League of Legends Match Win Classifier

Jonathan Ong

Abstract

League of Legends is internationally played game with a huge player base and large betting market. As result of this, it is of great interest to be able to predict the outcome of League of Legends matches before they are over as well as understand the factors that contribute to the outcome of a match. This project employs various binary classification techniques such as logistic regression, neural networks, and random forests to predict the outcome of a match given match data from 10 minutes into the game. Each model is trained on a training set, tuned on a development set, and finally evaluated with accuracy on a test set. Most of the methods manage to improve slightly on a baseline heuristic method, with random forests and neural networks performing the best on the test set.

1. Introduction

1.1. Overview of League of Legends

League of Legends of a team-based strategy game where two teams of 5 players (a red team and a blue team), each playing a different champion, work to destroy the other teams base (lea, 2023). This is achieved by destroying the enemy towers that defend the enemy's base and finally destroying the nexus at the center of the base. The players on a team of 5 usually take 1 of 5 positions (top, jungle, mid, bot, and support). Champions grow stronger for combat by earning gold through killing other players and minions and spending on this gold on items to gain combat stats. Additionally, champions gain experience that causes them to level up, allowing them to level up abilities and gain stats. A typical game of League of Legends lasts about 30 minutes with players being able to surrender as early as 15 minutes into the game (Mahmood, 2023).

1.2. Task Definition and Motivation

The goal of this project is to predict the outcome of a League of Legends match (which team wins) given a snapshot of the game data 10 minutes into the match. Given data about a match at 10 minutes (such as the champions on each team, the amount of gold on each team, champion stats) the model outputs the winning team. I chose this project because I really enjoy playing League of Legends and think it would be fun to apply machine learning techniques to a topic I enjoy. Additionally, this could have applications in providing insights into what metrics have the largest influence in game outcomes and in betting markets on league matches. It turns out that a baseline heuristic using the total gold of each team is extremely effective in determining the outcome of a match. Machine learning methods are only able to slightly improve on the baseline, with random forests models and neural networks performing the best.

2. Related Work

Past work in this field has been successful and generally falls into two categories.

2.1. Personal Player Related Statistics Approach

Past work has tried to use information about the players in the game to predict its outcome before it even starts after all players have selected the champion they will play. For example, Huang, Kim, and Leung found that each player's winrate on their chosen champion allowed for extremely accurate predictions decision trees and bayesian neural networks (Huang et al., 2015). They were even able to achieve an accuracy of 92.8% using the selected champion win-rates of players on each team. On the other hand, Hall used neural networks with data on each of the player's past ranked games (not including win rate) (2017). Hall was able to achieve an accuracy of 84.4% after some feature engineering. My project fundamentally differs from this approach though in that it is trying to make predictions based off the current state of the League of Legends match instead of the past history of the players.

2.2. Game Related Data Approach

Much other work aims to predict the winner of a League of Legends match with a snapshot of the state of the game at 10 minutes which is the approach that I take in this project. For example, Lee was able to predict the outcome of a match using logistic regression with an accuracy of 72.75% and also achieved an accuracy of 71.7% using a XGBoost Random Forest model (2021). Lee's results indicated that certain features (such as early kills) had an extremely high impact of the outcome of a match. Carroll was also able to achieve similar results training a random forest and XGBoost on game data from 10 minutes and discovered that a team's lead in gold was by far the largest factor in predicting a win (followed by lead in experience) (2020). This supports the use of the team with the lead in gold as a heuristic for preliminary predictions. A common challenge in related work is that League of Legends game data has features that are unnecessary or highly correlated. This is something I hope to address in my data preprocessing by removing unnecessary features. Additionally, none of the previous game data approaches to this task use neural networks and I am hope to use neural networks to achieve similar results.

3. Dataset and Evaluation

3.1. Plan to Produce a Dataset

I have produced my own dataset using the Riot API (Riot is the company that produces League of Legends). By making requests to the API for match statistics (such as the champions involved and outcome of the match) and for the state of each player at 10 minutes (stats, total gold, total experience, etc.), I have been able to extract 512 features that describe the state of a match at 10 minutes from 19242 ranked matches. The two teams involved in a match are labeled as team100 and team200 from the Riot API. The classification task will be to predict if team100 wins. Out of the 19242 matches, team100 wins on 9695 / 19242 = 50.2% of the matches. Since this is a rather sizable dataset, I do not need to sample matches from times other than 10 minutes. Additionally, this dataset is large enough to be split into a train, development, and test set. The dataset was randomized and split into a train, development, and test set with a 70:10:20 ratio each of size 13470, 1924, and 3848 respectively.

