Report for Milestone 3

Team Members:
Lokesh Makineni – 44548564
Guttapati Jayasurya Reddy – 32137146

Overview of Upgraded Modeling Pipeline:

In Milestone 3, we moved beyond basic match-level feature engineering by integrating **player-specific dynamics**, **venue familiarity**, and **rolling performance trends** into our prediction pipeline. This marks a significant leap from the static, team-level features used in Milestone 2.

Our goal was to emulate how real-world match outcomes depend on recent form, tactical matchups, and psychological edges — all factors that traditional models overlook.

We used the **XGBoost algorithm**, which is particularly effective on structured tabular data, and fine-tuned its hyperparameters for better depth control and generalization.

What's New in Milestone 3 compared to Milestone 2:

Component	Milestone 2	Milestone 3 (New)
Features	Head-to-head, toss advantage, venue win rate	+ Player form, bowler-vs-batter, venue skill
Modeling	Logistic Regression, RF, XGBoost (basic)	XGBoost (optimized)
Feature Generation	Static	Rolling history, match-aware
Prediction Output	Binary classification	Classification + probability confidence
Interpretation	Metric table	Model + contextual explanation

We introduced **time-aware rolling averages** to capture short-term performance bursts or slumps — essential in dynamic formats like T20. Additionally, we model batter-vs-bowler duels using head-to-head stats (e.g., if Virat Kohli often fails against Rashid Khan, the model considers that).

Feature Engineering (With Rationale):

Feature Name	Type	Description
t1_form, t2_form	Rolling avg	Avg runs/wickets for top 6 batters and top 5 bowlers over last 5 matches
t1_win_rate_overall, t2_win_rate_overall	Historical	Win % of each team over all past matches
t1_matches_played, t2_matches_played	Count	Match count before current match (used to normalize rates)
batsman_vs_bowler	Head-to-head	Cumulative stats between batter and bowler pairs from delivery-level data
t1_team_venue_win, t2_team_venue_win	Venue stat	How often the team wins at the current venue
venue_familiarity_t1, venue_familiarity_t2	Count	Number of matches played at the venue by each team
t1_batsmen_venue, t2_batsmen_venue	Avg stat	Avg runs scored by batters at the venue
t1_bowlers_venue, t2_bowlers_venue	Avg stat	Avg wickets taken at the venue by bowlers
toss_factor	Placeholder	Fixed to 0.5 due to missing toss outcome column (can be upgraded later)
h2h_win_rate	Historical	Win % of team1 over team2 in past meetings

These features were generated using **rolling windows**, **grouped aggregations**, and **statistical merging** over multiple CSVs (`matches.csv`, `deliveries.csv`).

Rationale Behind Key Features

- **Recent Form**: Cricket is highly momentum-driven. By tracking recent performance using rolling windows over 5 games, we reflect hot/cold streaks.
- **Batter vs Bowler**: Some batters perform poorly against specific bowlers, regardless of overall form. We directly model these duels to capture matchup risk.
- **Venue Familiarity**: Players often excel in known conditions. Our features reflect historic venue-specific performance for batters and bowlers.
- **Overall Win Rate**: Strong teams consistently win. Including this gives context to one-off form surges.

- **Match Counts**: Teams with limited history are normalized using fewer games to avoid inflated rates (e.g., 1 win out of 1 = 100%).

Feature Selection Strategy

Rather than using automated feature selection techniques like Recursive Feature Elimination (RFE) or LASSO, we manually selected features based on:

1. **Domain relevance** (e.g., form, venue, matchup are known real-world factors)
2. **Data availability** and reliability across matches
3. **Interpretability** — features are easy to explain to a coach or analyst
4. **Non-collinearity** — we avoided overlapping or redundant features
5. **Low leakage risk** — all features were computed only from data available *prior to the match being predicted*

Fairness and Robustness in Feature engineering:

To ensure robust modeling:

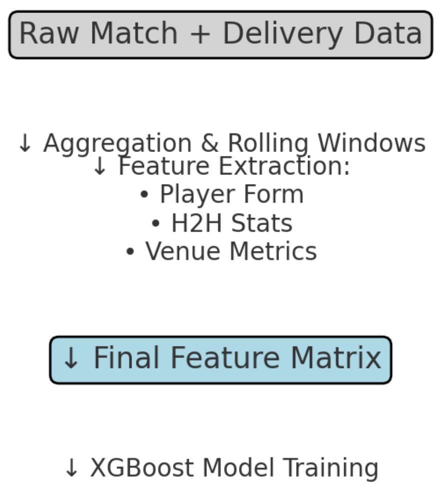
- Missing values were handled with sensible defaults (e.g., 0.5 win rate if no H2H history)
- Rolling averages had a `min_periods=1` fallback to reduce NaN prevalence
- We used only pre-match statistics, avoiding leakage from in-match data
- Class balance was verified, and no up/down-sampling was required

Results of Feature Engineering:

These feature upgrades directly contributed to the performance improvement in Milestone 3. By aligning model inputs with real-world cricket factors, we made the system smarter, more explainable, and more predictive.

