**Objective**

* To explain the steps undertaken to predict the winner of an IPL match using various machine learning algorithms.
* To discuss the features selected for the prediction model, including team information, toss results, and other match-related factors.
* To evaluate and compare the performance of different models (Logistic Regression, SVM, KNN, Decision Trees, Random Forest, and XGBoost) through metrics like accuracy, precision, recall, and F1-score.
* To detail the process of hyperparameter tuning and its impact on model performance.

**Data Understanding**

* The dataset contains 756 rows and 18 columns divided into categorical and numerical values, there are 4 numerical columns and 14 categorical columns.
* The description of each and every column is provided below for better understanding

1. Id - Indexing
2. Season - The respective year of the IPL season.
3. City - The city where the match was conducted.
4. Date - The date of the match.
5. team1 - Team1 playing
6. team2 - Team2 playing against Team1
7. toss winner - The team which won the toss
8. toss decision - Whether the team which won the toss chose batting or fielding
9. result – Result of the respective match
10. dl applied - Indicates if the Duckworth-Lewis method was applied.
11. Winner - The team that won the match.
12. win by runs - the details of on how much runs difference the team wins
13. win by wickets - The margin of victory by wickets.
14. player of match: The player awarded for stellar performance.
15. Venue - The stadium where the match was held.
16. umpires1 - Bowler’s end Umpire.
17. Umpire2 - Leg Umpire
18. Umpire3 - Offscreen Umpire

**Data Cleaning**

There were 656 null values from 6 different columns.

Bifurcation:

Umpire 3 - 637

City – 7

Winner – 4

Player of match – 4

Umpire1 – 2

Umpire2 – 2

Since these columns are categorical values, the null values were replaced with mode. For example – columns like City, Umpire1, Umpire2, Umpire3.

data\_ipl['city'] = data\_ipl['city'].fillna('Dubai')

data\_ipl['umpire1'] = data\_ipl['umpire1'].fillna(method = 'ffill')

data\_ipl['umpire2'] = data\_ipl['umpire2'].fillna(method = 'ffill')

data\_ipl['umpire3'] = data\_ipl['umpire3'].fillna(method = 'ffill')

Other columns such as Player of Match, winner cannot be replaced with repeated values so they were researched further. It was identified that the null values in these columns were because of having no results. Based on this observation the null values were replaced with “No Results”.

data\_ipl['winner'] = data\_ipl['winner'].fillna('No Result')

data\_ipl['player\_of\_match'] = data\_ipl['player\_of\_match'].fillna('No Result')

**Outliers**

There was a discrepancy in the team’s name for example Rising Pune Supergiant was mentioned as Rising Pune Supergiants which was corrected.

data\_ipl['team1'] = data\_ipl['team1'].replace('Rising Pune Supergiants', 'Rising Pune Supergiant')

The above provided code is for team1 and moreover the same team’s name was incorrect in different columns such as team2, toss winner and winner which was also replaced.

**Label Encoding – Value Mapping**

Since the values such as team1, team2 and all the other predictors has to be entered into the mode, they cannot remain categorical and has to be converted to numerical data.

data\_ipl['winner'] = data\_ipl['winner'].map({'Chennai Super Kings': 1, 'Mumbai Indians': 0,'Kolkata Knight Riders': 2,'Royal Challengers Bangalore': 3,'Kings XI Punjab': 4,'Rajasthan Royals': 5,'Delhi Daredevils': 6,'Sunrisers Hyderabad': 7,'Deccan Chargers': 8,'Rising Pune Supergiant': 9,'Gujarat Lions': 10,'Pune Warriors': 11,'Delhi Capitals': 12,'Kochi Tuskers Kerala': 13,'No Result': 14})

Manual Label Encoding

1. Values with binary kind of observations are converted to numerical data using Label Encoding. After the execution of the code, the column toss\_decision’s value bat will be converted to 1 and field will be converted to 0, the code is provided below for reference, the example code is provided below for reference.

data\_ipl['toss\_decision'] = data\_ipl['toss\_decision'].map({'bat':1, 'field':0})

