```
import tensorflow as tf
tf.test.gpu_device_name()
```

```
''
```

```
from google.colab import drive
drive.mount('/content/drive/')
```

```
Mounted at /content/drive/
```

```
import networkx as nx
G = nx.read_edgelist("/content/drive/MyDrive/Colab Notebooks/facebook_combined.txt", nodetype=int)
```

```
Complete network using networkx library. Following Parameters we have
of in all the Hops
1. degree of the node
2. sum of degrees of the neighbours (depending on the steps / hops)
3. Common connections in between neighbours
4. Average Degree Connectivity of the Node
5. Average Degree Connectivity of the Node's Network (hop step)
6. bridge count described as nodes which are end points which are not
with any other node. (hop step)
7. K core components of the Node
8. K core components of the Node's Network (hop step)
9. Onion Layers of the Node
10. Onion Layers of the Node's Network (hop step)
11. Boundary expansion of the Node netwrok (hop step)
12. eccentricity of a node v is the maximum distance from v to all oth
in G
13. Kemeny constant - measures the time needed for spreading across a
14. global efficiency  of the Node's Network (hop step)
15. max_weight_matching
16. random_probability -
17. Number of Triangles in the Network


Based on this data i have done 4 types of clustering


*   K Means
*   Heiarchial (Agglometric ) clustering


*   DBScan
*   Network X Clustering

For Score Calculation of metrics i have compared it to the NetworkX Cl

Finally i have tried community detection through various algorithms
```

Complete network using networkx library. Following Parameters we have take care of in all the Hops

1. degree of the node
2. sum of degrees of the neighbours (depending on the steps / hops)
3. Common connections in between neighbours
4. Average Degree Connectivity of the Node
5. Average Degree Connectivity of the Node's Network (hop step)
6. bridge count described as nodes which are end points which are not connected with any other node. (hop step)
7. K core components of the Node
8. K core components of the Node's Network (hop step)
9. Onion Layers of the Node
10. Onion Layers of the Node's Network (hop step)
11. Boundary expansion of the Node netwrok (hop step)
12. eccentricity of a node v is the maximum distance from v to all other nodes in G
13. Kemeny constant - measures the time needed for spreading across a graph
14. global efficiency of the Node's Network (hop step)
15. max_weight_matching
16. random_probability -
17. Number of Triangles in the Network

Based on this data i have done 4 types of clustering

- K Means
- Heiarchial (Agglometric) clustering
- DBScan
- Network X Clustering

For Score Calculation of metrics i have compared it to the NetworkX Clustering

Finally i have tried community detection through various algorithms

```
print(G.degree(1))
```

```
17
```

```
#all nodes and node degree in one frame for two hop
#node data
import pandas as pd
nodeData = []
for i in range(len(G)):
  ndData = []

  # Initialize deg outside the loop to avoid overwriting
  deg = G.degree[i]
  node_data = [i, deg]
  ndData.append(node_data)

  #extend it to the node data
  nodeData.extend(ndData)


# Define column names
columns = ['Node', 'NodeDegree']

# Create a DataFrame
nodedf = pd.DataFrame(nodeData, columns=columns)

# Display the DataFrame
print(nodedf)
```

```
      Node  NodeDegree
0        0         347
1        1          17
2        2          10
3        3          17
4        4          10
...    ...         ...
4034  4034           2
4035  4035           1
4036  4036           2
4037  4037           4
4038  4038           9

[4039 rows x 2 columns]
```

common connections between neighbours of a node

```
import pandas as pd
import networkx as nx
from itertools import combinations

onehopcommonsty_data = []

for node in range(len(G)):
    abData = []
    acData = sorted(nx.all_neighbors(G, node))

    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    onehopcommonsty_data.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
onehopcommondf = pd.DataFrame(onehopcommonsty_data, columns=["Common Connections"])

# Display the DataFrame
print(onehopcommondf)
```

```
      Common Connections
0                   2519
1                     57
2                     40
3                     86
```

```
4                    39
...                 ...
4034                  1
4035                  0
4036                  1
4037                  4
4038                 20

[4039 rows x 1 columns]
```

