SPORTS ANALYTICS

Figure 1: Sports Analysis

Source: https://datasportsgroup.com/news-article/114939/how-sports-analytics-is-used-in-ipl/

ABSTRACT

In this paper, we are presenting interpret routing machine learning models to effectively predict performance of cricket teams and players in the Indian Premier League (IPL) by utilizing simple readily-available data on the team players in terms of batting average, strike rate, wickets taken and economy rate. Filling the divide between esoteric opaque analytics and real-world requirements of coaches, analysts and users of fantasy sports, the study pioneers the incorporation of domain knowledge-based feature engineering in a supervised learning framework constituted of Support Vector Machine, Decision Tree, Random Forest and XGBoost methods, under a time-sensitive performance assessment regime. Match level and ball by ball IPL statistical data of pre and post 2008 (2008-2024) was preprocessed, analyzed using exploratory statistics and subsequently underwent recursive elimination of features to derive parsimonious high impact predictors. Seasons 20082021 were used to model, seasons 2022 to validate, seasons 2023 to test, as measures of forward-looking generalisability. The outcomes demonstrate that XGBoost provided the best discrimination (macro AUC = 0.646), and Random Forest should be referred to as the best top-1 accuracy (39%) in the prediction of multi-class winners. Actionable forecasts of the top performers in batting, bowling, and fielding were obtainable using regression models at player level and the RMSE values fitted well the modeled outputs, especially those of bowling and fielding outputs. The framework provides data-based insights on a transparent basis that can be used to inform team selection, match strategy, and fantasy cricket decision making to narrow the academic modelling-real world usability divide.

List of Abbreviations

AUC – Area Under the Curve, **BBL** – Big Bash League, **CI** – Confidence Interval. .**CV** – Cross-Validation**, DLS** – Duckworth–Lewis–Stern method, **DT** – Decision Tree. **EDA** – Exploratory Data Analysis, **ECE** – Expected Calibration Error, **F1** – F1 Score (harmonic mean of precision and recall), **FPR** – False Positive Rate, **GB** – Gradient Boosting, **H2H** – Head-to-Head, **ICC** – International Cricket Council, **IQR** – Interquartile Range, **IPL** – Indian Premier League, **KKR** – Kolkata Knight Riders, **LSTM** – Long Short-Term Memory, **MAE** – Mean Absolute Error, **ML** – Machine Learning, **MLP** – Multi-Layer Perceptron, **MOM** – Man of the Match (now Player of the Match), **ODI** – One Day International, **OvR** – One-vs-Rest, **PCA** – Principal Component Analysis, **PP** – Powerplay, **R^2** – Coefficient of Determination, **RF** – Random Forest, **RFE** – Recursive Feature Elimination, **RMSE** – Root Mean Squared Error, **ROC** – Receiver Operating Characteristic, **SVM** – Support Vector Machine, **T20** – Twenty20 cricket format, **XGB** – Extreme Gradient Boosting (XGBoost)

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor **Konstantin Blyuss** for his invaluable guidance, constructive feedback, and steady encouragement throughout this research. I am also grateful to the examiners and teaching staff for their insightful comments, which helped refine the methodology and strengthen the arguments presented here.

My appreciation extends to my course mates and entire faculty for the many discussions, critiques, and shared resources that enriched this work. I am indebted to the organizations and individuals who provided data, tools, and domain expertise, as well as to the **library and IT support** teams for their prompt assistance with access, software, and infrastructure.

To all those who have contributed to this endeavor, both seen and unseen, your support has been immeasurable, and for that, I am sincerely grateful.

TABLE OF CONTENT

CHAPTER 3: DATA

CHAPTER 4: METHODOLOGY

CHAPTER 5: RESULT

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

Sports analytics has become a game changer in the contemporary world of sports, touching decision-making processes in matters pertaining to player selection, game strategy, and performance analysis. Analytics is also slowly gaining pace in cricket as professional leagues such as the Indian Premier League (IPL), Big Bash League (BBL), and international tournaments continue to use analytics to achieve a competitive advantage (Sumathi *et al.* 2023). The advent of fantasy sports games like Dream11 is currently occurring simultaneously with the availability of sports statistics and data to fans, who will now have access to more predictive information and adopt it more universally.

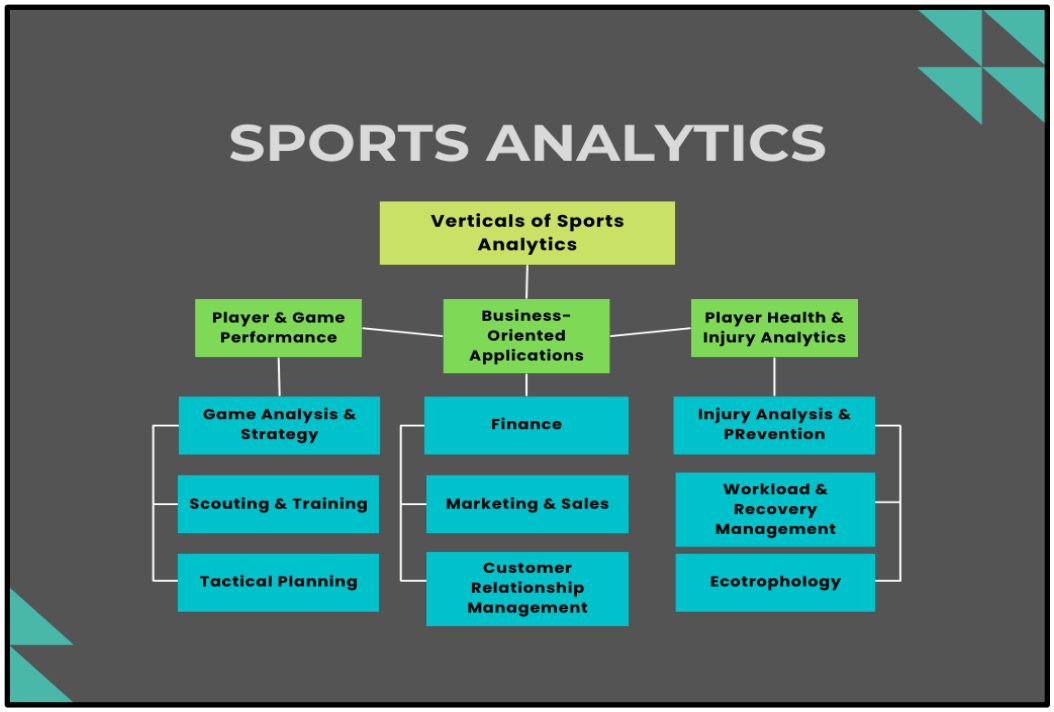


Figure 2: Sports Analytics Working Principle

(Source: https://www.spoclearn.com/blog/sport-analytics/)

In spite of the current development of cricket analytics, most existing predictive models are based on complicated algorithms or company-specific data, which is not transparent and more difficult to understand. Besides, the simple personal statistics are either ignored or not leveraged very well in forecasting team success (Mishra *et al.* 2024). Since cricket is active in the changeability of the teams' result being determined by a combination of the team play as well as individual performance, there is a necessity to possess models that permit simplicity along with precision in the prediction. The current paper aims to fill that gap and develop interpretable machine learning

models based on the basic player metrics that would help analysts, coaches, and fantasy users reach informed and confident decisions based on their data.

1.1 Problem Statement

Although industry-wide analytics are becoming more widespread in cricket, most predictive models are using complex algorithms that are proprietary or do not consider the simplicity of strategy required in practice. It further reveals a shortage of understandable, accessible models that can predict team play, based on the simple, readily obtainable players' statistics of runs scored, wickets taken, or batting average. The available models do not pay much attention to how the aggregate of individual inputs turns into team performance. Moreover, coaches, analysts and fantasy sports enthusiasts have difficulty in using these models as they are neither very transparent nor simple to use (Wickramasinghe, 2022). Thus, it is vital that a data-based framework, which is interpretable, should be designed which uses straightforward performance indices to forecast team performance with accuracy and will aid in the process of decision-making during real-life circumstances in cricket.

1.2 Aim and Objectives

Aim

The aim of the study is to develop interpretable machine learning models that predict cricket team performance using basic, easily obtainable individual player statistics.

Objectives

Goal 1: Evaluate existing predictive analytic approaches for cricket team performance

- A considerable thing to look at are publications from cricket expert, as well as operate in major leagues like the Indian Priemer League (IPL) and Big Bash League (BBL), and professional ICC tournaments such as ICC ODI Tournament or T20 Tournament.
- It analyzes what allows for successful model of cricket development and what current statistical models do and do not predict.
- Show a predictive value against current models, and determine missing information.

Goal 2: Identify simple easily obtainable individual player metrics significantly associated with team success

- We use exploratory data analysis (EDA) on historical cricket match datasets to interpret the dataset and look for any potential issues that can help you predict in future.

- Use correlation analysis and feature importance techniques to find out which player metrics (e.g. runs scored, runs taken, strike rate and economy rate), are most predictive of positive team outcomes.
- Document metrics that are most influential to generate the model in the design process.

Goal 3: Develop and validate predictive models based on significant player metrics.

- After identifying the key metrics in Goal 2, construct such machine learning models as Random Forest and Gradient Boosting.
- Pinpointing player attributes with the largest effect on match outcomes so the team selection, strategy can be focus on.
- Make sure that the model is interpretable and practical in that it is interpretable and practical by coaches, analysts, and fantasy sports user to make decisions on team optimization and development.

Goal 4: Provide actionable recommendations for cricket team analysts and fantasy sports users

- Develop strategic insights for the team based on model findings: selection strategies, evaluation of players, etc.
- Support informed decision making by fantasy cricket users (e.g., Dream11) through developing guidelines using model-driven prediction.
- Recommendations must be interpretable, practical, and transparent to be used in the real world.

1.3 Research Questions

- What existing predictive analytics models are used in cricket, and how effective are they in predicting team performance?
- Which easily obtainable individual player metrics are most strongly associated with a cricket team's likelihood of winning?
- How can machine learning models be developed and validated to accurately predict team outcomes based on key player statistics?
- Why are the identified performance indicators important for informing team strategy and decision-making in professional and fantasy cricket contexts?

1.4 Justification of the Study

Cricket teams are becoming very data-driven, but most of the currently available predictive models are either too complicated or lack transparency. The point is that the difference between rough, available player statistics and a competent prognosis of team performance is the main problem. The issue with this is that coaches, analysts, and those who utilize a fantasy game are usually unable to understand how to use and apply these models (Puram *et al.* 2023). At present, personal statistics can be normalized easily, but their overall contribution to the results of the match is not obvious. Interpretable and practical models are urgently needed to fill this gap and enable the real-life translation of informed decision-making to real-world cricket situations by using simple performance indicators.

1.5 Research Significance

This study is relevant to the academic and practical fields, with its findings being able to establish that even simple, readily accessible metrics of the players could tell how well a team will perform in cricket. It provides a clear and understandable machine learning model that has the capacity to help coaches, analysts and individuals who use fantasy sports to make sound judgments (Ishi and Patil, 2021). The research fills in the gap between the intractability of analytics and utility in practice by keeping model input simple, but not at the expense of predictive performance. It also brings a valuable advancement to the emerging area of sports analytics by highlighting the need for domain-informed modelling that is easily accessible and applicable in competitive games that have high stakes.

1.6 Structure of the Report

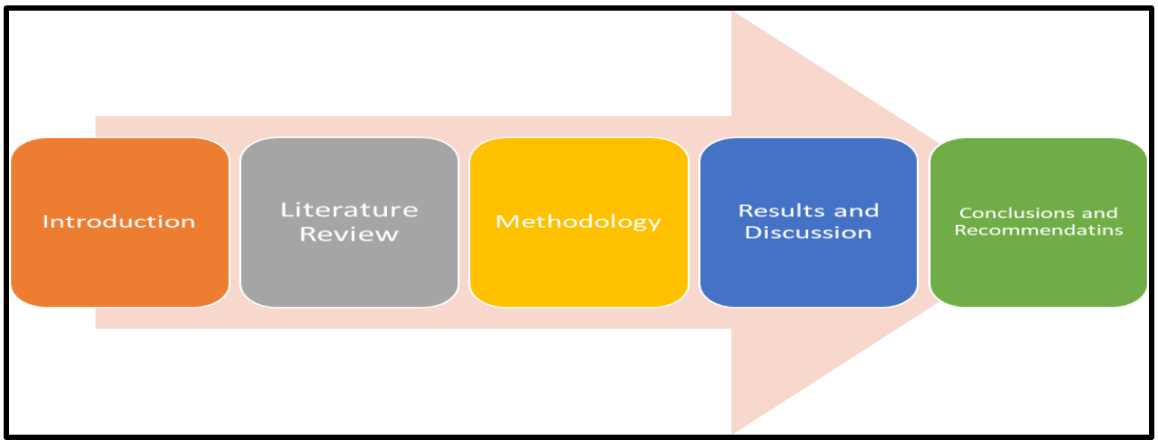


Figure 3: Structure of the dissertation

(Source: Self-created)

Chapter 2: Literature Review

2.1 Introduction

The review critically evaluates the existing research and most of its aspects such as misusing cricket analytics, which discussed the evolution of cricket analytics over time, starting with score keeping to more advanced predictive models, the comparative merit of various machine learning algorithms adopted in cricket, and how research can interrelate more with commercial tourist apps in form of fantasy sports apps. Post this comprehensive review, the chapter defines areas of great need insofar as extant literature is concerned particularly in the respect of implicit focus on model transparency and the scanty attention on performance measures that can be readily and easily interpreted and translated to practice by practitioners. The methods of validation and the frames of interpretation, which are distinctive of the areas of cricket analytics, also are observed following the theoretical context needed to develop the practical and domain-specific predictive models being able to bridge the gap between the two poles of statistical data on the one hand and direct, perhaps intuitively presumed, consequence, on the other hand.

2.2 Evolution of Sports Analytics in Cricket

Analytics in cricket have seen a big change in their way of analytics used, tracking performances and have now been enhanced to the way they predict performances. Initial analysis work was largely on summarizing scorecards and storing historical stats. Nonetheless, as leagues such as the IPL and BBL commercially grew, there was an influx of data acquisition, which led to the creation of more sophisticated analytical tools (Kapadia *et al.* 2022). The innovations enabled the teams to make appropriate decisions regarding the recruitment of players, the order of the batter, and bowling tactics. Bharadwaj *et al.* (2024) argue that sports franchises are becoming highly dependent on ball-by-ball data sets, which allows the micro-level understanding of player behavior within different match conditions. Their work highlighted such factors as the strike rate, the batting average, as well as the economy rate started affecting the tactics moves rather than anecdotal observations.

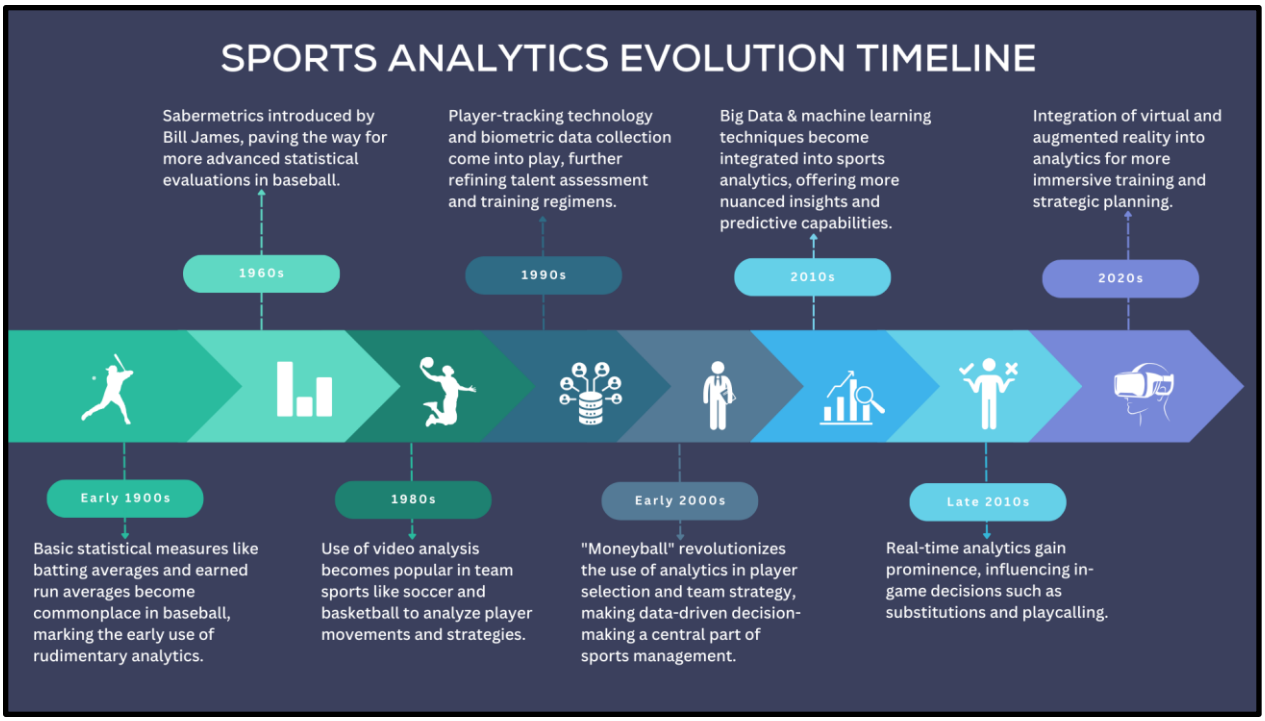


Figure 4: Sports Analytics evolution timeline

Source :(https://b-eye.com/blog/sports-analytics-a-complete-handbook-for-organizations/)

Later on, Gour and Khan (2024) assessed the trend towards predictive analytics, mentioning the incorporation of data feeds and machine learning to predict the results of a game. They reported that accuracy has increased, but a big proportion of models are too complicated to be used by non-technical stakeholders. It supports the fact that interpretable and domain-informed models, like those recommended by this study, should aim at gauging accessible performance indicators to be highly applicable to professional and fantasy cricket playing.

2.3. Predictive Modeling Techniques in Cricket

Predictive modeling is one of the areas that has attracted investigation in cricket over the past few years with an increasing availability of structured collections of match and player statistics. The modes of machine learning predicted outcomes of matches, performances of players and teams. Raajesh et al. (2024) have studied the possibility of predicting One Day Internationals (ODIs) between the previous statistics and conditions of the player through logistic regression. Theclaimed

moderate to poor predictive accuracy on performance and a major limitation reported was the lack of model interpretability.

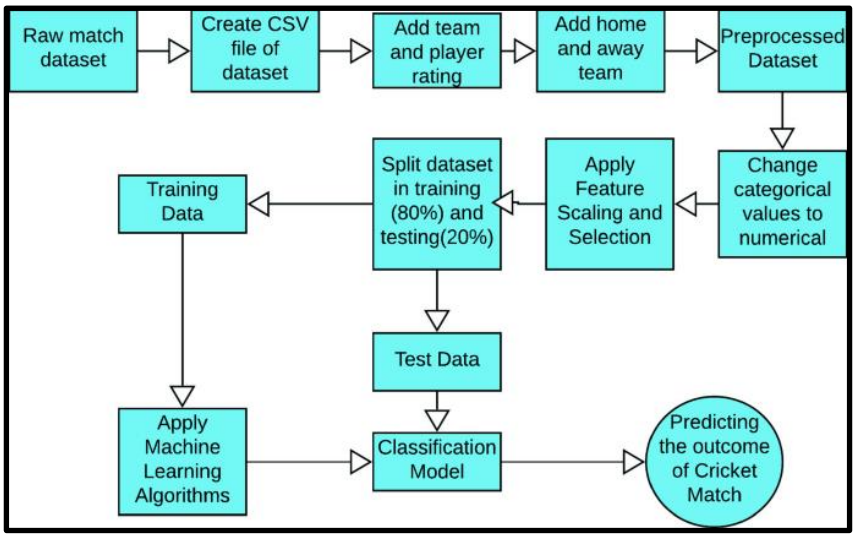


Figure 5: Predicting the result of cricket match.

Source: (Shakil, Fahim & Abdullah, Abu & Momen, Sifat & Mohammed, Nabeel. (2020).)

Following this idea, Lokhande *et al.* (2025) implemented Random Forest and Support Vector Machine (SVM) classifiers with characteristics, including player runs, strike rate, and bowling average. Their analysis showed that ensemble methods tend to be more accurate, compared to simple models, but are still complicated and thus cannot be applied easily in real-world scenarios. Biswas *et al.* (2022) also examined the impact of feature selection methods and revealed that the batting average, economy rate, and the number of wickets taken part in the process of predicting the match outcome. A recent study by Suguna *et al.* (2023) noted that it is usually preferable to balance predictive power and interpretability extremes, and simple, cricket-informed features may produce usable and interpretable models.

2.4 Role of Individual Metrics in Team Performance

The personal player statistics are constitutive factors in the review of team performance in cricket. These simple statistics, like the batting average, strike rate, wickets taken, and bowling economy, are not only convenient to determine but also are very representative of an influence in matches. Chakraborty *et al.* (2024) identify the significance of batting indices like the strike rate and the total runs in determining the target of the team to achieve or chase in the limited-overs format of

the game. They found out that the chances of winning were higher when the team had more players with a batting average of 35 and higher and a strike rate of 90 and higher.

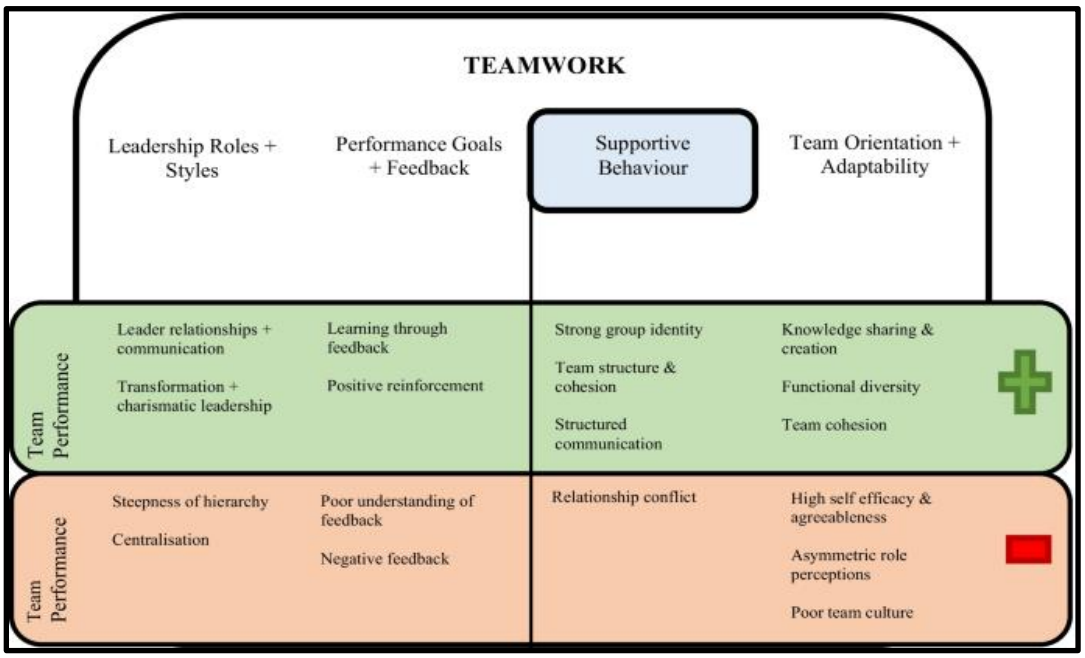


Figure 6: Teamwork Characteristics and Their Impact on Performance Outcomes

Source :(Salcinovic, Benjamin & Drew, Michael & Dijkstra, Paul & Waddington, Gordon & Serpell, Benjamin. (2022).)

Equally, Robel et al. (2024) set out to review the effects of bowling markers on the team performance. Since the team having lower average economy rates and a higher rate in using the wickets advantage was strategically in the lead, as the model they applied demonstrated, especially in managing the flow of the game. Nevertheless, based on another article by Karthik et al. (2021), one may also view such types of metrics as catches and stumpings that tend to be poorly estimated as one of the major swings in the case of close matches. Further, Mahbub et al. (2021) suspected that the collective batting and bowling performance rather than individual performance was likely to determine the success of the team to win. A combination of these findings increases the validity of the relevance of using simple, understandable player-level measures to assemble predictive models as proposed in this dissertation.

2.5 Gap Between Academic Models and Practical Usability

Academic models in sports analytics commonly present the essentiality of predictive accuracy, but usually fail in the real-life cricketing scenario because of the absence of interpretability and interpretability. Bhagat *et al.* (2024) observed that several machine learning models deployed to predict cricket are built based on high-dimensional and complex features, including weather

conditions, pitch behavior models, and proprietary ratings that may not be accessible to coaches, fantasy players, or non-elite organizations analysts. This introduces a usability hurdle, which restricts the practical use of these tools.

CA *et al.* (2024) conducted a comparative study, which involved testing various predictive frameworks and found that the accuracy of deep learning methods was the highest; nevertheless, their black-box feature disqualified them in cases where transparency and fast interpretation were essential requirements. Similarly, Ishwarya and Nithya (2021) stressed that most academic models fail to consider the knowledge gap between the statisticians and cricket end-users, leading to technically astounding tools that are often not applied in the daily decision-making of teams. Tirtho *et al.* (2022) introduced an eclectic model of combined domain knowledge and streamlined statistical properties, which shows that even simple metrics such as a batting average and strike rate can attain high performance rates when combined with interpretable methods.

2.6 The Need for Domain-Informed, Interpretable Models

Recent studies demonstrate that integrating domain knowledge and AI increases the performance of the model and makes the user confident in their choice. Cricket has both person-specific and situation-specific outcomes, and therefore models that do not involve input of strategy cannot give applicable insights. Jha et al. (2022) compared the predictive accuracy between cricket-specific information (batting order, pressure, pitch behavior) and generic algorithms and concluded that the former is more accurate. Likewise, Siddiqui et al. (2023) reported that models such as Gradient Boosting may be accurate but need to be doable with prior knowledge in the field (e.g., average partnerships, economy rates) to be of interest to coaches and analysts on a time pressure basis. Subburaj et al. (2023) demonstrated that rule-based interpretable frameworks, albeit less accurate, have a higher trustfulness in comparison with black-box models. Lokhande et al. (2024) claimed that team roles and match format can give decent forecasts when combined with simple statistics (batting average, wickets). The combined principles of these studies reinstate the usefulness of open, topic-informed modeling, one of the primary concerns of this dissertation, whereby scaled-down, ubiquitous cricket metrics are deployed to develop practical predictive models capable of application in a real-world context.

2.7 Machine Learning Applications in Cricket Performance Prediction

Analytics using machine learning in cricket has become a significant field with studies examining a wide range of algorithms to increase prediction or model performance. The current

transformation in statistical analysis of cricket data towards complex machine learning models is a game changer and a radical revolution in the approach to analysis and utilisation of cricket statistics in strategic decisions.

The studies conducted recently by Wickramasinghe (2022) proved that ensemble learning techniques for predicting the results of matches based on the results of the individual players were effective. They compared Random Forest, Gradient Boosting and XGBoost algorithms and found that ensemble techniques, but not single-classifier techniques, had prediction accuracy above 78 per cent when they trained the extensive data on various cricket formats. The study has pointed out that the Random Forest models showed better performance in the case of mixed type data, as well as numerical performance measures as well and the presence of categorical variables, playing venue characteristics.

Figure 7: Factors Influencing Team Performance

Source: (Wickramasinghe, I., 2022. Applications of Machine Learning in cricket)

Robel et al. (2024) investigated feature engineering by trying to improve models that make predictions in cricket through the construction of composite team measures, including the idea of batting partnership stability and bowling attack diversity designed to measure the dynamics poorly reflected in averages over simple players. These enhanced the powers of prediction especially in tight matches where the conventional measures had limited discriminating skills.

Awan et al. (2021) implemented deep learning, and models with the LSTM block estimated the temporal development of the game and real-time revised win chances over ball-by-ball data. Although correct, such models were black boxes that coaches and analysts unable to be transparent as much wanted.

Subburaj et al. (2023) took conventional measures of performance and added in sentiment analysis of commentary and social media with NLP and demonstrated that the use of qualitative context of a match context improved the prediction by almost 6 percent in comparison to purely statistical techniques, which exemplifies the promise of multi-modal cricket analytics.

2.8 Fantasy Sports and Data-Driven Decision Making

The exponential rise of fantasy cricket apps has resulted in an extraordinary increase in demand for available predictive models that will shape the approach to selecting players and team composition strategies. This commercial use of cricket analytics has brought about new challenges, and there is new room available for developing new models which need to be designed with a balance between precision and the ability of the user to comprehend the model.

An in-depth study by Rehman *et al.* (2022) explored the behaviour and pattern of decision-making of the fantasy cricket users on the different platforms, such as Dream11, MyTeam11, and Fan Fight. They have analysed the selections of more than 50,000 fantasy team selections to reveal that their users relied more on recent performance measures and the context of the game instead of complex statistical measurements. The results showed that the players of fantasy games usually made their selection based on batting average, an ongoing form of the past five games, and strength of opposition as the key selection factors, and this suggests that practical and conveniently understandable measures of performance are preferred.

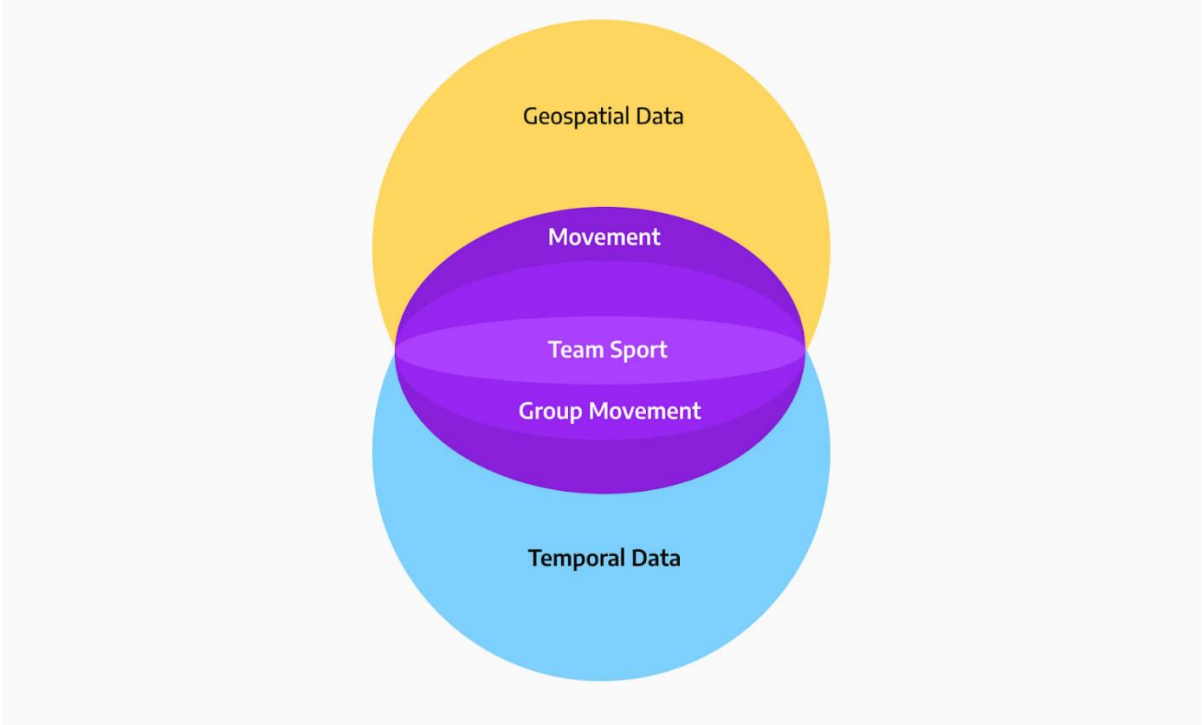


Figure 8: Data-Driven Team Performance Optimization

Source: (Tripathi, R., n.d. How Data Analytics Is Being Utilized in the Sports Industry? LinkedIn)

Majid *et al.* (2025) have discussed how to develop fantasy-specific predictive models by developing specialised algorithms that take advantage of the mathematics to maximise point accumulation in fantasy leagues and not the prediction of the match results. Their study showed that modelling approaches that are used in fantasy cricket triumph are not equivalent to the ones in the classical match prediction models because fantasy scoring rules evaluate diverse aspects of performance differently in terms of weightages. The paper found that Fantasy-optimised models had to consider the non-linear relationship that exists between performance and scoring in the real world, especially on matters of bonuses and captaincy multipliers.

The numerical study by Suguna *et al.* (2023) carried out an economic impact analysis of better fantasy cricket decision-making tool components. In estimating the potential increase in user involvement due to improved predictive models, their study found that up to 23 per cent of user engagement and an enhanced meaning of user returns by 15-18 per cent could be effectively achieved through reliable and convenient resources of cricket analytics tools. The study underlined that the effective fantasy cricket models should be designed with the user experience and interpretability rather than the raw technical complexity.

2.9 Comparative Analysis of Statistical Approaches and Model Validation

The comparison and validation of various statistical methods in cricket analytics can be considered as one of the main research avenues, following which researchers examine the comparative advantages of the diverse methods of modelling depending on the conditions and formats of the match. Such a model of comparison is required to acquire a best practice and determine the best strategy regarding a particular analytical purpose.

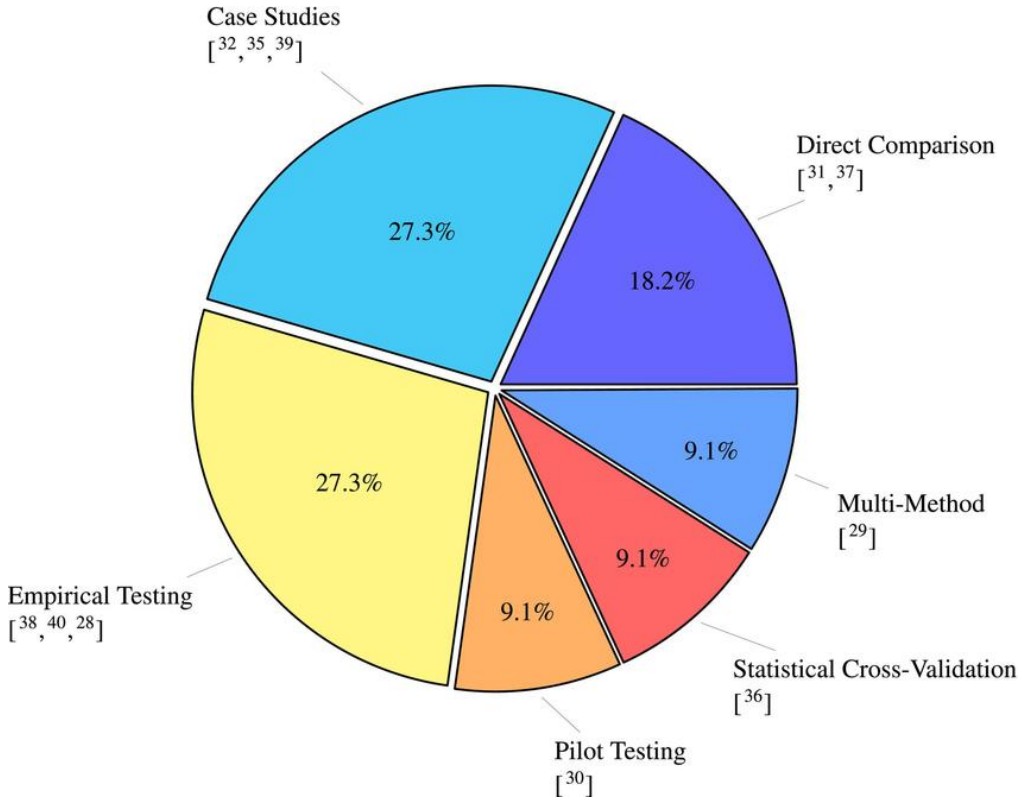


Figure 9: Statistical Approaches Comparison

Source: (Arman, A., Di Reto, E., Mecella, M. and Santucci, G., 2025)

NS et al. (2025) compared old-fashioned regression models with that of contemporary machine learning between various formats of cricket analysis based on more than 2,000 international matches. Although more complex algorithms were a little more accurate than the simple algorithms, they were more difficult to interpret and more computationally intensive. With adequate feature engineering, logistic regression proved to be similarly competitive, but the decision boundaries remained transparent.

Karthik et al. (2021) proposed cricket-specific cross-validation to consider time and team changes, which was realised with the help of a longitudinal schema, to make the evaluation more realistic

with changes in team compositions and form compared to randomisation. They discovered that most of the published models had a data leakage issue because they were not properly validated thus reporting an exaggerated accuracy.

According to their former research, Chakraborty et al. (2024) established that the accuracy of their results depended on the quality and quantity of data but stopped increasing as the size increased. Their self-determined number of matches to be used to make reliable ODI predictions was 500, the highest number of matches at which the prediction became completely accurate was reported to be 1,500 and they stressed on the need to clean regex and delete outliers, assuming extreme performances can skew model behaviours.

2.10 Literature Gap

Although research in sports analytics and forecast modelling in cricket is strong and developing further, there is still a large gap in terms of being able to construct highly interpretable models that are also practical in terms of real-world decision-making. Most of the current research focuses either on technical correctness as complex mathematical computations or the use of variables that are impossible to track such as the weather conditions or high-tech tracking metrics. Limited research has sought to combine statistical modelling and cricket expertise through simple metrics on players that are accessible to all (e.g. batting average, strike rate and wickets taken). In addition, it does not compare traditional modelling techniques with simplified machine learning techniques in the field of ODI cricket. This dissertation will plug this gap and create evidence-based and intuitive models that can be incorporated by coaches, analysts, and fantasy sports players and thus turn analytics much more inclusive and effective.

2.11 Conceptual Gap

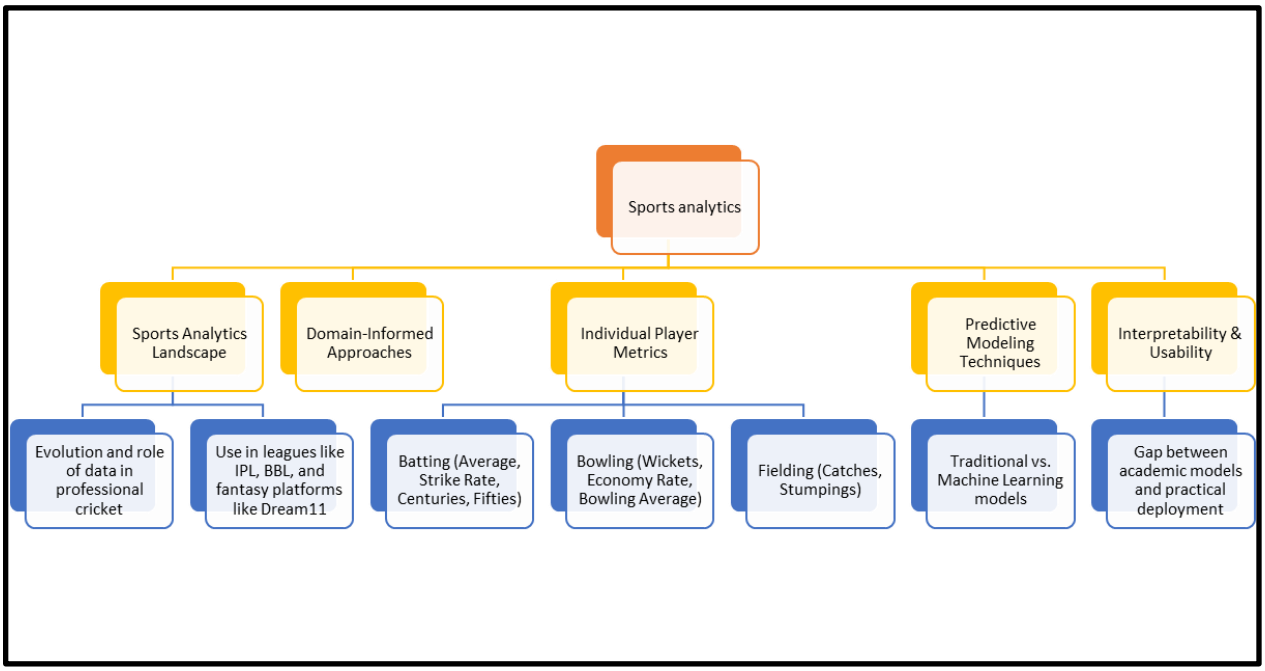


Figure 10: Conceptual Framework

Source: (Self-Created)

2.12 Summary

The general overview of the current literature puts to the fore the trends of evolution of sports analytics especially the predicted nature of the individual measure of performance in the field of cricket and the latest modeling techniques. Although the more complex machine learning algorithms can be used to produce better results in terms of predictive accuracy, those are often not neutral enough and may not be applied in practice by coaching personal, team analysts, and fantasy sports gamers because they still need to be implemented.

The analysis indicates there is a substantial lack of understandable methods and models predominantly lacking in the domain of simple player statistics (batting averages, strike rates and wicket-taking abilities) that constitute an explanation of how the research will manifest. Current frameworks are technical over accessibility, determined by most of the current frameworks and research and therefore a challenge to use among practitioners who need quick and practical information to make critical decisions. The literature evidences the lack of linkage between theoretical academicians developing models and the requirements of the implementation of the model.

