Time-Aware Network Centrality Measures & Link Prediction

Social Network Analysis Assignment

Ghada NAIT SAID **Ahmed NAIT SAID**

Erasmus Students from University of BATNA 2, Algeria

University of Piraeus – Spring 2022

Under Supervision of Dr. Dionisios N. Sotiropoulos

Abstract

This assignment is a hands-on algorithmic manipulation of the Stack Overflow Temporal Network. The network is in the form of timestamped edges list and is row-wise stored in the file “sx-stackoverflow.txt”, as consecutive triplets of the form (source_id target_id timestamp). Each edge’s timestamp indicates the exact time instance where the edge was created.

Due to the large volume of the data, and the small processing power, it would take a long time to process the whole data frame presented, in that regard, we only used a small portion of the data for computing.

Program Structure

The program is written purely in "python" since it has a very well-established set of libraries for graph manipulation and processing as well as graphical visualization. Using libraries such as:

- **Pandas:** a flexible and easy to use open-source data analysis and manipulation tool.
- **NetworkX:** for some graph related manipulations and processing.
- **Numpy:** very flexible array computing.
- **Matplotlib:** a comprehensive library for creating static, animated, and interactive visualizations.

The program is divided into different python file, in which each file corresponds to a question in the assignment. All these separate files' classes and methods, are used interchangeably and utilized all together in the main python file for the actual and full execution of the program.

In the next section, the functionalities of each py file will be explained, and the last section will present example runs of the program with two different time partitions *"N"*. The program explanation will be back and forth between the main.py file and the separate components PiQj.py (i: part no. & j: question no.). Each component is explained before its use in the main and highlighted with a gray background.

N.B. After almost every fraction of the program, whenever we're done using a set of variables, we delete them using **del**, then the **gc.collect()** garbage collector is called to collect the deleted (no longer referenced) variables. This step was added for space optimization purposes since the data files can have a large size.

The Program *main.py*

All needed modules are imported.

```
import pandas as pd
import networkx as nx
import P1Q1
import P1Q2
import P1Q3_1
import P1Q3_2
import P1Q4
import P2Q1
import P2Q2
import P3Q2
import P3Q3
import P3Extra
import gc
import os
```

Part I

- The file "sx-stackoverflow.txt" is read as a dataframe *"net"* and its columns are labeled as (src, dst, tstamp), then only the 1st 1K edges are extracted (for CPU and memory constraints). Secondly, *"net"* rows are sorted in an ascending chronological order which may trigger indices shuffling, so we reset the latter and drop the old indices.

```
# PART I -------------------------------------------------------------------------

# Reading the edges' set file -----------------------------------------------------
net = pd.read_csv("a.txt", sep=" ", header=None)
net = net.iloc[:1001, :]  # First 1K Edges
net.columns = ["src", "dst", "tstamp"]  # Labeling columns for easier data
                                          manipulation
net = net.sort_values(by=['tstamp'])  # Order data chronologically
net = net.reset_index(drop=True)  # Resetting indices after reordering
```

- Take the *"N"*: number of time partitions, as a user input.

  That input is checked since it has to be an int > 0, to avoid program execution failure.

```
# Getting N -----------------------------------------------------------------------
N = input('Please enter \"N\" the number of the network\'s time partitions: ')
while True:  # Input conflict (In case user inputs an invalid value)
    if not N.isnumeric() or int(N) == 0:  # Input != number or Input == 0
        N = input('Please enter a valid N: ')  # Read input again
    else:
        break
N = int(N)
```

P1Q1.py

Method: *TPartition (net, N): t_min*, *t_max*, *t* – the list of time instances $[t_0, \dots, t_N]$

Behavior: takes the timestamped & sorted edges dataframe, so *"t_min"* & *"t_max"* are respectively the first & last rows' tstamps. Then it divides the complete time period T = $[t_{\min}, t_{\max}]$ into N successive time intervals. *"Dt"* & *"dt"* correspond to Δt & δt respectively.

```
import csv


def TPartition(net, N):
    t_min = net.tstamp[0]  # instead  of min() that takes time we use [0], as we
                             already sorted the df by tstamp
    t_max = net.tstamp[len(net.index)-1]  # Same as min, we just get the last cell
    Dt = t_max - t_min  # The whole time length
    dt = Dt / N  # The interval length
    t = [t_min + i * dt for i in range(N + 1)]  # List of time intervals
    with open("t.csv", 'w') as f:  # Save time intervals in csv file
```

```
        csv.writer(f).writerow(t)
    return t_min, t_max, t
```

• **Q1: Partition of the complete time period T.**

P1Q1.TPartition(net, N) is called, and **t** is returned.

```
t_min, t_max, t = P1Q1.TPartition(net, N)
print("\nt_min: %s\nt_max:" % t_min, t_max)
print('t:', t, '\n')
```

P1Q2.py

Class: *SubGraph(N)*

```
import pandas as pd
import P1Q3_1
import csv
import os
import gc


class SubGraph:

    def __init__(self, N):
        self.N = N
```

Method 1: *SubgraphEdgeLists (self, net, t)*

Behavior: when j < N; for each time interval [tj-1, tj), it keeps the index of the first subgraph edge and iterates through the rows whose tstamp is within that time interval while keeping track if the last row in each iteration. Once that time interval edges end, it saves them in a “Edges-i.txt” file. When j = N, the rest of the edges automatically belong to the last subgraph.

```
def SubgraphEdgeLists(self, net, t):
    i = 0  # For iterating through df rows
    j = 1  # For iterating through "t" time list
    while j < self.N:  #
        i1 = i  # The index of the subgraph's first row
        while t[j - 1] <= net.tstamp[i] < t[j]:  # When 1 <= j < N
            i2 = i  # The index of the subgraph's last row
            i += 1  # We increment "i" as long as condition true, to keep all
                      values within the interval
        fname = "Edges-" + str(j) + ".txt"
        with open(fname, "w") as f:  # Save edge list in txt file
            net.iloc[i1:i2 + 1, 0:2].to_csv(f, sep=" ", header=None, index=False)
            # Writing only src & dst columns
```

```
        j += 1
    #  Once j == N, All the rest of the edges belong to the last subgraph
    i1 = i
    i2 = len(net.index) - 1
    fname = "Edges-" + str(j) + ".txt"
    with open(fname, "w") as f:
        net.iloc[i1:i2 + 1, 0:2].to_csv(f, sep=" ", header=None, index=False)
```

Method 2: *AdjLists (self)*

Behavior: creates adjacency lists from edge lists, by iterating through the "Edges-i.txt" files. Whenever a non-empty edge list is found, it's read in *"G"* and then gets its vertices and edges list with P1Q3_1.GetVertices(G)/.GetEdges(G) respectively (will be explained in P1Q3), creates a dictionary with those vertices as the keys and empty list values. Afterwards, for each key (vertex) it puts for the value a list of all dst vertices where that key is a src. At the end it saves the AdjList in a "AdjList-i.csv" file

```
def AdjLists(self):
    for i in range(1, self.N + 1):
        fname = "Edges-" + str(i) + ".txt"
        if os.stat(fname).st_size != 0:  # File is not empty
            G = pd.read_csv(fname, sep=' ', header=None)
            V = sorted(list(P1Q3_1.GetVertices(G)))  # Sorted list of vertices
            AdjList = dict.fromkeys(V, [])  # Dictionary with vertices as keys
            E = P1Q3_1.GetEdges(G)  # df of edges
            for key in AdjList:
                AdjList[key] = E[E.src == key].dst.tolist()  # Adding to each
                                         dictionary key values of dst vertices
            with open("AdjList-" + str(i) + ".csv", "w") as f:  # Save Adjacency
                                                                 list as csv file

                w = csv.writer(f)
                A = [[key] + value for key, value in AdjList.items()]
                w.writerows(A)
            del G, V, E, AdjList, A
            gc.collect()
```

- **Q2: an appropriate representation for each subgraph G = $[t_{j-1}, t_j]$ of the network.**

SubGraph(N) object *"Sub"* is instantiated then the methods *SubgraphEdgeLists(net, t)* and *AdjLists()* are called, for the creation of the subgraphs **edge lists** and **adjacency lists**.

```
# Subgraphs' Edge list creation -----------------------------------------------------
Sub = P1Q2.SubGraph(N)
Sub.SubgraphEdgeLists(net, t)
del net, t
gc.collect()
```

```
# Subgraphs' Adjacency Lists Creation ---------------------------------------------
Sub.AdjLists()
```

P1Q3_1.py

Method 1: *GetEdges(G): E* – df of edges

Behavior: takes an edges df, drops the duplicate edges (in case 2 vertices connect more than once per T), this step causes the deletion of some rows which requires the indices reset.

```
import csv


def GetEdges(G):
    E = G.drop_duplicates()  # Dropping duplicate edges, in case two vertices
                               connect more than once per T
    E = E.reset_index(drop=True)  # Resetting indices after dropping duplicate
                                    edges
    return E
```

Method 2: *SaveEdges(E)*

Behavior: Saves edge list in a “EdgesSet-i.csv” file.

```
def SaveEdges(E, i):
    with open("EdgesSet-" + str(i) + ".csv", "w") as f:  # Save edge list in csv
                                                             file
        w = csv.writer(f)
        for j in range(len(E.index)):  # writing edges in rows one by one
            w.writerow(E.iloc[j, :])
```

Method 3: *GetVertices(G): V* – set of vertices

Behavior: takes the edges list, and labels its columns as “src” & “dst”; after which it updates an empty set with the afore labeled columns, where duplicates are dropped automatically.

```
def GetVertices(G):
    G.columns = ["src", "dst"]
    V = set()  # Creating empty set for vertices
    V.update(G.src.tolist(), G.dst.tolist())
    # Updating set with vertices from src & dst where duplicates are automatically
      dropped
    return V
```

Method 4: *SaveVertices(E)*

Behavior: Saves vertices set in a “VerticesesSet-i.csv” file.

```
def SaveVertices(V, i):
    with open("VerticesSet-" + str(i) + ".csv", 'w') as f:  # Vertices set in csv
                                                               file
        csv.writer(f).writerow(V)
```

Method 5: *nVandnE(N): nV, nE* – Lists of vertices and edges sets cardinalities for each time interval

