David Z, Hannah H, Tony L, Zhisong L

Report Document

Project Overview

For our project, we decided to try and predict how well a board of units in the game *Teamfight Tactics* would perform. The dataset we used was obtained from the Riot Developer API. The data we obtained was last 25 games played by the top 250 players in the North American Server. Each game has 8 players, each of which has 8 units, so we have approximately 8 * 8 * 25 * 250 = around 400000 data points of units and placements. This data was aggregated into the average placement of each unit, which we added weights to, and created a driver program to access our aggregated data, and take a user inputted board to predict the average placement of that board.

Data

Our data was obtained from the Riot Developer API. The first part obtained was the ID's of the top 250 players of every region (NA, EU, KR, etc). We then used the ID's of the North American players to make API calls for their last 25 games played. One API call for a match returns a JSON string with info about the match, such as the version of the game played, the match ID, and a list of participants in the game. Each participant is another JSON string with info about their placement, what units they used, what items those units had, augments picked, the time they lost, the round they lost, and other data points. We only cared about the placement, and what units were used to achieve that placement

Scala/Spark Parsing

We used a spray JSON library to obtain the info from the match JSON, and parsed and filtered it into the data we wanted. Our first parse changed our data to rows consisting of: match_id, placement, list_of_units, where list_of_units consisted of unit data in the form of unit_name, rarity, tier. We then removed all duplicate lines, as there could be repeating games from our initial data. Each unit in the list was then associated with the placement, and then aggregated by key (unit_name, rarity, tier), to find the average of the values (placements). Each unit will have 3 rows in our final aggregation here, as we treat units with the same name, but different tiers as separate.

Adding Weights

Consideration about the weight / impact of a given unit was necessary, because units of different tiers and rarities will have different stats and damage scaling. Thus, there was a two-phased weighting system, first with the rarity on its own, then by a rarity and tier combined factor. This was to improve outlier predictability (albeit still not ideal).

Driver

In order to test and use our aggregated data, we created a driver app. This program takes a user input of any number of (unit, tier) pairs, and is used with our weighted data to generate an average placement of that input.

Workflow of Project:

User runs the program with program arguments of

teamComposition.input **and** championsFile.input

The team composition input from the user should be formatted to…

UnitName [String]	Tier [Int]
unitNameX	tierX
unitNameY	tierY
…	…

The champions file input is pre-generated / provided on the GitHub repository to the format of…

UnitName [String]	Rarity [Int]	Tier [Int]	Avg. Placement [Int]	Weight [Float]
unitNameX	rarityX	tierX	placementX	weightX
unitNameY	rarityY	tierY	placementY	weightY
…	…	…	…	…

Each file is remapped to use a key of (UnitName, Tier). The value of team composition being Null and champions file being (placement, weight). These RDDs are then joined on this key, and mapped again to remove the Null value.

A computation of a weighted average is performed via the following formula.

$$totalUnitWeight = \Sigma\,(unitWeight * tierRarityFactor)$$
$$expectedPlacement = \Sigma\,(unitPlacement * (unitWeight * tierRarityFactor\,/\,totalUnitWeight))$$

Results:

This project revealed that top players tend to optimize their unit selection to place between 4th and 5th place through units alone. This allows for the variability of other factors such as Augments, Items, and Traits to bring their placement higher. Beyond this, the usefulness of the project can be seen as a baseline for new and veteran players alike to see if their team composition is likely to make it to the top 4.

Obstacles:

An obstacle we ran into early on was retrieving the match data from the Riot API. There were many fields to parse through and filter to acquire the relevant data in a way that was accessible to our scala program, but we soon became familiar with the general format for parsing of these json objects.

The primary obstacle we continuously ran into in the process of developing this project is dealing with the multiple factors that determine a player's outcome. As aforementioned, to maintain a reasonable scope we chose to focus on tier, rarity, and placement data. This does not, for example, take into account how the different combination of characters impact the results. In turn, we decided to using weights, based on general player perception of tier / rarity value so that each field held a representative amount of impact on the placement prediction, which also proved to be a complex task.

In addition, we were working with a data set of top players specifically from North America, which provides a limited range in placement and playing style data to compare to, specifically data that reflects the impact of units on higher placing players. We do, however, realize that first and last placements are outliers when comparing the data, which make them significantly more difficult to predict.

Learning:

In this project, the usefulness of aggregation and joins were reemphasized. In addition to this, we were able to make predictions in spite of external factors, such that insights could still be obtained from our results.

Learning to pull from the Riot Games' API was a collateral of having to grab large datasets of information that pertains to our group members' interests.