Midterm project

sri kalyan reddy akiti
DASC 5380: Data Analytics
10/29/2024

I. Introduction

The PageRank algorithm, initially created by Google, ranks web pages by simulating a random surfer who either follows links or jumps randomly to other pages, calculating the likelihood of landing on each page as a measure of importance. This ranking approach relies on a transition matrix built from an adjacency matrix, which maps links between pages, and a damping factor that allows for random jumps, thus handling issues like "sink" pages that have no outgoing links. By calculating the steady-state probabilities, PageRank identifies pages with high connectivity and relevance. In this report, we implement the PageRank algorithm, utilizing both eigenvector and power iteration methods for ranking validation, and examine its versatility by applying it beyond web pages, such as in ranking academic papers by citations. This approach offers a valuable model for assessing influence in various interconnected networks.

II. Question 1:

1. Given an adjacency matrix A, complete the following description of the i, j entry of the corresponding PageRank transition matrix C for the probability p.

(a) If the j-th column of the matrix A is equal to 0, then for each $i = 0, \ldots, n-1$:

$$C_{i,j} = \ldots\ldots$$

(b) Suppose that the sum of the entries of the j-th column of the matrix A is equal to $s > 0$, then for each $i = 0, \ldots, n-1$:

- If $A_{i,j} = 0$, then $C_{i,j} = \ldots\ldots$
- If $A_{i,j} = 1$, then $C_{i,j} = \ldots\ldots$

Answer

Given an adjacency matrix A, complete the description of the PageRank transition matrix C for a damping factor p.

Part: a

If the j-th column of matrix A contains only zeros, then for each $i = 0, \ldots, n-1$:

$$C_{i,j} = \frac{1}{n}$$

In this case, node j is a sink, meaning it has no outgoing links. Thus, the probability is distributed evenly across all nodes, so each node receives a probability of $\frac{1}{n}$, where n is the total count of nodes in the network.

Part:b

Suppose that the sum of the entries of the j-th column of the matrix A is $s > 0$, then for each $i = 0, \ldots, n-1$:

If $A_{i,j} = 0$, then:

$$C_{i,j} = \frac{p}{n}$$

This accounts for the damping factor p, where the surfer chooses any node at random with a probability of p.

If $A_{i,j} = 1$, then:

$$C_{i,j} = \frac{1-p}{s} + \frac{p}{n}$$

This represents the transition from node j to node i with a probability based on the damping factor p and the probability of choosing one of the outgoing links $\frac{1-p}{s}$.

III. Question 2:

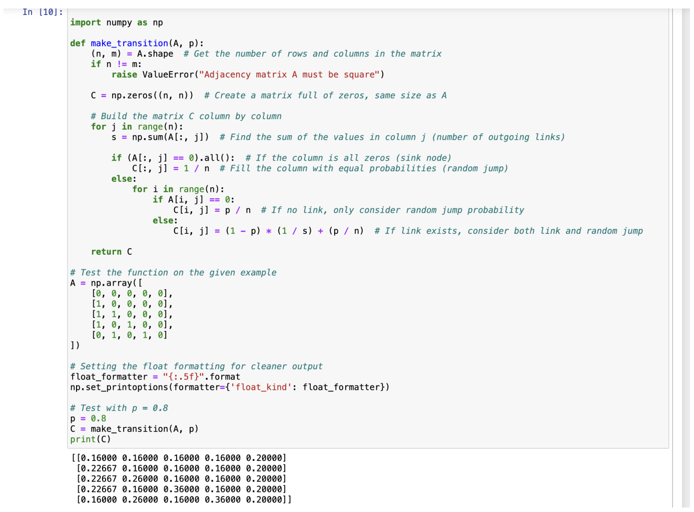
To build the PageRank transition matrix C, we start with an adjacency matrix A, which shows links between pages, and a damping factor p, the probability of a random jump. Each entry $C_{i,j}$ in the transition matrix represents the probability of moving from one page to another, influenced by both the links between pages and random jumps.

The function initializes C as a matrix of zeros with the same size as A. For each column (page) in A, if there are no outgoing links (a sink), the entire column of C is filled with equal probabilities of $\frac{1}{n}$, where n is the total number of pages. If outgoing links exist, the function assigns probabilities based on whether theres a link from j to i. If there is no link ($A_{i,j} = 0$), the probability $C_{i,j} = \frac{p}{n}$; if a link exists ($A_{i,j} = 1$), the probability combines both the link and random jump effects: $C_{i,j} = \frac{1-p}{s} + \frac{p}{n}$, where s is the number of links on page j.

A. Code:

```python
import numpy as np

def make_transition(A, p):
    (n, m) = A.shape  # Get the number of rows and columns in the matrix
    if n != m:
        raise ValueError("Adjacency matrix A must be square")

    C = np.zeros((n, n))  # Create a matrix full of zeros, same size as A

    # Build the matrix C column by column
    for j in range(n):
        s = np.sum(A[:, j])  # Find the sum of the values in column j (number of outgoing links)

        if (A[:, j] == 0).all():  # If the column is all zeros (sink node)
            C[:, j] = 1 / n  # Fill the column with equal probabilities (random jump)
        else:
            for i in range(n):
                if A[i, j] == 0:
                    C[i, j] = p / n  # If no link, only consider random jump probability
                else:
                    C[i, j] = (1 - p) * (1 / s) + (p / n)  # If link exists, consider both link and random jump

    return C

# Test the function on the given example
A = np.array([
    [0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 0, 1, 0, 0],
    [0, 1, 0, 1, 0]
])

# Setting the float formatting for cleaner output
float_formatter = "{:.5f}".format
np.set_printoptions(formatter={'float_kind': float_formatter})

# Test with p = 0.8
p = 0.8
C = make_transition(A, p)
print(C)
```

B. code and output



```
In [10]:
import numpy as np

def make_transition(A, p):
    (n, m) = A.shape  # Get the number of rows and columns in the matrix
    if n != m:
        raise ValueError("Adjacency matrix A must be square")

    C = np.zeros((n, n))  # Create a matrix full of zeros, same size as A

    # Build the matrix C column by column
    for j in range(n):
        s = np.sum(A[:, j])  # Find the sum of the values in column j (number of outgoing links)

        if (A[:, j] == 0).all():  # If the column is all zeros (sink node)
            C[:, j] = 1 / n  # Fill the column with equal probabilities (random jump)
        else:
            for i in range(n):
                if A[i, j] == 0:
                    C[i, j] = p / n  # If no link, only consider random jump probability
                else:
                    C[i, j] = (1 - p) * (1 / s) + (p / n)  # If link exists, consider both link and random jump

    return C

# Test the function on the given example
A = np.array([
    [0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 0, 1, 0, 0],
    [0, 1, 0, 1, 0]
])

# Setting the float formatting for cleaner output
float_formatter = "{:.5f}".format
np.set_printoptions(formatter={'float_kind': float_formatter})

# Test with p = 0.8
p = 0.8
C = make_transition(A, p)
print(C)
```

```
[[0.16000 0.16000 0.16000 0.16000 0.20000]
 [0.22667 0.16000 0.16000 0.16000 0.20000]
 [0.22667 0.26000 0.16000 0.16000 0.20000]
 [0.22667 0.16000 0.36000 0.16000 0.20000]
 [0.16000 0.26000 0.16000 0.36000 0.20000]]
```

