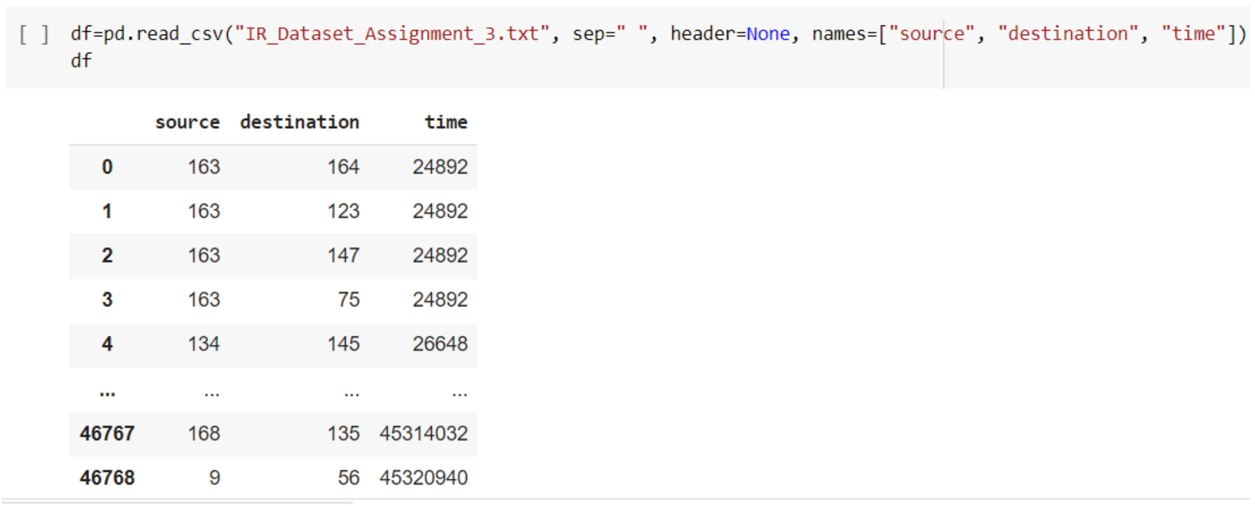
Assignment 3

Dataset:

- **About ===>** The network was generated using email data from a large European research institution. We have anonymized information about all incoming and outgoing emails between members of the research institution. A directed edge (u, v, t) means that person u sent an e-mail to person v at time t.

- **Source ===>** https://snap.stanford.edu/data/email-Eu-core-temporal.html
    - **File ===>** email-Eu-core-temporal-Dept2.txt.gz
    - **Description ===>** All e-mails between members of Department 2 at the institution

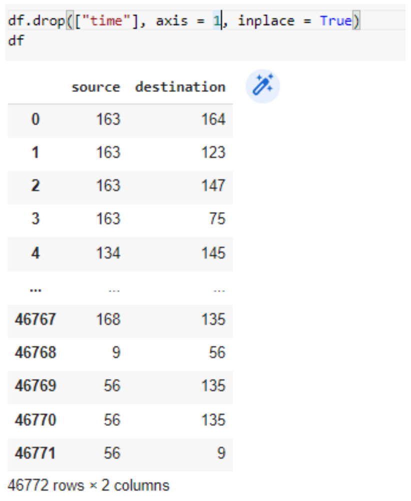
- Load dataset: name 3 columns sender, destination, time of email



```
[ ] df=pd.read_csv("IR_Dataset_Assignment_3.txt", sep=" ", header=None, names=["source", "destination", "time"])
    df
```

```
       source  destination      time
0         163          164     24892
1         163          123     24892
2         163          147     24892
3         163           75     24892
4         134          145     26648
...       ...          ...       ...
46767     168          135  45314032
46768       9           56  45320940
```

- The "**time**" column is not needed hence we can drop it



```
df.drop(["time"], axis = 1, inplace = True)
df
```

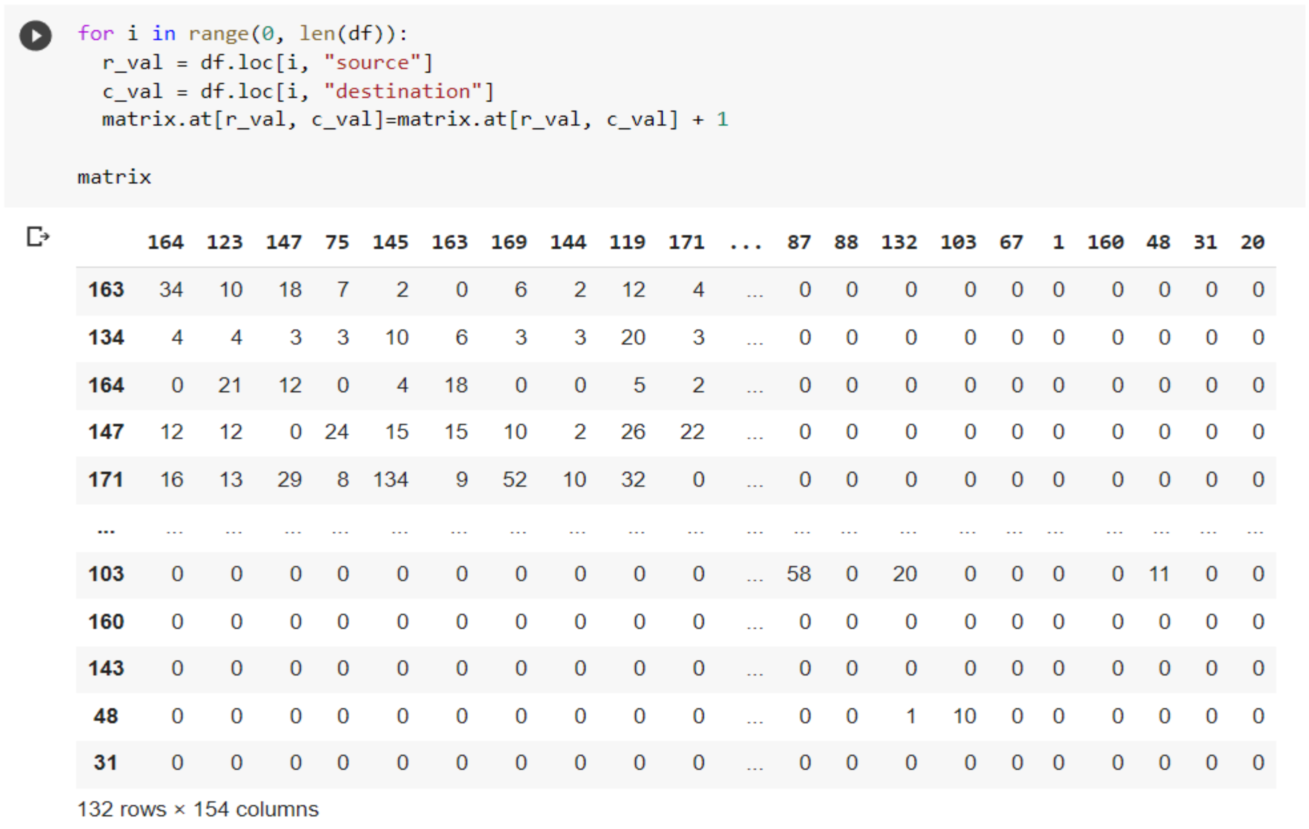
```
       source  destination
0         163          164
1         163          123
2         163          147
3         163           75
4         134          145
...       ...          ...
46767     168          135
46768       9           56
46769      56          135
46770      56          135
46771      56            9
46772 rows × 2 columns
```

Question 1:

Adjacency Matrix :

