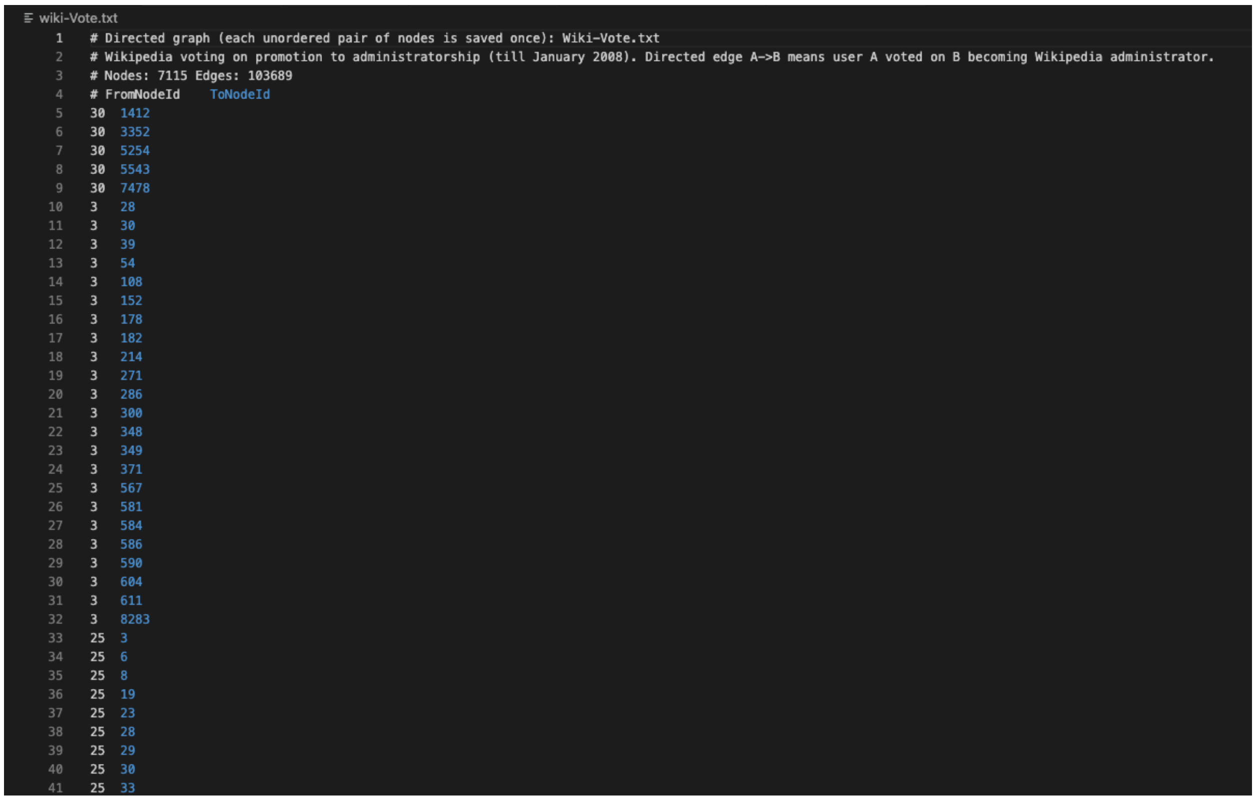
Group Number: 104
Team Members:
1) Akshita Gupta - 2020491
2) Shivam Agrawal - 2020124
3) Yash Kumar - 2020350

IR Assignment 3

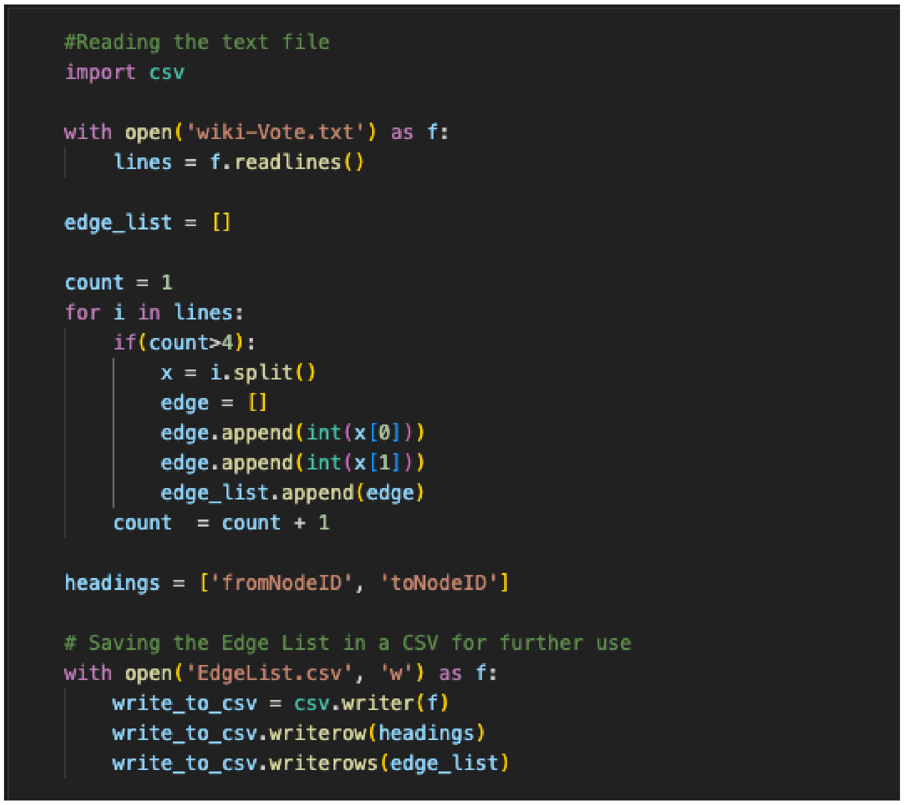
Q1)
Ans1)
For the first question we needed to Pick a real-world directed network dataset containing more than 100 nodes from the given dataset: https://snap.stanford.edu/data/index.html
For this task we selected the **Wikipedia Vote Network** containing 7115 nodes. It is a directed network as requested in the question. Our initial reasoning for selecting this dataset was the lesser number of nodes and edges as compared to the other datasets.

