Delft University of Technology

The AI that solves League of Legends: How to play a MOBA game professionally according to AI models

Author:
Yunsong Yao

Thesis Supervisors:
Prof.dr.ir. G. Jongbloed
Dr. N. (Nestor) Parolya
Dr. R.J. Fokkink

June 20, 2023

Abstract

This thesis examines the strategies used during the Ban/Pick phase of professional League of Legends matches by using a quantitative model to study its influencing variables, including winning rates of champions, team compositions, team sides and other components. Hence, by studying patterns and relationships between them, our purpose is to gain a deep understanding of the decision-making techniques in the aforementioned context, with which we aim to provide professional League of Legends coaches with state-of-the-art Ban/Pick strategies.

To successfully achieve this, we will first provide an extensive introduction to the objectives and mechanisms of League of Legends before discussing the structure and rules of the Ban/Pick phase, which is often regarded as the most important aspect of the game strategy-wise, within the game. Then, we discuss the methods and datasets used for our quantitative analysis by comparing the methods with different criteria and assess their efficiency, consistency and robustness. Last but not least, we finish off by summarizing our achievements and recommending further study directions in this area.

1 Introduction

League of Legends (LoL) is a 2009 multiplayer online battle arena video game developed and published by Riot Games. In the game, two teams of five players battle each other, with the mission of destroying the main base of the opponent, known as the Nexus. Every player chooses and controls a character, known as a "champion". These champions all have unique abilities and different play styles [Lea10]. During a game, champions become more powerful by collecting experience points and gold. With gold, they can purchase items from a wide range of collections to push the team to victory [Zha+].

The game is played on a map called Summoner's Rift, which consists of three lanes and two jungles, along with additional static objects such as turrets, inhibitors, and neutral monsters that provide strategic advantages when destroyed or defeated. Figure 1 displays the map of Summoner's Rift. In the figure, the name of each lane is shown, and during the laning phase, ally's laning champions (Top, Mid, Bot and Sup) proceed to the corresponding lanes to battle against enemy champions (Top, Mid, Bot and Sup). Meanwhile, jungle champions go into the jungle and combat with neutral monsters to gain experience and gold.

Figure 1: Map of Summoner's Rift of competitive League of Legends matches.

1.1 Champion selection

As mentioned earlier, one of the most crucial aspects of League of Legends is the pre-game champion selection process, commonly known as the "Ban/Pick (B/P) phase". In this phase, each team takes turns banning champions they do not want their opponents to pick, followed by selecting champions for their own team composition. Note that if a team bans a champion, then that champion cannot be chosen by either of the teams. The process is strategic, as players need to consider their team's strengths and weaknesses, synergies between champions, and counter-picks to enemy champions. The B/P phase follows the order *1*-*1*-*1*-*1*-*1*-*1*-**1**-**2**-**2**-**1**-*1*-*1*-*1*-*1*-**1**-**2**-**1** [Rob]. Here, the colors denote the team currently selecting, the numbers (1 or 2) signify how many champions each team should pick or ban, the italic letters denote the banning phase, and lastly, the bold letters denote the selection phase. In simple words, at the start of the B/P phase, the blue team and the red team ban champions alternately until each team has banned 3 champions, followed by

the picking phase where the blue team selects one champion first, succeeded by the red team selecting two champions, and so forth, until each team successfully selects a total of three champions. Subsequently, we go back to the banning phase where the red team and the blue team ban the champions alternately until in total of 10 champions are banned, lastly the teams select the champions by order **red-blue-blue-red** until all 10 champions are selected. An example of B/P phase in League of Legends is shown in Figure 2.

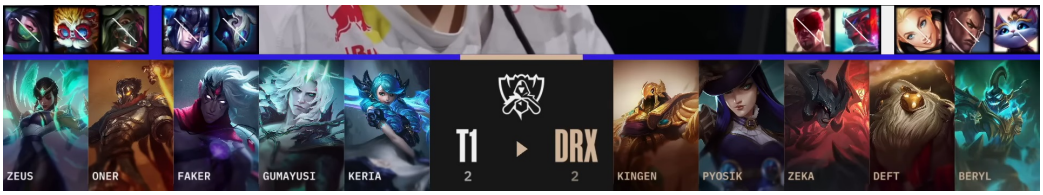


Figure 2: B/P phase of the final game of League of Legends 2022 Worlds Grand Final.

2 Methods

To determine the best method for selecting champions, we first need to predict the winning rate of all possible team compositions. In other words, when all 10 champions are chosen arbitrarily, we need to predict the winning rate of the blue side against the red side. We scrape first datasets from Fandom [LPL], [LCK], [wor]. In our dataset, all the variables we have are categorical variables, which consist of two different types: nominal categorical variables and ordinal categorical variables. An example of our dataset is shown in figure 3.

	patch	blue	red	...	red_mid	red_adc	red_sup
0	12.14	Victory Five	Ultra Prime	...	Cryin	Elk	Miaoniu
1	12.14	Victory Five	Ultra Prime	...	Cryin	Elk	Miaoniu
2	12.14	Victory Five	Ultra Prime	...	Cryin	Elk	Miaoniu
3	12.14	JD Gaming	Anyone's Legend	...	Forge	Betty	QiuQiu
4	12.14	Anyone's Legend	JD Gaming	...	Yagao	Hope	Missing

Figure 3: Example of the dataset.

Nominal categorical variables are variables that have two or more categories without any intrinsic ordering [Bru11]. In this dataset, the name of the blue team and the red team are examples of nominal variables. Each team has a unique name, but there is no intrinsic order to the team names.

Ordinal categorical variables, on the other hand, have categories that can be ordered or ranked [Has]. For instance, the variable "patch" can be considered as an ordinal categorical variable as patches are released in a chronological order, and a higher patch number implies that it was released later than the ones with lower numbers. However, in our case we consider "patch number" as a nominal categorical variable because a later patch number is not necessarily related to a higher value.

To perform any classification tasks on these categorical variables, they need to be transformed into numerical variables. In this section, Onehot encoding method, Binary encoding method and Embedding method are introduced and compared to see which one performs better.

There are a total of 493 games in the dataset, and for each game, 24 variables are available. These variables are nominal variables such as the name of the blue team, name of the red team, top lane champion of blue team, jungle champion of blue team, support player of red team, patch number, and

the winning team.

Before applying any transformation methods, we split the dataset into a training set (X_{train}, y_{train}) containing 75% of the total data, and a test set (X_{test}, y_{test}) containing the remaining 25%. We will repeat this behavior for a large number of times to ensure that the model can be trained and tested on a large variety of data.

2.1 Onehot encoding method

This subsection describes a method for representing categorical variables in a dataset using vectors. Specifically, for each categorical feature, a vector of length equal to the number of possible values of the feature is created, and each possible value is assigned a corresponding position in the vector. When a variable has a particular value, the corresponding position in the vector is set to 1, and all other positions are set to 0.

In the case of the given dataset, there are 23 features, each with a certain number of possible values as shown in the table. For example, the "Patch" feature has 6 possible values, so it would be represented by a vector of length 6. The exact mechanism is shown in Table 1.

Patch		Patch		Patch					
12.1		1		1	0	0	0	0	0
12.11		2		0	1	0	0	0	0
12.12	→	3	→	0	0	1	0	0	0
12.13		4		0	0	0	1	0	0
12.14		5		0	0	0	0	1	0
12.18		6		0	0	0	0	0	1

Table 1: Transforming mechanism for Onehot encoding method.

To represent a single game in the dataset, we stack all of the feature vectors vertically to form a single vector W that represents all of the values for a single game. The first N_1 entries of W correspond to the first variable, the next N_2 entries correspond to the second variable, and so on, until the last N_{23} entries correspond to the last variable. The dimensions of the variables after Onehot encoding is shown in Table 2.

Feature	Patch Blue pick 4 Red pick 5 Red top	Blue team Blue pick 5 Blue top Red jungle	Red team Red pick 1 Blue jungle Red mid	Blue pick 1 Red pick 2 Blue mid Red bot	Blue pick 2 Red pick 3 Blue bot Red support	Blue pick 3 Red pick 4 Blue support
Dimension	6 23 39 49	43 31 49 53	43 37 53 49	29 27 49 50	24 31 51 55	25 34 52

Table 2: Dimensions of the variables after applying Onehot encoding.

Using this representation, it is possible to perform various machine learning tasks on the dataset, such as classification. However, it should be noted that this method assumes that each feature is equally important, which may not always be the case in practice. Additionally, if the total number of possible values for the features is very large, the resulting vector representation may be very high-dimensional, which could pose challenges for the classification.

Adding up all the numbers yields the total length of the vector W, which is 892. Since the total number of matches available is 493, we are now able to represent all the data with a numerical matrix X of size 493×892.

2.2 Binary encoding method

Binary encoding is another popular technique used to convert categorical data to numerical data using binary digits (0s and 1s). In this process, each value of the variables is assigned a unique binary code, where all digits together represent a unique value that a variable can possibly take. By converting categorical variables into binary form, we can create a binary representation of the data that can be processed by machine learning algorithms.

To convert a categorical variable into binary form, we first determine the total number of unique values and label them using arbitrary integers from 2 to the total number of values plus 1. If, on the contrary, we started with 1, the binary representation of the very first value would be 00...00, resulting in a zero-row that may cause issues such as bias in the algorithm and inaccurate feature importance. Moreover, as a convention, we label the categorical variables in lexicographical order. Next, we convert each label from decimal to binary and place each digit in a separate column. As a

result, a categorical variable with N values can be represented using only $\lfloor \log_2 N + 1 \rfloor$ columns.

For example, consider the categorical variable "Patch" with the values "12.1", "12.11", "12.12", "12.13", "12.14", and "12.18". We label them from 2 to 7 and convert each label to its binary representation (001, 010, 011, 100, 101, 110). Finally, we place each digit in a separate column resulting in the representation $0|0|1, 0|1|0, 0|1|1, 1|0|0, 1|0|1, 1|1|0$. The mechanism is demonstrated in Table 3.

Patch		Patch		Patch		Patch		
12.1		2		001		0	0	1
12.11		3		010		0	1	0
12.12	→	4	→	011	→	0	1	1
12.13		5		100		1	0	0
12.14		6		101		1	0	1
12.18		7		110		1	1	0

Table 3: Transforming mechanism for Binary encoding method.

By converting categorical variables to binary form, the resulting matrix has significantly fewer dimensions, reducing the sparsity of the data. Table 4 displays the dimensions of the categorical variables after transformation.

Feature	Patch Blue pick 4 Red pick 5 Red top	Blue team Blue pick 5 Blue top Red jungle	Red team Red pick 1 Blue jungle Red mid	Blue pick 1 Red pick 2 Blue mid Red bot	Blue pick 2 Red pick 3 Blue bot Red support	Blue pick 3 Red pick 4 Blue support
Number of values	6 23 39 49	43 31 49 53	43 37 53 49	29 27 49 50	24 31 51 55	25 34 52

↓

Feature	Patch Blue pick 4 Red pick 5 Red top	Blue team Blue pick 5 Blue top Red jungle	Red team Red pick 1 Blue jungle Red mid	Blue pick 1 Red pick 2 Blue mid Red bot	Blue pick 2 Red pick 3 Blue bot Red support	Blue pick 3 Red pick 4 Blue support
Dimension	3 5 6 6	6 5 6 6	6 6 6 6	5 5 6 6	5 5 6 6	5 6 6

Table 4: Dimensions of the variables after applying Binary encoding.

