Exploring Google Page Recommendation with PageRank, Hyperlink-Induced Topic Search, and Personalized PageRank

Exploration the linear algebra concepts behind the PageRank and HITS algorithm, their implementation in Python, and further extensions involved Personalized PageRank

Michael Liu, Armaan Sidhu

I. Introduction

Introduction to PageRank

PageRank is an algorithm developed by Larry Page and Sergey Brin of Google used to rank the importance of website pages. The fundamental idea of PageRank stems from the idea of representing a network of websites as nodes of a directed graph and the edges to be links connecting such nodes together. The importance of a website is determined by the number of websites that link to each particular website and the importance of such websites, and the PageRank algorithm determines the probability distribution that an arbitrary surfer will reach each website after a series of random clicks. [3]

Introduction to HITS

The Hyperlink-Induced Topic Search HITS Algorithm was developed by Jon Kleinberg and is an another link ranking algorithm that is also widely used. The way that this algorithm works is that it computes two values for each page: the hub and the authority value. The higher that the hub value for a page is, the more important pages it links to, and the higher the authority value is, the more important pages link to it. To begin, the algorithm is run on a root set, which is the set of most relevant pages to a search query. Then, this can potentially lead to a larger base set, and the base set is formed by augmenting the root set with the web pages that are linked to and from those pages. From here, HITS is used to rank such websites based on such two key values: hub and authority. [2]

II. Linear Algebra Background and Applications

The fundamental idea of PageRank relies of the idea of probability vectors, Markov matrices, Markov chains, and steady state vectors. We start by exploring such ideas.

Foremost, we describe a vector $\mathbf{x} \in \mathbb{R}^n$ to be a probability vector if all entries of $\mathbf{x}, x_i, ..., x_n$ are nonnegative and sum to 1. Furthermore, we describe a matrix $M \in M_{n\times n}(\mathbb{R})$ to be a Markov matrix if all entries are non negative and each column sums to 1. A Markov matrix is used to describe the transition between states based on varying probabilities. We let entry $M_{i,j}$ represent the probability to transition from state i to state j. From here, we define a Markov chain to be a sequence $\mathbf{x}_0, ..., \mathbf{x}_k = M^k\mathbf{x}_0, ...$ given a Markov matrix M and an initial probability vector $\mathbf{x}_0 \in \mathbb{R}^n$. We know that each vector $\mathbf{x}_k$ is a probability vector.

When discussing PageRank, we consider it important to determine the probability vector at which a Markov chain will converge. Such a probability vector is called a steady state vector. Formally, we define $\mathbf{x} \in \mathbb{R}^n$ to be a steady state vector such that $M\mathbf{x} = \mathbf{x}$ where M is a Markov matrix. Importantly, by the Perron-Frobenius Theorem, we know that all positive Markov matrices (meaning Markov matrices with all positive entries) will have a unique steady state vector. Hence, we know that a regular Markov matrix (meaning at some power M^k, M is a positive Markov matrix) must have a unique steady state vector. Usually, we would determine such steady state vector by directly solving for the eigenvector of M associated with eigenvalue of 1, yet when considering PageRank, we will explore an iterative approach in the following sections.

Such ideas have a plethora of practical applications. For example, Markov chains can be used to offer a rough prediction of stocks and inform trading strategies where the states represent if a stock price will be up, down, or stable and the Markov matrix describes the probabilities of transitioning from one state to another. In this project, we will focus on its use in PageRank where each website will be represented as a state and the Markov matrix is the probability of transitioning from one state to another based on links. We seek to determine the steady state vector which represents the long run probability of being on each page. Hence, a page with a higher probability in the steady state vector will receive a higher PageRank score.

Though we discuss these fundamental concepts, note that the PageRank algorithm differs from theory due to the existence of "dangling pages" (where there are no outlinks, hence making our Markov matrix non positive), randomness, and other considerations in practical use. We explore such caveats below.

III. PageRank Algorithm

Now, we discuss the details of the PageRank algorithm. Fundamentally, we set the PageRank initially to be a uniform distributed probability vector in which we assume that a random surfer will have an equally likely probability to be on any page. Then, we find the steady state vector of the Google PageRank Markov matrix which represents the probabilities to transition from one page to another based on how many outbound links each page has and a dampening factor(which we will explore in more detail later). By iteratively multiplying this matrix by the PageRank vector, we can determine a vector at which the PageRank vector will converge, hence being the steady state vector representing the PageRank scores for each page. We formalize this process as follows.

Initialization

Let there be n pages. Then we let $PR(p_i)$ represent the PageRank for an arbitrary page p_i such that $1 \leq i \leq n$. Furthermore, we know that $PR(p_i)$ is between 0 and 1. For initialization, we assume that a random surfer will have an equal chance of landing on any page, hence we uniformly distribute the PageRank for each page as

$$PR(p_i) = \frac{1}{n} \quad \text{s.t.} \quad \sum_{i=1}^{n} PR(p_i) = 1$$

Iterative Step

After initialization, we seek to find the PageRank by iteratively updating for some number of steps until convergence. At each step, we let the Pagerank of a page p_i be determined by the PageRank of the pages that link to it as well as the number of links from each of such links. In this way, if a page links to many pages including page p_i, it will be deemed a less important factor to the PageRank of p_i than a page that links to less pages. Therefore, we formalize this by letting the $B(p_i)$ be the set of all pages that link to a page p_i and letting $L(v)$ be the number of links from page v. Then, we let

$$PR(p_i) = \sum_{v \in B(p_i)} \frac{PR(v)}{L(v)}$$

Dampening Factor

As explained in part II, the PageRank algorithm considers various considerations for practical use. Two of such considerations is if the random surfer will stop clicking links or jump to a random page. Hence, we introduce the dampening factor which is the probability that a random surfer will continue clicking links. We set the dampening factor d to be around 0.85 in practice according to research. Hence, we adjust the PageRank formula to be

$$PR(p_i) = \frac{1-d}{n} + d \sum_{v \in B(p_i)} \frac{PR(v)}{L(v)}$$

Note that $d \sum_{v \in B(p_i)} \frac{PR(v)}{L(v)}$ scales the contributions of each page by a factor of d, causing the sum of the PageRank of all our pages to no longer be a probability vector (as it will no longer be a sum of 1 as required). Hence, we add the remaining factor to be $\frac{1-d}{n}$ which gives each page an equally distributed possibility to be clicked, hence the randomness factor.