degree of the neighbours and eventually counting the number of suppose counting the number of 1 degree neighbours in the network

```
#near degree data
onedegree_data = []

for node in range(len(G)):
    abData = []
    acData = sorted(nx.all_neighbors(G, node))

    for j in acData:
        M =G.degree(j)
        abData.append(M)

    onedegree_data.append(abData)  # Append a new list as a row to all_data

# Create a DataFrame from the list of lists
onehopdegreedf = pd.DataFrame(onedegree_data, columns= [f'{i} degree connections' for i in range(1, 1046)])

# Display the DataFrame
print(onehopdegreedf)

# Assuming you have a list of 100 values named values_to_count
values_to_count = range(1046)  # Replace ... with the actual values

# Add columns for each value in values_to_count
for value in values_to_count:
    column_name = f'{value}_count'
    onehopdegreedf[column_name] = onehopdegreedf.apply(lambda row: row.eq(value).sum(), axis=1)

print(onehopdegreedf)
```

```
      1 degree connections  2 degree connections  3 degree connections  \
0                       17                  10.0                  17.0
1                      347                  22.0                  31.0
2                      347                  15.0                  21.0
3                      347                  57.0                  69.0
4                      347                   9.0                   5.0
...                    ...                   ...                   ...
4034                    59                  18.0                   NaN
4035                    59                   NaN                   NaN
4036                    59                   4.0                   NaN
4037                    59                  14.0                  12.0
4038                    59                   6.0                  10.0

      4 degree connections  5 degree connections  6 degree connections  \
0                     10.0                  13.0                   6.0
1                      8.0                  10.0                  20.0
2                     17.0                  14.0                  14.0
3                     68.0                  76.0                  24.0
4                     10.0                   9.0                   9.0
...                    ...                   ...                   ...
4034                   NaN                   NaN                   NaN
4035                   NaN                   NaN                   NaN
4036                   NaN                   NaN                   NaN
4037                   8.0                   NaN                   NaN
4038                   6.0                  12.0                   8.0

      7 degree connections  8 degree connections  9 degree connections  \
0                     20.0                   8.0                  57.0
1                     21.0                  62.0                   7.0
2                     26.0                  19.0                   8.0
3                     14.0                  63.0                  43.0
4                      9.0                  10.0                   9.0
...                    ...                   ...                   ...
4034                   NaN                   NaN                   NaN
4035                   NaN                   NaN                   NaN
4036                   NaN                   NaN                   NaN
4037                   NaN                   NaN                   NaN
4038                  18.0                   7.0                  11.0
```

```
      10 degree connections  ...  1036 degree connections  \
0                      10.0  ...                      NaN
1                      18.0  ...                      NaN
2                      18.0  ...                      NaN
3                      46.0  ...                      NaN
4                       9.0  ...                      NaN
...                     ...  ...                      ...
4034                    NaN  ...                      NaN
4035                    NaN  ...                      NaN
4036                    NaN  ...                      NaN
4037                    NaN  ...                      NaN
4038                    NaN  ...                      NaN

      1037 degree connections  1038 degree connections  \
0                         NaN                      NaN
1                         NaN                      NaN
2                         NaN                      NaN
3                         NaN                      NaN
4                         NaN                      NaN
```

```
df_dropped = onehopdegreedf.drop(onehopdegreedf.columns[:1045], axis=1)
print(df_dropped)
```

```
#dropping columns in degreedf
df_dropped = df_dropped.drop(df_dropped.columns[793:1045], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[756:792], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[548:755], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[348:547], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[295:347], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[292:294], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[255:291], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[246:254], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[236:245], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[232:234], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[230:231], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[227:229], axis=1)
df_dropped = df_dropped.drop(df_dropped.columns[225:226], axis=1)
# Display the DataFrame after dropping columns
print(df_dropped)
```

```
      0_count  1_count  2_count  3_count  4_count  5_count  6_count  7_count  \
0           0       14       27       22       17       12       15       18
1           0        0        0        0        0        0        0        1
2           0        0        0        0        0        0        0        0
3           0        0        0        1        0        1        0        0
4           0        0        0        0        0        1        0        0
...       ...      ...      ...      ...      ...      ...      ...      ...
4034        0        0        0        0        0        0        0        0
4035        0        0        0        0        0        0        0        0
4036        0        0        0        0        1        0        0        0
4037        0        0        0        0        0        0        0        0
4038        0        0        0        0        0        0        2        1

      8_count  9_count  ...  235_count  245_count  254_count  291_count  \
0          15       15  ...          0          0          0          0
1           1        0  ...          0          0          0          0
2           1        0  ...          0          0          0          0
3           0        0  ...          0          0          0          0
4           0        6  ...          0          0          0          0
...       ...      ...  ...        ...        ...        ...        ...
4034        0        0  ...          0          0          0          0
4035        0        0  ...          0          0          0          0
4036        0        0  ...          0          0          0          0
4037        1        0  ...          0          0          0          0
4038        1        0  ...          0          0          0          0

      294_count  347_count  547_count  755_count  792_count  1045_count
0             0          0          0          0          0           1
1             0          1          0          0          0           0
2             0          1          0          0          0           0
3             0          1          0          0          0           0
4             0          1          0          0          0           0
...         ...        ...        ...        ...        ...         ...
4034          0          0          0          0          0           0
4035          0          0          0          0          0           0
4036          0          0          0          0          0           0
4037          0          0          0          0          0           0
4038          0          0          0          0          0           0

[4039 rows x 239 columns]
```

```
import pandas as pd
A = nx.average_neighbor_degree(G)
B = nx.node_degree_xy(G)
print(A)
average_degrees = [round(value, 2) for value in A.values()]

# Create a DataFrame with a single column
adcdf = pd.DataFrame({'Average Degree Connectivity for Node': average_degrees})

# Display the DataFrame
print(adcdf)
```

```
{0: 18.959654178674352, 1: 48.23529411764706, 2: 49.9, 3: 59.76470588235294, 4: 42.6, 5: 50.61538461538461, 6: 63.5, 7: 45.9, 8: 48.375,
      Average Degree Connectivity for Node
0                                    18.96
1                                    48.24
2                                    49.90
3                                    59.76
4                                    42.60
...                                    ...
4034                                 38.50
4035                                 59.00
4036                                 31.50
4037                                 23.25
4038                                 15.22

[4039 rows x 1 columns]
```

```
import pandas as pd
import networkx as nx

dcData = []

for node in G.nodes():
    neighbors = list(G.neighbors(node))

    # Include the node itself in the subgraph
    subgraph = G.subgraph([node] + neighbors)

    # Calculate the average neighbor degree for the subgraph
    average_neighbor_degree = nx.average_neighbor_degree(subgraph)

    # Round the values to 2 decimal places
    average_degrees = {k: round(v, 2) for k, v in average_neighbor_degree.items()}

    # Sum the average degrees
    B = sum(average_degrees.values())

    dcData.append(B)

# Create a DataFrame with a single column
oneadcdf = pd.DataFrame({'Average Degree Connectivity for network': dcData})

# Display the DataFrame
print(oneadcdf)
```

```
      Average Degree Connectivity for network
0                                    28959.92
1                                      183.50
2                                      101.99
3                                      232.98
4                                      101.84
...                                       ...
4034                                     6.00
4035                                     2.00
4036                                     6.00
4037                                    17.34
4038                                    68.15

[4039 rows x 1 columns]
```

```
node_bridge_counts = {}

# Iterate over nodes from 0 to (number of nodes - 1)
for node in range(len(G)):
    # Create a subgraph containing only the current node and its neighbors
```

```
    subgraph = G.subgraph([node] + list(G.neighbors(node)))

    # Calculate local bridges for the subgraph
    local_bridges = list(nx.local_bridges(subgraph, with_span=False))

    # Count the number of bridges for the current node
    bridge_count = len(local_bridges)

    # Store the result in the dictionary
    node_bridge_counts[node] = bridge_count



# Convert the dictionary to a DataFrame
onehopbrdf = pd.DataFrame(list(node_bridge_counts.values()), columns=['Bridge_Count'])

# Display the DataFrame
print(onehopbrdf)
```

```
      Bridge_Count
0               14
1                0
2                0
3                0
4                0
...            ...
4034             0
4035             1
4036             0
4037             0
4038             0

[4039 rows x 1 columns]
```

```
#Returns the core number for each vertex.

#A k-core is a maximal subgraph that contains nodes of degree k or more.

#The core number of a node is the largest value k of a k-core containing that node.
C = nx.core_number(G)
print(C)

coredf = pd.DataFrame(list(C.values()), columns=['Core_Number'])

# Display the DataFrame
print(coredf)
```

```
{0: 21, 1: 13, 2: 9, 3: 13, 4: 9, 5: 10, 6: 5, 7: 12, 8: 5, 9: 21, 10: 10, 11: 1, 12: 1, 13: 21, 14: 10, 15: 1, 16: 9, 17: 9, 18: 1, 19:
      Core_Number
0              21
1              13
2               9
3              13
4               9
...           ...
4034            2
4035            1
4036            2
4037            4
4038            5

[4039 rows x 1 columns]
```

```
onekcore_counts = []

# Iterate over nodes from 0 to (number of nodes - 1)
for node in G.nodes():
    # Create a subgraph containing only the current node and its neighbors
    subgraph = G.subgraph([node] + list(G.neighbors(node)))

    # Calculate local bridges for the subgraph
    k_core = nx.core_number(subgraph)
    B = sum(list(k_core.values()))

    onekcore_counts.append(B)

# Convert the list to a DataFrame
onehopkcoredf = pd.DataFrame({'k_core for network': onekcore_counts})

# Display the DataFrame
print(onehopkcoredf)
```

```
      k_core for network
0                   3400
1                    106
2                     86
3                    170
4                     94
...                  ...
4034                   6
4035                   2
4036                   6
4037                  14
4038                  44

[4039 rows x 1 columns]
```

```
#Returns the layer of each vertex in an onion decomposition of the graph.

#The onion decomposition refines the k-core decomposition by providing information on the internal organization of each k-shell.
#It is usually used alongside the core numbers.#
O = nx.onion_layers(G)
print(O)
sorted_keys = sorted(O.keys())

# Create a DataFrame from the dictionary with sorted keys
oniondf = pd.DataFrame.from_dict({key: O[key] for key in sorted_keys}, orient='index', columns=['onion layers'])

# Display the DataFrame
print(oniondf)
```

```
{11: 1, 12: 1, 15: 1, 18: 1, 37: 1, 43: 1, 74: 1, 114: 1, 209: 1, 210: 1, 215: 1, 287: 1, 292: 1, 335: 1, 911: 1, 918: 1, 1096: 1, 1119:
      onion layers
0              108
1               51
2               30
3               49
4               30
...            ...
4034             2
4035             1
4036             2
4037             8
4038            14

[4039 rows x 1 columns]
```

```
oneonion_counts = []

# Iterate over nodes from 0 to (number of nodes - 1)
for node in G.nodes():
    # Create a subgraph containing only the current node and its neighbors
    subgraph = G.subgraph([node] + list(G.neighbors(node)))

    # Calculate local bridges for the subgraph
    k_core = nx.onion_layers(subgraph)
    sorted_keys = sorted(k_core.values())

    B = sum(list(sorted_keys))

    oneonion_counts.append(B)

# Convert the list to a DataFrame
oneoniondf = pd.DataFrame({'onion layers for network': oneonion_counts})

# Display the DataFrame
print(oneoniondf)
```

```
      onion layers for network
0                         8567
1                           47
2                           29
3                           87
4                           21
...                        ...
4034                         3
4035                         2
4036                         3
4037                         9
4038                        31

[4039 rows x 1 columns]
```

```
#Returns the boundary expansion of the set S.

#The boundary expansion is the quotient of the size of the node boundary and the cardinality of S. [1]
oneboundary_expansion_list = []

# Iterate over all nodes in the graph
for node in G.nodes():
    # Create a subgraph with the current node and its neighbors
    subgraph = G.subgraph([node] + list(G.neighbors(node)))

    # Calculate the boundary expansion for the current node
    boundary_expansion = nx.boundary_expansion(G, subgraph)

    # Append the boundary expansion to the list
    oneboundary_expansion_list.append(boundary_expansion)

# Create a DataFrame from the list
onebounddf = pd.DataFrame({'Boundary_Expansion': oneboundary_expansion_list})

# Display the DataFrame
print(onebounddf)
```

```
      Boundary_Expansion
0               3.364943
1              18.333333
2              30.636364
3              18.333333
4              30.636364
...                  ...
4034           19.000000
4035           29.000000
4036           19.000000
4037           11.000000
4038            5.000000

[4039 rows x 1 columns]
```

```
#The eccentricity of a node v is the maximum distance from v to all other nodes in G.
ecc = nx.eccentricity(G, v=None, sp=None, weight=None)
eccdf = pd.DataFrame(list(ecc.values()), columns=['eccentricity'])
print(eccdf)
```

```
      eccentricity
0                6
1                7
2                7
3                7
4                7
...            ...
4034             8
4035             8
4036             8
4037             8
4038             8

[4039 rows x 1 columns]
```

```
#The Kemeny constant measures the time needed for spreading across a graph. Low values indicate a closely connected graph whereas high value
onekemeny_list = []

# Iterate over all nodes in the graph
for node in G.nodes():
    # Create a subgraph with the current node and its neighbors
    subgraph = G.subgraph([node] + list(G.neighbors(node)))

    # Calculate the boundary expansion for the current node
    kemeny = nx.kemeny_constant(subgraph)

    # Append the boundary expansion to the list
    onekemeny_list.append(kemeny)

# Create a DataFrame from the list
onekemenydf = pd.DataFrame({'kemeny constant for network': onekemeny_list})

# Display the DataFrame
print(onekemenydf)
```

```
      kemeny constant for network
0                      406.232390
1                       17.493435
2                        9.177039
3                       16.649210
4                        9.196660
...                           ...
4034                     1.333333
4035                     0.500000
4036                     1.333333
4037                     3.362500
4038                     8.546052

[4039 rows x 1 columns]
```

```
#The average global efficiency of a graph is the average efficiency of all pairs of nodes
oneglobaleff_list = []
oneweight_matching_list = []
onerandom_list = []
panther_list = []
# Iterate over all nodes in the graph
for node in G.nodes():
    # Create a subgraph with the current node and its neighbors
    subgraph = G.subgraph([node] + list(G.neighbors(node)))

    # Calculate the boundary expansion for the current node
    globaleff = nx.global_efficiency(subgraph)

    weighteff = len(list(nx.max_weight_matching(subgraph, maxcardinality=False, weight='weight')))

    result = nx.non_randomness(subgraph, k=None, weight='weight')
    second_value = result[1]

    #panther = len(nx.panther_similarity(subgraph, 0, k=5, path_length=5, c=0.5, delta=0.1, eps=None, weight='weight'))

    # Append the boundary expansion to the list
    oneglobaleff_list.append(globaleff)
    oneweight_matching_list.append(weighteff)
    onerandom_list.append(second_value)
    #panther_list.append(panther)
# Create a DataFrame from the list
oneglobaldf = pd.DataFrame({
    'global_efficiency': oneglobaleff_list,
    'max_weight_matching': oneweight_matching_list,
    'random_probability': onerandom_list,
    # 'panther_similarity': panther_list  # Uncomment if needed
})
# Display the DataFrame
print(oneglobaldf)
```

```
/usr/local/lib/python3.10/dist-packages/networkx/algorithms/non_randomness.py:94: RuntimeWarning: invalid value encountered in double_sc
  nr_rd = (nr - ((n - 2 * k) * p + k)) / math.sqrt(2 * k * p * (1 - p))
/usr/local/lib/python3.10/dist-packages/networkx/algorithms/non_randomness.py:94: RuntimeWarning: divide by zero encountered in double_s
  nr_rd = (nr - ((n - 2 * k) * p + k)) / math.sqrt(2 * k * p * (1 - p))
      global_efficiency  max_weight_matching  random_probability
0              0.523734                 164          75.626117
1              0.741830                   9           0.814521
2              0.954545                   5           0.088836
3              0.836601                   9           1.395870
4              0.945455                   5           0.364554
...                 ...                 ...                ...
4034           1.000000                   1               -inf
4035           1.000000                   1                NaN
4036           1.000000                   1               -inf
4037           0.900000                   2          -0.135403
4038           0.822222                   5           0.266415

[4039 rows x 3 columns]
```

```
onegldf = oneglobaldf.fillna(0)
print(onegldf)
```

```
      global_efficiency  max_weight_matching  random_probability
0              0.523734                 164          75.626117
1              0.741830                   9           0.814521
2              0.954545                   5           0.088836
3              0.836601                   9           1.395870
4              0.945455                   5           0.364554
...                 ...                 ...                ...
4034           1.000000                   1               -inf
4035           1.000000                   1           0.000000
4036           1.000000                   1               -inf
4037           0.900000                   2          -0.135403
4038           0.822222                   5           0.266415

[4039 rows x 3 columns]
```

```
import numpy as np
onegldf.replace([np.inf, -np.inf], 0, inplace=True)
print(onegldf)
```

```
      global_efficiency  max_weight_matching  random_probability
0              0.523734                  164           75.626117
1              0.741830                    9            0.814521
2              0.954545                    5            0.088836
3              0.836601                    9            1.395870
4              0.945455                    5            0.364554
...                 ...                  ...                 ...
4034           1.000000                    1            0.000000
4035           1.000000                    1            0.000000
4036           1.000000                    1            0.000000
4037           0.900000                    2           -0.135403
4038           0.822222                    5            0.266415

[4039 rows x 3 columns]
```

```
#dataset final
onehopdf = pd.concat([nodedf, df_dropped, onehopcommondf, adcdf, oneadcdf, onehopbrdf, coredf, onehopkcoredf, oniondf, oneoniondf, oneboundd
print(onehopdf)
```

```
     4035  4035           1        0        0        0        0        0        0
     4036  4036           2        0        0        0        0        1        0
     4037  4037           4        0        0        0        0        0        0
     4038  4038           9        0        0        0        0        0        0

           6_count  7_count  ...  Core_Number  k_core for network  onion layers  \
     0          15       18  ...           21                3400           108
     1           0        1  ...           13                 106            51
     2           0        0  ...            9                  86            30
     3           0        0  ...           13                 170            49
     4           0        0  ...            9                  94            30
     ...       ...      ...  ...          ...                 ...           ...
     4034        0        0  ...            2                   6             2
     4035        0        0  ...            1                   2             1
     4036        0        0  ...            2                   6             2
     4037        0        0  ...            4                  14             8
     4038        2        1  ...            5                  44            14

           onion layers for network  Boundary_Expansion  \
     0                         8567            3.364943
     1                           47           18.333333
     2                           29           30.636364
     3                           87           18.333333
     4                           21           30.636364
     ...                        ...                 ...
     4034                         3           19.000000
     4035                         2           29.000000
     4036                         3           19.000000
     4037                         9           11.000000
     4038                        31            5.000000

           kemeny constant for network  global_efficiency  max_weight_matching  \
     0                      406.232390           0.523734                  164
     1                       17.493435           0.741830                    9
     2                        9.177039           0.954545                    5
     3                       16.649210           0.836601                    9
     4                        9.196660           0.945455                    5
     ...                           ...                ...                  ...
     4034                     1.333333           1.000000                    1
     4035                     0.500000           1.000000                    1
     4036                     1.333333           1.000000                    1
     4037                     3.362500           0.900000                    2
     4038                     8.546052           0.822222                    5

           random_probability  eccentricity
     0              75.626117             6
     1               0.814521             7
     2               0.088836             7
     3               1.395870             7
     4               0.364554             7
     ...                  ...           ...
     4034            0.000000             8
     4035            0.000000             8
     4036            0.000000             8
     4037           -0.135403             8
     4038            0.266415             8

     [4039 rows x 255 columns]
```

```
onehopdf.to_csv('/content/drive/MyDrive/Colab Notebooks/onehopdata.csv', index=False)
```

```
import pandas as pd

tri = nx.triangles(G, nodes=None)
tridf = pd.DataFrame(list(tri.values()), columns=['No of triangles with Node'])
print(tridf)
```

```
      No of triangles with Node
0                          2519
1                            57
2                            40
3                            86
4                            39
...                         ...
4034                           1
4035                           0
4036                           1
4037                           4
4038                          20

[4039 rows x 1 columns]
```

```
onetriangle_counts = []

# Iterate over nodes from 0 to (number of nodes - 1)
for node in G.nodes():
    # Create a subgraph containing only the current node and its neighbors
    subgraph = G.subgraph([node] + list(G.neighbors(node)))

    # Calculate local bridges for the subgraph
    triangle = nx.triangles(subgraph, nodes=None)
    B = sum(list(triangle.values()))

    onetriangle_counts.append(B)

# Convert the list to a DataFrame
onehopktriangledf = pd.DataFrame({'triangle for network': onetriangle_counts})

# Display the DataFrame
print(onehopktriangledf)
```

```
      triangle for network
0                    39777
1                      402
2                      381
3                      996
4                      378
...                    ...
4034                     3
4035                     0
4036                     3
4037                    15
4038                   114

[4039 rows x 1 columns]
```

```
path = '/content/drive/MyDrive/Colab Notebooks/onehopdata.csv'
onehop = pd.read_csv(path)
print(onehop)


#Final Dataset for Onehop step
onefinaldf = pd.concat([onehop, tridf, onehopktriangledf], axis=1)
print(onefinaldf)
```

