League of Legends Match Outcome Predictor

Bartłomiej Hryniewski, Kamila Kiełb

December 31, 2025

1 Introduction

1.1 What is League of Legends?

League of Legends (often referred to as LoL or League) is a multiplayer online battle arena video game developed by Riot Games. The game's main mode, Summoner's Rift, features two teams consisting of five players each. Each player controls an entity called a champion, which has a unique set of abilities. The objective of the game is to destroy a structure located in the opponent's base, called the Nexus. Each team must also defend a set of structures known as Turrets and Inhibitors. During a match, players can gain an advantage by collecting gold and exchanging it for items, earning experience points in order to upgrade their abilities, or by destroying neutral objectives that spawn throughout the map.

Since its release in 2009, League of Legends has grown to become one of the most popular video games worldwide. In 2025, the game had approximately 120–135 million monthly active players, peaking at around 180 million in 2022, despite being over a decade old.

League of Legends offers two competitive ranked modes: Ranked Solo/Duo (commonly referred to as solo queue), in which players compete alone or with one teammate, and Ranked Flex (flex queue), which allows teams of up to five players. Winning or losing ranked games results in gaining or losing League Points (LP), which are used to progress through a tiered ranking system consisting of ten ranks, from Iron to Challenger. A player's solo queue rank is commonly used as an indicator of individual skill level.

1.2 Motivation

A typical League of Legends match on Summoner's Rift lasts between 25 and 35 minutes. The first 15 minutes of gameplay are commonly referred to as the early game. Team performance during the early game has a significant impact on the final match outcome. The objective of this project was to develop an algorithm capable of predicting the match winner based solely on information available at the 15-minute mark. Knowledge of the predicted outcome may help losing teams assess their comeback potential and strategic priorities, while allowing winning teams to better understand their win conditions. Furthermore, the performance of the proposed models serves as evidence of the influence of early-game factors on the final match result.

Due to the large number of possible in-game scenarios and the inherent unpredictability of human decision-making, it is impossible to determine the match winner with certainty before the game concludes. Even a team with a substantial lead may commit critical mistakes, while a disadvantaged team may successfully outplay their opponents and reverse the outcome. Consequently, the goal of this project was not to create a model with near-perfect accuracy, but rather to achieve a prediction accuracy exceeding 70%, corresponding to correct predictions in approximately three out of four matches.

1.3 Input data

Several key parameters that strongly influence the probability of winning were identified, including champion kills, accumulated gold and experience, defeated neutral objectives (dragons and Void Grubs), destroyed enemy turrets, and creep score (the total number of enemy minions and neutral monsters killed). The data were obtained via the Riot Developer Portal, which provides access to

an Application Programming Interface (API). Using an API key, queries can be sent to Riot servers, which return detailed match timelines for the selected games.

1.4 Technical topic description

The presented task constitutes a classical machine learning (ML) problem. The algorithm is required to identify patterns within a provided dataset and generate predictions based on these patterns. The resulting model is a binary classifier, as it assigns each input instance to one of two possible outcomes: a given team either wins or loses the match.

2 State of the art

Machine learning is one of the fastest developing fields of computer science. New algorithms and solutions are being introduced daily. Thanks to it, there are multiple possible solutions for our ML problem.

2.1 Logistic regression

Logistic regression is a supervised machine learning algorithm commonly used for binary classification problems. In logistic regression, the target variable is binary and typically assumes values of 0 or 1. The model estimates the probability of each outcome, where the probabilities of the two classes necessarily sum to one. In the considered problem, the binary variable corresponds to a team either winning or losing the match.

An example application of logistic regression to League of Legends match prediction is presented in https://github.com/saikaryekar/lol-match-prediction, where the author achieved an accuracy of 82% when predicting match outcomes in professional games and identifying key factors influencing the result. However, the proposed solution had access to more comprehensive information, as it was not restricted to data available at the 15-minute mark.

The main advantages of logistic regression include very fast training time, low computational requirements, and a probabilistic output that directly maximizes the likelihood function, providing an interpretable measure of prediction uncertainty. Nevertheless, the model assumes a linear relationship between input features and the log-odds of the outcome, which limits its ability to capture non-linear interactions between parameters. For example, the impact of destroyed turrets may depend on team composition, such as the presence of assassin-type champions, which logistic regression cannot model explicitly.

2.2 Random forest

Random forest is an ensemble learning method used for classification and regression tasks. It consists of multiple decision trees trained on different subsets of the data, whose predictions are aggregated, typically by averaging probabilities or majority voting, to produce the final output.

This algorithm is well known for its ability to model nonlinear relationships and complex interactions between input features. It is relatively robust to outliers and can handle missing values more effectively than many other models. Additionally, random forests provide estimates of feature importance, which can be useful for model interpretation. However, training and inference are computationally more demanding compared to simpler models, and the predicted probabilities often remain close to the decision boundary (approximately 0.45–0.55), even in cases where one team has a substantial advantage.

