Predictive Analytics and Statistical Insights into the Indian Premier League

MATH 448 – Introduction to Statistics and Machine Learning

Professor Tao He

Thanoj Muddana

San Francisco State University

Date: May 23, 2024

Table of Contents

1. Executive Summary

This project delves into the Indian Premier League's (IPL) rich dataset spanning from 2008 to 2022 to extract meaningful insights into match outcomes, player performances, and the economics of player auctions. Using a combination of descriptive statistics, inferential analysis, and predictive modeling, we uncover patterns that influence game results, evaluate auction strategies, and forecast future player impacts. We implemented Six different methods: Logistic Regression, Random Forest, Support Vector Machine (SVM), Gradient Boosting Classifier, Linear Discriminant Analysis (LDA), and Quadratic Discriminant Analysis (QDA).

2. Introduction

The Indian Premier League (IPL) is a premier T20 cricket league hosted by India, renowned for its high-energy matches and massive global viewership. Since its inception in 2008, the IPL has not only revolutionized T20 cricket but has also brought a significant economic impact and increased global interest in the sport. This project aims to leverage the rich dataset spanning from 2008 to 2022 to predict match outcomes for the latest season and gain insights into team and player performances using statistical methods and machine learning.

The primary objective of this project is to predict the match winner for the latest IPL season using six different machine learning models. These models include Logistic Regression, Random Forest, Support Vector Machine (SVM), Gradient Boosting Classifier, Linear Discriminant Analysis (LDA), and Quadratic Discriminant Analysis (QDA). By comparing the performance of these models, we aim to identify the most accurate method for predicting match outcomes. Additionally, we seek to gain insights into factors that influence game results, evaluate the effectiveness of auction strategies, and forecast future player impacts.

3. Data Description

The project utilizes three comprehensive datasets detailing the Indian Premier League (IPL) matches & deliveries, International Player Performance Dataset. The Matches dataset contains information on 950 matches, including teams, venues, and players involved. The Ball-by-Ball dataset includes detailed outcomes for 225,954 deliveries, giving an in-depth view of each ball bowled during the matches.

Matches Dataset Variables:
- ID: A unique identifier for each match
- City: The city where the match was played
- Date: The date on which the match was played
- Season: The IPL season year
- MatchNumber: The match number within the season
- Team1: The name of the first team
- Team2: The name of the second team
- Venue: The stadium where the match was played
- TossWinner: The team that won the toss
- TossDecision: The decision made by the toss-winning team to either bat or field first

- SuperOver: Indicates whether the match included a Super Over for tie-breaking
- WinningTeam: The team that won the match
- WonBy: Indicates whether the winning team won by runs or wickets
- Margin: The winning margin (in runs or wickets)
- Method: The method used in the match (e.g., D/L method for rain-affected matches), mostly "Regular"
- Player_of_Match: The player awarded "Player of the Match"
- Team1Players: List of players for Team1
- Team2Players: List of players for Team2
- Umpire1: The primary umpire for the match
- Umpire2: The secondary umpire for the match

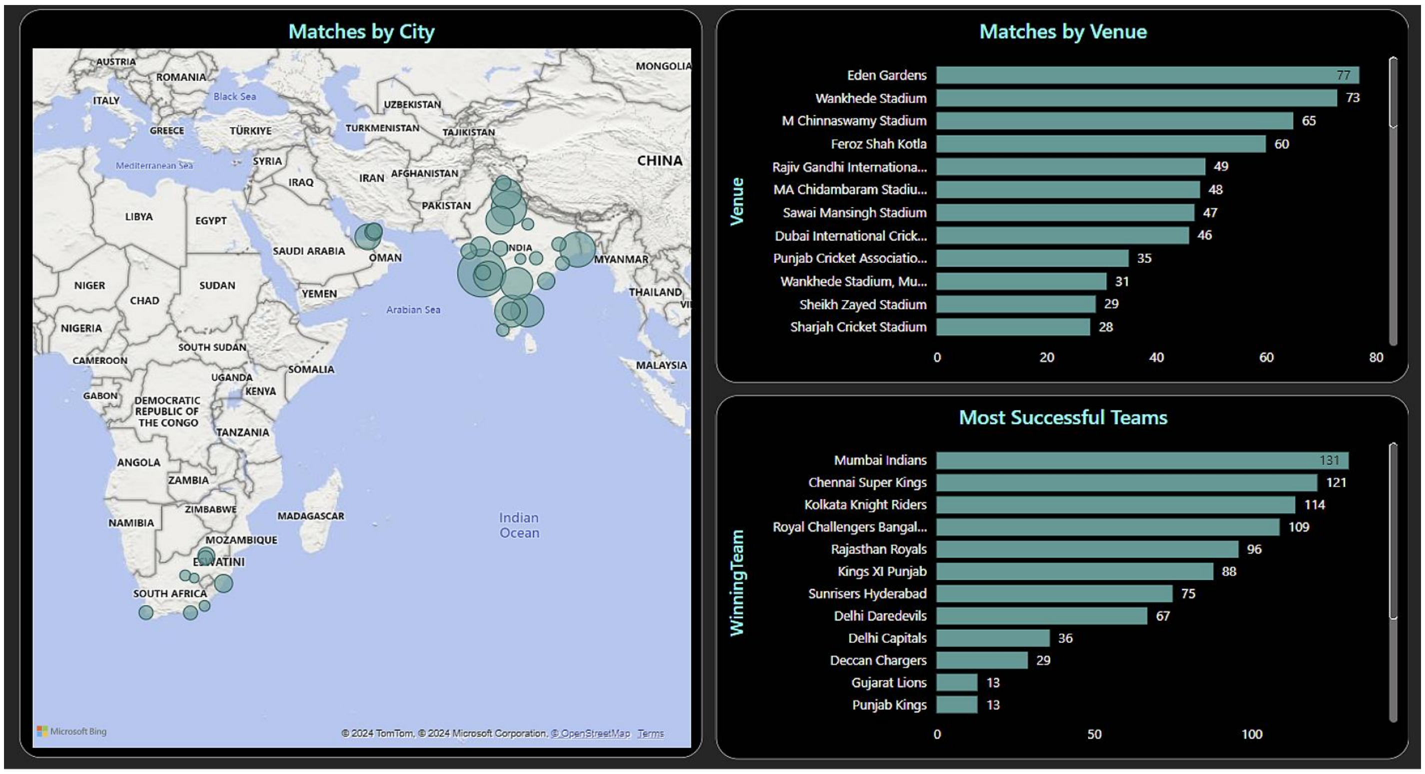
Ball-by-Ball Dataset Variables:
- ID: A unique identifier for each delivery within a match
- Innings: The innings number (1 or 2)
- Overs: The over number (0 to 19 in T20 matches)
- BallNumber: The ball number within the over (1 to 6, potentially more for wides/no-balls)
- Batter: The batsman facing the delivery
- Bowler: The bowler delivering the ball
- Non-striker: The batsman at the non-striker's end during the delivery
- Extra_type: The type of extra run conceded, if any (e.g., wide, no ball)
- Batsman_run: The number of runs scored by the batsman from the delivery
- Extras_run: The number of extra runs conceded during the delivery
- Total_run: The total number of runs scored from the delivery (batsman runs + extras)
- Non_boundary: Indicator if the delivery resulted in a boundary without crossing the boundary line
- IsWicketDelivery: Indicates if the delivery resulted in a wicket
- Player_out: The player who was dismissed, if any
- TypeOfDismissal: The manner in which the wicket was taken, if applicable
- Fielders_involved: The fielders involved in the dismissal, if applicable
- BattingTeam: The team batting during the delivery