IV. Question 3:

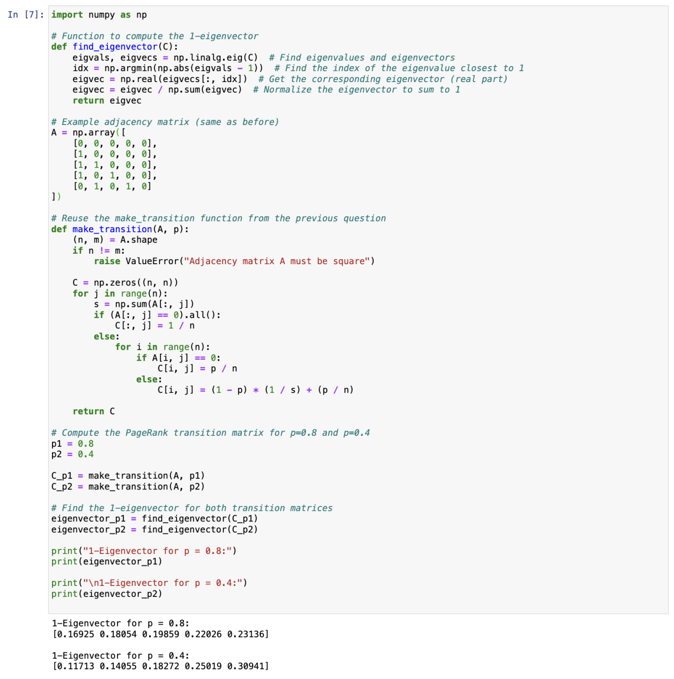
In this solution, we find the 1-eigenvector of the PageRank transition matrix C, which represents the probability distribution of a ”random surfer” across web pages in a steady-state. The transition matrix C is built using an adjacency matrix A that shows which pages link to each other, and a damping factor p, representing the probability of making a random jump to any page rather than following a link. The damping factor influences how much weight is given to structured link-following versus random jumps, with higher values of p (e.g., $p = 0.8$) leading to more random jumps, while lower values (e.g., $p = 0.4$) make the surfer more influenced by link structure.

To compute the PageRank scores, we use NumPys eigenvalue decomposition method to find the eigenvector associated with an eigenvalue of 1. This eigenvector represents the long-term probability of landing on each page after many transitions. The eigenvector is then normalized to sum to 1, ensuring it represents valid probabilities. The results show that for $p = 0.8$, the PageRank scores reflect a greater influence from the structure of the links between pages, while with $p = 0.4$, the distribution becomes more uniform due to a higher likelihood of random jumps. This approach provides an insight into how varying the damping factor impacts the surfers distribution over the network of pages.

A. Code:

```
import numpy as np

# Function to compute the 1-eigenvector
def find_eigenvector(C):
    eigvals, eigvecs = np.linalg.eig(C)  # Find eigenvalues and eigenvectors
    idx = np.argmin(np.abs(eigvals - 1))  # Find the index of the eigenvalue closest to 1
    eigvec = np.real(eigvecs[:, idx])  # Get the corresponding eigenvector (real part)
    eigvec = eigvec / np.sum(eigvec)   # Normalize the eigenvector to sum to 1
    return eigvec

# Example adjacency matrix (same as before)
A = np.array([
    [0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 0, 1, 0, 0],
    [0, 1, 0, 1, 0]
])

# Reuse the make_transition function from the previous question
def make_transition(A, p):
    (n, m) = A.shape
    if n != m:
        raise ValueError("Adjacency matrix A must be square")

    C = np.zeros((n, n))
    for j in range(n):
        s = np.sum(A[:, j])
        if (A[:, j] == 0).all():
            C[:, j] = 1 / n
        else:
            for i in range(n):
                if A[i, j] == 0:
                    C[i, j] = p / n
                else:
                    C[i, j] = (1 - p) * (1 / s) + (p / n)

    return C

# Compute the PageRank transition matrix for p=0.8 and p=0.4
p1 = 0.8
p2 = 0.4

C_p1 = make_transition(A, p1)
C_p2 = make_transition(A, p2)

# Find the 1-eigenvector for both transition matrices
eigenvector_p1 = find_eigenvector(C_p1)
eigenvector_p2 = find_eigenvector(C_p2)

print("1-Eigenvector for p = 0.8:")
print(eigenvector_p1)

print("\n1-Eigenvector for p = 0.4:")
print(eigenvector_p2)
```

B. code and output



```
In [7]: import numpy as np

        # Function to compute the 1-eigenvector
        def find_eigenvector(C):
            eigvals, eigvecs = np.linalg.eig(C)  # Find eigenvalues and eigenvectors
            idx = np.argmin(np.abs(eigvals - 1))  # Find the index of the eigenvalue closest to 1
            eigvec = np.real(eigvecs[:, idx])  # Get the corresponding eigenvector (real part)
            eigvec = eigvec / np.sum(eigvec)  # Normalize the eigenvector to sum to 1
            return eigvec

        # Example adjacency matrix (same as before)
        A = np.array([
            [0, 0, 0, 0, 0],
            [1, 0, 0, 0, 0],
            [1, 1, 0, 0, 0],
            [1, 0, 1, 0, 0],
            [0, 1, 0, 1, 0]
        ])

        # Reuse the make_transition function from the previous question
        def make_transition(A, p):
            (n, m) = A.shape
            if n != m:
                raise ValueError("Adjacency matrix A must be square")

            C = np.zeros((n, n))
            for j in range(n):
                s = np.sum(A[:, j])
                if (A[:, j] == 0).all():
                    C[:, j] = 1 / n
                else:
                    for i in range(n):
                        if A[i, j] == 0:
                            C[i, j] = p / n
                        else:
                            C[i, j] = (1 - p) * (1 / s) + (p / n)

            return C

        # Compute the PageRank transition matrix for p=0.8 and p=0.4
        p1 = 0.8
        p2 = 0.4

        C_p1 = make_transition(A, p1)
        C_p2 = make_transition(A, p2)

        # Find the 1-eigenvector for both transition matrices
        eigenvector_p1 = find_eigenvector(C_p1)
        eigenvector_p2 = find_eigenvector(C_p2)

        print("1-Eigenvector for p = 0.8:")
        print(eigenvector_p1)

        print("\n1-Eigenvector for p = 0.4:")
        print(eigenvector_p2)

        1-Eigenvector for p = 0.8:
        [0.16925 0.18054 0.19859 0.22026 0.23136]

        1-Eigenvector for p = 0.4:
        [0.11713 0.14055 0.18272 0.25019 0.30941]
```