2.3 Gradient boosting

Gradient boosting is a machine learning technique used for both classification and regression problems. The method constructs an ensemble of weak learners, typically decision trees, in a sequential manner. Each subsequent model is trained to minimize the loss function of the combined previous models by fitting to the gradient of the loss with respect to the current predictions.

On structured tabular data, gradient boosting algorithms often achieve performance improvements of approximately 2–5% over random forests. The method effectively captures non-linear relationships

even when shallow trees are used, while maintaining relatively strong predictive performance. Feature contributions can also be analyzed, allowing partial interpretability of the final model. Gradient boosting is computationally efficient for medium-sized datasets; however, it is more difficult to tune due to the larger number of hyperparameters that influence model behaviour.

3 Match outcome predictor

3.1 Obtaining the dataset

One of the advantages of choosing League of Legends as the game for match outcome prediction is the accessibility of the data. Through the Riot Developer Portal, it is possible to obtain either a 24-hour temporary API key or to register a product for private or commercial use, which removes the 24-hour limitation. In order to obtain an API key, the user is required to read and accept the General and Game Policies, Terms of Use, and Legal Notices. These documents prohibit the publication of any sensitive or personal data of League of Legends players, including player names and match identifiers. A personal API key allows the user to send up to 20 requests per second or 100 requests every two minutes to the Riot servers.

To gather all required information, we decided to create a custom match database instead of relying on existing datasets. For this purpose, a list of 374 players was collected based on their Riot IDs across different ranks from the Europe North and East (EUNE) server. The following APIs were used in the data collection pipeline: `account-v1`, which converts a Riot ID into a global Personal Universally Unique Identifier (PUUID); `summoner-v4`, which retrieves each player's current ranked solo queue tier and division; and `match-v5`, which provides access to up to 100 recent matches per player, along with information about match outcomes and second-by-second timeline events such as gold accumulation and monster kills.

The data collection process is handled by the `data.py` script. Its input consists of a list of Riot IDs stored as an array of strings in the format `["player1#tag", "player2#tag", ...]`, defined in the `players.py` file. For each Riot ID, the script calls the `account-v1` API to obtain the corresponding PUUID. For each PUUID, the `match-v5` API retrieves 80 unique match IDs, from which only ranked matches are selected, while other game modes are discarded. For each selected match ID, the timeline data are downloaded and parsed frame by frame, accumulating information on gold, experience, creep score (CS), champion kills, destroyed turrets, dragon kills, and Void Grub kills. The script then checks whether a given match is already present in the database; if not, the extracted data are appended to the `.csv` file. Each processed player is marked accordingly to prevent redundant processing during subsequent executions of the script.

A rate-limiting mechanism is implemented to comply with the API request constraints. When the request limit is exceeded, the script pauses execution for several seconds or minutes before issuing further requests.

For each match, the script generates 39 data columns, including the match outcome (binary indicator of blue team victory), match context (match ID, queue type, player rank and division), 15-minute statistics for both teams (gold, experience, kills, creep score, turrets, dragons, and Void Grubs), and draft-related information (champion IDs for each player, along with their champion classes, such as marksmen, fighter, or assasin, and assigned roles). The script appends new records to the existing dataset rather than overwriting it, allowing the dataset to be incrementally expanded by adding new players to the input list.

3.2 Data analysis

Due to the API request limitations, collecting a sufficient amount of data was a time-consuming process. In this study, the data collection script ran for approximately 17.5 hours and gathered information from 22 175 matches involving 374 players, resulting in a total of 221 750 champion entries (ten unique champions per match). The resulting dataset is well structured, containing no duplicate entries and no missing values. However, due to the Riot API terms of service it is prohibited to publish the dataset.

For the purpose of data analysis, the `visualize_data.py` script was developed. This script provides statistics on the total number of matches, queue distribution (solo queue versus flex queue), rank

distribution, overall win rate, and average team statistics at the 15-minute mark. It should be noted that players ranked in Iron, which is the lowest rank in the game, were excluded from the analysis.

The selected charts and graphs presented below were generated using the `visualize_data.py` script. Unfortunately, champion popularity varies over time, as the game undergoes balance updates approximately every two weeks, which adjust the relative strength of individual champions. These balance changes directly affect both champion popularity (stronger champions tend to be selected more frequently) and champion win rates (stronger champions typically achieve higher win rates). Since the collected matches span several months, temporal variations in champion strength introduce a slight bias into the dataset.

During data collection, the initial objective was to obtain a similar amount of data from each rank. In practice, however, the actual rank distribution resembles a normal distribution, with Silver being the most common rank. According to League of Graphs, approximately 22% of players are ranked Silver, while higher ranks are significantly less populated, with only 0.016% of players reaching the Challenger rank. Higher-ranked players tend to select different champions and exhibit different in-game statistics compared to lower-ranked players. To ensure that the trained models perform adequately across all ranks, players from higher ranks were intentionally overrepresented, while players from lower ranks were underrepresented. As a result, the presented graphs should not be interpreted as a faithful representation of the overall League of Legends player population.