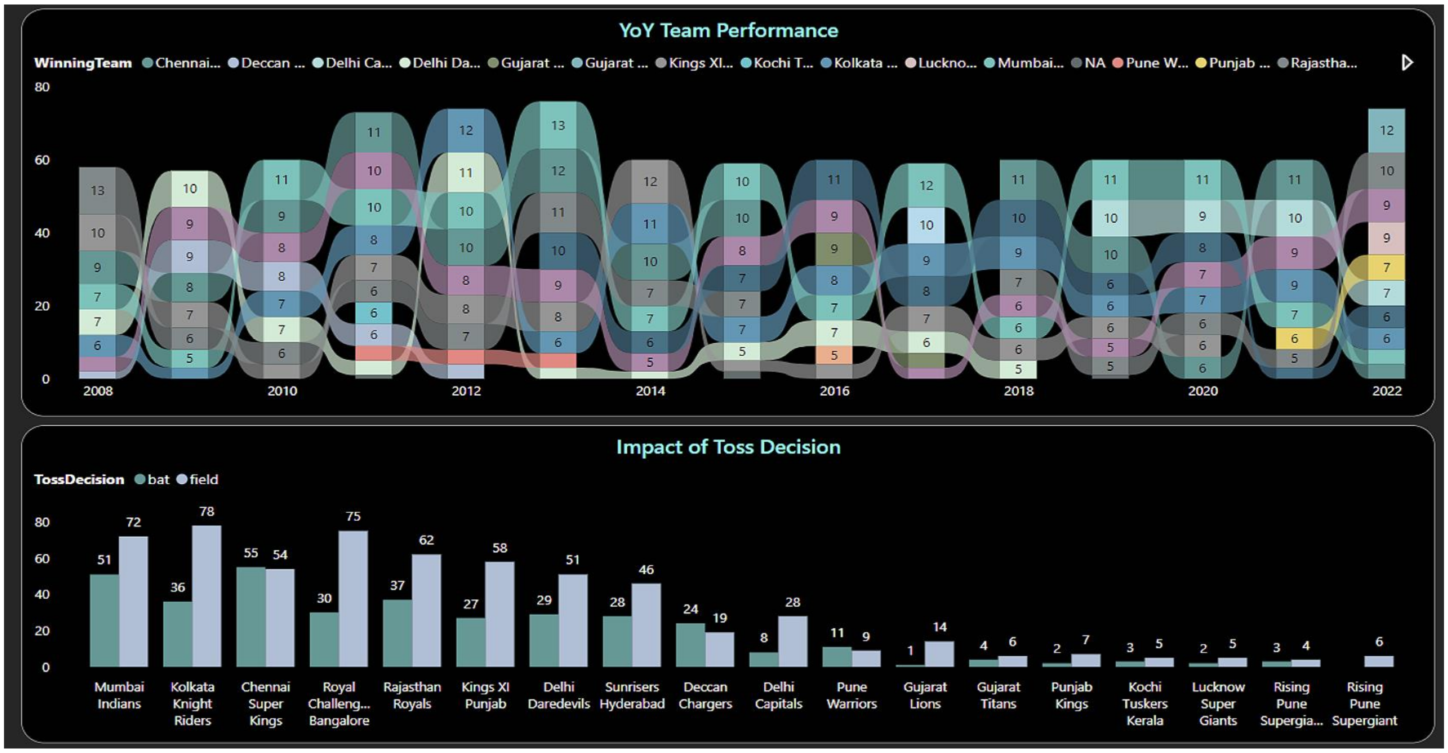
4. Exploratory Data Analysis (EDA)