1. Dummy variables are used for the column of result since it has only three known values – Tie, Normal and No Results. The values with highest count get their own column for example – If the normal results are more it will be categorized as column “Results\_normal” and the same goes for results tie. The code is provided below for reference.

data\_ipl = pd.get\_dummies(data\_ipl,columns = ['result'], drop\_first = True)

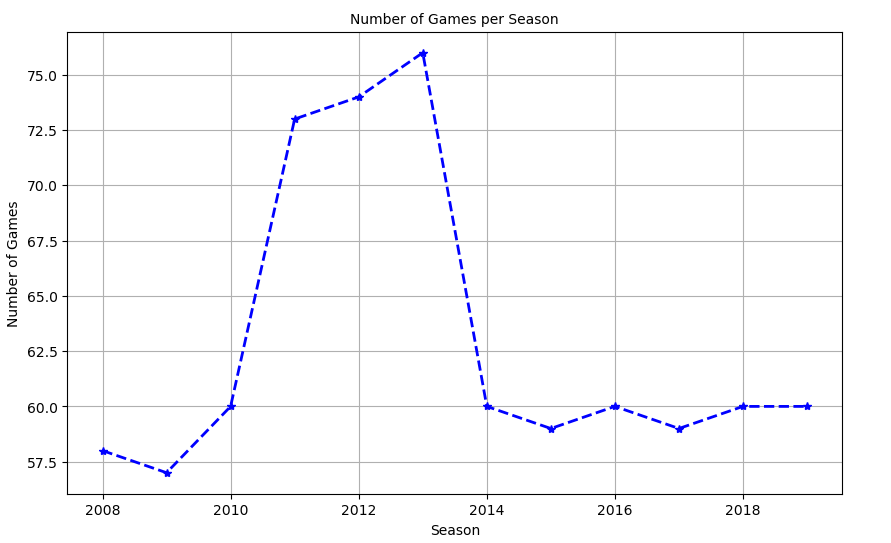
1. Using Label Encoder – We have converted all the categorical columns where the data type is object to numerical values after importing a module known as Label Encoder. The code is provided below for reference -

data\_ipl[data\_ipl.select\_dtypes(include=['object']).columns] = data\_ipl.select\_dtypes(include=['object']).apply(label\_encoder.fit\_transform)

**EDA (Exploratory Data Analysis)**

1. **Univariate Analysis**

Line chart is used for Univariate analysis where the X axis contains Season denoting the year in which the games were conducted and the number of games were taken into Y axis thus providing us a line chart denoting how many games were conducted in the respective years and also provides us the insight of which year had the highest games conducted and also the lowest. The visualization of the line chart is provided below for reference.