Behavior: empty lists for cardinalities are created. Empty edge list = empty V and E sets; |V|=|E|=0. Non-Empty edge list are read into *"G"* df, GetVertices/Edges(G) are called to get V and E sets and their cardinalities are calculated. Each time cardinalities are calculated, they're appended to nV and nE lists.

```
def nVandnE(N):
    nV = list()  # To get the number of vertices in every time interval
    nE = list()  # To get the number of edges in every time interval
    for i in range(1, N + 1):
        fname = "Edges-" + str(i) + ".txt"
        if os.stat(fname).st_size == 0:  # In case some time intervals have empty
                                           Subgraphs
            nE.append(0)  # The number of both vertices & edges is 0
            nV.append(0)
            open("VerticesSet-" + str(i) + ".csv", 'w').close()
            open("EdgesSet-" + str(i) + ".csv", 'w').close()
        else:
            G = pd.read_csv(fname, sep=' ', header=None)
            E = GetEdges(G)
            SaveEdges(E, i)
            V = GetVertices(G)
            SaveVertices(V, i)
            nE.append(len(E.index))
            nV.append(len(V))
            del G, E, V
            gc.collect()
    return nV, nE
```

P1Q3_2.py

Method: *GraphVandE(nE, nV)*

Behavior: plots Time evolution of |V[tj-1,tj]| and |E[tj-1,tj]| and saves the figure as a png file (Fig 1. 1/Fig 2. 1).

```
from matplotlib import pyplot as plt


def GraphVandE(nE, nV):  # Line graphs of |E| and |V|
    plt.rcParams['font.family'] = ['Times New Roman', 'serif']
    plt.subplot()  # Creating one subplot
    plt.plot(nV, 'c.-', label='|V[tj-1,tj]|')  # For creating graph of No. Vertices
    plt.plot(nE, 'm.-', label='|E[tj-1,tj]|')  # For creating graph of No. Edges
```

```
    plt.title('Time evolution of |V[tj-1,tj]| and |E[tj-1,tj]|', fontsize=12)
    plt.xlabel('Time intervals')
    plt.ylabel('Set cardinality')
    plt.legend(fontsize=9)  # To display plot labels
    plt.savefig('Time_evolution_of_nV_&_nE.png')
    plt.show()
    plt.close()
```

- **Q3: A graph depicting the time evolution of the quantities |V[tj-1,tj]| and |E[tj-1,tj]|.**

P1Q3_1.nVandnE(N) gets lists nV and nE that are plotted using *P1Q3_2.GraphVandE(nE, nV)*

```
# |V| and |E| time evolution Graph -----------------------------------------------
nV, nE = P1Q3_1.nVandnE(N)
P1Q3_2.GraphVandE(nE, nV)
print("-Plotting graph of |V[tj-1,tj]| and |E[tj-1,tj]| Time evolution finished-")
del nV, nE
gc.collect()
```

P1Q4.py

Method: *Cent(G, i)*

Behavior: All centrality measures are calculated using the network module and saved into files csv files. For each time interval 5 centrality measures relative frequency histograms are plotted in one figure and saved as a png file. Relative frequency is calculated by weighting each frequency with 1 / total sum of frequencies (Fig 1. 2/Fig 2. 2).

```
from matplotlib import pyplot as plt
import networkx as nx
import numpy as np
import csv
import gc


def Cent(G, i):

    fig, axs = plt.subplots(2, 3)  # Create 6 subplots
    fig.delaxes(axs[1, 2])  # Delete the last subplot as we only need 5 for the 5
                              centrality measures
    fig.suptitle('Centrality Measures G' + str(i))

    # Degree Centrality
    dc = nx.degree_centrality(G)
    dc = dict(sorted(dc.items(), key=lambda item: item[1]))  # Sort by value
    x = list(dc.values())
    axs[0, 0].hist(x, color="#1c2e4a", edgecolor='black', weights=np.ones_like(x) /
                   len(x))
    # First subplot in [0,0] is filled with relative frequency of degree centrality
      values
    # All values are weighted with 1/len(x) to get the relative frequency i.e. each
```

```
      x[i] is plotted x[i]/len(x)
    axs[0, 0].set_title('Degree centrality', fontsize=10)
    axs[0, 0].set_ylabel('Relative Frequency', fontsize=8)
    w = csv.writer(open("Degree_centrality_G" + str(i) + ".csv", "w"))
    for key, val in dc.items():  # Saving centrality values in csv file
        w.writerow([key, val])
    del dc  # We delete pointer to dc values for space optimization
    gc.collect()  # For the actual deletion of the values
    # Same goes for the rest of the centrality measures

    # Closeness Centrality
    cc = nx.closeness_centrality(G)
    cc = dict(sorted(cc.items(), key=lambda item: item[1]))  # Sort by value
    x = list(cc.values())
    axs[0, 1].hist(x, color="#8b008b", edgecolor='black', weights=np.ones_like(x) /
                   len(x))
    # plt.savefig('Closeness_centrality_' + str(i) + '.png', format="PNG")
    axs[0, 1].set_title('Closeness centrality', fontsize=10)
    w = csv.writer(open("Closeness_centrality_G" + str(i) + ".csv", "w"))
    for key, val in cc.items():
        w.writerow([key, val])
    del cc
    gc.collect()

    # Betweenness Centrality
    bc = nx.betweenness_centrality(G)
    bc = dict(sorted(bc.items(), key=lambda item: item[1]))  # Sort by value
    x = list(bc.values())
    axs[0, 2].hist(x, color="#8B8000", edgecolor='black', weights=np.ones_like(x) /
                   len(x))
    axs[0, 2].set_title('Betweenness centrality', fontsize=10)
    w = csv.writer(open("Betweenness_centrality_G" + str(i) + ".csv", "w"))
    for key, val in bc.items():
        w.writerow([key, val])
    del bc
    gc.collect()

    # Eigenvector Centrality
    ec = nx.eigenvector_centrality(G, 1000)
    ec = dict(sorted(ec.items(), key=lambda item: item[1]))  # Sort by value
    x = list(ec.values())
    axs[1, 0].hist(x, color="#d2691e", edgecolor='black', weights=np.ones_like(x) /
                   len(x))
    axs[1, 0].set_title('Eigenvector centrality', fontsize=10)
    axs[1, 0].set_ylabel('Relative Frequency', fontsize=8)
    w = csv.writer(open("Eigenvector_centrality_G" + str(i) + ".csv", "w"))
    for key, val in ec.items():
        w.writerow([key, val])
    del ec
    gc.collect()

    # Katz Centrality
    kc = nx.katz_centrality_numpy(G)
    kc = dict(sorted(kc.items(), key=lambda item: item[1]))  # Sort by value
    x = list(kc.values())
    axs[1, 1].hist(x, color="#006400", edgecolor='black', weights=np.ones_like(x) /
```

```
                len(x))
    axs[1, 1].set_title('Katz centrality', fontsize=10)
    w = csv.writer(open("Katz_centrality_G" + str(i) + ".csv", "w"))
    for key, val in kc.items():
        w.writerow([key, val])
    del kc
    gc.collect()

    plt.tight_layout()
    plt.savefig('Centrality_Measures_G' + str(i) + '.png')  # Saving plot as png
                                                              file
    # plt.show()  # Displaying plot
    plt.close()  # Closing plot
```

- **Q4: Computation and histograms of relative frequencies of each subgraph G[tj-1,tj] centrality measures (Degree/ Closeness/ Betweenness/ Eigenvector/ Katz Centralities).**

  For each non empty graph we read the edge list and apply *P1Q4.Cent(G, i)*

```
# Centrality Measures -------------------------------------------------------------
for i in range(1, N + 1):
    fname = "Edges-" + str(i) + ".txt"
    if os.stat(fname).st_size != 0:
        G = nx.read_edgelist(fname)  # Reading the graph from previously created
                                       edge list
        P1Q4.Cent(G, i)
        del G
        gc.collect()
print("---Plotting Centrality histograms finished---")
```

Part II

P2Q1.py

Class: *SubGraph(N)*

Constructor: Empty lists are initialized Vstar, Estar1 and Estar2 for respectively storing |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[t,tj+1]| for each 1 < j < N-1.

V*[tj-1,tj+1] represents the common vertices in the period tj-1 to tj+1.

E*[tj-1,tj] and E*[tj,tj+1] are the edges that form between the consistent vertices in the two successive periods tj-1 to tj and tj to tj+1.

The goal of this class is to create a set of persistent vertices between two successive time intervals (common nodes between two successive subgraphs), and the edges that those persistent vertices take part of from each subgraph, then graph the volume of these sets.

```
class VstarEstar:
    def __init__(self, N):
        self.N = N
        self.nVstar: list = []
        self.nEstar1: list = []
        self.nEstar2: list = []
```

