The PageRank Algorithm

A discussion and efficient implementation of the mathematics behind Google's PageRank algorithm

Arnav Gupta (arnavgu2) and Adarsh Nalamalpu (analamal)

21-241 Matrices and Linear Transformations
Carnegie Mellon University
Fall 2019
December 6, 2019

1 Introduction

The purpose of this paper is to discuss and implement the PageRank algorithm of ranking items of an interconnected graph such as that seen in the Internet's hyperlink structure. This technique is the underlying technology behind Google's search engine and is one of the main factors responsible for its rapid growth over the last two decades. PageRank was developed in 1996 by doctoral candidates Larry Page and Sergey Brin at Stanford University. Prior to PageRank and Google, early Internet users were struggling to efficiently find the specific web pages they were looking for, given that users knew the subject matter they were looking for but had never before been to some target web page. With the size of Internet and scale of hyperlinks between web pages increasing, search engines were stumbling to crawl and index all available web pages and serve a useful list of relevant web pages to users in a reasonable amount of time related to the user's initial query. PageRank provided a method to rank queried pages in such a way that the first handful of web pages would likely be what the user is looking for because they were more important websites, and additionally reduce the amount of real-time computation needed by storing "query-independent" information (information that is is used regardless of what the user searched for) in the backend. The result of these advantages produced an efficient and pertinent ranking of web pages that laid the foundation for one largest and most influential tech companies in the world. Google relies on a complex pipeline to take as input a query and output a sorted list of relevant pages, so this paper focuses specifically on one aspect of this pipeline. Namely, the "query-independent" ranking of web pages based on apparent popularity.

2 Mathematical Background

PageRank relies of the Internet's hyperlink structure to rank pages by the metric of apparent popularity based of the following thesis: "a web page is important if it is pointed to by other important pages." An additional subthesis is that a web page that points to many pages is less popular than a web page that points to few pages, given all other factors are equal. PageRank works on the idea of a random surfer who is traversing the internet by randomly visiting one of the links on the page is it currently on or some random page on the web. By constructing a Markov matrix where each row i sums to 1 and each entry i, j indicates the probability that the random surfer will visit page j from page i, PageRank is able to create a so-called "popularity" ranking of web pages based on the overall proportion of time the random surfer spends on each website when following the probabilities given in the Markov matrix. The genius of PageRank lies in the subtle adjustments and decisions used in constructing this Markov matrix to ensure correctness and efficient computation that allows Google to return a consistent and fast page of search results of the seemingly infinite set of possible queries.

3 Constructing an Adjacency Matrix

The first step in producing this Markov matrix is to construct an adjacency matrix on the Internet hyperlink graph. This adjacency matrix A will be a square matrix in which every entry $A_{ij} = 1$ iff there exists an outlink from web page i to web page j, else $A_{ij} = 0$. We wanted to see some real action, so we used real data. Specifically, we used a dataset of web pages and associated hyperlinks for a subgraph of the internet consisting of pages matching the query **California**.

This data came from Jon Kleinberg's 2002 "Structure of Information Networks" course at Cornell university. http://www.cs.cornell.edu/courses/cs685/2002fa/

The data was originally in a text file. We converted the text file into a csv file and split the data into two sets. The first set consisted of around 10000 strings of website urls indexed by the natural numbers. The second set consisted of all the hyperlinks in the graph where each row includes indices for the origin and the destination of a single hyperlink. Figure 1 shows a sample of the first set of indexed websites. Figure 2 shows a sample of the second set of hyperlinks. For example, the first hyperlink in the set represents a link on website 0 ("www.berkely.edu") to website 458 ("lib.berkeley.edu").

Figure 1: The first couple lines of the first partition of the original dataset: links.csv

```
0,http://www.berkeley.edu/
1,http://www.caltech.edu/
2,http://www.realestatenet.com/
3,http://www.ucsb.edu/
4,http://www.washingtonpost.com/wp-srv/national/longterm/50states/ca.htm
5,http://www-ucpress.berkeley.edu/
.
.
.
```

Figure 2: The first couple lines of the second partition of the original dataset: outlinks.csv

```
0,458
0,459
0,460
0,461
0,462
0,463
0,464
1,482
2,483
```

We iterated through these two datasets two construct an adjacency graph with the aforementioned definition.