First, we create an empty adjacency matrix with the help of source and destination nodes. The values for all are zero initially

```
matrix = pd.DataFrame(0, index=np.arange(len(src)), columns=des)
matrix.index = src
matrix
```



```
for i in range(0, len(df)):
  r_val = df.loc[i, "source"]
  c_val = df.loc[i, "destination"]
  matrix.at[r_val, c_val]=matrix.at[r_val, c_val] + 1

matrix
```

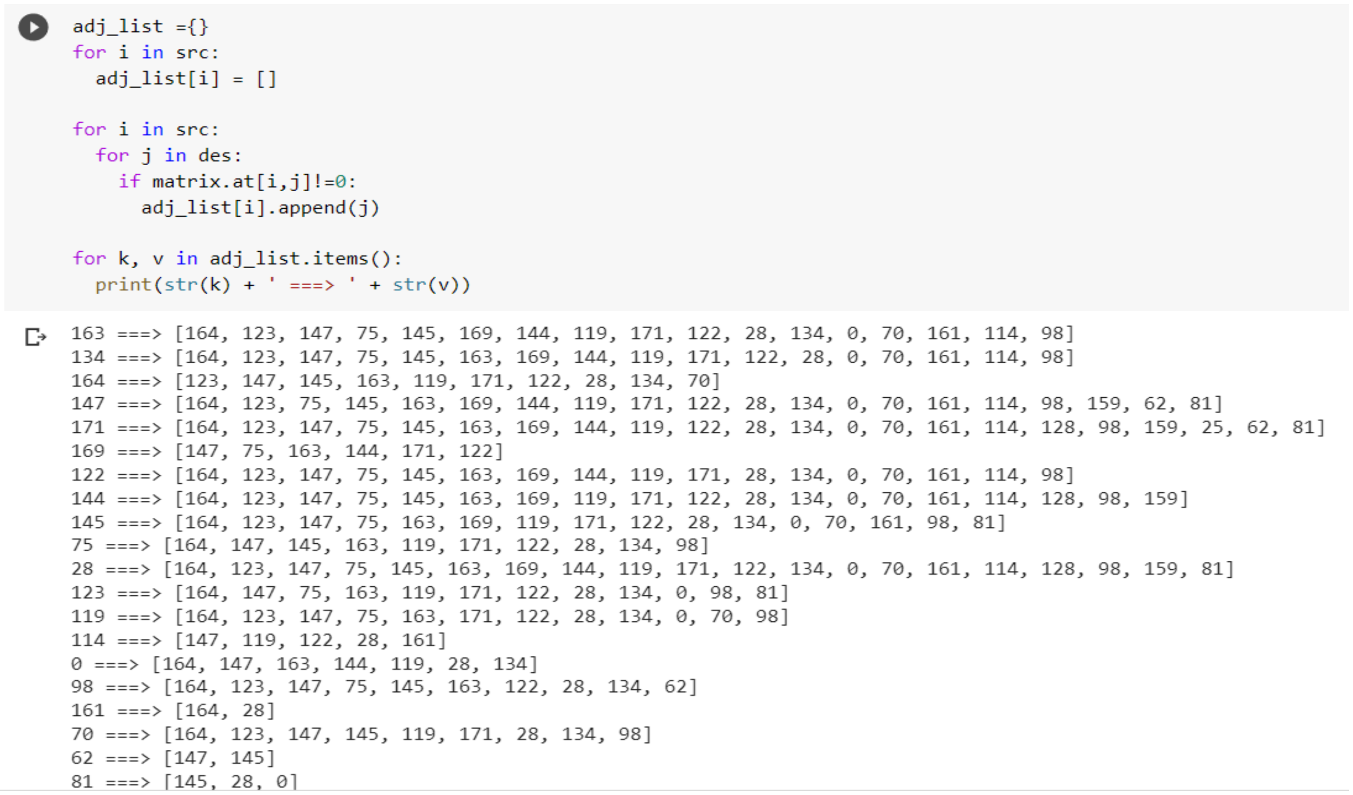
	164	123	147	75	145	163	169	144	119	171	...	87	88	132	103	67	1	160	48	31	20
163	34	10	18	7	2	0	6	2	12	4	...	0	0	0	0	0	0	0	0	0	0
134	4	4	3	3	10	6	3	3	20	3	...	0	0	0	0	0	0	0	0	0	0
164	0	21	12	0	4	18	0	0	5	2	...	0	0	0	0	0	0	0	0	0	0
147	12	12	0	24	15	15	10	2	26	22	...	0	0	0	0	0	0	0	0	0	0
171	16	13	29	8	134	9	52	10	32	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
103	0	0	0	0	0	0	0	0	0	0	...	58	0	20	0	0	0	0	11	0	0
160	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
143	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0	0	0	...	0	0	1	10	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

132 rows × 154 columns

Here matrix is a dataframe whose shape is number of source nodes and number of destination nodes. Initially dataframe is created with 0’s. The above code will update the values in the corresponding rows and columns.

The rows in this dataframe denotes the number of unique source nodes whereas the columns in this dataframe denotes the number of unique destination nodes.

Adjacency List :

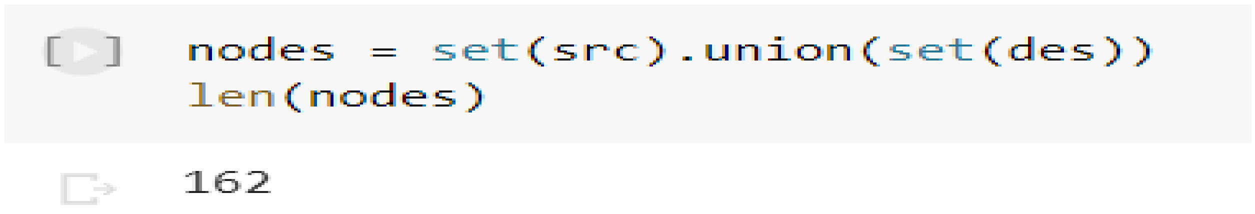


```
adj_list ={}
for i in src:
  adj_list[i] = []

for i in src:
  for j in des:
    if matrix.at[i,j]!=0:
      adj_list[i].append(j)

for k, v in adj_list.items():
  print(str(k) + ' ===> ' + str(v))
```

```
163 ===> [164, 123, 147, 75, 145, 169, 144, 119, 171, 122, 28, 134, 0, 70, 161, 114, 98]
134 ===> [164, 123, 147, 75, 145, 163, 169, 144, 119, 171, 122, 28, 0, 70, 161, 114, 98]
164 ===> [123, 147, 145, 163, 119, 171, 122, 28, 134, 70]
147 ===> [164, 123, 75, 145, 163, 169, 144, 119, 171, 122, 28, 134, 0, 70, 161, 114, 98, 159, 62, 81]
171 ===> [164, 123, 147, 75, 145, 163, 169, 144, 119, 122, 28, 134, 0, 70, 161, 114, 128, 98, 159, 25, 62, 81]
169 ===> [147, 75, 163, 144, 171, 122]
122 ===> [164, 123, 147, 75, 145, 163, 169, 144, 119, 171, 28, 134, 0, 70, 161, 114, 98]
144 ===> [164, 123, 147, 75, 145, 163, 169, 119, 171, 122, 28, 134, 0, 70, 161, 114, 128, 98, 159]
145 ===> [164, 123, 147, 75, 163, 169, 119, 171, 122, 28, 134, 0, 70, 161, 98, 81]
75 ===> [164, 147, 145, 163, 119, 171, 122, 28, 134, 98]
28 ===> [164, 123, 147, 75, 145, 163, 169, 144, 119, 171, 122, 134, 0, 70, 161, 114, 128, 98, 159, 81]
123 ===> [164, 147, 75, 163, 119, 171, 122, 28, 134, 0, 98, 81]
119 ===> [164, 123, 147, 75, 163, 171, 122, 28, 134, 0, 70, 98]
114 ===> [147, 119, 122, 28, 161]
0 ===> [164, 147, 163, 144, 119, 28, 134]
98 ===> [164, 123, 147, 75, 145, 163, 122, 28, 134, 62]
161 ===> [164, 28]
70 ===> [164, 123, 147, 145, 119, 171, 28, 134, 98]
62 ===> [147, 145]
81 ===> [145, 28, 0]
```

adj_list is a dictionary whose keys are source nodes and the value is a list where the outgoing edges reach (destination nodes)