Method: *VandEstar(self)*

Behavior: A loop goes through the *"N"* VerticesSets files created in the previous question; first if the file is empty, an empty Vstar file is created, otherwise it would create a file with the common nodes which is basically the intersection of 2 sets of nodes. Same thing applies to the set of edges, it creates 2 Edge lists that have their pairs available in the common vertices set (Estar[j-1,j] & Estar[j,j+1]) for each Vstar file one for the previous period and another for the next one. Each iteration, Sets are created and their cardinalities are kept in the object attributes Vstar, Estar1 and Estar2. Every resulting set will be saved in in a csv file, which can be used later whenever needed.

```
def VandEstar(self):
    for j in range(1, self.N):
        if os.stat('VerticesSet-%s.csv' % j).st_size == 0 or
os.stat('VerticesSet-%s.csv' % (j + 1)).st_size == 0:
            # In case some time intervals have empty Subgraphs
            self.nVstar.append(0)  # The number of both vertices & edges is 0
            self.nEstar1.append(0)
            self.nEstar2.append(0)
            open("Vstar-" + str(j) + ".csv", 'w').close()  # Create empty files
            open("Estar[j-1,j]-" + str(j) + ".csv", 'w').close()
            open("Estar[j,j+1]-" + str(j) + ".csv", 'w').close()
        else:
            # Creation of V*[tj-1,tj+1]
            Vprev = pd.read_csv('VerticesSet-%s.csv' % j)  # V[tj-1,tj]
            Vnext = pd.read_csv('VerticesSet-%s.csv' % (j + 1))  # V[tj,tj+1]
            Vstar = [i for i in Vprev if i in Vnext]  # V*[tj-1,tj+1] =
                                          intersect(V[tj-1,tj], V[tj,tj+1])
            del Vprev, Vnext
            gc.collect()
            if len(Vstar) == 0:  # If |V*| == 0 there will be no edges
                self.nVstar.append(0)  # The number of both vertices & edges is 0
                self.nEstar1.append(0)
                self.nEstar2.append(0)
                open("Vstar-" + str(j) + ".csv", 'w').close()  # Create empty files
                open("Estar[j-1,j]-" + str(j) + ".csv", 'w').close()
                open("Estar[j,j+1]-" + str(j) + ".csv", 'w').close()
                continue  # Moves to next iteration
            self.nVstar.append(len(Vstar))  # |V*[tj-1,tj+1]|
            with open("Vstar-" + str(j) + ".csv", 'w') as f:  # Save V* Set in csv
                                                                 file
                csv.writer(f).writerow(Vstar)

            # Creation of E*[tj-1,tj]
```

```
            Eprev = pd.read_csv('EdgesSet-%s.csv' % j, header=None)
            Eprev.columns = ["u", "v"]
            Vstar = [int(x) for x in Vstar]
            # Keep persistent edges whose vertices are part of V*
            Estar = [Eprev.iloc[i] for i in range(len(Eprev.index)) if
                     Eprev.u.iloc[i] in Vstar and Eprev.v.iloc[i] in Vstar]
            del Eprev
            gc.collect()
            self.nEstar1.append(len(Estar))  # |E*[tj-1,tj]|
            with open("Estar[j-1,j]-" + str(j) + ".csv", 'w') as f:  # Save
                                                    Previous E* Set in csv file
                for i in range(len(Estar)):
                    csv.writer(f, delimiter=' ').writerow(Estar[i])
            del Estar
            gc.collect()

            # Creation of E*[tj,tj+1]
            Enext = pd.read_csv('EdgesSet-%s.csv' % (j + 1), header=None)
            Enext.columns = ["u", "v"]
            Vstar = [int(x) for x in Vstar]
            Estar = [Enext.iloc[i] for i in range(len(Enext.index)) if
                     Enext.u.iloc[i] in Vstar and Enext.v.iloc[i] in Vstar]
            del Enext, Vstar
            gc.collect()
            self.nEstar2.append(len(Estar))
            with open("Estar[j,j+1]-" + str(j) + ".csv", 'w') as f:
                for i in range(len(Estar)):
                    csv.writer(f, delimiter=' ').writerow(Estar[i])
            del Estar
            gc.collect()
```

Method: *GraphVandEstar(self)*

Behavior: creation of a graph presenting the volumes of |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[t,tj+1]| computed previously (nVstar, nEstar1 & nEstar2) , and saves the figure as a PNG file (Fig 1. 3/Fig 2. 3).

```
def GraphVandEstar(self):
    plt.subplot()  # Creating one subplot
    plt.plot(self.nVstar, 'm.-', label='|V*[tj-1,tj+1]|')
    plt.plot(self.nEstar1, 'c.-', label='|E*[tj-1,tj]|')
    plt.plot(self.nEstar2, 'y.-', label='|E*[tj,tj+1]|')
    plt.legend(fontsize=9)  # To display plot labels
    plt.title('Time evolution of |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[tj,tj+1]|',
              fontsize=12)
    plt.xlabel('Time intervals')
    plt.ylabel('Set cardinality')
    # plt.xticks(np.arange(0, self.N-1, 1), np.arange(1, self.N, 1)) # Setting x
      ticks positions 0-(N-1) & labels 1-N
    plt.savefig('Time_evolution_of_nVstar_&_nEstar.png')
    # plt.show()
    plt.close()
    del self.nVstar, self.nEstar1, self.nEstar2
    gc.collect()
```

- **Q1: For each pair of successive network instances (G[tj−1, tj], G[tj , tj+1]), where 1 ≤ j ≤ N − 1, we compute: (a) V∗[tj−1, tj+1, (b) E∗[tj−1, tj] and (c) E∗[tj , tj+1].**

VstarEstar(N) object *"VEstar"* is instantiated then the methods *VandEstar()* and *GraphVandEstar()* are called, for the creation of the sets and their graphical representation.

```
# V* and E* sets -------------------------------------------------------------------
VEstar = P2Q1.VstarEstar(N)
VEstar.VandEstar()

# |V*| and |E*| time evolution Graph -----------------------------------------------
VEstar.GraphVandEstar()
print("---Plotting graph of |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[tj,tj+1]| Time
evolution finished---")
```

P2Q1.py

Method: *Similarities(N)*

Behavior: a loop going through all the E*[j-1,j] created and passing them through the similarity methods in this file, making sure that if there are some empty files we just create empty matrices for them. Else, a loop will call [GD, CN, JC, A, PA] methods for the creation of similarity matrices.

```
def Similarities(N):
    for j in range(1, N):
        if os.stat('Vstar-%s.csv' % j).st_size == 0:  # In case some time intervals
                                                        have empty Subgraphs
            for Sim in ['Sgd', 'Scn', 'Sjc', 'Sa', 'Spa']:  # Create empty
                                                              similarity matrices
                open(Sim + "_Estar[j-1,j]-" + str(j) + ".csv", 'w').close()
        else:
            Vstar = list(pd.read_csv('Vstar-%s.csv' % j))
            G = nx.read_edgelist('Estar[j-1,j]-' + str(j) + '.csv')
            G.add_nodes_from(Vstar)
            for Sim in [GD, CN, JC, A, PA]:
                Sim(j, G, Vstar)
```

All centrality measures are calculated using the methods of the network module.

The Similarity matrices are represented using pandas dataframe *"df"* which allowed us to index the rows and columns with the vertices' IDs. Those matrices are all saved in csv files.

Method: *GD(j, G, Vstar)*

Behavior: uses the method *floyd_warshall()* which is based on Floyd's algorithm, it's appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra's algorithm fails. It creates a matrix with all pair vertices' geodesic distances, that result is then mapped to the similarity matrix.

```
def GD(j, G, Vstar):
    Sgd = nx.floyd_warshall_numpy(G)  # returns a numpy matrix with all shortest
                                        paths
    Sgd = pd.DataFrame(Sgd, index=Vstar, columns=Vstar)  # The matrix put into df
                                                            with Vstar as indices & columns
    Sgd.to_csv('Sgd_Estar[j-1,j]-%s.csv' % j)
```

Method: *CN(j, G, Vstar)*

Behavior: using the *common_neighbors()* method, we can identify the common neighbors' list of a pair of vertices, then the number of these neighbors is counted and added to the similarity matrix.

```
def CN(j, G, Vstar):
    Scn = pd.DataFrame(0, index=Vstar, columns=Vstar)  # Initializing the Scn with
                                                          zeros & Vstar as indices & columns
    for u in Vstar:
        for v in Vstar:
            Scn.loc[u, v] = len(list(nx.common_neighbors(G, u, v)))  # The number
                                                                        of common  neighbors
    Scn.to_csv('Scn_Estar[j-1,j]-%s.csv' % j)
```

Method: *JC(j, G, Vstar)*

Behavior: the *jaccard_coefficient()* method returns a list of tuples of vertices pairs with their jaccard coefficient, as edge between u and v will provide same coefficient regardless of its direction, the jc score is scored in the matrix fin both (u,v) and (v,u) cells.

```
def JC(j, G, Vstar):
    Sjc = pd.DataFrame(0, index=Vstar, columns=Vstar)  # Initializing the Scj with
                                                          zeros & Vstar as indices & columns
    jc = nx.jaccard_coefficient(G)  # Returns an iterator of tuples (u, v, c): u &
                                      v are nodes & c is their jc
    for u, v, c in jc:
        Sjc.loc[u, v] = c
        Sjc.loc[v, u] = c  # The neighbors for u and v will be the same regardless
                             of the direction of the edge
    Sjc.to_csv('Sjc_Estar[j-1,j]-%s.csv' % j)
```

Method: *A(j, G, Vstar)*

Behavior: same as the *JC* method goes for this one as it calculates the Adamic Adar score using the *adamic_adar_index()*, we take the edge and score tuple and store the data it in the matrix.

```
def A(j, G, Vstar):
    Sa = pd.DataFrame(0, index=Vstar, columns=Vstar)  # Initializing the Scj with
                                                         zeros & Vstar as indices & columns
    AA = nx.adamic_adar_index(G)  # Returns an iterator of tuples (u, v, a): u & v
                                    are nodes & a is their aa
    for u, v, p in AA:
        Sa.loc[u, v] = p
        Sa.loc[v, u] = p  # The AA score for u and v will be the same regardless of
                            the direction of the edge
    Sa.to_csv('Sa_Estar[j-1,j]-%s.csv' % j)
```

Method: PA

Behavior: same idea as *JC* and *AA*.

```
def PA(j, G, Vstar):
    Spa = pd.DataFrame(0, index=Vstar, columns=Vstar)  # Initializing the Scj with
                                                          zeros & Vstar as indices & columns
    pa = list(nx.preferential_attachment(G))  # Returns an iterator of tuples (u,
                                                v, p): u & v are nodes & p is their pa
    for u, v, p in pa:
        Spa.loc[u, v] = p
        Spa.loc[v, u] = p  # The PA for u and v will be the same regardless of the
                             direction of the edge
    Spa.to_csv('Spa_Estar[j-1,j]-%s.csv' % j)
```

- **Q2: For each pair of nodes (u, v) ∈ V*[tj−1, tj+1] and for every set of common vertices V∗[tj−1,tj+1], where 1 ≤ j ≤ N − 1, we compute the similarity matrices: [Graph Distance] [Common Neighbors] [Jaccard's Coefficient] [Adamic / Adar] [Preferential Attachment]**

  Simply the *Similarities(N)* method is called.

```
# Similarity Matrices --------------------------------------------------------------
P2Q2.Similarities(N)
```

Part III

P3Q1.py

Class: *ACCFunc(N, x, ESet)*

Constructor: Initializes *"N"*, *"x"*: the similarity matrix name and "ESet": the E* set name we want to calculate the ACC for (Either E*[j-1,j] or E*[j,j+1]).

```
class AccFunc:
    def __init__(self, N, x, ESet):
        self.N = N
        self.x = x
        self.ESet = ESet
        self.Eps = sys.float_info.epsilon
```

Method: *PrvSimVal(self): R* – set of all similarity values in an ascending order

Behavior: A looping function that loops over each time interval similarity matrix and picks up the unique values (using *Numpy.unique()*). When all values are collected, they're sorted ascendingly (for easier range optimization in the next step of the program).

```
def PrvSimVal(self):  # Estar[j-1,j] Similarity values for training purposes
    R = set()
    for i in range(1, self.N):
        if os.stat('Estar[j-1,j]-%s.csv' % i).st_size != 0:  # In case some time
                                                  intervals have empty Subgraphs
            Sx = pd.read_csv(self.x + '_Estar[j-1,j]-%s.csv' % i, index_col=0)
            R.update(np.unique(Sx.to_numpy()).tolist())  # Getting all the possible
                                                         similarity values of Si
    R = sorted(list(R))  # Sorting all possible Similarity values of all time
                          intervals
    del Sx
    gc.collect()
    return R
```

Method: *ACC(self, Rx): np.mean(ACC)* – The mean value of the ACC values list

Behavior: for each *"x"* similarity matrix, an accuracy list that will store the accuracies of the partitioned data set is created. The calculations are done in a loop that passes through every E* partition and calculate the accuracy through the given formula (*eq 14* in the assignment) in the given range *"Rx"*. **Note** that we added an Epsilon value in the TNR formula, so we avoid a division by 0.