We first import the data.

```
using Pkg;
using CSV;
using DataFrames;
using LinearAlgebra;

links = CSV.read("//Users//arnavgupta//Documents//Fall19//241//cali-links.csv")
outlinks = CSV.read("//Users//arnavgupta//Documents//Fall19//241//cali-outlinks.csv")
```

Next, we construct an adjacency graph. $A_{ij} = 1$ if there is a hyperlink from website i to website j.

```
# n, number of websites in graph
n = size(links)[1]

# total_num_outlinks, number of directed edges in graph
total_num_outlinks = size(outlinks)[1]


# We create an adjacency matrix A for the directed graph
A = zeros(n,n);

for i = 1:total_num_outlinks
#     Note: hyperlinks refer to 0-indexed websites, but julia uses 1-index,
#           so we must increment the indices here
    A[outlinks.from[i] + 1, outlinks.to[i] + 1] = 1

end
```

4 Constructing a Substochastic Matrix H

Our goal is to create a Markov matrix to represent the probabilities that a random surfer may go to some website from another. One necessary condition of a Markov matrix is that all the rows sum to 1. A submarkov or substochastic matrix has rows that sum to either 0 or 1, so we begin by creating a substochastic matrix first.

As of now, each row i of A sums to the number of outlinks on p. If P_i is the number of outlinks on page i, then...

$$\sum_{j=1}^{n} A_{kj} = P_i$$

We normalize each nonzero row by dividing each element of row i by P_i so that that magnitude of each nonzero row is 1.

```
H = zeros(n,n)
for i = 1:n
    row_outlinks = sum(A[i, :])
    for k = 1:n
        if A[i, k] != 0
            H[i, k] = A[i,k]/row_outlinks
        end
    end
end
```

5 Stochasticity Adjustment

This matrix can exactly represent the transition states of a random walker traversing the graph only if the graph contains no dangling nodes. A dangling node is a webpage that has no outlinks. Such webpages are commonly found on the internet in the context of data tables or pdf documents that represent terminal information to be reached by the surfer. Every zero row in H exists because of a dongling node. We convert the substochastic matrix to a stochastic matrix that changing these zero rows of H with a row that sums to 1.

The reason we must correct for these dangling nodes is that we want to find a steady state for our graph: a vector representing a single sequence of proportions of time the random surfer spends on each page. Due to its being substochastic, the web surfer may enter a dangling node and then get stuck, because each entry in the dangling node's row is 0, i.e. there is a 0 probability for the random surfer to go anywhere. These dangling nodes will causes a sink on the proportion of time the surfer spends on the dangling nodes because once the surfer comes to a dangling node, he cannot visit any other website because he has no way to get there. Accordingly, the PageRank scores for the non-dangling nodes will transfer all their PageRank score to the dangling nodes.

To make H stochastic, we quite simply replace each zero row with a row that represents that the surfer may go to any web page on the graph with equal probability following a uniform distribution.

```
S = H
for i = 1:n
```

```
        if sum(H[i, :]) == 0
            for j = 1:n
                S[i,j] = (1 / n)
            end
        end
    end
```

We now have matrix S which is a stochastic matrix, i.e. each row sums to 1. What this means intuitively is that once the surfer encounters a dangling node, he will randomly teleport to some other page on the graph at random.

6 Primitivity Adjustment

We are not yet guaranteed to find a convergent steady state when there are cycles present in the graph. Consider a graph with two nodes that point to each other. In this graph, the random surfer will keep going back and forth between these two nodes. When PageRanks are iteratively determined, the matrix will keep transferring PageRank between the pages, causing there to be no convergent steady state. In the previous section where the matrix has not been adjusted to have stochasticity, the chain would have converged, but would have resulted in all the PageRank concentrated on the dangling nodes. Having ensured that there will be no PageRank sinks, we must now ensure that the chain will converge.