By applying Binary encoding to the categorical variables, we can significantly reduce the dimension of the resulting matrix from 493×892 to 493×128. However, it is important to note that Binary encoding has its limitations and drawbacks in our case. Many of the variables are unrelated to each other, representing distinct items such as players and champions. As a result, using Binary encoding may lead to a large number of dependent variables, with only $\lfloor \log_2 N + 1 \rfloor$ variables being independent based on simple linear algebra. Additionally, the dependencies between pairs of variables may vary as the dot products of pairs of vectors are different, which can pose challenges during classification. This must be taken into account during the analysis of the data.

2.3 Classification methods for basic encoding techniques

After transforming all features into either Onehot encoded form or Binary encoded form, our dataset contains only numerical values, and all features are considered equally important. Therefore, we do not perform dimensionality reduction and explore three classification techniques that are suitable for this situation.

Firstly, we consider Logistic Regression, which is appropriate for predicting the likelihood of an event occurring based on a set of independent vari-

ables [IBM]. In our case, the probability that the blue side wins lies between 0 and 1, which makes Logistic Regression an appropriate choice. However, we must assume that predictors are independent, which may not hold for champions since some of them interact with each other. Moreover, logistic regression is prone to overfitting, particularly when using Onehot encoding, where we have almost twice as many features as data points [Tra20].

The next method we use is random forest classification, which is more accurate and robust than logistic regression, as it reduces overfitting and variance by averaging the results of different trees [R21]. Since our dataset has a large number of features and classes, we expect random forest classification to perform well. However, this classifier may be slow to train when using a large number of trees. Still, this is not a significant issue since we only need to train once and store the classifier locally. In addition, random forest classification may not perform well on high dimensional or sparse data [Sha21]. This can be problematic for our Onehot encoded dataset, which contains many zeros, whereas the binary encoded dataset does not contain as much information.

As champions interact with one another, we cannot assume that the predictors are independent. The interactions between champions can be modeled using the encoding methods discussed in previous sections. With a total of 163 champions, employing the Onehot encoding method to represent interactions between two champions brings up the addition of 26,404 ($2C_{163}^2$) columns after the encoded columns. Out of the 26,404 entries, 20 ($2C_5^2$) would be ones, and the remaining 26,383 would be zeros. It becomes evident that even when attempting to represent pairwise interactions between champions, we are faced with an unmanageable number of columns, let alone accounting for higher-order (3, 4, 5) interactions that undoubtedly play a non-negligible role.

One solution to this challenge is the implementation of neural network methods, as they are designed to learn complex relationships between inputs and outputs. However, neural network methods are considered "black-box" approaches, meaning it is nearly impossible to comprehend precisely how the network arrives at its decisions or predictions.

In summary, we have three suitable classification techniques for our dataset, each with its advantages and limitations. The comparison between the methods are summarized in Table 5.

Methods	Advantages	Limitations
Logistic Regression	Easy to use and the output is a probability between 0 and 1	Independence does not always hold and is prone to overfitting
Neural Network	A good solution to the intractable interaction problem	Black-box method
Random Forest	More robust than logistic regression and reduces overfitting	May not perform well on highly sparse data

Table 5: Methods and their advantages and limitations.

2.4 Embedding

The Onehot encoding method creates a sparse matrix of size 493×892, resulting in a large number of explanatory variables (892) with mostly zeros. This poses a challenge known as the curse of dimensionality, which can adversely affect the accuracy of statistical models. To address this issue, we can use a method called embedding, which reduces the dimensionality of variables using deep learning [Koe18b].

Since we only have nominal categorical variables, we can order the values of each variable arbitrarily. As a convention, we order the values in a lexicographical order and assign the first value an order of 0 and the last value an order of n_i, where n_i is the total number of possible values of the i-th variable minus 1. We then decide how many dimensions we want to reduce.

It is important to find a balance between the number of dimensions we want to reduce and the amount of information we want to retain. If we reduce too few dimensions, we still have a large number of dimensions, and the curse of dimensionality persists. However, if we reduce too many dimensions, we may lose too much information and end up with strong collinearity between variables.

As a convention, we set the dimension of each transformed variable to be half of the number of possible values, rounded up to the closest integer. The dimensions of the transformed variables can be found in Table 6. By reducing the dimensionality of the variables through embedding, we can possibly improve the performance of our statistical models and avoid the curse of dimensionality.

Feature	Patch Blue pick 4 Red pick 5 Red top	Blue team Blue pick 5 Blue top Red jungle	Red team Red pick 1 Blue jungle Red mid	Blue pick 1 Red pick 2 Blue mid Red bot	Blue pick 2 Red pick 3 Blue bot Red support	Blue pick 3 Red pick 4 Blue support
Desired dimension of transformed values	3 12 20 25	22 14 25 27	22 19 27 25	15 14 25 25	12 16 26 28	13 12 26

Table 6: Dimensions of the variables after applying Embedding.

After having chosen the dimensions, the transformed variables are initialized arbitrarily while letting all variables have weights that add up to 1. That is, for variable i, if this variable can take n_i values, we let the transformed variable $T_i = [x_{i1}, ..., x_{in_i}]^T$.

Subsequently, all the vectors representing the categorical variables are concatenated to a vector of length $\sum_{i=0}^{N} n_i$, which is the sum of the lengths of all transformed vectors. This vector has the form

$$[x_{11}, ..., x_{1n_1}, x_{21}, ..., x_{2n_2}, ..., x_{N1}, ..., x_{Nn_N}].$$

All the data we have are transformed in this manner while letting the variables with the same values have the exactly same weight vectors and variables with different values have different weight vectors. After transforming (i.e. embedding) all the data into numerical values, those data are treated as the inputs of a shallow neural network. The neural network is custom built, it has arbitrarily determined fixed values of weights and bias, its number of layers, number of units per layer and its activation function are logically chosen. Finally, we may choose some suitable optimizer to optimize against the loss function that finds the embedded values.

In our case we create 2 dense neural network layers with 5000 units and 2500 units respectively, the numbers of units are decided by trial and error. We use ReLu as our activation function. Lastly we use sigmoid activation function to map the values of the 2500 units of the second layer to values between 0 and 1.

To determine the suitable loss-function, note that the output of the neural network is a vector consists of values between 0 and 1, we use binary cross-entropy as our loss function since the outcome of the training data is a binary vector. Lastly, we apply Adam stochastic gradient descent method with

a suitable step-size to find the weights. We notice that after 5 iterations we achieve an accuracy of 0.98645 for the training data. The embedding mechanism is summarized in Table 7.

To calculate the accuracy, we transform each value of each variable in the test data into the weights we found in the previous part. We therefore transform the 124×24 matrix consists of test data into an 124×455 matrix. Now we use the transformed matrix as our input to the deep neural network layers of the previous part, after rounding off the output, we obtain the accuracy simply by comparing the output to y_{test}.

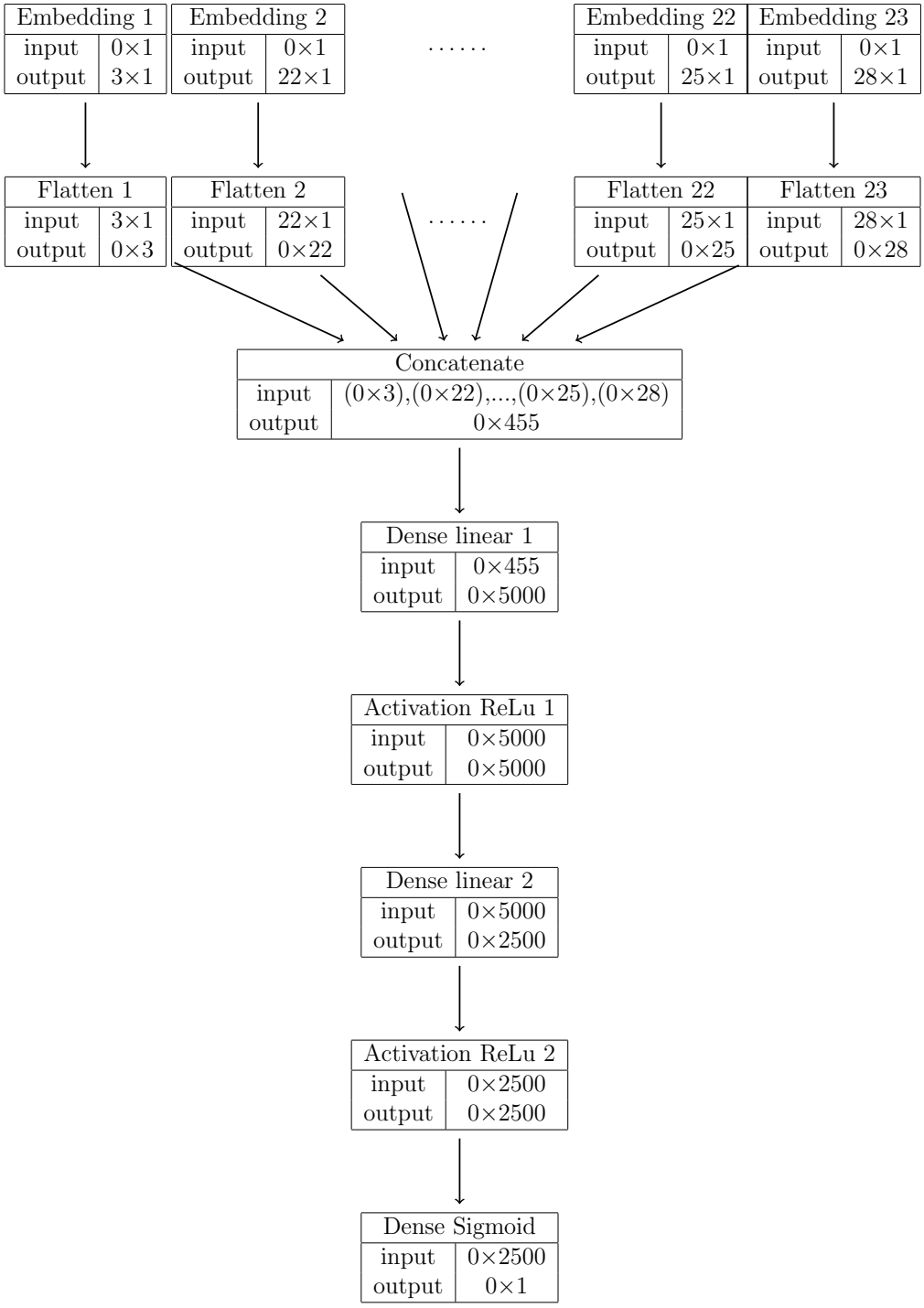


Table 7: Mechanism of Embedding.

3 Comparison between the methods

In this section we present comparison between the aforementioned methods using metrics such as accuracy, ROC, AUC and so on.

3.1 Accuracy

The traditional method of computing the accuracy is to test the accuracy of the test data and validate it through cross-validation. However, in our case, test sets created by using different portions of the whole data have a large difference in accuracy. A possible cause is that there are champions that are seldom selected, champions contained in the test sets might not even appear once in the training sets. A possible solution to circumvent this problem is to arbitrarily select a test set with replacement for a large number of times and test the accuracy of the selected test set. In Figure 4 we provide the comparison between embedding and various classification methods using Onehot encoding by splitting the data set into a 75% training set and a 25% test set arbitrarily 100 times. We observe that Random Forest classification method has the highest mean accuracy.