1. **Top 20 most popular champions**

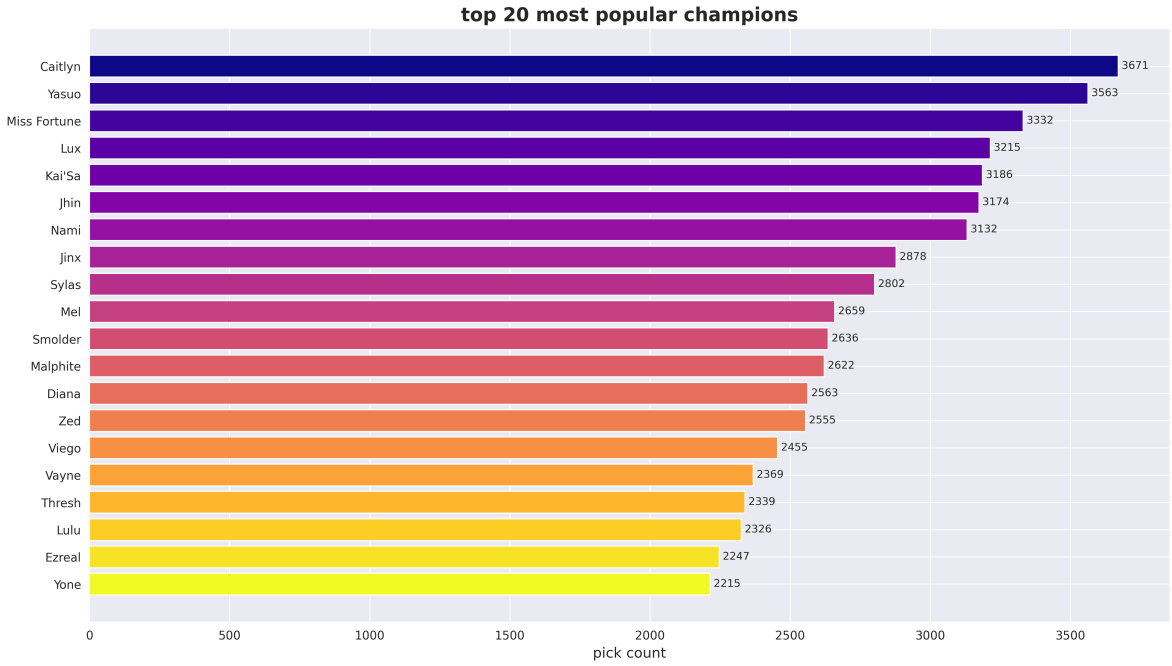


Figure 1: Most popular champions in the dataset

The most popular champions in the dataset can be compared with those observed in the actual game. According to the League of Graphs website, the following champions were the most popular on the Europe North and East server during the same period. The reference dataset was collected between 28 and 30 December 2025 and consists of 181 427 ranked matches:

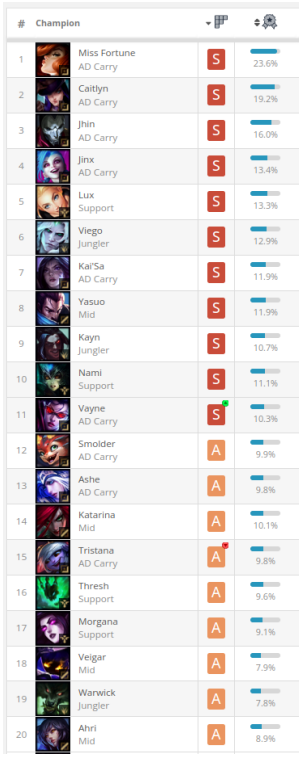


Figure 2: Most popular champions according to League of Graphs

Out of the 20 champions shown, 12 appear in both lists. This indicates that, despite not containing the most recent data, the constructed dataset remains a reasonably accurate representation of the actual game state.

