ECAMANIA

by Edwin Creten

Introduction

In 2025, ECAM launched a bold initiative: the creation of its very own esports team — *Ecamania*. Backed by the school director with the promise of an unlimited budget, the objective is simple but ambitious: bring the best players in the world to represent ECAM in the competitive League of Legends scene.

To support this project, I was recruited as the team's data analyst. My mission? Use in-game statistics from professional matches to guide three crucial decisions for the team:

1. **Predict the outcome of a game based on a single player's performance.**
2. **Identify the most important performance metrics that increase a player's chance to win.**
3. **Rank the most talented players to recruit for the next season.**

To achieve this, I used a dataset containing detailed stats from pro-level League of Legends games and designed a full machine learning pipeline — from preprocessing and feature engineering to model comparison and player ranking.

Data visualization and pre-processing

I started by visualizing the game_players_stats.csv file to get an overview of the data I was working with. This file contains information such as the game ID, player ID, team ID, player role, win status, and end-of-game statistics.

My first step was to look at global statistics to check for any abnormal values that could be cleaned. Since I've (maybe a bit too much) played the game myself, I had a pretty good idea of what values seemed suspicious. Right away, I noticed issues with the game_length, wards_placed, and level features.

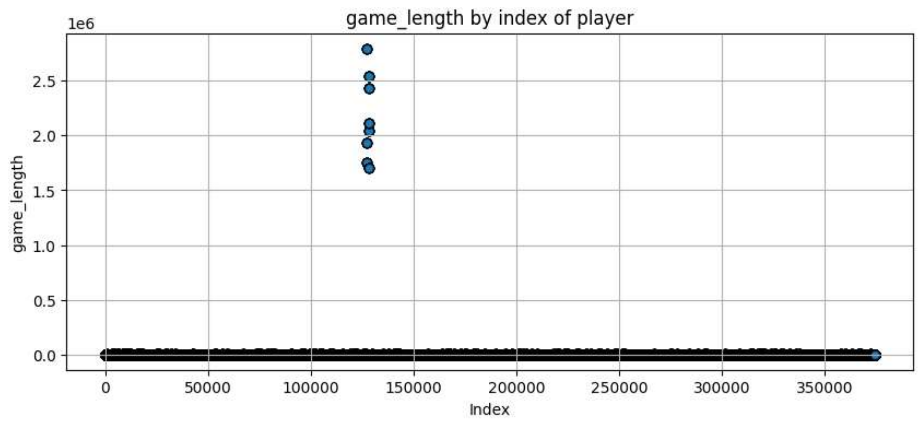
Data visualization confirmed it: some matches were way too long, while others ended way too early — with players still being very low level or possibly AFK. I cleaned the dataset by removing games that were too long, games where players ended at an unusually low level, and blocks of matches where no wards were placed, which looked suspicious.

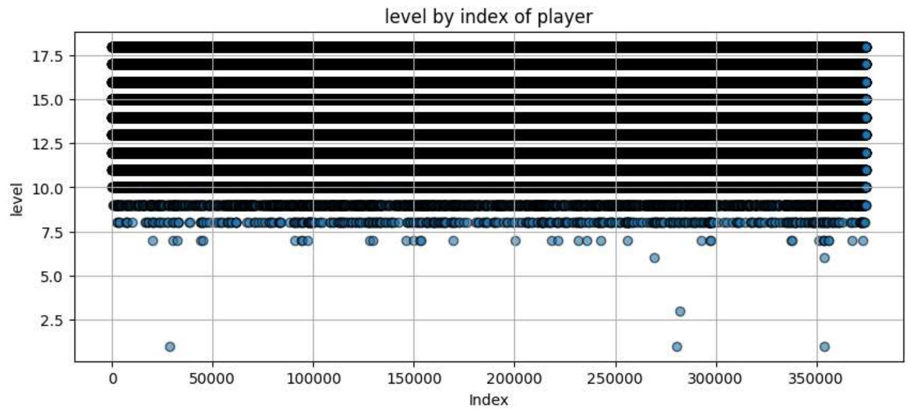
After this initial cleaning, I still found some matches where players had placed zero wards. That's not ideal in pro play, but upon further inspection, I found 30 games where players had 0 wards but were otherwise very active based on their other stats. Fair enough — I kept those.

