Eric Born

24 August 2019

# League of Legends

League of Legends (LoL for short), is a multiplayer online battle arena video game where two teams of five players each battle to destroy the opposite teams base. For this project I chose to use machine learning to predict which team will win the match.

## Dataset Features:

At a quick glance the dataset contains 61 features each representing a different aspect of how one of the two teams performed and 51491 rows, each row representing a single match between two teams. The features are outlined below:

Game ID, Creation Time (in Epoch format), Game Duration (in seconds), Season ID,

Winner (1 = team1, 2 = team2),

First Baron, dragon, tower, blood, inhibitor and Rift Herald (1 = team1, 2 = team2),

Champions and summoner spells for each team (Stored as Riot's champion and summoner spell IDs), The number of tower, inhibitor, Baron, dragon and Rift Herald kills each team has,

The 5 bans of each team (Again, champion IDs are used)

The dataset can be found here:

<https://www.kaggle.com/datasnaek/league-of-legends/downloads/league-of-legends.zip/9>

## Data Cleanup:

Before starting any work with the data, I examined and cleaned the data. I chose to remove the gameID, creationTime and seasonId columns as the entire dataset consisted of only data from season 9 and the gameID and creationTime are irrelevant to the study I will be performing. I also decided to rename the winner column to win and changed the values from 1 and 2 to 0 and 1 respectively.

This leaves 57 features plus the win column, which will be used for testing the models accuracy.

## Feature Selection:

Since there are so many columns in the dataset I decided to do some feature selection before building or training any machine learning models to boost their speed and hopefully their accuracy. I chose to use four methods for feature selection, Pearson’s correlation coefficient, ordinary least squares, recursive feature elimination and Lasso (least absolute shrinkage and selection operator).

|  |  |
| --- | --- |
| Pearson's Ten | |
| t1\_baronKills | 0.369472 |
| firstTower | 0.375697 |
| t2\_baronKills | 0.399249 |
| t1\_dragonKills | 0.472483 |
| t2\_dragonKills | 0.497206 |
| firstInhibitor | 0.536437 |
| t1\_inhibitorKills | 0.649405 |
| t2\_inhibitorKills | 0.660452 |
| t1\_towerKills | 0.771541 |
| t2\_towerKills | 0.785813 |

For Pearson’s I measured the correlation between each of the features to the win column and then set a threshold of at least 0.5 correlation, this resulted in five features. I performed the same test with a threshold of 0.35 which resulted in ten features.

|  |  |
| --- | --- |
| Pearson's Five | |
| firstInhibitor | 0.536437 |
| t1\_inhibitorKills | 0.649405 |
| t2\_inhibitorKills | 0.660452 |
| t1\_towerKills | 0.771541 |
| t2\_towerKills | 0.785813 |

To create the third feature set I used the ordinary least squares function from the statsmodel library by fitting the column data with the win column and creating a linear regression model. The p-value of the features were then evaluated and if they were greater than 0.05 they were eliminated. This process left me with 22 features.

|  |  |
| --- | --- |
| OLS | |
| firstBlood | t1\_baronKills |
| firstTower | t1\_dragonKills |
| firstInhibitor | t1\_riftHeraldKills |
| firstBaron | t2\_champ1id |
| firstRiftHerald | t2\_champ2id |
| t1\_champ3id | t2\_champ3id |
| t1\_champ3\_sum1 | t2\_towerKills |
| t1\_champ5id | t2\_inhibitorKills |
| t1\_champ5\_sum1 | t2\_baronKills |
| t1\_towerKills | t2\_dragonKills |
| t1\_inhibitorKills | t2\_riftHeraldKills |

I utilized the feature selection library from sklearn to perform RFE, or recursive feature selection. This library works to recursively consider smaller and smaller sets of features by measuring the features coefficient and pruning the least important features. The method eventually ended up selecting the top 39 features.

