21-241 Final Project – PageRank

Meher Mankikar and Bharat Narayanasamy
mmankika and bharatn

11 December 2020

1 Introduction

In this paper we will discuss Markov Matrices and Markov Chains and their application to algorithms that rank URLs in a link structure based on their connectivity. We first discuss the properties of Markov Matrices including eigenvalues and eigenvectors, diagonalization and matrix powers, and steady distributions. These concepts are then applied to implement three algorithms: Random Walks, PageRank, and the HITS Algorithm. The naive random walk starts at a vertex and then at each successive step picks one of its neighbors to move to. In creating PageRank, Google modified this algorithm through the inclusion of a damping factor. This damping factor accounts for users that stop searching as they are going through the web of links and therefore makes the algorithm more accurate for real world application. Finally, the HITS Algorithm is a modification of PageRank that defines nodes as authorities or hubs in order to find which nodes should be deemed "most important".

2 Eigenstuff and Diagonalization

EIGENVALUES AND EIGENVECTORS

Given a matrix A, $\vec{x}$ is an **eigenvector** if there exists a scalar **eigenvalue**, λ, such that $A\vec{x} = \lambda\vec{x}$.

To calculate a matrix's eigenvalue(s) and eigenvector(s), we do the following:

$$\begin{aligned} & A\vec{x} = \lambda\vec{x} \\ \implies & A\vec{x} - \lambda\vec{x} = 0 \\ \implies & A\vec{x} - \lambda I\vec{x} = 0 \\ \implies & (A - \lambda I)\vec{x} = 0 \end{aligned}$$

If $\vec{x} \neq 0$, we know that $A - \lambda I$ must be a singular or non-invertible matrix, since $\vec{x}$ is in the null space of $A - \lambda I$. Thus, to find the possible eigenvector $\vec{x}$ and eigenvalue λ, we solve for $det(A - \lambda I) = 0$. The polynomial when expanding $det(A - \lambda I)$ is also referred to as the **characteristic polynomial**.

Calculating the roots of the characteristic polynomial gives us all the eigenvalues, λ. Then, in order to find the eigenvectors, we can plug each of these values back into $A - \lambda I$ and find the basis for the null space.

DIAGONALIZABLE MATRICES AND MATRIX POWERS

An $n \times n$ matrix A is **diagonalizable** if it similar to a diagonal matrix.

In other words, if an $n \times n$ matrix A has n independent eigenvectors, $\{x_1, \cdots, x_n\}$ with eigenvalues $\{\lambda_1, \cdots, \lambda_b\}$ then A is diagonalizable.

If a matrix A is diagonalizable, it can be written as: $A = X\Lambda X^{-1}$

Where $X = (x_1|\cdots|x_n)$ and Λ is an $n \times n$ matrix with the eigenvalues on the diagonal and zeros in the other positions.

Matrix Powers

Diagonalization also makes computing matrix powers relatively simple.

Given that $A = X\Lambda X^{-1}$.

Then

$$\begin{aligned} A^n &= (X\Lambda X^{-1})^n \\ &= (X\Lambda X^{-1}) \cdots (X\Lambda X^{-1}) \\ &= X\Lambda^n X^{-1} \end{aligned}$$

So the n^{th} power of a matrix can be simply found by diagonalizing A and taking the diagonal matrix, Λ, to the n^{th} power.

3 Markov Matrices

DEFINITIONS:

An $n \times n$ matrix M is a **Markov Matrix** if:

1. $M_{ij} \geq 0$ for all i, j
2. $\sum_{i=1}^{n} M_{ij} = 1$ for all j

In other words, a matrix M is a Markov Matrix if all of its entries are greater than or equal to zero and each column sums to 1.

Note: A positive Markov matrix is one where $M_{ij} > 0$ for all i, j.

A **Markov chain** is a discrete-time stochastic process of n states that is defined in terms of a transition probability matrix, M.

- The transition probability of the current state only depends on the prior state
- All transition probabilities are ≥ 0 and must sum to 1
- $M_{ij} = \mathbb{P}$(transition from state i to state j)

PROPERTIES OF POSITIVE MARKOV MATRICES

- The largest eigenvalue $\lambda_1 = 1$
- All other eigenvalues: $|\lambda_i| < 1$
- The attracting steady state is the eigenvector corresponding to eigenvalue 1 (x_1)

- For large values of k, $M^k \to \begin{pmatrix} x_1 | \cdots | x_1 \end{pmatrix}$

STEADY STATE VECTORS

The steady state vector is a vector $\vec{v}$ such that for an $n \times n$ Markov matrix M, $M\vec{v} = \vec{v}$. This means that when we multiply an initial vector $\vec{x_0}$ by large powers of M, we will always approach the steady state vector.

