Google Pagerank

Xing Yang Lan
MAT167
University of California, Davis

Abstract

Google is one of the largest tech companies today, perhaps best known for their search engine, google.com. Receiving roughly 5.8 Billion search queries every day, Google's search engine is tasked with finding the most relevant results amongst the 2.9 billion (as of Jan 24, 2022) indexed webpages. After finding some pages relevant to the search query, a relevant page's pagerank is considered amongst the many algorithms that assemble the Google Search Algorithm's decision for the final pages that appear on the browser. This project will focus mainly on the page ranking aspect, done with Google's Pagerank algorithm, originally developed by Sergey Brin and Larry Page in 1998. Various interpretations of the Pagerank problem, approaches to solving the problem, and intuitive juxtapositions will be covered.

1 Introduction

A page is a document, and a webpage is a document on the World Wide Web, often written in HTML, with an associated path called a link. Because links can appear on another webpage as a hyperlink, an individual browsing the internet can go from webpage to webpage, surfing the World Wide Web.

To take a set of pages, and provide an ordered set of pages to display based off a search query, transformations must be applied on the information content in the pages and query. The first consideration is the content of the search query. Since webpages often provide information through strings of letters, or words divided by spaces, that is the focus of our first transformation.

2 Term Frequency

A generalized term frequency matrix transforms a set of pages into data based on their word content. All the unique words that appear across the pages can form a dictionary $D \in R^n$. For m pages each index by some $i, i = 1, .., m$ and n unique words each indexed by some $j, j = 1, ..., n$, and with rows of the matrix corresponding to a page, displaying the number of times each word in the dictionary appears along the column, the pages p and term frequency matrix T can be defined as,

$$p \in \mathbb{R}^m, T \in R^{mxn}$$

$$T_{ij} = count(p_i, w_j)$$

where, $count(p_i, w_j)$ is the number of occurrences of w_j in p_i.

A search query can be represented by a vector in $x \in R^n$ where the entries are the counts of the words in the query. Multiplying matrix T with x will result in a vector representing weighted calculations of words in the search query and the pages.

The linear algebra calculation is enough in a simple system with a limited number of documents to provide some decent document recommendations based on the relevancy scores of each document. But there are far more improvements.

3 Pagerank

3.1 Why is Pagerank Considered?

Not all pages are created equal. Pages that contain the most words in a query are not necessarily what the searcher wants to see. Sometimes page owners might even try to (and still do) game the Term Frequency part of page-query assessment by stuffing a list of all the awesome popular words on some obscure part of their page. Luckily Google's search algorithm next considers the Google's Pageranks of a narrowed down set of pages.

Generally, a rank refers to an integer, mapped to an item based off some score of the item. A general page rank is a floating point number for a page in a network of pages connected by hyperlinks and hopefully content as well. Using traffic as an indicator, the pages can have a popularity contest and receive a normalized score so that the sum of the page ranks of all the pages in the network is 1 (Just kidding, pages can't have popularity contests. Also there will be no more jokes when discussing the very serious Google Pagerank).

Ranking a network of nodes is common problem that appears in forms such as NCAA football teams and primate hierarchies [3]. The ranks of interest in this project function as an elo rating system; for webpages this is based upon perceived traffic or popularity. It is more likely that more popular pages should be what Google searches are looking for. Instead of actually monitoring traffic for every single web page, Google uses hyperlink connectivity to model supposed traffic.

3.2 Generalized Definition of Pagerank

The connectivity between the pages through hyperlinks can be thought of and represented as a directed graph of nodes where the graph is the network and the nodes are the pages, and the edges are the links. A person randomly clicking hyperlinks on whatever page they are on, surfing from page to page, is used to model traffic. Under this model, since the traffic for a given page P, depends on the traffic from pages that link to P, the in-links, and the same applies for the in-link pages to P, the problem is recursive in nature.

The formula for the page rank of p_j can be formulated as,

$$rank(p_j) = \sum_{p_i \in in(p_j)} \frac{rank(p_i)}{|out(p_i)|}$$

where,

$in(p_j)$ is the set of in-link pages to p_j

$|out(p_i)|$ is the number of outlinks from p_i

3.3 Recursion

Apart from being one of my favorite concepts, recursion is part of the page rank definition above. The rank of a page depends on the rank of more pages which depend on the rank of even more pages. A common assumption is that the page ranks for a system converge to their true value as some scale or count of the calculations approach infinity. Infinity is not always a concept practical for computers, and neither is recursion a surprisingly large amount of times. But using base cases allow for a very literal modeling of certain recursive or even transcendental problems.

```
[1]: def phi(n):
         if n == 1:
             return 1
         return 1/phi(n-1) + 1

     print(phi(50))
```

```
1.618033988749895
```

```
[2]: def e(n, v=1, c=1):
         if n == 1:
             return 1
         return 1/v + e(n-1, v * (c+1), c+1)

     print(e(50))
```

```
2.7182818284590455
```

Figure 1: ϕ and e in Python

The figure shows two recursive functions that calculate the Golden Ratio ϕ and Euler's number e; two things that can feel very disconnect from computers at times. Getting a computer to calculate page rank with recursive function calls is something I was not able to implement due to many overflow errors.

