Analyzing the Reddit Subreddit Network using the PageRank and HITS Algorithms

Kevin He, Tevin Wang

December 9th 2022

1 Linear algebra background

The math behind HITS and PageRank is important to understand because it allows us to understand why these algorithms work and appreciate the creativity that went into developing them. We'll be going over eigenvectors, Markov matrices, and singular value decomposition (SVD).

1.1 Eigenvectors

A square matrix is a matrix that has the same number of columns and rows. Formally, let A be an $n \times n$ matrix for some positive integer n. Then depending on the values in this matrix, we can calculate a set of vectors that we'll call *eigenvectors* and we can calculate a set of scalars that we'll call *eigenvalues*.

- (**Eigenvalues**) The identity matrix I_n is an $n \times n$ matrix with all ones on the diagonal and zeros everywhere else. Let scalar λ be an eigenvalue of matrix A. Then the equation:

  $$\det(A - \lambda I_n) = 0$$

  gives us a polynomial in terms of λ. Solving for the distinct roots of this polynomial gives us the eigenvalues of matrix A.

- (**Eigenvectors**) Each distinct eigenvalue corresponds to a set of eigenvectors. Let's say we're given an eigenvalue λ. Then the eigenvectors are the solutions $\mathbf{x}$ to the equation:

  $$(A - \lambda I_n)\mathbf{x} = \mathbf{0}$$

  where $\mathbf{0}$ is the zero vector and $\mathbf{x} \neq \mathbf{0}$. Note that if we find a vector $\mathbf{x}$ that satisfies the above equation, all scalar multiples of $\mathbf{x}$ will also satisfy the above equation. This is by linearity as for some scalar c and assuming $(A - \lambda I_n)\mathbf{x} = \mathbf{0}$:

  $$(A - \lambda I_n)(c\mathbf{x}) = c((A - \lambda I_n)\mathbf{x}) = c\mathbf{0} = \mathbf{0}$$

  As a form of standardization, some like to normalize the eigenvectors. But it's important to understand this concept that scalar multiples also count as valid eigenvectors.

This works because given an eigenvalue λ and a corresponding eigenvector $\mathbf{x}$ of matrix A, it turns out that the equation:

$$A\mathbf{x} = \lambda\mathbf{x}$$

is satisfied. The significance comes from the fact that multiplying vector $\mathbf{x}$ by a possibly huge $n \times n$ matrix A turns out to have the exact same effect as multiplying vector $\mathbf{x}$ by a scalar λ. As we'll see later, it also means that getting an equation of the form

$$(\mathbf{vector})(\mathbf{known\ scalar}) = (\mathbf{known\ matrix})(\mathbf{vector})$$

allows us to actually solve for (**vector**) by finding an eigenvector of (**known matrix**) corresponding to eigenvalue (**known scalar**).

1.2 Markov matrices

An $n \times n$ matrix M for some positive integer n can be called a *Markov matrix* if the following 2 properties hold:

1. All entries are nonnegative
2. The sum of the entries in each column is 1

Even more specific, a *positive Markov matrix* is a Markov matrix that actually has all positive entries. It turns out that positive Markov matrices have very interesting properties:

1. The largest eigenvalue is 1
2. Absolute value of all other eigenvalues is strictly less than 1
3. The eigenvector corresponding to an eigenvalue of 1 is an "attracting steady state"

What is an "attracting steady state"? It's a vector $\mathbf{x}$ that when transformed by multiplying it by positive Markov matrix M on the left, it actually remains unchanged. So $M\mathbf{x} = \mathbf{x}$. Thus, we see that $\mathbf{x}$ is the eigenvector of M corresponding to an eigenvalue of 1. It's also the vector that when given any vector $\mathbf{w}$, multiplying $\mathbf{w}$ by M many times converges to $\mathbf{x}$.

The significance comes from the fact that regular Markov matrices *don't guarantee an attractive steady state exists*. On the other hand, positive Markov matrices do. Having this notion of a vector that matrix M *converges to* is an extremely useful tool to have.

1.3 Singular value decomposition

Singular value decomposition allows us to break down *any matrix* into 3 parts. There's no requirement that a matrix be square, which allows this to become a rather universal tool. Let this matrix be A with dimension $m \times n$. Then we can write:

$$A = U\Sigma V^T$$

where U, Σ, and V^T are matrices. We'll be using the important fact that A^TA and AA^T are square matrices. In particular, A^TA is $n \times n$, AA^T is $m \times m$, and both turn out to actually be *symmetric*. A symmetric matrix doesn't change if you flip it over the main diagonal. The reason this is useful is because symmetric matrices have all non-negative eigenvalues, meaning that taking the square root of the eigenvalues will still result in real numbers.

Let's go over what each matrix is like:

- (Σ) This matrix is $m \times n$. First we calculate the eigenvalues of A^TA. Since A^TA is symmetric, we know the eigenvalues are non-negative. So we can take the square root of all the eigenvalues, order them from greatest to least, and then put them on the "diagonal" of matrix Σ.

Then set every other entry of Σ to zero.

What I mean by "diagonal" is you start from the upper left hand corner and work your way down and to the right. An example is given below:

$$\begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

- (U) This matrix is $m \times m$. Note AA^T is $m \times m$ and symmetric. Once we find the m eigenvectors of AA^T, we then put the unit eigenvector corresponding to the largest eigenvalue in column 1 of U, the unit eigenvector corresponding to the second largest eigenvalue in column 2 of U, etc. This should populate the entire $m \times m$ matrix.

  Each column, so each eigenvector, has a norm of 1. It also turns out that the columns of U are all orthogonal to each other. Thus, it turns out that U is actually an orthogonal matrix.

- (V) This matrix is $n \times n$. It's very to U. Note $A^T A$ is $n \times n$ and symmetric. Once we find the n eigenvectors of $A^T A$, we then put the unit eigenvector corresponding to the largest eigenvalue in column 1 of V, the unit eigenvector corresponding to the second largest eigenvalue in column 2 of V, etc. This should populate the entire $n \times n$ matrix.

  Each column, so each eigenvector, has a norm of 1. It also turns out that the columns of V are all orthogonal to each other. Thus, it turns out that V is actually an orthogonal matrix.

We call the eigenvector that corresponds to the largest eigenvalue the *principal eigenvector*. This vector will become important in the PageRank and HITS algorithms as we'll explore later. In this way, we can perform SVD on some matrix A to get the principal eigenvectors of $A^T A$ and AA^T by looking at V and U respectively.

We are now ready to see how the PageRank and HITS algorithms work.

2 Algorithms and computation

Everyone at some point in their lives have used the Google search engine. You type words into the search bar and then Google magically generates websites that relate to the words you input. In this way, Google is able to rank websites and display the top ones first for the user's convenience.

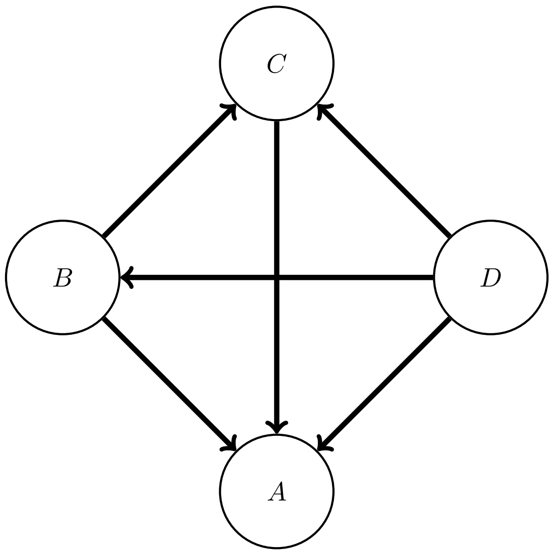
The PageRank and HITS algorithms are able to rank websites. For most of the population, this is good enough to understand. However, I'll be diving into the math and reasoning behind these algorithms to hopefully reveal the genius yet simple concepts behind them.

Both of these algorithms are rooted in linear algebra and they both rely on the idea that websites link to other websites. This gives rise to the idea of *graphs* that consist of nodes and directed edges (arrows). Yet, these algorithms differ in ways that I'll point out later in the section. We'll start with PageRank.

