ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Τμήμα Πληροφορικής



Εργασία Μαθήματος «Ανάλυση Κοινωνικών Δικτύων»

|  |  |
| --- | --- |
| Όνομα φοιτητή – Αρ. Μητρώου | Σμύρης Βασίλειος – ΜΠΣΠ2343 |
| Ημερομηνία παράδοσης | 17/11/2024 |

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

[1 Introduction 3](#_Toc182688805)

[2 Data loading 3](#_Toc182688806)

[3 Part I Questions 4](#_Toc182688807)

[3.1 Question 1.1 4](#_Toc182688808)

[3.2 Question 1.2 5](#_Toc182688809)

[3.3 Question 1.3 7](#_Toc182688810)

[3.4 Question 1.4 8](#_Toc182688811)

[4 Part II Questions 13](#_Toc182688812)

[4.1 Question 2.1 13](#_Toc182688813)

[4.2 Question 2.2 15](#_Toc182688814)

[5 Part III Questions 20](#_Toc182688815)

1. Introduction

The assignment was implemented using the python language. The code is included in the accompanying Jupiter notebook, along with execution results. The code can be executed for a reduced sample of the original data. Execution of the code on the full original data can take days. Below are the python libraries that were used for the assignments:

import pandas as pd

import numpy as np

import networkx as nx

import matplotlib.pyplot as plt

import json

1. Data loading

The project data are loaded into a pandas dataframe, using the read\_csv function. Labels that describe the data are assigned to the imported data columns.

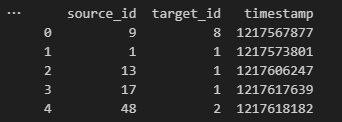
# Load the data

file\_path = 'sx-stackoverflow.txt'

df = pd.read\_csv(file\_path, sep=' ', header=None, names=['source\_id', 'target\_id', 'timestamp'])

print(df.head())

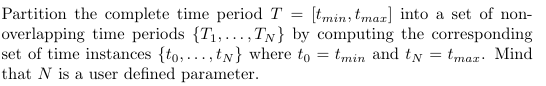
Below is a sample from the data.



1. Part I Questions

This section presents the implementations for the questions of the first part of the assignment. In this part, we split the graph of the data into N sub-graphs for N different non-overlapping time periods and we compute some centrality measures.

* 1. Question 1.1

**

In order to split the timeframe of the data into N parts, we first find the minimum and maximum timestamps. For this example, the timeframe will be split into 10 periods and the data will be subsequently split into 10 sub-graphs. We first compute how long the time periods should be, by dividing the duration of the whole timeframe by the number of time periods we want to split it into. We use this value, delta\_t, to compute N+1 time instances that will work as the lower and upper bounds for the N time periods, using the following formula:



Below is the code that implements this question:

# Find the minimum and maximum timestamps

tmin = df['timestamp'].min()

tmax = df['timestamp'].max()

# Define the number of partitions (N)

N = 10

# Calculate the time intervals

delta\_t = (tmax - tmin) / N

# Calculate the time instances {t0, t1, ..., tN}

time\_instances = [tmin + j \* delta\_t for j in range(N + 1)]

# Print the results

print(f"tmin: {tmin}, tmax: {tmax}")

print(f"Time instances (boundaries): {time\_instances}")

Below are the computed time instances:

tmin: 1217567877, tmax: 1457273428

Time instances (boundaries): [1217567877.0, 1241538432.1, 1265508987.2, 1289479542.3, 1313450097.4, 1337420652.5, 1361391207.6, 1385361762.7, 1409332317.8, 1433302872.9, 1457273428.0]

* 1. Question 1.2



We will use the networkx library’s Graph class for representing the subgraphs. The subgraph of each time period is populated with edges from the dataset that their timestamp falls into the bounds of the time period. Due to the very large size of the dataset, the computations for the next questions of the assignment take days to execute. For the quick execution of the assignment, two versions of this process were implemented. If the reduced variable is set to False, the program constructs the subgraphs including all the edges of the original dataset that they should have, which can be up to millions. If the reduced variable is set to True, the program creates the subgraphs with a random sample of size edges\_per\_reduced\_graph of the edges that should be included in the subgraph. This way, it is possible to execute the assignment code for both the full dataset and a limited sample of each time period.

reduced = True # Run the code for the whole graph or a sample

edges\_per\_reduced\_graph = 5000 # Number of samples for each time period in reduced mode

# Define a list to store the subgraphs

subgraphs = []

if not reduced:

    # Generate subgraphs for each time period

    for j in range(1, len(time\_instances)):

        t\_start = time\_instances[j-1]

        t\_end = time\_instances[j]

        # Filter the DataFrame for edges in the current time period

        if j < N:

            df\_sub = df[(df['timestamp'] >= t\_start) & (df['timestamp'] < t\_end)]

        else:

            df\_sub = df[(df['timestamp'] >= t\_start) & (df['timestamp'] <= t\_end)]

