Google PageRank with Markov Chains

Christina Lau, Amy Li
21-241 Linear Algebra, Professor Radcliffe

INTRODUCTION

When a user does a Google search, in order to determine which website page the user would be most interested in, links between pages can be evaluated to determine the reputation and importance of each page. The more a page is linked to, the more that page has been endorsed. The user would likely be more interested in the more endorsed pages. This was the idea behind PageRank, which Google founders Larry Page and Sergey Brin created in 1996. By 1998, PageRank had become the foundation of Google Search results, and the prototype was introduced to the public [6].

Before PageRank, search engines used text based ranking systems, so that the 1st ranked page would be the page that contained the user search word most frequently. The flaw is clear: it is unlikely that just because a page contains a search word more times, it is the page the user is looking for. The new idea of ranking by reputation thus made PageRank very powerful. With PageRank, Google could conduct an initial search for all web pages which contain the user search word, and then use PageRank to rank based on reputation as opposed to word count [5].

Since the initial prototype, Google has changed the specific implementation of its ranking systems many times. This was necessary not only to improve results so they are specific to unique users, but also because many companies exploited the known PageRank algorithm to boost their own rankings. In 2009, Google had deleted PageRank information from its Webmaster Tools / Search Console. Nonetheless, PageRank likely remains at least a small part of a Google Search's returned result. In this paper, we will look at the mathematical concepts behind PageRank and how they justify our implementation of PageRank [6].

DEFINITIONS

PageRank

A method for ranking a system of connected web pages based on their relative importance. The importance of each web page is based on structure of the links between pages. Each link that goes to a web page increases the relative importance of that page. Furthermore, a page that has less outgoing links will add more importance each page that is linked. The more important a web page is, the higher it is ranked.

Markov chain

A way to represent a system that changes according to given probabilities. The probability of transitioning to a future state of the system is determined only by the system's current state, and not by any previous states. In the graphical representation of a Markov chain, each possible state of the system is represented by a node, and the possible transitions between states are represented by arrows from one node to another [7].

In our PageRank algorithm, we will use a graphical representation of a Markov chain to represent the web page traversal of a web user. Each web page is represented as one node. Each hyperlink that takes the user from one web page to another is represented as an outgoing arrow from the node representing the current web page to the node representing the next web page. See for example, Figure 1 on page 4.

Valency

Number of outgoing links of a given node.

Stationary Distribution

A probability distribution for a Markov chain that remains the same at all points in time.

Pagerank vector

A vector with entries as the pageranks of the possible webpages to visit. The pagerank vector represents a stationary distribution of the Markov chain [1].

Probability Matrix

In order to easily compute at each step the probability of leaving a page X for another page Y, we summarize the network using a probability matrix. Let n be the total number of pages. Then we create a nxn probability matrix P. Each column represents the page we leave, and each row represents the page we arrive at. We can thus define P by defining the (i, j) entry of P:

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

Dangling node

A node in a Markov chain with no outgoing arrows. In our PageRank application of a Markov chain, a dangling node represents a web page which has no outgoing links to other web pages [5].

Damping factor

A fixed constant between 0 and 1. In our PageRank application of a Markov chain, this constant represents the small probability that the random web surfer will jump to a random page from their current page instead of clicking through a link [3].

Column-stochastic matrix

A square matrix in which all the entries are non-negative and the sum of the entries in each column is equal to 1. A column-stochastic matrix can be used to represent the transitions between states in a Markov chain [4].

Non-stochastic matrix

A matrix that is not column-stochastic [4].

Reducible (Markov chain)

A Markov chain in which it is not possible to get from any state in the system to every other state. This means that starting from a given node in a Markov chain graph, there is not necessarily a path to get to every other node [4].

Perron-Frobenius Theorem

Let B be an $n \times n$ matrix with nonnegative real entries. Then we have the following:

(1) B has a nonnegative real eigenvalue. The largest such eigenvalue, $\lambda(B)$, dominates the absolute values of all other eigenvalues of B. The domination is strict if the entries of B are strictly positive.

(2) If B has strictly positive entries, then $\lambda(B)$ is a simple positive eigenvalue, and the corresponding eigenvector can be normalized to have strictly positive entries.

(3) If B has an eigenvector v with strictly positive entries, then the corresponding eigenvalue λv is $\lambda(B)$. [2]

Convergence

The property that repeatedly multiplying a matrix A by a non-negative vector x will result in a stationary distribution. This is useful for our PageRank algorithm, as repeatedly taking random walks (using a probability matrix) would result in a stationary distribution independent of where one starts. Note that a column-stochastic matrix with columns summing to 1 will converge [4].

