TeamRank

Kahaan Shah & Rudransh Mukherjee

April 29, 2022

Implementing a modifying version of google's PageRank algorithm to rank the performance of all the teams that have played in the English Premier League from 1993 to 2018

Contents

The Problem

Football is fun. But sometimes, being a football fan can be a bit harmful for your blood pressure. The moment emotions rise, conversations lose any semblance of objectivity and all that's left is a shouting match where the one with the more decibels wins. We are no different. We've wasted countless hours on the internet fighting with anonymous people over football and the likes. In one such argument, we realised that in all of our conversations, we lacked a way to objectively compare the historical performances of our favourite teams. Sure there was a points table, but that gave the same weightage to a 1-0 win against a weak team as a 3-0 win against a top one. And that's when our eyes lit up. We had found a problem we wanted to solve. Ranking the performance of all the teams that have played in the English Premier League's history to find out who has truly been the most dominant team in the history of the competition.

Our Solution

When we thought of rating all teams, our first thought was referring to the points table of various seasons and after some deliberation, we realised that the cumulative points table might not be the best representation of the calibre of a team because not all teams have the same facilities and the resources at the same disposal. Especially teams that have been promoted from the lower divisions to the premier league. Hence, if in one season, a newly relegated won just 10 matches, but defeated the table-toppers while they were at it, those wins should be given more value than defeating another team that was also newly promoted.

Being chess fans as well as football fans, our natural instinct was merging the two sports and coming up with something along the lines of ELO Ratings, but for football teams (our code for ELO can be found in the appendix). After implementing the ELO system (in Appendix), we identified a few shortcomings with the same method:

a) There was a lot of weightage that we had to determine ourselves since ELO relies on some constant factors to give points to teams.

b) Since ELO deducted points for losing matches, we were unable to reward a team for simply staying up in the fiercely competitive premier league and losing as opposed to staying for a brief amount of time. For example: The ELO rating system was giving a higher rating to a team such as Charlton which played in the premier league for just a few years than Everton, who have been in the league for all years by virtue of the fact that Everton lost more matches in the Premier League then Charlton (who spent more time in the Lower divisions than in the premier league.)

After discarding ELO as a possible way of ranking teams, we then stumbled onto PageRank.

What is PageRank?
Named after the co-founder of Google, Lawrence "Larry" Page, it was first published in this paper. Essentially, it is a way of assigning ranks to web-pages in order to sequence them when showing the results of the searches in a search engine.

How does it work?
To understand how PageRank works, let's take a simplified version of the internet where there exist just 4 webpages: A,B,C and D (Figure 1).

Since an Arrow points from A to B, that means that A contains a link that takes the user to B. Initially, all pages will get a score of $\frac{1}{4}$ (Divide 1 by the number of webpages in the system) Now, the algorithm will start assigning scores to the pages. In the first iteration, Page A will get a score of $\frac{\frac{1}{4}}{3}$ from C since C has to divide a score of $\frac{1}{4}$ amongst the 3 links that it is linked to. Page B will

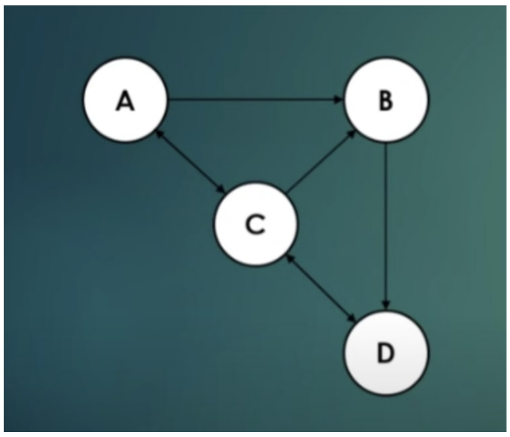
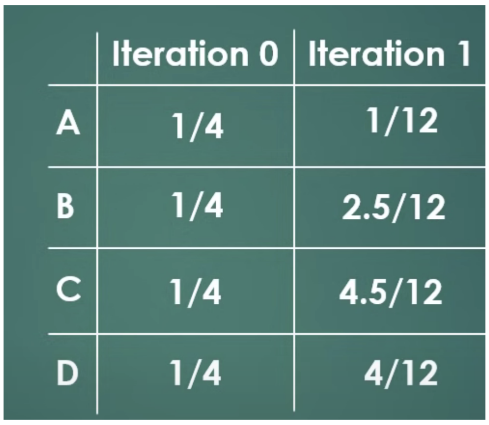


Figure 1: The four web-pages A,B,C and D where the arrows indicate links between the pages.