3.2. Plan for Model Evaluation

Using the test split, I will test the trained model using accuracy since this is a classification task. This is an appropriate metric to use because there is not a large class imbalance (team100 wins about half of the time).

3.3. Data Preprocessing

To preprocess the data, the label (team100Win) was separated out. Additionally, unimportant features such as matchID, teamID, and championName were removed because they aren't directly related to gameplay. The teamID feature was removed because the player data is always presented in the same order within the input. So, the position of player data within the input already encodes which team the player is on. Additionally, both the championName and championID features were removed. Originally, in the midterm report, I one-hot encoded the championID feature into 165 classes. But, this led to strong L1 regularization being the best performing logistic regression model which pushed the weights for a lot of features to zero (including all of the one-hot encoded features). So, the one-hot encoded features are not useful for the model and don't needed to be included in the features. The mean and standard deviation of each feature was calculated and data is normalized to have a mean of 0 and standard deviation of 1. After these preprocessing steps, the input size now has 470 features (down from the 2130 features used in the midterm report).

4. Methods

4.1. Baseline

The baseline model predicts the winner of a match based on the heuristic that the team with the most gold at 10 minutes will win. By identifying and hard-coding the locations in the feature vector that contain the gold for each player of a team, the baseline predicts that team100 wins if team100 has more total gold than team200. The input data for the baseline method does not need to be normalized because it is simply summing certain features. Total gold is a reasonable metric for predicting wins because gold is used to purchase items which directly influence combat stats. It would stand to reason that the team with more gold would have an advantage in combat against other team, leading to the gold lead increasing until the team in the lead wins. Additionally, previous work shows that a gold lead is a powerful predictor for the outcome of a League of Legends match.

4.2. Logistic Regression

To implement logistic regression, I use the sklearn.LogisticRegression class. After preprocessing inputs, I use this class to fit a Logistic Regression model to the training data. This model takes a takes the dot product between a weight vector and the input, adds a bias, then takes sigmoid of this value to make a prediction (True or False) whether Team100 wins or not. The hyperparameters of this model include type of regularization (L1 vs L2) and strength of regularization. I evaluate many different versions of logistic regression including no regularization and differing strengths of L2 or L1 regularization (chosen from [0.01, 0.1, 0.5, 1, 1.5, 2]). I evaluate each model on the development set and keep the model that performs best on this.

4.3. Support Vector Machines (SVMs)

To implement support vector machines, I used the sklearn.svm.SVC class. To map an input to a prediction, a SVM takes the kernel of an input with all of the training examples (which corresponds to a dot product in a different feature space) and sums them. Then, it takes sigmoid of this value to make a prediction (True of False) whether Team100 wins or not. The hyperparameters of this model include the type of kernel, polynomial or radial basis function (RBF). When working with a polynomial kernel, the degree of the kernel (chosen from [1, 2, 3, 4, 5]) is also a hyperparameter. When working with a RBF kernel, the strength of L2 regularization (chosen from [0.01, 0.1, 0.5, 1, 1.5, 2]) is also a hyperparameter. I evaluate the performance of each model with its varying hyperparameters on the development set and keep the model that performs best on this.

4.4. Neural Network

To implement logistic regression, I created a VariableLayerMLP class (a neural network with a variable number of hidden layers) using PyTorch. When a tensor of features is input into the model, a linear hidden layer is applied, the ReLU activation function is applied, and a drop out layer is applied. This process is repeated a set number of times before it is then passed through a final linear output layer. Sigmoid of the output layer then becomes the output of the model. If the output is greater than or equal to 0.5, then the model predicts that team100 wins. If it is less than 0.5, then the model predicts that team100 loses. Since this is a binary classification task, the loss function for the model was binary cross entropy.

