I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work, and including any code produced using generative AI systems. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Algorithms for massive data: Project 3: Link analysis

Pietro Manning

July 2024

Chapter 1

Loading and Analyzing Artworks Dataset

1.1 Loading and Preparing Data

I loaded the dataset `prado.csv` using `pandas`, ensuring that the `work_tags` column was present in the dataset.

1.2 Graph Construction

I constructed an undirected graph `G` using the `NetworkX` library. I added nodes to the graph representing each artwork, using their URLs as unique identifiers. I created edges between nodes based on shared tags, assuming that artworks with common tags are related.

Chapter 2

Graph Construction and Analysis

2.1 Code for Constructing the Graph

The following Python code was used to construct an undirected graph representing artworks, where nodes represent individual artworks and edges represent shared tags between artworks. The ‘NetworkX‘ library was utilized to build and analyze the graph.

```
import networkx as nx
import pandas as pd

# Load the dataset
df = pd.read_csv('prado.csv')

# Initialize an undirected graph
G = nx.Graph()

# Add nodes (pictures)
for work_url in df['work_url']:
    G.add_node(work_url)

# Add edges based on shared tags
tag_to_works = {}

for idx, row in df.iterrows():
    work_url = row['work_url']
    tags = str(row['work_tags']).split(',')
    for tag in tags:
        tag = tag.strip()
        if tag not in tag_to_works:
            tag_to_works[tag] = []
        tag_to_works[tag].append(work_url)

for tag, works in tag_to_works.items():
    for i in range(len(works)):
```

```
        for j in range(i + 1, len(works)):
            G.add_edge(works[i], works[j])
```

Listing 2.1: Graph Construction Code

2.2 Explanation

The code begins by importing the necessary libraries, `networkx` for graph operations and `pandas` for data manipulation. The dataset is loaded from a CSV file named `prado.csv`.

The graph `G` is initialized as an undirected graph. Nodes are added to the graph for each artwork using their URLs as unique identifiers.

Edges are added between nodes based on shared tags. A dictionary `tag_to_works` is used to map each tag to the list of artworks associated with it. The code iterates over each row in the dataset, extracting the URL and tags for each artwork. Tags are split and stripped of extra whitespace before being added to the dictionary.

2.3 Graph Visualization

To visualize the graph, I used the `nx.draw()` function from `NetworkX` along with `matplotlib.pyplot`, customizing the appearance of nodes and edges for clarity.

2.4 Basic Graph Analysis

I analyzed the graph by calculating and printing the total number of nodes and edges to understand its size, determining the node with the highest degree to find the artwork with the most connections, and identifying the number of connected components to reveal how many separate subgraphs exist within the overall graph.
-Number of nodes in the graph: 13487
-Number of edges in the graph: 129908
-Node with the highest degree: https://www.museodelprado.es/coleccion/obra-de-arte/el-nio-del-arbol/6469ab32-b795-4bc5-96b3-439fab6409e0 with a degree of 264
-Number of connected components in the graph: 8137
If the connected components are few, we can expect a very sparse network. In a sparse network, many nodes will have few or no connections. This can lead to a generally low PageRank for most nodes, as PageRank is influenced by the number of incoming and outgoing links.

2.5 PageRank Calculation

Finally, I computed PageRank scores for each artwork to rank them by their relative importance within the graph and printed the top 10 artworks with the highest PageRank scores to highlight the most influential artworks in terms of connectivity. This process allowed me to analyze which artworks are most interconnected based on shared tags, providing insights into the relationships and importance of different artworks within the dataset from the Museo del Prado. Here is the Python code used for calculating the PageRank:

```
import numpy as np

def pagerank(M, num_iterations=100, d=0.85, tol=1.0e-6):
    """
    Compute the PageRank of each node in the graph.

    Parameters:
    M (numpy array): Adjacency matrix where M[i][j] represents a
    link from node j to node i.
    num_iterations (int): Maximum number of iterations to run the
    algorithm.
    d (float): Damping factor, usually set to 0.85.
    tol (float): Tolerance for convergence.

    Returns:
    numpy array: PageRank vector
    """
    N = M.shape[1]  # Number of nodes in the graph
    v = np.random.rand(N, 1)
    v = v / np.linalg.norm(v, 1)  # Normalize initial vector
    M_hat = d * M + (1 - d) / N * np.ones((N, N))

    for _ in range(num_iterations):
        v_new = M_hat @ v
        if np.linalg.norm(v_new - v, 1) < tol:
            break
        v = v_new

    return v
```

Listing 2.2: PageRank Calculation Algorithm

2.6 Explanation

The provided Python code implements the PageRank algorithm, which is used to rank the importance of nodes in a graph. The algorithm operates on an adjacency matrix `M` where the element `M[i][j]` represents a link from node `j` to node `i`. The PageRank algorithm is an iterative method that computes the relative importance of nodes based on their link structures.

The function `pagerank` takes four parameters: the adjacency matrix `M`, the maximum number of iterations `num_iterations`, the damping factor `d`, and the convergence tolerance `tol`. The damping factor, commonly set to 0.85,

accounts for the probability that a user will randomly jump to any node rather than following links.

