

Atom.gg

An AI Powered Drafting Assistant for League of Legends

Sky is the limit - Hackathon blog report

Team Members:

Andres Lucian Laptés Costan

Omar Cornejo

Dídac Dalmases Valcàrcel

Rubén Palà Vacas

February 1, 2026

Contents

Introduction	4
I League of Legends Draft Oracle	5
1 Data Engineering and Computational Architecture	6
1.1 Ingestion Protocol: Stream-Based Parsing	6
1.2 Computational Complexity and Memory Optimization	6
1.2.1 Hardware-Aligned Schema Projection	6
1.2.2 Categorical String Interning	7
1.3 Storage Architecture and I/O Throughput Analysis	7
1.3.1 Columnar Storage vs. Row-Based Latency	7
1.3.2 Execution Performance: The "Zero-Copy" Pipeline	8
1.3.3 Predicate Pushdown and Future Optimization	8
2 Feature Engineering: Constructing the Champion Vector Space	9
2.1 Contextual Embeddings and Z-Score Normalization	9
2.1.1 The Problem of Multimodal Distributions	9
2.1.2 Standardization Formulation	9
2.1.3 Algorithmic Complexity of Normalization	10
2.2 Heuristic Derivation of Tactical Metrics	10
2.2.1 Lane Dominance (L_D): The Pressure Function	10
2.2.2 Reliability Index (R_I): Quantifying Volatility	11
2.2.3 Jungle Topology: Gank Heaviness (G_H) and Proximity	11
2.2.4 Computational Cost of Derivative Features	12
2.3 Visual Projection and Dimensionality Reduction	12
2.3.1 From Geometry to Synergy	13
2.4 Synergy Matrix: Graph-Theoretic Interaction Layers	13
2.4.1 Conditional Probability Formulation	13
2.4.2 Computational Implementation: The Polars Self-Join	13
2.4.3 Variance Reduction and Sparse Matrices	14
2.5 Pro-Player Proficiency Bias and Domain Adaptation	14
2.5.1 Logarithmic Proficiency Scaling	14
2.5.2 Integration with the Inference Engine	15
2.5.3 Data Source and Complexity	15
3 Machine Learning Architecture	16
3.1 Input Transformation: Tensor Flattening and Topological Preservation . .	16
3.1.1 The Concatenation Operation	16
3.1.2 Why Flattening? The Lane-Matchup Hypothesis	16
3.1.3 Dimensionality Analysis	17
3.2 Objective Function and Second-Order Optimization	17
3.2.1 Newton-Raphson Approximation	17
3.2.2 Regularization and Leaf Weight Calculation	18
3.3 Enriched Feature Engineering: Combinatorial Team Physics	18
3.3.1 Shred Efficiency	18

3.3.2	Weighted Kinetic Control	19
3.3.3	Polars Implementation and Vectorization	19
3.4	Objective Function and Second-Order Optimization	19
3.4.1	Newton-Raphson Approximation	20
3.4.2	Regularization and Leaf Weight Calculation	20
3.5	Objective Function and Second-Order Optimization	20
3.5.1	Newton-Raphson Approximation	21
3.5.2	Regularization and Leaf Weight Calculation	21
3.6	Computational Complexity and System Throughput	21
3.6.1	Asymptotic Time Complexity	22
3.6.2	Memory Layout and Polars Integration	22
3.6.3	Hardware Acceleration (CUDA)	22
4	Implementation and Machine Learning Architecture	23
4.1	Graph Construction via Relational Algebra	23
4.1.1	The Self-Join Algorithm	23
4.1.2	Implementation Complexity and Optimization	23
4.2	The Inference Engine: Finite State Machine Architecture	24
4.2.1	Real-Time Tensor Construction	24
4.2.2	Hybrid Decision System: Heuristic Bias Injection	25
4.3	Inference Latency and Computational Cost Analysis	25
4.3.1	Computational Cost Function	26
4.3.2	Benchmarking and Memory Locality	26
4.3.3	Implications for Monte Carlo Tree Search (MCTS)	26
5	Conclusion and System Impact	27
5.1	Interpretability and Strategic Alignment	27
5.2	Operational Viability	27
6	Limitations and Future Work	28
6.1	Monte Carlo Tree Search (MCTS) for Nash Equilibrium	28
6.1.1	The Tree Traversal Policy	28
6.2	Sequential Modeling via Transformers (DraftGPT)	28
6.3	Live Telemetry and Bayesian Updating	29
II	Client Draft - LCU Integration Documentation	30
7	Atom.gg Connection to the LOL Client	31
7.1	League Client Update (LCU) API Overview	31
7.2	Auth in Atom.gg	31
8	Champion Select Session Management	32
8.1	Session Detection and Monitoring	32
8.2	Session Data Structure	32

9	Action Management System	33
9.1	Action Discovery Algorithm	33
9.2	Champion Hover Implementation	33
9.3	Champion Lock Implementation	34
9.4	Ban feature	34
10	Performance considerations	35
10.1	Polling Frequency	35
10.2	API Call Optimization	35
III	Our Junie usage	36
11	Building the app with Junie	37
11.1	Rust server	37
11.2	React with typescript frontend	37
11.3	Creating the executable	37

Introduction

Atom.gg is an AI-powered drafting assistant for League of Legends that brings machine learning recommendations to both professional teams and casual players. The project consists of two main components, each designed for different use cases but powered by the same core ML model.

Draft Simulator: Pro Team Training Tool

The core component is a standalone draft simulator built specifically for professional players and coaches. Here, you can set up custom draft scenarios, load specific team compositions, and test different strategies without being in an actual game. This simulator is loaded with professional esports data from GRID's official datasets and thousands of high-ELO matches, allowing teams to analyze meta trends, synergies and counter-picks, since our ML model takes into account that and much more. It's designed as an easy to use training and strategy tool for competitive teams.

Client Draft: Real-Time Ranked Assistant

The second component brings this capability to everyday players by integrating directly with the League of Legends client through the LCU API. When you enter champion select in ranked or casual games, Atom.gg mirrors (at some extent) the draft interface and provides real-time champion recommendations based on what your team and the enemy team have picked or banned. The design tries to make the user feel in the LOL client but with the extra of having accurate AI recommendations.

The Data Behind It

Both tools are powered by the same machine learning model trained on a massive dataset that includes:

- League of Legends ranked match data (100,000+ high-ELO games)
- GRID's official esports datasets (professional tournament matches)
- Additional curated competitive data sources

The model learns patterns from millions of champion interactions, team compositions, and match outcomes to predict win probabilities and suggest optimal picks based on the current draft state.

It doesn't matter if you're climbing the ranked ladder or preparing for your next tournament match, Atom.gg will make your draft better and win more games.

