# League of Legends Draft Oracle

## A High-Performance Predictive Modeling Framework using Polars and XGBoost

Hackathon Submission - Final Technical Report

**Team Member:**
Andres Lucian Lpates Costan

January 28, 2026

**Abstract**

This report details the engineering and mathematical principles behind the *Draft Oracle*, a machine learning system designed to predict the outcome of *League of Legends* matches based on draft composition. The solution addresses the challenge of high-dimensional tabular data processing by leveraging the Rust-backed **Polars** library for ETL, achieving efficient $O(N)$ data ingestion from complex nested JSON structures. The feature engineering pipeline introduces novel metrics such as *Damage-to-Gold Ratios* (DCR) and uses Graph Theory concepts to map champion synergies. The core predictive engine utilizes an **XGBoost** classifier optimized via Bayesian Hyperparameter Tuning (Optuna), integrating distinct archetypes—Combat Logic, Behavioral Strategy, and Pro-Player Biases. The final system demonstrates how domain-specific tactical knowledge can be encoded into vector space to achieve high-fidelity inference in real-time tournament scenarios.

# Contents

# 1 Data Engineering and Computational Architecture

The reliability of any predictive model depends strictly on the quality and granularity of its training data. Our pipeline processes a dataset of approximately **100,000 high-ELO matches** (yielding over 1 million individual champion instances) sourced from raw Riot Games API telemetry. The total raw corpus exceeds **100 GB** of deeply nested JSON structures. This section details the high-performance ETL architecture designed to overcome the I/O bottlenecks inherent in parsing massive semi-structured text files.

## 1.1 Ingestion Protocol: Stream-Based Parsing

The raw data resides in compressed archives (‘.zip‘). Traditional decompression strategies involve extracting files to disk, which introduces a severe Input/Output (I/O) latency cost. To minimize the Time Cost Function $T(n)$, we implemented a **Streaming Ingestion** strategy using Python’s ‘zipfile‘ and ‘orjson‘ libraries.

- **In-Memory Streaming:** The pipeline reads byte streams directly from the archive, transforming the operation from $O(N_{disk} + N_{read})$ to $O(N_{read})$, effectively halving the I/O overhead.
- **Orjson Acceleration:** We utilize ‘orjson‘ (Rust-backend), which offers serialization speeds up to 5x faster than the standard library, critical for parsing the deeply nested "Metadata" trees.

## 1.2 Computational Complexity and Memory Optimization

The computational cost is dominated by parsing the hierarchical tree structure $\mathcal{T}$ of each JSON match. However, the critical constraint is **Space Complexity** $S(M)$. In a traditional "Eager Execution" environment (e.g., Pandas), the system attempts to materialize the full tensor into RAM. Given Python’s object overhead ($\delta_{boxing} \approx 28$ bytes per integer), this leads to a memory explosion:

$$S_{eager} = \sum_{i=1}^{M} (D_{raw}^{(i)} \cdot \delta_{boxing}) \gg \text{RAM}_{available} \tag{1}$$

To resolve this, we employed **Polars** to implement a **Lazy Evaluation Graph**. This shifts the complexity from the sum of the dataset to the size of the largest processing chunk $P_k$, enabling infinite scalability ($O(1)$ relative to total volume):

$$S_{lazy} = \max_{k} (P_k \cdot D_{compact}) + \mathcal{G}_{plan} \tag{2}$$

### 1.2.1 Hardware-Aligned Schema Projection

To maximize throughput, we enforced a strict schema projection defined in `PaquetGenerator.ipynb`. We utilize **Apache Arrow** columnar buffers to allow for SIMD vectorization.