The dataset was given to us in the form:



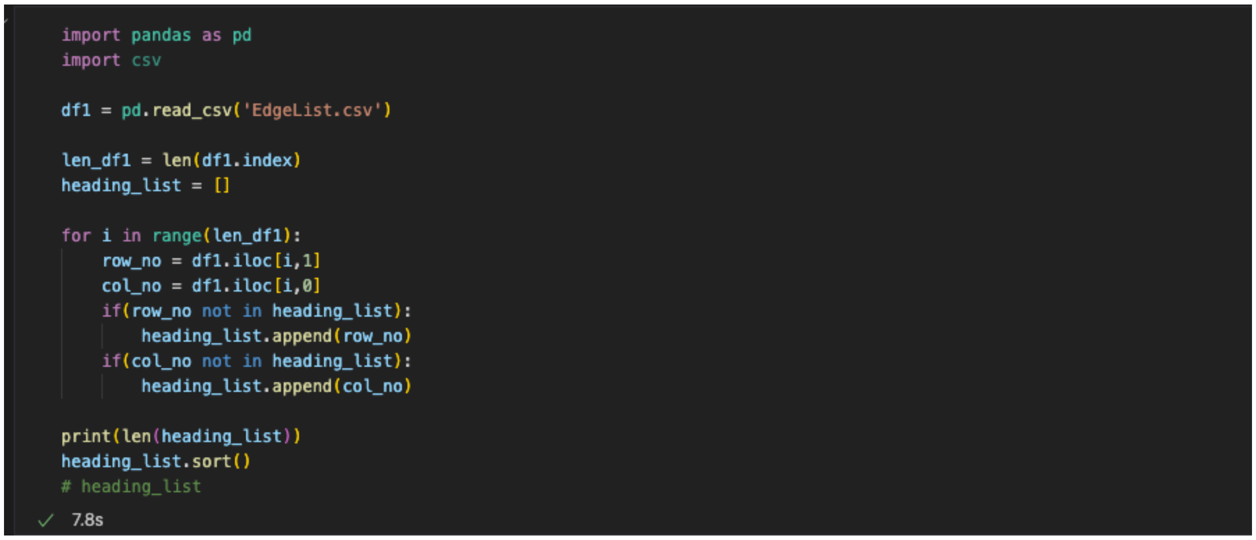
```
wiki-Vote.txt
# Directed graph (each unordered pair of nodes is saved once): Wiki-Vote.txt
# Wikipedia voting on promotion to administratorship (till January 2008). Directed edge A->B means user A voted on B becoming Wikipedia administrator.
# Nodes: 7115 Edges: 103689
# FromNodeId    ToNodeId
30  1412
30  3352
30  5254
30  5543
30  7478
3   28
3   30
3   39
3   54
3   108
3   152
3   178
3   182
3   214
3   271
3   286
3   300
3   348
3   349
3   371
3   567
3   581
3   584
3   586
3   590
3   604
3   611
3   8283
25  3
25  6
25  8
25  19
25  23
25  28
25  29
25  30
25  33
```

This actually is how an edge list would be represented, thus using a small piece of code we converted this text file to an Edge List and saved it in the CSV - ‘EdgeList.csv’ for use in the future.



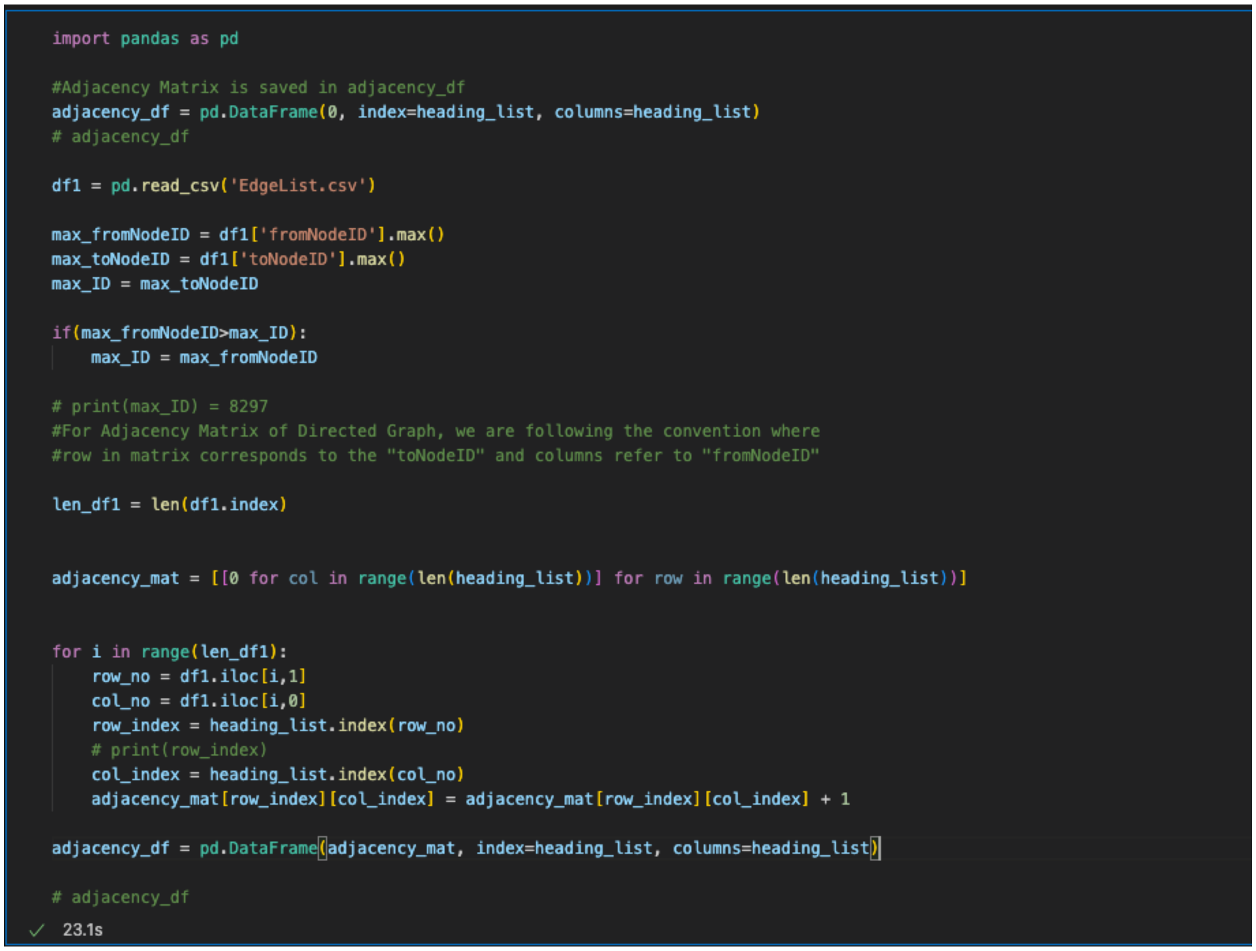
```python
#Reading the text file
import csv

with open('wiki-Vote.txt') as f:
    lines = f.readlines()

edge_list = []

count = 1
for i in lines:
    if(count>4):
        x = i.split()
        edge = []
        edge.append(int(x[0]))
        edge.append(int(x[1]))
        edge_list.append(edge)
    count  = count + 1

headings = ['fromNodeID', 'toNodeID']

# Saving the Edge List in a CSV for further use
with open('EdgeList.csv', 'w') as f:
    write_to_csv = csv.writer(f)
    write_to_csv.writerow(headings)
    write_to_csv.writerows(edge_list)
```

