Project 2

Nishant Sahni - 2023CSB1140

April 25, 2024

We are given an impression network , based on an experiment conducted among the students , and we are to run the following experiments on these dataset.

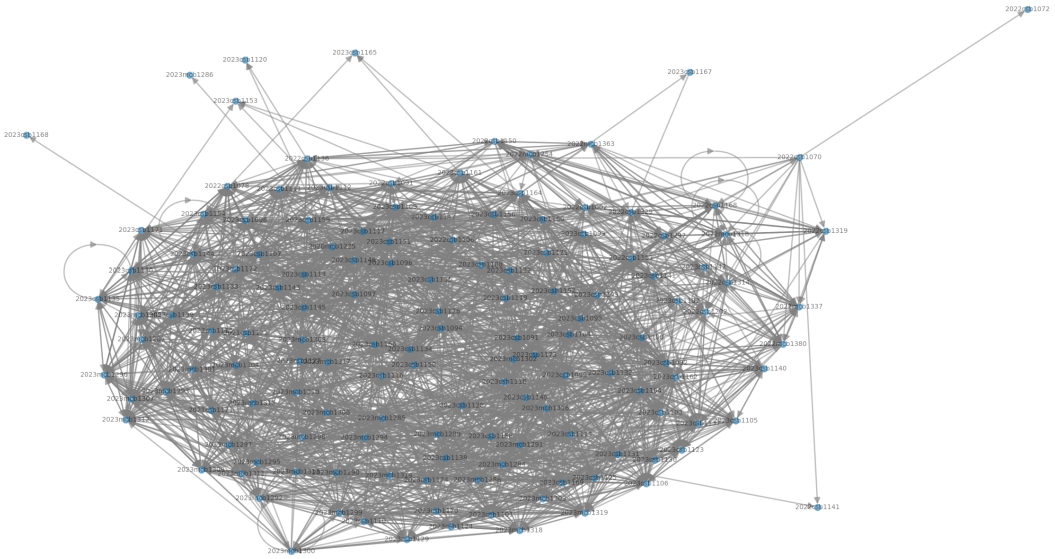
Python Implementation of said network into a Directed Graph :

```python
import networkx as nx
import pandas as pd
#importing all required libraries

G=nx.DiGraph()
Df=pd.read_csv(r'E:\Python Programs\Project 2\ImpressionNetwork.csv')
Df=Df.drop(columns=Df.columns[0])
Df['Email Address']=Df['Email Address'].str[0:11]
for column in Df.columns[1:]:
    Df[column] = Df[column].apply(lambda x: x[-11:].lower() if pd.notnull(x) else x)
Df.rename(columns={'Email Address':'Node'},inplace=True)
#Created a dataframe comprising only entry numbers to identify nodes

for node in Df.iloc[:, 0]:
    if pd.notnull(node):
        G.add_node(node)
#added all possible nodes to graoh

# Add edges from each node to non-empty cells in the same row
for index, row in Df.iterrows():
    node = row.iloc[0]
    for col_index, cell in enumerate(row):
        if pd.notnull(cell) and col_index != 0:
            G.add_edge(node, cell)
#Our directed graph is now made
```



The graph's visual representation through matplotlib is as shown above :

---

1 Question 1 - Top Leader

As discussed in the lectures and through lecture videos , one of the most optimal ways to find the top leader among an impression network is the random walk across the graph. The algorithm is as follows ;
Pseudo-Code:

- Initialize a dictionary to keep track of coins
- Start from a random node in the graph and drop a coin there
- If there are outgoing edges , go to a random neighbour , drop coin , repeat.
- If no outgoing edges , teleport to a random node in the graph and repeat the process
- Print the top leader (the one with highest number of coins)

Code Snippet :

```
#Q1 - RANDOM WALK
#Creating a dictionary to keep track of number of coins for each node
node_dict={node:0 for node in G.nodes}
#starting from a random node
start=random.choice(list(G.nodes()))
node_dict[start]+=1
next=start
#performing the walk for 1 million steps
for a in range(1000000):
    if(len(list(G.successors(start)))!=0):
        next=random.choice(list(G.successors(start)))
    else:
        #if no outgoing edges , teleport to a random node
        next=random.choice(list(G.nodes()))
    node_dict[next]+=1
    start=next

print(list(node_dict.keys())[list(node_dict.values()).index(max(list(node_dict.values())))])
```

The last line is a complicated way to print the node key corresponding to the largest value in the dictionary

The theory says that , given we traverse the nodes a large number of times , the node with the highest number of coins at the end will be the top leader.

So , through Python , I implemented exactly that using various libraries and methods . After traversing a million times (1000000 node steps) , I found the top leader to be 2023CSB1091.