2. **Top 10 champions by win rate**

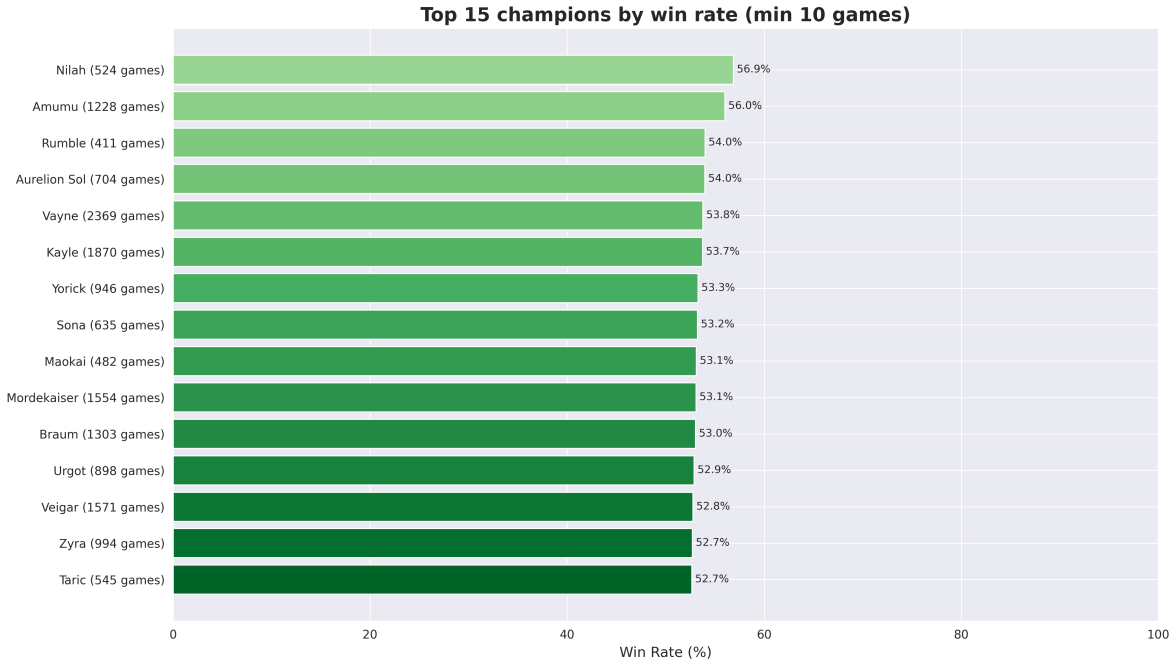
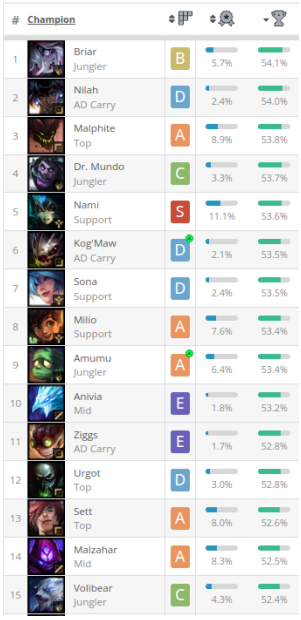


Figure 3: Champions with the highest win rates in the dataset

The top 10 champions by win rate according to League of Graphs are shown below:



#	Champion			
1	Briar Jungler	B	5.7%	54.1%
2	Nilah AD Carry	D	2.4%	54.0%
3	Malphite Top	A	8.9%	53.8%
4	Dr. Mundo Jungler	C	3.3%	53.7%
5	Nami Support	S	11.1%	53.6%
6	Kog'Maw AD Carry	D	2.1%	53.5%
7	Sona Support	D	2.4%	53.5%
8	Milio Support	A	7.6%	53.4%
9	Amumu Jungler	A	6.4%	53.4%
10	Anivia Mid	E	1.8%	53.2%
11	Ziggs AD Carry	E	1.7%	52.8%
12	Urgot Top	D	3.0%	52.8%
13	Sett Top	A	8.0%	52.6%
14	Malzahar Mid	A	8.3%	52.5%
15	Volibear Jungler	C	4.3%	52.4%

Figure 4: Champions with the highest win rates according to League of Graphs

Out of 15 champions considered across both lists, only four appear in both. This discrepancy indicates that the dataset is not sufficiently reliable for estimating individual champion win rates and should not be used for this purpose when selecting training features or model parameters.

