Markov Chains and PageRank

Ahana Mukhopadhyay, Jessica Xiao

December 13, 2018

Contents

1 Introduction

People search the web every day, expecting to find results that are most relevant to their search criterion. Google has become the dominant search engine because of the accuracy of its ranking algorithm. Without it, users would be left digging through page after page, trying to find relevant information. The PageRank algorithm essentially automates this process. Larry Page and Sergey Brin first developed the algorithm at Stanford University in 1996. It was based off the idea that information on the web should be ordered by popularity; pages with more links to them should rank higher than those with fewer links.

Google describes the algorithm as follows:
"PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites" [6].
This paper will describe the implementation of such an algorithm in Python.

2 Definitions

Definition 2.1. PageRank is a method used by the Google search engine to calculate the importance of a page in comparison to the rest of the pages on the web. A webpage's importance is determined by the number of other pages that link to it. If many pages or one single, authoritative page links to page j, then page j is called important. [1, 2]

Definition 2.2. A **PageRank vector** is the resulting vector that ranks the relative importance of each page. The ranking for a page indexed at i would be found in the i^{th} element of the vector. [2]

Definition 2.3. A **Markov Transition Matrix** contains the probabilities of reaching each page given the current page. Each column represents the current page, while the row represents the probability of landing on the specified page. The probability is always $\frac{1}{L}$, where L is the number of links coming out of the given page. The matrix should also be column-stochastic. For example, a set of two web pages (A and B) would have a two by two matrix, potentially containing these probabilities:

$$A = \begin{bmatrix} 0 & \frac{1}{2} \\ 1 & \frac{1}{2} \end{bmatrix}$$

Thus, if a user is on page A, they have no chance of going to page A through a link. If they were to go to another page, they would be guaranteed to be linked to page B. These probabilities are expressed in column one. If they started on page B, they have $\frac{1}{2}$ chance to be linked to page A or B. These probabilities are shown in column two. This algorithm may crash if there are dangling nodes, making the matrix not column-stochastic. Thus, for the functionality of the simplified PageRank algorithm, if a page is a dangling node, then the elements for the given column of the page will have $\frac{1}{n}$ probability, where n is the number of pages in consideration. [4, 5]

Definition 2.4. A **random walk** consists of starting at a web page, randomly choosing a link on the page, and following that link. This process can continue to iterate, with the same procedure. This is also known as a **Markov Chain**. [4]

Definition 2.5. A **node** represents one web page that has its own distinct URL. The two words, node and page, will be used interchangeably. [2]

Definition 2.6. If a web page, j, contains a link to another website, i, or vice versa, then the two webpages are said to be **linked**. Thus, there is a **link** between nodes i and j. Links are unique up to the direction. If j contains a link to i, then there is an arrow between the two nodes, with the arrow facing node i. [2]

Definition 2.7. A matrix is **column-stochastic** if all of its elements are greater than or equal to zero. Additionally, the entries in each column must add to one. [3]

Definition 2.8. A vector **converges** if when multiplied by the transition matrix multiple times, its elements stop changing. For the purposes of the project, an element converges when it differs from the previous result by less than three decimal places. [5]

Definition 2.9. A **dangling node** is a website that does not link to any other websites. [2]

Definition 2.10. A scalar λ is an **eigenvalue** for A if there exists a nonzero vector v such that $\mathrm{A}\mathrm{v} = \lambda\mathrm{v}$. [3]

Definition 2.11. An **eigenvector** is a nonzero vector such that $\mathrm{A}\mathrm{v} = \lambda\mathrm{v}$ for some scalar λ. [3]

Theorem 2.12. (Perron-Frobenius Theorem)
[2] Given that A is a positive, column-stochastic matrix:

1. An eigenvalue of A is 1, with a multiplicity of one.
2. The eigenvalue, 1, is the largest eigenvalue. All other values have an absolute value of less than one.
3. The eigenvectors corresponding to the eigenvalue of 1 will either have all positive or negative entries. Also, one corresponding unique eigenvector for the eigenvalue of 1 will have a sum of 1.

3 Math

The PageRank algorithm, more explicitly explained below, works on the assumption that the sequence of iterations $v, Av, ...A^k v$ eventually converges at a certain value of v (the final PageRank vector). The Perron-Frobenius Theorem helps explain this. This theorem states that if A is a positive, column stochaistic matrix, then

1. 1 is an eigenvalue with multiplicity 1
2. All other eigenvalues have an absolute value less than 1
3. There is a unique eigenvector that corresponds to the eignevalue 1 where the sum of its entries is 1

The PageRank vector is the eigenvector corresponding to an eigenvalue of 1. We know that the sequence $v, Av, ...A^k v$ converges due to the Power Method Convergence Theorem. This states that if A is a n x n positive column stochastic matrix and v is the n x 1 vector with all of its entries being equal to $\frac{1}{n}$, then the sequence $v, Av, ...A^k v$ will converge to the eigenvector corresponding to the eigenvalue 1.

