

Interpolation on Graphs: Predicting Overwatch League Matches*

Bansi Chhatrala[†] Dr. John P. Ward[‡] Joseph Graves[‡]

July 2020

Abstract

This paper describes how interpolation on graphs has been applied to predicting the results of matches in the Overwatch League, a professional esports. Interpolation is the act of estimating a value based on other known values. In this project, we worked with a complete set of match data from the 2019 season to develop and test our predictive functions. We used data science techniques to gather and organize our data, then constructed a derivative matrix and a weight function in order to make predictions with linear algebra. We conducted lots of testing on our weight function in order to make it as accurate as possible, and we learned the limits of predictive modeling.

1 Introduction

The problem we addressed is predicting the outcome of a sports match using previous matches. In this project, we visualized matches in a network and used adjacent matches to interpolate the outcome of our targets. We built our model based on data from the 2019 season of the esports Overwatch League. From our work, we determined what goes into this sort of predictive model and the limits these kinds of models have.

*this research was funded by the National Science Foundation

[†]North Carolina State University, Raleigh, NC

[‡]North Carolina A&T University, Greensboro, NC

2 Background

Interpolation is the calculation or estimation of an unknown value based on its surrounding values. Thus, graph interpolation is the estimation of a node's value in a graph using adjacent nodes. Since interpolation is predictive by nature, our problem was to be able to use it to predict the outcome of sports matches. Given prior match outcomes and team information, can we accurately determine the outcome of matches not yet played?

For this project, we chose to work with esports, which are multiplayer video games played competitively. Specifically, we worked with Overwatch League, played on Overwatch, a video game by the company Blizzard. Overwatch is a futuristic first-person shooter game. Players can choose from a roster of heroes to play, each with their own special abilities and classes. Six players at a time play for each team; in Overwatch League, they comprise two each of damage, tank, and support characters.

Many different factors go into an Overwatch match that helped us determine how similar teams are, which was important in our interpolation. For example, in Overwatch, there are four different types of maps: Assault, Escort, Hybrid, and Control. We could see whether certain teams consistently win or lose at certain maps; two teams were likely to have similar play styles if they consistently won at Control maps, for example. In this project, we wanted to test how descriptive different factors were of a team. We used these to determine which teams have the most similar outcomes in matches, helping us make our predictions.

We chose to use the Overwatch League because its data was readily available. Since it is an esports, computers are always collecting and storing data for it during games. Because of this, we were able to get lots of specific data. Additionally, the Overwatch League data set is complete; that is, each team played every team at least once. Developing a model using complete data helped us determine which factors were the most indicative of a game's outcome.

Although the data set we worked with is complete, our work is applicable to incomplete data. As the Overwatch League grows, the feasibility of having each team play each other team diminishes. A successful graph interpolation model can predict what would happen in a match between two teams that did not get to play each other. Moreover, we can apply the model to other Overwatch leagues. Notably, the college esports scene is growing rapidly, with over 100 college varsity esports programs in the National Association of Collegiate Esports [1].

We can also expand our model to work with different sports. For example, in college football, there are five Power Five conferences, but only four playoff spots. Being able to reasonably predict the outcomes of hypothetical matches between teams, especially across different conferences, will assist in determining which teams should make the playoffs. The overarching goal remains the same across all sports: to predict the outcomes of hypothetical matches between teams by analyzing ones already played.

Table 1 contains the definitions to some words specific to esports and Overwatch League which we use in this paper.

Hero	A playable character in Overwatch. Weapons, attacks, and abilities differ between heroes.
Map	The in-game location where an esports match is played. Each of Overwatch’s four map types has different objectives a team must complete in order to win.
Ultimate	A powerful ability that must be charged in order to be used. Each Overwatch hero has a distinct ultimate.

Table 1: Overwatch and esports terminology

3 Literature Review

3.1 Derivative Matrix

We used a derivative matrix to describe the connections between matches in our dataset. The inspiration behind this matrix comes from Benedetto and Koprowski’s paper, which focused on the Fourier transforms of graphs [4]. While we did not use Fourier transforms in this project, our derivative matrix was drawn from this paper’s incidence and weight matrices. Benedetto and Koprowski defined the $|E| \times |V|$ unnormalized incidence matrix and the $|E| \times |E|$ weight matrix for a graph, where E and V are the edge and vertex sets of the graph, respectively. Our derivative matrix had the dimensions of an incidence matrix, and the nonzero entries of the matrix were weights. These weights described the relationships between matches based on which teams played each match; we discuss this further in Methods.

3.2 Ranking and Comparing Teams

In our project, rankings referred to the official Overwatch League team rankings. Keener’s paper on the Perron-Frobenius theorem and the ranking of football teams [5] considered different methods that could be used to rank football teams. We drew inspiration from the ranking methods Keener discussed for determining team similarities. Keener’s paper described different ways to rank teams based on their performance against other teams. Similarly, we used teams’ performances against other teams in making our predictions; we used score differences with the functions in the minimization problem, which we discuss further in Methods.

3.3 The Importance of Weights

Shuman et al.’s paper about signal processing [7] helped us describe our motivation for using graphs as a model. The authors described how using a graph yields better results

over classical models in the context of signal processing. In addition, they demonstrated how weighted graphs in particular yield better results. An example from the paper is using graphs for image processing, specifically for denoising. The model gave better results when a weighted graph was used over an unweighted because it allowed more weight to be placed where there was a larger difference between pixel intensities. We applied this in our project by using a weighted graph. In our model, an edge connected two nodes, which represented matches, if they had a team in common. We determined how similar teams were based on how close they were in certain factors, such as overall ranking and healing done. Edges between matches with similar teams were weighted more heavily than those between matches with different teams. Thus, the team similarities in our model mirrored the differences between pixel intensities in the signal processing paper.

3.4 Splines

Pesenson’s paper [6] studied a version of splines on combinatorial graphs. Variational splines were introduced in this paper as a method of interpolation. These splines were also described as being able to provide optimal approximations to the functions they interpolated. A key difference between this paper and our work was that weighted graphs were not considered in Pesenson’s work, while ours relied heavily on weights.

Ward et al.’s paper worked with constructing splines for interpolation using the Laplacian as a derivative [8]. The authors considered how the smoothness of data affects interpolation. Ward et al.’s project specifically worked with finite, connected, weighted graphs, where data was known on a subset of the vertices, much like ours. They considered the decay and localization properties of different splines. We primarily focused on how well the splines we made represented team performances. A key difference from our project was that rather than using the Laplacian to construct splines, we used our derivative matrix. Furthermore, Ward et al.’s aim was to introduce more approximation tools for graphs; we expected to apply these to create an approximation tool specific to sports analytics and prediction.

Beautiful Soup	A library for extracting information from pages written in a markup language. We used Beautiful Soup to read HTML web pages.
NumPy	A Python library that supports arrays and matrices. We used it to store the data we scraped in arrays as well as create our derivative matrix later in the project.
re	A module for using regular expressions (regex) in Python. We used re to search for data that followed certain regex patterns in our web pages.
urllib	A Python module for opening urls. We used urllib to open and read a webpage into BeautifulSoup.

Table 2: Python libraries and their descriptions

4 Methods

4.1 Data Collection

We collected and organized our initial match data and team rankings from Liquipedia [2], which is an esports wiki run by volunteers. We used the urllib library to open the url to the webpage that we needed to scrape. Then, we used functions from the BeautifulSoup and re (regular expressions) Python libraries to scrape the data and arrays from the NumPy library to store it. Scraping is the process of collecting data from a human-readable format and storing it in a structure computers can work with. We wanted to scrape data from Liquipedia so that we could create the networks for our model to work with. Table 2 describes the functions of each Python library we used when collecting data. Figure 1 shows the table of match data we scraped from Liquipedia.