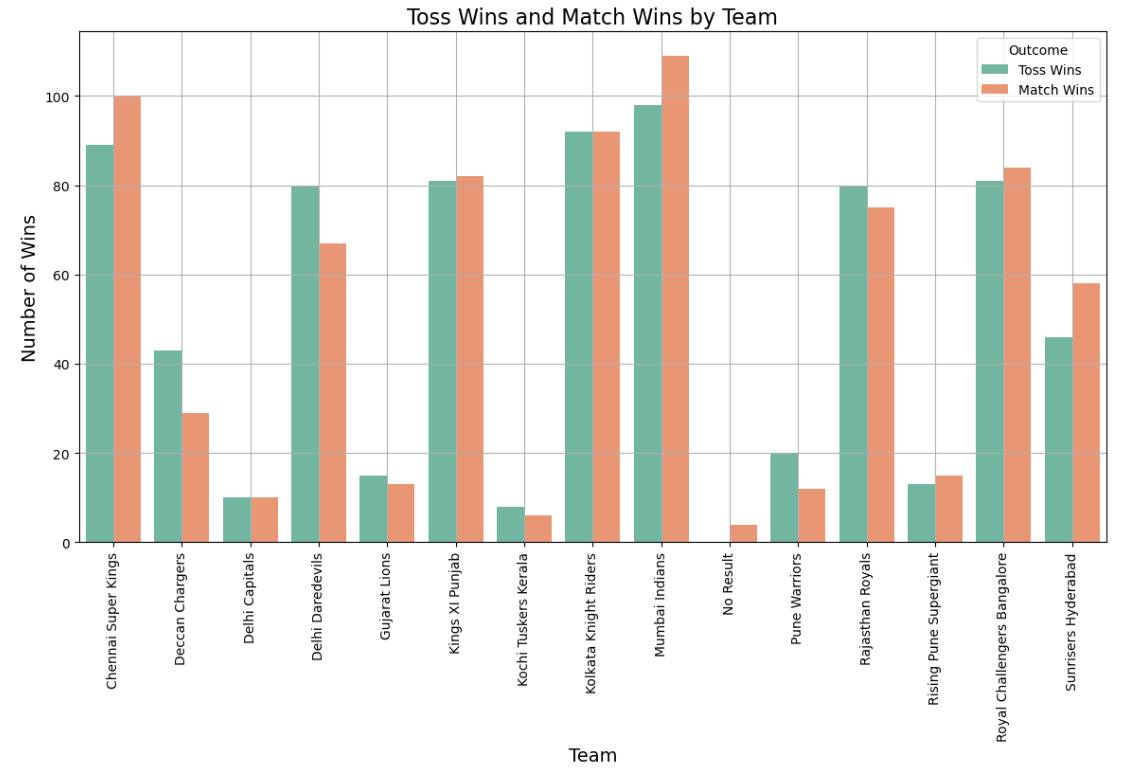


**Insights:**

* 2009 has the lowest number of games conducted with less than 57 games followed by 2008 which has almost near 60 games.
* The year in which the highest number of games were conducted is 2013 with more than 75 matches followed by 2012 which had more than 70 games conducted.
* After 2013 the number of games conducted were in the same range above 58 and below 60.
* Time period between 2011 and 2013 had the greater number of matches conducted in comparison to other years.

1. **Bi-Variate Analysis**

In Bi-Variate analysis, we have used two parameters for the Y axis, the first one being Count of Toss Wins and the second one being Count of Matches won. In X axis we have included the Team’s name in a way that the details of each team’s Toss wins and the match wins are displayed side by side for easier observation. Legend is provided for the Y axis labels – Toss wins, Match wins. The image of the Bar plot is provided below for reference.

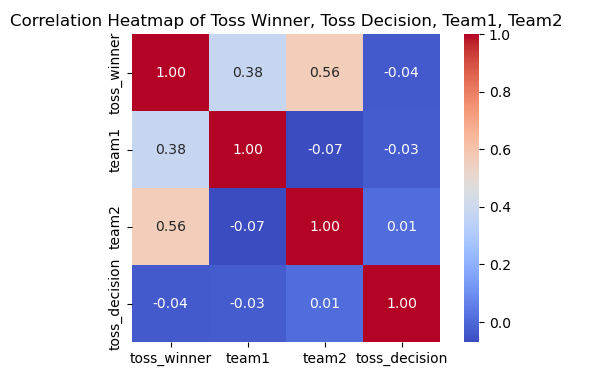


**Insights**

* Mumbai Indians has the Highest Number of Toss wins and also the highest number of Match wins thus remains being the team with most wins.
* Next to Mumbai Indians, we have Chennai Superkings as the team which also has More wins than the toss wins, however this team has less Toss wins than compared to the team with second most toss wins which is Kolkata Knight Riders.
* Kochi Tuskers Kerala has the lowest number of match wins and also toss wins as a result of playing only for a short time.

1. **Multi-Variate Analysis**

For multi-variate analysis because of not having multiple parameters in the same range having the relationship between one another, the parameters chosen for the multivariate analysis area three – toss decision, Team1, Team2 and Toss winner. A heatmap has been created to show the co-relation between these parameters.



**Insights**

* The relationship between team1 and team2 is the lowest compared to others.
* Team2 and Toss winner has the strongest relationship followed by toss winner and team1
* The week relationship status between these parameters shows the distribution between the range of values provided.