MAIN IDEAS

MARKOV CHAIN STRUCTURE

Here is the example Markov Chain we will use to demonstrate how our algorithm works:

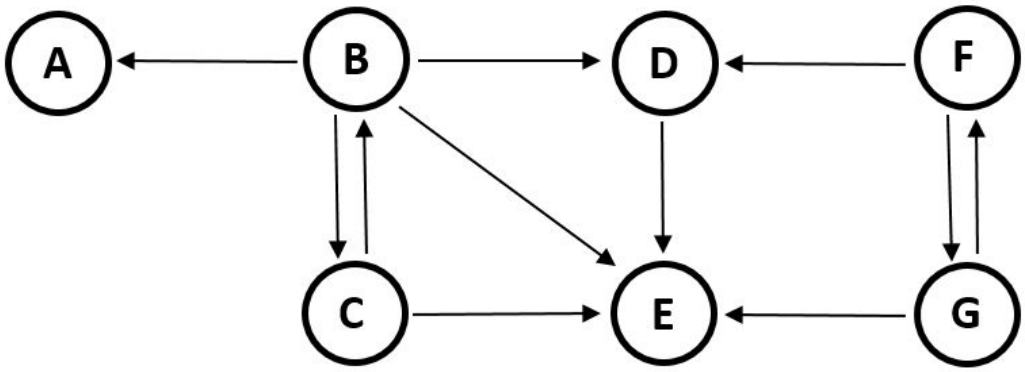

Figure 1

In the example above, there are 7 web pages represented: A, B, C, D, E, F, and G. If we are on web page B, we can choose to move to either A, C, D, or E. Meanwhile, if we are on web page D, then we can only move to webpage E. Notice that some web pages, such as web page A, may have no arrows going out of them, which means that there are no links to other URLs from that webpage. However, there must be at least one arrow that points to each web page so that every web page is accessible.

Let N be the total number of nodes in the Markov chain, or the number of web pages in the network. In our example above, there are seven nodes, so $N = 7$.

Code:

We can represent our Markov chain as a list of lists, or a 2D array, to input into our `ranks` function. We will call this 2D array `links`. Each `links[i]` is a list of integers representing all the webpages that can be traveled to from node `i`. Therefore, the input for our Markov chain above would be:

```
links = [[], [0, 2, 3, 4], [1, 4], [4], [5], [3, 6], [4, 5]]
```

In this case, `links[1]=[0, 2, 3, 4]` since we can travel to web pages A, C, D, and E from web page B, which correspond to indices 0, 2, 3, and 4. Also, `links[0]=[]` since we cannot travel to any other web pages from web page A.

GENERATE PROBABILITY MATRIX

Math:

In our ongoing example, the probability matrix P would then be:

```
[0.   0.25 0.   0.   0.   0.   0.  ]
[0.   0.   0.5  0.   0.   0.   0.  ]
[0.   0.25 0.   0.   0.   0.   0.  ]
[0.   0.25 0.   0.   0.   0.5  0.  ]
[0.   0.25 0.5  1.   0.   0.   0.5 ]
[0.   0.   0.   0.   1.   0.   0.5 ]
[0.   0.   0.   0.   0.   0.5  0.  ]
```

Each $P_{i,j}$ thus represents the probability that a web surfer on page i will move to page j. Note that all entries are positive or 0, because probability can never be negative. Also note that the sum of each column must be 1 or 0 [3].

Code:

To compute the probability matrix, we first initialized a nxn matrix filled with 0's, where `n = len(links)`. We then looped through the input Markov chain `links`, and for each node `j`, we found the number of outgoing links `numLinks = len(links[j])`. Then we looped through `links[j]` to find the specific pages it arrives at, `i`. We thus have both the page we leave `j` and the page we arrive at `i`, so as the math section explains, we set `P[i][j] = 1/numLinks`.

DANGLING NODE & DAMPING FACTOR

Math:

In our example, node A is a dangling node since there are no outgoing arrows from that node, so web page A does not have links to any other pages. Hence, the first column of our probability matrix P, which corresponds to node A, is filled with zeros, since the probability of travelling to any other page from page A is 0. When we have a dangling node, our probability matrix P is a non-stochastic matrix.

To calculate the pagerank vector, we first take a vector v representing our initial starting point on the web. Supposing that we are equally likely to start on any webpage, we can assign all the elements of v to be *(1 / N)*. In our example from above, we have 7 nodes, so we have:

```
v = [1/7, 1/7, 1/7, 1/7, 1/7, 1/7, 1/7]
```

A simple method to find the pagerank vector would be to multiply our probability matrix P by our vector v until it converges. However, note that the column of zeros in P from our dangling node results in the rank of some pages eventually being zero. This does not make sense, since all pages have incoming arrows and therefore must have some nonzero probability of being reached from an arbitrary starting point in the web. Therefore, we cannot find a pagerank vector from a non-stochastic probability matrix.