Although the dataset has 7115 nodes, the maximum NodeID is different. To find the unique and accurate NodeIDs, we run a for loop through the edge list and store the unique NodeIDs in a new list.



```python
import pandas as pd
import csv

df1 = pd.read_csv('EdgeList.csv')

len_df1 = len(df1.index)
heading_list = []

for i in range(len_df1):
    row_no = df1.iloc[i,1]
    col_no = df1.iloc[i,0]
    if(row_no not in heading_list):
        heading_list.append(row_no)
    if(col_no not in heading_list):
        heading_list.append(col_no)

print(len(heading_list))
heading_list.sort()
# heading_list
```
```
✓ 7.8s
```

Once we have the heading list with us, we now start creating the Adjacency Matrix for only the relevant NodeIDs. We initialize a zero matrix of size 7115*7115. For the Adjacency Matrix of a Directed Graph, we are following the convention where the row in the matrix refers to the “toNodeID” and the column refers to “fromNodeID”. We iterate through the edge list and fill the adjacency matrix. And convert this matrix into a DataFrame, so it is easier to use.

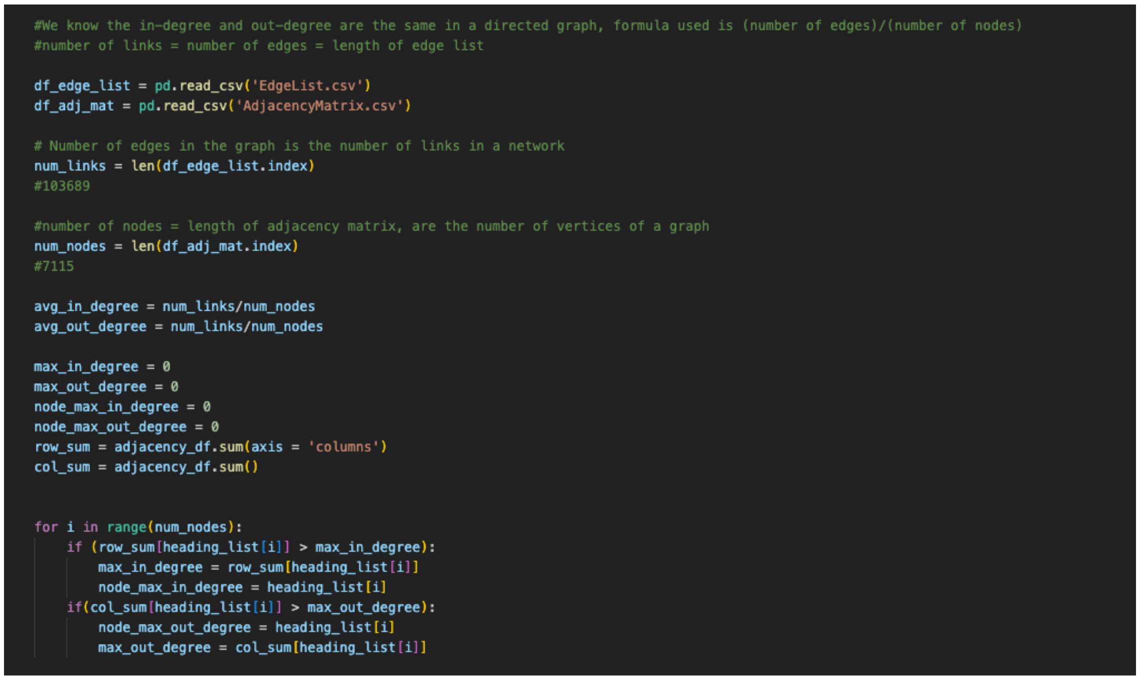


```
import pandas as pd

#Adjacency Matrix is saved in adjacency_df
adjacency_df = pd.DataFrame(0, index=heading_list, columns=heading_list)
# adjacency_df

df1 = pd.read_csv('EdgeList.csv')

max_fromNodeID = df1['fromNodeID'].max()
max_toNodeID = df1['toNodeID'].max()
max_ID = max_toNodeID

if(max_fromNodeID>max_ID):
    max_ID = max_fromNodeID

# print(max_ID) = 8297
#For Adjacency Matrix of Directed Graph, we are following the convention where
#row in matrix corresponds to the "toNodeID" and columns refer to "fromNodeID"

len_df1 = len(df1.index)


adjacency_mat = [[0 for col in range(len(heading_list))] for row in range(len(heading_list))]


for i in range(len_df1):
    row_no = df1.iloc[i,1]
    col_no = df1.iloc[i,0]
    row_index = heading_list.index(row_no)
    # print(row_index)
    col_index = heading_list.index(col_no)
    adjacency_mat[row_index][col_index] = adjacency_mat[row_index][col_index] + 1

adjacency_df = pd.DataFrame(adjacency_mat, index=heading_list, columns=heading_list)

# adjacency_df
✓ 23.1s
```