**Modelling Process**

**Libraries used**

# Step 1: Import necessary libraries

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

!pip install xgboost

from sklearn.preprocessing import MinMaxScaler

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

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier as KNN

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

import xgboost as xgb

from sklearn.model\_selection import GridSearchCV

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.preprocessing import LabelEncoder

from sklearn.model\_selection import cross\_val\_score, KFold

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.metrics import f1\_score, accuracy\_score, classification\_report, confusion\_matrix

**Inputs**

features = data\_ipl.drop(columns=['id','winner'], axis=1)

target = data\_ipl['winner']

print(features)

Every column except id and winner has been taken as the input parameters, we are assigning the input values as features and output as targets. In Inputs we are avoiding id and winner as Id is not relevant to the prediction and winner is what has to be predicted by the model.

**Normalization**

The values in the dataset after encoding are in multiple ranges for example – Win by runs can be more than 50 whereas win by wickets cannot be more than 10 and the dl applied has 0 and 1. To Normalize all the values within a specific range we are using MinMax Scaler so that the values from all the columns are between 0 and 1 making it easier for the model to read it.

scaler = MinMaxScaler()

scaler.fit(features) # Features indicate the details of input parameters

**Train & Split**

x\_train, x\_test, y\_train, y\_test = train\_test\_split(features,target,train\_size=0.7, random\_state=1)

We will then split the training and testing data in the ration of 7:3, 7 for training and 3 for testing data.

1. **Logistic Regression**

classifier = LogisticRegression()

classifier.fit(x\_train,y\_train)

We are using Logistic Regression as our first model to test the accuracy score for prediction, the model has been imported from sklearn library which can be found in the libraries used section.

We will use x train and y train data to train the model and use x test and y test to predict the accuracy score of the model.

x\_train\_prediction = classifier.predict(x\_train)

training\_data\_accuracy = accuracy\_score(y\_train, x\_train\_prediction)

We will train the accuracy score of the training data using x train and y train data.

**Accuracy score of training: 0.3497164461247637**

**x\_test\_prediction = classifier.predict(x\_test)**

**testing\_data\_accuracy = accuracy\_score(y\_test,x\_test\_prediction)**

We are using the test data for testing data accuracy and the result is provided below.

**Accuracy score of Logistic Regression Testing before Hyper Parameter tuning: 0.2422907488986784**

**Hyper Parameter Tuning Process**

log\_reg = LogisticRegression(max\_iter=1000)

param\_grid = {'C': [0.01, 0.1, 1, 10, 100],

'penalty': ['l1', 'l2'],

'solver': ['liblinear']}

grid\_search = GridSearchCV(log\_reg, param\_grid, cv=5, scoring='accuracy')

grid\_search.fit(x\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

best\_model = grid\_search.best\_estimator\_

y\_prediction\_log = best\_model.predict(x\_test)

print("Test Accuracy:", accuracy\_score(y\_test, y\_prediction\_log))

1. Searches for the best logistic regression hyperparameters using grid search with cv = 5
2. Trains the logistic regression model using the best-found parameters on the training data.
3. Predicts on the test data and prints the accuracy and best hyperparameter combination.

**The Testing score accuracy of Logistic regression has increased to 26%**

1. **Support Vector Machine – Classifier**

best\_classifier=SVC(kernel='linear',random\_state=0)

best\_classifier.fit(x\_train,y\_train)

Next to Logistic Regression we are using SVM – Classifier to see the accuracy score for the same dataset. We are using the training data for the classifier first.

y\_pred=best\_classifier.predict(x\_test)

We will get the accuracy score by using the testing data and predicting the data.

accuracy=accuracy\_score(y\_test,y\_pred)

print('accuracy score: ',accuracy)

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

print('confusion matrix: ',cm)

Now using the testing data and the prediction we are calculating the accuracy score.

