CSC111 Project: An Application of Graph: the Prediction of Soccer Match (La Liga) Based on the Past Match Results

Teddy Wang

Friday, Apr 16, 2021

Problem Description and Research Question

Soccer, which is also known as football or association football, is a team sport of 22 players (11 in each team) and one sphere ball. It is the most popular sport in the world with millions of players in more than 200 countries. Generally, the FIFA World Cup is the most valuable and reputed title for national teams. For clubs, the UEFA Champions League and FIFA Club World Cup often represent the best-performed club that year.

Each soccer tournament and league consist of many matches. Despite that different competitions might have different rules, the essence of them is to win as many matches as possible. Therefore, people are keen on predicting the outcome of matches. To analyze the upcoming matches and give an accurate prediction, soccer fans utilize various factors, such as the team roaster, the best player in the team, home/ away ground difference, etc. (Ibenegbu) The match history, which records the past match results among teams, represents the performance of teams when they faced each other in the past. People believe that the past winners will still win in most cases, and therefore this factor has been seen as one of the most important factors in predicting results. After collected the background information, the project is inspired to find out the influence of past games on future fixtures in soccer.

The match results of La Liga are used during the research. La Liga, established in 1928 and held by the Royal Spanish Football Federation, is the division one soccer league in Spain (RFEF). It is one of the top leagues in the world, where the clubs that participated in La Liga have won the most UEFA Champions League titles and FIFA club world cup titles. This particular football league is chosen because of three reasons: firstly, La Liga is one of the top leagues in the soccer world with well-maintained data systems and is therefore relatively easy to retrieve data; secondly, the long history of La Liga enables the project to get a large amount of data; thirdly, the author of this project is a fan of Barcelona, a club in La Liga. This increases the familiarity of the information in the dataset and makes the participator more motivated to do the project.

Taking La Liga's results into research, the goal of the project is **to create a predictor of soccer matches by using the graph object**. The discoveries during the development of the project and the final result will be used to answer the research question: **to what extent do the past results affect future matches in La Liga**?

Data set

The dataset that this project is going to use is retrieved from Kaggle. It is created by Kishan Kumar, a user of Kaggle, and updated on July 24, 2020. It is a **.csv** file that contains seasons, dates, the teams, half-time scores, half-time winners, full-time scores, and full-time winners in each row. All the matches in the dataset are sorted in chronological order. The following is an example of how the dataset looks like:

Season	Date	HomeTeam	AwayTeam	FTHG	FTAG	FTR	HTHG	HTAG	HTR
1995-96	2/9/1995	La Coruna	Valencia	3	0	H	2	0	H
1995-96	2/9/1995	Sp Gijon	Albacete	3	0	H	3	0	H
1995-96	3/9/1995	Ath Bilbao	Santander	4	0	H	2	0	H
1995-96	3/9/1995	Ath Madrid	Sociedad	4	1	H	1	1	D

Figure 1: a sample of the original dataset

Computational Overview

The computation of this project is separated into 3 parts: data loading, graph and predictor. To summarize, this project is a data-centric match predictor which uses graph as its essential part for data processing.

The first part is implemented in csc111project_data_loading.py. This module reads the data which has the shape with the above-mentioned data set and returns a list of matches. There will be only 4 columns after processing, each of them are: **season count**(now it starts from 0 rather than a value of year such as '1995-96' in the original data set), **home team name** (a label of 'home' is added to it for distinguishing it with the away teams), **away team name** (similarly, a label of 'away' is added), and **full-time result** (it is merged from the original full-time home goals and full-time away goals. It represents the difference between them, e.g. a game with score 1-3 is represented as -2). The following is an example of how the returned data will look like:

```
[[0, 'La Coruna home', 'Valencia away', 3],
 [0, 'Sp Gijon home', 'Albacete away', 3],
 [0, 'Ath Bilbao home', 'Santander away', 4],
 [0, 'Ath Madrid home', 'Sociedad away', 3],
 [0, 'Celta home', 'Compostela away', -1],
 ... ]
```

The second part is the essence of the project, which constructs the graph by the data. It is implemented in csc111project_graph.py. The graph is called **MatchGraph** and the node is called **_Team**. It includes 3 private instance attributes: **self.item, self.matches and self.past_matches**. **self.item** stores the name of the team; **self.matches** stores all the matches it played; **self.past_matches** stores the recent 10 matches it played. The MatchGraph object also includes 3 private instance attributes: **self._Teams** is the set of all the _Team nodes it includes; **self._all_matches** is a dict mapping a tuple (Home team, Away team) to the weight of the edge (itself represents the set of the edges as well); **self._seasonal_weight** is a fixed value represents the importance of the past matches. It is created with **graph.__init__** where the user gives this value to the graph.