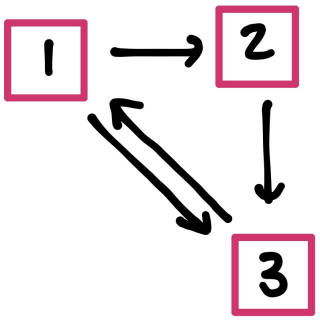
Convergence makes sense if we rely on a few simplifying assumptions. Let's assume that all eigenvalues are distinct. This implies that eigenvectors are also distinct and linearly independent. Define the matrix P to contain the eigenvectors of the transition matrix. Since the columns of P are linearly independent, the determinant of P is non-zero and P is invertible. Now, let's diagonalize the matrix A by writing it as $A = PDP^{-1}$ where D is a diagonal matrix containing the eigenvalues of A along its diagonal. We can rewrite $A^k v$ as

$$\begin{aligned} A^k v &= (PDP^{-1})^k v \\ &= (PDP^{-1})(PDP^{-1})...(PDP^{-1})v \\ &= PD^k P^{-1} v \end{aligned}$$

As k goes to infinity, the diagonal entries of D will go to 0 except for the entry that was initially 1. Therefore, the PageRank vector will converge to the eigenvector corresponding to the eigenvalue of 1.

4 Code

We'll walk through the steps of the PageRank algorithm using an example. The picture below represents the connections between different webpages. Each square is a page. An arrow pointing from one page to another represents a link from that page to the next page.



The example contains 3 pages. Define the function $N(p)$ to be the number of pages that a given page links to. For example, $N(1) = 2$ since page 1 links to pages 2 and 3. Let's now create the nxn matrix A (n = total number of pages) where the ij-entry is

$$a_{ij} = \begin{cases} \frac{1}{N(p)} & \text{if there is a link from j to i} \\ 0 & \text{otherwise} \end{cases}$$

In this case, the 3x3 matrix A would be

$$A = \begin{bmatrix} 0 & 0 & 1 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 1 & 0 \end{bmatrix}$$

A is column-stochastic (as defined above) if each of the pages links to at least one other page. Following along with the code excerpt from the rank() function below, the first few lines initialize the nxn matrix A with 0's in every position.

```
#creates matrix with all zeroes
    transitionM = []
    for i in range(len(links)):
        transitionM.append([])
        for j in range(len(links)):
            transitionM[i].append(0)
```

Next, we need to fill each position in A with the appropriate value as specified by the piecewise function defined above. A is called transitionM below.

```
#puts in values for matrix A
    for i in range(len(links)):
        probability = 1/(len(links[i]))
        for j in links[i]:
            transitionM[j][i] = probability
```

Next initialize the page-rank vector v with $\frac{1}{n}$ in each entry where n is the number of pages. The reason this makes sense is because initially, the probability of going to any given page is the same. In this case, there are 3 pages, so the initial vector v is as follows:

$$v = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}$$

The following code accomplishes this:

```
#fill vector v with 1/n where n is # of pages
v = [1\(len(transitionM)) for i in range(len(transitionM[0]))]
```

Upon each iteration, replace v with Av until the values converge. In order to multiply the matrix A by v, we defined the function matrixVectorMultiply(A, v) as can be seen below:

```
def matrixVectorMultiply(A, v):
    newV = []
    #checks that the dimensions of A and v match
    if (len(A[0]) != len(v)):
        return False
    #multiplies A with v and creates resulting vector "newV"
    for j in range(len(A[0])):
        sum = 0
        for i in range(len(v)):
            sum += v[i]*A[j][i]
        newV.append(sum)
    return newV
```

Now, to iteratively replace v with Av, we can cycle through a while loop until the values converge. First, we initialize the values of oldV and newV where oldV is initially Av and newV is intially AAv. Then, enter the while loop on the condition that the values haven't converged yet. We define a separate convergence function that checks when the difference between values in oldV and newV is almost 0 (essentially, the matrix stops changing in value).

```
#initialize values of oldV and newV
    oldV = matrixVectorMultiply(transitionM, v) #Av
    mult = np.matmul(transitionM, oldV) #AAv
    newV = np.ndarray.tolist(mult) #changes AAv to lists
    diff = True

    #continues looping until values converge
    while diff == True:
        oldV = newV #AAv
        mult = np.matmul(transitionM, newV) #AAAv
        newV = np.ndarray.tolist(mult) #changes AAAv to lists
        diff = convergence(oldV, newV)
    rankedList = createRank(newV)
    return rankedList
```

```
def convergence(oldV, newV):
    #checks if difference between all values is less than 0.001
    for i in range(len(oldV)):
        if abs(oldV[i]) - abs(newV[i]) >= 0.001:
            return True
    return False
```