To scrape the data from the table shown above, we first read it using urllib’s urlopen and read functions since it was on a web page. Then, we passed it through the HTML parser included in BeautifulSoup. This parser reads a page written in HTML and extracts the different components of it, such as the header, tags, and body text. Once we extracted









































		4-0 Aug 25	4-0 Feb 15	3-1 Aug 04	0-4 Apr 07	3-2 May 04	4-0 Aug 03	2-3 Jun 22	3-1 Feb 22	3-1 Apr 14										
	3-2 Apr 04		0-4 Aug 15	0-4 Jun 14	0-4 Jun 07	1-3 Aug 24	3-2 Jun 23	2-3 Jun 29	1-3 Mar 03	1-3 Aug 09										
	0-4 Jul 07	0-4 Mar 10		1-3 Mar 02	3-0 Aug 03	0-3 Jul 06	1-3 Jun 16	1-3 Apr 06	4-0 Aug 18	2-3 Aug 01										
	1-3 Mar 17	2-3 Feb 17	1-3 Jun 23		1-3 Aug 10	2-3 Jun 07	1-3 Jul 25	1-3 Aug 02	3-1 Jun 30	3-0 Jun 28										
	0-4 Aug 18	4-0 Apr 20	2-1 Apr 05	4-0 May 04		2-3 Jun 21	1-3 Feb 16	2-1 Apr 13	3-1 Jul 28	2-1 Jun 22										
	1-3 Apr 19	2-1 Feb 14	4-0 Apr 14	4-0 Feb 24	4-0 Jun 09		3-1 Aug 02	3-1 Apr 18	4-0 Jul 07	4-0 Apr 11										
	0-4 Mar 02	3-1 Jun 09	3-1 Apr 11	1-3 Jun 20	0-4 Apr 27	1-3 Jun 14		3-2 Aug 08	3-1 Jun 07	1-3 Aug 18										
	3-2 Feb 17	2-1 Jun 16	1-2 Feb 21	4-0 Apr 19	3-1 Feb 14	0-4 Apr 04	3-1 Mar 17		3-2 Aug 04	3-1 Jul 06										
	1-3 Jul 06	2-3 Apr 06	1-3 Aug 08	3-2 Feb 15	1-3 Jun 14	1-3 Mar 01	4-0 May 05	1-3 Apr 11		1-3 Jul 26										
	1-2 Aug 16	3-2 May 05	3-2 Mar 17	3-2 Aug 11	2-3 Feb 21	1-3 Feb 16	1-2 Mar 16	1-3 Feb 28	1-3 Apr 05											
	3-2 Mar 16	3-2 Jul 28	3-2 Feb 23	3-2 Aug 17	3-1 Apr 28	4-0 Apr 04	4-0 Apr 06	3-2 Jul 26	1-3 Mar 07	3-1 Apr 07		4-0 Jun 23	3-2 Feb 15	1-3 Aug 10	2-3 Aug 02	2-3 May 04	0-4 Aug 16	0-3 Jun 30	1-3 Apr 12	1-3 Jun 21
	1-3 Aug 24	2-3 Mar 16	3-1 May 02	3-1 Jun 29	1-3 Jul 26	2-3 Jun 29	2-1 Apr 14	3-1 Feb 23	4-0 Jun 09	3-0 Jun 15	1-3 Jun 13		0-4 Aug 18	1-3 Aug 15	2-3 Aug 09	1-3 Aug 03	0-4 Feb 15	0-4 Apr 20	4-0 Mar 08	0-4 Jun 15
	3-1 Apr 21	4-0 Jul 27	3-2 Feb 28	3-2 May 02	1-3 Aug 01	4-0 Aug 11	0-4 Apr 04	3-2 Jul 25	4-0 Jun 22	4-0 Jul 07	3-1 Jun 06	4-0 Feb 21		3-0 Aug 02	1-3 Mar 09	1-3 Jun 15	0-4 Apr 12	3-2 Aug 08	3-1 Jul 06	0-4 Mar 17
	1-3 Jul 27	3-2 Apr 12	4-0 Jun 29	1-3 Feb 22	1-3 Feb 24	2-3 Aug 09	3-2 Apr 28	4-0 Mar 09	3-0 Jun 15	3-1 Apr 27	3-0 Jun 20	3-2 Apr 19		3-1 Mar 03	3-2 Jun 27	0-4 Apr 21	3-2 Jun 22	4-0 Aug 25	4-0 Jun 13	1-3 Jun 13
	4-0 Mar 07	1-3 May 03	3-1 Apr 18	1-3 Jul 27	1-2 Mar 01	3-1 Jul 25	1-2 Feb 23	3-1 Jun 14	4-0 Jun 28	4-0 Jun 16	3-1 Apr 20	4-0 Jun 07	4-0 Apr 13	2-3 Aug 16		3-1 Aug 24	2-3 Apr 03	3-2 Apr 05	1-3 Jun 21	0-4 Jun 30
	3-2 Apr 12	3-1 Aug 17	2-1 Aug 10	1-2 Mar 09	4-0 Jun 28	2-3 Feb 22	3-2 Jul 27	2-3 Mar 07	1-2 Feb 24	3-1 Apr 19	4-0 Jun 08	4-0 Apr 27	4-0 Mar 03	1-3 Feb 16	2-3 Apr 14		0-4 Apr 05	2-3 Aug 01	1-3 Jun 06	3-1 Jun 23
	3-2 Jun 06	4-0 Jun 20	4-0 Jun 21	2-3 Jun 13	4-0 Jun 30	0-4 Mar 08	3-1 Mar 10	4-0 May 02	4-0 Apr 18	4-0 Mar 02	2-3 Jun 27	4-0 Aug 11	4-0 Apr 07	3-1 Feb 28	2-3 Feb 17	4-0 Aug 25		4-0 Jun 16	3-1 Aug 09	3-2 Aug 24
	4-0 Jun 15	1-3 Feb 28	4-0 Jun 06	4-0 Apr 27	3-0 Mar 16	1-3 Mar 03	3-1 Jun 27	1-3 Aug 17	2-1 Aug 10	3-1 Mar 09	4-0 Feb 21	1-3 Feb 17	4-0 Jun 08	1-3 May 05	3-1 Feb 14	3-2 Apr 28	1-3 Jul 26		3-1 Aug 15	0-4 May 04
	3-2 Jun 13	3-1 Feb 22	3-1 Apr 20	3-0 Apr 21	3-2 Mar 10	0-3 Apr 06	1-3 Aug 17	3-1 Jul 07	1-2 Aug 01	3-1 May 03	4-0 Mar 01	2-3 Mar 02	3-1 Jun 29	1-3 Feb 14	1-3 Apr 04	1-3 Jun 20	0-4 May 05	3-1 Jul 28		1-3 Jul 25
	3-1 Jun 08	4-0 Apr 21	4-0 Jul 28	3-1 Apr 13	3-2 Aug 08	3-2 Aug 25	4-0 Mar 08	4-0 Aug 11	3-1 May 03	0-4 Aug 04	3-2 Mar 10	4-0 Apr 18	4-0 Feb 23	4-0 Apr 07	3-1 Jun 09	3-1 Mar 01	3-1 Feb 24	3-1 Apr 11	4-0 Feb 16	
																				

Figure 1: Liquipedia’s match overview table

the text, we skimmed it to determine the format of the match data we wanted. “4-0Aug 25” was the text corresponding to the first match in the table, Atlanta Reign vs Boston Uprising. From this, we formed the regular expression pattern “[0-9]-[0-9][a-zA-Z]3 [0-9]2” to find the match data in the table. This pattern matches a hyphenated numerical score followed by a date that begins with a 3-letter month abbreviation.