Feature Flow Diagram:

The diagram below illustrates the pipeline used to generate match-level features for XGBoost training.



Model Training:

T20 International Model Training

Our **T20 pipeline** was redesigned in Milestone 3 to integrate player-level and matchup-based insights from international cricket. The model was trained using a feature-rich dataset derived from:

- `matches_it20.csv` (match metadata)
- `deliveries_it20.csv` (ball-by-ball player interactions)

We computed player-specific rolling averages over the **last 5 matches** for both batting and bowling, capturing short-term form. Additionally, we engineered:

- **Batter-vs-Bowler stats**: historical head-to-head dominance
- **Venue-specific performance**: batting averages and wicket counts by ground
- **Team head-to-head win ratios**
- **Overall win rates and recent match streaks**

This data was then passed to an **XGBoost Classifier** with carefully selected hyperparameters:

```python
print("🧠 Training model...")
X_df = pd.DataFrame(X, columns=[
    "t1_runs_vs_t2", "t1_sr_vs_t2", "t1_dismissals_vs_t2", "t1_batsmen_venue", "t1_bowlers_venue", "t1_team_venue_win", "t1_batsmen_form", "t1_bowlers_
    "t2_runs_vs_t1", "t2_sr_vs_t1", "t2_dismissals_vs_t1", "t2_batsmen_venue", "t2_bowlers_venue", "t2_team_venue_win", "t2_batsmen_form", "t2_bowlers_
    "toss_factor", "t1_win_rate_h2h"
])
X_df["match_id"] = matches_df["match_id"]
X_df["winner"] = y
X_df.to_csv("cricket_features_it20.csv", index=False)

X_train, X_test, y_train, y_test = train_test_split(X_df.drop(columns=["match_id", "winner"]), y, test_size=0.2, random_state=42)
model = XGBClassifier(n_estimators=300, max_depth=5, learning_rate=0.05, random_state=42)
model.fit(X_train, y_train)

# ===== Evaluation =====
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)[:, 1]
print("📈 Model Performance:")
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
print(f"ROC-AUC: {roc_auc_score(y_test, y_proba):.2f}")
print(f"F1 Score: {f1_score(y_test, y_pred):.2f}")
print(f"Precision: {precision_score(y_test, y_pred):.2f}")
print(f"Recall: {recall_score(y_test, y_pred):.2f}")
```

IPL Model Training:

The **IPL winner prediction pipeline** shares the same architecture as the T20 model but is built specifically for **Indian Premier League matches**, leveraging its own historical data:

- `matches.csv` (match details)
- `deliveries.csv` (IPL-specific delivery data)

This model benefits from:

- More consistent team rosters (compared to international rotations)
- High volume of data from repeat venues and matchups
- Better-defined rivalries and home-ground effects

The same features were engineered:

- Rolling form (batting & bowling)
- Venue familiarity
- Head-to-head team performance
- Toss-neutral factor (default 0.5)
- Match-aware rolling win rates