Next, I examined features with extreme values for damage dealt and taken. However, I didn't find any inconsistencies there. High damage values usually come from late-game hyper-carry champions, while high damage taken is typical of tanky champions. Nothing out of the ordinary.

I then converted the win column from True/False to 1/0 to make it easier to handle in machine learning models. Finally, I checked for null values: only a few team acronyms were missing, which I could safely drop.

As a last step, I converted the player roles into integer values. This helps simplify filtering and modeling processes later on — especially when separating the dataset by role or feeding it into models that expect purely numerical input.





Models

I built separate win prediction models for each role using Random Forest classifiers, based on end-of-game statistics. To do this, I extracted a cleaned dataset for each specific role in the game — top, jungle, mid, ADC, and support.

This separation is important because what defines a good performance for a top laner is completely different from what defines a good support.

Dataset creation

Before training the models, I normalized the game duration feature by converting it into minutes. Then, I created new time-relative features (e.g., damage per minute, gold per minute) to allow for fair comparison between games of different lengths.

The target label was defined as the win feature, which indicates whether the player's team won the game. I then split each role-specific dataset into training, validation, and test sets for model development and evaluation

Random forest

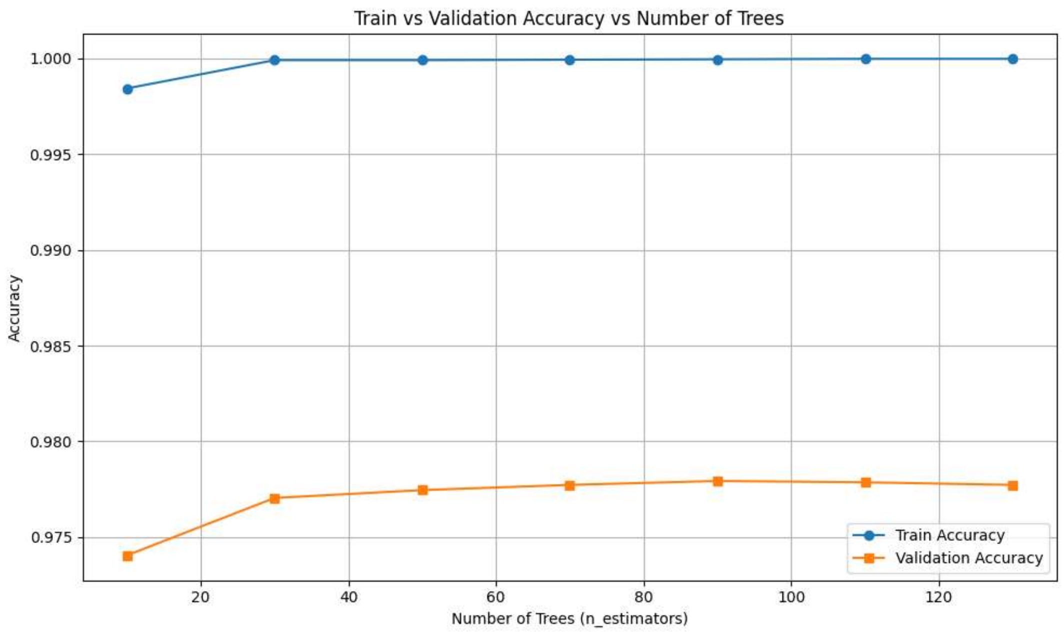
To tune my Random Forest models, I first focused on two key hyperparameters: the number of trees (n_estimators) and the maximum depth of each tree (max_depth). Instead of guessing these values, I tested them systematically to find what works best for my data.