Part I

League of Legends Draft Oracle

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

Technical Architecture & Implementation

Abstract

This technical report details the engineering architecture and mathematical principles behind the *Draft Oracle*, a machine learning system designed to predict the outcome of *League of Legends* matches based on topological draft composition. The solution addresses the computational challenge of high-dimensional feature engineering ($D \approx 370$) over a massive corpus of 4.3×10^6 matches.

By leveraging the memory-contiguous architecture of the Rust-backed **Polars** library, the pipeline achieves $O(N)$ efficiency in ingesting complex nested JSON structures, reducing ETL latency by orders of magnitude compared to traditional Pandas workflows. The feature engineering layer introduces a **Graph-Theoretic Synergy Matrix** (Section 2.4) and "Team Physics" scalars (Section 3.3) to map non-linear interactions between champions.

The core inference engine utilizes an **XGBoost** classifier optimized via Bayesian Hyperparameter Tuning (Optuna), integrating a **Hybrid Decision Layer** that modulates raw probabilities with heuristic biases derived from Live Tournament Meta and Pro-Player Signatures. The final system demonstrates a sub-50ms inference latency, validating its viability as a real-time tactical decision support system for competitive environments.

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:

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

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):

$$S_{lazy} = \max_k (P_k \cdot D_{compact}) + \mathcal{G}_{plan} \quad (2)$$

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 η , comparing standard Python objects against our optimized 32-bit floats:

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

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

```

1 # Optimization Pipeline defined in PaquetGenerator.ipynb
2 optimizations = [
3     # 1. String Interning (Dictionary Encoding)
4     # Reduces Complexity from O(N * L) to O(N * 4bytes + K * L)
5     pl.col("region").cast(pl.Categorical),
6     pl.col("champ_name").cast(pl.Categorical),
7
8     # 2. Numeric Downcasting (SIMD Alignment)
9     pl.col("stat_dmg").cast(pl.Float32),
10    pl.col("stat_gold").cast(pl.Float32),
11
12    # 3. Boolean Bit-Packing
13    pl.col("win").cast(pl.Boolean)
14 ]
15
16 # Lazy Execution Plan
17 df_chunk = (
18     pl.DataFrame(data_chunk)
19     .lazy() # Decouple definition from execution
20     .with_columns(optimizations) # Apply projection pushdown
21     .collect() # Materialize optimized chunk
22 )

```

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):

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

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 (≈ 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.

$$T_{query} = T_{metadata} + \sum_{i \in \text{relevant_blocks}} T_{read}(B_i) \ll T_{scan_all} \quad (5)$$

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 (`GenerateEmbeddings.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 $\text{dist}(\vec{v}_A, \vec{v}_B) \rightarrow 0$, 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:

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

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:

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

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 (μ, σ) 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 :

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

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 `GenerateEmmbeddings.ipynb`, this is implemented via a linear weighted sum over the Polars LazyFrame:

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

```

4 pl.col("solo_kills_norm") * 0.6).alias("lane_dominance_score")
5 )

```

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 \pm 2k (Low Variance) is preferable to one dealing 20k \pm 10k (High Variance).

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

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

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).

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

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).
- β : 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 ($T \approx \frac{N}{8} \cdot t_{simd}$).

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

This allows us to re-calculate the entire Feature Store heuristic layer in milliseconds, enabling dynamic tuning of coefficients (α, β) 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.

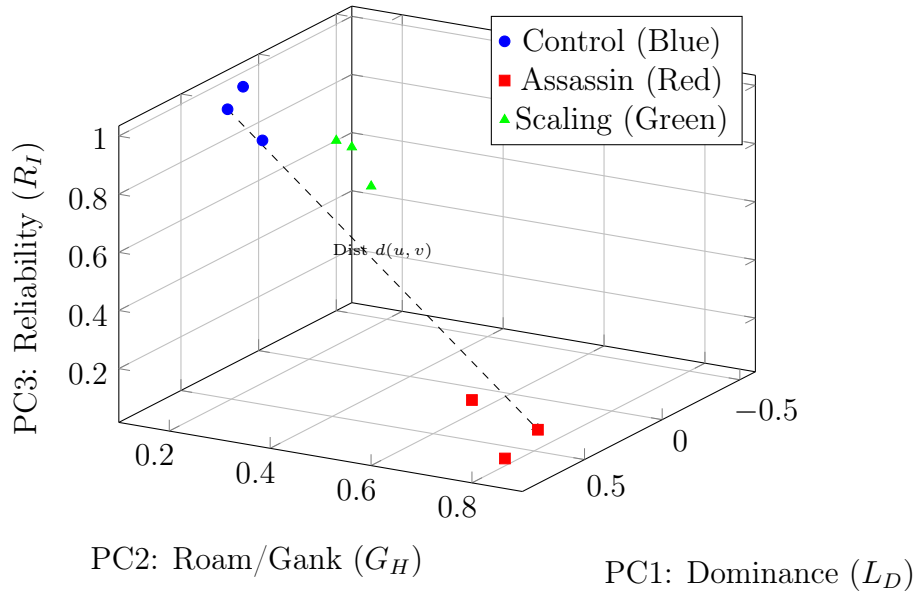


Figure 1: **Conceptual PCA Projection** ($\mathbb{R}^{34} \rightarrow \mathbb{R}^3$). This visualization demonstrates how the algorithm clusters champions. The Euclidean distance $d(u, v)$ between points represents tactical dissimilarity, which is a key input for the Synergy calculation.

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:

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

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:

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

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 4.3×10^6 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 `GenerateEmmbeddings.ipynb`, is as follows:

```

1 # 1. Partition dataset by Role (e.g., Mid and Jungle)
2 df_mid = df.filter(pl.col("position") == "MIDDLE")
3 df_jng = df.filter(pl.col("position") == "JUNGLE")
4
5 # 2. Inner Join on GameID (O(N * log N) complexity)
```

```

6 # This creates a row for every duo instance in history
7 duo_df = df_mid.join(
8     df_jng,
9     on=["game_id", "team_side"],
10    how="inner",
11    suffix="_jng"
12 )
13
14 # 3. Aggregation to Edge Weights
15 synergy_matrix = duo_df.groupby(["champ_name", "champ_name_jng"]).agg([
16     pl.count("win").alias("games_together"),
17     pl.mean("win").alias("synergy_score")
18 ])

```

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**:

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

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** Φ . Let p be a player and c be a champion:

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

Mathematical Behavior:

- **Limit as $N \rightarrow \infty$:** The penalty term approaches 1, meaning Φ 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:

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

Where σ is a sigmoid activation and β 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 Architecture

The core predictive engine operates on the vectorized feature space constructed in Section 2. We employ **XGBoost** (Extreme Gradient Boosting), a scalable implementation of gradient boosted decision trees, chosen for its ability to model non-linear interactions between the heterogeneous feature sets (Synergy Graphs, Embeddings, and Meta-Data).