**The accuracy score received for SVM-C before hyperparameter tuning is 0.2643171806167401**

**Hyper Parameter Tuning Process**

print(best\_classifier.get\_params())

param\_grid = {

'C': [0.1, 1, 10, 100], # Regularization parameter

'kernel': ['linear', 'rbf'], # Kernel types: linear and radial basis function (RBF)

'gamma': ['scale', 'auto'] # Gamma values for 'rbf' kernel}

grid\_search = GridSearchCV(estimator=SVC(), param\_grid=param\_grid, cv=10, n\_jobs=-1, verbose=4)

grid\_search.fit(x\_train, y\_train)

best\_classifier = grid\_search.best\_estimator\_

y\_pred = best\_classifier.predict(x\_test)

final\_accuracy\_SVC = accuracy\_score(y\_test, y\_pred)

print("Final Accuracy on SVC Test Set:", final\_accuracy\_SVC)

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

print('Confusion Matrix:\n', cm)

1. The grid search explores various combinations of the regularization parameter (C), kernel types (linear, rbf), and gamma values (scale, auto) to find the best set of parameters for the SVC model.
2. It uses 10-fold cross-validation (cv=10) to evaluate the model's performance more reliably, running in parallel with n\_jobs=-1 to speed up the process.
3. After identifying the best parameters, it evaluates the final model on the test set, printing the accuracy score and the confusion matrix for deeper insights into the model's performance.

**The testing score accuracy of SVM – Classifier has increased to 36%**

1. **KNN – Classifier**

clf = KNN(n\_neighbors = 5, metric='euclidean') #here use k=5

clf.fit(x\_train, y\_train)

test\_predict = clf.predict(x\_test)

k\_1 = f1\_score(y\_test, test\_predict, average='weighted') # or 'macro', 'micro'

print("F1 Score:", k\_1)

def Elbow(K):

test\_error = []

for i in K:

clf = KNN(n\_neighbors=i) # Use alias KNN here

clf.fit(x\_train, y\_train)

tmp = clf.predict(x\_test)

f1 = f1\_score(y\_test, tmp, average='weighted') # Multiclass-safe

error = 1 - f1

test\_error.append(error)

return test\_error

k = range(2, 20, 2)

test = Elbow(k)

plt.plot(k, test, marker='o')

plt.xlabel("Number of Neighbors (K)")

plt.ylabel("1 - F1 Score (Error)")

plt.title("Elbow Method using F1 Score")

plt.grid(True)

plt.show()

clf = KNN(n\_neighbors = 8) #after find K-value by elbow method

clf.fit(x\_train, y\_train)

test\_predict = clf.predict(x\_test,)

k\_2 = f1\_score(y\_test, test\_predict, average='weighted'

1. **Initial KNN Model**:  
   We have imported a KNN classifier with 5 neighbors and fit the model to the training data. It then calculates the F1 score on the test set using a weighted average.
2. **F1 Score Calculation**:  
   The initial F1 score (k\_1) is printed, showing the model's performance before tuning the k value.
3. **Elbow Method**:  
   The Elbow() function evaluates different values of k (2 to 20, with a step of 2), computing the error (1 - F1 score) for each. The goal is to find the k value that minimizes the error.
4. **Elbow Plot**:  
   The error values for different k values are plotted to visualize the "elbow," helping to select the optimal k value for better performance.
5. **Final Model Evaluation**:  
   After selecting the optimal k (e.g., k=8), the model is retrained and its F1 score (k\_2) on the test set is printed, showing the improvement after tuning.

**The accuracy score of KNN Model before Parameter tuning is 0.19045143793410693**

**Hyper-parameter tuning Process**

knn\_model = KNN()

param\_grid = {

'n\_neighbors': list(range(1, 21)),

'weights': ['uniform', 'distance'],

'metric': ['euclidean', 'manhattan']}

grid\_search = GridSearchCV(knn\_model, param\_grid, cv=5, scoring='accuracy')

grid\_search.fit(x\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

**print("Best Cross-Validated Accuracy:", grid\_search.best\_score\_)**

**best\_knn = grid\_search.best\_estimator\_**

**y\_pred\_knn = best\_knn.predict(x\_test)**

**print("\nClassification Report:\n", classification\_report(y\_test, y\_pred\_knn))**

**final\_accuracy\_knn = accuracy\_score(y\_test, test\_predict)**

**print("Final Accuracy on Test Set:", final\_accuracy\_knn)**

The grid search explores three hyperparameters: n\_neighbors (1 to 20), weights (uniform or distance), and metric (Euclidean or Manhattan) to find the best combination for the model.