Recursion tends to be slow and gimicky, but practical limitations aside, one great thing about it is how simple it is. It's a fairly intuitive explanation. The functions represent the essence of a transformation, in a manner similar to the eigenvectors and values for a matrix.

3.4 Fast and Intuitive Solution

I was not able to implement recursion, but since recursion is dynamic iteration, this iterative solution is what I am going to cover first. Given a Page object and a Network object, this following code will converge to the page rank in around 50 iterations, which is the parallel of 50 recursive iterations. In practice, the dynamic updates of page ranks makes it parallel a number higher than 50.

```
def surf(self, max_iter):
    for itr in range(max_iter):
        for page in self.pages:
            page.rank = sum([p.rank / len(p.out_links) for p in page.in_links])
```

Figure 2: Iterative Python Script (PyIter)

The Pagerank data that is used in this project come from a raw dataset of 500 webscraped webpages that were scraped and ranked by my STA 141C professor Bo Ning [1]. Links within this simulated network mostly belong to the University of California. As he provided the top 20 pages by Pageranks he got on R, I was able to confirm that my iterative implementation (PyIter) could approximate the Google Pagerank values within 4 significant digits in around 50 iterations.

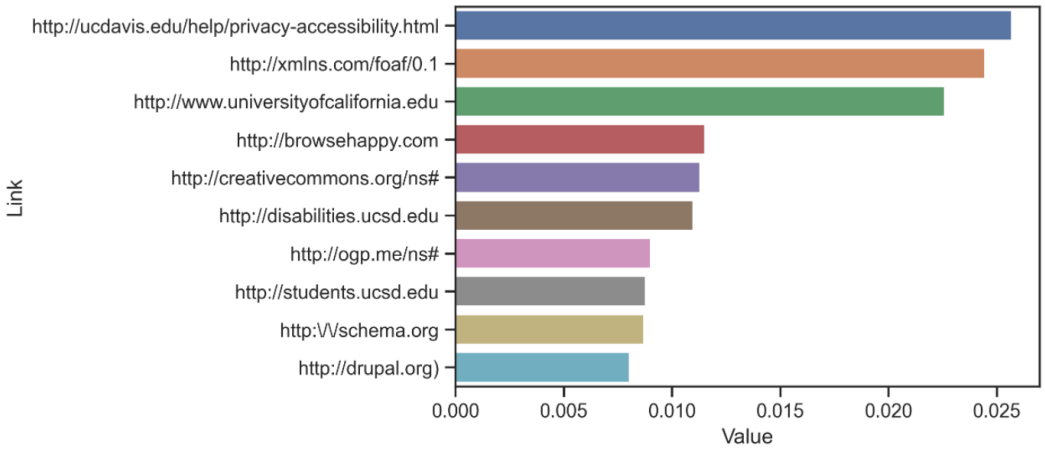

Figure 3: Top 10 Webpages via PyIter

4 The Adjacency Matrix and P Matrix

4.1 Solving Pagerank Given an Adjacency Matrix

A non-symmetric adjacency matrix can be used to represent the connectivity in a directed graph-vertices system or a network of pages. For n pages in the network, a non-symmetric adjacency matrix $A \in \mathbb{R}^{nxn}$ has entries $a_{ij} = 1$ if page i contains a hyperlink to page j and 0 otherwise. The outdegree of the jth page is then the sum of the jth row in A and the indegree of the jth page is the sum of the jth column in A.

Amongst many useful properties of this adjacency matrix A, it can be further converted into the Google matrix P. The first part of this transformation to normalize the rows of A. Going down rows n rows of matrix P, if the page has no hyperlinks, then row i of P, P_i, is a row of zeros, just like row i of A, A_i. Otherwise, the row P_i is the row A_i divided by the outdegree of page i, or the sum of A_i, $|A_i|$.

$$P = [v_1, v_2, ..., v_n]^T, v_i = \begin{cases} \frac{A_i}{|A_i|}, & \text{if } |A_i| > 0 \\ 0_n, & \text{otherwise} \end{cases} \tag{1}$$

where, A_i is the i^{th} row of A,

$|A_i|$ is the sum of the i^{th} row of A,

When the transpose of the above P matrix is multiplied against a vector of Pageranks $\pi \in \mathbb{R}^n$, this matrix P performs what an iteration of the python code does, except none of the Pagerank values update until the next iteration of π is calculated.