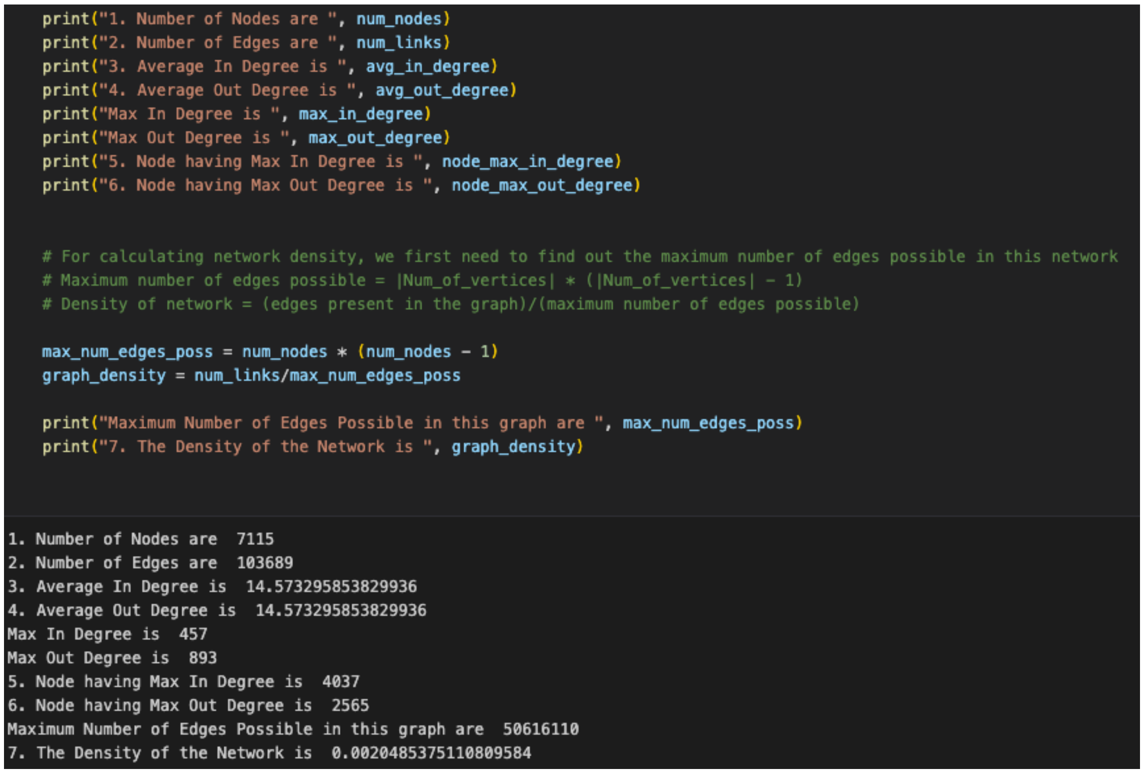
We have finally created the Adjacency Matrix and Edge List as requested in the question. Now we get onto answering the questions asked.

1. Number of Nodes = 7115
   Given in the dataset, also calculated after finding the unique nodes.
2. Number of Edges = 1,03,689
   Given in the dataset, also calculated as the length of the edge list.
3. Average In-Degree = 14.573295853829936
   Formula Used = (Number of Edges)/(Number of Nodes)
4. Average Out-Degree = 14.573295853829936
   Same as Average In-Degree
5. Node having Max In-Degree = 4037
   To find this node, we find the sum of each row in the Adjacency Matrix and then return the row with the maximum sum.
6. Node having Max Out-Degree = 2565
   To find this node, we find the sum of each column in the Adjacency Matrix and then return the column with the maximum sum.
7. The Density of the Network = 0.0020485375110809584

Formula Used = (Total Number of Edges existing)/(Total Number of Edges Possible)
To find Total Number of Edges Possible we used the formula
= Number of Vertices * (Number of Vertices - 1)
We used this formula because the graph is a directed graph and thus all the edges will be counted as unique edges.



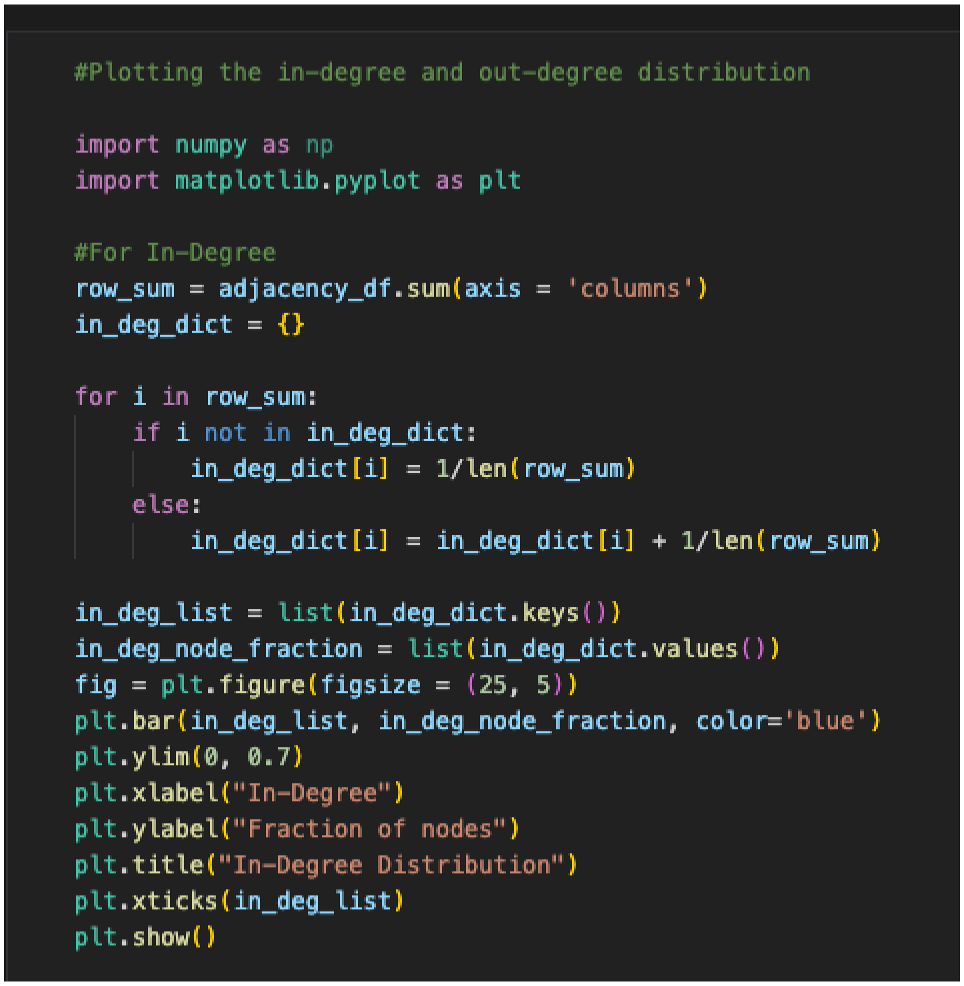
```python
#We know the in-degree and out-degree are the same in a directed graph, formula used is (number of edges)/(number of nodes)
#number of links = number of edges = length of edge list

df_edge_list = pd.read_csv('EdgeList.csv')
df_adj_mat = pd.read_csv('AdjacencyMatrix.csv')

# Number of edges in the graph is the number of links in a network
num_links = len(df_edge_list.index)
#103689

#number of nodes = length of adjacency matrix, are the number of vertices of a graph
num_nodes = len(df_adj_mat.index)
#7115

avg_in_degree = num_links/num_nodes
avg_out_degree = num_links/num_nodes

max_in_degree = 0
max_out_degree = 0
node_max_in_degree = 0
node_max_out_degree = 0
row_sum = adjacency_df.sum(axis = 'columns')
col_sum = adjacency_df.sum()


for i in range(num_nodes):
    if (row_sum[heading_list[i]] > max_in_degree):
        max_in_degree = row_sum[heading_list[i]]
        node_max_in_degree = heading_list[i]
    if(col_sum[heading_list[i]] > max_out_degree):
        node_max_out_degree = heading_list[i]
        max_out_degree = col_sum[heading_list[i]]
```



```python
print("1. Number of Nodes are ", num_nodes)
print("2. Number of Edges are ", num_links)
print("3. Average In Degree is ", avg_in_degree)
print("4. Average Out Degree is ", avg_out_degree)
print("Max In Degree is ", max_in_degree)
print("Max Out Degree is ", max_out_degree)
print("5. Node having Max In Degree is ", node_max_in_degree)
print("6. Node having Max Out Degree is ", node_max_out_degree)


# For calculating network density, we first need to find out the maximum number of edges possible in this network
# Maximum number of edges possible = |Num_of_vertices| * (|Num_of_vertices| - 1)
# Density of network = (edges present in the graph)/(maximum number of edges possible)

max_num_edges_poss = num_nodes * (num_nodes - 1)
graph_density = num_links/max_num_edges_poss

print("Maximum Number of Edges Possible in this graph are ", max_num_edges_poss)
print("7. The Density of the Network is ", graph_density)
```

```
1. Number of Nodes are  7115
2. Number of Edges are  103689
3. Average In Degree is  14.573295853829936
4. Average Out Degree is  14.573295853829936
Max In Degree is  457
Max Out Degree is  893
5. Node having Max In Degree is  4037
6. Node having Max Out Degree is  2565
Maximum Number of Edges Possible in this graph are  50616110
7. The Density of the Network is  0.0020485375110809584
```