The hyperparameters of this model include the type of optimizer (stochastic gradient descent or Adam), learning rate (or learning rate scheduler), number of epochs, batch size, number of hidden layers, size of each hidden layer, weight decay, and drop out probability. The optimizer is the algorithm used to update the parameters of the neural network. The number of epochs controls the maximum number of passes through the entire training data during training. The batch size controls how many examples the model sees before it updates itself. The number and size of the hidden layers controls the operations that the inputs are passed through to be mapped to the output. Weight decay is a version of L2 regularization that causes parameters to decay by a certain percentage each update. Drop out probability is the probability that a neurons are ignored in a specific forward pass of the model. While in the midterm report I only evaluated one version of the neural network with a set of hyperparameters. Now, I conduct experiments on different number of hidden layers, size of hidden layers, drop out probability, batch size, weight decay, and learning rate.

4.5. Decision Trees

To implement a decision tree, I used the sklearn.treeDecisionTreeClassifier class. At its core, a decision tree is a series of if-else statements. To map an input to an output, an input is put through a series of conditions based on single features. Once an input has gone through a certain number of conditions (nodes in the tree) and reaches a leaf node with no more conditions to be compared, the input is classified with an output based on the leaf node it landed at. A trained decision tree has all of its conditionals and output labels at each leaf node defined so an input simply needs to traverse the tree by checking itself against the conditions and assign itself the output of the leaf node it lands at.

Hyperparameters that control the training of a decision tree include its max depth, splitting criterion, maximum features to consider, and minimum impurity decrease. During training, a decision tree will create nodes that split the training data based on a condition involving a single feature. Instead of a traditional loss function, a decision tree uses an impurity function (criterion) which is a measure of how bad a leaf node is at classifying training examples. Some popular are criterion are gini, entropy, and log_ loss. Left unchecked, a decision tree can overfit very easily by continuously creating new conditions until it reaches 0 impurity, perfectly classifying all of the training data. To avoid this, the max depth hyperparameter stops the decision tree from becoming too deep by limiting the number of conditions it can learn to split the data. Additionally, the minimum impurity decrease can prevent overfitting to the training data by only allow the decision tree to create conditions that reduce the impurity (as defined by the criterion) by at least a certain amount. Limiting the number of features that a tree is allowed to split can also reduce overfitting.

4.6. Random Forests

A random forest is many decision trees working together to make predictions. To map an input to an output, an input is passed to many decision trees. The outputs of all of these decision trees are used in a majority vote system where the most common output becomes the output of the random forest.

A random forest uses many of the same hyperparameters as decision trees such as max depth, splitting criterion, max features, and minimum impurity decrease which all control the base decision tree that makes up the random forest. Additionally, to ensure that different decision trees are learned, bootstrap sampling is used. The dataset for each decision tree is generated by randomly sampling from the training dataset with replacement to produce a new (slightly different) dataset of the same size.

4.7. Improvements from Midterm Report

The final methods presented in the final report use a further cleaned version of the data as inputs. This should make it easier to train models and improve performance over the

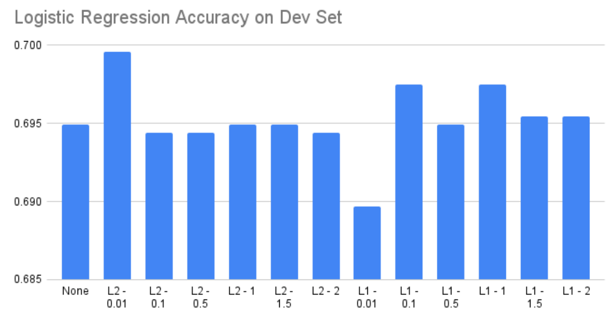

Figure 1. Accuracy of Logistic Regression with Different Regularization

midterm report. Additionally, the final methods used such as SVMs and random forests are much more powerful than what was used in the midterm report. In particular, given the success of the heuristic which only uses a few features, random forests which only consider single features at a time should perform very well.

5. Experiments

5.1. Baseline

The baseline model performs extremely well, achieving an accuracy of 0.703. It predicts that 1853 / 3848 = 0.482 of the matches are a win for team100. This is slightly off from the team100 winning 0.502 matches on average.

5.2. Logistic Regression

All of the logistic regression methods performed very similarly as can be seen in Figure 1. L2 regularization with a strength of 0.01 had the highest accuracy on the development set though and was chosen to represent the logistic regression model approach. L2 regularization of 0.01 had an accuracy of 0.706 on the test set which is an improvement over the baseline.