Putting It Together

From here, we represent this in terms of matrices and vectors. We let $\mathbf{R}$ be the PageRank vector where

$$\mathbf{R} = \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_n) \end{bmatrix} \text{where } \mathbf{R} = f + dL\mathbf{R},$$

such that

$$f = \begin{bmatrix} \frac{1-d}{n} \\ \frac{1-d}{n} \\ \vdots \\ \frac{1-d}{n} \end{bmatrix} \text{ and } L = \begin{bmatrix} \ell(p_1, p_1) & \cdots & \ell(p_1, p_n) \\ \ell(p_2, p_1) & \cdots & \ell(p_2, p_n) \\ \vdots & \vdots & \vdots \\ \ell(p_n, p_1) & \cdots & \ell(p_n, p_n) \end{bmatrix}$$

where

$$\ell(p_i, p_j) = \begin{cases} \frac{l}{o}, & \text{if page } p_j \text{ links to } p_i, \\ 0, & \text{if page } p_j \text{ does not link to } p_i, \end{cases}$$

where l is the number of links from page p_j to page p_i and o is the number of outbound links from page p_j. Note that we generally treat there to be 1 link from p_j to p_i, hence $l = 1$. Furthermore, every column of L must sum to 1. However, it is easily noticed that in the case that there is a dangling node (where the page has no outgoing links) that it will not. Therefore, in practice, we implement a technique where if a page is a dangling node and hence the column of that page in L is all zeros, we will simply replace it with a uniformly distributed column of $\frac{1}{n}$.

Now we will briefly discuss why this guarantees convergence. Note that $\mathbf{R} = f + dL\mathbf{R}$ is equivalent to $\mathbf{R} = G\mathbf{R}$ where G is the Google matrix where $G = (1-d)L + dB$ such that L is the link matrix after replacing dangling columns (where zero columns correspond to page that have no outlinks) and $B = \frac{1}{n}\mathbf{1}\mathbf{1}^\top$ (hence representing the matrix of dampening factors). This definition of the Google matrix is that provided in lecture notes.

We can determine that they are equal intuitively by analyzing what $G\mathbf{R}$ represents. We know that $G\mathbf{R} = (1-d)L\mathbf{R} + dB\mathbf{R}$. From here, the term $(1-d)L\mathbf{R}$ represents the contribution from the probability of following each link regularly based on the PageRank of all the pages that link to a page. By multiplying by $1-d$, we are scaling this contribution by the weight that the random surfer will actually follow a link instead of going to a random link. From here, we know that $dB\mathbf{R}$ accounts for the contribution to the PageRank that the random surfer will go to a random page. This is because $B\mathbf{R} = \frac{1}{n}\mathbf{1}\mathbf{1}^\top\mathbf{R} = \frac{1}{n}$ because $\mathbf{R}$ is a probability vector and hence all entries sum to 1. Hence, $dB\mathbf{R} = d\frac{1}{n}$ as required.

Therefore, because G must be a positive Markov matrix because each entry must be greater than or equal to $\frac{1}{n}$, by our discussion of steady state vectors in part II, we know that we must have a unique steady state vector $\mathbf{R}$.

Computation

In this project, we will apply PageRank in an iterative approach. We iteratively compute $\mathbf{R}_{i+1} = f + dL\mathbf{R}_i$ as described above until $|\mathbf{R}_{i+1} - \mathbf{R}_i| < \epsilon$ for some $\epsilon \in \mathbb{R}$. Here, we would deem the vector $\mathbf{R}_{i+1}$ to be close enough to the steady state vector, and we would hence let it represent our PageRank vector. [6]

IV. HITS Algorithm

Initialization

First, let's discuss how the authority and hub values are obtained for a website. These values are defined recursively in terms of each other as follows: the hub value is obtained by using the authority value of the pages that point from that page, and the authority value is obtained by using the hub values of the pages that point to that page. This algorithm is run iteratively a large number of times and each time, it will first update the authority value using the hub values that link to it and then update the hub value with the authority values of the pages it links to. After repeating this recursive process, the values are all normalized on each iteration and eventually converge.

Suppose our network contains n pages. We initialize the hub and authority vectors by setting all values to 1. Let a denote the authority vector and h denote the hub vector. Hence, we initialize

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad \text{and} \quad h = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Iterative Step

We update each page p_i's hub score by summing the authority scores of all pages that p_i links to. This is the same as just multiplying the adjacency matrix of outbound links with the authority vector as follows.

$$\text{hub}(p_i) = \sum_{q \in S(p_i)} \text{auth}(q) \quad \text{where } S(p_i) \text{ is the set of pages that } p_i \text{ links to}$$

Let L represent the adjacency matrix of outbound links:

$$L = \begin{bmatrix} \ell(p_1, p_1) & \cdots & \ell(p_1, p_n) \\ \vdots & \ddots & \vdots \\ \ell(p_n, p_1) & \cdots & \ell(p_n, p_n) \end{bmatrix} \text{ such that } \quad \ell(i,j) = \begin{cases} 1 & \text{if } i \text{ has an outgoing link to } j \\ 0 & \text{if } i \text{ does not link to } j \end{cases}$$

We then update the initial hub vector h_i to h_n where

$$h_n = La_i$$

At each iteration, we normalize the updated hub vector to keep it a probability vector as seen in lecture by doing $h_n = \frac{h_n}{||h_n||}$.

