# League of Legend Match Outcome Prediction at Different Stages

*Yunxuan Li*

*Jie Chen*

*May 4th, 2018*

## Abstract

In this report, we investigate on how to make accurate prediction on the outcome of a League of Legend ranked match at different times. We looked at information available at three time points: pre-match, 10-minute benchmark, and the end of match. Specifically, we compare between players with same roles on different team, and use the difference statistics as features. Machine learning methods including Logistic Regression, K-Nearest Neighbors(KNN), Support Vector Machine(SVM), Random Forest, Gradient Boosting Algorithms(Gradient Boosting and Xgboosting), and Multilayer Perceptrons(MLP) are implemented and their performance are compared. Discussions include conclusions and inferences from our data analysis.

## Introduction

League of Legends (LOL) is a multiplayer online battle arena (MOBA) video game operated by *Riot Games*. Each LOL match contains two teams (blue team and red team) of five players, where each player chooses one unique champion and collaborates with the team to destroy the nexus of the enemy team. In each team, each player has a different role: top, jungle, mid, adc and support. There are three lanes and jungle area along the map. Usually a match starts with two top players from each team playing against each other at the top lane, two mid players at the mid lane, the adcs and supports together at the bottom lane, and jungles in jungle area.

When a player entered the game, he/she is automatically matched with other four players and form a team to compete with the other randomly-matched team. Players on both team can each ban a champion, and pick a champion sequentially. The side which first-picks a champion is the blue team, and the other is the red team. Winning a match will increase the rank score of the player, and losing one will decrease that. Based on rank score, players are divided into different ranks: Bronze, Silver, Gold, Platinum, Diamond, Master, and Challenger.

Since players play ranked games to increase their rank and to climb higher on the rank ladder, winning is the sole purpose of a ranked game. It is thus natural to come up with the question: can we predict which team will win the game before a ranked game actually ends? Does our prediction accuracy vary based on the amount of information that we have at a specific time point?

Based on information and data that are availble, we build machine learning models for three different time points: before the match starts, when the match has started for 10 minutes, and right before the end of the game. At each of these three time points, the data we have are pre-match data, early-game statistics, and all in-game information. Pre-match data includes the the differences in current rank of players, and the differences in player's level of mastery on the selected champion in this game. Early-game statistics includes the differences in gold, experience, damage taken, etc., counted at 10-minute benchmark. Full in-game information includes all the statistics mentioned above, as well as some other statistics such as the total amount of damage dealt to the enemy, the number of kills and assists, and etc.

Due to the fact that the prediciton outcome is whether blue-team wins or loses, we explore several classification algorithms including Logistic Regression, K-Nearest Neighbors(KNN), Support Vector Machine(SVM), Random Forest, Gradient Boosting Algorithms (GBM). We compare how the prediction accuracy changes across these time points.

Our hypothesis is that the prediction accuracy increases as time goes, because the longer the play has played, the more information we will have and make use of to make predictions. Specifically, we hypothesized that the pre-match prediction accuracy is low and should be around 50%, because the matchmakinging system is supposed to create two balanced team for fairness. However, due to the champion pick uncertainty (e.g., if a player is very fluent at his champion while his enemy is not), pre-match information may still play a role in generating accurate predictions. Therefore, the prediction accuracy should be somewhere higher then 50%. Second, we hypothesized that prediction accuracy at 10-minute benchmark is high, because League of legend is mostly believed to be a snowballing game, where early advantage is more likely to lead to a win. Finally, we believe that with all in-game statistics, the prediction accuracy at the end of the match should reach 100%.

We believe this question is worth exploring for LOL players: if early-game-based prediction is accurate enough, players can adjust their strategy more accordingly. For instance, they can play more aggressive in early games to build up advantage and snowball til victory. Besides, it will be possible to know which team is the game in favor of at an early point when watching professional LOL champion series.

In addition, this problem shall also be interesting to non-players: the data for many sport matches are in the same format as the LOL match, where each player's statistics are recorded at different time points. Thus, the data processing and analyzing methods applied to this specific problem are also generalizable to a much broader perspective, to many kinds of different sports competition.

Currently there are two to three papers researching on relevant topics. For instance, Lucas Lin has tried gradient boosted trees with/without logistic regression in predicting game results with all in-game information. Similarly, Cabrera (2016) has studied the influence of early-game data on the outcome of matches. These previous researches are enlightening. However, past researches used dataset available online, which are now relatively out-dated and thus do not reflect true information in current version of LOL; besides, they did not investigate some varibales that we believe might be important. In our case, we will extract our own match dataset through official API and create our own unique set of variables in making predictions. We also explore some different algorithms.

## Methods

### Data Sampling and Pre-processing

We collect match dataset through the Riot Games API, official API of League of Legend. In order to generate a dataset consisting of random matches across different ranks, we used the following process: we started with a random player at each rank (Silver, Gold, Platinum, and Diamond), rertieve all his recent ranked games with his player id, selected one game as our sample and retrieve this specific match with match ID. We then randomly pick another player in those ranked games to repeat the procedure with the new player ID. Because the League matchmaking system matchs players with similar strentgh, our method is supposed to be able to collect matches only in selected rank. Note that we collect match data from the Korean server instead of North American server, because matches in NA region usually involves cross-server players from Latin American server and European server, which makes automatic data retrieval much harder.

Based to the rank distribution of League of Legend, we Collected 3000 matches for

Silver, 1500 matches for Gold, 800 matches for Platinum, and 300 matches for Diamond. Bronze matches are ignored because most players are placed and started with Silver, and in most times inactive players are more likely to be demoted from Silver to bronze. Therefore, retrieving matches from Bronze players are harder because many players play ranked games often, making the random generation of match dataset much harder.