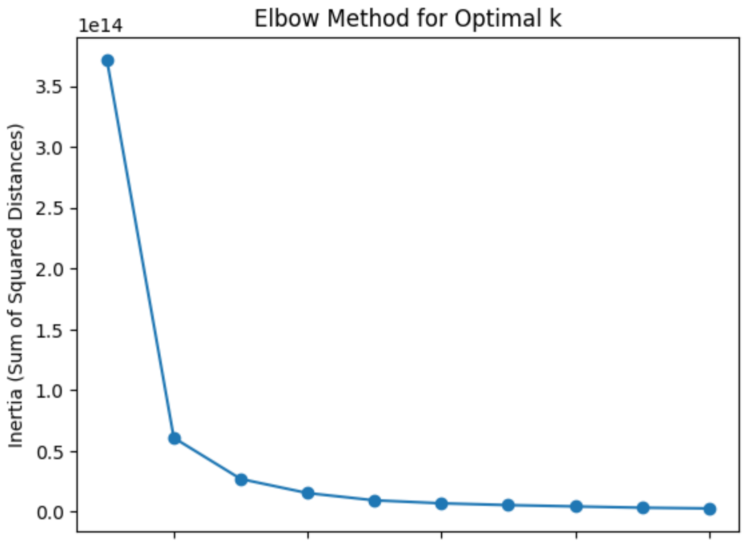
```
     4035  4035           1        0        0        0        0        0        0
     4036  4036           2        0        0        0        0        1        0
     4037  4037           4        0        0        0        0        0        0
     4038  4038           9        0        0        0        0        0        0

      6_count  7_count  ...  onion layers  onion layers for network  \
0          15       18  ...           108                      8567
1           0        1  ...            51                        47
```

```
4038        2        1  ...            14                      31

      Boundary_Expansion  kemeny constant for network  global_efficiency  \
0               3.364943                   406.232390           0.523734
1              18.333333                    17.493435           0.741830
2              30.636364                     9.177039           0.954545
3              18.333333                    16.649210           0.836601
4              30.636364                     9.196660           0.945455
...                  ...                          ...                ...
4034           19.000000                     1.333333           1.000000
4035           29.000000                     0.500000           1.000000
4036           19.000000                     1.333333           1.000000
4037           11.000000                     3.362500           0.900000
4038            5.000000                     8.546052           0.822222

      max_weight_matching  random_probability  eccentricity  \
0                     164           75.626117             6
1                       9            0.814521             7
2                       5            0.088836             7
3                       9            1.395870             7
4                       5            0.364554             7
...                   ...                 ...           ...
4034                    1            0.000000             8
4035                    1            0.000000             8
4036                    1            0.000000             8
4037                    2           -0.135403             8
4038                    5            0.266415             8

      No of triangles with Node  triangle for network
0                          2519                 39777
1                            57                   402
2                            40                   381
3                            86                   996
4                            39                   378
...                         ...                   ...
4034                          1                     3
4035                          0                     0
4036                          1                     3
4037                          4                    15
4038                         20                   114

[4039 rows x 257 columns]
```

K Means Clustering on the Data by determing the number of steps then plotting by the number of clusters which is determined by Elbow method and then plot accordingly

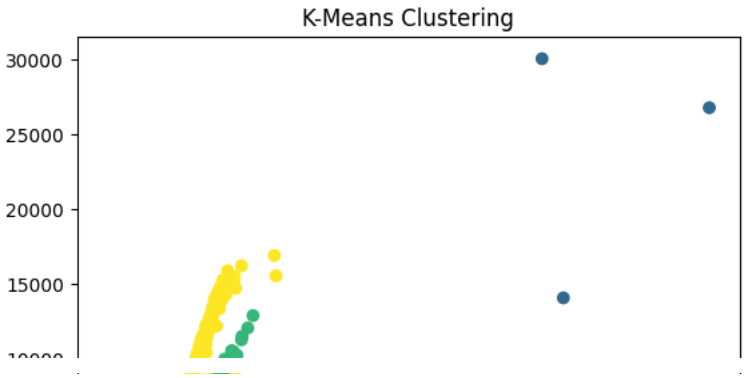
```
#determining the number of Steps for K means
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Assuming df is your DataFrame with standardized features
# Step 1: Standardize the data
scaler = StandardScaler()
features_standardized = scaler.fit_transform(onefinaldf)

# Store the sum of squared distances for different values of k
inertia_values = []

# Define a range of k values to try
k_values = range(1, 11)  # You can adjust the range based on your problem

# Apply k-means clustering for each k and store the inertia
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(onefinaldf)
    inertia_values.append(kmeans.inertia_)

# Plot the elbow curve
plt.plot(k_values, inertia_values, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia (Sum of Squared Distances)')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
```




```
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Step 1: Standardize the data
scaler = StandardScaler()
features_standardized = scaler.fit_transform(onefinaldf)

# Step 3: Apply k-means clustering
k = 4  # Replace with your optimal k
kmeans = KMeans(n_clusters=k, random_state=42)
onefinaldf['k=4_Cluster'] = kmeans.fit_predict(features_standardized)

# Display the DataFrame with cluster assignments
print(onefinaldf)

# Step 4: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
# Visualization is more challenging with a large number of features
# You may need to choose specific features for plotting or use advanced visualization techniques
plt.scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k=4_Cluster'], cmap='viridis')
plt.title('K-Means Clustering')
plt.show()
```

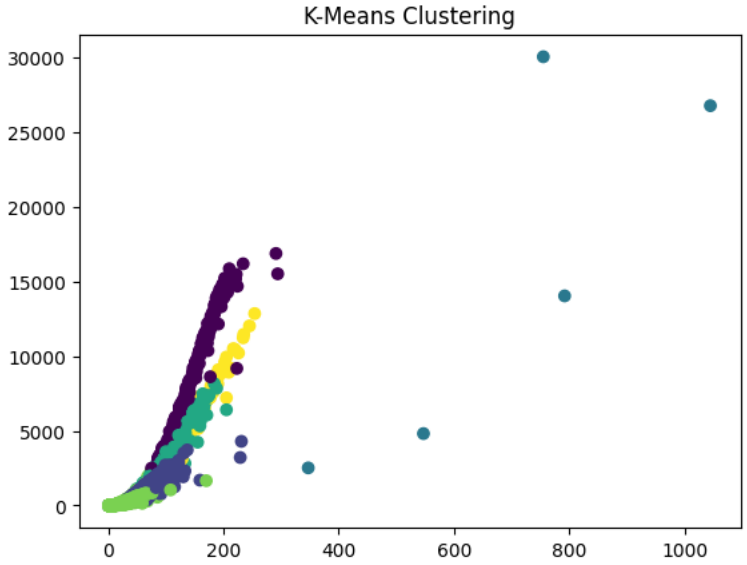
```
usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from
 warnings.warn(
     Node  NodeDegree  0_count  1_count  2_count  3_count  4_count  5_count  \
        0         347        0       14       27       22       17       12
        1          17        0        0        0        0        0        0
        2          10        0        0        0        0        0        0
        3          17        0        0        0        1        0        1
        4          10        0        0        0        0        0        1
..    ...         ...      ...      ...      ...      ...      ...      ...
034  4034           2        0        0        0        0        0        0
035  4035           1        0        0        0        0        0        0
036  4036           2        0        0        0        0        1        0
037  4037           4        0        0        0        0        0        0
038  4038           9        0        0        0        0        0        0

     6_count  7_count  ...  onion layers for network  Boundary_Expansion  \
          15       18  ...                      8567            3.364943
           0        1  ...                        47           18.333333
           0        0  ...                        29           30.636364
           0        0  ...                        87           18.333333
           0        0  ...                        21           30.636364
..       ...      ...  ...                       ...                 ...
034        0        0  ...                         3           19.000000
035        0        0  ...                         2           29.000000
036        0        0  ...                         3           19.000000
037        0        0  ...                         9           11.000000
038        2        1  ...                        31            5.000000

     kemeny constant for network  global_efficiency  max_weight_matching  \
                       406.232390           0.523734                  164
                        17.493435           0.741830                    9
                         9.177039           0.954545                    5
                        16.649210           0.836601                    9
                         9.196660           0.945455                    5
..                            ...                ...                  ...
034                      1.333333           1.000000                    1
035                      0.500000           1.000000                    1
036                      1.333333           1.000000                    1
037                      3.362500           0.900000                    2
038                      8.546052           0.822222                    5

     random_probability  eccentricity  No of triangles with Node  \
               75.626117             6                       2519
                0.814521             7                         57
                0.088836             7                         40
                1.395870             7                         86
                0.364554             7                         39
..                   ...           ...                        ...
034             0.000000             8                          1
035             0.000000             8                          0
036             0.000000             8                          1
037            -0.135403             8                          4
038             0.266415             8                         20

     triangle for network  k=4_Cluster
                    39777            1
                      402            0
                      381            0
                      996            0
                      378            0
..                    ...          ...
034                     3            0
035                     0            0
036                     3            0
037                    15            0
038                   114            0

4039 rows x 258 columns]
```



```
#Step 3: Apply k-means clustering
N = 6  # Replace with your optimal k
kmeans = KMeans(n_clusters=N, random_state=42)
onefinaldf['k = 6_Cluster'] = kmeans.fit_predict(features_standardized)


# Step 4: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
# Visualization is more challenging with a large number of features
# You may need to choose specific features for plotting or use advanced visualization techniques
plt.scatter(onefinaldf["Common Connections"], onefinaldf['NodeDegree'], c=onefinaldf['k = 6_Cluster'], cmap='viridis')
plt.title('K-Means Clustering')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
```