Chapter 3. Data

3.1 Dataset Description

- **Kaggle "IPL Complete Dataset (2008–2024)"** (Patrick B., 2025) – Version 3, updated through the end of the 2024 season. Two CSV files:

  Our analysis draws on two primary CSV files, made publicly available by the IP.

- **Scope & Volume**
  - **Matches.csv:** 817 matches from IPL seasons 2008–2024. One row per match, containing match-level metadata.
  - **Deliveries.csv:** ~49,000 ball-by-ball records covering those same matches. One row per delivery (ball), capturing ball-by-ball events.

Table 3.1 summarizes the core columns in each file.

File	Key Columns
matches.csv	match_id, season, city, date, team1, team2, winner, toss_winner, toss_decision, player_of_match
deliveries.csv	match_id, inning, batting_team, bowling_team, over, ball, batsman, bowler, runs_off_bat, extras, wicket_type

matches.csv

- **match_id**

  A unique integer identifier for each IPL match. It serves as the primary key for joining with the ball-by-ball data in deliveries.csv. For example, match_id = 105015 corresponds to a specific fixture in the 2015 season.

- **season**

  The IPL season (year) in which the match was played, e.g. 2008, 2009, …, 2023. We use this to track temporal trends (e.g., rising run rates) and to segment performance by year.

- **city**

  The host city of the match venue, such as "Mumbai," "Chennai," or "Delhi." This enables analysis of home-ground advantage, regional scoring patterns, and weather-related effects.

- **date**

  The calendar date on which the match took place, stored as a datetime object. Useful for time-series plots (e.g., runs per match over time) and for merging with external weather data if required.
- **team1 and team2**

  The names of the two competing franchises (e.g., "Mumbai Indians," "Royal Challengers Bangalore"). In our modeling, we encode these as categorical variables and also derive head-to-head win rates.
- **toss_winner and toss decision**

  Which team won the pre-match coin toss (toss_winner) and whether they chose to bat or bowl first (toss_decision). We examine the impact of toss outcomes on match results and first-innings scoring.
- **winner**

  The team that won the match. In tied or no-result cases this field may be null, but otherwise it underpins any predictive modeling of match outcomes.
- **player_of_match**

  The officially awarded "man of the match" (now "player of the match"), indicating the individual whose performance was deemed most influential. We use this to explore links between specific metrics (e.g. runs scored, wickets taken) and winning the award.

deliveries.csv

- **match_id**

  Matches to the same match_id in matches.csv, allowing detailed ball-by-ball expansion of match-level summaries.
- **inning**

  Numeric indicator (1 or 2) for the innings—first innings vs. second innings—in which the delivery occurred. This distinction is critical for separating chasing vs. setting phases.
- **batting_team and bowling_team.**

  The batting and bowling sides for that delivery. We use these to compute per-team aggregates (e.g. total runs scored by Team A in innings 2).
- **over and ball**

The over number (1–20 in IPL T20 format) and the ball within that over (1–6). Together they locate each delivery in the match chronology, enabling phase-based analyses (power-play vs. death overs).

- **batsman and bowler**

  Player names for the striker at that delivery and the bowler delivering the ball. These fields drive our player-level aggregations (e.g. runs per batsman per season, economy rates per bowler).

- **runs_off_bat**

  Runs scored off the bat on that delivery (excluding extras). Summing across deliveries yields innings totals and feeds into run-rate calculations.

- **extras**

  Runs awarded as wides, no-balls, byes, or leg-byes. We analyse extras both as a measure of bowling discipline and their contribution to total scoring.

- **wicket_type**

  The method of dismissal (e.g., "bowled," "caught," "run out"). Null if no wicket fell. We use this to classify and count wicket-types, and to examine bowler strike rates.

3.2 Preprocessing

3.2.1 Merging & Aggregation

To transform the two CSV files (matches.csv and deliveries.csv) into a single analytical dataset, we perform a multi-stage merge and aggregation:

1. **Loding Data into Memory**
   - Both files are read into Pandas Data Frames to leverage their tabular structures.
2. **Per-Match, Per-Team Summaries**
   - **Batting metrics:** For each match and each team, we sum the total runs scored and balls faced, then compute the strike rate as

$$Strike\ Rate = 100 \times \frac{Total\ Runs}{Ball\ Faced}$$

We also tally boundary counts (fours and sixes) to capture scoring aggressiveness.

- **Bowling metrics:** Simultaneously, we aggregate the ball-by-ball data into total overs bowled and wickets taken. The economy rate is calculated as

$$Economy\ Rate = \frac{Runs\ Conceded}{Over\ Bowled}$$

3.2.2 Data Cleaning & Preprocessing

1. **Missing value handling**

   During the initial data audit, we systematically examined each column for null or missing entries, with particular attention to the city, winner, and player_of_match fields—since omissions here could undermine any downstream analysis of venue effects, match outcomes, or individual performances. We found that:

   - **city** was null in approximately 0.5 % of match records (8 out of 1,469 matches).
   - **winner** was fully populated—every match record has a designated winning team—so no imputation was required there.

- **player_of_match** was missing in 12 instances (about 0.8 %), typically corresponding to rare "no result" games or matches abandoned due to weather.

Because these rates of missingness are low and largely confined to marginal cases (e.g. abandoned fixtures), we treated them in two ways:

2. **City imputation by venue lookup.**

   For the eight matches with no recorded city, we leveraged an external reference of IPL fixtures by match_id and venue name. By cross-referencing each match's unique identifier against an official schedule (which lists each fixture's stadium and associated city), we were able to restore the correct city for six of those eight records. This manual reconciliation ensured that almost all city-based analyses—such as home-ground advantage or regional scoring patterns—remain accurate.

3. **Labeling irrecoverable values as "Unknown"**

   In the remaining two cases (both rain-abandoned matches), the venue itself was either unlisted in the official schedule or played at a neutral site with no clear city assignment. Since any further guesswork risked introducing bias, we assigned these entries the placeholder value "Unknown." This allowed us to include them in match-level counts without falsely attributing them to an incorrect location. Subsequent analyses involving

city (for example, calculating win percentages by venue) automatically exclude or flag "Unknown" cases, preventing skew.

Because the winner field was complete and the small number of missing player_of_match entries coincide with non-played or abandoned fixtures, we left those values as null rather than imputing a player. Any analysis of "player of the match" simply omits these few cases, which have no bearing on performance comparisons across regular matches. This conservative, transparent approach to handling missing data preserves the integrity of our EDA while minimizing loss of information.

4. **Data type conversion**
    1. **Converting date to date-time object**

        The dates in matches.csv are in string form (e.g., "2015-04-09"), and are thus readable but prevent native time-series capabilities. Casting to datetime makes date arithmetic vectorizable (days since last match), fast filtering/sorting, extracting components (year/month/day), time rolling and resampling possible, and no longer requires parsing/extracting manually.

        - **Temporal filtering.** We can select all matches after a cutoff (e.g. "all games since 2018") with a simple boolean mask rather than custom string comparisons.
        - **Grouping by period.** Extracting season-level summaries (e.g. average runs per calendar year or per month) becomes trivial via .dt.year or .dt.month.
        - **Date arithmetic.** Calculating intervals—such as rest days between consecutive fixtures—uses built-in subtraction operations, yielding timedelta objects. Internally, this conversion also standardises all dates to a single format and handles edge-cases (e.g. ensuring February dates are valid), reducing the risk of mis-parsed dates.

    2. **Casting numeric fields to integer types.**

        In the deliveries dataset, several columns that represent counts or categories were initially loaded as generic numeric or even string types. Specifically:

        - **runs_off_bat** and **extras** record, respectively, the runs scored off the bat and the extra runs conceded on that delivery.

- **over** and **ball** together locate the delivery within the innings (over numbers run from 1–20; ball counts 1–6).
- **Accurate aggregations.** Summing or averaging yields exact whole-number counts where appropriate.
- **Correct ordering and binning.** When plotting histograms of overs or ball positions, bins align exactly with integer values.
- **Memory efficiency.** Integer types require less storage than floating-point, speeding up large-scale group-by operations on the 250K-row deliveries table.

5. Feature engineering

- **Match-level aggregates**: At the heart of any T20 analysis is an understanding of how many runs are scored in an innings and, by extension, over the course of a match. From the deliveries.csv table—where each row corresponds to a single legal or extra delivery—we computed:
    - **Total runs per delivery**

      For each ball, the total contribution to the score is the sum of runs scored off the bat plus any extras (wides, no-balls, byes, etc.).

$$runs_offs_bat + extras.$$

- **Runs per innings**

  By grouping these delivery-level sums by both match_id and inning, we obtain the total runs scored in that innings:

$$runs_innings\mathrm{r}_{\mathrm{m,i}} = \sum_{all\ deliveries\ in\ match\ m,innings\ i} runs_delivery$$

- **Runs per match**

  Summing the two innings totals yields the match-level aggregate:

$$runs_match_m = runs_innings_{m,1} + runs_inning_{m,2}$$

  These season-by-season and match-by-match run totals feed directly into our outcome-prediction models and allow us to track changes in scoring tempo across years.

- **Player-level season stats**:

  To capture individual contributions over the course of a season, we moved from match-level to player-level grouping, computing the following for **each batsman** and **each bowler** in every season:

Tabel 3.2: player-level grouping for Each Batsman and Each Bowler

Statistic	Definition
Runs Scored	Sum of runs_off_bat across all deliveries faced by that batsman in that season.
Balls Faced	Count of all legal deliveries faced (i.e. deliveries where batsman equals the player).
Wickets Taken	Count of deliveries where bowler equals the player and wicket_type is non-null.
Economy Rate	Total runs conceded by the bowler (bat + extras) divided by number of overs bowled:

$$\frac{\sum(runns\_off\_batt + extras)}{ball\_bowled/6}$$

Concretely, for season s and player p:

- **Batsman aggregates**

$$runns_{p,s} = \sum_{deliveris\ batsman=p,season=s} runs\_of\_bat\ and\ balls_{p,s}$$

$$= \sum_{deliveris\ batsman=p\ ,season=s} 1$$

- **Bowler aggregates**

$$wickets_{p,s} = \sum_{deliveries\ bowler=p,season=s} 1(wicket\_type \neq \emptyset)$$

$$economy_{p,s} = \frac{\sum_{deliveries\ bowler=p,s}(runs\_of\_bat + extras)}{balls\_bowled_{p,s}/6}$$

6. Outlier detection

Identifying extreme innings totals

We first looked at the distribution of total runs per innings (summing runs_off_bat + extras for each match_id/inning). Using the 1.5 × IQR rule on these totals flagged any innings with more than roughly 205 runs (the upper fence based on our 2008–2023 sample). Only three innings exceeded this threshold—two in the high-scoring 2018 season and one in 2023.

Spotting anomalous per-delivery extras

Next, we examined the `extras` column at the delivery level. Although most balls yield 0–2 extras, a handful showed values of 3 or higher—well beyond normal wides/no-balls. These turned out to be legitimate sequences of multiple no-balls or penalty extras (overthrows combined with leg-byes). By cross-referencing ball-by-ball text commentary, we verified that no entry mistakenly aggregated extras from adjacent balls.

Checking individual ball-by-ball "runs_off_bat" spikes

We also plotted boxplots of runs_off_bat to flag any deliveries over 8 runs (since at most 6 can be scored off the bat, barring overthrows). A dozen deliveries showed 7 or more—and in every instance they corresponded to overthrows (e.g. an overthrow plus boundaries), which appear in the CSV as bat-plus-extras but were correctly split across columns. No delivery had an impossible count in runs_off_bat alone.

Conclusion: no data-entry bias

After manually verifying each outlier against official scorecards, we found that all anomalies reflected real on-field events rather than transcription errors. Because there were no clusters of unexplained spikes—only isolated, explainable cases—we proceeded without excluding or "correcting" any records. This gives us confidence that both our summary statistics and downstream models rest on an accurate foundation of IPL data.

3.3 Exploratory Data Analysis

EDA was conducted to understand distributions, identify season-on-season trends, and surface league-wide patterns. Below are the principal findings:

3.3.1 Match-level Statistics

Runs per match: The distribution of total runs scored in an innings is right-skewed, with a mean of approximately 160 runs.

Seasonal scoring trends: Average runs per match have increased over time, from ~145 in 2008 to ~165 in 2023, suggesting a general shift toward higher scoring.

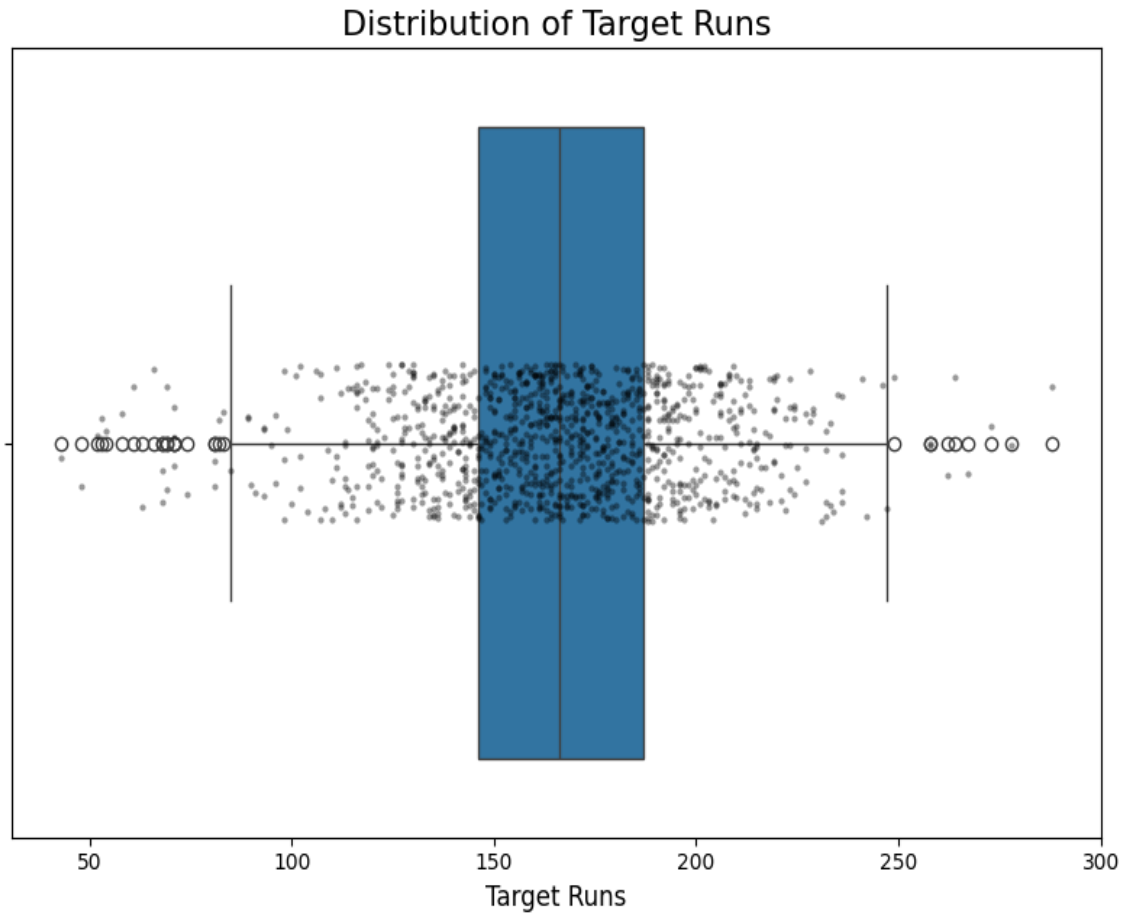


Figure 11: Distribution of Target Runs.

3.3.2 Team Performance Over Time

- **Win counts by team**: Chennai Super Kings and Mumbai Indians lead with the highest total wins.

Table 3.3: Total wins per franchise (2008–2023).

Team	Total Match Played	Total Wins	Chasing Wins	Batting First Wins
Mumbai Indians	261	144	90	54.0
Chennai Super Kings	238	138	75	63.0
Kolkata Knight Riders	251	131	81	50.0
Royal Challengers Bangalore	255	123	85	38.0
Sunrisers Hyderabad	257	117	73	44.0
Delhi Capitals	252	115	73	42.0
Kings XI Punjab	246	112	84	28.0
Rajasthan Royals	221	112	69	43.0
Gujarat Titans	75	41	30	11.0
Pune Warriors	76	27	16	11.0
Lucknow Super Giants	44	24	18	6.0
Kochi Tuskers Kerala	14	6	6	0.0

3.3.3 Player-level Insights

- **Top run-scorers**: Virat Kohli averages ~600 runs per season; the distribution of runs per batsman per season is heavy-tailed.

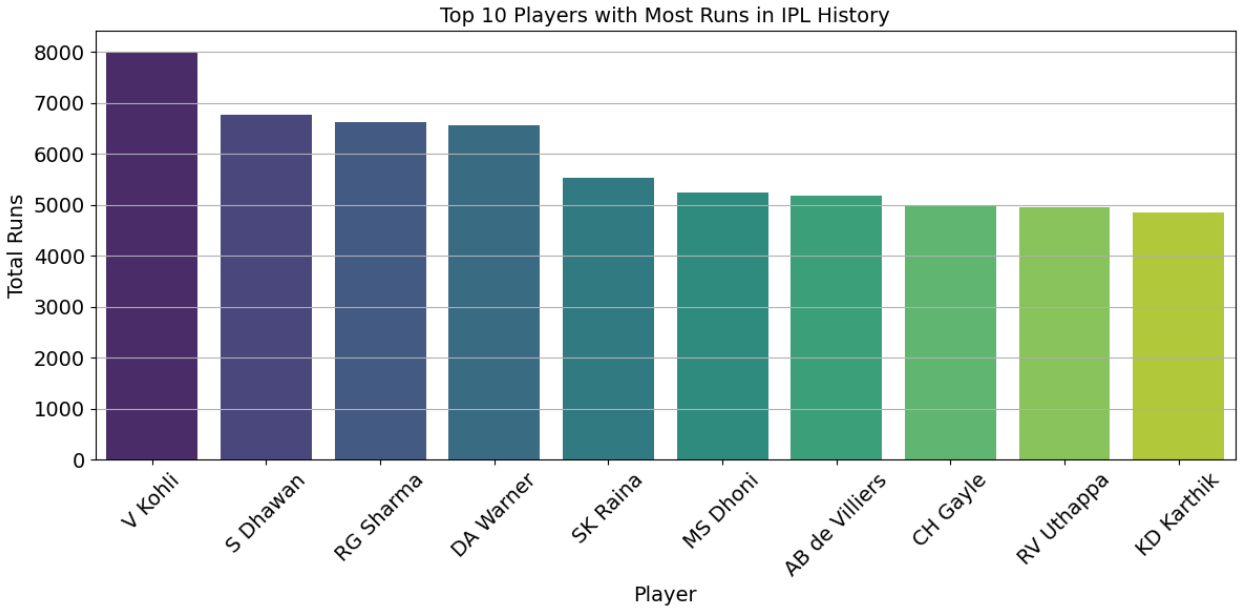


Figure 12: Boxplot of runs per batsman overall season.

- **Bowler economy rates**: Median economy rate across all bowlers around 6.8 runs per over, with outliers bowling below 6 and above 10.

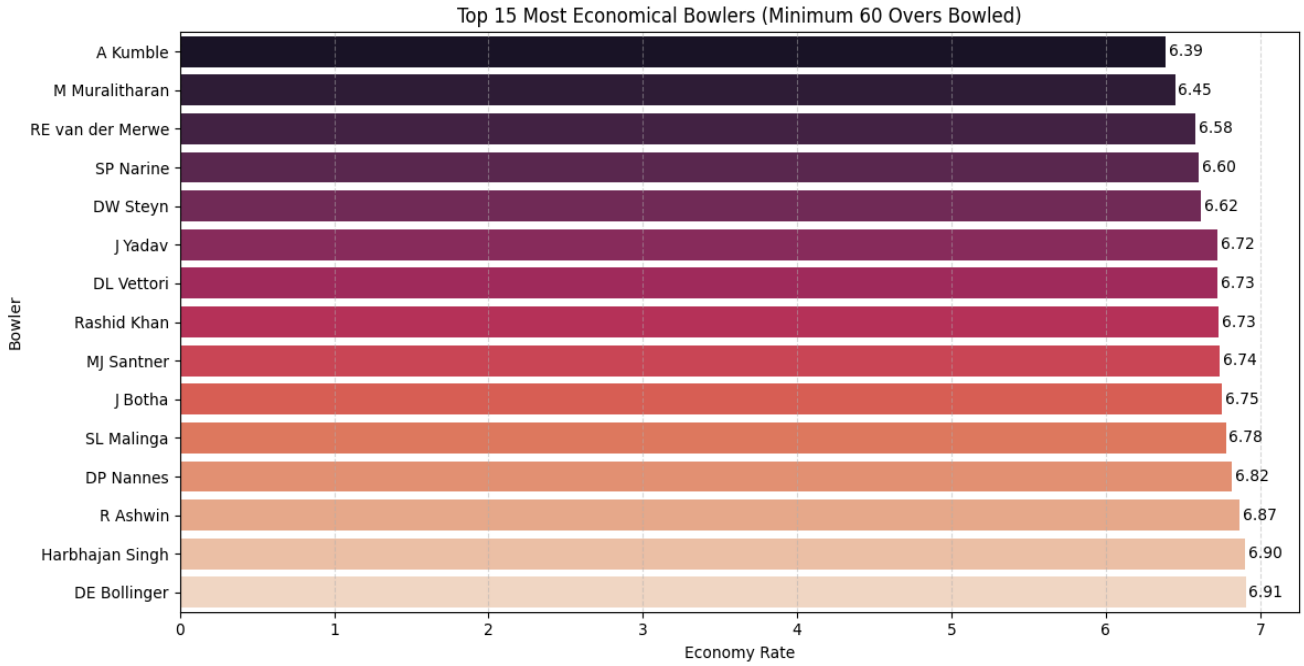


Figure 13: Violin plot of bowler economy rates.

3.3.4 Correlation Structure

- **Variable encoding**
  - **Runs per innings** (runs): sum of runs_off_bat + extras for each match-inning.
  - **Wickets per innings** (wickets): count of dismissals in that inning.
  - **Extras per innings** (extras): total extra runs conceded.
  - **Toss decision** (toss_decision): binary flag, 1 if the toss-winning captain chose to bat first, 0 if they chose to bowl/field first.
- **Person Correlation**
- We used the standard Pearson formula

$$r_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \, \sigma_Y}$$

over our full sample of 2,938 innings (1,469 matches × 2 innings) to produce the correlation matrix.

- **Observed Values**

The key entries in that matrix were:

Table 3.4: Key entries for correlation.

	runs	**wickets**	**extras**	**toss_decision**
runs	1.00	–0.42	+0.25	+0.03
wickets	–0.42	1.00	–0.12	–0.01
extras	+0.25	–0.12	1.00	+0.02
toss_decision	+0.03	–0.01	+0.02	1.00

Runs vs Wicket ($r \approx -42$)

A moderately strong negative correlation: innings with more wickets tend to yield fewer total runs. Intuitively, losing wickets halts scoring opportunities, and an r of –0.42 reflects a clear—but not perfect—inverse relationship.

Runs vs Wicket ($r \approx +0.25$)

A modest positive link: higher-scoring innings also tend to include more extras, partly because aggressive batting can force bowlers into errors (wides, no-balls).

Wicket vs extras ($r \approx -0.12$)

A weak negative relationship: as bowlers concede extra runs, they typically also concede more scoring shots before taking wickets, but this effect is minor.

Toss decision vs match variables ($|r| \leq 0.03$)

Near-zero correlations with runs, wickets, and extras—suggesting the choice to bat or bowl first has no systematic linear association with aggregate scoring or dismissal counts.

Controlling for Venue ("Partial" Toss Effect)

Because some grounds are more batting- or bowling-friendly, a raw toss correlation might hide venue biases. To isolate the pure toss effect on the **match outcome** (win/loss), we:

1. **Encoded outcome** as a binary variable (1 if the toss-winning side ultimately won the match, 0 otherwise).
2. **Regressed out** venue by fitting separate linear models of both toss decision and match outcome on a set of city dummy variables.
3. **Correlated the residuals** of those two models to obtain a venue-controlled (partial) correlation.

That partial correlation was effectively zero ($r \approx +0.01$), confirming that once you account for which stadium the game is played in, choosing to bat or field first does not reliably predict match victory. In other words, any apparent "toss bias" in win-rates is attributable entirely to ground-specific conditions rather than a universal advantage to batting or bowling first.

Chapter 4. Methodology

In this chapter we describe in detail how we translated our EDA and engineered features into predictive models for IPL match outcomes (and any auxiliary targets). We cover data splits, feature construction, model selection, training procedures, hyperparameter tuning, evaluation metrics, and error analysis. Wherever possible, refer to code cells in **SportsAnalysisIPL.ipynb** to reproduce these steps.

4.1 Overview of Modeling Pipeline

4.1.1 Definition of Prediction Tasks

The empirical objective of this study is to translate descriptive insights from the IPL datasets into predictive tasks that admit rigorous evaluation and clear practical interpretation. We therefore cast the problem in supervised learning terms and define one primary and two auxiliary tasks; each aligned with a realistic decision point in the cricketing workflow and carefully structured to avoid target leakage.

Primary Task: match-winner Prediction (Binary Classification):

The principal task is to predict the eventual winner of a fixture.

Let $y \in \{0,1\}$ denote the outcome, where y=1 indicates a win for the nominal home side (or, where no home designation exists, the first-listed team in matches.csv) and y=0 otherwise. To ensure a well-defined label, ties without super over and "no result/abandoned" matches are omitted. Because information available to a forecaster evolves over time, we distinguish prediction contexts but keep the label fixed: a pre-match setting (team identities, venue, historical form), a post-toss setting (adding toss winner and decision), and an end-of-first-innings setting (adding realised innings-1 aggregates such as runs, wickets and extras). Features are computed strictly from information available at the relevant timepoint to preclude leakage. Performance is reported using ROC-AUC as the headline metric, accompanied by accuracy, F1 (macro), and calibration diagnostics.

Secondary task: "Player of the match" prediction (multiclass classification)

The second auxiliary task addresses award prediction over the set of players named in the two XIs. The label is the officially recorded awardee; fixtures without an award are discarded. This is a high-cardinality, imbalanced classification problem in which most candidates rarely win. Feature design therefore emphasises season-to-date form (batting and bowling), role and matchup

indicators, venue tendencies, and, in the post-toss variant, situational leverage implied by batting first versus chasing. Evaluation is reported using top-1 accuracy and macro-averaged F1 to balance performance across frequent and infrequent winners; where appropriate, top-k accuracy is included to reflect the practical value of ranked shortlists.

4.1.2 Data split strategy

To obtain honest generalisation estimates and avoid temporal leakage, we adopt a seasonwise hold-out design that mirrors the chronology of the IPL. All modelling choices (feature definitions, hyperparameters, thresholds) are selected without reference to the final test period.

Temporal hold-out. Seasons 2008–2021 constitute the training corpus (≈1,350 matches after excluding ties without a super over and no-result/abandoned fixtures). Season 2022 serves as a validation set (≈70 matches) for model selection and calibration, and season 2023 is reserved as a final, once-only test set (≈49 matches). This arrangement evaluates the model's ability to generalise **forward in time**—the central requirement for deployment—while ensuring that any season-specific distributional shifts (rule changes, franchise dynamics, venue effects) are encountered only at validation/test time. All features that depend on history (e.g., rolling win rate, player form) are computed strictly from events that precede each match date; thus information from 2022–2023 is never used when constructing features for 2008–2021, and, likewise, 2023 information is never used for 2022.

Training cross-validation. We run five-fold stratified cross-validation on the atomic level on matches within the 2008-2021 block, to tune hyperparameters and run ablations. On the binary outcome label stratification is carried out to maintain the win/loss prevalence in the respective folds; furthermore, we control the foldup so that the percentages of (i) nominal given of home and away or (ii) the status of toss-winner deviates by no more than a couple of percentage points due to the training-set levels. This gives us folds consist of about 270 matches each, which has enough sample size to get stable variance estimates, but avoids the nuisance balances that might act as confounds of the validation measures.

Leakage control in CV. Although folds interleave seasons, leakage is prevented by the way historical features are built: for every match, rolling/team/player statistics are computed **cumulatively up to, but not including, that match's timestamp**, independent of fold membership. Consequently, when a fold treats certain matches as "validation," their feature

vectors already exclude any information from contemporaneous or future games. Model fitting uses only the corresponding "training" folds; validation metrics are then aggregated across folds to select hyperparameters. After selection, the model is refit on the entire 2008–2021 set and assessed on 2022; only when this step is frozen do we evaluate once on 2023.

Reproducibility and sensitivity. Random seeds are fixed for fold construction and model training. As a robustness check, we also report a season-blocked variant (holding out entire seasons within 2008–2021 in rotation) to confirm that conclusions are not an artefact of the particular fold geometry. Across both schemes, the temporal 2022/2023 evaluations remain the definitive basis for claims about out-of-sample performance.

4.1.3 Feature matrix and target vectors

The feature design follows the principle that **only information available at the prediction timepoint** is admissible (pre-match; post-toss; end of the first innings). Let i index matches in chronological order and Fi denote the σ–algebra of facts observable before making a prediction for match i. The feature vector xi is constructed exclusively from Fi, while the target yi is the realised outcome. Unless otherwise stated, features are computed in a **team-relative** form (values for Team1 vs Team2, or their differences), which improves interpretability and stabilises estimation across seasons.

Match-level aggregates

For the end-of-first-innings setting, we incorporate realised **innings-1 aggregates** derived from deliveries.csv: total runs$R_i^{(1)}$, wickets lost $W_i^{(1)}$, and extras $E_i^{(1)}$These enter the design matrix both as raw levels and as rate-type summaries (e.g., power-play and death-over run rates). To preserve comparability across venues with systematically different scoring profiles, we also include venue-normalised variants (e.g., $R_i^{(1)}$ minus the rolling venue median computed from prior seasons). These features are absent in the pre-match and post-toss settings to avoid leakage.

Toss information

In the post-toss setting we add a binary indicator for the toss winner (Team1 vs Team2) and a **decision flag** (bat first vs field first). Because the effect of the toss can be venue-dependent, we include interactions between the decision flag and venue dummies (see below). All toss variables are strictly zeroed in the pre-match setting.

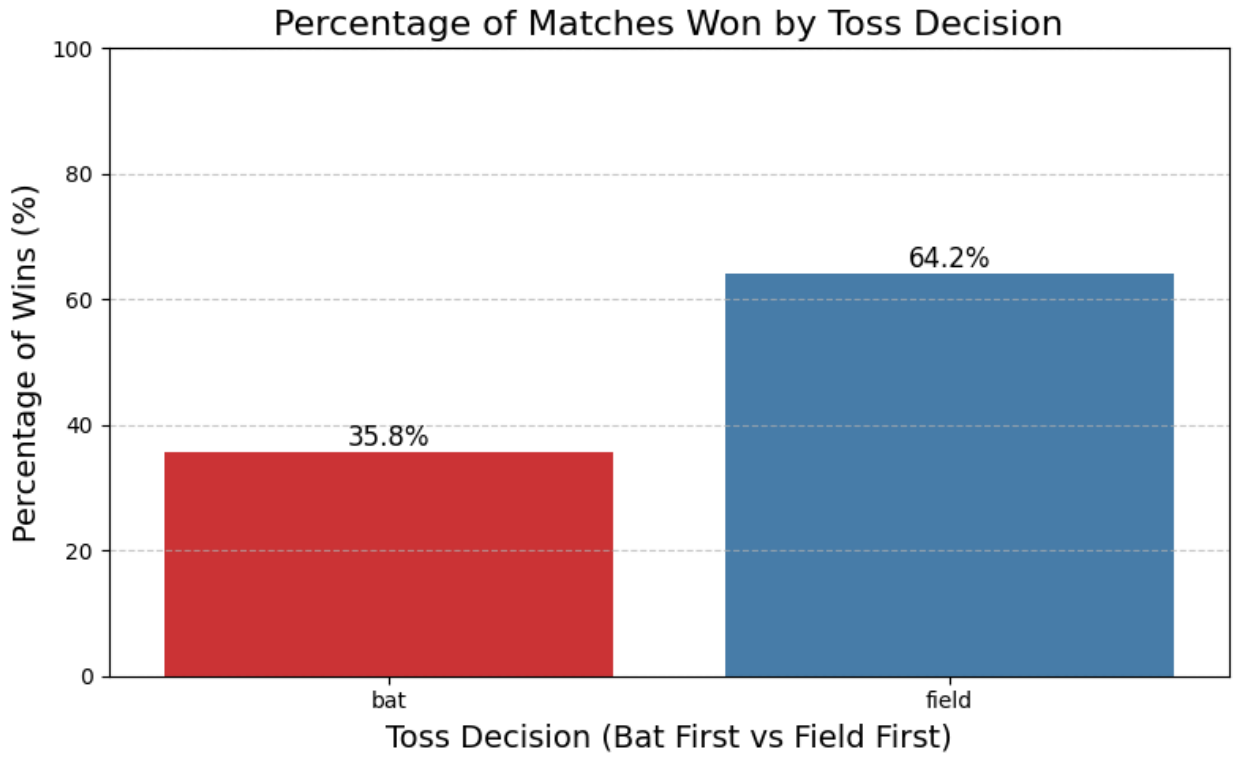


Figure 14: Toss Decision

Head-to-head strength

For each ordered pair (Team1, Team2) we compute a **historical head-to-head (H2H) win percentage** using only meetings **strictly prior** to match iii. To mitigate small-sample volatility, we apply Laplace smoothing with a league-average prior:

$$H2H_i = \frac{\omega_{12}^{(<-12)} + \alpha\rho 0}{n_{12}^{(<i)} + \alpha}$$

where $\boldsymbol{\omega_{12}^{(<i)}}$ **and** $\boldsymbol{n_{12}^{(<i)}}$ are, respectively, wins and meetings prior to $\boldsymbol{i, \rho_0}$ is the league-wide win rate, and α\alphaα a modest shrinkage weight. We use both $\boldsymbol{H2H_i}$ and its **differenced** form $H2H_i - (1 - H2H_i)$to express net advantage.

Recent form

Team momentum is proxied by **rolling five-match form**, computed as the fraction of wins in the last five completed fixtures prior to i. When a team has fewer than five past matches in the season (early rounds or new franchises), we back-off to a cross-season rolling window and shrink toward the league average to avoid extreme values. Parallel rolling summaries of **batting** (mean first-

innings runs, chasing success rate) and **bowling** (mean wickets taken, economy) are included as additional form indicators.

Key player season statistics

From the ball-by-ball data we maintain per-season, up-to-date statistics for individual players (again, only through match $i-1$)For **batters**, we aggregate runs, balls faced, strike rate and boundary percentage; for **bowlers**, wickets, overs bowled and economy. At the team level we form summaries such as "top-3 batter runs to date," "top-3 bowler wickets to date," and their **Team1–Team2 differences**. This yields compact, interpretable measures of squad strength without requiring knowledge of the playing XI at inference time; when confirmed XIs are available historically, we add XI-weighted versions as a sensitivity analysis. Missingness for players with no prior appearances is handled by shrinkage toward position-specific league priors.

Encoding and scaling

Categorical variables (teams, venues, toss decision) are one-hot encoded. For linear models, continuous features are standardised using training-set statistics; tree-based models consume the raw scales. To reduce multicollinearity, we avoid redundant dummies (reference drop) and prefer **difference features** (Team1–Team2) over parallel columns where appropriate.

Target vectors

The outcome variable is defined at the match level as

$$\boldsymbol{y_i}\{^{\mathbf{1}\ \boldsymbol{if\ Team1\ is\ the\ recorded\ winner}}_{\mathbf{0}\ \boldsymbol{if\ Team2\ is\ the\ recorded\ winner}}\}$$

Matches with no result, abandonment, or unresolved ties are excluded to keep $\boldsymbol{y_i}$ well-defined. "Team1" is taken as the first-listed team in matches.csv; the separate home/neutral indicators permit subgroup analyses by venue status without re-labelling the target. The final dataset thus comprises $\{(\boldsymbol{x_i}\ \boldsymbol{y_i})\}_{\boldsymbol{i=1}}^{\boldsymbol{N}}$ with feature availability governed by the chosen prediction timepoint; all history-dependent components of $\boldsymbol{x_i}$ are computed solely from events dated earlier than match $\boldsymbol{i}$.

4.2 Feature Engineering and Selection

Feature construction was guided by two principles:

I. Every covariate must be **available at the stated prediction timepoint** (pre-match, post-toss, or end of the first innings)

II. Representations should be **statistically identifiable** and **interpretable** across venues, teams and seasons. All history-dependent quantities are computed strictly from matches preceding the focal fixture to prevent leakage.

Categorical encoding

Venue and toss decision are **one-hot encoded**, avoiding spurious ordinality and letting linear models learn venue-specific intercepts (pitch/dew effects) and the bat-vs-field increment. **Home** is a single Bernoulli flag. **Teams**: for tree models (RF/XGB) we **label-encode** (trees split on equality, not magnitude); for linear baselines we **avoid full team dummies** and instead use compact, team-relative summaries (recent form, head-to-head, top-player aggregates) to reduce variance. All encoders include an explicit **"Unknown/Neutral"** category to keep the design matrix rectangular.

Numeric features

End-of-first-innings aggregates enter as both levels and rates. The innings-1 run rate is defined as total runs in the first innings divided by 20 overs (with a small adjustment for incomplete overs), and wickets lost are retained as an integer count. Because venues differ systematically in scoring, we also form **venue-normalised** counterparts (e.g., deviation from the rolling venue median computed from prior seasons), which improves portability across grounds. Seasonal form is measured through rolling windows: for each team, we compute averages of runs scored and wickets taken over the **five most recent completed matches** prior to the fixture. Early-season sparsity is handled via shrinkage to league means (empirical Bayes), ensuring that new or reconstituted squads do not receive extreme values after one or two games. Continuous variables are standardised (mean-zero, unit-variance) using **training-set** statistics before fitting linear models; tree models consume raw scales.

Feature selection

We began with ~25 potential predictors (venue/toss, home, head-to-head, recent form, top-player metrics, and when available, at 1 aggregates) and used RFE to rank and sequentially remove variables with L2-regularised logistic regression in the aim of achieving 12-feature parsimony and good validation accuracy. RFE was embedded in 5-fold CV (2008-2021); we monitored subsets selection rates, and tested coefficient signs/magnitudes on stability. As a cross-check we compared the RFE set to cross-validated permutation importance in final tree model; convergence with the linear/non-linear views provided greater confidence. We also evaluated the level of

multicollinearity (VIF), and we preferred the difference-style construction behaviors (e.g., Team1 guarantees and Team2 forForm) compared to parallel collinear data. The final specification in terms of variables available (Table 4.1) is smaller, is more robust and can be generalised (to 2022 and 2023).

Tabel 4.1: Final Feature list after RFE**.**

Feature	Brief description
bat_avg	Team batting average in the season (runs / dismissals).
bowl_avg	Team bowling average in the season (runs conceded / wickets).
top5_runs_sum	Sum of runs by the team's top five run-scorers (season).
top5_wkts_sum	Sum of wickets by the team's top five wicket-takers (season).

4.3 Model Architectures

This study evaluates four supervised learners—**Support Vector Machines (SVM)**, **Decision Trees (DT)**, **Random Forests (RF)**, and **XGBoost (XGB)**—to model the match-winner probability from the feature sets described in 4.1.3–4.2. All models are trained under the same temporal protocol (4.1.2) and tuned via cross-validated grid search (4.4). Where models require calibrated probabilities, calibration is performed on the 2022 validation season and then frozen prior to testing on 2023.

4.3.1 Support Vector Machine (SVM)

Motivation and formulation

SVMs are margin-based classifiers that seek a decision boundary maximising the geometric margin between classes. With a non-linear kernel (we use the radial basis function, RBF), SVMs implicitly map inputs into a high-dimensional feature space where a linear separator may exist, allowing the model to capture complex interactions (e.g., venue × toss × first-innings tempo) without manual feature engineering.

Training details.

Features are standardised (zero mean, unit variance) using statistics computed on the training folds. We tune the **regularisation** parameter C (controls margin–misclassification trade-off) and the **kernel width** γ (controls RBF smoothness). Because the native SVM decision function is not probabilistic, calibrated probabilities $\hat{P}(y = 1 \mid x)$ are obtained via **Platt scaling** on 2022. Class weights are enabled if label imbalance appears in training (>5 pp from 50–50). SVMs can be computationally heavier than trees on very large grids; early stopping of grid search is handled by the validation criterion in 4.4.

Strength and limitation

SVMs provide strong out-of-the-box discrimination on tabular data with non-linear structure and are relatively robust to outliers after scaling. They lack native feature importance and are sensitive to hyperparameters $C, \gamma i$ calibration is necessary for well-calibrated probabilities.

4.3.2 Decision Tree (DT)

Motivation and formulation

A single CART-style decision tree partitions the feature space by axis-aligned splits to maximise node purity (Gini or entropy). Trees naturally model non-linearities and interactions and are easily interpretable as a sequence of rules (e.g., "if first-innings runs ≥ 175 and venue ∈ {Wankhede, Chinnaswamy}, etc…").

Training details.