To account for a dangling node in our Markov chain which results in having a non-stochastic matrix P, we can replace any column of P that is all zeros with a column that is

filled with *(1 / N)*. This models the activity of a web surfer, who after reaching a page with no links, would jump to another random page with equal probability. Then, we would have:

$$P_{i,j} = \begin{cases} \frac{1}{val(j)}, & \text{if there is a link from j to i} \\ \frac{1}{N} & \text{if } i \text{ is a dangling node} \\ 0, & \text{otherwise} \end{cases}$$

Now, when we apply our method of repeatedly multiplying our probability matrix P by our starting point vector v, we would see that some of the columns that we previously filled with zeros are now non-zero. This means that some pages now have a nonzero probability of being reached, even after reaching the page with no outgoing links.

However, let us consider another scenario: once a user travels from page B to pages D or E or from page C to page E, they have no way of reaching web pages A, B, and C again since there are no arrows that go back the other way. They can only travel between pages D, E, F, and G after this point, which makes the graph reducible. Note that if there is a cycle in the graph, the probability does not converge, and so pagerank cannot be calculated.

To calculate page ranks correctly for a reducible graph, we can apply a damping factor B to our matrix P. The constant B represents the small probability that the random web surfer will jump to a random page from their current page instead of clicking through links. In our example, this allows the web surfer to get back to pages A, B, and C after reaching page D or E. (1-B) represents the probability that the web surfer will choose an outgoing link on the page to follow. We decided to use a damping factor of 0.15 since this what is used by Google. We also noticed that with too large of a damping factor, such as B = 0.5, the results of the page ranking algorithm were skewed since there was too much weight given to jumping around web pages rather than clicking through links, which is what usually occurs. We apply the damping factor to get our new matrix P_B , as follows:

$$P_B = (1-B)P + B(Q)$$

Since the damping factor B must be nonzero by definition, we know that none of the columns of P_B can be filled with zeros. Therefore, our new matrix P_B is column-stochastic [4].

Code:

The new modified matrix P_B is represented by the 2D array PB in our code. To compute the modified matrix PB that accounts for dangling nodes and reducible graphs, we first initialized an integer B with the value of 0.15 to account for our chosen damping factor. Then, we initialized an nxn matrix Q with every element set to *(1 / N)*. This accounted for dangling nodes. Next, we looped through matrix Q, which was represented as a 2D array, and multiplied every element in Q by the damping factor B, to turn Q into B(Q). Next, we looped through

matrix P, which was also a 2D array, and multiplied every element in P by B, to turn P into (1-B)P. Lastly, we set our new modified probability matrix PB to be the sum of P and Q.

CONVERGENCE OF PAGERANK THEOREM

Math:

The Perron-Frobenius Theorem gives us that for our column stochastic matrix PB, we can always find 1 as an eigenvalue, and all other eigenvalues have magnitude strictly less than 1. Furthermore, 1 is a root, but not a repeated root of the characteristic polynomial, so 1 is a simple eigenvalue. This eigenvalue 1 also corresponds to a positive eigenvector.

Let P_B 's eigenvectors be $\{ v_1 ... v_n \}$, where v_1 is the pagerank vector that corresponds to eigenvalue 1 in the Frobenius Theorem. Then $\forall v_i, \exists \lambda_i | (P_B)v_i = \lambda_i v_i$

Let X be the initial chosen pagerank vector s.t. $X^T = (x_1 ... x_n), x_i \in [0,1], \sum_{i=1}^{n} x_i = 1$ We express X in terms of P_B 's eigenvectors: $X = \sum_{i=1}^{n} a_i v_i$.

Then $P_B(X) = (P_B)(\sum_{i=1}^{n} a_i v_i) = \sum_{i=1}^{n} a_i (P_B) v_i = \sum_{i=1}^{n} a_i (\lambda_i) v_i$. After n iterations, the probability that the user will be on each of the pages becomes $(P_B)^n X = \sum_{i=1}^{n} a_i (\lambda_i)^n v_i$

But we know that all eigenvalues other the 1st have a magnitude strictly less than 1, so $\lim_{n \to \infty} (P_B)^n X = a_1 v_1 = v_1$. We know that $a_1 v_1 = v_1$ because for it to be a probability distribution in the first place, the absolute value of the sum of entries in v_i has to be 1 and so the limit leads to a_1 having to be 1. Now, we note that we had set v_1 to be the pagerank vector! Therefore, P_B converges to an eigenvector which corresponds to the eigenvalue 1. In order to calculate Pagerank then, there is no need to compute (P_B)X again and again, until it converges. Rather, we can simply find the eigenvector of P_B that corresponds to the eigenvalue 1 [3].

