Applications of Matrices and Linear Algebra in Predicting Sports Rankings

Maxwell Jones, Eric Gan

1 BACKGROUND

In this application of linear algebra, we decided to look into how matrices could predict how well sports teams would do by the end of the season. More specifically, we researched methods that could estimate the final rankings of sports teams so that we could utilize these results to predict which football team would win the Super Bowl at the end of any arbitrary year.

This paper focuses on three main methods that work towards accomplishing this goal. We looked into the Massey Method executed two ways and the Keener Method.

1.1 *Least Squares Massey Method*

The most basic and overarching description of the Massey Method of ranking for sports teams is an application of the least-squares approximation to the point differentials of the teams. Therefore, in order to understand the background behind the Massey Method of ranking, we need to first define what a point differential is.

The point differential for a team for a single game is the team's score - the opposing team's score. If we're looking at team i who's playing against team j for that single game, the point differential for that game is

$$\text{score of } i - \text{ score of } j$$

We would like the rankings of the two teams to differ by the same amount that the score differs by. Then, we apply this same process to all the games that were played during that season and there's enough data to apply the Massey Method of ranking. In essence, we want to find rankings such that for any game in the season, the score differential is equivalent to the ranking differential.

We use this data to craft a set of linear equations that we eventually want to approximate. For example, if we instead label the teams with numbers we will have the equations

$$r_1 - r_2 = \text{ point differential for team 1 in this game },$$

$$r_2 - r_3 = \text{ point differential for team 2 in this game }, \ldots$$

where $r_1, r_2, r_3, \ldots$ represent the "Massey rating" for the respective teams. This creates an equation for every game in the season. If you write out the matrix with each team having its own column, then you can see that each row of the matrix A will have a $1, -1$ and a bunch of 0's. This matrix will then be multiplied by a vector $\vec{r}$ which is filled with the variables $r_1, r_2, \ldots$. These "Massey ratings" are what we are trying to figure out in this process, with a higher rating representing a better team. Finally, the result vector $\vec{b}$ is filled with all the point differentials for the games. Since we cannot guarantee that this set of linear equations is solvable, we are stuck with the least-squares problem

$$A\vec{r} = \vec{b}$$

If we left-multiply both sides of the equation by A^T, we get

$$A^T A\vec{r} = A^T\vec{b}$$

It isn't guaranteed that M is invertible so in the Massey Method, so in the case that it is not, we replace all the zero terms with some small value like 10^{-15}. Then we multiply both sides of the equation by the inverse of the resulting Massey Matrix to find the ratings for each team. These ratings are then used to rank the teams. It is also important to notice that the result for the ratings Matrix is the same as a Least Squares approximation on the original equation:

$$\vec{r} = (A^T A)^{-1} A^T \vec{b}$$

1.2 *PCA Approximation with Massey Matrix*

The Conventional Massey Method used least-squares approximation but this method of approximating may not be the most accurate. As shown earlier, it also runs into the issue of if the matrix A isn't invertible. To combat these problems as well as to approximate minimizing a slightly different paramater, it is also possible to apply the PCA(Principle Component Analysis) approximation method, another way to find an approximation to a set of equations without a solution. This does not require any change for non-invertible matrices. We just apply PCA to the

$$A\vec{r} = \vec{b}$$

from the previous section to get another approximation which will give a different set of ratings for the sports teams. Later in the paper, we briefly discuss whether this is actually more accurate than the least-squares Massey Method or not.

1.3 *Keener Method*

The Keener Method also relies on its own rating system implementing so called "strength" of teams. In this method, the rating for some team i, r_i and strength for team i, s_i, are proportional to each other by some constant λ.

$$s_i = \lambda r_i$$

This makes sense because it's generally intuitive that if a team is stronger, it will have a higher rating. The strength is based off of some sort of factor between two teams, a_{ij}. a_{ij} can be many things, such as number of wins for team i against team j, total number of passing yards for team i in all the games against team j, etc. The easiest and most accurate factor for us was the total score for team i in all the games against team j, S_{ij} because the records for past games can easily be found online.

However, we don't immediately represent the strength of team i as the sum of these S_{ij} against the rest of the teams in the league. It's possible for some teams to score a lot and still lose a lot because they are very attack-focused teams, and vice-versa. In order to punish teams for allowing a high number of points scored against them in relation to points they score, we let our a_{ij} be as follows:

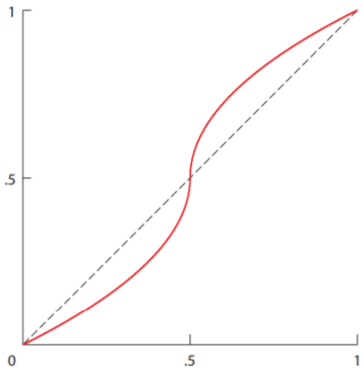
$$a_{ij} = \frac{S_{ij}}{S_{ij} + S_{ji}}$$

In order to make sure our fractions is nonzero and defined, we apply Laplace's rule of succession:

$$a_{ij} = \frac{S_{ij} + 1}{S_{ij} + S_{ji} + 2}$$

Finally, we slightly skew our a_{ij} because there is still the inconsistency that happens when a strong team completely demolishes a weaker team and racks up their own score. There are many skewing functions out there and you can even make your own, but we were given the nonlinear skewing function

$$h(x) = \frac{1}{2} + \frac{sgn(x - (1/2))\sqrt{\mid 2x - 1 \mid}}{2}$$



a_{ij} would usually also need to be normalized by being divided by the total number of games played by team i to account for a possible different amount of games played by every team, however, in a regular season for the NFL, each team plays the same number of games so this process isn't really necessary.

The relative strength of team i compared to team j is calculated by $a_{ij}r_j = S_{ij}$ where r_j is the rating of team j. We're trying to solve for this because we want to rank the teams by their ratings. The absolute strength of team i, s_i, is the sum of the relative strengths for all the other teams in the league. The vector $\vec{s}$ contains the absolute strengths for all the teams in the league and if the vector is written out with all the mathematical components substituted in, we get

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{bmatrix} \vec{r} = \vec{s} \Rightarrow A\vec{r} = \vec{s}$$

where $\vec{r}$ is the vector containing the ratings for all the teams. We also know from the beginning that strength and rating are proportional so we can substitue in this fact and get

$$A\vec{r} = \lambda\vec{r}$$

This becomes a simple eigenvalue-eigenvector problem and we can just plug in these matrices and vectors into Julia to get an answer. There is also the issue of which eigenvector and eigenvalue to pick. In the Keener Method, we just pick the largest positive eigenvalue and the eigenvector that is associated with that. This eigenvector becomes the rating vector and we use this to finally rank the teams.