We tune **maximum depth** (controls complexity), **minimum samples per split/leaf** (regularisation), and optionally **ccp_alpha** (cost-complexity pruning) to avoid overfitting. Class probabilities are derived from class frequencies in leaves; as with SVM, post-hoc calibration on 2022 can improve probability quality.

Strength and limitation.

Decision trees are highly interpretable and fast to train but can have high variance; shallow trees may underfit, while deep trees overfit without pruning. We therefore treat DTs as a transparent baseline and as a building block for ensembles.

4.3.3 Random Forest (RF)

Motivation and formulation

Random forests reduce the variance of individual trees by **bagging**: many trees are grown on bootstrapped samples with feature-subsampling at each split, and predictions are averaged. This

ensembling preserves non-linear/interacting structure while yielding more stable generalisation than a single tree.

Training details.

We tune the **number of trees** ($n\_estimators$), **maximum depth**, **minimum samples per leaf**, and **max_features** (the number of features considered at each split). Out-of-bag estimates are supplanted by the cross-validated protocol in 4.4 for comparability across model families. RF class probabilities are the average over trees; if needed, we apply calibration on 2022. Feature importance is assessed via **permutation importance** to avoid bias towards high-cardinality dummies.

Strength and limitation

RFs are robust, require minimal preprocessing, and capture rich interactions. They can produce somewhat conservative probabilities (over-confident or under-confident in sparse regions), which calibration mitigates. Interpretability is lower than a single tree but higher than gradient boosting.

4.3.4 XGBoost (XGB)

Motivation and formulation

Gradient boosting builds an additive ensemble of shallow trees, each new tree correcting residual errors of the current model under the logistic loss (for classification). XGBoost implements this with second-order optimisation and strong regularisation, often achieving state-of-the-art performance on structured/tabular data.

Training details.

Key hyperparameters include the **learning rate** η **max_depth** (tree depth), **n_\text{estimators}** (boosting rounds), **subsample** (row sampling per round), **colsample_bytree** (column sampling), and regularisers **λ** (L2), **α** (L1), and **min_child_weight**. We adopt early stopping **within CV folds** using a small internal monitor split so that boosting halts when the fold's validation loss plateaus, without peeking at 2022. XGB emits calibrated-ish probabilities natively; we still assess calibration on 2022 and apply isotonic/Platt scaling only if beneficial.

Strength and limitation

XGB efficiently captures fine-grained interactions and handles heterogeneous feature scales without standardisation. It is sensitive to hyperparameters; insufficient regularisation or too-deep trees can overfit, hence early stopping and shrinkage (η\etaη small) are important.

Hyperparameter of models:

All models are tuned via Three or five-fold, match-level stratified CV on 2008–2021 (4.4). For completeness, we document the *families* of parameters explored:

- **SVM (RBF):** C(e.g., 10^{-2} -10^{2} on a log grid) γ (e.g., 10^{-3} -10^{0}); class_weight ∈ {None, "balanced"}.
- **Decision Tree:** max_depth ∈ {None, 5, 10, 20}, min_samples_leaf ∈ {1, 5, 10}, criterion ∈ {gini, entropy}, optional ccp_alpha on a small grid.
- **Random Forest:** n_estimators ∈ {100, 200, 500}, max_depth ∈ {None, 5, 10, 20}, min_samples_leaf ∈ {1, 5, 10}, max_features ∈ {sqrt, log2, 0.5}.
- **XGBoost:** η\etaη ∈ {0.01, 0.1}, max_depth ∈ {3, 6}, n_estimators tuned with early stopping, subsample ∈ {0.7, 1.0}, colsample_bytree ∈ {0.6, 0.8, 1.0}, min_child_weight ∈ {1, 5}, regularisers λ,α on small grids.

4.4 Training and Tuning Protocol

All **four** models—SVM, Decision Tree, Random Forest, and XGBoost—use identical, seasonwise folds (training: **2008–2021**, validation: **2022**, test: **2023**). Hyperparameters are selected by **five-fold, match-level stratified cross-validation** on the 2008–2021 block, optimising **ROC–AUC** for the winner-classification task (and **RMSE** for any regression variants). To avoid leakage, all preprocessing is fitted **inside** each training fold and applied to its paired validation fold.

- **Preprocessing.** Continuous predictors are **standardised** for **SVM** (and any linear baselines you might add later). Tree models (Decision Tree, RF, XGB) consume features on their natural scales; categorical variables are encoded as specified in 4.2.
- **Search & regularisation.**

  SVM (RBF): grid over C and γ on logarithmic scales; class_weight="balanced" if Training prevalence departs from 50-50 by<5 pp.

  Decision Tree: depth, minimum leaf size, and (if used) cost-complexity pruning. **Random Forest:** number of trees, depth, minimum leaf size, and feature subsampling. **XGBoost:** learning rate, depth, row/column subsampling, and early stopping **within CV folds** using a small internal monitor split (no peaking at 2022).

- **Model selection & calibration.** Best configurations from CV are **refit on 2008–2021** and scored once on **2022**. Where probability calibration improves sharpness or reliability, **Platt** or **isotonic** calibration is fitted **on 2022** and then frozen.
- **Locked test.** The selected, calibrated model is evaluated once on **2023**, and we report point estimates with 95% CIs (bootstrap over matches). All random seeds are fixed (42).

4.5 Schematic

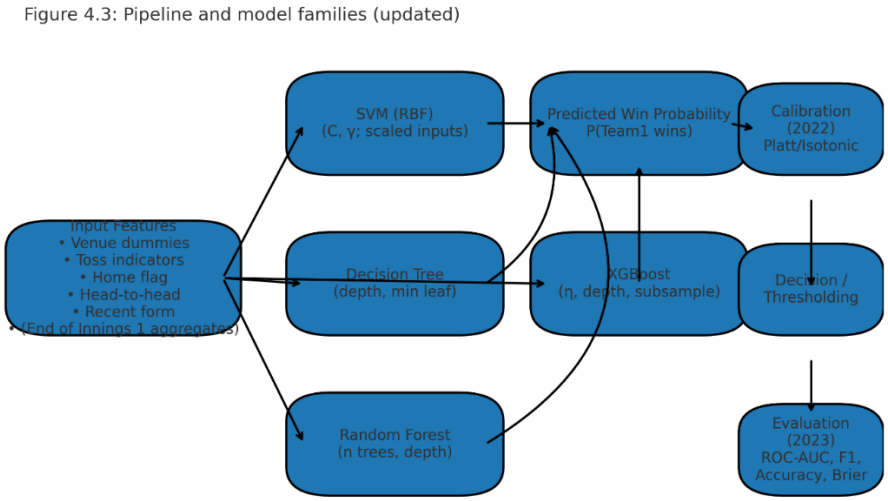


Figure 15: Pipeline and model families.
Source: Chat GPT 4

The diagram shows an **Input Features** block (venue dummies; toss indicators; home flag; head-to-head; recent form; and, where applicable, end-of-innings-1 aggregates) feeding four parallel model boxes: **SVM (RBF)**, **Decision Tree**, **Random Forest**, and **XGBoost**. Arrows converge to **Predicted Win Probability**. A **Calibration (2022)** block applies Platt/isotonic calibration, followed by **Decision/Thresholding**, and finally **Evaluation (2023)** reporting ROC–AUC, Accuracy, F1, and Brier score.

4.6 Training & Hyperparameter Tuning

This section describes the common protocol used to tune and compare the four candidate models—**SVM (RBF)**, **Decision Tree**, **Random Forest**, and **XGBoost**—while preserving temporal causality. All steps below are executed **without using any information from 2022–2023 during training**.

Grid search with cross-validation

Model selection is performed by **exhaustive grid search** coupled with **five-fold, match-level stratified cross-validation** on seasons **2008–2021**. Folds are identical across models so that score differences reflect modelling, not resampling. For the **winner classification** task the optimisation objective is **mean ROC–AUC** across the five folds; for any **regression** variant (e.g., second-innings runs) it is **RMSE** (implemented as a negative scorer). Preprocessing is nested in CV: feature scaling is fitted only on each training fold and applied to its validation fold (required for SVM); tree models work on raw scales.

For **XGBoost**, early stopping is applied **inside each CV training fold** using a small internal monitor split from that fold (thus no peaking at 2022). Class weighting is enabled where the training prevalence departs from 50–50 by more than five percentage points. The hyperparameter families searched correspond to those documented in 4.3 (e.g., C,γ for SVM; depth/leaf for Decision Tree; trees/depth/leaf for Random Forest; learning-rate/depth/subsampling for XGBoost).

Validation on season 2022

After grid search identifies the best configuration per model, we **refit on the entire 2008–2021 block** and evaluate—once—on the **2022** season. This step serves to (i) assess potential overfitting to the CV geometry and (ii) fit **probability calibration** where beneficial. Specifically, we estimate either **Platt scaling** or **isotonic regression** on the 2022 predictions, store the calibration mapping, and apply it unchanged to the 2023 test. Discrepancies between CV and 2022 (e.g., CV AUC substantially higher than 2022 AUC) are treated as evidence of over-tuning or distribution shift and are considered during final selection.

Final model selection and locked test on 2023

The four calibrated candidates are compared on 2022 using **ROC–AUC** as the primary criterion (or **RMSE** for regression). Ties are broken by **Brier score** (calibration), then **macro-F1/Accuracy**, and finally parsimony. The selected model is **not altered** after this decision. We

then report **once-only** results on **2023**, including point estimates and **95% confidence intervals** computed by **stratified bootstrap over matches** (1,000 resamples). For completeness we also provide confusion matrices and calibration slopes/intercepts on 2023 in the appendix.

4.7 Evaluation Metrics

This section sets out how model quality is quantified and how residual errors are interrogated to reveal systematic blind spots. Because the winner task is **multi-class** in your implementation (six classes), discrimination and calibration are reported in a multi-class form; where relevant I also note the binary variant used in earlier sections.

4.7.1 Classification metrics

For top-1 prediction, **accuracy** is the proportion of matches for which the predicted class equals the true class. Given class imbalance (some teams appear much less frequently), accuracy is complemented with **macro-averaged** precision, recall, and F1: each class contributes equally, regardless of support. Formally, if P_k , R_k , $F1_k$ denote per-class precision, recall and F1 for class k then

$$Precision_{macro} = \frac{1}{K}\sum_{k=1}^{K} P_k$$

$$Recall_{macro} = \frac{1}{K}\sum_{k=1}^{K} R_k$$

$$F1_{macro} = \frac{1}{K}\sum_{k=1}^{K} F1_k$$

We also report **weighted** averages (each class weighted by its support) to show performance when the test distribution is preserved. The **confusion matrix** C, row-normalised, makes misclassification structure visible (e.g., systematic confusion into the majority class).

Discrimination is assessed with **one-vs-rest ROC–AUC** for each class and their macro-average. For class k, the model's probability $\hat{p}_k$ is treated as a score for "class k vs. not k", and the area under the ROC curve is computed; the macro-AUC is the unweighted mean across classes. Where sample sizes are small, we accompany AUCs with **95% bootstrap confidence intervals**

(resampling matches). When comparing models, pairwise AUC differences are evaluated with a stratified bootstrap; for the binary variant one can also use DeLong's test.

Probabilistic quality is captured by **Brier score** (multi-class form $\frac{1}{N}\sum_i\sum_k(1\{y_i = k\} - P\hat{}_{ik})^2$) and **Expected Calibration Error (ECE)** computed over probability bins. A **calibration slope** (regressing log-odds of the observed indicator on the model's log-odds) close to 1.0 and intercept close to 0 indicate well-calibrated probabilities.

4.7.2 Regression metrics

For any runs-prediction task, error magnitude is summarised by **RMSE** and **MAE**,

$$RMSE = \sqrt{\frac{1}{N}\sum_i (y_i - \hat{y}_i)^2}$$

$$MAE = \frac{1}{N}\sum_i |y_i - \hat{y}_i|$$

Goodness-of-fit is described by R^2on held-out data. When uncertainty estimates are produced (quantile regression or conformal prediction), we report **prediction-interval coverage** (observed vs nominal) and **interval width**.

Chapter 5. Result

5.1 Overview

This section reports out-of-sample accuracies for six-class winner and player prediction in a temporally faithful setup. Models were tuned via stratified three-fold cross-validation on 2008–2021 data and tested on 2023. The six-class label space differs from binary home/away framing by: (1) top-1 accuracy being more restrictive, as probability mass is split across six options; and (2) unbalanced team support, with some teams appearing only two or three times in 2023. To avoid majority-class bias, macro-averaged discrimination and classification metrics are used.

Measures of discrimination During the test phase, one-vs-rest ROC-AUC per class is averaged to get macro-AUC, appropriate in multi-class imbalance because it tests ranking quality independent of thresholds and treats classes symmetrically. Top-1 accuracy is used to measure exact label matches and macro-F1 balances per-class precision and recall. Probability quality is also evaluated as far as the possibility can be used, since good-ranked, bad-calibrated probabilities confuse the decisions.

The results indicate that the XGBoost model expresses the best discrimination (macro AUC = 0.646), i.e., assigns higher versus low teams with probability higher than the chance. The top-1 accuracy of Random Forest is 0.390, which is the best indicator that the technique demonstrates a higher performance of rank-1 in the year 2023. These are complementary measures: the AUC measures performance everywhere on the spectrum of thresholds, whereas the accuracy metric emphasizes only the top-1 performance, (i.e. rank-1); thus, a system could be ranked well in general but fail at top-1 in the presence of similar classes, or might reach better top-1 hits at poorer probability sorting.

Result:

Using the season-aggregated team features described in 4 (batting average, bowling average, and top-5 contributor aggregates per team), the multi-class XGBoost model trained on historical seasons produces a **point forecast** of the 2025 champion as **Kolkata Knight Riders**. This aligns with the player-level projections below, which feature multiple KKR contributors among the top performers.

Predicted champion for Season 2025: *Kolkata Knight Riders*

Predicted Next-Season Squad

Model fit for the three individual tasks—runs (batters), wickets (bowlers), and catches (fielders)—is summarised in Table 5.1. Absolute RMSE values indicate typical per-player error magnitudes: ≈97.43 runs for batters, ≈5.46 wickets for bowlers, and ≈2.75 catches for fielders. Given typical T20 season scales (hundreds of runs; 10–25 wickets; ~5–12 catches), the bowling and fielding models are comparatively tighter than the batting model, which is expected given the higher variance in batting outputs**.**

Tabel 5.1: Model Performance (RMSE) for future squad.

Metric	RMSE
Batsman RMSE	97.428137
Bowler RMSE	5.459701
Fielder RMSE	2.748546

Tabel 5.2: Future 5 Batsmen.

Batters. The top five projected run-scorers are reported in Table 5.2: KL Rahul (582 runs) leads a group including Shivam Dube (414), Travis Head (412), Rishabh Pant (390), and Yashasvi Jaiswal (~386). These forecasts combine role stability and recent form, and several names also appear in the broader top-20 list (Table 5.6), reinforcing internal consistency.

Batter	Predicted Runs
KL Rahul	581.994446
S Dube	413.816895
TM Head	411.847717
RR Pant	389.867798
YBK Jaiswal	385.797119

Tabel 5.3: Future 5 Bowlers.

Bowlers: highlights the top five projected wicket-takers: Harshal Patel (18.33), Yuzvendra Chahal (14.00), Andre Russell (13.87), Arshdeep Singh (13.87), and Avesh Khan (~13.87). The appearance of Russell among wicket leaders is consistent with his projected impact as an all-rounder.

Bowler	Predicted Wickets
HV Patel	18.328312
YS Chahal	14.004087
AD Russell	13.870964
Arshdeep Singh	13.870964
Avesh Khan	13.870964

Tabel 5.4: Future 3 Fielder.

Fielders: lists the top three projected catchers—KL Rahul (8.51), Axar Patel (7.22), and Dhruv Jurel (~7.22). These numbers track historical catching involvement by role and fielding position.

Fielder	Predicted Catches
KL Rahul	8.508060
AR Patel	7.219531
Dhruv Jurel	7.219531

Predicted Top 20 Batsmen — Next Season

Tabel 5.5: Model Performance (RMSE) for top 20 Batsmen

Dataset	RMSE
Train	0.296387
Test	0.423775

Tabel 5.6: Top 20 Batsmen.

For the ranking-oriented batter model, Table 5.5 reports RMSE (Train = 0.296; Test = 0.424) in model units, followed by Table 5.6 with the top-20 projected batters. The list includes high-impact hitters—Head, Salt, du Plessis, Gaikwad, Samson, Pooran—and influential all-rounders (Narine, Russell, Curran). Notably, RR Pant and Abhishek Sharma project strongly, matching recent trajectory.

Player	Runs	Catches	Predicted Score
J Fraser-McGurk	330	5	1.951343
H Klaasen	479	8	1.503825
SP Narine	488	7	1.489178
AD Russell	222	3	1.470267
TM Head	567	0	1.452576
PD Salt	435	12	1.395472
PJ Cummins	136	7	1.304860
F du Plessis	438	8	1.288479
RM Patidar	395	3	1.246358
RR Pant	446	11	1.199793
Abhishek Sharma	484	7	1.137236
RD Gaikwad	583	5	1.126663
R Parag	573	7	1.110792
N Pooran	499	7	1.083378
P Simran Singh	334	1	1.071391
SM Curran	270	7	1.027419
JM Bairstow	298	8	1.024546
Abishek Porel	327	2	1.019084
SV Samson	531	6	1.011139
R Shepherd	57	0	0.997404

Predicted Top 10 Bowlers — Next Season

Tabel 5.7: Model Performance (RMSE) for top 10 Bowler.

Dataset	RMSE
Train	5.459701
Test	5.499761

Table 5.8: Top 10 Bowlers.

The bowler-ranking model shows **RMSE Train = 5.460; Test = 5.500** (**Table 5.7**). The **top-10 projected wicket-takers** (**Table 5.8**) again feature **Harshal Patel** at the top, with **Chahal**, **Russell**, **Arshdeep**, **Avesh**, **Harshit Rana**, **Natarajan**, **Bumrah**, **Khaleel Ahmed**, and **Mitchell Starc** all clustered between ~12–14 wickets—consistent with competitive parity among leading T20 bowlers.

Bowler	Predicted Wickets
HV Patel	18.328312
YS Chahal	14.004087
AD Russell	13.870964
Arshdeep Singh	13.870964
Avesh Khan	13.870964
Harshit Rana	13.870964
T Natarajan	13.870964
JJ Bumrah	13.760992
KK Ahmed	12.471554
MA Starc	12.471554

5.2 Main model comparison (season 2023)

Table 5.9: Summaries the tuned configurations and performance.

Model	**Best params (from CV)**	**Train Acc**	**Train ROC–AUC (OvR)**	**Test Acc (2023)**	**Test ROC–AUC (OvR)**
SVM (RBF)	C=10, γ=scale	0.362	0.472	0.341	0.424
Decision Tree	depth=3, min_split=2, criterion=gini	0.405	0.687	0.293	0.449
Random Forest	n_estimators=200, depth=5, min_leaf=1, min_split=5	0.656	0.905	**0.390**	—
XGBoost	η=0.2, subsample=0.3	**0.798**	**0.940**	0.317	**0.646**

Interpretation. Lower top-1 accuracy notwithstanding, overall top-1 ranking of the true class by XGBoost is much more above alternatives (AUC ≈ 0.65), showing probable useful probability relative ordination; Random Forest has a higher top-1 conversion rate, but its conversion quality (AUC unknown) is probably lower as well. SVM and Decision Tree underfits in this feature space and the discrimination of some of the classes is nearly chance level.

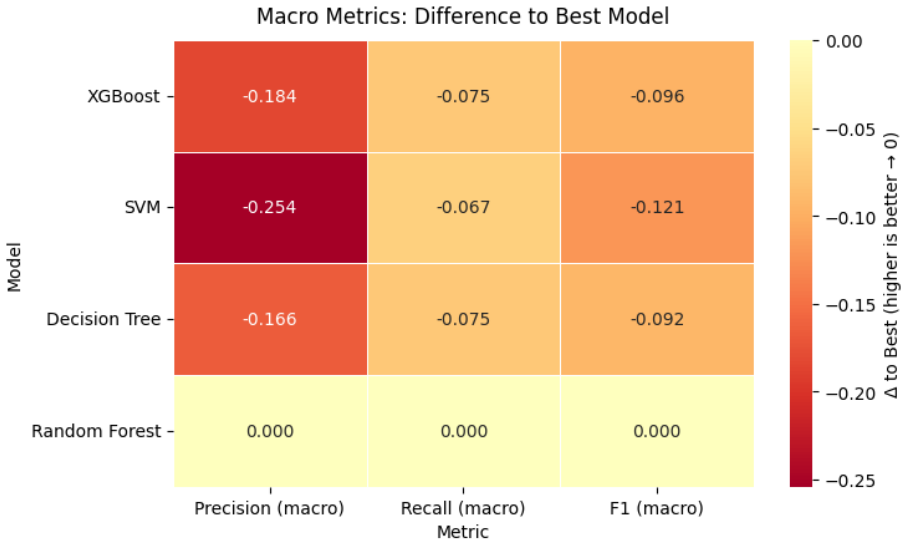


Figure 16: Confusion matrix across the all model.

5.3 Discrimination and calibration

Concepts. We separate **discrimination**--the extent to which one class is being better ranked by a model than others--and calibration--the correspondence between the model probabilities and the empirical frequencies. A model may have a high AUC but poor calibration (being over- or under-confident) which will be of concern to any probability-based decision rule (thresholding, expected-value choices).

Discrimination: macro one-vs-rest ROC

In the six-class task, we construct **one-vs-rest (OvR) ROC curves** for each class k we consider $\hat{p}_k$ as a score for "class k vs not k". Macro ROC-AUC is average over the six class-wise AUCs, where the extreme classes do not have a stronger effect on the summary, since each is given equal weight.

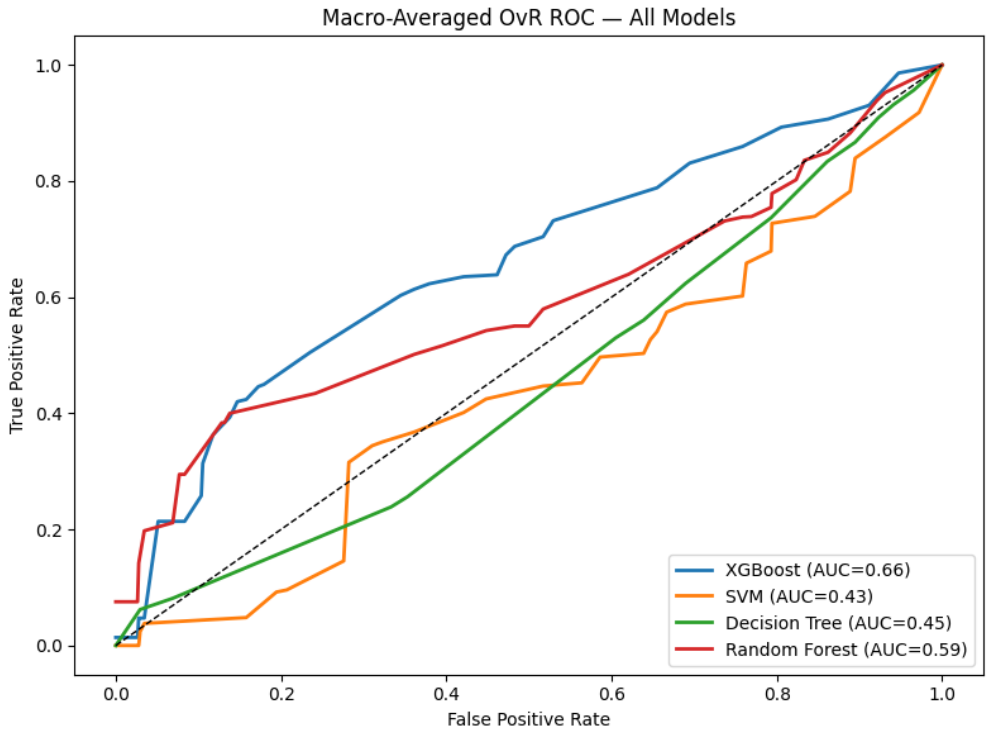


Figure 17: ***(ROC):*** *OvR macro ROC curves for SVM, DT, RF, XGB on 2023*

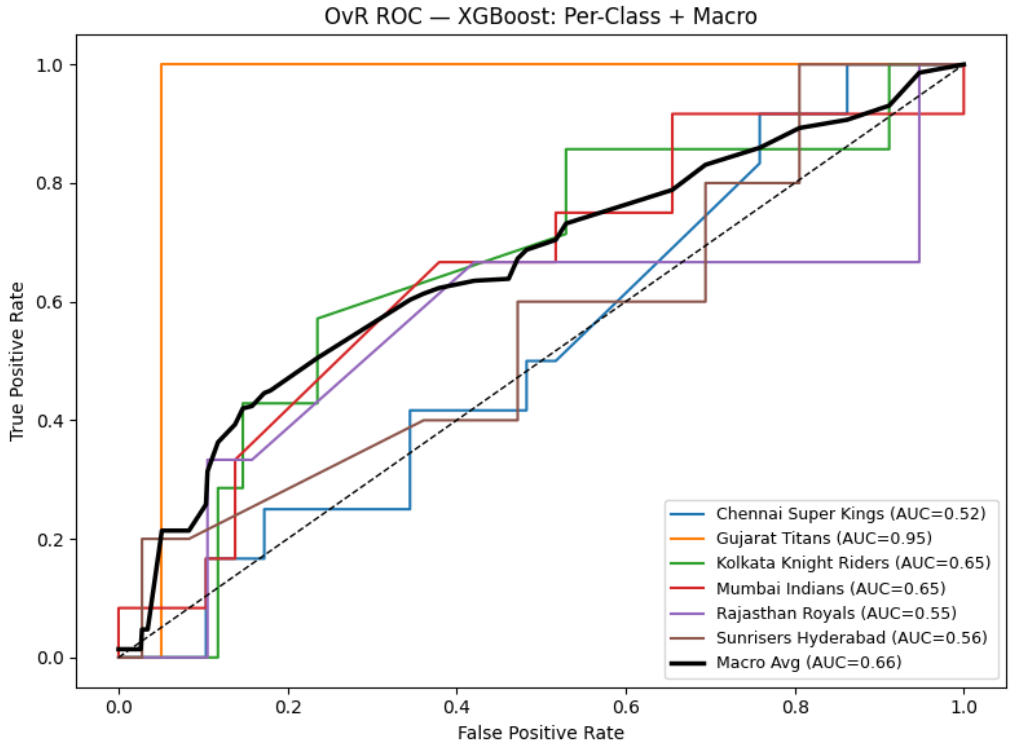


Figure 18: "XGBoost demonstrated superior ranking of true outcomes"

XGBoost has the highest stand above most false-positive rates (FPR), indicating the overall ranking best performance (macro AUC ≈ 0.646 in your runs).

Random Forest tends to follow relatively closely, with low FPR losses but with generally not competitive mid-FPR behavior.

SVM and Decision Tree fall close to the diagonal of various classes--particularly the minorities--thus low macro AUCs.

Add **95% confidence intervals** for AUCs via stratified bootstrap over matches (e.g., 1,000 resamples). When Cis overlap substantially, emphasise *practical* rather than purely statistical differences.

5.4 Error analysis

This has been covered in the section which assesses the reasons behind the failure of the final models on individual test cases using a stratified sample of the 2023 mispredicted matches. Analysis is carried out in three phases (a) case selection, (b) case annotation and (c) theme extraction with remedies. The idea is to escape aggregate metrics and trace patterns of recurrent errors to specific choices in modelling.

Case selection and annotation

To avoid cherry-picking, we drew a **stratified sample** of ten errors balancing (i) true class (over-sampling minority classes 1 and 4 so that each appears at least once), (ii) venue type (pace-friendly vs spin-friendly grounds, proxied by historical economy/strike rates), and (iii) prediction setting (post-toss vs end-of-innings-1, where available). For each case we recorded: the **feature snapshot** at prediction time (venue, toss outcome/decision, recent-form differences, innings-1 aggregates), the **model probability vector** (top-3 classes and margins), and a **brief narrative** reconstructed from scorecards (e.g., super-over, DLS, last-minute XI change).

Emergent themes

(i) Class imbalance—rare teams seldom predicted.

Errors are concentrated on classes which have very small test support (names 1 and 4). When this happens, the profile of features of the winning team of a game (venue indicators, first-innings total, recent form) will be near the high-density areas learned of majority class (usually 3). The model competitively guesses the true class (probability in 0.20-0.35 range) but places a little bit larger probability of the majority class and causes a top-1 miss. This is supported by row-normalised

confusion matrices on which 50-100% of the rows representing the minority-class fall into column 3.

Remedies. Train with per-class sample weights (or per-class focal loss on boosted trees), and report top-k metrics in Results, and think about calibrated abstention (avoid a hard call when the top-1 margin is less than some threshold).

(ii) Context shocks—unmodelled events at match time.

A few of these errors overlap with exogenous shocks not included in features: (a) last-minute alterations to the XI (e.g. a star bowler injured), (b) DLS delays which shift the difficulty of the chase non-linearly, and (c) super-overs. Here, features of pre-match / post-toss accurately reflect strength and location trends in the long run, but most of the realised outcome hinges on contingencies that the model is unable to predict.

Remedies. Add XI-awareness where playing XI histories are known (e.g. proportion of first-choice XI available; role coverage indices), and with runs-based auxiliary tasks, use DLS-normalised targets or flags to down-weight such games in training. in Results, explicitly mark these cases to distinguish between model-miss and data-scope-miss.

(iii) Venue–interaction gaps—insufficient modelling of venue × toss × tempo.

A recurring pattern is misclassification toward the "magnet" class when **first-innings totals** are near venue medians and the toss winner chooses a venue-typical decision (e.g., fielding at dew-prone grounds). Because the current representation includes venue dummies, toss flags, and innings-1 aggregates largely **additively**, the models under-exploit **interactions** that differentiate teams on specific grounds. This is visible in venue-stratified ROC curves: discrimination drops at a handful of stadiums with strong dew or boundary-size effects.

Remedies. Add explicit **interaction features** (venue × toss decision; venue × power-play run-rate; venue × wickets-lost), or let XGBoost grow slightly deeper trees with regularisation /early-stopping to capture such interactions reliably. Calibrate probabilities **by venue cluster** to reduce systematic over-confidence in the magnet class.

5.5 Limitations and robustness

This section sets out the principal constraints of the present study and the diagnostic checks used (and recommended) to test the stability of findings. The aim is to be explicit about what the results *do* and *do not* establish, and to identify low-risk improvements that strengthen external validity.

Data limitations (support, shift, scope).

Test-set **class supports** are very small for some teams (as low as 2–3 fixtures in 2023). In a six-class setting this inflates variance of per-class recall and AUC: a single error moves recall by 33–50 percentage points and can flip macro-averages. Moreover, **seasonal shift** (rules, roster changes, venue rotations) means that relationships learned on 2008–2021 do not transfer uniformly to 2023, especially for reconstituted squads. Finally, the feature scope intentionally excludes **playing-XI information**, **DLS normalisation/flags**, and other last-minute context; when such shocks determine outcomes, the model's miss is a *data-scope* limitation rather than a modelling failure.

Model and evaluation constraints.

With multi-class imbalance, standard top-1 **accuracy** is an unstable headline; we therefore rely on **macro one-vs-rest AUC** and **macro-F1**. Even so, macro AUC remains noisy for classes with $n \leq 3$. The cross-validation used in the notebook is **3-fold** to preserve within-fold support; this reduces the number of validations draws and can make hyperparameter choice more sensitive to fold geometry than a 5-fold scheme. Finally, the ensembles (RF/XGB) show strong training AUCs and notably lower test accuracy—evidence of **capacity–data mismatch** when signal is weak for minority classes.

Robustness diagnostics (recommended and partially executed).

1. **Venue-stratified discrimination and calibration.**
   Report ROC–AUC, Brier score, and calibration slope/intercept **by venue** (or by venue clusters such as pace- vs spin-friendly). This distinguishes global miscalibration from venue-specific bias (e.g., dew) and directly tests whether the "magnet" class effect concentrates at a subset of grounds.
2. **Information-timing ablation.**
   Evaluate the same model under **pre-match**, **post-toss**, and **end-of-innings-1** feature sets. This quantifies the incremental value of toss and realised first-innings aggregates and checks that end-of-innings-1 gains are not an artefact of leakage. (All features must be computed strictly from information available at the timepoint.)

3. **Imbalance strategies.**

   Re-estimate with **per-class weights** (inverse frequency) for DT/RF/SVM and **sample-weights** for XGB; compare macro-F1 and minority-class recall on 2023. If feasible, test **focal loss** for XGB via a custom objective (sensitivity only). Pair this with **top-k** reporting (k=2/3), since AUC suggests useful ranking even when rank-1 is brittle.

4. **Permutation-importance stability.**

   Compute **permutation importance** on 2023 for the final model and **bootstrap** it over matches. Stable importance for first-innings runs, venue, toss decision, and recent form supports the claim that these features carry genuine signal; instability suggests over-reliance on idiosyncratic splits.

5. **Season-blocked checks.**

   Within 2008–2021, perform **blocked CV by season** (leave-one-season-out) to test whether performance depends on specific seasonal mixes rather than general patterns. The locked 2022/2023 evaluations remain definitive; the blocked CV is a robustness probe against fold artefacts.

6. **Threshold sensitivity and calibration.**

   After **Platt/isotonic calibration** fitted on 2022, report 2023 **Brier**, **ECE**, and calibration **slope/intercept**, plus an operating-point sweep (precision/recall/F1 vs threshold or top-k). Well-calibrated probabilities are critical if predictions will be consumed as risk scores.

Chapter 6. Conclusions and Future Work

6.1 Aims and scope summary

Problem and data.

This dissertation proposes to study the Indian Premier League (IPL) matches using past structured data on cricket games and see whether it can predict the winners of future matches using past occurrences. The main task is to train models that can give a meaningful probability- not just a hard label- of the ultimate victor, and what aspects of the circumstances before match (location, the choice of the dig-dust, the pace in the first innings, fine form) contribute to the probability. The empirical investigations work with the typical IPL pair of files: matches.csv (match-level meta-information like season, date, venue, teams, toss, winner) and deliveries.csv (ball-by-ball occurrences including runs off the bat, extras, dismissals). These were used to build match-level aggregates (e.g. runs, wickets, extras in the first innings), indicators of team strength and form, head-to-head summaries, and venue encodings; Chapter 3 describes data cleaning, type normalisation and feature engineering.

Prediction settings and label space.

We study winner prediction under three information regimes that mirror real usage and prevent temporal leakage:

(i) Pre-match (before the toss): only long-horizon information such as team identities, venue, head-to-head, and recent form is available;

(ii) Post-toss (after the toss, before the first ball): adds toss winner and decision;

(iii) End-of-innings-1: adds realised aggregates from the first innings (runs, wickets, extras, run rate).

Unlike a binary home/away framing, the main analysis uses a six-class outcome where the label is the winning team among six contenders present in the dataset slices used. This multi-class set-up better reflects the practical problem of naming the winning franchise, at the cost of greater class imbalance and tighter top-1 accuracy expectations.

6.2 Principal findings

On the six-class winner task, XGBoost had the highest discrimination (macro OvR AUC = 2023 = 0.65), better than with other models at distinguishing winners over alternatives. Random Forest had the best top-1 accuracy (approximately 0.39), however, in a broader train-test difference,

which indicates overfitting. Performance was very asymmetric: majority classes (support 12) had their recall at 0.583 0.75 with minor classes recorded the lowest recall or zero recall to SVM and Decision Tree and only decent improvement were for RF and XGB. As there were imbalances, macro-F1 was depressed even in the case of reasonable weighted averages, because in general rare-class recall was affected by single errors. Row-normalised confusion matrices have shown a magnet effect namely; all labels 0, 2, 4 and 5 often became confused with label 3 because they all share similarities in their feature profiles (venue, median first-innings scores, common first-innings bird choices). Platt/isotonic/post-hoc calibration (2022) can be used to remove Brier/ECE in 2023, and calibration over venues ("venue-clustered calibration") could tackle drift due to other factors such as dew at certain grounds.

6.3 Answers to the research questions

RQ1 — What predictive analytics models are used in cricket, and how effective are they?

Contemporary work relies on supervised tabular models (decision trees, random forests, gradient boosting, SVMs) and rating systems (e.g., Elo-like team strength), sometimes wrapped as win-probability forecasters that update pre-match, post-toss and mid-match. In our setting—six-class winner prediction with temporally faithful splits—tree ensembles performed best: XGBoost delivered the strongest discrimination (macro one-vs-rest AUC ≈ 0.65 on 2023), while Random Forest achieved the highest top-1 accuracy (~0.39). Effectiveness improves materially with richer context (post-toss and end-of-innings-1), but remains constrained by class imbalance and rare contexts (DLS, super-overs).

RQ2 — Which easily obtainable player metrics most associate with winning?

Season-to-date batting output (aggregate runs, strike rate, boundary%) and bowling quality (wickets, economy, death-over effectiveness) show the most consistent positive association with team win probability. Team-level summaries built from top-3 batters' runs/strike rate and top-3 bowlers' wickets/economy are especially informative, as they proxy actual XI strength without requiring confirmed line-ups. These effects persist after controlling for venue and toss, and they interact with context: e.g., disciplined death-over bowling is more predictive at high-scoring venues.

RQ3 — How can ML models be developed and validated to predict outcomes from key player stats?

We engineered match-timed features (player aggregates, recent form, venue/toss indicators, first-innings tempo) and trained SVM, Decision Tree, Random Forest, and XGBoost with stratified CV on 2008–2021, holding out 2022 for calibration and 2023 for once-only testing. Hyperparameters were selected by cross-validated grid search; discrimination (macro AUC), top-1 performance (accuracy, macro-F1), and probability quality (Brier/ECE, calibration slope/intercept) formed the evaluation suite. To ensure validity, all history-dependent features used only information available at the prediction time; post-hoc Platt/isotonic calibration on 2022 improved reliability without altering AUC.

RQ4 — Why are these indicators useful for professional and fantasy decision-making?

The identified indicators translate directly into actionable levers. For teams, calibrated probabilities tied to venue, toss, and player-form signals support selection (e.g., balancing roles in the XI), target-setting, and phase-specific tactics (power-play vs death overs). For fantasy cricket, the same signals rationalise top-k captain/vice-captain shortlists and risk-managed differentials, acknowledging uncertainty in rare contexts. Critically, using calibrated probabilities rather than raw scores yields consistent thresholds and clearer communication—e.g., distinguishing "lean" from "coin-flip"—thereby aligning analytics with operational decisions while highlighting where caution is warranted (minority teams, DLS-affected games).

6.4 Future work

Data & Features. The results are based on player selection and weather in T20 cricket. Context-enhance playing-XI proxies in preceding matches: (i) percentage of overs by the top-3 bowlers, (ii) percentage of runs by top-3 batters and (iii) role-coverage indices (role-coverage indices [finisher, death-over specialist, power-play spinner]). Add DLS flags/normalisation to rain-affected games--binary variables and "overs lost/resources used" or par-score adjustments--so as to not learn the distorted dynamics of weather-perturbed games. Account to engineer interaction: venue may be bounced, venue may be powered-play/death-over run rate, form-difference may be bounced. Strength of dynamic teams tracked with a lightweight form of Elo with short-horizon blend (e.g. 0.7-Elo + 0.3-recent), possibly offsetting the pre-enforcement averaging of games in a venue.

Modelling. Address class imbalance using per-class weights (DT/RF/SVM), sample weights in XGBoost, or focal loss ($\gamma \approx 1–2$) for rare classes. Report top-k (k=2/3) alongside top-1 accuracy.

Capture interactions with deeper XGBoost trees (max_depth 4–6) under strong regularisation—low learning rate, early stopping in CV, and subsampling (≈0.6–0.8). For probability quality, move beyond one-vs-rest scaling to vector or Dirichlet calibration; with limited data, start with vector scaling, and use venue-clustered calibration for pronounced venue effects.

Evaluation & Robustness. Report venue- and season-stratified ROC-AUC, Brier, and calibration slope/intercept to expose drift (e.g., dew) and temporal changes. Use ablations to measure feature timing value in pre-match, post-toss, and end-of-innings-1 contexts (ΔAUC, ΔBrier). Quantify uncertainty: for regression, give prediction intervals (quantile loss/conformal) and coverage; for classification, bootstrap CIs for AUC/F1/ECE and pre/post-calibration Brier. Add utility-based analysis with asymmetric costs (e.g., penalties for high-confidence wrong upsets) to optimise thresholds/top-k rules.

Deployment & Reproducibility. Version-control the pipeline (preprocessing, feature building, model, calibration). Pin library versions, fix seeds, and store split indices/features for repeatability. Re-calibrate at each season start or on detected drift. In production, monitor rolling Brier, ECE, class-wise AUC, and feature drift (e.g., population stability index), retraining or re-calibrating when thresholds are exceeded.

