League Game Prediction: The First Ten Minutes

Jacob Lambert

***Abstract*—We present models to predict if a team wins based on data collected from the first ten minutes for the on-line video game League of Legends. Using data from 900 games played, we create a Decision Tree Classifier, Bagged Decision Tree Classifier, and Random Forest Classifier. These classifiers and their hyper-parameters are trained using two respective subsets of the data, and the classifiers are tested on a third subset of the data. The results show that, while these classifiers perform better than a random or majority selection, strong predictions will require more in-depth features and finer tuning of the hyper-parameters.**

***Keywords*—Decision Tree, Bagging Classifier, Random Forest, League of Legends**

I. Introduction

The industry of ESports, also known as Electronic Sports or competitive video gaming, has rapidly grown over the past decade [1]. Competitions for popular ESports like League of Legends (LoL) [2] can have millions of viewers and cash prizes in the millions of dollars [1].

Methods for accurately predicting game performance based on game data can be very useful for developing game-play strategy and tactics. For example, if a team can determine what behaviors in early game situations most commonly lead to successes, they can develop strategies to target those behaviors.

In section II, we use domain knowledge to determine what data can be transformed into relevant features. In sections III and IV we describe the machine learning methods applied to the generated features, and the motivation for applying each method. We also detail the experimental setup and how the methods were applied to the data and the features. Finally, in section IV we summarize the results of the experimentation and discuss possible improvements and future work.

II. Background

On a high-level, the premise of LoL is to collect resources, and to use those resources to capture objectives. The game consists of two teams of five players, and several different types of resources, some specific to each team and some neutral. The game ends when one team captures the enemy team's central objective. A game normally lasts 40-60 minutes.

Riot Games, the creators of LoL, provide an API [3] for fetching recorded data for LoL matches. The dataset used in this study consists of 9 files, where each file contains full-game data on 100 matches. For each match, we have data on each player's resources, each player's objectives captured, and team resources and objectives. We also have timestamped data of specific events that occur throughout the game, and the locations of players throughout the game. Finally, we have data on which team won the game, and which players belonged to the winning team.

For this study, we need data only from the first 10 minutes of each match. Fortunately, the Riot Games API collects summary data of each players average resources per minute in 10-minute intervals. For example, for each player we have the average gold per minute collected for minutes zero to ten. We use the following zero-to-ten minute summary data:

- average gold per minute
- average creep score (CS) per minute
- average damage taken per minute

Here CS refers to a specific type of resource in the game, closely related to gold acquisition.

Because the data is on a per-player basis, and we aim to make predictions on the team level, we can aggregate the individual data to generate features. For each category of summary data and for each team, we calculate the average, minimum, maximum, and variance over that category.

The resulting list contains 12 features, and two labels.

- Features
  - average, max, min, variance gold per minute
  - average, max, min, Variance CS per minute
  - average, max, min, variance damage taken per minute
- Labels
  - 1 (winning team)
  - 0 (losing team)

We use the above as an initial set of features for creating baseline classifiers.

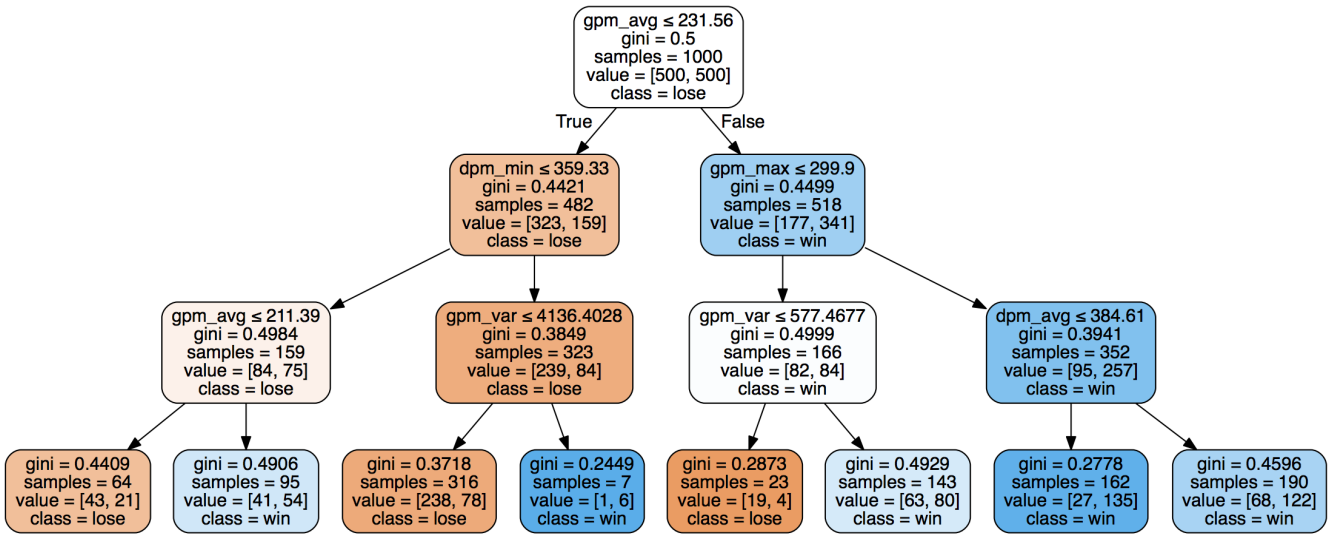

Fig. 1

III. Methods & Experiments

We partition the 1800 data items into training, validation, and testing sets as follows:

