Q1)

1.Adjacency Matrix and Edge List

```
# adjacency_list
adjacency_list = defaultdict(list)
for i in range(len(source_list)):
    adjacency_list[source_list[i]].append(target_list[i])
```

```
#adjacency_matrix
for i in range(len(source_list)):
    row = new_node_list.index(source_list[i])
    col = new_node_list.index(target_list[i])
    adjacent_matrix[row][col] = 1
```

2.Number of nodes

```
source_list = df['FromNodeId'].tolist()
target_list = df['ToNodeId'].tolist()
total_list = source_list+target_list
```

```
new_node_list = []
for node in total_list:
    if node not in new_node_list:
        new_node_list.append(node)
```

```
l=len(new_node_list)
dim = (l , l)
adjacent_matrix = np.zeros(dim)
```

```
#number of nodes
print(l)
```

List of source and target nodes is created and from that list of unique nodes is created and length of that list is calculated to find number of nodes.

3.Number of Edges

```
edge_count = 0;
for i in range(l):
    for j in range(l):
        if (loaded_model[i][j]==1):
            edge_count = edge_count+1
```

```
#number of edges
edge_count
```

Counting number of 1's in adjacency matrix gives number of edges in a graph

4.Average In-Degree

```
#avg in degree
indeg = 0
for j in range(l):
    for i in range(l):
        if(loaded_model[i][j]==1):
            indeg = indeg+1

avg_in_degree = indeg/l
print(avg_in_degree)
```

```
14.573295853829936
```

Calculating indegree using adjacency matrix and then calculating average of all indegrees to find average in degree.

5.Average Out-Degree

```
#avg out degree
outdeg = 0
for i in range(l):
    for j in range(l):
        if(loaded_model[i][j]==1):
            outdeg = outdeg+1
avg_out_degree = outdeg/l
print(avg_out_degree)
```

```
14.573295853829936
```

Calculating outdegree using adjacency matrix and then calculating average of all outdegrees to find average out-degree.

6.Density of network

```
#network density without self loops in network graph
network_dense = edge_count/(l*(l-1))
print(network_dense)
```

Density of network is being calculated by taking ratio of total number of edges and number of edges possible with given number of nodes

7.Node with Max-Indegree

```
#node with max indegree
indegrees = []
max_indeg = 0
for j in range(l):
    indeg=0
    for i in range(l):
        if(loaded_model[i][j]==1):
            indeg = indeg+1
    indegrees.append(indeg)
    if(indeg>=max_indeg):
        max_indeg=indeg
        index_max = j
print(new_node_list[index_max])
```

From the list of all indegrees first we are finding maximum indegree and using its index we are finding node with corresponding indegree

8. Node with Max-Outdegree

```
#node with max outdegree
max_outdeg = 0
outdegrees = []
for i in range(l):
    outdeg=0
    for j in range(l):
        if(loaded_model[i][j]==1):
            outdeg = outdeg+1
    outdegrees.append(outdeg)
    if(outdeg>=max_outdeg):
        max_outdeg=outdeg
        index_max = i
print(new_node_list[index_max])
```

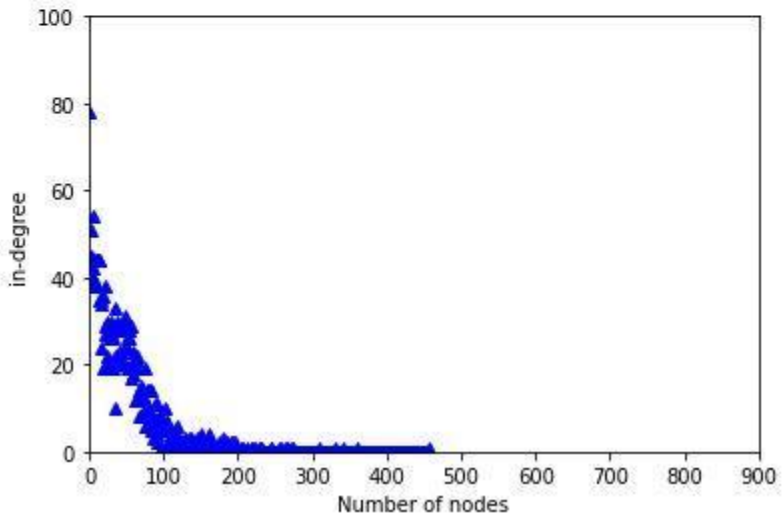
From the list of all outdegrees first we are finding maximum outdegree and using its index we are finding node with corresponding outdegree.