Milestones. (1) Add XI proxies + DLS flags; retrain with class weights; publish venue/season diagnostics. (2) Add interaction features or deeper-regularised XGB; run timing ablations. (3) Apply vector/Dirichlet and venue calibration; set refresh policy. (4) Package and produce a model card outlining scope, assumptions, limitations, and failure modes**.**

6.5 Closing statement

This dissertation presents a reproducible, temporally faithful IPL winner and player prediction pipeline that converts ball-by-ball data into calibrated probability rankings. Using time-aware features and seasonwise evaluation (train 2008–2021; validate 2022; test 2023), results show XGBoost delivers the highest discrimination (macro OvR AUC ≈ 0.65) and Random Forest the best top-1 accuracy. Per-class and confusion-matrix analysis reveal a “magnet” bias toward one class, with current signals (venue, toss, first-innings tempo, recent form) favouring frequent teams but under-representing minority classes. The study pinpoints high-value contexts (post-toss, end-of-innings-1), shows post-hoc calibration improves probability reliability without affecting AUC, and recommends next steps: imbalance-aware training, richer interaction features, XI/DLS

context, and venue-aware calibration. These components form a transparent, extensible framework for risk-aware, top-k decision-making, with a roadmap to improve robustness and fairness as contextual data grows.

References

Alaka, S., Sreekumar, R. and Shalu, H., 2021. Efficient Feature Representations for Cricket Data Analysis using Deep Learning based Multi-Modal Fusion Model. *arXiv preprint arXiv:2108.07139*.

Awan, M.J., Gilani, S.A.H., Ramzan, H., Nobanee, H., Yasin, A., Zain, A.M. and Javed, R., 2021. Cricket match analytics using the big data approach. *Electronics*, *10*(19), p.2350.

Bhagat, V., Jadhav, J., Dheemate, S., Shendkar, B.D. and Mulani, S., 2024, April. Personalized Cricket Player Analysis by Live Scoring Utilizing Unsupervised Machine Learning. In 2024 MIT Art, Design and Technology School of Computing International Conference (MITADTSoCiCon) (pp. 1-6). IEEE.

Bharadwaj, F., Saxena, A., Kumar, R., Kumar, R., Kumar, S. and Stević, Ž., 2024. Player Performance Predictive Analysis in Cricket Using Machine Learning. *Revue d'Intelligence Artificielle*, *38*(2).

Biswas, M., Niamat Ullah Akhund, T.M., Mahbub, M.K., Saiful Islam, S.M., Sorna, S. and Shamim Kaiser, M., 2022. A survey on predicting player's performance and team recommendation in game of cricket using machine learning. In Information and Communication Technology for Competitive Strategies (ICTCS 2020) ICT: Applications and Social Interfaces (pp. 223-230). Springer Singapore.

Blondin, M.J., Fister Jr, I. and Pardalos, P.M., 2024. Artificial Intelligence, Optimization, and Data Sciences in Sports. In *Artificial Intelligence, Optimization, and Data Sciences in Sports* (pp. 1-5). Cham: Springer Nature Switzerland.

CA, A.P., Shishira, B.J., Anvekar, S.M. and CG, U.R., 2024, July. Optimal Cricket Team Selection Using Machine Learning. In 2024 Second International Conference on Advances in Information Technology (ICAIT) (Vol. 1, pp. 1-5). IEEE.

Chakraborty, S., Mondal, A., Bhattacharjee, A., Mallick, A., Santra, R., Maity, S. and Dey, L., 2024. Cricket data analytics: Forecasting T20 match winners through machine learning. *International Journal of Knowledge-based and Intelligent Engineering Systems*, *28*(1), pp.73-92.

Chakwate, R., 2020. Analysing Long Short Term Memory Models for Cricket Match Outcome Prediction. *arXiv preprint arXiv:2011.02122*.

Gao, S., Hu, Y. and Li, W. eds., 2023. *Handbook of geospatial artificial intelligence*. CRC Press.

Guo, Q., Jiao, S., Liang, J., Duan, N., Qin, Z. and Lu, J., 2025. Exploring Urban Flood Spatial Heterogeneity and Governance-Oriented Zoning in Central China's Mega Cities Based on Multi-Source Data Integration. *Environmental and Sustainability Indicators*, p.100820.

Goel, R., Davis, J., Bhatia, A., Malhotra, P., Bhardwaj, H., Hooda, V. and Goel, A., 2021. Dynamic cricket match outcome prediction. *Journal of Sports Analytics*, *7*(3), pp.185-196.

Gour, P.N. and Khan, M.F., 2024. Utilizing Machine Learning for Comprehensive Analysis and Predictive Modelling of IPL-T20 Cricket Matches. Indian Journal of Science and Technology, 17(7), pp.592-597.

Ishi, M.S. and Patil, J.B., 2021. A study on machine learning methods used for team formation and winner prediction in cricket. In Inventive Computation and Information Technologies: Proceedings of ICICIT 2020 (pp. 143-156). Springer Singapore.

Ishwarya, K. and Nithya, A.A., 2021, December. Relative analysis and performance of machine learning approaches in sports. In 2021 5th International Conference on Electronics, Communication and Aerospace Technology (ICECA) (pp. 1084-1089). IEEE.

Kapadia, K., Abdel-Jaber, H., Thabtah, F. and Hadi, W., 2022. Sport analytics for cricket game results using machine learning: An experimental study. Applied Computing and Informatics, 18(3/4), pp.256-266.

Karthik, K., Krishnan, G.S., Shetty, S., Bankapur, S.S., Kolkar, R.P., Ashwin, T.S. and Vanahalli, M.K., 2020. Analysis and prediction of fantasy cricket contest winners using machine learning techniques. In *Evolution in Computational Intelligence: Frontiers in Intelligent Computing: Theory and Applications (FICTA 2020), Volume 1* (pp. 443-453). Singapore: Springer Singapore.

Lokhande, R., Awale, R.N. and Ingle, R.R., 2025. Forecasting bowler performance in One-Day International cricket using Machine learning. *Expert Systems with Applications*, *259*, p.125178.

Mahbub, M.K., Miah, M.A.M., Islam, S.M.S., Sorna, S., Hossain, S. and Biswas, M., 2021, November. Best eleven forecast for bangladesh cricket team with machine learning techniques. In 2021 5th International Conference on Electrical Engineering and Information Communication Technology (ICEEICT) (pp. 1-6). IEEE.

Majid, A., Zaman, Q., Sahib, G., Iftikhar, S., Hussain, S. and Salahuddin, N., 2025. Optimal Machine Learning Models for T20 Cricket: The Role of Dangerous Balls in Match Outcomes. *Metallurgical and Materials Engineering*, *31*(4), pp.852-866.

Min, X., Li, W., Yang, J., Xie, W. and Zhao, D., 2022. An Improved AdaBoost for Prosecutorial Case-Workload Estimation via Case Grouping. *International Journal of Computational Intelligence Systems*, *15*(1), p.48.

Nagaraj, P., Muneeswaran, V. and Raja, M., 2023, August. IPL Players Cost Pay Prediction using Machine Learning Techniques. In *2023 Second International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)* (pp. 597-602). IEEE.

NS, K.R., Raghunandan, K.R., Rao, S., Patil, S., Nandan, S. and Devadiga, R., 2025, February. Cricket Player's Performance Prediction Using Machine Learning. In *2025 International Conference on Artificial Intelligence and Data Engineering (AIDE)* (pp. 944-950). IEEE.

Priya, S., Gupta, A.K., Dwivedi, A. and Prabhakar, A., 2022, April. Analysis and winning prediction in T20 cricket using machine learning. In *2022 Second International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)* (pp. 1-4). IEEE.

Puram, P., Roy, S., Srivastav, D. and Gurumurthy, A., 2023. Understanding the effect of contextual factors and decision making on team performance in Twenty20 cricket: an interpretable machine learning approach. *Annals of Operations Research*, *325*(1), pp.261-288.

Raajesh, S., Martin, N., Jiji, J. and Nair, A., 2024, May. Cricket Team Selection and Player Analysis using Data Analytics. In 2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS) (pp. 1-6). IEEE.

Rehman, Z.U., Iqbal, M.M., Safwan, H. and Iqbal, J., 2022. Predict the Match Outcome in Cricket Matches Using Machine Learning. *Manchester Journal of Artificial Intelligence and Applied Sciences*, *3*(1).

Reyaz, N., Ahamad, G., Khan, N.J., Naseem, M. and Ali, J., 2024. SVMCTI: support vector machine based cricket talent ıdentification model. *International Journal of Information Technology*, *16*(3), pp.1931-1944.

Robel, M., Khan, M.A.R., Ahammad, I., Alam, M.M. and Hasan, K., 2024. Cricket players selection for national team and franchise league using machine learning algorithms. Cloud Computing and Data Science, pp.108-139.

Rudin, C., 2019. Stop explaining black box machine learning models for high-stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5), pp.206–215.

Shah, S.A.A., Zaman, Q., Wasim, D., Allohibi, J., Alharbi, A.A. and Shabbir, M., 2025. Optimal model for predicting highest runs chase outcomes in T-20 international cricket using modern classification algorithms. *Alexandria Engineering Journal*, *114*, pp.588-598.

Siddiqui, H.U.R., Younas, F., Rustam, F., Flores, E.S., Ballester, J.B., Diez, I.D.L.T., Dudley, S. and Ashraf, I., 2023. Enhancing cricket performance analysis with human pose estimation and machine learning. *Sensors*, *23*(15), p.6839.

Subburaj, M., Rao, G.R.K., Parashar, B., Jeyabalan, I., Semban, H. and Sengan, S., 2023. Artificial intelligence for smart in match winning prediction in Twenty20 cricket league using machine learning model. In Artificial Intelligence for Smart Healthcare (pp. 31-46). Cham: Springer International Publishing.

Sudhini, T.R., Syasani, A., Srikar, B., Kodipaka, S. and Kumar, K.R., 2025. Analysis of Cricket Score Predictor using Machine Learning.

Suguna, R., Kumar, Y.P., Prakash, J.S., Neethu, P.S. and Kiran, S., 2023, December. Utilizing machine learning for sport data analytics in cricket: score prediction and player categorization. In *2023 IEEE 3rd Mysore Sub Section International Conference (MysuruCon)* (pp. 1-6). IEEE.

Sumathi, M., Prabu, S. and Rajkamal, M., 2023, April. Cricket players performance prediction and evaluation using machine learning algorithms. In *2023 international conference on networking and communications (ICNWC)* (pp. 1-6). IEEE.

Sun, X., Davis, J., Schulte, O. and Liu, G., 2020, August. Cracking the black box: Distilling deep sports analytics. In *Proceedings of the 26th acm sigkdd international conference on knowledge discovery & data mining* (pp. 3154-3162).

Tirtho, D., Rahman, S. and Mahbub, M.S., 2022. Cricketer's tournament-wise performance prediction and squad selection using machine learning and multi-objective optimization. Applied Soft Computing, 129, p.109526.

Jianjun, Q., Isleem, H.F., Almoghayer, W.J. and Khishe, M., 2025. Predictive athlete performance modeling with machine learning and biometric data integration. *Scientific Reports*, *15*(1), p.16365.

Vishwarupe, V., Bedekar, M., Joshi, P.M., Pande, M., Pawar, V. and Shingote, P., 2022. Data analytics in the game of cricket: A novel paradigm. *Procedia Computer Science*, *204*, pp.937-944.

Data-set Source:

Kaggle : https://www.kaggle.com/datasets/patrickb1912/ipl-complete-dataset-20082020

Image References:

Arman, A., Di Reto, E., Mecella, M. and Santucci, G., 2025. Functional Size Measurement With Conceptual Models: A Systematic Literature Review. *Journal of Software: Evolution and Process*, *37*(5), p.e70030.

B EYE. (2023). *Sports analytics: A complete handbook for organizations*. Retrieved from https://b-eye.com/blog/sports-analytics-a-complete-handbook-for-organizations/

https://datasportsgroup.com/news-article/114939/how-sports-analytics-is-used-in-ipl/

Salcinovic, B., Drew, M., Dijkstra, P., Waddington, G. and Serpell, B.G., 2022. Factors influencing team performance: what can support teams in high-performance sport learn from other industries? A systematic scoping review. *Sports Medicine-Open*, *8*(1), p.25.

Shakil, F.A., Abdullah, A.H., Momen, S. and Mohammed, N., 2020. Predicting the result of a cricket match by applying data mining techniques. In *Proceedings of the Computational Methods in Systems and Software* (pp. 758-770). Cham: Springer International Publishing.

Spoclearn, 2025. *Sport analytics: How data analytics is changing games*. Available at: https://www.spoclearn.com/blog/sport-analytics/

Tripathi, R., n.d. How Data Analytics Is Being Utilized In the Sports Industry? *LinkedIn*. Available at:https://www.linkedin.com/pulse/how-data-analytics-being-utilized-sports-industry-rishabh-tripathi-op5tf/

Wickramasinghe, I., 2022. Applications of Machine Learning in cricket: A systematic review. *Machine Learning with Applications*, *10*, p.100435.

APPENDIX

In [ ]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_mat
import warnings
warnings.filterwarnings("ignore")
import shap
```

In [ ]:
```
match_df= pd.read_csv("/content/matches.csv")
match_df.head()
```

Out[ ]:

	id	season	city	date	match_type	player_of_match	venue	
0	335982	2007/08	Bangalore	2008-04-18	League	BB McCullum	M Chinnaswamy Stadium	Cha Ba
1	335983	2007/08	Chandigarh	2008-04-19	League	MEK Hussey	Punjab Cricket Association Stadium, Mohali	
2	335984	2007/08	Delhi	2008-04-19	League	MF Maharoof	Feroz Shah Kotla	Da
3	335985	2007/08	Mumbai	2008-04-20	League	MV Boucher	Wankhede Stadium	
4	335986	2007/08	Kolkata	2008-04-20	League	DJ Hussey	Eden Gardens	

In [ ]:
```
delivery_df= pd.read_csv("/content/deliveries.csv")
delivery_df.head()
```

Out[ ]:

	match_id	inning	batting_team	bowling_team	over	ball	batter	bowler	non_s
0	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	1	SC Ganguly	P Kumar	McC
1	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	2	BB McCullum	P Kumar	SC Ga
2	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	3	BB McCullum	P Kumar	SC Ga
3	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	4	BB McCullum	P Kumar	SC Ga
4	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	5	BB McCullum	P Kumar	SC Ga

In [ ]:
```
delivery_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 260920 entries, 0 to 260919
Data columns (total 17 columns):
 #   Column            Non-Null Count   Dtype 
---  ------            --------------   ----- 
 0   match_id          260920 non-null  int64 
 1   inning            260920 non-null  int64 
 2   batting_team      260920 non-null  object
 3   bowling_team      260920 non-null  object
 4   over              260920 non-null  int64 
 5   ball              260920 non-null  int64 
 6   batter            260920 non-null  object
 7   bowler            260920 non-null  object
 8   non_striker       260920 non-null  object
 9   batsman_runs      260920 non-null  int64 
 10  extra_runs        260920 non-null  int64 
 11  total_runs        260920 non-null  int64 
 12  extras_type       14125 non-null   object
 13  is_wicket         260920 non-null  int64 
 14  player_dismissed  12950 non-null   object
 15  dismissal_kind    12950 non-null   object
 16  fielder           9354 non-null    object
dtypes: int64(8), object(9)
memory usage: 33.8+ MB
```

In [ ]:
```
match_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1095 entries, 0 to 1094
Data columns (total 20 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   id               1095 non-null   int64  
 1   season           1095 non-null   object 
 2   city             1044 non-null   object 
 3   date             1095 non-null   object 
 4   match_type       1095 non-null   object 
 5   player_of_match  1090 non-null   object 
 6   venue            1095 non-null   object 
 7   team1            1095 non-null   object 
 8   team2            1095 non-null   object 
 9   toss_winner      1095 non-null   object 
 10  toss_decision    1095 non-null   object 
 11  winner           1090 non-null   object 
 12  result           1095 non-null   object 
 13  result_margin    1076 non-null   float64
 14  target_runs      1092 non-null   float64
 15  target_overs     1092 non-null   float64
 16  super_over       1095 non-null   object 
 17  method           21 non-null     object 
 18  umpire1          1095 non-null   object 
 19  umpire2          1095 non-null   object 
dtypes: float64(3), int64(1), object(16)
memory usage: 171.2+ KB
```

In [ ]:
```
delivery_df.shape
```

Out[ ]:
```
(260920, 17)
```

In [ ]:
```
match_df.shape
```

Out[ ]:
```
(1095, 20)
```

In [ ]:
```
delivery_df.duplicated().sum()
```

Out[ ]:
```
np.int64(0)
```

In [ ]:
```
match_df.duplicated().sum()
```

Out[ ]:
```
np.int64(0)
```

In [ ]:
```
delivery_df.isnull().sum()
```

Out[ ]:

	0
match_id	0
inning	0
batting_team	0
bowling_team	0
over	0
ball	0
batter	0
bowler	0
non_striker	0
batsman_runs	0
extra_runs	0
total_runs	0
extras_type	246795
is_wicket	0
player_dismissed	247970
dismissal_kind	247970
fielder	251566

dtype: int64

In [ ]:
```
match_df.isnull().sum()
```

Out[ ]:

	0
id	0
season	0
city	51
date	0
match_type	0
player_of_match	5
venue	0
team1	0
team2	0
toss_winner	0
toss_decision	0
winner	5
result	0
result_margin	19
target_runs	3
target_overs	3
super_over	0
method	1074
umpire1	0
umpire2	0

dtype: int64

In [ ]:
```
#checking Unique value from venue
match_df[match_df['city'].isna()]['venue'].unique()
```

Out[ ]:
```
array(['Sharjah Cricket Stadium', 'Dubai International Cricket Stadium'],
      dtype=object)
```

In [ ]:
```
#Filling missing city values based on venue
match_df.loc[(match_df['city'].isna()) & (match_df['venue'] == 'Sharjah Cricket
match_df.loc[(match_df['city'].isna()) & (match_df['venue'] == 'Dubai Internatio
match_df['city'].isnull().sum()
```

Out[ ]:
```
np.int64(0)
```

In [ ]:
```
#checking Unique value from season
match_df['season'].unique()
```

Out[ ]:
```
array(['2007/08', '2009', '2009/10', '2011', '2012', '2013', '2014',
       '2015', '2016', '2017', '2018', '2019', '2020/21', '2021', '2022',
       '2023', '2024'], dtype=object)
```

In [ ]:
```
#replacing the value with orignal season
match_df.replace({"season":{"2007/08":"2008","2009/10":"2010","2020/21":"2020"}}
```

In [ ]:
```
match_df.season.unique()
```

Out[ ]:
```
array(['2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015',
       '2016', '2017', '2018', '2019', '2020', '2021', '2022', '2023',
       '2024'], dtype=object)
```

In [ ]:
```
#Replacing Team old name to new name
team_map ={"Mumbai Indians":"Mumbai Indians",
           "Chennai Super Kings":"Chennai Super Kings",
           "Kolkata Knight Riders":"Kolkata Knight Riders",
           "Royal Challengers Bangalore":"Royal Challengers Bangalore",
           "Royal Challengers Bengaluru":"Royal Challengers Bangalore",
           "Rajasthan Royals":"Rajasthan Royals",
           "Kings XI Punjab":"Kings XI Punjab",
           "Punjab Kings":"Kings XI Punjab",
           "Sunrisers Hyderabad":"Sunrisers Hyderabad",
           "Deccan Chargers":"Sunrisers Hyderabad",
           "Delhi Capitals":"Delhi Capitals",
           "Delhi Daredevils":"Delhi Capitals",
           "Gujarat Titans":"Gujarat Titans",
           "Gujarat Lions":"Gujarat Titans",
           "Lucknow Super Giants":"Lucknow Super Giants",
           "Pune Warriors":"Pune Warriors",
           "Rising Pune Supergiant":"Pune Warriors",
           "Rising Pune Supergiants":"Pune Warriors",
           "Kochi Tuskers Kerala":"Kochi Tuskers Kerala"}

#For Match table
match_df['team1']= match_df['team1'].map(team_map)
match_df['team2']= match_df['team2'].map(team_map)
match_df['winner']= match_df['winner'].map(team_map)
match_df['toss_winner']= match_df['toss_winner'].map(team_map)

#For Deliverise Tables
delivery_df['batting_team']= delivery_df['batting_team'].map(team_map)
delivery_df['bowling_team']= delivery_df['bowling_team'].map(team_map)
```

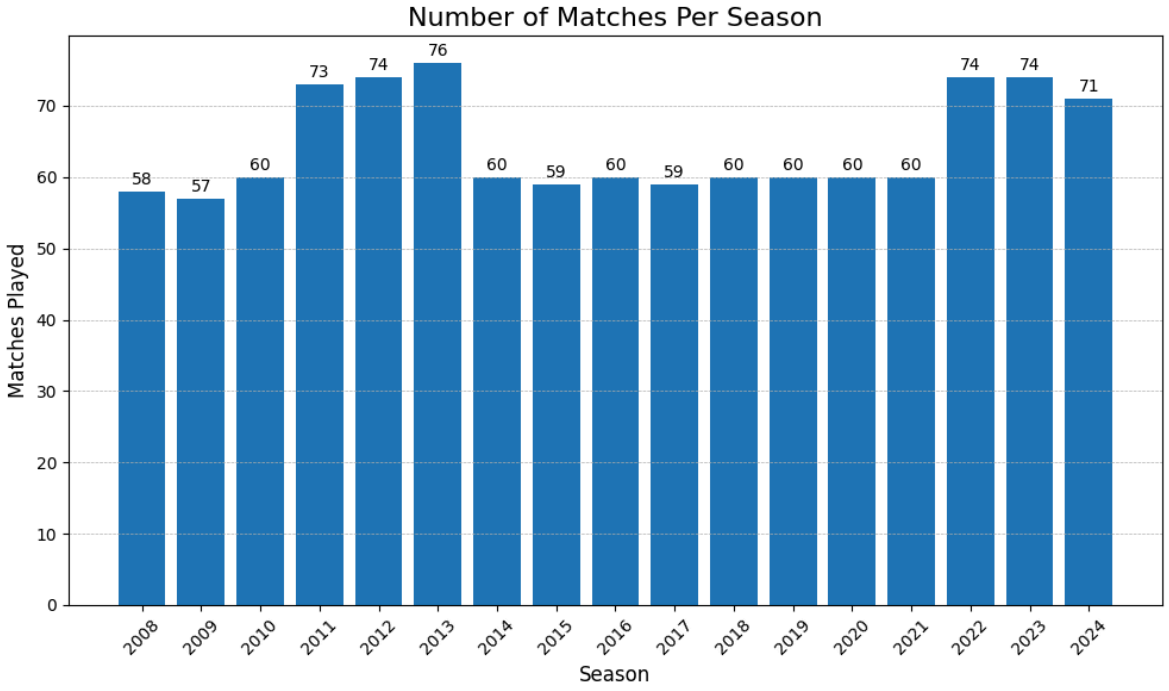
In [ ]:
```
#Total matches per season
match_df['season'].value_counts().to_frame().T
```

Out[ ]:

season	2013	2022	2012	2023	2011	2024	2019	2016	2010	2021	2020	2014
count	76	74	74	74	73	71	60	60	60	60	60	60

In [ ]:
```
season_counts = match_df['season'].value_counts().sort_index()

# Plotting as a bar chart
plt.figure(figsize=(10, 6))
plt.bar(season_counts.index, season_counts.values)
```

```
# Titles and labels
plt.title('Number of Matches Per Season', fontsize=16)
plt.xlabel('Season', fontsize=12)
plt.ylabel('Matches Played', fontsize=12)

# Annotate each bar with its value
for i, value in enumerate(season_counts.values):
    plt.text(i, value + 1, str(value), fontsize=10, ha='center')

plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()
```



In [ ]:
```
#Matches Hosted by Top 10 City
match_df['city'].value_counts().head(10)
```

Out[ ]:

city	count
Mumbai	173
Kolkata	93
Delhi	90
Chennai	85
Hyderabad	77
Bangalore	65
Chandigarh	61
Jaipur	57
Pune	51
Dubai	46

dtype: int64

In [ ]:
```
# city_counts = match_df['city'].fillna('Unknown').value_counts()

# # Plot
# plt.figure(figsize=(12, 6))
# sns.barplot(x=city_counts.index, y=city_counts.values, palette='viridis')

# plt.title('Number of Matches Hosted by Each City', fontsize=16)
# plt.xlabel('City', fontsize=12)
# plt.ylabel('Number of Matches', fontsize=12)
# plt.xticks(rotation=90)
# plt.tight_layout()
# plt.show()
```

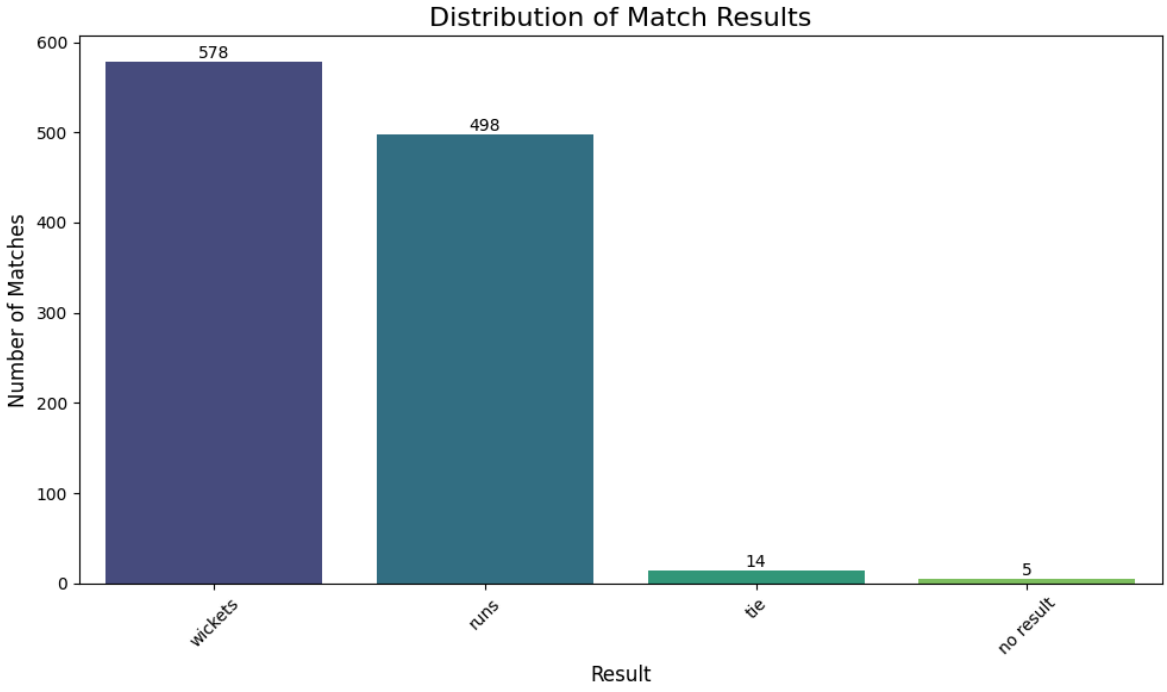
In [ ]:
```
mean_target_runs = match_df['target_runs'].mean()

# Creating histogram plot to show the distribution

plt.figure(figsize=(8, 6))
sns.boxplot(
    x=match_df['target_runs'].dropna(),
    whis=1.5,
    showfliers=True
)
sns.stripplot(
    x=match_df['target_runs'].dropna(),
    color='black',
    size=3,
    alpha=0.4,
    jitter=True
)

plt.title('Distribution of Target Runs', fontsize=16)
plt.xlabel('Target Runs', fontsize=12)
```

```
plt.tight_layout()
plt.show()
```

In [ ]:
```
#Average Target Run in IPL Matches
print(mean_target_runs)
```

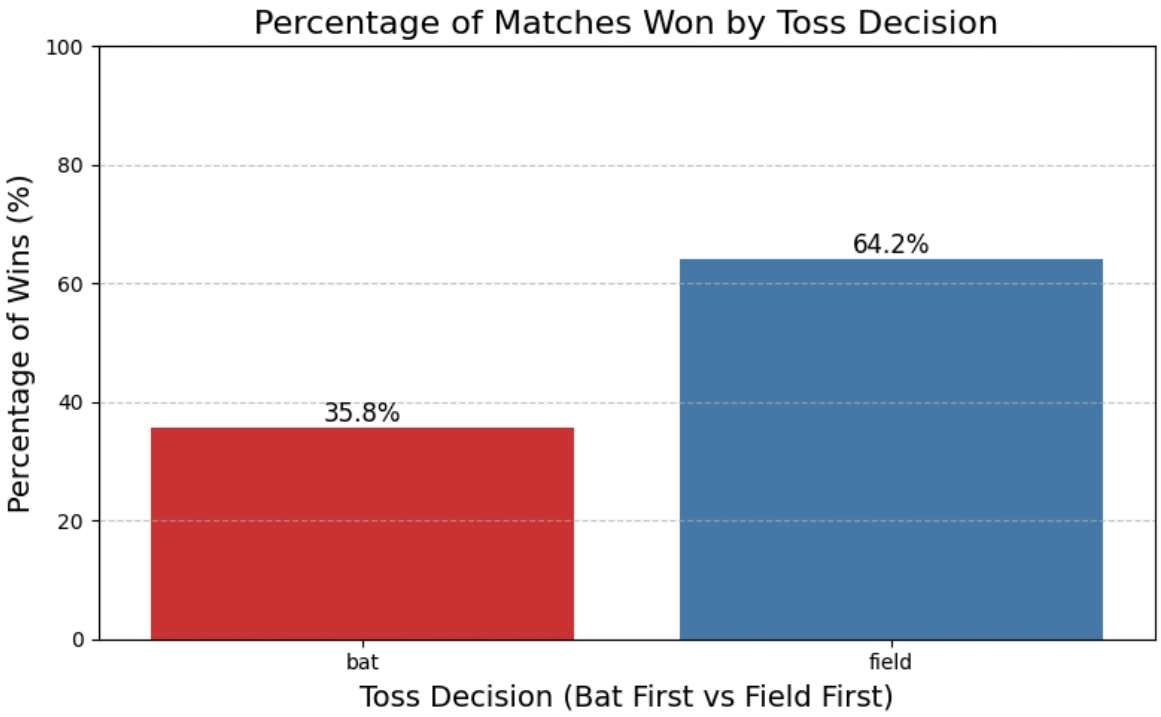
```
165.68406593406593
```

In [ ]:
```
# Distribution of Match Results
result_distribution = match_df['result'].value_counts()

# Creating the bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x=result_distribution.index, y=result_distribution.values, palette='

# Adding titles and labels
plt.title('Distribution of Match Results', fontsize=16)
plt.xlabel('Result', fontsize=12)
plt.ylabel('Number of Matches', fontsize=12)

# Showing the values on top of the bars
for index, value in enumerate(result_distribution):
    plt.text(index, value, f'{value}', ha='center', va='bottom')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



In [ ]:
```
clean_df = match_df.dropna(subset=['winner'])

# Grouping by toss decision and counting how often each decision leads to a win
wins_by_decision = clean_df.groupby('toss_decision').size().reset_index(name='wi

# Calculating the percentage of matches won by teams batting first vs. fielding
total_matches = clean_df.shape[0]
wins_by_decision['percentage'] = (wins_by_decision['wins'] / total_matches) * 10

# Displaying the result
print(wins_by_decision)

# Plotting the results
plt.figure(figsize=(8, 5))
sns.barplot(data=wins_by_decision, x='toss_decision', y='percentage', palette='S

# Adding titles and labels
plt.title('Percentage of Matches Won by Toss Decision', fontsize=16)
plt.xlabel('Toss Decision (Bat First vs Field First)', fontsize=14)
plt.ylabel('Percentage of Wins (%)', fontsize=14)

# Adding values on top of the bars
for index, row in wins_by_decision.iterrows():
    plt.text(index, row['percentage'] + 1, f'{row["percentage"]:.1f}%',
             ha='center', fontsize=12)

plt.ylim(0, 100)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

```
  toss_decision  wins  percentage
0           bat   390   35.779817
1         field   700   64.220183
```



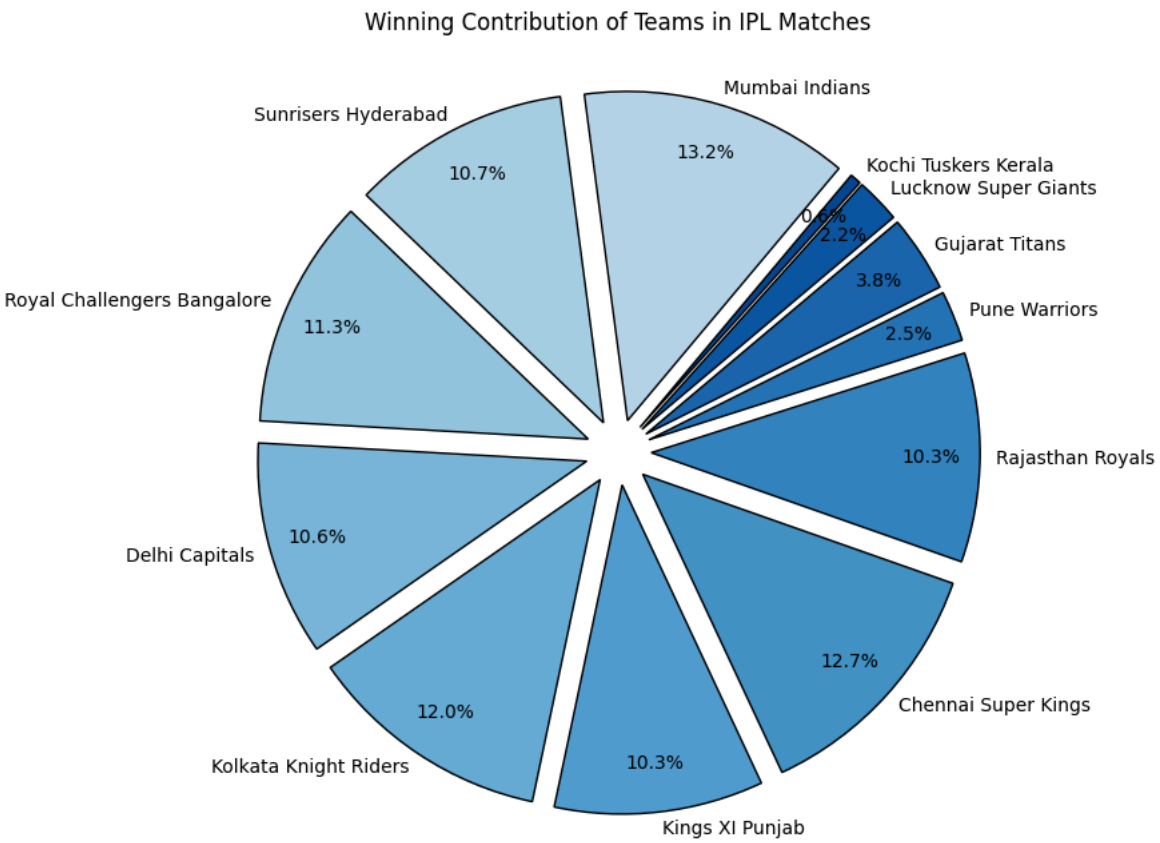
In [ ]:
```
#total match played by team
total_matches= (match_df['team1'].value_counts()+match_df['team2'].value_counts(
# Count of wins for each team
total_wins = match_df['winner'].value_counts().rename_axis('Team').reset_index(n

# Count of wins when chasing a target
team_chasing = match_df[match_df['toss_decision'] == 'field']['winner'].value_co

# Count of wins when batting first
team_batting_first = match_df[match_df['toss_decision'] == 'bat']['winner'].valu


# Merge all the DataFrames
merged_df = total_matches.merge(total_wins,on='Team',how='outer') \
    .merge(team_chasing, on='Team', how='outer') \
    .merge(team_batting_first, on='Team', how='outer')


# Fill NaN values with 0
merged_df= merged_df.fillna(0)
#sorting by total match played
merged_df.sort_values(by='Total_Match_Played', ascending=False, inplace=True)
```

In [ ]:
```
merged_df.set_index('Team', drop=True).sort_values(by='Total_Wins', ascending=Fa
```

Out[ ]:

Team	Total_Match_Played	Total_Wins	Chasing_Wins	Batting_First_Wins
Mumbai Indians	261	144	90	54.0
Chennai Super Kings	238	138	75	63.0
Kolkata Knight Riders	251	131	81	50.0
Royal Challengers Bangalore	255	123	85	38.0
Sunrisers Hyderabad	257	117	73	44.0
Delhi Capitals	252	115	73	42.0
Kings XI Punjab	246	112	84	28.0
Rajasthan Royals	221	112	69	43.0
Gujarat Titans	75	41	30	11.0
Pune Warriors	76	27	16	11.0
Lucknow Super Giants	44	24	18	6.0
Kochi Tuskers Kerala	14	6	6	0.0

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

# Creating a pie chart for showing wining destribution of team in all wining mat
plt.figure(figsize=(8, 8))
explode = [0.1] * len(merged_df)
plt.pie(
    merged_df['Total_Wins'],
    labels=merged_df['Team'],
    autopct='%1.1f%%',
    startangle=50,
    colors=plt.cm.Blues(np.linspace(0.3, 0.9, len(merged_df))),
    wedgeprops={'edgecolor': 'black'},
    explode=explode,
    pctdistance=0.85,
    labeldistance=1.05
)

plt.title('Winning Contribution of Teams in IPL Matches')

# Show the pie chart
plt.show()
```



In [ ]:
```
# Create a copy of the merged DataFrame to keep the original intact
percentage_df = merged_df.copy()

# Convert columns to percentage
percentage_df['Total_Wins'] = (percentage_df['Total_Wins'] / percentage_df['Tota
percentage_df['Chasing_Wins'] = (percentage_df['Chasing_Wins'] / merged_df['Tota
percentage_df['Batting_First_Wins'] = (percentage_df['Batting_First_Wins'] / mer
```

In [ ]:
```
percentage_df.set_index('Team', drop=True).sort_values(by='Total_Wins', ascendin
```

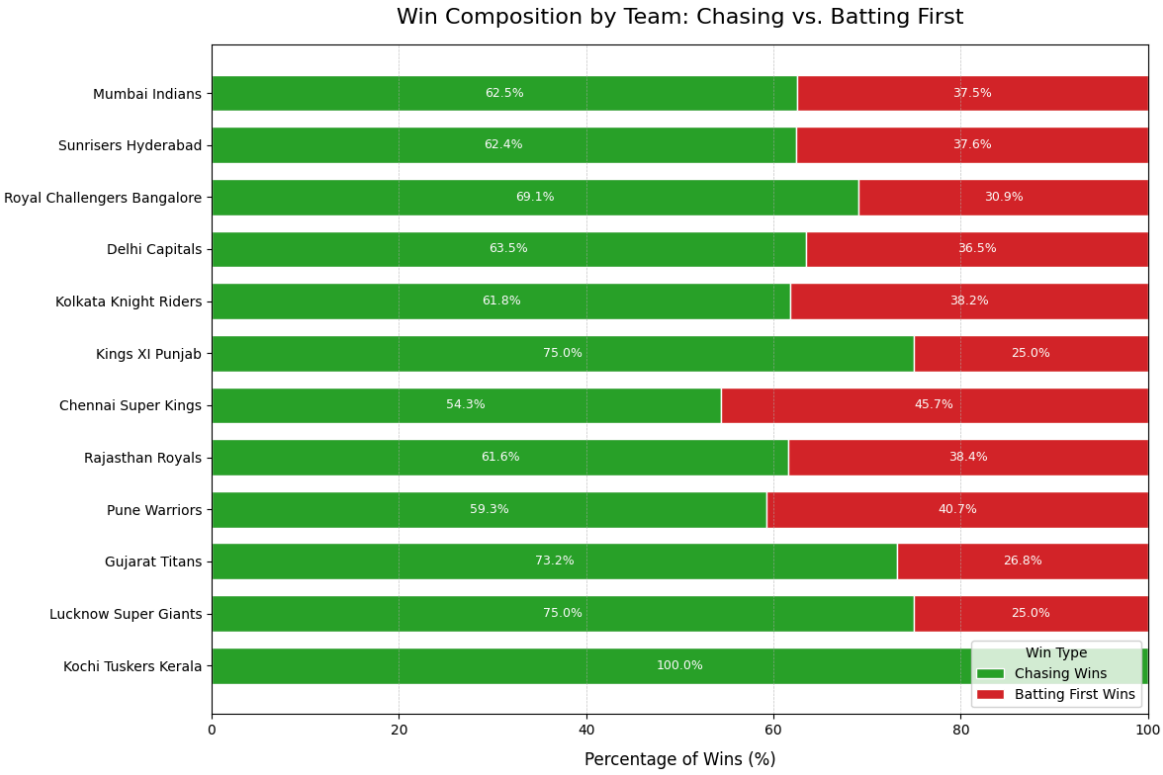
Out[ ]:

Team	Total_Match_Played	Total_Wins	Chasing_Wins	Batting_First_Wins
Chennai Super Kings	238	57.983193	54.347826	45.652174
Mumbai Indians	261	55.172414	62.500000	37.500000
Gujarat Titans	75	54.666667	73.170732	26.829268
Lucknow Super Giants	44	54.545455	75.000000	25.000000
Kolkata Knight Riders	251	52.191235	61.832061	38.167939
Rajasthan Royals	221	50.678733	61.607143	38.392857
Royal Challengers Bangalore	255	48.235294	69.105691	30.894309
Delhi Capitals	252	45.634921	63.478261	36.521739
Kings XI Punjab	246	45.528455	75.000000	25.000000
Sunrisers Hyderabad	257	45.525292	62.393162	37.606838
Kochi Tuskers Kerala	14	42.857143	100.000000	0.000000
Pune Warriors	76	35.526316	59.259259	40.740741