2 CODE EXPLORATION

All the code in this section is a slightly more pseudocode version of the actual code, for readability. The actual code will be included in the appendix at the end of this paper. These code processes are basically just transcribing the processes described in the background into code format.

2.1 *Massey Code*

To find the Massey Matrix and scores vector, we first utilize the code

```
MainMasseyMatrix = MakeNewMatrix(num_games_played by 32)
for i in range(num_games_played)
    MainMasseyMatrix[Home_Team_Index] = 1
    MainMasseyMatrix[Away_team_index] = -1
end
for i = 1:num_games_played
    Scores[i] = Home_score_game_i - Away_score_game_i
```

```
end
```

If we want to weigh this, multiply by $(1 + (\text{Week of game})\frac{1}{16})$ This gives the first game weight one, and every other game incrementally more until the last game which has weight two. If we also want to perturb the data, then we do

```
for i = 1:num_games_played
    for j = 1:num_games_played
        MainMasseyMatrix[i,j] += 1*10^(-15)
    end
end
```

Now that we have our main Massey matrix and our scores matrix, we just need to run the least-squares approximation on these matrices.

```
Ranks=inv(transpose(MainMasseyMatrix)*MainMasseyMatrix)*transpose(
    MainMasseyMatrix)*Scores
```

2.2 *PCA Code*

Let MainMasseyMatrix be the variable name for the Massey matrix and Scores be the variable name for the vector of score differences that we found in the first part.

$$MainMasseyMatrix * Ranks = Scores \Rightarrow U\Sigma V^T * Ranks = Scores$$

This is true because we apply SVD so we compute

$$Ranks = V\Sigma^{+}U^T * Scores$$

```
U,σ,V = svd(MainMasseyMatrix)
Eplus = Matrix(correct dimensions)
E = Matrix(correct dimensions)
for x = 1:32
    Eplus[x,x] = 1/σ[x]
    E[x,x] = σ[x]
End
PCARanks = V*Eplus*transpose(U)*Scores
```

If we wanted a k approximation, we would only use values for x less than some k. If $x < 17$, EplusApproximate[x,x] $= 1/\sigma$[x]. We could also use weighted Scores or approximate Ranks here.

2.3 *Keener Code*

```
KeenerMain = Matrix(32,32)
KeenerScores = Matrix(32,32)
For i = 1:num_games_played
    Scores(home_team_index,Away_team_index) += home_team_score_game_i
    Scores(Away_team_index, home_team_index) += away_team_score_game_i
```

We have now generated a scores matrix that has all points scored by team i on team j in position scores(i,j).

```
For i = 1:32
  For j = 1:32
    if(i_and_j_have_played())
    KeenerMatrix(A(i,j)) = normalize((points_i_scored_on_j + 1)/(
   points_i_scored_on_j + points_j_scored_on_i + 2)
    else
    KeenerMatrix(A(i,j)) = perturbedavalue
End
End
End
```

Now that we have our Keener matrix we just say

```
Answer = eigenvector_for_biggest_eigenvalue(KeenerMatrix)
```

3 DESIGN DECISIONS

When deciding to do a project that relates to ranking teams based on a season of games, there are two Linear Algorithms that are the most widely used. These two are the Massey Method and the Colley Method. Of these, we decided in favor of the Massey Method, but against Using the Colley Method. We also decided to use a lesser known method called the Keener Method as our second ranking system method. We did not just pick two methods that we found sources for, but instead made these choices proactively with specific ideas in mind.

The Massey Method is the most simple out of the three methods, but as a result allows for a lot of tweaking/enhancing that some of the other methods are more stringent about. As a quick summary, the Massey Method entails computation of a Matrix that stores information about which teams have played each other, and a vectors that keeps track of point differential in every game. We posit that there exists a rankings vector such that if we multiply it to the games matrix, we output the point differential matrix. Since it is unlikely that there will be an exact rankings vector that can calculate the point differential exactly, The solution to the Massey method is to do a least squares approximation on the data to compute a ranking system.

While this method is the most simple out of all of them, The key insight is recognizing that all that this method is really doing is approximating an $Ax = b$ equation when there is not a real solution. With this, based on 21-241, we know that there is not just one way to do this. We can not only use a least-squares approximation for the data, but we can also use a PCA approximation for our vector. Now we have two ways of calculating our ranking matrix, that are finding the minimum of different quantities(vertical distances versus orthogonal distances). We mentioned in class that the PCA approximation find more underlying information in data that we purport to be noisy, so it makes perfect sense to use for regular season games that have a built in randomness when trying to find the best team. We can also do a least squares approximation of the data in hopes of seeing if there are any underlying variables that play huge importance.

The Massey method also allows for easy weighting of data. Since all of our games are stored in a single vector, we can simply change these values, multiplying them by a factor of 1 to 2 depending on how close they are to the end of the season. By doing this, we are giving games that happen closer to the postseason more importance, in hopes that this will give teams that perform better towards the end an advantage over those whose performance goes down towards the end of the season. This can be applied to both the least-square and PCA method, since they only care about making sure that there is an $A\vec{x} = b$ equation to approximate.

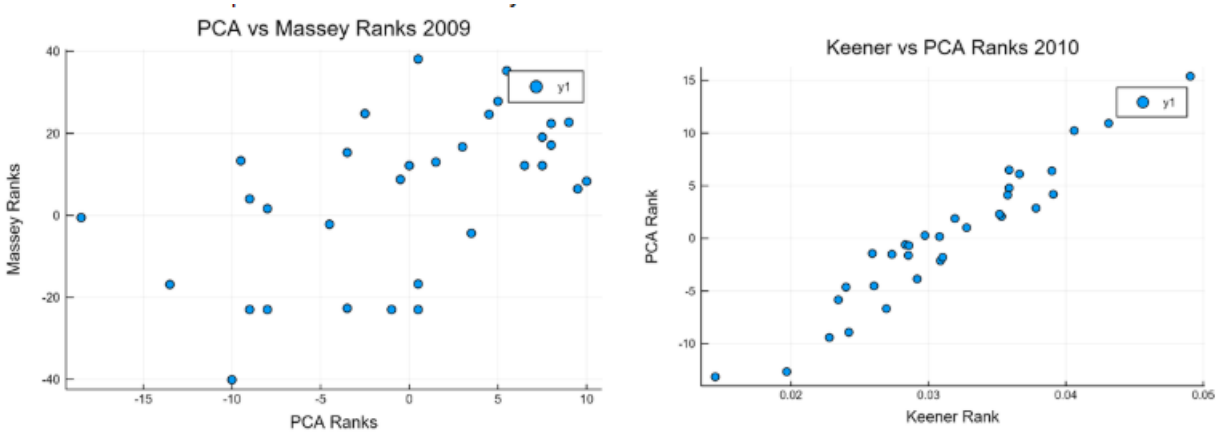
We chose not to use the Colley method because while it is widely respected as a good way to approximate data, there is no real way to change its results or add anything extra to it. One of the reasons why the Massey Method

