

BIG DATA

PROJECT

Team #1

- Ahmed Amin Sayed
- Yousef Mahmoud Ibrahim Gilany
- Mahmoud Khalid Mahmoud
- Ahmed Magdy Abdelaziz

1. Importing libraries & loading data

The script starts by loading several tools useful for data handling, modeling, and analysis, like pandas for data operations, matplotlib for plotting graphs, and several modules from sklearn for statistical modeling and evaluation.

Loading Data

It loads game data from a CSV file, making sure specific columns like the season year and club IDs are recognized as numbers (integers).

Initial Data Check

It checks how many records (games) are in the dataset and prints this number.



2. Filtering Data

Narrowing Down Data

The script filters the game data to include only games from certain major football leagues, as specified by their identifiers (like L1, ESl, etc.).

After filtering, it prints out how many games are left that match these criteria. This tells us the count of games in major leagues.


3. Data Preparation

Data Aggregation by Club and Competition

Two DataFrames, **home_club_goals** and **away_club_goals**, are created using groupby and agg methods. These DataFrames aggregate the total goals scored and conceded at home and away, respectively, grouped by club ID, season, and competition.

The aggregation sums up goals scored at home (home_club_goals) and goals conceded by the home team (away_club_goals) for both home and away scenarios.

Merging Home and Away Data

 **merged_goals** DataFrame merges the **home_club_goals** and **away_club_goals** based on club IDs, competition, and season. It uses an outer join to ensure all data from either frame is retained, filling missing values with zeros.

A new column **club_id** is created by combining **home_club_id** and **away_club_id**, and then the original club ID columns are dropped.

Calculating Total Goals and Differences

Additional aggregations create **total_home_goals** and **total_away_goals**, which sum the goals scored and conceded by clubs as home and away teams, respectively.

New columns **GS** (goals scored) and **GC** (goals conceded) are calculated by adding the appropriate columns, and then redundant columns are dropped.

These frames are merged into `total_goals_info`, where goal difference (**GD**) is calculated as `GS - GC`.

Collecting Goals by Match

The last part of the code collects lists of goals for and against each club in both home and away contexts. This provides a detailed match-by-match view of the goals statistics.

4. Analysis

Merging Detailed Match Data:

The script merges the detailed goals data for both home and away matches. `goals_for_home` and `goals_against_home` are merged to form `home_matches`. Similarly, `goals_against_away` and `goals_for_away` are merged to form `away_matches`.

These merges align goals scored and conceded in home and away contexts based on club ID, competition, and season, ensuring all matches are accounted for.

Transforming Data for Processing:

Two lists, **`home_matches_key_value`** and **`away_matches_key_value`**, are created to hold tuples that uniquely identify each match by its club ID, competition, and season, paired with the goals scored and conceded. These tuples prepare the data for further aggregation and analysis.



Process Data:

The function **process_data** takes a list of match data in the key-value format and processes it to calculate the number of wins, losses, and draws for each club.

It iterates through each match, computes the result by subtracting goals conceded from goals scored, and aggregates these results under each unique key (club, competition, season).

Combine Results:

The **combine_results** function merges the results from home and away data processing into a single dictionary that represents the total matches, wins, losses, and draws for each club across all competitions and seasons.

Final Aggregation and Output:

The script calls **process_data** for both home and away match data, then combines these results using **combine_results**.

It prints out the final dictionary **team_win_lose** which includes the aggregated results across all home and away matches.

5. Final club Information

Extracting and Organizing Data:

The code iterates over the **team_win_lose** dictionary, which contains aggregated results of matches played, including the number of wins, draws, and losses for each club.



Within the loop, it extracts the club ID, competition ID, and season (as keys) along with the associated match results (as values). These extracted values are appended to respective lists.

Creating a New DataFrame:

A new DataFrame, `new_df`, is constructed using these lists. This DataFrame organizes the club's performance data into columns representing the club ID, season, competition ID, matches played, wins, draws, and losses.