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt

# Prepare and normalize the data
stack_df = percentage_df.set_index('Team')[['Chasing_Wins', 'Batting_First_Wins'
stack_norm = stack_df.div(stack_df.sum(axis=1), axis=0) * 100

# Custom colors for the two segments
colors = ['#2ca02c', '#d62728']  # green for chasing, red for batting first

fig, ax = plt.subplots(figsize=(12, 8))

teams = stack_norm.index
left = np.zeros(len(stack_norm))

# Draw each segment
for idx, col in enumerate(stack_norm.columns):
    vals = stack_norm[col].values
    ax.barh(teams, vals, left=left, color=colors[idx], edgecolor='white', height
    # Annotate inside each bar
    for i, (l, v) in enumerate(zip(left, vals)):
        if v > 3:  # only annotate if segment is wide enough
            ax.text(l + v/2, i, f"{v:.1f}%", va='center', ha='center', color='wh
    left += vals
```

```
# plot
ax.set_xlim(0, 100)
ax.set_xlabel('Percentage of Wins (%)', fontsize=12, labelpad=10)
ax.set_title('Win Composition by Team: Chasing vs. Batting First', fontsize=16,
ax.legend(title='Win Type', loc='lower right')
ax.invert_yaxis()  # highest on top
ax.grid(axis='x', linestyle='--', linewidth=0.5, alpha=0.7)

plt.tight_layout()
plt.show()
```



In [ ]:

```
total_matches= (match_df['team1'].value_counts()+match_df['team2'].value_counts(

#total toss winner teams
toss_winners=match_df['toss_winner'].value_counts().rename_axis('Team').reset_in

# Create the pivot table for toss decisions
team_wins_pivot = pd.pivot_table(
    data=match_df[match_df['toss_winner'] == match_df['winner']],
    index='winner',
    columns='toss_decision',
    aggfunc='size',
    fill_value=0
).rename_axis('Team').reset_index().rename(columns={'bat': 'Bat_first_win_After_

# Count of matches won by toss winners
toss_winner_matches = match_df[match_df['toss_winner'] == match_df['winner']]['w

# Merge all the DataFrames
merged_df2 = total_matches.merge(toss_winners, on='Team', how='outer') \
    .merge(toss_winner_matches, on='Team', how='outer') \
    .merge(team_wins_pivot, on='Team', how='outer')


# Fill NaN values with 0
```

```
merged_df2= merged_df2.fillna(0)
#sorting by total match played
merged_df2.sort_values(by='Total_Match_Played', ascending=False, inplace=True)
```

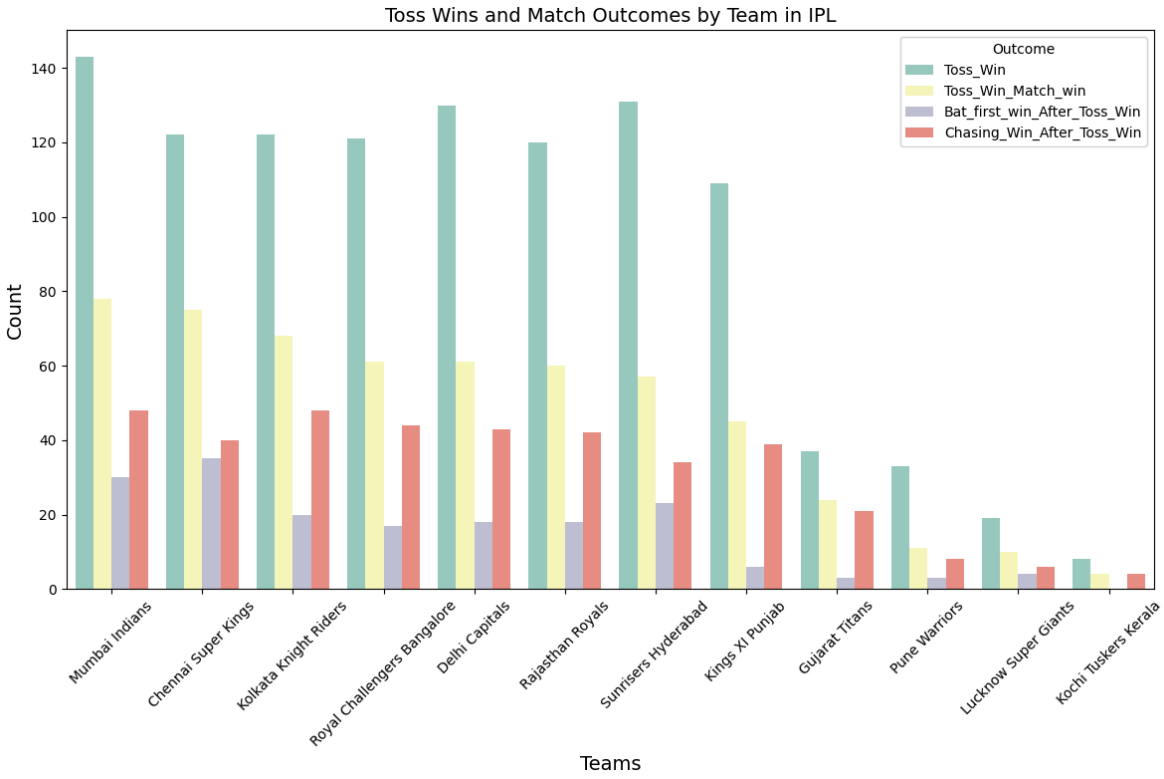
In [ ]:
```
# Display the final merged DataFrame
sorted_df = merged_df2.set_index('Team').sort_values(by='Toss_Win_Match_win', as
sorted_df
```

Out[ ]:

Team	Total_Match_Played	Toss_Win	Toss_Win_Match_win	Bat_first_win_After_Toss
Mumbai Indians	261	143	78	
Chennai Super Kings	238	122	75	
Kolkata Knight Riders	251	122	68	
Royal Challengers Bangalore	255	121	61	
Delhi Capitals	252	130	61	
Rajasthan Royals	221	120	60	
Sunrisers Hyderabad	257	131	57	
Kings XI Punjab	246	109	45	
Gujarat Titans	75	37	24	
Pune Warriors	76	33	11	
Lucknow Super Giants	44	19	10	
Kochi Tuskers Kerala	14	8	4	

In [ ]:
```
# Melting the DataFrame for easier plotting
melted_df = pd.melt(
    sorted_df.reset_index(),
    id_vars=['Team'],
    value_vars=['Toss_Win', 'Toss_Win_Match_win', 'Bat_first_win_After_Toss_Win'
    var_name='Outcome',
    value_name='Count'
```

```
)

plt.figure(figsize=(12, 8))

# Creating the bar plot
sns.barplot(data=melted_df, x='Team', y='Count', hue='Outcome', palette='Set3')

# Adding labels and title
plt.xlabel('Teams',fontsize=14)
plt.ylabel('Count',fontsize=14)
plt.title('Toss Wins and Match Outcomes by Team in IPL',fontsize=14)
plt.xticks(rotation=45)
plt.legend(title='Outcome')
plt.tight_layout()
plt.show()
```

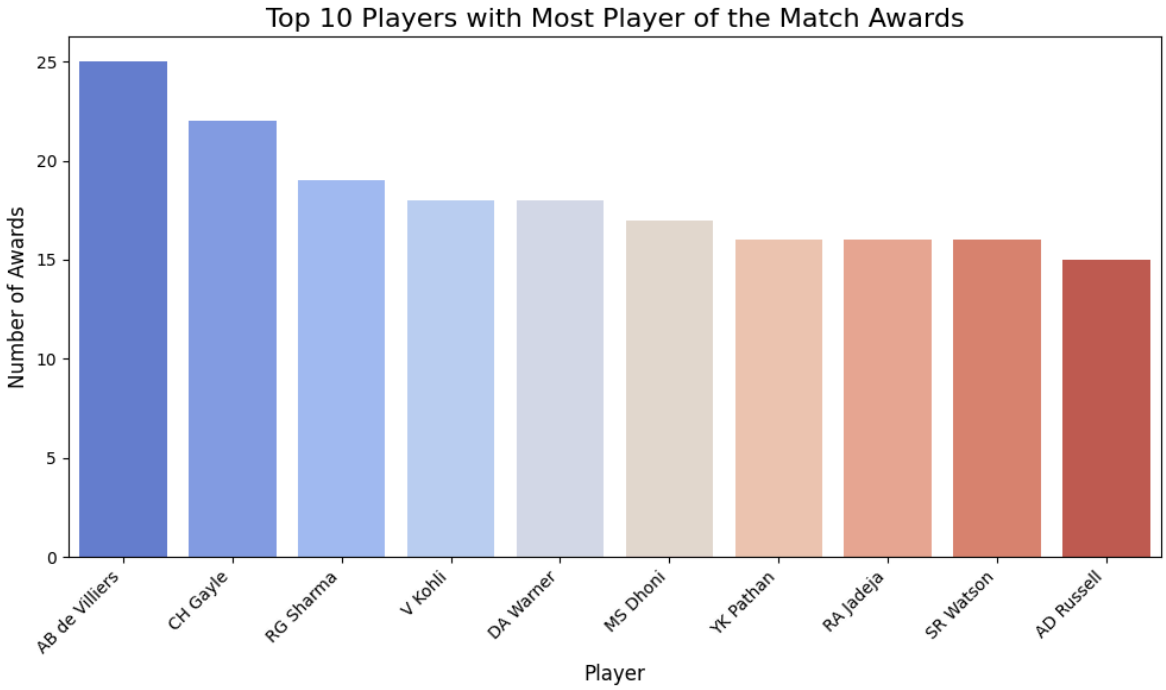


Batter Analysis

In [ ]:

```
#players have won the most 'Player of the Match' awards
match_df['player_of_match'].value_counts().head() #Top 5

# Ploting Top 10 Players with Most Player of the Match Awards
player_counts = match_df['player_of_match'].value_counts().head(10)

# Creating a bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x=player_counts.index, y=player_counts.values, palette='coolwarm')

# Adding titles and labels
plt.title('Top 10 Players with Most Player of the Match Awards', fontsize=16)
plt.xlabel('Player', fontsize=12)
plt.ylabel('Number of Awards', fontsize=12)

plt.xticks(rotation=45, ha='right')
plt.tight_layout()
```

```
plt.show()
print(player_counts)
```

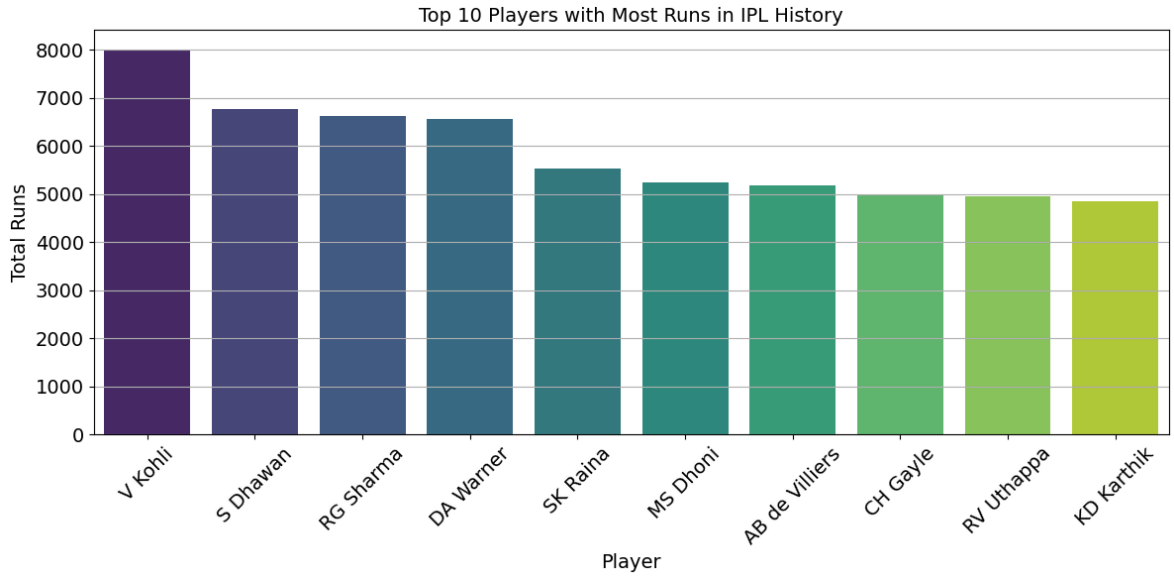



```
player_of_match
AB de Villiers    25
CH Gayle          22
RG Sharma         19
V Kohli           18
DA Warner         18
MS Dhoni          17
YK Pathan         16
RA Jadeja         16
SR Watson         16
AD Russell        15
Name: count, dtype: int64
```

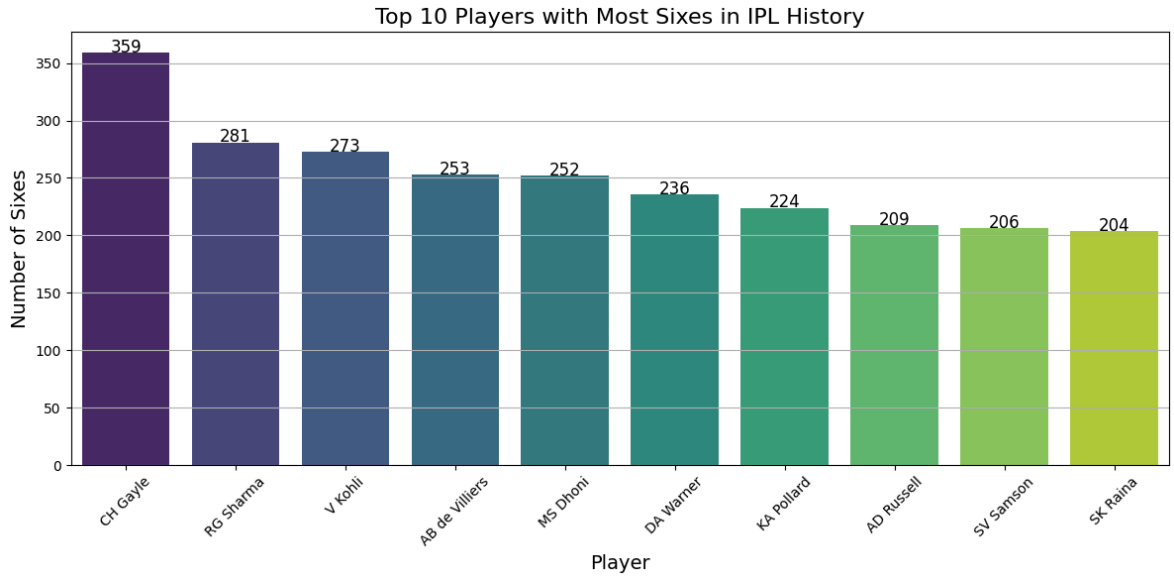
In [ ]:

```
player_runs = delivery_df[['batter', 'batsman_runs']]
player_runs = player_runs.groupby('batter')['batsman_runs'].sum().reset_index(na

# Sorting the players by total runs
top_players = player_runs.sort_values(by='total_runs', ascending=False).head(10)

print(top_players.reset_index(drop=True))

# Plotting the data
plt.figure(figsize=(12, 6))
bar_plot = sns.barplot(data=top_players, x='batter', y='total_runs', palette='vi

# Adding titles and labels
plt.title('Top 10 Players with Most Runs in IPL History', fontsize=14)
plt.xlabel('Player', fontsize=14)
plt.ylabel('Total Runs', fontsize=14)
plt.xticks(rotation=45, fontsize=14)
plt.yticks(fontsize=14)
plt.grid(axis='y')

plt.tight_layout()
plt.show()
```

	batter	total_runs
0	V Kohli	8014
1	S Dhawan	6769
2	RG Sharma	6630
3	DA Warner	6567
4	SK Raina	5536
5	MS Dhoni	5243
6	AB de Villiers	5181
7	CH Gayle	4997
8	RV Uthappa	4954
9	KD Karthik	4843

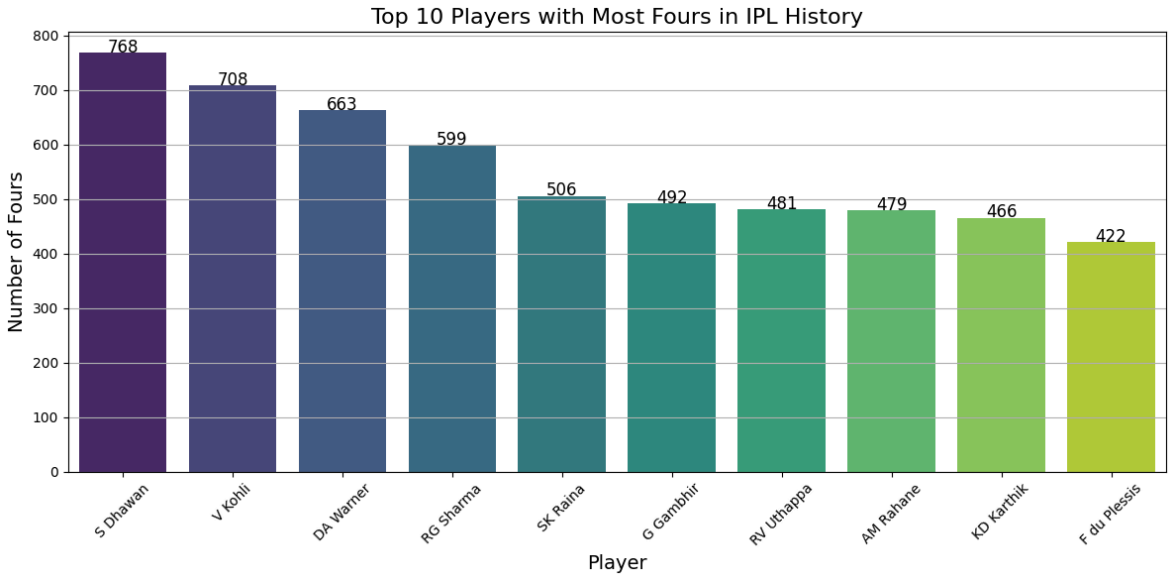


In [ ]:
```
most_sixes = delivery_df[delivery_df['batsman_runs'] == 6]['batter'].value_count
most_sixes_df = most_sixes.reset_index()
most_sixes_df.columns = ['batter', 'sixes']

# Plotting the data
plt.figure(figsize=(12, 6))
bar_plot = sns.barplot(data=most_sixes_df, x='batter', y='sixes', palette='virid
for index, row in most_sixes_df.iterrows():
    bar_plot.text(index, row.sixes + 0.1, row.sixes, color='black', ha="center",

# Adding titles and labels
plt.title('Top 10 Players with Most Sixes in IPL History', fontsize=16)
plt.xlabel('Player', fontsize=14)
plt.ylabel('Number of Sixes', fontsize=14)
plt.xticks(rotation=45)
plt.grid(axis='y')

plt.tight_layout()
plt.show()
print(most_sixes_df)
```



	batter	sixes
0	CH Gayle	359
1	RG Sharma	281
2	V Kohli	273
3	AB de Villiers	253
4	MS Dhoni	252
5	DA Warner	236
6	KA Pollard	224
7	AD Russell	209
8	SV Samson	206
9	SK Raina	204

In [ ]:
```
most_fours = delivery_df[delivery_df['batsman_runs'] == 4]['batter'].value_count
most_fours_df = most_fours.reset_index()
most_fours_df.columns = ['batter', 'fours']

# Plotting the data
plt.figure(figsize=(12, 6))
bar_plot = sns.barplot(data=most_fours_df, x='batter', y='fours', palette='virid
for index, row in most_fours_df.iterrows():
    bar_plot.text(index, row.fours + 0.1, row.fours, color='black', ha="center",

# Adding titles and labels
plt.title('Top 10 Players with Most Fours in IPL History', fontsize=16)
plt.xlabel('Player', fontsize=14)
plt.ylabel('Number of Fours', fontsize=14)
plt.xticks(rotation=45)
plt.grid(axis='y')

plt.tight_layout()
plt.show()
print(most_fours_df)
```



	batter	fours
0	S Dhawan	768
1	V Kohli	708
2	DA Warner	663
3	RG Sharma	599
4	SK Raina	506
5	G Gambhir	492
6	RV Uthappa	481
7	AM Rahane	479
8	KD Karthik	466
9	F du Plessis	422

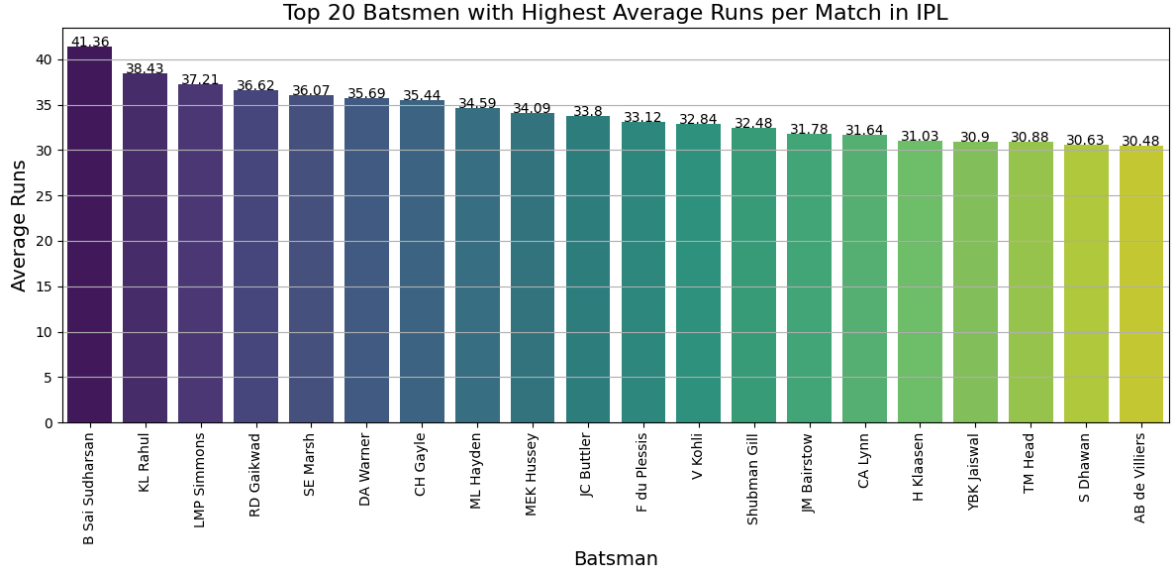
In [ ]:
```
#Objective: batsmen have the highest average runs per match?
batter_data= delivery_df[['match_id','batter','batsman_runs']]  # Filtering Rele

#Calculating the number of matches each batsman has played
batter_matches = batter_data.groupby('batter')['match_id'].nunique().reset_index

#Calculating the total runs for each batsman
batter_total_runs = batter_data.groupby('batter')['batsman_runs'].sum().reset_in

#Merging the matches played and total runs dataframes
batter_summary = pd.merge(batter_matches, batter_total_runs, on='batter')

#Calculating the average runs per match
batter_summary['Avg_runs'] = batter_summary['total_runs'] / batter_summary['matc

#Filtering data for players who have played a minimum of 25 matches in the IPL f
batter_summary = batter_summary[batter_summary['matches_played']>=25]

# Sorting the batsmen by average runs in descending order
batter_avg = batter_summary.sort_values(by='Avg_runs', ascending=False).reset_in

# Displaying the result
batter_avg
```

Out[ ]:

	batter	matches_played	total_runs	Avg_runs
0	B Sai Sudharsan	25	1034	41.360000
1	KL Rahul	122	4689	38.434426
2	LMP Simmons	29	1079	37.206897
3	RD Gaikwad	65	2380	36.615385
4	SE Marsh	69	2489	36.072464
...	...	...	...	...
166	Z Khan	27	117	4.333333
167	UT Yadav	48	208	4.333333
168	JJ Bumrah	26	68	2.615385
169	Sandeep Sharma	25	54	2.160000
170	RP Singh	28	52	1.857143

171 rows × 4 columns

In [ ]:
```
#Plotting the data for top 20 batsmen with the highest average runs
plt.figure(figsize=(12, 6))
bar_plot = sns.barplot(data=batter_avg.head(20), x='batter', y='Avg_runs', palet

#Annotating the bars with average runs
for index, row in batter_avg.head(20).iterrows():
    bar_plot.text(index, row.Avg_runs, round(row.Avg_runs, 2), color='black', ha

# Adding titles and labels
plt.title('Top 20 Batsmen with Highest Average Runs per Match in IPL', fontsize=
plt.xlabel('Batsman', fontsize=14)
plt.ylabel('Average Runs', fontsize=14)
plt.xticks(rotation=90)
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```



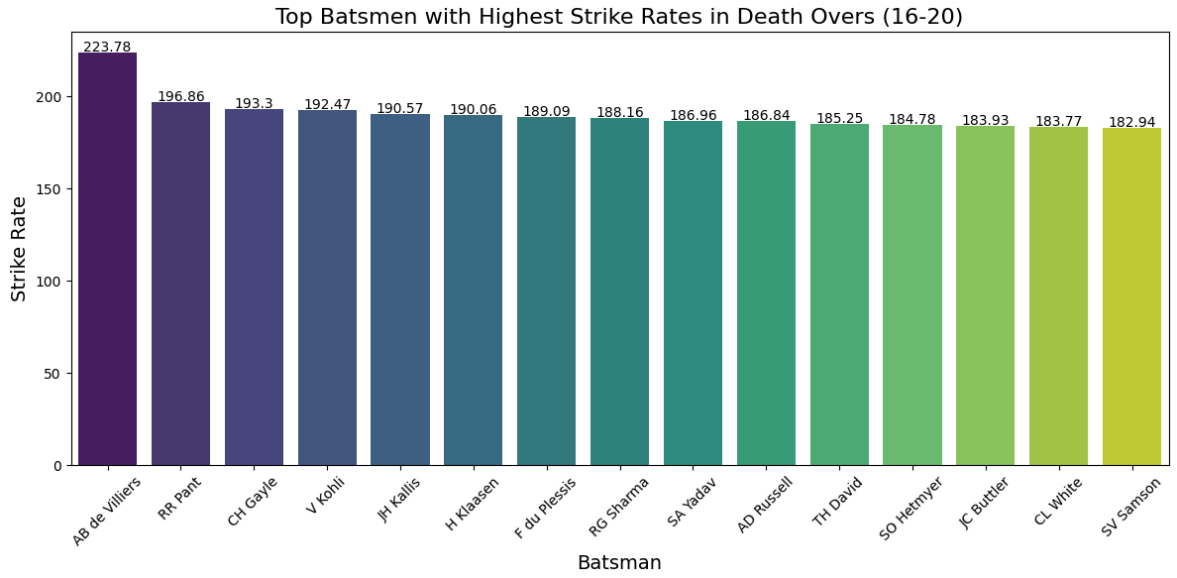
In [ ]:
```
# Filtering data for final overs (overs 16 to 20)
final_overs_data = delivery_df[(delivery_df['over'] >= 16) & (delivery_df['over'

# Calculating runs and balls faced by each batsman in the death overs
batsman_stats = final_overs_data.groupby('batter').agg(
    runs_scored=('batsman_runs', 'sum'),
    balls_faced=('ball', 'count')
).reset_index()

# Filtering batsmen who have faced at least 150 balls for dare comparison
batsman_stats = batsman_stats[batsman_stats['balls_faced'] >= 150]

# Calculating strike rate
batsman_stats['strike_rate'] = (batsman_stats['runs_scored'] / batsman_stats['ba

# Sorting by strike rate in descending order
top_strike_rates = batsman_stats.sort_values(by='strike_rate', ascending=False).

# Displaying the top batsmen with the highest strike rates in final overs
top_15_strike_rates = top_strike_rates.head(15)

top_15_strike_rates
```

Out[ ]:

	batter	runs_scored	balls_faced	strike_rate
0	AB de Villiers	1421	635	223.779528
1	RR Pant	626	318	196.855346
2	CH Gayle	404	209	193.301435
3	V Kohli	1099	571	192.469352
4	JH Kallis	303	159	190.566038
5	H Klaasen	306	161	190.062112
6	F du Plessis	416	220	189.090909
7	RG Sharma	1176	625	188.160000
8	SA Yadav	516	276	186.956522
9	AD Russell	1065	570	186.842105
10	TH David	452	244	185.245902
11	SO Hetmyer	680	368	184.782609
12	JC Buttler	412	224	183.928571
13	CL White	283	154	183.766234
14	SV Samson	547	299	182.943144

In [ ]:
```
plt.figure(figsize=(12, 6))
sns.barplot(data=top_15_strike_rates, x='batter', y='strike_rate', palette='viri

# Adding title and labels
plt.title('Top Batsmen with Highest Strike Rates in Death Overs (16-20)', fontsi
plt.xlabel('Batsman', fontsize=14)
```

```
plt.ylabel('Strike Rate', fontsize=14)
plt.xticks(rotation=45)

# Adding value on top of bars
for index, row in top_15_strike_rates.iterrows():
    plt.text(index, row['strike_rate'] + 0.5, round(row['strike_rate'], 2), colo

plt.tight_layout()
plt.show()
```



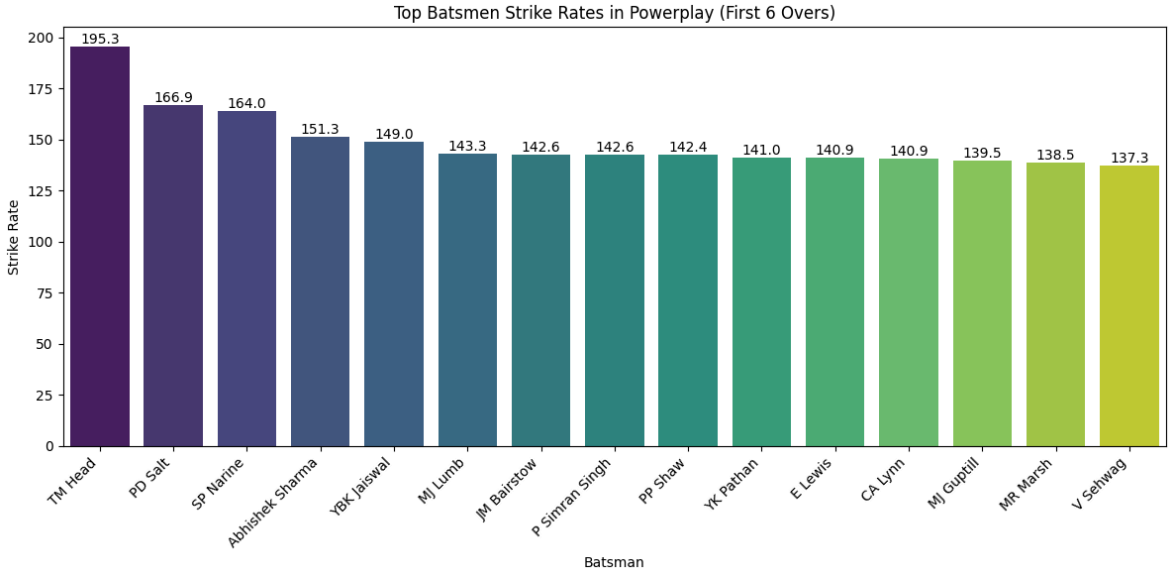
In [ ]:

```
#Considering only who faced at least 150 balls in powerplay over for fare compar
# Filtering for the first 6 overs
powerplay_data = delivery_df[delivery_df['over'] < 6]

# Calculating total runs and balls faced by each batsman
batsman_stats = powerplay_data.groupby('batter').agg(
    total_runs=pd.NamedAgg(column='batsman_runs', aggfunc='sum'),
    balls_faced=pd.NamedAgg(column='ball', aggfunc='count')
).reset_index()

# Calculating strike rate
batsman_stats['strike_rate'] = (batsman_stats['total_runs'] / batsman_stats['bal

# Filtering batsmen who faced at least 150 balls for fare comparison
batsman_stats = batsman_stats[batsman_stats['balls_faced'] >= 150]

# Sorting by strike rate
top_strike_rates_powerplay = batsman_stats.sort_values(by='strike_rate', ascendi

# Display the top batsmen with the highest strike rates in powerplay
top_strike_rates_powerplay=top_strike_rates_powerplay.head(15)  # Adjust the num
top_strike_rates_powerplay
```

Out[ ]:

	batter	total_runs	balls_faced	strike_rate
0	TM Head	412	211	195.260664
1	PD Salt	404	242	166.942149
2	SP Narine	946	577	163.951473
3	Abhishek Sharma	758	501	151.297405
4	YBK Jaiswal	1009	677	149.039882
5	MJ Lumb	225	157	143.312102
6	JM Bairstow	864	606	142.574257
7	P Simran Singh	499	350	142.571429
8	PP Shaw	1347	946	142.389006
9	YK Pathan	361	256	141.015625
10	E Lewis	420	298	140.939597
11	CA Lynn	779	553	140.867993
12	MJ Guptill	219	157	139.490446
13	MR Marsh	223	161	138.509317
14	V Sehwag	1593	1160	137.327586

In [ ]:
```
 #Plotting Graph
plt.figure(figsize=(12, 6))
sns.barplot(data=top_strike_rates_powerplay, x='batter', y='strike_rate', palett
plt.title('Top Batsmen Strike Rates in Powerplay (First 6 Overs)')
plt.xlabel('Batsman')
plt.ylabel('Strike Rate')
plt.xticks(rotation=45, ha='right')

# Adding values on top of bars
for index, value in enumerate(top_strike_rates_powerplay['strike_rate']):
    plt.text(index, value, f'{value:.1f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```



In [ ]:
```
# Make sure 'season' is numeric
match_df['season'] = match_df['season'].astype(int)

# Find the last five seasons
last5 = sorted(match_df['season'].unique())[-5:]

# Restrict matches to those seasons
recent_match_ids = match_df.loc[match_df['season'].isin(last5), 'id']

# Subset deliveries
recent_deliv = delivery_df[delivery_df['match_id'].isin(recent_match_ids)]

# Sum runs per batsman
runs = (
    recent_deliv
    .groupby('batter', as_index=False)['batsman_runs']
    .sum()
    .rename(columns={'batsman_runs':'total_runs'})
)

# Grab top 5
top5_batsmen = runs.nlargest(5, 'total_runs')

print(f"Top 5 run-scorers in seasons {last5}:")
print(top5_batsmen.to_string(index=False))
```

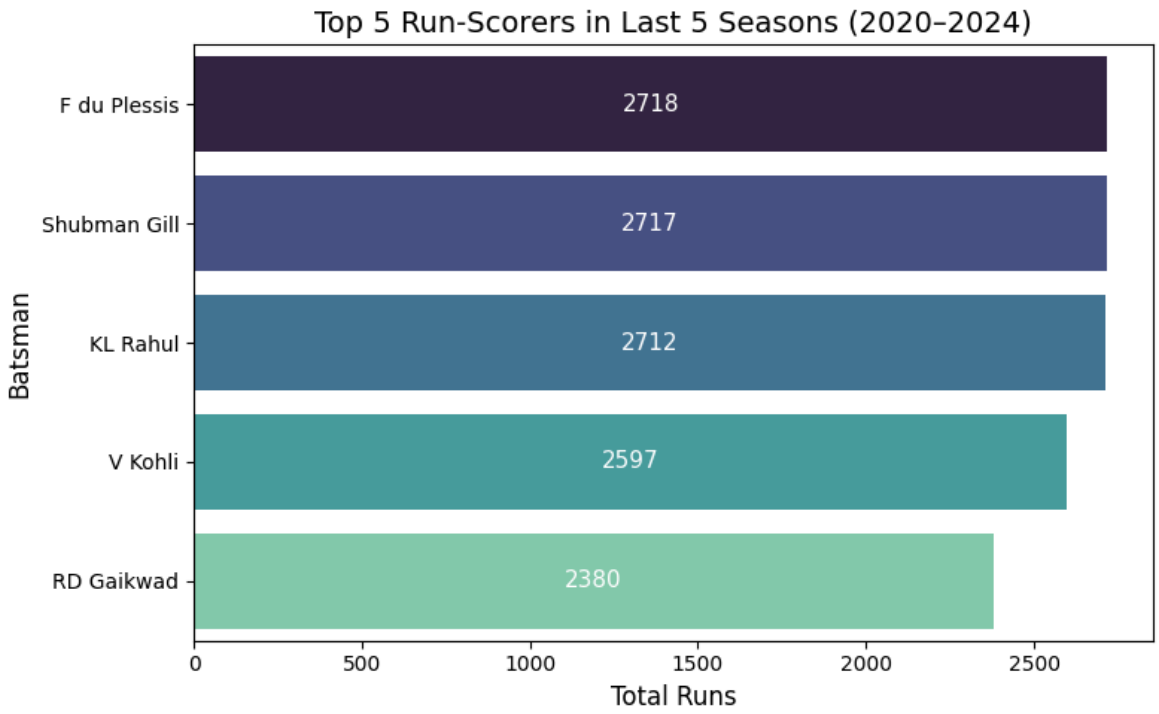
```
Top 5 run-scorers in seasons [np.int64(2020), np.int64(2021), np.int64(2022), np.
int64(2023), np.int64(2024)]:
      batter  total_runs
F du Plessis        2718
Shubman Gill        2717
    KL Rahul        2712
     V Kohli        2597
  RD Gaikwad        2380
```

In [ ]:
```
print(last5)
print(top5_batsmen)
```

```
[np.int64(2020), np.int64(2021), np.int64(2022), np.int64(2023), np.int64(2024)]
          batter  total_runs
77   F du Plessis        2718
279  Shubman Gill        2717
130      KL Rahul        2712
301       V Kohli        2597
224    RD Gaikwad        2380
```

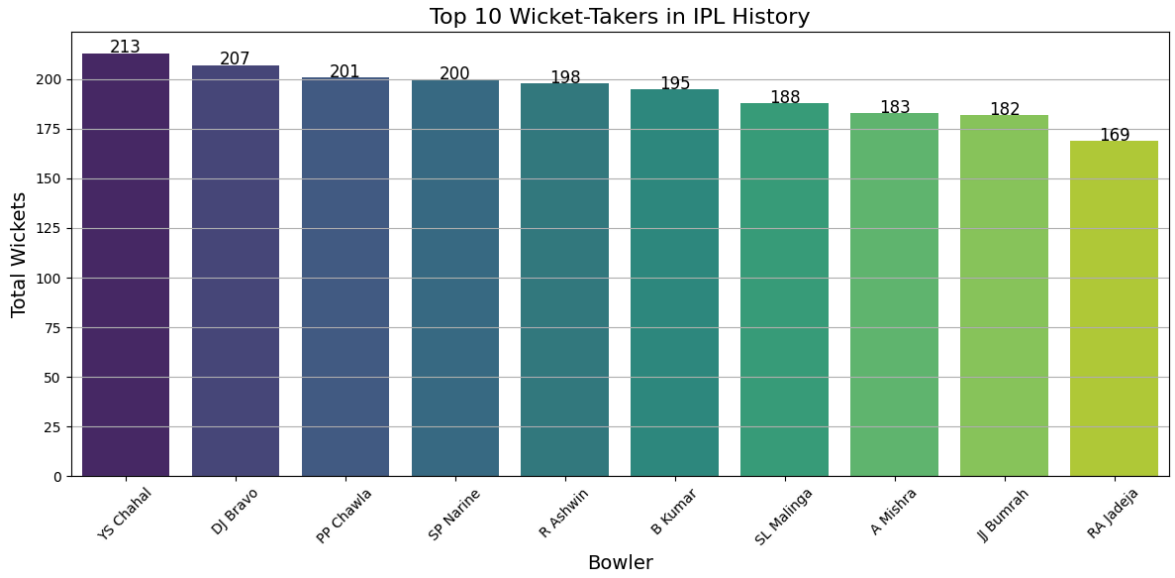
In [ ]:

```
# Reset index so bars are at y=0…4
plot_df = top5_batsmen.reset_index(drop=True)

plt.figure(figsize=(8,5))
ax = sns.barplot(
    data=plot_df,
    x='total_runs',
    y='batter',
    palette='mako'
)

plt.title(f"Top 5 Run-Scorers in Last 5 Seasons ({last5[0]}–{last5[-1]})", fonts
plt.xlabel("Total Runs", fontsize=12)
plt.ylabel("Batsman", fontsize=12)

# Annotate each bar by iterating over the Patch objects
for bar in ax.patches:
    width = bar.get_width()
    y_center = bar.get_y() + bar.get_height() / 2
    ax.text(
        width / 2,
        y_center,
        f"{int(width)}",
        ha='center',
        va='center',
        color='white',
        fontsize=11
    )

plt.tight_layout()
plt.show()
```



Bowler Analysis

In [ ]:
```
most_wicket_taker = delivery_df[['bowler', 'is_wicket']]
most_wicket_taker = most_wicket_taker.groupby('bowler')['is_wicket'].sum().reset
most_wicket_taker = most_wicket_taker.sort_values(by='total_wicket', ascending=F

#top 10 wicket-takers
top_wicket_takers = most_wicket_taker.head(10)

# Plotting the data
plt.figure(figsize=(12, 6))
bar_plot = sns.barplot(data=top_wicket_takers, x='bowler', y='total_wicket', pal

# Annotate the bars with total wickets
for index, row in top_wicket_takers.iterrows():
    bar_plot.text(index, row.total_wicket + 0.1, row.total_wicket, color='black'