Since we have many time partitions and thus the result is N-1 accuracy values, the mean value of these values is considered the ACC value of the whole set.

```
def ACC(self, Rx):
    Acc = []
    for i in range(1, self.N):
        G = nx.read_edgelist(self.ESet + '-%s.csv' % i)
        if not nx.is_empty(G):
            Vstar = list(pd.read_csv('Vstar-%s.csv' % i))
            Sx = pd.read_csv(self.x + '_Estar[j-1,j]-%s.csv' % i, index_col=0)
            Sx.index = Sx.index.map(str)  # Since indices are read as ints we
                                            convert them back to str for 'loc' use
            E = G.edges()
            # |E0| = |V* x V*| eq(10)
            lE0 = len(Vstar) ** 2  # The number of all possible permutations
                                    P(V*,2)
            # |^Ex*[tj-1,tj+1]| = |{edge in E0 if Sx(edge) in Rx}| eq(13)
            '''Ex = set()
            for u in Vstar:
                for v in Vstar:
                    if u < v and Rx[0] < Sx.loc[u, v] < Rx[-1]:
                        Ex.add((u, v))'''  # These lines are resumed in the next
                                             line list comprehension
            Ex = [(u, v) for u in Vstar for v in Vstar if u < v and Rx[0] <=
                  Sx.loc[u, v] <= Rx[-1]]
            EXEx = [e for e in E if e in Ex]  # Intersection(E, Ex) for the
                                                calculation of TPR & TNR
```

```
            # Some lengths for the calculation of TPR & TNR - calculate them once
              for all
            lE, lEXEx = len(E), len(EXEx)
            # TPR(Rx, E) eq(15)
            TPR = lEXEx / lE
            # TNR(Rx, E) eq(16)
            TNR = 1 - ((len(Ex) - lEXEx) + self.Eps) / ((lE0 - lE) + self.Eps)
            # Lambda eq(17)
            L = lE / lE0
            # ACC Calculation
            Acc.append(L * TPR + (1 - L) * TNR)
            del G, E, Ex, EXEx, Vstar, Sx
            gc.collect()
    return np.mean(Acc)
```

P3Q2.py

Method: *TrainACC(N): ACCListTrain* – the list of each similarity accuracy (ACC (Rx*; E*[tj-1, tj])).

RList – the list of each similarity optimal range (Rx*)

Behavior: the accuracy is calculated using the *ACC(Rx)* method using the E*[tj-1, tj] sets created for training the model. At first, ACC is sent the whole range which is assumed the best. After that the ACC is calculated using reduced intervals from the original Rx (first reduced one item from left and then one item from right), if a better accuracy value is found, it becomes the best ACC until we end up with a range that can't be reduce any longer, or the value of the ACC isn't changing anymore. The same applies for all similarity matrices, till we end up with a training ACCList for all similarities along with the optimal ranges list. Once again, we save these results in csv files.

```
def TrainACC(N):
    RList = {}
    ACCListTrain = {}
    for x in ['Sgd', 'Scn', 'Sjc', 'Sa', 'Spa']:
        ESet = 'Estar[j-1,j]'
        AcObj = P3Q1.AccFunc(N, x, ESet)
        R = AcObj.PrvSimVal()
        Rl, Rr = IdL, IdR = 0, len(R) - 1  # Min and Max values indices of R
        # IdLeft/Right are used for iterating through all the possible R (Range)
          list intervals
        # Rleft/right are used to keep indices of interval for best ACC
        ACC_best = AcObj.ACC((R[IdL], R[IdR]))  # We start assuming the whole range
                                                  gives the best ACC
        # then we reduce that range if we get a better ACC, till we reach an
          interval with the best ACC
        ACC_best1 = ACC_best
        while IdL < IdR - 1:
            ACC = AcObj.ACC((R[IdL + 1], R[IdR]))  # Calculate ACC with reduced
                                                     range one step from left
            if ACC > ACC_best:  # If calculated better than previous best ACC
                ACC_best = ACC  # Update best ACC
```

```
                Rl = IdL + 1  # Reduce "best" range one step from left
            ACC = AcObj.ACC((R[IdL], R[IdR - 1]))  # Calculate ACC with reduced
                                                     range one step from right
            if ACC > ACC_best:  # If calculated better than previous best ACC
                ACC_best = ACC  # Update best ACC
                Rr = IdR - 1  # Reduce "best" range one step from right
            ACC_best2 = ACC_best  # present best ACC
            if ACC_best1 == ACC_best2:  # If previous best ACC == present best ACC
                break  # To stop the while loop once the best accuracy isn't
                         changing anymore
            else:
                ACC_best1 = ACC_best2
            IdL += 1  # Reduce range one step from left
            IdR -= 1  # Reduce range one step from right
        # y = 'R' + x  # Rx* name for each Similarity Measurement
        # exec("y = (R[RLeft], R[RRight])")  # Applying the value to the name
        RList[x] = (R[Rl], R[Rr])
        ACCListTrain[x] = ACC_best
    ACCListTrain = {k: v for k, v in sorted(ACCListTrain.items(), key=lambda item:
                    item[1], reverse=True)}
    with open("Rx.csv", "w") as f, open("TrainACC.csv", "w") as t:  # Save Rx* and
                                                                       TrainACC as csv file
        w1 = csv.writer(f)
        w2 = csv.writer(t)
        w1.writerow(['Similarity', 'RMin', 'RMax'])
        w2.writerow(['Similarity', 'Accuracy'])
        A = [[key] + [value[0], value[1]] for key, value in RList.items()]
        B = [[key, value] for key, value in ACCListTrain.items()]
        w1.writerows(A)
        w2.writerows(B)
    return ACCListTrain, RList
```

- **Q1: The training algorithm for the maximization of the ACC to get the best optimal range Rx*.**

  P3Q2.TrainACC(N) is called and the Training accuracies and optimal Rx* ranges are calculated.

```
# Optimal Range Sets Rx* -----------------------------------------------------------
print('\n-- Optimal Range Sets Rx* --')
ACCListTrain, RList = P3Q2.TrainACC(N)
print(RList)
```

- **Q2: Evaluating and ranking ACC (Rx*; E*[tj-1, tj]).**

```
# ACC for Training Graphs ----------------------------------------------------------
print('\n---- Training Accuracy ---- Ranked from highest to lowest')
print(ACCListTrain)
```

P3Q3.py

Method: *TestACC(N, RList): ACCListTest* – the list of each similarity accuracy (ACC (Rx*; E*[t, tj+1])).

Behavior: the accuracy is calculated using the *ACC(Rx*)* method using the E*[tj-1, tj] sets created for training the model. Note that the testing model's ACC is calculated directly using an already determined (by the training model) optimal range Rx*. After that the result are simply saved in a CSV file.

```
def TestACC(N, RList):
    ACCListTest = {}
    for x in ['Sgd', 'Scn', 'Sjc', 'Sa', 'Spa']:
        ESet = 'Estar[j,j+1]'
        AcObj = P3Q1.AccFunc(N, x, ESet)
        ACC = AcObj.ACC(RList[x])  # Simply calculate ACC with given range
        ACCListTest[x] = ACC
    ACCListTest = {k: v for k, v in sorted(ACCListTest.items(), key=lambda item:
item[1], reverse=True)}
    with open("TestACC.csv", "w") as f:  # Save TestACC as csv file
        w = csv.writer(f)
        w.writerow(['Similarity', 'Accuracy'])
        A = [[key, value] for key, value in ACCListTest.items()]
        w.writerows(A)
    return ACCListTest
```

- **Q2: Evaluating and ranking ACC (Rx*; E*[tj, tj+1]).**

P3Q3.TestACC(N) is called and the Testing accuracies are calculated in the optimal Rx* ranges.

```
# ACC for Testing Graphs ------------------------------------------------------------
print('\n----- Testing Accuracy ----- Ranked from highest to lowest')
ACCListTest = P3Q3.TestACC(N, RList)
print(ACCListTest)
```

P3Extra.py

Method: *HistACC(Train, Test)*

Behavior: this method plots the Histograms that we use to compare the accuracy values of both the training and testing sets after which they're saved in a PNG file (Fig 1. 4/Fig 2. 4).

```
def HistACC(Train, Test):
    X = ['GD', 'CN', 'JC', 'A', 'PA']
    Tr = [Train[x] for x in ['Sgd', 'Scn', 'Sjc', 'Sa', 'Spa']]
    Ts = [Test[x] for x in ['Sgd', 'Scn', 'Sjc', 'Sa', 'Spa']]
    x_axis = np.arange(len(X))
    fig, ax = plt.subplots()
    plt.title("Training & Testing Accuracy", fontsize=10)
    for s in ['top', 'bottom', 'left', 'right']:  # Remove axes splines
        ax.spines[s].set_visible(False)
    ax.xaxis.set_ticks_position('none')  # Remove x, y Ticks
    ax.yaxis.set_ticks_position('none')
    ax.grid(b=True, color='grey', linestyle='-.', linewidth=0.7, alpha=0.2)  # Add
                                                                  x, y gridlines
    plt.bar(x_axis - 0.1, Tr, width=0.2, label='Training ACC')
```

```
    plt.bar(x_axis + 0.1, Ts, width=0.2, label='Testing ACC')
    plt.xticks(x_axis, X)
    plt.xlabel('Similarity Measures')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.savefig('Training & Testing Accuracy.png')  # Saving plot as png file
    plt.show()
    plt.close()
```