	Iteration 0	Iteration 1
A	1/4	1/12
B	1/4	2.5/12
C	1/4	4.5/12
D	1/4	4/12

Figure 2: The PageRank values of the pages after one iteration.

get a score of $\frac{\frac{1}{4}}{2} + \frac{\frac{1}{4}}{3}$ (since two pages link to B). And So on and so forth. After the first Iteration, the PageRank values of the pages will look like this:
Now, for the second iteration, the new values of the PageRank will be calculated by taking the values we got after the first iteration and using the same formula. For example the value of A will be $\frac{\frac{4.5}{12}}{3}$ and so on. We will keep iterating the values until they converge towards a constant set of ranks.

PageRank Score: The PageRank score that one page can give to the other is determined by the number of links between the pages/divided by the total number of links the page has with all pages.

Note: The PageRank value of a page ultimately computes the probability that a random internet user will end up on the page or not. This means that there always exists the possibility that the user just stops surfing the internet at any given time. (Maybe they just got bored, faced a power cut or any other reason). Hence, there also needs to be a damping factor that accounts for the probability of the user disconnecting from the internet. Our explanation did not account for the damping factor since it is not relevant to the problem that we are trying to solve.

Algorithm 1 – Iterative Algorithm

For the implementation we used this dataset.
Our initial implementation of PageRank was similar to the Algorithm explained above. We first cleaned the data (appendix) and then created three final python dictionaries:

1. OriginalValues: Where the key are all the team names and the values are initially 1/49 (1 vote divided equally amongst the number of teams)
2. NewValues: Where the keys are all the team names and the values are initially 0. This dictionary will store the values of the ranks of all teams after each computation.
3. recordOfLosses: a dictionary which stores the details of all the teams that have won against a certain team in the history of the league while also storing the relevant goal difference. So, the dictionary recordOfLosses will have 49 keys in accordance to the team names and the value of each key will be another dictionary which stores the details of the teams that the team in the key lost to. For example, The first key in recordOfLosses is Arsenal and the value of the key Arsenal is another dictionary which we can call dictionary B. Dictionary B has keys as the list of teams Arsenal has lost to and values as the total number of goals they have lost to that team by. So recordOfLosses[Arsenal][Liverpool] gives the total number of goals by which Arsenal has lost to Liverpool over all their matches. It also has a Total key that counts the total goals Arsenal has lost by.

Below is the pseudocode for the algorithm which is provided to illustrate exactly how it works:

1. Iterate through all the items in list teams(eg: Arsenal, Coventry, Chelsea etc). For each iteration j=recordOfLosses[team].
2. Iterate through each key in the dictionary returned (j). For r in 0 to length of the dictionary, the value of the rth key of the dictionary originalValue is newValue[team r] = newValue[team r]+originalValue[team i] (team i from the first loop) * j[team r]/j[total](refer above for the definition of a pageRank score.)
3. Calculate the average difference between newValues and originalValue
4. Set originalValue = newValues, newValues = 0
5. Stop when difference is less than the specified threshold

Here is the code implemented in Python:

```
i,difference = 0,1
#INV-I: difference>10^-20, i-1 iterations of pagerank completed

while(difference>math.pow(10,-20)):
    difference=0
    #INV-II: for 0<=i<=len(teams)-1, team=teams[i].
    #Assert: Teams is the listof teams, dictionaries as described above

    for team in teams:
        j = recordOfLosses[team]

        #INV-III: 0<=r<=len(dict.keys())-1, k=dict.keys()[r],
        #newValues[k]=originalValues[team]*j[k]/j["Total"] for first r-1 teams

        for k in recordOfLosses[team].keys():
            if(k=="Total"):
                continue
            newValues[k]=(newValues[k]+originalValues[team]*(j[k])/j["Total"])

        #Assert: newValues contains the updated pagerank
        #scores for all scores team i had to give out

    #Assert: newValues updated for all teams for all wins

    #INV-IV: 0<=i<=len(teams)-1, team=teams[i]
    #originalValues for first i-1 teams is set to the newValue
    #difference is the sum of difference for first i-1 teams
    #newValues for first i-1 teams is 0

    for team in teams:
        difference=difference+abs(originalValues[team]-newValues[team])
        originalValues[team] = newValues[team]
        newValues[team] = 0

    #Assert: originalValues=newValues, difference=sum of changes in scores,
    #newValues=0

    difference=difference/len(teams) #To get avg difference
    print(i,difference) #to check convergence
    i=i+1 #ticker to index the results

    #Assert: originalValues pagerank scores updated for all teams

#Assert: originalValues is the final pagerank scores
```

Now we will prove the invariants to show the algorithm is correct:

I. This invariant is trivial. At the beginning difference= $1 \geq 10^{-20}$, i=0. During execution difference is updated each time and each iteration is exactly 1 more iteration of pagerank so i=i+1 until at termination difference$\leq 10^{-20}$ and i iterations of pagerank are completed.

