PageRank and Beyond

Hanford Neild

May 17, 2020

1 PageRank

1.1 Background

In January of 1998, Sergey Brin and Lawrence Page developed the PageRank algorithm. It sought to bring order to the unruly mess that was the internet. At the time, many search engines such as Netscape and Yahoo worked very hard to maintain lists of relevant pages by topics like weather or finance. However, novel queries often produced junk results that would require the user to seek through many many pages before finding the answer. As noted in the original PageRank paper, " ..., as of November 1997, only one of the top four commercial search engines finds itself (returns its own search page in response to its name in the top ten results)." All this is to say, PageRank was written when the internet was still very new, and there was room for disruption. There existed a need to simplify the way we used the internet, and doing so would vastly increase the potential users of this new and exciting technology.

1.2 Introduction

The Internet can be seen as a graph where nodes represent pages and links represent edges between those pages. Viewed under these lenses, it was clear that the most popular and relevant pages are those with the most links pointing at them. However, the magic of PageRank comes from the realization that not all incoming links should be weighted equally. Links coming from reputable sources should carry more weight than ones coming from obscure edges. For example, a page that is linked from yahoo (a known collector of valid and relevant links) is probably more useful than a page on a blog that has other blog posts linking to it. The natural question that follows

is how to character the weighting or importance of some edges.
The answer is most easily described from the perspective of a page linking to many others. If our page has relevance x, then each of the m outgoing edges get a weight of $\frac{x}{m}$. From this, the formula for the PageRank(Page A) follows

$$PR(A) = \sum_{P \in N} \frac{PR(P)}{outdegree(P)}$$

where N is defined as the set of pages with links to Page A. However this formula is obviously recursive and it is not immediately obvious how one would compute such a function over every single page. Due to the exponentially increasing number of pages on the internet, it is essential that this computation be fast. To show how the PageRank team came up with an efficient implementation, it is helpful to introduce another concept.

1.3 Random Walks

In determining the best pages for a user, it is helpful to try and create a user behavior model. In the case of PageRank, this model was known as the random surfer. The random surfer starts at a random page, and randomly clicks links on the pages. This models a user seeking some information or learning about a topic. However, every so often, the random surfer randomly resets to another random page. This reset represents the user getting bored. Intuitively, the more often the random surfer visits a page, the more relevant the page is. As a quick aside, this random reset of the surfer is a vital component of the PageRank algorithm. For every company trying to rank pages, there are many more people who would seek to increase their rank artificially. This random reset avoids many problems, such as a page linking to itself or creating a cycle that would otherwise yield manipulated search results.

1.4 Principal eigenvector

After many iterations of the random surfer moving from page to page, the probability distribution over the pages (the probability that the random surfer visits the page) approaches the principal eigenvector of the normalized adjacency matrix. The adjacency matrix of a graph is simple enough to understand. For a graph of n nodes, the adjacency matrix, M, is a $n \times n$ matrix where $M_{i,j} = 1$ if $page_j$ links to $page_i$. The matrix is then normalized column-wise. The reason for this is quite simple. Each page distributes it page rank equally among the page's linked pages. While the

principal eigenvector could be found via the random surfer, this is not an efficient computation. Instead, we can use the power iteration method.

1.5 Power iteration method

The power iteration method is an iterative algorithm to approximate the largest eigenvector of a matrix. The method is described by the recurrence relation

$$PageRanks_{k+1} = \frac{M \times PageRanks_k}{||M \times PageRanks_k||}$$

So, at every iteration, the current PageRanks are multiplied by the normalized adjacency matrix, and the result is then normalized. However, PageRank adds another step to this process to take into account the random resets of the random surfer. In the following equation, d, $0 \leq d \leq 1$ is the dampening factor. The parameter d is often set to .85 meaning there is a 15% chance the random surfer randomly jumps to a new page. It represents the percent chance that the random surfer resets to a random node at any given node. R is a vector that represents a uniform page rank (Each of the n pages has rank$\frac{1}{n}$).

$$PageRanks_{k+1} = \frac{(1-d)(M \times PageRanks_k) + (d)(R)}{||(1-d)(M \times PageRanks_k) + (d)(R)||}$$

This change can be thought of as evenly spreading some relevance to all pages, on each iteration. When testing this empirically, I found that it took about 13 iterations to converge on average. While implementing PageRank, I discovered many different approaches while attempting to implement the PageRank algorithm on my own. I found PageRank implementation that was 5 lines long as others that were hundreds. Sometimes the dampening factor was applied (only once) to the matrix M before computing the eigenvalues. There are many ways to handle dangling nodes or pages that have no other pages linked to it. More sophisticated implementations that stop computing the eigenvector once a certain level of converge has been achieved. One issue with this method is that every time we get a new page added to the graph, our final page rank may change. It is not reasonable to recompute the rank of every page whenever a new page is added. To solve this problem, I will refer back to the equation for the page rank of a single page A,

$$PR(A) = \sum_{P \in N} \frac{PR(P)}{outdegree(P)}$$

If we simply know the PageRanks of all pages that link to our new page, then we can find a good enough approximation for the true PageRank of our new page.

