Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Group Project Report (CSU44061)

Classifying A Players Match Performance In The Online Video Game: ‘League of Legends’

Cornel Jonathan Cicai, Student: 19335265

Micheal Adebusuyi, Student: 16321842

Terlo Akintola, Student: 18326179

02/12/2022

Disclaimer: This project is different from the originally proposed, as we were requested to pick another.

https://github.com/League-of-legends-predictor/match-performance-ranker.git

Introduction

i) Context

The game 'League of Legends' (LoL) is a popular online video game where five players battle together as a team against another team of five. In this 5v5 format, each player is assigned a specific role.

These roles are commonly referred to as Top, Jungle, Mid, Bot and Support. Each role fulfils different duties to try and aid their team to victory.

Players can choose to play in a 'Ranked' game, which will affect their 'Rank' as a player. There are six main ranks: Iron, Bronze, Silver, Gold, Platinum and Diamond.

ii) Aim

We wanted to create a model that would allow players to analyse their performance and label it with a Rank.

This would be useful for players to affirm their skill level, and to see if they are improving or not at the game. This could also be used to predict a players current or future rank, if enough games were analysed and an aggregate Rank was calculated.

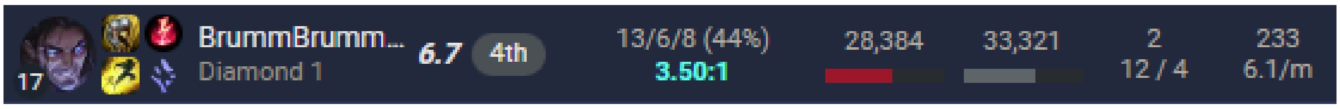
It could also be used for 'Matchmaking Systems', systems in charge of finding players of equal skill level to play together, to make the match as fair as possible.

iii) Model Input & Outputs

After a match, a player's performance is represented by various 'post-match statistics', which are various statistics such as their 'Kills/Deaths/Assists' score and Damage Dealt.

We will take these post-match statistics and engineer features using them, to use as inputs for our models (a kNN Classifier and a Random Forest Classifier).

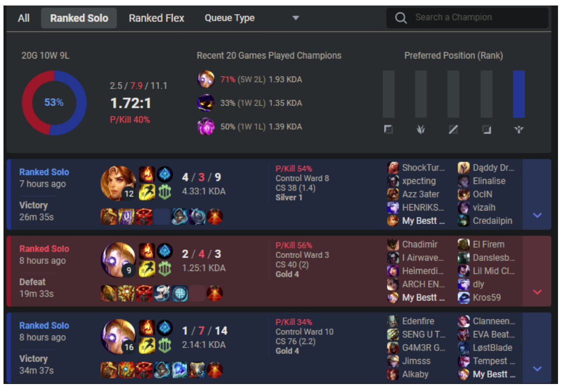
The models will label the players performance as a Rank, signifying which level they performed at for that particular game.


17 BrummBrumm... Diamond 1 6.7 4th 13/6/8 (44%) 3.50:1 28,384 33,321 2 12 / 4 233 6.1/m

Dataset & Features

i) Gathering Of Data

We needed samples of post-match statistics for each rank we would consider. One sample should represent one player's performance in a match. It contained information like what role they played, what their post-match statistics were, and what Rank they were at the time they played the match.


All | Ranked Solo | Ranked Flex | Queue Type | Search a Champion

20G 10W 9L — 53% — 2.5 / 7.9 / 11.1 — 1.72:1 — P/Kill 40%

Recent 20 Games Played Champions
71% (5W 2L) 1.93 KDA
33% (1W 2L) 1.35 KDA
50% (1W 1L) 1.39 KDA

Preferred Position (Rank)

Ranked Solo, 7 hours ago, Victory, 26m 35s — 12 — 4 / 3 / 9, 4.33:1 KDA — P/Kill 54%, Control Ward 8, CS 38 (1.4), Silver 1 — ShockTur..., xpecting, Azz 3ater, HENRIKS..., My Bestt ... / Daddy Dr..., Elinalise, OclN, vizaih, Credailpin

Ranked Solo, 8 hours ago, Defeat, 19m 33s — 9 — 2 / 4 / 3, 1.25:1 KDA — P/Kill 56%, Control Ward 3, CS 40 (2), Gold 4 — Chadimìr, I Airwave..., Heìmerdi..., ARCH EN..., My Bestt ... / El Firem, Danslesb..., Lil Mid Cl..., dly, Kros59

Ranked Solo, 8 hours ago, Victory, 34m 37s — 16 — 1 / 7 / 14, 2.14:1 KDA — P/Kill 34%, Control Ward 10, CS 76 (2.2), Gold 4 — Edenfire, SENG U T..., G4M3R G..., Jimsss, Alkaby / Clanneen..., EVA Beat..., LøstBlade, Tempest ..., My Bestt ...