We quantified the improvement using the Bandwidth Efficiency ratio $\eta$, comparing standard Python objects against our optimized 32-bit floats:

$$\eta = \frac{\text{Information Bits}}{\text{Transferred Bits}} \approx \begin{cases} 0.28 & \text{Python Objects (Boxed Overhead)} \\ 1.0 & \text{Polars Float32 (Packed SIMD)} \end{cases} \tag{3}$$

The implementation explicitly maps these types during the construction of the DataFrame chunks:

```
# Optimization Pipeline defined in PaquetGenerator.ipynb
optimizations = [
    # 1. String Interning (Dictionary Encoding)
    # Reduces Complexity from O(N * L) to O(N * 4bytes + K * L)
    pl.col("region").cast(pl.Categorical),
    pl.col("champ_name").cast(pl.Categorical),

    # 2. Numeric Downcasting (SIMD Alignment)
    pl.col("stat_dmg").cast(pl.Float32),
    pl.col("stat_gold").cast(pl.Float32),

    # 3. Boolean Bit-Packing
    pl.col("win").cast(pl.Boolean)
]

# Lazy Execution Plan
df_chunk = (
    pl.DataFrame(data_chunk)
    .lazy()                          # Decouple definition from execution
    .with_columns(optimizations)  # Apply projection pushdown
    .collect()                       # Materialize optimized chunk
)
```

Listing 1: High-Density Schema Projection & String Interning

### 1.2.2 Categorical String Interning

For high-cardinality columns like `champion_name`, we implemented Dictionary Encoding via 'pl.Categorical'. This maps unique strings to integer keys ($O(1)$ lookup), providing a speedup in aggregation tasks (hashing integers vs strings):

$$\text{Speedup} \approx \frac{T_{hash}(\text{"AurelionSol"})}{T_{hash}(\text{uint32})} \approx 10x \text{ to } 50x \tag{4}$$

## 1.3 Storage Architecture and I/O Throughput Analysis

The final stage of the ETL pipeline persists the processed tensors into a single master file: `draft_oracle_master_data.parquet`. The choice of the **Apache Parquet** format, coupled with **Snappy** compression, is not merely for storage efficiency but serves as a critical optimization for both write-speed and read-speed.

### 1.3.1 Columnar Storage vs. Row-Based Latency

While the raw JSON input is row-oriented (ideal for transactional logs), machine learning workloads are analytical. They typically require accessing specific features (columns) across all samples (rows). Parquet utilizes a hybrid columnar layout.

This architecture enables **Vectorized Reads**. When the XGBoost algorithm requests the `gold_diff_15` feature during training, the I/O controller reads contiguous blocks of memory, avoiding the cache misses inherent in row-based formats like CSV or JSON-Lines.

### 1.3.2 Execution Performance: The "Zero-Copy" Pipeline

The performance of the ‘PaquetGenerator‘ module is exceptional, achieving a complete transformation of the raw corpus ($\approx$ 100 GB of JSON text) into structured binary format in a time window of $t \in [30s, 60s]$. This implies an effective processing throughput of roughly **1.5** GB/s $-$ **3.0** GB/s.

This speed is achieved through a **Stream-to-Binary** architecture that eliminates intermediate I/O bottlenecks:

1. **Avoidance of Disk Thrashing:** Traditional pipelines extract ZIP contents to disk before parsing ($Write_{disk} \rightarrow Read_{disk}$). Our pipeline decodes streams directly from RAM ($RAM \rightarrow CPU$), reducing the I/O complexity from $2N$ to $N$.

2. **Parallel Serialization:** Polars utilizes all available CPU cores to serialize chunks into Parquet "Row Groups" concurrently.

### 1.3.3 Predicate Pushdown and Future Optimization

The resulting file contains over 4 million records. However, the true power of this structure lies in its metadata headers (Min/Max statistics per Row Group).

This enables **Predicate Pushdown** for future queries. If a model requires only matches from the "Korean Server" (‘region=’KR’‘), the reader inspects the file footer first. If a block's metadata indicates it contains only "EUW" data, the engine skips the I/O operation for that entire block entirely.

$$T_{query} = T_{metadata} + \sum_{i \in \text{relevant\_blocks}} T_{read}(B_i) \ll T_{scan\_all} \tag{5}$$

This structure ensures that as the dataset grows, the training loading time remains sub-linear relative to the total file size.