- Training: 1000
- Validation: 400
- Testing: 400

We use the training set to initially train each of the three classifiers. We subsequently use the validation set to train the hyper-parameters of each classifier. Finally we attempt to classify the training set.

We include the validation set to avoid over-fitting. Although we could train the hyper-parameters using the testing set, the hyper-parameters would then be dependent not only on already-seen training data but also unseen test data.

A. Decision Tree Classifier

As a baseline, we chose a Decision Tree classifier. The Decision Tree Classifier is well suited to our problem for several reasons.

First, the data set labels are already balanced, with 900 winning team labels and 900 losing team labels. Therefore there is no need to re-balance the data. Also the current feature space is relatively small. Finally, even though the current model only contains continuous data, the data-set has both continuous and categorical data. Modeling with a Decision Tree allows us to easily expand the feature space to include categorical data.

We use sklearn's [4] Decision Tree Classifier. This allows us to tune a wide variety of hyper-parameters when constructing the tree.

One of these hyper-parameters, the depth of the tree, was initially a major source of over-fitting. We therefore chose to train this hyper-parameter using 4-Fold Cross validation. To accomplish this, we partitioned the 1400 testing and validation data items into subsets of 200 items, each labeled 1-7. We then trained a classifier and tested against the validation set, where the validation set was equal to 1 or 2 of the 7 subsets. Restated, we performed training and classifying using the following data partitions:

- validate: 1, 2
  train: 3, 4, 5, 6, 7
- validate: 3, 4
  train: 1, 2, 5, 6, 7
- validate: 5, 6
  train: 1, 2, 3, 4, 7
- validate: 7
  train: 1, 2, 3, 4, 5, 6

Due to the file structure of the data, with 100 matches per file, 4-Fold Cross Validation allowed us to perform cross validation without restructuring the data files. Fig. 2 summarizes the results of the cross validation.

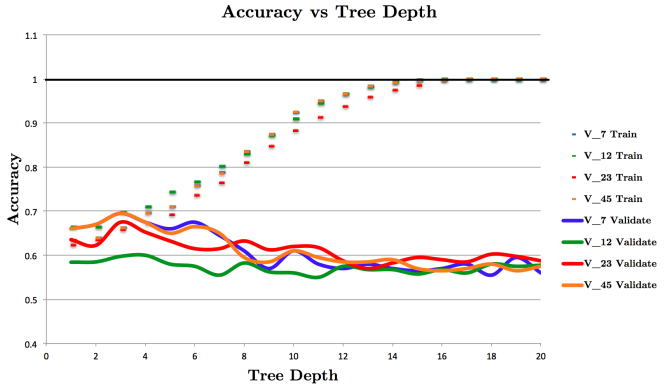

Fig. 2

We can see that, as the depth of the tree increases, the training accuracy tends to one in all examples. However the accuracy on the validation set decreases for most examples after a depth of three. Therefore, we construct a final Decision Tree Classifier with a maximum depth of three (Fig. 1).

B. Bagged Decision Trees

A Bagged Decision Tree classifier[5] is an ensemble method that expands upon the single decision tree. The bagged classifier consists of several decision trees. These trees are constructed using a subset of the training data. For testing, the test data is classified by each decision tree, and then every tree contributes a vote or probabilistic value to the overall classification.

We use sklearn's [4] Bagged Classifier and Decision Tree Classifier to construct the bagged decision tree classifier. Like the Decision Tree, the bagged classifier has many hyper-parameters that can be tuned using the validation set. We chose to tune the number of estimators parameter. For this Bagged Decision Tree Classifier, this represents the number of trees created in the ensemble. The results of this tuning are displayed in Fig. 3 below.

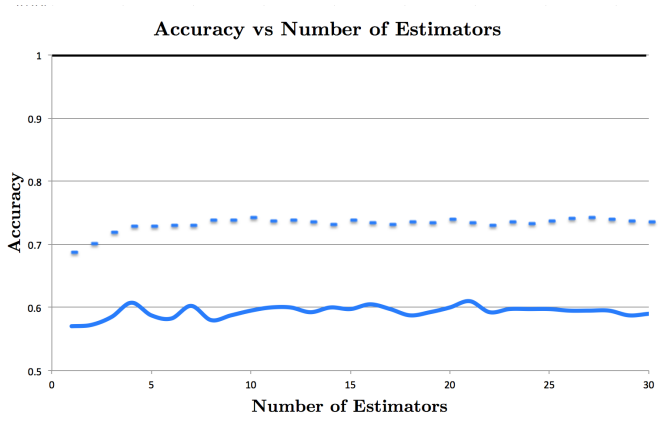

Fig. 3

We see that, unlike the tree depth parameter, number of estimators does not seem to over-fit the data at high numbers. Therefore we use the default value of ten to train the final classifier.