Initially, a random PageRank vector `v` is created and normalized. In each iteration, the algorithm updates the PageRank values using the matrix `M_hat`, which incorporates the damping factor. The iteration continues until the change in the PageRank vector is less than the tolerance `tol`, indicating that convergence has been reached.

After the iterations are complete, the final PageRank vector `v` is returned, representing the PageRank scores of the nodes in the graph.

2.6.1 Scalability Features

The `pagerank` function incorporates several features that contribute to its scalability, which are explained as follows:

- **Efficient Matrix Operations**:
  - The function uses the `@` operator for matrix multiplication, which is optimized in the NumPy library. NumPy provides high-performance computations for matrix operations, leveraging C-based implementations and optimized algorithms for large-scale matrix manipulations.
- **Memory Efficiency**:
  - NumPy arrays are designed for memory-efficient storage and manipulation of data. This efficiency is crucial for handling large adjacency matrices, ensuring that the function can work with extensive graphs without excessive memory consumption.
- **Convergence Criteria**:
  - The `tol` parameter sets a tolerance level for convergence, which allows the algorithm to terminate early when the PageRank vector stabilizes. This approach avoids unnecessary computations and reduces the number of iterations required for convergence.
- **Damping Factor**:
  - The `d` parameter, also known as the damping factor, controls the probability of following a link versus randomly jumping to any page. This factor helps stabilize the PageRank computation and ensures that the algorithm converges more effectively.
- **Sparse Matrix Handling**:
  - Although the function does not explicitly handle sparse matrices, the adjacency matrix `M` can be represented as a sparse matrix using libraries such as SciPy (`scipy.sparse`). This representation is beneficial for very large graphs, as it reduces both memory usage and computational costs.

- **Initial Vector Normalization**:
  - The initial PageRank vector is normalized with `v = v / np.linalg.norm(v, 1)` to ensure that it is a valid starting point for the PageRank calculations. This normalization helps achieve convergence more quickly.

2.6.2 Code Optimization for Large Graphs

For even better scalability, additional optimizations can be employed:

- **Sparse Matrix Representation**:
  - Convert the adjacency matrix to a sparse format to save memory and speed up computations:

```
from scipy.sparse import csr_matrix
M_sparse = csr_matrix(M)
```

    Use sparse matrix operations for the PageRank calculation:

```
M_hat = d * M_sparse + (1 - d) / N * np.ones((N, N))
v_new = M_hat.dot(v)
```

- **Adaptive Iteration Limit**:
  - Implement a dynamic iteration limit based on actual convergence progress, rather than a fixed number of iterations.

2.6.3 Summary Table of Scalability Features

Feature	Description
Efficient Matrix Operations	Uses optimized NumPy functions for matrix multiplications.
Memory Efficiency	NumPy arrays handle large datasets with minimal memory overhead.
Convergence Criteria	Uses a tolerance parameter to stop iterations when convergence is achieved.
Damping Factor	Stabilizes the PageRank computation and ensures proper convergence.
Sparse Matrix Support	For very large graphs, can be optimized with sparse matrix representations.
Initial Vector Normalization	Ensures that the initial vector is valid for PageRank computation and helps achieve convergence.

Table 2.1: Summary of scalability features in the PageRank function.

2.7 Results and Analysis

In my analysis, I obtained the following top nodes with their corresponding PageRank values:

Node ID	Artwork URL	PageRank Value
2969	https://www.museodelprado.es/coleccion/obra-de/2969	0.000280
5052	https://www.museodelprado.es/coleccion/obra-de/5052	0.000255
10811	https://www.museodelprado.es/coleccion/obra-de/10811	0.000255
573	https://www.museodelprado.es/coleccion/obra-de/573	0.000254
7430	https://www.museodelprado.es/coleccion/obra-de/7430	0.000254
2563	https://www.museodelprado.es/coleccion/obra-de/2563	0.000253
93	https://www.museodelprado.es/coleccion/obra-de/93	0.000249
1630	https://www.museodelprado.es/coleccion/obra-de/1630	0.000249
1892	https://www.museodelprado.es/coleccion/obra-de/1892	0.000247
8699	https://www.museodelprado.es/coleccion/obra-de/8699	0.000247
3663	https://www.museodelprado.es/coleccion/obra-de/3663	0.000244
3786	https://www.museodelprado.es/coleccion/obra-de/3786	0.000243
11901	https://www.museodelprado.es/coleccion/obra-de/11901	0.000243
4191	https://www.museodelprado.es/coleccion/obra-de/4191	0.000239
241	https://www.museodelprado.es/coleccion/obra-de/241	0.000238

Table 2.2: Top nodes by PageRank value with their corresponding artwork URLs.

2.7.1 Commentary

In my analysis, I obtained the following top PageRank values for the nodes in the network:

The PageRank values for the top nodes are very close to each other, ranging from approximately 0.000238 to 0.000280. This indicates that there is not a significant disparity in the importance of these nodes based on the PageRank algorithm.

- **Sparse Network**: The small differences in PageRank values suggest that the network might be sparse. In a sparse network, many nodes have few connections, which can lead to relatively uniform PageRank values among the top nodes.
- **Low PageRank Values**: The overall low PageRank values indicate that none of the nodes is significantly more important than the others in the context of this network. This result is consistent with the expectation of a sparse network where no single node dominates the importance metrics.