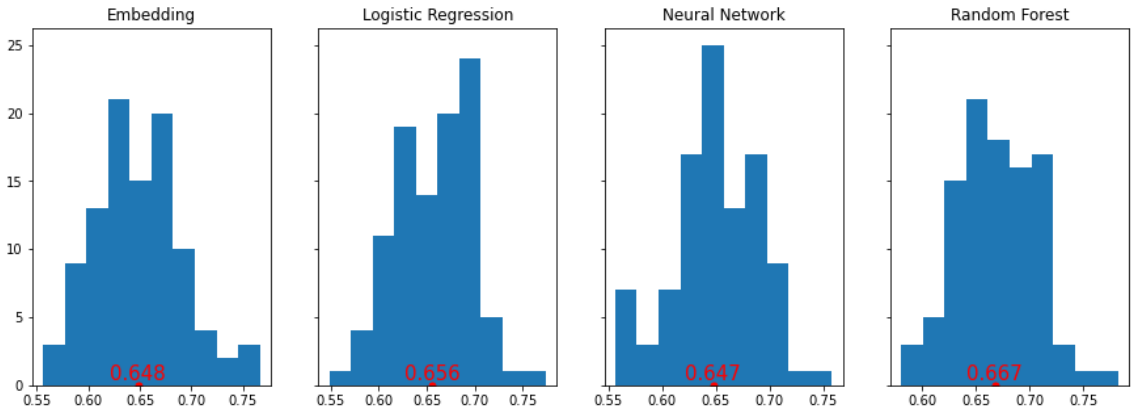


Figure 4: Comparison between Embedding and different Onehot classification methods.

In Figure 5, we compare the accuracy of different classification methods using Binary encoding. We observe that the accuracy of classification using Binary encoding is significantly lower than the accuracy using Onehot encoding, and on average, it is also lower than Embedding.

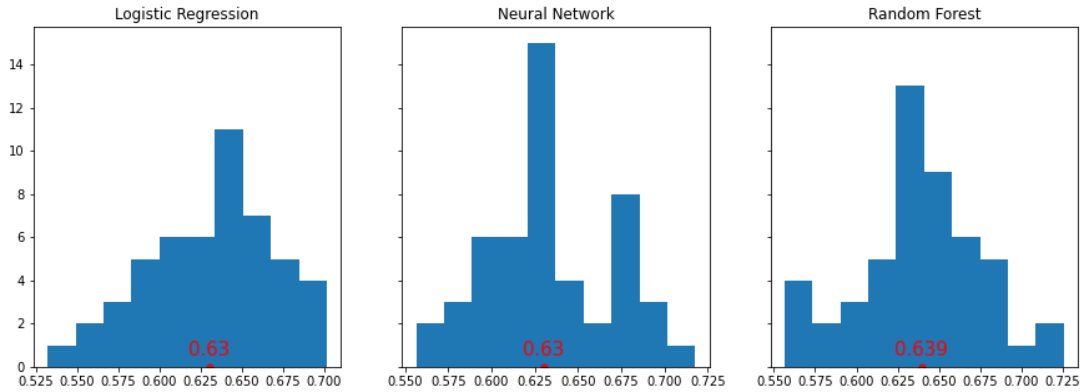
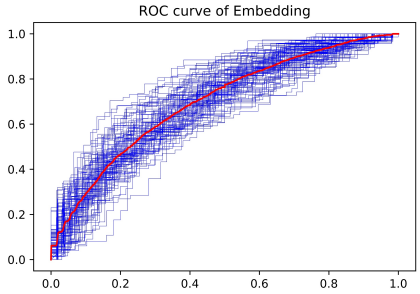


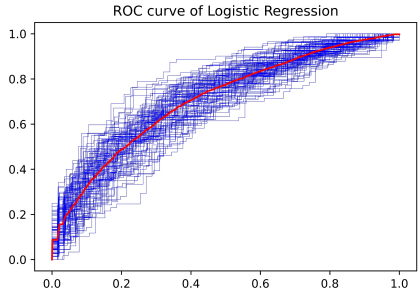
Figure 5: Comparison different classification methods using Binary encoding.

3.2 ROC and AUC

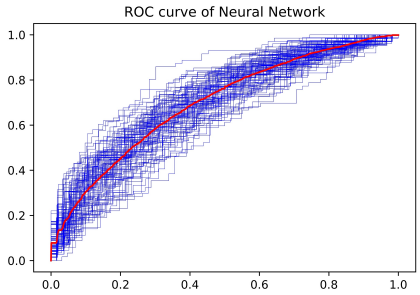
Another way to compare the methods is to find the ROC curve and the AUC parameter. Those are given in Figure 6, Figure 7 and Table 8. We see that when we use Onehot encoding method, not only does the Random Forest classifier have the highest average accuracy, but also the highest ROC curve and the largest area under curve. Embedding comes as the third regarding accuracy and is outperformed by Logistic Regression and Random Forest.



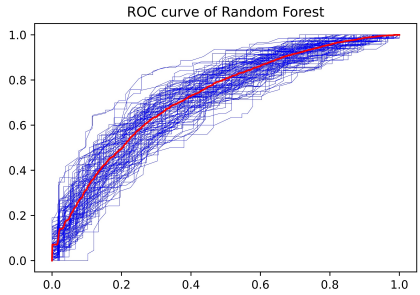
(a) ROC curve of Embedding.



(b) ROC curve of Logistic Regression.



(c) ROC curve of Neural Network.



(d) ROC curve of Random Forest.

Figure 6: Comparison between different classification methods by using Onehot encoding, the red curve denotes the average of the curves.

Area under curve	
Embedding	0.699
Logistic Regression	0.709
Neural Network	0.698
Random Forest	0.720

Table 8: The AUC parameter of different methods.

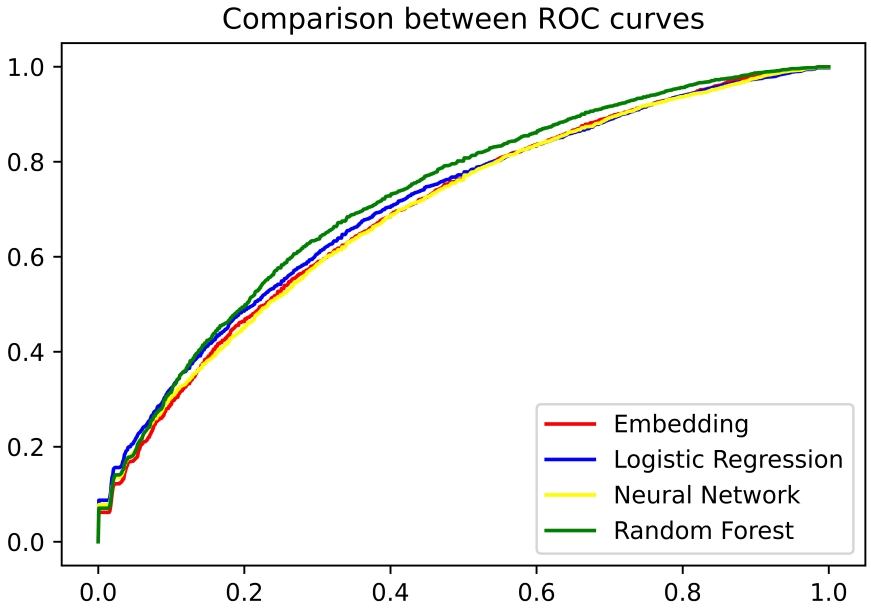


Figure 7: Comparison between the average ROC curve of Embedding and different Onehot classification methods.

3.3 Testing consistency

In this section, we present methods of testing the consistency of the classifiers. As we have successfully found a classification method that can predict the outcome of the games at an accuracy of approximately 0.67, we also need to make sure that the classifier has a strong consistency, meaning that the classifier should not make "simple" mistakes at a high probability.

In League of Legends, the two sides (blue and red) are designed to be nearly balanced, so playing on either side should be fair [Cri23]. Therefore, we expect that the outcome of a game should not be determined by the team's side, but rather by the strength of their champion selection. That means that if a given team plays against itself with a certain team composition (champion selection), the outcome of the game should be determined regardless which team it is. Therefore, suppose we have teams "A, B, ..., Y, Z" and our classifier predicts team A to win on the blue side against itself, we expect that with a high probability our classifier predicts team "B, ..., Y, Z" to win

on the blue side as well.

First we find the set of all teams that have appeared in our data set, then we take the whole data set and select one team, say team A, out of the set of teams and change the teams and the team members in the whole dataset to the current team (team A) and the corresponding players. Furthermore, we predict the outcome of the whole data set using the classifiers we have found in the previous sections, here we only consider Embedding and the Random Forest classifier using Onehot encoding. After this step we are left with a single vector of length 493 (number of games available) consisting only ones and zeros, those are expected to indicate the strength of the champion selection. We store this vector and continue selecting other teams out of the set of all teams and replace the "teams" and "teams members" of the dataset with the currently selected team. After looping over all the teams, we end up with 27 (number of teams) vectors of size 493, where the same position of the vectors indicates the same champion selection. The mechanism is shown in Table 9.

Game	Feature	Patch	Blue team	Red team	Blue team composition 1	Red team composition	Blue players	Red players
1	...	...	Blue team 1	Red team 1	Blue team composition 1	Red team composition 1	Blue players 1	Red players 1
2	...	...	Blue team 2	Red team 2	Blue team composition 2	Red team composition 2	Blue players 2	Red players 2
					...			
					...			
N	...	...	Blue team N	Red team N	Blue team composition N	Red team composition N	Blue players N	Red players N

↓

Game	Feature	Patch	Blue team	Red team	Blue team composition 1	Red team composition	Blue players	Red players
1	...	...	Blue team 1	Blue team 1	Blue team composition 1	Red team composition 1	Blue players 1	Blue players 1
2	...	...	Blue team 1	Blue team 1	Blue team composition 2	Red team composition 2	Blue players 1	Blue players 1
					...			
					...			
N	...	...	Blue team 1	Blue team 1	Blue team composition N	Red team composition N	Blue players 1	Blue players 1

↓

classification

↓

binary vector of length 493

Table 9: Mechanism of letting one team play against itself, this mechanism needs to be performed 27 times.

Now, for each of the 493 games, we examine the prediction made by the classifiers 27 times (for each of the 27 teams) and record the occurrence of ones (number of times that the blue team wins). We set up two hypotheses

here, H_0 : *The number of occurrences of ones is the half of the number of teams* against H_1 : *The number of occurrences of ones is not the half of the number of teams*. If the occurrences of ones is close to the number of teams (27) or close to 0, then H_1, our desired result, holds. That would mean that our classifier is therefore consistent. Suppose that the result is binomially distributed with $p = 0.5$ and $N = 27$. We use 0.05 as our critical p-value, and the percentage of the games where p-values of the occurrences of ones are less than 0.05 is given in the Table 10.

Percentage of the games where H_0 is rejected	
Random Forest	0.355
Embedding	0.462

Table 10: Percentage of the games where the occurrences of ones are close to 0 or 27.

We observe that for the Random Forest classifier, 35% of the p-values are less than the threshold 0.05, and for Embedding, 46% of the p-values are less than 0.05. Additionally, we calculate the average distance between the winning rate of the various team compositions and the boundary $\{0, 1\}$ when the same teams play against themselves. The distance is defined by $\frac{1}{P}\sum_{i=0}^{P} \mathbb{1}(p_i \leq 0.5)p_i^2 + \mathbb{1}(p_i > 0.5)(1-p_i)^2$, where $\mathbb{1}$ denotes the indicator function. The results are shown in Table 11. Note here that the smaller the mean distance is, the closer the winning rate is to the boundary, indicating a more consistent classification.

Mean distance to the boundary in L^2	
Random Forest	0.336
Embedding	0.303

Table 11: Mean distance between the winning rate and the boundary 0 and 1.

The next method we use to test the consistency of the classifier is to simulate games where we not only let the teams play against themselves, but also let the teams have the exactly same team composition when they play against themselves. We examine the situations where a certain team uses the composition chosen on the blue side to play against itself and where it uses

the composition chosen on the red site to play against itself. The mechanism is shown in Table 12 and 13.