        # Create a directed graph using NetworkX

        G = nx.Graph()

        # Add edges to the graph

        for \_, row in df\_sub.iterrows():

            G.add\_edge(row['source\_id'], row['target\_id'])

        #Remove self loops

        G.remove\_edges\_from(nx.selfloop\_edges(G))

        # Store the subgraph

        subgraphs.append(G)

        print(f"Subgraph {j} created. |V| = {len(G.nodes)}, |E| = {len(G.edges)}")

if reduced:

    for j in range(1, len(time\_instances)):

        t\_start = time\_instances[j-1]

        t\_end = time\_instances[j]

        # Filter the DataFrame for edges in the current time period

        if j < N:

            df\_sub = df[(df['timestamp'] >= t\_start) & (df['timestamp'] < t\_end)]

        else:

            df\_sub = df[(df['timestamp'] >= t\_start) & (df['timestamp'] <= t\_end)]

        # If the DataFrame has more than edges\_per\_reduced\_graph rows, sample edges\_per\_reduced\_graph edges

        if len(df\_sub) > edges\_per\_reduced\_graph:

            df\_sub\_limited = df\_sub.sample(n=edges\_per\_reduced\_graph)

        else:

            df\_sub\_limited = df\_sub  # If less than edges\_per\_reduced\_graph edges, keep all edges

        # Create a graph using NetworkX

        G = nx.Graph()

        # Add edges to the graph

        for \_, row in df\_sub\_limited.iterrows():

            G.add\_edge(row['source\_id'], row['target\_id'])

        #Remove self loops

        G.remove\_edges\_from(nx.selfloop\_edges(G))

        # Store the subgraph

        subgraphs.append(G)

        print(f"Subgraph {j} created. |V| = {len(G.nodes)}, |E| = {len(G.edges)}")

The following formula determines if a node should be included in a subgraph:



For all time periods only the lower bound is inclusive. For the Nth time period, both bounds are inclusive.

* 1. Question 1.3



The following code plots a graph that depicts the time evolution of the quantities of edges and vertices through the time periods.

subgraphs\_node\_num = []

subgraphs\_edge\_num = []

time\_periods = [f"T\_{i}" for i in range(1, N+1)]

for subgraph in subgraphs:

    subgraphs\_node\_num.append(len(subgraph.nodes()))

    subgraphs\_edge\_num.append(len(subgraph.edges()))

# Plotting

plt.figure(figsize=(12, 6))

# Plot number of nodes

plt.subplot(1, 2, 1)

plt.plot(time\_periods, subgraphs\_node\_num, marker='o', linestyle='-', color='b')

plt.xlabel('Time Periods')

plt.ylabel('Number of Nodes')

plt.title('Number of Nodes over Time')

plt.xticks(rotation=45)

# Plot number of edges

plt.subplot(1, 2, 2)

plt.plot(time\_periods, subgraphs\_edge\_num, marker='o', linestyle='-', color='r')

plt.xlabel('Time Periods')

plt.ylabel('Number of Edges')

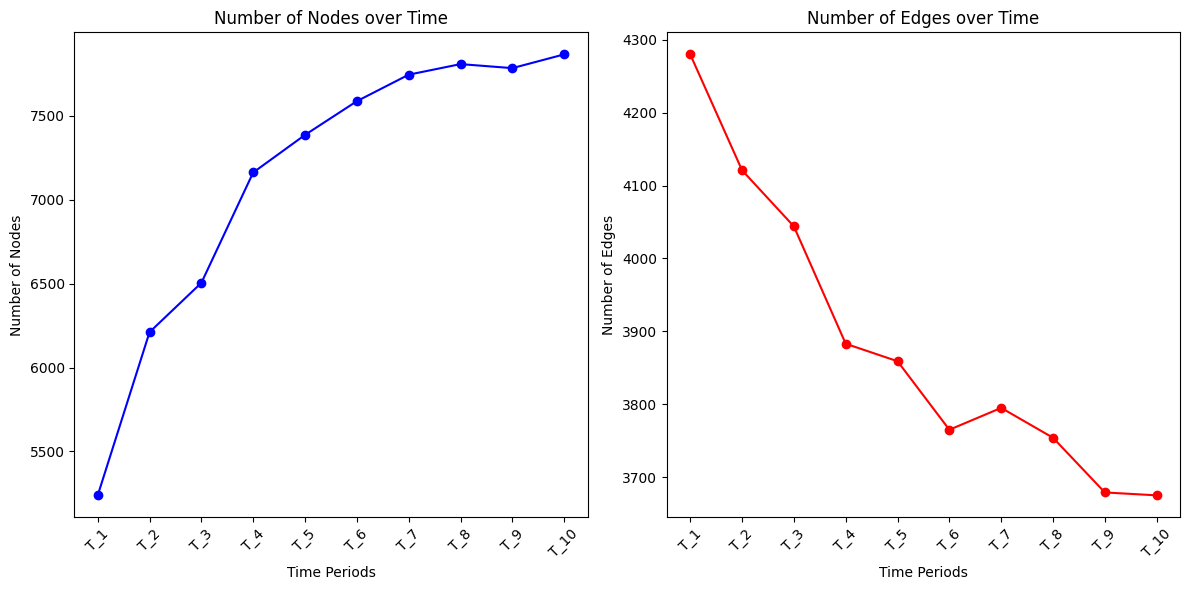
plt.title('Number of Edges over Time')

plt.xticks(rotation=45)