Method: *Hist(ACC, Str)*

Behavior: it produces the same graph but for one ranked accuracy result (testing or training) and displays them in a histogram as well. We also save the graph in a PNG file (Fig 1. 5 /Fig 2. 5, Fig 1. 6/Fig 2. 6).

```
def Hist(ACC, Str):
    X = list(ACC.keys())
    T = list(ACC.values())
    fig, ax = plt.subplots()
    ax.set_title(Str + "ing Accuracy", fontsize=10)
    for s in ['top', 'bottom', 'left', 'right']:  # Remove axes splines
        ax.spines[s].set_visible(False)
    ax.xaxis.set_ticks_position('none')  # Remove x, y Ticks
    ax.yaxis.set_ticks_position('none')
    ax.grid(b=True, color='gray', linestyle='-.', linewidth=0.7, alpha=0.2)  # Add
                                                                  x, y gridlines
    ax.xaxis.set_tick_params(pad=5)  # Add padding between axes and labels
    ax.yaxis.set_tick_params(pad=10)
    bars = ax.barh(X, T, 0.4)
    ax.bar_label(bars, padding=3)
    plt.ylabel('Similarity Measures')
    plt.xlabel('Accuracy')
    plt.savefig(Str + 'ing Accuracy.png')  # Saving plot as png file
    plt.show()
    plt.close()
```

- **ACC results are plotted for better results presentations:**

```
# Plotting ACC Results ------------------------------------------------------------
P3Extra.HistACC(ACCListTrain, ACCListTest)
print("\n---Plotting Accuracy bar chart finished---")
P3Extra.Hist(ACCListTrain, 'Train')
print("---Plotting Training Accuracy histogram finished---")
P3Extra.Hist(ACCListTest, 'Test')
print("---Plotting Testing Accuracy histogram finished---")
```

Example runs of the program

N = 70

Execution result

```
StackOverflow-main/main.py
      src  dst      tstamp
0       9    8  1217567877
1       1    1  1217573801
2      13    1  1217606247
3      17    1  1217617639
4      48    2  1217618182
...   ...  ...         ...
996    39  291  1218035753
997   383  476  1218035969
998   265  476  1218036416
999   342  370  1218036494
1000   51  192  1218037474

[1001 rows x 3 columns]
Please enter "N" the number of the network's time partitions: 70

t_min: 1217567877
t_max: 1218037474
t: [1217567877.0, 1217574585.5285714, 1217581294.057143, 1217588002.5857143,
1217594711.1142857, 1217601419.642857, 1217608128.1714287, 1217614836.7,
1217621545.2285714, 1217628253.7571428, 1217634962.2857144, 1217641670.8142858,
1217648379.3428571, 1217655087.8714285, 1217661796.4, 1217668504.9285715,
1217675213.4571428, 1217681921.9857142, 1217688630.5142858, 1217695339.0428572,
1217702047.5714285, 1217708756.1, 1217715464.6285715, 1217722173.1571429,
1217728881.6857142, 1217735590.2142856, 1217742298.7428572, 1217749007.2714286,
1217755715.8, 1217762424.3285713, 1217769132.857143, 1217775841.3857143,
1217782549.9142857, 1217789258.442857, 1217795966.9714286, 1217802675.5,
1217809384.0285714, 1217816092.557143, 1217822801.0857143, 1217829509.6142857,
1217836218.142857, 1217842926.6714287, 1217849635.2, 1217856343.7285714,
1217863052.2571428, 1217869760.7857144, 1217876469.3142858, 1217883177.8428571,
1217889886.3714285, 1217896594.9, 1217903303.4285715, 1217910011.9571428,
1217916720.4857142, 1217923429.0142858, 1217930137.5428572, 1217936846.0714285,
1217943554.6, 1217950263.1285715, 1217956971.6571429, 1217963680.1857142,
```

```
1217970388.7142856, 1217977097.2428572, 1217983805.7714286, 1217990514.3,
1217997222.8285713, 1218003931.357143, 1218010639.8857143, 1218017348.4142857,
1218024056.942857, 1218030765.4714286, 1218037474.0]

---Plotting graph of |V[tj-1,tj]| and |E[tj-1,tj]| Time evolution finished---
---Plotting Centrality histograms finished---
---Plotting graph of |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[tj,tj+1]| Time
evolution finished---

-- Optimal Range Sets Rx* --
{'Sgd': (3.0, 5.0), 'Scn': (2, 6), 'Sjc': (0.2, 0.2), 'Sa': (0.6213349345596119,
0.7213475204444817), 'Spa': (6, 24)}

---- Training Accuracy ---- Ranked from highest to lowest
{'Scn': 0.8943395650297616, 'Sjc': 0.8930777269612009, 'Spa': 0.8859392706464206,
'Sgd': 0.8805928297279259, 'Sa': 0.8799724792392748}

----- Testing Accuracy ----- Ranked from highest to lowest
{'Scn': 0.9109733051010912, 'Sjc': 0.9106080418786852, 'Sa': 0.9022526404971469,
'Spa': 0.9015366767048845, 'Sgd': 0.8981047600795967}

---Plotting Accuracy bar chart finished---
---Plotting Training Accuracy histogram finished---
---Plotting Testing Accuracy histogram finished---

Process finished with exit code 0
```

