GITHUB LINK:- https://github.com/shail10/SNA

Team Name:- INVOKER

Shail Kardani - 19UCS217

Kush Gandhi - 19UCS131

Gunjan Paghdar - 19UCS101

Pratik Gupta - 19UCS047

SNA Project Round - 1

For this project we have selected two datasets(Graphs). The google web graph which is a directed graph where Nodes represents web pages and directed edges represents hyperlinks between them. The second graph is the Facebook graph where the dataset consists of circles (or Friend list) from Facebook.

Libraries Used :-

```
In [1]: import numpy as np
        import pandas as pd
        import networkx as nx
        import matplotlib.pyplot as plt
        %matplotlib inline
        import random
```

Majorly we have used the Networkx library for creating graphs and printing information regarding graphs.

Reading CSV files and printing basic information regarding Graphs:-

```
In [2]: facebook = pd.read_csv('facebook.csv',header=None,names=["source","target"])
        google = pd.read_csv('google.csv',header=None,names=["source","target"])
```

```
In [3]: facebook.head()
```

Out[3]:

	source	target
0	0	1
1	0	2
2	0	3
3	0	4
4	0	5

```
In [4]: google.head()
```

Out[4]:

	source	target
0	0	11342
1	0	824020
2	0	867923
3	0	891835
4	11342	0

As we can see we have source nodes and target nodes for creating graphs.

Creating graphs:-

```
In [7]: graph_facebook = nx.Graph()
        for ind in facebook.index:
          graph_facebook.add_edge(facebook['source'][ind], facebook['target'][ind])
```

```
In [8]: graph_google = nx.read_edgelist('web-Google.txt', create_using = nx.DiGraph)
        # print(nx.info(G_tmp))
```

Creating google graph as Directed Graph

```
In [9]: nx.info(graph_facebook)
Out[9]: 'Graph with 4039 nodes and 88234 edges'

In [10]: nx.info(graph_google)
Out[10]: 'DiGraph with 875713 nodes and 5105039 edges'
```

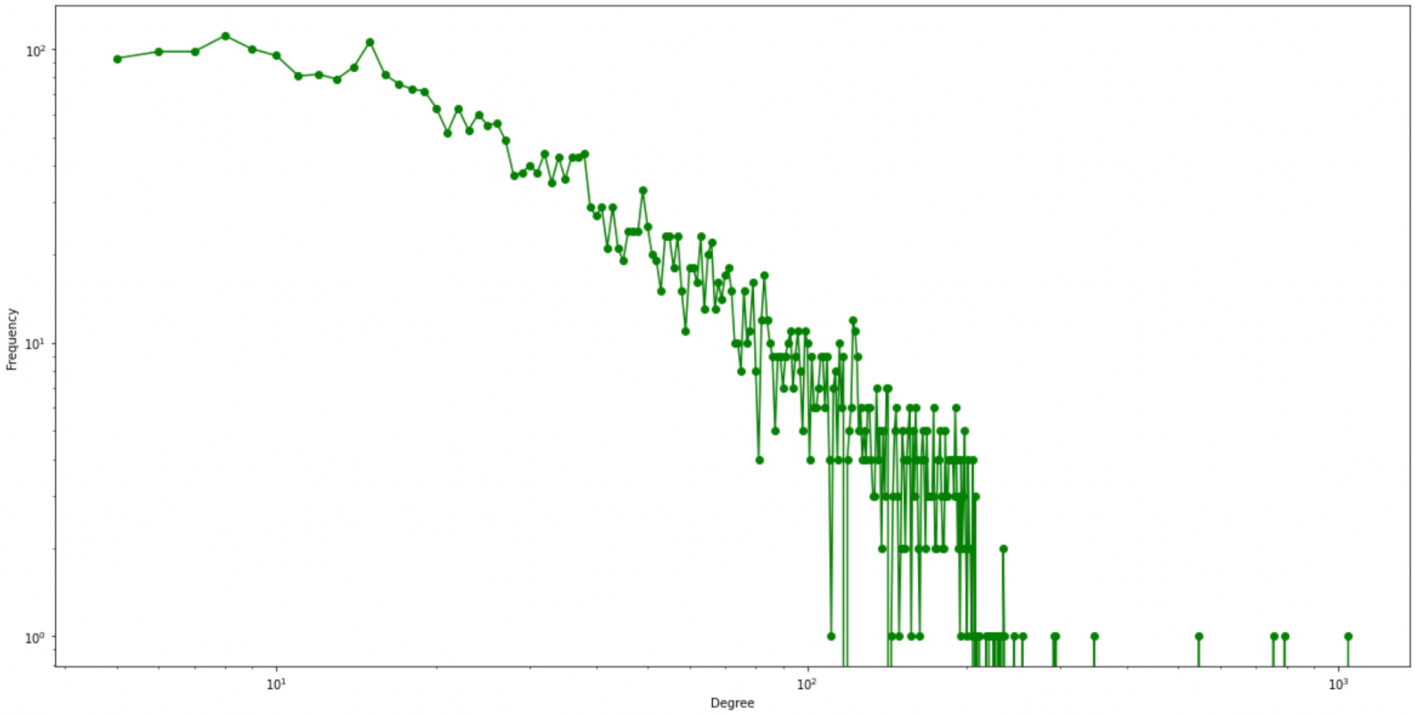
The facebook Graph has 4039 nodes and 88234 edges whereas Google Graphs has more than 800000 nodes and more than 5000000 edges.

Plotting the Degree Distribution of the Network:-

For Facebook:-

The Facebook dataset is an undirected graph therefore it is fairly straightforward to plot the degree distribution of the graph. For plotting the degree distribution we have used the nx.degree_histogram method.

```
In [14]: m=5

degree_freq = nx.degree_histogram(graph_facebook)
degrees = range(len(degree_freq))
plt.figure(figsize=(20, 10))
plt.loglog(degrees[m:], degree_freq[m:],'go-')
plt.xlabel('Degree')
plt.ylabel('Frequency')
```



Maximum Minimum and Standard Deviation of the Degrees for the Facebook Network:-

In [15]:
```
degrees = [val for (node, val) in graph_facebook.degree()]
```

In [16]:
```
max_degree = max(degrees)
print("Maximum Degree:-",max_degree)
```

```
Maximum Degree:- 1045
```

In [17]:
```
min_degree = min(degrees)
print("Minimum Degree:-",min_degree)
```

```
Minimum Degree:- 1
```

In [18]:
```
average_degree = sum(degrees)/len(degrees)
print("Average Degree:-",average_degree)
```

```
Average Degree:- 43.69101262688784
```