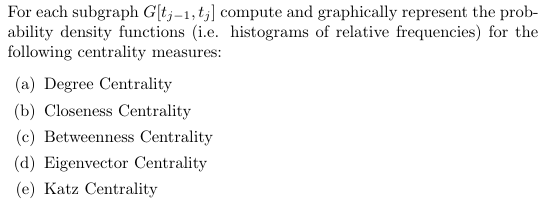
plt.tight\_layout()

plt.show()

Below are the graphs for a random reduced dataset:



* 1. Question 1.4



For this section, we compute five centrality measures for each of the N subgraphs, using functions provided by the networkx library, and we plot histograms that show the distribution of the centrality values among the vertices of each subgraph. Below are the computed measures and some of their characteristics.

**Betweenness Centrality**

* **Definition**: Measures the number of shortest paths that pass through a node.
* **Interpretation**: Nodes with high betweenness centrality act as intermediaries and are crucial for the distribution of information within the network.
* **Disadvantage**: It can yield very high values for nodes that connect two large communities.Degree Centrality

**Closeness Centrality**

* **Definition**: Measures the inverse of the average distance from a node to all other nodes in the network.
* **Interpretation**: A node with high closeness centrality has shorter overall distances to all other nodes, enabling faster communication throughout the network.
* **Disadvantage**: It is difficult to compute in large and complex networks.

**Degree Centrality**

* **Definition**: This measure is based on the number of connections (edges) a node has. For undirected networks, it is simply the number of a node's neighbors.
* **Interpretation**: A high degree of centrality indicates that the node is well-connected and may be considered important or central within the network.
* **Disadvantage**: It does not take into account the topology of the entire network.

**Eigenvector Centrality**

* **Definition**: Measures the influence of a node in the network by considering its connections to other important nodes.
* **Interpretation**: A node with high eigenvector centrality is connected to many other nodes that also have high centrality.
* **Disadvantage**: It can be less intuitive and more complex to compute.

**Katz Centrality**:

* **Definition**: Katz Centrality is an advanced measure based on the concept of Eigenvector Centrality but with an important distinction. Katz Centrality measures the influence of a node in the network, taking into account the number of neighboring nodes and the paths that lead to it, with a diminishing weight for each additional step away from the node. It uses a parameter α to adjust the significance of indirect connections and a parameter β that adds a constant value to each node.
* **Interpretation**: Katz Centrality gives weight to both direct and indirect connections, assigning greater importance to closer connections and reducing the impact of distant ones.
* **Application**: It is used to identify nodes that are significant not just because of their direct connections but also due to their broader relationships within the network. This makes it useful in situations where influence spreads through multiple paths.

The following code, implements the centrality measure computation. It computes all five measures for each time period’s subgraph and saves the results to files. The results are also saved in memory for use when plotting the frequency histograms. Exception handling is used for the eigenvector and katz centralities, as sometimes they do not converge for some subgraphs.

centrality\_measures = []

index = 1

for G in subgraphs:

    # Compute centrality measures

    degree\_centrality = nx.degree\_centrality(G)

    with open(f"centralities/degree{index}.json", "w") as f:

        json.dump({str(key): value for key, value in degree\_centrality.items()}, f)

    closeness\_centrality = nx.closeness\_centrality(G)

    with open(f"centralities/closeness{index}.json", "w") as f:

        json.dump({str(key): value for key, value in closeness\_centrality.items()}, f)

    betweenness\_centrality = nx.betweenness\_centrality(G)

    with open(f"centralities/betweenness{index}.json", "w") as f:

        json.dump({str(key): value for key, value in betweenness\_centrality.items()}, f)

    # Try computing eigenvector centrality, handle exceptions

    try:

        eigenvector\_centrality = nx.eigenvector\_centrality(G)

        with open(f"centralities/eigenvector{index}.json", "w") as f:

            json.dump({str(key): value for key, value in eigenvector\_centrality.items()}, f)

    except nx.PowerIterationFailedConvergence as e:

        print(f"Warning: Eigenvector centrality did not converge for subgraph {index}.")