The MatchGraph receives the data row by row. When a new row of data enters the graph, it first checks if the two nodes both exist. If any of them is missing, it creates the node for the team by calling the **add_team** method. Then, it checks if there is an edge between them. If not, it creates one by calling the **add_edge** method and assign the match score to the weight of the edge. If there is an edge, it calls the **update_edge** method to update the value of the edge. It simply adds the new value to the existing value of the edge. After all, this row of match data is updated to the **past_matches** of each team. A method called **update_season** will be used when there is a change of season. In this method, all the weights of the edges are going to multiply **self._seasonal_weight**. This method decreases the resources that the graph needs as it only needs to call this method in the predictor class when the season changes (which I will explain in the next paragraph) rather than always keeps an eye on the current season and the input season.

The edge of the graph is presented as a power series. In La Liga, the home team plays with the certain away team once a season. Then, their edges will only be changed once a season. Let there be n seasons and the score between teams A and B in each season be S_n. Let W represents the **_seasonal_weight** attribute of the graph. Then, the value of the edge E_n in each season is calculated as

$$E_n = S_n + S_{n-1}W + S_{n-2}W^2 + ... + S_1W^{n-1} \tag{1}$$

$$= \sum_{i=1}^{n} S_i W^{n-i} \tag{2}$$

From (2) we have the representation of the value of the edges by the sigma notation. However, in fact the idea of (1) is used in the actual computation. When there is no edges, we give it an edge and assign S_1 to it. This is E_1. When every match in the season is fed to the graph, we trigger the **update_season** method, which make every value multiply W and make the new value be S_1W. Then in the new season we add S_2 to the existing value when we call **update_edge** and we have $S_2 + S_1W$ which is exactly E_2. Repeating this steps, we can have the value of all the edges in any seasons: $E_1, E_2, ..., E_n$. It is done cumulatively but it is more efficient than calculating the sigma each time because that requires the graph to memorize all the past scores which increases the computing time.

The third part is the Predictor classes. They use the graph, feed the data into it, use the weight of the edge

to predict and calculate the accuracy. They also receive the data row by row. In each row, it first compares the season count of the input with the private attribute **_curr_season**. If they are different, it considers that there is a change of season in the data set and calls **graph.update_season**. Then, using the input teams, it makes a prediction first. Then, it sends the row to the graph by calling **graph.add_data**. This will trigger the whole process mentioned in the previous paragraphs. Then, it calculates the accuracy between the predicted result and the actual result from the data set. The accuracy is calculated differently. If the prediction is the same as the actual result, then the accuracy is 1 (100%). In other cases, if both of the prediction and the actual results are greater than 2 or less than -2, We consider the accuracy as 0.75 as it correctly predicted the winner. If the actual result is a draw (actu $= 0$) and the prediction is 1 or -1, we give it 0.5 as when the performance of the two teams are so close to each other it is indeed hard to predict the score. In all the other cases, the prediction will receive a 0. The average accuracy will be calculated each season and an attribute called **seasonal_accuracy** is created by those accuracies in one season.

There are 4 Predictor classes in csc111project_predictor.py. They are Predictor (the original one), active Predictor, random Predictor, and normal Predictor. To keep the clarity of the codes, the latter 3 predictors are designed to be the children classes of the Predictor class. They differ in the ways of making predictions. The Predictor and the active predictor class use the edge of the graph to make predictions. When the Predictor is initialized, the seasonal_weight of the graph is assigned by the user. It is then passed to the graph. The edges are calculated based on this. If the weight of the edge is valid, they round the value and return it; if it is not valid, they compare the recent performance of the two teams by finding the average of the **past_matches** in each Team node and calculate the difference. This value is then rounded and returned. If one of them does not have enough past matches to evaluate, they make random guesses between -1 and 1 as this is the most common case. The difference between these two predictors is that they use different ways to round the score. The Predictor class uses fixed boundaries to separate the scores into {-3, -2, -1, 0, 1, 2, 3}, while the active Predictor uses dynamic boundaries. The boundaries of the active Predictor are calculated by the average of the edge weights of the previous inputs. The random Predictor makes random guesses between -3 to 3; the normal predictor uses a normal distribution with mean 0 and standard deviation of 3 to give predictions. It is usually better than the random Predictor because there are scores between -1 and 1 than other scores. The normal Predictor is implemented with the help of **numpy.random.normal** function, which constructs a normal distribution and gives random values from it.