Merging DataFrames:

The script then merges `new_df` with `total_goals_info`, another DataFrame that presumably contains detailed goals data (total goals scored, goals conceded, and goal difference).

The merge is performed on the keys `club_id`, `season`, and `competition_id`, ensuring that all statistical data is aligned per club per season per competition.

Final Compiled DataFrame:

The resultant DataFrame, `final_club_information`, now contains a comprehensive set of statistics for each club, including matches played, wins, losses, draws, and detailed goal statistics.



6. Adding label

Champion Data Dictionary:

A dictionary named `champions` is created, which stores competition IDs as keys. Each key maps to a list of tuples, where each tuple consists of a `club_id` and a `season`. These tuples represent the champion clubs for each season in the respective competition.

Defining the Check Function:

The **`is_champion`** function is defined to check if a given club in a specific competition and season was the champion. It takes a row from the DataFrame as input:

- It retrieves the **`competition_id`**, **`club_id`**, and **`season`** from the row.
- The function checks if the competition ID exists in the `champions` dictionary and if the tuple (`club_id`, `season`) is in the list associated with that competition ID in the dictionary.
- If the club is a champion for the competition and season, the function returns 1; otherwise, it returns 0.

Applying the Function to DataFrame:

The **`is_champion`** function is applied to each row of the `final_club_information` DataFrame using the `apply` method with `axis=1` (which indicates that the function should be applied to each row rather than each column).

A new column, `champion`, is added to **`final_club_information`**, which will hold the value 1 for rows where the club is a champion and 0 otherwise.



Final Output:

The script prints the updated **final_club_information** DataFrame, which now includes the champion column, providing a quick reference to identify the championship status of clubs across various competitions and seasons.

7. Add Feature Engineering

Win Match Ratio (win_match_ratio):

This ratio measures the proportion of wins to total matches played, adjusted by adding 1 to the numerator to avoid division by zero when there are no wins. This technique is a form of Laplace smoothing commonly used in statistical calculations to handle zero occurrences.

Goals Scored per Match Ratio (gs_match_ratio):

Similar to the win match ratio, this calculates the average goals scored per match, again adjusted by adding 1 for smoothing.

Goals Conceded per Match Ratio (gc_match_ratio):

This ratio computes the average goals conceded per match, with an adjustment factor added to manage zero values.

Win to Goals Scored Ratio (win_gs_ratio):

It evaluates the efficiency of winning relative to the number of goals scored, providing insight into how effectively a club turns goals into victories.

Win to Loss Ratio (win_lost_ratio):

This metric compares the number of wins to losses, offering a perspective on a club's overall competitive performance.

Goal Difference Adjustment (gs_gc):



Adds a small constant (0.1) to the goal difference to slightly adjust the scale, potentially for better normalization or visualization in later analysis.

Wins to Draws Ratio (wins_draws_ratio):

Analyzes the relationship between wins and draws, adjusted to handle cases with zero draws.

Goals Scored and Goal Difference Combined (gs_gd):

Adds the total goals scored and the goal difference together, possibly to create a combined metric of offensive strength.

8. MI_SCORE



make_mi_scores(X, y) Function:

Purpose

This function calculates the mutual information scores for each feature in dataset X relative to the target variable y.

Process

It uses the `mutual_info_regression` function from Scikit-learn, which measures the dependency between the variables. A higher score means that knowing the feature would give more information about the target.

The scores are returned as a pandas Series, with the feature names as the index.

These scores are then sorted in descending order to prioritize features with the highest mutual information scores.