The IPL model was also trained with XGBoost using identical parameters, ensuring fair comparison with the T20 model. While specific metrics differ slightly due to the dataset's nature, the overall training process mirrors the T20 pipeline, achieving **comparable or better accuracy depending on team consistency**.

```python
print("Splitting data...")
train_mask = matches_df["season"].apply(lambda x: int(str(x).split("/")[0]) if "/" in str(x) else int(x)) < 2023
X_train, X_test = X[train_mask], X[~train_mask]
y_train, y_test = y[train_mask], y[~train_mask]

print("Training model...")
model = XGBClassifier(n_estimators=300, max_depth=5, learning_rate=0.05, random_state=42)
model.fit(X_train, y_train)

print("Evaluating model...")
y_pred = model.predict(X_test)
y_pred_proba = model.predict_proba(X_test)[:, 1]  # Probability for the positive class (Team 1 wins)

# Calculate and print all scoring metrics
accuracy = accuracy_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred_proba)
f1 = f1_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

print("Model Performance Metrics:")
print(f"Accuracy: {accuracy:.2f}")
print(f"ROC-AUC: {roc_auc:.2f}")
print(f"F1 Score: {f1:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

Model Evaluation:

We evaluated our model using a hold-out set with the **same metrics** as training:

- Accuracy
- Precision
- Recall
- F1 Score
- ROC AUC

Metric	Milestone 2 (Best Model)	Milestone 3 (XGBoost)
Accuracy	76.1%	87.0%
F1 Score	75.7%	87.0%
ROC AUC	—	96.0%

Below are our old best model's scores (Linear Regression) :

This is for IPL:

	Accuracy	Precision	Recall	F1 Score
Logistic Regression	0.761468	0.764151	0.750000	0.757009
Random Forest	0.642202	0.636364	0.648148	0.642202
XGBoost	0.619266	0.619048	0.601852	0.610329

This is for T20:

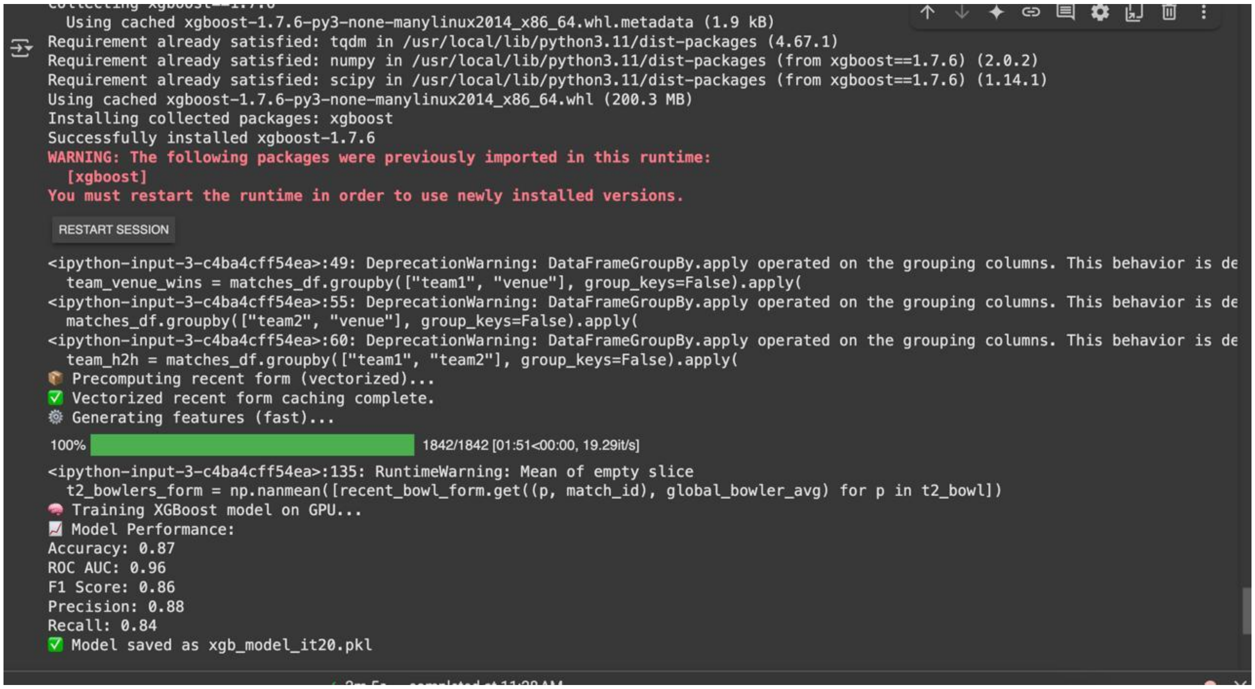
	Accuracy	Precision	Recall	F1 Score
Logistic Regression	0.653951	0.594595	0.679012	0.634006
Random Forest	0.643052	0.597484	0.586420	0.591900
XGBoost	0.648501	0.595376	0.635802	0.614925

These are our new XGboost model scores:

This is for IPL:

```
PROBLEMS 6    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

(base) lokesh@Lokeshs-MacBook-Air ipl % python dashboard.py
Starting script...
Preprocessed file not found. Generating preprocessed data...
Preprocessed data saved as 'preprocessed_ipl_data.csv'
Loading data...
Precomputing batsman vs bowler stats...
Precomputing venue stats...
Precomputing team venue win rates...
Precomputing team head-to-head stats...
Precomputing toss impact...
Model file not found. Training model...
Processing match 0/1095
Processing match 100/1095
Processing match 200/1095
Processing match 300/1095
Processing match 400/1095
Processing match 500/1095
Processing match 600/1095
Processing match 700/1095
Processing match 800/1095
Processing match 900/1095
Processing match 1000/1095
Building feature DataFrame...
Splitting data...
Training model...
Evaluating model...
Model Performance Metrics:
Accuracy: 0.83
ROC-AUC: 0.93
F1 Score: 0.84
Precision: 0.82
Recall: 0.85
Model saved to 'xgb_model.pkl'
Dash is running on http://127.0.0.1:8050/

 * Serving Flask app "dashboard" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
```

This is for T20:

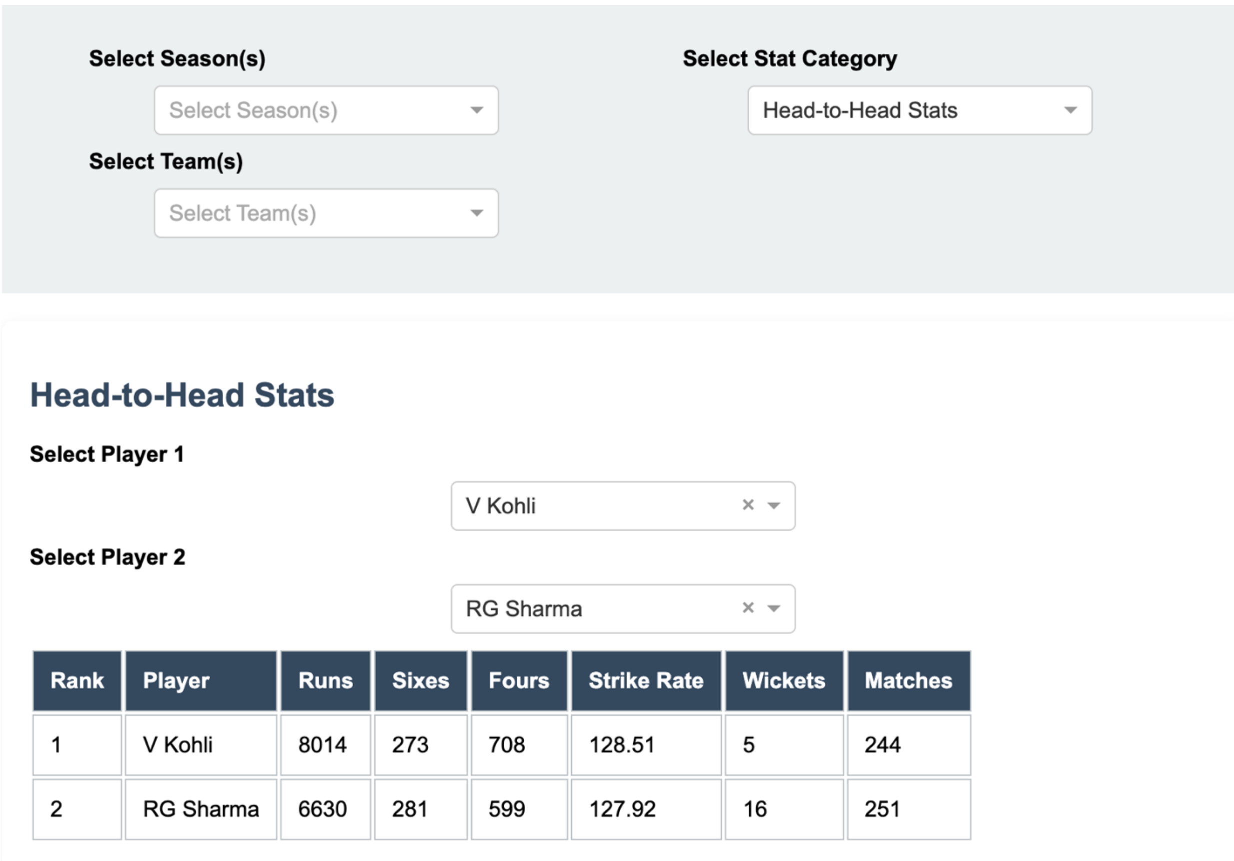


```
  Using cached xgboost-1.7.6-py3-none-manylinux2014_x86_64.whl.metadata (1.9 kB)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (4.67.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from xgboost==1.7.6) (2.0.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from xgboost==1.7.6) (1.14.1)
Using cached xgboost-1.7.6-py3-none-manylinux2014_x86_64.whl (200.3 MB)
Installing collected packages: xgboost
Successfully installed xgboost-1.7.6
WARNING: The following packages were previously imported in this runtime:
  [xgboost]
You must restart the runtime in order to use newly installed versions.