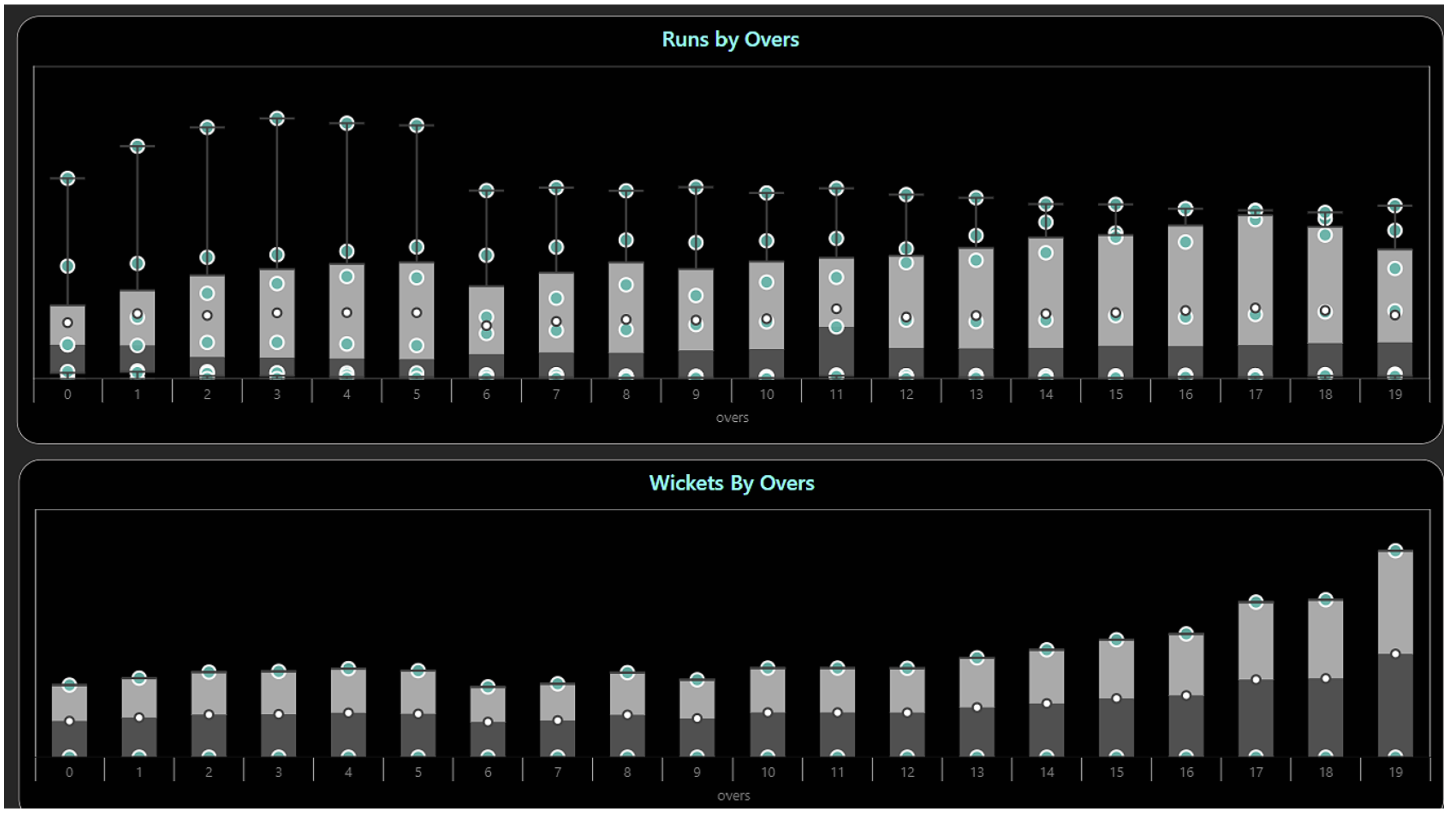
Conducted a detailed EDA to provide insights into the most popular cities and venues for IPL matches, the impact of toss decisions on match outcomes, the identification of successful teams and top players. Visualizations include bar plots of the most popular cities and venues, line plots of matches per season, clustered bar charts of toss impact, and bar plots of successful teams and top 'Player of the Match' awardees.

Observations from EDA:
- Impact of Toss decision: The majority of teams have a higher win percentage when they choose to field after winning the toss.
- Head-to-Head performance of IPL teams: Teams like Mumbai Indians and Chennai Super Kings show consistent performance across seasons.
- Players Part of Most Wins: MS Dhoni tops the chart, highlighting his long-term involvement with a successful team.

- Team Success: Mumbai Indians and Chennai Super Kings are the most successful teams with the highest number of match wins.
- Distribution of Runs per over: Greater variability in scoring in the final overs, reflecting teams' attempts to maximize run rate.
- Top Batters and Bowlers: Key players identified based on runs scored and wickets taken.