Now, we consider the authority scores. As stated before, we do this through our recursive definition depending on the hub scores. Thus, we update each page p_i's authority score by summing the hub scores of all pages pointing to p_i. This can be done by multiplying the adjacency matrix of inbound links with the hub vector as follows.

$$\text{auth}(p_i) = \sum_{q \in S(p_i)} \text{hub}(q) \quad \text{where } S(p_i) \text{ is the set of pages linking to } p_i$$

We can simply use L^T as the adjacency matrix of incoming links (since transposing the outbound links matrix gives the inbound links matrix). Updating the initial authority vector $\vec{a}_i$ to a_n yields:

$$a_n = L^T h_i$$

We then normalize the updated authority vector to keep it a probability vector as seen in lecture by doing $a_n = \frac{a_n}{||a_n||}$.

For some authority vector a and hub vector h, if we first update the hub vector to $h = La$, our subsequent step updates the authority vector to:

$$a = L^T h = L^T La$$

Since this process is mutually recursive, updating the hub vector yields:

$$h = La = LL^T h$$

Therefore, we have established that $a_{i+1} = L^T La_i$ and $h_{i+1} = LL^T h_i$ for any authority vector a_i and hub vector h_i. We know that $L^T L$ and LL^T must converge as they are both symmetric matrices and they are both positive semi-definite. [5]

Computation

We apply HITS in an iterative approach similar to PageRank. We iteratively compute $a_{i+1} = L^T La_i$ and $h_{i+1} = LL^T h_i$ as described above until $|a_{i+1} - a_i| < \epsilon \wedge |h_{i+1} - h_i| < \epsilon$ for some $\epsilon \in \mathbb{R}$. Hence, we would deem a_{i+1} to be an acceptable authority ranking.

V. PageRank Python Implementation

The following PageRank and HITS implementation code in Python can be accessed with this link to our Google Colab notebook. We use the NumPy library to perform vector and matrix calculations and Matplotlib to visualize iterations and convergence. In this section, we explore our implementation in detail.

We define a kind of adjacency matrix representation for the pages in the network. We have n pages in our network and we represent this in a list of lists where the ith index of the list is a list of all the other j pages that page i has an outgoing link to. For example we let this array `links` represent a network with the respective outlinks where `n` is the number of pages.

```
links = [
    [1, 2, 3, 4], #p0 -> p1, p2, p3, p4
    [2, 4],       #p1 -> p2, p4
    [3],          #p2 -> p3
    [],           #p3 -> none (dangling)
    [2, 0, 1],    #p4 -> p2, p0, p1
]
n = len(links)
```

Intuitively, we expect the importance of `p0` to be low because only `p4` links to it. We would expect `p2` to be relatively important because it has many incoming links. Importantly, `p3` would also be very important because it is linked by an important page `p2` where `p2` only links to it (making it very important) as well as being linked by `p0` (though this contribution would be lower).

As explained earlier, we initialize the PageRank to be evenly distributed among each page. Hence, we define the function `initial_pagerank` which takes in this list of links and returns a Numpy array (or vertex) of evenly distributed probabilities.

```
def initial_pagerank(links):
  entry = 1/n
  R = np.full(n, entry, dtype = float)
  return R
```

From here, we define the iterative step by directly translating the equation $\mathbf{R} = f + dL\mathbf{R}$. Hence,

```
def iterate_pagerank(R, L, d):
  f = np.full(n, (1-d)/n, dtype = float)
  Ri = f + d * np.dot(L, R)
  return Ri
```

However, we need to define a method to compute `L` which is the linking matrix as defined in section III. Hence, we define the function

```
def compute_link_mat(links):
  L = np.zeros((n, n), dtype = float)
  for i in range(n):
    for j in range(n):
      if len(links[j]) == 0:
        L[i][j] = 1/n
      elif i in links[j]:
        o = len(links[j])
        L[i][j] = 1/o
  return L
```

In this function, we create a new n by n matrix and iterate through every entry. By our equation

$$\ell(p_i, p_j) = \begin{cases} \frac{l}{o}, & \text{if page } p_j \text{ links to } p_i, \\ 0, & \text{if page } p_j \text{ does not link to } p_i, \end{cases}$$

from earlier, we see if page i is in the list of outlinks for page j by checking if it is an element of the list `links`[j]. If it is, then we know that page j links to page i, and hence, we populate it with $1/o$ where o is the number out outlinks of j, hence the length of `links`[j] as required (note we established earlier that we define l to be 1). Otherwise, it will just be the default value 0. Note, we also implement the fix with dangling nodes as described above.

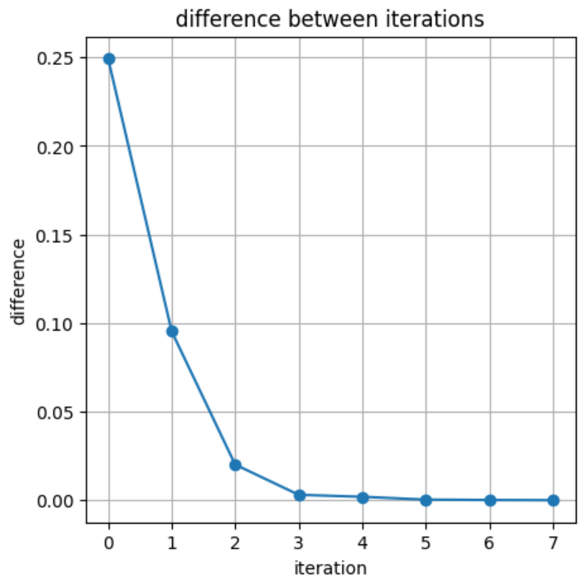
Now, we put it all together with the function `pagerank`. Note that the function in the Google colab notebook is slightly different as it includes code for tracing and plotting, but we simplify it here.