1. Number of Nodes :



```
nodes = set(src).union(set(des))
len(nodes)
```

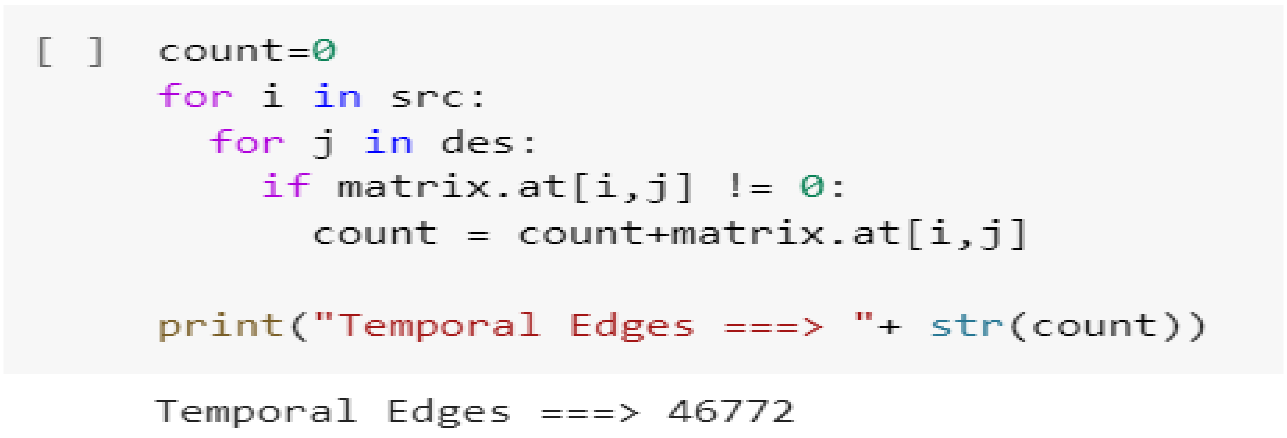
```
162
```

Here src and des are two lists taken from the given dataset. For both the list, no order is changed.

Formula -> Union(set(src) and set(des))

The reason for using the set is both lists contain duplicates. To avoid them we are converting the list to a set and taking union for both sets.

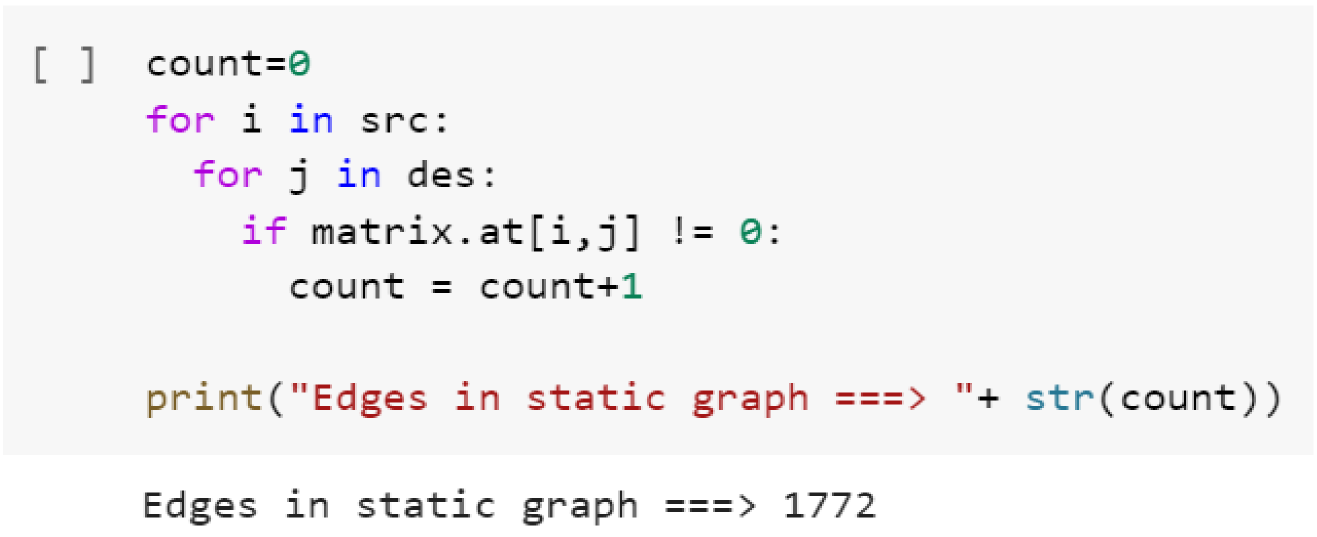
2. Number of Edges :



```
[ ]  count=0
     for i in src:
       for j in des:
         if matrix.at[i,j] != 0:
           count = count+matrix.at[i,j]

     print("Temporal Edges ===> "+ str(count))

     Temporal Edges ===> 46772
```

This code will calculate the temporal edges which means that while counting the number of edges if multiple edges present between 2 nodes we will count all the multiple edges.

Formula -> if there is non zero value in the matrix then it means there is an edge
Hence here we **count the number of non zero values in the matrix.**

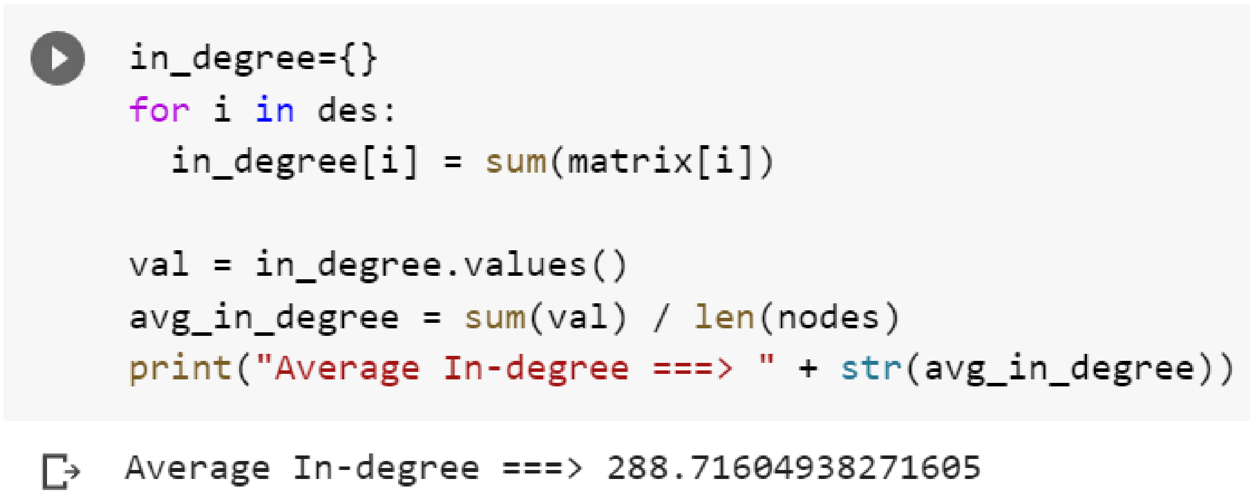


```
[ ]  count=0
     for i in src:
       for j in des:
         if matrix.at[i,j] != 0:
           count = count+1

     print("Edges in static graph ===> "+ str(count))

     Edges in static graph ===> 1772
```