For $\pi \in \mathbb{R}^n$, and π_k denoting the k^{th} iteration of Pagerank calculations, with $(\pi_k)_i = \frac{Q_i}{|Q_i|}$ where Q is a vector of pages,

$$\pi_{k+1} = P^T \pi_k \tag{2}$$

The transformation applied to the vector of Pageranks π is pretty much exactly what PyIter does in one iteration, but in one big step and a different data representation. Figuratively, they're the same transformation and made to model the same thing. The python function represents what the linear algebra represents. Literally, the choice of either method only represents what the language does in the background. But the linear algebra method has some cool properties.

4.2 Adding another Factor: Teleportation

When considering actual webpages however, there are going to be webpages that have a hyperlink to them, but do not have any hyperlinks themselves. Under the interpretation so far, the random walk interpretation, our supposed web surfer would theoretically get stuck on a webpage that does not have any hyperlinks to leave. Such pages are called sinks. Similarly, there are webpages that do not have hyperlinks pointing to them, but have hyperlinks to other pages. So the P matrix further accounts for these problems.

Webpages with no inlinks and no outlinks to any webpages to the network, and disconnected networks I would argue are beyond the scope of this paper, or at least individual networks under this interpretation thus far. With that being said, lets redefine the transformation $A \in \mathbb{R}^{nxn} \longrightarrow P \in \mathbb{R}^{nxn}$,

$$P = [v_1, v_2, ..., v_n]^T, v_i = \begin{cases} \frac{tA_i}{|A_i|} + \frac{1-t}{n} & \text{if } |A_i| > 0 \\ (\frac{1}{n})_n & \text{otherwise} \end{cases} \tag{3}$$

where t is the teleportation parameter, commonly set to 0.85.

To roughly describe the difference between this P matrix and the old one; when a page has no out-links then the surfer will randomly hop onto a new page on the network. When the surfer is on a page, there is a t chance they will click on a hyperlink from their current page, and a $1-t$ chance that they will randomly pick a page from all the pages in the network with the equal chance $\frac{1-t}{n}$. Pages that had no out-links now guarantee the surfer will pick a random page. Pages with no in-links now have a chance to be teleported into.

Once again, the formula for successive iterations is still,

$$\pi_{k+1} = P^T \pi_k \tag{4}$$

or alternatively, as Naoki Saito [1] wrote,

$$\pi_{k+1}^T = \pi_k^T P \tag{5}$$

5 Solving with the Power Method

The Pageranks can be solved by simply multiplying P^T with π, successively to create higher iterations of π, just like the iterative Python script. With 50 iterations, the Pagerank values for the UC network are also approximated within 4 significant digits.

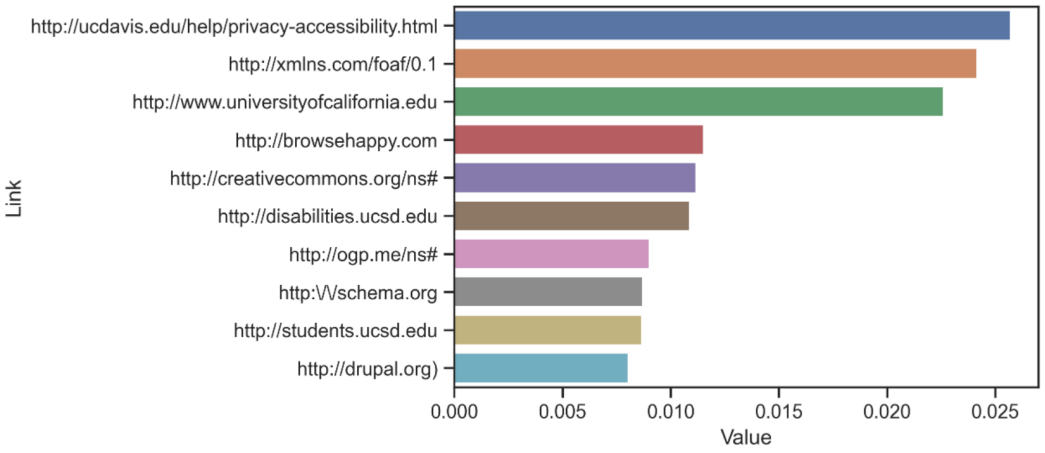

Figure 4: Top 10 Webpages via Power Method

6 Dense Solution

Going from the assumption that the Pageranks converge towards their theoretical values as k increases, which is based on an interpretation of walking randomly around on webpages, the model can also be interpreted as a Markov Chain. I would argue that a Markov Chain is more intuitive to the original definition, but with the assumption that the Markov Chain will converge towards a steady state, the formula is perhaps move intuitive.