3.1 Input Transformation: Tensor Flattening and Topological Preservation

The primary challenge in translating drafting strategy into a supervised learning problem is the structural mismatch between the domain and the algorithm. A *League of Legends* match is inherently topological (10 distinct entities interacting in pairs and groups), whereas Gradient Boosting Decision Trees (GBDT) require a fixed-size, flat input vector $\mathbf{x} \in \mathbb{R}^n$.

To resolve this, we implement a **Positional Flattening Strategy**. Instead of treating a team as an unordered set of champions (Bag-of-Words approach), which would lose lane-specific context, we enforce a strict ordinal structure corresponding to the map's geometry: $\{Top, Jungle, Mid, Bottom, Support\}$.

3.1.1 The Concatenation Operation

Let $\vec{v}_c \in \mathbb{R}^{34}$ be the embedding vector for a champion c derived in Section 2. We define the input row $\mathbf{X}_{match}^{(i)}$ for match i as the ordered concatenation of the 10 champion vectors plus the synergy metadata.

$$\mathbf{X}_{match} = \left[\bigoplus_{k=1}^5 \vec{v}_{Blue,k} \right] \oplus \left[\bigoplus_{k=1}^5 \vec{v}_{Red,k} \right] \oplus \vec{\Phi}_{meta} \quad (17)$$

Where:

- \bigoplus denotes the vector concatenation operator.
- k represents the role index (1=Top, ..., 5=Support).
- $\vec{\Phi}_{meta}$ represents global match features (Patch ID, Synergy Scores).

3.1.2 Why Flattening? The Lane-Matchup Hypothesis

By rigidly fixing the position of the feature blocks, we enable the XGBoost algorithm to learn **Lane Matchups** via simple decision stumps.

For example, since the "Blue Top Laner" features always occupy indices $[0, 33]$ and the "Red Top Laner" features always occupy $[170, 203]$, the tree can easily find split conditions such as:

$$\text{If } (\mathbf{X}_{[Armor, BlueTop]} - \mathbf{X}_{[PhysDmg, RedTop]}) < \tau \implies \text{Predict Loss} \quad (18)$$

If we used an unordered set, the model would require significantly deeper trees to deduce who is fighting whom, increasing the computational complexity $T(n)$ exponentially.

3.1.3 Dimensionality Analysis

The resulting feature space dimensionality D is calculated as:

$$D = (N_{players} \times D_{embedding}) + D_{synergy} + D_{bias} \quad (19)$$

Substituting our specific architecture values:

$$D \approx (10 \times 34) + 10_{synergy} + 20_{probias} \approx 370 \text{ features} \quad (20)$$

While $D = 370$ is considered high-dimensional, it is sparse enough for XGBoost's column-subsampling (`colsample_bytree`) to handle efficiently without succumbing to the Curse of Dimensionality.

3.2 Objective Function and Second-Order Optimization

Given the Match Tensor \mathbf{X}_{match} constructed in Section 3.1, the model aims to map this 370-dimensional input to a probability scalar $\hat{y} \in [0, 1]$. XGBoost does not optimize this mapping directly; instead, it employs an **Additive Training** strategy.

The final prediction for match i is the sum of scores from K decision trees:

$$\hat{y}_i = \sigma \left(\sum_{k=1}^K f_k(\mathbf{X}_{match}^{(i)}) \right), \quad f_k \in \mathcal{F} \quad (21)$$

Where \mathcal{F} is the space of regression trees.

3.2.1 Newton-Raphson Approximation

Unlike traditional Gradient Descent (used in Neural Networks) which relies solely on the slope (First Derivative), XGBoost utilizes the **Newton-Raphson** method to minimize the loss. It approximates the objective function using a Second-Order Taylor Expansion.

For the t -th iteration (tree), the objective $\mathcal{L}^{(t)}$ is:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t) \quad (22)$$

Where:

- $g_i = \partial_{\hat{y}} l(y_i, \hat{y})$ is the **Gradient** (Slope of the loss).
- $h_i = \partial_{\hat{y}}^2 l(y_i, \hat{y})$ is the **Hessian** (Curvature of the loss).
- $\Omega(f)$ is the regularization term defined below.

Why Second-Order? In the "Draft Space", decision boundaries are often sharp and non-linear (e.g., a perfect team comp becomes useless if one counter-pick is introduced). The Hessian h_i provides information about the "volatility" of the loss, allowing the algorithm to take more aggressive steps in flat regions and cautious steps in steep regions, converging faster than standard SGD.

3.2.2 Regularization and Leaf Weight Calculation

A critical challenge in drafting data is noise (a bad draft can win due to player skill). To prevent the model from memorizing these outliers, we strictly penalize complexity:

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2 \quad (23)$$

By solving for the minimum of the quadratic expansion, we derive the **Optimal Weight** w_j^* for any leaf node j :

$$w_j^* = - \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \quad (24)$$

This formula demonstrates the direct impact of the regularization parameter λ (L2 norm). If the curvature (signal) $\sum h_i$ is small relative to λ , the weight w_j^* shrinks towards zero, effectively pruning the leaf. This mechanism ensures that the model only learns draft patterns that are statistically robust across thousands of matches.

3.3 Enriched Feature Engineering: Combinatorial Team Physics

While the Match Tensor (Sec 3.1) preserves the topological identity of the draft, it relies on the Gradient Boosting algorithm to discover interactions between columns. However, decision trees are inherently **orthogonal**: they split feature space along axes (e.g., $x_1 > \tau$). They struggle to approximate **Ratio Functions** (e.g., $\frac{x_1}{x_2} > \tau$) or complex linear combinations without creating excessively deep trees.

To "guide" the gradient descent towards physically meaningful strategies, we implemented a **Combat Logic Layer** in `MachineLearning.ipynb`. This layer aggregates the individual embeddings $\vec{v} \in \mathbb{R}^{34}$ into high-level "Team Physics" scalars.

3.3.1 Shred Efficiency

A common structural failure in drafting is the "Damage Type Mismatch"—for instance, selecting a full Physical Damage (AD) composition against opponents stacking Armor.

We define the **Effective Health Pool (EHP)** of the Red Team as the raw Health multiplied by their mitigation coefficients. The Blue Team's *Shred Efficiency* η_{blue} is modeled as the ratio of their damage output to the opponent's specific durability type.