Here we are counting single edge even if we have multiple edges also. This is called as **static graph.**

3. Avg In-degree :

Formula -> **Average Indegree = Total Indegree / Number of Nodes**



```
in_degree={}
for i in des:
  in_degree[i] = sum(matrix[i])

val = in_degree.values()
avg_in_degree = sum(val) / len(nodes)
print("Average In-degree ===> " + str(avg_in_degree))
```

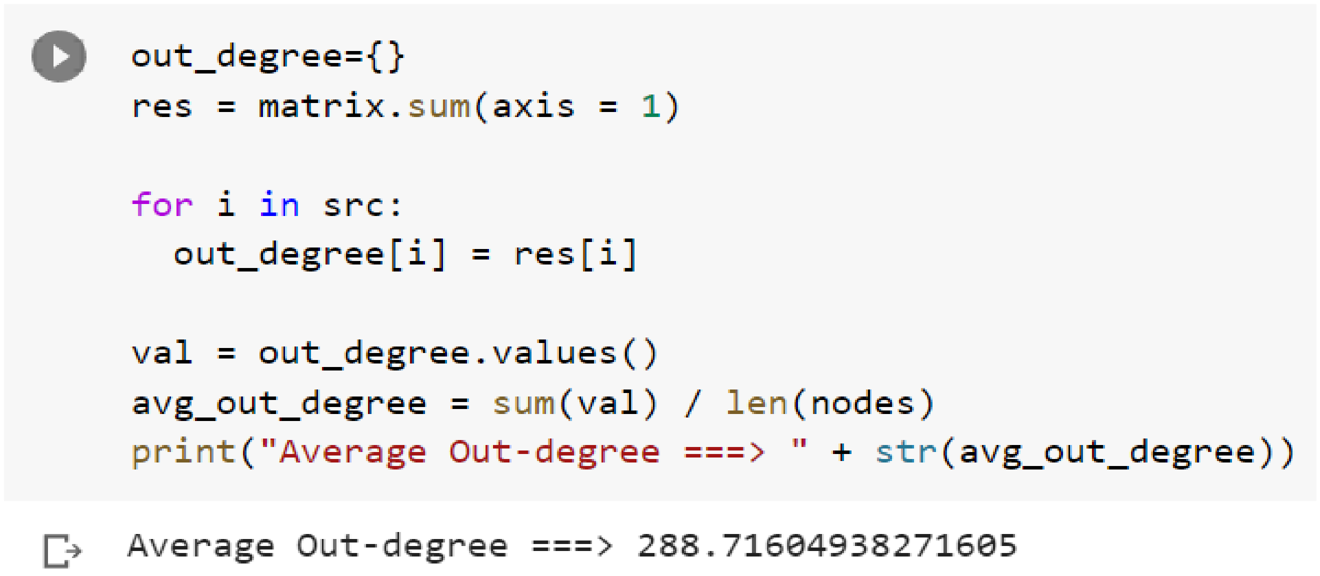
```
Average In-degree ===> 288.71604938271605
```

To calculate average in-degree, we first need to know the number of nodes in the graph. This we have calculated in 1. Next step is to calculate the in-degree for each node and then do sum of all in-degrees. Finally calculate the average in-degree by substituting the values in the formula.

Note that the average indegree gives you an idea of the typical number of incoming edges for a node in the graph. A high average indegree suggests that the graph is highly interconnected, while a low average indegree suggests that the graph is more sparse.

4. Avg. Out-Degree :

Formula -> Average Outdegree = Total Outdegree / Number of Nodes



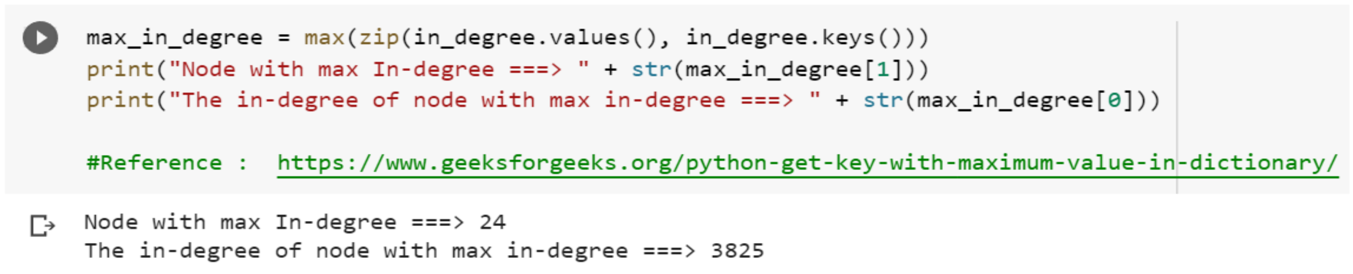
```
out_degree={}
res = matrix.sum(axis = 1)

for i in src:
  out_degree[i] = res[i]

val = out_degree.values()
avg_out_degree = sum(val) / len(nodes)
print("Average Out-degree ===> " + str(avg_out_degree))
```

```
Average Out-degree ===> 288.71604938271605
```

To calculate average out-degree, we first need to know the number of nodes in the graph. This we have calculated in 1. Next step is to calculate the out-degree for each node and then do sum of all out-degrees. Finally calculate the average out-degree by substituting the values in the formula.

5. Node with Max In-degree :

Formula -> Max(in_degree.values)
from all the nodes having indegree… return the node with max indegree and also its value



```
max_in_degree = max(zip(in_degree.values(), in_degree.keys()))
print("Node with max In-degree ===> " + str(max_in_degree[1]))
print("The in-degree of node with max in-degree ===> " + str(max_in_degree[0]))

#Reference :  https://www.geeksforgeeks.org/python-get-key-with-maximum-value-in-dictionary/
```

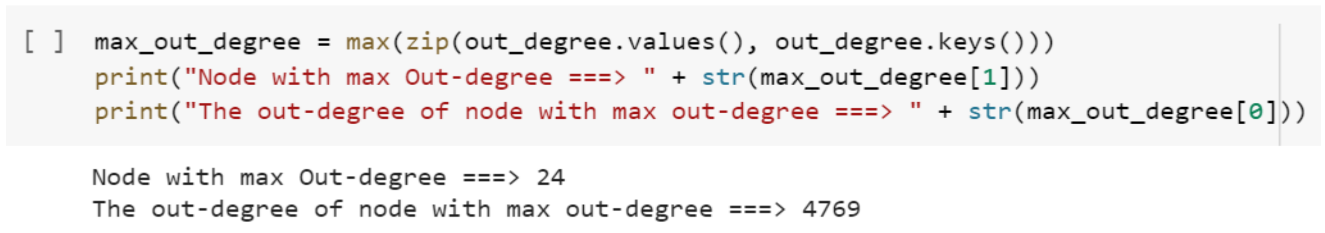
```
Node with max In-degree ===> 24
The in-degree of node with max in-degree ===> 3825
```

From 3 we can take the in-degree for each node. Now we need to check the list which is maximum and print that node. In addition to that we also printed the in-degree of the maximum in-degree node.

Note that the average outdegree gives you an idea of the typical number of outgoing edges for a node in the graph. A high average outdegree suggests that the graph is highly connected, while a low average outdegree suggests that the graph is more sparse.