II. This invariant is handled by the for loop function. The for loop iterates over every team in the list teams thus the invariant holds and at the i'th iteration team = teams[i], where i automatically=i+1.

III. This invariant is where newValue scores are updated for the score the current team i has to give out. The first part of the invariant is handled by the for loop as above. So r is incremented automatically and in each successive iteration k is the r'th value in the list of keys. Within the loop the second half of the invariant is maintained where each successive value for team k is updated in newValues (as discussed in the PageRank section).

IV. This invariant is also trivial. At the beginning i=0, difference=0 and originalValues and newValues have not changed (first 0 values exchanged). During execution i=i+! by the for loop, difference = difference+difference in ith teams, ith values updated in originalValues and ith newValues updated to 0. Thus the invariant is maintained during execution. Upon termination i=len(teams)-1, difference has the sum of each successive i'th value and originalValues is fully updated and newValues=0 for all values.

The time complexity of this algorithm is $O(n^3)$ since we traverse the list of n teams in the outer loop once, we traverse a worst case $(n-1)$ teams in the inner loop once and we again traverse the list of teams of size n to update the values of originalValues. The worst case complexity is $iO(n^3)$ depending on how fast the values converge, but this would change from value to value. The space complexity is $O(1)$ since no calculations are deferred. Overall the algorithm converged after 41 iterations.

Algorithm 2 – Eigenvector Approximation

Having created an algorithm in time complexity $O(n^3)$ we realised we needed a faster algorithm if this had to be viable for other such problems as well, not just specific to ours. We also realised that this would need to be achieved in a different way than just making more efficient loops because no matter what we had to iterate over the three loops in our initial algorithm. Thus we decided to implement a more efficient version of PageRank based on graph theory, linear algebra and Markov chains.

Essentially the graph's we drew for PageRank were Markov chains, which are directed graphs that tell us the probability of one event happening after another. Although (unlike the original PageRank algorithm) we weren't dealing with probability, our data could still be represented in a directed graph that had all the characteristics of a Markov chain. This graph could further be represented in a matrix called an adjacency matrix. Here each entry (i, j) in the matrix tells us proportional weight from team j to be given to team i. Since each team started off with a value of 1, the entries in the columns of the matrix would add up to exactly 1.

A further result of this is that this adjacency matrix is a stochastic matrix, i.e. a matrix where all the entries in the columns add up to 1. A property of this matrix is that it has a steady state vector q such that $Pq = q$ where P is the adjacency matrix. q would also be the principal eigenvector of the matrix and could thus be found using the power method for calculating eigenvectors. This gives us a recurrence relation of the form $q_k = Pq_{k-1}$ where q_0 is some random possible outcome of the system. This principal eigenvector q is exactly what we are looking for, since it is the final stable outcome of the system and thus our TeamRanks. So finally, we simply need to code a simple iterative function that fulfils the recurrence relation above that stops when the change in the eigenvector from one iteration to another is less than a specified value.

```
def teamrank(matrix, stopval):
#Assert: matrix is a schocastic matrix, stopval is a float
diff=stopval+1
i=1
n = matrix.shape[1]
eigenv = np.random.rand(n,1) #Generating a random vector with n entries
#Here we normalise it to make sure the entries add up to 1 i.e. it is a
#possible result of the system
eigenv = eigenv/np.linalg.norm(eigenv,1)
#INV: i>=1, diff>stopval, eigenv_i = matrix x eigenv_i-1
while diff>stopval:
    oldv = eigenv
    eigenv = matrix @ eigenv #numpy inbuilt matrix multiplication
    operator is ’@’
    changev = oldv-eigenv
    #Again using numpy inbuilt operations to calculate the average
    #change between 2 iterations
    diff = np.mean(np.absolute(changev))
    print(i, diff) #Printing to see how fast it converges
    i=i+1
#Assert: diff<= stopval, eigenv = eigenv_i
return eigenv
```

The invariant for this is that $diff > stopval$ and $eigenv_i = matrix \times eigenv_{i-1}$ for $i \geq 1$. That invariant is maintained before the loop since $diff = stopval + 1 > stopval$, $i = 1$ and $eigenv_0$ is initialized. During the loop the invariant is maintained because if $diff > stopval$ the loop stops, $eigenv = matix \times eigenv$ which maintains the invariant and $i = i + 1 \geq 1$. After the loop stops

$eigenv_i$ is returned, giving us the answer we want. The time complexity for this is $i * O(n^2)$ since we need to access every entry in the $n \times n$ matrix once for the matrix multiplication (although we used the inbuilt function which may be faster this is the worst case scenario).