Once we had successfully gathered all the matches that were played and removed the duplicates, we imported them into a NumPy array. Each row held team 1, team 1’s score, team 2, and team 2’s score. We realized that the teams were in the same order on both axes of the match overview table. Therefore, we were able to automate importing our match data into the array. We used nested for loops to do so, iterating through each pair of teams and adding their match to the array.

To scrape the team ranking data from Liquipedia, which was on the same page, we followed a process similar to what we had done for matches. We used the regular expression “\n [A-Z].+\n” to find team rankings. Team ranking data in the HTML file was surrounded by newlines, so we matched newline characters as well as a capital letter to ensure that we only matched team names and their rankings.

After gathering the data available on Liquipedia, we collected additional data from The Overwatch League Stats Lab [3]. Stats Lab is Blizzard’s official website that showcases Overwatch League data for fans to interact with. They have spreadsheets of data available to download, so we downloaded the 2019 player data as well as the map stats, as those were useful for making team comparisons. The StatsLab data was in csv format, so we used NumPy’s `genfromtxt` function to create NumPy arrays of the data contained in each spreadsheet. Since the spreadsheets were large files (around 40,000 kilobytes each), this section of our code took the longest time to execute.

4.2 Creating Our Graph

After we obtained the results for each match, we built our graph. We made each match a node and then generated our edge list by placing edges between any two matches that had at least one team in common. From our 280 matches, we found 7,470 edges. We were able to use Python’s NetworkX package to build a graph for visualization purposes. However, with 280 nodes and 7,470 edges, it was far too large to visualize. For graph visualizations throughout this paper, we will use smaller subgraphs. These snippets, such as Figure 2, reflect the structure of our graph.

4.3 Data Wrangling

We then proceeded to wrangle the scraped and collected data. Data wrangling is the process of organizing our data into a form that is more conducive to our work. Since the

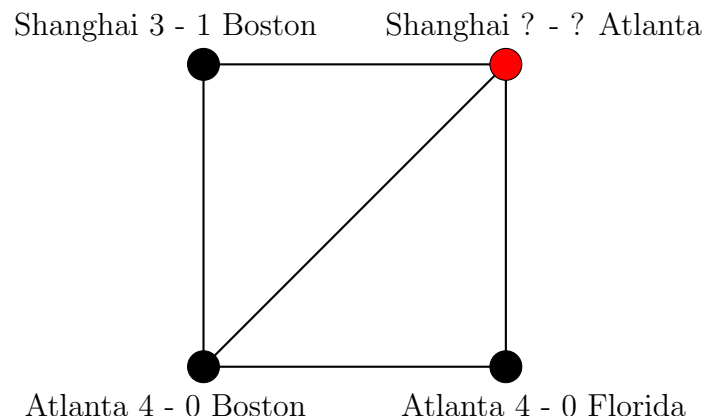


Figure 2: A graph snippet showing how matches are connected

weight function required data for each team, we made a Python dictionary, with the keys being team names and the values being lists of information we had about each team.

First, we collected the information on how many times each team had won, lost, and played on each map. In order to do this, we iterated once through the map array that we had generated from Stats Lab's maps spreadsheet to make a set of all of the maps played during the season. Then, we initialized three empty dictionaries (one for wins, one for losses, and one for plays), with maps as keys and frequencies as values, in each team information list. With everything initialized, we iterated over the map array once more, this time extracting which team won and lost at each map and adding the frequencies into their win and loss dictionaries, respectively. Once we had finished gathering win and loss data, we set the values in the plays dictionary for each team to equal the sum of their wins and losses at each map.

Once we had gathered our map frequencies, we needed to sort them. Since Python does not have a function to sort dictionaries by value, we acquired sorted lists instead, and then wrote a custom function to convert these sorted lists back into dictionaries. The code from 8.2.1 in the appendix shows how we wrangled the map data.

After organizing our map frequencies, we extracted the quantitative statistics that had the potential to be good indicators of team similarity. Table 3 contains short descriptors

of each stat we chose to take from the spreadsheets as well as the ones we calculated.

All Damage Done	The total damage output of a player, including damage done to heroes, shields, etc.
Assists	The number of times a player assisted in an elimination
Deaths	The number of times a player was eliminated
Eliminations	The number of times a player eliminated another player, also known as kills
Healing Done	A quantifier of how much a player healed their teammates
Hero Damage Done	The damage a player did to other heroes only
Time Played	The amount of time each player spent playing for a team; this includes time alive and time dead
Ultimates Used	The number of ultimates a player used
Weapon Accuracy	The ratio of how many weapon fires caused damage over how many times a weapon was fired
DPS	Damage Per Second, calculated with $(\text{Hero Damage Done}) / (\text{Time Played}^1)$
KDA	Kill-Death-Assist ratio, calculated with $(\text{Eliminations} + \text{Assists}) / (\text{Deaths})$

Table 3: Game statistics

These statistics were gathered and separated by player in the spreadsheets. For our purposes, we summed each statistic over an entire team, as we were working with team information to determine similarities between teams for the weight function, which we describe in section 4.5.

4.4 Creating The Derivative Matrix

To use the linear algebra functions we needed to use to make our predictions, we had to organize our match data into a derivative matrix. The dimensions of our derivative

matrix were $|E| \times |V|$, with E being our edge list and V being our node list. Our derivative matrix was akin to an incidence matrix [4], as mentioned in our literature review. Each row of the matrix contained values that corresponded to the nodes, aka matches, it connected. These values were weights that described how similar the two matches were. Since we made the derivative matrix before we wrote our weight functions, we used placeholder weights at first.

Using the same graph snippet from Figure 2, we can show how the derivative matrix is structured. In order to understand how we will use our derivative matrix, we must also understand the minimization problem. The value f at each node indicates the score difference of that match. The predictions we made were on the nodes with unknown f ; we found the values that minimized the functions shown in Figure 4.

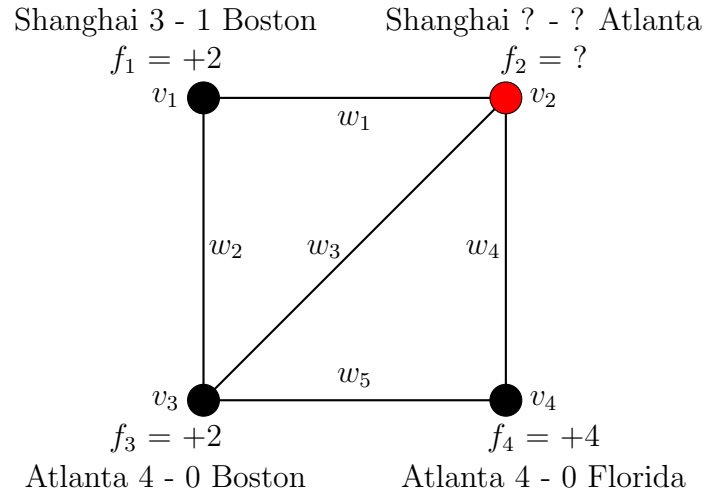


Figure 3: Graph snippet with labeled vertices and edges

$$\begin{array}{c}
w_1 \left| f_1 - f_2 \right| \\
w_2 \left| f_1 - f_3 \right| \\
w_3 \left| f_2 + f_3 \right| \\
w_4 \left| f_2 + f_4 \right| \\
w_5 \left| f_3 - f_4 \right|
\end{array}$$

Figure 4: Minimization problem functions

$$\begin{bmatrix}
w_1 & -w_1 & 0 & 0 \\
w_2 & 0 & -w_2 & 0 \\
0 & w_3 & w_3 & 0 \\
0 & w_4 & 0 & w_4 \\
0 & 0 & w_5 & -w_5
\end{bmatrix}$$

Figure 5: The derivative matrix for the graph from Figure 3

As shown here, we populated the derivative matrix with weights that corresponded to the minimization problem functions. Whether the labels were both positive or one positive and one negative was determined by the positions of the teams. In the first row, for example, we had one positive and one negative weight, since Shanghai was Team 1 in both of those matches.