We adjust this matrix S to create a new matrix G that has been adjusted for primitivity. As of now, the random surfer only has the option to follow the link structure he encounters unless he hits a dangling node, in which case he randomly goes to any page. This is not an accurate model of how people use the Internet, however. Most users will begin their traversal at some node following some path, but conclude their immediate path and go to another website even if they haven't reached a terminal website. We must allow the random surfer the ability to "teleport" to any other node regardless of where he is at the moment.

We use a dampening factor $\alpha \in (0,1)$ to do this primitivity adjustment. We choose $\alpha = .85$. This alpha signifies that there is a 85% chance that the random surfer will continue on his current traversal and a 15% chance that the random surfer will abandon his current path and teleport to a new node at random. We use a weighted average using the dampening fact in an effort to maintain the advantageous features that we have insofar obtained through our careful design choices of H and then S, namely stochasticity.

We use the vector $\vec{e}$ to be a vector of all 1's. Thus ee^T is an $n \times n$ matrix where each entry is 1.

$$G = (\alpha)S + (1-\alpha)(\frac{1}{n})\vec{e} \cdot \vec{e}^T$$

Because G is taken a convex combination of two stochastic matrices S and $(\frac{1}{n})\vec{e} \cdot \vec{e}^T$, G will also be stochastic.

```
 # alpha = the dampening factor that correlates to how likely a web surfer
# is to keep following hyperlinks he finds on the current site versus
# teleporting to another random site
alpha = 0.85;

e = ones(n)
G = (alpha) * S + (1 - alpha) (1 / n) (e * e')
```

We have now adjusted for primitivity and have solved the issue of getting stuck on cycles. G is a stochastic and primitive matrix. A primitive matrix is also a irreducible and aperiodic. Thus, G is a stochastic, and irreducible, and aperiodic matrix. By the Markov theory of random walks, these three qualities imply that there exists a unique attracting steady state $\vec{\pi}$ that the Markov chain will converge to. We are now ready to find our page rank vector.

7 Power Method

It is true that the dominant eigenvector of the matrix G will be the target PageRank vector $\vec{\pi}$. In practice, finding eigenvalues is done through iterative methods. The Power Method is an iterative method for finding the dominant eigenvector of a matrix (the eigenvector whose eigenvalue has the greatest magnitude). The Power Method defines the next closest eigenvector as a matrix-vector multiplication of the the previously obtained eigenvector.

$$\vec{\pi}^{k+1} = \frac{G \cdot \vec{\pi}^k}{||G \cdot \vec{\pi}^k||}$$

We have already shown that G is a positive Markov matrix, thus the dominant eigenvalue is unit, so we simplify the equation.

$$\vec{\pi}^{k+1} = \frac{G \cdot \vec{\pi}^k}{1}$$

$$\vec{\pi}^{k+1} = G \cdot \vec{\pi}^k$$

Therefore, it is clear that we can obtain a very close approximation to $\vec{\pi}$ by repeated multiplications by G. We can choose any vector for the initial vector $\vec{\pi}^0$ because the Markov chain will converge to the same steady state regardless of what the initial vector is. We perform this with the following code.

```
# we want our output pagerank vector to be within epsilon of the true pagerank vector
    epsilon = 10 ^ (-8)
# last_delta is initially set to an arbitrary large value
    last_delta = 10 ^ 8
```

```
# the initial pagerank vector pi0 we use is the uniform vector,
# but we could use any nonzero vector here as long as it
# has a component in the direction of the dominant eigenvector
    pi0= (1/n) * ones(n)
    pi = pi0


# we perform another iteration if we are not yet within epsilon of the true pi vector
    while (last_delta >= epsilon)
        old_pi = pi
        global pi = pi * G
        last_delta = abs(norm(pi - old_pi))
    end
```

Finally, we have obtained a vector of the PageRanks for the websites in the graph. Figure 3 below shows the first couple values of this vector.

