21-241 Final Project Paper

Emma Cohron and Michael OBroin

December 14, 2018

Contents

1 Introduction

We have implemented a version of the PageRank algorithm in Python using Markov chains and eigenvectors to determine the rank of a list of vertices given the connections between them. The problem of the PageRank algorithm asks us to rank a list of web pages, or more generally vertices, from greatest to least importance based on the probability of visiting each page by clicking links randomly, or more generally taking a "random walk," between the pages given the links, or connections, between each page or vertex. Our solution involves predicting the outcome of a random walk using linear algebra and displays the result as a highest-to-lowest rank list of the vertices.

2 Definitions and Notation

Throughout this paper n refers to the number of vertices in the input list.

Throughout this paper P refers to the $n \times n$ probability matrix where P_{ij} represents the probability of visiting the i^{th} vertex directly after visiting the j^{th} vertex.

Throughout this paper $totalConnections_i$ refers to the number of vertices that connect from vertex i to other vertices in the graph.

Throughout this paper a Markov probability matrix is considered *regular* if it fulfills two key characteristics:

1. It has an eigenvalue of 1 with algebraic multiplicity 1.
2. All other eigenvalues are less than 1.

A value λ is an *eigenvalue* of P if there exists a nonzero vector such that $Pv = \lambda v$. Any vector v that satisfies this for a given λ is called an *eigenvector*.[1]

[1]See Bibliography: Source 3

3 Mathematical Theorems

3.1 Overview

First, we have defined a function called *pageRank* which takes input that describes the connections between a set of vertices in the form of a list containing a list for each vertex of the other vertices it is connected to, and outputs a list ranking, from highest to lowest, each vertex by the probability of visiting it starting from any other vertex. The function uses the lists of connections input to create P. Next, after applying a damping factor to the probability matrix, we used the Perron-Frobenius Theorem to get the eigenvector for the probability matrix corresponding to an eigenvalue of 1. Finally, we used this vector to enter each vertex into a ranking list in order of greatest to least probability of being visited on a "random walk." This is the final output list.

3.2 Populating the Probability Matrix

There are several steps required to build the original probability matrix. We will first describe the mathematical representation of and reasoning behind this portion of the program, and will subsequently describe it's Python representation.
The first step in building the probability matrix is filling in zeros in every P_{ij} where it is impossible to visit the i^{th} vertex from the j^{th} vertex. This is all the positions where the i^{th} and j^{th} vertices have no edge connecting them. The next step is, at the P_{ij} location of any connect vertices i and j, to insert the probability of traveling to the j^{th} vertex from the i^{th} vertex. Given that it is equally as likely to travel to any vertex that i connects to from i, the probability of traveling to vertex j if vertex i connects to vertex j is $1/(totalConnections_i)$. Thus, the probability matrix is filled as each index either holds a 0 where no connection exists or a probability value for the likelihood of travelling from one vertex to another, as follows:

$$P_{ij} = \begin{cases} \frac{1}{totalConnections_j} & \text{if there is a link from j to i} \\ 0 & \text{otherwise} \end{cases}$$

The python representation of building this matrix begins with declaring a two-dimensional list of size $n \times n$. This list is originally filled with zeros, so it is initialized representing the zero matrix. Next, a nested for loop is used to traverse every index where there is a connection from the vertex represented by a column index, j, to the vertex represented by a row index, i. At each of these indices the zero that is originally in that position is replaced with $1/(totalConnections_i)$. Hence, the mathematical and python representations of the probability matrix match exactly.

3.3 Applying the Google Form

Once the probability matrix is populated with the initial probability values we apply a series of transformations to it. This is necessary because we want the probability matrix to be regular so that it has an eigenvalue of 1. This becomes necessary further down the line in calculating the ranking of the vertices. Note the following theorem[2]:

> We can replace the matrix P by the matrix P_{alpha} such that we multiply P by $alpha$ and then add $\frac{1-alpha}{n}$ to each value in P. Given any Markov transition matrix P, there always exists a positive $alpha$, as small as we wish, such that the matrix P_{alpha} is regular.

We have chosen to use the same $alpha$ value as Google uses in their implementation of PageRank, hence why this is called the Google Form. Their default value for $alpha$ is 0.85.
In our python program for this step we use a simple nested for loop to visit each element in the matrix and apply the transformation. For each element we first multiplied by $alpha$ and then added $\frac{1-alpha}{n}$, which is mathematically equivalent to multiplying the entire matrix by $alpha$, as scalar multiplication of a matrix is distributive to each element, and then adding $\frac{1-alpha}{n}$ to each element.