It is not possible to have a faster algorithm since in the original TeamRank graph there are at least n nodes and each node has approximately $n - 1$ edges connected to it, thus to traverse every node and edge would be approximately $O(n^2)$. The space complexity is $O(1)$ since there are no deferred calculations.

Upon running this algorithm there was no noticeable difference in speed (since there were only 49 entries) however the loop required significantly fewer iterations, it would converge in about 30 iterations (this would change since the initial vector $eigenv_0$ would be random). This was significantly quicker to converge than the naive algorithm, thus not only did this have a lower time complexity for each iteration it also converged to an answer faster.

Note: The NumPy library was used for faster calculations with arrays. The documentation can be found here.

Final Results

Having designed and implemented the algorithms for our problem we then compared the final results between the two algorithms as well as the total points the teams had accrued over those 25 years. The results, sorted by TeamRank are shown in the table below:

Team	Points	Algorithm 1	Algorithm 2
Man Utd	2018	0.08290174208265921	0.08290174208265916
Chelsea	1803	0.07631471575593464	0.07631471575593458
Arsenal	1829	0.07533713670089531	0.07533713670089526
Liverpool	1693	0.07430650748056783	0.07430650748056779
Tottenham	1465	0.052941406399029126	0.05294140639902911
Man City	1214	0.05233852477358815	0.052338524773588135
Newcastle	1227	0.048981874900679345	0.048981874900679304
Everton	1320	0.04799606209830908	0.047996062098309054
Aston Villa	1149	0.03955224199871734	0.0395522419987173
West Ham	1046	0.03633793361610519	0.03633793361610517
Blackburn	899	0.03389182570485981	0.03389182570485978
Southampton	839	0.03177963948667615	0.03177963948667612
Leeds	641	0.025565561979914633	0.02556556197991462
Middlesborough	617	0.024693420821054895	0.02469342082105488
Sunderland	618	0.02154282830956977	0.02154282830956976
Leicester	555	0.020456176890397488	0.020456176890397477
Fulham	586	0.020132915852392252	0.020132915852392245
Bolton	575	0.019677251101249554	0.019677251101249543
West Brom	464	0.01570111019858307	0.015701110198583057
Stoke	457	0.014684824560818682	0.014684824560818677
Sheff Wed	333	0.014521613427572235	0.014521613427572225
Coventry	357	0.01366624856424269	0.013666248564242678
Charlton	361	0.01283149279782422	0.012831492797824211
Wimbledon	320	0.011680242326621645	0.011680242326621643
Crystal Palace	331	0.010702019712277725	0.010702019712277716
Swansea	312	0.010657315160048416	0.010657315160048409
Portsmouth	293	0.010474967402608745	0.01047496740260874
Birmingham	301	0.010000873345094665	0.010000873345094658

Derby	274	0.009944887653504222	0.009944887653504217
Wigan	331	0.009526570444966267	0.009526570444966262
Norwich	287	0.00945861699563603	0.009458616995636023
QPR	245	0.008437289443516571	0.008437289443516568
Nott'm Forest	199	0.007784280646882116	0.0077842806468821114
Watford	178	0.005914582594404803	0.005914582594404799
Ipswich	172	0.005911249647553917	0.005911249647553914
Hull	171	0.005302124942739393	0.00530212494273939
Burnley	157	0.004407154508034395	0.004407154508034394
Bournemouth	132	0.004130363601129227	0.004130363601129224
Wolves	136	0.004130363601129227	0.004130363601129224
Reading	119	0.0037455768238247164	0.003745576823824714
Sheff Utd	80	0.0023340309770721835	0.0023340309770721827
Bradford	62	0.002229172394188878	0.0022291723941888766
Brighton	40	0.0013033920661827637	0.0013033920661827626
Oldham	40	0.0012611442833872655	0.0012611442833872647
Blackpool	39	0.0012107285485048044	0.001210728548504804
Huddersfield	37	0.0010843557791443592	0.0010843557791443586
Swindon	30	0.0009340230372928911	0.0009340230372928903
Barnsley	35	0.0007809965048829778	0.0007809965048829772
Cardiff	30	0.0006977121025315025	0.0006977121025315019