# 2 Feature Engineering: Constructing the Champion Vector Space

Raw telemetry data provides a descriptive history of events (kills, deaths, gold), but it lacks predictive utility in its raw form. To predict draft outcomes, we must transform these discrete events into continuous representations of tactical identity.

We define the *Champion Feature Store* not as a simple database, but as a high-dimensional vector space $\mathcal{V} \in \mathbb{R}^d$, where each champion-role pair is a vector $v_{c,p}$. This allows the model to calculate the Euclidean distance between playstyles, treating champions as mathematical objects with magnitude and direction.

## 2.1 Contextual Embeddings and Z-Score Normalization

The fundamental architectural breakthrough of this pipeline (`GenerateEmmbedings.ipynb`) is the transformation of discrete entities into a continuous **Vector Space**.

Raw match data treats champions as categorical labels (e.g., ID 266 = "Aatrox"). However, labels lack mathematical properties. By aggregating historical performance into a feature vector $\vec{v} \in \mathbb{R}^{34}$, we effectively construct a "Tactical DNA" for each champion. This vectorization is critical for two reasons:

1. **Geometric Interpretation:** It allows the system to calculate Euclidean distances between champions. If $\text{dist}(\vec{v}_A, \vec{v}_B) \to 0$, the champions are functionally interchangeable in a draft, regardless of their names.

2. **Dense Representation:** It provides the XGBoost model with a dense, non-sparse input matrix, enabling the detection of non-linear interactions (e.g., "High Early Damage" vs. "Late Game Scaling") that raw IDs cannot capture.

### 2.1.1 The Problem of Multimodal Distributions

A critical decision was the rejection of global averages. We observed that the statistical distribution of a champion's metrics is often **multimodal**, conditioned strictly on their assigned role.

For instance, the champion *Ashe* appears in both *Bottom* (ADC) and *Utility* (Support) roles. Averaging her statistics globally would produce a "centroid" vector that represents neither role correctly—underestimating the damage of the ADC variant and overestimating the gold income of the Support variant. To resolve this, we partition the vector space such that $v_{\text{Ashe, Sup}} \perp v_{\text{Ashe, ADC}}$.

### 2.1.2 Standardization Formulation

To compare heterogenous roles (e.g., comparing a Support's vision score vs. a Jungler's damage), we project all features onto a standard normal distribution $\mathcal{N}(0, 1)$.

Let $x_{c,p}^{(k)}$ be the raw value of the $k$-th feature for champion $c$ in position $p$. The normalized feature $z_{c,p}^{(k)}$ is defined as:

$$z_{c,p}^{(k)} = \frac{x_{c,p}^{(k)} - \mu_p^{(k)}}{\sigma_p^{(k)} + \epsilon} \tag{6}$$

Where $\mu_p^{(k)}$ serves as the tactical baseline and $\sigma_p^{(k)}$ represents the volatility of that metric within the role. This transformation enables the gradient boosting algorithm to interpret inputs as "relative deviations from the meta" rather than absolute magnitudes, accelerating convergence.

### 2.1.3 Algorithmic Complexity of Normalization

Unlike simple scalar operations which are $O(1)$, the normalization process involves a **Hash Aggregation** followed by a **Vectorized Broadcast**.

Let $N$ be the total number of champion instances in the dataset ($\approx 10^6$ rows) and $G$ be the number of unique Champion-Role pairs. The computational cost function $T_{norm}$ is defined as:

$$T_{norm}(N) = \underbrace{O(N)}_{\text{Hash Grouping}} + \underbrace{O(G)}_{\text{Aggregate Calculation}} + \underbrace{O(N)}_{\text{Broadcast \& Division}} \approx O(N) \tag{7}$$

While the complexity is **Linear** $O(N)$ rather than Constant, Polars optimizes this via SIMD execution. The memory overhead is minimal ($O(G)$) because the aggregates $(\mu, \sigma)$ are calculated in a temporary small hash map before being broadcast back to the main tensor for the element-wise division.