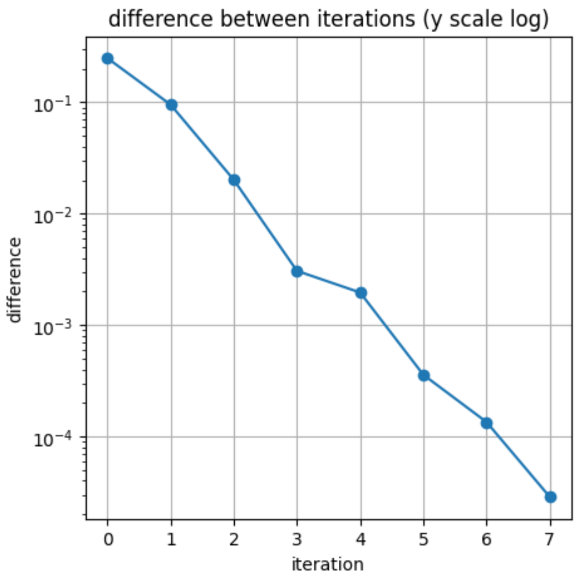
```
def pagerank(links, d = 0.85, epsilon = 0.0001):
  R0 = initial_pagerank(links)
  L = compute_link_mat(links)
  R1 = iterate_pagerank(R0, L, d)

  itr = 1
  while(np.sum(abs(R1 - R0)) > epsilon):
    R0 = R1
    R1 = iterate_pagerank(R0, L, d)
    itr = itr + 1

  print("iterations: " + str(itr))
  return R1
```

Here, we simply compute the initial PageRank `R0` with `initial_pagerank` function, compute the linking matrix `L` with out `compute_link_mat` function, and the first iteration `R1` by iterating. From here, we iteratively update `R1` until the absolute difference is less than a desired `epsilon` which we define as `0.0001`.Now we analyze and discuss our results. By fully tracing, we get

```
initial pagerank: [0.2 0.2 0.2 0.2 0.2]
iteration 1: [0.12066667 0.16316667 0.24816667 0.2765     0.1915    ] diff = 0.24933
iteration 2: [0.13126333 0.156905   0.22625083 0.31358833 0.1719925 ] diff = 0.09537
iteration 3: [0.13204123 0.15993468 0.22661931 0.30351668 0.1778881 ] diff = 0.02014
iteration 4: [0.13199946 0.16005822 0.22803047 0.30228301 0.17762884] diff = 0.00307
iteration 5: [0.13171628 0.15976617 0.22779091 0.30326389 0.17746274] diff = 0.00196
iteration 6: [0.13183597 0.15982568 0.2277263  0.30316685 0.17744519] diff = 0.00036
iteration 7: [0.1318145  0.15982965 0.22775556 0.30312087 0.17747942] diff = 0.00013
iteration 8: [0.13181638 0.15982697 0.22775457 0.30313336 0.17746873] diff = 3e-05
total iterations: 8
```

Hence, our final PageRank indeed confirms out initial hypothesis. `p0` had the lowest PageRank while `p3` had the highest followed by `p2`. Furthermore, we can notice that convergence is determined by the difference in PageRank between iterations. It is evident that such difference is great at first but swiftly tapers down until convergence is determined (in which such difference is lower than `epsilon`). In this way, we can conclude that PageRank convergence is generally logarithmic. By using Matplotlib, we can plot this and get:





VI. HITS Python Implementation

Now, we implement the HITS algorithm following the same mock network we established in the previous section about PageRank. We represent our test network as a list of lists, where each index i contains the pages that page i links to as follows (it is the same network so we can compare in a later section)

```
links = [
    [1, 2, 3, 4],  # p0 -> p1, p2, p3, p4
    [2, 4],        # p1 -> p2, p4
    [3],           # p2 -> p3
    [],            # p3 -> none (dangling)
    [2, 0, 1],     # p4 -> p2, p0, p1
]
n = len(links)
```

This representation allows us to easily construct the adjacency matrix and analyze link patterns. Intuitively, we expect different results from HITS compared to PageRank because the main difference in the algorithms lies in the way that HITS scores websites based on two metrics, namely the hub and authority scores. For instance, p_0 links to many pages, so it may have a high hub score, while p_3 receives links from important pages like p_2, so it may have a high authority score even though it is a dangling node.

First, we construct the initial authority and hub vectors. Hence, we define the function as follows

```python
def initial_hits(links):
  auth = np.full(n, 1, dtype = float)
  hub = np.full(n, 1, dtype = float)
  return auth, hub
```

Next, we construct the outbound link adjacency matrix using the function `compute_link_mat_hits` using our linking matrix representation we established previously.

```python
def compute_link_mat_hits(links):
  L = np.zeros((n, n), dtype = float)
  for i in range(n):
    for j in range(n):
      if j in links[i]:
        L[i][j] = 1
  return L
```

Running `build_link_matrix(links)` on our example network produces:

$$L = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Note that row i represents the outgoing links from page i. For example, row 0 has ones in columns 1, 2, 3, and 4, indicating that page 0 links to pages 1, 2, 3, and 4. Row 3 is all zeros, confirming that page 3 is a dangling node with no outbound links. Then, as stated earlier, the transpose L^T gives us the incoming link matrix, where column j shows which pages link to page j:

$$L^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

We now implement the iterative HITS algorithm. The function takes the network as input and returns both authority and hub vectors after convergence.

```python
def hits(links, epsilon=1e-6):
    L = compute_link_mat_hits(links)

    A0, H0 = initial_hits(links)
    A1, H1 = iterate_hits(A0, H0, L)
    authority_diff = np.linalg.norm(A1 - A0)
    hubs_diff = np.linalg.norm(H1 - H0)

    itr = 1
    while(authority_diff > epsilon or hubs_diff > epsilon):
      itr += 1

      A0 = A1
      H0 = H1
      A1, H1 = iterate_hits(A0, H0, L)

      authority_diff = np.linalg.norm(A1 - A0)
      hubs_diff = np.linalg.norm(H1 - H0)

    return A1, H1
```