There is a website called OP.GG that allows users to to search up a LoL profile in a specific region (there are multiple server regions around the world) and view their profile.

We decided to create a web scraper that would go on OP.GG, search for players in a specific Rank, look at their match history and extract samples of players that met certain criteria: the match had to have taken place one or two months ago (the Ranked season ended on Nov 14th, and players don't really play at their best after the season ends), the match had to not be a ‘Remake’ (‘Remade’ matches were matches that did not actually take place due to a player leaving in the early stages of the match), the player being inspected had to be the right Rank (the one being searched for) at the time the match was played, and the player should not have been ‘AFK’ (inactive) or ‘Trolling/Griefing’ (actively trying to lose the game because they were annoyed).

We aimed to get 20k samples per Rank (4k per role in a Rank), as we weren’t rate limited by an API, so we could get as many samples as we wanted. Since we were considering six ranks, that gave us a total of 120k raw samples.

We decided not to use the Riot Api ourselves for three reasons: it was not able to tell us the players rank at the time a game was played in the past (it could only give the players current rank), it was extremely rate limited, and we had enough data from OP.GG anyway.
We also wanted the experience of having to scrape raw data from the internet, as normally there is not an API that would just hand out data to you.

ii) Cleaning Of Data

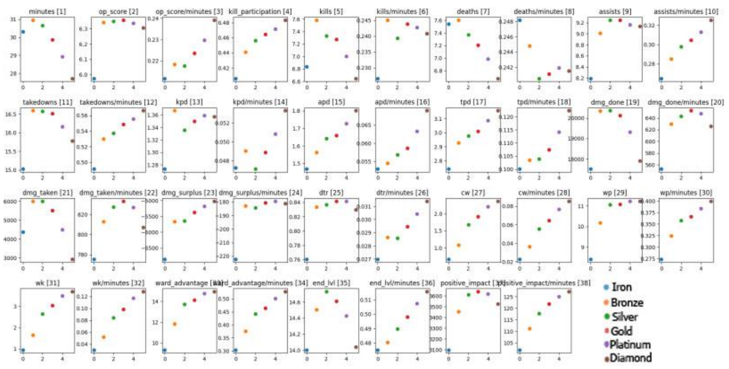
The web scraper provided us with relatively clean data, as it didn’t collect samples that didn’t meet our criteria outlined above. However, there were still samples in a given Rank where players overperformed or underperformed during their game (out of luck or due to the fact that they had good/bad teammates), meaning that their label wasn’t correct - they didn’t really perform at their usual performance. This could pose an issue for our models. We came up with a way to help ‘clean’ the dataset if models were not performing well on the full dataset.

We engineered a feature that represented the amount of ‘positive impact’ a player brought to their team (calculated by a weighted sum of other features like ‘Number of Kills’). We found the average value of this feature (per minute) in every role, at every Rank, and removed samples that were not close enough to the average (we could set how many standard deviations away they could be).

iii) Feature Selection & Engineering

Most of the data was numerical, the only piece of information that was categorical was the role they played and their rank. These were changed to numerical features (Ranks went from 0 to 5, and roles from 0 to 4). It made sense to map them to numerical values as a rank could just as well be a number representing your performance. As for the role, mapped it to a number just so that we could have all the data be numerical for the sake of consistency. We didn’t really have to do it, as ‘Role’ was irrelevant in the KNN Classifier, and the Random Forest Classifier could handle categorical data.

For feature engineering, we used our domain knowledge to come up with feature combinations that would be indicative of how a player performed in a match. Features like ‘Kill:Death’ ratios and ‘Takedowns’ (Kills + Assists) were added to the list of features we wanted to analyse and see how they affected the players performance.



We plotted the average value of a feature versus the Ranks of the players.

It was clear that there was some correlation between the features and the data. Some had a linear correlation, implying that they were independent of other features, while others had a quadratic or cubic relationship, implying that other features were influencing their effect on the output. We noticed that most of the ‘per minute’ features were linearly related to the output, so we knew these were strong features to consider. For each of the models below, we used what we thought was the best combination of features from the list above.