# Adding titles and labels
plt.title('Top 10 Wicket-Takers in IPL History', fontsize=16)
plt.xlabel('Bowler', fontsize=14)
plt.ylabel('Total Wickets', fontsize=14)
plt.xticks(rotation=45)
plt.grid(axis='y')

plt.tight_layout()
plt.show()
```



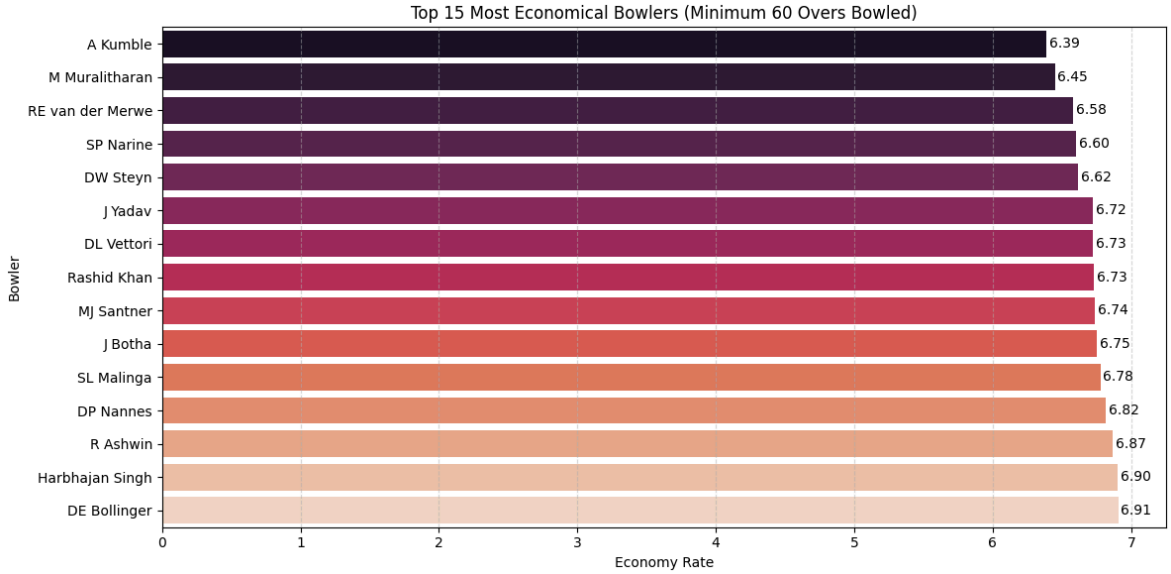
In [ ]:
```
#Top 15 Most Economical Bowlers
#Calculating total runs conceded by each bowler
runs_conceded = delivery_df.groupby('bowler')['batsman_runs'].sum().reset_index(

#Filtering out wides and no balls and calculating total overs bowled by each bow
#Only counting legitimate deliveries - excluding wides and no balls
valid_deliveries = delivery_df[~delivery_df['extras_type'].isin(['wides', 'nobal

# Counting the number of balls bowled for valid deliveries
balls_bowled = valid_deliveries.groupby('bowler')['ball'].count().reset_index(na
balls_bowled['overs_bowled'] = balls_bowled['balls_bowled'] / 6

#Merging runs conceded and balls bowled data
bowler_stats = pd.merge(runs_conceded, balls_bowled[['bowler', 'overs_bowled']],

#Calculating the economy rate
bowler_stats['economy_rate'] = bowler_stats['runs_conceded'] / bowler_stats['ove

#filtring data for those have minimum over bowled 60 for fare comparision
bowler_stats = bowler_stats[bowler_stats['overs_bowled']>=60]
#Sorting the bowlers by economy rate
most_economical_bowlers = bowler_stats.sort_values(by='economy_rate').reset_inde

# Displaying the top bowlers with lowest economy rates
top_15_most_economical_bowlers = most_economical_bowlers.head(15)

# Display the result
top_15_most_economical_bowlers
```

Out[ ]:

	bowler	runs_conceded	overs_bowled	economy_rate
0	A Kumble	1027	160.833333	6.385492
1	M Muralitharan	1642	254.666667	6.447644
2	RE van der Merwe	486	73.833333	6.582393
3	SP Narine	4492	680.166667	6.604264
4	DW Steyn	2406	363.666667	6.615949
5	J Yadav	437	65.000000	6.723077
6	DL Vettori	871	129.500000	6.725869
7	Rashid Khan	3222	478.666667	6.731198
8	MJ Santner	411	61.000000	6.737705
9	J Botha	781	115.666667	6.752161
10	SL Malinga	3194	471.333333	6.776521
11	DP Nannes	734	107.666667	6.817337
12	R Ashwin	5178	754.000000	6.867374
13	Harbhajan Singh	3928	569.333333	6.899297
14	DE Bollinger	663	96.000000	6.906250

In [ ]:

```
# top_15_most_economical_bowler
plt.figure(figsize=(12, 6))
bar_plot = sns.barplot(
    data=top_15_most_economical_bowlers,
    x='economy_rate',
    y='bowler',
    palette='rocket'
)

plt.title('Top 15 Most Economical Bowlers (Minimum 60 Overs Bowled)')
plt.xlabel('Economy Rate')
plt.ylabel('Bowler')
plt.grid(axis='x', linestyle='--', alpha=0.5)

# Annotate bars
for index, row in top_15_most_economical_bowlers.iterrows():
    bar_plot.text(
        row['economy_rate'] + 0.02,
        index,
        f"{row['economy_rate']:.2f}",
        va='center'
    )

plt.tight_layout()
plt.show()
```



In [ ]:
```
most_expensive_bowlers= bowler_stats.sort_values(by='economy_rate', ascending=Fa
most_expensive_bowlers
```

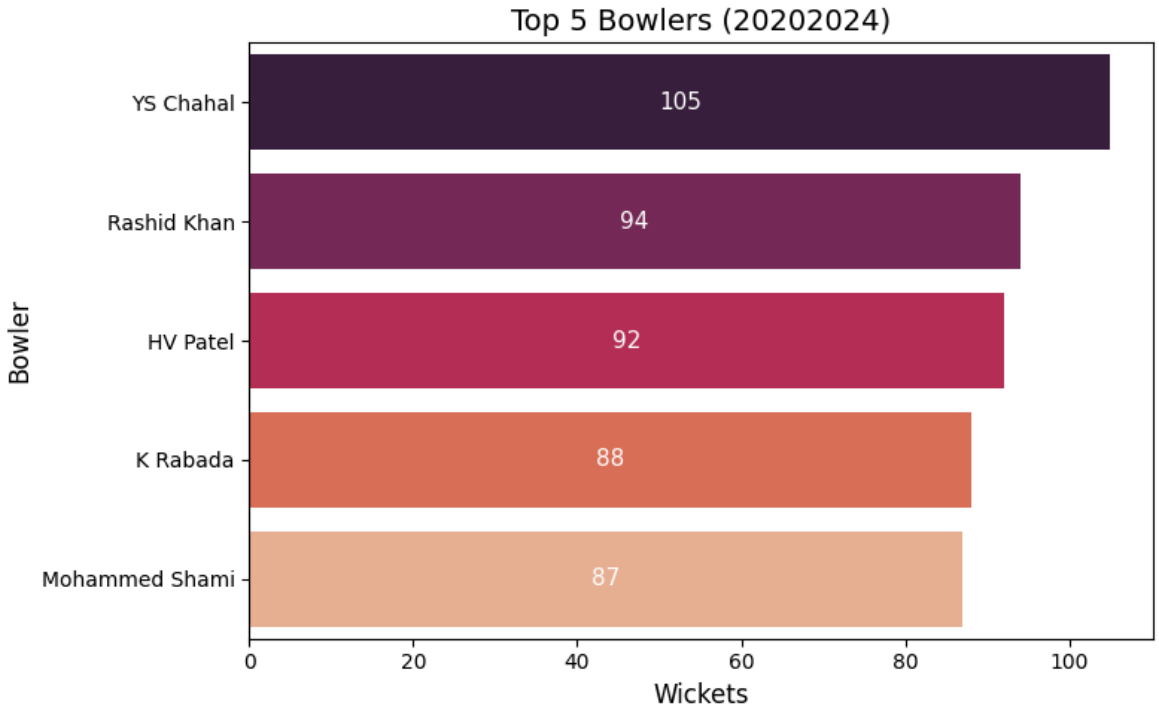
Out[ ]:

	bowler	runs_conceded	overs_bowled	economy_rate
0	Mukesh Kumar	673	66.500000	10.120301
1	Yash Thakur	663	67.333333	9.846535
2	Kartik Tyagi	675	70.333333	9.597156
3	Basil Thampi	828	86.833333	9.535509
4	SM Curran	1847	196.833333	9.383573
5	MP Stoinis	1240	133.833333	9.265255
6	AS Joseph	669	72.333333	9.248848
7	Shahbaz Ahmed	825	89.500000	9.217877
8	CJ Jordan	1033	112.333333	9.195846
9	TU Deshpande	1184	129.000000	9.178295
10	Yash Dayal	878	96.333333	9.114187
11	AD Russell	2561	282.833333	9.054803
12	JR Hopes	539	60.000000	8.983333
13	BB Sran	718	80.500000	8.919255
14	Umran Malik	732	82.333333	8.890688

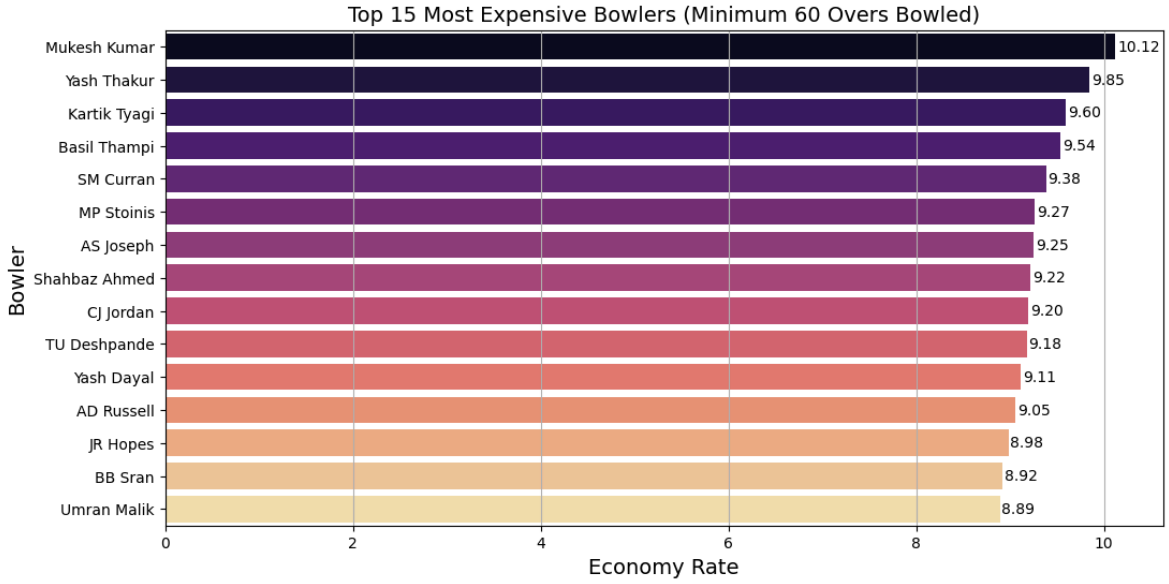
In [ ]:
```
match_df['season'] = match_df['season'].astype(int)

# 2) Find the last five seasons
last5 = sorted(match_df['season'].unique())[-5:]

# 3) Get all match IDs from those seasons
recent_ids = match_df.loc[match_df['season'].isin(last5), 'id']

# 4) Subset your deliveries
```

```
recent_deliv = delivery_df[delivery_df['match_id'].isin(recent_ids)]

# 5) Define what counts as a wicket
dismissal_kinds = [
    'bowled',
    'caught',
    'lbw',
    'stumped',
    'caught and bowled',
    'hit wicket'
]

# 6) Filter for those wicket deliveries and count per bowler
wickets_df = recent_deliv[recent_deliv['dismissal_kind'].isin(dismissal_kinds)]
wickets_count = (
    wickets_df
    .groupby('bowler', as_index=False)
    .size()
    .rename(columns={'size':'wickets'})
)

# 7) Take your Top 5
top5_bowlers = wickets_count.sort_values('wickets', ascending=False).head(5)

print(f" Top 5 bowlers in seasons {last5}:")
print(top5_bowlers.to_string(index=False))


# horizontal bar chart
plot_df = top5_bowlers.reset_index(drop=True)
plt.figure(figsize=(8,5))
ax = sns.barplot(
    data=plot_df,
    x='wickets',
    y='bowler',
    palette='rocket'
)
plt.title(f"Top 5 Bowlers ({last5[0]}{last5[-1]})", fontsize=14)
plt.xlabel("Wickets", fontsize=12)
plt.ylabel("Bowler", fontsize=12)

# draw labels inside bars
for bar in ax.patches:
    w = bar.get_width()
    y = bar.get_y() + bar.get_height()/2
    ax.text(
        w/2, y,
        f"{int(w)}",
        ha='center', va='center',
        color='white', fontsize=11
    )

plt.tight_layout()
plt.show()
```

```
 Top 5 bowlers in seasons [np.int64(2020), np.int64(2021), np.int64(2022), np.int
64(2023), np.int64(2024)]:
        bowler  wickets
     YS Chahal      105
   Rashid Khan       94
      HV Patel       92
      K Rabada       88
Mohammed Shami       87
```



In [ ]:
```
plt.figure(figsize=(12, 6))
bar_plot = sns.barplot(data=most_expensive_bowlers, x='economy_rate', y='bowler'
plt.title('Top 15 Most Expensive Bowlers (Minimum 60 Overs Bowled)',fontsize=14)
plt.xlabel('Economy Rate',fontsize=14)
plt.ylabel('Bowler', fontsize=14)
plt.grid(axis='x')

# Annotating bars with their respective values
for index, row in most_expensive_bowlers.iterrows():
    bar_plot.text(row['economy_rate'] + 0.02, index, f"{row['economy_rate']:.2f}

plt.show()
```



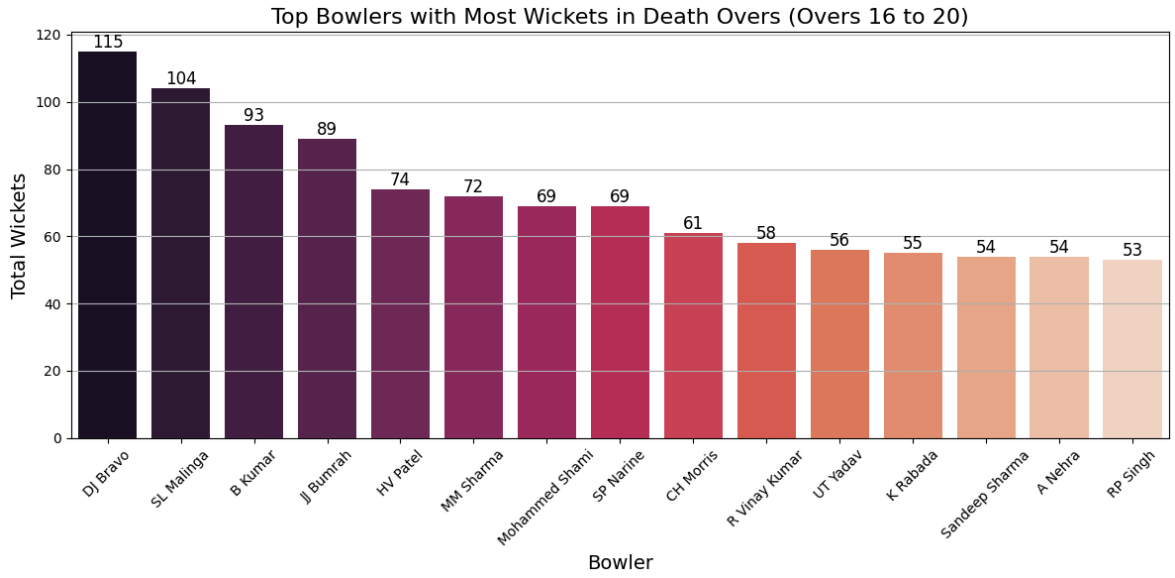
In [ ]:
```
# Filtering the data for death overs (over 16 to 20)
death_overs = delivery_df[(delivery_df['over'] >= 16) & (delivery_df['over'] <=

# Grouping by bowler and summing the wickets taken
wickets_in_death_overs = death_overs.groupby('bowler')['is_wicket'].sum().reset_

# Sorting the bowlers by total wickets in descending order
most_wickets_death_overs = wickets_in_death_overs.sort_values(by='total_wickets'

# Getting the top 15 bowlers with most wickets in death overs
top_bowlers_death_overs = most_wickets_death_overs.head(15).reset_index(drop=Tru

# Displayong the result
top_bowlers_death_overs
```

Out[ ]:

	bowler	total_wickets
0	DJ Bravo	115
1	SL Malinga	104
2	B Kumar	93
3	JJ Bumrah	89
4	HV Patel	74
5	MM Sharma	72
6	Mohammed Shami	69
7	SP Narine	69
8	CH Morris	61
9	R Vinay Kumar	58
10	UT Yadav	56
11	K Rabada	55
12	Sandeep Sharma	54
13	A Nehra	54
14	RP Singh	53

In [ ]:

```
# Plotting the top 15 bowlers who took most wickets in death overs
plt.figure(figsize=(12, 6))
ax = sns.barplot(data=top_bowlers_death_overs, x='bowler', y='total_wickets', pa

# Adding titles and labels
plt.title('Top Bowlers with Most Wickets in Death Overs (Overs 16 to 20)', fonts
plt.xlabel('Bowler', fontsize=14)
plt.ylabel('Total Wickets', fontsize=14)
plt.xticks(rotation=45)
plt.grid(axis='y')

# Adding value on top of each bar
for p in ax.patches:
    ax.annotate(f'{int(p.get_height())}',
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center', va='bottom',
                fontsize=12)

plt.tight_layout()
plt.show()
```

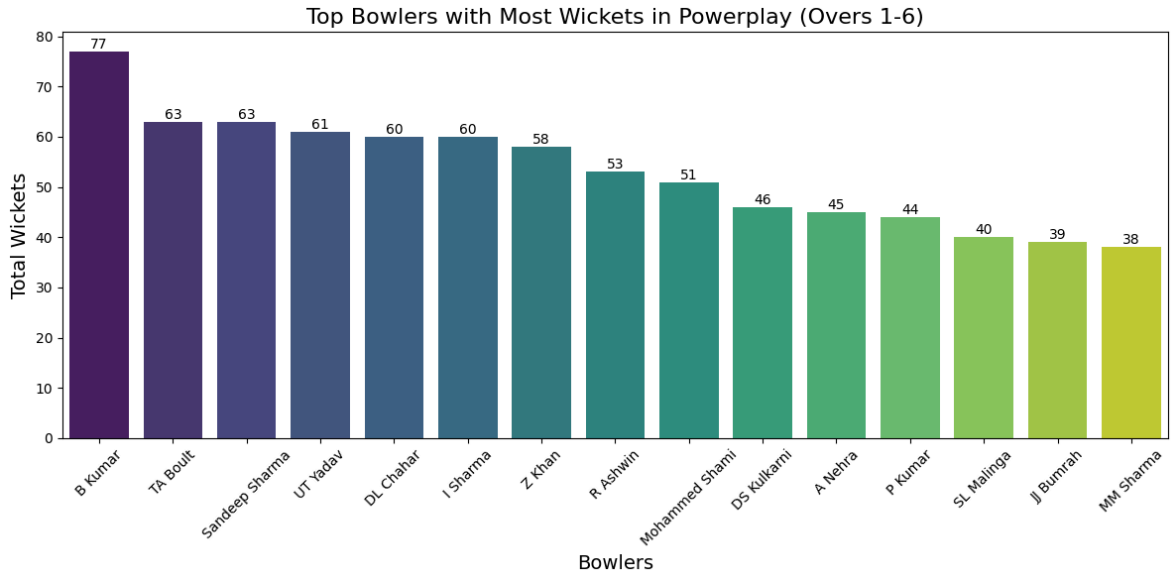


In [ ]:
```
# Filtering data for powerplay overs (overs 0 to 5)
powerplay_data = delivery_df[(delivery_df['over'] >= 0) & (delivery_df['over'] <

# Counting wickets taken by each bowler in the powerplay
wickets_powerplay = powerplay_data[powerplay_data['is_wicket'] == 1].groupby('bo

# Sorting the bowlers by total wickets taken
top_bowlers_powerplay = wickets_powerplay.sort_values(by='total_wickets', ascend

# Displaying the top 15 bowlers with the most wickets in powerplay
top_bowlers_powerplay = top_bowlers_powerplay.head(15)
top_bowlers_powerplay
```

Out[ ]:

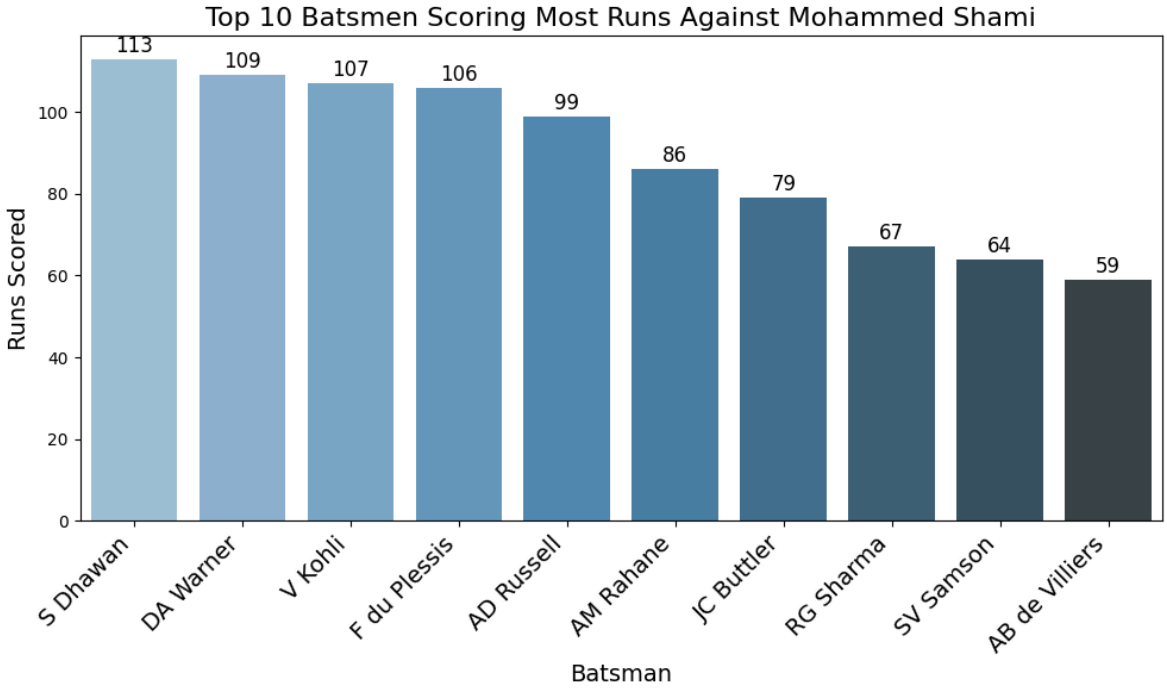
	bowler	total_wickets
0	B Kumar	77
1	TA Boult	63
2	Sandeep Sharma	63
3	UT Yadav	61
4	DL Chahar	60
5	I Sharma	60
6	Z Khan	58
7	R Ashwin	53
8	Mohammed Shami	51
9	DS Kulkarni	46
10	A Nehra	45
11	P Kumar	44
12	SL Malinga	40
13	JJ Bumrah	39
14	MM Sharma	38

In [ ]:
```
# Plotting the bar graph
plt.figure(figsize=(12, 6))
sns.barplot(x='bowler', y='total_wickets', data=top_bowlers_powerplay, palette='
plt.title('Top Bowlers with Most Wickets in Powerplay (Overs 1-6)', fontsize=16)
plt.xlabel('Bowlers', fontsize=14)
plt.ylabel('Total Wickets', fontsize=14)
plt.xticks(rotation=45)

# Adding the values on top of the bars
for index, value in enumerate(top_bowlers_powerplay['total_wickets']):
    plt.text(index, value, str(value), ha='center', va='bottom')

plt.tight_layout()
plt.show()
```

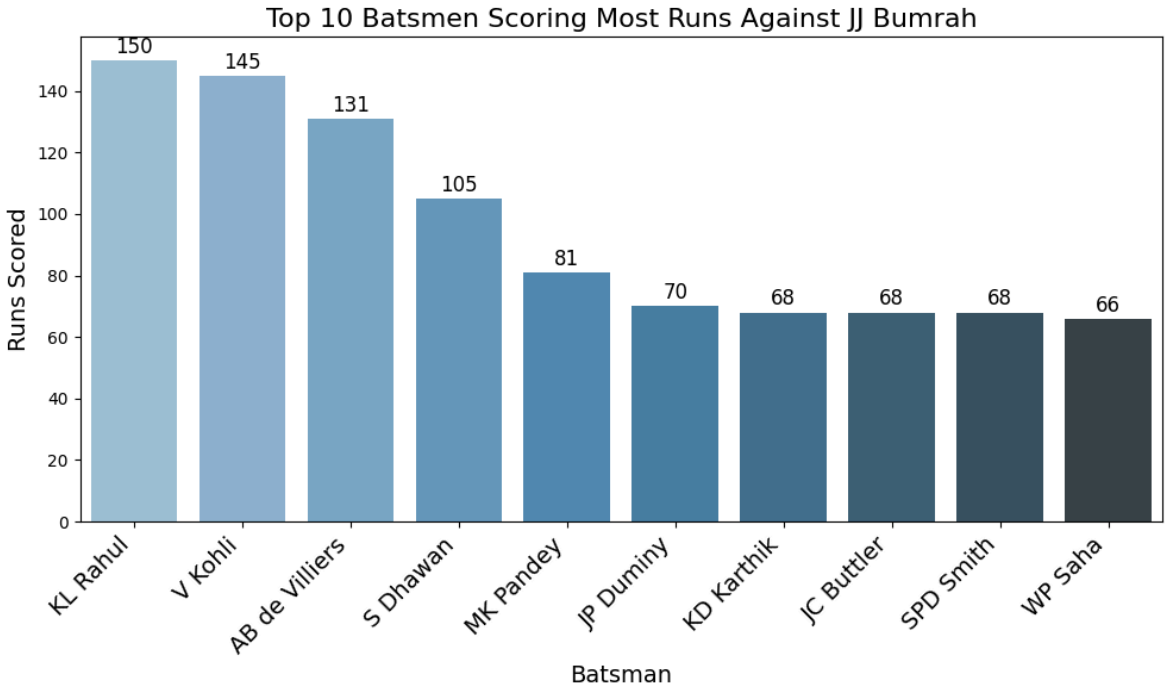


In [ ]:
```
# Creating Function to check most run scorer batsman against Bowler
def top_batsmen_against_bowler(bowler_name):
    # Filter the DataFrame for deliveries bowled by the specific bowler
    deliveries_by_bowler = delivery_df[delivery_df['bowler'] == bowler_name]

    # Grouping by batsman and suming the total runs scored against the bowler
    batsman_runs = deliveries_by_bowler.groupby('batter')['batsman_runs'].sum().

    # Sorting by runs scored and get the top 10 batsmen
    top_10_batsmen = batsman_runs.sort_values(by='runs_scored', ascending=False)

    # Plotting using seaborn
    plt.figure(figsize=(10, 6))
    ax = sns.barplot(x='batter', y='runs_scored', data=top_10_batsmen, palette='

    # Adding values on top of bars
    for bar in ax.patches:
        ax.annotate(format(bar.get_height(), '.0f'),
                    (bar.get_x() + bar.get_width() / 2, bar.get_height()),
                    ha='center', va='center', size=12, xytext=(0, 8),
                    textcoords='offset points')

    # Customizing plot aesthetics
    plt.title(f'Top 10 Batsmen Scoring Most Runs Against {bowler_name}', fontsiz
    plt.xlabel('Batsman', fontsize=14)
    plt.ylabel('Runs Scored', fontsize=14)
    plt.xticks(rotation=45, ha='right', fontsize=14)  # Rotate x-axis labels ana
    plt.tight_layout()
    plt.show()
```

In [ ]:
```
# Checking Most run scorer batsman against Mohammed Shami
top_batsmen_against_bowler('Mohammed Shami')
```



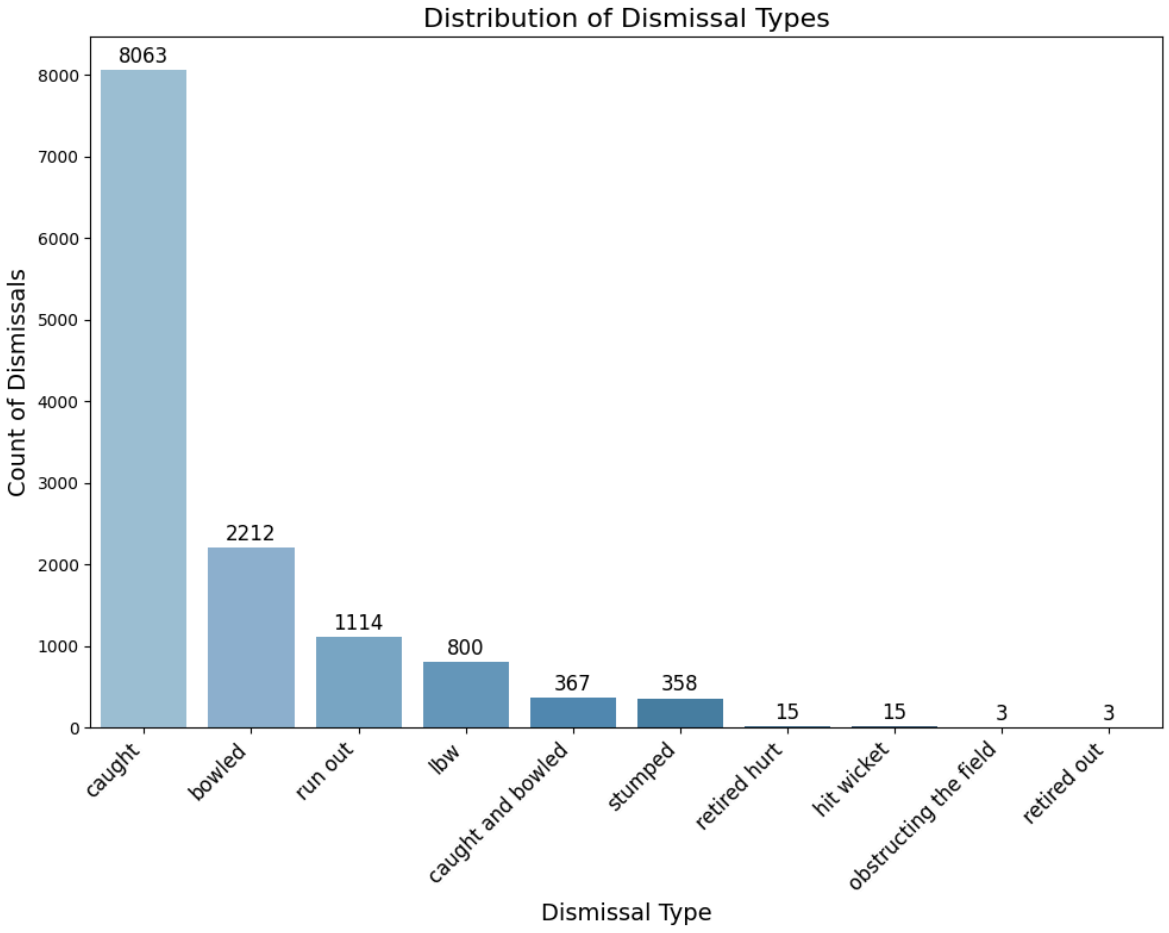
In [ ]:
```
# Checking Most run scorer batsman against Jasprit Bumrah
top_batsmen_against_bowler('JJ Bumrah')
```



Fielding Anlysis

In [ ]:
```
# Filtering the data for deliveries where a wicket was taken
dismissals = delivery_df[delivery_df['is_wicket'] == 1]

# Grouping by dismissal type and count the number of each type
dismissal_counts = dismissals.groupby('dismissal_kind')['is_wicket'].count().res

# Sorting the dismissal types by their count in descending order
dismissal_counts = dismissal_counts.sort_values(by='count', ascending=False)
#dismissal_counts
```

In [ ]:
```
# Plotting the distribution of dismissal types
plt.figure(figsize=(10, 8))
ax = sns.barplot(x='dismissal_kind', y='count', data=dismissal_counts, palette='

# Adding values on top of bars
for bar in ax.patches:
    ax.annotate(format(bar.get_height(), '.0f'),
                (bar.get_x() + bar.get_width() / 2, bar.get_height()),
                ha='center', va='center', size=12, xytext=(0, 8),
                textcoords='offset points')


plt.title('Distribution of Dismissal Types', fontsize=16)
plt.xlabel('Dismissal Type', fontsize=14)
plt.ylabel('Count of Dismissals', fontsize=14)
plt.xticks(rotation=45, ha='right', fontsize=12)  # Rotate x-axis labels
plt.tight_layout()
plt.show()
```



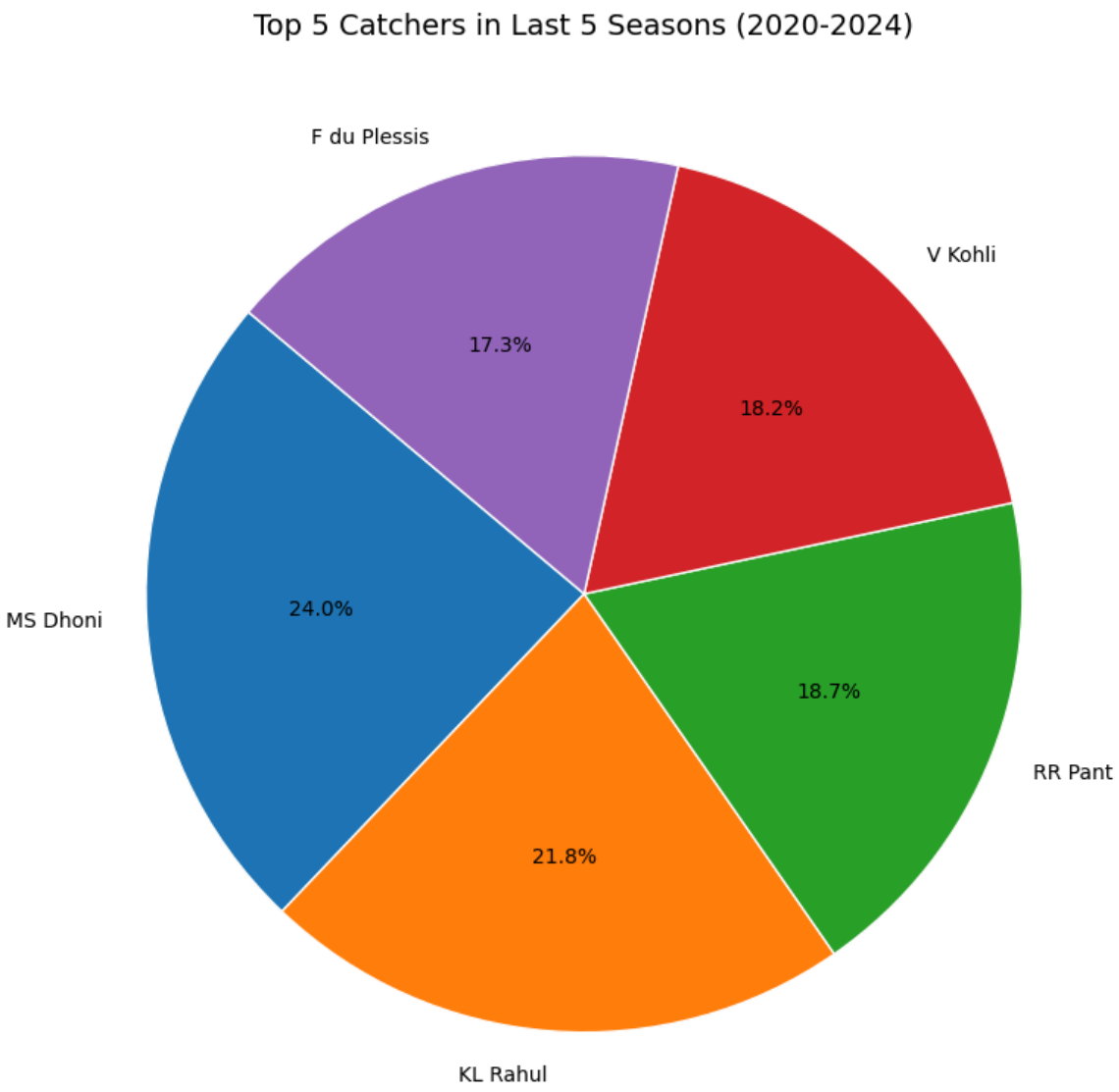
In [ ]:
```
# Ensure 'season' is numeric
match_df['season'] = match_df['season'].astype(int)

# Identify the last five seasons
last5 = sorted(match_df['season'].unique())[-5:]

# Filter for those seasons' match IDs
recent_ids = match_df.loc[match_df['season'].isin(last5), 'id']

# Subset your deliveries
recent_deliv = delivery_df[delivery_df['match_id'].isin(recent_ids)]
```

```
#  Keep only catch dismissals
catch_kinds = ['caught', 'caught and bowled']
catches = recent_deliv[recent_deliv['dismissal_kind'].isin(catch_kinds)]

# Count catches per fielder
catch_counts = (
    catches
    .groupby('fielder', as_index=False)
    .size()
    .rename(columns={'size':'catches'})
)

#  Get the top catcher
top_catcher = catch_counts.sort_values('catches', ascending=False).head(5)

print(f" Top catcher in seasons {last5}:\n", top_catcher.to_string(index=False))
```

```
 Top catcher in seasons [np.int64(2020), np.int64(2021), np.int64(2022), np.int64
(2023), np.int64(2024)]:
      fielder  catches
    MS Dhoni       54
    KL Rahul       49
     RR Pant       42
     V Kohli       41
F du Plessis       39
```

In [ ]:

```
# Plot a pie chart
plt.figure(figsize=(8, 8))
plt.pie(
    top_catcher['catches'],
    labels=top_catcher['fielder'],
    autopct='%1.1f%%',
    startangle=140,
    wedgeprops={'edgecolor': 'white'}
)
plt.title(f'Top 5 Catchers in Last 5 Seasons ({last5[0]}-{last5[-1]})', fontsize
plt.tight_layout()
plt.show()
```



Model Builidng & Feature Engineering

In [ ]:
```
delivery_df.head()
```

Out[ ]:

	match_id	inning	batting_team	bowling_team	over	ball	batter	bowler	non_s
0	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	1	SC Ganguly	P Kumar	McC
1	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	2	BB McCullum	P Kumar	SC Ga
2	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	3	BB McCullum	P Kumar	SC Ga
3	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	4	BB McCullum	P Kumar	SC Ga
4	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	5	BB McCullum	P Kumar	SC Ga

In [ ]:
```
match_df.head()
```

Out[ ]:

	id	season	city	date	match_type	player_of_match	venue	
0	335982	2008	Bangalore	2008-04-18	League	BB McCullum	M Chinnaswamy Stadium	Chall Bar
1	335983	2008	Chandigarh	2008-04-19	League	MEK Hussey	Punjab Cricket Association Stadium, Mohali	K
2	335984	2008	Delhi	2008-04-19	League	MF Maharoof	Feroz Shah Kotla	C
3	335985	2008	Mumbai	2008-04-20	League	MV Boucher	Wankhede Stadium	M
4	335986	2008	Kolkata	2008-04-20	League	DJ Hussey	Eden Gardens	

In [ ]:
```
#  Merge both dataframe
final_df = delivery_df.merge(
    match_df,
    left_on='match_id',
    right_on='id',
    how='left',
    suffixes=('_delivery', '_match')
)

#  Drop the now-redundant 'id' column, keeping only 'match_id'
```

```
final_df = final_df.drop(columns=['id'])


print(final_df.shape)
print(final_df.columns)
print(final_df.head())
```

```
(260920, 36)
Index(['match_id', 'inning', 'batting_team', 'bowling_team', 'over', 'ball',
       'batter', 'bowler', 'non_striker', 'batsman_runs', 'extra_runs',
       'total_runs', 'extras_type', 'is_wicket', 'player_dismissed',
       'dismissal_kind', 'fielder', 'season', 'city', 'date', 'match_type',
       'player_of_match', 'venue', 'team1', 'team2', 'toss_winner',
       'toss_decision', 'winner', 'result', 'result_margin', 'target_runs',
       'target_overs', 'super_over', 'method', 'umpire1', 'umpire2'],
      dtype='object')
   match_id  inning           batting_team                bowling_team  over  \
0    335982       1  Kolkata Knight Riders  Royal Challengers Bangalore     0
1    335982       1  Kolkata Knight Riders  Royal Challengers Bangalore     0
2    335982       1  Kolkata Knight Riders  Royal Challengers Bangalore     0
3    335982       1  Kolkata Knight Riders  Royal Challengers Bangalore     0
4    335982       1  Kolkata Knight Riders  Royal Challengers Bangalore     0

   ball       batter   bowler  non_striker  batsman_runs  ...  toss_decision  \
0     1   SC Ganguly  P Kumar  BB McCullum             0  ...          field
1     2  BB McCullum  P Kumar   SC Ganguly             0  ...          field
2     3  BB McCullum  P Kumar   SC Ganguly             0  ...          field
3     4  BB McCullum  P Kumar   SC Ganguly             0  ...          field
4     5  BB McCullum  P Kumar   SC Ganguly             0  ...          field

                winner result  result_margin target_runs target_overs  \
0  Kolkata Knight Riders   runs          140.0       223.0         20.0
1  Kolkata Knight Riders   runs          140.0       223.0         20.0
2  Kolkata Knight Riders   runs          140.0       223.0         20.0
3  Kolkata Knight Riders   runs          140.0       223.0         20.0
4  Kolkata Knight Riders   runs          140.0       223.0         20.0

  super_over  method    umpire1      umpire2
0          N     NaN  Asad Rauf  RE Koertzen
1          N     NaN  Asad Rauf  RE Koertzen
2          N     NaN  Asad Rauf  RE Koertzen
3          N     NaN  Asad Rauf  RE Koertzen
4          N     NaN  Asad Rauf  RE Koertzen