2.1 PageRank algorithm

Let's say we're given a bunch of websites along with the links among them. We'll treat websites as nodes and we'll draw an arrow from node a to node b if a link on website a takes someone to website b. This is a *directed graph.* From now on, I'll use the terms nodes and websites interchangeably.

The PageRank algorithm takes a graph as input and as output, assigns to each node a weight. Higher weights correspond to a higher probability that a surfer who's randomly clicking on links ends up at that website. We can think of an arrow as a "vote" for the website that the arrow points to. Websites with lots of arrows pointing to it logically will mean there's probably going to be more activity there.



In the graph above, we can predict that PageRank will rank node A higher because every other node points to it. Let's denote the *PageRank* of a node as the weight that is being assigned to it. For node u, that will be represented as $PR(u)$ from now on. Letting N be the number of nodes in the graph, let each node start off with a PageRank of $\frac{1}{N}$.

There are two ways to actually perform the PageRank algorithm. One way is iterative and one way involves matrices. I'll start by explaining the iterative method because only once you understand the iterative method can you understand the matrix method. Consider the graph above.

At each iteration, we update all the Page Ranks at once. B has 2 outward arrows so B transfers half its current PageRank through each of its arrows. D has 3 outward arrows so D transfers a third of its current PageRank through each of its arrows. C has 1 arrow so C transfers its entire PageRank through that arrow.

This tells us that after one iteration, we can denote

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}$$

We can do the same for the other nodes. Let's now generalize this. Let V be the set of all nodes. Then

$$\forall u \in V, PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)} \qquad \text{(Eq. 1)}$$

where B_u is the set of all nodes that have an arrow pointing to node u and $L(v)$ is the total number of outward-pointing arrows that node v has. Iterating to infinity should eventually lead to a fixed set of Page Ranks (in theory ...). Now we're ready to consider the matrix implementation.

Let's build an adjacency matrix M based on the information in the graph. Let p_i for $1 \leq i \leq N$ be the nodes of the graph. Then

$$M = \begin{pmatrix} l(p_1,p_1) & l(p_1,p_2) & \dots & l(p_1,p_N) \\ l(p_2,p_1) & l(p_2,p_2) & \dots & l(p_2,p_N) \\ \vdots & \vdots & \ddots & \vdots \\ l(p_N,p_1) & l(p_N,p_2) & \dots & l(p_N,p_N) \end{pmatrix}$$

Where

$$l(p_i,p_j) = \begin{cases} \frac{1}{L(p_j)} & \text{if } p_j \text{ has an arrow to } p_i \\ 0 & \text{if } p_j \text{ has no arrow to } p_i \end{cases}$$

In english, $l(p_i,p_j)$ is the ratio of the number of links from p_j to p_i to the total number of links coming out of node p_j. Thus, this means that the following equation holds:

$$\forall j \in \mathbb{Z}, 1 \leq j \leq N, \sum_{i=1}^{N} l(p_i,p_j) = 1$$

This makes intuitive sense because given a node p_j, distributing $\frac{1}{\text{number of nodes } p_j \text{ points to}}$ to each of the nodes p_j points to, we should get 1. So each column of M adds up to 1. This should ring a bell. We also notice that by the way we define function l, each entry of M is non-negative. So M is a Markov matrix!

Let vector $\mathbf{R}$ hold the PageRanks of every node in the graph so that

$$\mathbf{R} = \begin{pmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_N) \end{pmatrix}$$

Then consider the expression following expression:

$$
\begin{aligned}
M\mathbf{R} &= \begin{pmatrix} l(p_1,p_1) & l(p_1,p_2) & \dots & l(p_1,p_N) \\ l(p_2,p_1) & l(p_2,p_2) & \dots & l(p_2,p_N) \\ \vdots & \vdots & \ddots & \vdots \\ l(p_N,p_1) & l(p_N,p_2) & \dots & l(p_N,p_N) \end{pmatrix} \begin{pmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_N) \end{pmatrix} \\
&= \begin{pmatrix} l(p_1,p_1)PR(p_1) & l(p_1,p_2)PR(p_2) & \dots & l(p_1,p_N)PR(p_N) \\ l(p_2,p_1)PR(p_1) & l(p_2,p_2)PR(p_2) & \dots & l(p_2,p_N)PR(p_N) \\ \vdots & \vdots & \ddots & \vdots \\ l(p_N,p_1)PR(p_1) & l(p_N,p_2)PR(p_2) & \dots & l(p_N,p_N)PR(p_N) \end{pmatrix} \\
&= \begin{pmatrix} 0 & l(p_1,p_2)PR(p_2) & \dots & l(p_1,p_N)PR(p_N) \\ l(p_2,p_1)PR(p_1) & 0 & \dots & l(p_2,p_N)PR(p_N) \\ \vdots & \vdots & \ddots & \vdots \\ l(p_N,p_1)PR(p_1) & l(p_N,p_2)PR(p_2) & \dots & 0 \end{pmatrix}
\end{aligned}
$$

where $l(p_i,p_i) = 0$ since a node can't point to itself. Then we see that (Eq. 1) is represented in every row of MR. Looking at the 1st row of MR, we notice that $PR(p_1) = l(p_1,p_2)PR(p_2) + \ldots + l(p_1,p_N)PR(p_N)$ where if p_2 points to p_1, the exact form of (Eq. 1) is taken with

$$l(p_1,p_2)PR(p_2) = \frac{PR(p_2)}{L(p_2)}$$

and with $l(p_1,p_j) = 0$ removing all the terms where p_j doesn't actually point to p_1. Thus, it actually turns out that

$$\mathbf{R}_{t+1} = M\mathbf{R}_t \qquad \text{(Eq. 2)}$$

where if $\mathbf{R}_t$ represents the current PageRanks, $\mathbf{R}_{t+1}$ represents the PageRanks after an iteration. This gives rise to important observation:

Adjacency Matrix M updates the PageRank vector $\mathbf{R}$ of the graph.

Repeatedly applying M to the initial vector $\left(\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N}\right)$ should eventually result in a final set of PageRanks in the iterative method. This should remind us of a steady state vector. Referring back to the section on Markov matrices, we have that solving $M\mathbf{x} = \mathbf{x}$ finds the steady state vector $\mathbf{x}$. (Eq. 2) resembles that form.

With the insight that after many iterations, performing M should have little effect on the PageRanks, we should be able to write

$$\mathbf{R} = M\mathbf{R}$$

where solving for $\mathbf{R}$ ends up being the problem of solving for the eigenvector of M that corresponds to an eigenvalue of 1. Thus, we should be done, right? However, it turns out we can't guarantee a steady state based solely off a Markov matrix. Random walks might be cyclic! We need a *positive* Markov matrix. So we need to tweak our methods a bit. This leads into the introduction of the *damping factor*, d.

In the real world, a surfer who's browsing the internet won't click on links forever. At some point, you're going to stop. We let d be the probability that a person currently on a website decides to leave that website. Google has found d to be approximately 0.85. With this new factor, let's reconsider how we calculate $PR(u)$.