$$\lim_{k\to\infty} \pi_k = P^T \pi_k$$

$$\lim_{k\to\infty} \pi_k - P^T \pi_k = 0_n$$

$$\lim_{k\to\infty} (I - P^T)\pi_k = 0_n$$

or in true linear regression form, $Ax = b$,

$$x = (I - P^T)^{-1} b$$

To get a computer to solve this so that the coefficients of x, or the Pageranks π are not zero, a dummy variable with a rank of 1 needs to be at the top.

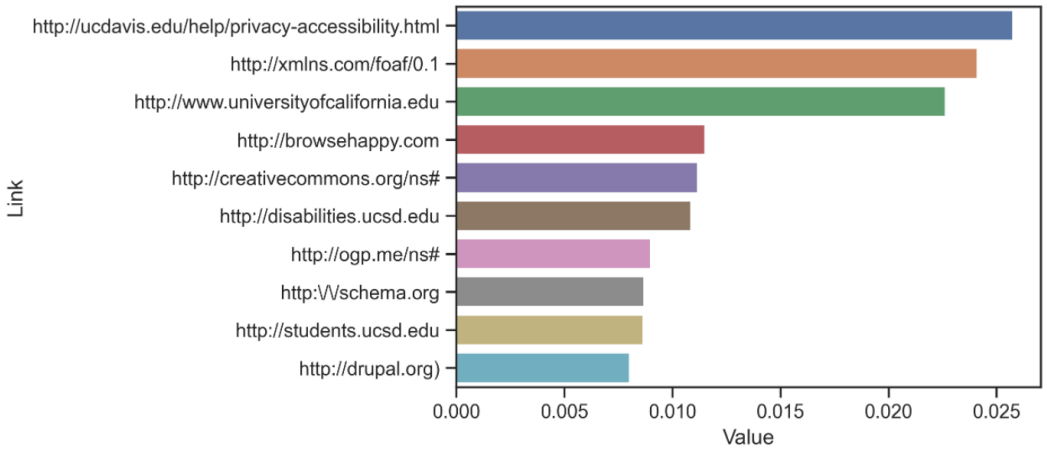

Figure 5: Top 10 Webpages via Linear Regression Module from sklearn

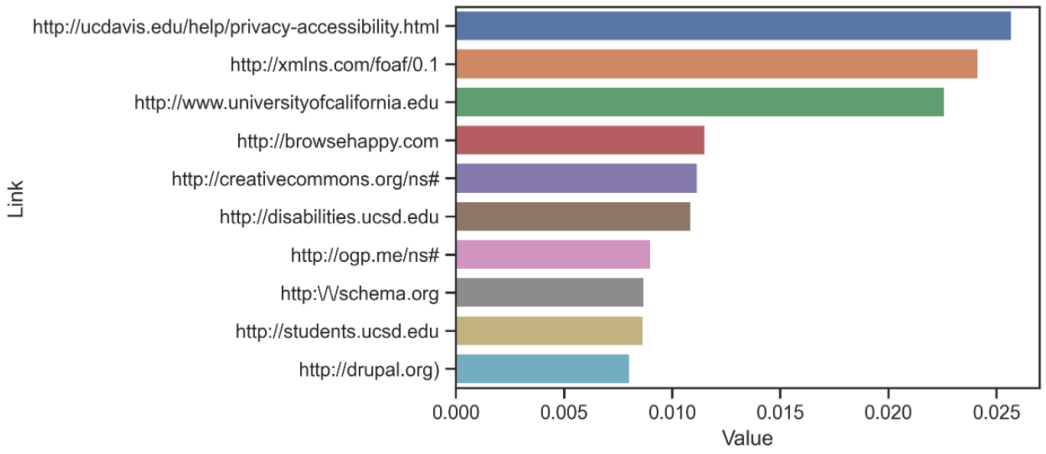

Figure 6: Top 10 Webpages via SVD

7 Comparison

7.1 Speed

The methods were timed using the timeit module's default_timer(), which helps eliminate runtime variability. The power method at 50 iterations was the fastest, followed by PyIter at 50 iterations, Linear Regression function from sklearn, and solving for the inverse by first calling numpy's SVD function.