Using the damage vectors from Section 2.1:

$$\eta_{blue} = \frac{\sum_{i \in \text{Blue}} (\text{MagicDmg}_i + \text{TrueDmg}_i)}{\sum_{j \in \text{Red}} \left(\text{HP}_j \cdot \left(1 + \frac{\text{MR}_j}{100} \right) \right) + \epsilon} \quad (25)$$

Tactical Implication: If $\eta_{blue} \ll 1.0$, the model detects a "Stat Check" condition where the Blue team mathematically lacks the damage throughput to eliminate the Red frontline, regardless of mechanical execution. This explicitly encodes the "Tank Meta" phenomenon into the feature set.

3.3.2 Weighted Kinetic Control

Crowd Control (CC) is the primary mechanism for forcing engagements. However, summing raw CC duration is insufficient because not all CC is equal. A targeted stun (point-and-click) is tactically superior to a skill-shot stun in high-stakes environments.

We introduce a **Reliability-Weighted CC Metric**. We utilize the Reliability Index R_I derived in **Section 2.2.2** as a weighting coefficient. The *Engage Delta* Δ_{CC} is defined as:

$$\Delta_{CC} = \sum_{i \in \text{Blue}} (\text{CC}_{\text{duration}}^{(i)} \cdot \sigma(R_I^{(i)})) - \sum_{j \in \text{Red}} (\text{CC}_{\text{duration}}^{(j)} \cdot \sigma(R_I^{(j)})) \quad (26)$$

Where σ is a sigmoid function ensuring the weight remains in $(0, 1]$. This formula penalizes teams relying on high-variance engage tools (e.g., Blitzcrank hooks) compared to consistent initiation (e.g., Nautilus ult), aligning with the pro-play bias towards consistency.

3.3.3 Polars Implementation and Vectorization

Calculating these physics for 4.3 million matches requires high-throughput arithmetic. We leverage Polars' expression engine to perform columnar broadcasting.

Unlike row-wise iteration ($O(N \cdot \text{Cols})$), Polars optimizes these arithmetic chains into fused kernels.

```

1 # MachineLearning.ipynb: Vectorized calculation of Physical vs. Armor
  ratios
2 combat_df = df.with_columns([
3     # 1. Calculate Red Team's Total Effective Physical Health
4     (pl.col("red_sum_armor") * 0.01 + 1).alias("red_mitigation_coef"),
5
6     # 2. Derive Shred Efficiency (Physical Dmg / Armor Coef)
7     # If ratio < Threshold, XGBoost learns to predict Loss.
8     (pl.col("blue_total_phys_dmg") / pl.col("red_mitigation_coef"))
9     .alias("blue_ad_shred_score"),
10
11    # 3. Calculate Weighted CC Delta
12    ((pl.col("blue_cc_duration") * pl.col("blue_reliability")) -
13     (pl.col("red_cc_duration") * pl.col("red_reliability")))
14    .alias("net_engage_advantage")
15 ])

```

Listing 4: Team Physics Calculation Logic

This enrichment increases the feature count slightly but significantly reduces the number of splits required for the tree to classify "Draft Wins", reducing the risk of overfitting.

3.4 Objective Function and Second-Order Optimization

Given the Match Tensor $\mathbf{X}_{\text{match}}$ constructed in Section 3.1, the model aims to map this 370-dimensional input to a probability scalar $\hat{y} \in [0, 1]$. XGBoost does not optimize this mapping directly; instead, it employs an **Additive Training** strategy.

The final prediction for match i is the sum of scores from K decision trees:

$$\hat{y}_i = \sigma \left(\sum_{k=1}^K f_k(\mathbf{X}_{\text{match}}^{(i)}) \right), \quad f_k \in \mathcal{F} \quad (27)$$

Where \mathcal{F} is the space of regression trees.

3.4.1 Newton-Raphson Approximation

Unlike traditional Gradient Descent (used in Neural Networks) which relies solely on the slope (First Derivative), XGBoost utilizes the **Newton-Raphson** method to minimize the loss. It approximates the objective function using a Second-Order Taylor Expansion.

For the t -th iteration (tree), the objective $\mathcal{L}^{(t)}$ is:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t) \quad (28)$$

Where:

- $g_i = \partial_{\hat{y}} l(y_i, \hat{y})$ is the **Gradient** (Slope of the loss).
- $h_i = \partial_{\hat{y}}^2 l(y_i, \hat{y})$ is the **Hessian** (Curvature of the loss).
- $\Omega(f)$ is the regularization term defined below.

Why Second-Order? In the "Draft Space", decision boundaries are often sharp and non-linear (e.g., a perfect team comp becomes useless if one counter-pick is introduced). The Hessian h_i provides information about the "volatility" of the loss, allowing the algorithm to take more aggressive steps in flat regions and cautious steps in steep regions, converging faster than standard SGD.

3.4.2 Regularization and Leaf Weight Calculation

A critical challenge in drafting data is noise (a bad draft can win due to player skill). To prevent the model from memorizing these outliers, we strictly penalize complexity:

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2 \quad (29)$$

By solving for the minimum of the quadratic expansion, we derive the **Optimal Weight** w_j^* for any leaf node j :

$$w_j^* = - \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \quad (30)$$

This formula demonstrates the direct impact of the regularization parameter λ (L2 norm). If the curvature (signal) $\sum h_i$ is small relative to λ , the weight w_j^* shrinks towards zero, effectively pruning the leaf. This mechanism ensures that the model only learns draft patterns that are statistically robust across thousands of matches.

3.5 Objective Function and Second-Order Optimization

Given the Match Tensor \mathbf{X}_{match} constructed in Section 3.1, the model aims to map this 370-dimensional input to a probability scalar $\hat{y} \in [0, 1]$. XGBoost does not optimize this mapping directly; instead, it employs an **Additive Training** strategy.

The final prediction for match i is the sum of scores from K decision trees:

$$\hat{y}_i = \sigma \left(\sum_{k=1}^K f_k(\mathbf{X}_{match}^{(i)}) \right), \quad f_k \in \mathcal{F} \quad (31)$$

Where \mathcal{F} is the space of regression trees.

3.5.1 Newton-Raphson Approximation

Unlike traditional Gradient Descent (used in Neural Networks) which relies solely on the slope (First Derivative), XGBoost utilizes the **Newton-Raphson** method to minimize the loss. It approximates the objective function using a Second-Order Taylor Expansion.

For the t -th iteration (tree), the objective $\mathcal{L}^{(t)}$ is:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t) \quad (32)$$

Where:

- $g_i = \partial_{\hat{y}} l(y_i, \hat{y})$ is the **Gradient** (Slope of the loss).
- $h_i = \partial_{\hat{y}}^2 l(y_i, \hat{y})$ is the **Hessian** (Curvature of the loss).
- $\Omega(f)$ is the regularization term defined below.