3.4 Frobenius Theorem

Next, we can use the power method to find the eigenvector that will help us generate our ranking.
Consider the following proof adapted from Rousseau[3]:
Let X_w be a random variable that represents the vertex we have reached after a random walk of length n. Since p_{ij} represents the conditional probability that we have gone to the i^{th} page on step $w+1$ from the j^{th} page where we were at step w, then the following is true:

$$p_{ij} = Prob(X_{n+1} = i | X_n = j)$$

[2]See Bibliography: Source 3
[3]See Bibliography: Source 3

Using this, we can go a step backwards:

$$Prob(X_{n+2} = i | X_n = j) = \sum_{k=1}^{N} Prob(X_{n+2} = i \wedge X_{n+1} = k | X_n = j)$$

(Law of Total Probabilities)

$$= \sum_{k=1}^{N} \frac{Prob(X_{n+2} = i \wedge X_{n+1} = k \wedge X_n = j)}{Prob(X_n = j)}$$

(Definition of Conditional Probability)

$$= \sum_{k=1}^{N} \frac{Prob(X_{n+2} = i \wedge X_{n+1} = k \wedge X_n = j)}{Prob(X_n = j)} \cdot \frac{Prob(X_{n+1} = k \wedge X_n = j}{Prob(X_n = j)}$$

$$= \sum_{k=1}^{N} Prob(X_{n+2} = i | X_{n+1} = k) \cdot Prob(X_{n+1} = k | X_n = j)$$

(Definition of Conditional Probability)

$$= \sum_{k=1}^{N} p_{ik} p_{ij}$$

$$= (P^2)_{ij}$$

Through iteration we can see that the entry $(P^m)_{ij}$ of the matrix P^m represents $Prob(X_{n+m} = i | X_n = j)$. Thus, after n steps, where n is sufficiently large, the probability of being at a given vertex is independent from where the random walk began.
Next, we want to find the stationary distribution vector that will ultimately allow us to rank the pages. This is a vector such that $Pv = v$, and is an eigenvector for the eigenvalue of 1.
Recall the Frobenius Theorem[4]:

> If we have an nxn Markov transition matrix P where $p_{ij} \in [0, 1]$ for all i, j, and the sum of the entries of each column is 1, then
>
> 1. $\lambda = 1$ is one eigenvalue of P.
> 2. Any eigenvalue of P satisfies $|\lambda| \leq 1$.
> 3. There exists an eigenvector v of the eigenvalue 1, all the coordinates of which are greater than or equal to zero. The sum of the coordinates are 1.

Consider the following proof adapted from Rousseau[5]:
Let the eigenvector basis for P be denoted $B = \{v_1, ..., v_n\}$ where v_1 is the vector v of the Frobenius Theorem. For each v_i there exists a λ_i such that $Pv_i =_i v_i$ by definition of eigenvector. Let X be the vector $X = (x_1, ..., x_n)$ where each

[4]See Bibliography: Source 3

[5]See Bibliography: Source 3

$x_i = \frac{1}{n}$. Note that each coordinate of X is less than 1 and the sum of all the coordinates of X is n. We can write X in terms of the basis B:

$$\begin{aligned}
X &= \sum_{i=1}^{N} a_i v_i \\
\implies PX &= \sum_{i=1}^{N} a_i P v_i = \sum_{i=1}^{n} a_i \lambda_i v_i && \text{(Definition of eigenvalue)} \\
\implies P^n X &= \sum_{i=1}^{n} a_i \lambda_i^n v_i && \text{(See proof above)} \\
\implies lim_{n\to\infty} P^n X &= a_1 v_1 && \text{(See * below)} \\
&= v
\end{aligned}$$

*Note that by a technical proof that we will not expound upon here, $a_1 = 1$. Hence, the more we multiply our matrix with itself, the closer we will get to an accurate estimate of our eigenvector for eigenvalue 1. After taking the limit to infinity of the matrix multiplied by itself, we have that the columns of the matrix each converge to the same values. Note that as a result of the Frobenius theorem, each column of our matrix now represents the eigenvector for eigenvalue 1. This column vector is referred to as the stationary distribution vector.

In python, we evaluate this using a while loop. In the loop, we set the probability matrix equal to itself multiplied by itself. The while loop continues until the maximum difference between the norms of any two adjacent columns of the matrix is less than the threshold amount. The threshold amount is equal to the machine precision of the float data type in python, approximately $2.2 * 10^{-16}$.