C. Random Forest

For the last classifier, we use another ensemble method, Random Forest [6]. Like the bagged classifier, random forest uses multiple decision trees that all contribute to the final classification. However, unlike the bagged classifier, when constructing each tree, the feature used at each node is chosen from a subset of the total features. This allows for more variety in the trees.

Again we used sklearn's [4] implementation of a Random Forest Classifier. For the random forest classifier, we use a grid-search technique to train both the tree depth and number of estimators hyper-parameters. Essentially, we tested all possible combinations of the parameters within a reasonable range.

Like the bagging ensemble, random forest did not seem to over-fit the validation set with a high number of estimators. Also like the single decision tree and bagging ensemble, random forest did over-fit the validation set with large values for the tree depth. For the final random forest classifier, we used a max tree depth of three and the default number of estimators, 10.

IV. Results & Conclusion

After training all hyper-parameters, the original training and validation data sets were combined to form a new training set of size 1400. This new training data set was used to reconstruct each classifier, without changing the hyper-parameters. With the newly trained classifiers, we then attempted to classify the unseen test data. The resulting accuracy of all classifications, including the classification of the unseen test data, is shown in Fig. 4.

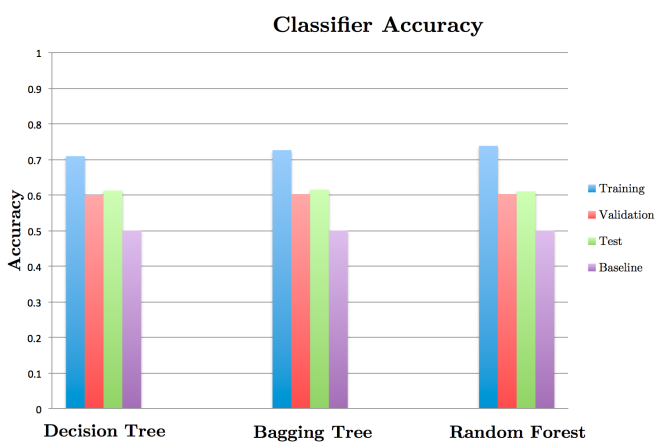

Fig. 4

As we can see, although the accuracy on the test set is better than a random or majority classifier, which would have an expected accuracy of 0.5, the test set accuracies of 0.6125, 0.615, and 0.610 for the decision trees is hardly impressive. It's especially discouraging that the ensemble classifiers fail to significantly outperform the decision trees.

There are several possibilities for the poor performance of the classifiers. One major flaw is the lack of features. Although we have 12 total features, these are derived from only three per-player data entries. Also, two

of the three data entries, gold and CS, are somewhat dependent on each other.

One solution would be to generate an expanded set of features. We can parse the event timestamped data mentioned in section II, and generate features for any events that occur in the first 10 minutes. We can also generate features from information provided about the players before the game begins. For example, the Riot Games API provides data on match history, player rank, and several other player-specific attributes.

Another improvement could be made by using sklearn's [4] grid parameter tuning algorithms, which should provide much better parameter tuning than our own simple tuning schemes and grid searches.

Finally, it may be the case that classifying whether a team loses or wins a game based on the first 10 minutes of data cannot have a significantly higher accuracy. To improve on this, we could use summary data from the first 20 minutes, or even the first 30 minutes.

Alternatively, instead of using absolute resources gained and treating the two teams in each game independently, we could generate features using resources gained relative to the opposing team. Although this would make the classifier less general, it would likely lead to better accuracy.

In conclusion, as competitive on-line games grow in popularity, the ability to make predictions about the results of in-game behaviors can be a powerful tool that players can use to develop strategies. We demonstrate three different classifiers that attempt to predict if a team wins or loses a game based on their resources attained in the first ten minutes of a game. Although the classifiers fail to accurately predict all winners, they have higher accuracy than a random prediction, and we provide several avenues for future improvement.

REFERENCES

[1] D. Segal, "Behind league of legends, e-sports's main attraction," Oct 2014. [Online]. Available: https://www.nytimes.com/2014/10/12/technology/riot-games-league-of-legends-main-attraction-esports.html

[2] Riot Games, "League of legends," 2008. [Online]. Available: http://na.leagueoflegends.com/

[3] ——, "Stats straight from the source," 2017. [Online]. Available: https://developer.riotgames.com/

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[6] ——, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.