We perform data sampling and preprocessing with R for convenience. We create functions which serve to automatically pick player ID, retrieve player's ranked games with player ID, automatically pick a match ID from games with a for-loop, without violating the API rate limit policy. We end with a list of match ID, with which we can retrieve the detailed match information from the API.

The match data returned by Riot API are in JSON format, containing various information for each of 10 players. We processed the JSON match data of each match into one row, so that the final dataset will be a dataset that is clear and easy to investigate.

We believe the differences in players who have same roles on different teams are worth comparing. For instance, two mid players start the game playing at the mid lane against each other, and their different behaviors and resulting statistics should be an influential feature on the match outcome. As a result, for each match, we start by writing scripts that identify each player's role in both teams: Top, Jungle, Mid, ADC, and Sup. Matches of which the player roles are vague and not identifiable are dropped. We then pair up two players whose roles are same, and calculate their difference statistics, (blue team player statistics - red tem player statistics, e.g., $\Delta_{damage\ taken\ at\ 10\ minutes}$). We calculate these numbers for each of the five roles, and formulate results into a single row. If some information of one player is missing (indicated by NaN), the difference statistics of that role is filled with 0, in accordance to our assumption that by matchmaking systems, players should have similar backgrounds and should not outplay corresponding enemy.

We then normalize the dataset. Because most of our variables are differences in some measurements between two players, these numbers should be flowing around 0. Therefore, we normalize our data with a standard scalar, to standardize our features by removing the main and scaling to unit variance. Scaling helps because it helps tuning parameters and optimizing performances of algorithms.

Feature selection is done for some algorithms, such as KNN and SVM. We use select features based on their feature scores, which are the ANOVA F-values of each feature. Feature selection helps as it removes abundant features, reduces over-fitting, and makes model easier to interpret. For algorithms such as Random Forest, we do not use feature selection, because the algorithm itself selects some predictors in each trees and there is no need to select anymore. We do not use feature extraction, which is usually applied when there is a huge amount of features.

The final dataset is thus a matrix with row number equaling to number of games, and column number equaling to number of features. Each row of our dataset contains information of one single match. For each match, the columns consist of three parts of information: pre-match information, players' statistics at 10-minute benchmark, and players' statistics before the end of the game. There is also a column recording the winning side.

**Core Methods: Algorithms**

As mentioned above, we are interested in how prediction accuracy changes with the amount of information we have, and the performances of different machine leaning algorithms. For each algorithm, we conduct three-stage predictions using pre-match data, data of first ten minutes, and full in-game data.

We split the data into training and test set, use cross-validation within training set to tune the parameter, and use test set to measure the performance of each algorithms. We implement the following methods on training data:

- Logistic Regression

- K-Nearest Neighbor Algorithm
- Support Vector Machines
- Random Forest
- Gradient Boosting

***Logistic Regression:***

Logistic Regression Classifer is the most primary method for binary classification problems. $Y$ has two possible outcomes, 0 and 1.

$$p(Y) = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n}}{1+e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n}}$$

. With $P$ as the probability of Y occuring, $\beta_0$ is the interception at $y-axis$, $x_1,\ldots,x_n$ as the predictors and $\beta_n$ as the regression coefficient of $x_n$. Logistic regression predicts the probability of $Y$ taking a specific value (Cohen, J., P. Cohen, S.G. West, L.S. Aiken. 2003).

***K-Nearest Neighbors Algorithm (KNN):***

K-Nearest Neighbors Algorithms is suitable for this problem, since our response variables include only 0 and 1 to indicate blue team winning or losing the game. The idea behind KNN methodology is to at first learn from the training set to determine the number of near neighbors that are needed to predict a label, and then predict the label of new instances based on the labels of its K closet neighbors in the training set. The mathematical way to understand this is assuming that our instance domain $\chi$ is endowed with a metric function $\rho$, which returns the distance between two elements of $\chi$. For each of the $x \in \chi$, let $\pi_1(\mathbf{x}), \pi_m(\mathbf{x})$ be a reordering of $\{1,\cdots,m\}$ according to distance to $\mathbf{x}$, $\rho(\mathbf{x},\mathbf{x_i})$. Given a training sample $S=(x_1,y_1),\ldots,(x_m,y_m)$, for every point $\mathbf{x}\in\chi$, the KNN method returns the majority label among $\{y_{\pi_i(x)} : i \leq k\}$.

To implement KNN, we need to first decide the numbers of features we would like to use that would lead to the most accurate prediction possible. This is done using cross-validation. Noticing that we also need to decide the number of neighbors, we ends up with two layers of cross-validation. The cross-validation gives us the best combination of the number of features and the number of neighbors for KNN algorithm. We then do feature selection based on features selected in cross-validation, feed the entire training set to the KNN model with the fixed number of neighbors determined via cross-validation, and make predictions with the test set. We do the cross-validation and predictions for each of three stages.