was appealing was because it allowed for a large amount of changeability and adaptation, but the Colley Method is almost set in stone. We could weight the data, but as it turned out weighting didn't play a huge role and our results would have been just copying the exact algorithm presented.

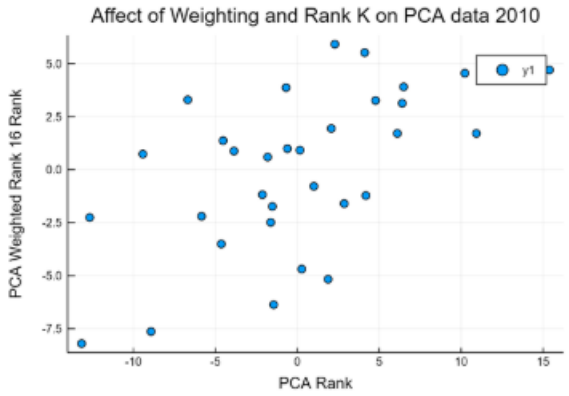
The Keener method was used both for it's variability and its use of eigenvectors and eigenvalues. We did not find other methods that applied eigenvectors and eigenvalues, and it was an interesting idea to use a principle almost built in to the idea of matrices and linear algebra, the existence of eigenvectors and eigenvalues, to find rankings. The Keener data also allowed for normalization(or not) as well as weighting the data. The final interesting thing about the Keener data was the ability to perturb the matrix or not. Perturbance of the matrix means adding a small component(we used 10^{-15}) to every value of the matrix in order to guarantee we have a real largest eigenvalue. This idea was both interesting and useful later on, and was described in the background.

4 RESULTS

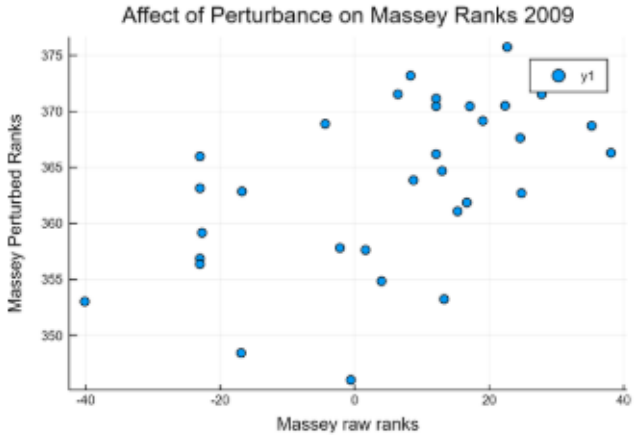
The results of our calculations were not totally unexpected. We base our performance off of the prediction of the best ranks, since not all NFL teams make the playoffs anyway, and of those top teams what we really want is a prediction of which team will win the Super Bowl. It is also important to note that while most years' data had a non-singular Massey Matrix, the 2010 data was singular so we had to employ the changes to the Massey Matrix from the background section in order to make it non-singular. The first thing that we noticed was that the Massey Least-Squares approximation performed the worst out of the three main methods (Least Squares, PCA, and Keener). While the PCA Method had the Super Bowl winning New Orleans Saints as the second best team likely to win a championship that year and the Keener had them as the favorites, the least-squares approximation had them as the sixth most likely to win the title. As we can see, the Keener and PCA data are very similar, while the least-squares data from that year is much different. Another interesting result is that a least-squares approximation was very bad at getting the data correct. This can be seen as very little correlation to the mostly correct PCA data. All this data is shown in the two graphs below.



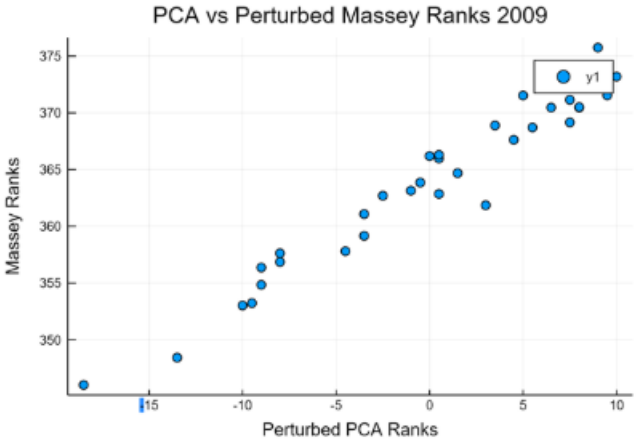
The data goes from accurate to considerably less accurate after a PCA rank 16 approximation, which is not that much considering there are only 32 total ranks in 2009 and a significant amount for all other years tested as well. Investigating further, we noticed that the Σ matrix in $U\Sigma V^T$ had significant weights for 31 out of the 32 singular values. This can be seen below.