Methods

Baseline Model

i) Model Explanation

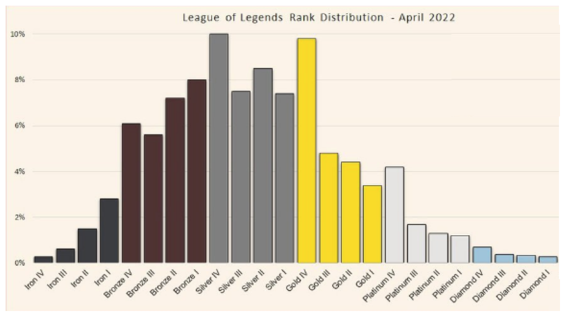
The baseline model is a model that we can use to compare our engineered machine learning model to. It sets a target for us to beat, and can confirm that we have indeed achieved something.

The baseline classifier we have decided to use predicts labels at random ignoring input values.

This is because each label appears evenly throughout the dataset. With our dataset, this should have an accuracy of about 1/6 (17%).

If you were to always predict either 'Bronze', 'Silver' or 'Gold', you would have an accuracy of about 25%-30% in the real world. However, because the labels appear an even number of times in our dataset, we should not use this sort of baseline - that is why we used a baseline classifier that randomly outputs a label.

In the real world, the ranks are distributed as shown in the figure below.



ii) Model Result

The accuracy of the dummy model is 0.17176263929296812.
[dummy.py]

K Nearest Neighbours (kNN) Classifier

i) Model Explanation

K Nearest Neighbours is an algorithm that can be used for classification problems.The operating principle on which K Nearest Neighbours is based on, is that for every data point, each point should fall near to data points which share a similar class. A new data point is classified by calculating the closest K neighbouring points to said point and labelling it based on the 'votes' of the neighbours (neighbours' votes can have a uniform weight or be weighted based on their distance, you must choose the weighting type when tuning hyperparameters).

ii) Features Selected

We wanted as few features as possible for kNN, to reduce the dimensions of its feature space. We wanted to limit the feature space to 3D (so around three features) to best utilise the 'minkowski' distance metric (we decided to use this over 'euclidean').

We experimented with a number of 'valid' features (feature validity was established as described in the 'Feature Selection & Engineering' section above), and found that Kill Participation, (Takedown/Death)/Minutes and Impact/Minutes were the best features to use.We did not need to clean the dataset for this model (We used all 120k samples!)

ii) Hyperparameter Selection

When choosing hyperparameters, one should pick them such that accuracy on test data is as close to the accuracy on training data as possible (otherwise it is over-fitted if training data accuracy is much higher) and such that the accuracy of the model is "high enough" (otherwise it is under-fitted). We used KFold cross validation (with 5 splits, holding out on 20% of the data for testing) to analyse our model with different hyperparameter values