Choosing the right number of trees

I ran several models by varying the number of trees from 10 to 150, with steps of 20. For each value, I trained the model and checked its accuracy on both the training set and the validation set.

The idea is simple: too few trees and the model underfits; too many, and I risk unnecessary complexity and longer training time. I plotted the train and validation accuracy to visually spot the sweet spot — where validation accuracy was high and stable, without overfitting.

Based on the results, the best performance on the validation set was reached around 90 trees, so I picked that as my final choice for n_estimators.



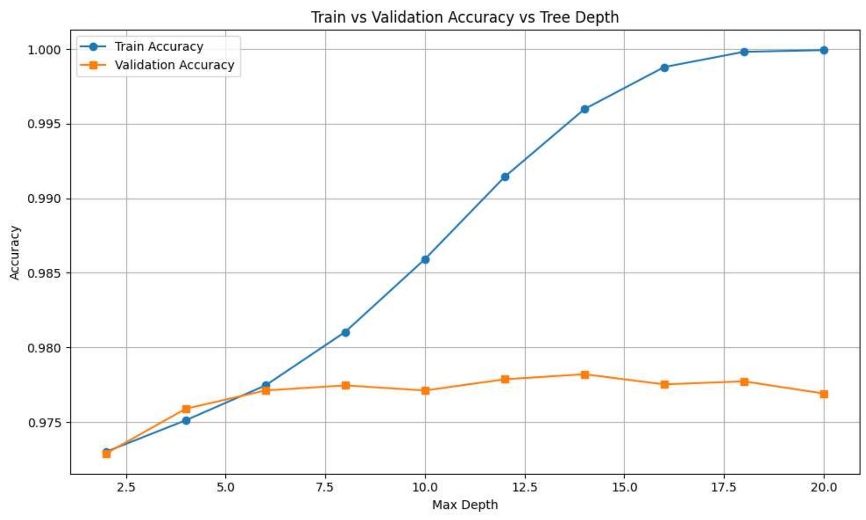
I could have chosen a smaller n_estimators value to reduce execution time, as it wouldn't have significantly impacted the model's accuracy.

Choosing the right depth

Once the number of trees was set, I tuned the maximum depth of the trees by trying values from 2 to 20. The logic here was the same: shallow trees might miss patterns (underfitting), while very deep trees might memorize the training data (overfitting).

Again, I plotted the training and validation accuracy to get a clear view. The validation accuracy peaked at a depth of 14 but I deliberately chose to limit the depth to 7.

As we can see on the graph, the improvement in validation accuracy beyond depth 7 is minimal — it stays mostly flat, with only small fluctuations.



This two-step tuning process allowed me to build a model that generalizes well without being overly complex.

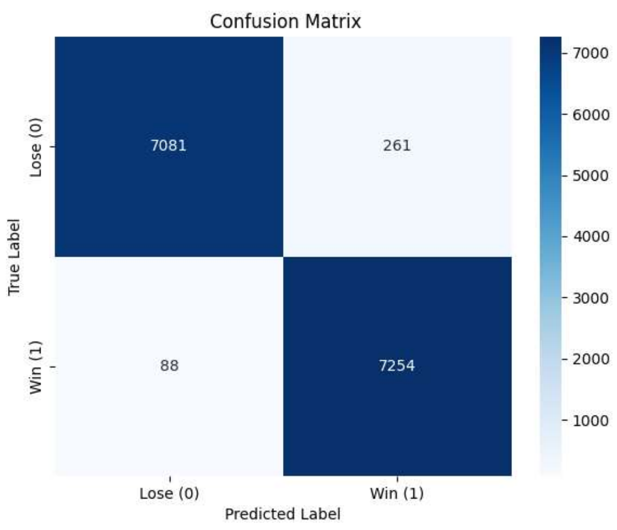
Final training

After selecting the best hyperparameters (n_estimators = 90 and max_depth = 7), I trained the final Random Forest model on the training set. Once trained, I used it to predict the outcome (win or lose) on the test set — which the model had never seen before.