5.3. Support Vector Machines (SVMs)

The results experiments with different kernels and regularizations for SVMs can be seen in Figure 2. A polynomial kernel with degree 3 had the highest accuracy of the development set and was chosen to represent the SVM approach. A degree 3 polynomial kernel SVM had an accuracy of 0.707 on the test set which is slight improvement over logistic regression.

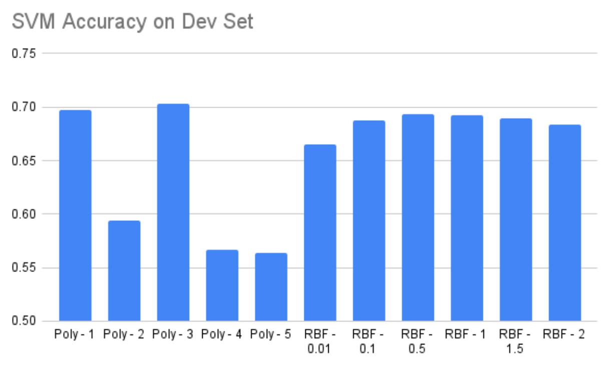

Figure 2. Accuracy of SVMs with Different Kernels and Regularization

5.4. Neural Network

After finding a bug in the implementation of the neural network model, I trained a model with the same hyperparameters as the midterm report[1] but with a different optimizer. This neural network had a highest development set accuracy of 0.7074 on epoch 67 which is pretty good. But given the success of the heuristic baseline model which only takes into account a handful of features, I conducted further experiments to see if similar performance could be achieved with much smaller neural networks.

In testing different hidden layer sizes[2], I discovered that a hidden layer size of 2 and 4 achieved comparable development set performance to the first model with a hidden layer size of 200 (see Figure 3). This led to further experiments with model architecture shown in table 1. A neural network with 6 hidden layers of size 4 achieved the best development accuracy.

This led to more experiments to tune drop out probability, then learning rate, then weight decay, then batch size. I tuned each of these individually and in order, keeping all other hyperparameters that I wasn't tuning constant to avoid a huge grid search through all combinations of these hyperparameters. The results of these experiments are summarized in Table 2, Table 3, Table 4, and Figure 4.

Despite the slightly weaker performance of drop out probability 0.15, I decided to choose this to be the drop out probability moving forward because I was worried about

[1]This was a neural network using 2 hidden layers of 200 neurons. Dropout probability was 0.1 and weight decay was 0.001. The learning rate was 0.01 and I used Adam as the optimizer instead of stochastic gradient descent. The model was trained for 100 epochs with a batch size of 32.

[2]All hidden layer sizes were trained with 200 epochs except hidden layer size 2 was trained with 400 epochs because it was still improving at the 200th epoch.

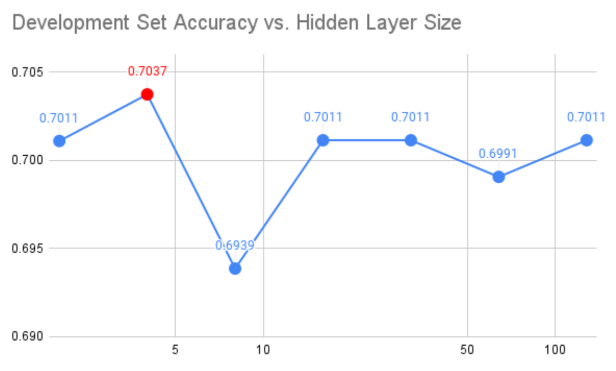

Figure 3. Accuracy of 2 layer neural network models on the development sets with varying hidden layer sizes

Table 1. Model Architecture Experiment

Num Hidden Layers	Hidden Size	Dev Accuracy
2	2	0.6975
2	4	0.7037
4	2	0.4844
4	4	0.7027
6	2	0.5156
6	***4***	***0.7048***
8	2	0.4844
8	4	0.5156
10	2	0.5156
10	4	0.5156