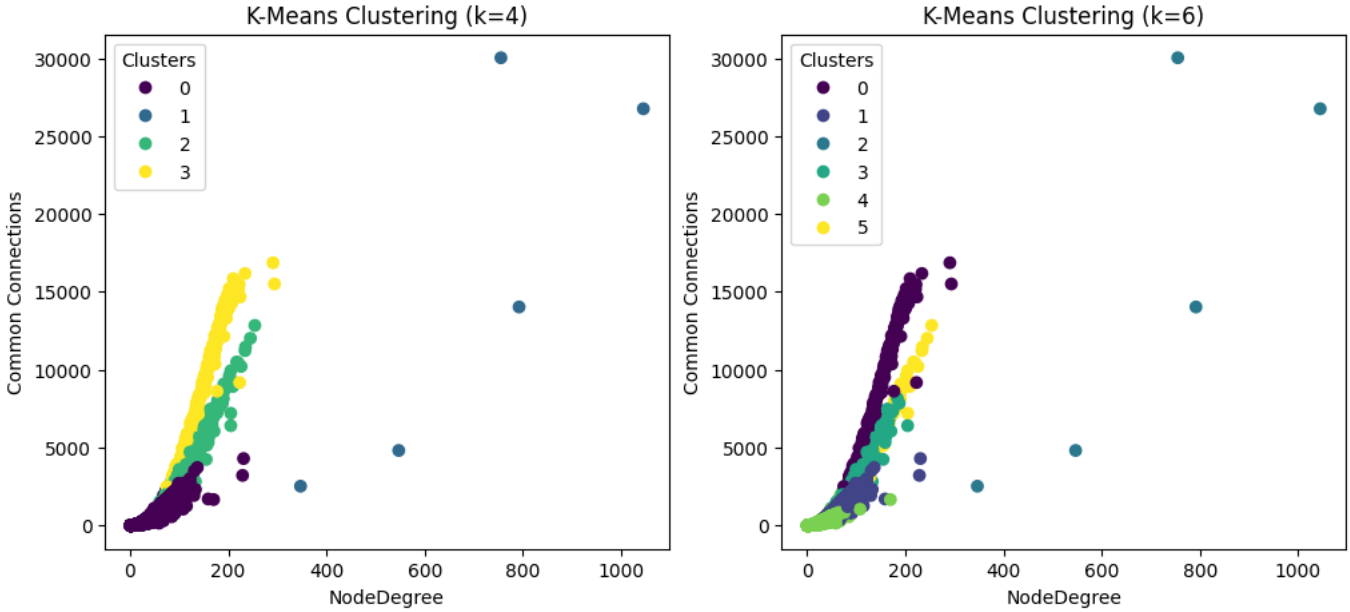

```
# You may need to choose specific features for plotting or use advanced visualization techniques
plt.scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k = 6_Cluster'], cmap='viridis')
plt.title('K-Means Clustering')
plt.show()
```



showing both for 4 clusters and 6 clusters in a same graph

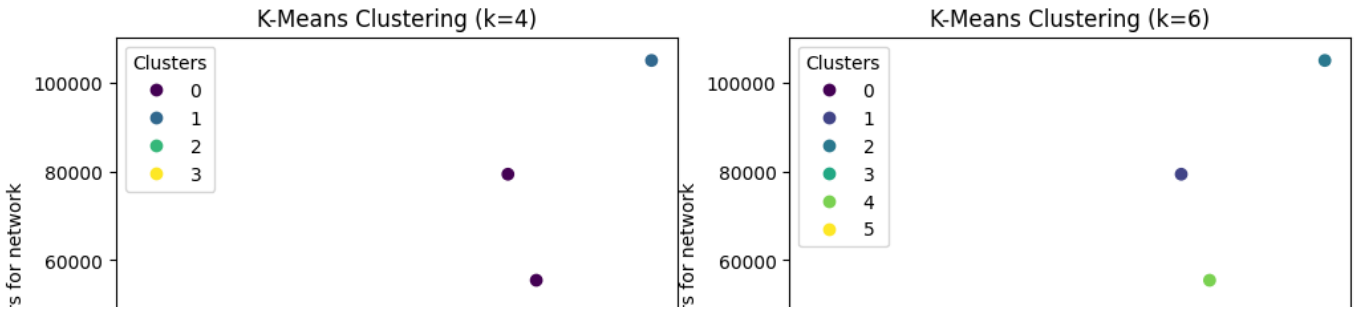
```
# Step 4: Visualize the clusters
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot for k=4 clusters
scatter_4 = axes[0].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k=4_Cluster'], cmap='viridis')
axes[0].set_title(f'K-Means Clustering (k=4)')
axes[0].set_xlabel('NodeDegree')
axes[0].set_ylabel('Common Connections')
axes[0].legend(*scatter_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters
scatter_6 = axes[1].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k = 6_Cluster'], cmap='viridis')
axes[1].set_title(f'K-Means Clustering (k=6)')
axes[1].set_xlabel('NodeDegree')
axes[1].set_ylabel('Common Connections')
axes[1].legend(*scatter_6.legend_elements(), title='Clusters')
```

```
<matplotlib.legend.Legend at 0x7a199b5abe80>
```



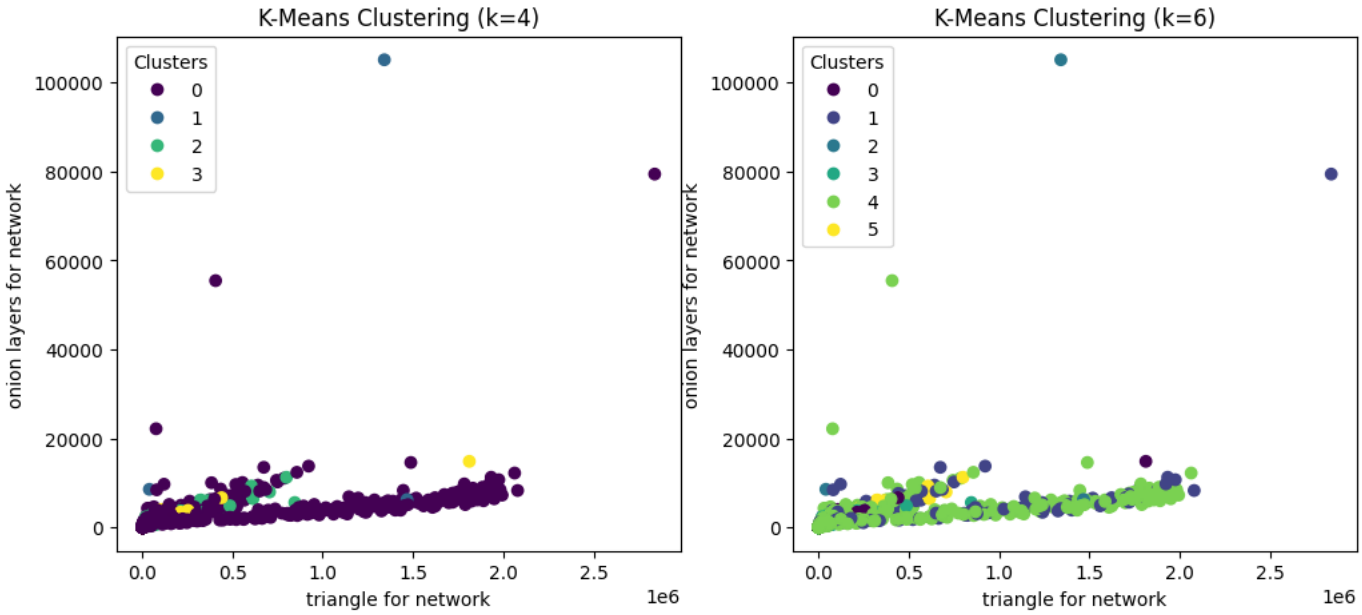

```
# visualize with kemeny constant - time and Onion layers of the network for clustering
# Step 4: Visualize the clusters
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot for k=4 clusters
scatter_4 = axes[0].scatter(onefinaldf["kemeny constant for network"], onefinaldf['onion layers for network'], c=onefinaldf['k=4_Cluster'],
axes[0].set_title(f'K-Means Clustering (k=4)')
axes[0].set_xlabel('kemeny constant for network')
axes[0].set_ylabel('onion layers for network')
axes[0].legend(*scatter_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters
scatter_6 = axes[1].scatter(onefinaldf["kemeny constant for network"], onefinaldf['onion layers for network'], c=onefinaldf['k = 6_Cluster']
axes[1].set_title(f'K-Means Clustering (k=6)')
axes[1].set_xlabel('kemeny constant for network')
axes[1].set_ylabel('onion layers for network')
axes[1].legend(*scatter_6.legend_elements(), title='Clusters')
```

```
<matplotlib.legend.Legend at 0x7a198ff2f0a0>
```



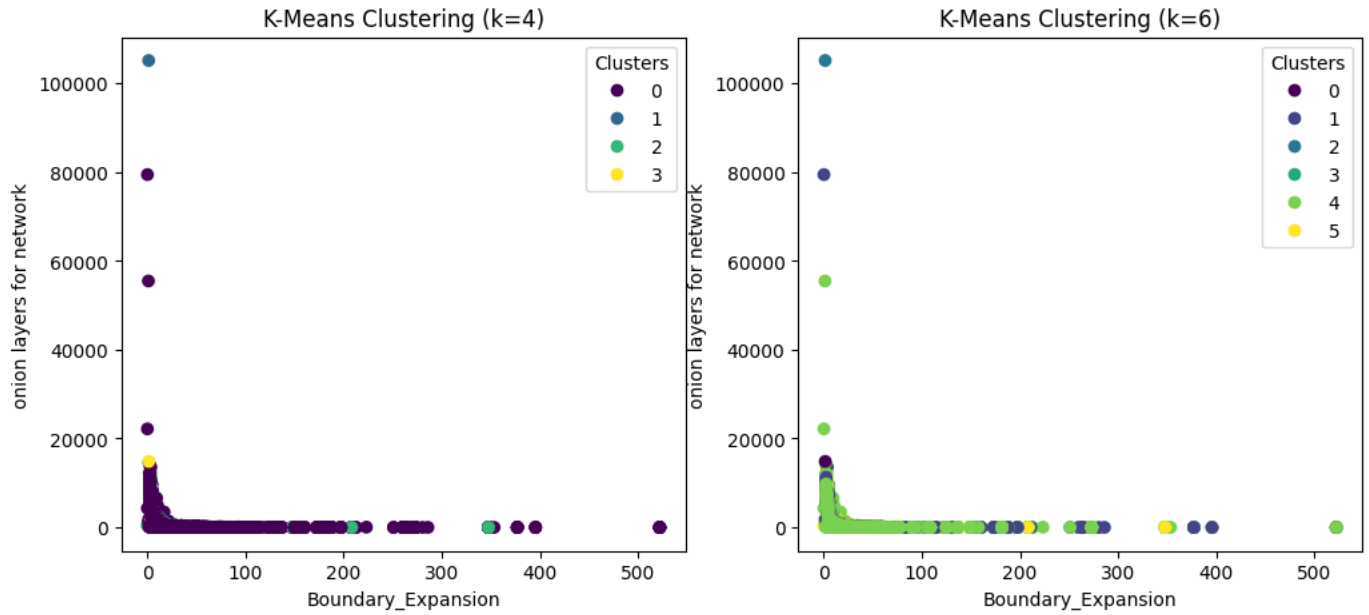

```
# visualize with No of triangle in the network and Onion layers of the network for clustering
# Step 4: Visualize the clusters
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot for k=4 clusters
scatter_4 = axes[0].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['k=4_Cluster'], cmap='v
axes[0].set_title(f'K-Means Clustering (k=4)')
axes[0].set_xlabel('triangle for network')
axes[0].set_ylabel('onion layers for network')
axes[0].legend(*scatter_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters
scatter_6 = axes[1].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['k = 6_Cluster'], cmap=
axes[1].set_title(f'K-Means Clustering (k=6)')
axes[1].set_xlabel('triangle for network')
axes[1].set_ylabel('onion layers for network')
axes[1].legend(*scatter_6.legend_elements(), title='Clusters')
```

```
<matplotlib.legend.Legend at 0x7a199b411f00>
```