***Support Vector Machines Algorithm (SVM):***

SVM is also popular method for binary classification problems. We use sklearn.svm.SVC in python to fit a model and make predictions( http://scikit-learn.org/stable/modules/svm.html#svm-classification). Given training vectors $x_i \in \mathbb{R}^p, i = 1, \ldots, n$, in two classes, and a vector $y \in \{1, -1\}^n$, SVC solves the following problems:

$$min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

subject to $y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i$,

$$\zeta_i \geq 0, i = 1, \ldots, n$$

Its dual is

$$min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$

subject to $y^T \alpha = 0$

$$0 \leq \alpha_i \leq C, i = 1, \ldots, n$$

where $e$ is the vector of all ones, $C \geq 0$ is the upper bound, $Q$ is an $n$ by $n$ positive semidefinite matrix,$Q_{ij} = y_i y_j K(x_i, x_j)$, where $K(x_i, x_j)$ is the kernel. The kernel we specify in our model is $rbf : exp(-\gamma||x - x'||^2)$, with $\gamma > 0$.

To implement SVM, we need to first decide the numbers of features that lead to the best performance. We also need to decide the value of $\gamma$ in the kernel function and the the upper bound $C$. In this case, we design a cross-validation with three layers. After feeding training set into the cross-validation process, we have the best combination of features, $\gamma$ and $C$ that can be used to build the model. We then fit a SVM model with these parameters with training set, and make predictions with the test set. We do the cross-validation and predictions for each of three time-stages.

***Random Forest:***

A random forest consists of a collection of decision trees on various sub-samples of the dataset. By averaging over a bunch of single decision trees, this algorithm reaches the higher predictive accuracy and controls over-fitting. sklearn.ensemble.RandomForestClassifier allows us to make predictions using random forest classifer. We set the boostrap to be true so that each tree in the ensemble is built from a sample drawn with replacement from the training set. This will reduce variability of decision trees.

To implement random forest, we need to decide the value of n_estimators, which is the number of trees in the forest, and max_depth, which is the maximum depth of each tree. The cross-validation gives us the best n_estimators and max_depth. We train the random forest classifier with training set and parameters from cross-validation, and then make predictions using pre-match data, early-game data and full match data.

Specifically, because pre-match and full-match analysis are not that helpful in predicting the outcome of a game, we focus on early-game data and look at which variables are most important in the random forest model.

***Gradient Boosting:***

In addition to random forest, we also try gradient boosting classifier to fit the model and make predictions. We got references from Jerome H. Friedman's 2001 paper and sklearn.emnsemble.GradientBoostingRegressor for gradient boosting algorithm and implementation methods.

For gradient boosting, the number of weak learners (i.e. regression trees) is controlled by the parameter n_estimators; The size of each tree can be controlled either by setting the tree depth via max_depth or by setting the number of leaf nodes via max_leaf_nodes. The learning_rate is a hyper-parameter in the range $(0.0, 1.0]$ that controls overfitting via shrinkage. This leads to a three-stage cross validation process. The cross validation gives us the best combination of the n_estimators, max_depth and learning rate. We use gradient boosting to make predictions using pre-match data, early-game data and full match data.

***XGBoosting:***

In addition to gradient boosting, we also use XGBoost to fit the model and make predictions. According to Tianqi Chen, the author of XGBoosting, the fitting procedure of XGBoosting is similar to gradient boosting, but XGBoosting used a more regularized model formalization to control over-fitting, which gives better performance. The objective functions for XGBoosting contains two parts: training loss and regularization.

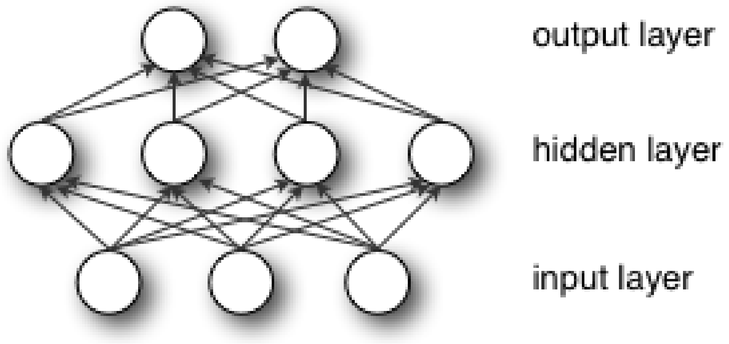
$$obj(\theta) = L(\theta) + \Omega(\theta)$$

, where L is the training loss function, and $\Omega$ is the regularization term. The training loss measures how predictive our model is on training data. The regularization term controls the complexity of the model, which helps avoid overfitting. Similar to Gradient Boosting Classifier, max_depth controls maximum depth of a tree (increase this value will make the model more complex / likely to be overfitting). The number of trees is controlled by n_estimators and learning_rate controls overfitting via shrinkage. This leads to a cross-validation with three layers.

***Multilayer Perceptrons:***

Multiplayer Perceptrons can be viewd as a logistic regression classifier where the input is first transformed using a learned non-linear transformation $\Phi$. DeepLearning 0.1 documentation has a detailed explanation of MLP. This transformation projects the input data into a space where it becomes linearly separable.This intermediate layer is referred to as a hidden layer. A MLP with a single hidden layer can be represented graphically as:

Figure 1: MLP graphical representation



Formally, a one-hidden-layer MLP is a function $f : R^D \rightarrow R^L$, where $D$ is the size of input vector $x$ and $L$ is he size of the output vector $f(x)$ such that:

$$f(x) = G(b^{(2)} + W^{(2)}(s(b^{(1)} + W^{(1)}x))),$$

with bias vector $b^{(1)}, b^{(2)}$,weight matrices $W^{(1)}, W^{(2)}$ and activation functions $G$ and $s$. The vector $h(x) = \Phi(x) = s(b^{(1)} + W^{(1)}x)$ constitutes the hidden layer. $W^{(1)} \in R^{D \times D_h}$ is the weight matrix connecting the input vector to the hidden layer. The output vector is obtained as $o(x) = G(b^{(2)} + W^{(2)}h(x))$.

