Modifications to the Colley Matrix

C. Hayes & D. Schafer

November 22, 2014

Abstract

Modifications to the Colley Matrix are suggested to allow various game factors to affect the rankings. Suggestions are offered for general modifications, and specific examples are offered to allow for margin of victory consideration. Additionally, a method for creating Colley Matrix rankings for nonstable systems is proposed.

1 Introduction

One of the most mathematically pure methods for determining the relative strengths of teams is through the Colley Matrix. However, in the initial presentation of this method, no allowance is made for future modifications. In many ranking situations, factors other than the result of the game must be considered. This paper offers a methodology for modifying the Colley Matrix to include such factors and provides detailed descriptions on how the Colley Matrix can be modified to include margin of victory and game weighting.

Code examples will be offered throughout this article in Python, and the final version of the code used in the Hayes/Schafer rankings is in the appendix. Note that all of the data stored will be assumed to be delineated by commmas. Additionally, the scope of this paper does not extend to methods for solving the matricies generated; the Python code outputs A and b in Matlab format, and expects that the results will be inputted in Matlab format as well.

2 Colley Method

The full description of the Colley method is beyond the scope of this report; for the full details, [**?**] contains the most complete description of the mathematical basis of the Colley Matrix.

Many ranking techniques rely upon win percentage, defined as:

$$r = \frac{n_w}{n_{tot}}$$

where n_w is the number of wins and n_{tot} is the number of games played. However, this formula has numerous flaws. Consider two teams, each of which has played one game. The first team won its game, the second team lost. According to this formula, the first team is infinitely better than the second; this is clearly both difficult to work with mathematically, and probably incorrect.

The Colley Method relies upon the assumption that the Laplace win percentage is mathematically pure. It asserts that a fair win percentage can be expressed as:

$$r = \frac{1 + n_w}{2 + n_{tot}}$$

where n_w is the number of wins and n_{tot} is the number of games played. Note that with infinitly many games, this formula reduces to the previous win percentage value. [**?**] then shows that n_w can be written as follows:

$$n_w = \frac{n_w - n_l}{2} + \frac{n_{tot}}{2}$$

However, $\frac{n_{tot}}{2}$ can also be written as $\sum_{j=1}^{n_{tot}} \frac{1}{2}$. As the initial Laplace win percentage is $\frac{1}{2}$, this

last term can be considered equivelent to $\sum_{j=1}^{n_{tot}} r_j$, where r_j is the Laplace win percentage of team j. Clearly, there are t equations with t variables (where t is the number of teams). A matrix method for solving this set of equations is then offered. Consider the following equation rearrangement:

$$r_i = \frac{\frac{n_{w,i}-n_{l,i}}{2} + \sum r_j + 1}{n_{tot,i} + 2}$$

$$(n_{tot,i} + 2)r_i - \sum r_j = 1 + \frac{n_{w,i} - n_{l,i}}{2}$$

Thus, the formation of the matrix A will be as follows:

$$A_{i,i} = (n_{tot,i} + 2)$$

$$A_{i,j} = -n_{i,j}$$

Where $n_{i,j}$ is the number of games played between i and j. The b vector is formed as such:

$$b_i = 1 + \frac{n_{w,i} - n_{l,i}}{2}$$

The ratings can thus be found by solving $Ar = b$ for r.

3 Modifying the Colley Method

The Colley method works quite well, but it does not make customization especially easy. [**?**] asserts that the only way to find mathematically pure rankings is to only use wins and losses, but for a small price in purity, margin of victory and other factors can be included in the rankings.

3.1 General Concept

Consider the following code, which forms the heart of a basic Colley method program:

```
for game in games_array:
    winner = game[0]
    loser = game[1]
    winnerID = id_array[winner]
    loserID = id_array[loser]
    AMatrix[winnerID][loserID] -= 1
    AMatrix[loserID][winnerID] -= 1
    AMatrix[winnerID][winnerID] += 1
    AMatrix[loserID][loserID] += 1
    bVector[winnerID] += .5
    bVector[loserID] -= .5
```

Where $A = \texttt{AMatrix}$ and $b = \texttt{bVector}$. Clearly, this method will create A and b as specified above. One benefit of the Colley Matrix is the fact that it is both diagonally dominant and positive definite, which allows most matrix solving programs (including Matlab) to solve it quickly. In the modifications made to the program, the only changes are adding different weights to the games; that is, a game that under Colley recieves a weight of 1 might recieve a weight of .5 under the new system. Note, however, that [**?**] proves that the Colley Matrix has certain properties for any **real** number of games, and the modifications map $\mathbb{R} \to \mathbb{R}$. Thus, the modifcations maintain the original traits of the Colley matrix.