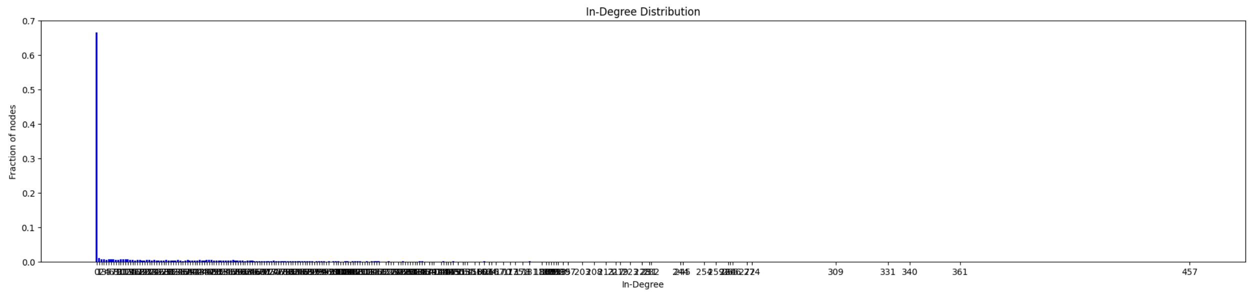
For plotting the In-Degree and Out-Degree Distribution we have taken the frequency of occurrence of the degree on the y-axis and the value of degree on the x-axis. For finding the In-Degree distribution we find the sum of each row, as the row in Adjacency Matrix corresponds to the “toNodeID”. We store the sum of each row in another list called “row_sum”. We created a dictionary to store the degree as key and its frequency as value. We traverse the row_sum list and fill the dictionary using it. We store the keys and values of the dictionary in another list and use them to plot the In-Degree Distribution.



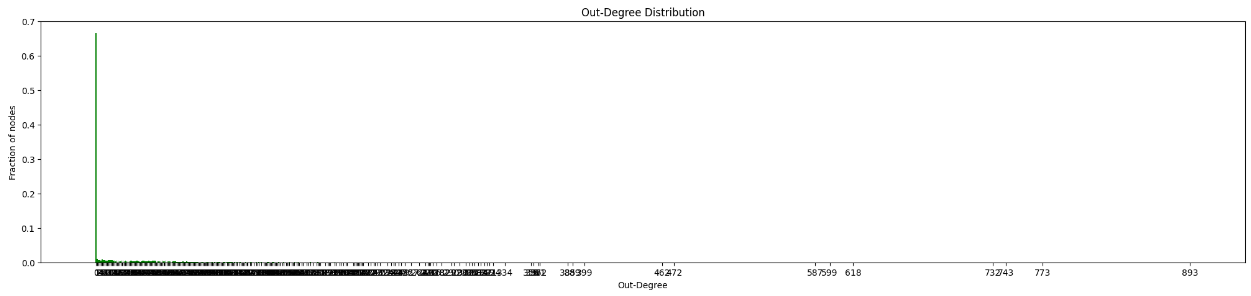
```
#Plotting the in-degree and out-degree distribution

import numpy as np
import matplotlib.pyplot as plt

#For In-Degree
row_sum = adjacency_df.sum(axis = 'columns')
in_deg_dict = {}

for i in row_sum:
    if i not in in_deg_dict:
        in_deg_dict[i] = 1/len(row_sum)
    else:
        in_deg_dict[i] = in_deg_dict[i] + 1/len(row_sum)

in_deg_list = list(in_deg_dict.keys())
in_deg_node_fraction = list(in_deg_dict.values())
fig = plt.figure(figsize = (25, 5))
plt.bar(in_deg_list, in_deg_node_fraction, color='blue')
plt.ylim(0, 0.7)
plt.xlabel("In-Degree")
plt.ylabel("Fraction of nodes")
plt.title("In-Degree Distribution")
plt.xticks(in_deg_list)
plt.show()
```

Similar code was used to plot the Out-Degree, in place of summing the row, we find the sum of the column. Frequency is calculated by finding the total number of times an In-Degree or Out-Degree exists divided by the total number of nodes.