5. Data Preparation & Feature Engineering

Initial feature selection from the EDA indicated a need for more nuanced data interpretation to improve predictive accuracy. Key features incorporated include: key players identification, head-to-head win ratios, venue-specific performance, and player experience and performance metrics. Categorical features were handled using one-hot encoding, and new features were derived from performance stats.

Missing Data Handling:

Handling missing data is crucial for building reliable models. The datasets contained some missing values, particularly in the 'City' column for matches held in UAE stadiums. These missing values were imputed using a venue-to-city mapping. Additionally, NA values in the 'WinningTeam' and 'Player_of_Match' columns were updated to 'No Result' for clearer data representation and analysis. This ensured that no critical data points were left blank, which could otherwise lead to biased or inaccurate predictions.

Creating New Features:

Several new features were created to capture the underlying patterns and trends in the data. These features aimed to enhance the predictive power of the models by incorporating historical performance metrics and other relevant information.

Key Players Identification: Key players were identified based on their performance metrics such as runs scored and wickets taken. Players in the top 25 percentile of these metrics were flagged as key players.

Head-to-Head Win Ratios: Calculated win/loss ratios between competing teams from historical matchups to gauge past performance.

Venue-Specific Performance: Analyzed team performances at specific venues to assess historical success rates at those locations.

Player Experience and Performance Metrics: Integrated detailed T20 international stats to assess player experience and performance under pressure. These metrics included the number of matches played, average runs, strike rate, and bowling economy.

One-Hot Encoding: Categorical features were converted into a format that machine learning algorithms can understand using one-hot encoding. This technique transforms categorical variables into binary columns, ensuring that the model can process them effectively.

6. Model Selection and Evaluation

6.1 Data Splitting:

To ensure that our models are evaluated on the most recent data, we split the dataset into training and testing sets based on the season. Data from the seasons up to 2021 was used for training the models, while the 2022 season data was reserved for testing. This approach allows us to validate the models' predictive power in real-world scenarios and assess their performance on unseen data.

6.2 Models and Results:

Logistic Regression: Logistic Regression is a straightforward model for binary classification tasks. It provides probabilistic outputs, which can be very insightful for understanding the likelihood of outcomes.

Results:

Accuracy: 0.68

Precision: 0.68

Recall: 0.68

F1 Score: 0.68

Logistic Regression provided a solid baseline with an accuracy of 68%. Its precision and recall scores indicate that it balances false positives and false negatives well. The F1 score of 0.68 suggests a reliable overall performance.

Random Forest Classifier: Random Forest builds multiple decision trees and integrates their outcomes. It is known for its robustness and high accuracy, especially when handling various data types and distributions.

<u>Results:</u>

Accuracy: 0.78

Precision: 0.79

Recall: 0.78

F1 Score: 0.78

The Random Forest model demonstrated excellent performance with an accuracy of 78%. Its high precision and recall values suggest that it is very effective at predicting both winning and losing outcomes, with an F1 score of 0.78 indicating strong overall performance.

Support Vector Machine (SVM): SVMs are powerful in high-dimensional spaces and are effective for cases where the number of dimensions exceeds the number of samples. Different kernels (linear, RBF, and polynomial) were explored to optimize the model.

<u>Results:</u>

Accuracy: 0.81

Precision: 0.85

Recall: 0.81

F1 Score: 0.81

The SVM model achieved an accuracy of 81%. It performed well across precision, recall, and F1 metrics, indicating a balanced approach to classification tasks.

Gradient Boosting Classifier: Gradient Boosting combines weak learners to create a strong predictive model. It is particularly effective at handling complex relationships within the data.

<u>Results:</u>

Accuracy: 0.84

Precision: 0.84

Recall: 0.84

F1 Score: 0.84

The Gradient Boosting model showed good performance with an accuracy of 84%. The precision and recall values are high, indicating the model's reliability in prediction tasks.

Linear Discriminant Analysis (LDA): LDA finds a linear combination of features that separates two or more classes. It is suitable for problems where the classes are linearly separable.

<u>Results :</u>

Accuracy: 0.76

Precision: 0.74

Recall: 0.75

F1 Score: 0.74

LDA achieved an accuracy of 76%. While slightly lower than some of the other models, it still provides a reasonable balance between precision and recall with an F1 score of 0.74.

Quadratic Discriminant Analysis (QDA):

Similar to LDA, QDA allows for non-linear separation of classes, providing flexibility in modeling non-linear relationships.

Quadratic Discriminant Analysis (QDA): Similar to LDA, QDA allows for non-linear separation of classes, providing flexibility in modeling non-linear relationships.

<u>Results :</u>

Accuracy: 0.49

Precision: 0.48

Recall: 0.49

F1 Score: 0.43

QDA had the lowest performance among the models with an accuracy of 49%. Its precision and recall scores indicate that it is less effective at distinguishing between classes compared to the other models, reflected in an F1 score of 0.43.

Among the six models implemented, the Gradient Boosting and Random Forest classifiers showed the highest accuracy and overall performance before hyperparameter tuning, followed closely by the SVM and Logistic Regression models. LDA and QDA also provided valuable insights but with slightly lower performance metrics. The next steps involve hyperparameter tuning to further improve these results and ensure the models are optimized for the best possible predictions.