3.2 Margin of Victory

The first change that was considered was the addition of margin of victory as an indicator of the game quality. Although it was initially thought that adding this would invariably make the formula no longer mathematically pure, discussions with M. Lee and P. Diao inspired a new technique to implement fair game weighting.

The game conisdered here operates under the following ruleset: A game is played between 2 players. Each player attempts to score points, and whoever reaches a fixed number of points first wins. There is no need to win by 2. The best example of a game using this ruleset would be foosball, which is what these ratings were originally developed for.

The methodology behind the game weighting determines how likely a given victory was to be an honest indication of the two players skill levels, and how likely it was to be purely random chance. For example, a 5-0 victory is much less likely to occur randomly than a 5-4 victory. Thus, the first step is to compute the probability that a certain result would occur in a game between two equally skilled players (such that the probabilty of either scoring any given goal is $\frac{1}{2}$).

Consider first that in a game played to n goals, there will be $2n-1$ goals scored (this assumes that the game does not end once a player reaches n; note that this will never change the result. There is a bijection, therefore, between game results and a list of the $2n-1$ goals scored, with W indicating a goal scored by a winning team, L indicating a goal scored by a losing team, and O indicating a goal scored by either team.

In a n-0 victory, the list of goals scored will be as follows:

$$WWW \cdots WWOO \cdots OO$$

Note that there are n Ws and $n-1$ Os. The probability of any of these configurations occuring is $2^{-(2n-1)}$, but there are 2^{n-1} configurations of the Os at the end, and 1 rearrangement of the Ws. Thus,

$$P_{n-0} = \frac{2^{n-1}}{2^{2n-1}} = \frac{1}{2^n}$$

For a $(n-1)$-0 victory, the list of goals scored will be as follows:

$$WWW \cdots WLWWOO \cdots OO$$

Where there are n Ws, $n-2$ Os, and 1 L, and the L can be anywhere before the Os **except** the last spot (for the winning team cannot win on a losing team's goal). Thus, there are 2^{n-2} conigurations of the Os, and $\binom{n-1}{n}$ rearrangements of the W and Ls. As the probability of any configuration occuring is $2^{-(2n-1)}$,

$$P_{n-1} = \frac{(2^{n-2})(\binom{n-1}{n})}{2^{2n-1}} = \frac{n-1}{2^{n+1}}$$

In general, then, in an n-p victory, there will be 2^{n-p-1} configurations of the Os, and $\binom{n-p+1}{p}$ rearrangements of the W and Ls, so:

$$P_{n-p} = \frac{(2^{n-p-1})(\binom{n+p-1}{n-1})}{2^{2n-1}}$$

Note that $\sum_{i=0}^{n-1} P_{n-i} = \frac{1}{2}$, that is, the probability that one of the players will win is $\frac{1}{2}$.

The multiplier is derived using these values. For example, conisider a n-p victory for a certain player. A random player is going to achieve this result, or a result better than this, with probability $\sum_{i=0}^{p} P_{n-i}$. Thus, the probability that this result is a valid indication of the relative skill level of the two players is $1 - \sum_{i=0}^{p} P_{n-i}$.

The code used to create this method is shown below.

```
def getScalingFactor(winnerScore,loserScore):
    probRand = 0
    for i in range(loserScore+1):
        probRand += (1.0)*(2**(winnerScore-i-1)) * (factorial(↩
            winnerScore+i-1)/
                    (factorial(winnerScore-1)*factorial(i)))/(2**(2*↩
                        winnerScore-1))
    scaleF = 1-probRand
    return scaleF

...

    for game in games_array:
        winnerID = game[1]
        loserID = game[2]
        winner_score = game[3]
        loser_score = game[4]
        score_scale_factor = getScalingFactor(int(winner_score),int(↩
            loser_score))
        AMatrix[winnerID][loserID] -= 1*score_scale_factor
        AMatrix[loserID][winnerID] -= 1*score_scale_factor
        AMatrix[winnerID][winnerID] += 1*score_scale_factor
        AMatrix[loserID][loserID] += 1*score_scale_factor
```

5-0 games	22
5-1 games	85
5-2 games	107
5-3 games	141
5-4 games	126
Total Games	481