6. Node with Max out-degree :

Formula -> **from all the nodes having out degree… return the node with max out degree and also its value**



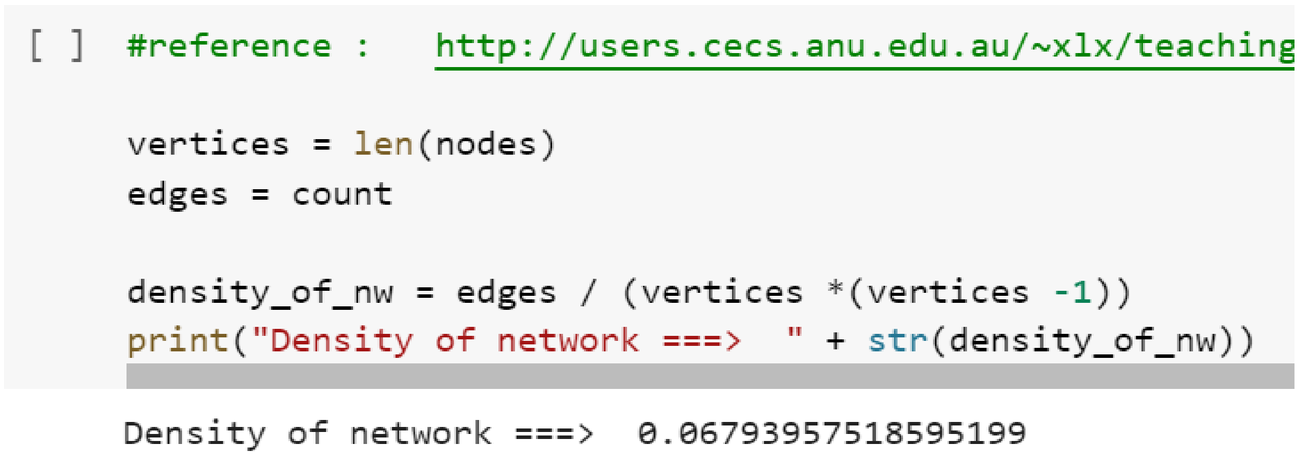
```
[ ]  max_out_degree = max(zip(out_degree.values(), out_degree.keys()))
     print("Node with max Out-degree ===> " + str(max_out_degree[1]))
     print("The out-degree of node with max out-degree ===> " + str(max_out_degree[0]))
```

```
Node with max Out-degree ===> 24
The out-degree of node with max out-degree ===> 4769
```

From 4 we can take the out-degree for each node. Now we need to check the list which is maximum and print that node. In addition to that we also printed the out-degree of the maximum out-degree node.

7. The density of the network :

Formula -> **Density = Number of Edges / Number of Possible Edges**



```
[ ]  #reference :   http://users.cecs.anu.edu.au/~xlx/teaching

     vertices = len(nodes)
     edges = count

     density_of_nw = edges / (vertices *(vertices -1))
     print("Density of network ===>  " + str(density_of_nw))

     Density of network ===>  0.06793957518595199
```

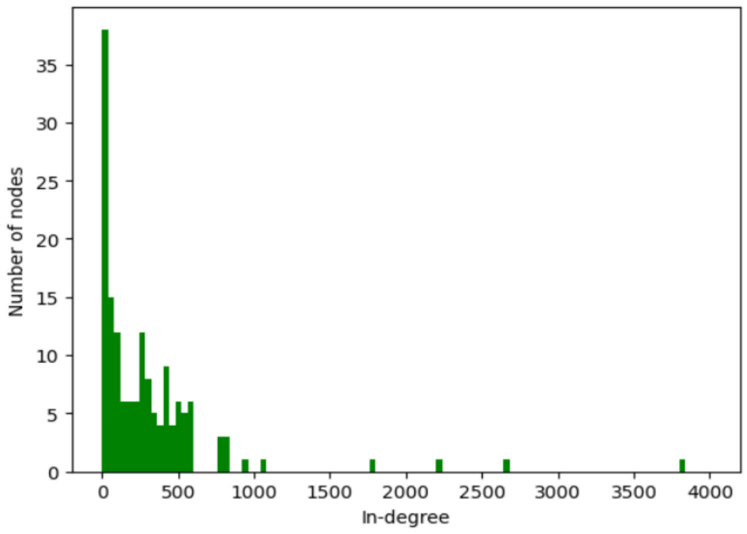
To calculate the density of the network, we need to count the number of edges in the graph which we can get from 2. Then count the number of possible number of edges in the graph. The maximum number of edges in a directed graph is n*(n-1) where n is the number of nodes in the graph and for undirected graph the maximum number of edges is n*(n-1)/2. Now substitute the values in the formula and calculate the density of the network.

The density of a network can range from 0 (for a completely disconnected graph) to 1 (for a fully connected graph). A higher density indicates a more tightly connected network, while a lower density suggests a more sparse or loosely connected network.

1. Plot degree distribution of the network (in case of a directed graph, plot in-degree and out-degree separately).

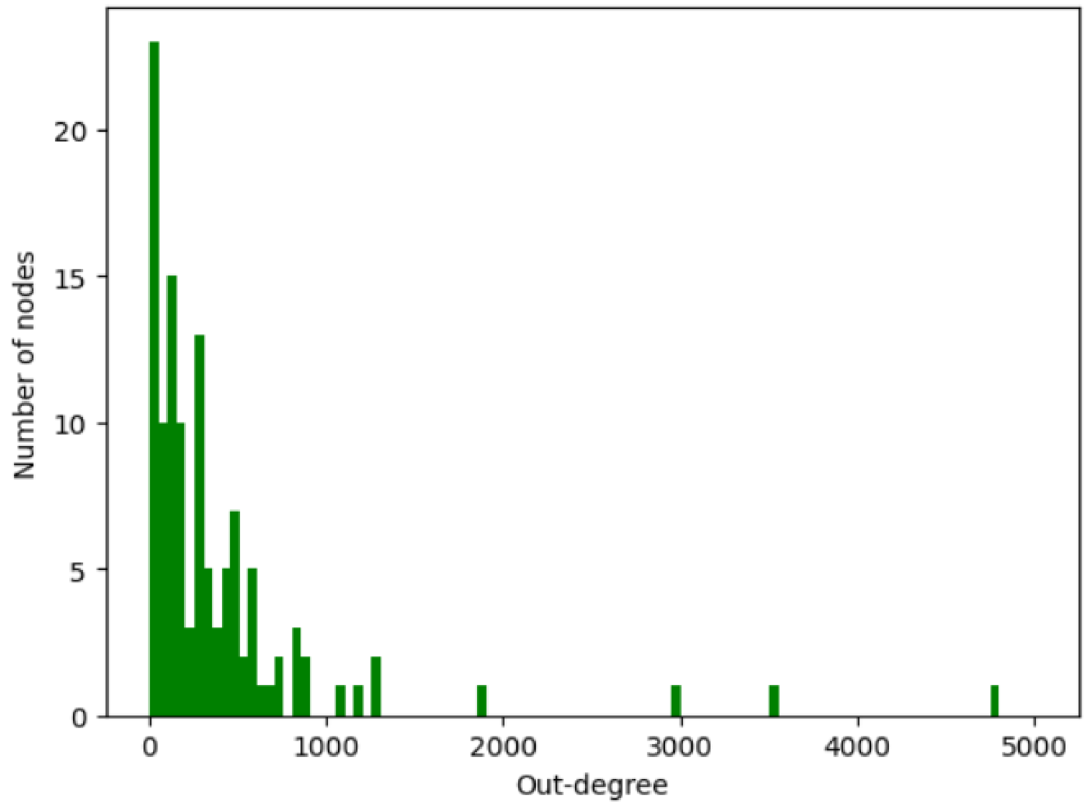
Plot for in-degree distribution ->

```
#reference :   https://mathinsight.org/degree_distribution#:~:text=By%20counting%20how%20many%20nodes,%2C%20and%20k10%3D1.

plt.hist(list(in_degree.values()),bins=100,range=[1,4000],color='green')
plt.xlabel('In-degree')
plt.ylabel('Number of nodes')
plt.show()
```