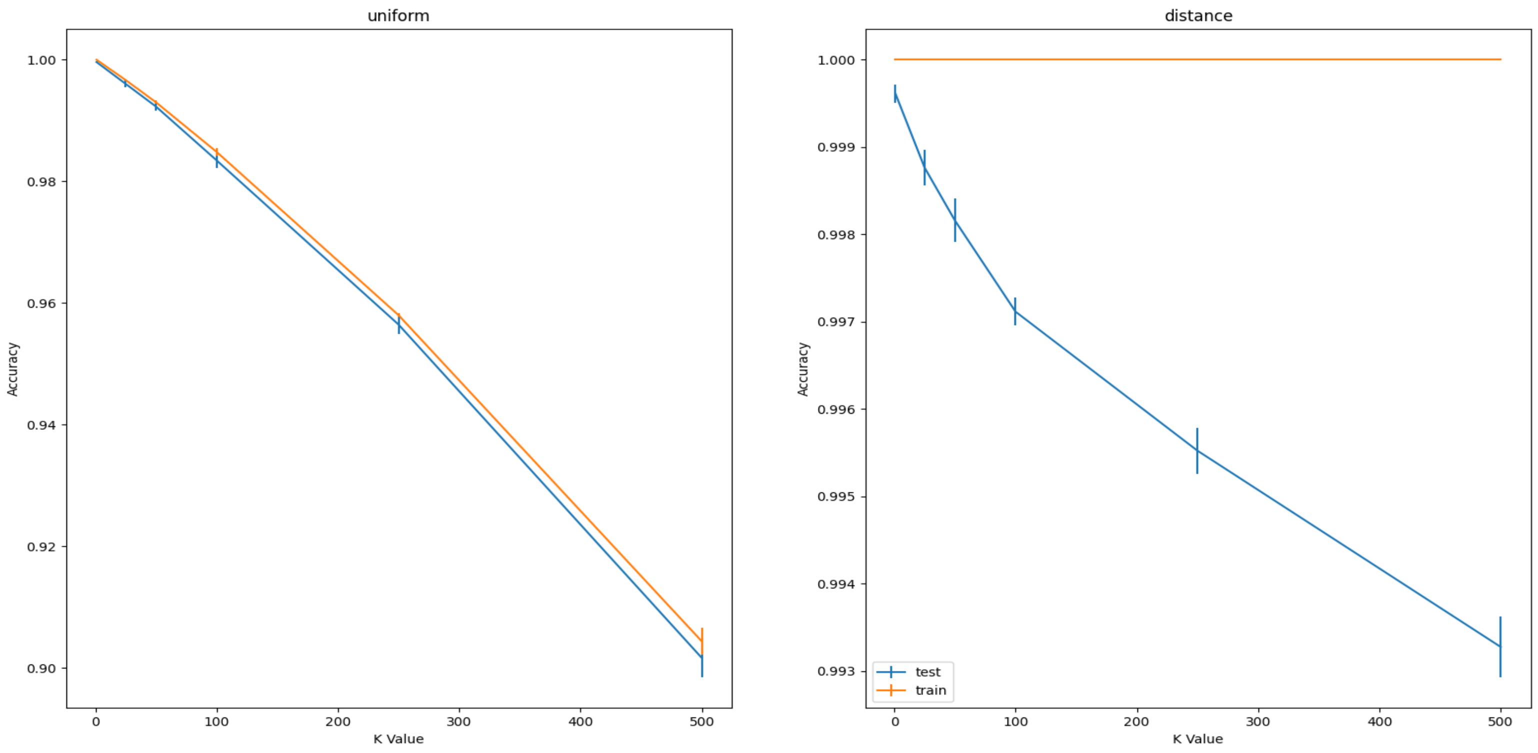
There are a few different hyperparameters to tune for kNN classifiers, but we decided to tune the two hyperparameters that had the most impact: the weighting type (uniform or distance based) and k (the number of neighbours to get votes from). Another hyperparameter we could have tested was the search algorithm that was used to find the k nearest neighbours (and also tune its own corresponding hyperparameters), but we left this to the default value of ‘auto’, as it attempts to pick the most appropriate search algorithm based on the values passed to the fit() method.

The accuracy plots below allowed us to identify the optimal distance weighting and k-value. Higher values of k tended to decrease the models performance, while the weighting of the votes seemed to prevent overfitting. From the plots below we would recommend a k-value of 5 or 6, and to use the distance weighing type on the neighbours when they cast their vote.

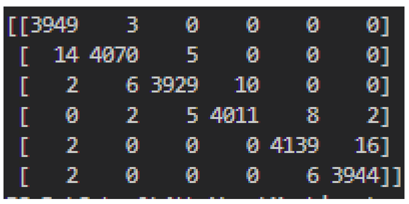
The best value for K is a value < 25, as it has more than 99.9% accuracy.

iii) Model Results

Model accuracy was calculated by dividing the number of correct predictions by the total number of predictions.



‘Uniform’ weights control overfitting better than ‘Distance’ weights, but ‘Distance’ is more accurate



```
[[3949    3    0    0    0    0]
 [  14 4070    5    0    0    0]
 [   2    6 3929   10    0    0]
 [   0    2    5 4011    8    2]
 [   2    0    0    0 4139   16]
 [   2    0    0    0    6 3944]]
```

Model accuracy reaches 99.99% for test data on both weight types, ~99.99% for training data with the uniformly weighted model, and is always 100% accurate on training data with the ‘distance’ weighted model. Accuracy decreases as we increase K.

The model isn’t under-fitted because the accuracy is very high. The ‘distance‘ weighted model is more overfitted than the ‘uniform’ weighted model, and gets more and more overfitted as we increase K. However, since the difference is < 1%, we believe the models ‘overfitted-ness’ is controlled. The results of the model show that our model is very accurate, after getting high accuracy percentages, and that it definitely beats our Baseline. As mentioned above, we used KFold cross validation, so we were splitting our data and holding out on test data within the train/test splits.