By calling $\texttt{auth}, \texttt{hub} = \texttt{hits}(\texttt{links})$ and tracing, we get

```
iteration 1:
   auth =[0.21320072 0.42640143 0.63960215 0.42640143 0.42640143]diff auth =1.31756809
   hub =[0.7448453  0.41380294 0.16552118 0.         0.49656353]diff hub =1.53575196
iteration 2:
   auth =[0.19185884 0.47964711 0.63952948 0.35174121 0.44767064]diff auth =0.09652518
   hub =[0.74091015 0.41984908 0.13583353 0.         0.5062886 ]diff hub =0.03206205
iteration 3:
   auth =[0.19549538 0.48158618 0.6437043  0.33854078 0.44820892]diff auth =0.01445527
   hub =[0.73828583 0.4216146  0.13071894 0.         0.50998797]diff hub =0.00706034
iteration 4:
   auth =[0.1969177  0.48198627 0.64478105 0.33554207 0.44786335]diff auth =0.00352904
   hub =[0.737559   0.42189363 0.12956005 0.         0.5111034 ]diff hub =0.001787
iteration 5:
   auth =[0.19734799 0.48213535 0.64503753 0.33481327 0.44768954]diff auth =0.00091352
   hub =[0.73736669 0.42192533 0.12927858 0.         0.51142588]diff hub =0.00047032
iteration 6:
   auth =[0.19747248 0.48218554 0.64509995 0.33463029 0.44762747]diff auth =0.00024341
   hub =[0.73731552 0.42192544 0.12920792 0.         0.51151741]diff hub =0.00012644
iteration 7:
   auth =[0.19750782 0.48220112 0.64511557 0.33458325 0.44760775]diff auth =6.586e-05
   hub =[0.73730179 0.42192386 0.12918975 0.         0.5115431 ]diff hub =3.437e-05
iteration 8:
   auth =[0.19751774 0.48220574 0.64511957 0.33457093 0.44760183]diff auth =1.796e-05
   hub =[0.73729807 0.42192312 0.129185   0.         0.51155026]diff hub =9.39e-06
iteration 9:
   auth =[0.1975205  0.48220707 0.64512062 0.33456766 0.44760011]diff auth =4.91e-06
   hub =[0.73729707 0.42192286 0.12918373 0.         0.51155224]diff hub =2.57e-06
iteration 10:
   auth =[0.19752127 0.48220745 0.64512089 0.33456678 0.44759963]diff auth =1.35e-06
   hub =[0.73729679 0.42192278 0.1291834  0.         0.51155279]diff hub =7.1e-07
iteration 11:
   auth =[0.19752148 0.48220755 0.64512097 0.33456655 0.44759949]diff auth =3.7e-07
   hub =[0.73729672 0.42192276 0.1291833  0.         0.51155294]diff hub =1.9e-07
```

To produce final rankings from the HITS scores, we sort pages by their authority and hub values separately.

```
def determine_hits_order(auth, hub):
  auth_order = np.argsort(-auth)
  hub_order = np.argsort(-hub)
  for i in range(n):
    print("rank " + str(i + 1) + ": page " + str(auth_order[i]))
  return auth_order, hub_order

auth_order, hub_order = determine_hits_order(auth, hub)
print("auth order: " + str(auth_order))
print("hub order: " + str(hub_order))
```

Thus, we get

```
rank 1: page 2
rank 2: page 1
rank 3: page 4
rank 4: page 3
rank 5: page 0
auth order: [2 1 4 3 0]
hub order: [0 4 1 2 3]
```

Comparison with PageRank

Comparing HITS results to the PageRank scores from Section IV reveals interesting differences. Recall that PageRank yielded final scores:

```
PageRank: [0.13181638, 0.15982697, 0.22775457, 0.30313336, 0.17746873]
```

We can This gives us the PageRank ranking: $p_3 > p_2 > p_4 > p_1 > p_0$.

In contrast, HITS provides two separate rankings:

- **HITS authority ranking:** $p_2 > p_1 \approx p_3 > p_0 > p_4$
- **HITS hub ranking:** $p_0 \approx p_4 > p_1 \approx p_2 > p_3$

From these results, we can see that PageRank gives us that p_3 is the most likely to be clicked on from a random click and that p_0 is the least likely to be clicked on. Further, we see from the HITS authority ranking that p_2 has the most valuable content while p_4 has the least valuable content. Also, from the HITS hub ranking we see that p_0 and p_4 have the highest value of links to other pages, while p_3 has the lowest value of links to other pages.

These results show interesting patterns as we have that even though p_3 is the most likely to be clicked on, it actually only has the 2nd most valuable content and the least value of links to other pages. Also, even though p_0 is the least likely to be clicked on, it has the 2nd lowest valuable content but the most value of links to other pages. These differences arise from the different ways that each algorithm computes these rankings.

Further, the differences in how PageRank and HITS algorithms rank pages becomes apparent and we can see that these can be helpful in different use cases. For example, if someone highly values the content of a website and the other content it links to, HITS will provide much better results, while for other users who don't have as high of a priority for those things and are more generally browsing, PageRank provides the generic probabilities of each site being clicked on. Thus, whether or not we are using importance of the page itself and the other important pages it links to as a key factor can affect the type of algorithm to be used.

Conversation on Dangling Nodes

A fundamental difference that can be noticed is that we do not explicitly handle dandling nodes unlike in the PageRank implementation. This is because HITS does so naturally. Take p_3 into account for example. When computing the hub score where $h = La$, such pages with no outlinks will have a row of zeros in L, hence resulting in a hub score of zero due to matrix vector multiplication. This is desired and makes sense intuitively as a page cannot be a hub if it doesn't like to any other pages.

However, dangling nodes can still have a high authority score. For example, p_3 is a dangling node and by definition does not have any outbound links. we established it has a hub score of 0. However, because it is linked by pages p_0 and p_2, it receives a high authority score of 0.33456655. This asymmetry between how dangling nodes affect authority and hub scores is a fundamental difference between HITS and PageRank. As detailed above, PageRank forces us to directly handle dangling nodes via the dampening factor and replacing columns in the linking matrix, while in HITS, this is handled by design.