For instance, let's say we were to attempt to determine the ratio of two countries' populations were after 1,000 years given the initial populations and a Markov transition matrix for moving from one country to another. Eventually the populations would come to a "steady state" where the number of people from each country that move out will be the same as the number of people from each country that move in. Thus, the steady state vector can be extremely useful in probability distributions in PageRank.

Indeed, to find the steady state vector, we must find the eigenvector with eigenvalue 1 of M. This is because the steady state vector is defined as a vector $\vec{v}$ such that $M\vec{v} = \vec{v}$. Then, by the definition of eigenvector and eigenvalue, for matrix M, 1 is an eigenvalue and $\vec{v}$ is the eigenvector. So, by solving for the eigenvector with eigenvalue 1 of matrix M, we can find the steady state vector for any Markov matrix.

4 Random Walks

A random walk is a stochastic process describing a path of successive steps. In this paper we will look at random walks in the context of directed graphs.

First we can define a directed graph, $G = (V, E)$ where V are the vertices of the graph and E are the edges. A random walk can then be represented by starting at some vertex, v_0. Then at each iteration, we move to one of the neighbors of the current vertex.

Random Walks with Markov Matrices

We can then use Markov Matrices to think about random walks. The initial position, $\vec{x}_0$, can be represented as a vector that shows the probability of being at point $i \in S$, where S is the state space.

Let $\vec{x_0} \in \mathbb{R}^n$, such that the elements $\vec{x_0}_i \geq 0$ and $\sum_i \vec{x_0}_i = 1$.

In addition, we define the $n \times n$ probability matrix P such that P_{ij} represents the probability that a person will end up on the i^{th} website at step n + 1 given that they were on the j^{th} website at step n. So

$$P_{ij} = \mathbb{P}(x_{n+1} = i | x_n = j)$$

In order to determine the probability of being at space i after one step, we can multiply $\vec{x_0}$ on the left by this matrix P. The resulting vector, $P\vec{x_0}$, is the new distribution.

Then, we may want to find the distribution vector at some step n, represented by $\vec{x}_n$. In order to do this, we use the fact that the random walk process is *memoryless*. Using this fact and the properties of Markov Matrices, it can be proven that $\vec{x}_n = P^n\vec{x}_0$ (Rousseau, 2015).

Stationary Distribution

For sufficiently large n, we can see that $\vec{x}_n$ begins to level off and eventually, $P\pi = \pi$ for some vector π. At this point, we can see that the probability of being on a page is independent of the page we started at. We call π the stationary distribution where π_i represents the probability of being at page i. π can be calculated by finding the eigenvector corresponding to $\lambda = 1$ for P (Rousseau, 2015).

5 Naive Random Walk without Markov Matrices

When implementing a random walk, we began by creating a random directed graph with 5 nodes.

```
# create graph with 5 vertices and 7 random edges
g = SimpleDiGraph(5, 7)
```

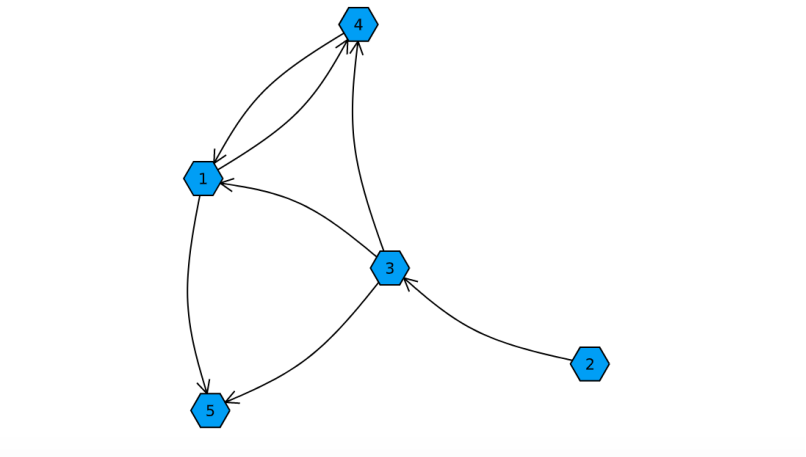

Figure 1. The random directed graph used in this implementation of a random walk. Each node represents a different website. The edges represent links between the pages.