To implement MLP, we need to decide the value for learning rate, number of hidden units in layer 1 (n1 setting) and the number of hidden units in layer 2 (n2 setting). Many literatures mentioned that decaying learning rate overtime may be a good choice for MLP, but in terms of simplicity, we choose to implement constant learning rate. In cross validation, we tried several log-spaced values $(10^{-1}, 10^{-2}, \ldots)$ in our search. The hidden units in each layer is dataset-dependent. Based on the number of input variables in prematch, early game and full game dataset, we choose different cross validation values for hidden units in each layer. After feeding training set into the cross-validation process, we have the best combination of learning rate, number of hidden units in layer 1 and number of hidden units in layer 2 that can be used to build the model. We then fit a MLP with these parameters with training set, and make predictions with the test set. We do the cross-validation and predictions for each of the three time-stages.

## Results

In this section, the parameters chosen from cross-validation process for each algorithm are reported, and their prediction accuracy on the test dataset are shown in $Table$ 0.1. We include the cross-validation result plot of early-game based prediction for each of the algorithms as a way to show how to determine parameters through corss-validation.

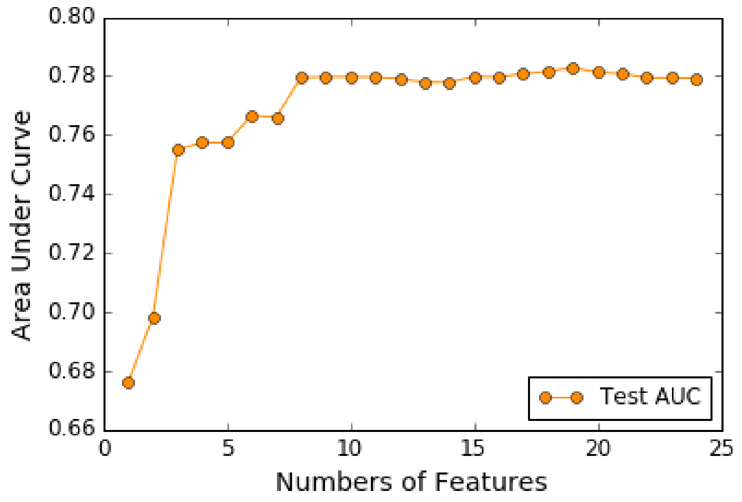
In addition, we would like to investigate whether the current accuracy achieved is the maximum ability of the model, or is still limited by the amount of traning data used. We choose the model which has the best performance, and plot the test accuracy achieved against the amount of training data that we feed into the model.

Graphical and numerical results are presented in this sections; inferences and further discussions of results are included in next section.

***Logistic Regression:***

- Prematch: 4 features.
- Early-game: 18 features. Figure 2.
- Full-game: 6 features.

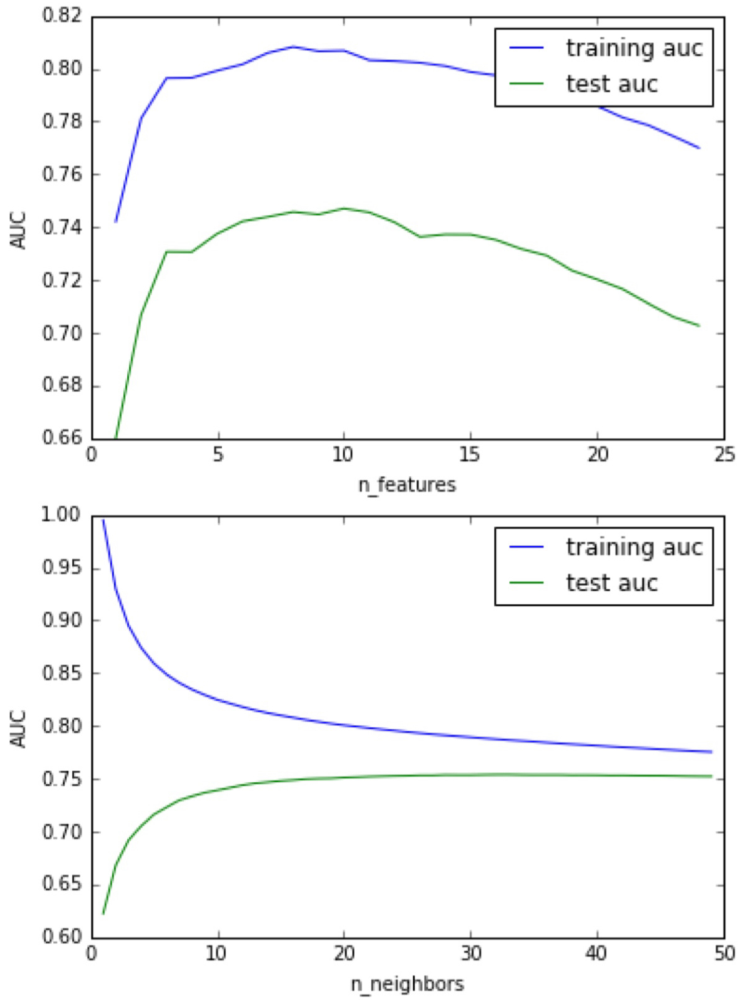
Figure 2: Logistic Regression Early-Game Cross-Validation Plots



***K-Nearest Neighbors Algorithm (KNN):***

- Prematch: 4 features, 50 neighbors.
- Early-game: 10 features, 50 neighbors. Figure 3.
- Full-game: 5 features, 18 neighbors.