VII. Extension: Personalized PageRank

Motivation and Background

PageRank is a much more general algorithm that simply provides a general ranking that applies to all users, so there is not personalized element in this. Personalized PageRank (PPR) gives rankings that take into account certain personalized biases to offer a more tailored experience to the user. These biases are in the form of sets of seed pages that are subsets of all of the pages (essentially, only the ones that the user has an interest in). [4]

Initialization

From section III, we have that standard PageRank uses the formula $R = f + dLR$ where $f = \begin{bmatrix} \frac{1-d}{n} & \frac{1-d}{n} & \cdots & \frac{1-d}{n} \end{bmatrix}^T$ distributes the teleportation probability uniformly across all pages.

In Personalized PageRank, we replace the uniform teleportation vector f with a personalized teleportation vector v that encodes user preferences. Let $S \subseteq \{p_1, p_2, \ldots, p_n\}$ be a set of seed pages that the user is interested in. We define the personalization vector as

$$v_i = \begin{cases} \frac{1-d}{|S|} & \text{if } p_i \in S \\ 0 & \text{if } p_i \notin S \end{cases}$$

Hence, we modify the basic PageRank formula and the Personalized PageRank formula becomes:

$$R_{PPR} = v + dLR_{PPR}$$

This way, the scores that the algorithm gives back are much more biased towards the user's preferences and interests rather than generic as PageRank normally is. [1]

Implementation

We implement Personalized PageRank by modifying our existing PageRank implementation to accept a personalization vector instead of the original teleportation one, so the rest of the algorithm remains the same except we create a new function to create the personalization vector as shown above

```python
def personalization_vector(seeds, d):
    v = np.zeros(n, dtype=float)
    for page in seeds:
        v[page] = (1 - d) / len(seeds)
    return v
```

We put it all together with the following function.

```
def personalized_pagerank(links, seed_pages, d = 0.85, epsilon = 0.0001):
  R0 = initial_pagerank(links)
  L = compute_link_mat(links)
  v = personalization_vector(seed_pages, d)
  R1 = v + d * np.dot(L, R0)

  diff = np.sum(abs(R1 - R0))
  differences = [diff]

  itr = 1
  while(diff > epsilon):
    itr = itr + 1
    R0 = R1
    R1 = v + d * np.dot(L, R0)

    diff = np.sum(abs(R1 - R0))
    differences.append(diff)

  return R1
```

Examples

We test Personalized PageRank on our example network with different seed sets to demonstrate how personalization affects the rankings.

Using a Single Seed page

First, we compute Personalized PageRank with p_2 as the only seed page. We run the following

```
seed_pages_1 = [2]
R1 = personalized_pagerank(links, seed_pages)
```

Output:

```
Personalization vector (seed pages: [2]):
[0.   0.   0.15 0.   0.  ]

initial pagerank: [0.2 0.2 0.2 0.2 0.2]
iteration 1: [0.09066667 0.13316667 0.36816667 0.2465     0.1615    ] diff = 0.42933
iteration 2: [0.08766333 0.10693    0.31352583 0.37411333 0.1177675 ] diff = 0.25523
iteration 3: [0.09696673 0.11559518 0.31104043 0.34872468 0.12767297] diff = 0.05575
iteration 4: [0.09545721 0.11606263 0.31519059 0.34427299 0.12901658] diff = 0.01192
iteration 5: [0.09508111 0.11536576 0.31469238 0.34672306 0.12813768] diff = 0.0049
iteration 6: [0.0952486  0.11545333 0.31448378 0.34663618 0.1281781 ] diff = 0.00059
iteration 7: [0.09524528 0.11548561 0.31455327 0.34647969 0.12823614] diff = 0.00032
iteration 8: [0.09523512 0.11547474 0.31455613 0.34651145 0.12822255] diff = 7e-05
total iterations: 8
```

Hence, our personalized PageRank is

```
[0.09523512 0.11547474 0.31455613 0.34651145 0.12822255]
```

Now, we can create a helper function to show the difference between our PageRank and Personalized PageRank

```
def compare_personalized_pagerank(PR, PPR, seed_pages):
  print("Page         PageRank      Personalized PR    Difference")
  for i in range(n):
    seed_inf = ""
    if(i in seed_pages):
      seed_inf = " (seed)"
    print("Page " + str(i) + ":      " + str(round(PR[i], 8)) + "     " + str(round
                                                (PPR[i], 8)) + "         " + str(
                                                round((PPR[i] - PR[i]), 8)) + "  " +
                                                 seed_inf)

compare_personalized_pagerank(pr, R1, seed_pages_1)
```

```
Page         PageRank      Personalized PR    Difference
Page 0:      0.13181638    0.09523512         -0.03658126
Page 1:      0.15982697    0.11547474         -0.04435222
Page 2:      0.22775457    0.31455613         0.08680156   (seed)
Page 3:      0.30313336    0.34651145         0.0433781
Page 4:      0.17746873    0.12822255         -0.04924618
```

We now analyze the changes. Page p_2, our seed page, shows a dramatic increase in PageRank from 0.2275457 to 00.31455613174. This makes sense because all teleportation probability is now directed to p_2. Additionally, pages that are closely connected to p_2 maintain relatively high scores since they somewhat relate to the page that the user is interested in. For example, page p_3, which is directly linked from p_2, remains relatively important and actually increased as well from 0.30313336 to 0.34651145. The remaining pages' PageRank score decreased as desired.