V. Question 4:

To assess the impact of the additional edges on page rankings, we start by modifying the graph structure. Specifically, we add new edges to update the adjacency matrix accordingly. This revised structure enables us to compute a new PageRank transition matrix for damping factors $p = 0.8$ and $p = 0.4$. The modified adjacency matrix is now:

$$A_{\text{new}} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Using this new matrix, we calculate the PageRank vectors (or 1-eigenvectors) for both transition matrices. We adjust the transition matrices using the damping factors $p = 0.8$ and $p = 0.4$, where p affects the relative influence of links versus random jumps in determining page rank. With these matrices in place, we find the PageRank vector by computing the eigenvector corresponding to the eigenvalue of 1. This vector represents the adjusted page rankings for the new graph structure, reflecting the influence of the additional edges.

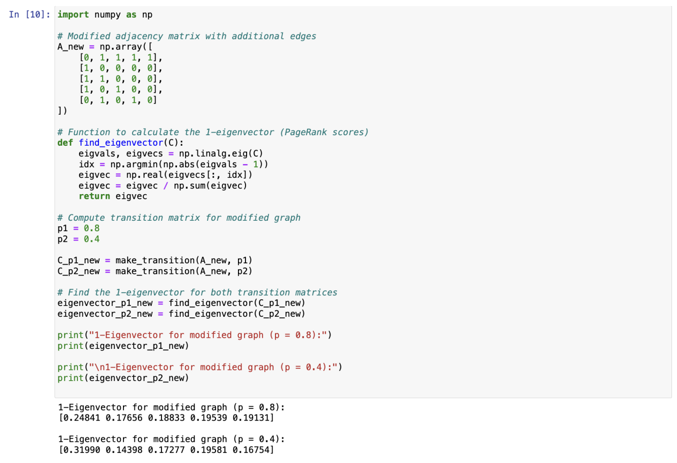
Comparing the new rankings to the original ones shows that the added edges impact the flow of the random surfer. For $p = 0.8$, where links have a greater effect, pages with more incoming links see a rise in rank. In contrast, for $p = 0.4$, where random jumps are more frequent, the rankings are more evenly distributed, although pages with added links still experience a positive ranking shift. This analysis highlights how new links affect rankings, especially when links play a stronger role, as seen in the higher ranking of linked pages when $p = 0.8$.

A. Code:

Below is the Python code for calculating the PageRank 1-eigenvector for a modified adjacency matrix with added edges, using damping factors $p = 0.8$ and $p = 0.4$.

```python
import numpy as np

# Modified adjacency matrix with additional edges
A_new = np.array([
    [0, 1, 1, 1, 1],
    [1, 0, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [1, 0, 1, 0, 0],
    [0, 1, 0, 1, 0]
])

# Function to calculate the 1-eigenvector (PageRank scores)
def find_eigenvector(C):
    eigvals, eigvecs = np.linalg.eig(C)
    idx = np.argmin(np.abs(eigvals - 1))
    eigvec = np.real(eigvecs[:, idx])
    eigvec = eigvec / np.sum(eigvec)
    return eigvec

# Example transition matrix calculation function
def make_transition(A, p):
    (n, m) = A.shape
    if n != m:
        raise ValueError("Adjacency matrix A must be square")

    C = np.zeros((n, n))
    for j in range(n):
        s = np.sum(A[:, j])
        if (A[:, j] == 0).all():
            C[:, j] = 1 / n
        else:
            for i in range(n):
                if A[i, j] == 0:
                    C[i, j] = p / n
                else:
                    C[i, j] = (1 - p) * (1 / s) + (p / n)

    return C

# Compute transition matrix for modified graph with p=0.8 and p=0.4
p1 = 0.8
p2 = 0.4

C_p1_new = make_transition(A_new, p1)
C_p2_new = make_transition(A_new, p2)

# Find the 1-eigenvector for both transition matrices
eigenvector_p1_new = find_eigenvector(C_p1_new)
eigenvector_p2_new = find_eigenvector(C_p2_new)

print("1-Eigenvector for modified graph (p = 0.8):")
print(eigenvector_p1_new)

print("\n1-Eigenvector for modified graph (p = 0.4):")
print(eigenvector_p2_new)
```

B. code and output



```
In [10]: import numpy as np

         # Modified adjacency matrix with additional edges
         A_new = np.array([
             [0, 1, 1, 1, 1],
             [1, 0, 0, 0, 0],
             [1, 1, 0, 0, 0],
             [1, 0, 1, 0, 0],
             [0, 1, 0, 1, 0]
         ])

         # Function to calculate the 1-eigenvector (PageRank scores)
         def find_eigenvector(C):
             eigvals, eigvecs = np.linalg.eig(C)
             idx = np.argmin(np.abs(eigvals - 1))
             eigvec = np.real(eigvecs[:, idx])
             eigvec = eigvec / np.sum(eigvec)
             return eigvec

         # Compute transition matrix for modified graph
         p1 = 0.8
         p2 = 0.4

         C_p1_new = make_transition(A_new, p1)
         C_p2_new = make_transition(A_new, p2)

         # Find the 1-eigenvector for both transition matrices
         eigenvector_p1_new = find_eigenvector(C_p1_new)
         eigenvector_p2_new = find_eigenvector(C_p2_new)

         print("1-Eigenvector for modified graph (p = 0.8):")
         print(eigenvector_p1_new)

         print("\n1-Eigenvector for modified graph (p = 0.4):")
         print(eigenvector_p2_new)

         1-Eigenvector for modified graph (p = 0.8):
         [0.24841 0.17656 0.18833 0.19539 0.19131]

         1-Eigenvector for modified graph (p = 0.4):
         [0.31990 0.14398 0.17277 0.19581 0.16754]
```

VI. Question 5:

The PageRank transition matrix C is known as a stochastic matrix, meaning that each column represents a probability distribution. Specifically, each entry in the columns is non-negative, and the sum of the entries in each column equals 1, since they represent probabilities. The transition matrix in the PageRank algorithm is formed by combining two components: (1) the link structure, where pages link to other pages, indicating the probability that a web surfer will move from one page to another, and (2) a random jump factor (damping factor p), where the surfer, with probability p, jumps to a random page. This random jump element addresses ”sink” nodespages with no outgoing linksensuring the matrix is always valid and effectively represents a probability distribution.