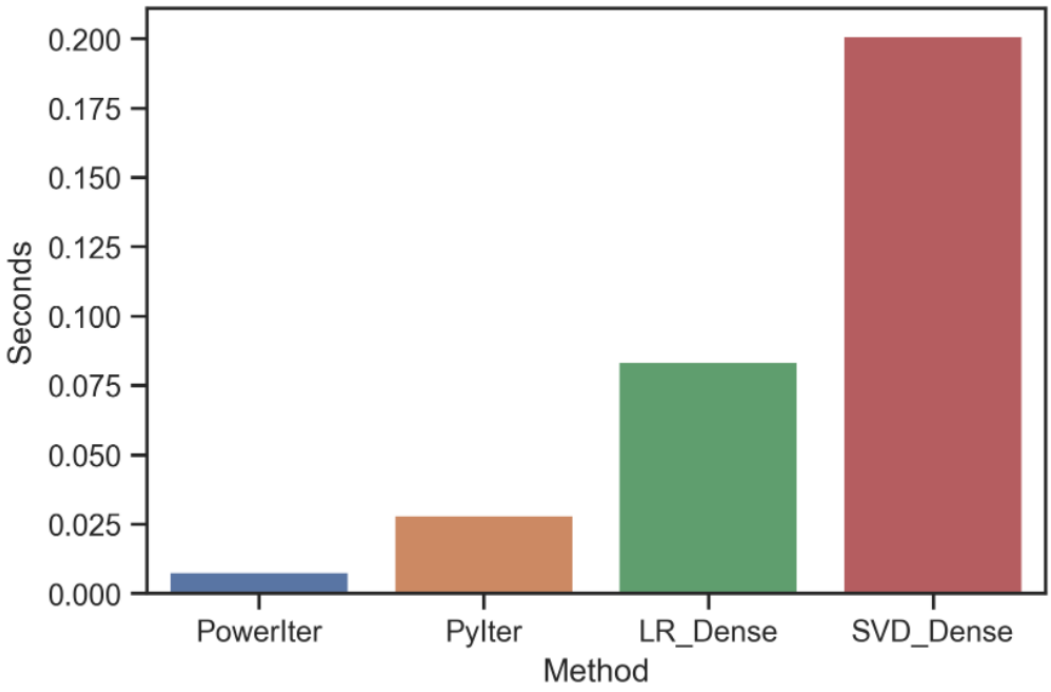

Figure 7: Runtimes for the 4 methods (with Teleportation)

Python is known for being slow, but BLAS and LAPACK are known for being extremely optimized and fast. Scikit-learn and Numpy both use BLAS and LAPACK solvers. This is done in a roundabout

manner of calling upon more libraries, and error checking, which makes these tools popular but also makes time comparison difficult.

LinearRegression() from sklearn and SVD() from numpy.linalg both use the same solver which uses SVD, but calling LinearRegression() solves it much faster than manually solving $Ax = b$ with the optimized SVD linear algebra functions. From my experience, importing things tends to be very slow. Realistically, without the redundancies and checks, the actual time to do these computations in say, assembly, would be much faster.

I wanted to point out that PyIter, when there is no teleportation (or if I were able to remove the redundancy of calculating teleportation for every page for every iteration) should theoretically be the fastest solution in the methods described. However, as it does not use any libraries, it relies on Python's interpretation, which attempts to do what the BLAS and LAPACK wrappers in scikit-learn do. PyIter in theory tries to save time by pointing directly to where the ranks are stored, but the pointers end up most likely pointing to wrappers with the actual pointers.

Perhaps even more interesting is something to note about what the observed fastest solution, Power Iteration with numpy, could do without teleportation. Sparse "ndarray"s optimize by ignoring the zeros, so the pointers to non-zero elements do what I was trying to do in the very first place. They both calculate Pagerank with a Markov Chain model, and also only normalize once at the end.

8 My Takeaway

With so many algorithms, models, interpretations and assumptions, even an oversimplification of the solution to a simple problem (Google ranking their pages) can become very complex. My takeaway is that vastly different solutions to the same fundamental problem find synchronicity in modeling chaos. Leslie Lamport, who created the Latex I am writing, used concepts from relativity when creating his work on distributed systems. These days, quantum interpretations seem to be very popular. So maybe the solution depends on the interpretation of the problem, and how people understand it with what they have learned.

9 References

[1] Bo Y.C. Ning, UC Davis Department of Statistics.

[2] Naoki Saito, UC Davis Department of Mathematics.

[3] Fushing, UC Davis Department of Statistics. "Computing a ranking network with confidence bounds from a graph-based Beta random field". Aug 3, 2011.

All trivia and rough estimates of Google's statistics were provided by Google searches.