CS101 Link Analysis Project
Name-Navneet kaur
Entry no.-2023csb1137

Problem 1. *Choose the top leader by running a random walk on the graph with teleportation.*

Solution.

1 Introduction

PageRank is an algorithm used to measure the importance of nodes in a network, particularly in the context of web pages on the internet. Originally developed by Larry Page and Sergey Brin at Google, PageRank assigns a numerical weight to each node in a network based on the structure of links between them. The fundamental idea behind PageRank is that important nodes are likely to be linked to by other important nodes.

In this report, I have discussed how PageRank is calculated, and the Top Leader is found using a random walk with teleportation approach, implemented in Python using the NetworkX library.

2 Random Walk with Teleportation

Random walk with teleportation is a method to simulate the movement of a particle through a network, where at each step, the particle either moves to a neighbouring node or teleports to a random node with a certain probability. This approach is used in PageRank to model the behavior of a user navigating through web pages, where they may follow links or randomly jump to a different page.

3 Implementation

The provided Python code implements the random walk with teleportation algorithm to calculate PageRank for nodes in a given directed graph and then provide the Leader. Let's break down the implementation:

```
import pandas as pd
import networkx as nx
import random
import numpy as np

#Reading the csv file
data = pd.read_csv('modified_impression_network.csv')

#Creating directed graph from the csv file data
G = nx.DiGraph()
for index, row in data.iterrows():
```

```
    node = row.iloc[0]  # Assuming the first column contains the node
    neighbors = row.iloc[1:].dropna().tolist()  # Assuming the rest columns are neighbors
    G.add_node(node)  # Adding nodes to the graph
    for neighbor in neighbors: # Adding neighbors to the graph
        G.add_edge(node, neighbor)

"""
    Performing a random walk with teleportation on the given graph.

    Parameters used:
        G (networkx.DiGraph): Directed graph to perform random walk on.
        teleport_prob : Probability of teleportation (default is 0.15).
        num_steps : Number of steps in the random walk (default is 1000000).

    Returns:
        tuple: A tuple containing two lists - nodes and their random walk points.
    """
def random_walk_with_teleportation(G, teleport_prob=0.15, num_steps=1000000):
    nodes = list(G.nodes())# Extracting all nodes from the graph
    rw_points = {node: 0 for node in nodes}# Initializing random walk points for each node
    current_node = random.choice(nodes)# Choosing a random starting node
    for _ in range(num_steps):

        # Perform teleportation with a certain probability
        if random.random() <= teleport_prob:
            current_node = random.choice(nodes)# Teleporting to a random node
        else:
            neighbors = list(G.neighbors(current_node))# Getting neighbors of the current node
            if neighbors: # If neighbors exist
                current_node = random.choice(neighbors)# Moving to a random neighbor
            else:
                current_node = random.choice(nodes)# Teleporting if no neighbors exist
        rw_points[current_node] += 1# Incrementing random walk points for the current node
```

```
45
46      return nodes, rw_points
47
48  #Sorting nodes based on their random walk points in descending order
49  def nodes_sorting(nodes, points):
50      points_array = np.array(list(points.values()))  # Extract values
            from the dictionary
51      sorted_indices = np.argsort(-points_array)# Sorting indices in
           descending order of points
52      sorted_nodes = [nodes[i] for i in sorted_indices] # Sorting
           nodes based on sorted indices
53      return sorted_nodes
54
55  #Performing the functions defined above
56  nodes, rw_points = random_walk_with_teleportation(G, teleport_prob
       =0.15, num_steps=1000000)
57  sorted_nodes = nodes_sorting(nodes, rw_points)
58
59  # Printing the top 10 nodes with their random walk points
60  print("Node \t\t Random Walk Points")
61  for node in sorted_nodes[:10]:
62      print(f"{node} \t {rw_points[node]}")
63
64  #Printing the leader with max random walk points
65  leader=sorted_nodes[0]
66  print ("The Top Leader=", leader)
```

1. **Reading Data**: The code reads data from a CSV file containing information about nodes and their neighbours.

2. **Creating the Directed Graph**: Using NetworkX, a directed graph is constructed from the data where each node represents a student, and edges represent links between students.

3. **Random Walk with Teleportation Function**: The `random_walk_with_teleportation` function performs the random walk simulation on the graph. At each step, the algorithm either moves to a neighbouring node or teleports to a random node with a specified teleportation probability.

4. **Sorting Nodes by Random Walk Points**: After simulating the random walk, nodes are sorted based on their accumulated random walk points in descending order. This sorting identifies the nodes with higher random walk points.