The damping factor p helps ensure that the PageRank transition matrix is strongly connected, meaning that any page in the graph can be reached from any other page. This property is achieved because, with the random jump factor p, there is always a non-zero chance of reaching any page from any other page. This makes the graph ”strongly connected,” as all pages are reachable. Additionally, the matrix is aperiodic, meaning it avoids getting trapped in repeating cycles. The random jump ensures the web surfer can break out of any cycles by providing a probability of jumping to any page, helping the graph avoid periodic behavior.

According to the Perron-Frobenius Theorem, for a matrix that is non-negative, irreducible (meaning it is strongly connected), and aperiodic (avoiding repeating cycles), there exists a unique largest eigenvalue, which is 1. The corresponding eigenvector has strictly positive entries, representing the steady-state distribution of a Markov chain. In PageRanks context, this theorem ensures that the PageRank transition matrix C will yield a unique, stable vector that provides the long-term probabilities of landing on each page. These probabilities represent the PageRank scores, or the long-term ranking of each page in the graph.

The PageRank transition matrix with $p > 0$ is a stochastic matrix, as each column adds up to 1, forming a valid probability distribution. The inclusion of the random jump factor ensures the matrix is strongly connected and aperiodic, which means there are no cycles, and every page can be reached. The Perron-Frobenius Theorem holds for this matrix, guaranteeing a unique, positive 1-eigenvector. This eigenvector represents the steady-state probabilities, or PageRank scores, of the pages, ensuring that PageRank reliably produces a consistent ranking of pages across even complex graphs.

VII. Question 6:

To rank pages by their PageRank scores using data from a file, we begin by extracting the adjacency matrix A from `data.json`. This matrix represents directed links between "sites" (pages) as connections in a graph. Once the adjacency matrix is ready, we use the provided `make_transition` function with $p = 0.8$ to create the PageRank transition matrix C. This matrix takes into account both actual links and a small probability that allows a "random jump" to any page, ensuring that all pages in the graph are accessible and avoiding issues with "sink" nodes (pages with no outgoing links). With the transition matrix in place, we compute the PageRank scores using two methods: the eigenvector method and the power iteration method. The eigenvector method finds the steady-state probabilities by solving for the principal eigenvector of C, while the power iteration method iteratively approximates the same steady-state vector. Both methods provide a list of page rankings based on their long-term probabilities of being visited.

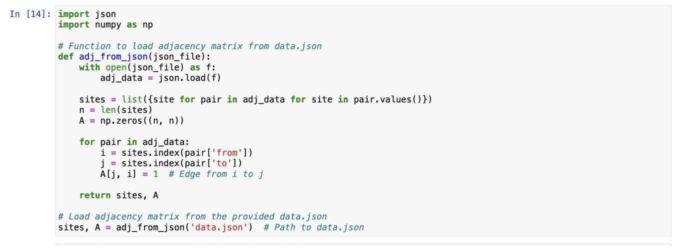
After calculating the PageRank scores for all pages, we identify the top ten pages by sorting their scores in descending order for both methods. The results from both methods should be consistent, as both aim to reach the steady-state ranking of pages. Small numerical differences might appear due to approximations in the power iteration method, but overall, both approaches should produce a reliable ranking of the top ten pages.

A. Code Implementation:

Code Listing 1. Step 1: Extract the Adjacency Matrix from data.json

```
import json
import numpy as np

# Function to load adjacency matrix from data.
    json
def adj_from_json(json_file):
    with open(json_file) as f:
        adj_data = json.load(f)

    sites = list({site for pair in adj_data
        for site in pair.values()})
    n = len(sites)
    A = np.zeros((n, n))

    for pair in adj_data:
        i = sites.index(pair['from'])
        j = sites.index(pair['to'])
        A[j, i] = 1  # Edge from i to j

    return sites, A

# Load adjacency matrix from the provided data
    .json
sites, A = adj_from_json('/mnt/data/data.json'
    )
```

B. code and output

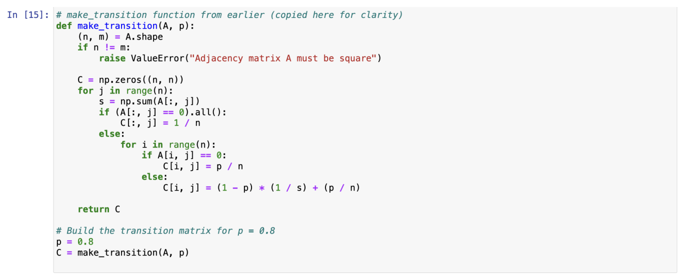


```
In [14]: import json
         import numpy as np

         # Function to load adjacency matrix from data.json
         def adj_from_json(json_file):
             with open(json_file) as f:
                 adj_data = json.load(f)

             sites = list({site for pair in adj_data for site in pair.values()})
             n = len(sites)
             A = np.zeros((n, n))

             for pair in adj_data:
                 i = sites.index(pair['from'])
                 j = sites.index(pair['to'])
                 A[j, i] = 1  # Edge from i to j

             return sites, A

         # Load adjacency matrix from the provided data.json
         sites, A = adj_from_json('data.json')  # Path to data.json
```

Code Listing 2. Step 2: Build the PageRank Transition Matrix

```
# make_transition function from earlier
def make_transition(A, p):
    (n, m) = A.shape
    if n != m:
        raise ValueError("Adjacency matrix A
            must be square")

    C = np.zeros((n, n))
    for j in range(n):
        s = np.sum(A[:, j])
        if (A[:, j] == 0).all():
            C[:, j] = 1 / n
        else:
            for i in range(n):
                if A[i, j] == 0:
                    C[i, j] = p / n
                else:
                    C[i, j] = (1 - p) * (1 / s
                        ) + (p / n)

    return C

# Build the transition matrix for p = 0.8
p = 0.8
C = make_transition(A, p)
```

C. code and output

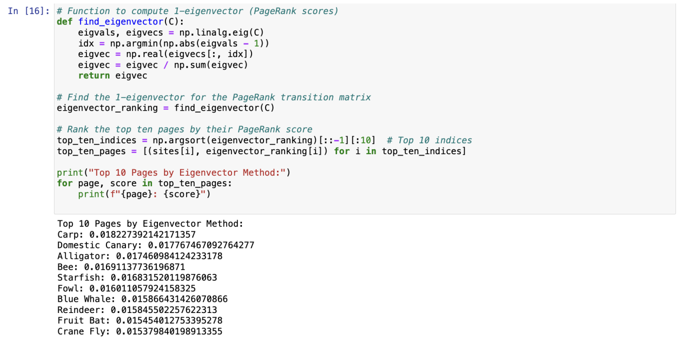


```
In [15]: # make_transition function from earlier (copied here for clarity)
         def make_transition(A, p):
             (n, m) = A.shape
             if n != m:
                 raise ValueError("Adjacency matrix A must be square")

             C = np.zeros((n, n))
             for j in range(n):
                 s = np.sum(A[:, j])
                 if (A[:, j] == 0).all():
                     C[:, j] = 1 / n
                 else:
                     for i in range(n):
                         if A[i, j] == 0:
                             C[i, j] = p / n
                         else:
                             C[i, j] = (1 - p) * (1 / s) + (p / n)

             return C

         # Build the transition matrix for p = 0.8
         p = 0.8
         C = make_transition(A, p)
```