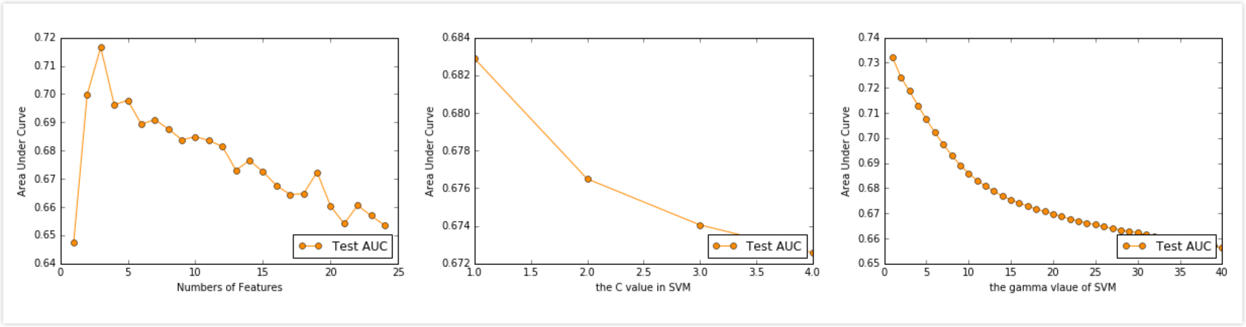
Figure 3: KNN Early-Game Cross-Validation Plots



***Support Vector Machines Algorithm (SVM):***

- Prematch: 4 features, $\gamma = 0.2$,$C = 1$.
- Early-game: 3 features, $\gamma = 0.1$,$C = 1$. Figure 4.
- Full-game: 6 features, $\gamma = 0.1$,$C = 2$

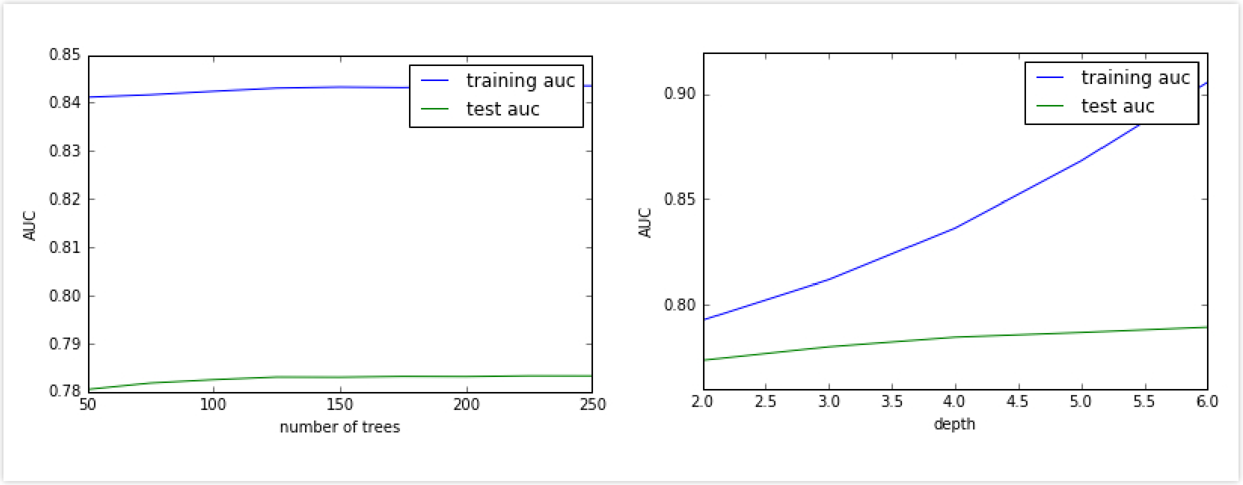
Figure 4: SVM Early-Game Cross-Validation Plots



***Random Forest:***

- Prematch: $n\ estimators = 125$, $max\ depth = 6$
- Early-game: $n\ estimators = 130$, $max\ depth = 8$. Figure 5.
- Full-game: $n\ estimators = 125$, $max\ depth = 6$.

Figure 5: Random Forest Early-Game Cross-Validation Plots

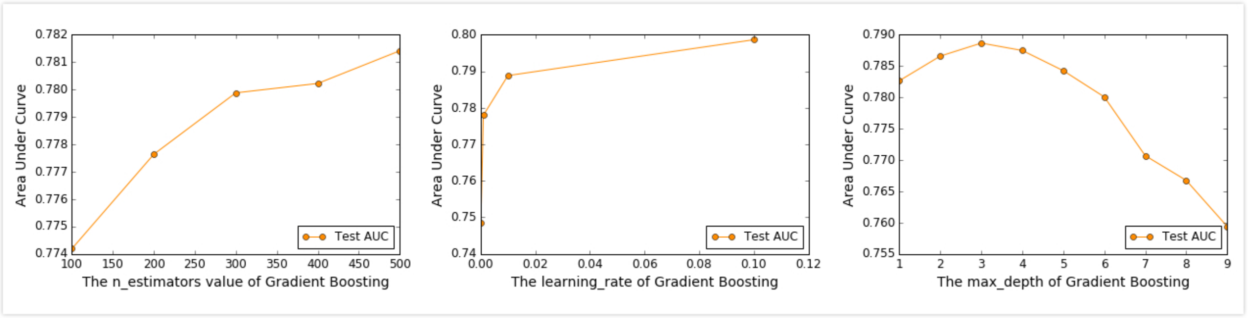


The three most important variables in the variable importance plot of Random Forest: 'Gold10Dif_Mid', 'Gold10Dif_Jg', and 'Gold10Dif_ADC', with corresponding variable importance: 0.11780101, 0.11171752, 0.10754643. We will discuss this later.