We next implemented our random walk algorithm with the following function.

```
# NAIVE RANDOM WALK IMPLEMENTATION
# Starts at a random node on the graph
# At each iteration pick on of the edges of the current node to
   move to
# Currently running 11 steps

function randomWalk(g)
    rwp = [0 for i=1:5]
    vert = vertices(g)
    random = rand(1:5)
    rwp[random] += 1
    neighbors = outneighbors(g, random)

    z = 0
    while(z != 10)
        if (length(neighbors) == 0)
            focus = rand(1:5)
        else
            r1 = rand(1:length(neighbors))
            focus = neighbors[r1]
        end
        rwp[focus] += 1
        neighbors = outneighbors(g, focus)
        z += 1
    end
    return rwp
end
```

Running this method on the graph in Figure 1 resulted in the following output. The list below shows the number of times each node was visited during this random walk.

```
5-element Array{Int64,1}:
 Vertex 1: 3
 Vertex 2: 1
 Vertex 3: 3
 Vertex 4: 2
 Vertex 5: 2
```

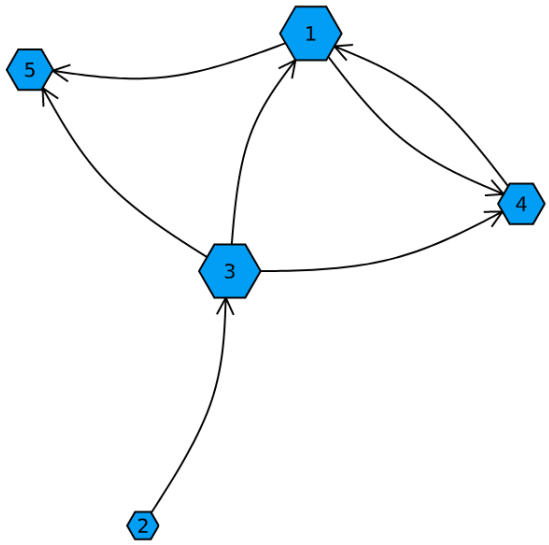

Figure 2. Shows the weighted graph of the random walk. Larger nodes were visited more frequently.

6 Data Preprocessing

To implement Random Walks, we first collected 5 different websites that formed a link structure.

```
# Data -- 5 connected webpages

url1 = "https://cs.cmu.edu"
url2 = "http://cmu.edu/"
url3 = "https://www.instagram.com/carnegiemellon/"
url4 = "http://www.ml.cmu.edu"
url5 = "https://www.cmu.edu/about/rankings.html"

urls = [url1, url2, url3, url4, url5]
```

Next, we found which of these webpages contains hyperlinks to the other webpages:

```
# Helper function gets all the links from the webpage

function getLinkList(url)
    resp = HTTP.get(url)
    respText = String(resp.body)
    links = collect(m.match for m in eachmatch(r"http[s]?://(?:[a-zA-Z]|[0-9]|[-_@.&+]|[!*\(\),]|(?:[0-9a-fA-F][0-9a-fA-F]))+", respText))
    return links
end
```

This helper function allows us to scour each URL and return a list of all the URLs hyperlinked to it. Then we can form a list of hyperlinks indexed by each URL; for each URL (urls[i]), we store the list of outneighbors at lists[i].

```
# Creates list of links on each page
function listOfLinks()
    lists = Any[]
    for i in 1:length(urls)
        push!(lists, getLinkList(urls[i]))
    end
    lists
end
```

To create the probability matrix P, we created a 5×5 matrix, with each row corresponding to a URL in our dataset. We then updated the matrix such that P_{ij} equals 1 if urls[j] is in lists[i]. Afterwards, we divide each term by the sum of the terms in its column so that P is now a Markov Matrix.

```
# Creates the probability matrix P

function createP()
    # Creates the probability matrix P

    P = zeros(Float64, length(urls), length(lists))

    for i in 1:length(lists)
        for j in 1:length(urls)
            if i == j
                continue
            end
            if urls[j] in lists[i]
                P[j,i] = 1
            end
        end
    end
    sum1 = sum(P,dims=1)


    for i in 1:length(urls)
        for j in 1:length(lists)
            if sum1[i] == 0
                break
            end
            P[j,i] = P[j,i]/sum1[i]
        end
    end
    P
end
```

Now, we have all the data we need to implement random walks with Markov matrices!