In-Degree Distribution:

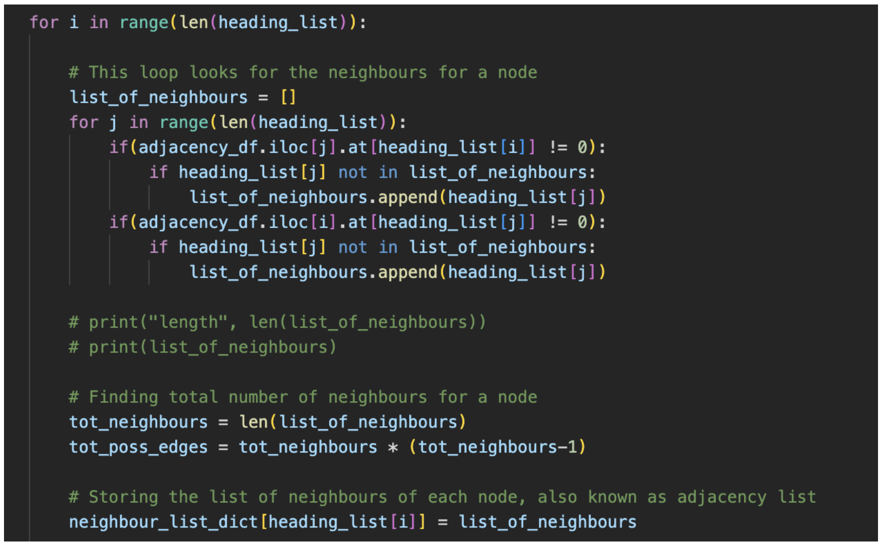


Out-Degree Distribution:

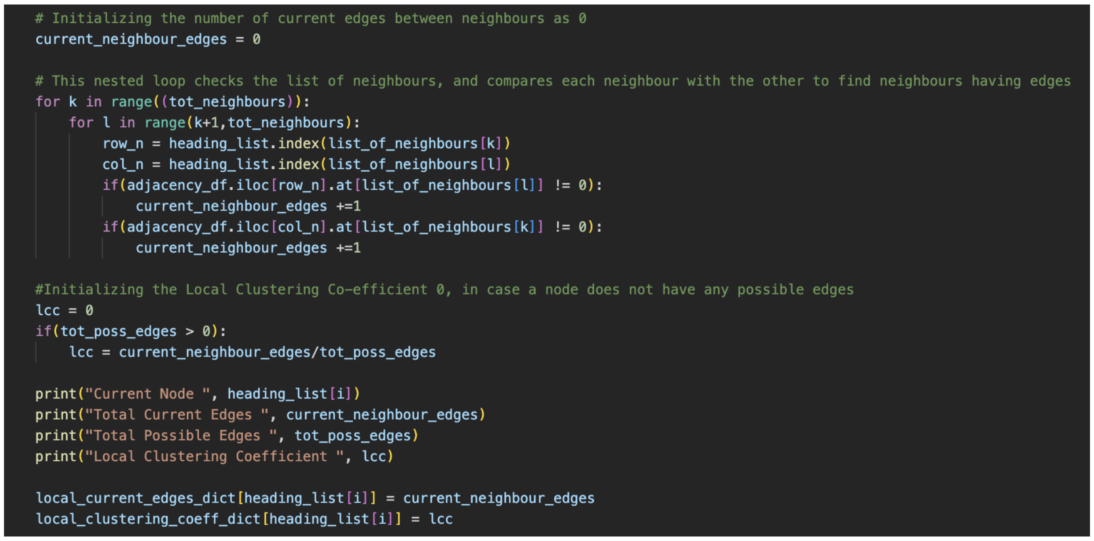


Calculating Local Clustering Co-efficient:

For calculating the local clustering co-efficient for a node, we first need to find the list of neighbors for that node. Using that list of neighbors, we first check for the number of edges existing between those neighbors. We have done that using a nested for loop of complexity O(len(list_of_neighbors)^2). Once we find the number of edges existing between neighbors, we find the maximum number of edges possible between these neighbors. As it is a Directed Graph, the maximum number of edges possible are equal to (Number of Neighbors)*(Number of Neighbors - 1). And finally the Local Clustering Co-efficient is calculated using the formula = (Number of Edges Existing Between Neighbors)/(Maximum Number of Edges Possible). This is stored in a separate dictionary to find the frequency of each LCC value.

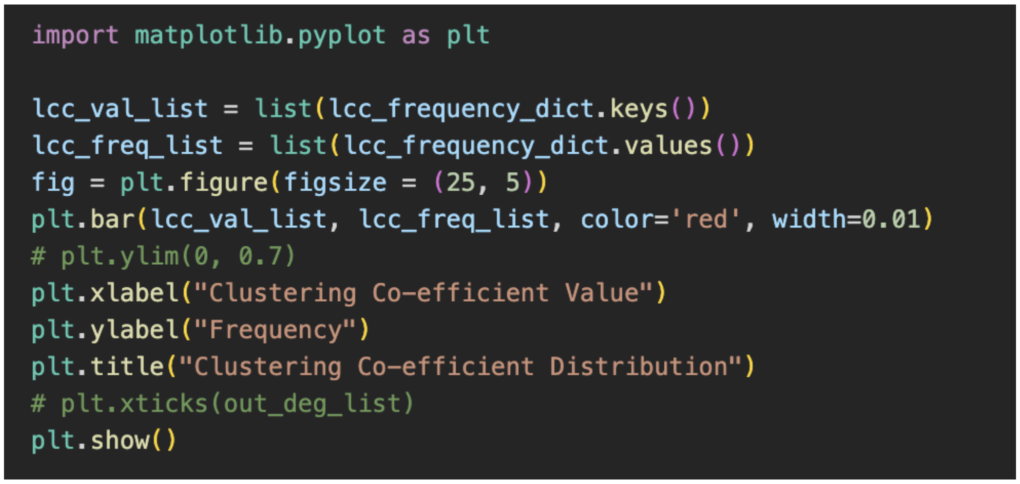


```python
for i in range(len(heading_list)):

    # This loop looks for the neighbours for a node
    list_of_neighbours = []
    for j in range(len(heading_list)):
        if(adjacency_df.iloc[j].at[heading_list[i]] != 0):
            if heading_list[j] not in list_of_neighbours:
                list_of_neighbours.append(heading_list[j])
        if(adjacency_df.iloc[i].at[heading_list[j]] != 0):
            if heading_list[j] not in list_of_neighbours:
                list_of_neighbours.append(heading_list[j])

    # print("length", len(list_of_neighbours))
    # print(list_of_neighbours)

    # Finding total number of neighbours for a node
    tot_neighbours = len(list_of_neighbours)
    tot_poss_edges = tot_neighbours * (tot_neighbours-1)

    # Storing the list of neighbours of each node, also known as adjacency list
    neighbour_list_dict[heading_list[i]] = list_of_neighbours
```