Before, we added up the PageRanks coming from nodes pointing to node u. However, this new probability means it's possible that the arrows pointing to u won't even be "traveled" on. So the expected number of "votes" coming from other websites is now

$$(d)\left(\sum_{v\in B_u}\frac{PR(v)}{L(v)}\right)$$

But then isn't there a chance that a user on website u decides to stay on u? With N total nodes, a user is on node u with probability $\frac{1}{N}$ and there's a $1-d$ probability of staying, so in total:

$$PR(u) = \frac{1-d}{N} + d\sum_{v\in B_u}\frac{PR(v)}{L(v)} \qquad \text{(Eq. 3)}$$

Writing this in terms of matrices, we have that

$$\mathbf{R}_{t+1} = \begin{pmatrix} \frac{1-d}{N} \\ \frac{1-d}{N} \\ \vdots \\ \frac{1-d}{N} \end{pmatrix} + d\begin{pmatrix} l(p_1,p_1) & l(p_1,p_2) & \dots & l(p_1,p_N) \\ l(p_2,p_1) & l(p_2,p_2) & \dots & l(p_2,p_N) \\ \vdots & \vdots & \ddots & \vdots \\ l(p_N,p_1) & l(p_N,p_2) & \dots & l(p_N,p_N) \end{pmatrix}\mathbf{R_t}$$

For convenience and future reference, we let $\mathbf{1} = (1,1,\dots,1)$ to get that

$$\mathbf{R}_{t+1} = \frac{1-d}{N}\mathbf{1} + dM\mathbf{R_t}$$

Since the entries of $\mathbf{R}_t$ sum to 1, we know that $E\mathbf{R}_t = \mathbf{1}$ where E is the square matrix of all ones. So we can actually do some factoring:

$$\mathbf{R}_{t+1} = \frac{1-d}{N}(E\mathbf{R}_t) + dM\mathbf{R_t} = \left(\frac{1-d}{N}E + dM\right)\mathbf{R}_t = \hat{M}\mathbf{R}$$

where we define

$$\hat{M} = \frac{1-d}{N}E + dM$$

I now claim that $\hat{M}$ is a *positive* Markov matrix, which means that a random walk leads to an attracting steady state vector. We've shown previously that M is a Markov matrix. That tells us every column of dM sums to d. Since $d \in (0,1)$, we know that the entries of dM remain non-negative as well ($d = 0, 1$ seems unreasonable and is nearly impossible for practical purposes).

$\frac{1-d}{N}E$ is a matrix where every term is $\frac{1-d}{N}$, so adding that to dM results in every column now summing up to $d + (N)(\frac{1-d}{N}) = d + (1-d) = 1$. More importantly, $d \in (0,1) \implies \frac{1-d}{N} > 0$ so when we add a matrix of all $\frac{1-d}{N}$ to dM, which has all non-negative entries, we actually end up with a matrix of all strictly positive entries! So $\hat{M}$ is a positive Markov matrix.

Now we can look at the equation $\mathbf{R}_{t+1} = \hat{M}\mathbf{R}_t$ and with the same intuition that after many iterations, performing $\hat{M}$ should have little effect on the PageRanks and we're able to write

$$\mathbf{R} = \hat{M}\mathbf{R}$$

So finding $\mathbf{R}$ consists of finding the steady state vector of the positive Markov matrix $\hat{M}$, which we now know exists for sure given the properties of positive Markov matrices. Since positive Markov matrices have largest eigenvalue 1, it turns out that $\mathbf{R}$ is also the principal eigenvector of $\hat{M}$.

In summary, we're given a graph with websites as nodes and links as edges. In particular, there are N nodes. We find an adjacency matrix M from this. Then, using a damping factor d, which is typically set to 0.85, we find matrix $\hat{M} = \frac{1-d}{N}E + dM$. Finding the principal eigenvector of $\hat{M}$ gives us $\mathbf{R}$, which when scaled so entries sum to 1, gives us the PagesRanks of all nodes and thus all websites.

2.2 HITS algorithm

In the PageRank algorithm, a website's value is solely based on how many arrows point to it. The HITS algorithm provides an entirely new dimension. HITS says that a website with many arrows pointing to it as a high *authority score*. But HITS also says that a website pointing to many websites with good authority scores has a higher *hub score*.

Like PageRank, we're given a graph. A graph is actually constructed based off of the words used in the search bar. Ask.com, a search engine, still implements this HITS algorithm. The HITS algorithm takes as input a graph and as output gives each node an authority score and a hub score. We'll consider the iterative method first and that should lead into a derivation of the way to use SVD in order to calculate final scores.

At each step, we update authority scores and then update hub scores. We update authority scores by going to each node u and adding the sum of the hub scores of the nodes pointing to u. We write this as

$$A(u) = \sum_{v \in B_u} H(v)$$

where $A(x)$ and $H(x)$ are the authority and hub scores, respectively, of node x and B_u is the set of nodes that point to node u. We then update hub scores by going to each node u and adding the sum of the authority scores of all the nodes that u points to. We write this as

$$H(u) = \sum_{v \in C_u} A(v)$$

where C_u is the set of nodes that u points to. This makes intuitive sense. A good hub should have links to high quality websites, while a high quality website should be linked to by a good hub. Now let's add some linear algebra. Let vector

$$\mathbf{v} = \begin{pmatrix} A(p_1) \\ A(p_2) \\ \vdots \\ A(p_N) \end{pmatrix}$$

represent the authority scores of all the nodes and let vector

$$\mathbf{u} = \begin{pmatrix} H(p_1) \\ H(p_2) \\ \vdots \\ H(p_N) \end{pmatrix}$$

represent the hub scores all the nodes where N is the total number of nodes and p_i for $1 \leq i \leq N$ are the nodes. For iteration, we let $\mathbf{v}_k$ and $\mathbf{u}_k$ represent the authority and hub scores, respectively, at iteration k. We will need an adjacency matrix A that comes from the graph. We construct it by letting $A_{ij} = 1$ if there exists an arrow from node i to node j and $A_{ij} = 0$ if not.

Let's represent updating authority scores using matrices. It turns out that

$$\mathbf{v} = A^T \mathbf{u} \qquad \text{(authority update)}$$

To see this consider the first entry of $\mathbf{v}$. This comes from $A^T_{*1}\mathbf{u}$, which represents the sum of the hub scores of all the nodes that point to node 1. From before, this should be the authority score of node 1. The same reasoning is valid for other nodes. Then updating hub scores turns out to be

$$\mathbf{u} = A\mathbf{v} \qquad \text{(hub update)}$$

To see this consider the first entry of $\mathbf{u}$. This comes from $A^T_{1*}\mathbf{v}$, which represents the sum of the authority scores of all the nodes that node 1 points to. From before, this should be the hub score of node 1. The same reasoning is valid for other nodes.

We start by letting

$$\mathbf{u}_0 = \mathbf{1}$$

$$\mathbf{v}_0 = A^T\mathbf{1}$$

where $\mathbf{v}_0$ is made that way to account for edge cases in the future formula. First we update authority scores. Then to find $\mathbf{v}_1$, we have $\mathbf{v}_1 = A^T\mathbf{u}_0$. Second we update hub scores. To find $\mathbf{u}_1$, we have $\mathbf{u}_1 = A\mathbf{v}_1$ where $\mathbf{v}_1$ is used rather than $\mathbf{v}_0$ because it's more recent. Substituting gives us

$$\mathbf{u}_1 = A\mathbf{v}_1 = A(A^T\mathbf{u}_0) = (AA^T)\mathbf{u}_0$$

Now solving for $\mathbf{v}_2$, we get $\mathbf{v}_2 = A^T\mathbf{u}_1 = A^T(A\mathbf{v}_1) = (A^TA)\mathbf{v}_1$. Continuing, we observe a pattern. It turns out that

$$\mathbf{v}_k = (A^TA)\mathbf{v}_{k-1}$$

$$\mathbf{u}_k = (AA^T)\mathbf{u}_{k-1}$$

for all $k \geq 1$. These are the ultimate iterative formulas. Repeatedly performing this iteration until $\mathbf{v}_k$ and $\mathbf{u}_k$ aren't changing much and remembering to normalize the vectors either after every iteration or after finitely many iterations to prevent infinite values, we should get the result of the HITS algorithm.

Now, we start deriving a method that uses SVD and eigenvectors to implement HITS. Let's start substituting:

$$\mathbf{v}_k = (A^TA)\mathbf{v}_{k-1} = (A^TA)^2\mathbf{v}_{k-2} = \ldots = (A^TA)^k\mathbf{v}_0$$

Similarly,

$$\mathbf{u}_k = (AA^T)\mathbf{u}_{k-1} = (AA^T)^2\mathbf{u}_{k-2} = \ldots = (AA^T)^k\mathbf{u}_0$$

Our goal is to compute

$$\lim_{k\to\infty} \mathbf{v}_k, \mathbf{u}_k$$

So we need to figure the behavior of $(A^TA)^k$ and $(AA^T)^k$ for very large k. Let's first consider A^TA without the k exponent. Since A^TA is a symmetric square matrix, there exists a way to write $A^TA = Q\Lambda Q^T$ for some orthogonal matrix Q that holds eigenvectors and diagonal matrix Λ that holds eigenvalues. Let's say that the eigenvalues in Λ are ordered from greatest to least, left to right.