To evaluate the performance, I computed the accuracy, which gave me a score of 0.9762 . This gives a good first idea of the model's general performance.

But accuracy alone doesn't tell the full story. So I also printed a classification report including precision, recall, and F1-score for both classes (win and lose). This gave a more detailed view of how well the model handles each outcome.

Finally, I plotted a confusion matrix to visually check how many correct predictions the model made, and where it made mistakes. Most of the predictions were correct, with a few misclassifications — which is expected.

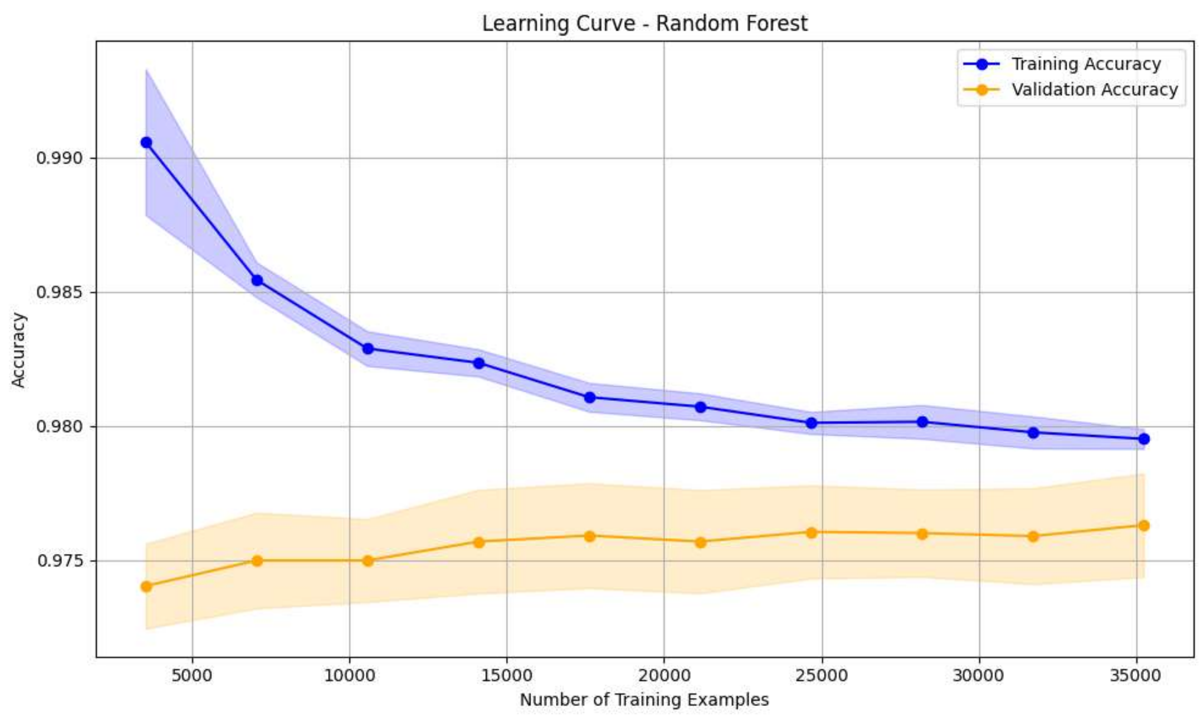


All in all, the final model performs well and generalizes correctly on unseen data. It's now ready to be used to evaluate player performance based on their match stats.

Learning curve

To better understand how my model generalizes and whether adding more data would help, I plotted a learning curve. This graph shows how the training and validation accuracy evolve as we increase the number of training examples.

I used scikit-learn's learning_curve function with 5-fold cross-validation and gradually increased the training size from 10% to 100% of the available data. For each size, the model was trained and evaluated, and I computed the mean accuracy and standard deviation for both the training and validation scores.