Using Multiple Seed Pages

Next, we use multiple seed pages representing a user interested in pages p_0 and p_4:

```
seed_pages_2 = [0, 4]
R2 = personalized_pagerank(links, seed_pages_2, trace = True, plot_diff = False)
```

Output:

```
Personalization vector (seed pages: [0, 4]):
[0.075 0.    0.    0.    0.075]

initial pagerank: [0.2 0.2 0.2 0.2 0.2]
iteration 1: [0.16566667 0.13316667 0.21816667 0.2465     0.2365    ] diff = 0.20233
iteration 2: [0.18391333 0.1441175  0.20071333 0.26255083 0.208705  ] diff = 0.0905
iteration 3: [0.17876673 0.14284831 0.20409825 0.25432156 0.21996516] diff = 0.02929
iteration 4: [0.18055813 0.14354606 0.20425659 0.2547061  0.21693313] diff = 0.00606
iteration 5: [0.17976442 0.14313303 0.2041401  0.25528674 0.21767571] diff = 0.00265
iteration 6: [0.18007353 0.14327347 0.20410501 0.25511777 0.21743022] diff = 0.0009
iteration 7: [0.17997525 0.14324088 0.2041321  0.2551249  0.21752687] diff = 0.00026
iteration 8: [0.18000385 0.14324859 0.20412596 0.25512826 0.21749335] diff = 8e-05
total iterations: 8
```

By using the same compare function, we run `compare_personalized_pagerank(pr, R2, seed_pages_2)`

```
Page        PageRank      Personalized PR    Difference
Page 0:     0.13181638    0.18000385         0.04818746   (seed)
Page 1:     0.15982697    0.14324859         -0.01657838
Page 2:     0.22775457    0.20412596         -0.02362861
Page 3:     0.30313336    0.25512826         -0.0480051
Page 4:     0.17746873    0.21749335         0.04002462   (seed)
```

We now analyze the changes. Page p_0 increases from 0.13181638 to 0.18000385 and page p_4 increases from 0.17746873 to 0.21749335. The ranking order changes significantly as page p_4 goes from third to second, while p_0 goes from last place to second to last. These changes can be attributed to how Personalized PageRank places a greater emphasis on the user's interests. Pages p_1 and p_2, which are linked from both seed pages, also are affected by this slightly as seen above as they are penalized less than $p3$ and have a smaller decrease (since they are given more value since they relate somehow to pages that the user is interested in). This makes sense because both seed pages $p0, p4$ link to $p1, p2$ but $p4$ does not link to $p3$.

Comparison with Standard PageRank and HITS

Personalized PageRank extends the standard PageRank algorithm by introducing user-specific or topic-specific biases through the teleportation distribution. Though HITS is closer to this implementation since it takes into consideration value of the page and the other pages it links to, Personalized PageRank provides a more user-based and tailored method of ranking the pages. Thus, Personalized PageRank can be especially useful in user-focused use cases, such as personalized searches or a movie recommendation system.

References

[1] Polo Chau. What is PageRank? Why does it matter? How is it calculated? — Medium, Polo Club of Data Science — Georgia Tech, 2023. [Online; accessed 05-December-2025].

[2] Raluca Tanase. HITS algorithm - hubs and authorities on the internet — cornell.edu, 2025. [Online; accessed 05-December-2025].

[3] Raluca Tanase and Remus Radu. Lecture #3 — PageRank algorithm - the mathematics of google search, 2009. [Online; accessed 05-December-2025].

[4] Stanford University. Topic-Specific PageRank — Stanford NLP IR Book, 2025. [Online; accessed 05-December-2025].

[5] Wikipedia contributors. Hits algorithm — Wikipedia, the free encyclopedia, 2025. [Online; accessed 29-November-2025].

[6] Wikipedia contributors. Pagerank — Wikipedia, the free encyclopedia, 2025. [Online; accessed 26-November-2025].