This means that if

$$Q = (\mathbf{q}_1 \mid \mathbf{q}_2 \mid \ldots \mid \mathbf{q}_n)$$

$\mathbf{q}_1$ is the principal eigenvector of A^TA. It's another property that $(A^TA)^k = (Q\Lambda Q^T)^k = Q\Lambda^kQ^T$ using the fact that Q is orthogonal. Then consider the following algebra that deals with matrices:

$$\begin{aligned}
(A^TA)^k &= Q\Lambda^kQ^T \\
&= \big(\mathbf{q}_1 \mid \mathbf{q}_2 \mid \cdots \mid \mathbf{q}_n\big) \begin{pmatrix} \lambda_1^k & 0 & \cdots & 0 \\ 0 & \lambda_2^k & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \lambda_n^k \end{pmatrix} \begin{pmatrix} \mathbf{q}_1^T \\ \hline \mathbf{q}_2^T \\ \hline \cdots \\ \hline \mathbf{q}_n^T \end{pmatrix} \\
&= \big(\mathbf{q}_1 \mid \mathbf{q}_2 \mid \cdots \mid \mathbf{q}_n\big) \begin{pmatrix} \lambda_1^k\mathbf{q}_1^T \\ \hline \lambda_2^k\mathbf{q}_2^T \\ \hline \cdots \\ \hline \lambda_n^k\mathbf{q}_n^T \end{pmatrix} \\
&= \lambda_1^k\mathbf{q}_1\mathbf{q}_1^T + \lambda_2^k\mathbf{q}_2\mathbf{q}_2^T + \ldots + \lambda_n^k\mathbf{q}_n\mathbf{q}_n^T
\end{aligned}$$

Since $\lambda_1 > \lambda_2 > \ldots > \lambda_n$, we have that λ_1^k dominates all the other eigenvalues as k approaches infinity. So in the limit, we can say that

$$(A^T A)^k = \lambda_1^k \mathbf{q}_1 \mathbf{q}_1^T$$

Then going back to $\mathbf{v}_k$, we have that

$$\mathbf{v}_k = \lambda_1^k \mathbf{q}_1 \mathbf{q}_1^T \mathbf{v}_0$$

Since $\mathbf{q}_1^T \mathbf{v}_0$ is a dot product, it's a scalar. λ_1^k is also a scalar, so it turns out that $\mathbf{v}_k$ is a multiple of $\mathbf{q}_1$. Thus, normalizing at the end or after every iteration allows us to drop the scalars to get that $\mathbf{v}_k = \mathbf{q}_1$, using the fact that the vectors that form the columns of Q have norms of 1. Thus, it turns out that $\lim_{k\to\infty} \mathbf{v}_k = \mathbf{q}_1$, or that the authority scores vector approaches the principal eigenvector of $A^T A$.

Doing the exact same thing with $\mathbf{u}_k$ and with the same reasoning, we get that $\mathbf{u}_k$ approaches the principal eigenvector of AA^T as k approaches infinity. This relates to SVD because using the conventional way to represent $A = U\Sigma V^T$, the principal eigenvector of $A^T A$ turns out to be the first column of V and the principal eigenvector of AA^T turns out to be the first column of U.

2.3 Power method

To implement these algorithms in code, it's necessary to efficiently be able to find the principal eigenvector of a matrix A. Especially in the case of large data sets, solving for the exact eigenvector is slow. Instead, an accurate approximation can be obtained by a method called the *Power method.* We use a built-in Power method function in our Reddit analysis later on. Here's how it works.

We can randomly generate a nonzero vector $\mathbf{x}_0$. This serves as an initial approximation. Then, if we're trying to find the dominant eigenvector of matrix A, we repeatedly multiply vector $\mathbf{x}_0$ by A. This gives us

$$\mathbf{x}_k = (A)^k \mathbf{x}_0$$

We can't iterate to infinity, but it turns out that it doesn't actually take too long to obtain a close enough approximation. With scaling, this method is extremely useful for analyzing data sets in the real world.

3 Algorithm and code design decisions

It is important to understand how mathematical intuition and proofs correspond with algorithms and code design. This is why we must describe a general summary of our code implementation in this section. Full code will be provided as an appendix.

3.1 Trivial network

A trivial network means that the graph only has 4 nodes, and is not large enough where time complexity matters.

We start by taking in the input of website vectors. For example, one possible arrangement is $\begin{pmatrix} "twitter.com" \\ "youtube.com" \\ "facebook.com" \\ "instagram.com" \end{pmatrix}$

Then, we create a list of lists called *lists*: The ith index of lists is itself a list. This list corresponds to the index of where the directed edges are pointing (in this case, hyperlinks) of a particular source based on the website vector. The website vector is 1-indexed. For example: $[[2, 3], [3], [1, 2], [2, 3, 4]]$

For PageRank:

We can now create our Markov matrix based on these lists. We first initialize a *graph* with zeros (size $N \times N$ where N is the number of sources) and then loop through each cell, where $\text{graph}_{ij} = \frac{1}{\text{number of directed edges for a particular source}}$ if and only if there exists a directed edge from j to i.

Now that we have a Markov matrix, in order to make a random walk correspond with an attracting steady state (*positive* Markov matrix), we must adjust the matrix by a damping factor. We follow the mathematical formula shown in part 2, except that we first fill a matrix with $\frac{1}{N}$, where N is the number of sources, and then multiply that by $1 - d = 0.15$. Then we simply add $d * graph$ to complete the formula.

Once we finish creating our positive Markov matrix, we can call the built-in function *eigvecs*, and take the last vector (one of that corresponds with the highest eigenvalue, which must be 1).

Finally, we scale the resulting eigenvector so that it is positive and sums to 1, a requirement of PageRank. We can now append our website vector and sort by PageRank, and we are done.

For HITS:

We utilize the same structure: a website vector and a list of lists. Instead of creating a Markov matrix, we proceed by making an adjacency matrix. We do this by first creating a matrix of all zeros of size $N \times N$ where N is the number of sources. Then, we iterate through every cell and set $\text{graph}_{ij} = 1$ if and only if there exists a directed edge between i and j indexes.

Then, we can get the spectral value decomposition to find authority rank vector $\mathbf{v}_1$ and hub rank vector $\mathbf{u}_1$. To sort by the corresponding ranks, we take the absolute value of each vector.

Then, we can again append the website vector and sort by authority as well as sort by hub. Then, we are done.

3.2 Reddit network

I took a subreddit dataset from Stanford SNAP that provided information about directed edges between a SOURCE_SUBREDDIT and a STARTING_SUBREDDIT. The dataset looked at hyperlinks from the post body of subreddits from January 2014 to April 2017.

Since the file comes as a TSV file, I need to first turn the table into a DataFrame for further analysis. We get that there are 286561 edges, that is a lot!

Then, we want to create a dictionary that maps sources to indices for $O(1)$ access when we create our list of lists of links. We also want to simultaneously gather our sources vector.

We do this by first looping through the DataFrame and if SOURCE_SUBREDDIT is not already in the dictionary, we must add it to our dictionary and add it to our sources vector. We then loop through the Dataframe again and if TARGET_SUBREDDIT is not already in the dictionary, we must add it to our dictionary and add it to our sources vector (some subreddits may not have hyperlinks).

Then to finally get our *links* list of lists, we must create a new dictionary, to check if we've added a source already to links. Then if we have not added the source, we push a set onto links to ensure fast iteration and fast insertion while preventing any duplicates. If we already have, we simply push the index of the target onto the specific list in links. We proceed to push empty sets for the remaining links, since they do not link to anything.

Finally, we have our data ready for PageRank and HITS.

For PageRank:

We create the graph as shown in the trivial network. However, creating a new filled matrix takes way too much memory for our computers to handle. Therefore, the better approach is to loop through the graph and apply the damping factor (this is slow time complexity, but it is the better choice).

Then, let's use the power method (built-in function) to find the attracting steady state vector. We start with an estimate vector where each element is $\frac{1}{N}$ where N is the number of sources. This way, we assume initially that each vector has the same PageRank.

The power method is a lot faster than simply computing all the eigenvalues and eigenvectors since we only care about the dominant eigenvalue, which is 1.

We then scale the eigenvector so that the sum is 1, append our sources vector, and then sort by PageRank.