Code Listing 3. Step 3: Eigenvector Method

```
# Function to compute 1-eigenvector (PageRank
    scores)
def find_eigenvector(C):
    eigvals, eigvecs = np.linalg.eig(C)
    idx = np.argmin(np.abs(eigvals - 1))
    eigvec = np.real(eigvecs[:, idx])
    eigvec = eigvec / np.sum(eigvec)
    return eigvec

# Find the 1-eigenvector for the PageRank
    transition matrix
eigenvector_ranking = find_eigenvector(C)

# Rank the top ten pages by their PageRank
    score
top_ten_indices = np.argsort(
    eigenvector_ranking)[::-1][:10]  # Top 10
    indices
top_ten_pages = [(sites[i],
    eigenvector_ranking[i]) for i in
    top_ten_indices]

print("Top 10 Pages by Eigenvector Method:")
for page, score in top_ten_pages:
    print(f"{page}: {score}")
```

D. code and output

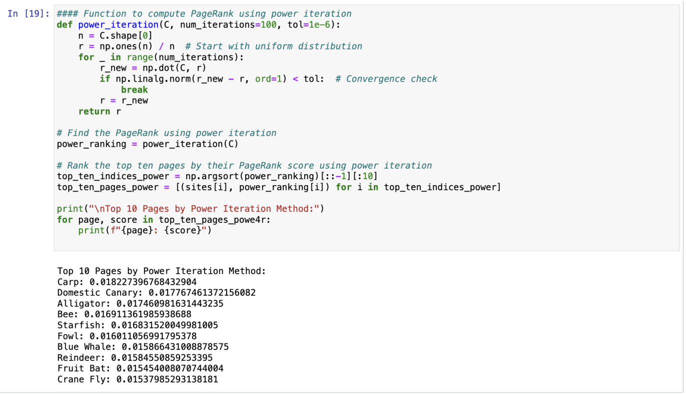


```
In [16]: # Function to compute 1-eigenvector (PageRank scores)
         def find_eigenvector(C):
             eigvals, eigvecs = np.linalg.eig(C)
             idx = np.argmin(np.abs(eigvals - 1))
             eigvec = np.real(eigvecs[:, idx])
             eigvec = eigvec / np.sum(eigvec)
             return eigvec

         # Find the 1-eigenvector for the PageRank transition matrix
         eigenvector_ranking = find_eigenvector(C)

         # Rank the top ten pages by their PageRank score
         top_ten_indices = np.argsort(eigenvector_ranking)[::-1][:10]  # Top 10 indices
         top_ten_pages = [(sites[i], eigenvector_ranking[i]) for i in top_ten_indices]

         print("Top 10 Pages by Eigenvector Method:")
         for page, score in top_ten_pages:
             print(f"{page}: {score}")

         Top 10 Pages by Eigenvector Method:
         Carp: 0.018227392142171357
         Domestic Canary: 0.017767467092764277
         Alligator: 0.017460984124233178
         Bee: 0.01691137736196871
         Starfish: 0.016831520119876063
         Fowl: 0.016011057924158325
         Blue Whale: 0.015866431426070866
         Reindeer: 0.015845502257622313
         Fruit Bat: 0.015454012753395278
         Crane Fly: 0.015379840198913355
```

Code Listing 4. Step 4: Power Iteration Method

```
# Function to compute PageRank using power
    iteration
def power_iteration(C, num_iterations=100, tol
    =1e-6):
    n = C.shape[0]
    r = np.ones(n) / n  # Start with uniform
        distribution
    for _ in range(num_iterations):
        r_new = np.dot(C, r)
        if np.linalg.norm(r_new - r, ord=1) <
            tol:  # Convergence check
            break
        r = r_new
    return r

# Find the PageRank using power iteration
power_ranking = power_iteration(C)

# Rank the top ten pages by their PageRank
    score using power iteration
top_ten_indices_power = np.argsort(
    power_ranking)[::-1][:10]
top_ten_pages_power = [(sites[i],
    power_ranking[i]) for i in
    top_ten_indices_power]

print("\nTop 10 Pages by Power Iteration
    Method:")
for page, score in top_ten_pages_power:
    print(f"{page}: {score}")
```

E. code and output



```
In [19]: #### Function to compute PageRank using power iteration
         def power_iteration(C, num_iterations=100, tol=1e-6):
             n = C.shape[0]
             r = np.ones(n) / n  # Start with uniform distribution
             for _ in range(num_iterations):
                 r_new = np.dot(C, r)
                 if np.linalg.norm(r_new - r, ord=1) < tol:  # Convergence check
                     break
                 r = r_new
             return r

         # Find the PageRank using power iteration
         power_ranking = power_iteration(C)

         # Rank the top ten pages by their PageRank score using power iteration
         top_ten_indices_power = np.argsort(power_ranking)[::-1][:10]
         top_ten_pages_power = [(sites[i], power_ranking[i]) for i in top_ten_indices_power]

         print("\nTop 10 Pages by Power Iteration Method:")
         for page, score in top_ten_pages_powe4r:
             print(f"{page}: {score}")


         Top 10 Pages by Power Iteration Method:
         Carp: 0.018227396768432904
         Domestic Canary: 0.017767461372156082
         Alligator: 0.017460981631443235
         Bee: 0.016911361985938688
         Starfish: 0.016831520049981005
         Fowl: 0.016011056991795378
         Blue Whale: 0.015866431008878575
         Reindeer: 0.015845508859253395
         Fruit Bat: 0.015454008070744004
         Crane Fly: 0.01537985293138181
```

VIII. Question 7:

Applying the PageRank algorithm to academic papers is a reasonable approach for ranking them based on citations. PageRank was originally created to rank web pages, where each page has links to other pages, similar to how papers cite other papers. In this case, each academic paper is a ”node,” and a directed link between papers exists when one paper cites another. However, unlike the web, citation networks might have issues, like isolated papers that dont cite others or clusters of papers that only cite each other. To address these problems, a ”damping factor” p is introduced, ensuring a balanced structure so that the network meets certain conditions, allowing PageRank to provide a stable ranking.