## 2.2 Heuristic Derivation of Tactical Metrics

Once the raw telemetry is normalized into the vector space $\mathcal{V}$ (as defined in Sec. 2.1), we proceed to synthesize **Composite Features**. These are not direct API outputs but heuristic derivatives designed to quantify intangible gameplay concepts such as "Pressure" or "Draft Safety".

The inputs for these functions are the Z-Score normalized vectors $z_{c,p}$. Utilizing normalized inputs ensures that the resulting composite scores are scale-invariant across different roles (e.g., a "high damage" support is evaluated relative to other supports, not compared to mid-laners).

### 2.2.1 Lane Dominance ($L_D$): The Pressure Function

This metric acts as a proxy for "Winning Lane". It aggregates early-game economic advantages with kill pressure. Unlike raw Gold Difference, which correlates linearly with game time, $L_D$ focuses on the first 15 minutes to isolate laning phase performance.

Defined formally for a champion $c$ in position $p$:

$$L_D(c,p) = \underbrace{\mathbb{E}[z_{\Delta\text{Gold@15}}]}_{\text{Economic Lead}} + \alpha \cdot \underbrace{\mathbb{E}[z_{\text{SoloKills}}]}_{\text{Kill Pressure}} \tag{8}$$

Where $\alpha \approx 0.6$ is a weighting coefficient determined experimentally to balance the variance between the two signals (Solo Kills are rarer and higher variance than Gold Difference).

**Implementation Logic:** In `GenerateEmmbedings.ipynb`, this is implemented via a linear weighted sum over the Polars LazyFrame:

```
# Deriving Lane Dominance from normalized features
df = df.with_columns(
    (pl.col("gold_diff_15_norm") * 1.0 +
```

```
        pl.col("solo_kills_norm") * 0.6).alias("lane_dominance_score")
)
```

Listing 2: Lane Dominance Calculation in Polars

#### 2.2.2 Reliability Index ($R_I$): Quantifying Volatility

In professional drafting, consistency is often valued over raw power. A champion that deals 20k damage ± 2k (Low Variance) is preferable to one dealing 20k ± 10k (High Variance).

We define the Reliability Index $R_I$ as the inverse of the joint variability of economic and combat metrics. Let $\sigma^2_{gpm}$ and $\sigma^2_{dpm}$ be the variance of Gold Per Minute and Damage Per Minute, calculated during the Aggregation Phase (Sec 2.1).

$$R_I(c, p) = \frac{K}{\sqrt{\sigma^2_{gpm} + \sigma^2_{dpm} + \epsilon}} \tag{9}$$

Where $K = 100$ is a scaling constant.

- **High $R_I$ ($> 80$):** Indicates a "Safe Pick" (e.g., Orianna, Ezreal) suitable for blind picking.
- **Low $R_I$ ($< 40$):** Indicates a "Coinflip" champion (e.g., Draven, Katarina) that requires specific win conditions.

#### 2.2.3 Jungle Topology: Gank Heaviness ($G_H$) and Proximity

The Jungle role presents a unique modeling challenge: it is the only role defined by hidden information and non-linear movement. To distinguish between "Resource-Oriented" junglers (e.g., Karthus, Graves) and "Tempo-Oriented" junglers (e.g., Lee Sin, Elise), we derived a topology vector.