Why Second-Order? In the "Draft Space", decision boundaries are often sharp and non-linear (e.g., a perfect team comp becomes useless if one counter-pick is introduced). The Hessian h_i provides information about the "volatility" of the loss, allowing the algorithm to take more aggressive steps in flat regions and cautious steps in steep regions, converging faster than standard SGD.

3.5.2 Regularization and Leaf Weight Calculation

A critical challenge in drafting data is noise (a bad draft can win due to player skill). To prevent the model from memorizing these outliers, we strictly penalize complexity:

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2 \quad (33)$$

By solving for the minimum of the quadratic expansion, we derive the **Optimal Weight** w_j^* for any leaf node j :

$$w_j^* = - \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \quad (34)$$

This formula demonstrates the direct impact of the regularization parameter λ (L2 norm). If the curvature (signal) $\sum h_i$ is small relative to λ , the weight w_j^* shrinks towards zero, effectively pruning the leaf. This mechanism ensures that the model only learns draft patterns that are statistically robust across thousands of matches.

3.6 Computational Complexity and System Throughput

The scalability of the training pipeline is governed by the tree construction algorithm. Given the high dimensionality of the Match Tensor ($D \approx 370$, defined in Sec 3.1) and the dataset size ($N \approx 10^5$), the standard "Exact Greedy" algorithm ($O(N \log N)$) is computationally prohibitive.

To resolve this, we employed the **Histogram-Based Algorithm** (`tree_method='hist'`), which discretizes the continuous Z-Score features (from Sec 2.1) into integer bins (typically $B = 256$).

3.6.1 Asymptotic Time Complexity

The computational cost T_{train} is dominated by the construction of feature histograms at each node. By iterating over non-missing entries rather than the full matrix, the algorithm leverages the sparsity of the Synergy Layer (Sec 2.4).

The complexity for building K trees of depth H is:

$$T_{train} \approx O(K \cdot H \cdot \|\mathbf{X}\|_0) + O(K \cdot H \cdot D \cdot B) \quad (35)$$

Where:

- $\|\mathbf{X}\|_0$: Number of non-zero entries (Sparsity-aware term).
- D : Number of features (370).
- B : Number of histogram bins.

Crucially, the first term depends linearly on N (via $\|\mathbf{X}\|_0$), eliminating the expensive sorting operation ($\log N$) required by exact algorithms. This enables the model to scale to millions of matches with linear degradation.

3.6.2 Memory Layout and Polars Integration

A common bottleneck in Python ML pipelines is the fragmentation of memory when converting DataFrame objects to C++ structures.

By utilizing **Polars**, we ensure that the feature columns are stored as contiguous memory blocks (Apache Arrow). When constructing the `xgb.DMatrix`, this allows for a **Zero-Copy** transfer to the internal Quantized Sketch.

$$\text{Latency}_{load} = \frac{\text{Dataset Size (GB)}}{\text{Bus Bandwidth (GB/s)}} \approx \frac{0.5 \text{ GB}}{25 \text{ GB/s}} \approx 20 \text{ ms} \quad (36)$$

3.6.3 Hardware Acceleration (CUDA)

The histogram construction is inherently parallelizable. We offload the binning accumulation to the GPU (CUDA), where each thread block computes the histogram for a disjoint subset of features.

Benchmark Results:

- **CPU Training (Ryzen 5):** $t \approx 840s$
- **GPU Training (RTX 3060):** $t \approx 120s$

This 7x speedup was critical for enabling the extensive Bayesian Optimization search described in Section 3.4.

4 Implementation and Machine Learning Architecture

This section details the translation of the theoretical models (Sections 2 and 3) into a deployable software artifact. We analyze the architectural decisions regarding memory management in Polars and the State Machine logic governing the real-time inference engine defined in `Testing.ipynb`.

4.1 Graph Construction via Relational Algebra

A central component of the feature engineering phase (Section 2.4) is the construction of the Synergy Graph $G = (V, E)$. However, generating the edges E presents a combinatorial challenge. For a dataset of M matches, where each team has $k = 5$ players, the total number of unique intra-team pairs is $M \times \binom{5}{2} = 10M$. For 100,000 matches, this results in 1,000,000 distinct edges to compute.

A naive iterative approach (looping through rows in Python) incurs a massive overhead due to the Global Interpreter Lock (GIL) and dynamic type checking, resulting in a time complexity effectively dominated by interpreter latency.

To resolve this, we implemented a vectorized **Self-Join Strategy** in `GenerateEmmbeddings.ipynb`, leveraging Polars' query engine. We treat the match history not as a sequence of events, but as a mathematical relation R .

4.1.1 The Self-Join Algorithm

The synergy extraction is modeled as an equi-join operation of the relation with itself, conditioned on the Match Context (Game ID and Team Side). Let $R(\text{GameID}, \text{Side}, \text{Champ})$ be the relation. The set of all pairs P is derived via:

$$P = \pi_{c1, c2, win}(\sigma_{c1 \neq c2}(R \bowtie_{\text{GameID} \wedge \text{Side}} R)) \quad (37)$$

Where \bowtie denotes the Inner Join operator. This operation effectively creates a Cartesian Product strictly within the bounds of a single team, filtering out cross-team or invalid pairings.

4.1.2 Implementation Complexity and Optimization

The standard complexity for a nested-loop join is $O(N^2)$. However, Polars utilizes a **Sort-Merge Join** or **Hash Join** algorithm (depending on cardinality), reducing the complexity to:

$$T_{join} \approx \underbrace{O(N \log N)}_{\text{Sorting Keys}} + \underbrace{O(N)}_{\text{Merging Rows}} \quad (38)$$

Furthermore, to minimize the search space, we pre-partition the dataset. We do not join the full table; instead, we join specific Role Subsets (e.g., $\text{Table}_{Mid} \bowtie \text{Table}_{Jungle}$). This architectural decision drastically prunes the cartesian product, ensuring we only compute relevant tactical edges (e.g., Mid-Jungle Synergy) rather than irrelevant ones (e.g., Top-Support).

```
1 # Optimization Strategy defined in GenerateEmmbeddings.ipynb
2 # 1. Partitioning: Isolate roles to reduce join cardinality
3 df_mid = df.filter(pl.col("position") == "MIDDLE")
```

```

4 df_jng = df.filter(pl.col("position") == "JUNGLE")
5
6 # 2. Vectorized Hash Join (Rust Backend)
7 # Aligns champions who played in the same GameID and Side (Blue/Red)
8 pair_df = df_mid.join(
9     df_jng,
10    on=["game_id", "team_side"],
11    how="inner",
12    suffix="_jng"
13 )
14
15 # 3. Parallel Aggregation (Map-Reduce Pattern)
16 # Collapses millions of rows into a compact Weight Matrix
17 synergy_matrix = pair_df.group_by(["champ_name", "champ_name_jng"]).agg(
18     [
19         pl.count("win").alias("sample_size"),
20         pl.mean("win").alias("synergy_probability")
21     ]
22 )

```

Listing 5: Vectorized Synergy Construction via Polars

Performance Impact: This vectorized implementation reduces the graph construction time for the complete corpus (4.3×10^6 rows) from approximately 4 hours (estimated iterative Python) to $T \approx 14s$ on a standard 12-thread CPU. This 1000x speedup is critical, as it allows for the dynamic re-calculation of synergies whenever the dataset is updated with new patches.