For HITS:

We create the adjacency matrix in the exact same way.

Now we must find singular vectors $\mathbf{v}_1$ and $\mathbf{u}_1$. We can do so by power method (built-in function), for efficiency. We set each vector to an initial estimate where each element is $\frac{1}{N}$ where N is the number of sources. Then we apply the power method on $\text{graph}^T\text{graph}$ for $\mathbf{v}_1$ and $\text{graph} * \text{graph}^T$ for $\mathbf{u}_1$.

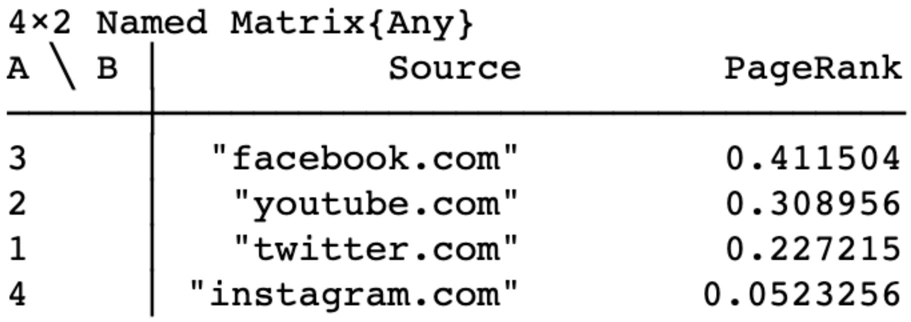
We then concatenate the sources vector and sort by authority and by hub accordingly to produce our results.

4 Results and interpretation

For clarification, the first column corresponds to the index of the source, not the rank. The ranks are sorted in decreasing order from top to bottom.

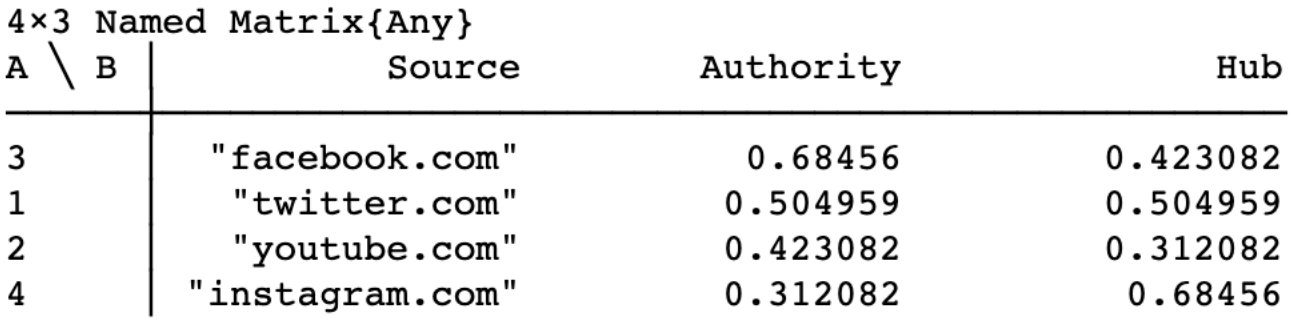
4.1 Trivial network

PageRank:



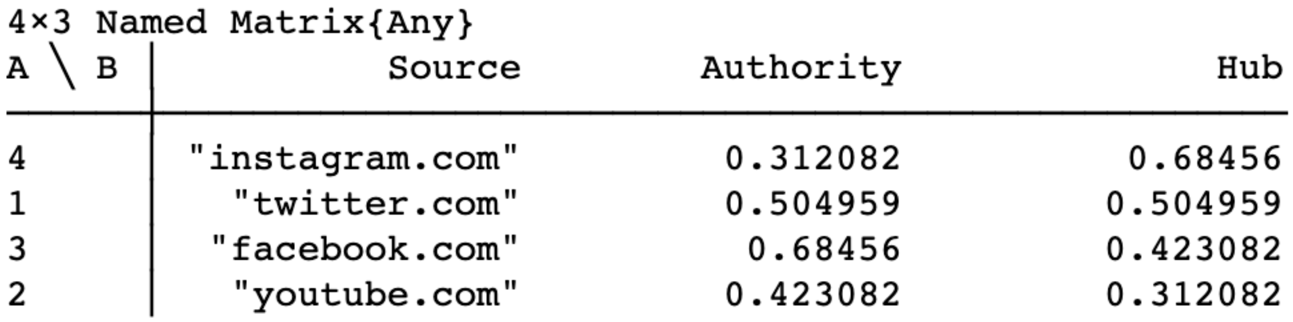
```
4×2 Named Matrix{Any}
A ╲ B │           Source      PageRank
──────┼─────────────────────────────────
3     │   "facebook.com"      0.411504
2     │    "youtube.com"      0.308956
1     │    "twitter.com"      0.227215
4     │ "instagram.com"      0.0523256
```

HITS by Authority:



```
4×3 Named Matrix{Any}
A ╲ B │           Source     Authority           Hub
──────┼──────────────────────────────────────────────
3     │   "facebook.com"       0.68456      0.423082
1     │    "twitter.com"      0.504959      0.504959
2     │    "youtube.com"      0.423082      0.312082
4     │ "instagram.com"      0.312082       0.68456
```

HITS by Hub:



```
4×3 Named Matrix{Any}
A ╲ B │           Source     Authority           Hub
──────┼──────────────────────────────────────────────
4     │ "instagram.com"      0.312082       0.68456
1     │    "twitter.com"      0.504959      0.504959
3     │   "facebook.com"       0.68456      0.423082
2     │    "youtube.com"      0.423082      0.312082
```

4.2 Reddit network

PageRank:

```
20×2 Named Matrix{Any}
A ╲ B │              Source      PageRank
──────┼─────────────────────────────────
35    │         "askreddit"     0.0121459
23    │              "iama"     0.0117198
7742  │              "pics"    0.00586916
11295 │            "videos"    0.00563957
127   │      "outoftheloop"    0.00402394
27866 │     "todayilearned"    0.00391077
5297  │              "mhoc"     0.0038693
563   │            "gaming"    0.00380185
1     │   "leagueoflegends"    0.00379357
43    │    "writingprompts"    0.00374519
738   │             "funny"    0.00364018
1386  │         "worldnews"    0.00340026
1257  │              "news"    0.00304047
258   │      "pcmasterrace"    0.00300983
6495  │        "technology"    0.00282534
89    │  "explainlikeimfive"   0.00275315
1828  │           "science"    0.00265422
491   │             "games"    0.00253972
140   │            "movies"    0.00241172
17463 │        "bestof2015"    0.00237747
```

HITS by Authority:

```
20×3 Named Matrix{Any}
A ╲ B │                 Source     Authority             Hub
──────┼───────────────────────────────────────────────────────
35    │            "askreddit"      0.286246        0.107565
23    │                 "iama"      0.225876       0.0923254
7742  │                 "pics"      0.181623     0.000563776
11295 │               "videos"      0.173853     0.000201055
27866 │        "todayilearned"      0.172722             0.0
738   │                "funny"      0.155721       0.0259283
1386  │            "worldnews"      0.153163      0.00320554
1257  │                 "news"      0.142677      0.00810208
27868 │        "adviceanimals"      0.118612             0.0
563   │               "gaming"      0.118028       0.0446724
89    │    "explainlikeimfive"      0.117069       0.0828201
1828  │              "science"      0.112375      0.00659387
127   │         "outoftheloop"      0.110032        0.133766
6495  │           "technology"      0.108211       0.0197608
27870 │                  "wtf"       0.10584             0.0
269   │       "showerthoughts"      0.104325        0.073693
3594  │             "politics"      0.102769        0.015815
7461  │                 "gifs"      0.102288     3.19847e-5
71    │       "subredditdrama"      0.101395        0.208824
145   │                 "tifu"     0.0960996       0.0508054
```