9. Indegree Distribution

```
pr_ind_distri = []
indegree_list = []
for degree in range(1,max_indeg+1):
    indegree_list.append(degree)
    pr = indegrees.count(degree)
    pr_ind_distri.append(pr)
```

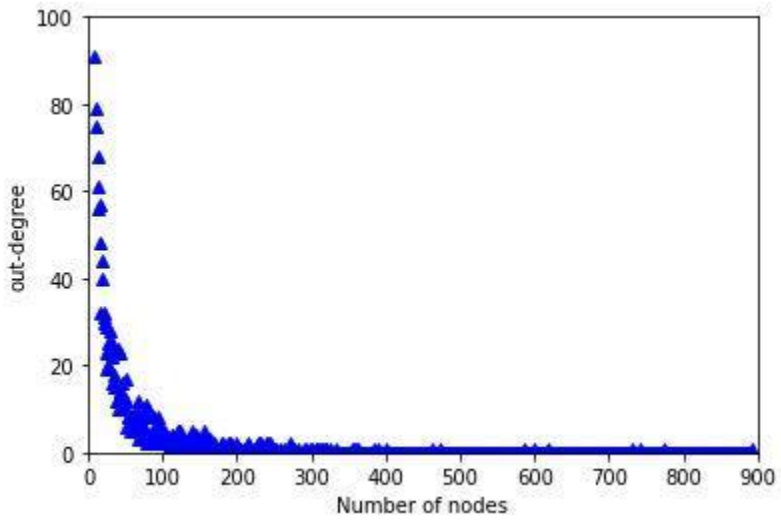
```
import matplotlib.pyplot as plt
plt.plot(indegree_list,pr_ind_distri,'b^')
plt.xlim(0, 900)
plt.ylim(0, 100)
plt.ylabel("in-degree")
plt.xlabel("Number of nodes")
```

```
Text(0.5, 0, 'Number of nodes')
```



Here we calculated number of nodes with corresponding degrees and stored that frequency in a list , now with this information we draw a distribution in form of graph.

10. Outdegree Distribution




```
pr_out_distri = []
outdegree_list = []
for degree in range(1,max_outdeg+1):
    outdegree_list.append(degree)
    pr = outdegrees.count(degree)
    pr_out_distri.append(pr)
```

```
plt.plot(outdegree_list,pr_out_distri,'b^')
plt.xlim(0, 900)
plt.ylim(0, 100)
plt.ylabel("out-degree")
plt.xlabel("Number of nodes")
```

Here we calculated number of nodes with corresponding degrees and stored that frequency in a list , now with this information we draw a distribution in form of graph.