        eigenvector\_centrality = {node: None for node in G.nodes()}

    # Try computing Katz centrality, handle exceptions

    try:

        katz\_centrality = nx.katz\_centrality(G)

        with open(f"centralities/katz{index}.json", "w") as f:

            json.dump({str(key): value for key, value in katz\_centrality.items()}, f)

    except nx.PowerIterationFailedConvergence as e:

        print(f"Warning: Katz centrality did not converge for subgraph {index}.")

        katz\_centrality = {node: None for node in G.nodes()}

    # Store centrality measures

    centrality\_measures.append({

        'degree': list(degree\_centrality.values()),

        'closeness': list(closeness\_centrality.values()),

        'betweenness': list(betweenness\_centrality.values()),

        'eigenvector': list(eigenvector\_centrality.values()),

        'katz': list(katz\_centrality.values())

    })

    print(f"Centralities computed for T\_{index}")

    index += 1

After computing the centralities, frequency histograms are plotted for each measure and time period, showing the distribution of centrality values among the vectors of each graph.

# Plot histograms for centrality measures

measures = ['degree', 'closeness', 'betweenness', 'eigenvector', 'katz']

for measure in measures:

    num\_subgraphs = len(centrality\_measures)

    # Dynamically adjust the figure size based on number of subgraphs

    plt.figure(figsize=(14, 10 \* num\_subgraphs))  # Set height based on subgraphs

    # Loop over subgraphs

    for i, measure\_values in enumerate([cm[measure] for cm in centrality\_measures]):

        # Filter out None values

        measure\_values\_filtered = [val for val in measure\_values if val is not None]

        # Only plot if there are valid values to plot

        if len(measure\_values\_filtered) > 0:

            plt.subplot(num\_subgraphs, 1, i+1)  # Create a subplot for each subgraph

            plt.hist(measure\_values\_filtered, bins=20, density=True, alpha=0.7)

            plt.xlabel(f'{measure.capitalize()} Centrality')

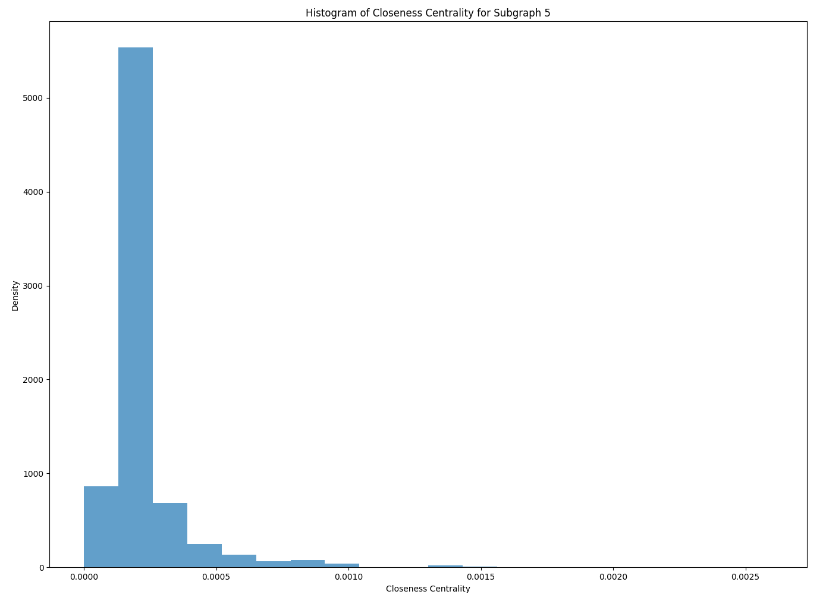
            plt.ylabel('Density')

            plt.title(f'Histogram of {measure.capitalize()} Centrality for Subgraph {i+1}')

    plt.tight\_layout()

    plt.show()

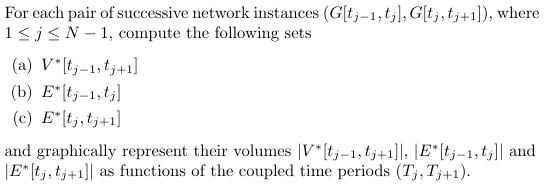
Below is an example of such a histogram. All histograms are included in the accompanying Jupiter notebook.



1. Part II Questions

This section presents the implementations for the questions of the second part of the assignment.

* 1. Question 2.1



The following code, compute the required vortex and edge sets. The subgraphs are accessed in pairs of two consecutive time periods. In V\_star we keep the vertices that are common in the graphs of the the two time periods. In E\_start\_prev and E\_star\_next we keep the edges of the two graphs that form between vertices that belong in V\_star. We then add these sets to lists, that can then be used to iterate over the time period pairs to process the data.