5. **Printing Results**: The code prints the top 10 nodes along with their random walk points, providing insights into the most important nodes in the network, it identifies the node with the highest PageRank (random walk points) score as the top leader.

4 Conclusion

The provided code demonstrates how PageRank can be calculated using the random walk with teleportation algorithm. By simulating the movement of a particle through the network, the algorithm assigns importance scores to each node, helping to identify the most influential nodes in the graph. This approach is fundamental to various applications such as web search, recommendation systems, and network analysis.

□

Problem 2. *Recommend missing links using the matrix method explained in the class.*

Solution.

Introduction

Social network analysis is a critical tool in understanding relationships and interactions within a network. One common task in social network analysis is to predict missing links between nodes, which can provide insights into potential connections that have not yet been established. In this report, I present a method for predicting missing links in a social network graph using linear algebra techniques (Matrix method) and least squares regression.

Method

The method used in this report involves the following steps:

1. **Data Preparation**: The input data is assumed to be in the form of a CSV file containing information about the network, with each row representing a node and its associated neighbors.

2. **Graph Construction**: A directed graph (DiGraph) is created using the NetworkX library in Python. Nodes represent individuals, and edges represent relationships between them. The graph is constructed based on the information provided in the CSV file.

3. **Adjacency Matrix Generation**: An adjacency matrix is generated from the constructed graph using the `nx.adjacency_matrix` function. This matrix represents the relationships between nodes in the graph.

4. **Prediction of Missing Links**: The `predict_zero_values` function is employed to predict missing links in the graph. This function iterates through the adjacency matrix, identifies zero elements (indicating missing links), and predicts their values using linear algebra techniques.

5. **Linear Least Squares Regression**: Within the `predict_zero_values` function, linear least squares regression is performed to predict the value of each missing link. This involves expressing the row containing the zero element as a linear combination of the

remaining rows in the adjacency matrix. The coefficients obtained from the regression are then used to predict the value of the missing link based on the corresponding column values.

6. **Thresholding**: Predicted values exceeding a certain threshold (in this case, 0.6 as if the predicted value is less than or equal to zero then there is no edge between that nodes but as by this method the values approximately range from some -0.9 to 3 so for more accuracy I took 0.6 as the threshold) are considered significant and are added as edges to the graph.

Implementation

The provided Python code implements the aforementioned method as follows:

```
import numpy as np
import networkx as nx
import pandas as pd

def predict_zero_values(adj_matrix):
    zero_indices = np.argwhere(adj_matrix == 0)  # Finding indices of zeroes in the matrix
    if len(zero_indices) == 0:
        print("No zero found in the adjacency matrix.")
        return None

    predicted_values = []#List for storing the predicted values

    for zero_index in zero_indices:
        row, col = zero_index #Rows and columns corresponding to zeroes in the matrix
        deleted_row = adj_matrix[row]# Extracting the row containing the zero
        deleted_row = np.delete(deleted_row, col, axis=0)# Deleting the zero element from the row
        deleted_col = adj_matrix[:, col]# Extracting the column containing the zero
        deleted_col = np.delete(deleted_col, row, axis=0) # Deleting the zero element from the column

        # Deleting row and column
        adj_matrix_temp = np.delete(adj_matrix, row, axis=0)# Deleting the row from the adjacency matrix
        adj_matrix_temp = np.delete(adj_matrix_temp, col, axis=1)# Deleting the column from the adjacency matrix

        # Expressing deleted row as linear combination of remaining rows using numpy built in function (matrix method)
```

```
        '''working of the built in function used here:

        np.linalg.lstsq: This function computes the least-squares
           solution to a linear matrix equation. It is commonly used
            when you have an overdetermined system of linear
           equations, meaning there are more equations than unknowns
           .

        adj_matrix_temp.T: This is the matrix of independent
           variables , it represents the remaining rows of the
           adjacency matrix after deleting the row corresponding to
           the zero element.

        deleted_row: This is the dependent variable vector,
           representing the row containing the zero element that we
           want to express as a linear combination of the remaining
           rows.

        rcond=None: This argument specifies the cutoff for small
           singular values. When rcond is set to None, NumPy
           internally determines the threshold for determining rank
           of the coefficient matrix, which is used to solve the
           least squares problem. It essentially controls the
           numerical stability of the solution.

        [0]: The result of np.linalg.lstsq is a tuple containing
           several elements, including the solution to the least
           squares problem. By accessing element [0], we're
           extracting the solution coefficients from the tuple.
                '''
        coefficients = np.linalg.lstsq(adj_matrix_temp.T,
           deleted_row, rcond=None)[0]

        # Predicting the value of the zero in the deleted row/column
        predicted_value = np.dot(coefficients, deleted_col)

        # Storing the predicted value along with its index
        predicted_values.append((zero_index, predicted_value))

    return predicted_values


#Reading the csv file
data = pd.read_csv('modified_impression_network.csv')

#Creating directed graph from the csv file data
G = nx.DiGraph()
```

```
for index, row in data.iterrows():
    node = row.iloc[0]  # Assuming the first column contains the
       node
    neighbors = row.iloc[1:].dropna().tolist()  # Assuming the rest
       columns are neighbors
    G.add_node(node)  # Adding nodes to the graph
    for neighbor in neighbors: # Adding neighbors to the graph
        G.add_edge(node, neighbor)

adj_matrix = nx.adjacency_matrix(G).todense()# Generating the
   adjacency matrix from the graph

predicted_values = predict_zero_values(adj_matrix)# Predicting zero
   values in the adjacency matrix
missing_links = []# List to store missing links
persons = list(G.nodes())# List of persons in the graph so that i
   can get the entry no. of the students corresponding to its index
   in adjacency matrix
for zero_index, predicted_value in predicted_values:

    if predicted_value > 0.6:# Checking if predicted value exceeds a
        threshold
        i,j = zero_index# Extracting row and column indices
        missing_links.append((persons[i], persons[j]))# Storing
           missing links in the form of entry no. of the students
        G.add_edge(persons[i], persons[j])# Adding missing links to
           the graph

# Printing missing links
print("missing links")
for m in missing_links:
    print(m)
```