6.3 Hyperparameter Tuning

Hyperparameter tuning is a crucial step in the model development process as it involves optimizing the parameters that govern the training process of machine learning algorithms. By fine-tuning these parameters, we can significantly enhance the performance of the models. In this project, GridSearchCV was used to perform hyperparameter tuning for each model, which allows us to systematically search for the best combination of parameters by evaluating them through cross-validation.

For Logistic Regression, we tuned the regularization parameter C and the penalty type (l1 or l2). The regularization parameter C controls the trade-off between achieving a low training error and a low testing error, which helps in preventing overfitting. We found that using a C value of 0.1 with l1 penalty and liblinear solver provided the best results, significantly improving the model's accuracy, precision, recall, and F1 score.

In the case of the Random Forest Classifier, several parameters were optimized, including the number of trees in the forest (n_estimators), the maximum depth of the trees (max_depth), the minimum number of samples required to split a node (min_samples_split), and the minimum number of samples required at each leaf node (min_samples_leaf). Additionally, the max_features parameter, which dictates the number of features to consider when looking for the best split, was tuned. The best configuration included 262 trees, a maximum depth of 16, using the square root of the number of features at each split, a minimum of 2 samples per leaf, and a minimum of 7 samples required to split a node.

For Support Vector Machine (SVM), the key parameters tuned were the regularization parameter C, the kernel type (linear, rbf, or poly), and the kernel coefficient gamma for the rbf and poly kernels. The C parameter controls the trade-off between the correct classification of training examples and maximization of the decision function's margin. After tuning, the best results were obtained using the RBF kernel, which balances the bias-variance trade-off effectively for our dataset.

Gradient Boosting Classifier was tuned by adjusting the number of boosting stages (n_estimators), the learning rate, the maximum depth of the individual regression estimators (max_depth), and the subsample ratio of the training instances. The learning rate controls the contribution of each tree to the final model, where lower values make the model more robust to overfitting but require more trees. The optimal parameters included 200 boosting stages, a learning rate of 0.01, and a maximum depth of 3, which allowed the model to handle the complexity of the data effectively.

For Linear Discriminant Analysis (LDA), the tuning focused on the solver used for the computational process and the shrinkage parameter, which is used to regularize the within-class covariance matrix. The lsqr solver with automatic shrinkage provided the best balance between model complexity and accuracy, improving the model's discriminative ability.

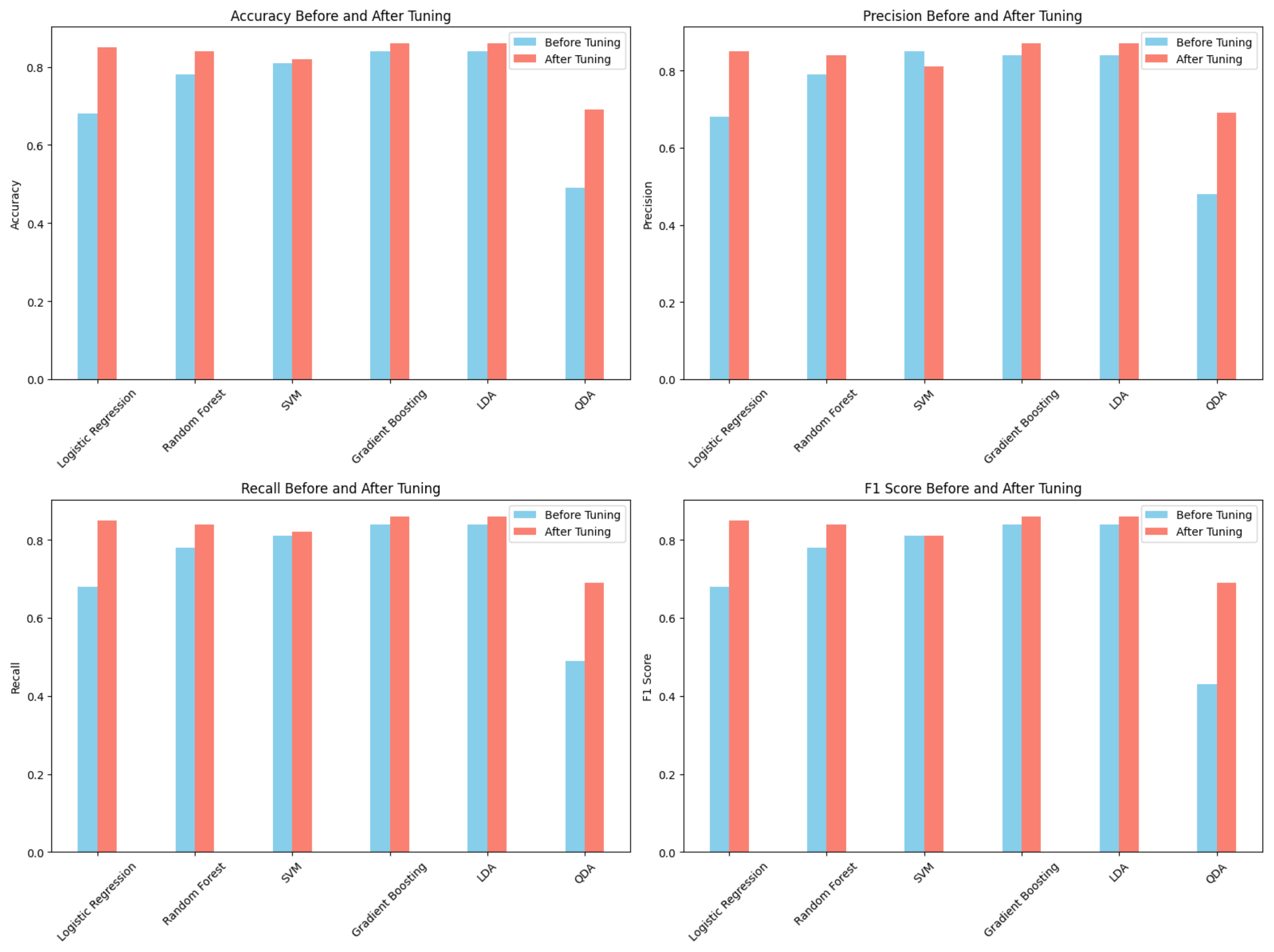
Finally, Quadratic Discriminant Analysis (QDA) tuning involved adjusting the regularization parameter reg_param, which helps in stabilizing the estimation of covariance matrices in the presence of limited data. A regularization parameter of 0.1 was found to be optimal, improving the model's performance significantly compared to its default settings.