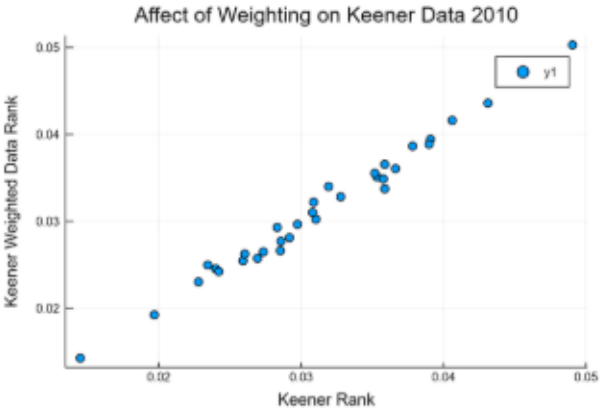
One of our most interesting results occurred with our least-square approximation Massey matrix. After perturbing our Keener data, we decided to perturb our Massey matrix as well, To see what effect it would have. The results were astonishing. It not only changed the least-squares approximation such that it now correlated with the other two, but the least-squares approximation also outperformed both the Keener(which predicted the right Super Bowl Champ) and the PCA(Which had them third) by picking the correct champion by a higher margin. This success continued into 2010 Where the least-squares method performed similarly to the Keener which both had the Super Bowl winning Green Bay Packers as second best ranking and outperformed the PCA approximation(where the perturbed matrix was used) which had the Packers out of the top three. The graphs below show how perturbation both drastically changes Massey least square ranks and how they correlate to PCA ranks after perturbation for the

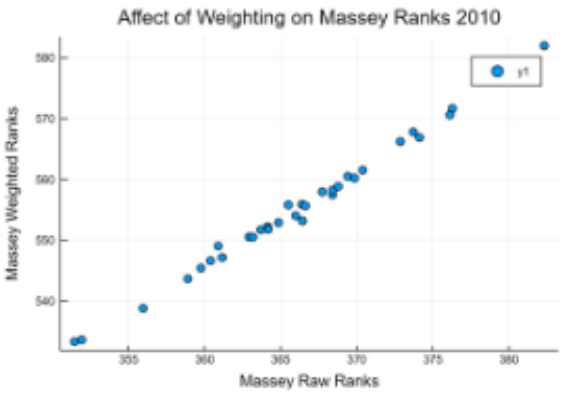


2009 data.



With respect to weighting, it seemed to not play as important a role as we had first postulated. When looking at weighting data where games go from a factor of one importance to two, we notice that none of the approximations change in a significant way in the graphs below.

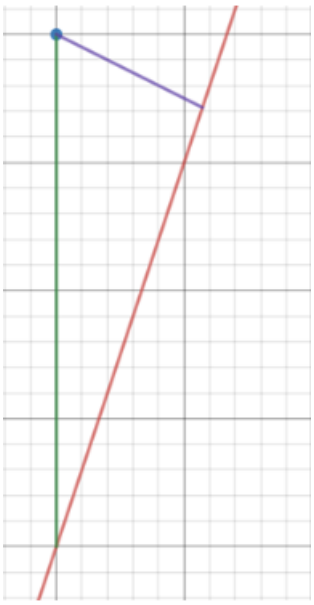




This may be because no teams had any huge winning streaks toward the end of the season. If this weighting is made more extreme. For example, if the last game is weighted 3 times a much as the first game, then the predictions become much less accurate.

5 INTERPRETATION

The first thing we notice is that the PCA data performed better than the least-squares data for the same matrix. This intuitively makes sense because PCA data is better suited to finding underlying data in blurry measurements. Since every football game has a relatively high measure of randomness, it makes sense that the PCA data would perform better. The next logical question is, what distances are the least-squares data minimizing? While the high number of dimensions may seem a little bit daunting, the idea seems to be that the least-squares approximation is trying to minimize each game without respect to the other games. The PCA approximation, by looking at orthogonality, respects the other games because the distance to the line is not solely dependent on the error based on one game, a situation analogous to the graph below.



The red line here represents a rankings guess, and the blue point represents a game. As we can see, while it's possible that the guess might be off for that particular game, if we find the orthogonal distance, representing the distance with respect to the line itself, it is quite small in comparison. With blurry data that only has two nonzero terms per row, it is possible that many lines in the higher dimension have this type of data, since every team is likely to have at least one "off" game, where they lose when they should not or even if they win by a small margin against a team that is expected to be much worse than them.

In terms of weightings, it seems that no teams in the 2009 or 2010 seasons went on to have much larger or smaller point differentials then at the beginning of the season, and so the results did not change as much. While we experimented with making the last games much more important than the first, 3 or more times as important, this just made our data worse and so we quickly did away with this approach. It is possible that because of the low number of games that every team plays in the regular season, 16 games, there wasn't much opportunity for this weighting system to have a large affect. The results may have been different if looking at the NBA or MLB seasons, when teams play 82 and over 100 games, respectively. This discovery with weightings held true for all three ranking systems.

The PCA data had many important singular values, which indicates that the Massey matrix was constructed in such a way that made it very hard for any few singular values to take into effect. This is likely because of the large amount of zeros/small values that the matrix contained. As a result of this, the least-squares approximation was very bad at being able to predict anything useful about the data.

The Keener Method was very good at predicting the data, regardless of whether the weighting system worked. It is another real world example of how eigenvectors and eigenvalues can be used, and possibly one of the most interesting, because the use of eigenvalues comes from intuition about how strengths and rankings should be related as opposed to trying to save space or just using them for their data-storage purposes. It is also important to note that without normalization or perturbation of the scores in the Keener method, it performed worse that the PCA and the perturbed Massey matrix. This indicates that normalization is very important when computing eigenvalues and vectors when it comes to accuracy of rankings.

Possibly the biggest discovery was that perturbation of the Massey matrix lead to slightly increased performance in the PCA Matrix, but a massive increase in performance in the least-squares approximation matrix. Since we

have to calculate an inverse in the PCA Matrix, it seems that this inverse is not only sensitive to perturbations, but it becomes more accurate with them. In all honesty, the idea of perturbing a matrix is a problem within itself, and it would be possible to do a whole project on different types of perturbations and their uses. Without going too deep into that field, we will examine this topic a little. When computing an inverse, we know that AA^{-1} must equal the identity matrix. Since our original Massey matrix contained a lot of zeros, A^{-1} was forced to obtain the identity matrix with very little data points to do so. In this case A is M^TM, but the concept still holds. With perturbations, it allows The inverse to store very large values instead to offset for these zeros when multiplied with the small perturbation values, which keeps some integrity of the original matrix while also pushing all the final rankings up (the average ranking for the perturbed data was centered around 365 while the unperturbed rankings were centered around 0).