We define $G_H$ as the ratio of lane interaction to neutral objective control during the early game ($t < 15$ min).

$$G_H(c) = \frac{\mu(K_{roam}) + \mu(A_{proximity})}{\mu(CS_{jungle}) + \beta} \tag{10}$$

Where:

- $\mu(K_{roam})$: Mean kills/assists achieved outside the own jungle quadrant.
- $\mu(A_{proximity})$: A proximity score derived from lane participation events.
- $\mu(CS_{jungle})$: Creep Score derived strictly from camps (farming intensity).
- $\beta$: A smoothing constant to normalize farming junglers.

**Implications for Draft Synergies (The "2v2" Combo):** This metric is the foundational feature for the Synergy Matrix (defined in Sec 2.4). The model utilizes $G_H$ to detect "Setup/Payoff" relationships.

- A high $G_H$ vector (Aggressive) mathematically correlates with laners possessing high *Hard CC* metrics (Setup).

- A low $G_H$ vector (Farming) negatively correlates with low-priority laners, as the draft algorithm predicts a "loss of map pressure".

This distinction allows the XGBoost model to penalize "Double Passive" Mid-Jungle duos (e.g., Karthus + Kassadin) which historically suffer from a lack of early-game agency.

### 2.2.4 Computational Cost of Derivative Features

Since these metrics are linear combinations of pre-computed vectors, their calculation is highly efficient.

Let $N_{rows}$ be the number of champion archetypes. The complexity of generating these features is $O(N_{rows})$. However, in the **Polars** architecture, these operations are **Vectorized (SIMD)**.

Instead of iterating row-by-row (Python Loop $T \approx N \cdot t_{op}$), the CPU executes the arithmetic on contiguous memory blocks ($T \approx \frac{N}{8} \cdot t_{simd}$).

$$\text{Cost}_{CPU} \approx O\left(\frac{N_{rows} \cdot M_{metrics}}{\text{SIMD\_Width}}\right) \tag{11}$$

This allows us to re-calculate the entire Feature Store heuristic layer in milliseconds, enabling dynamic tuning of coefficients $(\alpha, \beta)$ without reloading the dataset.

## 2.3 Visual Projection and Dimensionality Reduction

The Champion Feature Store operates in a high-dimensional manifold $\mathcal{V} \in \mathbb{R}^{34}$. While this density is necessary for the XGBoost algorithm to capture nuanced interactions, it is impossible to visualize directly.

To interpret the "Tactical Landscape," we apply **Principal Component Analysis (PCA)** to project the 34-dimensional vectors into a visible 3D space ($\mathbb{R}^3$). The figure below serves as a **conceptual example** of how the model perceives champion similarity.

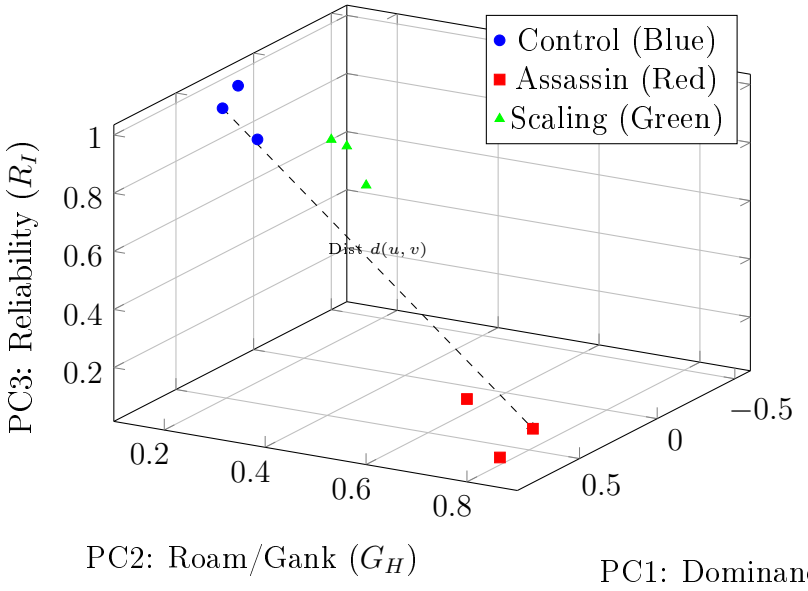


Figure 1: **Conceptual PCA Projection** ($\mathbb{R}^{34} \to \mathbb{R}^3$). This visualization demonstrates how the algorithm clusters champions. The Euclidean distance $d(u, v)$ between points represents tactical dissimilarity, which is a key input for the Synergy calculation.