For efficiency and the ability to rebuild this matrix whenever we changed the weight function, we automated its creation. We made a NumPy array of zeros for our matrix, with our edge list length and game list length as dimensions. Then, we used a for loop to populate the matrix. For each row of the matrix, we extracted the corresponding edge, and the matches connected by that edge. We extracted the teams from those matches, then used a series of conditions to determine whether we called our weight function or entered a large constant weight. Additionally, we were determining the signs of the weights we entered.

These signs corresponded to the signs from the minimization problem functions. The code for creating the derivative matrix is in section 8.4 of the appendix.

Since we had not yet developed a weight function, we created a placeholder that only calculated weights based on team rankings. This way, we could confirm that we built our derivative matrix properly. We wrote the code for making predictions at this point so that we would be able to test our derivative matrix and weight functions. Once we were confident that the matrix was built as expected, we proceeded to create our weight function.

4.5 Developing A Weight Function

To determine which factors we wanted to use for our weight function, we tested a variety of factors for their prediction accuracy. We also used these tests to determine how heavily each factor affected the weights.

We tested each weight function against the results of our existing match data to determine the predictive accuracy of each function. For each of these tests, we used the first N matches as our known matches and had our model make predictions on the remaining matches. Section 4.6 describes the prediction-making process in more detail. Once we had these predictions, we compared the signs of our predicted score differences against the actual ones to determine how frequently our model accurately predicted a winner. The tests we ran were as follows:

1. Using the first 200 matches to predict the last 80 matches
2. Using the first 230 matches to predict the last 50 matches
3. Using the first 250 matches to predict the last 30 matches
4. Using the first 270 matches to predict the last 10 matches

For the purpose of gauging the accuracy of our model, we focused on the results of tests 1-3 the most, as the larger samples better represented the predictive results and were

less prone to being drastically affected by upsets.

At first, we gradually developed our weight function by adding factors to it and testing every time. However, this did not give us an accurate representation for how each factor alone describes the teams, so we decided to test each factor independently. We wrote individual weight functions for each factor we wanted to run tests on. Each function had roughly the same structure, excepting the one we used for determining how similar two different teams' frequently played maps were. The weight function for each non-map factor followed this structure:

```
1 # this version only considers FACTOR
2 def weight(ti1, ti2):
3     fac1 = ti1[4]['FACTOR']
4     fac2 = ti2[4]['FACTOR']
5     distance = abs(fac1 - fac2)
6     weight = 1 / distance
7     return(weight)
```

Figure 6: Structure of a single-factor weight function

The function took in the lists of information we had on both teams we wanted to compare. At index 4 within these lists was a dictionary of stats, from which we extracted the value for the stat name we wanted to use. In order for our weights to come out similarly each time, we normalized each factor by finding the largest instance of it across all teams and dividing each team's instance of that factor by the largest. For example, since 20 was the highest ranking, we divided each team's rank by 20 to normalize it.

In order to test the predictive accuracy when using map frequencies alone, we needed a different function. We had 3 map lists, wins, losses, and plays, sorted from most frequent to least frequent for each team. We gauged how similar or different two teams' frequently played maps were by testing whether each map was at the same position frequency-wise for both teams in any of these 3 lists. The code for this function is shown in Figure 7.

We included Map Frequency with our individual factor tests to determine how much it factored in to determining how similar teams were, in order to be thorough with our

```

1 def weight(ti1, ti2):
2     plays1 = list(ti1[3].keys())
3     plays2 = list(ti2[3].keys())
4     wins1 = list(ti1[1].keys())
5     wins2 = list(ti2[1].keys())
6     losses1 = list(ti1[2].keys())
7     losses2 = list(ti2[2].keys())
8
9     mapFactor = 0.1;
10    # iterate over both lists and consider how many are or aren't in the
    same position
11    for i in range(len(plays1)):
12        if(plays1[i] != plays2[i] and wins1[i] != wins2[i] and losses1[i]
    != losses2[i]):
13            mapFactor += 0.1
14    weight = 1 / mapFactor
15    return(weight)

```

Figure 7: Weight function that only considers map frequency similarity

testing. However, we used Map Frequency differently from the other factors in our conglomerate weight function. The results of testing our predictions with each individual factor determining the weight function are shown in Table 4.

Once we had determined how accurate each of these predictive factors were on their own, we pulled them together to make a conglomerate weight function. Our goal was to make our predictions as accurate as possible; by combining the more accurate predictive factors, we could create a weight function that best represented the similarities between teams and, therefore, could make relatively accurate predictions on match outcomes.

To begin building our conglomerate weight function, we started with a base that only used team ranking, as that had been one of our best performing individual factors in our tests. Then, we attempted to factor in map frequencies. We could not normalize map frequencies as we had our other factors; the other factors had all been separated by team, while our map factor was based on the similarities between two teams' maps played. We originally tested it out as an additive factor alongside ranking, but its accuracy was on par with using maps alone, and we wanted our function to be more accurate than that. For a different approach, we tried making map frequencies a multiplicative factor. This gave us better results than before, so we stuck with making maps a multiplicative factor in future

	% of correct predictions			
Factor	Test 1	Test 2	Test 3	Test 4
All Damage Done	66.25	56.00	56.67	80.00
DPS ²	76.25	66.00	73.33	80.00
Healing Done	75.00	76.00	83.33	80.00
Hero Damage Done	73.75	68.00	73.33	80.00
KDA	72.50	72.00	76.67	90.00
Map Frequency	71.25	68.00	73.33	70.00
Ranking	73.75	76.00	83.33	90.00
Weapon Accuracy	62.50	62.00	73.33	90.00
Ultimates Used	70.00	70.00	76.67	80.00

Table 4: Predictive Accuracy of Individual Factors

iterations of the weight function.

For other additive factors besides ranking, we looked to the ones that had performed best in our individual tests. We also wanted to avoid any overlap between factors; for instance, we did not want to use both DPS and a damage stat since damage went into the DPS calculation.

Due to time constraints, we were not able to test several different versions of the weight function. The one we ended up with is in the code snippet below, and it performed with about 80% accuracy across all tests. In section 5.2, we will discuss potential improvements that could be made to the weight function.

We used this complete weight function to populate our derivative matrix in order to make predictions. These weights determined how similar two matches were; the higher the weight, the more similar the matches.