In [19]:
```
mean = sum(degrees) / len(degrees)
variance = sum([((x - mean) ** 2) for x in degrees]) / len(degrees)
std = variance ** 0.5

print("Standard deviation:- " + str(std))
```

```
Standard deviation:- 52.41411556737502
```

For Google:-

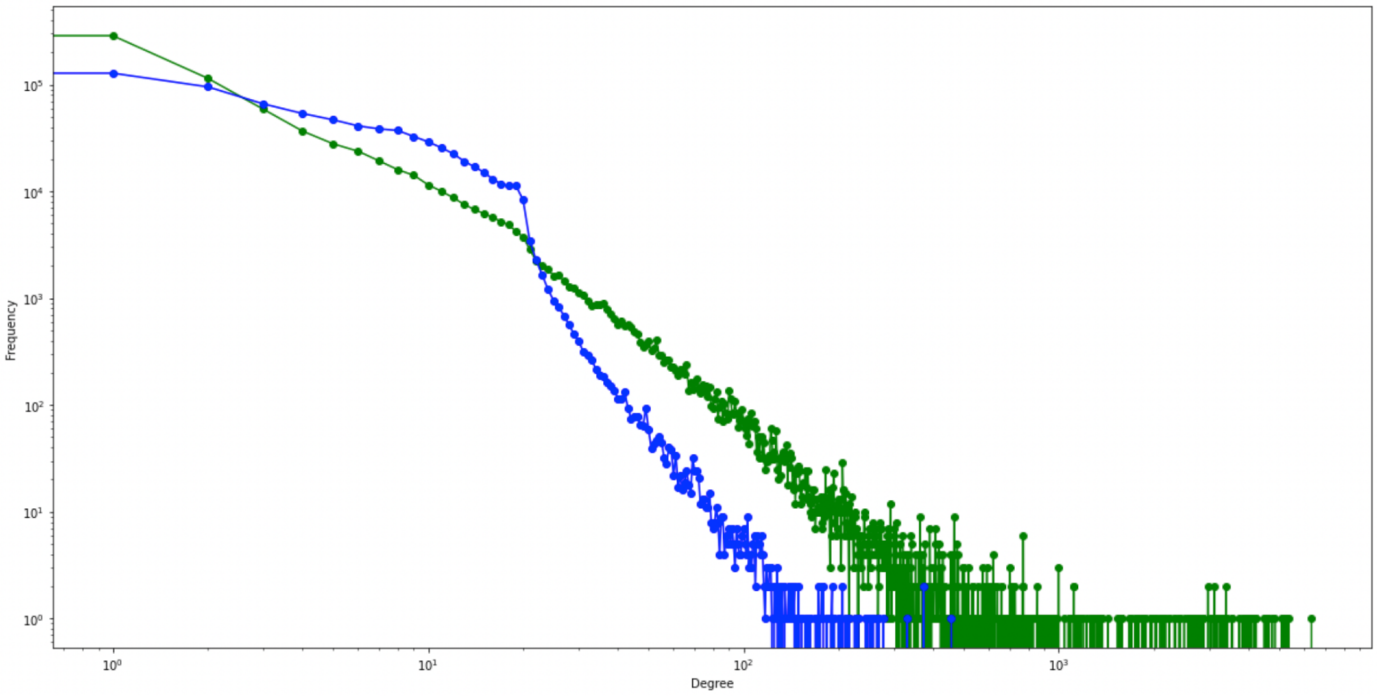
In the case of directed graphs(Google) we need to slightly modify the function to account for In degree and out degree.
First we will calculate the indegree and the out degree and the calculate the frequency distribution.

Modified Function:-

In [20]:
```
def degree_histogram_directed(G, in_degree=False, out_degree=False):
    nodes = G.nodes()
    if in_degree: #for finding the indegree
        in_degree = dict(G.in_degree())
        degseq=[in_degree.get(k,0) for k in nodes]
    elif out_degree: #for finding the outdegree
        out_degree = dict(G.out_degree())
        degseq=[out_degree.get(k,0) for k in nodes]
    else:
        degseq=[v for k, v in G.degree()]
    dmax=max(degseq)+1
    freq= [ 0 for d in range(dmax) ]
    for d in degseq:
        freq[d] += 1
    return freq
```

In [21]:
```
in_degree_freq = degree_histogram_directed(graph_google, in_degree=True)
out_degree_freq = degree_histogram_directed(graph_google, out_degree=True)
degrees = range(len(in_degree_freq))
plt.figure(figsize=(20, 10))
plt.loglog(range(len(in_degree_freq)), in_degree_freq, 'go-', label='in-degree')
plt.loglog(range(len(out_degree_freq)), out_degree_freq, 'bo-', label='out-degree')
plt.xlabel('Degree')
plt.ylabel('Frequency')
```

Plots for Indegree and Out degree:-



Maximum Minimum and Standard Deviation of the Degrees for the Google Network:-

In Degree:-

In [22]:
```
# in_degree = dict(graph_google.in_degree())
in_degree = [val for (node, val) in graph_google.in_degree()]
```

In [23]:
```
max_in_degree = max(in_degree)
print("Maximum In Degree:-",max_in_degree)
```

```
Maximum In Degree:- 6326
```

In [24]:
```
min_in_degree = min(in_degree)
print("Minimum In Degree:-",min_in_degree)
```

```
Minimum In Degree:- 0
```

In [25]:
```
average_in_degree = sum(in_degree)/len(in_degree)
print("Average In Degree:-",average_in_degree)
```

```
Average In Degree:- 5.829580010802626
```

In [26]:
```
mean = sum(in_degree) / len(in_degree)
variance = sum([((x - mean) ** 2) for x in in_degree]) / len(in_degree)
std = variance ** 0.5

print("Standard deviation:- " + str(std))
```

```
Standard deviation:- 39.22897027717595
```

Out Degree:-

In [27]:
```
out_degree = [val for (node, val) in graph_google.out_degree()]
```

In [28]:
```
max_out_degree = max(out_degree)
print("Maximum Out Degree:-",max_out_degree)
```

```
Maximum Out Degree:- 456
```

In [29]:
```
min_out_degree = min(out_degree)
print("Minimum Out Degree:-",min_out_degree)
```

```
Minimum Out Degree:- 0
```

In [30]:
```
average_out_degree = sum(out_degree)/len(out_degree)
print("Average Out Degree:-",average_out_degree)
```

```
Average Out Degree:- 5.829580010802626
```