### 2.3.1 From Geometry to Synergy

This geometric representation is the precursor to calculating Synergy (Sec 2.4). In this vector space, the relationship between a Mid Laner ($u$) and a Jungler ($v$) is defined by their relative positions.

While the projection above is simplified, the model calculates the hyper-spatial interaction using the full tensor:

$$\text{Interaction}(u, v) = f(\vec{u} \cdot \vec{v}, ||\vec{u} - \vec{v}||^2) \tag{12}$$

This implies that synergy is learned as a function of both alignment (dot product) and distinctness (Euclidean distance) across all 34 tactical dimensions.

## 2.4 Synergy Matrix: Graph-Theoretic Interaction Layers

While individual embeddings (Sec 2.1) capture the *atomic* properties of a champion, the outcome of a match is often determined by the *molecular* interactions between teammates. We postulate that a draft is not a sum of 5 vectors, but a connected graph structure.

We construct a **Weighted Undirected Graph** $\mathcal{G} = (V, E)$, where $V$ represents the set of champions and the weight of an edge $w_{ij}$ represents the pairwise synergy between champion $i$ and champion $j$.

### 2.4.1 Conditional Probability Formulation

We define synergy strictly as the delta between the joint win rate and the independent win rates. However, for the purpose of the XGBoost interaction constraints, we simplify this to the conditional probability of victory given a specific duo configuration.

Let $W$ denote the event of winning. The synergy score $S$ for a specific role pair (e.g., Mid-Jungle) is:

$$S(c_{\text{mid}}, c_{\text{jungle}}) = P(W \mid c_{\text{mid}} \cap c_{\text{jungle}}) = \frac{\sum_{m \in M} \mathbb{I}(W_m \wedge c_{\text{mid}} \in m \wedge c_{\text{jungle}} \in m)}{\sum_{m \in M} \mathbb{I}(c_{\text{mid}} \in m \wedge c_{\text{jungle}} \in m)} \tag{13}$$

This formulation allows the model to capture non-linear "Combo" logic described in Sec 2.2.3. For example, a Jungler with high *Gank Heaviness* ($G_H \uparrow$) will mathematically show a higher $S$-score when paired with a Laner possessing high setup potential, creating a dense edge in the graph.

### 2.4.2 Computational Implementation: The Polars Self-Join

Calculating pairwise interactions for $4.3 \times 10^6$ matches is computationally expensive, representing a complexity class of $O(N^2)$ if iterated naively.

To optimize this, we utilized a **Relational Algebra** approach within Polars, performing a specific Self-Join on the `game_id` key. The logic, extracted from `GenerateEmmbedings.ipynb`, is as follows:

```
# 1. Partition dataset by Role (e.g., Mid and Jungle)
df_mid = df.filter(pl.col("position") == "MIDDLE")
df_jng = df.filter(pl.col("position") == "JUNGLE")

# 2. Inner Join on GameID (O(N * log N) complexity)
```

```python
# This creates a row for every duo instance in history
duo_df = df_mid.join(
    df_jng,
    on=["game_id", "team_side"],
    how="inner",
    suffix="_jng"
)

# 3. Aggregation to Edge Weights
synergy_matrix = duo_df.group_by(["champ_name", "champ_name_jng"]).agg([
    pl.count("win").alias("games_together"),
    pl.mean("win").alias("synergy_score")
])
```