Main.py is the place that all the components are utilized and all the results are displayed. The data is retrieved from csc111project_data_loading.py first. Then, since the weight of the graph needs to be provided with the initialization, this project is going to discover the performance of them in different _seasonal_weight values. Therefore, for each case from 0.1, 0.2 to 0.9, 1.0, all the four predictors are initialized and the data are fed to them respectively. Then, from the **seasonal_accuracy** method we can have their accuracy history by season. The accuracy of the Predictor is divided by the accuracy of random and normal predictor. The average of the two ratios is recorded. The same procedure is also done for the active Predictor (The reason for doing such things is explained in the Discussion Section). Therefore, now we have a list of Predictor and active Predictor's performance with different seasonal_weights.

The data is displayed in bar charts with the help of **numpy** and **matplotlib**. Since we have a list of performances of the predictor and active predictor, we use them to make bar charts. The outcome is double-sided bar charts whose x-axis is the values of seasonal_weight and the bar values are the performance score of Predictor and the active Predictor. The performance of the predictor will be displayed in the top half of the graph and the active Predictor will be displayed in the bottom half by calling **matplotlib.bar** and labeled by calling **matplotlib.text**. They will be showed in the Discussion section. The above-mentioned two libraries are also the two external libraries used by the project. **Numpy** provides more options and more flexibility to manipulate lists and gives a very simple, easy solution for making a normal distribution. **Matplotlib** is one of the most popular graph-drawing libraries and fits with numpy very well.

Instruction for Running

1. Install all the external libraries listed under requirement.txt.

2. Download the data set from www.kaggle.com/kishan305/la-liga-results-19952020. The same item is also sent to csc111-2021-01@cs.toronto.edu by https://send.utoronto.ca. It can be picked up by using Claim ID: oKKu2qKE4HQdMsm3 and Claim Passcode: yEsvrnPcYjxfMd3r.

3. Download all the python modules from MarkUs. Put all the data and python documents in the same folder. If everything is done properly, the folder will look like this (ignore the "Paperworks" folder):

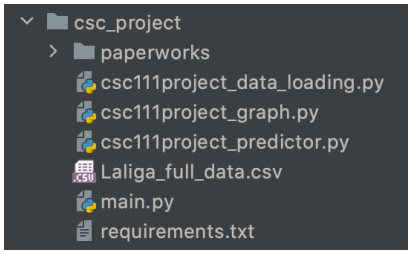

Figure 2: Project Overview in Pycharm

4. If you do differently, change the file path in main.py. **str** in line 48 refers to the path of **Laliga_full_data.csv**. It should also be changed if you renamed it.

5. Run main.py. Note that Line 66 and 67 give different ways to get the seasonal_accuracy result. You can change between them by simply commenting and uncommenting each other and observe the change of results.

Changes After Proposal

There are not too many changes after the proposal is submitted and the feedback is received. The only significant change is the display of outcome. In the original plan, it is displayed directly. But in the new plan, the performance of the two predictors is evaluated by their relative performance with the two random predictors.

Discussion Section

The project was intended to show the absolute performance of the predictor and the active predictor. However, the result seems not very optimistic. Therefore, in the end, the plan was changed. Two more predictor classes are introduced so we can compare the performance of the original predictors with those random predictors to see how much the project had done to improve the accuracy of prediction from random guess. The new results will be displayed below.

Displaying Result

There are two ways to get the performance result, but all of them depend on the **seasonal_accuracy** attributes in the predictors. In the first one, the average of seasonal accuracy is used for discussion, and in the second one, the accuracy value of the final season is used.

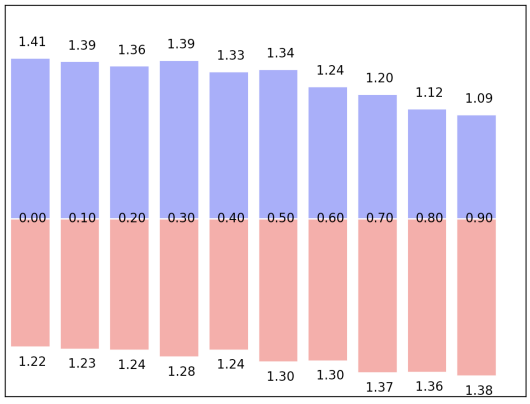

Figure 3: The relative accuracy of the Predictor and the active Predictor (version 1)

The graph above is the one that uses the average seasonal accuracy for evaluation. The one in blue represents the

performance of the Predictor (original) and the one in red represents the performance of the active Predictor. The numbers that appear on the graph are the average relative advantages of the predictor to random guesses. For example, 1.30 means that the predictor performs 30% better on average than the random predictor and the normal predictor. The active Predictor has better performance in higher seasonal_weight values while the original Predictor performed better when the value is low. There is a decreasing trend in Predictor when the seasonal_weight is increasing, while active Predictor performs almost equally well in all conditions. The trend is displayed even clearer in the next graph, where the last season's prediction accuracy is used for evaluation.