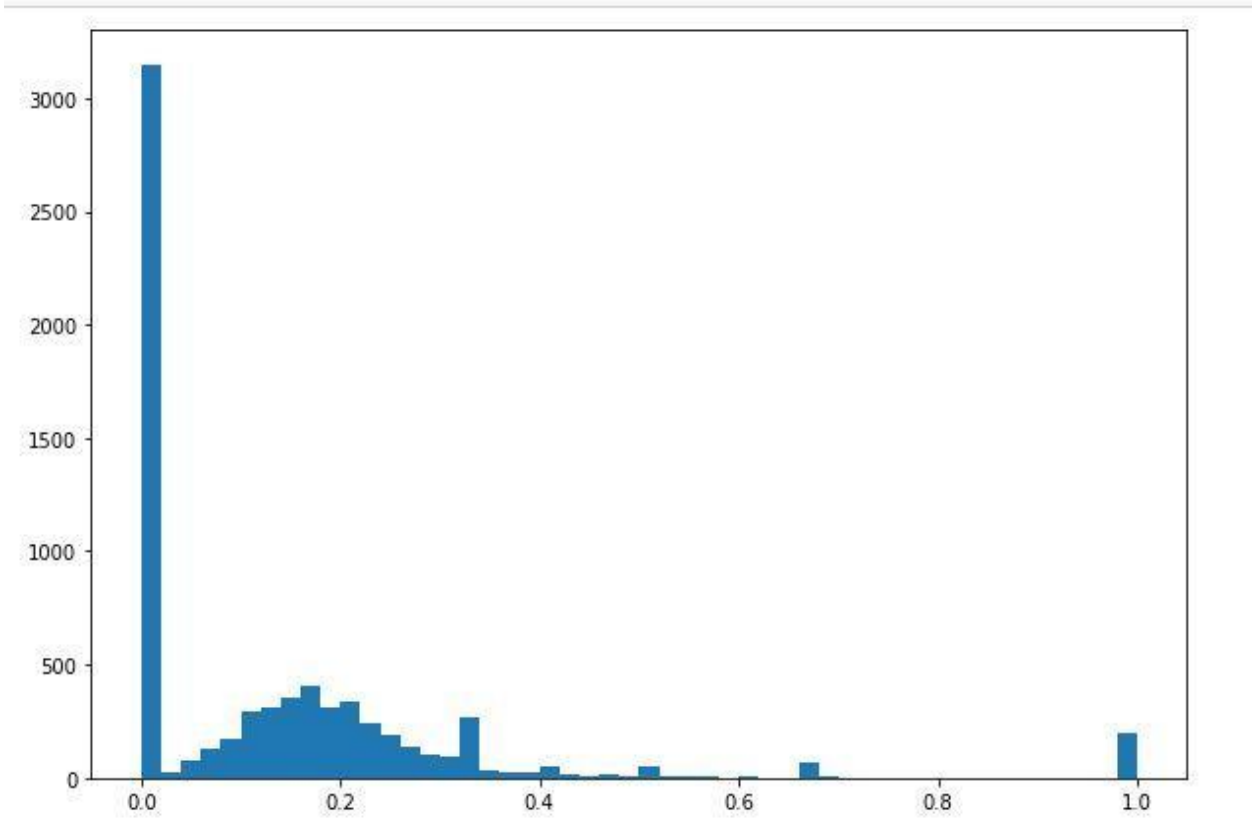
Local clustering coefficient:

```
lcc = []
for i in range(l):
    neighbor_list = []
    for j in range(l):
        if(new_adjacent_matrix[i][j]==1):
            neighbor_list.append(j)
    new_neighbor_list = neighbor_list

    nv=0
    for u in neighbor_list:
        for v in new_neighbor_list:
            if(new_adjacent_matrix[u][v]==1):
                nv=nv+1
    kv=degree_list[i]
    if(kv==1):
        lcc.append(0)
    else:
        kv1=kv*(kv-1)
        cc=nv/kv1
        lcc.append(cc)
```

First nv is calculated using adjacency matrix ie by calculating degree of that particular node and then kv is calculated as number of links between neighbors of that particular node and now by using formula for lcc ie nv/(kv*(kv-1)) we find lcc of that particular node.

Local clustering coefficient Distribution:



Using lcc which is calculated above and using node frequency lcc distribution is plotted as shown above using histogram.

Ques2

Creating the directed graph using networkx library and adjacent matrix.

```
G = nx.DiGraph()
```

```
for i in range(l):
    for j in range(l):
        if adjacent_matrix[i][j] == 1:
            G.add_edge(i,j)
```

1. **Page Rank score for each node by implementing by scratch**

```
def get_page_rank(itera , page_rank_list , G):
    d=0.85
    n = len(page_rank_list)
    for i in range(itera):
        for node in range(len(page_rank_list)):
            in_neighbors = G.predecessors(node)
            page_rank_sum=0
            for neighbor in in_neighbors:
                page_rank_sum = page_rank_sum + page_rank_list[neighbor]/len(list(G.successors(neighbor)))


            random_walk = d/n
            page_rank_list[node] = random_walk + (1-d) * page_rank_sum

    return page_rank_list
```

```
#without library
itera = 100
final_page_rank = get_page_rank(itera ,page_rank_list , G)
```

```
] #with library
  pr = nx.pagerank(G , max_iter=100)
```