A 5-fold cross-validation is used with GridSearchCV to evaluate the model on different subsets of the training data, ensuring robust performance.

After finding the best parameters, the best hyperparameters and cross-validated accuracy are displayed, followed by the classification report and final test accuracy.

**The testing score accuracy of KNN – Classifier has increased 21%**

1. **Decision Tree**

model = DecisionTreeClassifier()

model.fit(x\_train,y\_train)

y\_predict\_DT = model.predict(x\_test)

accuracy\_score(y\_test,y\_predict\_DT)

**The accuracy score received for Decision Tree before parameter tuning is 0.7533039647577092**

**Hyper Parameter Tuning Process**

dt\_model = DecisionTreeClassifier()

param\_grid = {

'criterion': ['gini', 'entropy'],

'max\_depth': [None, 5, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 5],

'max\_features': [None, 'sqrt', 'log2'],

'splitter': ['best', 'random']}

grid\_search = GridSearchCV(dt\_model, param\_grid, cv=5, scoring='accuracy', verbose=4, n\_jobs=-1)

grid\_search.fit(x\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

print("Best Cross-Validated Accuracy:", grid\_search.best\_score\_)

best\_dt\_model = grid\_search.best\_estimator

y\_pred\_dt = best\_dt\_model.predict(x\_test)

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred\_dt))

final\_accuracy\_dt = accuracy\_score(y\_test, y\_pred\_dt)

print("Final Accuracy on Test Set:", final\_accuracy\_dt)

The grid search explores a wide range of hyperparameters for the Decision Tree model, including criterion, max\_depth, min\_samples\_split, min\_samples\_leaf, max\_features, and splitter, to find the best combination for optimal performance.

It uses 5-fold cross-validation with GridSearchCV to evaluate the performance of different hyperparameter combinations, selecting the best parameters based on accuracy.

After identifying the best model, it prints the best hyperparameters, evaluates the model's performance on the test set, and displays the classification report and final accuracy.

**The testing score accuracy of Decision Tree has increased to 83%**

1. **Random Forest Classifier**

clf=RandomForestClassifier(n\_estimators=100,random\_state=2)

clf.fit(x\_train,y\_train)

y\_pred\_RF=clf.prediczt(x\_test)

from sklearn import metrics

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

**The accuracy score received for Random Forest before Parameter tuning is 0.7312775330396476**

**Hyper Parameter Tuning Process**

rf\_model = RandomForestClassifier(random\_state=2)

param\_grid = {

'n\_estimators': [100, 200, 300, 500],

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4],

'max\_features': ['auto', 'sqrt', 'log2'],

'bootstrap': [True, False],

'criterion': ['gini', 'entropy']}

grid\_search = GridSearchCV(estimator=rf\_model, param\_grid=param\_grid, cv=5, scoring='accuracy', verbose=4, n\_jobs=-1)

grid\_search.fit(x\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

print("Best Cross-Validated Accuracy:", grid\_search.best\_score\_)

best\_rf\_model = grid\_search.best\_estimator\_

y\_pred\_rf = best\_rf\_model.predict(x\_test)

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred\_rf))

final\_accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf)

print("Final Accuracy on Test Set:", final\_accuracy\_rf)

The grid search explores multiple hyperparameters of the Random Forest model, including the number of trees, max depth, split criteria, and feature selection strategies to find the best-performing configuration.

It uses 5-fold cross-validation with GridSearchCV to evaluate each hyperparameter combination based on accuracy, running in parallel to speed up the process.

After fitting, it outputs the best parameters, evaluates the model on the test set, and prints the classification report and final test accuracy.