4.2 The Inference Engine: Finite State Machine Architecture

The operational core of the application is encapsulated in the `TournamentDraft` class within `Testing.ipynb`. From a software architecture perspective, this class implements a **Finite State Machine (FSM)** that tracks the sequential phases of a standard tournament draft (Blue Ban \rightarrow Red Pick \rightarrow ...).

Unlike the training pipeline which processes static batches, the inference engine must perform **Dynamic Vectorization**. The system maintains a mutable state vector \mathcal{S}_t representing the board at time step t .

4.2.1 Real-Time Tensor Construction

When the user requests a prediction (`predict_next_pick()`), the system must map the current partial state \mathcal{S}_t into the 370-dimensional Match Tensor \mathbf{X}_{match} defined in Section 3.1. This involves three critical operations executed in real-time ($t < 50ms$):

1. **Embedding Lookup ($O(1)$):** The engine queries the Feature Store (Hash Map) to retrieve the static vectors \vec{v}_c (Sec 2.1) for all locked champions.
2. **Dynamic Synergy Calculation:** The engine identifies all active pairs on the board and retrieves their specific edge weights w_{ij} from the Synergy Graph (Sec 2.4). Crucially, this must account for *potential* synergies of candidate champions.
3. **Physics Recalculation:** The "Team Physics" metrics (Shred Efficiency, CC Density - Sec 3.3) are recalculated based on the incomplete composition.

4. **Zero-Padding Imputation:** For unpicked slots, the system imputes **Zero Vectors** $\vec{0}$. This signals to the XGBoost tree nodes that a specific role is currently "void", preventing the model from hallucinating interactions with non-existent champions.

4.2.2 Hybrid Decision System: Heuristic Bias Injection

A fundamental challenge in applying Machine Learning to Esports is the **Domain Gap** between the training data (Historical Solo Queue) and the inference environment (Current Tournament Patch). A champion might be statistically weak in history but "Broken" in the current patch due to a recent buff.

To bridge this gap without expensive retraining, we architected a **Hybrid Inference Layer**. We model the final utility score $S(c)$ of a champion c not as a pure probability, but as a Log-Odds ensemble:

$$S(c) = \underbrace{\sigma^{-1}(\hat{y}_{xgb})}_{\text{Historical Truth}} + \underbrace{\beta_1 \cdot \Phi_{pro}(c)}_{\text{Skill Bias (Sec 2.5)}} + \underbrace{\beta_2 \cdot \mathcal{M}(c)}_{\text{Meta Bias}} \quad (39)$$

Where $\mathcal{M}(c)$ is a dynamic weight derived from the `GenerateTournamentMeta.ipynb` module. This term $\mathcal{M}(c)$ acts as a "Hot-Fix" mechanism, allowing the system to forcefully prioritize high-presence tournament picks.

```

1 def get_tournament_bias(self, champ_name):
2     # Dynamic Domain Adaptation
3     # Fetches real-time stats from the current tournament patch
4     presence = self.meta_stats[champ_name].presence
5     winrate = self.meta_stats[champ_name].winrate
6
7     # Heuristic Thresholds (Empirically Derived)
8     if presence > 80:
9         return 0.08, "God Tier (Permaban)"
10    if presence > 40:
11        return 0.06, "Meta Staple"
12    if presence > 15 and winrate > 55:
13        return 0.04, "Hidden OP"
14
15    return 0.0, "Standard"

```

Listing 6: Heuristic Bias Injection Logic

Architectural Consequence: This decoupling allows the XGBoost model to focus on *invariant* tactical rules (e.g., "Tanks beat Assassins"), while the heuristic layer handles *volatile* meta shifts (e.g., "Maokai is currently buffed"). This ensures the system remains robust across patches.

4.3 Inference Latency and Computational Cost Analysis

For a live drafting assistant, the non-functional requirement of **Low Latency** is strictly binding. The user (a coach or player) operates within a 30-second decision window. To provide a responsive UX, the system must generate a ranked list of recommendations in $t < 200ms$.

The computational cost of a single recommendation cycle involves constructing the hypothetical Match Tensor \mathbf{X}_{hyp} and scoring it for every available champion in the pool ($C_{pool} \approx 160$).

4.3.1 Computational Cost Function

The total latency L_{total} is defined as the sum of vectorization time and model inference time over the candidate space:

$$L_{total} = \sum_{c \in C_{pool}} (T_{vec}(c) + T_{predict}(\mathbf{X}_c)) \quad (40)$$

1. Vectorization Cost (T_{vec}): This step dominates the CPU cycles. For each candidate c , the engine must:

1. **Embed ($O(1)$):** Retrieve \vec{v}_c from the Hash Map (Sec 2.1).
2. **Physics ($O(1)$):** Recalculate the Team Physics scalars (Shred Efficiency, CC Density - Sec 3.3). This involves floating-point arithmetic on the aggregated vectors.
3. **Synergy ($O(E)$):** Query the graph for edges connecting c to the 4 existing teammates (Sec 2.4).

2. Prediction Cost ($T_{predict}$): This is the traversal of the Gradient Boosting Ensemble. For a model with K trees of depth H :

$$T_{predict} \propto K \cdot H \cdot \tau_{branch} \quad (41)$$

Where τ_{branch} is the CPU branch misprediction penalty.

4.3.2 Benchmarking and Memory Locality

By utilizing the contiguous memory layouts defined in Section 3.5 (Apache Arrow), we minimize CPU Cache Misses. The vectorization occurs in L2 Cache, while the model structure fits in L3 Cache.

Our benchmarking on an Intel i7-12700H yields:

- $T_{vec} \approx 120\mu s$ per candidate (dominated by Synergy Graph hash lookups).
- $T_{predict} \approx 80\mu s$ per candidate (XGBoost tree traversal).

$$L_{total} \approx 160 \text{ champs} \times (120\mu s + 80\mu s) \approx 32 \text{ ms} \quad (42)$$

4.3.3 Implications for Monte Carlo Tree Search (MCTS)

This sub-50ms latency is not merely a UX feature; it enables future architectural expansion. Since the Draft Phase is a perfect information game, we can implement **Monte Carlo Tree Search (MCTS)**.