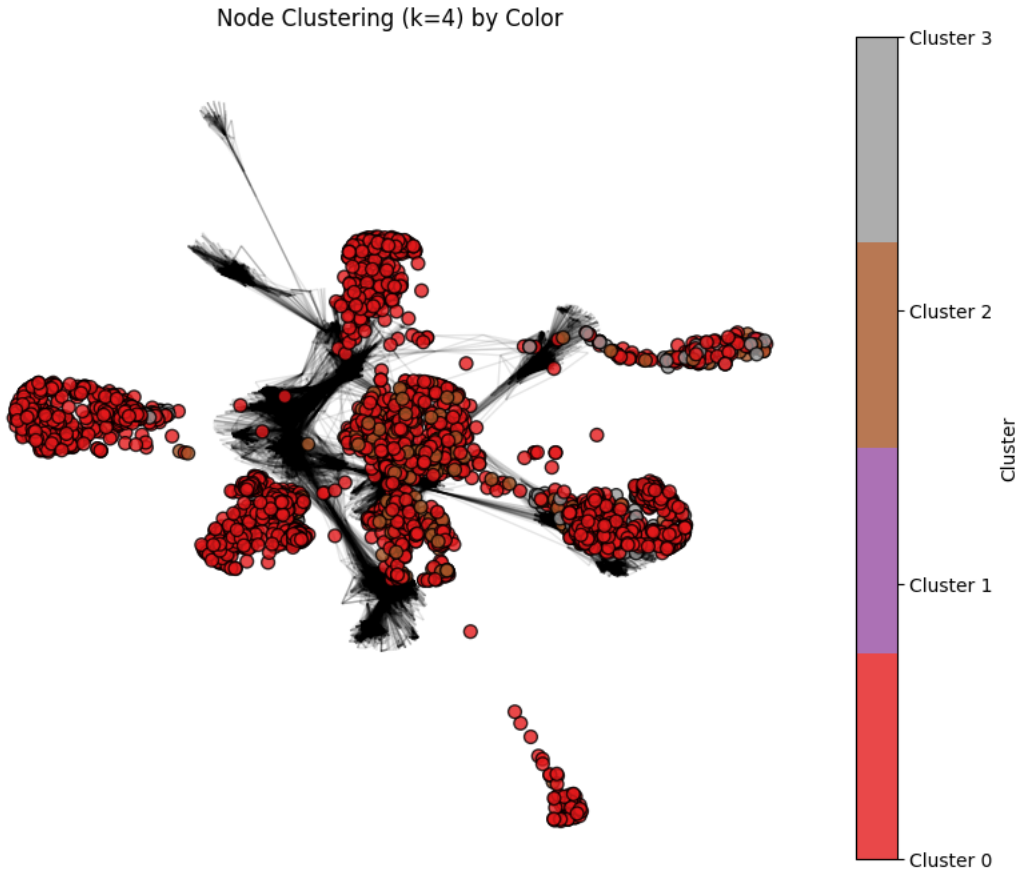

```
# visualize with Boundary Expansion of network and Onion layers of the network for clustering
# Step 4: Visualize the clusters
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot for k=4 clusters
scatter_4 = axes[0].scatter(onefinaldf["Boundary_Expansion"], onefinaldf['onion layers for network'], c=onefinaldf['k=4_Cluster'], cmap='vir
axes[0].set_title(f'K-Means Clustering (k=4)')
axes[0].set_xlabel('Boundary_Expansion')
axes[0].set_ylabel('onion layers for network')
axes[0].legend(*scatter_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters
scatter_6 = axes[1].scatter(onefinaldf["Boundary_Expansion"], onefinaldf['onion layers for network'], c=onefinaldf['k = 6_Cluster'], cmap='v
axes[1].set_title(f'K-Means Clustering (k=6)')
axes[1].set_xlabel('Boundary_Expansion')
axes[1].set_ylabel('onion layers for network')
axes[1].legend(*scatter_6.legend_elements(), title='Clusters')
```

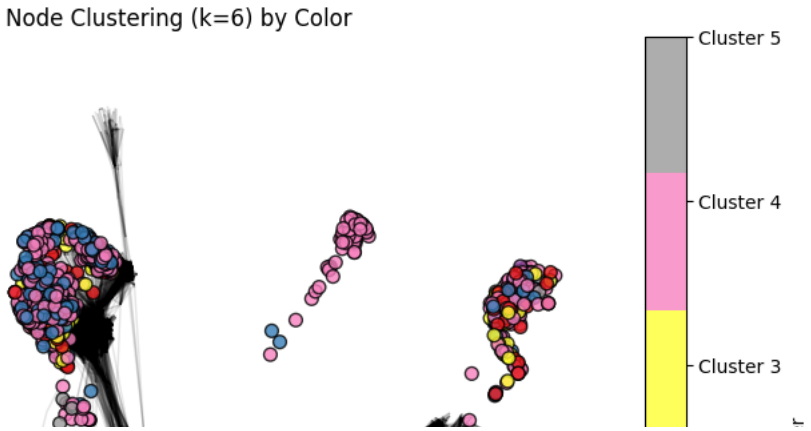
```
<matplotlib.legend.Legend at 0x7a199b675870>
```



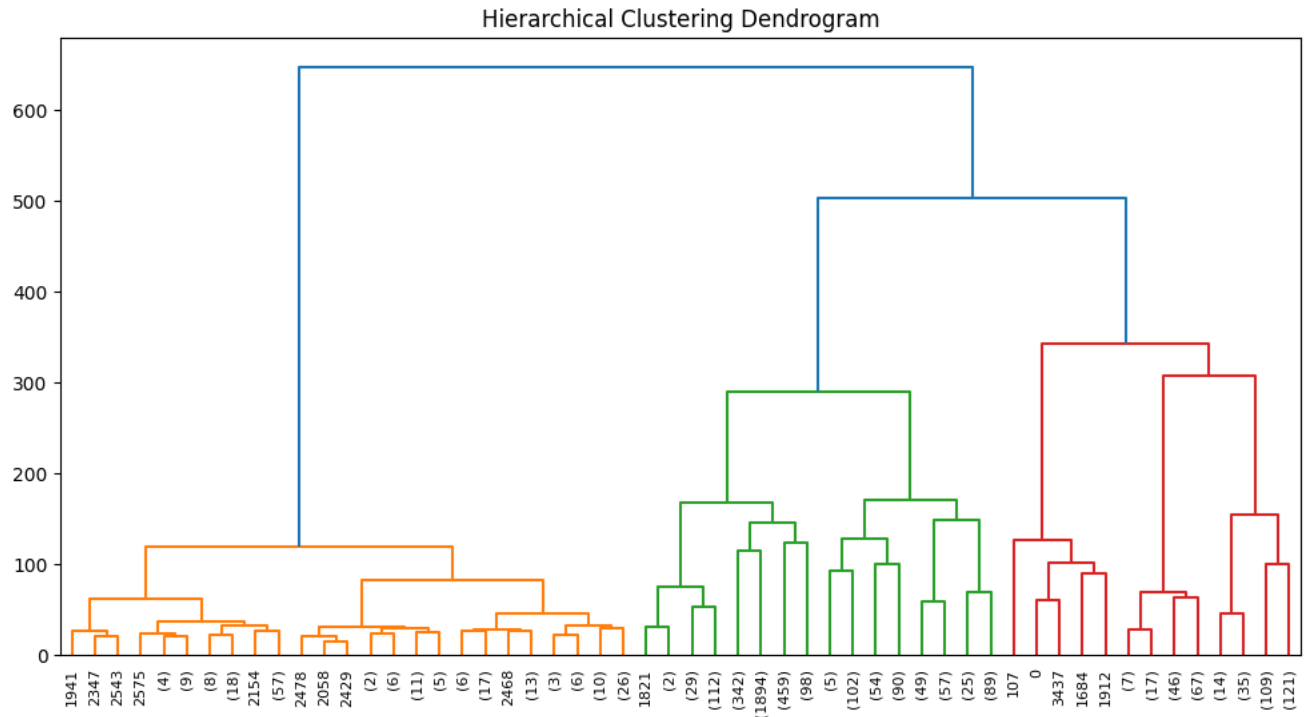

```
import seaborn as sns
def plot_clusters(graph, clusters, k_value):
    # Choose a colormap for clusters
    cluster_colors = sns.color_palette('Set1', n_colors=len(clusters.unique()))

    # Create a figure and axis
    fig, ax = plt.subplots(figsize=(10, 8))

    # Draw nodes with colors based on clusters
    nodes = nx.draw_networkx_nodes(graph, pos=nx.spring_layout(graph), node_color=clusters, cmap=plt.cm.get_cmap('Set1', len(clusters.unique

    # Draw edges
    nx.draw_networkx_edges(graph, pos=nx.spring_layout(graph), alpha=0.1)

    # Add a colorbar
    cbar = plt.colorbar(nodes, ticks=clusters.unique(), label='Cluster')
    cbar.set_ticklabels([f'Cluster {i}' for i in clusters.unique()])

    plt.title(f'Node Clustering (k={k_value}) by Color')
    plt.axis('off')
    plt.show()



# Plot for k=4
plot_clusters(G, onefinaldf['k=4_Cluster'], 4)

# Plot for k=6
plot_clusters(G, onefinaldf['k = 6_Cluster'], 6)
```

```
<ipython-input-113-98016605a855>:10: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will
  nodes = nx.draw_networkx_nodes(graph, pos=nx.spring_layout(graph), node_color=clusters, cmap=plt.cm.get_cmap('Set1', len(clusters
```





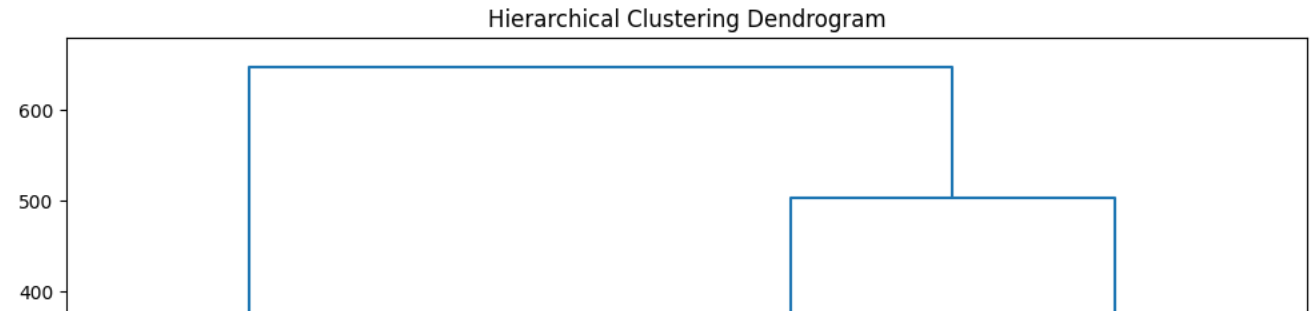
```
# Agglomerative Clustering
import pandas as pd
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage



# Step 1: Standardize the data (optional but recommended for hierarchical clustering)
scaler = StandardScaler()
data_standardized = scaler.fit_transform(onefinaldf)

# Step 2: Determine the linkage matrix using the ward method
linkage_matrix = linkage(data_standardized, method='ward')

# Step 3: Plot the dendrogram (optional but useful for visualizing hierarchical clustering)
plt.figure(figsize=(12, 6))
dendrogram(linkage_matrix, leaf_rotation=90., leaf_font_size=8., truncate_mode='level', p=5)
plt.title('Hierarchical Clustering Dendrogram')
plt.show()

# Step 4: Apply Agglomerative Hierarchical Clustering
# Choose the appropriate number of clusters (n_clusters) based on the dendrogram
n_clusters = 4  # Replace with your desired number of clusters
hierarchical_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward')
onefinaldf['agg4Cluster'] = hierarchical_clustering.fit_predict(data_standardized)

# Display the DataFrame with cluster assignments
print(onefinaldf)
```