The plot shows a few important things:

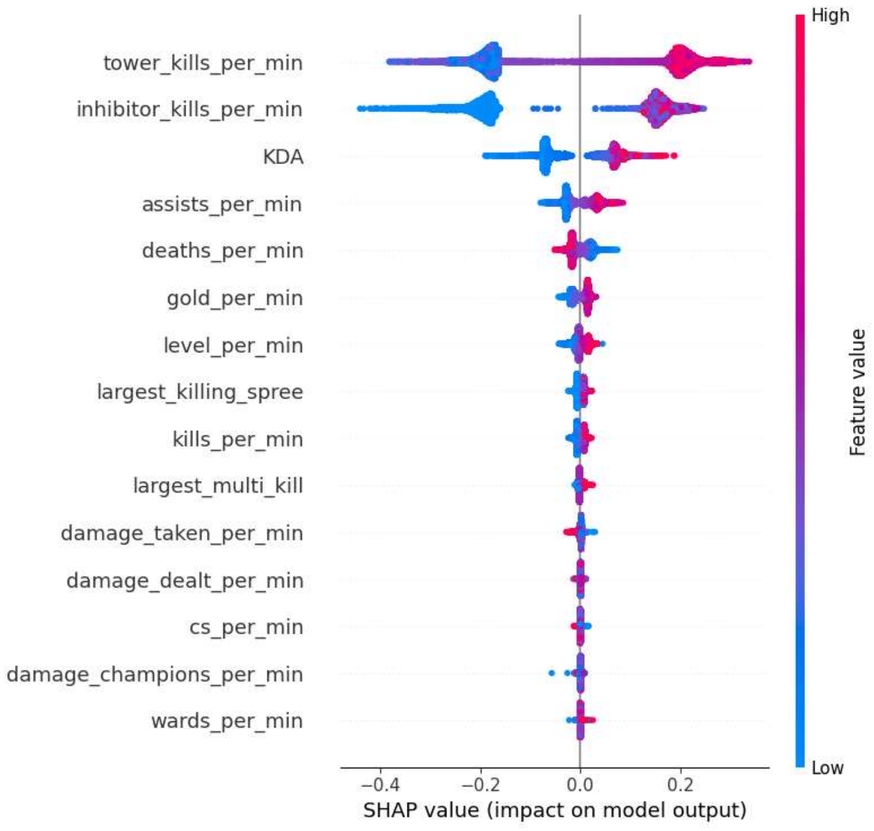
- Training accuracy starts very high (above 99%) but decreases as the dataset grows, which is expected: the more data the model sees, the harder it is to overfit.
- Validation accuracy stays relatively stable around 97.5%, with only minor variations.

From this curve, we can also see that adding more data probably wouldn't boost performance significantly, as the validation accuracy has already plateaued. The model is learning well and generalizing reasonably.

Feature Importance Analysis — SHAP Values

To understand which statistics had the biggest impact on the model's predictions, I used SHAP values (SHapley Additive exPlanations). This method helps to explain how much each feature contributes to the prediction for each individual example.

I used the TreeExplainer from the SHAP library, which is specifically optimized for tree-based models like Random Forests. Since we're predicting a binary outcome (win or lose), SHAP gives us a 3D array: one set of SHAP values for each class (0 and 1). I focused on class **1**, which corresponds to predicting a win.