Game	Feature	Patch	Blue team	Red team	Blue team composition	Red team composition	Blue players	Red players
1	...	...	Blue team 1	Red team 1	Blue team composition 1	Red team composition 1	Blue players 1	Red players 1
2	...	...	Blue team 2	Red team 2	Blue team composition 2	Red team composition 2	Blue players 2	Red players 2
					...			
					...			
N	...	...	Blue team N	Red team N	Blue team composition N	Red team composition N	Blue players N	Red players N

↓

Game	Feature	Patch	Blue team	Red team	Blue team composition	Red team composition	Blue players	Red players
1	...	...	Blue team 1	Blue team 1	Blue team composition 1	Blue team composition 1	Blue players 1	Blue players 1
2	...	...	Blue team 1	Blue team 1	Blue team composition 2	Blue team composition 2	Blue players 1	Blue players 1
					...			
					...			
N	...	...	Blue team 1	Blue team 1	Blue team composition N	Blue team composition N	Blue players 1	Blue players 1

Table 12: Modification of the DataFrame where a certain team uses the composition on the blue side to play against itself.

Game	Feature	Patch	Blue team	Red team	Blue team composition	Red team composition	Blue players	Red players
1	...	...	Blue team 1	Red team 1	Blue team composition 1	Red team composition 1	Blue players 1	Red players 1
2	...	...	Blue team 2	Red team 2	Blue team composition 2	Red team composition 2	Blue players 2	Red players 2
					...			
					...			
N	...	...	Blue team N	Red team N	Blue team composition N	Red team composition N	Blue players N	Red players N

↓

Game	Feature	Patch	Blue team	Red team	Blue team composition	Red team composition	Blue players	Red players
1	...	...	Blue team 1	Blue team 1	Red team composition 1	Red team composition 1	Blue players 1	Blue players 1
2	...	...	Blue team 1	Blue team 1	Red team composition 2	Red team composition 2	Blue players 1	Blue players 1
					...			
					...			
N	...	...	Blue team 1	Blue team 1	Red team composition N	Red team composition N	Blue players 1	Blue players 1

Table 13: Modification of the DataFrame where a certain team uses the composition on the red side to play against itself.

In this case, we expect that nearly 50% of the situations our classifier predicts the blue side to win since we have made the blue team and the red team equally strong for each of the games. We now have H_0 : *The number of occurrences of ones is half the number of teams* against H_1 : *The number of occurrences of ones is not half the number of teams.* Now suppose again that the result is binomially distributed with $p = 0.5$ and $N = 27$, we calculate the p-value for each of the games and reject H_0 when $p < 0.05$. The percentage of the total 493 games that we do not reject H_0 is given in Table 14. Note that a higher percentage of not rejecting H_0 means less influence of the sides to the outcome of the games, indicating a more consistent classifier.

Percentage of the games where H_0 is not rejected	
Random Forest blue comp	0.737
Embedding blue comp	0.564
Random Forest red comp	0.809
Embedding red comp	0.570

Table 14: Percentage of the games where the occurrences of ones are near the half of the number of teams.

Also, the distance between the percentage of the games where the blue side wins and the baseline percentage 0.5, given by $\frac{1}{P}\sum_{i=0}^{P}(p_i - 0.5)^2$ is shown in Table 15.

Mean distance to the baseline 0.5 in L^2	
Random Forest blue comp	0.186
Embedding blue comp	0.255
Random Forest red comp	0.166
Embedding red comp	0.254

Table 15: Mean distance between the winning rates and the baseline winning rate 0.5.

We note that Embedding performs better when the teams play against themselves while keeping the original composition, and the Random Forest classifier with Onehot encoding performs better when the teams play against themselves with exactly the same team composition. This can be explained by the domain knowledge of League of Legends. In League of Legends, different teams have different strategies and different ways of playing. A certain team composition may not be suitable for some teams, possibly due to the fact that players have different masteries for the champions, players favor a certain side of the map, etc. Therefore, it is understandable and likely that one team favors the composition on the blue side and the other team favors the composition on the red side when the teams play against themselves. This can lead to completely different outcomes. On the other hand, if a team plays against itself with exactly the same team composition on both sides, the only factor that decides the outcome of the game is the side. However, as we discussed before, no side has an advantage over the other. Therefore, the outcome is more or less decided by a coin-flip.

3.4 Limitations

We see that the combination of Onehot encoding technique and Random Forest classifier has overall the best performance among all other classifiers. However, it also has limitations. In our dataset, we only have game records of 112 champions while there are in total 163 playable champions in League of Legends [Xu23]. This is due to the fact that in professional matches, players tend to choose champions that are strongest in the current meta, while the relatively weak champions are usually left out. The dataset available covers in total of 6 patches, while we have around 20 patches per year.

Suppose now we have a new dataset in which all the 163 champions are chosen at least once, and in the luckiest case no champion is chosen by players with different roles (so suppose a multi-role champion is chosen by a *Top* player, it is not chosen by any *Jun* player anymore). In this case, we would end up having a matrix of size $N \times 1008$. The number 1008 is obtained by noting that now we have 14 additional columns for patches (as the number of patches increases from 6 to 20) and 2*51 (as ally and enemy both has a increased champion pool from 112 to 163) more columns for champions. However, if we use Embedding or Binary encoding techniques, the number of the features in the matrix will be significantly smaller.

4 Fine-tuning of the Methods

4.1 Hyperparameter tuning

We see that Random Forest classifier has the best performance among all classifiers we have used. However, the classifier we used are with the default setting in Python. The setting is shown in Table 16.

parameters	values
bootstrap	True
ccp alpha	0
class weight	None
criterion	gini
max depth	None
max features	sqrt
max leaf nodes	None
max samples	None
min impurity decrease	0.0
min samples leaf	1
min samples split	2
min weight fraction leaf	0.0
n estimators	100
n jobs	None
oob score	False
random state	None
verbose	0
warm start	False

Table 16: Default parameters of the Random Forest classifier.

Note that the most important criteria are the number of trees ("n_estimators") and the max number of features when splitting at each leaf node [Koe18a]. Therefore, we take a selection of the parameters and give them a range of available values and search for the best value. The selection we take consists of the following parameters: "bootstrap", "criterion", "max_depth", "max_features", "min_samples_leaf", "min_samples_split", and "n_estimators". In order to determine a reasonable range of values for "max_depth", we check the values that this parameter currently has for all 100 trees. The depths of

the trees are [40, 36, 35, 48, 46, 37, ..., 36, 41, 37, 43, 42]. None of the trees has a depth more than 50, so it is reasonable to limit the "max_depth" of the trees to any positive integer below 45. The proposed range of values is given in Table 17.

parameters	values
bootstrap	{True, False}
criterion	{gini, entropy, log-loss}
max depth	{20,25,30,35,40,45,None}
max features	{sqrt, log2, auto}
min samples leaf	{1,2,3}
min samples split	{2,4,8}
n estimators	{100,200,400,800,1600}

Table 17: Range of parameters of the Random Forest classifier

Now we use the RandomizedSearchCV module from the "sklearn" package to find the best possible parameters with a randomized training set. The best parameters found are given in Table 18.

parameters	values
bootstrap	True
criterion	log-loss
max depth	40
max features	auto
min samples leaf	3
min samples split	2
n estimators	100

Table 18: Best parameter for Random Forest classifier on one random training set.

As we mentioned before, the selection of the training set greatly affects the accuracy of the classification. Therefore, We also need to test the accuracy on different training sets. As before, we will arbitrarily select a large number of training sets and compare the best parameters found by RandomizedSearchCV to determine the ultimately best parameters for our dataset. We take 50 random training sets and perform 100 searchings for each of the

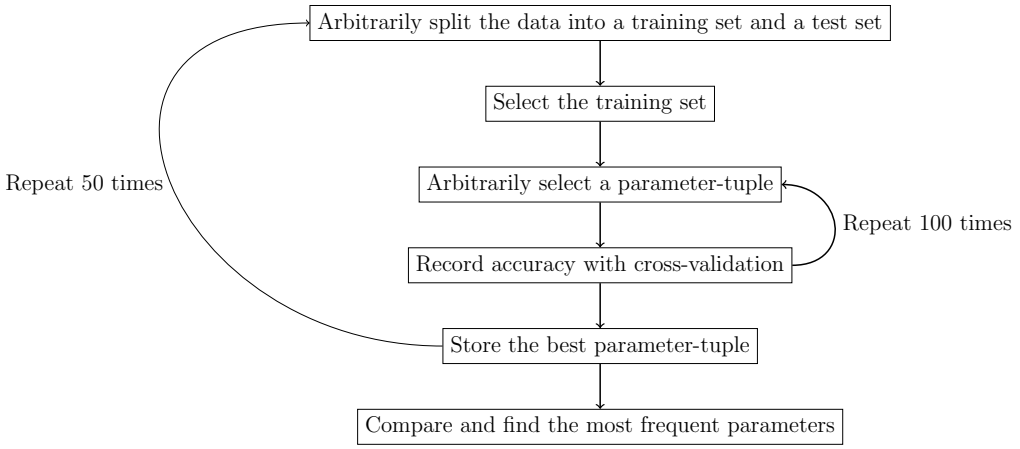


Figure 8: Mechanism of performing RandomizedSearchCV on 50 random training sets.

50 training sets. The mechanism is shown in Figure 8. The running time is 5070 seconds and the most frequent best value for each of the parameters is given in Table 19.

parameters	default values	most frequent best values
bootstrap	True	True
ccp alpha	0	0
class weight	None	None
criterion	gini	entropy
max depth	None	None
max features	sqrt	sqrt
max leaf nodes	None	None
max samples	None	None
min impurity decrease	0	0
min samples leaf	1	3
min samples split	2	3
min weight fraction leaf	0	0
n estimators	100	100
n jobs	None	None
oob score	False	False
random state	None	None
verbose	0	0
warm start	False	False

Table 19: Comparison between default parameters and the most frequent best parameters of the Random Forest classifier, the values of the parameters that can be improved are given in red.

4.2 Comparison between the default classifier and the fine-tuned classifier

After obtaining the best fine-tuned values, we again arbitrarily split the dataset into a training set and a test set for 100 times and evaluate the accuracy, ROC-curve and AUC, those are given in Figure 9, Figure 10, Figure 11 and Table 20.

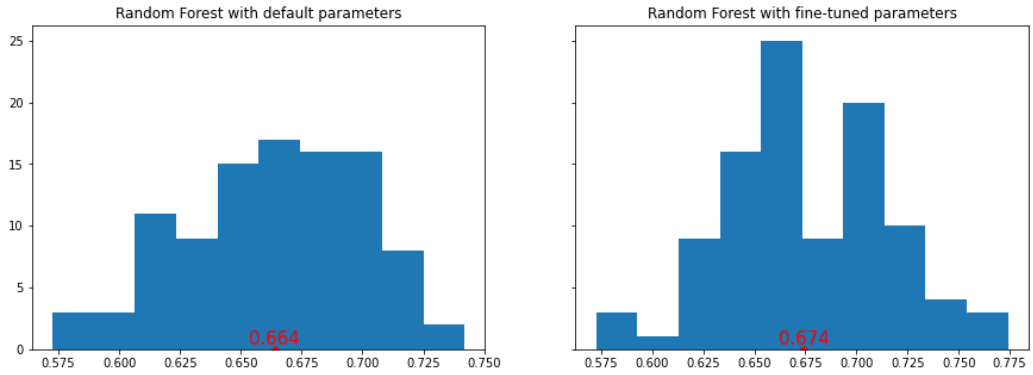
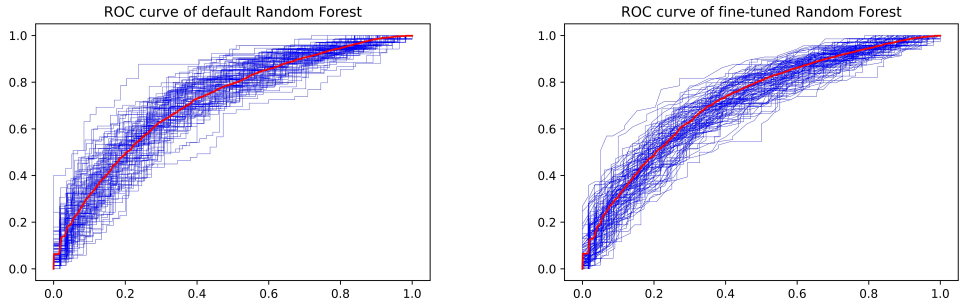


Figure 9: Comparison between the accuracy between Random Forest classifier with default parameters and with fine-tuned parameters.