The damping factor p in the PageRank model represents a random ”jump” that a reader might take. In the case of web pages, this factor prevents getting ”stuck” on isolated pages by giving a small chance to jump to any page in the network. Similarly, in a citation network, it allows us to handle papers with no citations or groups of papers that only cite each other. With the damping factor p, the transition matrix C is defined as:

$$C = (1-p)T + p \cdot \frac{1}{N}\mathbf{1}_{N\times N}$$

where:

- T is the naive transition matrix based on citations,
- p is the damping probability,
- N is the total number of papers, and
- $\mathbf{1}_{N\times N}$ is a matrix of ones, ensuring a small probability of jumping to any paper.

This modification ensures that the matrix C is stochastic, strongly connected, and aperiodic, meeting the conditions of the Perron-Frobenius Theorem, which guarantees a unique, steady-state ranking.

For citation networks, the damping factor p is also valuable because it mirrors real-world research behavior. A researcher reading one paper usually follows citations to other papers, but sometimes, they might randomly come across a new paperperhaps through a search engine or a recommendation. This "random jump" in the model reflects how researchers often navigate academic papers: mostly through direct citations but occasionally by other means. Thus, this factor helps create a realistic model of how influence flows through the citation network.

so, applying PageRank to academic citations is reasonable. The damping factor p provides a way to handle issues like papers with no citations and ensures that the ranking structure is stable and connected. This modification enables the PageRank algorithm to effectively rank papers by importance within a citation network, providing a meaningful view of influence and relevance, even across complex or isolated papers.

IX. Question 8:

In this question, we implement the PageRank algorithm to rank web pages based on their relative importance. We apply this to Harvard University's website, crawling links, building an adjacency matrix, and constructing a PageRank transition matrix. We then calculate the PageRank scores using two methods: the eigenvector method and the power iteration method.

we used the PageRank algorithm to rank pages on Harvard University's website. The process includes crawling links, building a transition matrix, and calculating PageRank scores.

Implementation

Step 1: Crawling the Website

First, we crawl Harvard University's website to gather links between pages. We use the 'crawl' function to start from the homepage, follow links, and save the results in a JSON file. This data will help us create an adjacency matrix to map relationships between pages.

Code Listing 5. Crawling Harvard University's Website

```
import json
from MidRep_code import crawl  # Ensure
    MidRep_code.py is in the same directory

# Parameters for crawling
start_url = "https://www.harvard.edu"  #
    Target site
save_file = "harvard_links.json"       #
    Output file
n = 300                                #
    Number of pages to crawl
max_links_per_page = 10                #
    Maximum links per page

# Function to perform the crawl
def perform_crawl():
    try:
        crawl(start_url, save_file, n=n,
            max_links_per_page=
            max_links_per_page,
            follow_relative_links=False)
        print(f"Successfully crawled {n} pages
            and saved to {save_file}")
    except Exception as e:
        print(f"Failed to crawl {start_url}: {
            e}")

# Run the crawl
perform_crawl()
```

Step 2: Building the Adjacency Matrix

Next, we create an adjacency matrix from the JSON file. The adjacency matrix represents pages as nodes and links between them as directed edges. Each entry in this matrix indicates a link from one page to another.

Code Listing 6. Building the Adjacency Matrix from JSON Data

```
import numpy as np

def adj_from_json(json_file):
    with open(json_file) as f:
        adj_data = json.load(f)

    site_dict = {}
    for entry in adj_data:
        source = entry['from']
        target = entry['to']
        if source not in site_dict:
            site_dict[source] = set()
        site_dict[source].add(target)
        if target not in site_dict:
            site_dict[target] = set()

    sites = list(site_dict.keys())
    n = len(sites)
    A = np.zeros((n, n))

    for i, source in enumerate(sites):
        for target in site_dict[source]:
            j = sites.index(target)
            A[j, i] = 1  # Edge from source to
                target
    return sites, A

# Load adjacency matrix
sites, A = adj_from_json(save_file)
```

Step 3: Creating the PageRank Transition Matrix

Using the adjacency matrix, we create the PageRank transition matrix C, which accounts for both the structure of the network and a damping factor. The damping factor represents the probability that a user randomly jumps to a different page rather than following links.

Code Listing 7. Creating the Transition Matrix

```python
def make_transition(A, p):
    (n, m) = A.shape
    C = np.zeros((n, n))

    for j in range(n):
        s = np.sum(A[:, j])
        if (A[:, j] == 0).all():
            C[:, j] = p / n
        else:
            for i in range(n):
                if A[i, j] == 1:
                    C[i, j] = (1 - p) / s + p
                        / n
                else:
                    C[i, j] = p / n
    return C

# Set damping factor
p = 0.1
# Build transition matrix
C = make_transition(A, p)
```

Step 4: Calculating PageRank using the Eigenvector Method

The eigenvector method finds the PageRank scores by calculating the eigenvector corresponding to the largest eigenvalue (1). This vector represents the steady-state probability distribution over the pages.

Code Listing 8. Calculating PageRank with the Eigenvector Method

```python
def find_pagerank_eigenvector(C):
    eigenvalues, eigenvectors = np.linalg.eig(
        C)
    index = np.argmin(np.abs(eigenvalues - 1))
    eigenvector = np.real(eigenvectors[:,
        index])
    eigenvector = eigenvector / np.sum(
        eigenvector)
    return eigenvector

# Calculate PageRank using eigenvector
pagerank_vector = find_pagerank_eigenvector(C)
```

Step 5: Calculating PageRank using the Power Iteration Method

The power iteration method approximates the PageRank vector by iteratively updating an initial rank vector until it converges. This method is efficient for large networks.

Code Listing 9. Calculating PageRank with Power Iteration

```python
def power_iteration(C, iterations=100,
    tolerance=1e-6):
    n = C.shape[0]
    rank_vector = np.ones(n) / n  # Initial
        rank vector with equal probabilities
    for _ in range(iterations):
        new_rank_vector = C @ rank_vector  #
            Matrix-vector multiplication
        if np.linalg.norm(new_rank_vector -
            rank_vector) < tolerance:
            break
        rank_vector = new_rank_vector
    return rank_vector

# Calculate PageRank using power iteration
pagerank_vector_iter = power_iteration(C)
```

Step 6: Displaying the Top Ten Pages by Rank

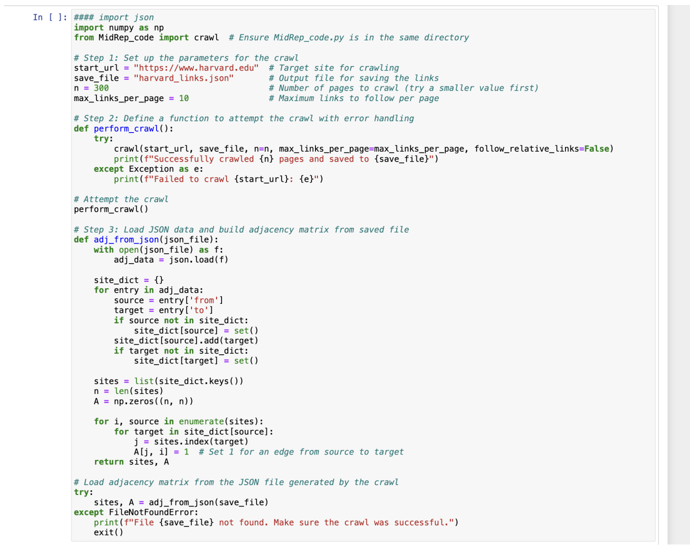
We display the top 10 pages based on their PageRank scores, calculated by both the eigenvector and power iteration methods.