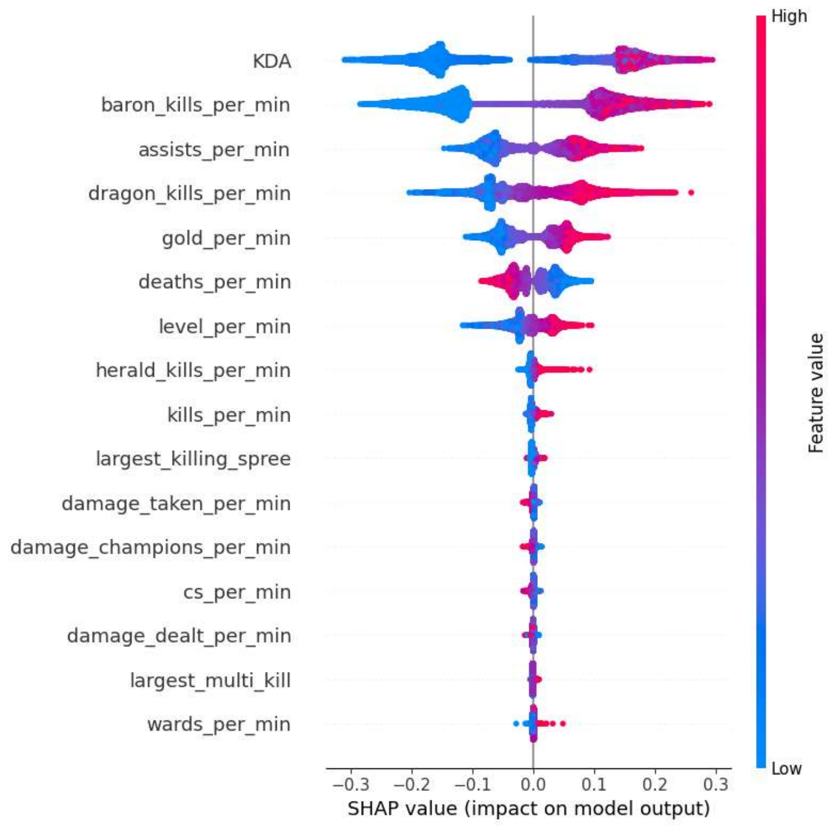
The beeswarm plot shows the impact of each feature on the model's output, across all validation samples. Each dot represents a player, and its horizontal position shows how much that feature pushed the prediction toward win (right) or loss (left). The color indicates the actual value of the feature for that sample (blue = low, red = high).

From the plot of the Top lane model, we can clearly see that:

- **Tower kills per minute and inhibitor kills per minute are** the most impactful features. That makes perfect sense — destroying objectives directly increases your chances to win.
- KDA (kill/death/assist ratio) also plays an important role, as expected.
- Assists per minute, gold per minute, and deaths per minute follow closely — all solid indicators of team contribution and game control.
- Vision control (wards_per_min) and farming (cs_per_min) had surprisingly little impact in this model.

Overall, the SHAP values helped validate that the model is relying on sensible features, and not on noise or irrelevant stats. They also give practical insight into what players can focus on to maximize their win probability according to their respective role. In all my models I give very high value for destroying objectives such turrets, inhibitor, dragons and barons.

Here's the jungler SHAP that prioritize neutral objectives.