Plot for out-degree distribution ->

```
plt.hist(list(out_degree.values()),bins=100,range=[10,5000],color='green')
plt.xlabel('Out-degree')
plt.ylabel('Number of nodes')
plt.show()
```

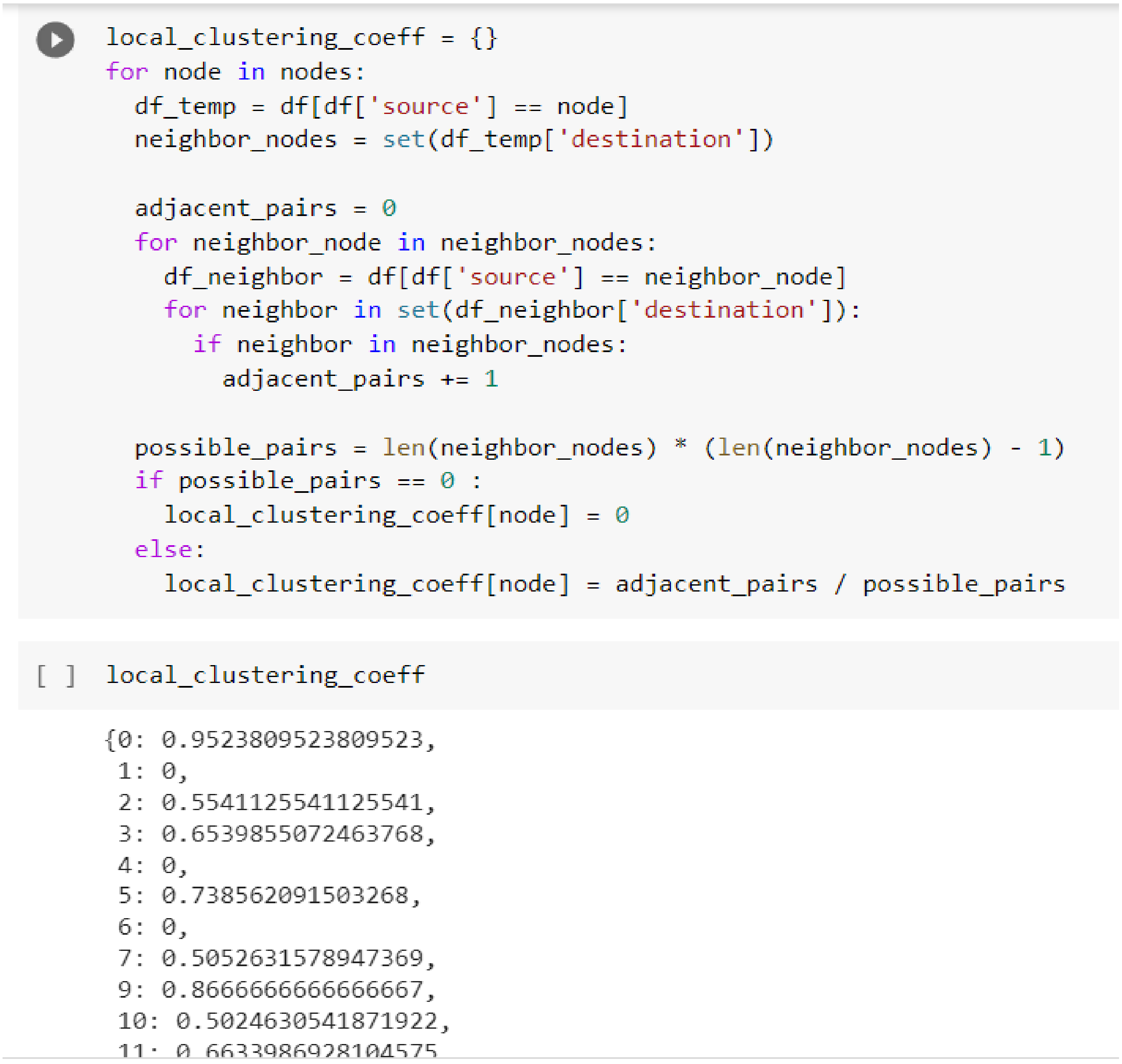


2. [10 points] Calculate the local clustering coefficient of each node and plot the clustering-coefficient distribution (lcc vs frequency of lcc) of the network.

Local clustering coefficient ->

Formula ->

LCC = T / (k * (k - 1))
where:
T is the number of strongly connected triangles that the node is part of.
k is the number of neighbors that the node has (i.e., its in-degree or out-degree, depending on the definition of the LCC).



```
local_clustering_coeff = {}
for node in nodes:
  df_temp = df[df['source'] == node]
  neighbor_nodes = set(df_temp['destination'])

  adjacent_pairs = 0
  for neighbor_node in neighbor_nodes:
    df_neighbor = df[df['source'] == neighbor_node]
    for neighbor in set(df_neighbor['destination']):
      if neighbor in neighbor_nodes:
        adjacent_pairs += 1

  possible_pairs = len(neighbor_nodes) * (len(neighbor_nodes) - 1)
  if possible_pairs == 0 :
    local_clustering_coeff[node] = 0
  else:
    local_clustering_coeff[node] = adjacent_pairs / possible_pairs
```

```
[ ]  local_clustering_coeff
```

```
{0: 0.9523809523809523,
 1: 0,
 2: 0.5541125541125541,
 3: 0.6539855072463768,
 4: 0,
 5: 0.738562091503268,
 6: 0,
 7: 0.5052631578947369,
 9: 0.8666666666666667,
 10: 0.5024630541871922,
 11: 0.6633986928104575
```

Below code is for calculating the frequency of each local clustering coefficient other than 0.

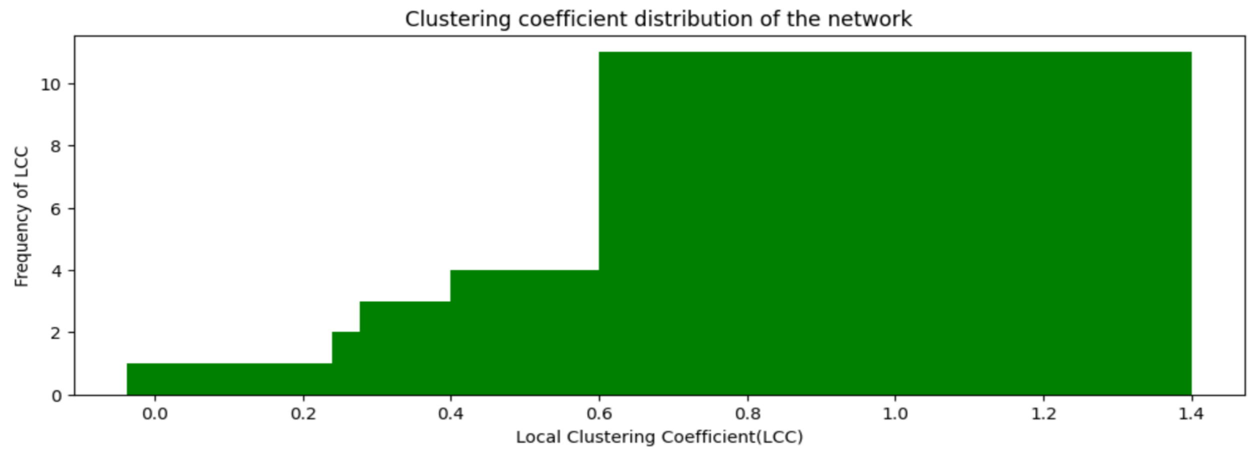
```
[ ]  local_clustering_coeff_freq = {}
     for coefficient in local_clustering_coeff.values():
       if coefficient != 0:
         if coefficient in local_clustering_coeff_freq:
           local_clustering_coeff_freq[coefficient] += 1
         else:
           local_clustering_coeff_freq[coefficient] = 1
```

```
local_clustering_coeff_freq
```

```
{0.9523809523809523: 2,
 0.5541125541125541: 1,
 0.6539855072463768: 1,
 0.738562091503268: 1,
 0.5052631578947369: 1,
 0.8666666666666667: 1,
 0.5024630541871922: 1,
 0.6633986928104575: 1,
 0.7573529411764706: 1,
 0.8636363636363636: 1,
 1.0: 11,
 0.8589743589743589: 1,
```