	Node	NodeDegree	0_count	1_count	2_count	3_count	4_count	5_count	\
0	0	347	0	14	27	22	17	12	
1	1	17	0	0	0	0	0	0	
2	2	10	0	0	0	0	0	0	
3	3	17	0	0	0	1	0	1	
4	4	10	0	0	0	0	0	1	
...	...	...	...	...	...	...	...	...	
4034	4034	2	0	0	0	0	0	0	
4035	4035	1	0	0	0	0	0	0	
4036	4036	2	0	0	0	0	1	0	
4037	4037	4	0	0	0	0	0	0	
4038	4038	9	0	0	0	0	0	0	

	6_count	7_count	...	kemeny constant for network	global_efficiency	\
0	15	18	...	406.232390	0.523734	
1	0	1	...	17.493435	0.741830	
2	0	0	...	9.177039	0.954545	

```
linkage_matrix = linkage(data_standardized, method='ward')

# Step 3: Plot the dendrogram (optional but useful for visualizing hierarchical clustering)
plt.figure(figsize=(12, 6))
dendrogram(linkage_matrix, leaf_rotation=90., leaf_font_size=8., truncate_mode='level', p=7)
plt.title('Hierarchical Clustering Dendrogram')
plt.show()

# Step 4: Apply Agglomerative Hierarchical Clustering
# Choose the appropriate number of clusters (n_clusters) based on the dendrogram
n_clusters = 6  # Replace with your desired number of clusters
hierarchical_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward')
onefinaldf['agg6Cluster'] = hierarchical_clustering.fit_predict(data_standardized)
```

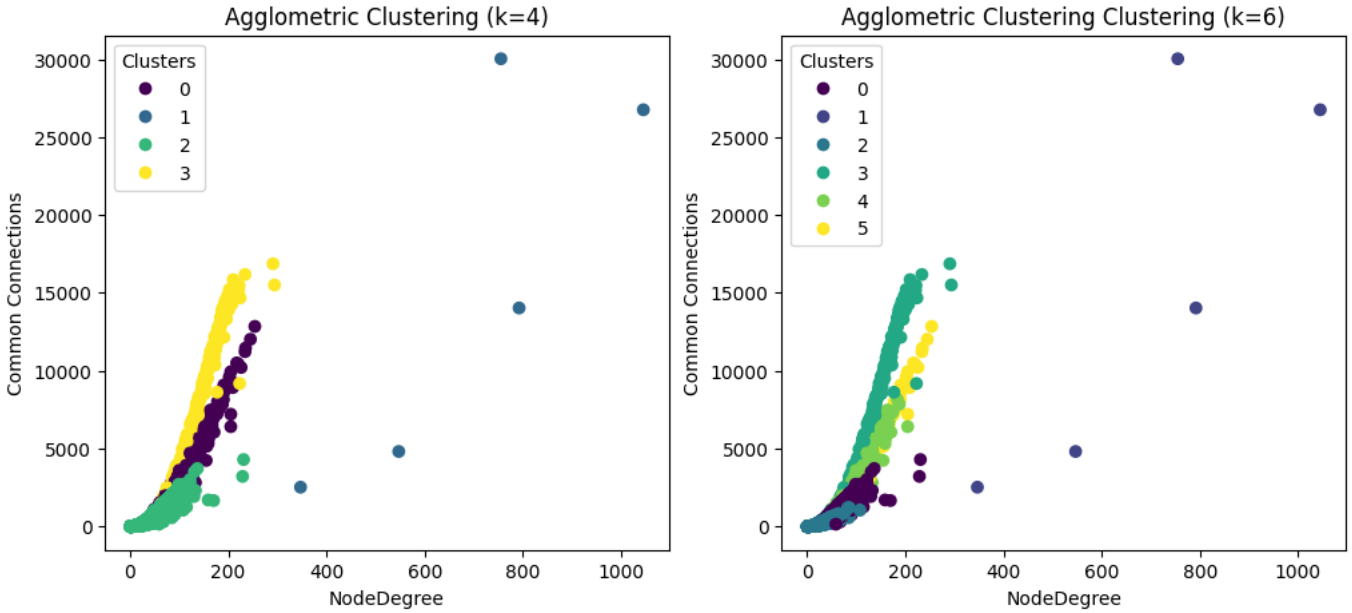



```
print(onefinaldf)
```

```
      4035  4035          1       0       0       0       0       0       0
      4036  4036          2       0       0       0       0       1       0
      4037  4037          4       0       0       0       0       0       0
      4038  4038          9       0       0       0       0       0       0

            6_count  7_count  ...  global_efficiency  max_weight_matching  \
      0          15       18  ...           0.523734                  164
      1           0        1  ...           0.741830                    9
      2           0        0  ...           0.954545                    5
      3           0        0  ...           0.836601                    9
      4           0        0  ...           0.945455                    5
      ...       ...      ...  ...                ...                  ...
      4034        0        0  ...           1.000000                    1
      4035        0        0  ...           1.000000                    1
      4036        0        0  ...           1.000000                    1
      4037        0        0  ...           0.900000                    2
      4038        2        1  ...           0.822222                    5

            random_probability  eccentricity  No of triangles with Node  \
      0             75.626117             6                       2519
      1              0.814521             7                         57
      2              0.088836             7                         40
      3              1.395870             7                         86
      4              0.364554             7                         39
      ...                 ...           ...                        ...
      4034           0.000000             8                          1
      4035           0.000000             8                          0
      4036           0.000000             8                          1
      4037          -0.135403             8                          4
      4038           0.266415             8                         20

            triangle for network  k=4_Cluster  k = 6_Cluster  agg4Cluster  \
      0                   39777            1              2            1
      1                     402            0              4            2
      2                     381            0              4            2
      3                     996            0              4            2
      4                     378            0              4            2
      ...                   ...          ...            ...          ...
      4034                    3            0              4            2
      4035                    0            0              4            2
      4036                    3            0              4            2
      4037                   15            0              4            2
      4038                  114            0              4            2

            agg6Cluster
      0               1
      1               2
      2               2
      3               2
      4               2
      ...           ...
      4034            2
      4035            2
      4036            2
      4037            2
      4038            2

      [4039 rows x 261 columns]
```

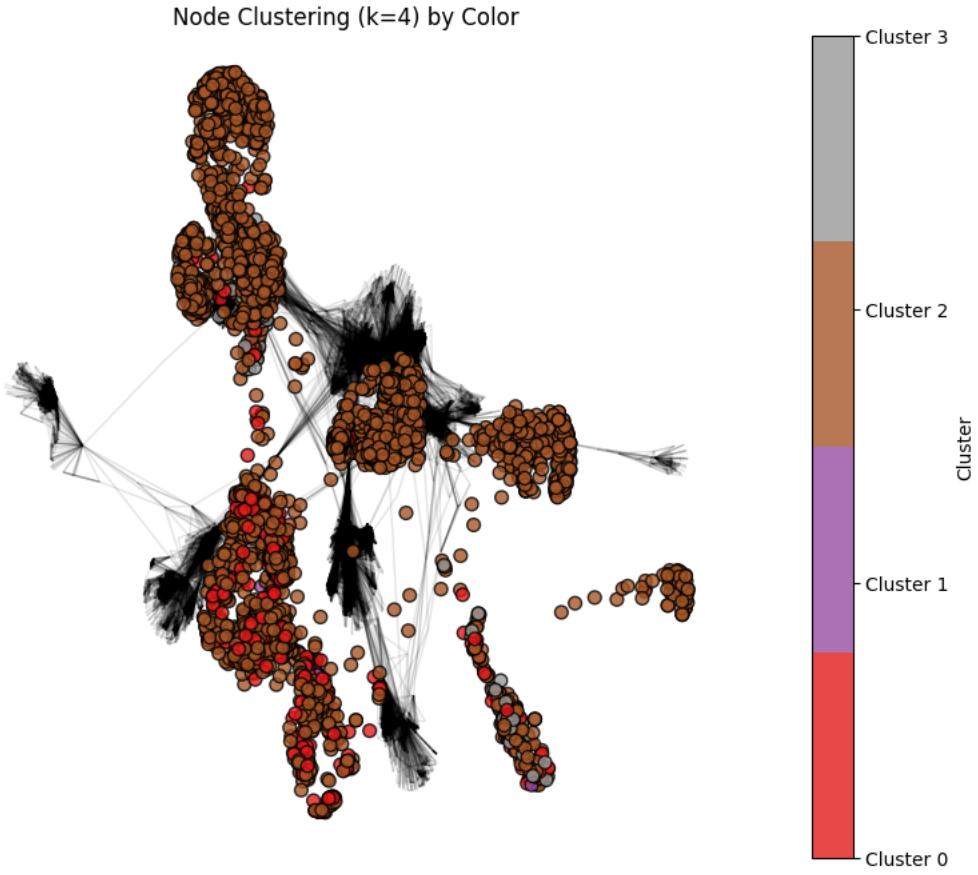
```
# Step 4: Visualize the clusters for agglometric clustering
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot for k=4 clusters
scatter_4 = axes[0].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['agg4Cluster'], cmap='viridis')
axes[0].set_title(f'Agglometric Clustering (k=4)')
axes[0].set_xlabel('NodeDegree')
axes[0].set_ylabel('Common Connections')
axes[0].legend(*scatter_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters
scatter_6 = axes[1].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['agg6Cluster'], cmap='viridis')
axes[1].set_title(f'Agglometric Clustering Clustering (k=6)')
axes[1].set_xlabel('NodeDegree')
axes[1].set_ylabel('Common Connections')
axes[1].legend(*scatter_6.legend_elements(), title='Clusters')
```

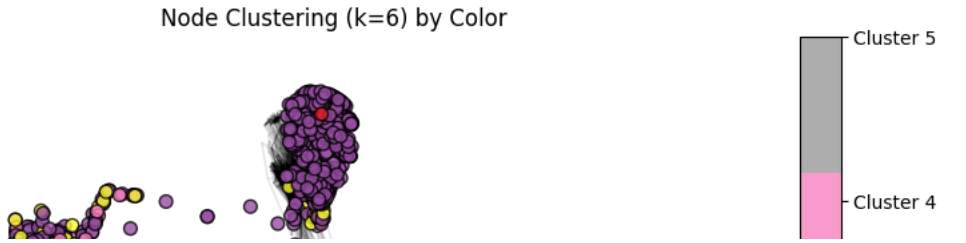
```
<matplotlib.legend.Legend at 0x7a198e411f00>
```