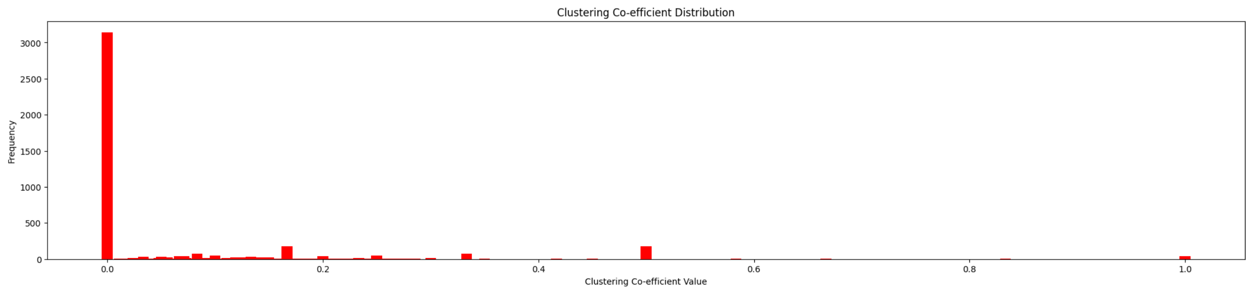
```python
# Initializing the number of current edges between neighbours as 0
current_neighbour_edges = 0

# This nested loop checks the list of neighbours, and compares each neighbour with the other to find neighbours having edges
for k in range((tot_neighbours)):
    for l in range(k+1,tot_neighbours):
        row_n = heading_list.index(list_of_neighbours[k])
        col_n = heading_list.index(list_of_neighbours[l])
        if(adjacency_df.iloc[row_n].at[list_of_neighbours[l]] != 0):
            current_neighbour_edges +=1
        if(adjacency_df.iloc[col_n].at[list_of_neighbours[k]] != 0):
            current_neighbour_edges +=1

#Initializing the Local Clustering Co-efficient 0, in case a node does not have any possible edges
lcc = 0
if(tot_poss_edges > 0):
    lcc = current_neighbour_edges/tot_poss_edges

print("Current Node ", heading_list[i])
print("Total Current Edges ", current_neighbour_edges)
print("Total Possible Edges ", tot_poss_edges)
print("Local Clustering Coefficient ", lcc)

local_current_edges_dict[heading_list[i]] = current_neighbour_edges
local_clustering_coeff_dict[heading_list[i]] = lcc
```

Plotting the Graph:



```python
import matplotlib.pyplot as plt

lcc_val_list = list(lcc_frequency_dict.keys())
lcc_freq_list = list(lcc_frequency_dict.values())
fig = plt.figure(figsize = (25, 5))
plt.bar(lcc_val_list, lcc_freq_list, color='red', width=0.01)
# plt.ylim(0, 0.7)
plt.xlabel("Clustering Co-efficient Value")
plt.ylabel("Frequency")
plt.title("Clustering Co-efficient Distribution")
# plt.xticks(out_deg_list)
plt.show()
```

<u>Clustering Co-efficient Distribution Graph:</u>



Q2)
Ans2) In this question, we use the NetworkX library of Python to analyze a directed graph and calculate the PageRank and HITS (Hypertext Induced Topic Selection) scores for each node. The input graph is read from a file named "edited_wiki-Vote.txt", which contains pairs of nodes separated by whitespace indicating directed edges in the graph.

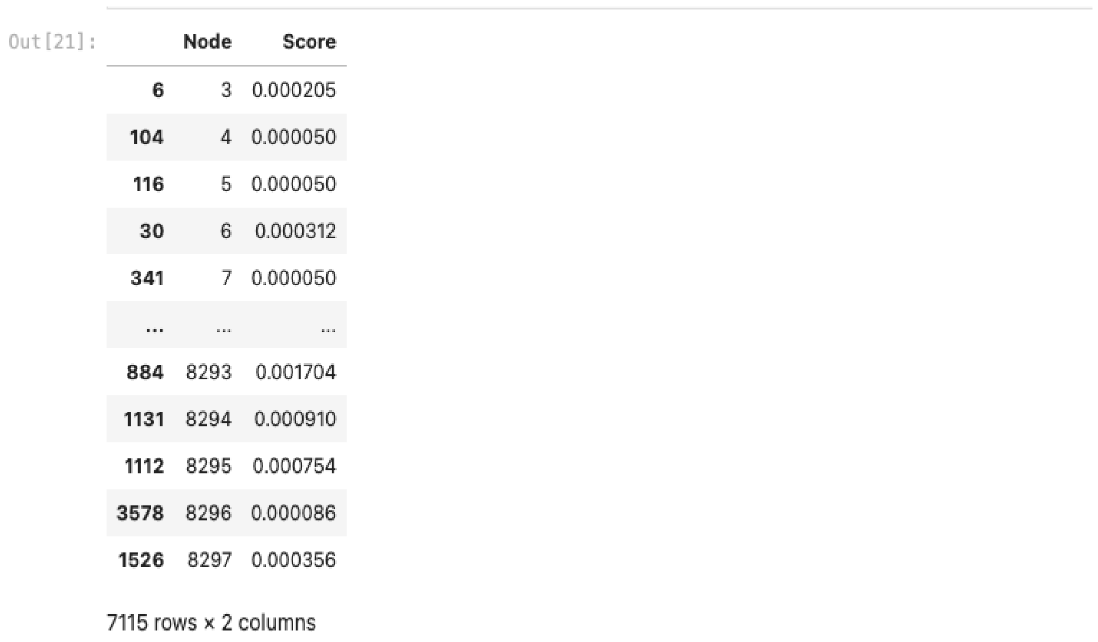
First, a directed graph G is created using the DiGraph() method from the NetworkX library.

Next, the pagerank() method of the nx module in NetworkX is used to calculate the PageRank score for each node in the graph G. These scores are then added to a Pandas DataFrame pageranks with two columns, "Node" and "Score", and saved to a CSV file as **"pageranks.csv"**.