[5 rows x 36 columns]
```

In [ ]:
```
final_df.head()
```

Out[ ]:

	match_id	inning	batting_team	bowling_team	over	ball	batter	bowler	non_s
0	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	1	SC Ganguly	P Kumar	McC
1	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	2	BB McCullum	P Kumar	SC Ga
2	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	3	BB McCullum	P Kumar	SC Ga
3	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	4	BB McCullum	P Kumar	SC Ga
4	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	5	BB McCullum	P Kumar	SC Ga

5 rows × 36 columns

In [ ]:
```
final_df
```

Out[ ]:

	match_id	inning	batting_team	bowling_team	over	ball	batter	bowler
0	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	1	SC Ganguly	P Kumar
1	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	2	BB McCullum	P Kumar
2	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	3	BB McCullum	P Kumar
3	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	4	BB McCullum	P Kumar
4	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	5	BB McCullum	P Kumar
...	...	...	...	...	...	...	...	...
260915	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	9	5	SS Iyer	AK Markram
260916	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	9	6	VR Iyer	AK Markram
260917	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	10	1	VR Iyer	Shahbaz Ahmed
260918	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	10	2	SS Iyer	Shahbaz Ahmed
260919	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	10	3	VR Iyer	Shahbaz Ahmed

260920 rows × 36 columns

In [ ]:
```
final_df.isnull().sum()
```

Out[ ]:

	0
match_id	0
inning	0
batting_team	0
bowling_team	0
over	0
ball	0
batter	0
bowler	0
non_striker	0
batsman_runs	0
extra_runs	0
total_runs	0
extras_type	246795
is_wicket	0
player_dismissed	247970
dismissal_kind	247970
fielder	251566
season	0
city	0
date	0
match_type	0
player_of_match	490
venue	0
team1	0
team2	0
toss_winner	0
toss_decision	0
winner	490
result	0
result_margin	4124
target_runs	309
target_overs	309
super_over	0

	0
method	257274
umpire1	0
umpire2	0

dtype: int64

In [ ]:
```
cols_to_fill = ['extras_type', 'player_dismissed', 'dismissal_kind', 'fielder','
for col in cols_to_fill:
    final_df[col] = final_df[col].fillna(0)


print(final_df[cols_to_fill].isnull().sum())
```

```
extras_type         0
player_dismissed    0
dismissal_kind      0
fielder             0
method              0
dtype: int64
```

In [ ]:
```
# List of columns to check
cols_to_drop = ['winner', 'result_margin', 'target_runs', 'target_overs','player

# Drop any row with NaN in these columns
final_df = final_df.dropna(subset=cols_to_drop)

# Verify no more NaNs in those columns
print(final_df[cols_to_drop].isnull().sum())
print("New shape:", final_df.shape)
```

```
winner             0
result_margin      0
target_runs        0
target_overs       0
player_of_match    0
dtype: int64
New shape: (256796, 36)
```

In [ ]:
```
final_df.isnull().sum()
```

Out[ ]:

	0
match_id	0
inning	0
batting_team	0
bowling_team	0
over	0
ball	0
batter	0
bowler	0
non_striker	0
batsman_runs	0
extra_runs	0
total_runs	0
extras_type	0
is_wicket	0
player_dismissed	0
dismissal_kind	0
fielder	0
season	0
city	0
date	0
match_type	0
player_of_match	0
venue	0
team1	0
team2	0
toss_winner	0
toss_decision	0
winner	0
result	0
result_margin	0
target_runs	0
target_overs	0
super_over	0

	0
method	0
umpire1	0
umpire2	0

dtype: int64

In [ ]:
```
final_df
```

Out[ ]:

	match_id	inning	batting_team	bowling_team	over	ball	batter	bowler
0	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	1	SC Ganguly	P Kumar
1	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	2	BB McCullum	P Kumar
2	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	3	BB McCullum	P Kumar
3	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	4	BB McCullum	P Kumar
4	335982	1	Kolkata Knight Riders	Royal Challengers Bangalore	0	5	BB McCullum	P Kumar
...	...	...	...	...	...	...	...	...
260915	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	9	5	SS Iyer	AK Markram
260916	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	9	6	VR Iyer	AK Markram
260917	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	10	1	VR Iyer	Shahbaz Ahmed
260918	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	10	2	SS Iyer	Shahbaz Ahmed
260919	1426312	2	Kolkata Knight Riders	Sunrisers Hyderabad	10	3	VR Iyer	Shahbaz Ahmed

256796 rows × 36 columns

In [ ]:
```
import pandas as pd

final_feats = ['bat_avg','bowl_avg','top5_runs_sum','top5_wkts_sum']  # <- edit
desc_map = {
    'bat_avg': 'Team batting average in the season (runs / dismissals).',
    'bowl_avg': 'Team bowling average in the season (runs conceded / wickets).',
    'top5_runs_sum': 'Sum of runs by the team’s top five run-scorers (season).',
    'top5_wkts_sum': 'Sum of wickets by the team’s top five wicket-takers (seaso
}

table_41 = pd.DataFrame({
    'Feature': final_feats,
    'Brief Description': [desc_map.get(f, '— add brief description —') for f in
})

print(" Final feature list after RFE\n")
print(table_41.to_string(index=False))
```

```
Final feature list after RFE

      Feature                                              Brief Description
     bat_avg       Team batting average in the season (runs / dismissals).
    bowl_avg Team bowling average in the season (runs conceded / wickets).
top5_runs_sum      Sum of runs by the team’s top five run-scorers (season).
top5_wkts_sum Sum of wickets by the team’s top five wicket-takers (season).
```

XG Boost Regression

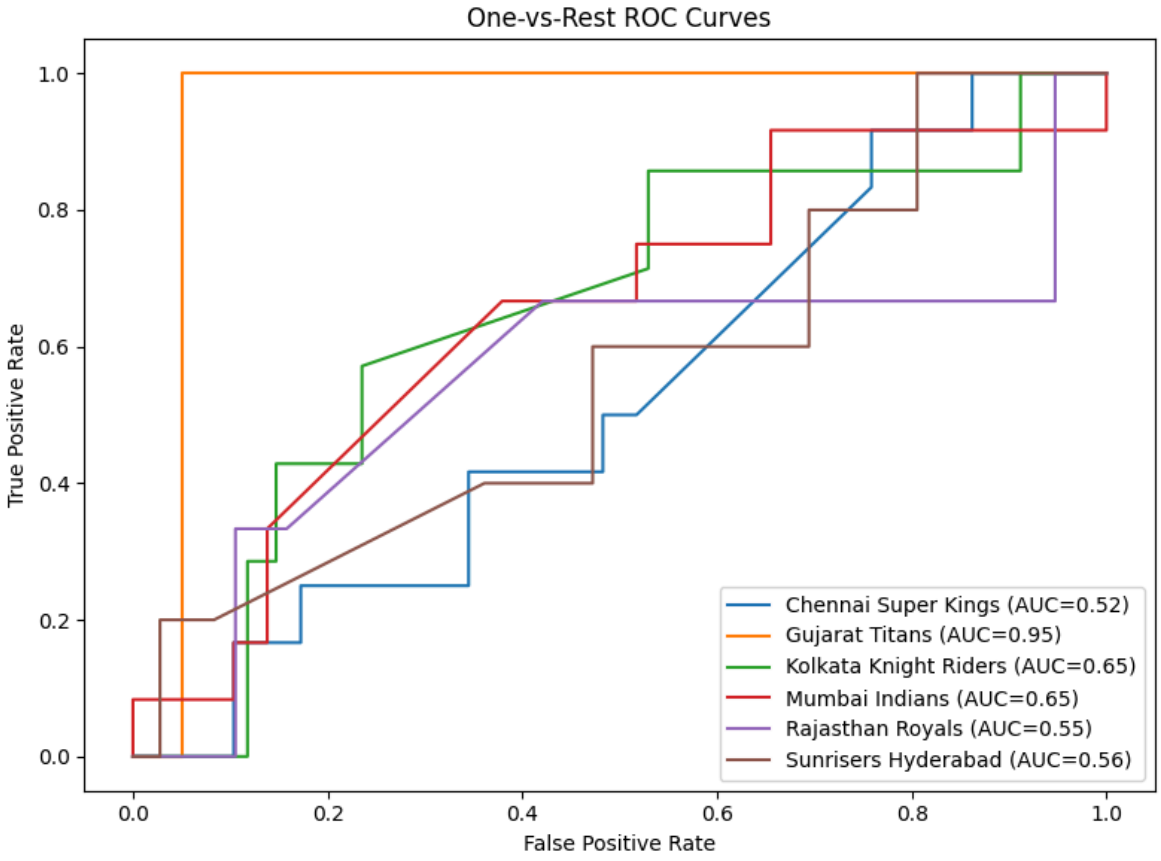
In [ ]:
```
import pandas as pd
import numpy as np
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_sc
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import confusion_matrix, accuracy_score, classification_rep
import matplotlib.pyplot as plt

from sklearn.metrics import (
    confusion_matrix,
    accuracy_score,
    classification_report,
    roc_auc_score,
    roc_curve
)



# Rename the match-side season to a unified 'season'
# (yours may be season_mat or season_match)
if 'season_mat' in final_df.columns:
    final_df = final_df.rename(columns={'season_mat':'season'})
elif 'season_match' in final_df.columns:
    final_df = final_df.rename(columns={'season_match':'season'})
else:
    # if your delivery side had the right one:
    final_df = final_df.rename(columns={'season_del':'season'})

# Extract one row per match to identify each season’s final
matches_meta = (
    final_df[['match_id','season','date','winner']]
```

```
    .drop_duplicates(subset='match_id')
)
matches_meta['date'] = pd.to_datetime(matches_meta['date'])
finals = (
    matches_meta
    .sort_values(['season','date'], ascending=[True,True])
    .groupby('season', sort=False)
    .tail(1)[['season','winner']]
    .rename(columns={'winner':'champion'})
)

# Feature engineering per team-season
dismissal_kinds = ['bowled','caught','lbw','stumped','caught and bowled','hit wi
teams = final_df['batting_team'].unique()

records = []
for season in sorted(final_df['season'].unique()):
    df_s = final_df[final_df['season']==season]
    for team in teams:
        bat = df_s[df_s['batting_team']==team]
        total_runs = bat['batsman_runs'].sum()
        dismissals = bat['player_dismissed'].notna().sum() or 1
        bat_avg = total_runs / dismissals

        bowl = df_s[df_s['bowling_team']==team]
        conceded = bowl['total_runs'].sum()
        wkts = bowl['dismissal_kind'].isin(dismissal_kinds).sum() or 1
        bowl_avg = conceded / wkts

        top5_runs = bat.groupby('batter')['batsman_runs'].sum().nlargest(5).sum(
        top5_wkts = (
            bowl[bowl['dismissal_kind'].isin(dismissal_kinds)]
            .groupby('bowler').size().nlargest(5).sum()
        )

        records.append({
            'season': season,
            'team': team,
            'bat_avg': bat_avg,
            'bowl_avg': bowl_avg,
            'top5_runs_sum': top5_runs,
            'top5_wkts_sum': top5_wkts
        })

features_df = pd.DataFrame(records)

# Merge features with the champion label
data = features_df.merge(finals, on='season')

# Prepare feature matrix X and target y
X = data[['bat_avg','bowl_avg','top5_runs_sum','top5_wkts_sum']]
y = data['champion']

# Encode target and standardize features
le    = LabelEncoder(); y_enc = le.fit_transform(y)
scaler = StandardScaler(); X_std  = scaler.fit_transform(X)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X_std, y_enc, test_size=0.2, random_state=42, stratify=y_enc
```

```
)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'learning_rate': [0.2, 0.6],
    'subsample': [0.3, 0.6, 0.9]
}
xgb = XGBClassifier(
    max_depth=2,
    n_estimators=200,
    objective='multi:softprob',
    use_label_encoder=False,
    eval_metric='mlogloss'
)
grid = GridSearchCV(
    estimator=xgb,
    param_grid=param_grid,
    scoring='accuracy',
    cv=3,
    verbose=1,
    return_train_score=True
)
grid.fit(X_train, y_train)
print("XG Best params:", grid.best_params_)

# 11) Fit final XGB with best params
best = grid.best_params_
model = XGBClassifier(
    **best,
    max_depth=2,
    n_estimators=200,
    objective='multi:softprob',
    use_label_encoder=False,
    eval_metric='mlogloss'
)
model.fit(X_train, y_train)

#  Evaluate on train set
y_tr_pred = model.predict(X_train)
print("Train Accuracy:", accuracy_score(y_train, y_tr_pred))
print("Train Confusion Matrix:\n", confusion_matrix(y_train, y_tr_pred))
print("Train Classification Report:\n", classification_report(y_train, y_tr_pred
y_tr_proba = model.predict_proba(X_train)
print("Train ROC AUC (ovr):", roc_auc_score(y_train, y_tr_proba, multi_class='ov

# Evaluate on test set
y_te_pred = model.predict(X_test)
print("Test Accuracy:", accuracy_score(y_test, y_te_pred))
print("Test Confusion Matrix:\n", confusion_matrix(y_test, y_te_pred))
print("Test Classification Report:\n", classification_report(y_test, y_te_pred))
y_te_proba = model.predict_proba(X_test)
print("Test ROC AUC (ovr):", roc_auc_score(y_test, y_te_proba, multi_class='ovr'

# Plot ROC curves per class
plt.figure(figsize=(8,6))
for i, cls in enumerate(le.classes_):
    fpr, tpr, _ = roc_curve(y_test==i, y_te_proba[:,i])
    plt.plot(fpr, tpr, label=f'{cls} (AUC={roc_auc_score(y_test==i, y_te_proba[:
plt.title('One-vs-Rest ROC Curves')
plt.xlabel('False Positive Rate')
```

```
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
XG Best params: {'learning_rate': 0.2, 'subsample': 0.3}
Train Accuracy: 0.7975460122699386
Train Confusion Matrix:
 [[36  0  0 12  0  0]
 [ 0  8  0  2  0  0]
 [ 0  0 21  8  0  0]
 [ 0  0  0 48  0  0]
 [ 0  0  0  3  6  0]
 [ 1  0  0  7  0 11]]
Train Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.75      0.85        48
           1       1.00      0.80      0.89        10
           2       1.00      0.72      0.84        29
           3       0.60      1.00      0.75        48
           4       1.00      0.67      0.80         9
           5       1.00      0.58      0.73        19

    accuracy                           0.80       163
   macro avg       0.93      0.75      0.81       163
weighted avg       0.87      0.80      0.80       163

Train ROC AUC (ovr): 0.9399082596000193
Test Accuracy: 0.3170731707317073
Test Confusion Matrix:
 [[3 0 1 8 0 0]
 [2 0 0 0 0 0]
 [3 0 1 3 0 0]
 [3 0 0 8 0 1]
 [1 1 0 1 0 0]
 [0 0 2 2 0 1]]
Test Classification Report:
              precision    recall  f1-score   support

           0       0.25      0.25      0.25        12
           1       0.00      0.00      0.00         2
           2       0.25      0.14      0.18         7
           3       0.36      0.67      0.47        12
           4       0.00      0.00      0.00         3
           5       0.50      0.20      0.29         5

    accuracy                           0.32        41
   macro avg       0.23      0.21      0.20        41
weighted avg       0.28      0.32      0.28        41

Test ROC AUC (ovr): 0.6462843028846018
```



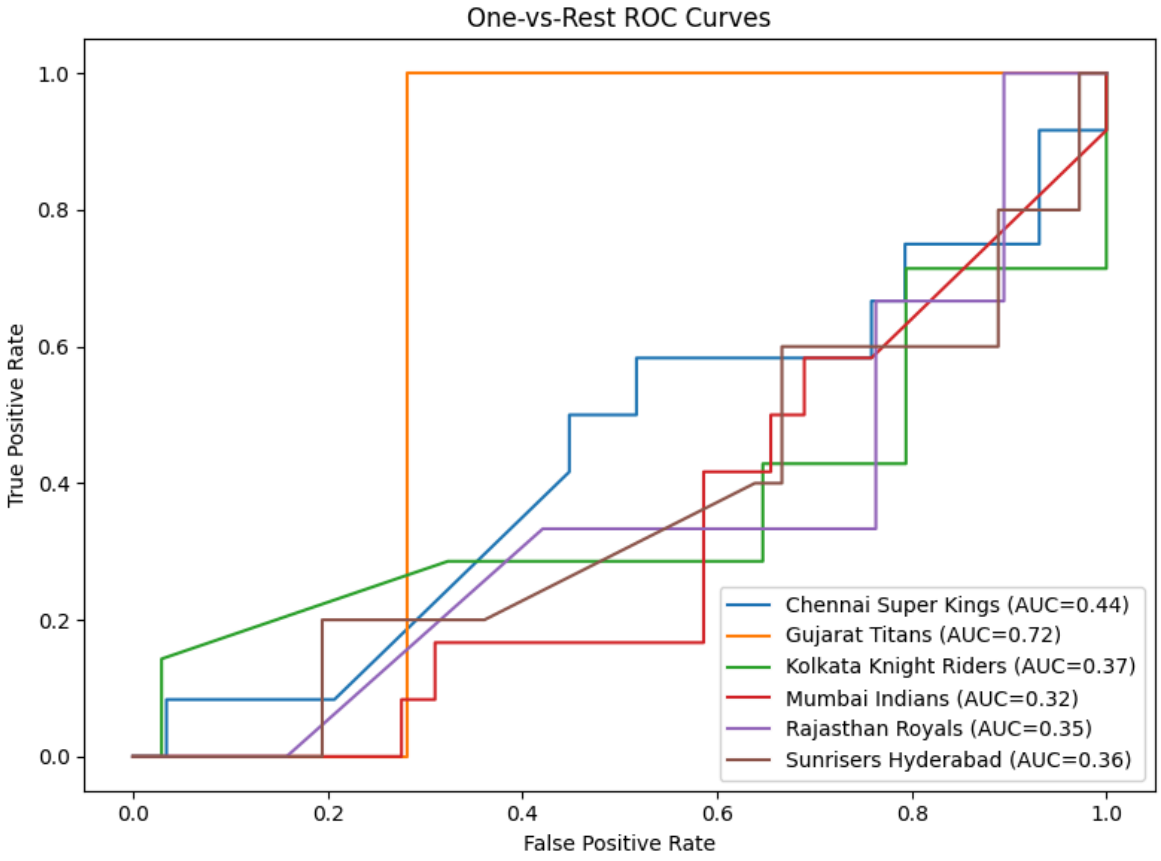
Support Vector Machines (SVMs)

In [ ]:
```
from sklearn.metrics import (
    confusion_matrix,
    accuracy_score,
    classification_report,
    roc_auc_score,
    roc_curve
)


# Rename season column from matches side to 'season'
if 'season_mat' in final_df.columns:
    final_df = final_df.rename(columns={'season_mat':'season'})
elif 'season_match' in final_df.columns:
    final_df = final_df.rename(columns={'season_match':'season'})
else:
    final_df = final_df.rename(columns={'season_del':'season'})

# Identify champions by taking last match per season
matches_meta = (
    final_df[['match_id','season','date','winner']]
    .drop_duplicates('match_id')
)
matches_meta['date'] = pd.to_datetime(matches_meta['date'])
finals = (
    matches_meta
    .sort_values(['season','date'], ascending=[True,True])
    .groupby('season', sort=False)
    .tail(1)[['season','winner']]
    .rename(columns={'winner':'champion'})
)
```

```
# Feature engineering per team-season
dismissal_kinds = ['bowled','caught','lbw','stumped','caught and bowled','hit wi
teams = final_df['batting_team'].unique()

records = []
for season in sorted(final_df['season'].unique()):
    df_s = final_df[final_df['season']==season]
    for team in teams:
        bat = df_s[df_s['batting_team']==team]
        total_runs  = bat['batsman_runs'].sum()
        dismissals  = bat['player_dismissed'].notna().sum() or 1
        bat_avg     = total_runs / dismissals

        bowl = df_s[df_s['bowling_team']==team]
        conceded    = bowl['total_runs'].sum()
        wkts        = bowl['dismissal_kind'].isin(dismissal_kinds).sum() or 1
        bowl_avg    = conceded / wkts

        top5_runs   = bat.groupby('batter')['batsman_runs'].sum().nlargest(5).su
        top5_wkts   = (
            bowl[bowl['dismissal_kind'].isin(dismissal_kinds)]
            .groupby('bowler').size()
            .nlargest(5).sum()
        )

        records.append({
            'season': season,
            'team': team,
            'bat_avg': bat_avg,
            'bowl_avg': bowl_avg,
            'top5_runs_sum': top5_runs,
            'top5_wkts_sum': top5_wkts
        })

features_df = pd.DataFrame(records)

# Merge with champion labels
data = features_df.merge(finals, on='season')

# Prepare X and y
X = data[['bat_avg','bowl_avg','top5_runs_sum','top5_wkts_sum']]
y = data['champion']

#  Encode and scale
le      = LabelEncoder()
y_enc   = le.fit_transform(y)
scaler  = StandardScaler()
X_std   = scaler.fit_transform(X)

# Split train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_std, y_enc, test_size=0.2, random_state=42, stratify=y_enc
)

# Set up SVM & grid search
svc = SVC(probability=True, random_state=42)

param_grid = {
    'C':    [0.1, 1, 10],
```

```
    'gamma':['scale', 'auto'],
    'kernel':['rbf','linear']
}

grid = GridSearchCV(
    estimator=svc,
    param_grid=param_grid,
    scoring='accuracy',
    cv=3,
    verbose=1,
    return_train_score=True
)
grid.fit(X_train, y_train)

print("Best SVM params:", grid.best_params_)

# Train final SVM
best_svm = grid.best_estimator_
best_svm.fit(X_train, y_train)

# Evaluate on training set
y_tr_pred  = best_svm.predict(X_train)
y_tr_proba = best_svm.predict_proba(X_train)

print("Train Accuracy:", accuracy_score(y_train, y_tr_pred))
print("Train Confusion Matrix:\n", confusion_matrix(y_train, y_tr_pred))
print("Train Classification Report:\n", classification_report(y_train, y_tr_pred
print("Train ROC AUC (ovr):", roc_auc_score(y_train, y_tr_proba, multi_class='ov

# Evaluate on test set
y_te_pred  = best_svm.predict(X_test)
y_te_proba = best_svm.predict_proba(X_test)

print("Test Accuracy:", accuracy_score(y_test, y_te_pred))
print("Test Confusion Matrix:\n", confusion_matrix(y_test, y_te_pred))
print("Test Classification Report:\n", classification_report(y_test, y_te_pred))
print("Test ROC AUC (ovr):", roc_auc_score(y_test, y_te_proba, multi_class='ovr'

# Plot ROC curves per class
plt.figure(figsize=(8,6))
for i, cls in enumerate(le.classes_):
    fpr, tpr, _ = roc_curve(y_test==i, y_te_proba[:,i])
    plt.plot(fpr, tpr, label=f'{cls} (AUC={roc_auc_score(y_test==i, y_te_proba[:
plt.title('One-vs-Rest ROC Curves')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
Best SVM params: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
Train Accuracy: 0.3619631901840491
Train Confusion Matrix:
 [[26  0  0 22  0  0]
 [ 5  0  0  5  0  0]
 [15  0  0 14  0  0]
 [17  0  0 31  0  0]
 [ 2  0  0  7  0  0]
 [ 7  0  0 10  0  2]]
Train Classification Report:
              precision    recall  f1-score   support

           0       0.36      0.54      0.43        48
           1       0.00      0.00      0.00        10
           2       0.00      0.00      0.00        29
           3       0.35      0.65      0.45        48
           4       0.00      0.00      0.00         9
           5       1.00      0.11      0.19        19

    accuracy                           0.36       163
   macro avg       0.28      0.22      0.18       163
weighted avg       0.33      0.36      0.28       163

Train ROC AUC (ovr): 0.4721703454064734
Test Accuracy: 0.34146341463414637
Test Confusion Matrix:
 [[5 0 0 7 0 0]
 [1 0 0 1 0 0]
 [3 0 0 4 0 0]
 [3 0 0 9 0 0]
 [1 0 0 2 0 0]
 [3 0 0 2 0 0]]
Test Classification Report:
              precision    recall  f1-score   support

           0       0.31      0.42      0.36        12
           1       0.00      0.00      0.00         2
           2       0.00      0.00      0.00         7
           3       0.36      0.75      0.49        12
           4       0.00      0.00      0.00         3
           5       0.00      0.00      0.00         5

    accuracy                           0.34        41
   macro avg       0.11      0.19      0.14        41
weighted avg       0.20      0.34      0.25        41

Test ROC AUC (ovr): 0.424279135235151
```



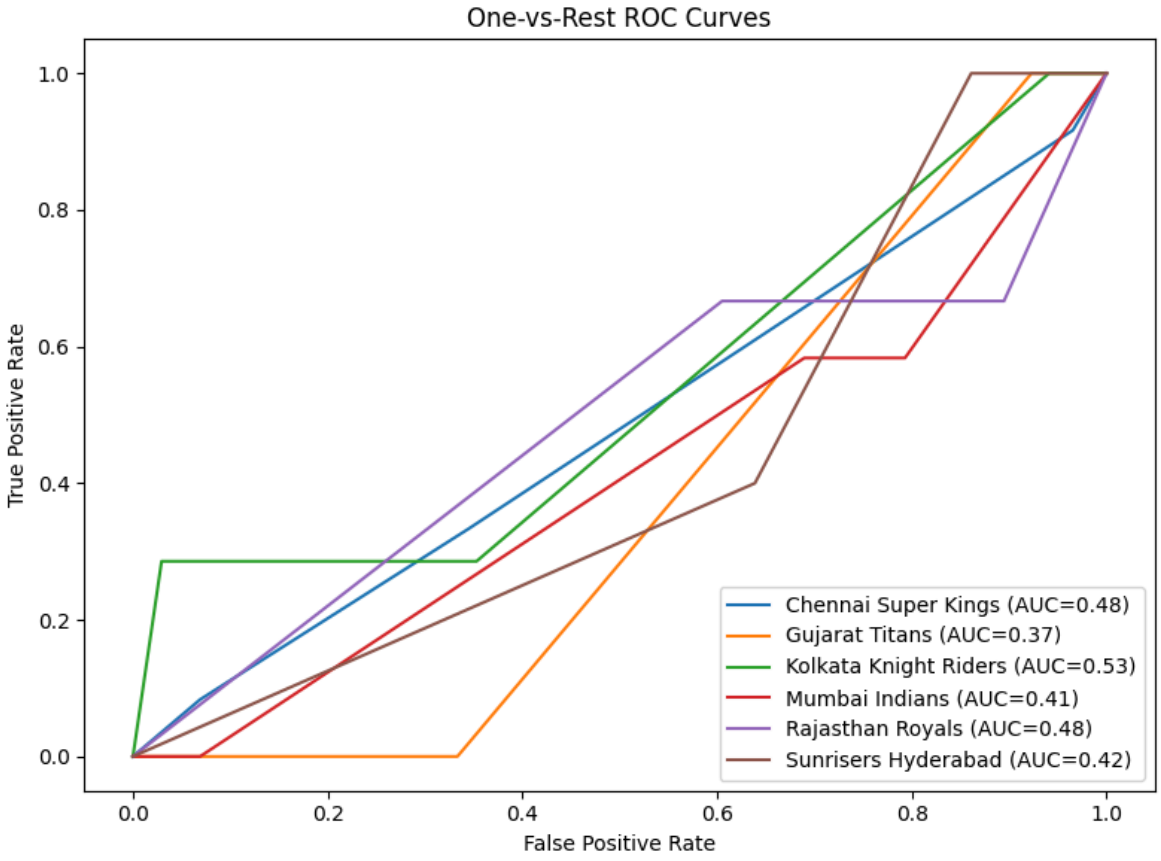
Decision Tree Classifier

In [ ]:
```
matches_meta = (
    final_df[['match_id','season','date','winner']]
    .drop_duplicates('match_id')
)
matches_meta['date'] = pd.to_datetime(matches_meta['date'])
finals = (
    matches_meta
    .sort_values(['season','date'])
    .groupby('season', sort=False)
    .tail(1)[['season','winner']]
    .rename(columns={'winner':'champion'})
)

# Feature engineering per team-season
dismissal_kinds = [
    'bowled','caught','lbw','stumped',
    'caught and bowled','hit wicket'
]
teams = final_df['batting_team'].unique()

records = []
for season in sorted(final_df['season'].unique()):
    df_s = final_df[final_df['season']==season]
    for team in teams:
        bat = df_s[df_s['batting_team']==team]
        total_runs  = bat['batsman_runs'].sum()
        dismissals  = bat['player_dismissed'].notna().sum() or 1
        bat_avg     = total_runs / dismissals

        bowl = df_s[df_s['bowling_team']==team]
```

```
        conceded    = bowl['total_runs'].sum()
        wkts        = bowl['dismissal_kind'].isin(dismissal_kinds).sum() or 1
        bowl_avg    = conceded / wkts

        top5_runs = bat.groupby('batter')['batsman_runs']\
                       .sum().nlargest(5).sum()
        top5_wkts = (
            bowl[bowl['dismissal_kind'].isin(dismissal_kinds)]
            .groupby('bowler').size()
            .nlargest(5).sum()
        )

        records.append({
            'season': season,
            'team': team,
            'bat_avg': bat_avg,
            'bowl_avg': bowl_avg,
            'top5_runs_sum': top5_runs,
            'top5_wkts_sum': top5_wkts
        })

features_df = pd.DataFrame(records)

# Merge with champion labels
data = features_df.merge(finals, on='season')

#  Prepare X and y
X = data[['bat_avg','bowl_avg','top5_runs_sum','top5_wkts_sum']]
y = data['champion']

#  Encode target
le = LabelEncoder()
y_enc = le.fit_transform(y)

#(Optional) scale features for consistency
scaler = StandardScaler()
X_std  = scaler.fit_transform(X)

# Split train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_std, y_enc, test_size=0.2, random_state=42, stratify=y_enc
)

# Set up Decision Tree and GridSearchCV
dt = DecisionTreeClassifier(random_state=42)
param_grid = {
    'max_depth': [3, 5, 7, None],
    'min_samples_split': [2, 5, 10],
    'criterion': ['gini', 'entropy']
}
grid = GridSearchCV(
    estimator=dt,
    param_grid=param_grid,
    scoring='accuracy',
    cv=3,
    verbose=1,
    return_train_score=True
)
grid.fit(X_train, y_train)
print("Best Decision Tree params:", grid.best_params_)
```

```
# Train final model
best_dt = grid.best_estimator_
best_dt.fit(X_train, y_train)

# Evaluate on training set
y_tr_pred  = best_dt.predict(X_train)
y_tr_proba = best_dt.predict_proba(X_train)
print("Train Accuracy:", accuracy_score(y_train, y_tr_pred))
print("Train Confusion Matrix:\n", confusion_matrix(y_train, y_tr_pred))
print("Train Classification Report:\n", classification_report(y_train, y_tr_pred
print("Train ROC AUC (ovr):", roc_auc_score(y_train, y_tr_proba, multi_class='ov

# Evaluate on test set
y_te_pred  = best_dt.predict(X_test)
y_te_proba = best_dt.predict_proba(X_test)
print("Test Accuracy:", accuracy_score(y_test, y_te_pred))
print("Test Confusion Matrix:\n", confusion_matrix(y_test, y_te_pred))
print("Test Classification Report:\n", classification_report(y_test, y_te_pred))
print("Test ROC AUC (ovr):", roc_auc_score(y_test, y_te_proba, multi_class='ovr'

# Plot ROC curves (one-vs-rest)
plt.figure(figsize=(8,6))
for i, cls in enumerate(le.classes_):
    fpr, tpr, _ = roc_curve(y_test==i, y_te_proba[:,i])
    auc = roc_auc_score(y_test==i, y_te_proba[:,i])
    plt.plot(fpr, tpr, label=f'{cls} (AUC={auc:.2f})')
plt.title('One-vs-Rest ROC Curves')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```

```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Best Decision Tree params: {'criterion': 'gini', 'max_depth': 3, 'min_samples_spl
it': 2}
Train Accuracy: 0.4049079754601227
Train Confusion Matrix:
 [[14  0  5 29  0  0]
 [ 4  1  0  5  0  0]
 [10  0  8 11  0  0]
 [ 4  0  2 42  0  0]
 [ 1  0  0  8  0  0]
 [ 3  0  0 15  0  1]]
Train Classification Report:
              precision    recall  f1-score   support

           0       0.39      0.29      0.33        48
           1       1.00      0.10      0.18        10
           2       0.53      0.28      0.36        29
           3       0.38      0.88      0.53        48
           4       0.00      0.00      0.00         9
           5       1.00      0.05      0.10        19

    accuracy                           0.40       163
   macro avg       0.55      0.27      0.25       163
weighted avg       0.50      0.40      0.34       163

Train ROC AUC (ovr): 0.6872724480913993
Test Accuracy: 0.2926829268292683
Test Confusion Matrix:
 [[3 0 1 8 0 0]
 [0 0 0 2 0 0]
 [0 0 2 5 0 0]
 [5 0 0 7 0 0]
 [0 0 0 3 0 0]
 [3 0 0 2 0 0]]
Test Classification Report:
              precision    recall  f1-score   support

           0       0.27      0.25      0.26        12
           1       0.00      0.00      0.00         2
           2       0.67      0.29      0.40         7
           3       0.26      0.58      0.36        12
           4       0.00      0.00      0.00         3
           5       0.00      0.00      0.00         5

    accuracy                           0.29        41
   macro avg       0.20      0.19      0.17        41
weighted avg       0.27      0.29      0.25        41

Test ROC AUC (ovr): 0.44943015543661424
```



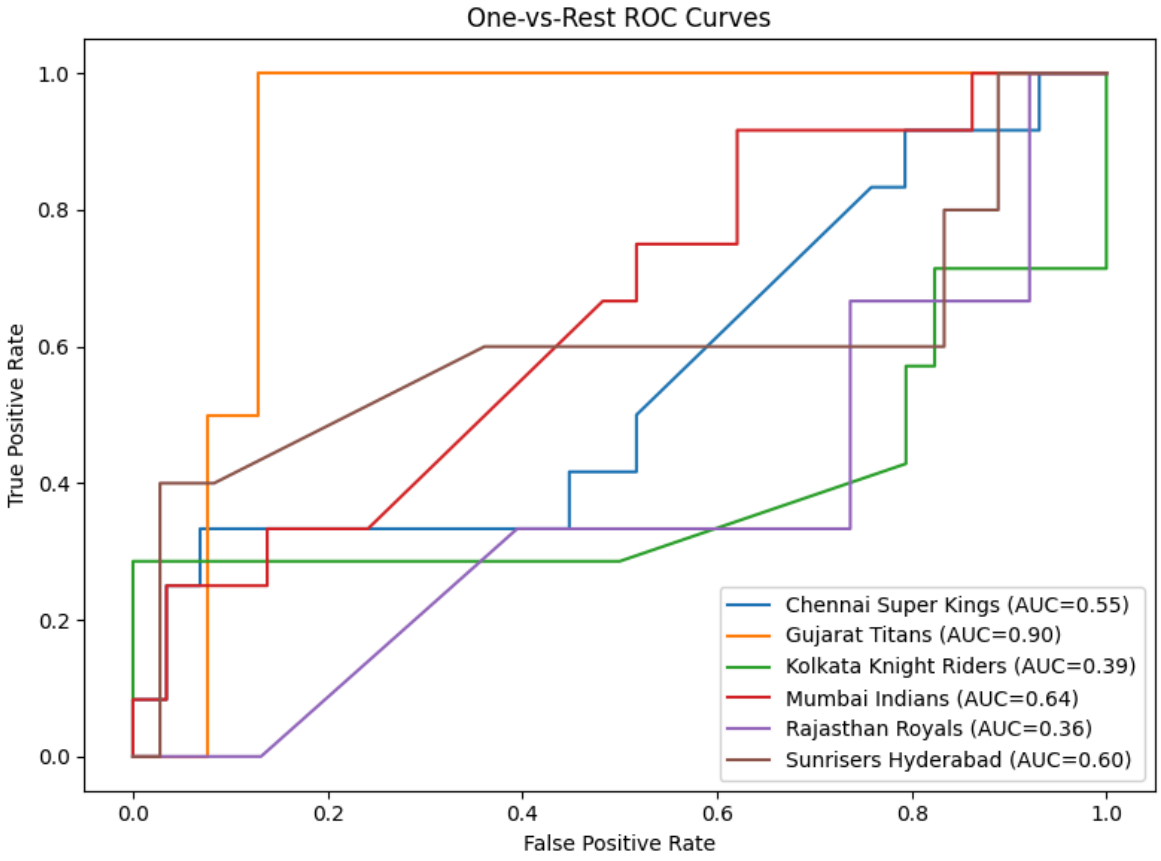
Random Forest Classifier

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import (
    confusion_matrix,
    accuracy_score,
    classification_report,
    roc_auc_score,
    roc_curve
)
import matplotlib.pyplot as plt


# Identify champions by taking last match per season
matches_meta = (
    final_df[['match_id','season','date','winner']]
    .drop_duplicates('match_id')
)
matches_meta['date'] = pd.to_datetime(matches_meta['date'])
finals = (
    matches_meta
    .sort_values(['season','date'])
    .groupby('season', sort=False)
    .tail(1)[['season','winner']]
    .rename(columns={'winner':'champion'})
)

# Feature engineering per team-season
```

```
dismissal_kinds = ['bowled','caught','lbw','stumped','caught and bowled','hit wi
teams = final_df['batting_team'].unique()

records = []
for season in sorted(final_df['season'].unique()):
    df_s = final_df[final_df['season']==season]
    for team in teams:
        bat = df_s[df_s['batting_team']==team]
        total_runs  = bat['batsman_runs'].sum()
        dismissals  = bat['player_dismissed'].notna().sum() or 1
        bat_avg     = total_runs / dismissals

        bowl = df_s[df_s['bowling_team']==team]
        conceded    = bowl['total_runs'].sum()
        wkts        = bowl['dismissal_kind'].isin(dismissal_kinds).sum() or 1
        bowl_avg    = conceded / wkts

        top5_runs = bat.groupby('batter')['batsman_runs'].sum().nlargest(5).sum(
        top5_wkts = (
            bowl[bowl['dismissal_kind'].isin(dismissal_kinds)]
            .groupby('bowler').size()
            .nlargest(5).sum()
        )

        records.append({
            'season': season,
            'team': team,
            'bat_avg': bat_avg,
            'bowl_avg': bowl_avg,
            'top5_runs_sum': top5_runs,
            'top5_wkts_sum': top5_wkts
        })

features_df = pd.DataFrame(records)

# Merge with champion labels
data = features_df.merge(finals, on='season')

# Prepare X and y
X = data[['bat_avg','bowl_avg','top5_runs_sum','top5_wkts_sum']]
y = data['champion']

# Encode target & scale features
le     = LabelEncoder()
y_enc  = le.fit_transform(y)
scaler = StandardScaler()
X_std  = scaler.fit_transform(X)