Plotting the LCC vs Frequency distribution of LCC

```
coeff = list(local_clustering_coeff_freq.keys())
freq = list(local_clustering_coeff_freq.values())
fig = plt.figure(figsize=(12, 4))
plt.bar(coeff, freq, color='green')
plt.title("Clustering coefficient distribution of the network")
plt.xlabel("Local Clustering Coefficient(LCC)")
plt.ylabel("Frequency of LCC")
plt.show()
```



Question 2:

Objective:

1. Taking out the Pagerank score for each of the node from the given preprocessed data from Q-1
2. Taking out the Authority and Hub score for each node.

Part-i….

Load preprocessed dataset from above

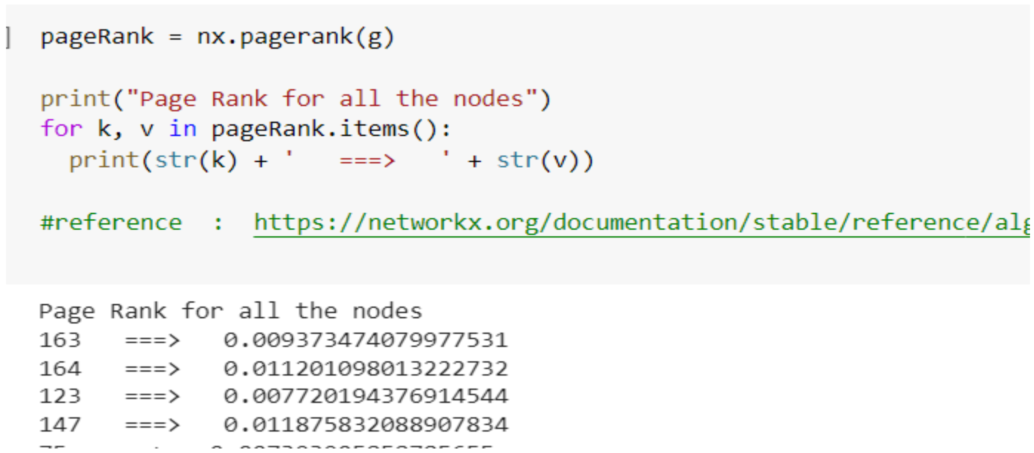
Draw the edges between source and destination using the networx library.

```
[ ]  g = nx.DiGraph()
```

Add edges to the Graph

```
[ ]  for i in range(0, len(df)):
       r_val = df.loc[i, "source"]
       c_val = df.loc[i, "destination"]

       g.add_edge(r_val,c_val)
```

Thus, each source and destination can be treated as node of the transformed graph from the above dataset.



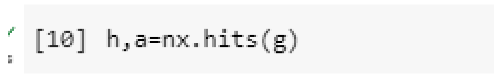
```
] pageRank = nx.pagerank(g)

  print("Page Rank for all the nodes")
  for k, v in pageRank.items():
    print(str(k) + '    ===>    ' + str(v))

  #reference  :  https://networkx.org/documentation/stable/reference/alg
```

```
Page Rank for all the nodes
163   ===>   0.009373474079977531
164   ===>   0.011201098013222732
123   ===>   0.007720194376914544
147   ===>   0.011875832088907834
```

Pagerank of all the nodes are computed using the pagerank function of the networkx library.

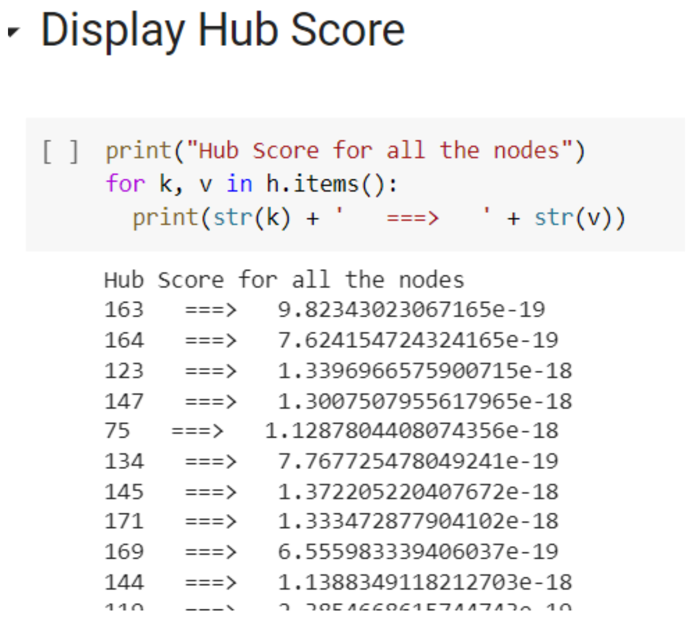
In the above image you can see the node with its corresponding page rank value.

Likewise HubScore and Authority Score of all the nodes are computed using the Hubscore and Authority score function of the networkx library respectively.



```
[10] h,a=nx.hits(g)
```

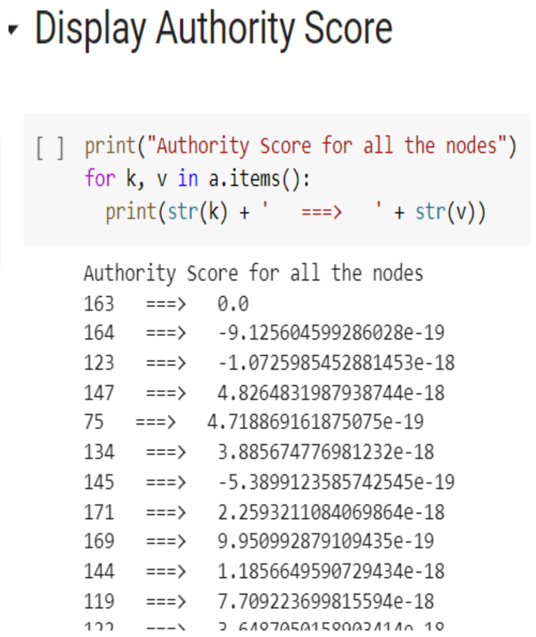
In the below two images, you can see the nodes with their corresponding Hub score and Authority score.