Output

Returns a Series of mutual information scores indexed by feature names, sorted from highest to lowest.

plot_mi_scores(scores) Function:

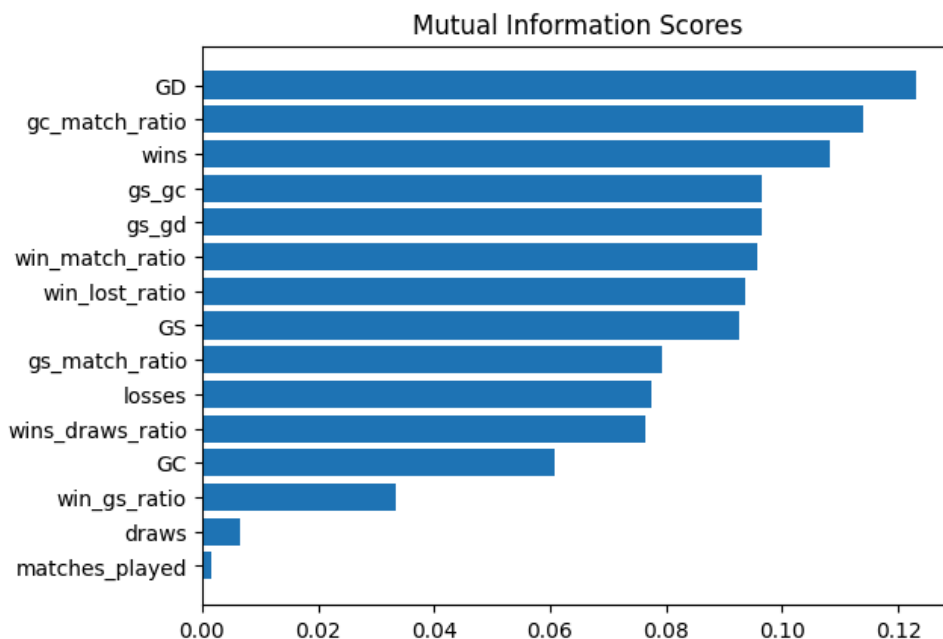
Purpose

Visualizes the mutual information scores using a horizontal bar chart, which can help in understanding the importance of various features visually.

Process

The scores are sorted in ascending order for the purpose of plotting. This makes the chart easier to read, with the highest scores appearing at the bottom of the chart.

A bar chart is created using matplotlib.pyplot. The function uses barh for horizontal bars, where width defines the x-positions of the bars, and ticks are the labels on the y-axis.



9. Smote For Balancing Data

the Synthetic Minority Over-sampling Technique (SMOTE) to balance an imbalanced dataset.

SMOTE Instantiation:

smote = SMOTE(): This line creates an instance of the SMOTE class from the `imblearn.over_sampling` library. SMOTE is a popular method used to generate synthetic samples from the minority class in a dataset, thereby balancing the class distribution without losing important information.



Resampling Data:

x_sm, y_sm = smote.fit_resample(x, y): This method applies SMOTE to the feature matrix `x` and the target vector `y`. The `fit_resample` method fits the SMOTE model to the data and then oversamples the minority class(es) in `y`, returning a new feature matrix `x_sm` and a new target vector `y_sm` with a balanced class distribution.

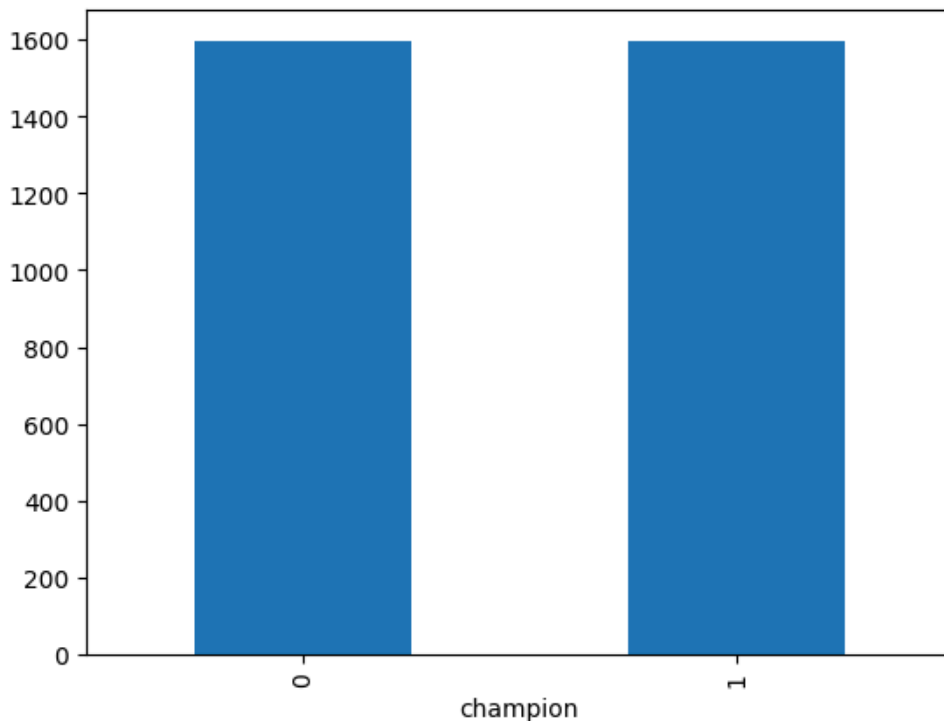
Visualisation of Resampled Target Variable:

pd.DataFrame(y_sm).champion.value_counts().plot(kind = 'bar'): This line performs several steps:

- **pd.DataFrame(y_sm):** Converts the resampled target vector `y_sm` into a pandas DataFrame. Assuming `y_sm` contains a column named

champion (as suggested by your syntax, though it might require adjustment based on actual column names).

- **.champion.value_counts():** Counts the occurrences of each class in the champion column, which are then returned as a Series.
- **.plot(kind = 'bar'):** create a bar chart displaying the frequency of each class in the resampled target variable. This visualisation is particularly useful for confirming that the classes have been balanced as intended by the SMOTE algorithm.



Splitting into train and test set Percentage 75% to 25%

train_test_split function from Scikit-learn's model_selection module to divide your dataset into training and testing subsets.

10. Classification Report

Report function generates a classification report, confusion matrix, and receiver operating characteristic (ROC) curve.

Classification Report:

The classification report provides a summary of precision, recall, F1-score, and support for each class. This gives us a detailed understanding of how well the classifier performs for each individual class.

Confusion Matrix:

The confusion matrix helps identify where the model is getting things right or wrong.



ROC Curve:

The ROC curve is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. It plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.

11. Logistic Regression

Logistic Regression Initialization:

A LogisticRegression model is initialized with specific parameters (C, penalty, solver, random_state).

Grid Search for Hyperparameter Tuning:

GridSearchCV is used to find the optimal **max_iter** parameter, which controls the number of iterations the solver performs before it converges. This is set up within a parameter grid and configured to perform a 10-fold cross-validation (cv=10), optimizing for accuracy (scoring='accuracy').



Re-initializing and Fitting the Logistic Regression:

The logistic regression classifier is re-initialized using the best max_iter value found (max_iter=150 in this case, assuming it's the best found by the grid search) and the other initial parameters.

The model is then trained on the full training dataset.

Cross-Validation for Model Evaluation:

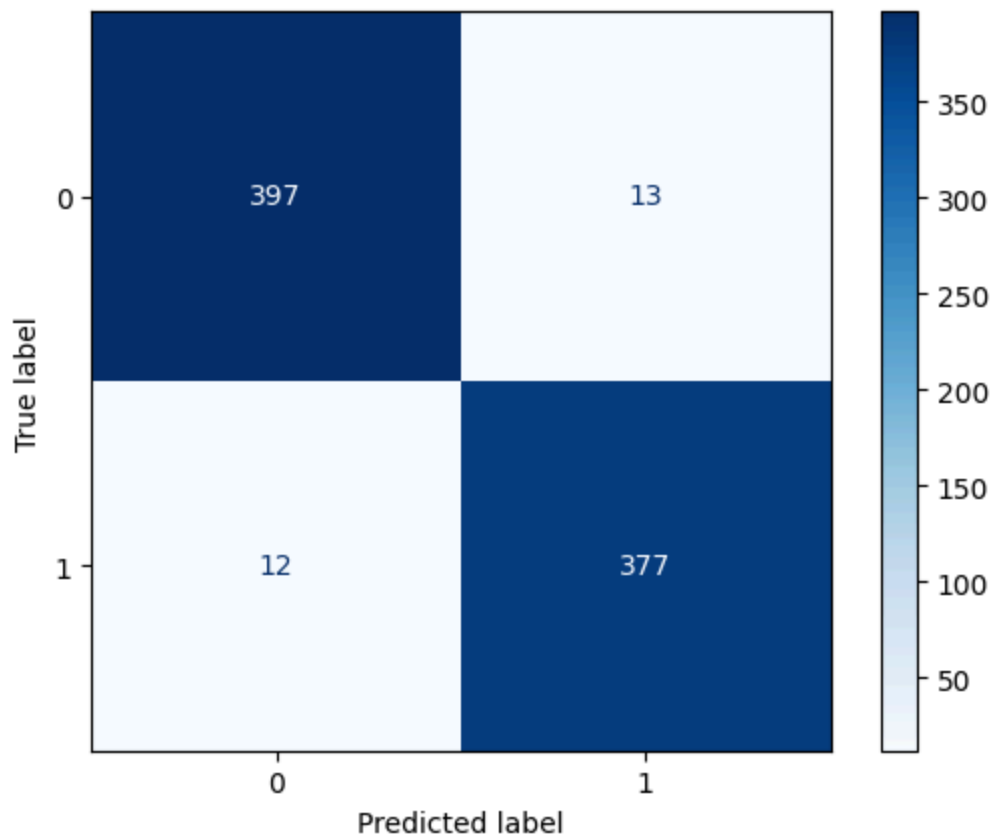
Cross-validation is executed using cross_val_score with 10 folds (cv=10), which evaluates the model's accuracy across different subsets of the training data, providing a robust estimate of its performance.

The average of the cross-validation scores is computed and printed, giving an overview of how well the model performs on average across all the validation sets.

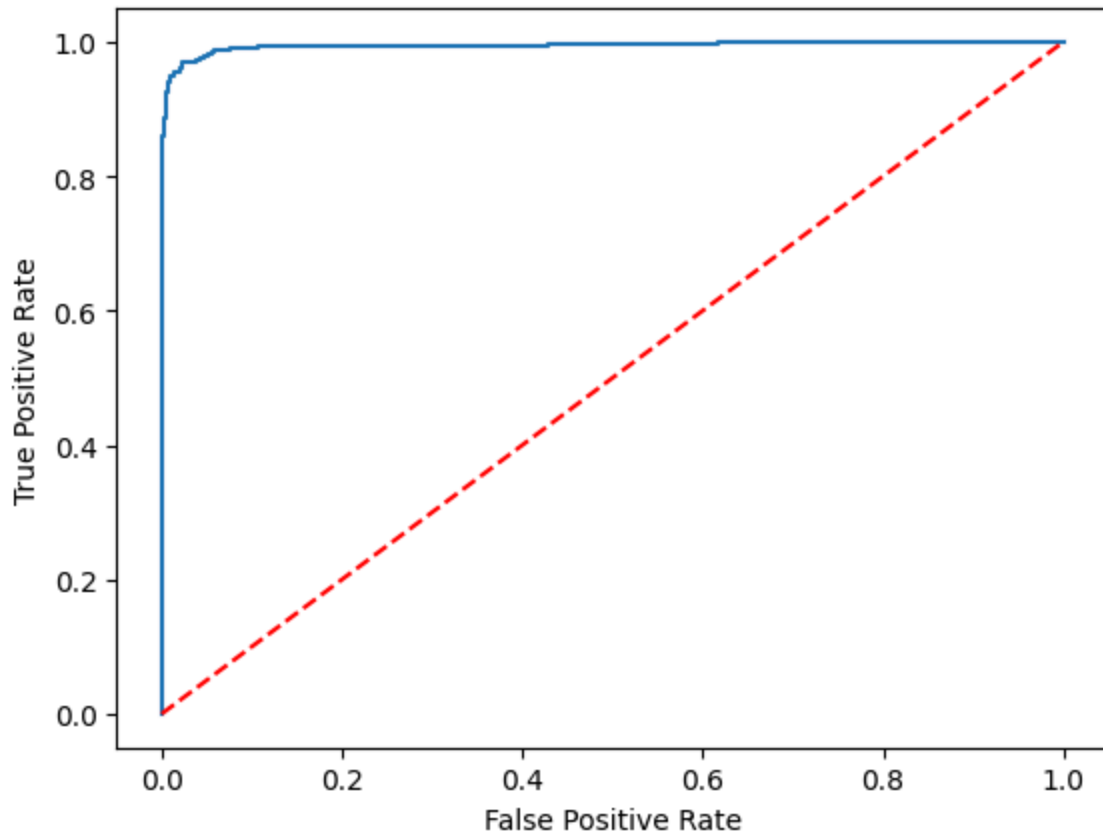
Model Performance Reporting:

Finally, the report function is called with the trained logistic regression model to print out the classification report, confusion matrix, and ROC curve, providing detailed insights into the model's classification capabilities on the test dataset.

Confusion Matrix



ROC

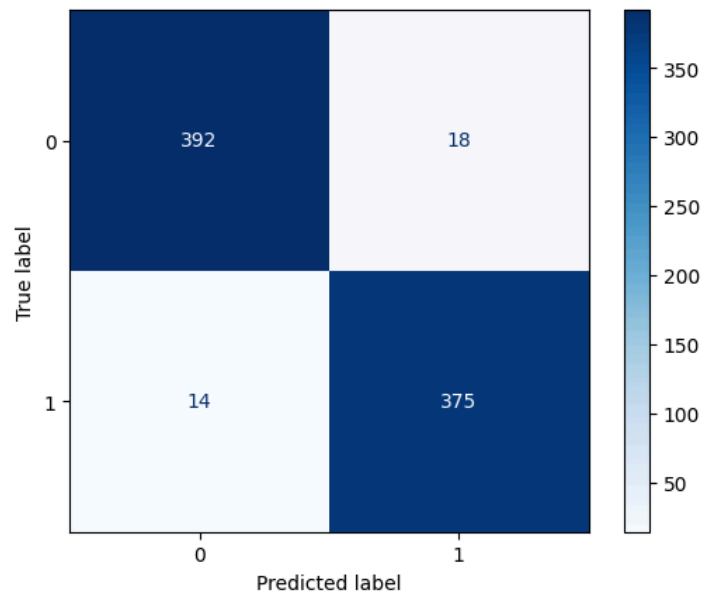


12. KNN

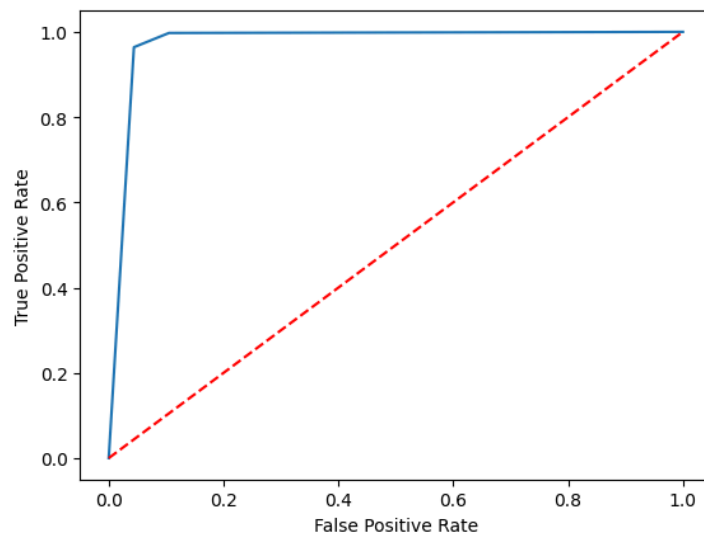
Same steps were done using KNN

We configured the KNN with certain parameters to best fit your data, and then trained it on your dataset. The configuration includes using two neighbors and the Manhattan distance metric