In [31]:
```
mean = sum(out_degree) / len(out_degree)
variance = sum([((x - mean) ** 2) for x in out_degree]) / len(out_degree)
std = variance ** 0.5

print("Standard deviation:- " + str(std))
```

```
Standard deviation:- 6.593697311106739
```

Calculating Transitivity and Reciprocity of both Graphs:-

Transivity

In [20]:
```
print('Transitivity of our Facebook Network is : ', transitivity(graph_facebook))
```

```
Transitivity of our Facebook Network is :  0.5191742775433075
```

In [21]:
```
print('Transitivity of our Google Network is : ', transitivity(graph_google))
```

```
Transitivity of our Google Network is :  0.449540234004255
```

Reciprocity

In [22]:
```
print('Reciprocity of our Facebook Network is : ', overall_reciprocity(graph_facebook))
```

```
Reciprocity of our Facebook Network is :  0.0
```

In [23]:
```
print('Reciprocity of our Google Network is : ', overall_reciprocity(graph_google))
```

```
Reciprocity of our Google Network is :  0.3067509755065582
```

In [24]:
```
clustering_coeff_dict = clustering(graph_facebook)

facebook_clustering = ({'Node' : [node for node in graph_facebook.nodes()],
                  'Clustering_Coeff' : [clustering_coeff_dict[node] for node in graph_facebook.nodes()]
})
```

In [25]:
```
clustering_coeff_dict = clustering(graph_google)

google_clustering = ({'Node' : [node for node in graph_google.nodes()],
                  'Clustering_Coeff' : [clustering_coeff_dict[node] for node in graph_google.nodes()]
})
```

Calculating Local Clustering and Global Clustering of both Graphs:-

In [68]:
```
clustering_coeff_dict = clustering(graph_facebook)

facebook_clustering = ({'Node' : [node for node in graph_facebook.nodes()],
                  'Clustering_Coeff' : [clustering_coeff_dict[node] for node in graph_facebook.nodes()]
})
```

In [69]:
```
clustering_coeff_dict = clustering(graph_google)

google_clustering = ({'Node' : [node for node in graph_google.nodes()],
                  'Clustering_Coeff' : [clustering_coeff_dict[node] for node in graph_google.nodes()]
})
```

In [70]:
```
facebook_clustering = pd.DataFrame.from_dict(facebook_clustering)
```

In [71]:
```
google_clustering = pd.DataFrame.from_dict(google_clustering)
```

In [68]:
```
clustering_coeff_dict = clustering(graph_facebook)

facebook_clustering = ({'Node' : [node for node in graph_facebook.nodes()],
                  'Clustering_Coeff' : [clustering_coeff_dict[node] for node in graph_facebook.nodes()]
})
```

In [69]:
```
clustering_coeff_dict = clustering(graph_google)

google_clustering = ({'Node' : [node for node in graph_google.nodes()],
                  'Clustering_Coeff' : [clustering_coeff_dict[node] for node in graph_google.nodes()]
})
```

In [70]:
```
facebook_clustering = pd.DataFrame.from_dict(facebook_clustering)
```

In [71]:
```
google_clustering = pd.DataFrame.from_dict(google_clustering)
```

The global clustering is defined as 3*(Total number of closed triangles)/Total number of triplets which turns out to be equal to the average local clustering of the graph.

Centrality Measure:-

For Facebook:-

Facebook networks have considerably lower number of nodes and edges hence we can find the centrality measure of the whole graph.

In [21]:
```
from networkx.algorithms.centrality import degree_centrality, eigenvector_centrality_numpy\
                                           ,katz_centrality_numpy, closeness_centrality\
                                                                  ,betweenness_centrality
```

In [22]:
```
degree_centrality_dict = degree_centrality(graph_facebook)
eigenvector_centrality_dict = eigenvector_centrality_numpy(graph_facebook)
katz_centrality_dict = katz_centrality_numpy(graph_facebook)
pagerank_centraliy_dict = nx.pagerank_numpy(graph_facebook)
closeness_centrality_dict = closeness_centrality(graph_facebook)
betweenness_centrality_dict = betweenness_centrality(graph_facebook)

facebook_centrality = ({'Node' : [node for node in graph_facebook.nodes()],
                  'Degree_Centrality' : [degree_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Closeness_Centrality' : [closeness_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Betweenness_Centrality' : [betweenness_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Eigenvector_Centrality' : [eigenvector_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Katz_Centrality' : [katz_centrality_dict[node] for node in graph_facebook.nodes()],
                  'PageRank_Centrality' : [pagerank_centraliy_dict[node] for node in graph_facebook.nodes()]
})
```

```
/var/folders/gj/35ct11l52fq913rx4n_ts6cr0000gn/T/ipykernel_1197/3988097465.py:4: DeprecationWarning: networkx.pageran
k_numpy is deprecated and will be removed in NetworkX 3.0, use networkx.pagerank instead.
  pagerank_centraliy_dict = nx.pagerank_numpy(graph_facebook)
```

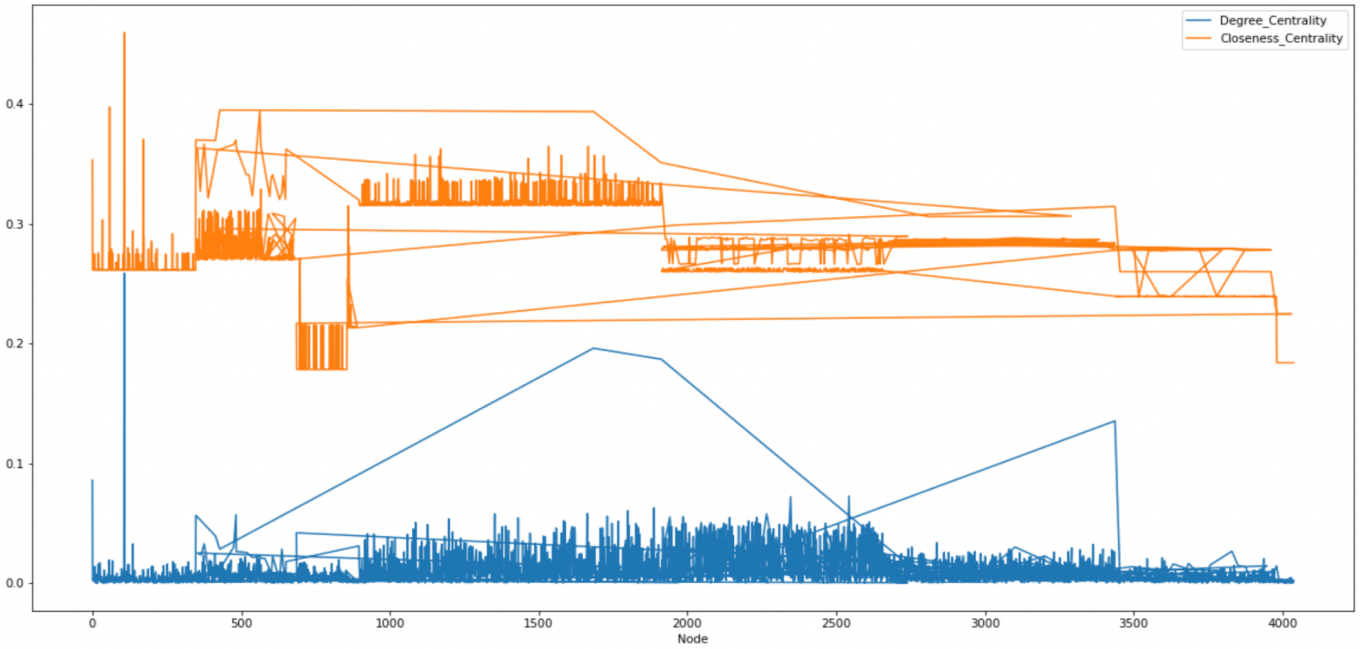
In [21]:
```
from networkx.algorithms.centrality import degree_centrality, eigenvector_centrality_numpy\
                                           ,katz_centrality_numpy, closeness_centrality\
                                                                  ,betweenness_centrality
```