|  |  |
| --- | --- |
| RFE | |
| firstBlood | t2\_inhibitorKills |
| t2\_champ3\_sum1 | t1\_champ4\_sum1 |
| firstTower | t1\_champ4\_sum2 |
| firstInhibitor | t1\_champ5id |
| firstBaron | t1\_champ5\_sum1 |
| t2\_champ3\_sum2 | t1\_champ5\_sum2 |
| t2\_champ4\_sum1 | t1\_towerKills |
| firstDragon | t1\_inhibitorKills |
| firstRiftHerald | t2\_riftHeraldKills] |
| t1\_champ1\_sum1 | t1\_baronKills |
| t1\_champ1\_sum2 | t1\_dragonKills |
| t1\_champ2\_sum1 | t1\_riftHeraldKills |
| t1\_champ3\_sum2 | t2\_champ1\_sum1 |
| t1\_champ2\_sum2 | t2\_champ1\_sum2 |
| t1\_champ3\_sum1 | t2\_champ2id |
| t2\_champ4\_sum2 | t2\_champ2\_sum1 |
| t2\_champ5\_sum1 | t2\_champ2\_sum2 |
| t2\_champ5\_sum2 | t2\_towerKills |
| t1\_champ3id | t2\_baronKills |
| t2\_dragonKills |  |

The final feature selection method used was lasso cv, least absolute shrinkage and selection operator with cross validation. This was also implemented from the SKlearn library. The method resulted in 32 features selected which are listed below:

|  |  |
| --- | --- |
| Lasso | |
| gameDuration | t2\_champ1\_sum1 |
| firstTower | t2\_champ2id |
| firstRiftHerald | t2\_champ2\_sum2 |
| t1\_champ1id | t2\_champ3id |
| t1\_champ2id | t2\_champ3\_sum2 |
| t1\_champ3id | t2\_champ4id |
| t1\_champ3\_sum1 | t2\_champ5id |
| t1\_champ4id | t2\_towerKills |
| t1\_champ5id | t2\_baronKills |
| t1\_champ5\_sum1 | t2\_ban1 |
| t1\_towerKills | t2\_ban2 |
| t1\_baronKills | t2\_ban3 |
| t1\_dragonKills | t2\_ban4 |
| t1\_ban1 | t2\_ban5 |
| t1\_ban2 | t2\_champ1id |
| t1\_ban3 | t1\_ban4 |

## Scaled and Non-Scaled Datasets:

After establishing the features identified by these five methods, I needed to create both scaled and non-scaled versions of each for use with the various algorithms. The scaled versions were created with the standard scaler from the sklearn preprocessing library. This library transforms the dataframe into an array and then scales the data to center by removing the mean value of each feature, then dividing non-constant features by their standard deviation. This step is also needed for any algorithm that operates based upon distance, such as KNN, as large variations in values translate to large distances between neighbors and make it difficult for the algorithms to learn from and predict on.

Both datasets were split into training and test sets using 66.6% for the training and 33.3 for testing.

## Algorithms:

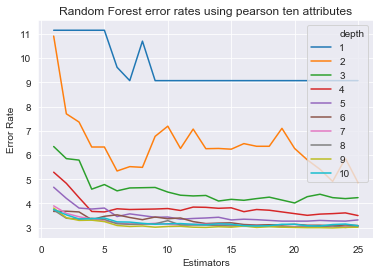
Finally, after all of the prep and setup, it’s time to train and test some algorithms!

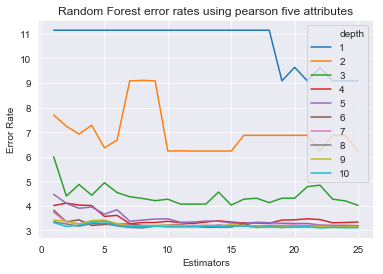
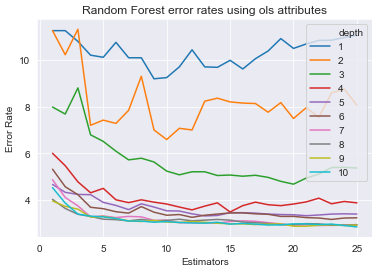
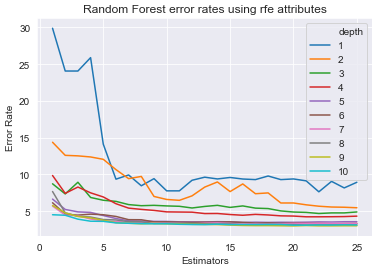
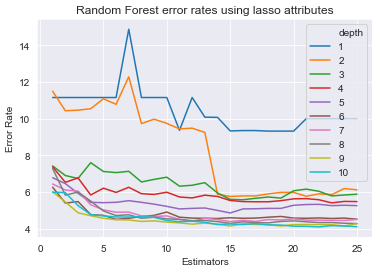
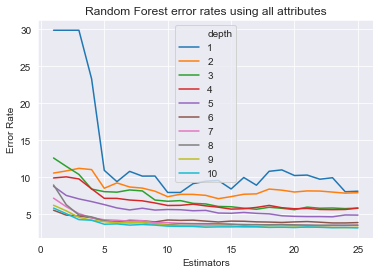
I chose to implement six different algorithms, three using scaled data sets and three using non-scaled. For the scaled group I implemented KNN, SVM and Logistic Regression and the non-scaled on Decision tree, Naïve Bayes and Random Forest.