**The testing score accuracy of Random Forest has increased to 81%**

1. **XGBoost Classifier**

xgb\_cfl = xgb.XGBClassifier(n\_jobs = -1)

xgb\_cfl.get\_params()

xgb\_cfl.fit(x\_train, y\_train) # default

xgb\_predictions = xgb\_cfl.predict(x\_test)

print(xgb\_predictions)

xgb\_predictions\_prob = xgb\_cfl.predict\_proba(x\_test)

print(xgb\_predictions\_prob)

acc=accuracy\_score(y\_test, xgb\_predictions)

print(acc)

**The accuracy score received for XGboost Classifier before Parameter tuning is 0.9427312775330396**

**Hyper Parameter Tuning Process**

**params = {** 'n\_estimators' : [100, 200, 500, 750], # no of trees

'learning\_rate' : [0.01, 0.02, 0.05, 0.1, 0.25], # eta

'min\_child\_weight': [1, 5, 7, 10],

'gamma': [0.1, 0.5, 1, 1.5, 5],

'subsample': [0.6, 0.8, 1.0],

'colsample\_bytree': [0.6, 0.8, 1.0],

'max\_depth': [3, 4, 5, 10, 12]}

folds = 5

param\_comb = 100

random\_search = RandomizedSearchCV(xgb\_cfl, param\_distributions=params, n\_iter=param\_comb, scoring='accuracy', n\_jobs=-1, cv=5, verbose=3, random\_state=42)

random\_search.fit(x\_train, y\_train)

print('\n Best accuracy for %d-fold search with %d parameter combinations:' % (folds, param\_comb))

print(random\_search.best\_score\_ )

The code performs a **randomized search** on hyperparameters for an XGBoost model (xgb\_cfl) using 5-fold cross-validation (cv=5). It randomly tests 100 combinations of hyperparameters like n\_estimators, learning\_rate, and max\_depth to find the best-performing configuration.

The RandomizedSearchCV is set to optimize for accuracy and runs in parallel with n\_jobs=-1 for faster execution. After fitting the model, it prints the best accuracy score found during the search along with the number of folds and parameter combinations tested.

**The testing score accuracy of XGBoost after tuning is 92%**

**Hyper Parameter Tuning**

Why Grid Search is used for Logistic, SVM, KNN, Decision Tree, Random Forest?

These models have fewer hyperparameters to tune. Grid Search works well when the number of combinations is small and manageable.

Why Random Search is used for XGBoost Classifier?

Randomized Search randomly selects a fixed number of combinations to test allowing to cover a wider hyperparameter space in less time.

**Model Comparison:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Models** | **Accuracy before Tuning** | **Hyper-Parameter Tuning** | **Accuracy after Tuning** | **Acceptance** |
| **Logistic Regression** | **24%** | **Grid Search** | **26%** | No |
| **SVM - Classifier** | **26%** | **Grid Search** | **36%** | No |
| **KNN - Classifier** | **20%** | **Grid Search** | **22%** | No |
| **Decision Tree** | **75%** | **Grid Search** | **83%** | Yes |
| **Random Forest** | **73%** | **Grid Search** | **81%** | Yes |
| **XGBoost Classifier** | **92%** | **Randomized Search** | **94%** | Yes |

**Conclusion**

1. XGBoost Performance:  
   Among the six models tested (Logistic Regression, SVM, KNN, Decision Trees, Random Forest, and XGBoost), XGBoost performed the best, delivering the highest accuracy. This demonstrates its effectiveness in capturing complex patterns in the data.
2. Model Improvement:  
   Models like Logistic Regression, SVM, and KNN performed poorly (below 50% accuracy), indicating that the feature relationships in the data may be non-linear or complex. This suggests the need for models that can better handle such relationships, like tree-based models (e.g., Random Forest or XGBoost).
3. Key Features for Prediction:  
   The most important features identified by XGBoost for predicting match outcomes include toss winner (15.57%), venue (12.41%), and team2 (11.05%). These factors suggest that the outcome of the toss, the match location, and the opposing teams are critical in determining the match result.
4. Low Impact Features:  
   Features such as results tie (0.00%) and umpires (with low importance scores, e.g., umpire3 at 4.53%) were found to have minimal influence on the match outcome. This indicates that tie-related factors or umpire information are less relevant for predicting match results in this dataset.
5. Future Enhancements:  
   The model's performance could potentially be further improved by exploring additional features such as player performance stats, weather conditions, or historical match data. Also, tuning the hyperparameters of models like XGBoost or Random Forest might improve prediction accuracy.