Code:

We used `numpy.linalg.eig` to obtain the eigenvalues `w` and eigenvectors `v` of the matrix `PB`. Because `numpy.linalg.eig` approximates eigenvalues and does not order them, and because we knew by the Perron-Frobenius Theorem the largest eigenvalue would be 1, we looped through `w` to find the index of the largest eigenvalue. Then we found the eigenvector corresponding to index of largest eigenvalue 1 `pagerankVector = v[:,largestIndex]`.

COMPUTING INDEXED PAGERANK VECTOR

Math:

To compute the page rank from the pagerank vector, we first sort all the elements in the chosen eigenvector by order of magnitude. These magnitudes correspond to the probabilities of reaching each web page from an arbitrary starting point on the web. Then, we use the order of elements in the sorted array to create a new pagerank array containing the ranking of each of the eigenvector elements, from 0 to (number of nodes - 1). Therefore, the pagerank array contains the ranking of each element in the original list. The smaller the value, the more popular the web page is [5].

Code:

In this step of our function, we first initialized an array called `indexedPagerankVector`. We set each element in the new vector array to be a tuple containing the eigenvalue elements from the `pagerankVector`, and the original index of the element. These indices correspond to web pages in the Markov chain, with 0 = A, 1 = B, and so one. We also created a helper function called sortFirst to help sort the indexed pagerank vector by returning the absolute values of the eigenvector elements. Then, we sorted `indexedPagerankVector` by the absolute values of the eigenvector elements. Since we had the original indices saved as the second element in the sorted tuples, we were able to create a final `pagerank` array with elements as the second elements of tuples in `indexedPagerankVector`. These integers are the indices of the sorted eigenvector values. Lastly, we returned our `pagerank` array, containing the ranking of each element in the original list.

OBSTACLES ENCOUNTERED

This project was very fascinating and rewarding for us, but we did run into a few challenges along the way. One coding challenge we encountered in the beginning was figuring out how to correctly compute PB from P and Q. We realized that our multiplication algorithm was incorrect because we had not saved the multiplied values in the 2D arrays after exiting the for loop. We fixed this bug by traversing by index inside the array.

Another problem arose when we were finding the index corresponding to the eigenvalue of 1. Since numpy estimates eigenvalues, none of the values in the returned array were exactly equal to 1.

However, we knew by the Perron-Frobenius Theorem that:

$$1 = \lambda_1 > |\lambda_2| \geq |\lambda_3| \geq |\lambda_4| \geq \cdots \geq |\lambda_N|$$

Therefore, we decided to save the indices of each eigenvalue in a new vector array. Then, we sorted the indexed vector array by eigenvalue and retrieved the index corresponding to the largest eigenvalue, since we knew the largest eigenvalue must be 1. The eigenvector corresponding to the index of the largest eigenvalue is our pagerank vector.

CONCLUSION

While this PageRank algorithm can be used to rank pages in order of importance for search engines, it can also be used in other areas as well. For example, advertising companies can use previous user data to rank what types of ads are most relatable to the user, and then present similar ads so that the user is more likely to buy those products. Similarly, Twitter, Instagram, Facebook and other social networks could use such an algorithm to determine which users are the most popular in a given sub-network. Note however, that bots may easily take advantage of this algorithm and push certain nodes to the top if they continuously link to each other. For example, fake news could easily rise to the top with such exploitation, and spread quicker than real news. Thus, companies that implement this algorithm must add more private features to it so that people do not exploit the known PageRank algorithm.

BIBLIOGRAPHY

1. Ipsen, I. C., & Wills, R. S. (2006). Mathematical properties and analysis of Google's PageRank. Retrieved November 30, 2018, *Bol. Soc. Esp. Mat. Apl*, *34*, 191-196, from http://www4.ncsu.edu/~ipsen/ps/cedya.pdf
2. Khim, S. (2007). The Frobenius-Perron Theorem. Retrieved November 30, 2018, from http://www.math.uchicago.edu/~may/VIGRE/VIGRE2007/REUPapers/FINALAPP/Khim.pdf
3. Rousseau, C. (2015, May 30). How Google works: Markov chains and eigenvalues. Retrieved November 30, 2018, from http://blog.kleinproject.org/?p=280
4. Shum, K. (2013, April 3). Notes on PageRank Algorithm. Retrieved November 30, 2018, from http://home.ie.cuhk.edu.hk/~wkshum/papers/pagerank.pdf
5. Tanase, R., & Radu, R. (2009, Winter). Lecture #3: PageRank Algorithm - The Mathematics of Google Search. Retrieved November 30, 2018, from http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html
6. Whitmill, D. (2017, January 10). A History Lesson on PageRank. Retrieved November 30, 2018, from https://www.boostability.com/a-history-lesson-on-pagerank
7. Wikimedia Foundation. (2018, November 18). Markov chain. Retrieved November 30, 2018, from https://en.wikipedia.org/wiki/Markov_chain