Figure 9 gives us a visual of what a weighted graph of our problem would look like


```

1 def weight(ti1,ti2):
2     distance = abs(ti1[0]-ti2[0])
3
4     k1 = ti1[4]['KDA']
5     k2 = ti2[4]['KDA']
6     kdafac = abs(k1 - k2)
7
8     h1 = ti1[4]['Healing Done']
9     h2 = ti2[4]['Healing Done']
10    hfac = abs(h1 - h2)
11
12    distance = distance * .4 + kdafac * .3 + hfac * .3
13
14    plays1 = list(ti1[3].keys())
15    plays2 = list(ti2[3].keys())
16    wins1 = list(ti1[1].keys())
17    wins2 = list(ti2[1].keys())
18    losses1 = list(ti1[2].keys())
19    losses2 = list(ti2[2].keys())
20
21    mapFactor = 0.1;
22
23    for i in range(len(plays1)):
24        if(plays1[i] != plays2[i] and wins1[i] != wins2[i] and losses1[i]
25        != losses2[i]):
26            mapFactor += 0.1
27    weight = 1 / (distance * mapFactor)
28    return(weight)

```

Figure 8: Weight function final version

if we plotted it. Suppose we did not know the score of the Shanghai vs Atlanta game and wanted to make a prediction of it. Based on the weight of the middle edge, we can see that Shanghai and Boston are not very similar, and neither are Shanghai and Florida based on the right edge. However, the top edge has a relatively large weight, showing that Atlanta and Boston are similar, so we can see that the results of Shanghai vs Atlanta would be more similar to Shanghai vs Boston than the other matches we know.

4.6 Making Predictions

With the weight function complete, we had a working model to make predictions. The prediction-making process involved using linear algebra from SciPy. Specifically, we used the `lstsq` function, which computes the least-squares solution to a problem in the form $Ax = b$, where A and b are arrays. The least-squares solution this provided us with solved

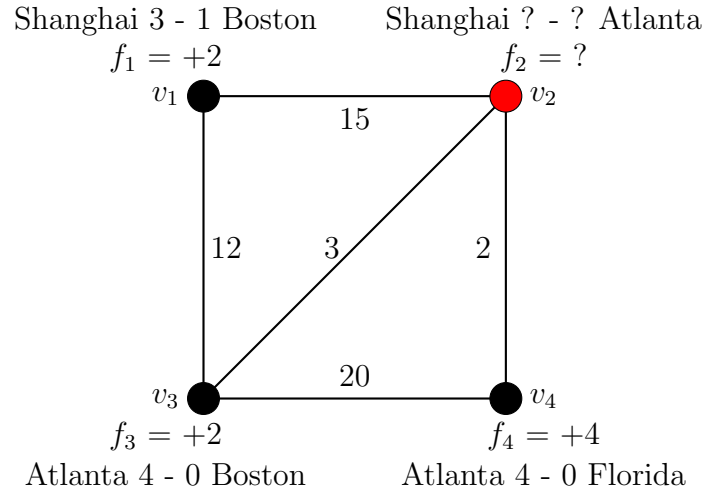


Figure 9: Weighted graph snippet with one match

the minimization problem from Figure 4.

The code we used to make predictions and determine how accurately the correct winner was predicted is shown in Figure 10.

```

1 # array of score differences for comparisons
2 sd = np.array([GameTuples[i][1]-GameTuples[i][3] for i in range(len(
   GameTuples))])
3
4 N = 200 # suppose first N games known, predict last games
5 prediction = sp.linalg.lstsq(Derivative[:,N:], - Derivative[:,N:]@sd[:N] )
6 predicted = list(prediction[0])
7 actual = sd[N:]
8 pSize = len(predicted)
9
10 correct = 0
11 for i in range(pSize):
12     if((predicted[i] < 0 and actual[i] < 0) or (predicted[i] > 0 and
   actual[i] > 0)):
13         correct += 1
14 print((correct / pSize * 100), "% correctly predicted winner \\\\")
15 print(correct, "/", pSize, "correct predictions")

```

Figure 10: Code for making predictions and determining accuracy

5 Results

5.1 Findings

Based on our work in this project, we have determined that certain factors in an Overwatch game are more predictive of a team’s performance than others, especially healing done, KDA, and team ranking. We tested several iterations of the weight function using different factors. Table 5 shows are their accuracy levels, ordered chronologically by when we created and tested them.

Factors	% of correct predictions			
	Test 1	Test 2	Test 3	Test 4
Ranking and Maps (Additive)	71.25	68.00	73.33	70.00
Ranking and Maps (Multiplicative) ³	75.00	76.00	80.00	80.00
Ranking, KDA, and Maps	73.75	70.00	70.00	80.00
Adding Multipliers to Ranking and KDA ⁴	73.75	76.00	76.67	90.00
Ranking, KDA, Healing, and Maps	75.00	80.00	80.00	80.00

Table 5: Predictive Accuracy of Conglomerate Weight Functions

Most versions of our weight function did not go above 80% prediction accuracy, which shows us there is a limit to the predictions we can make. After all, where there are sports, there are bound to be upsets, which by nature are unpredictable. We also tested out the weight functions with different proportions on the factor multipliers (for example, making healing contribute 40% to the distance rather than 30%) and ended up with nearly identical results. This implies that there is a cap to how much these factors affect our outcome.

5.2 Directions for Future Research

One way to extend this research in the future would be conducting more testing of different weight functions. Automating that process would allow us to test all sorts of

combinations of factors and how much they affect the weight function, finding us the most optimal weight function based on the factors we used.

Ranking Overwatch League teams using the Perron-Frobenius scheme and other ranking schemes Keener considered [5] would also make for an interesting question in future research. By using ranking schemes that diverge from the typical linear ranks, we may be able to get more accurate predictions. This is especially true since our weight function tests show that ranking is an important descriptor of a team.

6 Conclusion

We applied many different data science techniques to create our predictive model. In order to gather the match data we used from Liquipedia [2], we learned how to use data scraping. We also learned how to use data wrangling to get data from giant spreadsheets into a format we could use. Once we had our data together, we automated the building of a graph from it, and then created a derivative matrix to represent all of it. The culmination of our project was putting together a weight function and testing it.

Through this process, we learned which factors were the most indicative of an Overwatch League team's performance; in our project, we decided these were ranking, healing, and KDA. There is more work to be done in the field of sports prediction; Moving forward, something to consider is how different team ranking methods affect the predictive capabilities of our model. It is also worth determining exactly which factors to include to yield the most accurate predictions possible. We expect that future work will extend our model for predicting sports matches.

7 References

References

- [1] National Association of College Esports. <https://nacesports.org/about/>.
- [2] Overwatch League - 2019 Regular Season. https://liquipedia.net/overwatch/Overwatch_League/Season_2/Regular_Season.
- [3] The Overwatch League - Stats Lab (beta). <https://overwatchleague.com/en-us/statslab>.
- [4] J.J. Benedetto and P.J. Koprowski. Graph theoretic uncertainty principles. In *Sampling Theory and Applications (SampTA), 2015 International Conference on*, pages 357–361. IEEE, 2015.
- [5] James P Keener. The perron–frobenius theorem and the ranking of football teams. *SIAM review*, 35(1):80–93, 1993.
- [6] I. Pesenson. Variational splines and paley–wiener spaces on combinatorial graphs. *Constructive Approximation*, 29(1):1–21, Feb 2009.
- [7] D.I. Shuman, S.K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.
- [8] John Paul Ward, Francis J Narcowich, and Joseph D Ward. Interpolating splines on graphs for data science applications. *Applied and Computational Harmonic Analysis*, 2020.