Results Obtained

```
pr
```

```
{0: 0.00017349553934328338,
 2716: 0.0017851250122027217,
 4124: 0.0021500675059293226,
 4314: 0.0010508052619841283,
 6110: 0.0008141761230496596,
 6111: 0.000812430352613478,
 1: 0.00020539498232448021,
 24: 0.00034765464971898025,
 25: 0.0016986730322136935,
 32: 0.0003439790689580259,
 103: 0.0004398371153454517,
 137: 0.0005817197428805889,
 159: 0.00029758488331950184,
 163: 0.00016083873728146714,
 302: 0.00017393564565284635,
 305: 9.460415271381966e-05,
 322: 0.0002892903392357495,
 463: 0.00033152691295165293,
 479: 0.00022615441013923313,
 481: 0.00010518825019481072,
 484: 0.00019458075864204938,
 496: 0.00018151640169193398,
 6112: 0.0016599199669365457,
 6113: 0.0013349240914416604,
 6114: 0.0001736775777030509,
```

2. HUB and Authority

```
hub,authority = nx.hits(G)
```

```
my_hub_key = hub.keys()
my_auth_key = authority.keys()
my_pr_key = pr.keys()
```

```
my_hub_key = list(my_hub_key)
my_auth_key = list(my_auth_key)
my_pr_key = list(my_pr_key)
```

```
new_auth_values = list(authority.values())
new_hub_values = list(hub.values())
new_pr_values = list(pr.values())
```

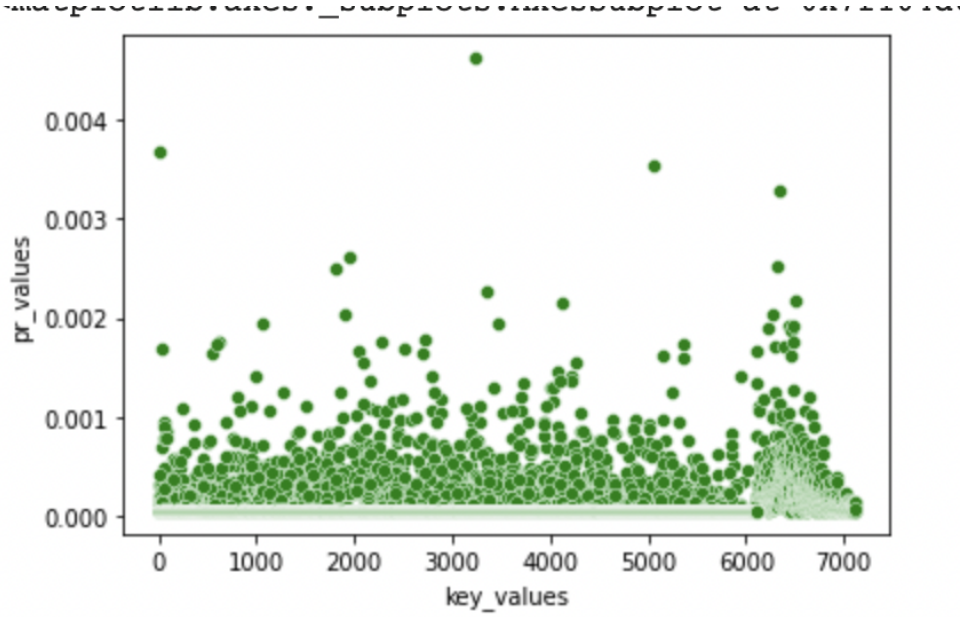
```
hub
```

```
{0: 7.925509536668041e-05,
 2716: 0.0033814231063449985,
 4124: 0.00037720963088326646,
 4314: 0.0013969722537516508,
 6110: 0.0,
 6111: 0.0,
 1: 4.021031639777635e-05,
 24: 2.526962059829082e-05,
 25: 0.0003569336095194368,
 32: 0.00010662223573949801,
 103: 2.588983177975365e-06,
 137: 5.991557851971008e-05,
 159: 0.0004364871929359797,
 163: 0.0006660507491502769,
 302: 9.313148244802727e-05,
 305: 1.0425226312650495e-06,
 322: 0.0009447618732908875,
 463: 1.690402245546196e-06,
 479: 8.003211011036128e-06,
 481: 6.262609343164861e-05,
 484: 3.97592563596769e-05,
 496: 2.3955216460508118e-05,
 6112: 0.0,
 6113: 0.0,
 6114: 0.0,
 6115: 0.0,
```