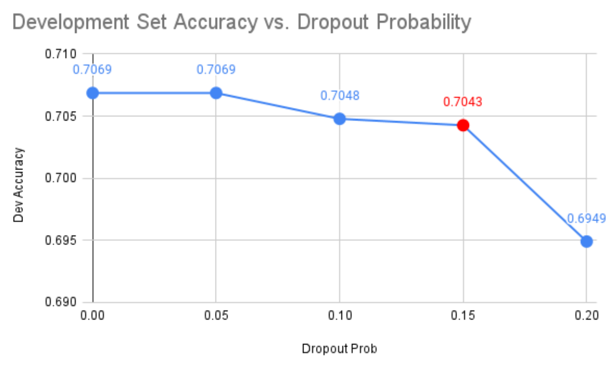

Figure 4. Accuracy of a neural network with 6 layers of 4 neurons with varying dropout probabilities

Table 2. Learning Rate Experiment

Learning Rate	Dev Accuracy
0.01	0.5156
0.001	0.6960
0.0001	0.7027
1.00E-05	0.7032
1.00E-06	0.4844
1.00E-05	0.7032
2.00E-05	***0.7053***
3.00E-05	***0.7053***
4.00E-05	0.7048
5.00E-05	0.7022

Table 3. Weight Decay Experiment

Weight Decay	Dev Accuracy
0.1	0.5156
0.01	0.5156
0.003	0.5156
0.005	0.5156
0.008	0.5156
0.001	***0.7053***
0.0001	0.7017

overfitting and it didn't have a large effect on the development accuracy when compared with smaller drop out probabilities. To break the tie between 2.00E-05 and 3.00E-05 learning rate, I chose 2.00E-05 for more stable training.

The final hyperparameters for the neural network has the following hyperparameters:

- 6 hidden layers of size 4 each
- drop out = 0.15
- learning rate = 2.00E-05
- weight decay = 0.001
- batch size = 32

Table 4. Batch Size Experiment

Batch Size	Dev Acccuracy
16	0.7053
32	***0.7084***
64	0.7074
128	0.6939

Table 5. Decision Tree Max Depth Experiment

Max Depth	Dev Acccuracy	Train Accuracy
2	0.5894	0.6066
4	0.6201	0.6530
5	***0.6279***	***0.6694***
6	0.6159	0.6907
7	0.6081	0.7205
8	0.6034	0.7601
16	0.5894	0.9812
32	0.5697	1
64	0.5697	1
128	0.5697	1
256	0.5697	1

Table 6. Decision Tree Hyperparameters Experiment

Criterion	Max Features	Dev Acc.	Train Acc.
gini	sqrt	0.6066	0.6347
gini	log2	0.5858	0.6226
gini	***None***	***0.6279***	***0.6694***
entropy	sqrt	0.6045	0.6301
entropy	log2	0.5847	0.6210
entropy	None	0.6273	0.6693
log_loss	sqrt	0.6048	0.6301
log_loss	log2	0.5847	0.6210
log_loss	None	0.6273	0.6693

This model achieved a development set accuracy of 0.708 which is better than the original neural network model with much fewer parameters. The accuracy of the model on the test set is 0.7102 which which is an improvement over the SVM.

Table 7. Decision Tree Min Impurity Experiment

Min Impurity Decrease	Dev Accuracy	Train Accuracy
0	0.6278586279	0.6694135115
1.00E-06	0.6278586279	0.6694135115
1.00E-04	0.6278586279	0.6694135115
1.00E-03	***0.632016632***	***0.6590200445***
1.00E-02	0.5899168399	0.588047513
1.00E-01	0.5155925156	0.5032665182

5.5. Decision Trees

The success of extremely small neural network models (as well as the great success of a simple baseline heuristic) suggest that there are a few features that are very powerful predictors for the winner of a League of Legends match. So, it is expected that decision tree models which make simple decisions based on single features will have a perform well on this task.

After trying many different values for the max depth (see Table 5), and then running an experiment on different combinations of criterion and max features (see Table 6), a decision tree with a max depth of 5, gini criterion, and no max features performs the best on the development set with a development set accuracy of 0.628. I further improved this by experimenting with different values for the minimum impurity decrease (see Table 7). The best value for minimum decrease was 0.001 with a development accuracy of 0.632. This decision tree achieves an accuracy of 0.632 on the test set which is significantly under the baseline and even the neural network model. Despite this, what the decision tree learns generalizes well because there isn't a drop in accuracy between the development set and the test set.