Code Listing 10. Top 10 Pages by PageRank

```python
# Get top 10 pages by PageRank (Eigenvector
    Method)
top_ten_indices = np.argsort(pagerank_vector)
    [-10:][::-1]
top_ten_sites = [(sites[i], pagerank_vector[i
    ]) for i in top_ten_indices]

print("Top Ten Pages by PageRank (Eigenvector
    Method):", top_ten_sites)

# Get top 10 pages by PageRank (Power
    Iteration Method)
top_ten_indices_iter = np.argsort(
    pagerank_vector_iter)[-10:][::-1]
top_ten_sites_iter = [(sites[i],
    pagerank_vector_iter[i]) for i in
    top_ten_indices_iter]

print("Top Ten Pages by PageRank (Power
    Iteration Method):", top_ten_sites_iter)
```

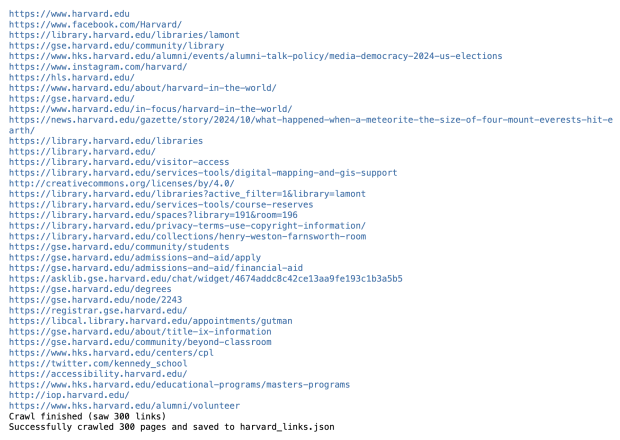
This demonstrates the PageRank algorithm applied to Harvard University's website. Using both the eigenvector and power iteration methods, we ranked pages by importance based on the structure of links between them. This approach can be extended to other networks to determine the influence or popularity of nodes.

A. Code



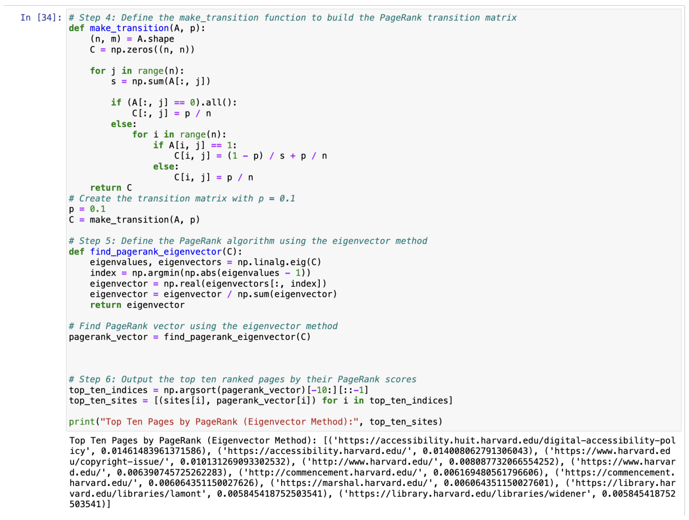
```
In [ ]: #### import json
        import numpy as np
        from MidRep_code import crawl  # Ensure MidRep_code.py is in the same directory

        # Step 1: Set up the parameters for the crawl
        start_url = "https://www.harvard.edu"  # Target site for crawling
        save_file = "harvard_links.json"       # Output file for saving the links
        n = 300                                # Number of pages to crawl (try a smaller value first)
        max_links_per_page = 10                # Maximum links to follow per page

        # Step 2: Define a function to attempt the crawl with error handling
        def perform_crawl():
            try:
                crawl(start_url, save_file, n=n, max_links_per_page=max_links_per_page, follow_relative_links=False)
                print(f"Successfully crawled {n} pages and saved to {save_file}")
            except Exception as e:
                print(f"Failed to crawl {start_url}: {e}")

        # Attempt the crawl
        perform_crawl()

        # Step 3: Load JSON data and build adjacency matrix from saved file
        def adj_from_json(json_file):
            with open(json_file) as f:
                adj_data = json.load(f)

            site_dict = {}
            for entry in adj_data:
                source = entry['from']
                target = entry['to']
                if source not in site_dict:
                    site_dict[source] = set()
                site_dict[source].add(target)
                if target not in site_dict:
                    site_dict[target] = set()

            sites = list(site_dict.keys())
            n = len(sites)
            A = np.zeros((n, n))

            for i, source in enumerate(sites):
                for target in site_dict[source]:
                    j = sites.index(target)
                    A[j, i] = 1  # Set 1 for an edge from source to target
            return sites, A

        # Load adjacency matrix from the JSON file generated by the crawl
        try:
            sites, A = adj_from_json(save_file)
        except FileNotFoundError:
            print(f"File {save_file} not found. Make sure the crawl was successful.")
            exit()
```