HITS by Hub:

```
20×3 Named Matrix{Any}
A ╲ B │                    Source     Authority             Hub
──────┼──────────────────────────────────────────────────────────
71    │          "subredditdrama"      0.101395        0.208824
80    │               "copypasta"     0.0127593        0.140558
127   │            "outoftheloop"      0.110032        0.133766
17    │             "circlebroke"     0.0295397         0.13034
32    │    "circlejerkcopypasta"     0.0100197        0.128305
627   │                   "drama"     0.0316244        0.127558
6401  │         "shitliberalssay"     0.0106842        0.122778
9245  │            "justunsubbed"    0.00955009        0.117095
131   │              "conspiracy"     0.0739141        0.115691
46    │           "hailcorporate"     0.0278471        0.113726
327   │                    "self"     0.0588681        0.110519
35    │               "askreddit"      0.286246        0.107565
344   │      "nostupidquestions"     0.0562355        0.107181
7375  │   "bestofoutrageculture"     0.0134311        0.102165
163   │              "karmacourt"     0.0116986       0.0986079
264   │            "circlebroke2"     0.0194316       0.0950776
43    │          "writingprompts"     0.0757081       0.0938505
190   │          "shitredditsays"      0.039926       0.0934937
184   │                    "help"     0.0250544       0.0930448
23    │                    "iama"      0.225876       0.0923254
```

4.3 Conclusion

Let's consider the Reddit network results. We'll consider PageRank and HITS separately at first and gauge whether the outputs of the algorithms make sense. Then, we'll analyze the differences between the two algorithms and how they reveal the behind-the-scenes implementation details of these methods.

"askreddit" is a subreddit where questions are posed as posts and answered by a comment section of Reddit users. This subreddit has 38.8 million members and some posts can be seen as having nearly 30,000 comments. It's also active, so there's so many posts that it's hard to keep track of. As a result, if a user in another subreddit poses a question, there's a decent chance that it's already been answered in "askreddit" and a user will subsequently provide a link to "askreddit."

Treating subreddits as nodes, then algorithmically speaking, this increases the number of arrows pointing to "askreddit" and thus increases both PageRank and authority score. This makes it hardly surprising that "askreddit" ended up having the highest PageRank and authority score among all subreddits.

The subreddit with, by far, the highest hub score is **"subredditdrama"**. After a little bit of exploring, it's once again clear why. The subreddit specializes in discussing the drama that goes on in every other subreddit. As a result, every post references some other subreddit. This subreddit isn't negligible either, since it has nearly 900,000 members. Algorithmically speaking, this means that the node representing "subredditdrama" has lots of edges that point to other nodes and thus leads to a higher hub score.

For practical purposes, this means that if you searched up drama in the search bar hoping to learn more about drama in Reddit, a search engine implementing HITS would put "subredditdrama" as the first result. This makes sense, but it also reveals that an entirely new dimension is needed to effectively implement an actual search engine. There needs to be another algorithm that matches the topic of the user's interest to the subreddit's topic in a way that almost seems to *override* hub score in certain situations.

Subreddit "pics" was third on both PageRank and authority score. This subreddit is full of pictures of any kind. With similar reasoning to "askreddit," this means that any picture in some other subreddit could be directed to "pics" by users who found the image cool. This creates arrows to "pics" and thus higher PageRank and authority scores.

Now let us **compare and contrast** the results of the two algorithms. Since PageRank doesn't really consider the notion of a "hub score," we'll mainly be analyzing PageRank and authority score.

Looking at the top 17 subreddits in PageRank and authority score, 13 of them appear in both. Additionally, the top results in PageRank and authority cover general areas. Some examples include "gaming", "science", "funny", and "videos". This makes sense, as niche topics have a hard time getting referenced by other subreddits by the nature of being "niche."

The significance comes from looking at which subreddits are in one list and not in the other. We see that **"leagueoflegends"** is ranked using the PageRank algorithm but not ranked among the top authority scores using the HITS algorithm. However, this makes sense due to subtle differences in the way PageRank and authority score is calculated.

A node under PageRank distributes its PageRank equally among all the nodes it points to. Thus, more total nodes leads to a *decrease in value* in each "vote" or arrow that a node has to another node. Note that this means PageRank may give an advantage to subreddit S that has lots of subreddits with all their nodes pointed to S and nowhere else.

However, the HITS algorithm calculates authority score a little differently. Each arrow is treated the same, and a node with a fixed hub score will transfer that entire hub score through every outward arrow. Thus, more outwards arrows just means a greater increase in authority score, which means each arrow has a *constant value.*

League of legends has a rather tight community because someone who knows nothing about their culture has no chance of enjoying game play or understanding discussions. This is different than sports, which people can begin to understand just by watching a couple plays. This means that the subreddits that link to “leagueoflegends” likely don’t link to much else, favoring PageRank because of the fact that more outward links diminishes the value of each link.

This subtle difference in implementation has a drastic influence on results, as “leagueoflegends” is ranked 9th in PageRank while not ranked using HITS. The PageRank algorithm thus has the potential to reveal large but rather niche communities, which is something important to consider when actually developing a search engine to match whatever marketing pitch or selling point.

All in all, the representation of the Reddit network using PageRank and HITS was rather accurate and it’s cool to see how behind all the results lie a pool of linear algebra concepts that can be learned in just around 3 months.

5 Bibliography

1. “Hits Algorithm.” *Wikipedia*, Wikimedia Foundation, 26 Aug. 2022, https://en.wikipedia.org/wiki/HITS_algorithm.

2. “PageRank.” *Wikipedia*, Wikimedia Foundation, 22 Nov. 2022, https://en.wikipedia.org/wiki/PageRank.

3. “Social Network: Reddit Hyperlink Network.” *SNAP*, Stanford, https://snap.stanford.edu/data/soc-RedditHyperlinks.html.

4. Strang, Gilbert. Introduction to Linear Algebra. Wellesley-Cambridge Press, 2021.

5. Raluca Tanase, Remus Radu. “Lecture #4.” *HITS Algorithm - Hubs and Authorities on the Internet*, http://pi.math.cornell.edu/ mec/Winter2009/RalucaRemus/Lecture4/lecture4.html.

6. “Reddit.” *Reddit*, https://www.reddit.com/.

7. Rousseau, Christiane. “How Google Works: Markov Chains and Eigenvalues.” *Klein Project Blog*, 7 Feb. 2017, http://blog.kleinproject.org/?p=280.

6 Appendix: Full code, output, and commentary

Importing packages:

```
[1]: using LinearAlgebra, NamedArrays, CSVFiles, JSON, DataFrames, IterativeSolvers
```

We start by generating a sample dataset of 4 websites and links for each website.

```
[24]: sources = ["twitter.com", "youtube.com", "facebook.com", "instagram.com"]
      links = [[2,3], [3], [1,2], [1,3,4]]
```

```
[24]: 4-element Vector{Vector{Int64}}:
       [2, 3]
       [3]
       [1, 2]
       [1, 3, 4]
```

Google's PageRank Algorithm

Markov Matrix

Now, we must create our markov matrix based on these websites and links. We can represent these websites and links as nodes and directed edges, respectively. This creates an matrix with each column representing each website, and each element representing the directed edge. It's value is $\frac{1}{\text{number of links}}$.

```
[25]: graph = zeros((length(links),length(links)))
      for i in range(1, length(links))
        for j in links[i]
          graph[j, i] = 1/length(links[i])
        end
      end
      graph
```

```
[25]: 4×4 Matrix{Float64}:
       0.0  0.0  0.5  0.333333
       0.5  0.0  0.5  0.0
       0.5  1.0  0.0  0.333333
       0.0  0.0  0.0  0.333333
```

We must now account for a slight modification (a *damping factor*) in order to ensure that we are able to produce a *positive* markov matrix and an attracting steady state vector; that is, a sole eigenvector with eigenvalue 1. Google's PageRank algorithm utilizes a damping factor of $d = 0.85$. We modify the graph with the formula as follows: $dP + (1 - d)Q$, where $Q_{ij} = \frac{1}{N}$ and $N = $ the number of websites.

```
[26]: d = 0.85
      graph = (d)*graph + (1-d)*fill(1/length(links),(length(links),length(links)))
```

```
[26]: 4×4 Matrix{Float64}:
       0.0375  0.0375  0.4625  0.320833
       0.4625  0.0375  0.4625  0.0375
       0.4625  0.8875  0.0375  0.320833
       0.0375  0.0375  0.0375  0.320833
```

Let's find the steady state vector. Per the PageRank algorithm, we must scale the vector so that it sums to 1.