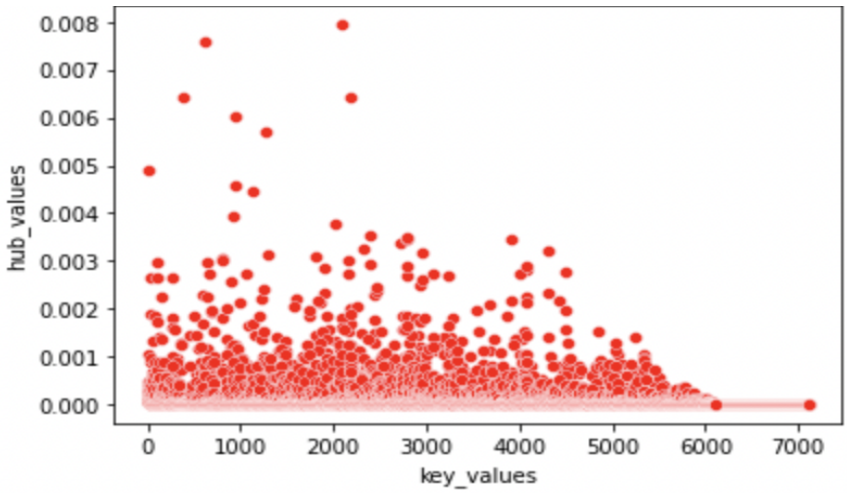
```
authority
```

```
{0: 9.508954835822218e-05,
 2716: 0.0023284150914976826,
 4124: 0.001824739664365004,
 4314: 0.0012852668947678197,
 6110: 0.00012172939864267837,
 6111: 0.0007626588344747699,
 1: 9.501171858460713e-05,
 24: 0.0001402071358643648,
 25: 0.00025475061390751937,
 32: 6.13295050360769e-05,
 103: 4.840902379896683e-06,
 137: 0.00012871399918704812,
 159: 0.00012808339450860097,
 163: 9.346016586177434e-05,
 302: 0.00011304120192581299,
 305: 2.9996853208330798e-05,
 322: 7.973278770685004e-05,
 463: 7.273924780779132e-05,
 479: 5.512405621214168e-05,
 481: 2.7879633455625446e-05,
 484: 8.786023670037264e-05,
 496: 2.7141024149939694e-05,
 6112: 0.0008251269691977225,
 6113: 0.0007442312255899237,
 6114: 8.219949441209396e-05,
 6115: 3.7566410592876524e-05,
 6116: 2.3189352656171763e-05,
```

Distribution of Page rank using Scatter Plot

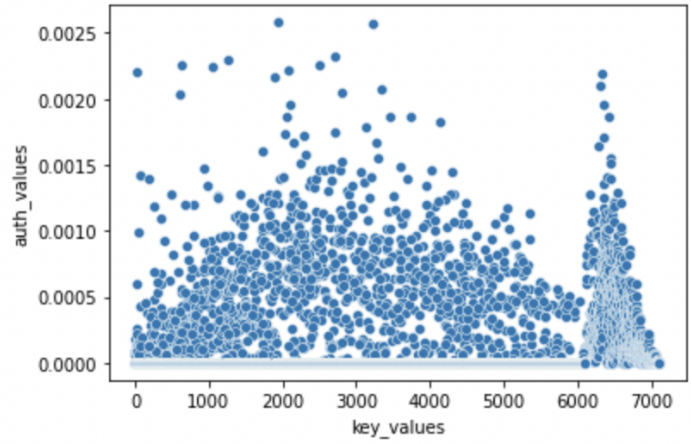


Distribution of HUB scores using Scatter Plot



Distribution of authority scores using Scatter Plot

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f104ac8b390>
```



Analysis

1. Most of the values are scattered between 0 and 0.001 in PageRank.
2. Most of the values are scattered between 0 and 0.010 in the authority and the score goes to 0.0025.

3. Most of the values are scattered between 0 and 0.003 in hits and the score goes to 0.008.
4. On fetching the hits score and page rank scores, we get different results since we know both runs on different techniques.

Comparison between the PageRank and HITS

Pagerank finds the importance of the webpage by counting the number of links it is connected to. It also takes the quality of the links into consideration. It is represented as a graph-like structure in which nodes represent the webpage and edges represent the links between the pages. Weights are then calculated and allotted to the web pages, high weight means it is a high-ranked page.

HITS also finds the importance of a webpage but it appoints 2 numbers - a hub score and an authority score.

A webpage is of high quality if its hub and authority score is high that is if it is linked to high-ranked pages.

Pagerank is query independent while HITS is query dependent.

HITS depends on both hub and authority scores and is more time-consuming than PageRank.