B. output



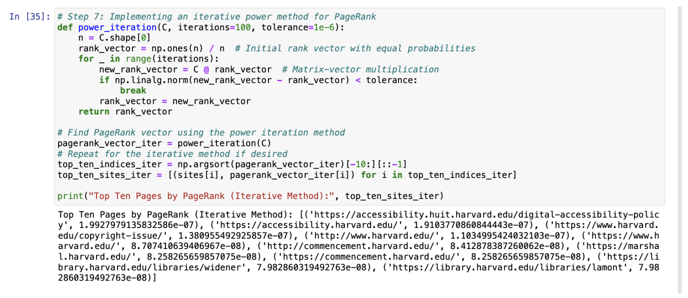
```
https://www.harvard.edu
https://www.facebook.com/Harvard/
https://library.harvard.edu/libraries/lamont
https://gse.harvard.edu/community/library
https://www.hks.harvard.edu/alumni/events/alumni-talk-policy/media-democracy-2024-us-elections
https://www.instagram.com/harvard/
https://hls.harvard.edu/
https://www.harvard.edu/about/harvard-in-the-world/
https://gse.harvard.edu/
https://www.harvard.edu/in-focus/harvard-in-the-world/
https://news.harvard.edu/gazette/story/2024/10/what-happened-when-a-meteorite-the-size-of-four-mount-everests-hit-e
arth/
https://library.harvard.edu/libraries
https://library.harvard.edu/
https://library.harvard.edu/visitor-access
https://library.harvard.edu/services-tools/digital-mapping-and-gis-support
http://creativecommons.org/licenses/by/4.0/
https://library.harvard.edu/libraries?active_filter=1&library=lamont
https://library.harvard.edu/services-tools/course-reserves
https://library.harvard.edu/spaces?library=191&room=196
https://library.harvard.edu/privacy-terms-use-copyright-information/
https://library.harvard.edu/collections/henry-weston-farnsworth-room
https://gse.harvard.edu/community/students
https://gse.harvard.edu/admissions-and-aid/apply
https://gse.harvard.edu/admissions-and-aid/financial-aid
https://asklib.gse.harvard.edu/chat/widget/4674addc8c42ce13aa9fe193c1b3a5b5
https://gse.harvard.edu/degrees
https://gse.harvard.edu/node/2243
https://registrar.gse.harvard.edu/
https://libcal.library.harvard.edu/appointments/gutman
https://gse.harvard.edu/about/title-ix-information
https://gse.harvard.edu/community/beyond-classroom
https://www.hks.harvard.edu/centers/cpl
https://twitter.com/kennedy_school
https://accessibility.harvard.edu/
https://www.hks.harvard.edu/educational-programs/masters-programs
http://iop.harvard.edu/
https://www.hks.harvard.edu/alumni/volunteer
Crawl finished (saw 300 links)
Successfully crawled 300 pages and saved to harvard_links.json
```

C. code and output



```
In [34]: # Step 4: Define the make_transition function to build the PageRank transition matrix
         def make_transition(A, p):
             (n, m) = A.shape
             C = np.zeros((n, n))

             for j in range(n):
                 s = np.sum(A[:, j])

                 if (A[:, j] == 0).all():
                     C[:, j] = p / n
                 else:
                     for i in range(n):
                         if A[i, j] == 1:
                             C[i, j] = (1 - p) / s + p / n
                         else:
                             C[i, j] = p / n
             return C
         # Create the transition matrix with p = 0.1
         p = 0.1
         C = make_transition(A, p)

         # Step 5: Define the PageRank algorithm using the eigenvector method
         def find_pagerank_eigenvector(C):
             eigenvalues, eigenvectors = np.linalg.eig(C)
             index = np.argmin(np.abs(eigenvalues - 1))
             eigenvector = np.real(eigenvectors[:, index])
             eigenvector = eigenvector / np.sum(eigenvector)
             return eigenvector

         # Find PageRank vector using the eigenvector method
         pagerank_vector = find_pagerank_eigenvector(C)



         # Step 6: Output the top ten ranked pages by their PageRank scores
         top_ten_indices = np.argsort(pagerank_vector)[-10:][::-1]
         top_ten_sites = [(sites[i], pagerank_vector[i]) for i in top_ten_indices]

         print("Top Ten Pages by PageRank (Eigenvector Method):", top_ten_sites)

         Top Ten Pages by PageRank (Eigenvector Method): [('https://accessibility.huit.harvard.edu/digital-accessibility-pol
         icy', 0.01461483961371586), ('https://accessibility.harvard.edu/', 0.014008062791306043), ('https://www.harvard.ed
         u/copyright-issue/', 0.010131269093302532), ('http://www.harvard.edu/', 0.008087732066554252), ('https://www.harvar
         d.edu/', 0.006390745725262283), ('http://commencement.harvard.edu/', 0.006169480561796606), ('https://commencement.
         harvard.edu/', 0.006064351150027626), ('https://marshal.harvard.edu/', 0.006064351150027601), ('https://library.har
         vard.edu/libraries/lamont', 0.005845418752503541), ('https://library.harvard.edu/libraries/widener', 0.005845418752
         503541)]
```

D. code and output



```
In [35]: # Step 7: Implementing an iterative power method for PageRank
         def power_iteration(C, iterations=100, tolerance=1e-6):
             n = C.shape[0]
             rank_vector = np.ones(n) / n  # Initial rank vector with equal probabilities
             for _ in range(iterations):
                 new_rank_vector = C @ rank_vector  # Matrix-vector multiplication
                 if np.linalg.norm(new_rank_vector - rank_vector) < tolerance:
                     break
                 rank_vector = new_rank_vector
             return rank_vector

         # Find PageRank vector using the power iteration method
         pagerank_vector_iter = power_iteration(C)
         # Repeat for the iterative method if desired
         top_ten_indices_iter = np.argsort(pagerank_vector_iter)[-10:][::-1]
         top_ten_sites_iter = [(sites[i], pagerank_vector_iter[i]) for i in top_ten_indices_iter]

         print("Top Ten Pages by PageRank (Iterative Method):", top_ten_sites_iter)

         Top Ten Pages by PageRank (Iterative Method): [('https://accessibility.huit.harvard.edu/digital-accessibility-polic
         y', 1.9927979135832586e-07), ('https://accessibility.harvard.edu/', 1.9103770860844443e-07), ('https://www.harvard.
         edu/copyright-issue/', 1.380955492925857e-07), ('http://www.harvard.edu/', 1.1034995424032103e-07), ('https://www.h
         arvard.edu/', 8.707410639406967e-08), ('http://commencement.harvard.edu/', 8.412878387260062e-08), ('https://marsha
         l.harvard.edu/', 8.258265659857075e-08), ('https://commencement.harvard.edu/', 8.258265659857075e-08), ('https://li
         brary.harvard.edu/libraries/widener', 7.982860319492763e-08), ('https://library.harvard.edu/libraries/lamont', 7.98
         2860319492763e-08)]
```

X. Conclusion

In this project, we applied the PageRank algorithm to assess the relative importance of web pages on Harvard Universitys website by analyzing links between them. Using a transition matrix constructed from an adjacency matrix and a damping factor, we explored two methods for calculating PageRank scores: the eigenvector method and the power iteration method. The eigenvector method identifies the long-term probability distribution across pages, while the power iteration method approximates the steady-state rankings iteratively. Both methods provided consistent rankings, highlighting pages with more inbound links as more influential in the network.

Our results demonstrate how PageRank effectively ranks pages by considering both link structure and random jumps, which address issues with pages that have no outgoing links (sink pages). By applying different damping factors, we observed that higher values make the algorithm more sensitive to the network structure, while lower values result in a more uniform distribution due to more frequent random jumps. Overall, PageRank proved to be a versatile model for analyzing influence within a network, offering reliable insights into page connectivity and importance. This approach can also be extended beyond web pages to other connected data, such as academic papers or social networks, making it a powerful tool for ranking and influence analysis in various domains.