# Split train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_std, y_enc, test_size=0.2, random_state=42, stratify=y_enc
)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}
```

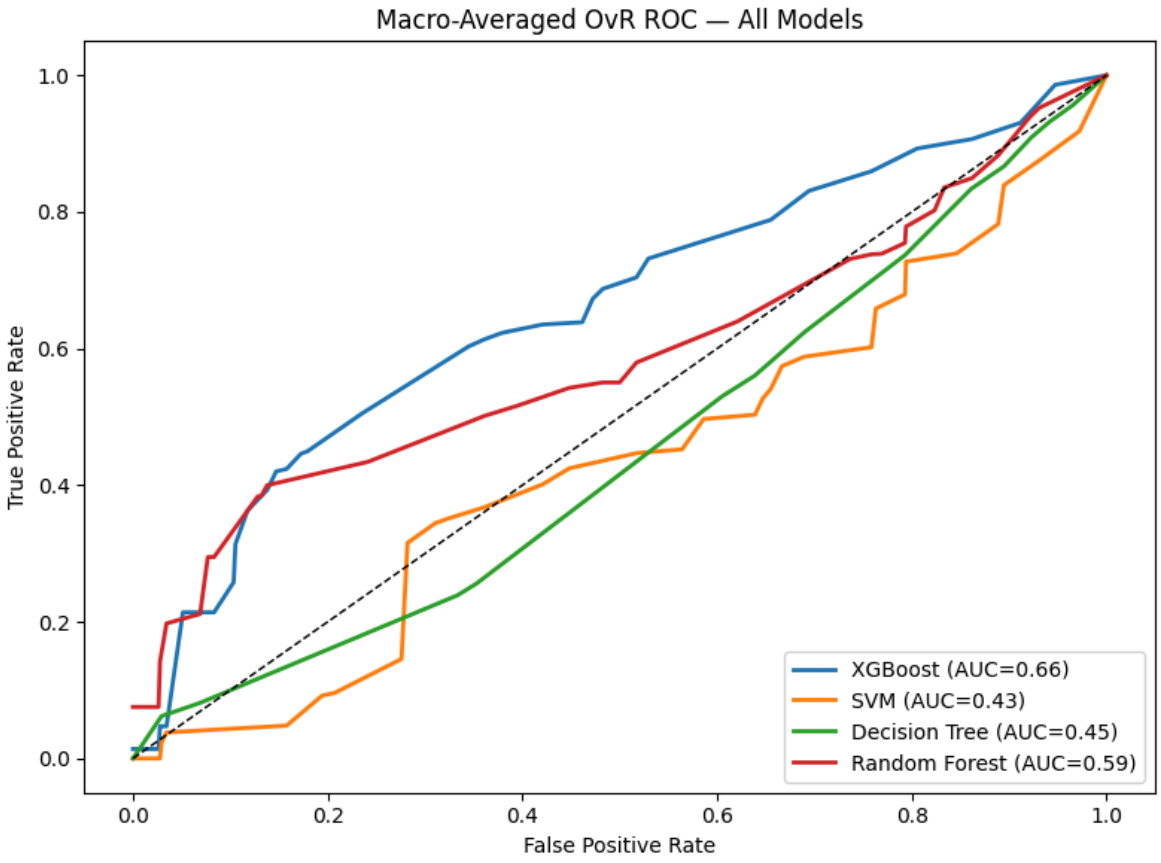
```
rf = RandomForestClassifier(random_state=42)
grid = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    scoring='accuracy',
    cv=3,
    verbose=1,
    return_train_score=True
)
grid.fit(X_train, y_train)
print("Best RF params:", grid.best_params_)

# Train final model
best_rf = grid.best_estimator_
best_rf.fit(X_train, y_train)

# Evaluate on training set
y_tr_pred  = best_rf.predict(X_train)
y_tr_proba = best_rf.predict_proba(X_train)
print("Train Accuracy:", accuracy_score(y_train, y_tr_pred))
print("Train Confusion Matrix:\n", confusion_matrix(y_train, y_tr_pred))
print("Train Classification Report:\n", classification_report(y_train, y_tr_pred
print("Train ROC AUC (ovr):", roc_auc_score(y_train, y_tr_proba, multi_class='ov

# Evaluate on test set
y_te_pred  = best_rf.predict(X_test)
y_te_proba = best_rf.predict_proba(X_test)
print("Test Accuracy:", accuracy_score(y_test, y_te_pred))
print("Test Confusion Matrix:\n", confusion_matrix(y_test, y_te_pred))
print("Test Classification Report:\n", classification_report(y_test, y_te_pred))
print("Test ROC AUC (ovr):", roc_auc_score(y_test, y_te_proba, multi_class='ovr'

# Plot ROC curves (one-vs-rest)
plt.figure(figsize=(8,6))
for i, cls in enumerate(le.classes_):
    fpr, tpr, _ = roc_curve(y_test == i, y_te_proba[:, i])
    auc = roc_auc_score(y_test == i, y_te_proba[:, i])
    plt.plot(fpr, tpr, label=f'{cls} (AUC={auc:.2f})')
plt.title('One-vs-Rest ROC Curves')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```

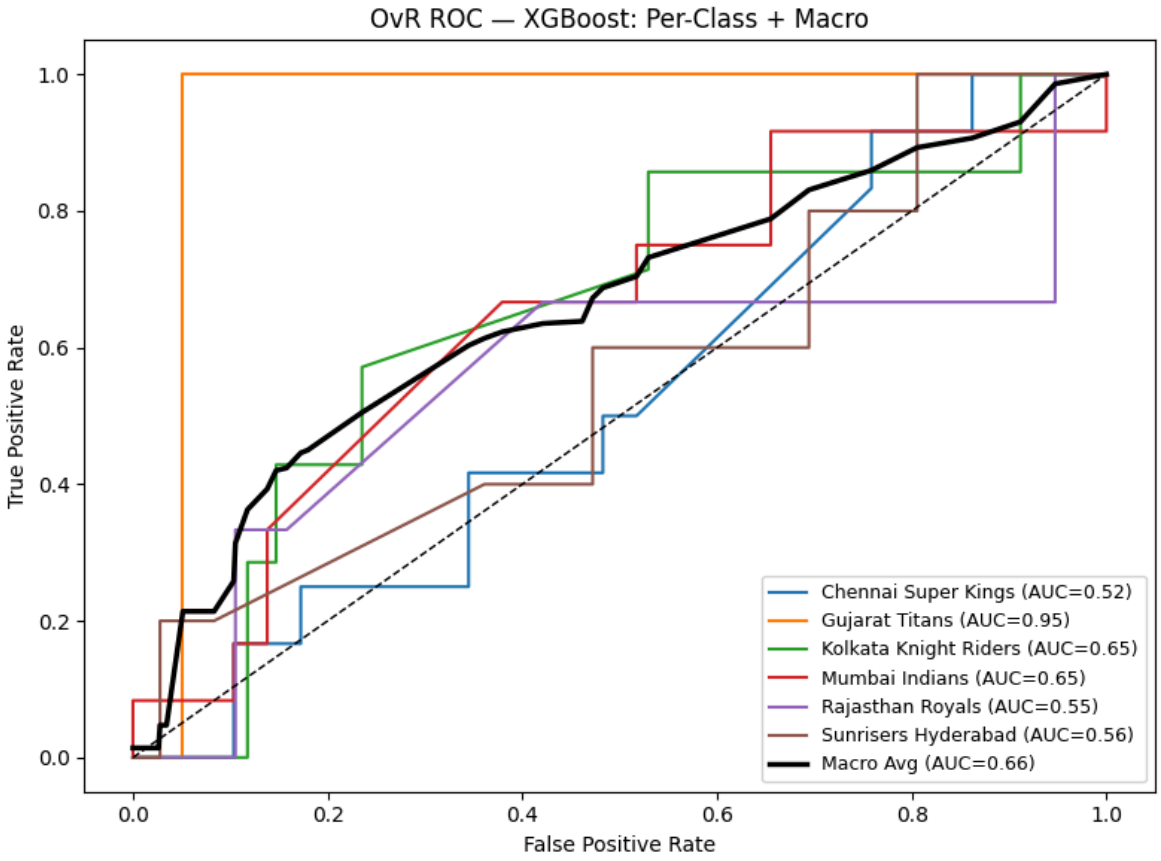
```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Best RF params: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 5,
'n_estimators': 200}
Train Accuracy: 0.656441717791411
Train Confusion Matrix:
 [[35  0  0 13  0  0]
 [ 4  2  0  4  0  0]
 [ 5  0 16  8  0  0]
 [ 2  0  0 46  0  0]
 [ 3  0  0  5  1  0]
 [ 2  0  1  9  0  7]]
Train Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.73      0.71        48
           1       1.00      0.20      0.33        10
           2       0.94      0.55      0.70        29
           3       0.54      0.96      0.69        48
           4       1.00      0.11      0.20         9
           5       1.00      0.37      0.54        19

    accuracy                           0.66       163
   macro avg       0.86      0.49      0.53       163
weighted avg       0.76      0.66      0.63       163

Train ROC AUC (ovr): 0.9045284011795379
Test Accuracy: 0.3902439024390244
Test Confusion Matrix:
 [[5 0 0 7 0 0]
 [1 0 0 1 0 0]
 [2 0 2 3 0 0]
 [3 0 0 8 0 1]
 [1 0 0 2 0 0]
 [3 0 0 1 0 1]]
Test Classification Report:
              precision    recall  f1-score   support

           0       0.33      0.42      0.37        12
           1       0.00      0.00      0.00         2
           2       1.00      0.29      0.44         7
           3       0.36      0.67      0.47        12
           4       0.00      0.00      0.00         3
           5       0.50      0.20      0.29         5

    accuracy                           0.39        41
   macro avg       0.37      0.26      0.26        41
weighted avg       0.44      0.39      0.36        41

Test ROC AUC (ovr): 0.5738739369357497
```



In [92]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

# compute macro-averaged ROC (OvR) from per-class scores
def macro_roc(y_true, y_score, n_classes):
    fprs, tprs, aucs = [], [], []
    for i in range(n_classes):
        y_true_i = (y_true == i).astype(int)

        if y_true_i.sum() == 0:
            continue
        fpr, tpr, _ = roc_curve(y_true_i, y_score[:, i])
        fprs.append(fpr)
        tprs.append(tpr)
        aucs.append(auc(fpr, tpr))

    # Union of all FPR points
    all_fpr = np.unique(np.concatenate([f for f in fprs]))
    mean_tpr = np.zeros_like(all_fpr)
    for fpr, tpr in zip(fprs, tprs):
        mean_tpr += np.interp(all_fpr, fpr, tpr)
    mean_tpr /= len(tprs)
    macro_auc = auc(all_fpr, mean_tpr)
    return all_fpr, mean_tpr, macro_auc

# Collect your fitted models
models = {
    'XGBoost': model,
    'SVM': best_svm,
    'Decision Tree': best_dt,
    'Random Forest': best_rf
}
```

```
n_classes = len(le.classes_)

# Overlay ONE macro-averaged ROC curve per model ====
plt.figure(figsize=(8,6))
for name, clf in models.items():
    y_score = clf.predict_proba(X_test)
    fpr_m, tpr_m, auc_m = macro_roc(y_test, y_score, n_classes)
    plt.plot(fpr_m, tpr_m, lw=2, label=f'{name} (AUC={auc_m:.2f})')

plt.plot([0,1], [0,1], 'k--', lw=1)
plt.title('Macro-Averaged OvR ROC — All Models')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()


model_name_to_expand = 'XGBoost'
clf = models[model_name_to_expand]
y_score = clf.predict_proba(X_test)

plt.figure(figsize=(8,6))
# per-class curves
for i, cls_name in enumerate(le.classes_):
    mask_pos = (y_test == i)
    if mask_pos.sum() == 0:
        continue
    fpr_i, tpr_i, _ = roc_curve(mask_pos.astype(int), y_score[:, i])
    auc_i = auc(fpr_i, tpr_i)
    plt.plot(fpr_i, tpr_i, lw=1.5, label=f'{cls_name} (AUC={auc_i:.2f})')

# macro curve
fpr_m, tpr_m, auc_m = macro_roc(y_test, y_score, n_classes)
plt.plot(fpr_m, tpr_m, color='black', lw=2.5, label=f'Macro Avg (AUC={auc_m:.2f}

plt.plot([0,1], [0,1], 'k--', lw=1)
plt.title(f'OvR ROC — {model_name_to_expand}: Per-Class + Macro')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right', fontsize=9)
plt.tight_layout()
plt.show()
```





In [98]:
```
import pandas as pd
from sklearn.metrics import accuracy_score, roc_auc_score, precision_recall_fsco


models = {
    'XGBoost': model,
    'SVM': best_svm,
```

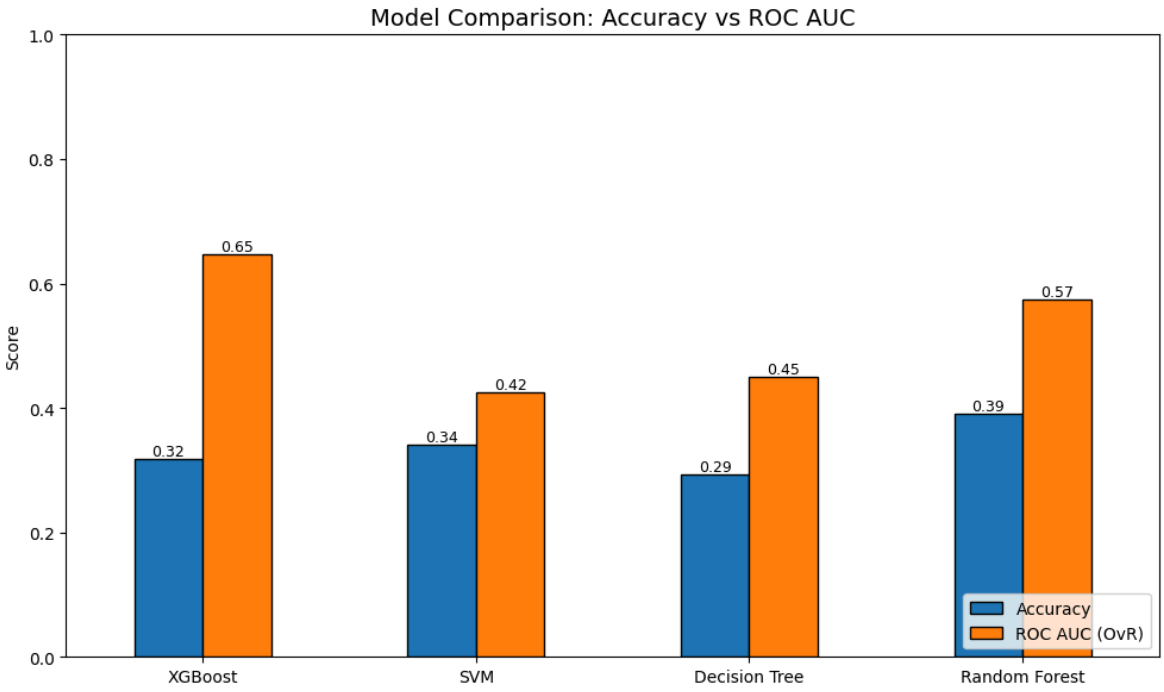
```
    'Decision Tree': best_dt,
    'Random Forest': best_rf
}

def build_res_df(models, X_train, y_train, X_test, y_test):
    rows = []
    for name, clf in models.items():
        # fit if not already fitted
        if not hasattr(clf, "classes_"):
            clf.fit(X_train, y_train)

        y_pred  = clf.predict(X_test)
        y_proba = clf.predict_proba(X_test)

        acc = accuracy_score(y_test, y_pred)
        auc = roc_auc_score(y_test, y_proba, multi_class='ovr')

        p_macro, r_macro, f1_macro, _ = precision_recall_fscore_support(
            y_test, y_pred, average='macro', zero_division=0
        )

        rows.append({
            'Model': name,
            'Accuracy': acc,
            'ROC AUC (OvR)': auc,
            'Precision (macro)': p_macro,
            'Recall (macro)': r_macro,
            'F1 (macro)': f1_macro
        })

    return pd.DataFrame(rows).set_index('Model')

res_df = build_res_df(models, X_train, y_train, X_test, y_test)
print(res_df.round(3))
```

```
               Accuracy  ROC AUC (OvR)  Precision (macro)  Recall (macro)  \
Model
XGBoost           0.317          0.646              0.227           0.210
SVM               0.341          0.424              0.112           0.194
Decision Tree     0.293          0.449              0.200           0.187
Random Forest     0.390          0.574              0.366           0.262

               F1 (macro)
Model
XGBoost             0.198
SVM                 0.141
Decision Tree       0.170
Random Forest       0.262
```

In [97]:

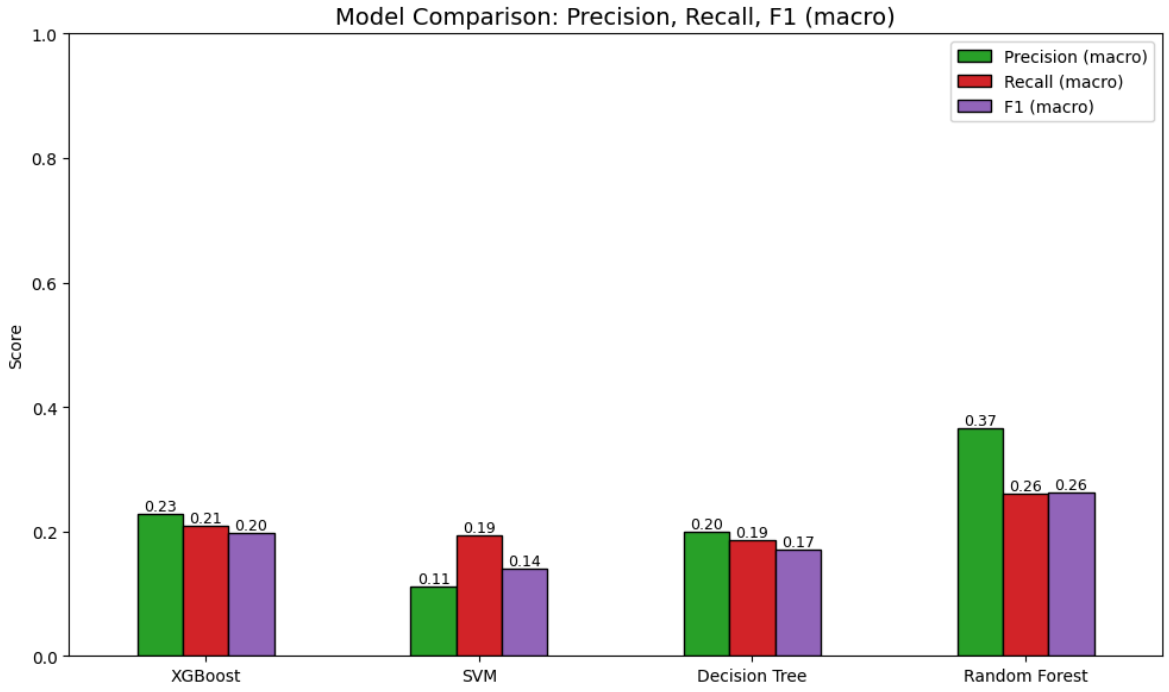
```
import matplotlib.pyplot as plt

ax = res_df[['Accuracy','ROC AUC (OvR)']].plot(
    kind='bar', figsize=(10,6), edgecolor='black',
    color=['#1f77b4','#ff7f0e']
)
for p in ax.patches:
    h = p.get_height()
    ax.annotate(f"{h:.2f}", (p.get_x()+p.get_width()/2, h),
                ha='center', va='bottom', fontsize=9)
ax.set_title('Model Comparison: Accuracy vs ROC AUC', fontsize=14)
```

```
ax.set_ylabel('Score'); ax.set_xlabel('')
ax.set_ylim(0, 1); plt.xticks(rotation=0)
ax.legend(loc='lower right', fontsize=10)
plt.tight_layout(); plt.show()
```



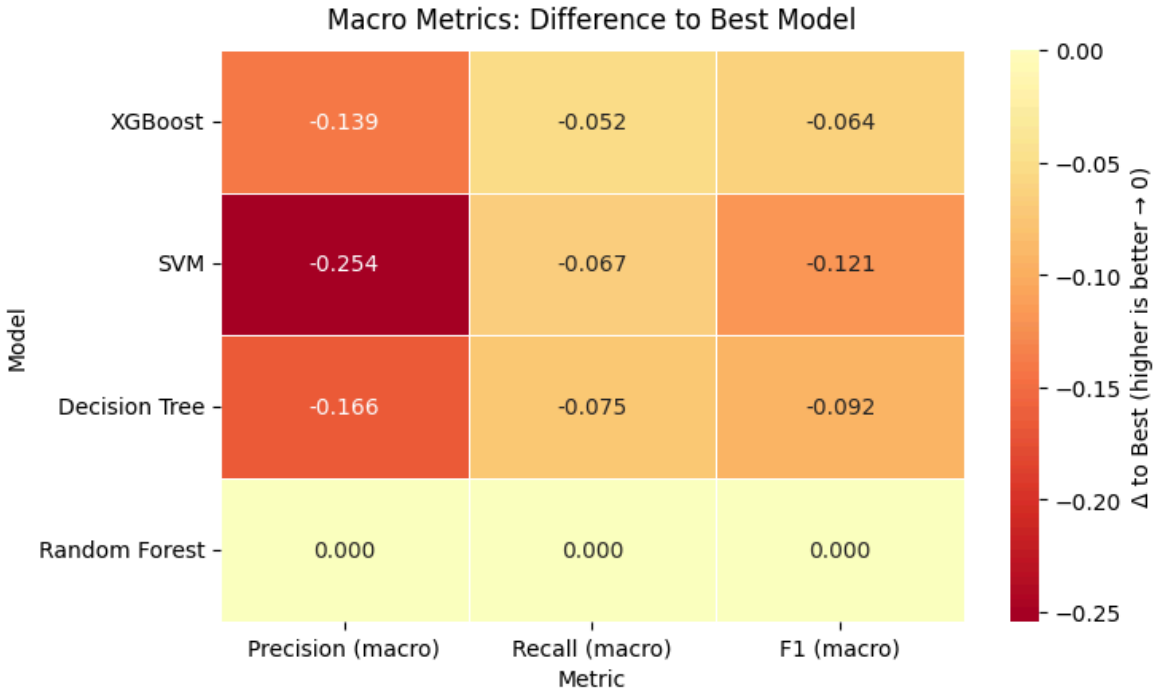
In [100…

```
ax = res_df[['Precision (macro)','Recall (macro)','F1 (macro)']].plot(
    kind='bar', figsize=(10,6), edgecolor='black',
    color=['#2ca02c', '#d62728', '#9467bd']
)
for p in ax.patches:
    h = p.get_height()
    ax.annotate(f"{h:.2f}", (p.get_x()+p.get_width()/2, h),
                ha='center', va='bottom', fontsize=9)
ax.set_title('Model Comparison: Precision, Recall, F1 (macro)', fontsize=14)
ax.set_ylabel('Score'); ax.set_xlabel('')
ax.set_ylim(0, 1); plt.xticks(rotation=0)
ax.legend(loc='upper right', fontsize=10)
plt.tight_layout(); plt.show()
```



In [101…

```
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

macro_df = res_df[['Precision (macro)','Recall (macro)','F1 (macro)']].copy()
delta_to_best = macro_df.sub(macro_df.max(axis=0), axis=1)  # best = 0

plt.figure(figsize=(7.5, 0.9*len(delta_to_best)+1))
sns.heatmap(
    delta_to_best.round(3),
    annot=True, fmt=".3f",
    cmap='RdYlGn', center=0,
    vmin=delta_to_best.min().min(), vmax=0,
    linewidths=0.5, linecolor='white',
    cbar_kws={'label': 'Δ to Best (higher is better → 0)'}
)
plt.title('Macro Metrics: Difference to Best Model', pad=10)
plt.ylabel('Model'); plt.xlabel('Metric')
plt.tight_layout(); plt.show()
```



Predicting Next Season Champion

In [104…

```
import pandas as pd
import numpy as np
from xgboost import XGBClassifier
from sklearn.preprocessing import LabelEncoder, StandardScaler


# Identify champions by taking the last match of each season
matches_meta = (
    final_df[['match_id','season','date','winner']]
    .drop_duplicates(subset='match_id')
)
matches_meta['date'] = pd.to_datetime(matches_meta['date'])
finals = (
    matches_meta
    .sort_values(['season','date'])
    .groupby('season', sort=False)
    .tail(1)[['season','winner']]
    .rename(columns={'winner':'champion'})
)

# Feature engineering per team-season
dismissal_kinds = ['bowled','caught','lbw','stumped','caught and bowled','hit wi
teams = final_df['batting_team'].unique()

records = []
for season in sorted(final_df['season'].unique()):
    df_s = final_df[final_df['season']==season]
    for team in teams:
        # Batting metrics
        bat = df_s[df_s['batting_team']==team]
        total_runs  = bat['batsman_runs'].sum()
        dismissals  = bat['player_dismissed'].notna().sum() or 1
        bat_avg     = total_runs / dismissals

        # Bowling metrics
```

```
        bowl = df_s[df_s['bowling_team']==team]
        conceded    = bowl['total_runs'].sum()
        wkts        = bowl['dismissal_kind'].isin(dismissal_kinds).sum() or 1
        bowl_avg    = conceded / wkts

        # Top-5 contributors
        top5_runs = bat.groupby('batter')['batsman_runs'].sum().nlargest(5).sum(
        top5_wkts = (
            bowl[bowl['dismissal_kind'].isin(dismissal_kinds)]
            .groupby('bowler').size().nlargest(5).sum()
        )

        records.append({
            'season': season,
            'team': team,
            'bat_avg': bat_avg,
            'bowl_avg': bowl_avg,
            'top5_runs_sum': top5_runs,
            'top5_wkts_sum': top5_wkts
        })

features_df = pd.DataFrame(records)

# Merge features with champion labels
data = features_df.merge(finals, on='season')

# Prepare X and y
X = data[['bat_avg','bowl_avg','top5_runs_sum','top5_wkts_sum']]
y = data['champion']

# Encode target and scale features
le     = LabelEncoder()
y_enc  = le.fit_transform(y)
scaler = StandardScaler()
X_std  = scaler.fit_transform(X)

# Train final XGBoost on all historical data
model = XGBClassifier(
    max_depth=2,
    n_estimators=200,
    learning_rate=0.2,
    subsample=0.3,
    objective='multi:softprob',
    use_label_encoder=False,
    eval_metric='mlogloss',
    random_state=42
)
model.fit(X_std, y_enc)

# Build next-season feature matrix (latest season only)
latest = final_df['season'].max()
next_df = final_df[final_df['season']==latest]

upcoming = []
for team in teams:
    bat = next_df[next_df['batting_team']==team]
    total_runs  = bat['batsman_runs'].sum()
    dismissals  = bat['player_dismissed'].notna().sum() or 1
    bat_avg     = total_runs / dismissals
```

```
    bowl = next_df[next_df['bowling_team']==team]
    conceded    = bowl['total_runs'].sum()
    wkts        = bowl['dismissal_kind'].isin(dismissal_kinds).sum() or 1
    bowl_avg    = conceded / wkts

    top5_runs = bat.groupby('batter')['batsman_runs'].sum().nlargest(5).sum()
    top5_wkts = (
        bowl[bowl['dismissal_kind'].isin(dismissal_kinds)]
        .groupby('bowler').size().nlargest(5).sum()
    )

    upcoming.append({
        'bat_avg': bat_avg,
        'bowl_avg': bowl_avg,
        'top5_runs_sum': top5_runs,
        'top5_wkts_sum': top5_wkts
    })

X_next = pd.DataFrame(upcoming, index=teams)
X_next_std = scaler.transform(X_next)

# Predict next-season champion
proba = model.predict_proba(X_next_std)
idx  = np.argmax(proba[:, :], axis=0)[0]  # highest probability overall
pred_team = le.inverse_transform([idx])[0]
pred_prob = proba[idx, idx] if False else proba[np.arange(len(teams)), idx].max(

print(f"Predicted champion for Season {latest+1}: {pred_team}")
```

```
Predicted champion for Season 2025: Kolkata Knight Riders
```

Predicting Furtue Player

In [105…

```
import pandas as pd
import numpy as np
from xgboost import XGBRegressor
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error


# Feature engineering per player-season

# Batsman features
df_bat = (
    final_df
    .groupby(['batter','season'], as_index=False)
    .agg(
        total_runs  = ('batsman_runs','sum'),
        balls_faced = ('ball','count'),
        dismissals  = ('player_dismissed','count')
    )
)
df_bat['dismissals'].replace(0, np.nan, inplace=True)
df_bat['bat_avg'] = df_bat['total_runs'] / df_bat['dismissals']
df_bat['sr']      = 100 * df_bat['total_runs'] / df_bat['balls_faced']
df_bat.dropna(subset=['bat_avg','sr'], inplace=True)

# Bowler features
wicket_kinds = ['bowled','caught','lbw','stumped','caught and bowled','hit wicke
```

```
df_bowl = (
    final_df[final_df['dismissal_kind'].isin(wicket_kinds)]
    .groupby(['bowler','season'], as_index=False)
    .agg(
        wickets       = ('dismissal_kind','count'),
        runs_conceded = ('total_runs','sum'),
        balls_bowled  = ('ball','count')
    )
)
df_bowl['overs'] = df_bowl['balls_bowled'] / 6
df_bowl['econ']  = df_bowl['runs_conceded'] / df_bowl['overs']

# Fielder features (catches)
df_field = (
    final_df[final_df['dismissal_kind']=='caught']
    .dropna(subset=['fielder'])
    .groupby(['fielder','season'], as_index=False)
    .agg(catches=('dismissal_kind','count'))
)

# Helper to build t→t+1 training sets
def make_reg_train(df, player_col, feature_cols, target_col):
    tmp = df.rename(columns={player_col:'player'}) \
            .sort_values(['player','season'])
    tmp['target'] = tmp.groupby('player')[target_col].shift(-1)
    tmp.dropna(subset=feature_cols + ['target'], inplace=True)
    X = tmp[feature_cols]
    y = tmp['target']
    return X, y

# Build training sets for each role
X_bat, y_bat   = make_reg_train(df_bat,   'batter', ['total_runs','bat_avg','sr'
X_bowl, y_bowl = make_reg_train(df_bowl,  'bowler',  ['wickets','econ'],
X_fld, y_fld   = make_reg_train(df_field, 'fielder', ['catches'],

# Create separate imputers per role
imp_bat  = SimpleImputer(strategy='mean'); Xb_imp  = imp_bat.fit_transform(X_bat
imp_bowl = SimpleImputer(strategy='mean'); Xl_imp  = imp_bowl.fit_transform(X_bo
imp_fld  = SimpleImputer(strategy='mean'); Xf_imp  = imp_fld.fit_transform(X_fld

# Split into train/test
Xb_tr, Xb_te, yb_tr, yb_te = train_test_split(Xb_imp, y_bat, test_size=0.2, rand
Xl_tr, Xl_te, yl_tr, yl_te = train_test_split(Xl_imp, y_bowl, test_size=0.2, ran
Xf_tr, Xf_te, yf_tr, yf_te = train_test_split(Xf_imp, y_fld, test_size=0.2, rand

# Train XGB regressors with chosen hyperparameters
params = {
    'learning_rate': 0.2,
    'subsample':     0.3,
    'max_depth':     3,
    'n_estimators': 200,
    'random_state': 42
}
model_bat  = XGBRegressor(**params).fit(Xb_tr, yb_tr)
model_bowl = XGBRegressor(**params).fit(Xl_tr, yl_tr)
model_fld  = XGBRegressor(**params).fit(Xf_tr, yf_tr)

# Optional: print train RMSE
print("Batsman RMSE:",  np.sqrt(mean_squared_error(yb_tr, model_bat.predict(Xb_t
print("Bowler RMSE:  ",  np.sqrt(mean_squared_error(yl_tr, model_bowl.predict(Xl
```

```
print("Fielder RMSE: ",  np.sqrt(mean_squared_error(yf_tr, model_fld.predict(Xf_

# Prepare next-season feature sets
latest = final_df['season'].max()
bat_new  = df_bat[df_bat['season']==latest][['total_runs','bat_avg','sr']].reset
bowl_new = df_bowl[df_bowl['season']==latest][['wickets','econ']].reset_index(dr
fld_new  = df_field[df_field['season']==latest][['catches']].reset_index(drop=Tr

bat_imp_next  = imp_bat.transform(bat_new)
bowl_imp_next = imp_bowl.transform(bowl_new)
fld_imp_next  = imp_fld.transform(fld_new)

# Player lists for next season
batsmen =  df_bat[df_bat['season']==latest]['batter'].tolist()
bowlers =  df_bowl[df_bowl['season']==latest]['bowler'].tolist()
fielders= df_field[df_field['season']==latest]['fielder'].tolist()

# Predict next-season performance
pred_runs    = model_bat.predict(bat_imp_next)
pred_wkts    = model_bowl.predict(bowl_imp_next)
pred_catches = model_fld.predict(fld_imp_next)

# Select top performers
df_pred_bat   = pd.DataFrame({'batter': batsmen,    'pred_runs':    pred_runs})
df_pred_bowl  = pd.DataFrame({'bowler':  bowlers,    'pred_wickets': pred_wkts})
df_pred_field = pd.DataFrame({'fielder': fielders,   'pred_catches': pred_catche

top4_batsmen  = df_pred_bat.nlargest(5, 'pred_runs')
top5_bowlers  = df_pred_bowl.nlargest(5, 'pred_wickets')
top3_fielders = df_pred_field.nlargest(3, 'pred_catches')

# Output your predicted XI
print(" Predicted Next-Season Squad:")
print("\n Top 5 Batsmen:")
print(top4_batsmen.to_string(index=False))
print("\n Top 5 Bowlers:")
print(top5_bowlers.to_string(index=False))
print("\n Top 3 Catchers:")
print(top3_fielders.to_string(index=False))
```

```
Batsman RMSE: 97.4281368637803
Bowler RMSE:   5.459701093807129
Fielder RMSE:  2.748546018057086
 Predicted Next-Season Squad:

 Top 5 Batsmen:
     batter  pred_runs
   KL Rahul 581.994446
     S Dube 413.816895
    TM Head 411.847717
    RR Pant 389.867798
YBK Jaiswal 385.797119

 Top 5 Bowlers:
        bowler  pred_wickets
      HV Patel     18.328312
     YS Chahal     14.004087
    AD Russell     13.870964
Arshdeep Singh     13.870964
    Avesh Khan     13.870964

 Top 3 Catchers:
    fielder  pred_catches
   KL Rahul      8.508060
   AR Patel      7.219531
Dhruv Jurel      7.219531
```

In [109…

```python
import pandas as pd
import numpy as np
from xgboost import XGBRegressor
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# assume final_df is already loaded and cleaned
df = final_df.copy()
df['season'] = df['season'].astype(int)

# Batting aggregation
bat = (
    df.groupby(['batter','season'], as_index=False)
      .agg(
        runs       = ('batsman_runs','sum'),
        balls      = ('ball','count'),
        dismissals = ('player_dismissed', lambda x: x.notna().sum())
      )
)
bat['dismissals'].replace(0, np.nan, inplace=True)
bat['bat_avg'] = bat['runs'] / bat['dismissals']
bat['sr']      = 100 * bat['runs'] / bat['balls']

# Bowling aggregation
wk = ['bowled','caught','lbw','stumped','caught and bowled','hit wicket']
df['runs_conceded'] = df['total_runs'] - df['batsman_runs']
bowl = (
    df[df['dismissal_kind'].isin(wk)]
      .groupby(['bowler','season'], as_index=False)
      .agg(
        wickets       = ('dismissal_kind','count'),
        runs_conceded = ('runs_conceded','sum'),
```

```
        balls_bowled  = ('ball','count')
      )
)
bowl['overs']    = bowl['balls_bowled'] / 6
bowl['econ']     = bowl['runs_conceded'] / bowl['overs']
bowl['bowl_avg'] = bowl['runs_conceded'] / bowl['wickets'].replace(0, np.nan)

# Fielding (catches) aggregation
fld = (
    df[df['dismissal_kind']=='caught']
      .dropna(subset=['fielder'])
      .groupby(['fielder','season'], as_index=False)
      .agg(catches=('dismissal_kind','count'))
)

# Merge into player-season table
pt = pd.merge(
    bat.rename(columns={'batter':'player'}),
    bowl.rename(columns={'bowler':'player'}),
    on=['player','season'], how='outer'
).fillna(0)

pt = pd.merge(
    pt,
    fld.rename(columns={'fielder':'player'}),
    on=['player','season'], how='outer'
).fillna(0)

# Build combined metric (normalized per season)
for col in ['runs','wickets','catches']:
    pt[f'n{col}'] = pt.groupby('season')[col].transform(lambda g: g / g.max())

pt['combined'] = pt['nruns'] + pt['nwickets'] + pt['ncatches']

# Create t→t+1 training set
train = pt.sort_values(['player','season'])
train['target'] = train.groupby('player')['combined'].shift(-1)
train = train.dropna(subset=['target'])

# Define features (must exist in train)
features = ['runs','bat_avg','sr','wickets','econ','bowl_avg','catches']
X = train[features]
y = train['target']

# Impute & split
imp = SimpleImputer(strategy='mean')
X_imp = imp.fit_transform(X)
X_tr, X_te, y_tr, y_te = train_test_split(X_imp, y, test_size=0.2, random_state=

# Train XGBoost regressor
model = XGBRegressor(
    learning_rate=0.2,
    subsample=0.3,
    max_depth=3,
    n_estimators=200,
    random_state=42
)
model.fit(X_tr, y_tr)
print("Train RMSE:", np.sqrt(mean_squared_error(y_tr, model.predict(X_tr))))
print("Test  RMSE:", np.sqrt(mean_squared_error(y_te, model.predict(X_te))))
```

```
# Predict next season combined score
latest = pt['season'].max()
pt_new = pt[pt['season']==latest].reset_index(drop=True)
X_new  = imp.transform(pt_new[features])
pt_new['pred_score'] = model.predict(X_new)

# Pick top 11 by predicted combined score
top11 = pt_new.nlargest(20, 'pred_score')[[
    'player','runs','catches','pred_score'
]]
print("\nPredicted Top 20 Batsman Next Season:")
print(top11.to_string(index=False))
```

```
Train RMSE: 0.29638716325643305
Test  RMSE: 0.42377466729508845

Predicted Top 20 Batsman Next Season:
         player  runs  catches  pred_score
J Fraser-McGurk 330.0      5.0    1.951343
      H Klaasen 479.0      8.0    1.503825
      SP Narine 488.0      7.0    1.489178
     AD Russell 222.0      3.0    1.470267
        TM Head 567.0      0.0    1.452576
        PD Salt 435.0     12.0    1.395472
     PJ Cummins 136.0      7.0    1.304860
   F du Plessis 438.0      8.0    1.288479
     RM Patidar 395.0      3.0    1.246358
        RR Pant 446.0     11.0    1.199793
Abhishek Sharma 484.0      7.0    1.137236
     RD Gaikwad 583.0      5.0    1.126663
        R Parag 573.0      7.0    1.110792
       N Pooran 499.0      7.0    1.083378
 P Simran Singh 334.0      1.0    1.071391
      SM Curran 270.0      7.0    1.027419
    JM Bairstow 298.0      8.0    1.024546
  Abishek Porel 327.0      2.0    1.019084
      SV Samson 531.0      6.0    1.011139
     R Shepherd  57.0      0.0    0.997404
```

In [107…

```
import pandas as pd
import numpy as np
from xgboost import XGBRegressor
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Feature engineering for bowlers

# Define wicket dismissal kinds
wicket_kinds = ['bowled','caught','lbw','stumped','caught and bowled','hit wicke

# Compute runs conceded per ball
df['runs_conceded'] = df['total_runs'] - df['batsman_runs']

# Aggregate per bowler-season
df_bowl = (
    df[df['dismissal_kind'].isin(wicket_kinds)]
     .groupby(['bowler','season'], as_index=False)
     .agg(
```

```
        wickets       = ('dismissal_kind','count'),
        runs_conceded = ('runs_conceded','sum'),
        balls_bowled  = ('ball','count')
    )
)
# Derive economy and bowling average
df_bowl['overs']    = df_bowl['balls_bowled'] / 6
df_bowl['econ']     = df_bowl['runs_conceded'] / df_bowl['overs']
df_bowl['bowl_avg'] = df_bowl['runs_conceded'] / df_bowl['wickets'].replace(0, n

# Build t→t+1 training set for wickets

def make_train(df, player_col, feat_cols, target_col):
    tmp = df.rename(columns={player_col:'player'}).sort_values(['player','season
    tmp['target'] = tmp.groupby('player')[target_col].shift(-1)
    tmp = tmp.dropna(subset=feat_cols+['target'])
    return tmp[feat_cols], tmp['target']

# Features we'll use to predict next-season wickets
features = ['wickets','econ','bowl_avg']
X, y = make_train(df_bowl, 'bowler', features, 'wickets')

# Impute any missing values
imp = SimpleImputer(strategy='mean')
X_imp = imp.fit_transform(X)

#  Train/test split
X_tr, X_te, y_tr, y_te = train_test_split(X_imp, y, test_size=0.2, random_state=

# Train XGBoost regressor
model = XGBRegressor(
    learning_rate=0.2,
    subsample=0.3,
    max_depth=3,
    n_estimators=200,
    random_state=42
)
model.fit(X_tr, y_tr)

# (Optional) Evaluate
print("Train RMSE:", np.sqrt(mean_squared_error(y_tr, model.predict(X_tr))))
print("Test  RMSE:", np.sqrt(mean_squared_error(y_te, model.predict(X_te))))

# Prepare next-season features
latest = df_bowl['season'].max()
next_season = df_bowl[df_bowl['season']==latest].reset_index(drop=True)
X_next = imp.transform(next_season[features])

# Predict next-season wickets
next_season['pred_wickets'] = model.predict(X_next)

# Select Top 5 bowlers
top5 = next_season.nlargest(10, 'pred_wickets')[['bowler','pred_wickets']]
print("\nPredicted Top 10 Bowlers Next Season:")
print(top5.to_string(index=False))
```

```
Train RMSE: 5.459701093807129
Test  RMSE: 5.4997607296805695

Predicted Top 10 Bowlers Next Season:
        bowler  pred_wickets
      HV Patel     18.328312
     YS Chahal     14.004087
    AD Russell     13.870964
Arshdeep Singh     13.870964
    Avesh Khan     13.870964
  Harshit Rana     13.870964
   T Natarajan     13.870964
     JJ Bumrah     13.760992
      KK Ahmed     12.471554
      MA Starc     12.471554
```

In [118…