7 Naive Random Walk with Markov Matrices

In order to implement this version of a naÃ¯ve random walk, we first initialized the distribution vector $\vec{x}_0$.

```
# Initialize X_0 so that all the values are 1/n
function initializeVec()
    X_0 = zeros(Float64, length(urls))
    fill!(X_0, 1/length(urls))
    X_0
end
```

And then implemented the random walk algorithm.

```
function randWalk()
    i = 0
    X = P * X_0
    while i < 150
        X = P * X
        if i % 25 == 0
            print("X_#i: ")
            println(X)
        end
        i += 1
    end
end
```

The output of this function was as follows:

```
X_0: [0.1, 0.4, 0.30000000000000004, 0.1, 0.1]
X_25: [2.44140625e-5, 0.39997558593750004, 0.5999755859375, 1.220703125e-5, 1.220703125e-5]
X_50: [2.9802322387695314e-9, 0.5999999940395355, 0.3999999970197678, 2.9802322387695314e-9, 2.9802322387695314e-9]
X_75: [7.275957614183426e-13, 0.39999999999927244, 0.5999999999992724, 3.637978807091713e-13, 3.637978807091713e-13]
X_100: [8.881784197001253e-17, 0.5999999999999999, 0.39999999999999997, 8.881784197001253e-17, 8.881784197001253e-17]
X_125: [2.168404344971009e-20, 0.4, 0.6, 1.0842021724855045e-20, 1.0842021724855045e-20]
```

As can be seen from this output, for $n = 125$, the steady state vector began to approach $(0, 0.4, 0.6, 0, 0)^T$. However, if we look at when $n = 100$, we can see that we get components that seem to approach $(0, 0.6, 0.4, 0, 0)^T$. In fact, each timestep our vector alternates between $(0, 0.4, 0.6, 0, 0)^T$ and $(0, 0.6, 0.4, 0, 0)^T$. We see that the reason behind this is because the only URL in list[2] is url3 and the only URL in list[3] is url2. So, we must average the values of 0.4 and 0.6 since each vector alternates through both every iteration; we then get the steady state vector $(0, 0.5, 0.5, 0, 0)^T$.

We then used the method of diagonalization described in Section 2 to calculate the steady state vector of P and confirm these results. As expressed in the Section 3, we can find the steady state by finding eigenvector of P with eigenvalue 1.

```
function steadyState(P)
    evalue, evec = eigen(P)
    return evec[:,n]
end
```

Which outputed the steady state vector:

```
5-element Array{Float64,1}:
 -6.206335383118171e-16
  0.5000000000000001
  0.5
 -3.2272943992214487e-16
 -3.2272943992214487e-16
```

As we can see, this vector is approaching $(0, 0.5, 0.5, 0, 0)^T$, which confirms the results of our random walk implementation.

8 Necessary Modifications to Naive Random Walks in PageRank

In the implementation of random walks, users were expected to continue to click on links for the number of iterations that we specified. This, however, does not take into account the fact that a user may simply want to quit from the page and not search further for links. The inclusion of this factor leads to more accurate results as humans are certainly more likely to quit from a page; this brings up the idea of a **damping factor**.

A damping factor is the probability that the user leaves the site altogether without clicking on the next webpage. For their implementation of PageRank with a damping factor, Google assigned $\beta = 0.15$. In this scenario, the algorithm resets to an equal probability to choose each webpage to start on.

Now, to implement PageRank with the damping factor at each iteration we update P_B to be: (Rousseau, 2015)

$$P_B = ((1 - \beta) * P) + (\beta) * Q$$

Where P = the probability matrix and Q = $n \times n$ matrix where all entries are $\frac{1}{n}$.

9 PageRank Implementation

To implement PageRank, we first initialize Q so that all entries are $\frac{1}{n}$.

```
# Creates Q which is initialized with all values being 1 / n
function createQ()
    Q = zeros(Float64, length(urls), length(lists))
    numNodes = length(urls)
    fill!(Q, 1/numNodes)
    return Q
end
```

Next, we initialize P_B so that each entry, $(P_B)_{ij}$, represents the probability that the user will be at page j given that they were at page i previously.

```
# Creates P matrix with damping factor Beta
function createPB()
    P_B = ((1 - Beta) * P) + (Beta * Q)
    return P_B
end
```