***Gradient Boosting:***

- Prematch: $n\ estimator = 500$, $learning\ rate = 0.1$ and $max\ depth = 9$.
- Early-game: $n\ estimator = 500$, $learning\ rate = 0.1$ and $max\ depth = 3$. Figure 6.
- Full-game: $n\ estimator = 500$, $learning\ rate = 0.1$ and $max\ depth = 2$.

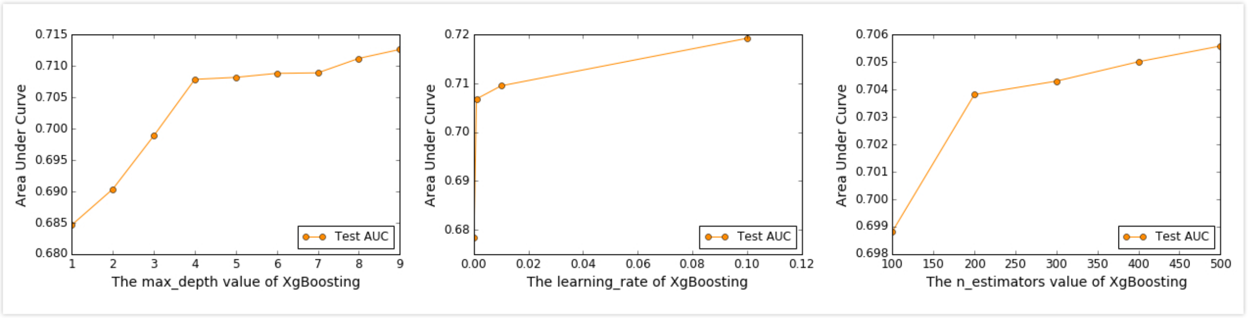
Figure 6: Gradient Boosting Early-Game Cross-Validation Plots



***XGBoosting:***

- Prematch: $n\ estimator = 500$, $learning\ rate = 0.1$ and $max\ depth = 9$.
- Early-game: $n\ estimator = 500$, $learning\ rate = 0.1$ and $max\ depth = 9$. Figure 7.
- Full-game: $n\ estimator = 500$, $learning\ rate = 0.1$ and $max\ depth = 9$.

Figure 7: XGBoosting Early-Game Cross-Validation Plots



***Multilayer Perceptron:***

- Prematch: $n1\ setting = 3$, $n2\ setting = 10$ and $learning\ rate = 0.05$.
- Early-game: $n1\ setting = 10$, $n2\ setting = 20$ and $learning\ rate = 0.05$. Figure 8.

- Full-game: $n1\ setting = 5$, $n2\ setting = 25$ and $learning\ rate = 0.05$

Figure 8: Multilayer Perceptron Full Match Cross-Validation Plots

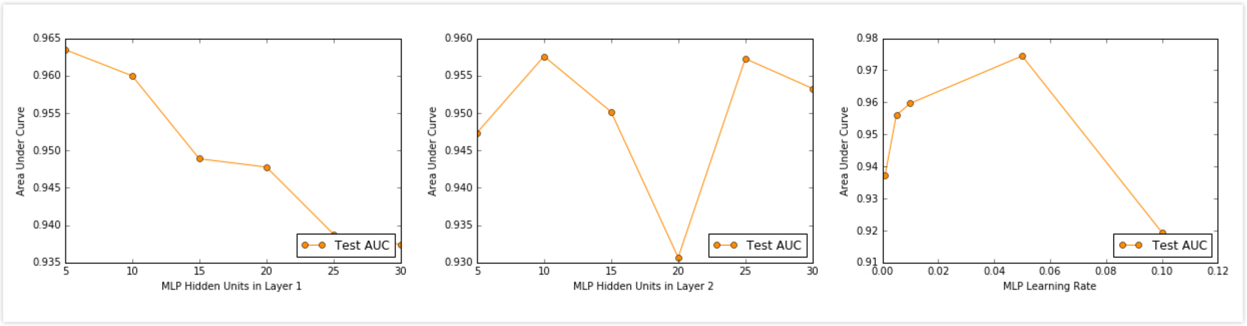


Table 1: Test Prediction Accuracy (%)

| **Method** | **Pre-match** | **Early-game** | **Full-game** |
|---|---|---|---|
| **Logistic Regression** | 49.09 | 71.48 | 82.445 |
| **KNN** | 54.94 | 79.49 | 99.49 |
| **SVM** | 55.44 | 63.32 | 99.61 |
| **Random Forest** | 53.4 | 78.03 | 99.89 |
| **GBM** | 61.24 | 75.15 | 99.72 |
| **XGBoosting** | 56.16 | 65.32 | 97.99 |
| **MLP** | 56.73 | 69.67 | 96.27 |

## Discussions

Results are discussed sequentially. Overall, our hypothesis that prediction accuracy increases as the amount of information we have increases is supported by the results. Cross-validation is implemented for each algorithm to ensure that best parameters are selected and that the model is generalizable to more real-world data.

For the prediction made on pre-match information, our algorithms generally do not perform very well. The prediction accuracies flow around 50%, which are not much better than random guessing. This makes sense, as the pre-match information we use, i.e., players' current ranks, are also the only information that LOL matchmaking system uses to match players together to form a match. The fact that prediction accuracy is around 50% indicates that the matchmaking system is pretty fair and balanced.