In [22]:
```
degree_centrality_dict = degree_centrality(graph_facebook)
eigenvector_centrality_dict = eigenvector_centrality_numpy(graph_facebook)
katz_centrality_dict = katz_centrality_numpy(graph_facebook)
pagerank_centraliy_dict = nx.pagerank_numpy(graph_facebook)
closeness_centrality_dict = closeness_centrality(graph_facebook)
betweenness_centrality_dict = betweenness_centrality(graph_facebook)

facebook_centrality = ({'Node' : [node for node in graph_facebook.nodes()],
                  'Degree_Centrality' : [degree_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Closeness_Centrality' : [closeness_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Betweenness_Centrality' : [betweenness_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Eigenvector_Centrality' : [eigenvector_centrality_dict[node] for node in graph_facebook.nodes()],
                  'Katz_Centrality' : [katz_centrality_dict[node] for node in graph_facebook.nodes()],
                  'PageRank_Centrality' : [pagerank_centraliy_dict[node] for node in graph_facebook.nodes()]
})
```

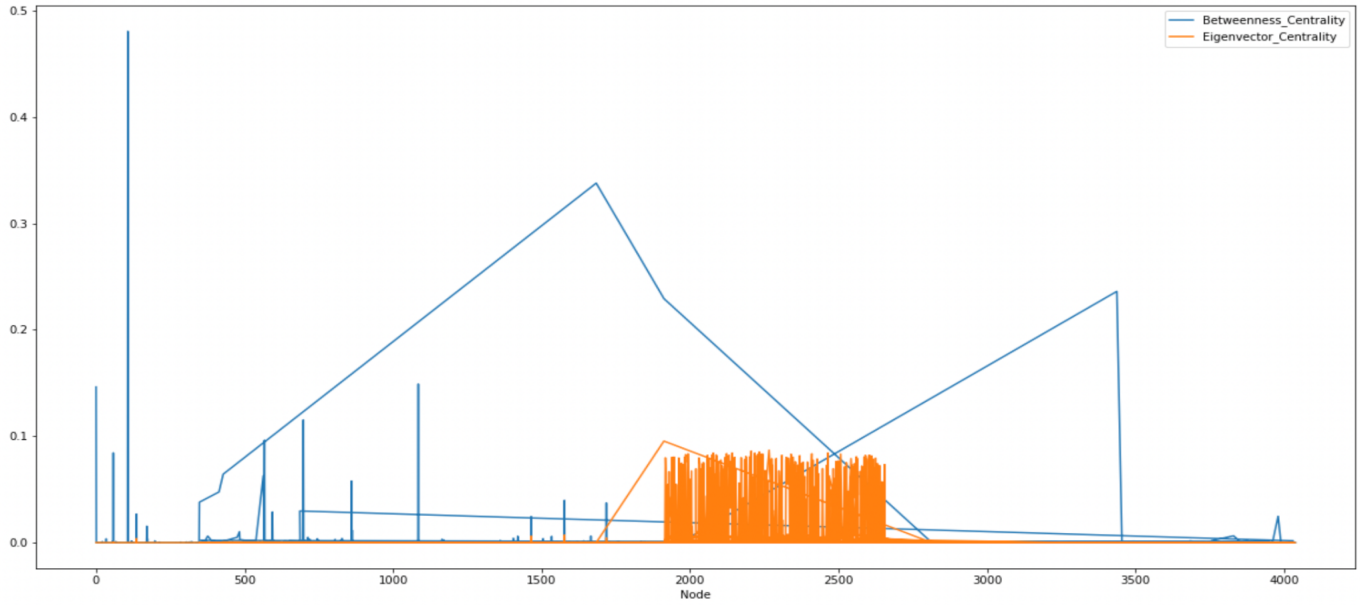
```
/var/folders/gj/35ct11l52fq913rx4n_ts6cr0000gn/T/ipykernel_1197/3988097465.py:4: DeprecationWarning: networkx.pageran
k_numpy is deprecated and will be removed in NetworkX 3.0, use networkx.pagerank instead.
  pagerank_centraliy_dict = nx.pagerank_numpy(graph_facebook)
```

Now we will group the centrality measure according to their value and plot them on the graph for better understanding.