Coin Rank List :

1. 2023CSB1091
2. 2023MCB1316
3. 2023MCB1284
4. 2023CSB1162

It is worth noting that if we run the code multiple times , the rank list fluctuates for ranks below 5.

---

2 Question 2 - Missing Links

As discussed in the lectures , we are to find the missing links (edges) in our impression network through the following steps:

- Convert our directed graph to its adjacency matrix (a form of representation of graphs)
- For each row , find coefficients (of linear combination) of the other rows , that would linearly combine to make up the current particular row
- Find what the linear combination is summing up to
- Set a threshold for this linear combination
- If it crosses the threshold , we will recommend that edge as a missing link , else we won't

Now , to find the combinations , I will make use of the SciKit-Learn Library , in particular , it's Linear Regression Module. I will be applying linear regression on the rows to find the coefficients of linear combinations that best fit the inputs.

```
#Missing Links

from sklearn.linear_model import LinearRegression as LR
model=LR() #creating linear regression model

X=adjacency_matrix  #defined above

coeff=[]
for i in range(len(X)):
    tar=X[i] #ith row
    Others=X.copy()
    Others=np.delete(Others,(i),axis=0) # storing all others except ith row
    Others=Others.T
    model.fit(Others,tar)
    coeff.append(model.coef_)
coeff=np.array(coeff)
#for i in range(len(coeff)):
#    print(np.all(coeff[i]==0))
def linearcomb(i,j):
    targetcol=np.array(X)[:,j]
    targetcol=np.delete(targetcol,i)
    #print(coeff)
    if(np.all(coeff[i]==0)):
        #if all entries in a row are zero , then model.fit will return all coefficients corresponding to it to be zero
        #so , if we don't make a special case for this , and just find the linear combination score , then we will not be able to pred
        #so we need to assign some sort of score , so that we can identify whether this edge is worthy of being a missing link or not
        #one measure of popularity is the PageRank score
        score_jth=nx.pagerank(G)[str(list(G.nodes)[j])] #pagerank score of the jth node
        #set a threshold on this score , if its above some value , then we will reccomend it as a missing link
        averagescore=sum(list(nx.pagerank(G).values()))/len(nx.pagerank(G).values())
        if score_jth>=averagescore:
            return np.random.choice([1,1,1,-1])  #will be recommended as a missing link with 2/3 probability
        else:
```

```
            return np.random.choice([-1,-1,-1,1]) #will not be recommended as a missing link with 1/3 probability
    else:
        return np.dot(coeff[i],targetcol)
def identifylinks(i,j):
    MissingLink=()
    predictionscore=linearcomb(i,j) #this will either be dependent on the pagerank score or the dot product of np.dot
    if predictionscore>=0: #after testing the code numerous , I found this threshold/tolerance to be most appropriate
        MissingLink=(list(G.nodes)[i],list(G.nodes)[j])
        return MissingLink
    return None

def recommend():
    MissingLinks=[]
    for i in range(0,len(G.nodes())):
        for j in range(0,len(G.nodes())):
            if(i!=j and identifylinks(i,j)!=None):
                #print(list(G.nodes)[i],'-',list(G.nodes)[j])
                MissingLinks.append(identifylinks(i,j))
    return MissingLinks
print('There are ',len(recommend()),'number of missing links (edges) after using the specified tolerance')
#print(recommend()) #prints all the edges that are missing
```

Pseudo-Code :

1. Define adjacency matrix for the graph
2. Iterate through the rows , for each row , find the coefficients of the other n-1 rows using model.fit
3. Store the lists of coefficients in a numpy array
4. Define a function linearcomb that will deal with assigning a score to each edge.
5. 2 cases , if the current row has all entries zero or not.
6. Case 1: If the current row has all entries zero , the function model.fit will return all coefficients zero. This will not allow us to predict whether the edge is a missing link or not. So , instead we should assign some sort of score that will let us determine whether the edge should be recommended or not. One measure of a node's popularity , is its PageRank score. (If a node is popular in terms of its PageRank score , then it is way more likely to be part of a missing link with another node.) So , I set a threshold , that if a node's PageRank is more than average , its edge with the current row node has a 75% probability of being recommended , and if less than average , it has a 25% probability of being recommended.
7. Case 2: If current row's all entries are non-zero , apply dot product with current row and column to find the sum of the linear combinations we got model.fit If this is higher than a specified threshold , recommend it , otherwise not.
8. Define 2 other functions to collect all the recommended missing links.

Now comes the matter of what to set as threshold for Case 2.

I simply set the threshold as 0.5 for the linear combination sum , after observing the values that were being generated for each iteration. 0.5 seemed like a good cutoff considering the distribution of data seemed to be centering around that neighbourhood.