[27]:
```
vecs = eigvecs(graph)
ss = abs.(vecs[:,length(links)])/sum(abs.(vecs[:,length(links)]))
```

[27]:
```
4-element Vector{Float64}:
 0.22721481386043674
 0.3089555283557729
 0.4115040763884415
 0.05232558139534881
```

Sorting and displaying the page ranks in a organized form:

[28]:
```
PR = NamedArray(hcat(sources,ss))
setnames!(PR, ["Source", "PageRank"], 2)
PR[sortperm(PR[:, 2], rev=true), :]
```

[28]:
```
4×2 Named Matrix{Any}
A  B           Source         PageRank

3       "facebook.com"         0.411504
2        "youtube.com"         0.308956
1        "twitter.com"         0.227215
4      "instagram.com"        0.0523256
```

HITS Algorithm

Singular Value Decomposition

We start by creating an adjacency matrix with $A_{ij} = 1$ if and only if i links to j.

[29]:
```
graph = zeros((length(links),length(links)))
for i in range(1, length(links))
  for j in links[i]
    graph[i, j] = 1
  end
end
graph
```

[29]:
```
4×4 Matrix{Float64}:
 0.0  1.0  1.0  0.0
 0.0  0.0  1.0  0.0
 1.0  1.0  0.0  0.0
 1.0  0.0  1.0  1.0
```

Now let's find the SVD of such adjacency matrix.

[30]:
```
U, , V = svd(graph)
```

[30]:
```
SVD{Float64, Float64, Matrix{Float64}, Vector{Float64}}
U factor:
4×4 Matrix{Float64}:
 -0.504959   0.226443   0.68456    0.474465
 -0.312082  -0.366393   0.423082  -0.7677
 -0.423082   0.7677    -0.312082  -0.366393
 -0.68456   -0.474465  -0.504959   0.226443
singular values:
4-element Vector{Float64}:
 2.1935270853310547
 1.2949628992915996
 1.193527085331054
 0.2949628992915991
Vt factor:
4×4 Matrix{Float64}:
 -0.504959  -0.423082  -0.68456   -0.312082
  0.226443   0.7677    -0.474465  -0.366393
 -0.68456    0.312082   0.504959  -0.423082
 -0.474465   0.366393  -0.226443   0.7677
```

The largest eigenvalue dominates over time, so we take $\vec{v}_1$ as the authority rank vector and $\vec{u}_1$ as the hub rank vector.

[31]:
```
v1 = abs.(V[:, 1])
```

[31]:
```
4-element Vector{Float64}:
 0.5049593141482909
 0.42308157087882825
 0.6845603616956413
 0.31208201907947963
```

[32]:
```
u1 = abs.(U[:, 1])
```

[32]:
```
4-element Vector{Float64}:
 0.5049593141482911
 0.3120820190794794
 0.4230815708788275
 0.6845603616956408
```

We can now construct and sort the ranks by authority (column 2 represents authority and column 3 represents hub)

[33]:
```
PR = NamedArray(hcat(sources,v1,u1))
setnames!(PR, ["Source", "Authority", "Hub"], 2)
PR[sortperm(PR[:, 2], rev=true), :]
```

[33]:
```
4×3 Named Matrix{Any}
A   B           Source        Authority              Hub

3       "facebook.com"          0.68456         0.423082
1        "twitter.com"         0.504959         0.504959
2        "youtube.com"         0.423082         0.312082
4      "instagram.com"         0.312082          0.68456
```

Sorting the ranks by hub:

[34]:
```
PR[sortperm(PR[:, 3], rev=true), :]
```

[34]:
```
4×3 Named Matrix{Any}
A   B           Source        Authority              Hub

4      "instagram.com"         0.312082          0.68456
1        "twitter.com"         0.504959         0.504959
3       "facebook.com"          0.68456         0.423082
```

```
 2       "youtube.com"        0.423082        0.312082
```

Reddit Network

As a website network is simply too large to be ran here, a more practical application of PageRank is a network on Reddit, where we look at the hyperlinks inside the bodies of reddit posts in particular subreddits. We look at data from Jan 2014 to April 2017. The dataset can be found here: http://snap.stanford.edu/data/soc-RedditHyperlinks.html

We start by loading the data from the tsv file as a dataframe.

[14]:
```
using CSVFiles, DataFrames
df = DataFrame(load("reddit.tsv"))
```

[14]:

	SOURCE_SUBREDDIT	TARGET_SUBREDDIT	POST_ID	TIMESTAMP	LINK_SENTIMENT	
	String	String	String	DateTime	Int64	
1	leagueoflegends	teamredditteams	1u4nrps	2013-12-31T16:39:58	1	...
2	theredlion	soccer	1u4qkd	2013-12-31T18:18:37	-1	...
3	inlandempire	bikela	1u4qlzs	2014-01-01T14:54:35	1	...
4	nfl	cfb	1u4sjvs	2013-12-31T17:37:55	1	...
5	playmygame	gamedev	1u4w5ss	2014-01-01T02:51:13	1	...
6	dogemarket	dogecoin	1u4w7bs	2013-12-31T18:35:44	1	...
7	locationbot	legaladvice	1u4wfes	2014-01-07T20:17:41	1	...
8	indiefied	aww	1u50pos	2014-03-03T17:00:35	1	...
9	posthardcore	bestof2013	1u5ccus	2013-12-31T23:16:20	1	...
10	posthardcore	corejerk	1u5ccus	2013-12-31T23:16:20	1	...
11	gfycat	india	1u5df2s	2013-12-31T22:27:50	1	...
12	metalcore	bestof2013	1u5iets	2014-01-01T04:15:54	1	...
13	metalcore	corejerk	1u5iets	2014-01-01T04:15:54	1	...
14	suicidewatch	offmychest	1u5k33s	2014-01-01T03:01:53	1	...
15	dogecoin	novacoin	1u5olgs	2014-01-01T05:58:10	1	...
16	gaming4gamers	fallout	1u5q84s	2014-01-01T06:55:04	1	...
17	kpop	dota2	1u5qg2s	2014-01-01T07:05:10	1	...
18	airsoft	airsoftmarket	1u5r7js	2014-01-01T07:09:16	1	...
19	circlebroke	childfree	1u5rs9s	2014-01-01T06:51:30	1	...
20	tribes	games	1u5syks	2014-01-01T09:06:30	1	...
21	oldschoolcoolnsfw	pics	1u5t29s	2014-01-02T15:13:58	1	...
22	fl_vapers	vaperequests	1u5uchs	2014-01-03T07:21:14	1	...
23	jailbreak	flextweak	1u5vxys	2014-01-01T08:33:33	1	...
24	corejerk	bestof2013	1u5w42s	2014-01-01T04:58:04	1	...
25	iama	todayilearned	1u5yqfs	2014-01-01T09:09:40	1	...
26	bandnames	books	1u62xgs	2014-01-01T10:14:40	1	...
27	thedoctorstravels	hungergamesrp	1u638cs	2014-01-01T10:08:48	1	...
28	politicaldiscussion	todayilearned	1u64d9s	2014-01-01T12:50:53	1	...
29	uncomfortableqs	debatereligion	1u64jfs	2014-01-24T15:07:06	1	...
30	connecticut	ctbeer	1u64lgs	2014-01-01T09:59:45	1	...
...	...	...	...	...	...	

Next, we create the sources vector as a hash map mapped from subreddit to its index. This allows for easy access later to create our links.

[15]:
```
d = Dict()
sources = String[]
i = 1
for row in eachrow(df)
  if !haskey(d, row.SOURCE_SUBREDDIT)
```

```
    d[row.SOURCE_SUBREDDIT] = i
    i = i + 1
    push!(sources, row.SOURCE_SUBREDDIT)
  end
end
links = []
l = Dict()
for row in eachrow(df)
  if !haskey(d, row.TARGET_SUBREDDIT)
    d[row.TARGET_SUBREDDIT] = i
    i = i + 1
    push!(sources, row.TARGET_SUBREDDIT)
  end
end
```

Next, we create the links array, where each index corresponds to a subreddit and its links to other subreddit indexes. We implement the links of each subreddit as a hash set to prevent any duplicates from occurring. The order of the links does not matter, so this still allows for fast iteration and insertion.