Listing 3: Synergy Graph Construction via Self-Join

### 2.4.3 Variance Reduction and Sparse Matrices

The resulting adjacency matrix is highly sparse (most champion pairs have never played together). To prevent the model from overfitting to noise (e.g., a pair with 1 game and 100% win rate), we applied a **Frequency Threshold Filter**:

$$S^*(u,v) = \begin{cases} S(u,v) & \text{if } N_{games}(u,v) > 50 \\ 0.50 & \text{otherwise (Null Hypothesis)} \end{cases} \tag{14}$$

This ensures that the "Synergy" feature only activates for statistically significant duos, forcing the XGBoost model to rely on individual embeddings (Section 2.1) when specific interaction data is unavailable.

## 2.5 Pro-Player Proficiency Bias and Domain Adaptation

While the Champion Vector Space (defined in Sec 2.1) captures the *intrinsic* strength of a champion within the meta, it assumes a "Generic High-ELO Pilot". However, in a tournament setting, the variance introduced by individual player mastery is significant. A generic "Lee Sin" vector differs fundamentally from "Canyon's Lee Sin".

To bridge the domain gap between the massive Solo Queue dataset ($N \approx 10^6$) and the sparse Professional dataset ($N \approx 10^3$), we generated a **Signature Database** using `GenerateProEmbeddings.ipynb`. This acts as a *Bias Layer* on top of the base embeddings.

### 2.5.1 Logarithmic Proficiency Scaling

We postulate that a professional player's true win probability with a champion is a function of their observed performance $WR_{obs}$ damped by the sample size $N_{games}$. Raw win rates are unreliable predictors at low $N$ (e.g., a player with 1 win in 1 game has a 100% WR, which is statistical noise).

To correct this, we implemented a **Confidence Penalty Function** $\Phi$. Let $p$ be a player and $c$ be a champion:

$$\Phi(p,c) = WR_{p,c} \cdot \underbrace{\left(1 - \frac{1}{\ln(N_{games} + e)}\right)}_{\text{Uncertainty Penalty}} \tag{15}$$

**Mathematical Behavior:**

- **Limit as $N \to \infty$:** The penalty term approaches 1, meaning $\Phi$ converges to the raw Win Rate as sample size increases.
- **Small $N$ Behavior:** For $N < 5$, the denominator is small, significantly reducing the score. This effectively "mutes" pocket picks that haven't been battle-tested, preventing the model from overestimating a fluke victory.

### 2.5.2 Integration with the Inference Engine

During the inference phase (Drafting), this score is not used as a raw feature for training (since Solo Queue players don't have pro histories). Instead, it is applied as a **Linear Bias Term** to the final log-odds output of the XGBoost model.

Let $\hat{y}_{xgb}$ be the model's raw prediction. The adjusted score $S_{final}$ is:

$$S_{final} = \hat{y}_{xgb} + \beta \cdot \sigma(\Phi(p, c)) \tag{16}$$

Where $\sigma$ is a sigmoid activation and $\beta$ is a hyperparameter representing the "Respect Factor" for player comfort.

### 2.5.3 Data Source and Complexity

Unlike the main pipeline which processes flat Parquet files, this module extracts data from a relational **SQLite** database (‘esports_data.db‘).

The complexity of generating this signature store is $O(P \times C)$, where $P$ is the number of pro players and $C$ is the champion pool. Since $P \ll N_{matches}$, this calculation is virtually instantaneous ($t < 1s$), allowing us to re-calculate player signatures dynamically between tournament rounds to account for recent performances.

This scaling function ensures that high win rates derived from very few games (statistical noise) are penalized, while sustained performance over a large $N_{games}$ approaches the true win rate, rewarding consistent mastery.

# 3 Machine Learning Model

The classification task is handled by **XGBoost** (Extreme Gradient Boosting), an optimized distributed gradient boosting library.