Confusion Matrix



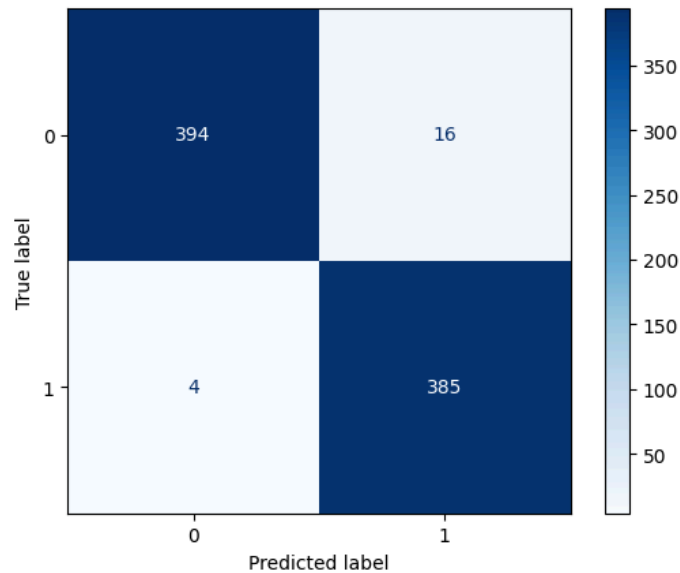
ROC



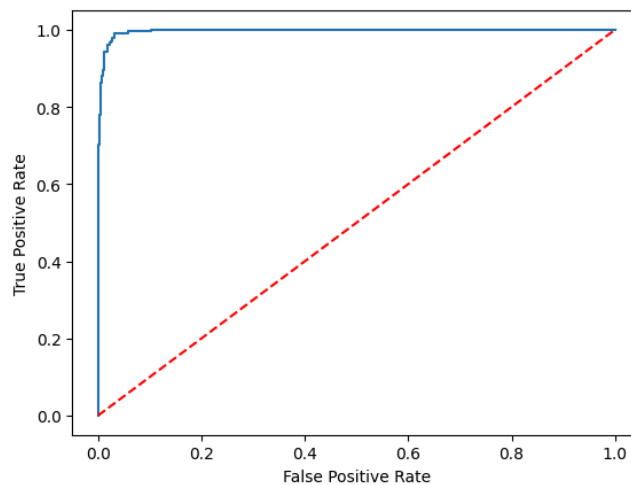
13. Random Forest

Same steps were done using Random Forest

Confusion Matrix



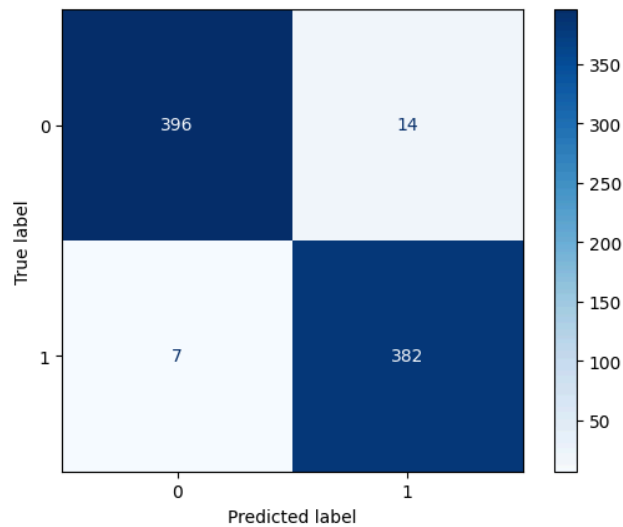
ROC



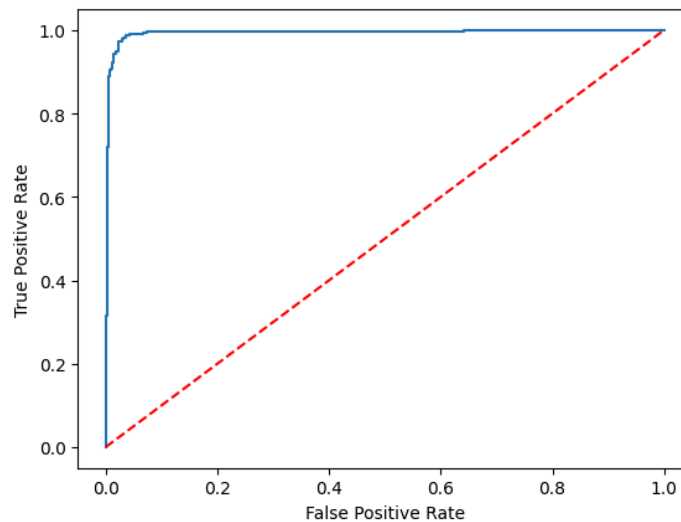
14. Gradient Boost

Same steps were done using Gradient Boost

Confusion Matrix



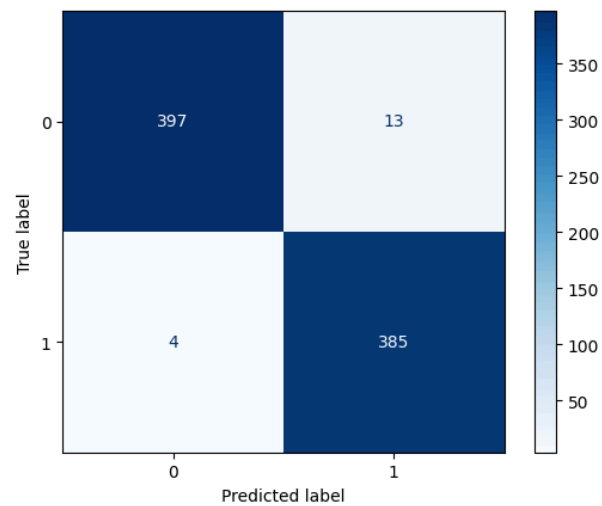
ROC



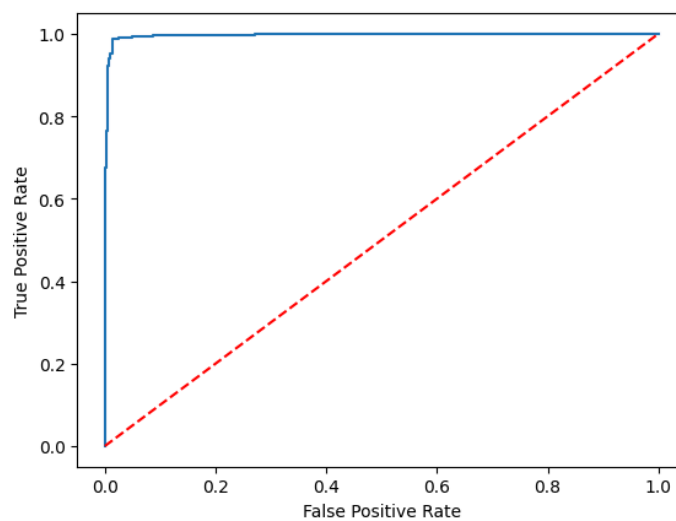
15. LightBGM

Same steps were done using LightBGM

Confusion Matrix



ROC



16. Score Analysis

Model	Score
Logistic Regression	0.975790
KNN	0.962420
Random Forest	0.970356
Gradient Boost	0.970352
LightBGM	0.977455

- LightGBM has the highest accuracy at approximately 97.7%. LightGBM is known for its efficiency and speed, especially on large datasets, and it often performs well on classification tasks due to its sophisticated handling of categorical features and gradient boosting technique.
- The KNN model, while still performing well, has the lowest accuracy at about 96.2%. KNN's performance can be sensitive to the number of neighbors chosen and the distance metric used. The slightly lower performance might indicate that KNN is either not capturing the complexities in the data as well as the other models, or it might be slightly overfitting to the noise in the training data.