8 Appendix

8.1 Data Collection

```
1 teams = ["Atlanta Reign", "Boston Uprising", "Florida Mayhem",
2          "Houston Outlaws", "London Spitfire", "New York Excelsior",
3          "Paris Eternal", "Philadelphia Fusion", "Toronto Defiant",
4          "Washington Justice", "Chengdu Hunters", "Dallas Fuel",
5          "Guangzhou Charge", "Hangzhou Spark", "Los Angeles Gladiators",
6          "Los Angeles Valiant", "San Francisco Shock", "Seoul Dynasty",
7          "Shanghai Dragons", "Vancouver Titans"]
8
9
10 # Attempting to read Liquipedia's Overwatch Season 19 data
11 url0 = "https://liquipedia.net/overwatch/Overwatch_League/Season_2/
12        Regular_Season#Match_Overview"
13
14 # read the website
15 page = urlopen(url0).read()
16
17 # parse the html using beautiful soup and store in variable 'soup'
18 soup = BeautifulSoup(page, 'html.parser')
19
20 # extract the text
21 text = soup.get_text()
22
23 mostart = text.index("4-0Aug 25")
24 moend = text.rfind("\nAdditional Data[edit]")
25
26 mo = text[mostart:moend] # match overview
27
28 # regex pattern of the data we want to extract
29 pattern = r'[0-9]-[0-9][a-zA-Z]{3} [0-9]{2}'
30
31 # list of all pattern matches
32 matchesd = re.findall(pattern, mo) # for 2019 data there should be 280
33 matches
34 # something went wrong when scraping - pacific v atlantic matches are all
35 # recorded twice in the html for some reason
36 # going to manually remove them for now as i'm not sure how else to go
37 # about it
38 matches = matchesd[0:109] + matchesd[119:138] + matchesd[148:167]
39 matches += matchesd[177:196] + matchesd[206:225] + matchesd[235:254]
40 matches += matchesd[264:283] + matchesd[293:312] + matchesd[322:341]
41 matches += matchesd[351:370]
42
43 # now we are going to group this data into tuples of matches
44 # first, initialize the empty array
45 # GameArray = np.empty( (len(matches), 4), dtype = object)
46 GameTuples = []
47
48 placeholder = 0
49
50 # first, the Atlantic division's games within the division
51 for i in range(10):
52     for j in range(10):
53         if(i != j):
```

```

51         t1 = teams[i]
52         s1 = matches[placeholder][0]
53         t2 = teams[j]
54         s2 = matches[placeholder][2]
55         GameTuples.append((t1,int(s1),t2,int(s2)))
56         placeholder += 1
57
58 # now, the Pacific division's games against all teams
59 for i in range(10,20):
60     for j in range(20):
61         if(i != j):
62             t1 = teams[i]
63             s1 = matches[placeholder][0]
64             t2 = teams[j]
65             s2 = matches[placeholder][2]
66             GameTuples.append((t1,int(s1),t2,int(s2)))
67             placeholder += 1
68
69 # read in team rankings, which will be used in calculating weights
70 rkstart = text.index("\n\n\n\n\n League Standings")
71 rkend = text.rfind("Stage Results")
72 rk = text[rkstart:rkend] # rankings
73
74 tpattern = r'\n [A-Z].+\n'
75 teamrank = re.findall(tpattern, rk)
76 teamrank = teamrank[1:]
77
78 TeamInfo = {}
79 for i in range(0, len(teamrank)):
80     teamrank[i] = teamrank[i][2:-1]
81     TeamInfo[teamrank[i]] = [float((i+1)/len(teamrank)),]
82
83
84 # read in all the data from StatsLab that we might want to use when
    calculating weights
85 # the csv files referenced here are the StatsLab spreadsheets
86 s2019_stage1 = np.genfromtxt('phs_2019_stage_1.csv', delimiter=',', dtype=
    None, encoding="utf8")
87 s2019_stage2 = np.genfromtxt('phs_2019_stage_2.csv', delimiter=',', dtype=
    None, encoding="utf8")
88 s2019_stage3 = np.genfromtxt('phs_2019_stage_3.csv', delimiter=',', dtype=
    None, encoding="utf8")
89 s2019_stage4 = np.genfromtxt('phs_2019_stage_4.csv', delimiter=',', dtype=
    None, encoding="utf8")
90 s2019_playoffs = np.genfromtxt('phs_2019_playoffs.csv', delimiter=',',
    dtype=None, encoding="utf8")
91 maps = np.genfromtxt('match_map_stats.csv', delimiter=',', dtype=None,
    encoding="utf8")

```

8.2 Data Wrangling

8.2.1 Map Frequencies

```

1 # put empty dictionaries in the TeamInfo objects to count wins, losses,
    plays, and players
2 for i in range(len(teamrank)):

```

```

3     TeamInfo[teams[i]].append({}) # wins
4     TeamInfo[teams[i]].append({}) # losses
5     TeamInfo[teams[i]].append({}) # plays
6     TeamInfo[teams[i]].append({}) # stats
7
8 mapSet = set() # a set to account for ALL maps
9
10 # first let's get a list of all the maps and initialize each win, loss,
    and play
11 # dictionary accordingly (ie list all maps as keys, make all values 0)
12 for i in range(1, len(maps)):
13     mapSet.add(maps[i,8])
14
15 for i in range(len(TeamInfo)):
16     for cMap in mapSet:
17         entry = TeamInfo[teams[i]]
18         entry[1][cMap] = 0
19         entry[2][cMap] = 0
20         entry[3][cMap] = 0
21
22 # let's determine the frequency of wins and losses per map for each team
23 # does not factor in draws
24 for i in range(1, len(maps)):
25     winner = maps[i,6]
26     loser = maps[i,7]
27     if((winner == "draw") or (loser == "draw")):
28         continue
29
30     mapName = maps[i,8]
31     TeamInfo[winner][1][mapName] += 1
32     TeamInfo[loser][2][mapName] += 1
33
34
35 # this function will convert our sorted lists into dictionaries
36 # this is just to help us determine a team's most common maps in the
    weight function
37 def listToDict(sortedList):
38     d = {}
39     for i in range(len(sortedList)):
40         d[sortedList[i][0]] = sortedList[i][1]
41     return(d)
42
43
44 # now that we have dictionaries of win and loss frequencies,
45 # let's condense it all into the amount of times each team played at each
    map
46 # once again not factoring draws
47 # from here, we can update the weight function to factor maps played
48 # this also sorts our map dictionaries in a TeamInfo object
49 # from most commonly played map to least commonly played
50 for i in range(len(teamrank)):
51     for cMap in mapSet:
52         entry = TeamInfo[teams[i]] # for readability
53         entry[3][cMap] = entry[1][cMap] + entry[2][cMap]
54     tempSort = sorted(entry[1].items(), key=lambda x: x[1], reverse=True)
55     entry[1] = listToDict(tempSort)
56     tempSort = sorted(entry[2].items(), key=lambda x: x[1], reverse=True)
57     entry[2] = listToDict(tempSort)

```



```

58     tempSort = sorted(entry[3].items(), key=lambda x: x[1], reverse=True)
59     entry[3] = listToDict(tempSort)