1.6 Conclusion

PageRank is an algoritihm that has has a significant impact on the course of humanity. Every day google serves 3.5 Billion searches, and while their algorithm has surely changed over the years, I suspect it still has some elements of the original page rank algorithm. PageRank is a beautiful and simple algorithm, and it is a good example of how it doesn't take a revolutionary idea to make a revolutionary change, which I find quite inspiring.

2 MemeRank

2.1 Introduction

While researching PageRank, I had an idea for another problem I was trying to solve. For the past few months, myself and 5 others have been building a web app, called MemeHub. Memes are a cultural phenomenon by which people share humor and opinions through images. Memehub aggregates meme from popular social media platforms such as Facebook and Reddit. However, sourcing memes is easy. The challenge is, how do you find out which memes are best for a user. Humor, in general, is a tricky thing to quantify. Similar images may have completely different meanings. Likewise, captions may share many words but also have different meanings. Sarcasm may be obvious to us, but understanding it with natural language processing is a brutal problem. While my teammates and I are analyzing the image and caption of a meme to make smart recommendations, this has not been very effective. Instead, we must turn to our intelligent users to reason about these memes. Thus I introduce MemeRank.

2.2 Algorithm

Before discussing the algorithm itself, it is important to note the difference in the goals of PageRank and MemeRank. We seek to provide a ranking of memes that is specific to each user, whereas this is not a primary goal of PageRank. However we can still take inspiration from PageRank. Much like how pages formed a graph, so to do memes and users. In our graph, we have users and memes as the nodes. A edge exists between a user and a meme if the user likes the meme. One simple algorithm, would be that the MemeRank of a meme is equal to the number of likes edges for that meme. Obviously this would not yield a personalized solution. Just as with PageRank all incoming links were not weighted equally, likes from users should

not be weighted equally. Intuitively, a recommendation from a friend carries more weight than a stranger. From this we get the MemeRank formula.

$$MemeRank(Meme\ a,\ User\ u_0) = \sum_{u\ \in\ UsersLikingMemeA} similarity(u_0, u)$$

This algorithm is good, but incomplete. How do we find the similarity between two users? I define the user similarity function as

$$similarity(u_1, u_2) = \frac{|intersection|}{|union|} = \frac{|liked(u_1) \cap liked(u_2)|}{|liked(u_1) \cup liked(u_2)|}$$

Where $liked(u)$ is the set of all memes liked by user u. Again, drawing from PageRank, it need not be that all memes are weighted equally when computing user similarity. For example, two users liking an extremely popular meme means less than two users liking a niche meme. This helps avoid the snowball effect where popular memes are recommended so strongly that they are seen a disproportionate amount and thus liked disproportionately. Thus, if we define $intersection$ and $union$ as we did above, a better user similarity function would be

$$similarity(u_1, u_2) = \frac{\sum_{m \in intersection} likes(m)^{-1}}{\sum_{m \in union} likes(m)^{-1}}$$

.

2.3 Conclusion

MemeRank is a very intuitive algorithm. Given a user, we get recommendations from all other users, where more similar users have more input. Similar users are users that liked many of the same memes, where more niche memes carry more weight. I have implemented a naive version of this algorithm and am pleased with the results I see when I test it in my web app. However, much like PageRank, these ranking change every single time a user likes a meme and the cost to compute ranking is high. Of course, we need to be able to serve memes very fast (this is the point of the app) and recomputing rankings must be done thoughtfully. Thus, as per my professors, advice, my team and I have split up the process of ranking memes and serving memes. We have one process that reranks the memes when there have been a large number of likes and another that serves them. This way we can always serve memes quickly since the ranking are precomputed, and we can be sure the ranking are not too stale since we are updating them when significant changes occur. My team and I hope to fully implement this feature in our next sprint.

3 Code

The code for this project is attached in the submission zip file. There is a README.md that describes each of the files, including the memeRank.java file. Additionally, the code for this project can be found on my github at https://github.com/fordneild/pagerank