Team Making

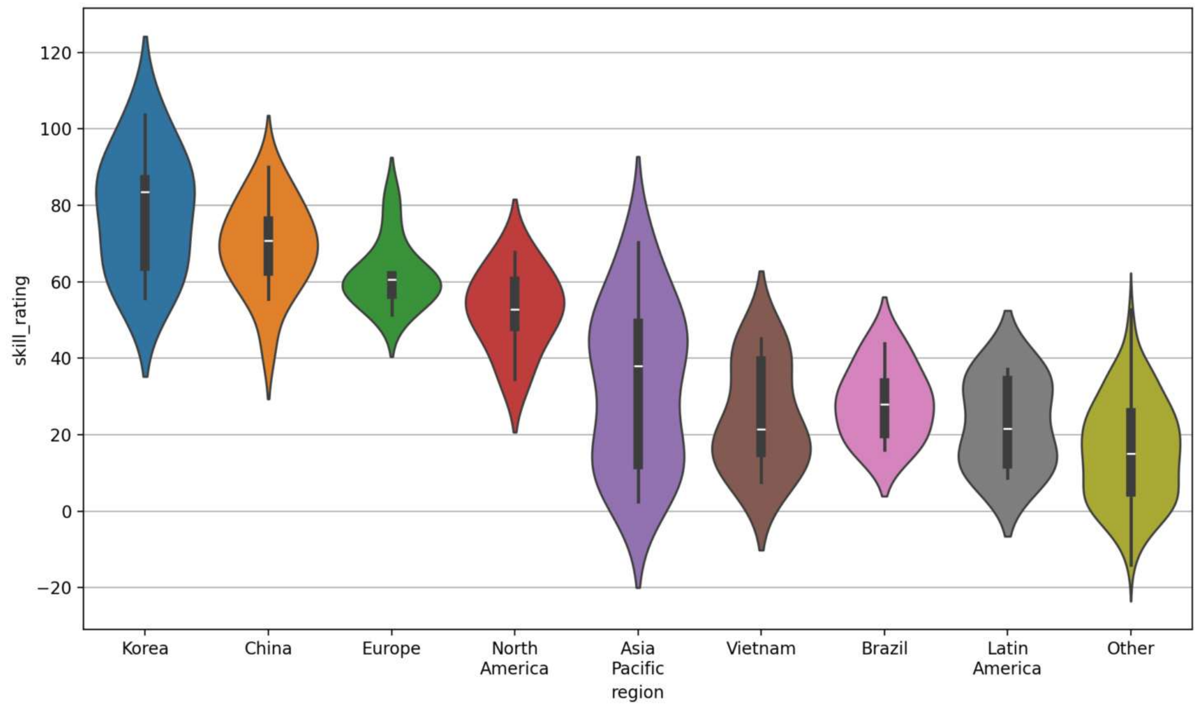
The final major goal of the project is to identify the best players in the dataset to build our dream team. When training my model, I deliberately excluded out-of-game features — even though they can have a huge impact.

Data preprocessing

To build the final player ranking across all roles, I first restored the identification columns in each role-specific dataset. This allowed me to merge them into a single unified dataset and link each player performance back to the original game_metadata.csv file — which contains useful information such as the league and date of each match.

I started by filtering out non-major leagues. To do that, I arbitrarily removed all leagues with fewer than 500 games, assuming they didn't have a strong enough sample size to be reliable.

Next, I assigned weights to each league, giving more importance to regions known for their high level of competition, such as Korea, China, and international tournaments. These weights were inspired by the league distribution used in the *PandaSkill* paper. The goal was to give more credit to good performances in tougher environments, and not treat a win in a top-tier match the same as one in a minor showmatch.



I also added a time-based weight to the games. After all, we're building a team for 2025, so we want players who are currently strong — not those who peaked years ago.

To reflect this, I used an exponential decay function that gives less importance to older games.

$$timeweight = e^{-\lambda \cdot time}$$

I used a decay rate λ = 0.002, which I found gave a nice balance

Finally, I introduced a weight based on the number of games played by each player. The idea is simple: we want to prioritize established players who have shown consistent performance over many matches, rather than players with only one or two lucky games.

To do that, I used a logarithmic scaling function

In the end, I computed what I called the performance score for each player in each match. It combines their predicted win probability (from the model) with all the weights I had previously defined$Performance\ score = Win\ \Pr o\, babilit\acute{e} \cdot (w_{time} + w_{league} + w_{games})$

Final team composition

Finally, I'm proud to say that I was able to rank the top-performing players in the world, across all leagues and roles. The results I obtained are surprisingly close to those published in the *PandaSkill* paper, which confirms the reliability of my methodology.

Based on my model's predictions and the weighted performance scores, here is the final dream team I would recruit for Ecamania:

- Top: Bin (BLG)
- Jungle: Canyon (Gen.G)
- Mid: Chovy (Gen.G)
- ADC: Ruler (JDG)
- Support: Keria (T1)

These players consistently showed strong in-game performances, high win probabilities, and played in the most competitive environments — making them ideal picks for a team aiming to dominate the international scene.