Now, with the help of a damping factor, we will get results that are more plausible. The following function allows us to calculate a steady state that does not contain 0's as the previous example did; therefore, this is a better representation of the steady state vector.

```
# Implements PageRank algorithm
function pagerank()
    i = 0
    X = P_B * X_0
    while i < 100
        X = P_B * X
        if i % 25 == 0
            print("X_#i: ")
            println(X)
        end
        i += 1
    end
    Xsort = sort(X, rev=true)
    used = []
    for i in 1:length(X)
        ind2 = findall(x->x == Xsort[i], X)
        ind = findfirst(x->x == Xsort[i], X)

```

```
        for j in 1:length(ind2)
            if(!(ind2[j] in used))
                println("Rank #i: (urls[(ind2[j])])")
                push!(used, ind2[j])
                break
            end
        end
    end
end
```

Running this algorithm outputs the ranks of each website.

```
X_0: [0.12775, 0.3445000000000001, 0.29775000000000007, 0.115, 0.115]
X_25: [0.08688865585259019, 0.3731109555009213, 0.40614503117178685, 0.06692767873735082, 0.06692767873735082]
X_50: [0.08688845401210182, 0.37427255857285874, 0.4049838015042113, 0.06692759295541392, 0.06692759295541391]
X_75: [0.08688845401174167, 0.37425258449547427, 0.40500377558280354, 0.0669275929549902, 0.06692759295499019]
Rank 1: https://www.instagram.com/carnegiemellon/
Rank 2: http://cmu.edu/
Rank 3: https://cs.cmu.edu
Rank 4: http://www.ml.cmu.edu
Rank 5: https://www.cmu.edu/about/rankings.html
```

We can see that when we get to the vectors X_{50} and X_{75}, which can be written as $(P_B)^{50} * X_0$ and $(P_B)^{75} * X_0$ respectively, the components of the vectors seem to be stagnant, meaning that we are closer and closer to a steady state.

So, we can then take our steady state vector π to be $\begin{pmatrix} 0.0868 \\ 0.3743 \\ 0.4050 \\ 0.0669 \\ 0.0669 \end{pmatrix}$.

These results can be verified by diagonalizing P_B and then finding the eigenvector for eigenvalue 1 as we did for the naive random walk implementation.

We built the following function to do so:

```
# Finds the Steady State of P_B
function steadyState1(P_B)
    evalue, evec = eigen(P_B)
    Λ = Diagonal(evalue)
    N = evec
    Ninv = inv(N)
    return evec[:,n]
end
```

This takes in a matrix P_B as the parameter, calculates the eigenvectors and eigenvalues, and returns the eigenvector for the largest eigenvalue, which will always be 1 for Markov matrices.

Running this function results in the following:

```
5-element Array{Float64,1}:
 -0.15345444359686275
 -0.6609712946745814
 -0.7152800440702566
 -0.11820139574352939
 -0.11820139574352939
```

Now, to make all the components in the vector add up to 1, we will sum up the components and divide each component by the sum:

```
# Manipulates steady state vector so values sum to 1
function getSteadyState2(π)
    s = sum(π)
    π = π / s
    return π
end
```

By calling this function on the result above, we get the same result as we did in our PageRank function.

```
5-element Array{Float64,1}:
 0.08688845401174161
 0.3742529221981279
 0.40500343788015014
 0.06692759295499015
 0.06692759295499015
```

Thus, we are able to use damping to produce a more accurate PageRank algorithm and rank websites based on the probability that a user will be at each webpage at a certain point in time. We have shown 2 ways here: using Julia to take large matrix powers and finding the eigenvector for eigenvalue 1. Both ways reveal the same answer for properly ranking websites through this algorithm!

10 HITS Algorithm

The Hypertext Induced Topics Search (HITS) Algorithm is an extension of the PageRank Algorithm implemented above. HITS has the same purpose of taking in a link structure of webpages and outputting ranks for the pages based on their connectivity. The HITS Algorithm defines two types of important nodes: hubs and authorities. Hubs are nodes which point to many important nodes. Authorities are these important nodes that have many links to them. In this way, the definitions of hubs and authorities is relatively circular (Benzi et al, 2012).