After specifying such a tolerance , the list of missing edges was obtained , length of which is in the range of [3900,4500] changing with each run of the code.
The output list of missing links is commented in the code attached

---

3 Question 3

My brand new problem is a three-parter ,I'm going to investigate the measures of *ASSORTATIVITY* , *MUTUALITY* and *CENTRALITY* in the graph.

3.1 Assortativity

★ This is the extent to which the nodes in graph have edges to similar/dissimilar nodes. I will be defining similarity by the person's branch/department .

```
#Assortativity

total=G.number_of_edges() #storing cardinality of edge-set
#print(G.has_edge(list(G.nodes)[4],list(G.nodes)[5]))
similar=0
dissimilar=0
selfedges=0
for i in list(G.nodes):
    for j in list(G.nodes):
            if (i!=j and G.has_edge(i,j)):
                if i[4:7]==j[4:7]:
                    similar+=1     #if same branch
                else:
                    dissimilar+=1    #if different branch
            if(i==j and G.has_edge(i,j)):
                selfedges+=1            #increment number of people impressed with themself
assortativity=similar/(total-selfedges) #this is the ratio of similar edges to total non-self edges
print("The percentage of edges between people of the same branch is",assortativity*100,'%')
```

In the above code snippet , I found the number of edges between people of the same branch , and calculated its ratio with total number of non-self edges.
Self-edges being those in which a node is narcissistically impressed with itself.

The extent of assortativity in the graph came out to be 72.89%

3.2 Mutuality

★ This is the extent to which out of all the node pairs in the graph , which nodes reciprocate the impressiveness , that is , how many nodes have both incoming and outgoing edges from the same node.

```
#Mutuality

double=0
container=set()  #initialized a set to keep track of the pairs of nodes already checked
for i in list(G.nodes):
    for j in list(G.nodes):
        if(i!=j and G.has_edge(i,j) and G.has_edge(j,i) and ((i,j) not in container) and ((j,i) not in container)):
            double+=1
            container.add((i,j))
            container.add((j,i))
print("There are a total of",double,'mutual edges in the network.')
#-------------------------------------------------------
```

In the output of this code snippet , I found the total number of double-sided edges to be 1204. (2 mutual directed-edges count as one double-edge).

3.3 Centrality

Now , we come to the main component of Question 3 , Centrality.

Centrality is a relatively blanket term , there are different types of centrality. Centrality is basically a measure of node's influence in a social network.
By finding the most influential person in a network , we can apply this result and model in various scenarios like spreaders in a disease network , influencers in a social media setting and so on.

3.3.1 Closeness Centrality

Closeness centrality of a node is mathematically defined as the summation of the reciprocal of the distance of the shortest paths between that node and all other nodes in the graph.

Closeness centrality of node X = C(X) = $\frac{N-1}{\sum_y d(y,X)}$
Where d(y,X) is the length of shortest path from node y to X.

Physically , closeness centrality has a variety of applications such as ;

1. Identifying the way academics choose which papers and documentations to take their reference/inspirations from
2. Analysing customer data of a service company and finding significant nodes
3. Identifying the significance of city in the air transport network

and so on.....

```python
#Centrality
store={}
for X in list(G.nodes):
    CX=0
    if (len(list(G.neighbors(X)))==0):
        continue
    else:
        for y in list(G.nodes):
            if y!=X:
                if(len(list(G.neighbors(y)))==0):
                    continue
                else:
                    CX+=1/(nx.shortest_path_length(G,source=y,target=X))
        CX=CX*(len(list(G.nodes))-1)
    store[X]=CX
sortedC=sorted(list(store.values()),reverse=True)
top10=[]
for a in sortedC[:10]:
    ind=list(store.values()).index(a)
    top10.append(list(store.keys())[ind])
print("The top 10 nodes ranked by closeness centrality are :",top10)
```

In the above code , I have stored the values of each node's closeness centrality in the form of a dictionary , and then I have printed the top 10 nodes (ranked by closeness centrality).

The top 10 came out to be:

```
['2023csb1091', '2023csb1132', '2023csb1099', '2023csb1162', '2023mcb1316', '2023mcb1302', '2023csb1126', '2023mcb1284', '2023csb1166', '2023csb1145']
```

Coincidentally , **our top leader by the Random Walk is the same as our leader by Closeness Centrality**. This shows how closely linked such concepts are when discussing the influence of a node in a social impression network.

References:

1. GeeksForGeeks SciKit-Learn Documentation
2. GeeksForGeeks Linear Regression Documentation
3. Social Network Analysis - Wiki
4. NetworkX Documentation
5. Centrality - Wiki