5.6. Random Forests

Initially, similar experiments to the decision tree experiments were conducted for the random forest to determine its underlying decision tree. A max depth of 8 (see Table 8) achieves the highest development accuracy but seems to be overfitting on the training data, achieving a training accuracy nearly 20% greater than the development accuracy. A max depth of 5 has the second highest development accuracy, but seems to be slightly overfitting on the training data. A max depth of 4 was chosen because it has a very similar development accuracy to the max depth of 5, without overfitting as much on the training data. Both entropy criterion wtih sqrt max features and log_loss criterion with sqrt max features had the highest development accuracy. For future experiments, I arbitrarily decided to use the entropy criterion with sqrt max features. Additionally, a minimum purity decrease of 0.0001 had the highest accuracy on the development set.

Finally I ran experiments on the number of estimators used for the random forest (see Table 9). I found that 200 random estimators had the best development accuracy. So after all the experiments, a random forest model with a max depth of 4, entropy criterion, sqrt max features, minimum impurity decrease 0.0001, and 200 random estimators was chosen to be the representative random forest model. This model achieved a test accuracy of 0.7105 which is the best model discovered, slightly outperforming the neural network.

Table 8. Random Forest Max Depth Experiment

Max Depth	Dev Acccuracy	Train Accuracy
1	0.6856	0.6844
2	0.6861	0.6950
3	0.6913	0.7033
4	***0.6960***	***0.7223***
5	0.6970	0.7417
6	0.6975	0.7721
7	0.6840	0.8100
8	***0.6975***	***0.8602***
10	0.6939	0.9500
12	0.6757	0.9918
14	0.6913	0.9990
16	0.6850	1
32	0.6840	1
64	0.6772	1
128	0.6772	1
256	0.6772	1

Table 9. Random Forest Num Estimators Experiment

Max Depth	Dev Acccuracy	Train Accuracy
1	0.5936	0.6062
5	0.6419	0.6670
10	0.6663	0.6828
20	0.6793	0.7038
50	0.6897	0.7161
100	0.6970	0.7240
150	0.6954	0.7254
175	0.6985	0.7247
200	***0.7027***	***0.7251***
225	0.7006	0.7257
250	0.7011	0.7252
275	0.7027	0.7250
300	0.6980	0.7269
400	0.6975	0.7269
500	0.6991	0.7270
600	0.6928	0.7270

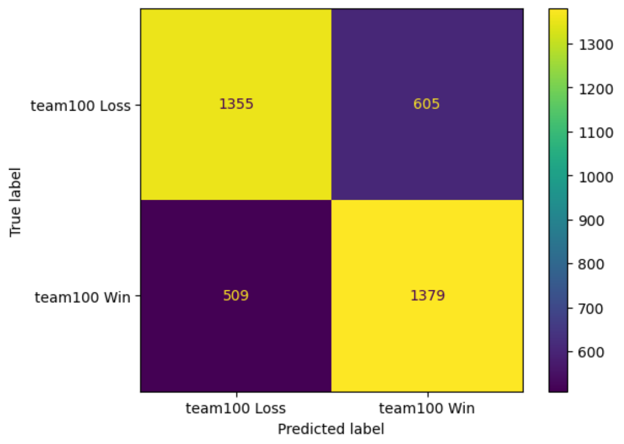

Figure 5. Confusion matrix for the random forest model on the test set

6. Discussion

6.1. Analysis of Performance and Improvement From Midterm Report

For logistic regression, L2 regularization doing better than L1 regularization is an indication that they remaining input features for the model are useful for the model. It still seems like out the few remaining features, a few dominate with as predictors for the output though. This can be seen in the success of very shallow decision trees used in the most successful machine learning method, random forests. Surprisingly, a random forest using just 4 deep decision trees achieves the best performance on the data, indicating that only a few features are needed to make accurate predictions. This explains the poor performance of a normal decision tree because many of the other features would have be noise that made it easy to overfit on the training data, and harder to produce a generalizing model. Random forests took the idea that a few features could be used to make accurate predictions, but addressed the overfitting issues of decision trees by averaging the predictions of many decision trees.

6.2. Error Analysis

Overall the random forest model is very balanced in its predictions as can be seen in a relatively equal rate of false positives to false negatives as can be seen in Figure 5.

Out of the top 10 features used for splits in the estimators of the random forest model, the top 8 were total gold for different players. This indicates that the the random forest model relied heavily on the total gold features to make its predictions.