```

8.2.2 Quantitative Player/Team Statistics

```

1  # pick out the stats we want to extract for weighing
2  statNames = ['All Damage Done', 'Assists', 'Deaths', 'Eliminations', '
    Healing Done',
3      'Hero Damage Done', 'Time Played', 'Ultimates Used', 'Weapon
    Accuracy',
4      'DPS', 'KDA']
5
6  # initialize each team's stats to all be 0
7  for i in range(len(teamrank)):
8      for j in range(len(statNames)):
9          entry = TeamInfo[teams[i]] # for readability
10         entry[4][statNames[j]] = 0
11
12  def readStats(filename):
13      for i in range(1, len(filename)):
14          team = filename[i][6]
15          # player = filename[i][5]
16          statName = filename[i][7]
17          # hero = filename[i][8]
18          statAmt = float(filename[i][9])
19          entry = TeamInfo[team]
20          if (statName in statNames):
21              entry[4][statName] += statAmt
22
23
24  readStats(s2019_stage1)
25  readStats(s2019_stage2)
26  readStats(s2019_stage3)
27  readStats(s2019_stage4)
28  readStats(s2019_playoffs)
29
30  # calculate other stats we want to use, such as DPS and KDA
31  # DPS will be Damage / Time Played
32  # (can't use Time Alive because Junkrat can do damage while dead)
33  # KDA is (K+A)/D
34  for i in range(len(teamrank)):
35      entry = TeamInfo[teams[i]]
36      damage = entry[4]['Hero Damage Done']
37      time = entry[4]['Time Played']
38      kills = entry[4]['Eliminations']
39      deaths = entry[4]['Deaths']
40      assists = entry[4]['Assists']
41      DPS = damage / time
42      KDA = (kills + assists) / deaths
43      entry[4]['DPS'] = DPS
44      entry[4]['KDA'] = KDA
45
46  # Need to normalize stats
47  def normalize(statName):
48      m = TeamInfo[teams[0]][4][statName]
49      for i in range(1, len(teams)):
50          n = TeamInfo[teams[i]][4][statName]

```

```

51     m = max(m, n)
52     for i in range(len(teams)):
53         TeamInfo[teams[i]][4][statName] /= m
54
55 # Normalize every stat we want to work with
56 for i in range(len(statNames)):
57     normalize(statNames[i])

```

8.3 Finding Edges

```

1 # find all the edges
2 # edges go between any match with at least one team in common
3 edge_list = []
4 for i in range(len(GameTuples)):
5     for j in range(i+1, len(GameTuples)):
6         m1t1 = GameTuples[i][0]
7         m2t1 = GameTuples[j][0]
8         m1t2 = GameTuples[i][2]
9         m2t2 = GameTuples[j][2]
10        m1 = GameTuples[i]
11        m2 = GameTuples[j]
12        if ((m1t1 == m2t1) or (m1t1 == m2t2) or (m1t2 == m2t1) or (m1t2 ==
m2t2)):
13            edge_list.append((m1, m2))

```

8.4 Derivative Matrix

```

1 # let's make a derivative matrix full of zeros first
2 Derivative = np.zeros((len(edge_list), len(GameTuples)), dtype = np.float64)
3
4 for i in range(len(Derivative)):
5     m1 = edge_list[i][0]
6     m2 = edge_list[i][1]
7     m1t1 = m1[0]
8     m1t2 = m1[2]
9     m2t1 = m2[0]
10    m2t2 = m2[2]
11    x = GameTuples.index(m1)
12    y = GameTuples.index(m2)
13    if ((m1t1 == m2t1) and (m1t2 == m2t2)):
14        cw = 25 # relatively large number
15        Derivative[i][x] = cw
16        Derivative[i][y] = -cw
17        continue
18    if ((m1t1 == m2t2) and (m1t2 == m2t1)):
19        cw = 25 # relatively large number
20        Derivative[i][x] = cw
21        Derivative[i][y] = cw
22        continue
23    if ((m1t1 == m2t1)):
24        cw = weight(TeamInfo[m1t2], TeamInfo[m2t2])
25        Derivative[i][x] = cw
26        Derivative[i][y] = -cw
27        continue
28    if ((m1t2 == m2t2)):

```

```

29         cw = weight(TeamInfo[m1t1], TeamInfo[m2t1])
30         Derivative[i][x] = cw
31         Derivative[i][y] = -cw
32         continue
33     if((m1t1 == m2t2)):
34         cw = weight(TeamInfo[m1t2], TeamInfo[m2t1])
35         Derivative[i][x] = cw
36         Derivative[i][y] = cw
37         continue
38     if((m1t2 == m2t1)):
39         cw = weight(TeamInfo[m1t1], TeamInfo[m2t2])
40         Derivative[i][x] = cw
41         Derivative[i][y] = cw
42         continue

```

8.5 Weight Functions

Instead of adding everything to the main weight function, we are testing each factor separately here. The outcomes from these tests help us determine how important each factor is in calculating weights.

8.5.1 Ranking

```

1 def weight(ti1, ti2):
2     distance = abs(ti1[0] - ti2[0])
3     weight = 1 / distance
4     return(weight)

```

8.5.2 Map Frequency

```

1 def weight(ti1, ti2):
2     plays1 = list(ti1[3].keys())
3     plays2 = list(ti2[3].keys())
4     wins1 = list(ti1[1].keys())
5     wins2 = list(ti2[1].keys())
6     losses1 = list(ti1[2].keys())
7     losses2 = list(ti2[2].keys())
8
9     mapFactor = 0.1;
10    # iterate over both lists and consider how many are or aren't in the
    same position
11    for i in range(len(plays1)):
12        if(plays1[i] != plays2[i] and wins1[i] != wins2[i] and losses1[i]
    != losses2[i]):
13            mapFactor += 0.1
14    weight = 1 / mapFactor
15    return(weight)

```

8.5.3 All Damage Done

```

1 def weight(ti1, ti2):
2     dmg1 = ti1[4]['All Damage Done']
3     dmg2 = ti2[4]['All Damage Done']
4     distance = abs(dmg1 - dmg2)
5     weight = 1 / distance
6     return(weight)

```

8.5.4 Hero Damage Done

```

1 def weight(ti1, ti2):
2     dmg1 = ti1[4]['Hero Damage Done']
3     dmg2 = ti2[4]['Hero Damage Done']
4     distance = abs(dmg1 - dmg2)
5     weight = 1 / distance
6     return(weight)

```

8.5.5 Healing Done

```

1 def weight(ti1, ti2):
2     dmg1 = ti1[4]['Healing Done']
3     dmg2 = ti2[4]['Healing Done']
4     distance = abs(dmg1 - dmg2)
5     weight = 1 / distance
6     return(weight)

```

8.5.6 DPS

```

1 def weight(ti1, ti2):
2     dmg1 = ti1[4]['DPS']
3     dmg2 = ti2[4]['DPS']
4     distance = abs(dmg1 - dmg2)
5     weight = 1 / distance
6     return(weight)

```

8.5.7 KDA

```

1 def weight(ti1, ti2):
2     dmg1 = ti1[4]['KDA']
3     dmg2 = ti2[4]['KDA']
4     distance = abs(dmg1 - dmg2)
5     weight = 1 / distance
6     return(weight)

```

8.5.8 Weapon Accuracy

```

1 def weight(ti1, ti2):
2     dmg1 = ti1[4]['Weapon Accuracy']
3     dmg2 = ti2[4]['Weapon Accuracy']
4     distance = abs(dmg1 - dmg2)
5     weight = 1 / distance
6     return(weight)

```

8.5.9 Ultimates Used

```

1 def weight(ti1, ti2):
2     dmg1 = ti1[4]['Ultimates Used']
3     dmg2 = ti2[4]['Ultimates Used']
4     distance = abs(dmg1 - dmg2)
5     if(distance == 0):
6         return(25)
7     weight = 1 / distance
8     return(weight)

```

8.5.10 Factoring Team Ranking, Maps Played

```

1 # structure of a TeamInfo entry:
2 # 'name': [rank, {wins}, {losses}, {plays}]
3 def weight(ti1,ti2):
4     # first, consider ranking
5     distance = abs(ti1[0]-ti2[0])
6
7     # then, frequently played maps
8     plays1 = list(ti1[3].keys())
9     plays2 = list(ti2[3].keys())
10    simCount = 0
11    # iterate over both lists and count how many are in the same position
12    for i in range(len(plays1)):
13        if(plays1[i] == plays2[i]):
14            simCount += 1
15    # or one off
16    for i in range(1, len(plays1)):
17        if(plays1[i] == plays2[i-1]):
18            simCount += 0.5
19    for i in range(0, len(plays1) - 1):
20        if(plays1[i] == plays2[i+1]):
21            simCount += 0.5
22    distance += ((12 - simCount) / 12)
23    weight = 1 / distance
24    return(weight)

```

Test 1:

71.25 % correctly predicted winner

57 / 80 correct guesses

Test 2:

68.00 % correctly predicted winner

34 / 50 correct guesses

Test 3:

73.33 % correctly predicted winner

22 / 30 correct guesses

Test 4:

70.00 % correctly predicted winner

7 / 10 correct guesses

This iteration of the weight function performs worse overall than using rankings alone. Its results are consistently around 70%.