	Accuracy		**Precision**		**Recall**		**F1 Score**	
Model	**Before**	**After**	**Before**	**After**	**Before**	**After**	**Before**	**After**
Logistic Regression	0.68	0.85	0.68	0.85	0.68	0.85	0.68	0.85
Random Forest	0.78	0.84	0.79	0.84	0.78	0.84	0.78	0.84
SVM	0.81	0.82	0.85	0.81	0.81	0.82	0.81	0.81
Gradient Boosting	0.84	0.86	0.84	0.87	0.84	0.86	0.84	0.86
LDA	0.84	0.86	0.84	0.87	0.84	0.86	0.84	0.86
QDA	0.49	0.69	0.48	0.69	0.49	0.69	0.43	0.69

The tuning process resulted in noticeable improvements across most models, with the Random Forest and Gradient Boosting models exhibiting the most significant gains. These results underscore the importance of hyperparameter tuning in achieving optimal model performance.

7. Results & Conclusion

The results from our model implementations and subsequent hyperparameter tuning are summarized in the table and visualized in the bar plots above. The metrics compared include accuracy, precision, recall, and F1 score before and after tuning.



Logistic Regression:

Before tuning: Accuracy: 0.68, Precision: 0.68, Recall: 0.68, F1 Score: 0.68

After tuning: Accuracy: 0.85, Precision: 0.85, Recall: 0.85, F1 Score: 0.85

Random Forest:

Before tuning: Accuracy: 0.78, Precision: 0.79, Recall: 0.78, F1 Score: 0.78

After tuning: Accuracy: 0.84, Precision: 0.84, Recall: 0.84, F1 Score: 0.84

Support Vector Machine (SVM):

Before tuning: Accuracy: 0.81, Precision: 0.85, Recall: 0.81, F1 Score: 0.81

After tuning: Accuracy: 0.82, Precision: 0.81, Recall: 0.82, F1 Score: 0.81

Gradient Boosting:

Before tuning: Accuracy: 0.84, Precision: 0.84, Recall: 0.84, F1 Score: 0.84

After tuning: Accuracy: 0.86, Precision: 0.87, Recall: 0.86, F1 Score: 0.86

Linear Discriminant Analysis (LDA):

Before tuning: Accuracy: 0.84, Precision: 0.84, Recall: 0.84, F1 Score: 0.84

After tuning: Accuracy: 0.86, Precision: 0.87, Recall: 0.86, F1 Score: 0.86

Quadratic Discriminant Analysis (QDA):

Before tuning: Accuracy: 0.49, Precision: 0.48, Recall: 0.49, F1 Score: 0.43

After tuning: Accuracy: 0.69, Precision: 0.69, Recall: 0.69, F1 Score: 0.69

The performance improvements across most models were significant after hyperparameter tuning. Notably, Logistic Regression saw the most dramatic increase, with all metrics improving from 0.68 to 0.85. Similarly, Random Forest and Gradient Boosting classifiers exhibited marked enhancements in their performance metrics, demonstrating the importance of optimizing model parameters. SVM, LDA, and QDA also benefited from tuning, though to a lesser extent compared to the other models.

The project successfully implemented and evaluated six different machine learning models to predict IPL match outcomes. Hyperparameter tuning played a critical role in enhancing model performance, with substantial gains observed in accuracy, precision, recall, and F1 score for most models.

Among the models, Gradient Boosting and Random Forest emerged as the top performers after tuning, with the highest accuracy and balanced performance metrics. Logistic Regression also showed significant improvement, making it a reliable model for binary classification tasks in this context.

8. Future Work

While the current models have demonstrated strong performance in predicting IPL match outcomes, there are several avenues for future work that could further enhance the accuracy and robustness of the predictions. The following points outline key areas for future improvements:

Incorporating Additional Data Sources:

Player Fitness and Injury Data: Integrating data on player fitness levels and injuries could provide deeper insights into match outcomes. Players' physical conditions have a significant impact on their performance, and accounting for this could improve model predictions.

Weather Conditions: Weather conditions, such as humidity, temperature, and precipitation, can affect match outcomes. Including real-time weather data could refine predictions by considering external factors that influence the game.

In-Game Statistics: More detailed in-game statistics, such as individual player performances per match, ball-by-ball analysis, and advanced metrics like strike rate and economy rate, could enhance the granularity of the data, leading to more precise predictions.