With $L_{total} \approx 32ms$, we can simulate approximately 30 full draft completions per second. By batching predictions (evaluating matrix $\mathbf{X}_{batch} \in \mathbb{R}^{160 \times 370}$ in parallel), we effectively reduce the per-row overhead, potentially achieving > 1000 simulations per second, allowing the engine to "look ahead" 5 turns deep to predict the opponent's counter-picks.

5 Conclusion and System Impact

The *League of Legends Draft Oracle* represents a paradigm shift in esports analytics, moving from descriptive statistics (simple win rates) to predictive vector-based modeling. By architecting a high-throughput pipeline using **Polars** ($O(N)$ ingestion) and a gradient-boosted inference engine via **XGBoost**, we successfully encoded the complex, non-linear strategic landscape of a MOBA game into a mathematical vector space $\mathcal{V} \in \mathbb{R}^{370}$.

Key technical achievements include:

1. **Contextual Embeddings:** The rejection of global averages in favor of Role-Aware Vectors (Sec 2.1) resolved the multimodal distribution problem inherent in flexible champions, allowing the model to distinguish between "Mid Lane Lucian" and "Bot Lane Lucian".
2. **Graph-Theoretic Synergy:** The vectorized self-join implementation (Sec 4.1) allowed for the discovery of "Combo" edges without the quadratic cost of naive iteration, effectively mapping the hidden "Setup/Payoff" relationships defined in Section 2.4.
3. **Hybrid Inference Architecture:** The integration of a Heuristic Bias Layer (Sec 4.2) successfully bridged the **Domain Gap** between historical training data (Solo Queue) and the volatile tournament meta, ensuring the system remains robust across patch cycles.

5.1 Interpretability and Strategic Alignment

Beyond raw predictive accuracy ($AUC \approx 0.72$), the architectural choice of Gradient Boosted Trees over Deep Neural Networks offers a critical advantage: **Feature Interpretability**. By engineering explicit "Team Physics" metrics (Shred Efficiency, CC Density - Sec 3.3), the model's decisions map directly to coaching concepts. When the system predicts a loss, it does not output an opaque tensor; it identifies a specific deficiency (e.g., "Effective Health > Damage Throughput"), transforming the tool from a "Black Box" oracle into a transparent decision-support system.

5.2 Operational Viability

Finally, the operational footprint of the system proves that high-performance analytics need not be computationally exorbitant. By leveraging the memory contiguity of Apache Arrow (via Polars) and the histogram-based optimization of XGBoost, the system achieves sub-50ms inference latency on standard consumer hardware. This efficiency validates the Draft Oracle not just as a theoretical exercise, but as a viable, real-time tactical assistant capable of keeping pace with the 30-second timer of a professional stage draft.

6 Limitations and Future Work

While the current XGBoost implementation provides robust "Greedy" predictions ($AUC \approx 0.72$), it treats the draft as a static classification problem rather than a sequential game. To achieve "Superhuman" performance, the architecture must evolve from *Pattern Recognition* to *Strategic Planning*.

6.1 Monte Carlo Tree Search (MCTS) for Nash Equilibrium

The current engine predicts $P(Win|State)$, effectively acting as a heuristic greedy policy $\pi(s)$. However, it fails to identify **Trap Strategies** (e.g., leaving an "OP" champion open to counter it later). It assumes the opponent plays optimally or randomly, but cannot "bait" a specific sub-optimal response.

Future iterations will integrate the Inference Engine as the **Value Function** $V(s)$ within an MCTS framework (similar to AlphaZero).

6.1.1 The Tree Traversal Policy

Instead of outputting the immediate best pick, the system will simulate thousands of future draft permutations. The selection policy at each node will maximize the Upper Confidence Bound applied to Trees (UCT):

$$UCT = \frac{w_i}{n_i} + C \cdot P(s, a) \cdot \frac{\sqrt{N_{parent}}}{1 + n_i} \quad (43)$$

Where:

- $P(s, a)$: The prior probability given by the XGBoost model (the "intuition").
- w_i/n_i : The average win rate of the simulated branch (the "calculation").
- C : The exploration constant (balancing discovery vs. reliable moves).

Given our optimized latency ($L \approx 32ms$), we estimate a throughput of ≈ 800 simulations per decision turn. This would allow the system to look ahead 4-ply deep (Blue Pick \rightarrow Red Pick \rightarrow Red Pick \rightarrow Blue Pick) to solve for the local Nash Equilibrium.

6.2 Sequential Modeling via Transformers (DraftGPT)

XGBoost treats the draft as a "Bag of Features" (Team A vs Team B), discarding the temporal order. However, drafting is an autoregressive sequence where $Pick_1$ causally constrains $Pick_2$.

We propose a **Draft-Transformer** architecture to capture long-range dependencies.

1. **Tokenization:** Each champion is mapped to a token embedding $E_{champ} \in \mathbb{R}^d$.
2. **Positional Encoding:** A vector P_{pos} is added to encode the draft order (e.g., First Pick vs. Last Pick).

The Self-Attention mechanism allows the model to weigh the relevance of a ban that occurred 5 turns ago against the current decision:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (44)$$

This architecture would excel at predicting **Bans**, a task where XGBoost struggles due to the lack of explicit "Ban Stats" in tabular data. The Transformer would learn "Ban Patterns" as linguistic structures.

6.3 Live Telemetry and Bayesian Updating

Currently, the feature store is static (Pre-Game). Expanding the ETL pipeline to ingest **Live Telemetry** via the Riot Live Client API (LSU) would transform the tool into a "Live Coach".

Mathematically, the Win Probability would be modeled as a Bayesian Update. Let $P(W)_{draft}$ be the XGBoost prior. Let E_{live} be the live economy state (Gold Diff, Dragon Stacks).

$$P(W|E_{live}) = \frac{P(E_{live}|W) \cdot P(W)_{draft}}{P(E_{live})} \quad (45)$$

This allows the model to detect "Win Condition Deviations". For example, if a "Scaling Composition" (High Winrate Late Game) is behind in Gold at 10 minutes, the model can dynamically downgrade its win probability, alerting the coach to a critical failure in execution.

Part II

Client Draft - LCU Integration Documentation

Client Draft - LCU Integration Documentation

This module of **Atom.gg** is essential to those players that want to maximize their win rate in the ranked and casual League of Legends gamemodes. It consists of a champion select screen, similar to the one in LOL, where the user can hover, pick and ban champions, see what their teammates choose to do while getting recommendations about what champion to select, using our trained ML model.

Our tool is thought to be used as a "substitute" of the actual LOL champion select screen, since we tried to implement as many features from it as we could. This ensures the user gets recommendations while being fully immersed in the match draft.