8.5.11 Making Maps a Multiplicative Factor

```
1 # structure of a TeamInfo entry:
2 # 'name': [rank, {wins}, {losses}, {plays}]
3 def weight(ti1,ti2):
4     # first, consider ranking
5     distance = abs(ti1[0]-ti2[0])
6
7     # then, frequently played maps
8     plays1 = list(ti1[3].keys())
9     plays2 = list(ti2[3].keys())
10    wins1 = list(ti1[1].keys())
11    wins2 = list(ti2[1].keys())
12    losses1 = list(ti1[2].keys())
13    losses2 = list(ti2[2].keys())
14
15    mapFactor = 0.1;
16    # iterate over both lists and consider how many are or aren't in the
17    # same position
18    for i in range(len(plays1)):
19        if(plays1[i] != plays2[i] and wins1[i] != wins2[i] and losses1[i]
20        != losses2[i]):
21            mapFactor += 0.1
22    weight = 1 / (distance * mapFactor)
23    return(weight)
```

Test 1:

75.00 % correctly predicted winner

60 / 80 correct guesses

Test 2:

76.00 % correctly predicted winner

38 / 50 correct guesses

Test 3:

80.00 % correctly predicted winner

24 / 30 correct guesses

Test 4:

80.00 % correctly predicted winner

8 / 10 correct guesses

This iteration of the weight function performs slightly better than the one before it, predicting between 75% and 80% of matches correctly.

8.5.12 Using Ranking, KDA, and (Multiplicative) Maps

```
1 def weight(ti1,ti2):
2     distance = abs(ti1[0]-ti2[0])
3
4     # then, frequently played maps
5     plays1 = list(ti1[3].keys())
6     plays2 = list(ti2[3].keys())
7     wins1 = list(ti1[1].keys())
8     wins2 = list(ti2[1].keys())
9     losses1 = list(ti1[2].keys())
10    losses2 = list(ti2[2].keys())
11
12    k1 = ti1[4]['KDA']
13    k2 = ti2[4]['KDA']
14    distance += abs(k1 - k2)
15
16    mapFactor = 0.1;
17    # iterate over both lists and consider how many are or aren't in the
18    # same position
19    for i in range(len(plays1)):
20        if(plays1[i] != plays2[i] and wins1[i] != wins2[i] and losses1[i]
21        != losses2[i]):
22            mapFactor += 0.1
23    weight = 1 / (distance * mapFactor)
24    return(weight)
```

Test 1:

73.75 % correctly predicted winner

59 / 80 correct guesses

Test 2:

70.00 % correctly predicted winner

35 / 50 correct guesses

Test 3:

70.00 % correctly predicted winner

21 / 30 correct guesses

Test 4:

80.00 % correctly predicted winner

8 / 10 correct guesses

This version performed slightly worse, potentially due to the extra additive factor.

8.5.13 Adding Multipliers to Ranking and KDA

```
1 def weight(ti1,ti2):
2     # this version combines factors from the other versions
3     distance = abs(ti1[0]-ti2[0])
4
5     # then, frequently played maps
6     plays1 = list(ti1[3].keys())
7     plays2 = list(ti2[3].keys())
8     wins1 = list(ti1[1].keys())
9     wins2 = list(ti2[1].keys())
10    losses1 = list(ti1[2].keys())
11    losses2 = list(ti2[2].keys())
12
13    k1 = ti1[4]['KDA']
14    k2 = ti2[4]['KDA']
15    kdafac = abs(k1 - k2)
16
17    distance = distance * .4 + kdafac * .6
18
19    mapFactor = 0.1;
20    # iterate over both lists and consider how many are or aren't in the
    same position
21    for i in range(len(plays1)):
22        if(plays1[i] != plays2[i] and wins1[i] != wins2[i] and losses1[i]
    != losses2[i]):
23            mapFactor += 0.1
24    weight = 1 / (distance * mapFactor)
25    return(weight)
```

Test 1:

73.75 % correctly predicted winner

59 / 80 correct guesses

Test 2:

76.00 % correctly predicted winner

38 / 50 correct guesses

Test 3:

76.67 % correctly predicted winner

23 / 30 correct guesses

Test 4:

90.00 % correctly predicted winner

9 / 10 correct guesses

By multiplying our additive factors by some percent (in this case 40% ranking, 60% KDA), we were able to have slightly better predictions than when we just added KDA and ranking before multiplying by the map factor.

8.5.14 Adding Healing Factor

```
1 def weight(ti1,ti2):
2     # this version combines factors from the other versions
3     distance = abs(ti1[0]-ti2[0])
4
5     # then, frequently played maps
6     plays1 = list(ti1[3].keys())
7     plays2 = list(ti2[3].keys())
8     wins1 = list(ti1[1].keys())
9     wins2 = list(ti2[1].keys())
10    losses1 = list(ti1[2].keys())
11    losses2 = list(ti2[2].keys())
12
13    k1 = ti1[4]['KDA']
14    k2 = ti2[4]['KDA']
15    kdafac = abs(k1 - k2)
16
17    h1 = ti1[4]['Healing Done']
18    h2 = ti2[4]['Healing Done']
19    hfac = abs(h1 - h2)
20
21    distance = distance * .4 + kdafac * .3 + hfac * .3
22
23    mapFactor = 0.1;
24    # iterate over both lists and consider how many are or aren't in the
    same position
25    for i in range(len(plays1)):
26        if(plays1[i] != plays2[i] and wins1[i] != wins2[i] and losses1[i]
    != losses2[i]):
27            mapFactor += 0.1
28    weight = 1 / (distance * mapFactor)
29    return(weight)
```

Test 1:

75.00 % correctly predicted winner

60 / 80 correct guesses

Test 2:

80.00 % correctly predicted winner

40 / 50 correct guesses

Test 3:

80.00 % correctly predicted winner

24 / 30 correct guesses

Test 4:

80.00 % correctly predicted winner

8 / 10 correct guesses

8.6 Making Predictions

```
1 # begin making predictions
2 sd = np.array([GameTuples[i][1]-GameTuples[i][3] for i in range(len(
    GameTuples))])
3
4 N = 200 # suppose first N games known, predict last games
5 prediction = sp.linalg.lstsq(Derivative[:,N:], - Derivative[:,N:]@sd[:N] )
6 predicted = list(prediction[0])
7 actual = sd[N:]
8 pSize = len(predicted)
9
10 # determine how many times we predicted the winner correctly
11 correct = 0
12 for i in range(pSize):
13     if((predicted[i] < 0 and actual[i] < 0) or (predicted[i] > 0 and
        actual[i] > 0)):
14         correct += 1
15 print((correct / pSize * 100), "% correctly predicted winner \\\\")
16 print(correct, "/", pSize, "correct predictions")
```