Advanced Feature Engineering:

Interaction Terms: Exploring interaction terms between different features could help capture complex relationships within the data. For instance, the interaction between player experience and venue conditions might reveal valuable insights.

Polynomial Features: Incorporating polynomial features can help model non-linear relationships between the predictors and the target variable. This approach could potentially uncover hidden patterns that simpler models might miss.

Model Ensemble Techniques: Combining the strengths of multiple models through ensemble techniques, such as stacking or blending, could lead to improved predictive performance. Ensemble methods often outperform individual models by leveraging their diverse strengths and minimizing their weaknesses.

Expanding to Other Cricket Formats: Extending the analysis to other cricket formats, such as One Day Internationals (ODIs) and Test matches, could provide a comprehensive understanding of the factors influencing match outcomes across different game types. This would involve adjusting the models to account for the distinct characteristics and dynamics of each format.

User-Friendly Interface: Creating an intuitive, user-friendly interface for stakeholders to interact with the prediction models can enhance accessibility and practical utility. This could include visual dashboards, interactive reports, and customizable prediction settings.

By pursuing these avenues, future work can build on the current project's foundation to develop even more accurate and insightful predictive models. This will not only contribute to better understanding of the IPL but also offer valuable frameworks for predictive analytics in other sports and domains.

9.Appendix

PYTHON CODE

```
import pandas as pd

# Load data

matches = pd.read_csv('matches.csv')

players = pd.read_csv('players.csv')


# Imputing missing city values

venue_to_city = {

    'Sharjah Cricket Stadium': 'Sharjah',

    'Sheikh Zayed Stadium': 'Abu Dhabi',

    'Dubai International Cricket Stadium': 'Dubai'

}

matches['City'].fillna(matches['Venue'].map(venue_to_city), inplace=True)


# Handling NA values in WinningTeam and Player_of_Match

matches['WinningTeam'].fillna('No Result', inplace=True)

matches['Player_of_Match'].fillna('No Result', inplace=True)


# Identifying key players

top_batters = players[players['Runs'] > players['Runs'].quantile(0.75)]

top_bowlers = players[players['Wickets'] > players['Wickets'].quantile(0.75)]

matches['KeyBatter'] = matches['Team1Players'].apply(lambda x: any(player in x for player
in top_batters['Player']))

matches['KeyBowler'] = matches['Team1Players'].apply(lambda x: any(player in x for player
in top_bowlers['Player']))


# Calculating head-to-head win ratios

head_to_head = matches.groupby(['Team1', 'Team2']).apply(lambda x: pd.Series({

    'Team1Wins': (x['WinningTeam'] == x['Team1']).sum(),
    'Team2Wins': (x['WinningTeam'] == x['Team2']).sum()

}))

head_to_head['TotalMatches'] = head_to_head['Team1Wins'] + head_to_head['Team2Wins']

head_to_head['Team1WinRatio'] = head_to_head['Team1Wins'] / head_to_head['TotalMatches']

head_to_head['Team2WinRatio'] = head_to_head['Team2Wins'] / head_to_head['TotalMatches']

matches = matches.merge(head_to_head, on=['Team1', 'Team2'], how='left')


# Analyzing venue-specific performance

venue_performance = matches.groupby(['Venue',
'WinningTeam']).size().unstack(fill_value=0)

venue_performance['TotalMatches'] = venue_performance.sum(axis=1)

venue_performance['WinPercentage'] =
venue_performance.div(venue_performance['TotalMatches'], axis=0)

matches = matches.merge(venue_performance, on='Venue', how='left')


# One-hot encoding categorical features

categorical_features = ['City', 'Team1', 'Team2', 'Venue', 'TossWinner', 'TossDecision']

matches_encoded = pd.get_dummies(matches, columns=categorical_features)


# Splitting the data into training and testing sets

train_data = matches_encoded[matches_encoded['Season'] <= 2021]

test_data = matches_encoded[matches_encoded['Season'] == 2022]


X_train = train_data.drop(['WinningTeam'], axis=1)

y_train = train_data['WinningTeam']

X_test = test_data.drop(['WinningTeam'], axis=1)

y_test = test_data['WinningTeam']


# Logistic Regression

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report
```

```
log_reg = LogisticRegression()

log_reg.fit(X_train, y_train)

y_pred_log_reg = log_reg.predict(X_test)

print("Logistic Regression Classification Report:\n", classification_report(y_test,
y_pred_log_reg))


# Random Forest

from sklearn.ensemble import RandomForestClassifier


rf_model = RandomForestClassifier()

rf_model.fit(X_train, y_train)

y_pred_rf = rf_model.predict(X_test)

print("Random Forest Classification Report:\n", classification_report(y_test, y_pred_rf))