Graphical representations

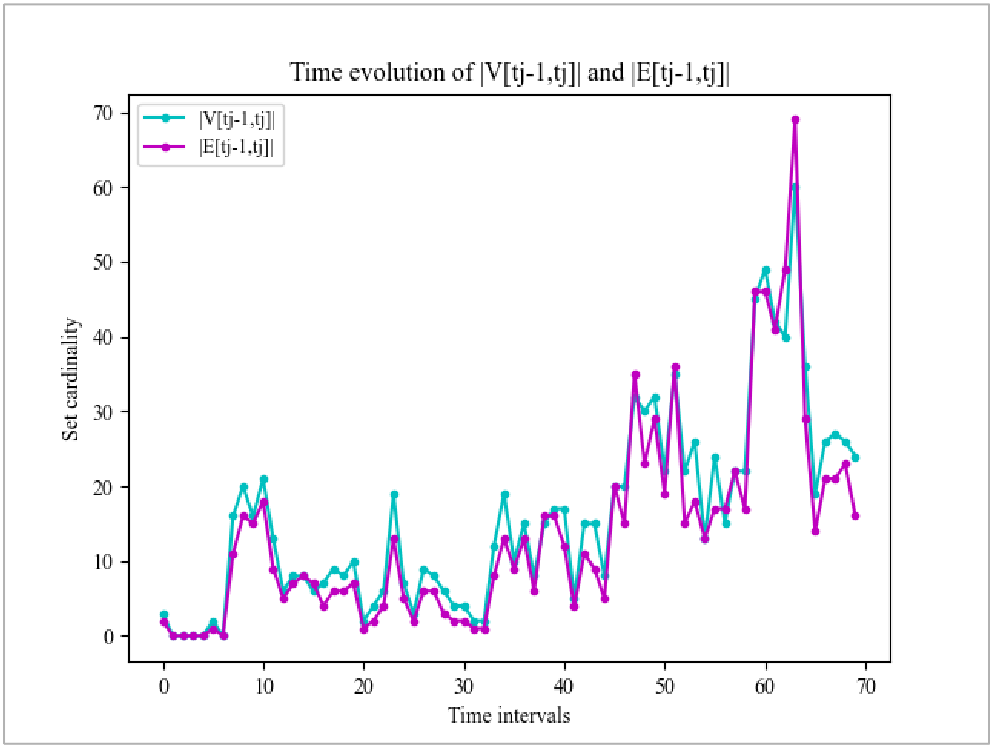


Fig 1. 1

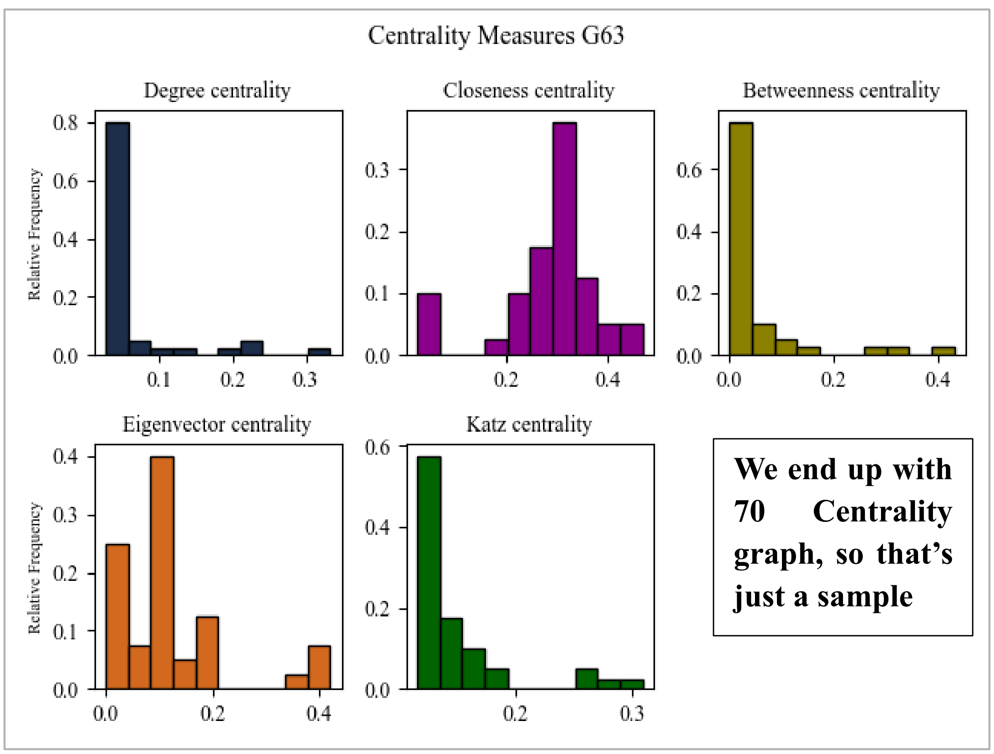


Fig 1. 2

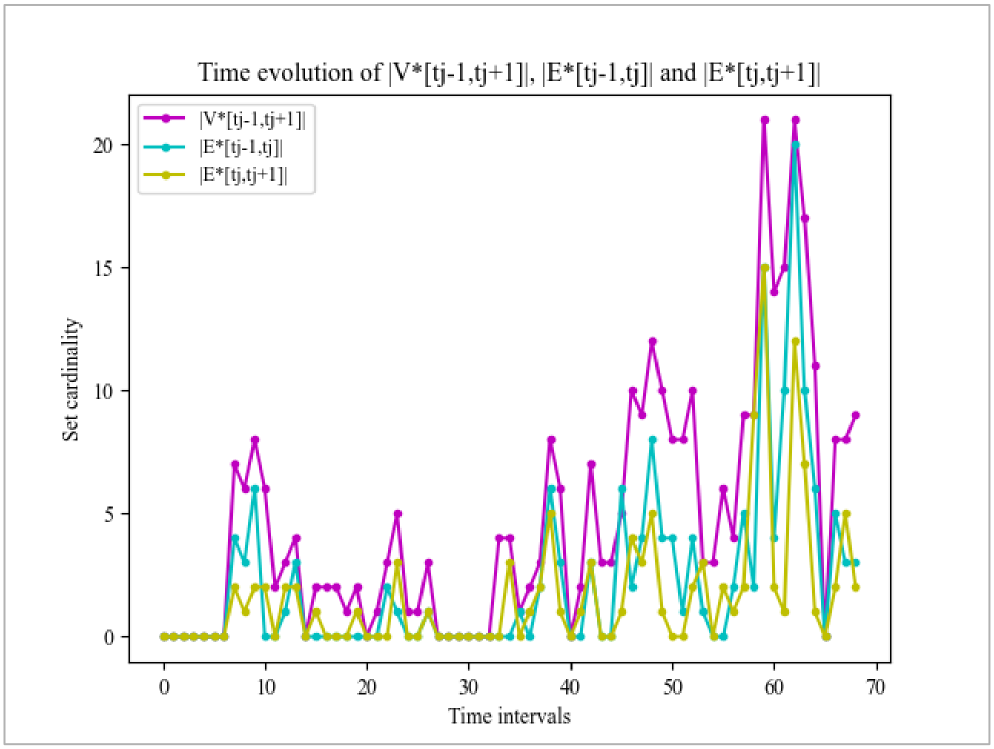

Time evolution of |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[tj,tj+1]|
|V*[tj-1,tj+1]|
|E*[tj-1,tj]|
|E*[tj,tj+1]|
Set cardinality
Time intervals
0
5
10
15
20
0
10
20
30
40
50
60
70

Fig 1. 3

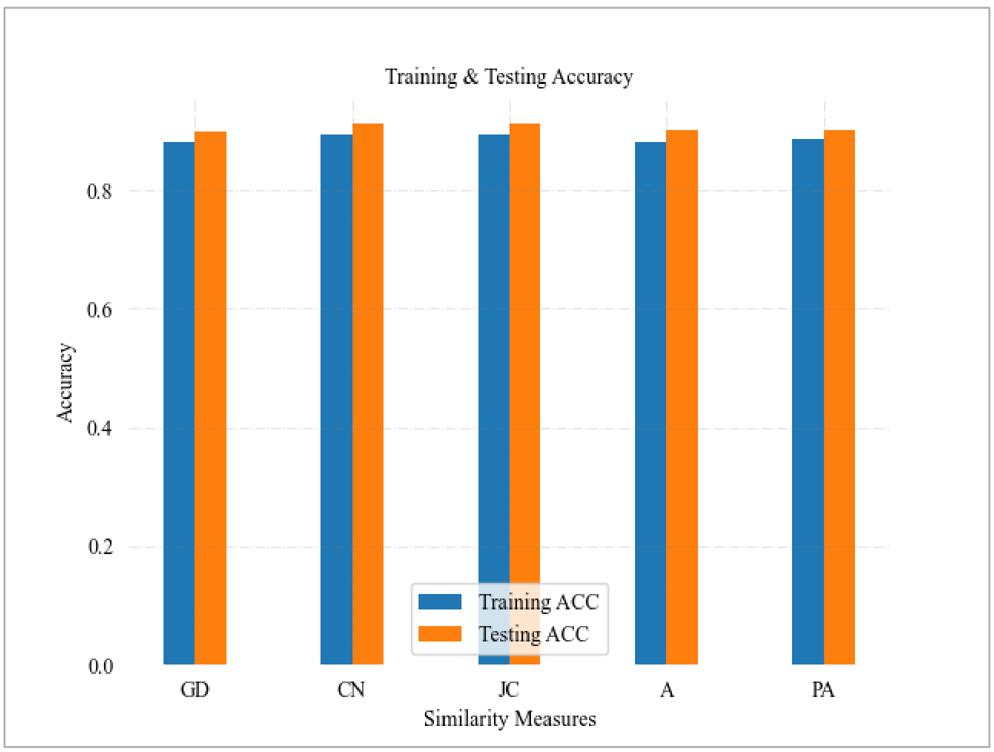

Training & Testing Accuracy
Accuracy
0.0
0.2
0.4
0.6
0.8
Training ACC
Testing ACC
GD
CN
JC
A
PA
Similarity Measures

Fig 1. 4

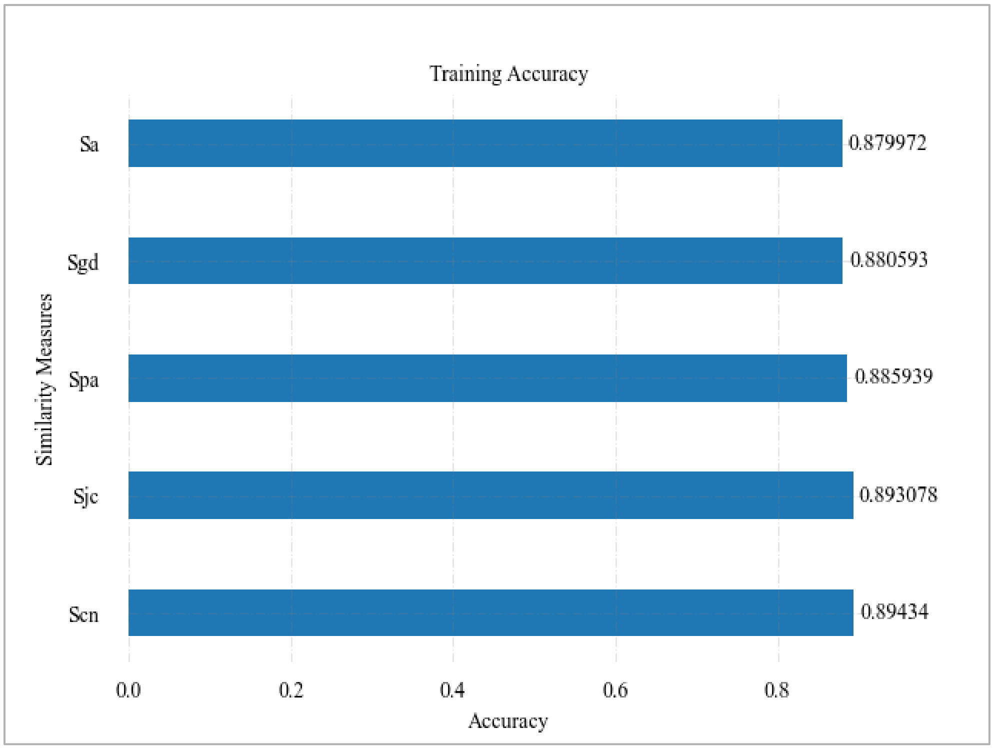

Training Accuracy
Sa
Sgd
Spa
Sjc
Scn
0.879972
0.880593
0.885939
0.893078
0.89434
Similarity Measures
0.0
0.2
0.4
0.6
0.8
Accuracy

Fig 1. 5

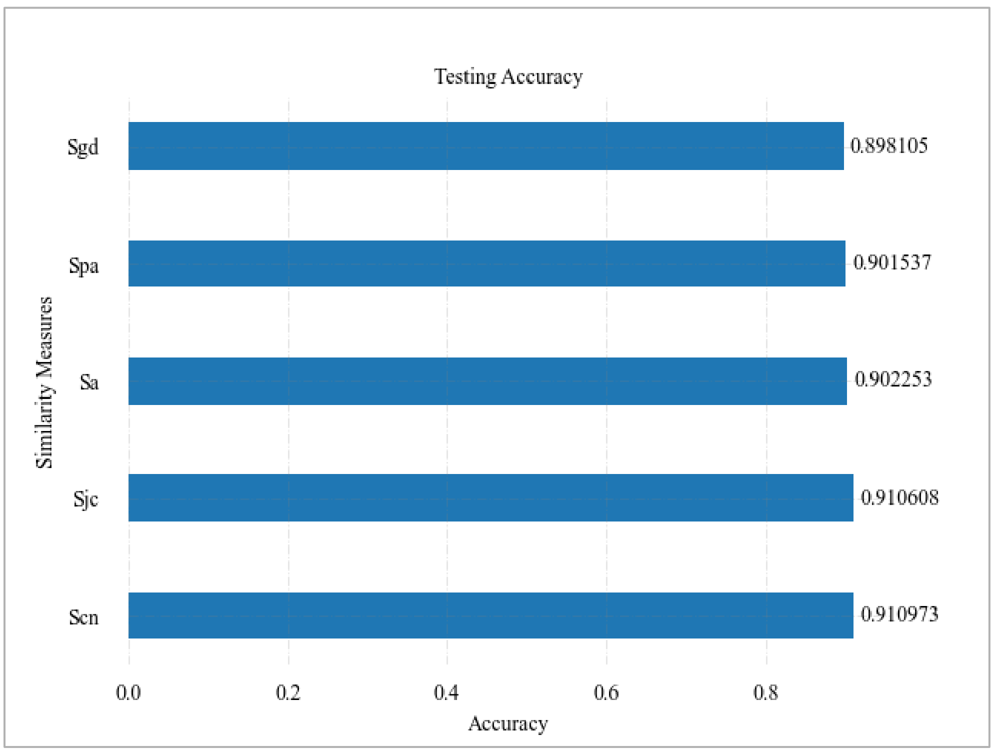

Testing Accuracy
Sgd
Spa
Sa
Sjc
Scn
0.898105
0.901537
0.902253
0.910608
0.910973
Similarity Measures
0.0
0.2
0.4
0.6
0.8
Accuracy

