Gavin Saldanha
UNI: gvs2106
May 19, 2015

EECS E6895
Advanced Big Data Analytics
Final Project

For my final project, I decided to develop a tool for predicting which team in a League of Legends game will win, based solely off of the characters (champions) chosen. To make it more useful to players of the game, I then added a tool to aid in choosing a champion; it chooses the champion that will maximize the player's chance to win.

League of Legends is the most played online game in the world with almost 100 million players. In the game, two teams of 5 players compete to destroy their opponent's base. The details of the gameplay are not important. However, the team setup is critical. Before each game starts, all 10 players have to choose their champion for the game. Each champion has a unique skillset that makes them strong in certain aspects of the game (dealing damage, taking damage, healing, etc), while lacking in others. Thus it is important that the team as a whole has a balanced set of champions. This is hard enough when playing with people you know - and in the ranked mode, you are paired with 4 teammates randomly. Thus, having a tool to inform you about what is best given your team's situation can be very useful.

There are 124 champions right now, and each game features 10 of them (each player must pick a unique champion, no duplicates allowed[1]), there are a total of

$$\binom{124}{10} = 1.63 * 10^{14}$$

possible matchups. Currently, the only real way to estimate which team has a stronger set of champions is to rely on one's previous experience (or someone else's experience). With 163 trillion matchup possibilities, this isn't a very reliable method. For this project, therefore, I decided to come up with a machine learning based approach that would be able to accurately predict which team composition has the upper hand. It is important to note that League of Legends is a strategic game, and relies on the player's mechanical ability and decision making - thus even with the "best" possible team comp, that team can still lose to the "worst" team comp based on the players' performances.

The first step was acquiring the training data. Riot Games, the company behind League, has published an API for developers to use that allows them to query for past and present match information, among other things. However, they do not provide a way to get a large amount of random game data - one can only get the previous matches of a given particular player. To generate my dataset therefore, I started by looking up my own prior matches. For each match, I also stored the other 9 players in the game. Then

[1]This is technically only true of ranked games, but the only type of game I considered was ranked solo queue.

for each of those, I retrieved their match histories and did the same. I made sure I never counted a match or player twice. Using this sort of rippling approach, I was able to get a pretty sizeable dataset - about 1.5 GB. The Riot API has some pretty severe rate limiting, so for an initial application such as this, getting that much data took about 36 hours. I have included my script for generating data in `dataset_builder.py`. It can be run with no arguments, but requires that there exist a file `saved_state` containing (at least) 4 lines - the set of players that have already been processed, the set of players yet to be processed but who have been discovered, the set of match ID's that have been processed, and an integer denoting which output file to write to next. This is all needed by the script in order to avoid double counting either matches or players. The script generates many output files, each containing several hundred matches. It is divided into many scripts to be fault tolerant - if the program crashes, it will only affect the one file. With code that has to execute for as long as this, that was an important concern.

At this point I had a directory `data` containing about 45 thousand games. In order to efficiently deal with this volume of data - and be able to scale up for substantially more data in the future - I decided to use Spark (pySpark). Spark is able to process the data amazingly quickly and has all the machine learning algorithms I could possibly want built in. Running locally, Spark is able to run my complete code in a few minutes time, where a typical, sequential Python program could take up to an hour.

The main part of my code is basically in 2 parts: first the machine learning system, and second a webserver to allow players to use the model to predict the outcomes of their games. The webserver is very basic, intended only as a proof of concept. The majority of my work was done on the machine learning system.

The file `learner.py` contains all the data processing and machine learning algorithms. The `main` function trains several different models on the training data and evaluates all of them, both on the training data and a held out test set. It outputs the evaluation statistics to a file and returns the models with highest scores. The models I have tried include SVM, Logistic Regression using Stochastic Gradient Descent, Logistic Regression using Limited Memory BFGS, Decision Trees, and Random Forests. I also have a baseline model which always predicts the winner to be the team whose average champion win rate is higher. For each model (except the baseline) I also have two vector formats derived from the match data, one that only incorporates which champions are being played, and one that also includes in which role they are played. Unsurprisingly, using the role information provides better results. I also tuned the various parameters for the models to see if I could squeeze any extra performance, but was unsuccessful. I then created several aggregate models, which used any odd number of models to independently predict the winner, then used a majority vote system to output its final guess. However, these seemed to do worse than any of the models. Attached at the end of the report is the output file containing the evaluation statistics for the various algorithms.

Given the nature of the game, I did not expect to get better than 55% accuracy - the outcomes of the games are determined primarily by player skill. The baseline method gets about 53.5% accuracy on the held out test set I used. This was pretty reasonable. Both SVM and logistic regression outperformed the baseline. SVM achieved a best accuracy of about 55.5%, while logistic regression was able to get 56.5% with the BFGS method of converging. That's about a 5% improvement over the baseline, and shows that the models were successful. The decision trees and random forests did not perform well, only getting between 50.5 and 53% accuracy. Again it's important to realize that the team that plays better will probably win, and the team composition will simply give an edge to one team.

To make this project useful for League players, I also created a very simple Django-based webserver. The web page simply asks a player to input the two team compositions - all 10 champions - and outputs which team is more likely to win, based on the trained model. It also gives a sort of margin that tells whether the two comps are close or if one is a heavy favorite. There is an option to include the role information for better accuracy. The page can also be uesd for players currently choosing their champion. During the champion select, teams alternate picking their champions. This page asks the user to enter the picks thus far and which pick is next, and then responds with the best champion for that pick, according to the trained model. This portion is completely experimental as I have no way of verifying its results. However, the results seems reasonable - based on my experience playing the game.

The Django code basically uses `learner.py`'s functions to produce its results. It runs `main` on startup of the server to get the best model for each vector format. Then whenever a user submits the form for either predicting the winner or getting the best champion, the appropriate function in `learner.py` is called, passing in the model, champions, and whatever other inputs are needed. It then displays the output - either the predicted winner (and a measure of confidence), or the best champion to pick next.