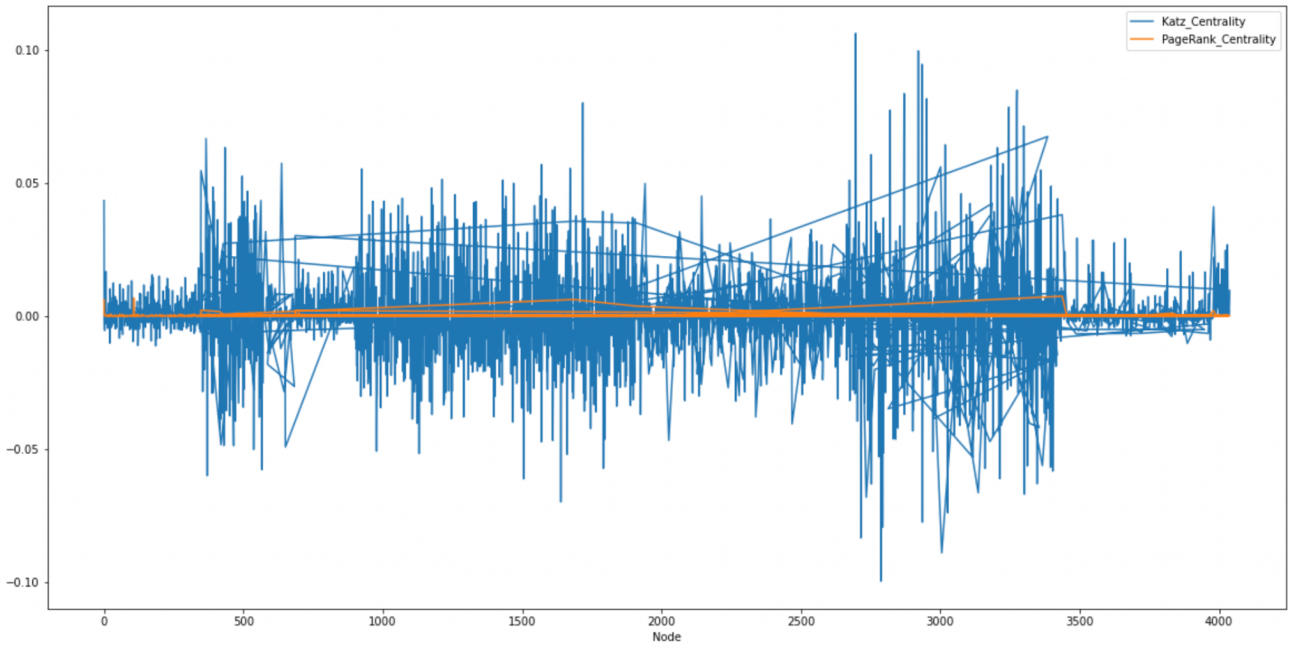
Degree_Centrality and Closeness_Centrality:-



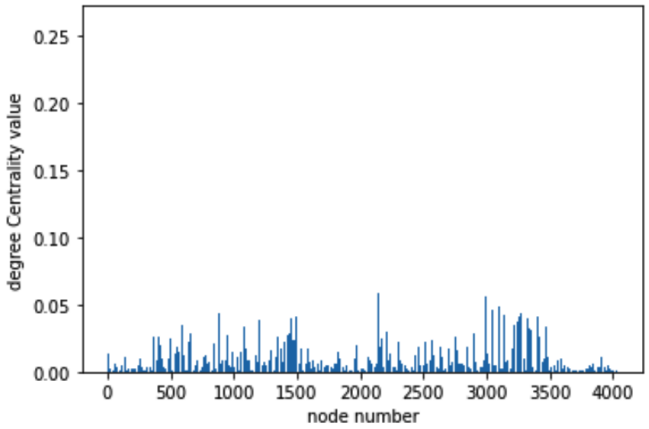
Betweenness_Centrality and Eigenvector_Centrality:-

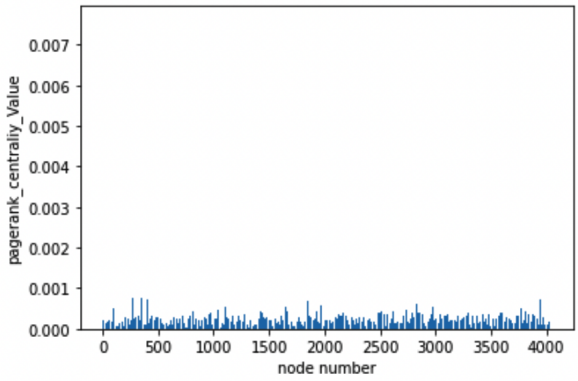


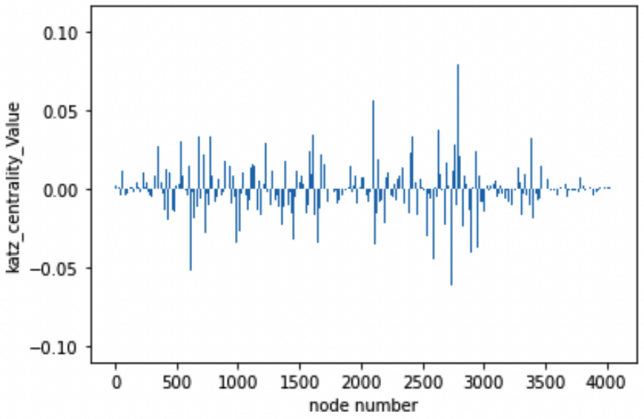
Katz Centrality and PageRank Centrality:-

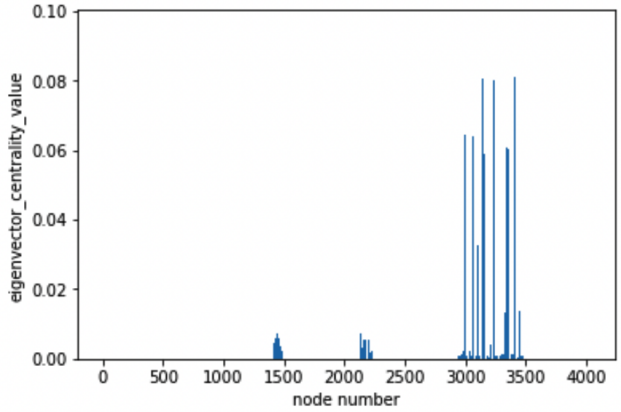


Since it's unclear from the above line graph, let’s plot the individual centrality measure on the bar graph.









For Google:-

For Google Graph we need to create a subgraph of the original dataset. We do this because the original graph contains too many nodes to process at once which was causing the kernel to die multiple times and had to restart it.