Once we have our sufficient approximation of $\lim_{n\to\infty} P^n$, we then calculate the stationary vector. We do this by initializing a n-vector filled with each entry $\frac{1}{n}$, and then multiplying it by our probability matrix.

3.5 Ranking

Once we have our stationary vector, ranking is straightforward. The nodes are ranked according to their corresponding entry in the stationary vector, going from greatest to least.

In python, we do this destructively. First, we make a copy of our stationary vector, represented as a list and an empty list, ranked, in which we will store our ranking of the nodes. We then have a nested for loop, iterating over the copy on the outside and the original on the inside. We then loop through to find the greatest value in the original list, find its index in the copy, append that value to the list of ranks, and then remove that value from the original list. Once the loop terminates, we have our ranked list, which has the indices of the original input in order with respect to their importance.

4 Conclusion

The importance of solving this problem spans many different fields and applications. The most obvious use of Markov chains in this manner is the official PageRank algorithm used by Google. Google, and other search engines like it, use the algorithm when determining the order in which to present results from a search. Those pages which are most likely to be clicked and linked to from other sites are ranked most highly, as they are assumed to be most useful, and are therefore displayed at the top of the results page. Constrastingly, computational biologists have used a version of this algorithm to rank species in a given ecosystem by their importance to predict which are most likely to cause a collapse of the ecosystem if they were to go extinct. These are just two examples of how wide-ranging the concepts behind this algorithm truly are, however there are so many more in use today and likely many more uses that have yet to be discovered.

In reality, when using PageRank for search results, the number of elements in the probability matrix would number in the millions, with columns and rows corresponding to different websites and the entries the links between them. This was the first method Google used to rank its search results, and though it is no longer in use in the format described in this paper, methods since have been based on it. It is because of this practical use of this algorithm that an alternate method for solving the problem, directly calculating the eigenvectors, was not used. Calculating eigenvectors is more computationally complex than simply multiplying matrices, which becomes a very practical issue when used as described above to rank websites for search results. Most eigenvalue decomposition functions have a $O(n^3)$ run time, while the matrix multiplication method runs in approximately $O(n^\omega)$ where $2 < \omega < 2.376$.[6]

Regardless of which method is used the time complexity of finding eigenvalues and vectors is relatively large. Further, the space complexity of either method is about $O(n^3)$, which is unfavorable[7]. This issue has caused even Google, who first produced this version of the algorithm, to move on to move sophisticated methods, though they are still based on this algorithm. Further, the spacial cost of running these types of algorithms is high and only adds to their need for increasingly massive server farms across the US. This is an unsolved problem currently standing in the way the widespread use of these very valuable linear-algebra-based tools for developing solutions to other similar problems. While the time complexity would be difficult to reduce, it is possible that the space complexity could be reduced in other languages or with updated algorithms. This is something that would be valuable to explore further.

[6]See Bibliography: Source 6
[7]See Bibliography: Source 6

5 Bibliography

1. Leggett, Hadley. “Google Algorithm Predicts When Species Will Go 404, Not Found.” Wired, 4 Sept. 2009. Retrieved December 11, 2018, from www.wired.com/2009/09/googlefoodwebs/.

2. Manning, Christopher D, et al. Introduction to Information Retrieval. Cambridge University Press, 2008. Retrieved December 9, 2018, from https://nlp.stanford.edu/IR-book/html/htmledition/markov-chains-1.html.

3. Rousseau, Christiane. “How Google Works: Markov Chains and Eigenvalues.” Klein Project Blog, 30 May 2015. Retrieved December 12, 2018, from blog.kleinproject.org/?p=280.

4. Shum, Kenneth. “Notes on PageRank Algorithm.” ENGG2012B Advanced Engineering Mathematics Lecture 13. Retrieved December 9, 2018, from home.ie.cuhk.edu.hk/ wkshum/papers/pagerank.pdf.

5. Tanase, Raluca, and Remus Radu. “Lecture #3: PageRank Algorithm - The Mathematics of Google Search.” The Mathematics of Web Search. Retrieved December 9, 2018, from pi.math.cornell.edu/ mec/Winter2009/RalucaRemus/index.html.

6. Demmel, James, et al. “Fast Linear Algebra Is Stable.” Numerical Analysis, vol. 108, no. 1, 28 Aug. 2007. Cornell University Library. Retrieved December 10, 2018, from arxiv.org/abs/math/0612264.