(a) ROC curve of Random Forest classifier with default parameters.

(b) ROC curve of Random Forest classifier with fine-tuned parameters.

Figure 10: Comparison between the ROC curve between Random Forest classifier with default parameters and with fine-tuned parameters, the red curve denotes the average of the curves.

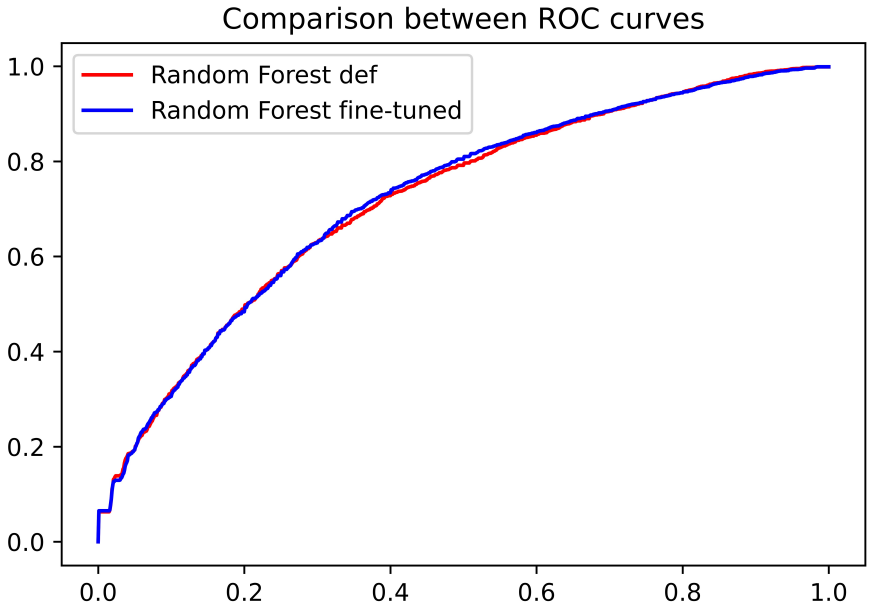


Figure 11: Comparison between the average ROC curve of Random Forest classifier with default parameters and with fine-tuned parameters.

Classifier	Accuracy	Area under curve
Random Forest default	0.664	0.713
Random Forest fine-tuned	0.674	0.723
Increase	1.5%	1.4%

Table 20: Increase in accuracy and AUC after fine-tuning Random Forest classifier.

We see that fine-tuning the Random Forest classifier improves not only the accuracy but also the AUC parameter. Therefore, we can conclude that the Random Forest classifier with the fine-tuned parameters is, for now, the best classifier for our model.

5 MCTS algorithm

In League of Legends, there are currently 163 champions. Considering the roles of the champions, the number of different team compositions is at least $(32 \times 31)^5 = 9.60 \times 10^{14}$, i.e. picking 2 champions for each of the 5 roles, one for the blue team and the other one for the red team. Due to the enormous number of team compositions and countless interactions between champions, it is nearly impossible for humans to draft the best possible team composition against the enemy team. Nowadays, the best champion selection strategies are based on several criteria, such as (i) player proficiency with the champions, for instance, factors like winning rate on that champion and mastery points, (ii) countering relationships between champions, for instance, the overall winning rate of a champion against another, projectile speed/width of the champion spells (a faster projectile gives an upper hand in the laning phase), and (iii) overall popularity of the champions (usually, the higher the popularity, the higher the tolerance of mistakes, the easier to snowball with the champions picked). Therefore, most coaches nowadays are retired professional players with little or no statistical knowledge who have played a large number of games. They select the champions for the players mostly based on their intuition.

In the previous section, we saw that the best classifier is the fine-tuned Random Forest classifier. In this section, we will use the results of the Random Forest classifier to determine the outcome of the games. We will consider the B/P problem as a two-person zero-sum game with perfect information. We will first present a method for solving a single best-of-1 League of Legends match, and then we will compare this method with other existing methods and examine the winning rate.

5.1 Selection criteria

As introduced in section 1 .The champion selection in League of Legends follows the order *1*-*1*-*1*-*1*-*1*-*1*-**1**-**2**-**2**-**1**-*1*-*1*-*1*-*1*-**1**-**2**-**1**. However, in League of Legends matches, banning champions has a strong connection with personal favorites. Therefore, in most cases, coaches ban specific champions not because of their strength, but to prevent their opponents from playing with comfort. Therefore, we may consider a simpler case that has the following selection order: **Blue-Red-Red-Blue-Blue-Red-Red-Blue-Blue-Red**. Note that we also simplified the cases where a player is allowed to

select 2 champions simultaneously to selecting 2 champions in a row. There are thus 10 rounds in the selection step. In each round, we need to find the "legal move", which is the set of all available champions at that round. The legal move must satisfy the following criteria:

- Any champion that is already selected, either by ally or enemy team, cannot be selected again.
- Any number of champions selected cannot have combined less number of roles than the number of champions selected.

The first criterion must be satisfied because in League of Legends professional matches, no two identical champions are allowed to appear in the same match.

The second criterion also must be satisfied because otherwise we would have multiple champions having the same role and possibly zero champion for some roles, this results in a extremely bad team composition due to the unique characteristics and roles that different champions have. In other words, any distinct role has a specific job that can only be done by the champions with that role. This criterion must be satisfied for every single round of champion selection. However, it would be extremely computationally heavy if we verify whether this criterion holds in each round for each iteration of the MCTS algorithm. To circumvent this problem, we note that the most champions have only one role and therefore after selecting such a champion, we can simply remove all other champions with only one role that is the same as the role of the champion we have selected from the champion set corresponding to the same side (blue or red). In this manner we end up having a list of champions chosen with at most 3 champions with multiple roles with a high probability. Also, we are certain that the champions selected with a single role must be assigned to that role during the process of predicting the outcome of the game. Therefore, all what is left is to verify whether we can assign the remaining roles to the multi-role champions left. This is known as the distinct representatives problem, and we will address it in the next subsection.

5.2 Distinct representative problem

Note that the dataset we have consists of 5 different positions (Top, Jungle, Mid, Bot, Sup) on each side. In the previous section, we saw that the MCTS algorithm selects the champions in an arbitrary order. Moreover, it is likely

that at least 2 champions out of 5 have multiple roles. To predict the outcome of the game, we need to find the corresponding role for each of the champions.

The distinct representative problem is a combinatorial problem in mathematics that involves finding representatives from a collection of sets, where each representative is an element in every other set in the collection. The problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve it for all instances [JR11].

In our case, we have a collection of 5 sets containing numbers ranging from 0 to 4, representing all the possible roles of the champions, where we let $0 = Top$, $1 = Jungle$, $2 = Mid$, $3 = ADC$, $4 = Sup$. Our goal is to find a way to assign 0, 1, 2, 3, 4 to all 5 sets, with the criterion that each number must be contained in every other set. As stated before, with a high probability, a champion selection sequence contains at most 3 champions with multiple roles. Therefore, the problem is reduced to assigning up to 3 numbers from 0 to 4 to a collection of up to 3 sets containing 0 to 4, since we must assign the only element of the singleton sets to themselves to find such a system of distinct representatives.

We present a method for finding a system of distinct representatives where the cardinality of the collection of sets is less than or equal to 3. Denote N as the size of the collection. First, for the trivial case, suppose we have a list of representatives $[n_0]$ and a collection of sets $[S_0]$. We can immediately determine whether a system of distinct representatives exists by inspecting whether n_0 is in S_0.

Now, suppose we have a list of representatives $[n_0, n_1]$ and a collection of sets $[S_0, S_1]$. We first check whether n_0 is in one of the sets S_0 or S_1 following the lexicographic order. If it is not, then we are done and it means that a system of distinct representatives does not exist.

Suppose n_0 is in S_0. We remove n_0 temporarily from the list and S_0 from the collection, and we are left with the trivial case where N = 1. If we can find a representative for N = 1, then we are done. Otherwise, we revert n_0 back into the list and add S_0 to the end of the collection. We then permute the collection and end up with $[n_0, n_1]$ and $[S_1, S_0]$. We perform the exactly same search again to find the distinct representatives, if they exist.

At last, suppose we have a list of representatives $[n_0, n_1, n_2]$ and a collection of sets $[S_0, S_1, S_2]$. Analogously, we check first whether n_0 is in one of S_0, S_1, S_2. If it is not, then we are done. Suppose it is in S_0, then as before, we temporarily remove n_0 from the list and S_0 from the collection. We are left with the case where N = 2. We then check whether we can find

a solution. If we can, then we are done. If we cannot, we revert n_0 back into the list and put S_0 at the end of the collection. We then have $[n_0, n_1, n_2]$ and $[S_1, S_2, S_0]$. If n_0 is only in S_0, then we are done. If n_0 is in S_1, we perform the same mechanism as before. We perform at most two permutations to find at least one system of distinct representatives.

5.3 Minimax algorithm

Based on the winning rate of the champions and their countering relationships, a natural approach is to use the Minimax algorithm to find the best champion of the current stage. Prior to Stockfish and Deep Blue, numerous chess engines made use of the Minimax algorithm [Mis19]. For chess engines, it ensures that the picks made by the Minimax algorithm are the most optimal ones. However, for League of Legends, considering picking a champion as a move in chess, there are approximately 3 times (31.1 moves in chess [Bar19] vs 100 moves in League of Legends) as many moves as in chess. Moreover, the Minimax algorithm is known as an extremely computationally heavy algorithm as it must store all game states that grow exponentially.

For a two-person-zero-sum game, the Minimax algorithm applies often when players take turns sequentially (or without knowing the opponent's move prior to their own move). However, in League of Legends, the teams do not select the champions simultaneously and they do not move in an alternating order. Suppose we consider a pair of champions picked by one team as one pick, we would be left with a matrix of size 112×10000 approximately for the very first turn. To elaborate, in the first turn, the blue side has 112 available picks and the red side has approximately 100×100 available combinations of picks since they have to pick 2 champions in this turn. For the blue team, completely not knowing what the red team would pick, approximately 10^6 outcomes (reward/loss) must be found and filled into the payoff matrix to derive the optimal pick using the Minimax strategy, which requires an outrageously large computational power. To circumvent this problem, we may place "bars" between rounds with each pair of picks as one single group of picks, and we consider each of those groups as one turn of a two-person-zero-sum game. Therefore, the whole selection process becomes $\underbrace{Blue - Red}_{turn\,1} | \underbrace{Red - Blue}_{turn\,2} | \underbrace{Blue - Red}_{turn\,3} | \underbrace{Red - Blue}_{turn\,4} | \underbrace{Blue - Red}_{turn\,5}$.