For creating subgraphs, we randomly take 5000 nodes and create a graph from it and then calculate the centrality measure on it.

```
res=[]
with open("web-Google.txt") as f:
  data = f.readlines()
for l in data:
  if l:
    words = l.split()
    res.append(words)
```

```
sample = random.sample(res, 5000)
filename=open('data.txt','a')
for item in sample:
  filename.write(item[0] + " " + item[1] + "\n")
```

```
graph_google_sample = nx.read_edgelist('data.txt',nodetype=int,create_using=nx.DiGraph())
```

```
gree_centrality_dict = degree_centrality(graph_google_sample)
genvector_centrality_dict = eigenvector_centrality_numpy(graph_google_sample)
tz_centrality_dict = katz_centrality_numpy(graph_google_sample)
oseness_centrality_dict = closeness_centrality(graph_google_sample)
tweenness_centrality_dict = betweenness_centrality(graph_google_sample)
gerank_centraliy_dict = nx.pagerank_numpy(graph_google_sample)

ogle_centrality = ({'Node' : [node for node in graph_google_sample.nodes()],
                'Degree_Centrality' : [degree_centrality_dict[node] for node in graph_google_sample.nodes()],
                'Closeness_Centrality' : [closeness_centrality_dict[node] for node in graph_google_sample.nodes()],
                'Betweenness_Centrality' : [betweenness_centrality_dict[node] for node in graph_google_sample.nodes()],
                'Eigenvector_Centrality' : [eigenvector_centrality_dict[node] for node in graph_google_sample.nodes()],
                'Katz_Centrality' : [katz_centrality_dict[node] for node in graph_google_sample.nodes()],
                'PageRank_Centrality' : [pagerank_centraliy_dict[node] for node in graph_google_sample.nodes()]
```

```
/var/folders/gj/35ct11l52fq913rx4n_ts6cr0000gn/T/ipykernel_1085/644764353.py:6: DeprecationWarning: networkx.pagerank
_numpy is deprecated and will be removed in NetworkX 3.0, use networkx.pagerank instead.
  pagerank_centraliy_dict = nx.pagerank_numpy(graph_google_sample)
```

SNA Project Round - 2

Finding the Giant Components:

For finding the Giant Components of the graph we will find all the connected components(in case of undirected graph) or Strongly connected components(in case of directed graph) and then sort them in reverse order to find the largest component(Giant Component) at the 0th index.

For this purpose we use the inbuilt functions of nx.connected_component and nx.strongly_connected_component in the networkx library.

Facebook:-

```
[75] facebook_c = sorted(nx.connected_components(graph_facebook), key=len, reverse=True)
     facebook_giant = graph_facebook.subgraph(facebook_c[0])

[80] print(facebook_giant.number_of_nodes()/facebook_nodes)

     1.0
```

Here, facebook_nodes = Number of nodes in the graph of Facebook network.

Google:-

```
[76] largest = max(nx.strongly_connected_components(graph_google), key=len)

[77] print(len(largest)/google_nodes)

     0.4965291219175188
```

Here, google_nodes = Number of nodes in the graph of Google network.

Creating Scale Free Network using Barabasi-Albert Algorithm and Applying ICM to find the number of steps required to get the maximum number of nodes:-

For creating Scale Free Network, we can use the inbuilt function in the networkx library barabasi_albert_graph. This function requires 3 inputs:- n(number of nodes), m(Number of edges to attach from a new node to existing nodes) and seed. Seed is used for random number generation which will be taken as default if not provided.

(creating Scale Free Network)

```
graph = barabasi_albert_graph(10000, 250)
print('Barabasi Alber Model has been created for 10000 nodes and 250 edges\n')


print('\n')
```

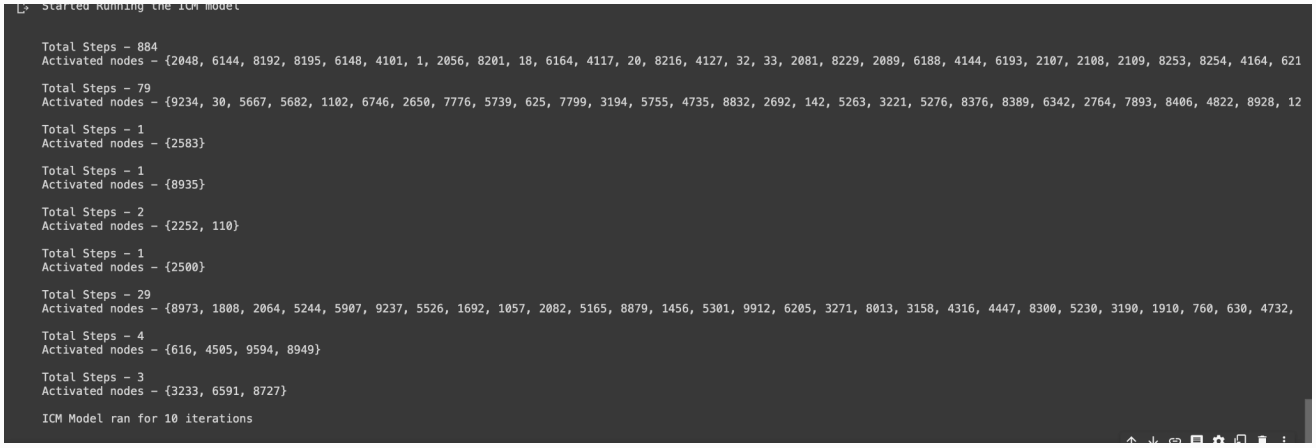
In order to apply the Independent Cascade Model to find the steps required to get the maximum number of nodes we first need to assign Activation Probability of each node. For this purpose we will traverse the entire graph(scale free network) using the Breadth First Search(BFS) algorithm and set the activation probability. Entire Process has been explained in the code along with comments.

```
print('\n')


set_edge_attributes(graph, values=0, name='weight')
#Setting edge attributes to initialize the activation probability of each edege.

arr = [[False] * 10000 for i in range(10000)]
visited = set()
#We create a set visited to mark all the nodes that have been visited.
#If in our algorithm we visit these nodes again it will run the entire process again and just leave the node in its original state
q = Queue()
visited.add(0)
q.put(0)

#Now we will run BFS algorithm to Assign Activate Probability
while not q.empty():
    u = q.get()

    visited.add(u)
    x = 1
    f = 0

    for v in graph.neighbors(u):
        #Here we consider all the nodes that are neighbours of the current node recieved from the front of the queue.

        if v not in visited:
            visited.add(v)
            q.put(v)

        if arr[u][v] is True or arr[v][u] is True:
            x -= graph[u][v]['weight']
        else:
            f += 1

    k = 0
    lst = np.random.dirichlet(np.ones(f), size=1).tolist()[0]

    for v in graph.neighbors(u):
        if arr[u][v] is False or arr[v][u] is False:
            arr[u][v] = arr[v][u] = True
            graph[u][v]['weight'] = lst[k] * max(x, 0)
            k += 1

print('Activation Probability has been assigned for every node\n')
```