Figure 3: the first couple entries of our output PageRank vector. The sum of all entries in this vector is 1.

```
    0.0041974078249338445
    0.0011434030804152878
    9.971562820765948e-5
    0.0014325364390488002
    0.00010499445365887654
    .
    .
    .
```

We can determine the most important web page in the whole graph by finding the web page that has the greatest PageRank value. We do so with the following code.

```
    enumeration = [Float64(1) Float64(0)]

    for i=1:n
        if (pi[i] > enumeration[2])
            enumeration[1] = i
            enumeration[2] = pi[i]
        end
    end

    println(links[Int64(enumeration[1]), :])
```

Figure 4: console output when finding the most important web page. ucdavis.edu is the most important webpage in the graph of websites matching the query “California”, because it has the greatest PageRank score.

```
DataFrameRow
| Row  | index | url                    |
|      | Int64 | String                 |
|---------------------------------------|
| 1489 | 1488  | http://www.ucdavis.edu/ |
```

8 Using Sparse Matrix and Rank-one Adjustments to Decrease Computation

When Google performs their calculations of the PageRank vector, they must analyse the entire graph of the Internet which is on the order of billions of web pages. The power method requires multiple calculations of the vector-matrix multiplication between $vec\pi$ and G (in practice around 50-100). Each vector-matrix multiplication requires $O(n^2)$ calculations. Furthermore, the Internet is constantly changing its own hyperlink structure. It is evident that the computation required to find a vector that must be performed daily if not multiple times a day.

We extend our implementation to decrease the cost of computation. If you imagine the average web page, it might have a 100 links. Links are not often just listed on a website, usually they have a context and purpose for being there else the site will be too cluttered. Let’s say that the average web page has 1000 links as an good upper bound approximation. 1000 links is much less than the total size of the graph, so using an sparse matrix data structure representation instead of a dense matrix data structure will allow for better computation complexity. How much better will our complexity be if we use a sparse matrix representation? Julia 1.3 documenation for the library SparseArrays states that multiplcation of a vector by a sparse matrix will require $O(z)$ where z is the number of nonzero elements in the matrix. This will be substantially lower that n^2 is the matrix has many zeros.

The adjacency matrix A is only sparse matrix we have so far. The substochastic matrix H can also be represented sparse since it has nonzero elements in exactly the same locations as A.

```
using Pkg;
using CSV;
using DataFrames;
using LinearAlgebra;
using SparseArrays; #Sparse array library
```

```
    links = CSV.read("//Users//arnavgupta//Documents//Fall19//241//cali-links.csv")
    outlinks = CSV.read("//Users//arnavgupta//Documents//Fall19//241//cali-outlinks.csv"

    n = size(links)[1]

    total_num_outlinks = size(outlinks)[1]

    A = zeros(n,n);

    for i = 1:total_num_outlinks
        A[outlinks.from[i] + 1, outlinks.to[i] + 1] = 1
    end

    for i = 1:n
        row_outlinks = sum(A[i, :])
        for k = 1:n
            if A[i, k] != 0
                A[i, k] = A[i,k]/row_outlinks
            end
        end
    end

#NEW: uses sparse matrices increase computation efficiency
    H = sparse(A)
```

The next step in our pipeline was to make H stochastic. This was done by replacing all zero rows with uniform distribution rows over the total number of nodes n, i.e. replacing $\vec{0}^T$ with $\frac{1}{n}\vec{e}^T$ where $\vec{e}$ is a vector of all 1's. If we do this, we eliminate a large portion of zeros that are currently in the graph, increasing the number calculations we need to do. Rather than make the matrix dense, we try to represent the matrix as a sparse matrix with some rank 1 updates in addition to it. To do this, we create a new vector $\vec{a}$ called the dangling node vector such that $\vec{a}_i = 1$ iff web page i is a dangling node, else $\vec{a}_i = 0$.

```
#a, the dangling node vector
    a = zeros(n);

    for i = 1:n
        if sum(A[i, :]) == 0
            a[i] = 1;
        end
    end
```

Let $\vec{a}$ be the dangling node vector such that $\forall i \in [n], \vec{a}_i = 1$ iff web page i is a dangling node, else $\vec{a}_i = 0$.

$$G = G$$

$$G = \alpha S + (1-\alpha)\frac{1}{n}\vec{e} \cdot \vec{e}^T$$

$$G = \alpha(H + \frac{1}{n}\vec{a} \cdot \vec{e}^T) + (1-\alpha)\frac{1}{n}\vec{e} \cdot \vec{e}^T$$