To test the random forest model's reliance on the total gold feature, I created a new dataset which was the subset of

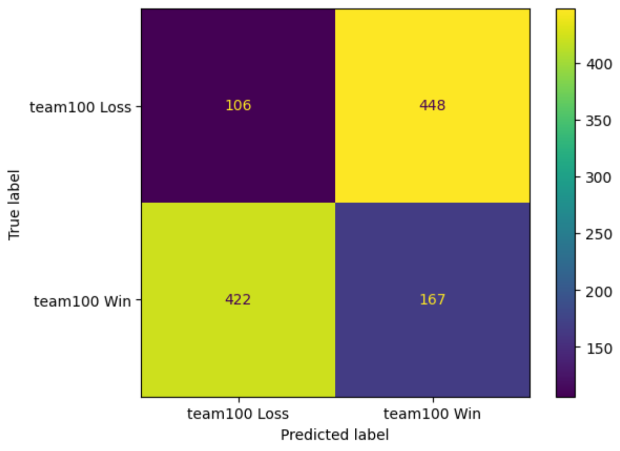

Figure 6. Confusion matrix for the random forest model on a subset of the test set where the total gold heuristic fails

the testing data that the heuristic approach incorrectly classified. This dataset consisted of matches where team100 had more total gold and lost or team100 had less total gold and won. The random forest model still had fairly balanced predictions as can be seem in Figure 6, but has a terrible accuracy of 23.9%. Evidently the model has learned to rely very heavily on the total gold features and performs very poorly when total gold isn't an accurate predictor of the winning team. It could be the case that there is an element of randomness in League of Legends matches and that beyond total gold, there are not many good predictors of the outcome of a match.

7. Conclusion

Overall, I found that total gold is a key factor in determining the outcome of a League of Legends match. In fact, total gold is such a strong predictor of the winner that machine learning models learn to rely on it heavily to make their predictions and achieve accuracies comparable to a simple heuristic based off total gold. One of the biggest (and most surprising) takeaways I have from this project is that sometimes simple heuristics are effective.

I learned a lot about the importance of data preprocessing. Before I preprocessed my data by removing unneccessary features, I wasn't able to train a SVM in reasonable time because the data was too noisy to fit a model to. After I updated my data, I was able to train SVMs and neural networks to achieve respectable accuracies.

Additionally, I have learned a lot about debugging neural networks in Pytorch and the importance of sanity checks such as fitting your model to a small training set. Debugging my neural network model on a small training set that it wouldn't fit enabled me to discover a bug in my implementation.

I also got a sense of how easy it is to overfit on data and learned how important it is to take measures to watch for overfitting such as tracking training loss or accuracy in comparison with development loss. In particular, it amazed me how quickly decision trees overfit, with even a 6 or 7 layer decision overfitting to the training set to the point of hurting development accuracy.

8. Code and Data Submission

The code and dataset for this project are available at this Google Drive folder. To run the code for this project, run the cells in Models.ipynb. Run the "Set Up" and "Data Preprocessing" sections before running any of the other sections. The %cd command in the very first "Set Up" cell may need to be changed to reference location of the data folder in your drive. All of the data used can be found in the data folder. Raw data from experiments can be found in the Final Documentation folder.

References

How to play - league of legends, 2023. URL `https://www.leagueoflegends.com/en-us/how-to-play/`.

Carrol, T. Predicting the winner of league of legends games. *Medium*, 2020. URL `https://medium.com/analytics-vidhya/predicting-the-winner-of-league-of-legends-games-c6fb3513b3d4`.

Hall, K. T. Lol-match-prediction. 2017. URL `https://minihat.github.io/LoL-Match-Prediction/`.

Huang, T., Leung, G., and Kim, D. League of legends win predictor, 2015. URL `https://thomasythuang.github.io/League-Predictor/`.

Lee, J. Using machine learning to understand league of legends. *Towards Data Science*, 2021. URL `https://towardsdatascience.com/the-path-to-a-victorious-league-of-legends-match-40d51a1a089e`.

Mahmood, R. How long is an average match in league of legends 2023. 2023. URL `https://gameriv.com/how-long-is-an-average-match-in-league-of-legends/#average-match-time-for-summoner8217s-rift`.