With SVM I wanted to compare using three different kernels, linear, Gaussian and polynomial and logistic regression using three solvers, liblinear, sag and newton-cg to see if there was any difference in speed or accuracy. With KNN I tested K at a value between 3 to 25 in increments of 2 and random forest with a range of 1 to 26 estimators and 1 to 10 depth.

Each of these was tested on the five datasets previous created with an additional test for Random Forest on a fully featured dataset, just to see how it would perform. This brought the total implementations to 51.

## Random Forest

The graphs shown below relfect the changes in the error rate when utilizing differet numbers of estimators and depth settings with the Random Forest algorithm. Each one is built from the datasets previous described.

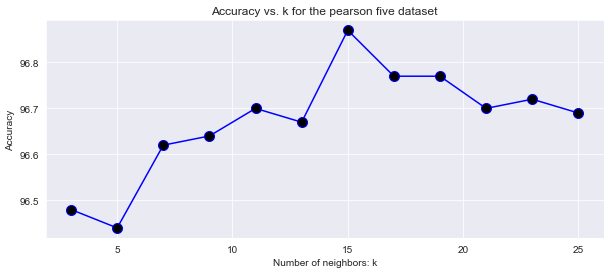


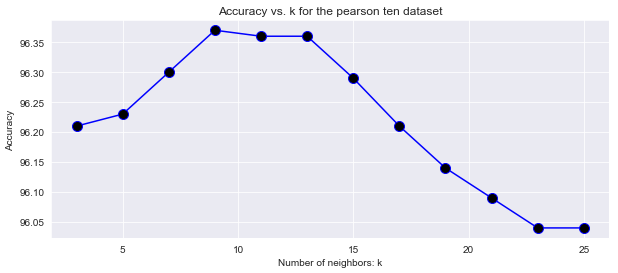
Each performed similarly, but interestingly enough all of them had a different optimum.

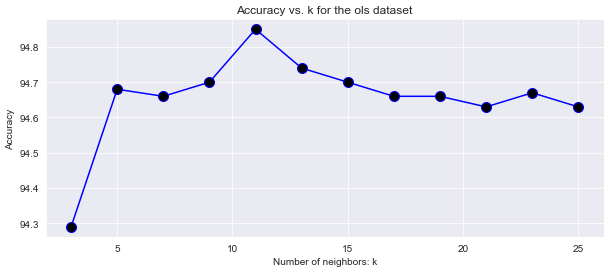
|  |  |  |
| --- | --- | --- |
| Random Forest | 8 trees, 8 depth | Pearson Five |
| Random Forest | 23 trees, 9 depth | Pearson Ten |
| Random Forest | 25 trees, 10 depth | OLS |
| Random Forest | 20 trees, 9 depth | RFE |
| Random Forest | 22 trees, 10 depth | Lasso |
| Random Forest | 25 trees, 10 depth | Full |

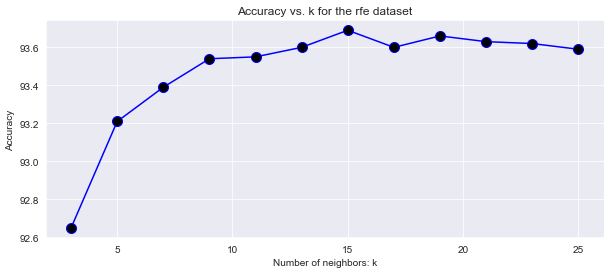
## KNN:

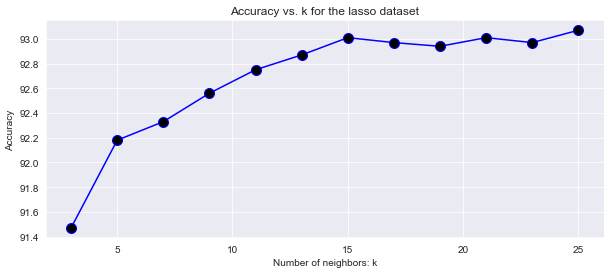
Just as with Random forest, implementing various values for K resulted in a variety of different accuracies depending on the dataset used. For the Pearson’s top 5, a K value of 15 was optimal, 9 for Pearson’s 10, 11 for the OLS dataset, 15 for RFE and 25 for Lasso.











## Durations:

I gathered timings for each of the major steps in the process. As we can see from below, the slowest to build was KNN at almost two minutes and the fastest was Naïve Bayes at less than a second. The three non-scaled algorithms took only four seconds total to train and test, while the scaled took over four minutes, that’s 65 times slower! The following numbers are displayed in seconds:

Data clean up duration: 1

Features selection duration: 5

Dataframe build duration: 0

Decision tree duration: 1

Naive Bayes duration: 0

Random forest duration: 3

Non-scaled duration: 4 (total duration of all non-scaled algorithms)

KNN duration: 112

SVM linear duration: 53

SVM Gaussian duration: 38

SVM poly duration: 54

Logistic Regression liblinear duration: 1

Logistic Regression sag duration: 2

Logistic Regression newton-cg duration: 0

Scaled time 260 (total duration of all scaled algorithms)

Global duration: 270

## Accuracy:

Below are the highest accuracy scores for each algorithm across the five datasets. Random forest and KNN outperformed all others achieving an accuracy of 99.97%, with the second highest going to SVM using the Gaussian kernel at 96.93%. While the two most accurate algorithms both achieved the same score, the time required to train and test KNN was close to a second, which doesn’t seem like very much time, but considering Random forest finished the same task in 0.06 seconds, KNN was over 15 times slower.

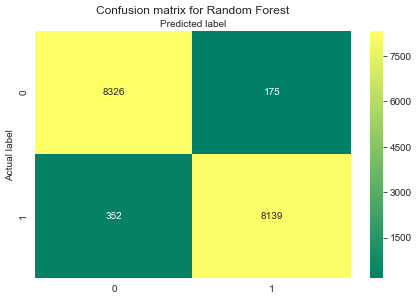
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Classifier** | **Details** | **Features** | **Accuracy** | **Time** |
| Decision Tree | Entropy | Pearson Five | 96.69 | 0.014959 |
| Naive Bayes | Gaussian | Pearson Ten | 94.38 | 0.012941 |
| Random Forest | 8 trees, 8 depth | Pearson Five | 99.97 | 0.063829 |
| KNN | K - 15 | Pearson Five | 99.97 | 0.927157 |
| SVM | Linear | Pearson Ten | 96.06 | 2.849993 |
| SVM | Gaussian | OLS | 96.93 | 6.205704 |
| SVM | Poly | Pearson Ten | 96.64 | 3.293677 |
| Logistic Regression | Liblinear | Lasso | 96.02 | 0.201585 |
| Logistic Regression | Sag | RFE | 96.03 | 0.476487 |
| Logistic Regression | Newton-cg | RFE | 96.03 | 0.249333 |

## Speed:

While all of the algorithms performed above 90% accuracy, some were much slower than others. When utilizing the RFE and Lasso datasets, which have the two highest number of features, the KNN algorithm time jumped from only 0.927 seconds on the Pearson’s five to 49 seconds on the RFE and 41 seconds on the Lasso. SVM had similar issues with large increases in time across all three of its kernels.

## Conclusion:

In conclusion if I was to continue to evaluate data from League of Legends I would chose to use the top five features chosen by Pearson’s correlation along with the Random forest algorithm set to 8 trees and a depth of 8. This combination was able to achieve 99.97% overall accuracy, a 95.85% true positive rate and a 97.94% true negative rate in just 0.06 seconds.