We now include the PageRank vector to find the expression evaluated at each iteration of the power method.

$$\vec{\pi}_{k+1} = G \cdot \vec{\pi}_k$$

$$\vec{\pi}_{k+1} = \left(\alpha(H + \frac{1}{n}\vec{a} \cdot \vec{e}^T) + (1-\alpha)\frac{1}{n}\vec{e} \cdot \vec{e}^T\right) \cdot \vec{\pi}^k$$

$$\vec{\pi}_{k+1}^T = \alpha \vec{\pi}_k^T H + \left(\frac{1}{n}\right)(\alpha \vec{\pi}_k^T \cdot \vec{a} + 1 - \alpha)\vec{e}^T$$

Having represented G solely in terms of a sparse matrix and 2 rank one updates, we save enormously on computations where the number of multiplications needed to be performed with each iteration of the power method is only $O(\text{nnz}(H) + n)$ where $\text{nnz}(H)$ is the number of nonzero elements in H. We know this is complexity because multiplication the G will require $O(\text{nnz}(H) + n)$ multiplications to find $\alpha \vec{\pi}_k^T H$, O(n) multiplications to find $\vec{\pi}_k^T \cdot \vec{a}$, and $O(n)$ multiplications to find $(\alpha \vec{\pi}_k^T \cdot \vec{a} + 1 - \alpha)\vec{e}^T$. The total complexity will be $O(n)$ if the graph is close to being a complete graph, otherwise it will be less since $O(\text{nnz}(H) + n) \subseteq O(n)$. Thus, we are able to reap the same benefits as the dense matrix G, guaranteeing that we find a convergent PageRank vector, yet we only need to do $O(\text{nnz}(H) + n)$ multiplcations for each iteration of the power method. It should be noted that because web pages have much less links on them than total number of web pages, we can reduce this further. Assuming there are somewhere around 100-1000 links per page, an overestimate, the complexity per power method iteration step is $O(1000n + n) = O(n)$.

We now reveal the final code for the project that is both correct and efficient.

```
using Pkg;
using CSV;
using DataFrames;
using LinearAlgebra;
using SparseArrays;

links = CSV.read("//Users//arnavgupta//Documents//Fall19//241//cali-links.csv")
outlinks = CSV.read("//Users//arnavgupta//Documents//Fall19//241//cali-outlinks.csv")
```

```
# n, number of websites in graph
n = size(links)[1]

# total_num_outlinks, number of directed edges in graph
total_num_outlinks = size(outlinks)[1]


# We create an adjacency matrix A for the directed graph
A = zeros(n,n);

for i = 1:total_num_outlinks
#     Note: hyperlinks refer to 0-indexed websites, julia uses 1-index, so we must incre
    A[outlinks.from[i] + 1, outlinks.to[i] + 1] = 1

end

for i = 1:n
    row_outlinks = sum(A[i, :])
    for k = 1:n
        if A[i, k] != 0
            A[i, k] = A[i,k]/row_outlinks
        end
    end
end

H = sparse(A)


for i = 1:n
    row_outlinks = sum(A[i, :])
    for k = 1:n
        if A[i, k] != 0
            H[i, k] = A[i,k]/row_outlinks
        end
    end
end

a = zeros(n);
for i = 1:n
    if sum(A[i, :]) == 0
```

```
        a[i] = 1;
    end
end

# we want our output pagerank vector to be within epsilon of the true pagerank vector
epsilon = 10 ^ (-8)
# last_delta is initially set to an arbitrary large value
last_delta = 10 ^ 8

# the initial pagerank vector pi0 we use is the uniform vector,
# but we could use any nonzero vector here as long as it
# has a component in the direction of the dominant eignvector
pi0= (1/n) * ones(n)
pi = pi0


# we perform another iteration if we are not yet within epsilon of the true pi vector
while (last_delta >= epsilon)
    old_pi = pi
    global pi = (alpha * pi' * H + (alpha * pi' * a + 1 - alpha) * e' * (1 / n))'
    last_delta = abs(norm(pi - old_pi))
end
```