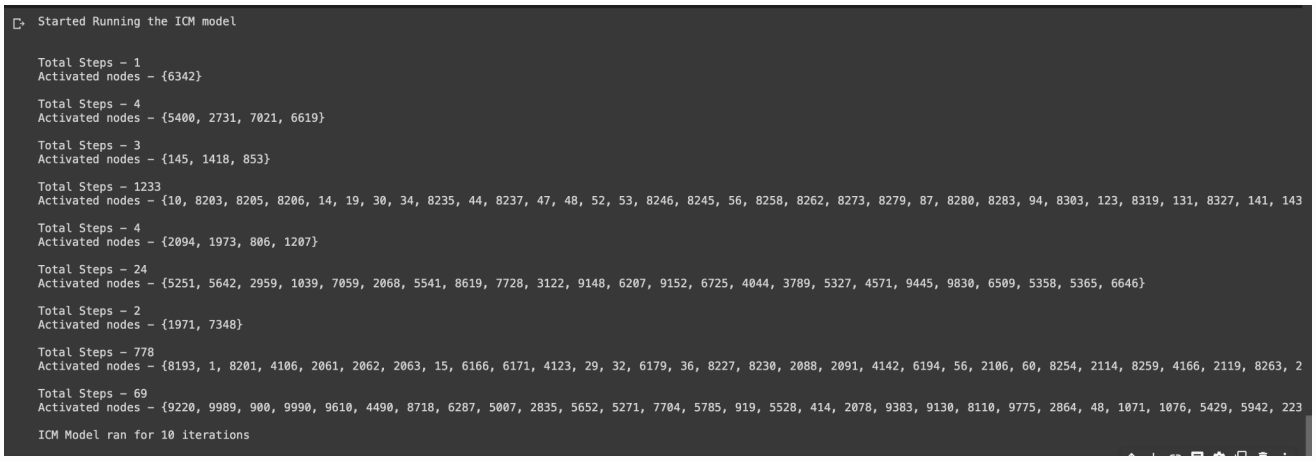
The next step is to find the number of steps, for this a have written a function icm that takes two inputs:- Initial Activation Probability and a list to store the steps after each iteration.
In each iteration a random node will be selected and then we will apply the BFS algorithm to find the steps required to get maximum number of nodes.

```
def icm(graph,ls):
    print("Started Running the ICM model")
    print('\n')
    i = 9;
    #Here we run BFS for 10 iteration.
    #Then we will find the average and median of all the steps in these 10 iterations
    for itr in range(0, i):
        r = random.randint(0, 9999)
        vis = set()
        q = Queue()
        q.put(r)
        activated = set()
        activated.add(r)
        steps = 0
        # print("Initial set - " + str(activated))

        while not q.empty():
            steps += 1
            u = q.get()

            for v in graph.neighbors(u):
                if v not in activated:
                    rand = random.uniform(0, 1)
                    #The node will only be activated if the random number that is generated is less than the activate probability that is stored in the graph variable.
                    if rand < graph[u][v]['weight']:
                        q.put(v)
                        activated.add(v)

        print("Total Steps - " + str(steps))
        print("Activated nodes - " + str(activated))
        print()
        ls.append(steps)

    print("ICM Model ran for %d iterations" % (i+1))
```

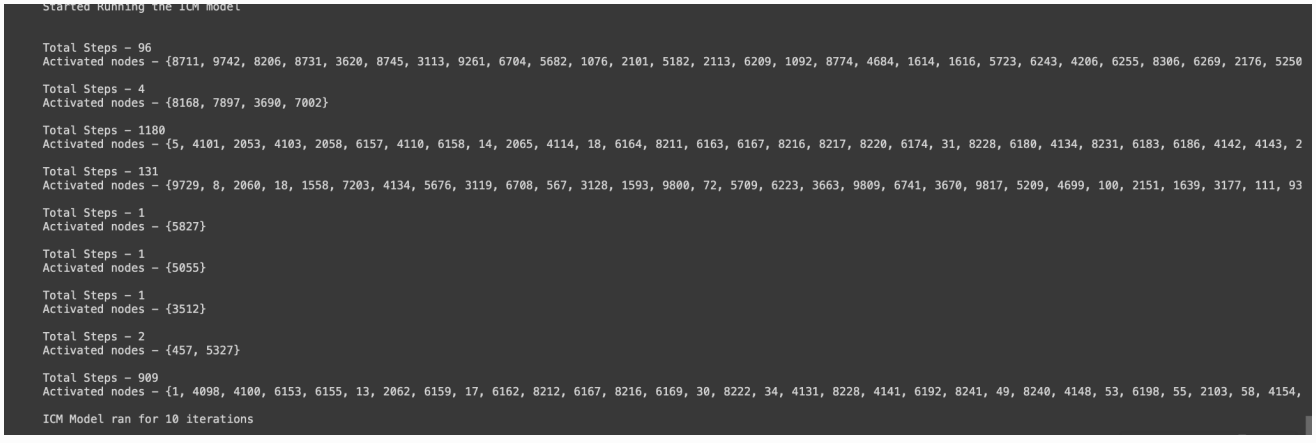
Here are some of the snippets from when this function was run for a few times. Each time it ran for 10 iterations.



```
Started Running the ICM model

Total Steps - 884
Activated nodes - {2048, 6144, 8192, 8195, 6148, 4101, 1, 2056, 8201, 18, 6164, 4117, 20, 8216, 4127, 32, 33, 2081, 8229, 2089, 6188, 4144, 6193, 2107, 2108, 2109, 8253, 8254, 4164, 621

Total Steps - 79
Activated nodes - {9234, 30, 5667, 5682, 1102, 6746, 2650, 7776, 5739, 625, 7799, 3194, 5755, 4735, 8832, 2692, 142, 5263, 3221, 5276, 8376, 8389, 6342, 2764, 7893, 8406, 4822, 8928, 12

Total Steps - 1
Activated nodes - {2583}

Total Steps - 1
Activated nodes - {8935}

Total Steps - 2
Activated nodes - {2252, 110}

Total Steps - 1
Activated nodes - {2500}

Total Steps - 29
Activated nodes - {8973, 1808, 2064, 5244, 5907, 9237, 5526, 1692, 1057, 2082, 5165, 8879, 1456, 5301, 9912, 6205, 3271, 8013, 3158, 4316, 4447, 8300, 5230, 3190, 1910, 760, 630, 4732,

Total Steps - 4
Activated nodes - {616, 4505, 9594, 8949}

Total Steps - 3
Activated nodes - {3233, 6591, 8727}

ICM Model ran for 10 iterations
```