# Prepare lists for persistent nodes and edges

persistent\_nodes = []

persistent\_edges\_prev = []

persistent\_edges\_next = []

persistent\_nodes\_num = []

persistent\_edges\_prev\_num = []

persistent\_edges\_next\_num = []

# Compute persistent nodes and edges

for j in range(len(subgraphs) - 1):

    # Get nodes and edges for periods [T\_{j-1}, T\_j] and [T\_j, T\_{j+1}]

    V\_prev = set(subgraphs[j].nodes())

    V\_next = set(subgraphs[j+1].nodes())

    V\_star = V\_prev.intersection(V\_next)

    E\_prev = set(subgraphs[j].edges())

    E\_next = set(subgraphs[j+1].edges())

    E\_star\_prev = {(u, v) for (u, v) in E\_prev if u in V\_star and v in V\_star}

    E\_star\_next = {(u, v) for (u, v) in E\_next if u in V\_star and v in V\_star}

    persistent\_nodes.append(V\_star)

    persistent\_edges\_prev.append(E\_star\_prev)

    persistent\_edges\_next.append(E\_star\_next)

    persistent\_nodes\_num.append(len(V\_star))

    persistent\_edges\_prev\_num.append(len(E\_star\_prev))

    persistent\_edges\_next\_num.append(len(E\_star\_next))

# Prepare time\_periods for plotting

time\_periods = [f"T\_{i + 1}, T\_{i + 2}" for i in range(len(subgraphs) - 1)]

# Plot persistent nodes and edges

plt.figure(figsize=(12, 6))

# Plot number of persistent nodes

plt.subplot(1, 2, 1)

plt.plot(time\_periods, persistent\_nodes\_num, marker='o', linestyle='-', color='g')

plt.xlabel('Time Periods')

plt.ylabel('Number of Persistent Nodes')

plt.title('Number of Persistent Nodes over Time')

plt.xticks(rotation=45)

# Plot number of persistent edges

plt.subplot(1, 2, 2)

plt.plot(time\_periods, persistent\_edges\_prev\_num, marker='o', linestyle='-', color='m', label='Edges [t\_{j-1}, t\_j]')

plt.plot(time\_periods, persistent\_edges\_next\_num, marker='o', linestyle='-', color='c', label='Edges [t\_j, t\_{j+1}]')

plt.xlabel('Time Periods')

plt.ylabel('Number of Persistent Edges')

plt.title('Number of Persistent Edges over Time')

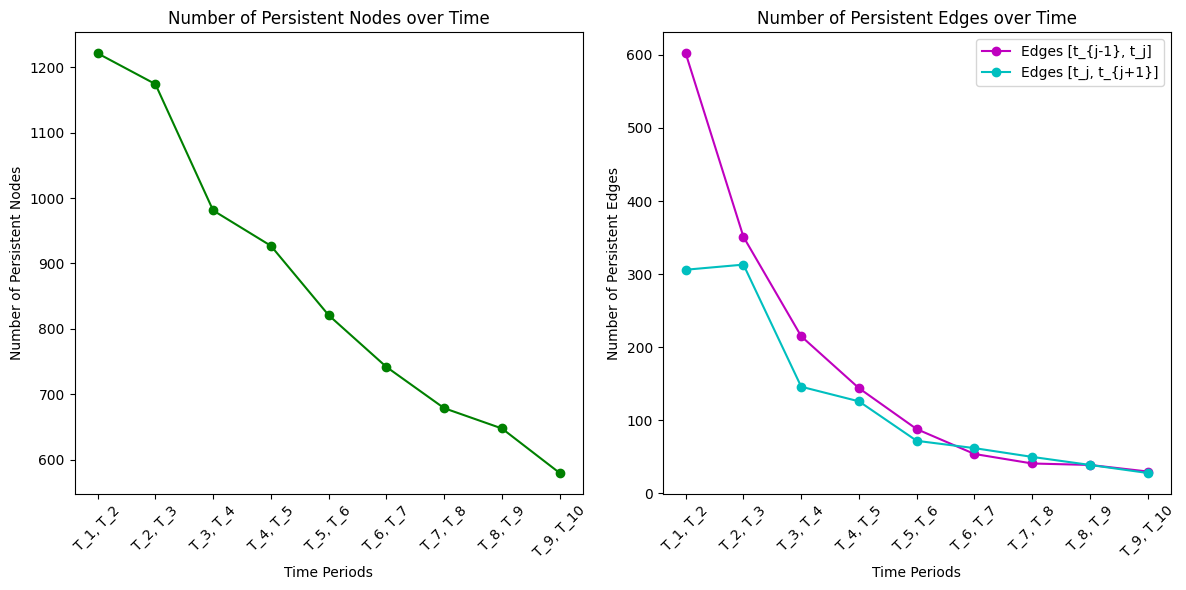
plt.xticks(rotation=45)

plt.legend()

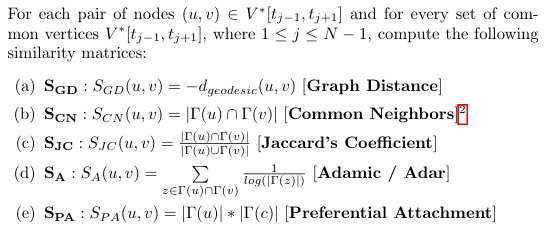
plt.tight\_layout()

plt.show()

We then collect all the vortex and edge counts and plot a graph that shows how these values fluctuate over time. We can see in the below example, that for the given sample data, the number of peristing vertices between two time periods drops. The number of edges between common vertices that two paired time period has, also drops.



