## 1 Model Development and Analysis

#### Exploratory Data Analysis (EDA):

##### Data Cleaning and Preparation

* The initial dataset contained 15,691 rows.
* During cleaning, 2 rows with missing values (NaN) were removed.
* Final dataset size after cleaning: 15,689 rows.

##### Class Distribution and Imbalance Handling

* Class distribution in the cleaned dataset (**15,689 rows**):
  + **Class 1 (Won Match):** 9,747 instances (**62%**)
  + **Class 0 (Lost Match):** 5,942 instances (**38%**)

To address the imbalance, the following strategies were used or evaluated:

1. **Class Weight Adjustment:**

Applied class\_weight='balanced' in scikit-learn models (e.g., Logistic Regression, Random Forest) to penalize minority class errors more heavily.

1. **Oversampling (SMOTE):**

Used **SMOTE** on the **training split only** to synthesize additional minority-class samples (not included in the shared production code but tested during experimentation).

1. **Undersampling:**

Randomly removed a portion of majority-class samples to further balance the classes (evaluated during experimentation)

#### 1.2 Model Training & Comparison:

##### Match‑Based Train–Validation Split:

* A naive row-wise train\_test\_split initially produced inflated accuracies (~97%) for tree-based models (Random Forest, XGBoost).
* **Root cause:** rows from the *same match* leaked into both train and validation sets, allowing models to effectively memorize match trajectories.
* **Solution:** split the data by complete matches using a match\_id, and stratify by the match-level outcome (won) so the win/loss balance is similar across splits.
* This approach ensures the validation set contains entirely unseen matches, yielding realistic generalization estimates.

##### Training Algorithms:

**Logistic Regression —** Baseline & Interpretability

* Serves as a **baseline** for performance and sanity‑checking linear signal.
* Coefficients provide direct interpretability of feature impact on win probability.

**Random Forest —** Robust Non‑Linear Learner

* Ensemble of decision trees that captures non‑linearities and feature interactions.
* More robust than a single tree and provides feature importance for insights.

**XGBoost — High‑Performance Gradient Boosting**

* Sequential boosting of trees that corrects prior errors, typically achieving top predictive accuracy.
* Efficient and widely successful in structured tabular problems.

**Support Vector Machine (SVM)**

* Evaluated as an additional strong baseline; comparable to Logistic Regression on the full dataset.

**SVM results (example best run):**

|  |  |  |
| --- | --- | --- |
| **Metric** | **Value** |  |
| Best Params | {'C': 0.1, 'gamma': 'scale', 'kernel': 'rbf'} |  |
| Accuracy | 0.813 |  |
| F1-score | 0.867 |  |
| AUC-PR | 0.841 |  |

##### Hyperparameter Tuning & Cross‑Validation

* Used **GridSearchCV (cv=5)** to search over predefined hyperparameter grids for each model.
* Cross‑validation is embedded within GridSearchCV, ensuring robust estimates per configuration.
* Example grids:
  + **Logistic Regression:** C, solver, class\_weight
  + **Random Forest:** n\_estimators, max\_depth, min\_samples\_split, class\_weight
  + **XGBoost:** n\_estimators, scale\_pos\_weight (and others explored as needed)

##### Model Evaluation Metrics and Interpretation

Because of class imbalance, we report multiple metrics to capture different aspects of performance:

* **Accuracy**
  + Fraction of correctly predicted samples.
* **F1‑Score**
  + Harmonic mean of **precision** and **recall**.
  + More reliable than accuracy in imbalanced settings because it penalizes extreme trade‑offs between precision and recall.
* **AUC-PR**
  + Plots **precision vs. recall** across different thresholds.
  + Especially informative for **imbalanced datasets**, where ROC-AUC can give an overly optimistic view.
  + A higher AUC-PR indicates better ability to detect the minority class while maintaining precision.

##### Model Evaluation on Validation set:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Accuracy | F1-Score | AUC-PR |
| Logistic Regression | 0.84 | 0.88 | 0.89 |
| Random Forest | 0.86 | 0.90 | 0.95 |
| XGBoost | 0.85 | 0.88 | 0.89 |

`

##### Limitations

1. Simplified Features

* The model currently uses only a few features (total\_runs, wickets, balls\_left, target).
* Real match outcomes depend on many more factors (pitch conditions, team strength, player form, toss result, etc.) which are not captured.

1. Data imbalance

* The dataset has an imbalanced target variable (won: ~62% vs ~38%).
* This can cause the model to favor the majority class, reducing fairness and accuracy for the minority outcome.