Appendix

A: Python Code

```
#import numpy for vector and matrix arithmetic and matplotlib for plotting
import numpy as np
import matplotlib.pyplot as plt

#intialize an list of lists where list at entry i represents the outlinks for
                                                    such page

links = [
    [1, 2, 3, 4], #p0 -> p1, p2, p3, p4
    [2, 4],       #p1 -> p2, p4
    [3],          #p2 -> p3
    [],           #p3 -> none (dangling)
    [2, 0, 1],    #p4 -> p2, p0, p1
]

#size of network, number of links
n = len(links)

#return a uniform probablity vector where each pagerank for page i is set to 1/n
def initial_pagerank(links):
  entry = 1/n
  R = np.full(n, entry, dtype = float)
  return R

#by the iterative equation, returns the pagerank vector Ri = f + dLR
def iterate_pagerank(R, L, d):
  f = np.full(n, (1-d)/n, dtype = float)
  Ri = f + d * np.dot(L, R)
  return Ri

#creates an n by n linking matrix where the Li,j entry is 1/o if
#page pj links to pi and 0 otherwise. note that o is the number of
#outbound links from pj. we implement the dangling node fix by
#replacing zero columns with a uniformly distributed column of entries
#1/n
def compute_link_mat(links):
  L = np.zeros((n, n), dtype = float)
  for i in range(n):
    for j in range(n):
      if len(links[j]) == 0:
        L[i][j] = 1/n
      elif i in links[j]:
        o = len(links[j])
        L[i][j] = 1/o
  return L
```

```
#iteratively determine pagerank until the difference is negligible.
#included code for plotting and tracing for comparisons
def pagerank(links, d = 0.85, epsilon = 0.0001, trace = True, plot_diff = True):
  R0 = initial_pagerank(links)
  L = compute_link_mat(links)
  R1 = iterate_pagerank(R0, L, d)

  diff = np.sum(abs(R1 - R0))
  differences = [diff]

  if trace:
    print("initial pagerank: " + str(R0))
    print("iteration 1: " + str(R1)  + " diff = " + str(round(diff, 5)))

  itr = 1
  while(diff > epsilon):
    itr = itr + 1
    R0 = R1
    R1 = iterate_pagerank(R0, L, d)

    diff = np.sum(abs(R1 - R0))
    differences.append(diff)

    if trace:
      print("iteration " + str(itr) + ": " + str(R1) + " diff = " + str(round(
                                                  diff, 5)))

  if trace:
    print("total iterations: " + str(itr))

  if plot_diff:
    plt.figure(figsize=(5, 5))
    plt.title("difference between iterations")
    plt.plot(differences, "-o")
    plt.xlabel("iteration")
    plt.ylabel("difference")

    plt.grid(True)
    plt.show()

    plt.figure(figsize=(5, 5))
    plt.title("difference between iterations (y scale log)")
    plt.plot(differences, "-o")
    plt.xlabel("iteration")
    plt.ylabel("difference")
    plt.yscale("log")

    plt.grid(True)
    plt.show()
  return R1
```

```
pr = pagerank(links)
print("final pagerank: " + str((pr, 5)))

"""### HITS Implementation

"""

#compute the linking matrix L for HITS whre ethe Li,j entry is 1 if
#page i has an outgoing link to page j and 0 otherwise
def compute_link_mat_hits(links):
  L = np.zeros((n, n), dtype = float)
  for i in range(n):
    for j in range(n):
      if j in links[i]:
        L[i][j] = 1
  return L

#set the inital hits and authority vectors to all 1 as required
def initial_hits(links):
  auth = np.full(n, 1, dtype = float)
  hub = np.full(n, 1, dtype = float)
  return auth, hub

#compute a = L^Th and h = La on each iteration
def iterate_hits(auth, hub, L):
  auth = np.dot(L.T, hub)
  auth = auth / np.linalg.norm(auth)
  hub = np.dot(L, auth)
  hub = hub / np.linalg.norm(hub)
  return auth, hub

#iteratively determine the authority and hub score until the differences
#are negligible. included tracing code.
def hits(links, epsilon=1e-6, trace=True):
    L = compute_link_mat_hits(links)

    A0, H0 = initial_hits(links)
    A1, H1 = iterate_hits(A0, H0, L)

    authority_diff = np.linalg.norm(A1 - A0)
    hubs_diff = np.linalg.norm(H1 - H0)

    if trace:
        print("iteration 1: \n    auth =" + str(A1) + "diff auth =" + str(round(
                                                  authority_diff, 8)) +
              "\n    hub =" + str(H1) + "diff hub =" + str(round(hubs_diff, 8)))

    itr = 1
    while authority_diff > epsilon or hubs_diff > epsilon:
        itr += 1
        A0, H0 = A1, H1
        A1, H1 = iterate_hits(A0, H0, L)

        authority_diff = np.linalg.norm(A1 - A0)
        hubs_diff = np.linalg.norm(H1 - H0)

        if trace:
              print("iteration " + str(itr) + ": \n   auth =" + str(A1) + "diff
                                                         auth =" + str(round(
                                                         authority_diff, 8)) +
              "\n   hub =" + str(H1) + "diff hub =" + str(round(hubs_diff, 8)))

    return A1, H1

auth, hub = hits(links)

#determine the indicicies of the auth and hub verticies based
#increasing order of their entries
def determine_hits_order(auth, hub):
  auth_order = np.argsort(-auth)
  hub_order = np.argsort(-hub)
  for i in range(n):
    print("rank " + str(i + 1) + ": page " + str(auth_order[i]))
  return auth_order, hub_order

auth_order, hub_order = determine_hits_order(auth, hub)
print("auth order: " + str(auth_order))
print("hub order: " + str(hub_order))

"""### Extension: Personalized PageRank

"""

#returns a vector for the personalization vector by dividing 1 - d by
#the number of seed pages as required.
def personalization_vector(seeds, d):
    v = np.zeros(n, dtype=float)
    for page in seeds:
        v[page] = (1 - d) / len(seeds)
    return v

#modifies the pagerank code above by computing Ri = v + dLR instead of
#Ri = f + dLR where v is now a biased vector which prioritizes pages
#that are considered seeds instead of being distributed uniformally.
def personalized_pagerank(links, seed_pages, d = 0.85, epsilon = 0.0001, trace =
                                         True, plot_diff = True):
  R0 = initial_pagerank(links)
  L = compute_link_mat(links)
  v = personalization_vector(seed_pages, d)
  R1 = v + d * np.dot(L, R0)
  diff = np.sum(abs(R1 - R0))
  differences = [diff]

  if trace:
    print("personalization vector: " + str(v) + " \n")

    print("initial pagerank: " + str(R0))
    print("iteration 1: " + str(R1)  + " diff = " + str(round(diff, 5)))

  itr = 1
  while(diff > epsilon):
    itr = itr + 1
    R0 = R1
    R1 = v + d * np.dot(L, R0)

    diff = np.sum(abs(R1 - R0))
    differences.append(diff)

    if trace:
      print("iteration " + str(itr) + ": " + str(R1) + " diff = " + str(round(
                                                diff, 5)))

  if trace:
    print("total iterations: " + str(itr))

  if plot_diff:
    plt.figure(figsize=(5, 5))
    plt.title("difference between iterations")
    plt.plot(differences, "-o")
    plt.xlabel("iteration")
    plt.ylabel("difference")

    plt.grid(True)
    plt.show()

    plt.figure(figsize=(5, 5))
    plt.title("difference between iterations (y scale log)")
    plt.plot(differences, "-o")
    plt.xlabel("iteration")
    plt.ylabel("difference")
    plt.yscale("log")

    plt.grid(True)
    plt.show()
  return R1

#example with one seed page, p2
seed_pages_1 = [2]
R1 = personalized_pagerank(links, seed_pages_1, trace = True, plot_diff = False)

#example with two seed pages, p0 and p4
```

```
seed_pages_2 = [0, 4]
R2 = personalized_pagerank(links, seed_pages_2, trace = True, plot_diff = False)

#function to print the difference between pagerank scores in a
#visually understandable manner
def compare_personalized_pagerank(PR, PPR, seed_pages):
  print("Page        PageRank      Personalized PR    Difference")
  for i in range(n):
    seed_inf = ""
    if(i in seed_pages):
      seed_inf = " (seed)"
    print("Page " + str(i) + ":     " + str(round(PR[i], 8)) + "    " + str(round
                                            (PPR[i], 8)) + "         " + str(
                                            round((PPR[i] - PR[i]), 8)) + "  " +
                                             seed_inf)

#compare original pagerank and personalized pagerank with single seed [2]
compare_personalized_pagerank(pr, R1, seed_pages_1)

#compare original pagerank and personalized pagerank with two seeds [0, 4]
compare_personalized_pagerank(pr, R2, seed_pages_2)
```