* 1. Question 2.2



To compute the similarity matrices for the common nodes of each pair of consecutive time periods, we use the relevant functions from the network library, wich receive a graph and a list of node pairs for which we want to find their similarity. Below are the similarity measures we are computing and some of their characteristics:

**Graph Distance**

* **Definition**: The distance in a graph is the minimum number of edges that must be traversed to travel from one node to another. It is also known as "shortest path distance" or "geodesic distance."
* **Application**: Used to calculate the shortest path between two nodes in a network.

**Common Neighbors**

* **Definition**: Common neighbors of two nodes are the nodes that are adjacent to both nodes simultaneously. In other words, they are the shared connections between two nodes in a graph.
* **Application**: Used to assess the likelihood of two nodes connecting in the future and to measure the similarity between nodes.

**Jaccard’s Coefficient**

* **Definition**: Jaccard's coefficient measures the similarity between two sets. It is defined as the ratio of the size of the intersection of the two sets to the size of their union.
* **Application**: Used for estimating the similarity between nodes in social networks or comparing the similarity between different data sets.

**Adamic/Adar**

* **Definition**: The Adamic/Adar measure is a metric that evaluates the similarity between two nodes in a network by considering their common neighbors, with greater weight given to common neighbors with fewer connections.
* **Application**: Used to predict the likelihood of new connections forming between nodes, emphasizing rarer common neighbors.

**Preferential Attachment**

* **Definition**: This measure explains how nodes with many existing connections (high degree nodes) are more likely to acquire new connections. It follows the principle of "the rich get richer."
* **Application**: Used to model the growth of networks such as social networks, where popular nodes continue to become more popular over time.

For the Graph Distance measure, we divide 1 by the computed distance between two vectors, so the measure will have a lesser value the greater the distance between the two nodes will be. This is done because the greater the distance between two vectors, the less similar they are. For each set of common vectors, their similarity is computed for both the graph of the graph of the previous time period and the next.

# Helper function to convert matrix keys to strings for JSON compatibility

def stringify\_keys(matrix):

    return {str(key): value for key, value in matrix.items()}

# Placeholder for similarity matrices

similarities = {

    "SGD" : {

        "prev": {},

        "next": {}

    },

    "SCN" : {

        "prev": {},

        "next": {}

    },

    "SJC" : {

        "prev": {},

        "next": {}

    },

    "SA" : {

        "prev": {},

        "next": {}

    },

    "SPA" : {

        "prev": {},

        "next": {}

    }

}

# Iterate through the graph pairs for each time period

for j in range(len(persistent\_nodes)):

    V\_star = persistent\_nodes[j]

    G\_prev = nx.Graph()

    G\_prev.add\_edges\_from(persistent\_edges\_prev[j])

    G\_next = nx.Graph()

    G\_next.add\_edges\_from(persistent\_edges\_next[j])

    # Initialize similarity matrices for each measure

    similarities["SGD"]["prev"][j] = {}

    similarities["SCN"]["prev"][j] = {}

    similarities["SJC"]["prev"][j] = {}

    similarities["SA"]["prev"][j] = {}

    similarities["SPA"]["prev"][j] = {}

    similarities["SGD"]["next"][j] = {}

    similarities["SCN"]["next"][j] = {}

    similarities["SJC"]["next"][j] = {}

    similarities["SA"]["next"][j] = {}

    similarities["SPA"]["next"][j] = {}

    # Compute the similarity for every pair of persistent nodes (u, v)

    for u in V\_star:

        for v in V\_star:

            if u != v:

                # SGD: Shortest path distance

                try:

                    similarities["SGD"]["prev"][j][(u, v)] = 1 / nx.shortest\_path\_length(G\_prev, u, v)

                except:

                    pass

                try:

                    similarities["SGD"]["next"][j][(u, v)] = 1 / nx.shortest\_path\_length(G\_prev, u, v)

                except:

                    pass

                # SCN: Common neighbors squared

                try:

                    similarities["SCN"]["prev"][j][(u, v)] = len(list(nx.common\_neighbors(G\_prev, u, v)))

                except:

                    pass

                try:

                    similarities["SCN"]["next"][j][(u, v)] = len(list(nx.common\_neighbors(G\_prev, u, v)))

                except:

                    pass

                #SJC: Jaccards Coe cient

                try:

                    for u, v, p in nx.jaccard\_coefficient(G\_prev, [(u, v)]):