Finally, we rank the importance of the pages based on the index in v that has the highest value. The indices of v with higher values correspond to pages with more incoming links. The code for doing so is below:

```
def createRank(v):
    list = copy.deepcopy(v)
    rankedList = []
    #finds max element, appends to new list, removes element from original list
    while len(v) > 0:
        m = max(v)
        i = list.index(m)
        rankedList.append(i)
        v.pop(v.index(m))
    return rankedList
```

In some cases, we might have a dangling node. In this case, the transition matrix A will have a column of all 0's. We need to change this to a column with every entry of this column being $\frac{1}{n}$ where n is the number of pages. The following code accomplishes this.

```
def danglingNode(transitionM):
    #transpose matrix to get columns
    transitionMT = np.ndarray.tolist(np.transpose(transitionM))
    #total number of pages
    n = len(transitionM)
    #loop through columns to check if it's a danging node
    for col in transitionMT:
        if isDanglingNode(col):
            #change all elements in column to 1/n
            for elem in col:
                elem = 1/n
    return np.ndarray.tolist(np.transpose(transitionMT))

def isDanglingNode(col):
    #checks if column contains all zeroes
    for elem in range(len(col)):
        if elem != 0:
            return False
    return True
```

If we don't account for dangling nodes, then the importance of pages that do not link to other pages will be lost. Instead, if we initialize the column corresponding to the dangling node with $\frac{1}{n}$

(n being the number of web pages), the importance of this dangling node gets redistributed among the other pages.

5 Conclusion

Limitations:
A significant limitation of this algorithm is that it is only applicable for irreducible graphs. An irreducible graph is when any given pair of distinct nodes in the graph of the page links could start from one node and eventually end on the other node, and vice versa. A reducible graph can be defined through the following example:

Given five pages, A, B, C, D, E, once a web surfer lands on pages C, D, or E, it cannot reach pages A or B.

Thus, this graph is reducible, because once reaching a node, a web surfer can only reach a certain set of nodes, and not all of the available nodes.
The algorithm does not apply for reducible graphs, because the algorithm may rank C, D, E, as higher than A or B, but A or B should actually be ranked higher. The inability to be applied to reducible graphs significantly limits the usability of this algorithm.

One main disadvantage of the page rank algorithm is that it favors older pages over newer pages simply because older pages have more links to other pages. However, users often want updated information, as opposed to older websites. Additionally, the PageRank algorithm does not consider current events, which are often of most importance. Lastly, web creators could exploit this algorithm, by ensuring that other pages link to their own.

Alternative:
A scenario that the transition matrix does not take account of is the possibility that a random web surfer can exit the current page and choose a random page to enter into. In that case, each page has equal probability to be chosen. Thus, a damping factor, p, can be chosen to remedy this case. This factor will be a number between 0 and 1, representing the probability that the previously specified scenario occurs. Then, an n x n matrix B can be created, where n equals the number of pages. All elements in the matrix equals $\frac{1}{n}$, since each page has the same probability of being landed on. Then, using the original transition matrix, A, the overall transition matrix could found through this equation:

$$M = (1 - p) * A + p * B$$

Thus, this combines the probability of the random web surfer following the regular method, following from page to page, and the probability of the surfer exiting and entering a random page.

References

[1] Ian Rogers, *Pagerank Explained Correctly with Examples* (2018), https://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm. Accessed 13 December 2018.

[2] Raluca Tanase, *PageRank Algorithm - The Mathematics of Google SearchPagerank Explained Correctly with Examples* (2018), http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html. Accessed 13 December 2018.

[3] Raluca R. Tanase, *Linear Algebra - in a Nutshell.* (2018), http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture1/lecture1.html. Accessed 13 December 2018.

[4] Christiane Rousseau, *How Google works: Markov chains and eigenvalues.* (2015), http://blog.kleinproject.org/?p=280. Accessed 13 December 2018.

[5] K Shum, *Notes on PageRank Algorithm. Lecture 13 notes.* (n.d.), http://home.ie.cuhk.edu.hk/wkshum/papers/pagerank.pdf. Accessed 13 December 2018.

[6] *PageRank* (2018), https://en.wikipedia.org/wiki/PageRank. Accessed 13 December 2018.