The hits() method of the nx module is used to calculate the HITS scores for each node in the graph G. These scores are then saved in a CSV file as **"hits.csv".**

Here are the outputs of the code written for part 1 and 2-

1. **PageRank Scores**

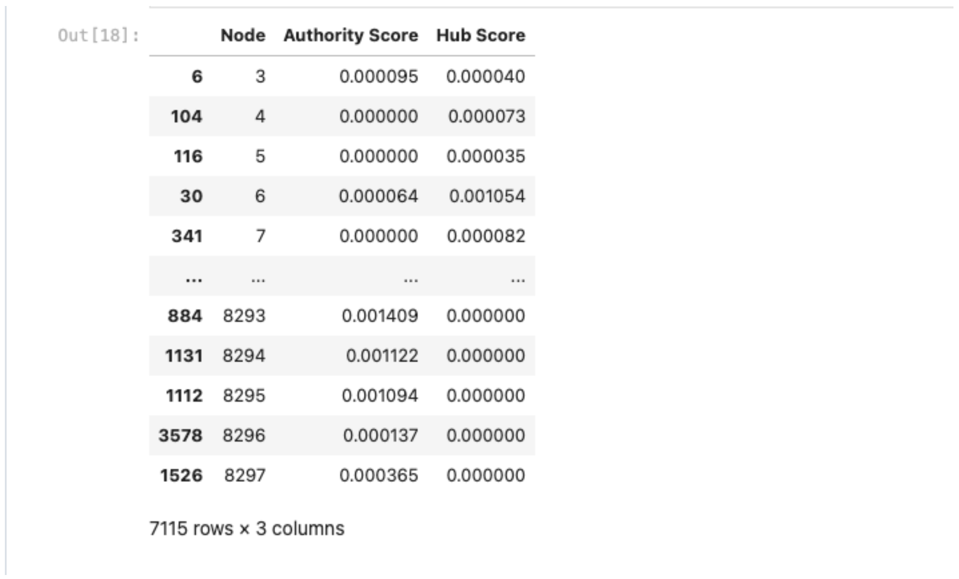


Out[21]:

	Node	Score
6	3	0.000205
104	4	0.000050
116	5	0.000050
30	6	0.000312
341	7	0.000050
...	...	...
884	8293	0.001704
1131	8294	0.000910
1112	8295	0.000754
3578	8296	0.000086
1526	8297	0.000356

7115 rows × 2 columns

2. **Authority and Hub Scores**



Out[18]:

	Node	Authority Score	Hub Score
6	3	0.000095	0.000040
104	4	0.000000	0.000073
116	5	0.000000	0.000035
30	6	0.000064	0.001054
341	7	0.000000	0.000082
...	...	...	...
884	8293	0.001409	0.000000
1131	8294	0.001122	0.000000
1112	8295	0.001094	0.000000
3578	8296	0.000137	0.000000
1526	8297	0.000365	0.000000

7115 rows × 3 columns

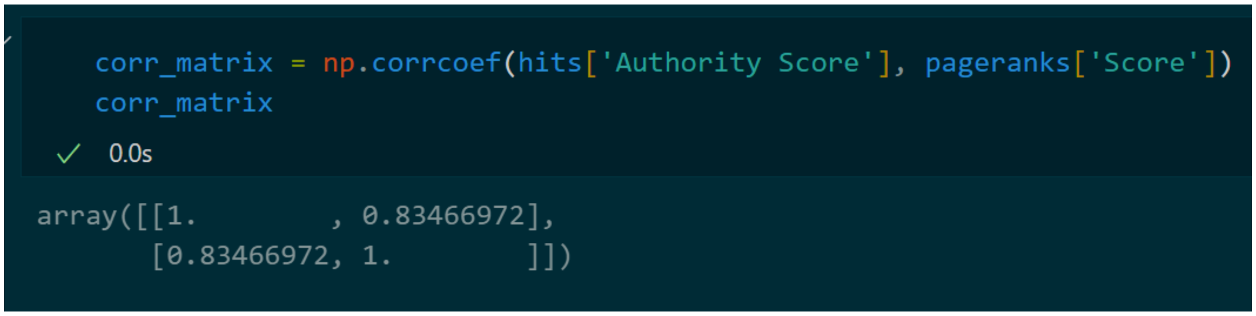
Results Comparison

Both the PageRank algorithm and HITS algorithm calculate scores for each node in a graph, but they do so in different ways.

A node with a high Hub score is considered a good hub because it has many outgoing links that point to other nodes in the graph. Whereas, Pagerank and Authority algorithms, assign scores based on the incoming links to that node.

A comparison between the two scoring schemes can be done by calculating the correlation coefficient.

Correlation coefficient between pagerank score and authority score is 0.83, which is decently high and signifies that both algorithms give similar scores to the nodes, which is what we expected, as both use incoming links to calculate the scores.

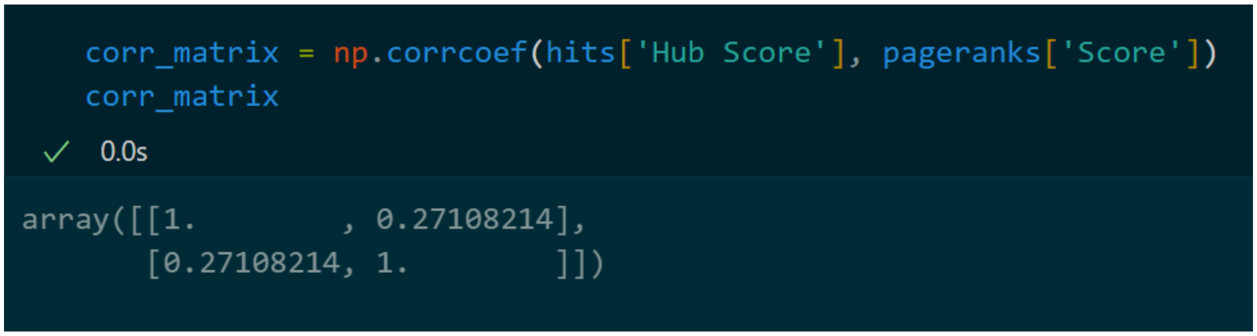


```
corr_matrix = np.corrcoef(hits['Authority Score'], pageranks['Score'])
corr_matrix
```

✓ 0.0s

```
array([[1.        , 0.83466972],
       [0.83466972, 1.        ]])
```

Correlation coefficient between pagerank score and hub score is 0.27, which as expected must be low, as the former uses inlinks while the latter uses outlinks to assign the scores, hence they both evaluate the nodes on different criterias.



```
corr_matrix = np.corrcoef(hits['Hub Score'], pageranks['Score'])
corr_matrix
```

✓ 0.0s

```
array([[1.        , 0.27108214],
       [0.27108214, 1.        ]])
```