3. **Rank distribution**

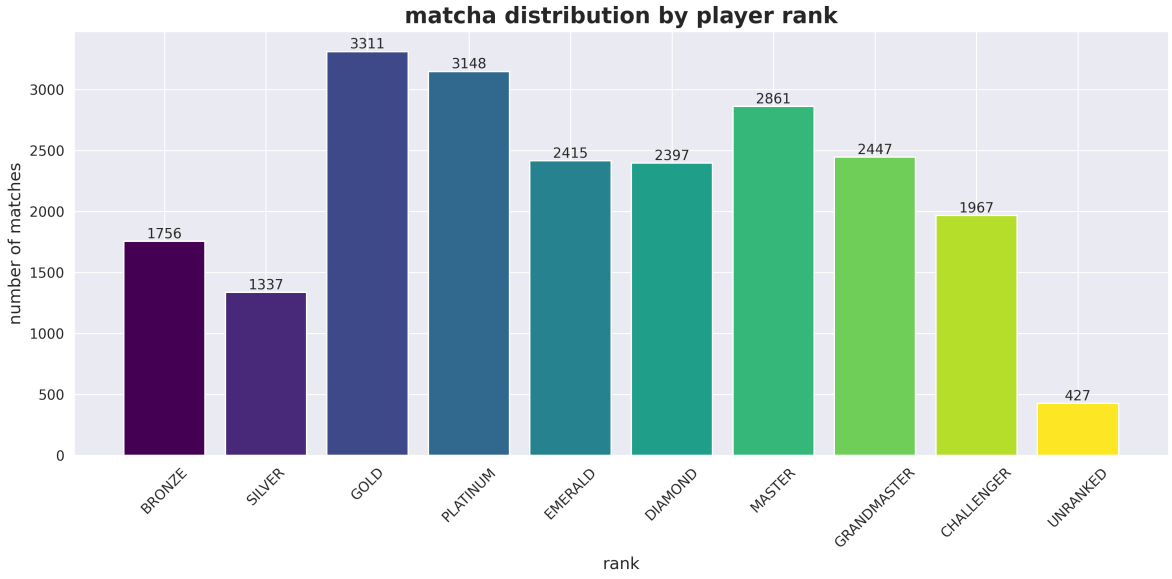


Figure 5: Rank distribution in the dataset

This graph illustrates the number of matches from each rank included in the dataset.

Other parameters, such as class distribution or game stats distribution are available inside of Github repository.

3.3 Preprocessing

The next step in the development of the predictive model is preprocessing the data, transforming raw string-based information into a numerical feature table. For this purpose, the `preprocessing.py` script was developed. The script first loads the dataset and removes matches that include more than three Void Grubs, as the maximum number of Void Grubs per match was reduced from six to three in May 2025. This criterion therefore serves as an effective indicator for filtering out very old matches. Subsequently, the match statistics obtained from `match_data.py` are converted into differential values at the 15-minute mark, as such representations are translation-invariant and better capture relative team advantage. The core differentials include gold difference, kill difference, creep score (CS) difference, turret difference, dragon difference, Void Grub difference, and experience difference. An additional engineered feature, referred to as the `early_impact_score`, is then introduced. This metric is defined as a weighted average of the aforementioned differentials, with weights selected based on domain knowledge derived from gameplay experience. A queue-type indicator is also added, with a value of 1 representing ranked solo queue and 0 representing ranked flex queue. Player rank is encoded into categorical tiers, where Bronze and Silver are treated as low ranks, Gold and Platinum as mid ranks, Emerald and Diamond as high ranks, and Master, Grandmaster, and Challenger as elite ranks. Furthermore, information regarding champion classes for both teams is incorporated into the feature set. The resulting feature matrix is constructed alongside a separate label vector indicating whether the blue team won the match. Finally, the dataset is split into training and testing subsets using an 80/20 ratio. The processed data are then stored in appropriate locations, organized according to queue type and rank tier.

3.4 Model

Finally, the `model.py` script was developed to train the predictive model on the preprocessed data and evaluate its performance. For this task, the gradient-boosted decision tree algorithm XGBoost was selected, as it is widely regarded as state of the art for medium-sized, structured tabular datasets. The desired output of the model is a calibrated probability of the blue team winning the match. Model performance is evaluated using log-loss, the receiver operating characteristic (ROC) curve, the area under the curve (AUC), and the confusion matrix, allowing both accuracy and prediction quality to be assessed.

The script supports two training modes: a mode with fixed, manually tuned hyperparameters and an automatic tuning mode that employs randomized search to identify near-optimal hyperparameter configurations.

The XGBoost models are defined using the following hyperparameters:

1. `max_depth` – maximum depth of individual trees
2. `gamma` – minimum loss reduction required to perform an additional split at a leaf node
3. `min_child_weight` – minimum sum of instance weights required in a leaf
4. `learning_rate` – scaling factor controlling the contribution of each tree
5. `num_boost_round` – total number of boosting iterations (trees)
6. `subsample` – fraction of training instances sampled for each tree
7. `colsample_bytree` – fraction of features sampled for each tree
8. `tree_method = hist` – histogram-based tree construction
9. `objective = binary:logistic` – binary classification objective

The `LoLPredictor` class constitutes the core component of the script. It handles data conversion by transforming the input features into the `xgb.DMatrix` format, which is optimized for XGBoost. The model then constructs an ensemble of decision trees, with early stopping enabled to terminate training if no improvement is observed for 30 consecutive boosting rounds, thereby reducing the risk of overfitting. Following training, the class evaluates the model's performance using the selected metrics.

Hyperparameter autotuning is implemented using a randomized search strategy, which samples random combinations from predefined parameter ranges. This approach is typically sufficient to identify well-performing configurations while maintaining relatively low computational cost. Additionally, the script determines which input features have the greatest impact on the predicted match outcome.

3.5 Training multiple models

Because match dynamics differ substantially across rank tiers and ranked queue types, multiple specialized models were trained. In total, ten models were developed based on combinations of rank tier and queue type:

1. all ranks, both queues (full dataset)
2. all ranks, solo queue only
3. low ranks, solo queue
4. mid ranks, solo queue
5. high ranks, solo queue
6. elite ranks, solo queue
7. low ranks, both queues
8. mid ranks, both queues
9. high ranks, both queues
10. elite ranks, both queues

To support this process, command-line argument handling was added to both `preprocessing.py` and `model.py`, enabling data preprocessing and model training to be restricted to specific subsets, such as solo queue matches in a particular rank tier. Additionally, the scripts `multi_preprocessing.py` and `multi_models.py` were introduced to automate the creation of datasets and models for each of the specified configurations.