#### 1.3 Feature Engineering

During feature engineering, we explored adding four new features:

* required\_run\_rate
* total\_runs\_left
* balls\_per\_wicket
* run\_rate\_difference

The aim was to provide more domain-specific insights. However, these additions did not improve model performance.

The main reasons:

* **Multicollinearity & Redundancy**: The new features were simple combinations of existing ones (e.g., total\_runs\_left = target – total\_runs).
* **Model Strength:** Algorithms like RandomForest and XGBoost can already capture these derived relationships implicitly.

Thus, adding these engineered features offered no new non-redundant signal, leading to a plateau in performance.

## 2: Production API

We built a FastAPI app with a **/predict** POST endpoint. It takes the test dataset (cricket\_dataset\_test.csv), validates the required columns, filters the relevant rows, and makes predictions (with confidence scores if available).

##### Findings on Model Saving, Registry & Versioning:

1. **Model Development**

* Trained **Logistic Regression, Random Forest,** and **XGBoost** using a **match-based train-test split** to prevent leakage.
* Each trained model was saved with a versioned filename (e.g., random\_forest\_v1.pkl), ensuring older versions remain available.

1. **Model Versioning**

* Models are stored with version numbers (v1, v2, v3).
* Each version is documented with:
  + Training data used
  + Hyperparameters and techniques (e.g., SMOTE, class weights)
  + Validation metrics (Accuracy, F1, AUC-PR).

1. **Model Registry**

* A **registry.json** file logs all trained models, their versions, and performance.
* This provides a single source of truth for comparing models.

1. **Active Model**

* The model with the highestAUC-PRscore is promoted as the active model.
* Its details are stored in active\_model.json, making it the one used for deployment.
* Older active models are archived for rollback if needed.

1. **API Integration**

* The API automatically loads the active model from active\_model.json.
* All /predict requests run through this model, ensuring predictions always use the best available version.

##### Error Handling

**Common Errors:**

* Malformed CSV → "Missing required column: target"
* No Rows Matching Filter → "No rows matched filter condition: balls\_left < 60 and target > 120"
* Model Not Found → "Active model not set. Please train a model first."
* Invalid Prediction ID → "Prediction ID 99 not found in results.csv"

All errors return HTTP 400 with a descriptive message.

##### Logging

* Prediction requests, errors, and explanations are logged.
* Logging helps monitor model drift and track prediction reliability.

## 3: Prompt Engineering:

* We added a new FastAPI endpoint /explain/{prediction\_id} that uses the gemini-2.5-flash-lite LLM to generate human-readable explanations of predictions.
* It loads the saved results.csv, retrieves the selected row, and passes the match context (runs, wickets, target, balls left, prediction, and confidence) into a structured prompt.
* The model then returns a concise (≤5 sentences) explanation tailored by confidence levels: high (>0.65), medium (0.5–0.65), or low (<0.5).
* The endpoint responds with prediction, confidence, and natural language explanation.
* Prompt: “You are a cricket match prediction assistant.

A machine learning model predicted the following

Prediction: {row['prediction']}

Confidence: {row['confidence']:.2f}

Match context:

Runs Scored: {row['total\_runs']}

Wickets Lost: {row['wickets']}

Target: {row['target']}

Balls Left: {row['balls\_left']}

Explain the prediction in clear, human terms:

If confidence is high (>0.65), emphasize certainty and key drivers.

If confidence is medium (0.5–0.65), explain uncertainty and influencing factors.

If confidence is low (<0.5), emphasize unpredictability and risk factors.

Make it concise but insightful (max 5 sentences)”

## 4: Testing

**Summary: Integration Test with Pytest**

* **File:** integration\_pytest.py
* **Purpose:** End-to-end integration test of the cricket training pipeline using pytest.

**What It Does**

1. Fixture (sample\_dataset)
   * Creates a synthetic cricket dataset (match\_id, balls\_left, total\_runs, wickets, target, won).
   * Saves it as a temporary CSV file for testing.
2. Integration Test (test\_integration\_train\_and\_evaluate)
   * Runs the full pipeline (train\_and\_evaluate) on the synthetic dataset.
   * Stores models, predictions, and registry files in a temporary directory.

**Key Assertions**

* Results Validity
  + Returns a dictionary containing metrics for Logistic, RandomForest, and XGBoost.
  + Metrics (accuracy, f1, auc\_pr) are in [0,1].
* Artifacts Saved
  + registry.json and active\_model.json exist.
  + At least one trained .pkl model is saved.
  + Validation prediction CSVs (\*\_val\_predictions.csv) are saved for each model (3 total).

This test validates the entire ML pipeline (data → training → evaluation → model saving → predictions), ensuring all critical outputs are generated correctly.