                        similarities["SJC"]["prev"][j][(u, v)] = p

                except:

                    pass

                try:

                    for u, v, p in nx.jaccard\_coefficient(G\_next, [(u, v)]):

                        similarities["SJC"]["next"][j][(u, v)] = p

                except:

                    pass

                # SA: Adamic/Adar score

                try:

                    for u, v, p in nx.adamic\_adar\_index(G\_prev, [(u, v)]):

                        similarities["SA"]["prev"][j][(u, v)] = p

                except:

                    pass

                try:

                    for u, v, p in nx.adamic\_adar\_index(G\_next, [(u, v)]):

                        similarities["SA"]["next"][j][(u, v)] = p

                except:

                    pass

                # SPA: Preferential attachment (degree product)

                try:

                    for u, v, p in nx.preferential\_attachment(G\_prev, [(u, v)]):

                        similarities["SPA"]["prev"][j][(u, v)] = p

                except:

                    pass

                try:

                    for u, v, p in nx.preferential\_attachment(G\_next, [(u, v)]):

                        similarities["SPA"]["next"][j][(u, v)] = p

                except:

                    pass

    print("Similarities computed for", j)

with open("similarities/SGD\_matrix\_prev.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SGD"]["prev"][j]) for j in similarities["SGD"]["prev"]}, f)

with open("similarities/SCN\_matrix\_prev.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SCN"]["prev"][j]) for j in similarities["SCN"]["prev"]}, f)

with open("similarities/SJC\_matrix\_prev.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SJC"]["prev"][j]) for j in similarities["SJC"]["prev"]}, f)

with open("similarities/SA\_matrix\_prev.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SA"]["prev"][j]) for j in similarities["SA"]["prev"]}, f)

with open("similarities/SPA\_matrix\_prev.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SPA"]["prev"][j]) for j in similarities["SPA"]["prev"]}, f)

with open("similarities/SGD\_matrix\_next.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SGD"]["next"][j]) for j in similarities["SGD"]["next"]}, f)

with open("similarities/SCN\_matrix\_next.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SCN"]["next"][j]) for j in similarities["SCN"]["next"]}, f)

with open("similarities/SJC\_matrix\_next.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SJC"]["next"][j]) for j in similarities["SJC"]["next"]}, f)

with open("similarities/SA\_matrix\_next.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SA"]["next"][j]) for j in similarities["SA"]["next"]}, f)

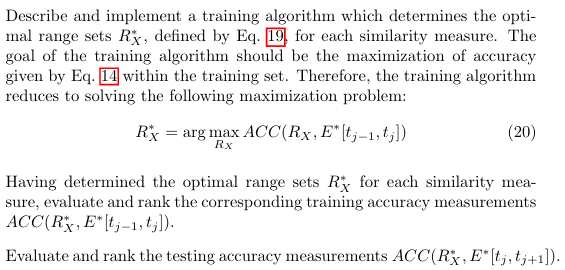
with open("similarities/SPA\_matrix\_next.json", "w") as f:

    json.dump({j: stringify\_keys(similarities["SPA"]["next"][j]) for j in similarities["SPA"]["next"]}, f)

print("Matrices saved to files.")

The above code produces lists of similarity matrices for each similarity measure, for both time periods of a pair. The results are also saved to files for viewing.

1. Part III Questions



# Example setup for training on each similarity measure for each period

optimal\_ranges = {}

training\_accuracies = {}

testing\_accuracies = {}

for measure, similarity\_matrices in similarities.items():

    optimal\_ranges[measure] = {}

    training\_accuracies[measure] = {}

    testing\_accuracies[measure] = {}

    for period in range(N - 1):

        training\_edges = persistent\_edges\_prev[period]  # Training set

        actual\_edges = persistent\_edges\_next[period]  # Testing set

        all\_possible\_edges = {(u, v) for u in persistent\_nodes[period] for v in persistent\_nodes[period] if u != v}

        # Optimize range for current similarity measure

        optimal\_range, accuracy = optimize\_range(similarity\_matrices['prev'][period], similarity\_matrices['next'][period], training\_edges, actual\_edges, all\_possible\_edges)

        optimal\_ranges[measure][period] = {'range': optimal\_range, 'accuracy': accuracy}

        training\_accuracies[measure][period] = {'accuracy': accuracy}

        print(f"Optimal range for {measure} in period {period}: {optimal\_range} with accuracy {accuracy}")

        # Evaluate testing accuracy using the optimal range found

        pred\_edges\_test = {edge for edge, score in similarity\_matrices['next'][period].items() if optimal\_range[0][0] <= score <= optimal\_range[0][1]}

        test\_accuracy = calculate\_accuracy(pred\_edges\_test, actual\_edges, all\_possible\_edges)

        testing\_accuracies[measure][period] = {'accuracy': test\_accuracy}

        print(f"Testing accuracy for {measure} in period {period}: {test\_accuracy}")

The following function is used to find the optimal range of smiliarity that accurately predicts the most edges. It is applied for two consecutive time periods, with the data of the previous period being used for training and the data of the next period being used for testing. The function receives as input the similarity matrices for a single metric for a pair of time periods and their edges. It finds the lowest and highest similarity values in the training data, and tries to find a range of similarities, that most accurately predict which connections will be made in the next time period.