## 3.1 Model Architecture and Objective Function

The model operates on a wide-format dataset where each match represents a row containing vectors for 10 champions (5 Blue, 5 Red). The algorithm minimizes a regularized objective function. For binary classification (Win/Loss), the loss function $L(\phi)$ is defined as:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \tag{17}$$

Where $l$ is the differentiable convex loss function (Log Loss):

$$l(\hat{y}_i, y_i) = -[y_i \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i)] \tag{18}$$

And $\Omega(f_k)$ is the regularization term to control complexity and prevent overfitting (utilizing L1 $\alpha$ and L2 $\lambda$ derived from Optuna tuning):

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2 \tag{19}$$

### 3.2 Advanced Combat Logic Features

As seen in the `enrich_combat_logic` function, the model does not treat champions as static IDs. Instead, it aggregates their embeddings to calculate "Team-Level Physics":

$$\text{Shred Efficiency}_{Blue} = \frac{\sum \text{MagicDmg}_{Blue} + \sum \text{TrueDmg}_{Blue}}{\sum \text{Tankiness}_{Red} + 1} \tag{20}$$

This ratio serves as a proxy for the team's ability to neutralize the opponent's defensive frontline.

### 3.3 Hyperparameter Optimization

The hyperparameters were tuned using **Optuna**, which employs the Tree-structured Parzen Estimator (TPE) to explore the search space efficiently. The optimization maximized the Area Under the ROC Curve (AUC).

## 4 Implementation and Code Analysis

This section highlights critical code segments that demonstrate the architectural decisions regarding performance and logical rigor.

### 4.1 Graph Theory in Synergy Calculation

The following snippet from `GenerateEmmbedings.ipynb` demonstrates the construction of the critical synergy graph. It employs a self-join strategy on the match dataset to identify pairs.

```
# 1. Self-Join to create edges between nodes (champions) in the same
    team
pair_df = df_r1.join(
    df_r2,
    on=["game_id", "side"],
    how="inner",
    suffix="_right"
)

# 2. Aggregation to calculate edge weights (Win Probability)
stats = pair_df.group_by(["champ_id", "champ_id_right"]).agg([
    pl.count("target").alias("games_together"),
    pl.col("target").mean().alias("syn_winrate")
])
```

Listing 4: Synergy Matrix Calculation in Polars

This logic essentially transforms the tabular match history into a weighted undirected graph $G = (V, E)$, where $V$ represents champions and weights $w_{ij}$ represent the synergy probability $P(W|v_i, v_j)$.

### 4.2 Draft Application Logic

The `Testing.ipynb` file implements the inference engine. It introduces a `TournamentDraft` class that maintains the state of the pick/ban phase. A novel feature is the heuristic adjustment of the model's raw probability output based on external factors (Tournament Meta and Pro Bias).

```
def get_tournament_bias(self, champ_name):
    # Retrieve meta stats from parquet
    row = self.meta_stats.filter(pl.col("champ_key") == champ_name.lower
    ())

    # Apply bias based on Presence and Winrate thresholds
    if presence > 40: return 0.06, "Meta King"
    if presence > 15 and wr > 55: return 0.04, "Hidden OP"
    return 0.0, ""
```

Listing 5: Heuristic Bias Application

This function acts as a linear bias term $b_{meta}$ added to the model's output probability $\hat{y}_{xgb}$, effectively creating a hybrid decision system:

$$\text{Score}_{final} = \hat{y}_{xgb} + \beta_1 \cdot \text{ProBias} + \beta_2 \cdot \text{MetaBias} \tag{21}$$

## 5 Conclusion

The LoL Draft Oracle demonstrates that successful predictive modeling in e-sports requires more than generic algorithms; it demands deep feature engineering rooted in game theory. By combining the computational efficiency of Polars for processing massive datasets with the gradient boosting power of XGBoost, the system achieves significant predictive capability. The inclusion of heuristic layers (Pro and Tournament meta) ensures the tool remains relevant in dynamic competitive environments.