Table 1: Number of Games by Score

Score	Theorectical	Actual
5-0	.96875	.97713
5-1	.890625	.88878
5-2	.7734375	.77755
5-3	.63671875	.63098
5-4	.5	.5

Table 2: Score Probability Comparison

```
bVector[winnerID] += .5*score_scale_factor
bVector[loserID] -= .5*score_scale_factor
cVector[winnerID] += 1
cVector[loserID] += 1
```

To verify this assumption, we considered data after 12 weeks of play, which resulted in the statistics shown in Table 1. We therefore consider the amount of games that was won by each score or better over the total number of games. This is the probability that either player will win by that score. We therefore must divide by 2, to determine the probability that a specific player will win by that score, then we subtract from 1, to determine the multiplier.:

$$
\begin{aligned}
P_{5-0,actual} &= \frac{2-\frac{22}{481}}{2} \\
P_{5-1,actual} &= \frac{2-\frac{22+85}{481}}{2} \\
P_{5-2,actual} &= \frac{2-\frac{22+85+107}{481}}{2} \\
P_{5-3,actual} &= \frac{2-\frac{22+85+107+141}{481}}{2} \\
P_{5-4,actual} &= \frac{2-\frac{22+85+107+141+126}{481}}{2}
\end{aligned}
$$

A comparison between the theoretical multipliers and the actual multipliers is shown in Table 2. Clearly, the assumptions made in this sections are corroborated by the actual data.

3.3 Nonstable systems

Another feature that can be added to the Colley Matrix fixes a major problem with the original. Under the original system, a game counts the same no matter when it occurs. However, in an unstable environment, recent games are more indicative of the rankings than games in the far past. Although there is no mathematically pure justification for the weighting chosed, it was decided that $(.8)^n$, where n =the number of weeks that have passed since the game was plyed, would be a good multiplier for these purposes.

The code used to create this method is shown below.

```
def getDecay(weeksSincePlayed):
    return .8**weeksSincePlayed

...

    for game in games_array:
        winnerID = game[1]
        loserID = game[2]
        winner_score = game[3]
        loser_score = game[4]
        exp_decay_const = getDecay(maxWeek-game[5])
        AMatrix[winnerID][loserID] -= 1*exp_decay_const
        AMatrix[loserID][winnerID] -= 1*exp_decay_const
        AMatrix[winnerID][winnerID] += 1*exp_decay_const
        AMatrix[loserID][loserID] += 1*exp_decay_const
        bVector[winnerID] += .5*exp_decay_const
        bVector[loserID] -= .5*exp_decay_const
        cVector[winnerID] += exp_decay_const
        cVector[loserID] += exp_decay_const
```

4 Conclusions

In the modifications made to the Colley Matrix, many beneficial changes have been made. A methodology for creating fair, stable modificaitons was developed, and two modifications, one entirely justifiable using combinatorics and statistics and one *ad hoc*, were implemented using this technique. The results that were developed in the testing were reasonably accurate, validating the changes made.

5 Acknowledgements

First, we would like to thank W. Colley, for developing the original paper on his ranking technique. His work has been an invaluable resoure during the creation of both our program and this report. We also would like to thank J. Geneson, one of the original supporters of the rating system we developed, and P. Diao and M. Lee, who worked with us to develop the mathematically justifiable version of the margin of victory function.

A Final Code

All of the following code is written either in Python 2.3.5, or for Matlab 7.1.0.183.