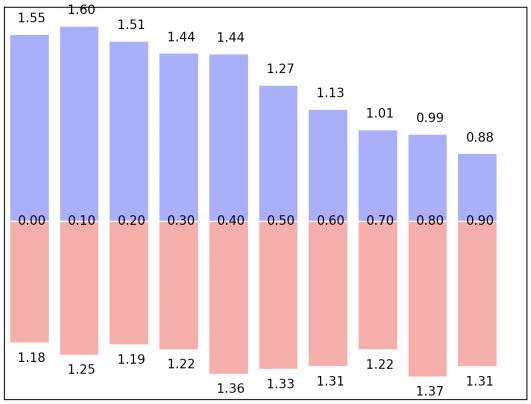

Figure 4: The relative accuracy of the Predictor and the active Predictor (version 2)

Clearly, the original Predictor's performance reaches its maximum when the weight value is 0.10, and then it decreases drastically. When the season weights more than 0.8, the performance of the original predictor drops to a place that even the random guesses can give higher accuracy prediction than it. The active Predictor maintains stable around 1.2 to 1.3, which means that it is constantly 20%-30% better than the normal predictor and the random predictor.

Answering the research question, although there is some progress and the project shows some extent of dominance over random guesses, one should not say that it can predict the football result accurately as the best prediction accuracy this project can give to a match is around 40%, which apparently does not satisfy people especially the football fans. There might be two parts of reasons for that: first, football is a sport that involves various factors, thus using such a simple way to predict football cannot reach high accuracy; second, graph, as a relatively simple data type, is not able to handle a complicated model for prediction. The merit and flaws of the project are discussed below:

Merit and significance

The significance of this project is that it allows the developer (or in other words, me) to connect the knowledge he learned in different courses, such as the graph object learned in computer science and the power series in mathematics, with some real-life topics. It is also a great experience that allows him to do modeling by himself and reached a relatively satisfactory outcome. Also, the project is planned and implemented systematically with lots of plans and deadlines, which increases the efficiency of working. This is one of the soft skills one can learn and apply in the future. Moreover, this project proved the unpredictability of football results, or at least by graph or some easier techniques. As a sport that has hundreds of years of history, it should be unpredictable to make it thrilling and engaging.

Flaw and improvement

There are not too many flaws in the implementation of this project. Apart from the modeling, another possible improvement in the project to increase efficiency would be the running time. The main.py runs the data set which has nearly 10,000 rows 40 times (10 iterations for all four types of predictors). Moreover, the OOP part in csc111project_predictor can be simplified as well. Currently, the random predictor and the normal predictor are the children classes of the Predictor object, however, they did not even use any of the data from the data set. The feed_data method is not overwritten, which means that they still have to go through the data and add them to the graph, despite they do not use them. Simplify them can greatly decrease the running time of the main part, thus makes the project more efficient.

Further Research Suggestions

This project can be extended further in three ways.

First, the model can be changed to a more efficient one. The weighted graph is sufficient for a first-year student to discover but is not enough if the project is going to be used further. Therefore, the model and the calculation of the weight of the edges can be updated and replaced by some better models and ideas.

Second, the data set can be processed in a better way. the current data set includes a lot of noises. The outcome of the project will improve if some of the noises can be canceled by some techniques when the data set is read.

Third, football is a complicated sport with a lot of factors involved. Therefore, focusing on simply the full-time score should be not enough to predict the future result. Hence, it is suggested that more factors related to football can be considered and used to build a better model for the project.

References

1. Homeyer, Brad. "How Champions League Scoring Works (Away Goals Explained)." The18, 17 Apr. 2019, the18.com/soccer-entertainment/how-champions-league-scoring-works.

2. Ibenegbu, George. "How to Predict Football Match Results? - 7 Tips to Increase Your Chances." Legit.ng - Nigeria News., 28 July 2020, www.legit.ng/1121607-useful-tips-predict-football-matches-accurately.html.

3. Kumar, Kishan. "La Liga (1995 - 2020)." Kaggle, 24 July 2020, www.kaggle.com/kishan305/la-liga-results-19952020.

4. "Statistics LaLiga Santander 2020/21." LaLiga, www.laliga.com/en-ES/stats.

5. "Visualization with Python." Matplotlib, matplotlib.org/.

6. "Numpy", https://numpy.org/.

7. Rfef.es, www.rfef.es/competiciones/futbol-masculino/calendario/2/0.