Suppose the blue team uses the Minimax algorithm to select the champions and the red team uses some other algorithm. Then we see in turns 1, 3,

and 5, without knowing what the red team would select, the blue team has to list all the possibilities of the blue champion-red champion pairs to find the optimal strategy. In turn 1 there are approximately $100 \times 100 = 10000$ distinct pairs, for each of the pairs, we first complete the rest of the 8 picks arbitrarily a large number of times using rules defined in section 5.1, then we predict the outcomes and average them using the fine-tuned Random Forest classifier in section 4. This yields the reward/loss of all the distinct pairs and we can now simply put the outcomes in a payoff matrix to find the optimal strategy, which is the best champion to pick at the current stage. However, at turns 2 and 4, the blue team picks after seeing the picks of the red team. We therefore only have to list all the legal picks of the blue team (instead of red-blue pairs) to find the optimal strategy. The optimal strategy is obvious, it is the pick whose predicted outcome has the highest reward for the blue side. Using the Minimax algorithm on the red side is analogous, at turns 1, 3, 5, we predict the outcomes of all the legal picks and take the pick that yields the least reward/loss with respect to the blue side and at turns 2, 4 we predict the outcomes of all combinations of the blue team-red team pairs and take the pick that minimizes the maximal reward for the blue team.

5.4 Introduction to MCTS algorithm

Monte Carlo Tree Search (MCTS) is a decision-making algorithm used in for finding the best move in a game or other types of decision-making problems. MCTS is a heuristic search algorithm that uses random simulations to build a search tree, where each node represents a possible game state or decision point, and each edge represents a possible move or action [Pet+02]. The algorithm builds the search tree incrementally, by repeatedly selecting nodes and simulating random moves from a non-leaf node to the leaf node. Once the leaf node is reached, the result is known and it will be sent back to all the preceding nodes.

The MCTS algorithm consists of four steps: Selection, Expansion, Simulation, and Backpropagation. In the selection step, the algorithm traverses the search tree from the root node to a leaf node based on a proper selection strategy, such as Upper Confidence Bounds 1(UCT1) [Pet+02], which tends to select the node with the highest payoff and the least visited node. In the expansion step, the algorithm creates child nodes of the current node consisting of legal moves. In the simulation step, the algorithm simulates a game by arbitrarily selecting from the legal set of moves. Finally, in the

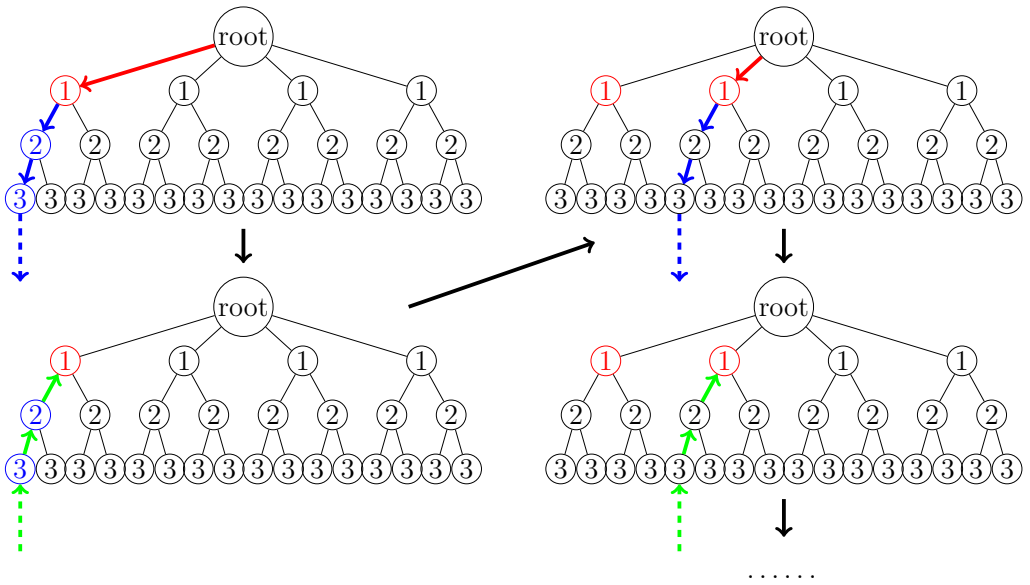


Figure 12: The mechanism of MCTS algorithm.

backpropagation step, the algorithm updates the results from the leaf node to all its ancestor nodes. The mechanism of MCTS algorithm is shown in Figure 12.

In this figure, the root node is the state in which we apply the MCTS algorithm. For a chess game, it is then the position in which we "turn on" the chess engine. The numbered nodes are the states that are 1, 2 or 3 moves away from the root node, the numbers indicate the depth of the searching tree. The uncolored nodes (black nodes) are unvisited nodes. Moreover, the red arrow indicates the selection process, the blue solid arrow indicates the expansion process, the blue dashed arrow indicates the simulation process, the green dashed arrow indicates the process where we input the result to the leaf node and finally the green solid arrow indicates the backpropagation process. The entire process repeats itself for a large number of times.

5.5 MCTS algorithm for League of Legends

After finding the legal set in each step, we let the algorithm select the champions by the formula (derived from UCT1) $U_1 = Q_i + C\sqrt{\log N_i/n_i}$ and

$U_2 = Q_i - 1 - C\sqrt{\log N_i / n_i}$, where Q_i stands for the average winning rate of the blue side at the current node, N_i stands for the number of visits of the parent node of the current node, n_i stands for the number of visits of the current node and C stands for the exploitation rate that theoretically equals to $\sqrt{2}$ in the optimal situation. Due to the champion selection order shown in the previous subsection, when it is round 1, 4, 5, 8, 9 the blue side selects and the red side selects in the rest of the rounds, which are 2, 3, 6, 7, 10. Moreover, we must also assume that the opponent selects the champion that most likely leads them to a win. Therefore, when it is round 1, 4, 5, 8, 9, the algorithm selects the champion that corresponds to the node with the highest U_1 and in other rounds, the algorithm selects the champion that corresponds to the node with the lowest U_2. To wit, U_1 increases when the winning rate for blue increases but decreases when the number of selection increases and U_2 decreases when the winning rate for blue decreases (therefore the winning rate for red increases) but increases when the number of selection increases. In a nutshell, the MCTS algorithm tends to choose the champions that lead to a higher winning rate but it will also choose champions with a low number of visits in case those champions have a lower winning rate by accident.

The MCTS algorithm for League of Legends can therefore be summarized as in Figure 13.

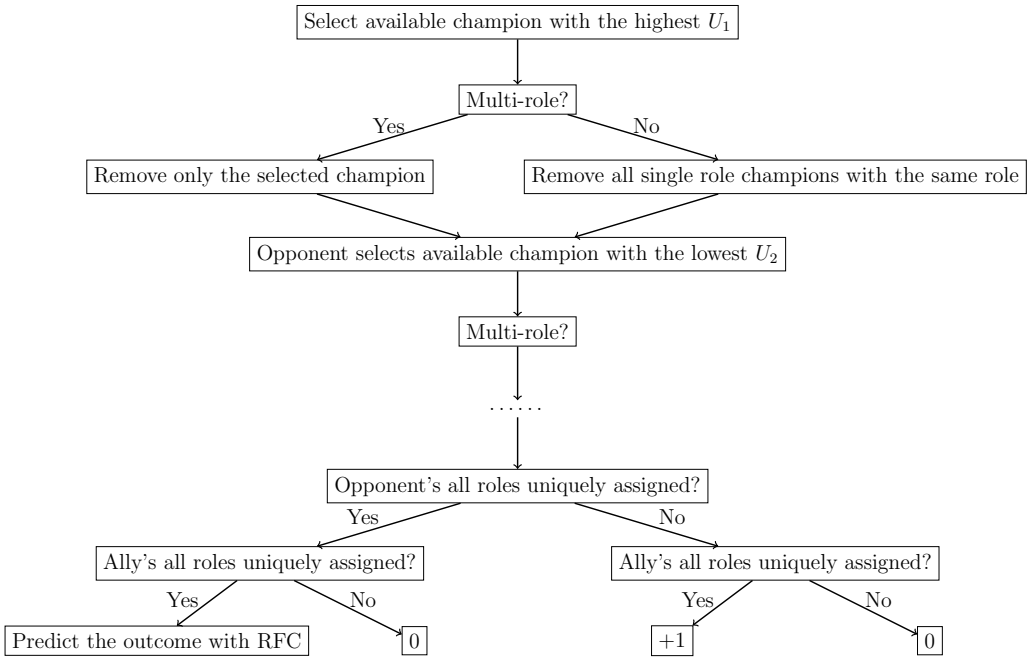


Figure 13: MCTS algorithm for League of Legends summarized.

5.6 Parallel MCTS algorithm

We see that the MCTS algorithm can help us find the best picking sequence through a random searching technique. However, the MCTS algorithm is computationally heavy as we have on average approximately 100 legal moves per step, compared to only 31.1 moves on average in chess. Parallelizing the MCTS algorithm is a decent solution to the mentioned problem. There are multiple types of parallelization including Tree Parallelization, Leaf Parallelization, and Root Parallelization. For a Go game, we see that Root Parallelization and Leaf Parallelization with local mutexes and virtual loss yield the best result [CWH08].

- Root Parallelization of MCTS: To realize root parallelization of the MCTS algorithm, we first run the MCTS algorithm simultaneously on all cores of the CPUs and record all the results. Next, we may either compare the results and select the one with the highest visit count or select the one that appears most frequently. We will see later in this

section that the more iterations each of the MCTS algorithms has, the more likely repetitions of the results appear.

- Leaf Parallelization with local mutexes and virtual loss: We may consider the standard MCTS algorithm to be a thread trying to visit the nodes by some criteria and send back information along the visited path. This Parallel MCTS can therefore be considered as multiple threads trying to visit the nodes simultaneously. Moreover, we should note that in order for different threads to process the searching algorithm in different nodes, we should wisely impose an auxiliary parameter l for all nodes that are searched in parallel. The parameter l stands for the virtual loss. Whenever a node is visited, the virtual loss increases. Thus, threads in other cores tend to visit other nodes. One possible way of applying virtual loss is to leave the UCT1 formula unchanged, but to let virtual loss be 1 and add it to the number of visits of the nodes. Threads tend to visit nodes with high winning rates as well as a small number of visits, adding virtual loss to the number of visits parameter makes different threads choose different nodes unless some node has a significantly high winning rate, as desired. Note that in this manner, computing N iterations with multi-threading is no other than computing kN iterations of the normal version of the MCTS algorithm, where k stands for the number of cores available. The mechanism of this type of Parallel MCTS algorithm is shown in Figure 14.

5.7 Results

Suppose at the start of the B/P phase we are on the blue side and our opponent is on the red side. We use the above algorithm to find the best champion possible. The result is shown in Table 21. We see that more iterations lead to a more accurate result. This can be observed by noting that the predicted winning rate decreases as the number of iterations increases. Because the blue side and the red side are supposed to be equally strong and the MCTS algorithm always assumes that the opponent has the best counter-pick possible, the predicted winning percentage eventually converges to the baseline winning rate of 0.5. Therefore, when the winning rate of the selected champion is close to 0.5, we know that the number of iterations is large enough for us to get a good pick.