What follows is the list of URLs that we used in this implementation of the HITS Algorithm.

```
url1A = "https://en.wikipedia.org/wiki/Carnegie_Mellon_University"
url2A = "https://en.wikipedia.org/wiki/
   Carnegie_Mellon_School_of_Computer_Science"
url3A = "https://en.wikipedia.org/wiki/Human-
   Computer_Interaction_Institute"
url4A = "https://en.wikipedia.org/wiki/Pittsburgh"
url5A = "https://en.wikipedia.org/wiki/
   Carnegie_Mellon_University_traditions"
url6A = "https://en.wikipedia.org/wiki/
   List_of_Turing_Award_laureates_by_university_affiliation"
url7A = "https://en.wikipedia.org/wiki/Robotics"
```

In order to begin the implementation of the HITS Algorithm, we first defined A to be the adjacency matrix representing the link structure of the webpages.

$$(A)_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$

```
7x7 Array{Int64,2}:
 0  1  1  1  1  1  0
 1  0  1  1  1  0  1
 1  1  0  1  1  0  0
 1  1  0  0  0  0  1
 1  1  1  1  0  0  0
 1  0  0  0  0  0  0
 1  0  0  0  0  0  0
```

Thereafter, we initialized x_0 and y_0 to be the initial authority weights and hub weights of each vertex respectively. The initial values of $(x_0)_i$ and $(y_0)_i$ for all $0 \leq i \leq n$ were set to be $1/\sqrt{n}$.

```
function createVecs()
    x_0 = zeros(Float64, length(urlsA))
    numNodes = length(urlsA)
    fill!(x_0, 1/sqrt(numNodes))
    return x_0
end
```

```
7-element Array{Float64,1}:
 0.3779644730092272
 0.3779644730092272
 0.3779644730092272
 0.3779644730092272
 0.3779644730092272
 0.3779644730092272
 0.3779644730092272
```

At each iteration, the authority score of each vertex is updated to the sum of the hub scores of all its neighbors. In addition, the hub score of each vertex is updated to equal the sum of the authority scores of all its neighbors. This can be expressed as

$$\vec{x}_k = A^T \vec{y}_k \text{ and } \vec{y}_k = A\vec{x}_k$$

Therefore the process that occurs at each iteration is as follows.

$$\vec{x}_k = c_k A^T A \vec{x}_{k-1} \text{ and } \vec{y}_k = c_k' A A^T \vec{y}_{k-1}$$

Our implementation of the HITS Algorithm follows.

```
function hits()
    x_k = normalize(transpose(A1) * (A1) * x_0)
    y_k = normalize(A1 * transpose(A1) * y_0)
    i = 0
    while i < 100
        x_k = normalize(transpose(A1) * A1 * x_k)
        y_k = normalize(A1 * transpose(A1) * y_k)
        i += 1
    end
    return x_k, y_k
end
```

Using the URL link structure shown above, the authority and hub distributions after 100 iterations were

```
x_100 = [0.5100828571185775, 0.43116815209447085,
   0.3640946773280714, 0.4830588696890527, 0.36409467732807144,
   0.11703846508823482, 0.20429332216312873]

y_100 = [0.4537883802440175, 0.4966458691951298,
   0.4612549226258964, 0.2954521448140663, 0.46125492262589646,
   0.1315576098111577, 0.1315576098111577]
```

As we can see from the explanations in Section 2 and 3, the steady distribution for x_k can be represented by the dominant eigenvector for A^TA. Similarly. the steady distribution for y_k can be represented by the dominant eigenvector for AA^T (Benzi et al, 2012).

Calculating these values we see that the dominant eigenvectors for A^TA and AA^T are as follows.

```
A^TA (steady distribution for x_k):
7-element Array{Float64,1}:
 0.5100828571185776
 0.43116815209447085
 0.36409467732807144
 0.48305886968905276
 0.3640946773280714
 0.1170384650882349
 0.204293322163129

AA^T (steady distribution for y_k):
7-element Array{Float64,1}:
 0.4537883802440176
 0.49664586919512976
 0.4612549226258963
 0.2954521448140661
 0.4612549226258963
 0.13155760981115738
 0.13155760981115783
```

Since the dominant eigenvectors for A^TA and AA^T are the same as the steady distributions received in our HITS Algorithm function, we can confirm our results!

References

[1] Arora, S. (2013, November). *Random walks, Markov chains, and how to analyse them.* https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec12.pdf.

[2] Benzi, M., Estrada, E., and Klymko, C. (2013, March 1). *Ranking Hubs and Authorities Using Matrix Functions.* http://www.mathcs.emory.edu/~benzi/Webinar_papers/rankinghubs.pdf.

[3] Rousseau, C. (2010, August 5). *How Google works.* http://dmuw.zum.de/images/f/f8/Google_klein_2.pdf.

[4] Spielman, D. A. (2018, October 1). *Random Walks on Graphs - Yale University.* Random Walks on Graph. https://www.cs.yale.edu/homes/spielman/561/lect10-18.pdf.