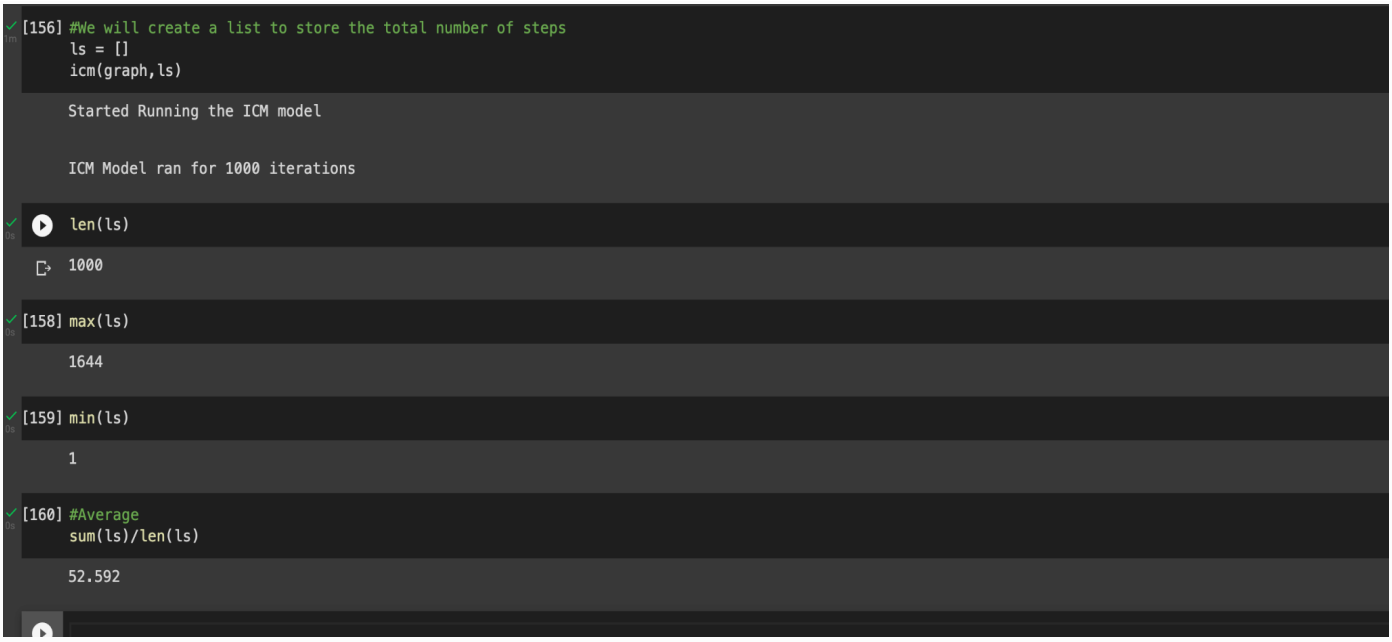


```
Started Running the ICM model

Total Steps - 1
Activated nodes - {6342}

Total Steps - 4
Activated nodes - {5400, 2731, 7021, 6619}

Total Steps - 3
Activated nodes - {145, 1418, 853}

Total Steps - 1233
Activated nodes - {10, 8203, 8205, 8206, 14, 19, 30, 34, 8235, 44, 8237, 47, 48, 52, 53, 8246, 8245, 56, 8258, 8262, 8273, 8279, 87, 8280, 8283, 94, 8303, 123, 8319, 131, 8327, 141, 143

Total Steps - 4
Activated nodes - {2094, 1973, 806, 1207}

Total Steps - 24
Activated nodes - {5251, 5642, 2959, 1039, 7059, 2068, 5541, 8619, 7728, 3122, 9148, 6207, 9152, 6725, 4044, 3789, 5327, 4571, 9445, 9830, 6509, 5358, 5365, 6646}

Total Steps - 2
Activated nodes - {1971, 7348}

Total Steps - 778
Activated nodes - {8193, 1, 8201, 4106, 2061, 2062, 2063, 15, 6166, 6171, 4123, 29, 32, 6179, 36, 8227, 8230, 2088, 2091, 4142, 6194, 56, 2106, 60, 8254, 2114, 8259, 4166, 2119, 8263, 2

Total Steps - 69
Activated nodes - {9220, 9989, 900, 9990, 9610, 4490, 8718, 6287, 5007, 2835, 5652, 5271, 7704, 5785, 919, 5528, 414, 2078, 9383, 9130, 8110, 9775, 2864, 48, 1071, 1076, 5429, 5942, 223

ICM Model ran for 10 iterations
```



```
Started Running the ICM model

Total Steps - 96
Activated nodes - {8711, 9742, 8206, 8731, 3620, 8745, 3113, 9261, 6704, 5682, 1076, 2101, 5182, 2113, 6209, 1092, 8774, 4684, 1614, 1616, 5723, 6243, 4206, 6255, 8306, 6269, 2176, 5250

Total Steps - 4
Activated nodes - {8168, 7897, 3690, 7002}

Total Steps - 1180
Activated nodes - {5, 4101, 2053, 4103, 2058, 6157, 4110, 6158, 14, 2065, 4114, 18, 6164, 8211, 6163, 6167, 8216, 8217, 8220, 6174, 31, 8228, 6180, 4134, 8231, 6183, 6186, 4142, 4143, 2

Total Steps - 131
Activated nodes - {9729, 8, 2060, 18, 1558, 7203, 4134, 5676, 3119, 6708, 567, 3128, 1593, 9800, 72, 5709, 6223, 3663, 9809, 6741, 3670, 9817, 5209, 4699, 100, 2151, 1639, 3177, 111, 93

Total Steps - 1
Activated nodes - {5827}

Total Steps - 1
Activated nodes - {5055}

Total Steps - 1
Activated nodes - {3512}

Total Steps - 2
Activated nodes - {457, 5327}

Total Steps - 909
Activated nodes - {1, 4098, 4100, 6153, 6155, 13, 2062, 6159, 17, 6162, 8212, 6167, 8216, 6169, 30, 8222, 34, 4131, 8228, 4141, 6192, 8241, 49, 8240, 4148, 53, 6198, 55, 2103, 58, 4154,

ICM Model ran for 10 iterations
```

As we can see it is very difficult to summarize the data that was gathered with mere 10 iterations. Therefore, we ran the function for 1000 iterations and stored all the steps after each iteration. Then we will check the average, minimum, maximum and various other parameters more properly.



References:-

https://networkx.org/

https://hamketab.ir/wp-content/uploads/edd/2020/03/Social-Media-Mining-book.pdf

http://home.iitj.ac.in/~suman/articles/detail/how-to-code-independent-cascade-model-of-information-diffusion/#:~:text=Independent%20Cascade%20Model%20(ICM)%20is,the%20information%20or%20not%20influenced.

https://cambridge-intelligence.com/keylines-faqs-social-network-analysis/#:~:text=Centrality%20measures%20are%20a%20vital,but%20they%20all%20work%20differently.

http://networksciencebook.com/