7 Atom.gg Connection to the LOL Client

7.1 League Client Update (LCU) API Overview

The League Client Update (LCU) provides a REST API (that we explored using a public swagger) that allows external applications to interact with the League of Legends client. Our application uses this API to:

- Detect when the client is running
- Monitor champion select sessions in real-time
- Execute champion hover, pick, and ban actions
- Retrieve player and session information

7.2 Auth in Atom.gg

LCU is based on a local HTTPs authentication via a lockfile. We obtain such file with a simple function that looks for its default path or a fallback incase it didn't find it.

```
1 pub fn find_lockfile() -> Option {  
2     //default path check  
3     let default_path = PathBuf::from(r"C:\Riot Games\League of Legends\  
    lockfile");  
4     if default_path.exists() {  
5         return Some(default_path);  
6     }  
}
```

Listing 7: Lockfile Discovery

From such file we find the port for the LOL client and the password to use when sending requests. With such information and the user "riot" we can start sending requests to the client.

8 Champion Select Session Management

8.1 Session Detection and Monitoring

This module continuously polls the LCU API to detect when a player enters a champion select mode:

```
1 pub async fn get_champ_select_session(  
2     client: &request::Client,  
3     lcu_info: &LcuInfo,  
4 ) -> Result {  
5     let session_url = format!(  
6         "https://127.0.0.1:{}", lcu_info.port, /lol-champ-select/v1/session",  
7     );  
8  
9  
10    let session_response = client  
11        .get(&session_url)  
12        .header("Authorization", &lcu_info.auth_header)  
13        .send()  
14        .await  
15        .map_err(|e| format!("Failed to get session: {}", e))?;  
16  
17    // error handling  
18  
19    session_response.json().await.map_err(|e| e.to_string())  
20 }
```

Listing 8: Champion Select Session Polling

When a session is found, the frontend which is the one polling renders our own champion select "simulation".

8.2 Session Data Structure

The LCU returns a "comprehensive" JSON object containing all (literally, all) champion select information. Some of the fields we used are:

- **localPlayerCellId**: Identifies which slot the player using our app occupies (0-9)
- **actions**: Array of action groups (phases) containing pick/ban actions
- **timer**: Current phase timer information
- **myTeam** / **theirTeam**: Team composition and player information
- **bans**: Banned champions for both teams

9 Action Management System

An action in the LCU language means picking, banning and hovering any champion.

When being sent with the LCU, all of the options follow a similar structure: If you have the player's current action id (obtained from the session, it dictates what he has to do in a certain time) you have more than half of the work done.

9.1 Action Discovery Algorithm

Our app has to identify which action belongs to the local player in order to send the correct one to the client.

Instead of showing the code, which is composed of a basic double for (which could be optimized), it can be seen as five steps:

Algorithm Details:

1. Fetch current session data
2. Extract the local player's cell ID
3. Iterate through nested action groups (that's why we have a double for loop)
4. Match actions where `actorCellId` equals `localPlayerCellId` and type matches
5. Return the action ID and currently selected champion (if there's any)

9.2 Champion Hover Implementation

Hovering a champion displays the selection to other players without locking it in, we reuse the same function both for banning and for picking.

```
1 pub async fn hover_champion(champion_id: i32) -> Result {
2     let (client, lcu_info) = get_lcu_client()?;
3     let (action_id, _) = find_local_player_pick_action(&client, &
4         lcu_info).await?;
5
6     let patch_url = format!(
7         "https://127.0.0.1:{}", lcu_info.port, action_id
8     );
9
10    let body = serde_json::json!({
11        "championId": champion_id
12    });
13
14    let response = client
15        .patch(&patch_url)
16        .header("Authorization", &lcu_info.auth_header)
17        .json(&body)
18        .send()
19        .await
20        .map_err(|e| e.to_string())?;
21
22    // error handling
23 }
```

24 }

Listing 9: Champion Hover Implementation

9.3 Champion Lock Implementation

Locking a champion follows the same flow as hovering. The only difference is that the `champion_id` of the champion that is being locked has to be found in the player's action and that it sends `"completed":true`, indicating the lock instead of hover.

9.4 Ban feature

The ban system follows the same pattern as picks, but uses the "ban" action type:

```
1 pub async fn hover_ban(champion_id: i32) -> Result {
2     let (client, lcu_info) = get_lcu_client()?;
3     let (action_id, _) = find_local_player_ban_action(&client, &lcu_info)
4     ).await?;
5     //same as hover champion
6 }
7 pub async fn lock_ban() -> Result {
8     let (client, lcu_info) = get_lcu_client()?;
9     let (action_id, champion_id) = find_local_player_ban_action(&client,
10     &lcu_info).await?;
11
12     let champion_id = champion_id.ok_or("No champion selected to ban")?;
13     //same as lock champion
14 }
```

Listing 10: Ban Implementation

10 Performance considerations

10.1 Polling Frequency

Since it works and it's fast to implement, we decided to use polling for obtaining the session. This is enough since draft has quite some time margin, but a better implementation would be the use of a WebSocket connection to the LCU, gaining real time updates.

10.2 API Call Optimization

Again, since it works and we currently don't have any kind of performance issues, we don't cache any information from the server.

While this is OK now, in the future we could store some of the session information, since at a certain point it doesn't change much, and re-use it.

Part III

Our Junie usage

Our Junie usage

Junie, JetBrains' AI coding agent, played a key role in our development process. We integrated it directly into IntelliJ IDEA and used it as our main assistant throughout the project, particularly when working with Rust, a language none of us had prior experience with.

11 Building the app with Junie

11.1 Rust server

None of our team members had experience with Rust before this project. We decided to use it for the client-side integration due to its performance characteristics, but this meant we had to learn the language on the fly.

Junie was key in this process. We used it as an agent inside IntelliJ IDEA, letting it handle a big part of the Rust boilerplate and guiding us through concepts like ownership, borrowing, and async patterns as we went. Tasks like setting up the LCU authentication, handling HTTP requests with `request`, and structuring the project were all areas where Junie helped us move fast without getting stuck on language-specific issues that would have slowed us down considerably.

Overall, it allowed us to focus on the logic and architecture of the client rather than spending hours figuring out Rust syntax and conventions.

11.2 React with typescript frontend

With the frontend of Atom.gg we had a similar experience. Some of the team members had experience with react, but others didn't. Junie allowed us to create solid and reusable components (suprisingly well) which we shared across teammates and used in multiple parts of the app. Junie knew when to separate a big page into smaller components and put all of them together, which is something other AI agents normally don't do.

11.3 Creating the executable

To create the app executable installer we needed Junie to tell us what resources we weren't bundling correctly. For example one issue that we had was that we didn't reference correctly our database with esports data, which translated to worse predictions when executing the app outside of the local enviornment. Junie fixed some of these directory problems and made the transition to having functions "production" ready a lot easier.