Previously we planned to also include players' level of mastery on their selected champions for a specific game as part of pre-match information. Champion selection happens after the matchmaking system matches players together, and before the match actually begins. As mentioned in introduction part, we believe this is a useful information to increase the prediction accuracy before the game actually starts, because if two players of same role on different teams have a big difference on their champions, the player who has a higher mastery level is more likely to win the lane and more likely to lead to the overall victory. However, during implementing scripts to retrieve each player's mastery information of a champion in each match, we found that the Korean API server for mastery retrieval is very unstable, producing error code from time to time. In addition, the fact that 10 requests are needed for one match because one request is necessary per player, together with the fact that the Riot API rate limit is very strict, makes mastery retrieval really time-consuming. Given the time limit for this assignment, we give up on collecting players' mastery level information. We will bring this issue in the discussion later.

Next, we look at the prediction results based on early-game information. Most algorithms have a prediction accuracy around 75%, with KNN almost 80%. Overall, KNN and Random Forest are two classifiers that performs the best. This result supports our belief that early-game advantage is important for LOL games. In fact, the prediction accuracy of 80% is surprisingly satisfactory to us. In previous literatures, prediction accuracy based on 10-minute information has at most an accuracy of 70% (Cabrera, 2016). Having a accuracy of 80% means that if we have the information of the first 10 minutes, we are 80% likely to successfully predict the outcome of the match, which on average lasts for 25 to 30 minutes. There are things happening at later game that may influence the outcome, but with only a small portion of the whole game information, we are already able to give an accurate prediction to things happening in future. We also notice that Multilayer Perceptron does not perform well on predictions based on early-game information. We think this is related to the nature of our data set. Neural network methods generally perform well on data set with larger number of observations and number of features. However, we only have aroud 25 features in 10-minute match data and 3400 observations, so there is not much structure left for MLP to explore and perform well.

We further look at the variable importance provided by the Random Forest classifier. The 3 most important features that have variable importance over 0.1, as described in previous section, are the gold difference between middle players, that between jungle players, and that between adc players. This makes sense, as middle players and adc players are normally responsible of doing damage in team fight. Having more gold than opponents in these roles mean the players can make more damage, and thus have a higher chance of leading the team to victory.

We believe our unique way of manipulating and creating variables is why we achieved a much higher accuracy. We are the first to use "differences in a certain aspect between two players of same role on two teams" as predictors. Players of the same role play against each other at the beginning of each game, so analyzing differencese in different roles individually is more helpful than simply analyzing differences in two teams.

Finally, we look at the full-game statistics based predictions. Nearly all algorithms have a prediction accuracy of almost 100%. This makes sense, as we now make predictions based on all information of a game, which makes prediction much easier. This result confirms that our algorithms can learn from the game and make almost perfect predictions if all information is given. Note that logistic regression has a very low accuracy of 82.4%, which may be due to that the logistic model itself does not have an complicated structure to fit the data.

We also explore the relationship between the amount of training data we feed into the classifiers and the resulting test accuracy. Overall, KNN does the best job in predicting match outcome, so we focus on KNN classifier when investigating on this question. Looking at Figure 9, which is the early-game model result of KNN classifier, test accuracy increases as the amount of train data increases, but it is increasing more and more slowly. Based on the graph, it seems that the model has not achieved its maximum ability of prediciton accuracy, and increasing amount of train data a bit more (possibly to 4000 observations) would help the model perform even better. We would like to collect more data and test the limit of the KNN model, and get the highest prediction accuracy possible in future. On the other hand, Figure 10, which shows the trend of test accuracy of a the full-game KNN classifier, demonstrates a much more converged result. Here, increasing amount of training data are not likely to icnrease the performance anymore. The plot for pre-match looks a little wierd, as it increases a lot at first and then gradually decreases as the training set size becomes larger. We think this may show the effect of "law of large numbers". As the number of experiments increases, the actual ratio of outcomes will converge to the theoretical ratio of outcomes, which, in our cases, is around 50% for pre-match predictions.

Figure 9: How Test Accuracy Varies with Tranining Data Size(Early Game)

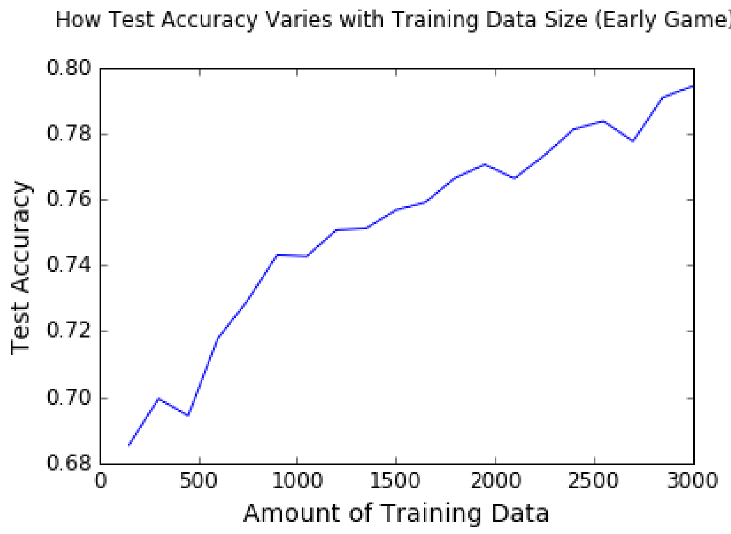


Figure 10: How Test Accuracy Varies with Tranining Data Size(Full Match)