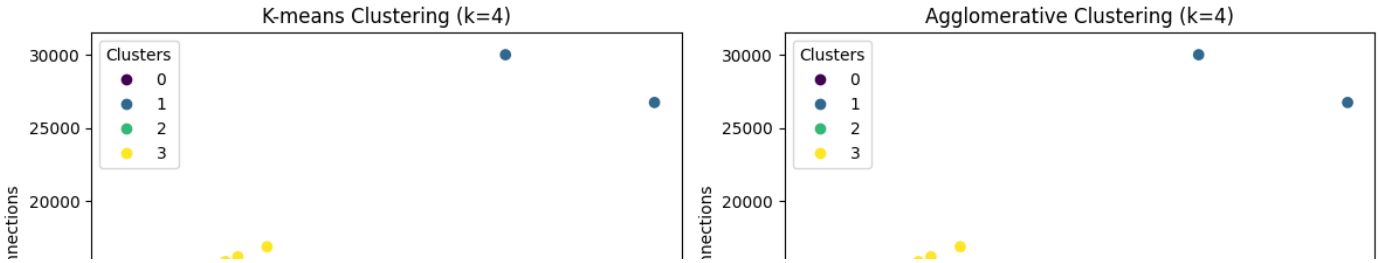

```
# Plot for k=4
plot_clusters(G, onefinaldf['agg4Cluster'], 4)

# Plot for k=6
plot_clusters(G, onefinaldf['agg6Cluster'], 6)
```

```
<ipython-input-113-98016605a855>:10: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will
  nodes = nx.draw_networkx_nodes(graph, pos=nx.spring_layout(graph), node_color=clusters, cmap=plt.cm.get_cmap('Set1', len(clusters
```



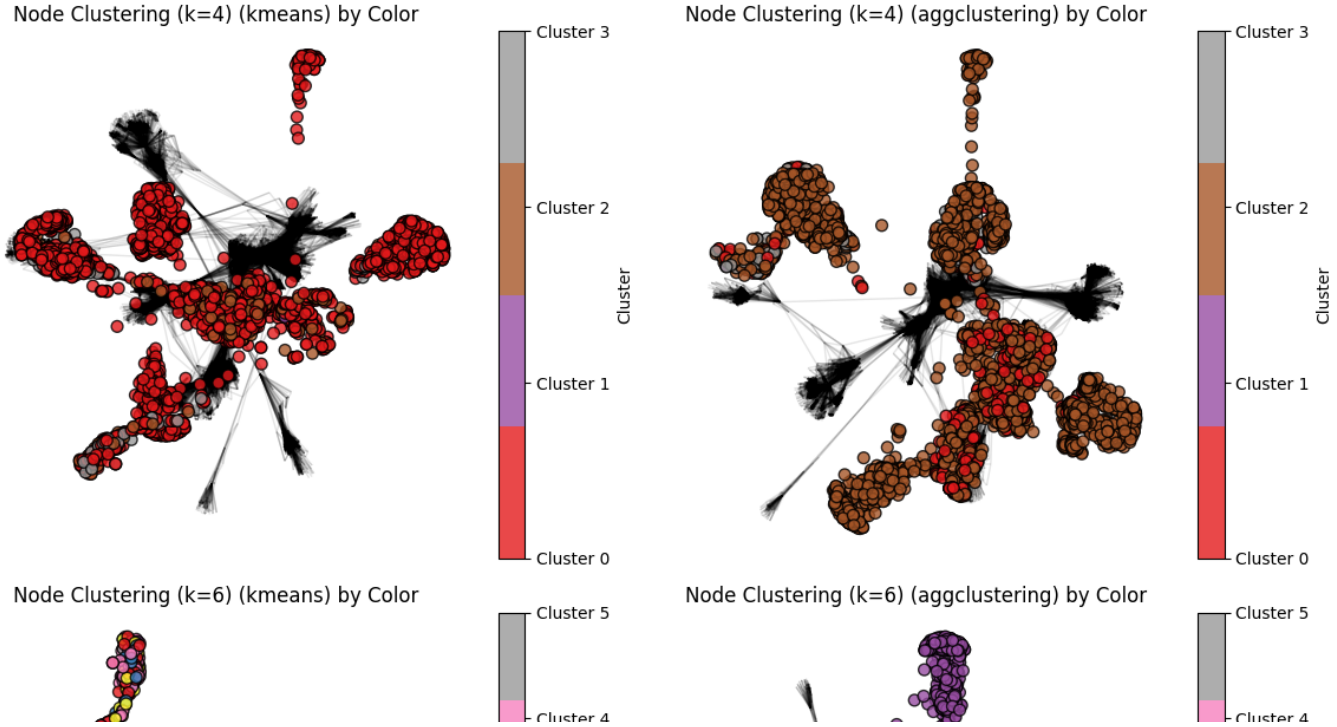


```
# Step 4: Visualize the clusters for k-means and agglomerative clustering
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))

# Plot for k=4 clusters in k-means
scatter_4_kmeans = axes[0, 0].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k=4_Cluster'], cmap='viridis
axes[0, 0].set_title('K-means Clustering (k=4)')
axes[0, 0].set_xlabel('NodeDegree')
axes[0, 0].set_ylabel('Common Connections')
axes[0, 0].legend(*scatter_4_kmeans.legend_elements(), title='Clusters')

# Plot for k=4 clusters in agglomerative clustering
scatter_4_agg = axes[0, 1].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['agg4Cluster'], cmap='viridis')
axes[0, 1].set_title('Agglomerative Clustering (k=4)')
axes[0, 1].set_xlabel('NodeDegree')
axes[0, 1].set_ylabel('Common Connections')
axes[0, 1].legend(*scatter_4_agg.legend_elements(), title='Clusters')

# Plot for k=6 clusters in k-means
scatter_6_kmeans = axes[1, 0].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k = 6_Cluster'], cmap='virid
axes[1, 0].set_title('K-means Clustering (k=6)')
axes[1, 0].set_xlabel('NodeDegree')
axes[1, 0].set_ylabel('Common Connections')
axes[1, 0].legend(*scatter_6_kmeans.legend_elements(), title='Clusters')

# Plot for k=6 clusters in agglomerative clustering
scatter_6_agg = axes[1, 1].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['agg6Cluster'], cmap='viridis')
axes[1, 1].set_title('Agglomerative Clustering (k=6)')
axes[1, 1].set_xlabel('NodeDegree')
axes[1, 1].set_ylabel('Common Connections')
axes[1, 1].legend(*scatter_6_agg.legend_elements(), title='Clusters')

plt.tight_layout()
plt.show()
```

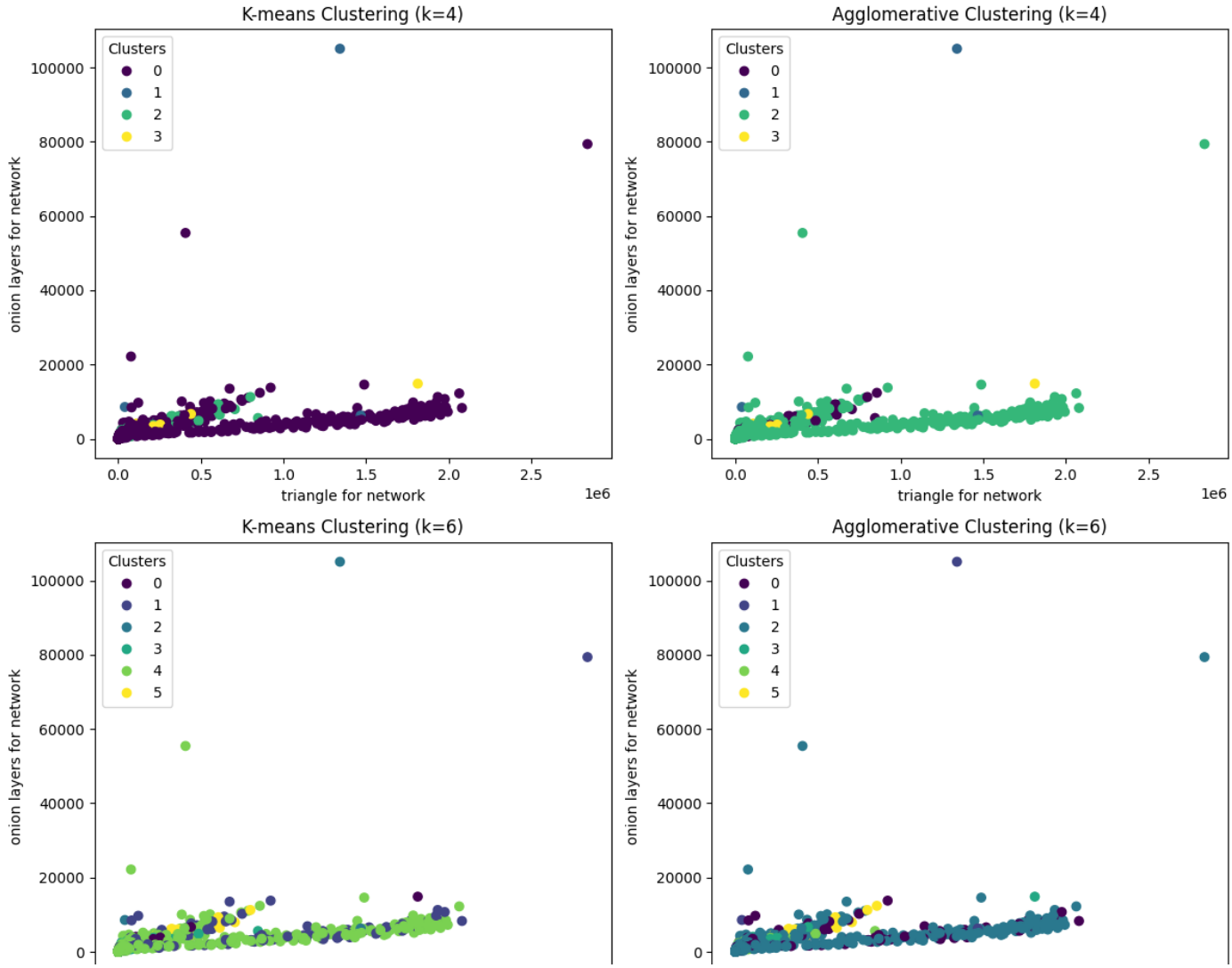



```
import seaborn as sns

def plot_clustersname(graph, clusters, k_value, name, ax):
    # Choose a colormap for clusters
    cluster_colors = sns.color_palette('Set1', n_colors=len(clusters.unique()))

    # Draw nodes with colors based on clusters
    nodes = nx.draw_networkx_nodes(graph, pos=nx.spring_layout(graph), node_color=clusters, cmap=plt.cm.get_cmap('Set1', len(clusters.unique

    # Draw edges
    nx.draw_networkx_edges(graph, pos=nx.spring_layout(graph), alpha=0.1, ax=ax)

    # Add a colorbar
    cbar = plt.colorbar(nodes, ticks=clusters.unique(), label='Cluster', ax=ax)
    cbar.set_ticklabels([f'Cluster {i}' for i in clusters.unique()])

    ax.set_title(f'Node Clustering (k={k_value}) ({name}) by Color')
    ax.axis('off')

# Assuming G is your graph

# Create a 2x2 grid for subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Plot for k=4
plot_clustersname(G, onefinaldf['k=4_Cluster'], 4, 'kmeans', axes[0, 0])

# Plot for k=4 - agg
plot_clustersname(G, onefinaldf['agg4Cluster'], 4, 'aggclustering', axes[0, 1])

# Plot for k=6
plot_clustersname(G, onefinaldf['k = 6_Cluster'], 6, 'kmeans', axes[1, 0])

# Plot for k=6 - agg
plot_clustersname(G, onefinaldf['agg6Cluster'], 6, 'aggclustering', axes[1, 1])

# Adjust layout
plt.tight_layout()
plt.show()
```

```
<ipython-input-117-83f07bba3dce>:8: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be rem
  nodes = nx.draw_networkx_nodes(graph, pos=nx.spring_layout(graph), node_color=clusters, cmap=plt.cm.get_cmap('Set1', len(clusters.uniq
```




```
# visualize with No of triangle in the network and Onion layers of the network for both of the Clustering
# Step 4: Visualize the clusters for k-means and agglomerative clustering
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))

# Plot for k=4 clusters in k-means
scatter_4_kmeans = axes[0, 0].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['k=4_Cluster'
axes[0, 0].set_title('K-means Clustering (k=4)')
axes[0, 0].set_xlabel('triangle for network')
axes[0, 0].set_ylabel('onion layers for network')
axes[0, 0].legend(*scatter_4_kmeans.legend_elements(), title='Clusters')

# Plot for k=4 clusters in agglomerative clustering
scatter_4_agg = axes[0, 1].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['agg4Cluster'],
axes[0, 1].set_title('Agglomerative Clustering (k=4)')
axes[0, 1].set_xlabel('triangle for network')
axes[0, 1].set_ylabel('onion layers for network')
axes[0, 1].legend(*scatter_4_agg.legend_elements(), title='Clusters')

# Plot for k=6 clusters in k-means
scatter_6_kmeans = axes[1, 0].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['k = 6_Cluste
axes[1, 0].set_title('K-means Clustering (k=6)')
axes[1, 0].set_xlabel('triangle for network')
axes[1, 0].set_ylabel('onion layers for network')
axes[1, 0].legend(*scatter_6_kmeans.legend_elements(), title='Clusters')

# Plot for k=6 clusters in agglomerative clustering
scatter_6_agg = axes[1, 1].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['agg6Cluster'],
axes[1, 1].set_title('Agglomerative Clustering (k=6)')
axes[1, 1].set_xlabel('triangle for network')
axes[1, 1].set_ylabel('onion layers for network')
axes[1, 1].legend(*scatter_6_agg.legend_elements(), title='Clusters')

plt.tight_layout()
plt.show()
```




```
#dbscan algorithm
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Assuming df is your DataFrame

# Step 1: Standardize the data (required for DBSCAN)
scaler = StandardScaler()
data_standardized = scaler.fit_transform(onefinaldf)

# Step 2: Apply DBSCAN
# Choose appropriate values for `eps` (maximum distance between two samples for one to be considered as in the neighborhood of the other)
# and `min_samples` (the number of samples in a neighborhood for a point to be considered as a core point)
dbscan = DBSCAN(eps=1, min_samples=20)
onefinaldf['dbs1Cluster'] = dbscan.fit_predict(data_standardized)

# Display the DataFrame with cluster assignments
print(onefinaldf)

# Step 3: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
plt.scatter(onefinaldf['NodeDegree'], onefinaldf['Common Connections'], c=onefinaldf['dbs1Cluster'], cmap='viridis')
plt.title('DBSCAN Clustering')
plt.show()
```