▸ Display Hub Score

```
[ ]  print("Hub Score for all the nodes")
     for k, v in h.items():
       print(str(k) + '    ===>    ' + str(v))
```

```
Hub Score for all the nodes
163    ===>    9.82343023067165e-19
164    ===>    7.624154724324165e-19
123    ===>    1.3396966575900715e-18
147    ===>    1.3007507955617965e-18
75    ===>    1.1287804408074356e-18
134    ===>    7.767725478049241e-19
145    ===>    1.372205220407672e-18
171    ===>    1.333472877904102e-18
169    ===>    6.555983339406037e-19
144    ===>    1.1388349118212703e-18
```

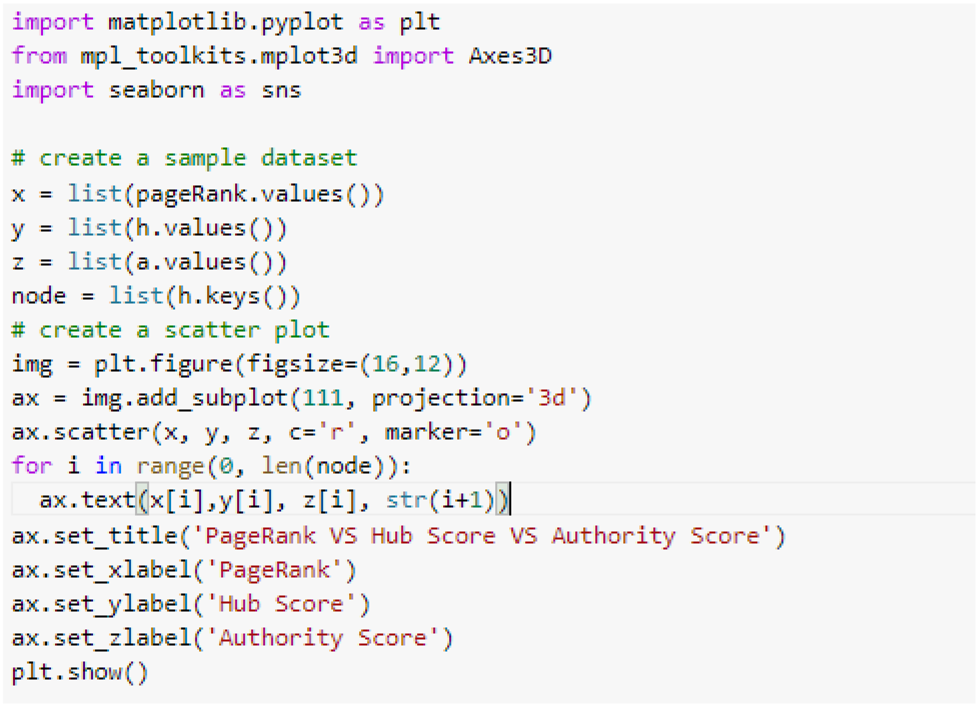


▸ Display Authority Score

```
[ ]  print("Authority Score for all the nodes")
     for k, v in a.items():
       print(str(k) + '    ===>    ' + str(v))
```

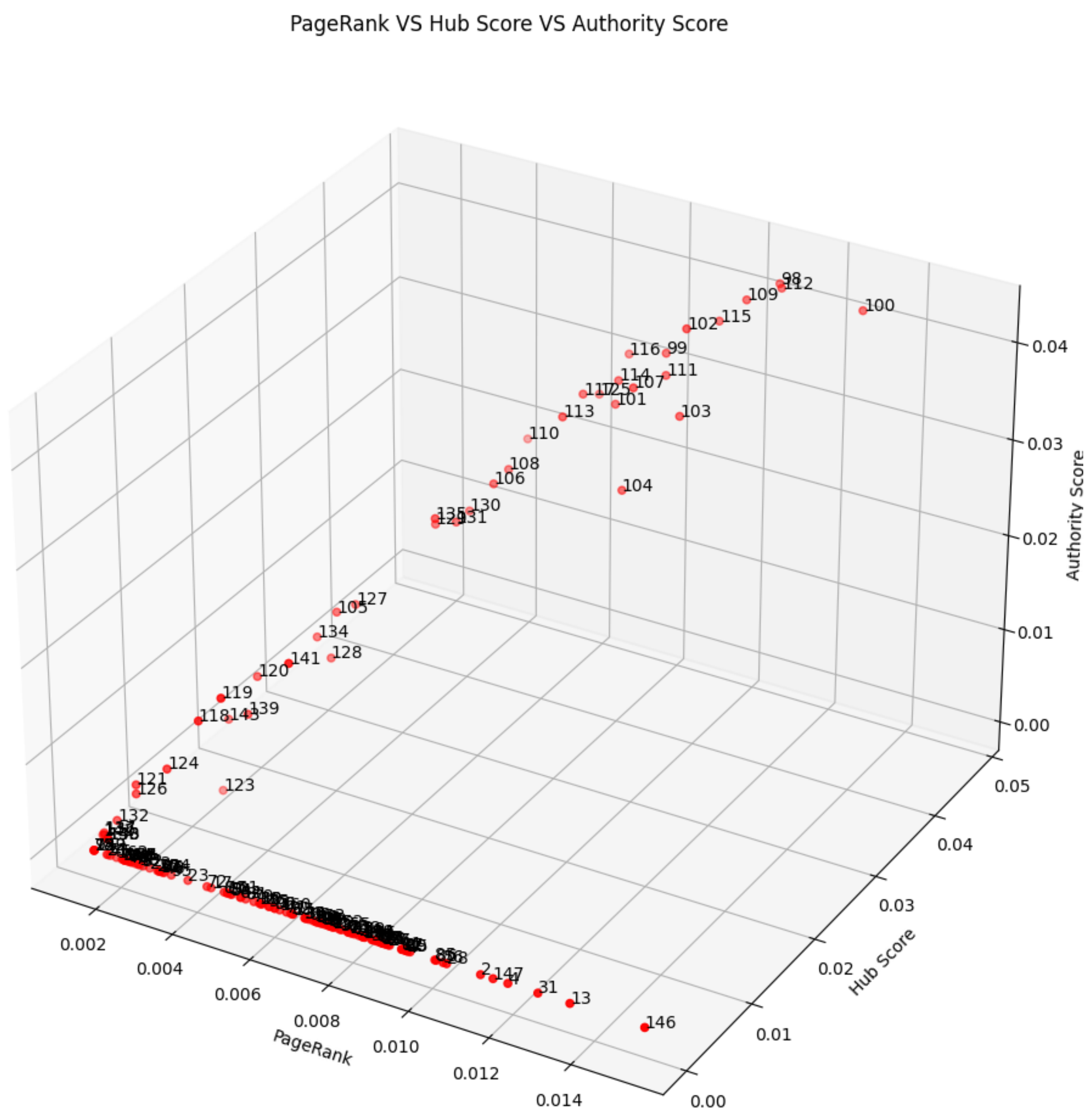
```
Authority Score for all the nodes
163   ===>   0.0
164   ===>   -9.125604599286028e-19
123   ===>   -1.0725985452881453e-18
147   ===>   4.8264831987938744e-18
75   ===>   4.718869161875075e-19
134   ===>   3.885674776981232e-18
145   ===>   -5.3899123585742545e-19
171   ===>   2.2593211084069864e-18
169   ===>   9.950992879109435e-19
144   ===>   1.1856649590729434e-18
119   ===>   7.709223699815594e-18
```

Now we plot a scatterplot to compare each node on the basis of page rank, hub score, and authority score of each node



```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns

# create a sample dataset
x = list(pageRank.values())
y = list(h.values())
z = list(a.values())
node = list(h.keys())
# create a scatter plot
img = plt.figure(figsize=(16,12))
ax = img.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c='r', marker='o')
for i in range(0, len(node)):
  ax.text(x[i],y[i], z[i], str(i+1))
ax.set_title('PageRank VS Hub Score VS Authority Score')
ax.set_xlabel('PageRank')
ax.set_ylabel('Hub Score')
ax.set_zlabel('Authority Score')
plt.show()
```

Comparison result is shown below



Comparison outcome:

PageRank gives the importance of a web page based on the incoming links to the page. **Hub Score** gives the relevance of a web page based on the outgoing links from it. **Authority Score** gives the relevance of a web page based on the incoming links to it. Higher values of the above scores indicate that the page is largely important.

From the comparison we have found that node 100 is more important than other nodes as it has higher scores on all 3 aspects (Page Rank, Hub Score, and Authority Score). Other important nodes are 98, 112, 109, etc