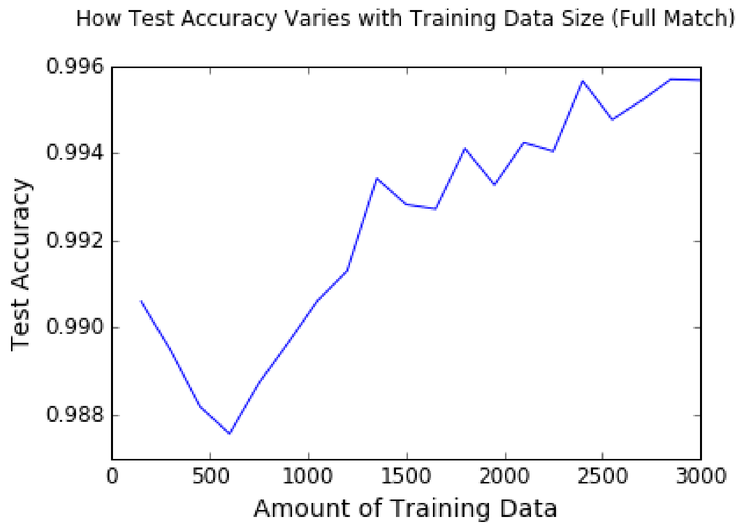
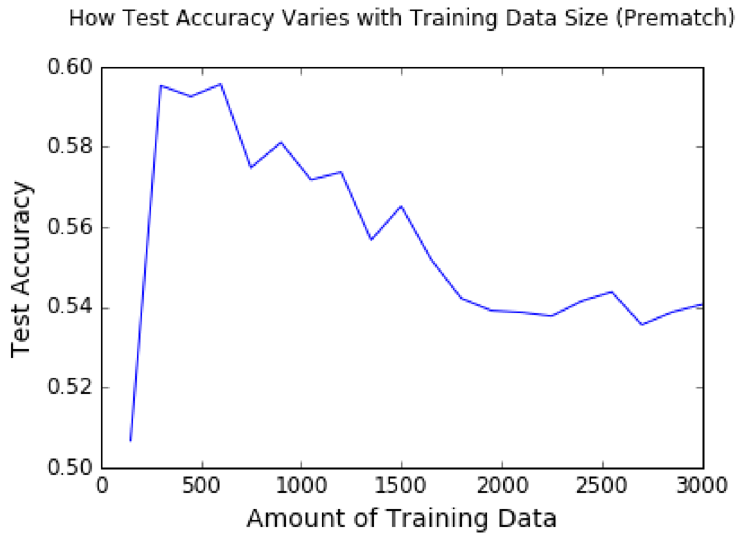


Figure 11: How Test Accuracy Varies with Tranining Data Size(Pre Match)



There are something else we would like to explore in future based on this project. We are still interested in how much pre-match based prediction accuracy can be improved by adding players' mastery information. As mentioned above, the mastery information retrieval is very time-consuming and we cannot finish collecting data before this project ends. We will collect the mastery data for matches afterwards, and re-run our algorithms to obtain the new prediction accuracy. In addition, we are interested in how prediction accuracy changes across different regions (NA, EU, KR, etc.). Players in some regions, such as Korean, are known for strategic group-play, while players in other regions prefer solo-play. These behavior differences may change early-game based prediction accuracy. If more data can be collected across regions, it will be possible to evaluate the prediction accuracies across regions.

## References

1. Cohen, J., P. Cohen, S.G. West, L.S. Aiken. 2003. Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences. London: Lawrence Erlbaum.

2. Friedman, Jerome H. "Greedy Function Approximation: A Gradient Boosting Machine." The Annals of Statistics, vol. 29, no. 5, 2001, pp. 1189–1232. JSTOR, JSTOR, www.jstor.org/stable/2699986.Shai

3. Sklearn.linear_model.LogisticRegression¶." Sklearn.linear_model.LogisticRegression - Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/modules/ generated/sklearn.linear_model.LogisticRegression.html.

4. "Sklearn.svm.SVC¶." Sklearn.svm.SVC - Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

5. Shalev-Shwartz, Shai Ben-David. 2014. Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press.

6. Cabrera, Matias S. Predicting the winner of a League of Legend match at the 10-minutemark. https://www.dropbox.com/s/hb9t1nfc0uhb9ge/Predicting%20the%20Winner%20of%20a%20League%20of%20Legends%20Match.pdf?dl=0

7. Lin, Lucas. League of Legend outcome prediction. http://cs229.stanford.edu/proj2016/report/Lin-LeagueOfLegendsMatchOutcomePredictio n-report.pdf

8. "Introduction to Boosted Trees¶." Introduction to Boosted Trees - Xgboost 0.71 Documentation, xgboost.readthedocs.io/en/latest/model.html.

9. Chen, Tianqi. What Is the Difference between the R Gbm (Gradient Boosting Machine) and Xgboost (Extreme Gradient Boosting)? www.quora.com/What-is-the-difference-between-the-R-gbm-gradient-boosting-machine-and-xgboost-extreme-gradient-boosting.

10. CS231n Convolutional Neural Networks for Visual Recognition, cs231n.github.io/neural-networks-3/#anneal.

11. “Multilayer Perceptron¶.” Multilayer Perceptron - DeepLearning 0.1 Documentation, deeplearning.net/tutorial/mlp.html.

**We affirm that we adhered to the Duke Honor Code.**