[16]:
```
for row in eachrow(df)
  if !haskey(l, row.SOURCE_SUBREDDIT)
    push!(links, Set())
    l[row.SOURCE_SUBREDDIT] = 1
  end
  push!(links[d[row.SOURCE_SUBREDDIT]], d[row.TARGET_SUBREDDIT])
end
for i in range(length(links)+ 1, size(sources)[1])
  push!(links, Set())
end
```

Now we must create our Markov matrix and account for a damping factor by first iterating through links and adding the links to the appropriate cells ((j,i) if i links to j), and then iterate once again for each element of the matrix to apply the damping factor formula. While this is quite slow and a solution that adds the damping factor to all elements at once is faster, it uses too much memory for the comupter to handle.

[17]:
```
graph = zeros(length(links),length(links))
for i in range(1, length(links))
  for j in links[i]
    graph[j, i] = 1/length(links[i])
  end
end
d = 0.85
for j = 1:size(graph,2)
  for i = 1:size(graph,1)
      graph[i,j] = d*graph[i,j] + (1-d)*(1/length(links))
  end
end
graph
```

[17]:
```
35776×35776 Matrix{Float64}:
 4.19275e-6  4.19275e-6  4.19275e-6  ...  4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 0.00332451  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6  …  4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6  …  4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 0.00332451  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6

 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6  …  4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6  …  4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6     4.19275e-6  4.19275e-6  4.19275e-6
 4.19275e-6  4.19275e-6  4.19275e-6  …  4.19275e-6  4.19275e-6  4.19275e-6
```

We then apply the power method with a balanced starting vector (assuming all subreddits have the same influence as an original estimate) to find the dominant eigenvalue and eigenvector. The eigenvector must sum to 1, so we scale the steady state vector as such.

```
[18]: ss = vec(fill(1/length(links), (length(links),1)))
      powm!(graph, ss)
      ss = ss/sum(ss)
```

```
[18]: 35776-element Vector{Float64}:
       0.003793573721956673
       7.042346225625251e-6
       2.0316603152421813e-5
       0.0016126283772358498
       0.00015113802866155768
       0.000507849770218283
       4.715301657625223e-6
       4.715301657625223e-6
       0.00014397729224427874
       6.554354380528157e-5
       0.00028987663293599306
       0.0003317549368681413
       0.0015235940888843394
       ⋮
       6.492951695642393e-6
       5.093565199272388e-6
       1.69853077505414e-5
       9.22297286089656e-6
       9.988424976734344e-6
       5.0159928607763886e-6
       2.1946796346132804e-5
       4.981899135660787e-6
```

```
 4.87512466553193e-6
 1.3929494791052338e-5
 9.222972860896559e-6
 6.969137259260894e-6
```

Note that the dominant eigenvalue is not quite equivalent to 1, but its a close-enough approximation to provide us with some useful results. We now create a named array and sort it to find the most influential subreddits.

[19]:
```
PR = NamedArray(hcat(sources,ss))
setnames!(PR, ["Source", "PageRank"], 2)
PR[sortperm(PR[:, 2], rev=true), :][1:20, 1:2]
```

[19]:
```
20×2 Named Matrix{Any}
A   B               Source              PageRank

35              "askreddit"            0.0121459
23                   "iama"            0.0117198
7742                 "pics"           0.00586916
11295              "videos"           0.00563957
127            "outoftheloop"         0.00402394
27866         "todayilearned"         0.00391077
5297                 "mhoc"            0.0038693
563                "gaming"           0.00380185
1           "leagueoflegends"         0.00379357
43           "writingprompts"         0.00374519
738                 "funny"           0.00364018
1386            "worldnews"           0.00340026
1257                 "news"           0.00304047
258           "pcmasterrace"          0.00300983
6495           "technology"           0.00282534
89        "explainlikeimfive"         0.00275315
1828              "science"           0.00265422
491                 "games"           0.00253972
140                "movies"           0.00241172
17463          "bestof2015"           0.00237747
```

We see that AskReddit and iAMA are the most influential subreddits from Jan 2014 to April 2017, by a long shot. This makes sense intuitively as AskReddit provides useful information for a large portion of the reddit population and iAMA hosts celebrities on its page.

Now let's test the reddit network on the HITS algorithm. We first create an adjacency matrix where if the element at row i and column j is equal to 1, then there exists a directed edge from subreddit i to j. We can do this by iterating through links.

[20]:
```
graph = zeros((length(links),length(links)))
for i in range(1, length(links))
  for j in links[i]
    graph[i, j] = 1
  end
end
```

Next we find the eigenvectors v_1 and u_1 by applying the power method on $A^T A$ and AA^T respectively.

```
[21]: v1 = vec(fill(1/length(links), (length(links),1)))
      powm!(graph'*graph, v1)
      u1 = vec(fill(1/length(links), (length(links),1)))
      powm!(graph*graph', u1)
```

```
[21]: (8877.8311828487, [0.05411352866169992, 0.010565961512690703,
      0.0007499341525331654, 0.015256165299901396, 0.007882067335624753,
      0.005816260837258981, 0.0009738135297897343, 0.0039045897424879613,
      0.0009554711055651921, 0.0053228273784484866  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
      0.0, 0.0, 0.0, 0.0])
```

We can then organize the data by concatenating our 3 vectors horizontally and sorting by authority.

```
[22]: PR = NamedArray(hcat(sources,v1,u1))
      setnames!(PR, ["Source", "Authority", "Hub"], 2)
      show(IOContext(stdout, :limit=>false), PR[sortperm(PR[:, 2], rev=true), :][1:20, 1:3])
```

```
20×3 Named Matrix{Any}
A   B                  Source           Authority              Hub

35              "askreddit"              0.286246          0.107565
23                   "iama"              0.225876         0.0923254
7742                 "pics"              0.181623       0.000563776
11295              "videos"              0.173853       0.000201055
27866        "todayilearned"             0.172722               0.0
738                 "funny"              0.155721         0.0259283
1386            "worldnews"              0.153163        0.00320554
1257                 "news"              0.142677        0.00810208
27868        "adviceanimals"             0.118612               0.0
563                "gaming"              0.118028         0.0446724
89       "explainlikeimfive"             0.117069         0.0828201
1828              "science"              0.112375        0.00659387
127           "outoftheloop"             0.110032          0.133766
6495            "technology"             0.108211         0.0197608
27870                 "wtf"               0.10584               0.0
269        "showerthoughts"              0.104325          0.073693
3594              "politics"             0.102769          0.015815
7461                  "gifs"             0.102288       3.19847e-5
71         "subredditdrama"              0.101395          0.208824
145                   "tifu"            0.0960996         0.0508054
```

As well as sorting by hub:

```
[23]: PR = NamedArray(hcat(sources,v1,u1))
      setnames!(PR, ["Source", "Authority", "Hub"], 2)
      show(IOContext(stdout, :limit=>false), PR[sortperm(PR[:, 3], rev=true), :][1:20, 1:3])
```

```
20×3 Named Matrix{Any}
A   B                     Source             Authority                Hub

71            "subredditdrama"                0.101395           0.208824
80                 "copypasta"               0.0127593           0.140558
127             "outoftheloop"                0.110032           0.133766
17                "circlebroke"              0.0295397            0.13034
32       "circlejerkcopypasta"               0.0100197           0.128305
```

```
627                  "drama"            0.0316244            0.127558
6401        "shitliberalssay"           0.0106842            0.122778
9245           "justunsubbed"          0.00955009            0.117095
131              "conspiracy"           0.0739141            0.115691
46            "hailcorporate"           0.0278471            0.113726
327                    "self"           0.0588681            0.110519
35                "askreddit"            0.286246            0.107565
344       "nostupidquestions"           0.0562355            0.107181
7375   "bestofoutrageculture"           0.0134311            0.102165
163              "karmacourt"           0.0116986           0.0986079
264            "circlebroke2"           0.0194316           0.0950776
43           "writingprompts"           0.0757081           0.0938505
190          "shitredditsays"            0.039926           0.0934937
184                    "help"           0.0250544           0.0930448
23                     "iama"            0.225876           0.0923254
```