## Accuracy and Durations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Details** | **Features** | **Accuracy** | **Time** |
| Decision Tree | Entropy | Pearson Five | 96.69 | 0.014959 |
| Decision Tree | Entropy | Pearson Ten | 96.23 | 0.026928 |
| Decision Tree | Entropy | OLS | 95.85 | 0.105721 |
| Decision Tree | Entropy | RFE | 95.88 | 0.138358 |
| Decision Tree | Entropy | Lasso | 94.69 | 0.221848 |
| Naive Bayes | Gaussian | Pearson Five | 93.92 | 0.006981 |
| Naive Bayes | Gaussian | Pearson Ten | 94.38 | 0.012941 |
| Naive Bayes | Gaussian | OLS | 94.03 | 0.027930 |
| Naive Bayes | Gaussian | RFE | 93.81 | 0.054855 |
| Naive Bayes | Gaussian | Lasso | 94.5 | 0.045877 |
| Random Forest | 8 trees, 8 depth | Pearson Five | 99.97 | 0.063829 |
| Random Forest | 23 trees, 9 depth | Pearson Ten | 99.97 | 0.198542 |
| Random Forest | 25 trees, 10 depth | OLS | 99.97 | 0.398216 |
| Random Forest | 20 trees, 9 depth | RFE | 99.97 | 0.381977 |
| Random Forest | 22 trees, 10 depth | Lasso | 99.96 | 0.763644 |
| Random Forest | 25 trees, 10 depth | Full | 99.97 | 0.855404 |
| KNN | K - 15 | Pearson Five | 99.97 | 0.927157 |
| KNN | K - 9 | Pearson Ten | 99.96 | 1.691211 |
| KNN | K - 11 | OLS | 99.95 | 9.204163 |
| KNN | K - 15 | RFE | 99.94 | 49.233297 |
| KNN | K - 25 | Lasso | 93.08 | 41.710265 |
| SVM | Linear | Pearson Five | 95.85 | 2.099777 |
| SVM | Linear | Pearson Ten | 96.06 | 2.849993 |
| SVM | Linear | OLS | 95.99 | 6.325226 |
| SVM | Linear | RFE | 96.05 | 22.123563 |
| SVM | Linear | Lasso | 95.99 | 16.437155 |
| SVM | Gaussian | Pearson Five | 96.87 | 2.624671 |
| SVM | Gaussian | Pearson Ten | 96.79 | 3.524398 |
| SVM | Gaussian | OLS | 96.93 | 6.205704 |
| SVM | Gaussian | RFE | 96.72 | 11.894439 |
| SVM | Gaussian | Lasso | 96.07 | 11.371954 |
| SVM | Poly | Pearson Five | 95.92 | 2.805235 |
| SVM | Poly | Pearson Ten | 96.64 | 3.293677 |
| SVM | Poly | OLS | 96.6 | 5.655072 |
| SVM | Poly | RFE | 96.06 | 19.053048 |
| SVM | Poly | Lasso | 95.57 | 17.492169 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Details** | **Features** | **Accuracy** | **Time** |
| Logistic Regression | Liblinear | Pearson Five | 95.93 | 0.028921 |
| Logistic Regression | Liblinear | Pearson Ten | 95.94 | 0.058818 |
| Logistic Regression | Liblinear | OLS | 95.94 | 0.218912 |
| Logistic Regression | Liblinear | RFE | 96.02 | 0.30934 |
| Logistic Regression | Liblinear | Lasso | 96.02 | 0.201585 |
| Logistic Regression | Sag | Pearson Five | 95.93 | 0.140656 |
| Logistic Regression | Sag | Pearson Ten | 95.94 | 0.233349 |
| Logistic Regression | Sag | OLS | 95.93 | 0.261301 |
| Logistic Regression | Sag | RFE | 96.03 | 0.476487 |
| Logistic Regression | Sag | Lasso | 96.01 | 0.425001 |
| Logistic Regression | Newton-cg | Pearson Five | 95.93 | 0.083143 |
| Logistic Regression | Newton-cg | Pearson Ten | 95.94 | 0.102719 |
| Logistic Regression | Newton-cg | OLS | 95.93 | 0.148603 |
| Logistic Regression | Newton-cg | RFE | 96.03 | 0.249333 |
| Logistic Regression | Newton-cg | Lasso | 96.02 | 0.134640 |

Green rows represent top accuracy from each algorithm and detail combination.

yellow cells indicate a run duration over 1 second.

red cells indicate run duration over 10 seconds.