Fig 1. 6

N = 150

Execution result

```
 src  dst      tstamp
0       9    8  1217567877
1       1    1  1217573801
2      13    1  1217606247
3      17    1  1217617639
4      48    2  1217618182
...   ...  ...         ...
996    39  291  1218035753
997   383  476  1218035969
998   265  476  1218036416
999   342  370  1218036494
1000   51  192  1218037474

[1001 rows x 3 columns]
Please enter "N" the number of the network's time partitions: 150

t_min: 1217567877
t_max: 1218037474
t: [1217567877.0, 1217571007.6466668, 1217574138.2933333, 1217577268.94,
1217580399.5866666, 1217583530.2333333, 1217586660.88, 1217589791.5266666,
1217592922.1733334, 1217596052.82, 1217599183.4666667, 1217602314.1133332,
1217605444.76, 1217608575.4066668, 1217611706.0533333, 1217614836.7,
1217617967.3466666, 1217621097.9933333, 1217624228.64, 1217627359.2866666,
1217630489.9333334, 1217633620.58, 1217636751.2266667, 1217639881.8733332,
1217643012.52, 1217646143.1666667, 1217649273.8133333, 1217652404.46,
1217655535.1066666, 1217658665.7533333, 1217661796.4, 1217664927.0466666,
1217668057.6933334, 1217671188.34, 1217674318.9866667, 1217677449.6333334,
1217680580.28, 1217683710.9266667, 1217686841.5733333, 1217689972.22,
1217693102.8666666, 1217696233.5133333, 1217699364.16, 1217702494.8066666,
1217705625.4533334, 1217708756.1, 1217711886.7466667, 1217715017.3933334,
1217718148.04, 1217721278.6866667, 1217724409.3333333, 1217727539.98,
1217730670.6266668, 1217733801.2733333, 1217736931.92, 1217740062.5666666,
1217743193.2133334, 1217746323.86, 1217749454.5066667, 1217752585.1533334,
```

```
1217755715.8, 1217758846.4466667, 1217761977.0933332, 1217765107.74,
1217768238.3866668, 1217771369.0333333, 1217774499.68, 1217777630.3266666,
1217780760.9733334, 1217783891.62, 1217787022.2666667, 1217790152.9133334,
1217793283.56, 1217796414.2066667, 1217799544.8533332, 1217802675.5,
1217805806.1466668, 1217808936.7933333, 1217812067.44, 1217815198.0866666,
1217818328.7333333, 1217821459.38, 1217824590.0266666, 1217827720.6733334,
1217830851.32, 1217833981.9666667, 1217837112.6133332, 1217840243.26,
1217843373.9066668, 1217846504.5533333, 1217849635.2, 1217852765.8466666,
1217855896.4933333, 1217859027.14, 1217862157.7866666, 1217865288.4333334,
1217868419.08, 1217871549.7266667, 1217874680.3733332, 1217877811.02,
1217880941.6666667, 1217884072.3133333, 1217887202.96, 1217890333.6066666,
1217893464.2533333, 1217896594.9, 1217899725.5466666, 1217902856.1933334,
1217905986.84, 1217909117.4866667, 1217912248.1333334, 1217915378.78,
1217918509.4266667, 1217921640.0733333, 1217924770.72, 1217927901.3666666,
1217931032.0133333, 1217934162.66, 1217937293.3066666, 1217940423.9533334,
1217943554.6, 1217946685.2466667, 1217949815.8933334, 1217952946.54,
1217956077.1866667, 1217959207.8333333, 1217962338.48, 1217965469.1266668,
1217968599.7733333, 1217971730.42, 1217974861.0666666, 1217977991.7133334,
1217981122.36, 1217984253.0066667, 1217987383.6533334, 1217990514.3,
1217993644.9466667, 1217996775.5933332, 1217999906.24, 1218003036.8866668,
1218006167.5333333, 1218009298.18, 1218012428.8266666, 1218015559.4733334,
1218018690.12, 1218021820.7666667, 1218024951.4133334, 1218028082.06,
1218031212.7066667, 1218034343.3533332, 1218037474.0]

---Plotting graph of |V[tj-1,tj]| and |E[tj-1,tj]| Time evolution finished---
---Plotting Centrality histograms finished---
---Plotting graph of |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[tj,tj+1]| Time
evolution finished---

-- Optimal Range Sets Rx* --
{'Sgd': (3.0, 5.0), 'Scn': (2, 6), 'Sjc': (0.25, 0.25), 'Sa': (0.9102392266268372,
0.9102392266268372), 'Spa': (4, 16)}

---- Training Accuracy ---- Ranked from highest to lowest
{'Scn': 0.8291193278601974, 'Sjc': 0.8282363341398334, 'Sgd': 0.8232441462360163,
'Sa': 0.8231308546483257, 'Spa': 0.8211452977328605}
```

```
----- Testing Accuracy ----- Ranked from highest to lowest
{'Scn': 0.8140765850095326, 'Sjc': 0.813442854393103, 'Sa': 0.8130849336119995,
'Spa': 0.8079745517262258, 'Sgd': 0.8067616426362805}

---Plotting Accuracy bar chart finished---
---Plotting Training Accuracy histogram finished---
---Plotting Testing Accuracy histogram finished---

Process finished with exit code 0
```