- It imports necessary libraries such as NumPy for numerical operations, NetworkX for graph manipulation, and Pandas for data handling.
- The `predict_zero_values` function is defined to predict missing links in the adjacency matrix.
- The CSV file containing network data is read using Pandas, and a directed graph is constructed based on the information in the file.
- The adjacency matrix of the graph is generated.
- Missing links are predicted using the `predict_zero_values` function, and significant predictions are added as edges to the graph.
- Finally, the missing links are printed for analysis.

Conclusion

The method described provides a systematic approach for predicting missing links in a social network graph. By leveraging linear algebra techniques and least squares regression, it offers insights into potential connections that can enhance our understanding of network dynamics. Further experimentation and validation on real-world datasets can validate the effectiveness of this approach in practical scenarios.

□

Problem 3. *Propose a brand new problem based on this dataset and provide a solution for the same. Be as creative as possible.*

Solution.

1 Problem : Find the Strongly Connected Components (SCC) in the Graph

The Question is to identify the strongly connected components in a given directed graph. This involves partitioning the graph into subsets of nodes where each subset forms an SCC, meaning that every node in an SCC can reach every other node within the same SCC.

2 Introduction

Strongly Connected Components (SCCs) are fundamental concepts in graph theory, particularly in the analysis of directed graphs. An SCC is a subset of nodes in a directed graph where every node is reachable from every other node within the subset. SCCs play a crucial role in understanding the structure and connectivity of directed graphs.

3 What are SCCs and How are They Used in Examining the Graph

Strongly Connected Components provide valuable insights into the structure and connectivity of directed graphs. They help in identifying clusters or communities within the graph where nodes have strong interactions or dependencies. SCCs are used in various graph analysis tasks, including:

1. Graph Visualization: SCCs can be visualized as distinct clusters within a directed graph, providing a clear representation of the graph's connectivity patterns.

2. Network Analysis: Identifying SCCs allows for the examination of interconnected subgraphs, revealing important structures and relationships within the larger graph.

3. Algorithm Design: SCCs are utilized in algorithms for tasks such as finding shortest paths, detecting cycles, and identifying central nodes in a graph.

4 Algorithm Used for Finding SCCs

The algorithm that I have used for finding strongly connected components in a directed graph is Kosaraju's algorithm. Kosaraju's algorithm is a two-pass algorithm that efficiently identifies SCCs in linear time complexity.

1. First Pass (Forward DFS): Perform a depth-first search (DFS) traversal of the graph to compute the finishing times for each node. This step assigns a finishing time to each node, with nodes finishing later having higher finishing times.

2. Second Pass (Backward DFS): Reverse the direction of all edges in the graph. Perform another DFS traversal, this time starting from nodes with the highest finishing times computed in the first pass. This step identifies the SCCs by exploring nodes in the reverse topological order of the original graph.