You might be wondering why we went down this tedious route of ranking teams when we could simply refer to a cumulative points table such as this to understand which team has dominated the premier league over the past 25 years. That's where our algorithm comes in. A quick glance at the table above will show that despite winning less points than Arsenal (1829), Chelsea (1803) have a higher TeamRank score. This is because over the time period, Chelsea defeated bigger teams with a bigger margin than Arsenal. The same can be observed in the case of Manchester City and Newcastle, Leicester and Fulham amongst others. By accounting for the quality of the wins and losses we have managed to create a more realistic depiction of the tale of the past 25 years. This however, is just the beginning. We could apply this algorithm in a variety of ways to see how the change in playing conditions, time of the year, away vs home matches and such endogenous factors change the performances of teams.

References

1. https://www.kaggle.com/datasets/thefc17/epl-results-19932018
2. https://en.wikipedia.org/wiki/Elo_rating_system
3. http://infolab.stanford.edu/pub/papers/google.pdf
4. https://youtu.be/P8Kt6Abq_rM
5. https://www.w3schools.com/python/python_dictionaries.asp
6. https://youtu.be/9gtLOS87ZPs
7. https://en.wikipedia.org/wiki/PageRank
8. https://www.transfermarkt.co.in/premier-league/ewigeTabelle/wettbewerb/GB1/plus/?saison_id_von=1993saison_id_bis=2017&tabellenart=alle
9. Linear Algebra and its Applications by David Lay

Appendix

Code for cleaning up data, making dictionaries and matrix:

```
import numpy as np
import math
import operator
from operator import itemgetter
file_name = 'EPL_set.csv'

matchdata=[]
teams=[]

text = open(file_name).read()
for line in text.split('\n'):
    match = []
    data = line.split(',')

    if len(data)<2:
        continue
    if data[1] == 'date':
        continue

    if data[2] == 'Middlesboro':
        data[2] = 'Middlesbrough'

    if data[3] == 'Middlesboro':
        data[3] = 'Middlesbrough'

    if data[2] not in teams:
        teams.append(data[2])

    if int(data[4])>int(data[5]):
        match.append(data[2])
        match.append(data[3])
        match.append(int(data[4])-int(data[5]))
        matchdata.append(match)
    elif int(data[5])>int(data[4]):
        match.append(data[3])
        match.append(data[2])
        match.append(int(data[5])-int(data[4]))
        matchdata.append(match)
    else:
        match.append(data[2])
        match.append(data[3])
        match.append(0.5)
        matchdata.append(match)

        match=[]
        match.append(data[3])
        match.append(data[2])
        match.append(0.5)
        matchdata.append(match)

teams = sorted(teams)
```

```
teamlosses=dict()
for team in teams:
    teamlosses[team] = []

for match in matchdata:
    winner = [match[0],match[2]]
    teamlosses[match[1]].append(winner)

givedict=dict()
for team in teams:
    givedict[team] = dict()

for team in teams:
    total = 0
    for loss in teamlosses[team]:
        if loss[0] not in givedict[team].keys():
            givedict[team][loss[0]] = 0

        givedict[team][loss[0]] = givedict[team][loss[0]]+loss[1]
        total = total+loss[1]

    givedict[team]['Total'] = total

arrlist = []
for team in teams:
    teamlist = []
    teamdict = givedict[team]
    for team in teams:
        try:
            teamlist.append(teamdict[team]/teamdict['Total'])
        except:
            teamlist.append(0)
    arrlist.append(teamlist)

adjmat = np.matrix(arrlist)
adjmat = np.transpose(adjmat)
```

Code for ELO Implementation:

```
def Probability(r1,r2):
    return 1.0*1.0/(1 + 1.0*math.pow(10,1.0*(r1-r2)/400))

def EloChange(team1,team2,rating_dict):
    k=10
    p_t1 = Probability(rating_dict[team1],rating_dict[team2])
    p_t2 = Probability(rating_dict[team2],rating_dict[team1])

    rating_dict[team1] = rating_dict[team1]+k*(1-p_t1)
    rating_dict[team2] = rating_dict[team2]+k*(0-p_t2)
ratings = dict()
for team in teams:
    ratings[team] = 1500
for match in matchdata:
    EloChange(match[0],match[1],ratings)
```