Random Forest (RF) Classification.

i) Model Explanation

We chose to analyse the performance of a random forest model as it seemed interesting to use a relatively non-conventional classifier, since we already achieved a high enough accuracy with the kNN Classifier.

A random forest classifier is a model that combines the output of multiple Decision Tree classifiers to reach a single output. It works by usually 'bootstrapping' and creating a smaller version of the dataset for each decision tree in the 'forest' (collection of decision trees) by sampling *with replacement*. Each decision tree is built using its own sample dataset and analyses randomly selected features.
When a new sample is inputted to the RF, it gets the aggregate of the labels given by the forest. Decision Tree classifiers, and by extension, RF classifiers, tend to overfit the data, and are really hard to tune. Decision trees don't have cost functions, they have criterions by which they calculate the quality of a split. We used the default 'gini' criterion. Decision trees try to minimise this value when creating a split. We specifically held out on 20% of the data to test after the hyperparameters were picked. We had to clean the dataset for this model using the method described in the 'Cleaning of Dataset' section above, and only used about 17k/120k samples in the end.

ii) Features Selected

We picked the features that seemed to be the most linearly correlated with the rank, as we deduced they'd be the easiest to split nodes with. We had 11 features in total (the role + 10 'linear' engineered features).

iii) Hyperparameter Selection

RF Classifiers have seven main hyperparameters: The number of trees in the forest, the max depth of each tree, the number of samples the trees can bootstrap, the maximum number of features each tree can analyse, the minimum number of samples in a leaf, the minimum number of samples in a leaf before splitting, and many more! All these hyperparameters are very useful to prevent overfitting, but because there are so many, it is very difficult to tune them.

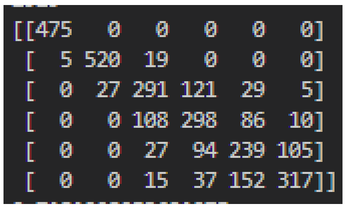
Instead of manually tuning the model, we used GridSearchCV to cross validate the model. We didn't have the computing resources and time to fully check across the full ideal range of values (which is included in the code), we had to truncate them to about 2 or 3 values per hyperparameter, and used 3-fold cross validation.

iv) Model Results

Model accuracy was calculated by dividing the number of correct predictions by the total number of predictions.

The best model test accuracy was ~72% and its training accuracy was ~86%. This was achieved with the following hyperparameters:

```
'bootstrap': True,
'max_depth': 2980,           'min_samples_split':
'max_features': None                        0.001,
'min_samples_leaf':10        'n_estimators': 12
```



```
[[475   0   0   0   0   0]
 [  5 520  19   0   0   0]
 [  0  27 291 121  29   5]
 [  0   0 108 298  86  10]
 [  0   0  27  94 239 105]
 [  0   0  15  37 152 317]]
```

This model is definitely over-fitted, as training accuracy is much higher than test accuracy. But this is expected of a model that relies on Decision Trees. It is interesting to note that the 'best' model has only 12 trees in its forest, when it could have had up to 100. The model accuracy is not very high, but one can't really argue that it is under-fitted.

Summary

The kNN Classifier has an amazing accuracy of over 99.99% when the right value for k is chosen with respect to the weighting type ('uniform' or 'distance' weighting). Intuitively it made sense that players would be close to each other in a feature space if the right features were chosen, so we are pleased we were able to prove this with our model.

The Random Forest Classifier didn't perform as well but it still had a relatively high performance (~70% accuracy), avoiding under-fitting. Unfortunately, as is characteristic of such a classifier, it was starting to become over-fitted. If we had more computing resources or time, we might have been able to run the full ideal range of hyperparameters, and we might've been able to get better results for this classifier. But overall, it had a decent performance with minimal hyperparameter tuning, and we are glad we took this opportunity to explore a model outside the course material.

Contributions

- Report: Everyone completed the report details of what they worked on
- Data Gathering (Web Scraper): Cornel
- Data Cleaning: Everyone contributed
- Overall Feature Engineering: Cornel & Terlo
- Model Selection: Everyone contributed
- kNN Classifier Training & Hyperparameter Tuning: Cornel & Michael
- kNN Experiments: Michael
- Random Forest Classifier Training & Tuning: Terlo & Michael
- Random Forest Experiments: Terlo

Cornel I Cicai

Micheal Adebusuyi

T Akintola