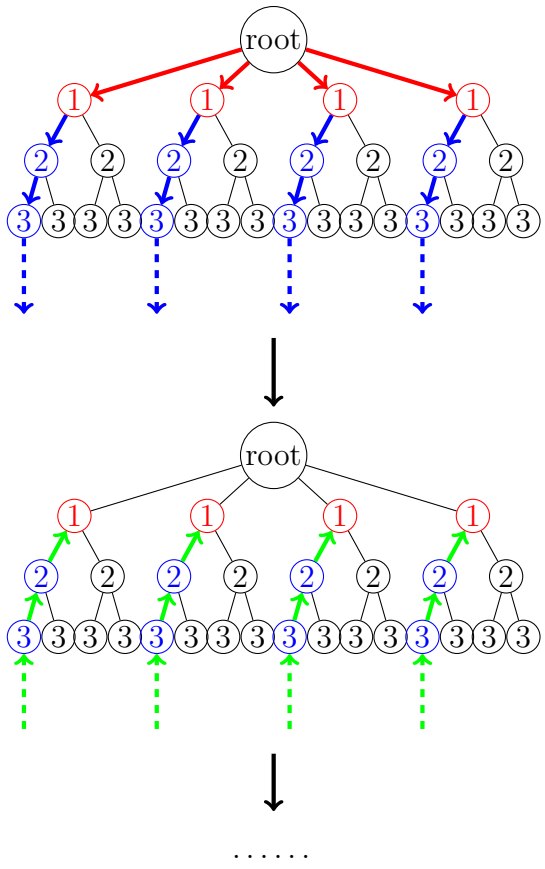


Figure 14: The mechanism of the Parallel MCTS algorithm.

Selected champion	Winning rate	Iterations	Computation time
Kalista	0.85	1000	3.38
Zeri	0.62	5000	18.91
Xin Zhao	0.60	10000	40.88
Miss Fortune	0.54	50000	163.10
Zeri	0.53	100000	331.50

Table 21: Picks made by standard MCTS algorithm.

Next, we show the theoretical results that can be obtained by using root parallelization. That is, using the same settings as before, we run the standard MCTS algorithm eight times simultaneously and select the champion with the most picks out of the eight best picks the standard MCTS algorithm generates. The result is shown in Table 22.

Selected champion	Winning rate	Iterations	Computation time
Caitlyn, Zoe, Blitzcrank, Kindred, Nocturne, Kog'Maw, Azir, Xayah	0.65, 0.73, 0.71, 0.75, 0.74, 0.74, 0.77, 0.76	1000	~ 4
Volibear, Olaf, Caitlyn, Karthus, Rek'Sai, Miss Fortune, Poppy, Vi	0.65, 0.66, 0.63, 0.64 0.61, 0.61, 0.62, 0.61	5000	~ 20
Viego, Ezreal, Kalista, Skarner, Bard, Skarner, Skarner, Kai'Sa	0.57, 0.57, 0.58, 0.6, 0.56, 0.56, 0.58, 0.61	10000	~ 40
Jhin, Zeri, Poppy, Bel'Veth, Zeri, Lee Sin, Zeri, Wukong	0.54, 0.53, 0.55, 0.53, 0.54, 0.54, 0.54, 0.54	50000	~ 200

Table 22: Picks made by Parallel MCTS algorithm, only theoretical computation time is given.

In the above table, the champions highlighted in red denote the champions with the highest winning rate and the champions highlighted in purple denote the champions that appear to be the best pick most frequently. We notice that more iterations lead to more accurate results as before; at 50000 iterations, the Parallel MCTS algorithm selects "Zeri" most frequently, which is in accordance with the standard MCTS algorithm with 100000 iterations. Moreover, we observe that the more iterations, the more likely our algorithm selects the same best champion in eight separate runs, which gives us even more confidence in picking the champion with the most appearances.

5.7.1 MCTS vs Minimax

To compare the performance of the MCTS algorithm against the Minimax algorithm, one of the difficulties we may encounter, compared with the method mentioned in [Che+20], is that we cannot ensure that both sides of the teams are equally strong since our algorithm is "tailored" for specific teams while the authors in the mentioned paper only compare the strengths of the team composition. In other words, letting the best team play on the blue side and the worst team play on the red side would almost certainly lead to a win for the blue side, no matter what champions are picked. Therefore, we aim to select teams that have approximately the same winning rates on both sides and we let this team play against itself. In other words, we try to eliminate all factors other than the team compositions that influence the outcome of the game. In Figure 15, we show the winning rate of all teams when they are playing on the blue side and the red side. The blue bars denote the winning rate on blue sides and the red bars denote the winning rates on red sides; the purple bars denote the overlapping between the blue and the red bars. It is obvious that the larger the ratio of the purple bars, the closer the winning rates are between the blue and the red side. We observe that "Weibo Gaming" has the closest winning rate on the blue side and the red side, and it is highlighted in red. We will therefore compare the algorithms in various aspects while letting both sides of the teams be "Weibo Gaming".

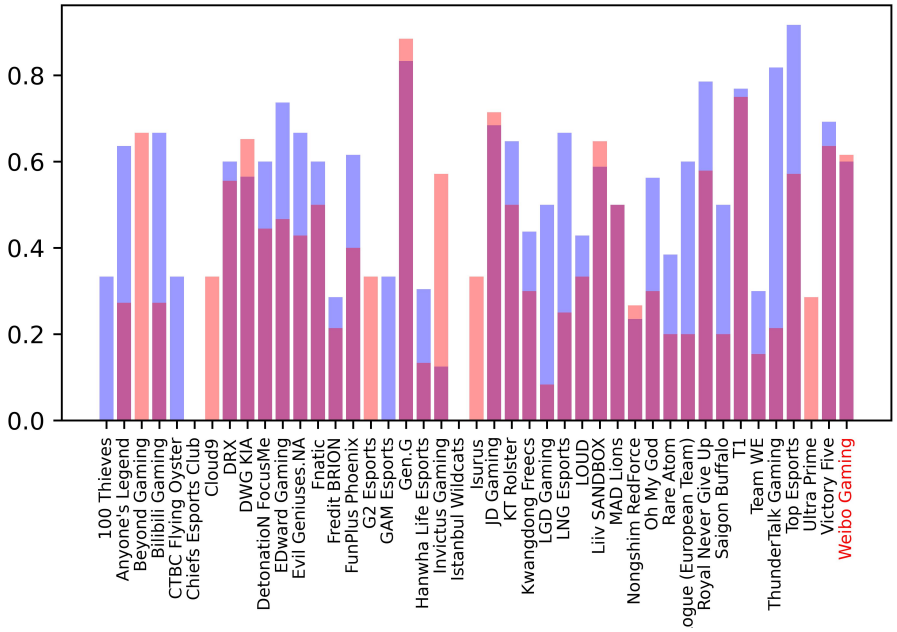


Figure 15: Winning rate of the teams when playing on the blue side and the red side.

First, we let the MCTS algorithm select the champions on the blue side and, as its adversary, the Minimax algorithm selects on the red side. Then we let the Minimax algorithm select the champions on the blue side and the MCTS algorithm select on the red side. Moreover, we impose a time limit when running the algorithms and we will compare the winning rate of one algorithm against the other with respect to computation time. The results are listed in Figure 16.

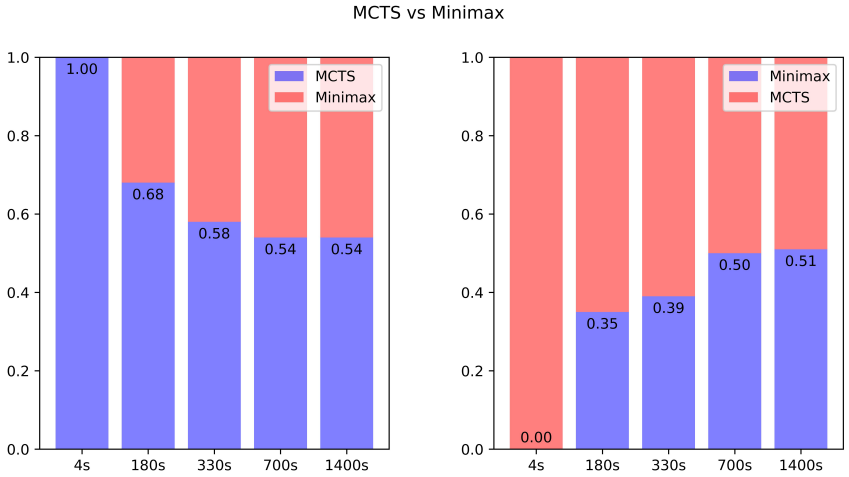


Figure 16: Winning rate of the MCTS algorithm against the Minimax algorithm when imposing a time limit.

We notice that when the allowed computation time is short, the MCTS algorithm beats the Minimax algorithm with an overwhelmingly large advantage. However, as the computation time limit becomes larger, the two algorithms gradually become tied. This is due to the fact that both the MCTS algorithm and the Minimax algorithm eventually find the optimal pick; the only difference is that the MCTS algorithm does not select all combinations of picks equally often while the Minimax algorithm does. The MCTS algorithm selects the "good" picks more often and has a larger depth in searching.

Moreover, we are also interested in the number of iterations of the MCTS algorithm that is necessary to at least tie the well-performed Minimax algorithm. We see that using the time limit of 1400 seconds, the Minimax algorithm is able to find near-optimal picks in the previous section. The time limit of 1400 seconds corresponds to 100 Monte Carlo simulations for predicting the payoffs of all pairs of champions, we therefore use 100 and the baseline 10 as our parameters for the Minimax algorithm and we compare it with the MCTS algorithm with different parameters. The results are shown in Figure 17 and Figure 18.

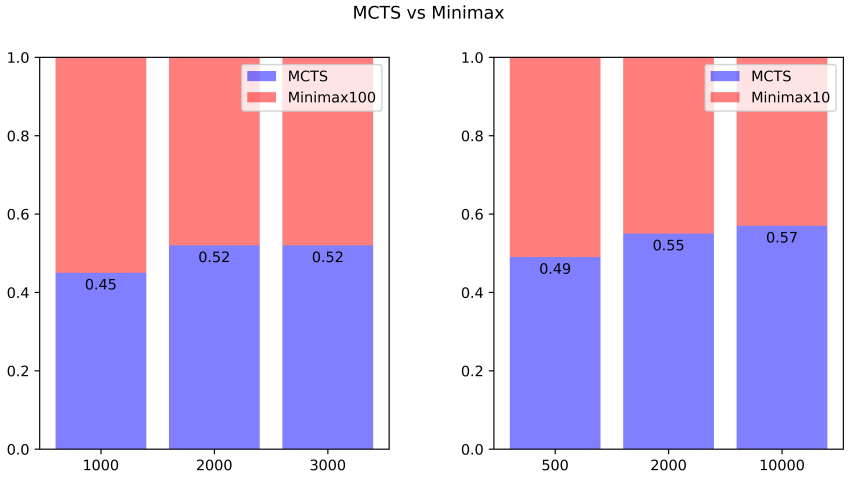


Figure 17: Winning rate of MCTS algorithm on the blue side with different number of iterations against Minimax algorithm with 100 and 10 iterations.

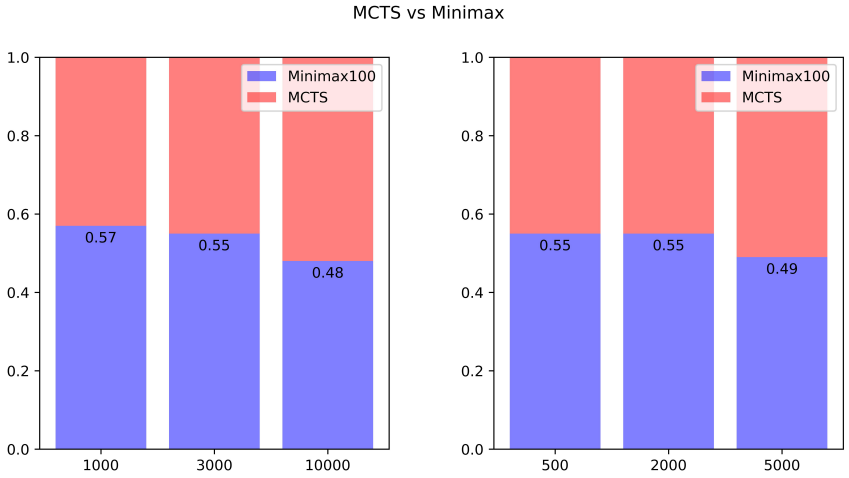


Figure 18: Losing rate of MCTS algorithm on the red side with different number of iterations against Minimax algorithm with 100 and 10 iterations.