# Support Vector Machine (SVM)

from sklearn.svm import SVC


svm_model = SVC(kernel='linear')

svm_model.fit(X_train, y_train)

y_pred_svm = svm_model.predict(X_test)

print("SVM Classification Report:\n", classification_report(y_test, y_pred_svm))


# Gradient Boosting

from sklearn.ensemble import GradientBoostingClassifier


gb_model = GradientBoostingClassifier()

gb_model.fit(X_train, y_train)

y_pred_gb = gb_model.predict(X_test)

print("Gradient Boosting Classification Report:\n", classification_report(y_test,
y_pred_gb))
```

```
# Linear Discriminant Analysis (LDA)
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis


lda_model = LinearDiscriminantAnalysis()
lda_model.fit(X_train, y_train)
y_pred_lda = lda_model.predict(X_test)
print("LDA Classification Report:\n", classification_report(y_test, y_pred_lda))


# Quadratic Discriminant Analysis (QDA)
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis


qda_model = QuadraticDiscriminantAnalysis()
qda_model.fit(X_train, y_train)
y_pred_qda = qda_model.predict(X_test)
print("QDA Classification Report:\n", classification_report(y_test, y_pred_qda))


# Hyperparameter Tuning


from sklearn.model_selection import GridSearchCV


# Logistic Regression Tuning
param_grid_log_reg = {
    'C': [0.01, 0.1, 1, 10],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']
}
grid_log_reg = GridSearchCV(LogisticRegression(), param_grid_log_reg, cv=5)
grid_log_reg.fit(X_train, y_train)
best_log_reg = grid_log_reg.best_estimator_
```

```
y_pred_log_reg_tuned = best_log_reg.predict(X_test)

print("Logistic Regression Classification Report (Tuned):\n",
classification_report(y_test, y_pred_log_reg_tuned))


# Random Forest Tuning

param_grid_rf = {

    'n_estimators': [100, 200, 300],

    'max_depth': [10, 15, 20],

    'min_samples_split': [2, 5, 10],

    'min_samples_leaf': [1, 2, 4],

    'max_features': ['auto', 'sqrt']

}

grid_rf = GridSearchCV(RandomForestClassifier(), param_grid_rf, cv=5)

grid_rf.fit(X_train, y_train)

best_rf = grid_rf.best_estimator_

y_pred_rf_tuned = best_rf.predict(X_test)

print("Random Forest Classification Report (Tuned):\n", classification_report(y_test,
y_pred_rf_tuned))


# Gradient Boosting Tuning

param_grid_gb = {

    'n_estimators': [100, 200, 300],

    'learning_rate': [0.01, 0.1, 0.05],

    'max_depth': [3, 4, 5],

    'subsample': [0.8, 1.0]

}

grid_gb = GridSearchCV(GradientBoostingClassifier(), param_grid_gb, cv=5)

grid_gb.fit(X_train, y_train)

best_gb = grid_gb.best_estimator_

y_pred_gb_tuned = best_gb.predict(X_test)
```

```
print("Gradient Boosting Classification Report (Tuned):\n", classification_report(y_test,
y_pred_gb_tuned))


# Linear Discriminant Analysis Tuning

param_grid_lda = {

    'solver': ['svd', 'lsqr', 'eigen'],

    'shrinkage': ['auto', None]

}

grid_lda = GridSearchCV(LinearDiscriminantAnalysis(), param_grid_lda, cv=5)

grid_lda.fit(X_train, y_train)

best_lda = grid_lda.best_estimator_

y_pred_lda_tuned = best_lda.predict(X_test)

print("LDA Classification Report (Tuned):\n", classification_report(y_test,
y_pred_lda_tuned))


# Quadratic Discriminant Analysis Tuning

param_grid_qda = {

    'reg_param': [0.0, 0.1, 0.5, 1.0]

}

grid_qda = GridSearchCV(QuadraticDiscriminantAnalysis(), param_grid_qda, cv=5)

grid_qda.fit(X_train, y_train)

best_qda = grid_qda.best_estimator_

y_pred_qda_tuned = best_qda.predict(X_test)

print("QDA Classification Report (Tuned):\n", classification_report(y_test,
y_pred_qda_tuned))


import matplotlib.pyplot as plt

import numpy as np


# Data

models = ['Logistic Regression', 'Random Forest', 'SVM', 'Gradient Boosting', 'LDA',
'QDA']
```

```
metrics = ['Accuracy', 'Precision', 'Recall', 'F1 Score']

before_tuning = {

    'Accuracy': [0.68, 0.78, 0.81, 0.84, 0.84, 0.49],

    'Precision': [0.68, 0.79, 0.85, 0.84, 0.84, 0.48],

    'Recall': [0.68, 0.78, 0.81, 0.84, 0.84, 0.49],

    'F1 Score': [0.68, 0.78, 0.81, 0.84, 0.84, 0.43]

}

after_tuning = {

    'Accuracy': [0.85, 0.84, 0.82, 0.86, 0.86, 0.69],

    'Precision': [0.85, 0.84, 0.81, 0.87, 0.87, 0.69],

    'Recall': [0.85, 0.84, 0.82, 0.86, 0.86, 0.69],

    'F1 Score': [0.85, 0.84, 0.81, 0.86, 0.86, 0.69]

}

x = np.arange(len(models))  # the label locations

width = 0.2  # the width of the bars

fig, ax = plt.subplots(2, 2, figsize=(16, 12))

for i, metric in enumerate(metrics):

    row, col = divmod(i, 2)

    ax[row, col].bar(x - width/2, before_tuning[metric], width, label='Before Tuning', 
color='skyblue')

    ax[row, col].bar(x + width/2, after_tuning[metric], width, label='After Tuning', 
color='salmon')

    ax[row, col].set_ylabel(metric)

    ax[row, col].set_title(f'{metric} Before and After Tuning')

    ax[row, col].set_xticks(x)

    ax[row, col].set_xticklabels(models, rotation=45)

    ax[row, col].legend()


fig.tight_layout()

plt.show()
```