3.6 Visualising model performance

Finally, the `visualize_models.py` script was created to present the performance of the trained models in a graphical form. The script loads all trained models, reconstructs them, performs predictions on the corresponding datasets, and computes evaluation metrics such as accuracy and AUC. Based on these results, charts illustrating accuracy by rank tier, comparisons between queue types, and AUC scores are generated and saved to the `/charts` directory alongside the data analysis figures.

4 Training results

In order to obtain all desired models, the corresponding datasets were first generated using the `multi_preprocessing.py` script. Subsequently, the `multi_models.py` script was executed in autotuning mode. Below is a snippet of terminal output illustrating the training and evaluation process for the first model, trained on the full dataset:

```
================================================================
training model: all data
================================================================
loaded data from: data/all/

============================================================
TRAINING CONFIGURATION
============================================================
tier filter: ALL
```

```
queue filter: ALL
training samples: 14787
test samples: 3697
============================================================

================================================
your autotune is starting now
================================================
looking for best parameters using randomized search. sit back and relax for 5-10 minutes
Fitting 3 folds for each of 30 candidates, totalling 90 fits

best parameters found:
  subsample: 0.7
  reg_lambda: 0.5
  min_child_weight: 2
  max_depth: 3
  learning_rate: 0.05
  gamma: 2.0
  colsample_bytree: 0.9
best cross-validation auc: 0.843

[0]   train-logloss:0.67683   eval-logloss:0.67799
[50]  train-logloss:0.49080   eval-logloss:0.51372
[100] train-logloss:0.47910   eval-logloss:0.50771
[150] train-logloss:0.47338   eval-logloss:0.50691
[200] train-logloss:0.46928   eval-logloss:0.50606
[232] train-logloss:0.46681   eval-logloss:0.50637

================================================
model performance on test set:
accuracy: 74.52%
auc score: 0.830

confusion matrix:
                 predicted
                 loss | win
actual loss     1418 |   453
       win       489 |  1337

  true negatives  (correct loss predictions): 1418
  false positives (predicted win, was loss):  453
  false negatives (predicted loss, was win):  489
  true positives  (correct win predictions):  1337
================================================

top features by importance (gain):
  early_impact_score: 131.27
  gold_diff_15: 37.11
  dragon_diff_15: 18.90
  experience_diff_15: 11.98
  grub_diff_15: 9.90
  cs_diff_15: 8.74
  red_assassin_count: 7.85
  blue_assassin_count: 7.29
  tower_diff_15: 6.58
  blue_mage_count: 6.19
```

```
model saved to models/lol_predictor.json
```

This model triggered the early stopping mechanism, terminating training at boosting round 232, as no further improvement in validation performance was observed. Autotuning determined `max_depth` equal to 3, `gamma` equal to 2.0, and `learning_rate` equal to 0.05 as the most effective hyperparameter values for this dataset. The resulting classification accuracy reached 74.52%, satisfying the predefined performance requirements. The AUC score of 0.830 indicates that the model is effective at distinguishing between winning and losing teams.

Analysis of the confusion matrix reveals that the model is slightly more likely to miss a comeback scenario than to misclassify a throw, where a team with an early advantage ultimately loses the match. Overall, the confusion matrix demonstrates a balanced error distribution.

The model also ranked input features according to their importance, measured by gain. The most influential feature by a large margin was the artificially introduced `early_impact_score`, which achieved a gain approximately four times higher than that of the second most important feature. This result suggests that the selected feature weights were appropriate. The next most important features were gold difference, dragon difference, and the presence of assassin-class champions. Notably, dragon objectives were found to be approximately twice as impactful as Void Grubs, indicating that dragon control plays a more significant role in securing victory.

The achieved performance demonstrates that League of Legends match outcomes are highly predictable as early as 15 minutes into the game. Early map control and gold accumulation have a dominant influence on the final result, rewarding teams that adopt a proactive early-game strategy.

4.1 Performance comparison of different models

Using the `visualize_models.py` script, a quantitative comparison of all trained models was obtained:

```
================================================================================
MODEL PERFORMANCE SUMMARY
================================================================================
tier                                    queue      accuracy     auc      samples
--------------------------------------------------------------------------------
ALL                                     Both        74.52%      0.830   3697
ELITE                                   Both        74.45%      0.827   1366
HIGH                                    Both        75.53%      0.845   756
LOW                                     Both        77.23%      0.860   470
MID                                     Both        74.06%      0.826   1037
ALL                                     SoloQ       75.10%      0.842   2257
ELITE                                   SoloQ       75.03%      0.844   793
HIGH                                    SoloQ       77.51%      0.860   489
LOW                                     SoloQ       74.92%      0.842   299
MID                                     SoloQ       78.33%      0.868   669
================================================================================

BEST MODELS:
  highest accuracy: MID (SoloQ) - 78.33%
  highest auc:      MID (SoloQ) - 0.868
```

The accuracy of all models consistently remained within the 70–80% range. The model that clearly outperformed the others was the mid-rank tier solo queue model, which achieved an accuracy of 78.33% and an AUC score of 0.868. This result suggests that players in this tier are generally capable of converting early-game advantages into victories, in contrast to lower-rank players, while still being less proficient at recovering from early disadvantages compared to higher-rank players.