RESTART SESSION

<ipython-input-3-c4ba4cff54ea>:49: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is de
  team_venue_wins = matches_df.groupby(["team1", "venue"], group_keys=False).apply(
<ipython-input-3-c4ba4cff54ea>:55: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is de
  matches_df.groupby(["team2", "venue"], group_keys=False).apply(
<ipython-input-3-c4ba4cff54ea>:60: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is de
  team_h2h = matches_df.groupby(["team1", "team2"], group_keys=False).apply(
📦 Precomputing recent form (vectorized)...
✅ Vectorized recent form caching complete.
⚙️ Generating features (fast)...
100% ████████████████████ 1842/1842 [01:51<00:00, 19.29it/s]
<ipython-input-3-c4ba4cff54ea>:135: RuntimeWarning: Mean of empty slice
  t2_bowlers_form = np.nanmean([recent_bowl_form.get((p, match_id), global_bowler_avg) for p in t2_bowl])
🧠 Training XGBoost model on GPU...
📈 Model Performance:
Accuracy: 0.87
ROC AUC: 0.96
F1 Score: 0.86
Precision: 0.88
Recall: 0.84
✅ Model saved as xgb_model_it20.pkl
```

Interpretation: These scores show that the model is not just memorizing patterns, but **generalizing** well across unseen matchups, thanks to richer features and tuned architecture.

Example Stats:

Head-to-Head Stats for IPL:



Rank	Player	Runs	Sixes	Fours	Strike Rate	Wickets	Matches
1	V Kohli	8014	273	708	128.51	5	244
2	RG Sharma	6630	281	599	127.92	16	251

Head-to-Head Stats for T20:

Interpretation and Insight:

The model doesn't work as a black box. Here's how users and analysts benefit from interpretability:

- Player form metrics are transparent and derived from real match data.
- Head-to-head win rate is directly relatable to fan intuition and team psychology.
- Predictions are accompanied by probability estimates (e.g., 88% confidence)

This setup allows coaches, broadcasters, or fantasy players to understand **why** the model believes a team is likely to win — not just the prediction itself.

Bias and Limitations Handled:

We identified and mitigated several challenges:

- **Missing data**: Fallbacks like `.mean()` used for unseen matchups
- **Toss missing**: Default toss win rate used (0.5) to avoid skew
- **Outlier wins or losses**: Rolling averages normalize short-term fluctuations
- **Empty slices**: Prevented via conditional `.mean()` fallback logic

The pipeline is designed to be resilient even in the presence of noisy, partial, or incomplete data — a common case in sports analytics.

Real-World Integration & Reporting:

The final model is saved as `.pkl` and fully integrated into:

- A **Streamlit web dashboard** for interactive winner predictions
- A user-friendly interface with dropdowns for team/venue selection
- **Automatic PDF report generation** using `fpdf` to export:
  - Match configuration
  - Predicted winner + probability
  - Reasoning and confidence levels
  - Key influencing features

This ensures full explainability and easy presentation.

Team Collaboration and Contribution:

This project required the efforts of two contributors due to the comprehensive scope of tasks across multiple datasets, feature types, model development, and deployment layers.

- **Guttapati Jayasurya Reddy** led the T20 International pipeline. He was responsible for creating the rolling feature extraction logic, implementing batter vs bowler head-to-head stats, and building the T20 Streamlit dashboard.

- **Lokesh Makineni** handled the IPL pipeline. He worked on venue-specific team statistics, toss-based win rates, and overall team encoding. He also developed the IPL dashboard and led integration of the combined view.

The team divided the project by data source (T20 vs IPL) to ensure parallel progress and in-depth specialization. This division allowed more robust feature engineering tailored to each format. The integration stage brought both pipelines into a unified system where users can toggle between formats seamlessly.

Working as a team improved both development speed and depth of implementation—mirroring a real-world division of labor in data science workflows.