Graphical representations

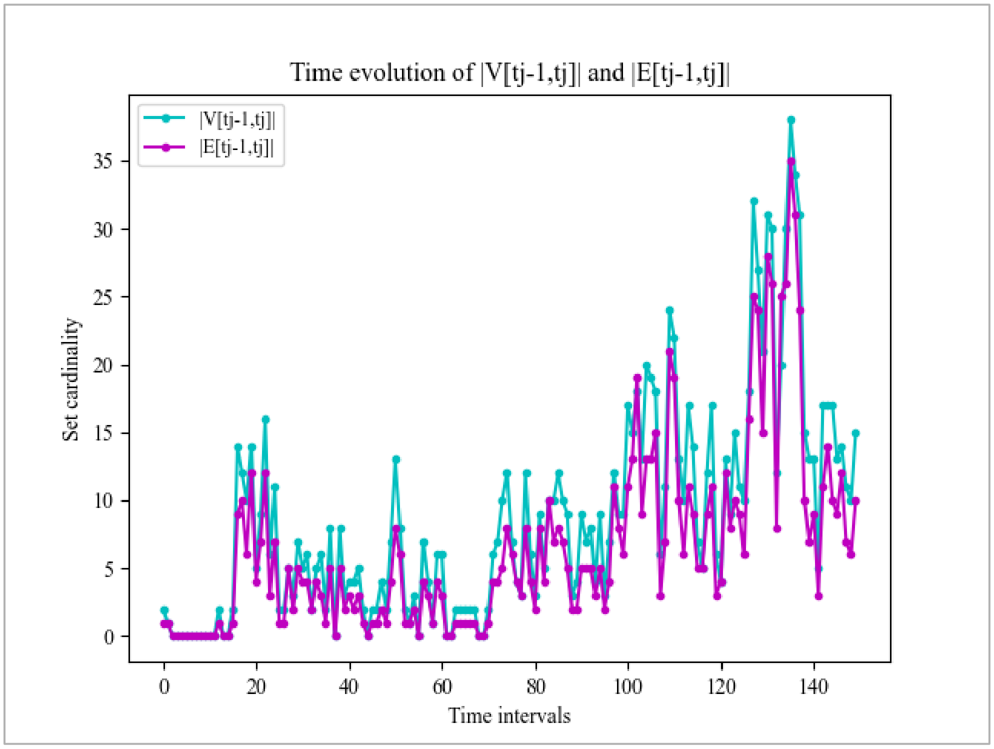


Fig 2. 1

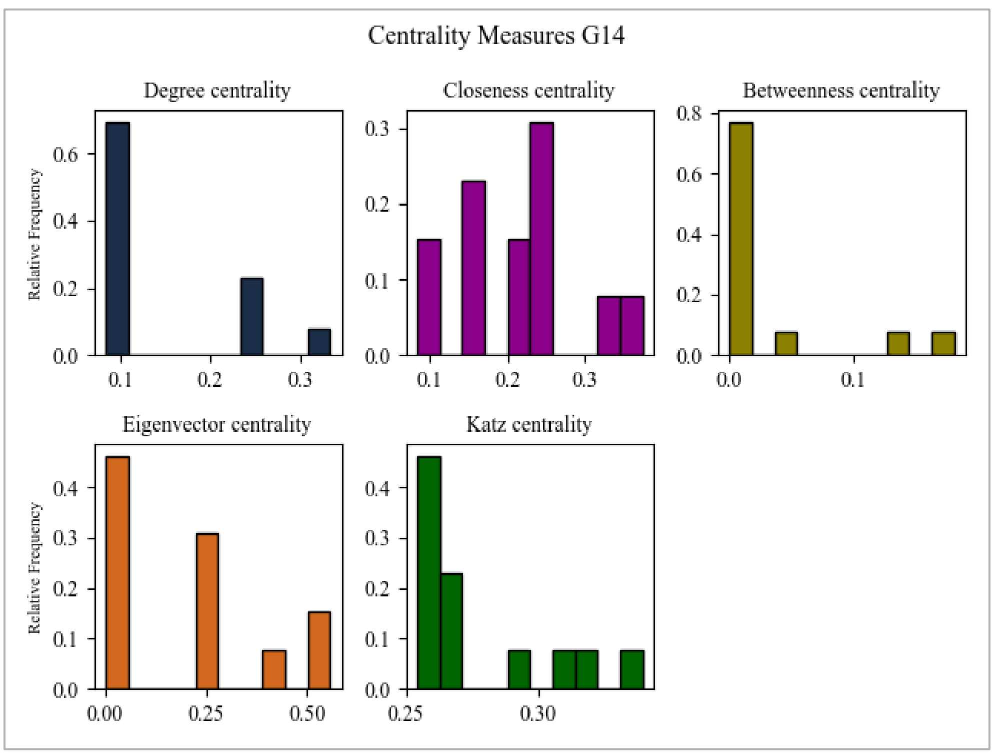
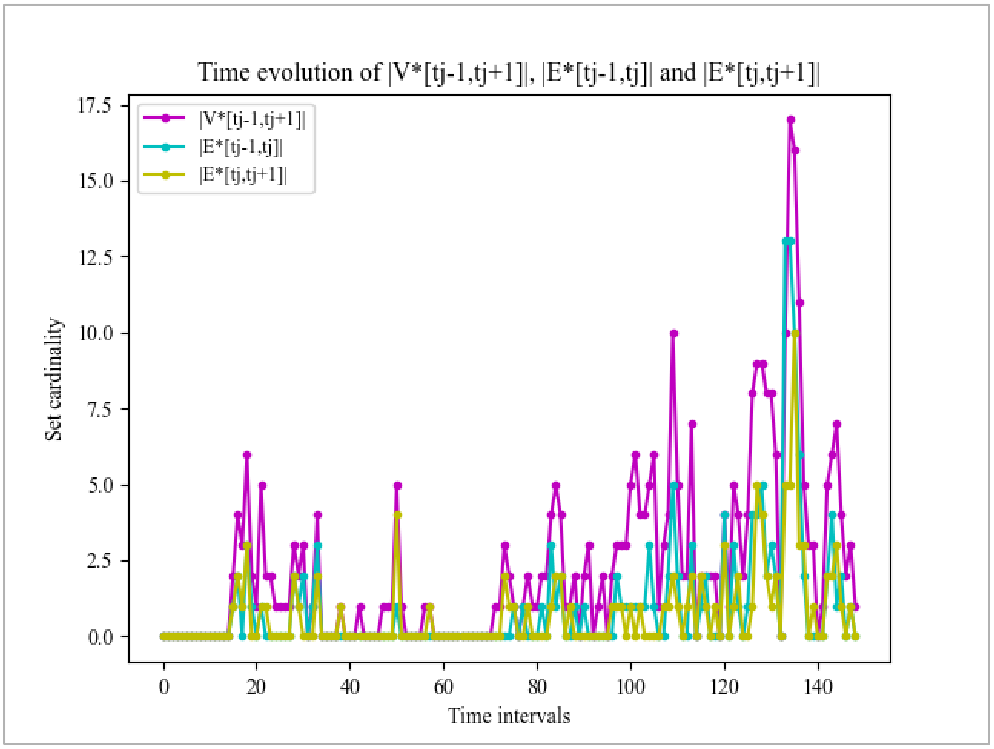
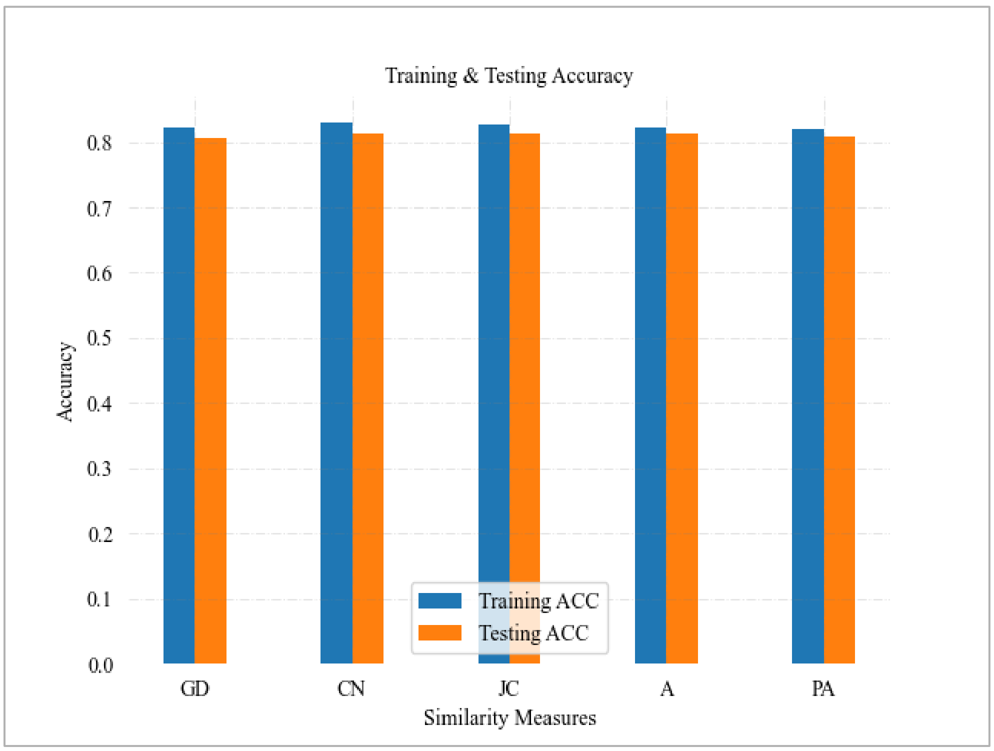


Fig 2. 2



Time evolution of |V*[tj-1,tj+1]|, |E*[tj-1,tj]| and |E*[tj,tj+1]|
|V*[tj-1,tj+1]|
|E*[tj-1,tj]|
|E*[tj,tj+1]|
Set cardinality
Time intervals
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
20
40
60
80
100
120
140

Fig 2. 3



Training & Testing Accuracy
Accuracy
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Training ACC
Testing ACC
GD
CN
JC
A
PA
Similarity Measures

Fig 2. 4

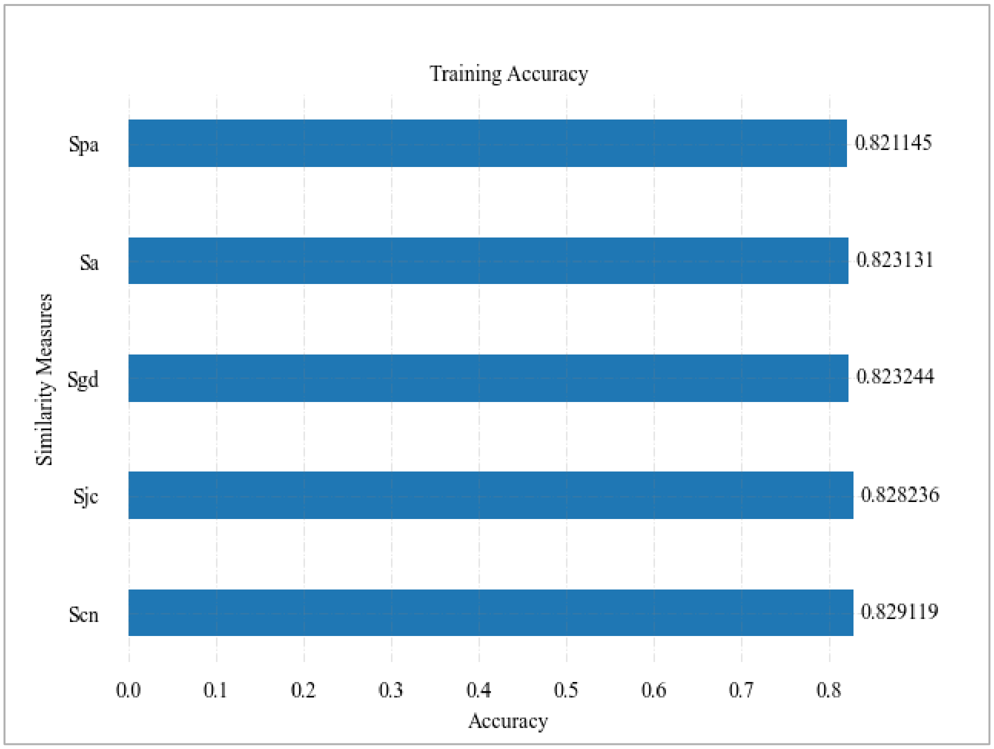

Training Accuracy
Spa
Sa
Sgd
Sjc
Scn
0.821145
0.823131
0.823244
0.828236
0.829119
Similarity Measures
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Accuracy

Fig 2. 5

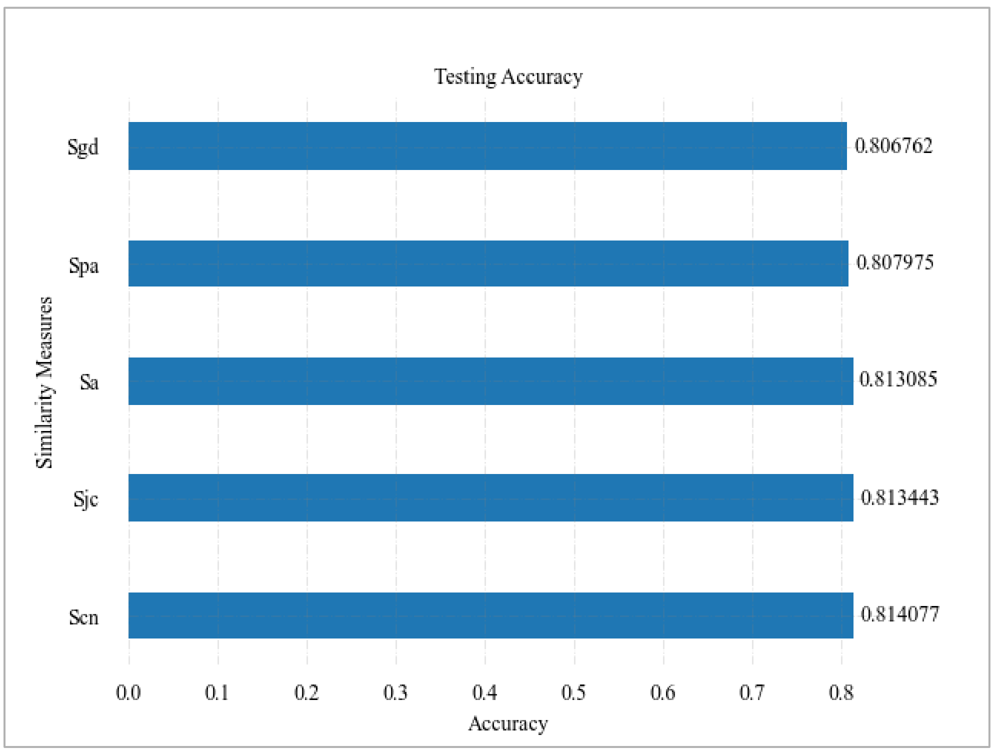

Testing Accuracy
Sgd
Spa
Sa
Sjc
Scn
0.806762
0.807975
0.813085
0.813443
0.814077
Similarity Measures
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Accuracy

Fig 2. 6