5 Implementation

The provided Python code implements Kosaraju's algorithm to find strongly connected components in a given directed graph. It utilizes the NetworkX library to represent and traverse the graph efficiently. After identifying SCCs, the code outputs the nodes belonging to each SCC, providing valuable insights into the graph's structure and connectivity.

```
import pandas as pd
import networkx as nx

class Graph:
    def __init__(self):
        self.graph = {} # Initializing an empty dictionary to
           represent the graph

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []# Adding node 'u' to the graph if not
               already present
        self.graph[u].append(v)# Adding edge (u, v) to the graph

    def dfs(self, v, visited, stack):
        visited.add(v)# Marking node 'v' as visited
        for neighbor in self.graph.get(v, []):
            if neighbor not in visited:
                self.dfs(neighbor, visited, stack)# Recursively
                   performing DFS on unvisited neighbors
        stack.append(v) # Adding node 'v' to the stack after
           visiting all its neighbors

    def transpose(self):
        transposed = Graph()# Creating a new instance of Graph to
           represent the transposed graph
```

```
22          for u in self.graph:
23              for v in self.graph[u]:
24                  transposed.add_edge(v, u) #Reversing the direction
                        of each edge in the original graph
25          return transposed# Returning the transposed graph
26
27      def fill_order(self, v, visited, stack):
28          visited.add(v)# Marking node 'v' as visited
29          for neighbor in self.graph.get(v, []):
30              if neighbor not in visited:
31                  self.fill_order(neighbor, visited, stack)#
                        Recursively performing DFS on unvisited neighbors
32          stack.append(v)# Adding node 'v' to the stack after visiting
                all its neighbors
33
34      def get_sccs(self):
35          stack = [] # Initializing a stack to store nodes in order of
                their finishing times
36          visited = set()# Initializing a set to store visited nodes
37          for node in self.graph:
38              if node not in visited:
39                  self.dfs(node, visited, stack)# Performing DFS on
                        all unvisited nodes
40          # Getting the transposed graph
41          transposed = self.transpose()
42          # Clearing the visited set to reuse it for the next DFS
                traversal
43          visited.clear()
44          sccs = []# Initializing a list to store strongly connected
                components
45
46          while stack:
47              v = stack.pop() # Popping a node from the stack
48              if v not in visited:
49                  scc = [] # Initializing a list to store nodes in the
                        current strongly connected component
50                  transposed.fill_order(v, visited, scc)# Performing
                        DFS on the transposed graph to get the SCC
51                  sccs.append(scc)# Adding the SCC to the list of SCCs
52
53          return sccs
54
55  #Reading the csv file
56  data = pd.read_csv('modified_impression_network.csv')
57
58  #Creating directed graph from the csv file data
59  G = nx.DiGraph()
```

```
for index, row in data.iterrows():
    node = row.iloc[0]  # Assuming the first column contains the node
    neighbors = row.iloc[1:].dropna().tolist()  # Assuming the rest columns are neighbors
    G.add_node(node)  # Adding nodes to the graph
    for neighbor in neighbors: # Adding neighbors to the graph
        G.add_edge(node, neighbor)

# Converting NetworkX graph to adjacency list
adjacency_list = {}
for edge in G.edges:
    if edge[0] not in adjacency_list:
        adjacency_list[edge[0]] = []# Adding node 'u' to the adjacency list if not already present
    adjacency_list[edge[0]].append(edge[1])# Adding edge (u, v) to the adjacency list

# Creating a Graph object and initialize it with the adjacency list
g = Graph()# Creating an instance of the Graph class
for node, neighbors in adjacency_list.items():
    for neighbor in neighbors:
        g.add_edge(node, neighbor)# Adding edges to the graph using the adjacency list

# Finding strongly connected components using Kosaraju's algorithm
sccs = g.get_sccs()# Getting the strongly connected components
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

# Getting the largest strongly connected component
largest_scc = max(sccs, key=len)

# Number of nodes in the largest strongly connected component
num_nodes_in_largest_scc = len(largest_scc)

print("Number of nodes in the largest strongly connected component:", num_nodes_in_largest_scc)
```

6 Conclusion

Finding strongly connected components in a directed graph is a crucial task in graph analysis. SCCs help in understanding the underlying structure and connectivity patterns of directed graphs. If the biggest strongly connected component (SCC) found has a large number of nodes, approximately equal to the total number of nodes in the graph, it implies that

the graph is highly connected and lacks distinct substructures. Such a scenario suggests that the graph forms a single cohesive unit where nearly every node is reachable from every other node. This can indicate a tightly interconnected network with strong relationships and dependencies between its constituents. It suggests a homogeneous structure where information or influence can flow freely across the entire network without encountering significant barriers.

□