# Function to determine optimal range R\*X for a given similarity measure

def optimize\_range(similarity\_matrix\_prev, similarity\_matrix\_next, training\_edges, actual\_edges, all\_possible\_edges):

    best\_accuracy = 0

    best\_ranges = []

    training\_similarities = {t: similarity\_matrix\_prev.get(t, None) for t in training\_edges if similarity\_matrix\_prev.get(t, None) is not None}

    if not training\_similarities:

        return best\_ranges, best\_accuracy

    min\_score, max\_score = min(training\_similarities.values()), max(training\_similarities.values())

    for lower\_bound in np.linspace(min\_score, max\_score, num=20):

        for upper\_bound in np.linspace(lower\_bound, max\_score, num=20):

            pred\_edges = {edge for edge, score in similarity\_matrix\_next.items() if lower\_bound <= score <= upper\_bound}

            accuracy = calculate\_accuracy(pred\_edges, actual\_edges, all\_possible\_edges)

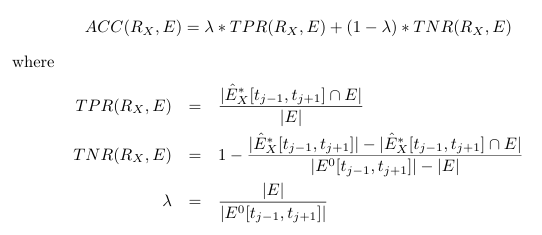
            if accuracy > best\_accuracy:

                best\_accuracy = accuracy

                best\_ranges = [(lower\_bound, upper\_bound)]

    return best\_ranges, best\_accuracy

The following function, calculates the accuracy of a prediction based on the given formula from the assignment:



def calculate\_accuracy(pred\_edges, actual\_edges, all\_possible\_edges):

    tp = len(pred\_edges & actual\_edges)

    tpr = tp / len(actual\_edges) if actual\_edges else 0

    tn = len(all\_possible\_edges - pred\_edges - actual\_edges)

    tnr = tn / (len(all\_possible\_edges) - len(actual\_edges)) if len(all\_possible\_edges) > len(actual\_edges) else 0

    lamda = len(actual\_edges) / len(all\_possible\_edges) if all\_possible\_edges else 0

    return lamda \* tpr + (1 - lamda) \* tnr

We find the maximum accuracy for each measure for training and testing and we rank the measures for their accuracy in predicting new edges.

max\_training\_accuracies = {}

for measure, periods in training\_accuracies.items():

    max\_accuracy = max(periods.values(), key=lambda x: x['accuracy'])  # Find the period with max accuracy

    max\_training\_accuracies[measure] = max\_accuracy['accuracy']

max\_testing\_accuracies = {}

for measure, periods in testing\_accuracies.items():

    max\_accuracy = max(periods.values(), key=lambda x: x['accuracy'])  # Find the period with max accuracy

    max\_testing\_accuracies[measure] = max\_accuracy['accuracy']

# Sort the measures by accuracy in descending order

sorted\_max\_training\_accuracies = sorted(max\_training\_accuracies.items(), key=lambda x: x[1], reverse=True)

sorted\_max\_testing\_accuracies = sorted(max\_testing\_accuracies.items(), key=lambda x: x[1], reverse=True)

# Print the rankings

print("Ranking of Measures by Maximum Training Accuracy:")

for rank, (measure, accuracy) in enumerate(sorted\_max\_training\_accuracies, start=1):

    print(f"Rank {rank}: {measure} with accuracy {accuracy:.6f}")

print("\nRanking of Measures by Maximum Testing Accuracy:")

for rank, (measure, accuracy) in enumerate(sorted\_max\_testing\_accuracies, start=1):

    print(f"Rank {rank}: {measure} with accuracy {accuracy:.6f}")

Below is an example output of the rating:

Ranking of Measures by Maximum Training Accuracy:

Rank 1: SPA with accuracy 0.999946

Rank 2: SCN with accuracy 0.999822

Rank 3: SJC with accuracy 0.999822

Rank 4: SA with accuracy 0.999822

Rank 5: SGD with accuracy 0.999787

Ranking of Measures by Maximum Testing Accuracy:

Rank 1: SPA with accuracy 0.999946

Rank 2: SCN with accuracy 0.999822

Rank 3: SJC with accuracy 0.999822

Rank 4: SA with accuracy 0.999822

Rank 5: SGD with accuracy 0.999787