The next question to ask is, are these models really better than just looking at the records of the best teams and picking whichever has the best rating. The answer is yes. In 2009, the New Orleans Saints had the third best record in the NFL, not the best, yet two of the three methods guessed them as the Super Bowl champs. In addition, the best team in the NFL was the Indianapolis Colts, and all models had the Saints having a significantly higher ranking. This was shown as at least a good guess since the Saints won the Super Bowl by 14 points, which is a considerable margin for a football game. It is also good to notice that the other 13-3 team in 2009 was the San Diego Chargers. They lost to the 9-7 New York Jets in their first game of the playoffs, and both the perturbed Massey and PCA Rankings predicted this result by giving the Jets a higher ranking than the Colts. The Keener ranking had them 1 spot above the Jets, but with almost the same ranking, so it at least predicted a close matchup. In 2010, the models(perturbed Massey, PCA, and Keener) did not predict the correct champion Packers, but two had them at the second spot and one had them at the third. This is still very impressive considering that there were 6 teams that had a better record than the Packers in 2010 and 5 that had the same record.

Even with all this being said, some years have such high playoff variance that the models are not the best. In 2012, the 9-7 Giants somehow made it to the Super Bowl and defeated the 16-0 Patriots, and the next year, the 49ers quarterback got hurt towards the end of the season, and the backup quarterback, Colin Kaepernick, led the team to the Super Bowl(though they lost). Because of such a small amount of games, and a playoff where it is single game elimination, instead of a 7 game series or some other series like in the NBA or MLB, the models and analysts, can fail when the data simply does not correlate to who wins in the playoffs and Super Bowl. As an estimate, however, the models are still very good at being able to guess who wins a certain game or Super Bowl in a given year. In all likelihood they would be even more accurate for an NBA or MLB season, where there are more games and less chances for upsets due to the increased number of games in a playoff series and the lower number of moving pieces in one team. A football team has around 50 people while baseball team has 25 and a basketball team has 15, of which around 7 to 9 are important.

REFERENCES

1. "Chapter 2. Massey's Method." Who's 1? The Science of Rating and Ranking, by Amy N Langville and Carl D Meyer, Princeton University Press, 2012, pp. 9-13.
2. "Chapter 4: Keener's Method." Who's 1? The Science of Rating and Ranking, by Amy N Langville and Carl D Meyer, Princeton University Press, 2012, pp. 29-50.

In [1]:

```
#Function to display rankings of teams
function showrankings(IndextoTeams,Rankings)
           HelperDict = Dict{Int64, String}()
           RatingsDict = Dict{Float64, String}()
           for key in collect(keys(IndextoTeams))
              HelperDict[IndextoTeams[key]] = key
           end
           for i = 1:32
             RatingsDict[Rankings[i,1]] = HelperDict[i]
           end
           for key in sort(collect(keys(RatingsDict)), rev=true)
               println("$(RatingsDict[key]) => $(key)")
           end
    end
```

Out[1]:

```
showrankings (generic function with 1 method)
```

In [2]:

```
#Reading in Data
using DelimitedFiles
using LinearAlgebra
using Plots.PlotMeasures
Data = readdlm("reg_games_2009.csv.txt", ',')
```

Out[2]:

```
257×10 Array{Any,2}:
 "type"            "game_id"   "home_team"  …    "home_score"    "away_score"
 "reg"    2009091000           "PIT"             13              10
 "reg"    2009091304           "CLE"             20              34
 "reg"    2009091307           "NO"              45              27
 "reg"    2009091308           "TB"              21              34
 "reg"    2009091305           "HOU"         …    7              24
 "reg"    2009091306           "IND"             14              12
 "reg"    2009091303           "CIN"              7              12
 "reg"    2009091302           "CAR"             10              38
 "reg"    2009091301           "BAL"             38              24
 "reg"    2009091300           "ATL"         …   19               7
 "reg"    2009091309           "ARI"             16              20
 "reg"    2009091311           "SEA"             28               0
 ⋮                                          ⋱
 "reg"    2010010307           "MIN"         …   44               7
 "reg"    2010010306           "MIA"             24              30
 "reg"    2010010305           "HOU"             34              27
 "reg"    2010010304           "DET"             23              37
 "reg"    2010010302           "CLE"             23              17
 "reg"    2010010312           "DEN"         …   24              44
 "reg"    2010010311           "ARI"              7              33
 "reg"    2010010314           "SD"              23              20
 "reg"    2010010313           "OAK"             13              21
 "reg"    2010010303           "DAL"             24               0
 "reg"    2010010315           "SEA"         …   13              17
 "reg"    2010010308           "NYJ"             37               0
```

In [3]:

```
#Assigning every team an index
teamsToIndex = Dict("NYJ" => 1, "SD" => 2, "NYG" => 3, "DAL" => 4, "PIT" => 5, "DET" => 6, "ARI" =>
7, "MIN" => 8, "NE" => 9, "TB" => 10, "GB" => 11, "SF" => 12, "BUF" => 13, "TEN" => 14, "DEN" => 15
, "IND" => 16, "PHI" => 17, "HOU" => 18, "WAS" => 19, "CIN" => 20, "OAK" => 21, "BAL" => 22, "CHI"
=> 23, "ATL" => 24, "CLE" => 25, "STL" => 26, "JAC" => 27, "MIA" => 28, "SEA" => 29, "NO" => 30, "K
C" => 31, "CAR" => 32)
```

Out[3]:

```
Dict{String,Int64} with 32 entries:
```

```
  "DET" => 6
  "BUF" => 13
  "CIN" => 20
  "SD"  => 2
  "PIT" => 5
  "HOU" => 18
  "CLE" => 25
  "NE"  => 9
  "STL" => 26
  "MIA" => 28
  "IND" => 16
  "TEN" => 14
  "SF"  => 12
  "DAL" => 4
  "BAL" => 22
  "PHI" => 17
  "NYG" => 3
  "ARI" => 7
  "NO"  => 30
  "MIN" => 8
  "TB"  => 10
  "DEN" => 15
  "GB"  => 11
  "CHI" => 23
  "SEA" => 29
  ⋮     => ⋮
```

In [4]:

```
#Initializing Scores, Weighted Scores, and Main Matrix for Massey method
Scores = zeros(Int, size(Data,1)-1,1)
WeightedScores = zeros(Float64, size(Data, 1)-1,1)
MainMasseyMatrix = zeros(Float64, size(Data,1)-1,32)
PMainMasseyMatrix = zeros(Float64, size(Data,1)-1,32)
```

Out[4]:

```
256×32 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                        ⋮              ⋱  ⋮                        ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

In [5]:

```
#Filling Main Massey Matrix, Scores, and Weighted Scores with correct values
for i = 2:size(Data,1)
    MainMasseyMatrix[i-1,teamsToIndex[Data[i,3]]] = 1
    MainMasseyMatrix[i-1,teamsToIndex[Data[i,4]]] = -1
end
for i = 2:size(Data,1)
    PMainMasseyMatrix[i-1,teamsToIndex[Data[i,3]]] = 1
    PMainMasseyMatrix[i-1,teamsToIndex[Data[i,4]]] = -1
```

```
end
for i = 2:size(Data,1)
    Scores[i-1,1] = Data[i,9] - Data[i,10]
    WeightedScores[i-1,1] = (Data[i,9] - Data[i,10])*(1 + (Data[i,5]-1)*(3/16))
end
#Perterbing the PMainMasseyMatrix
for i = 2:size(Data,1)
    for j = 1:32
        PMainMasseyMatrix[i-1,j] += 1*10^(-15)
    end
end
PMainMasseyMatrix
```

Out[5]:

```
256×32 Array{Float64,2}:
  1.0e-15  1.0e-15   1.0e-15  …  1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0       1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
 -1.0      1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15  …  1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15  -1.0      1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15  …  1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0      1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0         1.0e-15  1.0e-15   1.0e-15  1.0e-15
  ⋮                             ⋱                    ⋮
  1.0e-15  1.0e-15  -1.0         1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15  …  1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15  -1.0      1.0e-15
  1.0e-15  1.0e-15   1.0e-15  …  1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0       1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0e-15  1.0e-15   1.0e-15  1.0e-15
  1.0e-15  1.0e-15   1.0e-15     1.0      1.0e-15   1.0e-15  1.0e-15
  1.0      1.0e-15   1.0e-15  …  1.0e-15  1.0e-15   1.0e-15  1.0e-15
```

In [6]:

```
#Finding least Squares Approximation of equation Massey Matrix * ranks = Scores
MasseyRanks = inv(transpose(MainMasseyMatrix)*MainMasseyMatrix)*transpose(MainMasseyMatrix)*Scores
PMasseyRanks = inv(transpose(PMainMasseyMatrix)*PMainMasseyMatrix)*transpose(PMainMasseyMatrix)*Sc
ores
WeightedMasseyRanks = inv(transpose(MainMasseyMatrix)*MainMasseyMatrix)*transpose(MainMasseyMatrix)
*WeightedScores
using Plots
#show Massey Team Rankings
println("Massey Ranks 2009")
showrankings(teamsToIndex,MasseyRanks)
```

```
Massey Ranks 2009
MIA => 38.125
PHI => 35.25
IND => 27.8125
TEN => 24.8125
ATL => 24.625
NO => 22.6875
GB => 22.375
DAL => 19.0625
SD => 17.125
NYG => 16.6875
BUF => 15.3125
OAK => 13.3125
CIN => 13.0
BAL => 12.125
DEN => 8.75
NE => 8.3125
NYJ => 6.4375
CLE => 4.0
```

```
TB => 1.625
STL => -0.5625
WAS => -2.1875
CAR => -4.375
ARI => -16.75
DET => -16.875
CHI => -22.6875
SEA => -23.0
KC => -40.125
```

In [7]:

```
#show perturbed data
println("Perturbed Massey Ranks 2009")
showrankings(teamsToIndex,PMasseyRanks)
```

```
Perturbed Massey Ranks 2009
NO => 375.7477141818027
NE => 373.1930266818027
NYJ => 371.5367766818027
IND => 371.53287043180273
BAL => 371.15005793180273
GB => 370.50943293180273
MIN => 370.47037043180273
SD => 370.45865168180273
DAL => 369.15787043180273
CAR => 368.8922454318027
PHI => 368.7125579318027
ATL => 367.6266204318027
MIA => 366.2984954318027
PIT => 366.19302668180273
HOU => 365.98208918180273
CIN => 364.6930266818028
DEN => 363.8609954318027
SF => 363.1461516818027
ARI => 362.85708918180273
TEN => 362.70083918180273
NYG => 361.8649016818027
BUF => 361.0836516818027
CHI => 359.16568293180273
WAS => 357.81412043180273
TB => 357.6305266818027
JAC => 356.86490168180273
SEA => 356.37271418180273
CLE => 354.8453704318027
OAK => 353.2399016818027
KC => 353.02896418180273
DET => 348.43521418180273
STL => 346.0289641818027
```

In [8]:

```
#Plotting MasseyRanks vs Massey Perturbed data - the farther the scatter plot is from a line, the
greater significance perturbance has
plot(MasseyRanks, PMasseyRanks, seriestype=:scatter, title = "Affect of Perturbance on Massey
Ranks 2009", xlabel = "Massey raw ranks", ylabel = "Massey Perturbed Ranks")
```

Out[8]:

In [9]:

```
#Initializing PCA data
using LinearAlgebra
U,σ,V = svd(MainMasseyMatrix)
#U,σ,V = svd(PMainMasseyMatrix) Run This line instead of the one above for PCA on the perturbed Ma
trix
Eplus = zeros(Float64, 32,32)
EplusApproximate = zeros(Float64, 32, 32)
E = zeros(Float64, 32,32)
for x = 1:32
    Eplus[x,x] = 1/σ[x]
```

```
    E[x,x] = σ[x]
    if x<17
        EplusApproximate[x,x] = 1/σ[x]
    end
end
```

In [10]:

```
#Calculate PCA approximation for weighted and unweighted data
PCARanks = V*Eplus*transpose(U)*Scores
WeightedPCARanks = V*EplusApproximate*transpose(U)*WeightedScores
PCAApproximateRanks = V*EplusApproximate*transpose(U)*Scores
sum = 0;
Weightedsum = 0;
Approximatesum = 0;
for i = 1:32
    sum+= PCARanks[i,1]
    Weightedsum +=WeightedPCARanks[i,1]
    Approximatesum += PCAApproximateRanks[i,1]
end
avg = sum/32
Weightedavg = Weightedsum/32
Approximateavg = Approximatesum/32
for i = 1:32
    PCARanks[i,1] = PCARanks[i,1] - avg
    WeightedPCARanks[i,1] = WeightedPCARanks[i,1] - Weightedavg
    PCAApproximateRanks[i,1] = PCAApproximateRanks[i,1] - Approximateavg

end
```

In [22]:

```
#show PCA rankings
println("PCA rankings 2009")
showrankings(teamsToIndex,PCARanks)
```

```
PCA rankings 2009
NE => 10.0
NYJ => 9.5
NO => 9.0
GB => 8.0
BAL => 7.5
MIN => 6.5
PHI => 5.5
IND => 5.0
ATL => 4.5
CAR => 3.5
NYG => 3.0
CIN => 1.5
MIA => 0.5
PIT => 0.0
DEN => -0.5
SF => -1.0
TEN => -2.5
CHI => -3.5
WAS => -4.5
JAC => -8.0
SEA => -9.0
OAK => -9.5
KC => -10.0
DET => -13.5
STL => -18.5
```

In [12]:

```
#Plotting PCARanks vs PCA Aproximate Weighted data - the farther the scatter plot is from a line,
the greater significance weighting has
plot(PCARanks,PCAApproximateRanks,seriestype=:scatter, title = "Affect of Weighting and Rank K on
PCA data 2009", xlabel = "PCA Ranks", ylabel = "PCA Weighted Rank 16 Ranks")
```

Out[12]:

In [13]:

```
#plot Weighted Approximate PCARanks vs MasseyRanks, the closer this data is to a line, the more th
e two methods agree on rankings
plot(PCARanks, MasseyRanks,seriestype=:scatter, title = "PCA vs Massey Ranks 2009", xlabel = "PCA R
anks", ylabel = "Massey Ranks")
```

Out[13]:

In [14]:

```
#plot Weighted Approximate PCARanks vs MasseyRanks, the closer this data is to a line, the more th
e two methods agree on rankings
plot(PCARanks, PMasseyRanks,seriestype=:scatter, title = "PCA vs Perturbed Massey Ranks 2009",
xlabel = "Perturbed PCA Ranks", ylabel = "Massey Ranks")
```

Out[14]:

In [15]:

```
#Initialize Keener Matrix, and Helper scores Matrix
KeenerMain = zeros(Float64,32,32)
ScoreMatrix = zeros(Int64,32,32)
```

Out[15]:

```
32×32 Array{Int64,2}:
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 ⋮              ⋮              ⋮        ⋱  ⋮              ⋮              ⋮
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0
```

In [16]:

```
#Calculate Main Keener Matrix
for i = 2:size(Data,1)
    #populate scores matrix
    ScoreMatrix[teamsToIndex[Data[i,3]],teamsToIndex[Data[i,4]]] += Data[i,9]
    ScoreMatrix[teamsToIndex[Data[i,4]],teamsToIndex[Data[i,3]]] += Data[i,10]
end
for i = 2:size(Data,1)
    #calculate (A_{ij}+1)/(A_{ij} + A_{ji} + 2) given results from scores matrix
    ScoresTeam1 = ScoreMatrix[teamsToIndex[Data[i,3]],teamsToIndex[Data[i,4]]]
    ScoresTeam2 = ScoreMatrix[teamsToIndex[Data[i,4]],teamsToIndex[Data[i,3]]]
    point1 = (ScoresTeam1 + 1)/(ScoresTeam1 + ScoresTeam2 + 2)
```

```
    point2 = (ScoresTeam2 + 1)/(ScoresTeam1 + ScoresTeam2 + 2)
    #normalize data
    KeenerMain[teamsToIndex[Data[i,3]],teamsToIndex[Data[i,4]]] = (1/2) + ((sign(point1 - (1/2)))*(s
qrt(abs(2*point1 -1))))/2
    KeenerMain[teamsToIndex[Data[i,4]],teamsToIndex[Data[i,3]]] = (1/2) + ((sign(point2 - (1/2)))*(s
qrt(abs(2*point2 -1))))/2
end
#Sloppy notation here, I really should be doing a double for loop through every i,j index, but thi
s works too because any
#teams that play each other twice will just get reset to the same value at position i,j twice
#same process for weighted data
KeenerMainWeighted = zeros(Float64,32,32)
ScoreMatrixWeighted = zeros(Float64,32,32)
for i = 2:size(Data,1)
    ScoreMatrixWeighted[teamsToIndex[Data[i,3]],teamsToIndex[Data[i,4]]] += Data[i,9]*(1 + (Data[i,
5]-1)*(2/16))
    ScoreMatrixWeighted[teamsToIndex[Data[i,4]],teamsToIndex[Data[i,3]]] += Data[i,10]*(1 + (Data[i
,5]-1)*(2/16))
end
for i = 2:size(Data,1)
    ScoresTeam1 = ScoreMatrixWeighted[teamsToIndex[Data[i,3]],teamsToIndex[Data[i,4]]]
    ScoresTeam2 = ScoreMatrixWeighted[teamsToIndex[Data[i,4]],teamsToIndex[Data[i,3]]]
    point1 = (ScoresTeam1 + 1)/(ScoresTeam1 + ScoresTeam2 + 2)
    point2 = (ScoresTeam2 + 1)/(ScoresTeam1 + ScoresTeam2 + 2)
    KeenerMainWeighted[teamsToIndex[Data[i,3]],teamsToIndex[Data[i,4]]] += (1/2) + ((sign(point1 -
(1/2)))*(sqrt(abs(2*point1 -1))))/2
    KeenerMainWeighted[teamsToIndex[Data[i,4]],teamsToIndex[Data[i,3]]] += (1/2) + ((sign(point2 -
(1/2)))*(sqrt(abs(2*point2 -1))))/2
    #normalize data
end
```

In [17]:

```
#Calculate eigenvectors and eigenvalues
Keenervals, Keenervecs = eigen(KeenerMain)
KeenervalsWeighted, KeenervecsWeighted = eigen(KeenerMainWeighted)
```

Out[17]:

```
Eigen{Complex{Float64},Complex{Float64},Array{Complex{Float64},2},Array{Complex{Float64},1}}
eigenvalues:
32-element Array{Complex{Float64},1}:
    7.484518989834809 + 0.0im
   4.5001623566266655 + 0.0im
    4.421936442233129 + 0.0im
     3.56060768059194 + 0.0im
    2.545213140681959 + 0.0im
  -0.7041745668460162 + 1.1964550068886681im
  -0.7041745668460162 - 1.1964550068886681im
  -1.1823542723611946 + 1.0668993413610355im
  -1.1823542723611946 - 1.0668993413610355im
 -0.23787601638998296 + 0.9660868649069986im
 -0.23787601638998296 - 0.9660868649069986im
   0.2456824638886411 + 0.0im
 -0.44305865847774145 + 0.9597107597345892im
                      ⋮
  -0.7659331006001069 - 0.8097712993309971im
  -1.2626606740135204 + 0.29354211152498416im
  -1.2626606740135204 - 0.29354211152498416im
  -0.9621039006100025 + 0.48082700769183284im
  -0.9621039006100025 - 0.48082700769183284im
 -0.20879567650991337 + 0.0im
  -0.2712252216874335 + 0.0im
  -0.4800512752794581 + 0.29606673863092914im
  -0.4800512752794581 - 0.29606673863092914im
  -0.9481604120775583 + 0.09286564561265448im
  -0.9481604120775583 - 0.09286564561265448im
  -0.7466381153662018 + 0.0im
eigenvectors:
32×32 Array{Complex{Float64},2}:
  -0.220424+0.0im     0.18205+0.0im   -0.166956+0.0im  …    0.354603+0.0im
  -0.231844+0.0im    -0.152011+0.0im    0.103158+0.0im      0.0722618+0.0im
  -0.174605+0.0im    0.0276732+0.0im  -0.0808085+0.0im        0.11296+0.0im
  -0.226045+0.0im    0.0824117+0.0im   -0.142309+0.0im      -0.299906+0.0im
  -0.196386+0.0im    -0.259349+0.0im    0.259122+0.0im      -0.129186+0.0im
  -0.082252+0.0im   -0.0848873+0.0im   0.0909648+0.0im  …    0.237039+0.0im
```

```
  -0.15805+0.0im   -0.014765+0.0im   0.0490189+0.0im       0.111929+0.0im
 -0.198656+0.0im    -0.27477+0.0im    0.299152+0.0im     -0.0989065+0.0im
 -0.254321+0.0im    0.321553+0.0im   -0.305993+0.0im      -0.217102+0.0im
 -0.110663+0.0im   0.0931253+0.0im  -0.0976034+0.0im       0.248257+0.0im
 -0.198064+0.0im   -0.224173+0.0im      0.2488+0.0im  …    -0.11789+0.0im
 -0.162067+0.0im  -0.0159889+0.0im     0.06325+0.0im       0.262652+0.0im
 -0.162766+0.0im    0.209173+0.0im   -0.201125+0.0im     -0.0722145+0.0im
          ⋮                                          ⋱
 -0.117522+0.0im   -0.147634+0.0im    0.123243+0.0im  …   0.0895441+0.0im
 -0.198648+0.0im   -0.283192+0.0im    0.276838+0.0im       0.16169+0.0im
 -0.128491+0.0im   -0.153359+0.0im     0.17104+0.0im      -0.459053+0.0im
 -0.193608+0.0im    0.214714+0.0im   -0.223231+0.0im      -0.127332+0.0im
 -0.118828+0.0im   -0.125088+0.0im    0.122651+0.0im       0.115992+0.0im
-0.0723586+0.0im  0.00681359+0.0im   0.0056419+0.0im  …   -0.175229+0.0im
  -0.14501+0.0im    0.133171+0.0im   -0.104351+0.0im      -0.138521+0.0im
 -0.179362+0.0im    0.235973+0.0im   -0.226037+0.0im       -0.16601+0.0im
 -0.102463+0.0im   0.0159737+0.0im   0.0126404+0.0im       0.273692+0.0im
  -0.23454+0.0im    0.294203+0.0im   -0.320728+0.0im     -0.0994801+0.0im
 -0.129675+0.0im   -0.119964+0.0im    0.092051+0.0im  …    0.101116+0.0im
 -0.197739+0.0im    0.177923+0.0im   -0.198824+0.0im      -0.107466+0.0im
```

In [18]:

```
Keenersolution = zeros(Float64, 32, 1)
KeenersolutionWeighted = zeros(Float64, 32, 1)
```

Out[18]:

```
32×1 Array{Float64,2}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 ⋮
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

In [19]:

```
#Calculate solution vectors
for i = 1:32
  KeenersolutionWeighted[i] =convert(Float64, KeenervecsWeighted[i,1])
  Keenersolution[i] =convert(Float64, Keenervecs[i,1])
end
#make solution vectors have sum of one overall
sum = 0
sumWeighted = 0
for i = 1:32
  sumWeighted+=KeenersolutionWeighted[i]
  sum+=Keenersolution[i]
end
for i = 1:32
    KeenersolutionWeighted[i] = KeenersolutionWeighted[i]/sumWeighted
    Keenersolution[i] = Keenersolution[i]/sum
end
println("Keener 2009 predictions")
```

```
showrankings(teamsToIndex,Keenersolution)
```

```
Keener 2009 predictions
NO => 0.04448417448343444
NE => 0.04409103414172255
SD => 0.041475024332727234
NYJ => 0.04131262542107667
DAL => 0.04053851781463917
IND => 0.039904989836407084
GB => 0.03836155784338129
PHI => 0.038033151917279635
BAL => 0.037618546875070304
PIT => 0.0372150977796062
ATL => 0.03642275482594184
MIN => 0.036041199312961375
HOU => 0.03503614669028825
DEN => 0.034995167786062056
CAR => 0.033027586190638426
CIN => 0.03179049649030801
MIA => 0.03149025316469775
ARI => 0.030407361499862368
NYG => 0.030180561661230836
SF => 0.0294039373213542
TEN => 0.028777764176872674
BUF => 0.02760098720263444
JAC => 0.02513707094054378
WAS => 0.02506221093911111
CHI => 0.024273069708989754
KC => 0.023656068523359667
CLE => 0.021854687796847065
OAK => 0.021481223739572036
TB => 0.02046882620808455
SEA => 0.01852567131699633
DET => 0.016431405025446873
STL => 0.014900829032852104
```

In [20]:

```
plot(Keenersolution, KeenersolutionWeighted,seriestype=:scatter, title="Affect of Weighting on Keener Data", xlabel = "Keener Rankings", ylabel = "Keener Weighted Data Rankings")
```

Out[20]:

In [21]:

```
#plot Keener Ranks vs PCA, the closer this data is to a line, the more the two methods agree on rankings
plot(Keenersolution, PCARanks,seriestype=:scatter, title = "Keener vs PCA Ranks 2009", xlabel = "Keener Ranks", ylabel = "PCA Ranks")
```

Out[21]:

In [ ]:

```

```