We notice that when the MCTS algorithm is on the blue side, it requires 2000 and 500 iterations to beat the Minimax algorithm with 100 and 10 iterations half of the time while it requires far more iterations, 10000 and

5000 when the MCTS algorithm is on the red side. 3000 and 10000 iterations of the MCTS algorithm cost 12 seconds and 40 seconds respectively, so on average we are 50 times faster by using the MCTS algorithm instead of the Minimax algorithm.

6 Conclusion and future works

In this paper, we proposed a new method of selecting champions during the B/P phase of League of Legends professional matches by using a fine-tuned Random Forest classifier together with the MCTS algorithm. We compared different methods of predicting the winner of a game including Logistic Regression, Neural Network, Random Forest, and Embedding, given all the related information. After finding that the Random Forest outperformed the rest, we further fine-tuned this classifier and found the best parameters that increased the AUC and accuracy even more. Furthermore, we used the MCTS algorithm to select champions by finding a system of distinct representatives and compared this algorithm to the well-known Minimax algorithm. With merely 1/50 of the computation time, the MCTS algorithm is already able to beat the Minimax algorithm.

However, there are also limitations which can be summarized in the following aspects. First, we did not take the banning phase into account because banning champions is closely related to personal favorites. Moreover, we have a limited number of matches available. Out of 163 champions, we only have 112 champions in our record; the algorithm did not consider the rest of the 51 champions at all. Also, we were not able to perform true root or leaf parallelization because the programming language we use, i.e., Python, has the notorious GIL (Global Interpreter Lock), which does not allow true multi-threading. Because of that, we could only calculate the theoretical computation time. Lastly, we did not consider the Multi-round version of the MCTS algorithm, while most professional matches are either best-of-3 or best-of-5. The Multi-round MCTS is introduced in the following subsection and, in the future, we will work on this aspect.

6.1 Future work: Multi-round MCTS

In some of the League of Legends regions in 2024, officials will be implementing a new regulation for competitive matches. According to this regulation, any champions that a team picks in a round cannot be picked again by that team in later rounds, but the same champions that have been banned can be banned again. Therefore, in order to win the entire match, teams must choose their champions wisely, considering their picking strategy for future rounds. To address this, we modify our MCTS algorithm to the Multi-round Parallel MCTS algorithm. Figure 19 shows the mechanism of this algorithm.

The nodes and arrows have the same indications as before. The difference between the Multi-round MCTS and the normal MCTS is that after the searching algorithm reaches the leaf node of the first round, it continues its simulation to further rounds, and the legal moves reset after each round with the previously selected champions removed. Additionally, as each time the thread reaches a leaf node, a round ends, we need to predict the outcome of the game for every visited leaf node. As the results of early rounds influence the results of later rounds, but not vice versa, we let the total reward be the reward backpropagated from all the leaf nodes further down the tree. In other words, the reward is given by the formula $\sum_{i=n}^{N} v_i$, where N denotes the total number of rounds, and n denotes the current round of the node. Therefore, the reward in the figure is given by $v_1 + v_2$, as we are currently at nodes of depth 1 in the very first round.

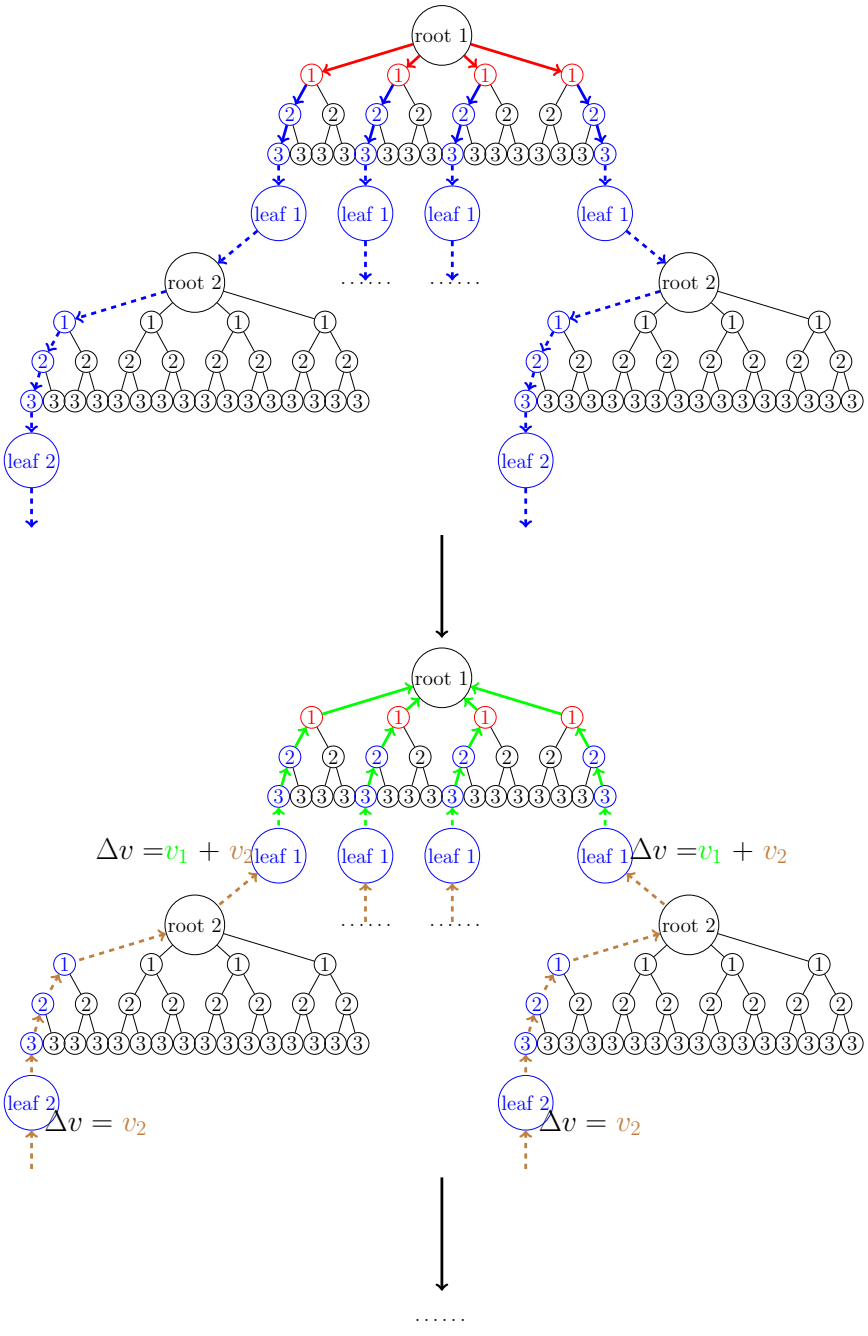


Figure 19: The mechanism of the Multi-round Parallel MCTS algorithm

References

[Pet+02] Nicolò Cesa-Bianchi Peter Auer et al. "Finite-time Analysis of the Multiarmed Bandit Problem." In: *Machine Learning* (2002), pp. 235–256. DOI: 10.1023/A:1013689704352.

[CWH08] Guillaume M. J. -B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. "Parallel Monte-Carlo Tree Search". In: *Computers and Games*. Springer Berlin Heidelberg, 2008, pp. 60–71. DOI: 10.1007/978-3-540-87608-3_6. URL: https://doi.org/10.1007/978-3-540-87608-3_6.

[Lea10] League of Legends. *League of Legends — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-December-2022]. 2010. URL: https://en.wikipedia.org/wiki/League_of_Legends.

[Bru11] J. Bruin. *newtest: command to compute new test @ONLINE*. Feb. 2011. URL: https://stats.oarc.ucla.edu/stata/ado/analysis/.

[JR11] Marshall HALL JR. "Distinct Representatives". In: *Combinatorial Theory*. John Wiley & Sons, Inc., Aug. 2011, pp. 48–72. DOI: 10.1002/9781118032862.ch5. URL: https://doi.org/10.1002/9781118032862.ch5.

[Koe18a] Will Koehrsen. *Hyperparameter Tuning the Random Forest in Python — towardsdatascience.com*. https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74. 2018.

[Koe18b] Will Koehrsen. *Neural Network Embeddings Explained — towardsdatascience.com*. https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526. 2018.

[Bar19] David Barnes. *What is the average number of legal moves per turn? — chess.stackexchange.com*. https://chess.stackexchange.com/questions/23135/what-is-the-average-number-of-legal-moves-per-turn. 2019.

[Mis19] Mistreaver. *History Of Chess Computer Engines - Chessentials — chessentials.com*. https://chessentials.com/history-of-chess-computer-engines/. 2019.

[Che+20] Sheng Chen et al. *Which Heroes to Pick? Learning to Draft in MOBA Games with Neural Networks and Tree Search*. 2020. DOI: 10.48550/ARXIV.2012.10171. URL: https://arxiv.org/abs/2012.10171.

[Tra20] Eoin Travers. *Why does logistic regression overfit in high-dimensions? — Eoin Travers — eointravers.com*. http://eointravers.com/post/logistic-overfit/#:~:text=Logistic%20regression%20models%20tend%20to,closely%20to%20th%20trainin%20data.. 2020.

[R21] Sruthi E R. *Understand Random Forest Algorithms With Examples (Updated 2023) — analyticsvidhya.com*. https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/#:~:text=Random%20Forest%20reduces%20overfitting%20by,feature%20selection%20and%20data%20interpretation.. 2021.

[Sha21] Mohammed Shammeer. *Random Forest Fails — medium.com*. https://medium.com/swlh/random-forest-fails-a8ca2d46c312. 2021.

[Cri23] Jacob Crick. *League of Legends — Red Side vs. Blue Side — pinnacle.com*. https://www.pinnacle.com/en/esports-hub/betting-articles/league-of-legends/league-of-legends-red-side-vs-blue-side/kg2jey3lv9q726yz. 2023.

[Xu23] Davide Xu. *How many Champions are in League of Legends? - List by Class and Role — esports.net*. https://www.esports.net/news/lol/how-many-champions-are-in-league-of-legends/. 2023.

[Has] Muhammad Hassan. *Categorical Variable - Definition, Types and Examples — researchmethod.net*. https://researchmethod.net/categorical-variable/.

[IBM] IBM. *What is Logistic regression? — IBM — ibm.com*. https://www.ibm.com/topics/logistic-regression.

[LCK] LCK. *LCK/2022 Season/Summer Season/Picks and Bans — lol.fandom.com*. https://lol.fandom.com/wiki/LCK/2022_Season/Summer_Season/Picks_and_Bans.

[LPL] LPL. *LPL/2022 Season/Summer Season/Picks and Bans — lol.fandom.com*. https://lol.fandom.com/wiki/LPL/2022_Season/Summer_Season/Picks_and_Bans.

[Rob] Chris Roberts. *Matchmaking and Champion Select - Fall 2022 - League of Legends — leagueoflegends.com*. https://www.leagueoflegends.com/en-gb/news/dev/matchmaking-and-champion-select-fall-2022/.

[wor] worlds. *2022 Season World Championship/Match History — lol.fandom.com*. https://lol.fandom.com/wiki/2022_Season_World_Championship/Match_History.

[Zha+] Brandon Zhao et al. *League of Legends: An Exploratory Data Analysis — ucladatares.medium.com*. https://ucladatares.medium.com/league-of-legends-an-exploratory-data-analysis-11f6022f18be.