A.1 colley.py

```
from string import join
import os
import sys
from sys import argv
import MySQLdb

if len(argv) > 1:
    modifier = argv[1]
else:
    modifier = "none"

def factorial(x):
    x = int(x)
    if x < 0:
        return 0
    if x == 0:
        return 1
    return x*factorial(x-1)

def readFile(filename):
    file = open(filename, "r")
    data = file.readlines()
    file.close()
    trueData = [r[:-1] for r in data]
    return trueData

def convertMatrixToStrings(matrix):
    return [[str(j) for j in i] for i in matrix]

def convertListToStrings(matrix):
    return [str(i) for i in matrix]

def sort_by_value(d):
    items=d.items()
    backitems=[[v[1][-1],v[0]] for v in items]
    backitems.sort()
    backitems.reverse()
```

```
    return [ backitems[i][1] for i in range(0,len(backitems))]

def matlabFormat(AMatrix):
    numP = len(AMatrix)
    s = "["
    for i in range(numP):
        for j in range(numP):
            s += str(AMatrix[i][j])
            s += ","
        s = s[:-1]
        s += "; "
    s = s[:-2]
    s += "]"
    return s

def getScalingFactor(winnerScore,loserScore):
    probRand = 0
    for i in range(loserScore+1):
        probRand += (1.0)*(2**(winnerScore-i-1)) * (factorial(↩
           winnerScore+i-1)/
                   (factorial(winnerScore-1)*factorial(i)))/(2**(2*↩
                      winnerScore-1))
    scaleF = 1-probRand
    return scaleF

def getDecay(weeksSincePlayed):
    return .8**weeksSincePlayed

def displayRatings(ratingArray):
    print "Player\t\tRating"
    for p in sort_by_value(ratingArray):
        if len(p) < 8:
            print p+"\t\t"+str(ratingArray[p][-1])
        else:
            print p+"\t"+str(ratingArray[p][-1])


def generateAB():
    db = MySQLdb.connect()
    cursor = db.cursor();
    cursor.execute("SELECT * FROM foosballGames;");
    games_array = cursor.fetchall();
    cursor.execute("SELECT * FROM foosballPlayers;");
    user_array = cursor.fetchall();
    cursor.execute("UPDATE foosballGames SET new=0;");
```

```
maxWeek = 1
for game in games_array:
    if game[5] > maxWeek:
        maxWeek = game[5];

numP = len(user_array)+1;
bVector = [1 for i in range(numP)]
AMatrix = [[0 for j in range(numP)] for i in range(numP)]
cVector = [0 for i in range(numP)]
for i in range(numP):
    AMatrix[i][i] = 2

for game in games_array:
    winnerID = game[1]
    loserID = game[2]
    winner_score = game[3]
    loser_score = game[4]
    exp_decay_const = getDecay(maxWeek-game[5])
    score_scale_factor = getScalingFactor(int(winner_score),int(↩
       loser_score))
    AMatrix[winnerID][loserID] -= 1*exp_decay_const*↩
       score_scale_factor
    AMatrix[loserID][winnerID] -= 1*exp_decay_const*↩
       score_scale_factor
    AMatrix[winnerID][winnerID] += 1*exp_decay_const*↩
       score_scale_factor
    AMatrix[loserID][loserID] += 1*exp_decay_const*↩
       score_scale_factor
    bVector[winnerID] += .5*exp_decay_const*score_scale_factor
    bVector[loserID] -= .5*exp_decay_const*score_scale_factor
    cVector[winnerID] += exp_decay_const
    cVector[loserID] += exp_decay_const

file = open("aMatrix.csv","w")
file.write("\n".join([",".join(row) for row in ↩
   convertMatrixToStrings(AMatrix)]))
file.close()

file = open("bVector.csv","w")
file.write(",".join(convertListToStrings(bVector)))
file.close()

file = open("activeness.csv","w")
file.write(",".join(convertListToStrings(cVector)))
```

```
    file.write("\n");
    file.close()

    print "A and b successfully generated!"

def generateRankings():

    db = MySQLdb.connect()

    matlabResults = readFile("matlabRes.csv")[0][:-1]
    newRankings = matlabResults.split(",")

    activenessResults = readFile("activeness.csv")[0]
    activenessRankings = activenessResults.split(",")

    for i in range(1,len(newRankings)):
        cursor = db.cursor()
        cursor.execute("UPDATE foosballPlayers
                    SET rating="+str(newRankings[i])+",
                    activeness="+str(activenessRankings[i])+"
                    WHERE playerID="+str(i)+";")

    print "Ratings updated!"

def main():
    if modifier == "gmf": #Generate Matlab Files
        generateAB()
        return
    if modifier == "urf": #Update Ranking Files
        generateRankings()
        return
    while 1:
        os.system("clear")
        print "Main Menu"
        print
        print "1. Generate AB"
        print "2. Update Current Week Rankings"
        print "3. Exit"
        print
        i = raw_input("?")
        if i == "1":
            generateAB()
            raw_input()
        elif i == "2":
            generateRankings()
```

```
            raw_input()
        else:
            return

if __name__ == "__main__":
    main()
```

A.2 getResults.m

```
A = dlmread('aMatrix.csv',',');
b = dlmread('bVector.csv',',');
results = A\b';
fid = fopen('matlabRes.csv','w');
fprintf(fid,'%f,',results);
'Matrix equations successfully solved!'
```