Across all rank tiers, models trained exclusively on solo queue data consistently outperformed models trained on data from both queues. Interestingly, in elite-rank matches, early-game performance appears to be slightly less decisive for the final match outcome. Overall, the imperfect accuracy of all models indicates that early-game advantage does not guarantee victory, highlighting the importance of continued strategic play even when a team falls behind.

The graphs comparing model performance across different configurations are presented below:

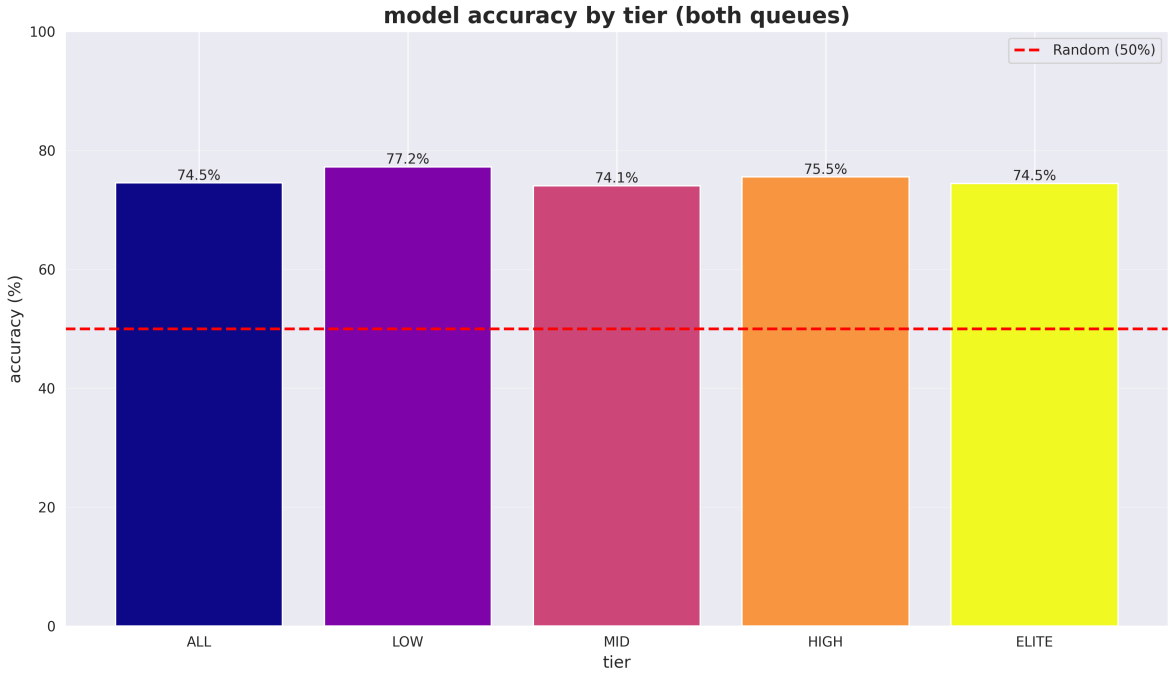


Figure 6: Model accuracy by rank tier for both queues

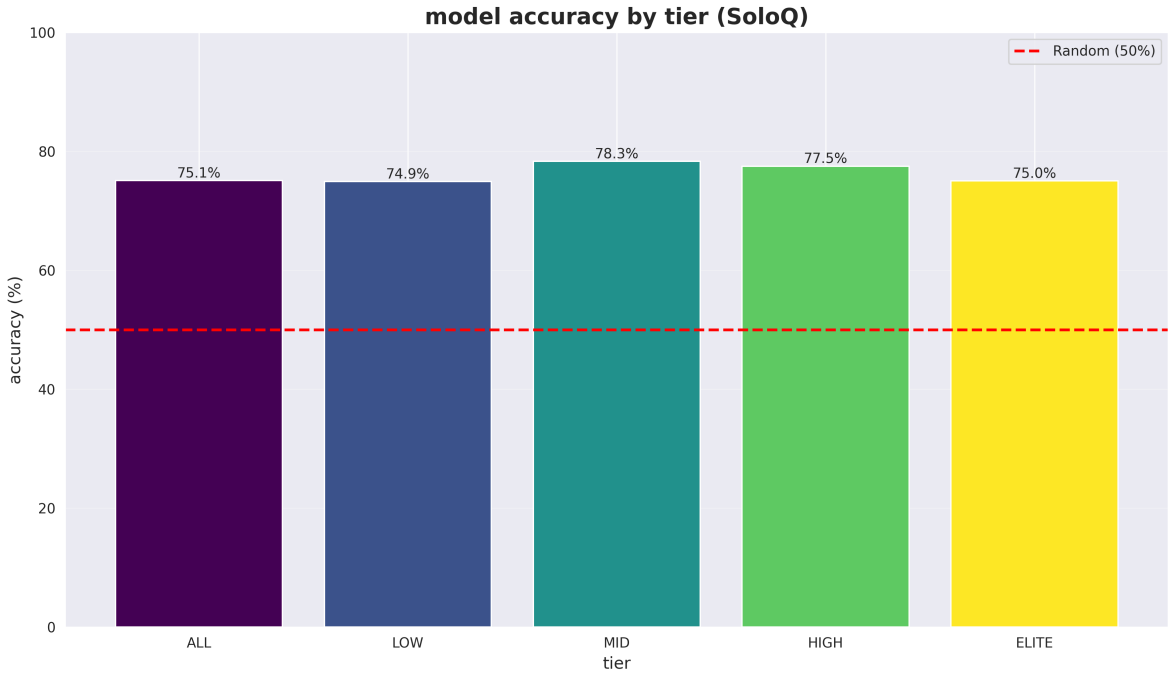


Figure 7: Model accuracy by rank tier for solo queue

The obtained accuracy remains relatively consistent across all rank tiers and is significantly higher than 50%, which corresponds to random guessing.

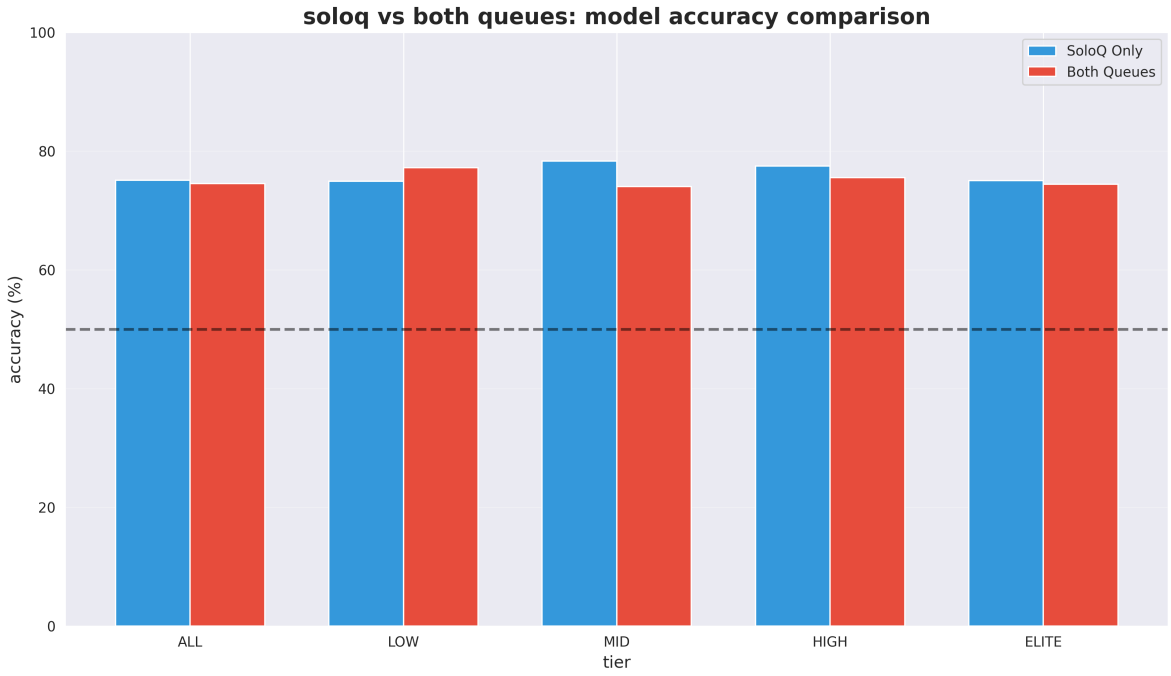


Figure 8: Comparison of model accuracy between solo queue and both queues

Models trained exclusively on solo queue data outperformed the general models in all rank tiers except the low-rank tier, suggesting that early-game performance has a stronger influence on match outcomes in solo queue games.

5 Conclusion

Recent advances in machine learning techniques have enabled the identification of meaningful patterns across many aspects of digital activity, allowing for the prediction of future system states. Online competitive games, such as *League of Legends*, constitute no exception to this trend. The models developed in this work were capable of predicting match outcomes at the 15-minute mark with satisfactory accuracy, despite the inherently complex and highly variable nature of the game.

The obtained results demonstrate that early-game information contains significant predictive value and allows for the identification of key factors influencing the final match outcome. Moreover, the proposed approach shows potential for further extension, including real-time prediction of live matches and adaptation to additional gameplay features or competitive environments.