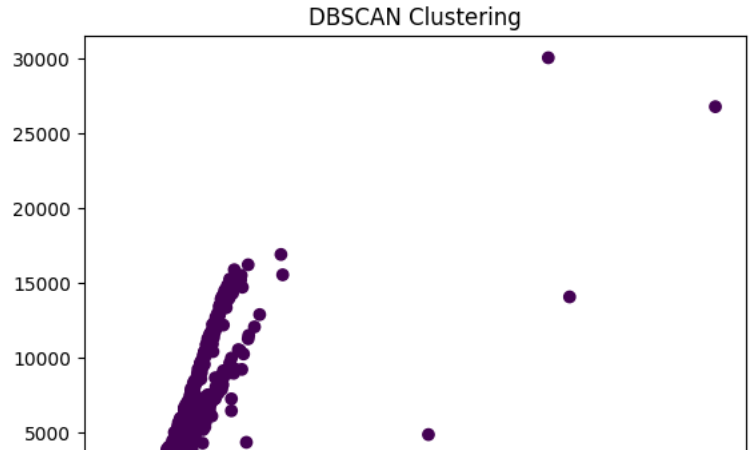
	Node	NodeDegree	0_count	1_count	2_count	3_count	4_count	5_count	\
0	0	347	0	14	27	22	17	12	
1	1	17	0	0	0	0	0	0	
2	2	10	0	0	0	0	0	0	
3	3	17	0	0	0	1	0	1	
4	4	10	0	0	0	0	0	1	
...	...	...	...	...	...	...	...	...	
4034	4034	2	0	0	0	0	0	0	
4035	4035	1	0	0	0	0	0	0	
4036	4036	2	0	0	0	0	1	0	
4037	4037	4	0	0	0	0	0	0	
4038	4038	9	0	0	0	0	0	0	

	6_count	7_count	...	max_weight_matching	random_probability	\
0	15	18	...	164	75.626117	
1	0	1	...	9	0.814521	
2	0	0	...	5	0.088836	
3	0	0	...	9	1.395870	
4	0	0	...	5	0.364554	
...	...	...	...	...	...	
4034	0	0	...	1	0.000000	
4035	0	0	...	1	0.000000	
4036	0	0	...	1	0.000000	
4037	0	0	...	2	-0.135403	
4038	2	1	...	5	0.266415	

	eccentricity	No of triangles with Node	triangle for network	\
0	6	2519	39777	
1	7	57	402	
2	7	40	381	
3	7	86	996	
4	7	39	378	
...	...	...	...	
4034	8	1	3	
4035	8	0	0	
4036	8	1	3	
4037	8	4	15	
4038	8	20	114	

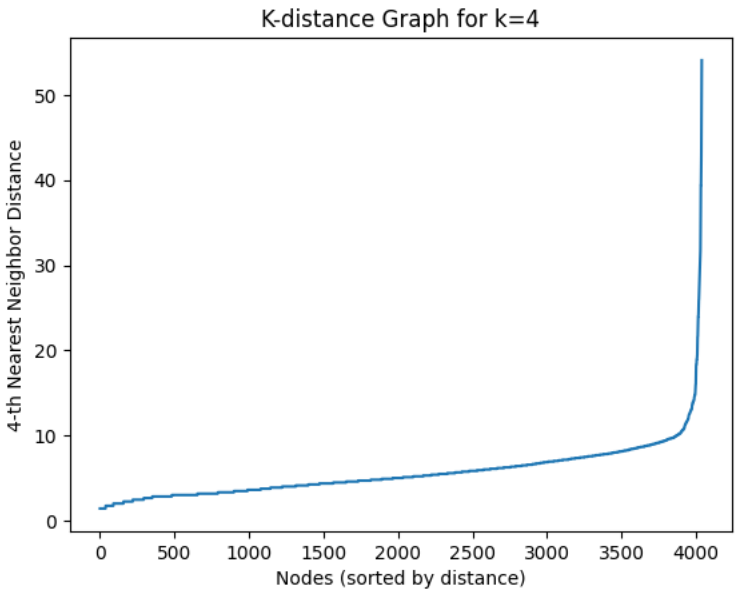
	k=4_Cluster	k = 6_Cluster	agg4Cluster	agg6Cluster	dbs1Cluster
0	1	2	1	1	-1
1	0	4	2	2	-1
2	0	4	2	2	-1
3	0	4	2	2	-1
4	0	4	2	2	-1
...	...	...	...	...	...
4034	0	4	2	2	-1
4035	0	4	2	2	-1
4036	0	4	2	2	-1
4037	0	4	2	2	-1
4038	0	4	2	2	-1

[4039 rows x 262 columns]



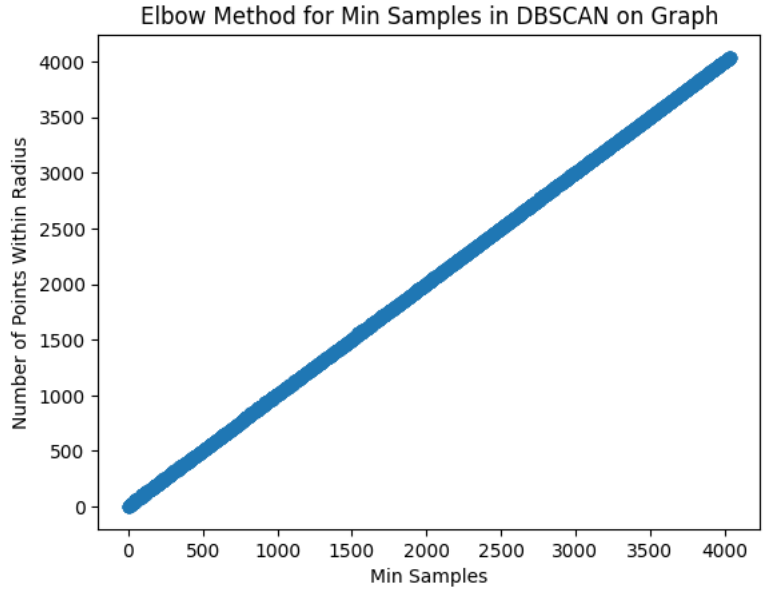
we have to change eps and min samples value for the dbscan

```
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt

# Choose a suitable value for k (number of neighbors)
k = 4

 #Compute pairwise distances between nodes
distances = nx.floyd_warshall_numpy(G)

# Choose a suitable value for k
k = 4

# Fit a nearest neighbors model
neigh = NearestNeighbors(n_neighbors=k)
neigh.fit(distances)

# Calculate distances to k-th nearest neighbors
distances_knn, _ = neigh.kneighbors(distances)


import numpy as np
import matplotlib.pyplot as plt

# Take the distances to the k-th nearest neighbors
distances_knn = np.max(distances_knn, axis=1)
distances_knn.sort()

# Plot the k-distance graph
plt.plot(distances_knn)
plt.xlabel('Nodes (sorted by distance)')
plt.ylabel(f'{k}-th Nearest Neighbor Distance')
plt.title(f'K-distance Graph for k={k}')
plt.show()
```



so it is increasing sharply near 11 , so i will take eps value as 11

```
# Compute pairwise distances between nodes (you may use other metrics based on your problem)
distances1 = nx.floyd_warshall_numpy(G)

# Choose a suitable range of values for min_samples
min_samples_values = range(1, 4040)  # Adjust the range as needed

# Fit a nearest neighbors model
neigh = NearestNeighbors(n_neighbors=max(min_samples_values))
neigh.fit(distances1)

# Calculate distances to k-th nearest neighbors
distances_knn1, _ = neigh.kneighbors(distances)

# Sort the distances and calculate the number of points within each distance
distances_knn1 = np.max(distances_knn1, axis=1)
distances_knn1.sort()

num_points_within_radius = [np.sum(distances_knn1 <= d) for d in distances_knn1]

# Plot the elbow graph
plt.plot(min_samples_values, num_points_within_radius, marker='o')
plt.xlabel('Min Samples')
plt.ylabel('Number of Points Within Radius')
plt.title('Elbow Method for Min Samples in DBSCAN on Graph')
plt.show()
```




```
dbminvaluedf = onefinaldf.copy()
print(dbminvaluedf)
```

```
      Node  NodeDegree  0_count  1_count  2_count  3_count  4_count  5_count  \
0        0         347        0       14       27       22       17       12
1        1          17        0        0        0        0        0        0
2        2          10        0        0        0        0        0        0
3        3          17        0        0        0        1        0        1
4        4          10        0        0        0        0        0        1
...    ...         ...      ...      ...      ...      ...      ...      ...
4034  4034           2        0        0        0        0        0        0
4035  4035           1        0        0        0        0        0        0
4036  4036           2        0        0        0        0        1        0
4037  4037           4        0        0        0        0        0        0
4038  4038           9        0        0        0        0        0        0

      6_count  7_count  ...  max_weight_matching  random_probability  \
0          15       18  ...                  164           75.626117
1           0        1  ...                    9            0.814521
2           0        0  ...                    5            0.088836
3           0        0  ...                    9            1.395870
4           0        0  ...                    5            0.364554
...       ...      ...  ...                  ...                 ...
4034        0        0  ...                    1            0.000000
4035        0        0  ...                    1            0.000000
4036        0        0  ...                    1            0.000000
4037        0        0  ...                    2           -0.135403
4038        2        1  ...                    5            0.266415
```

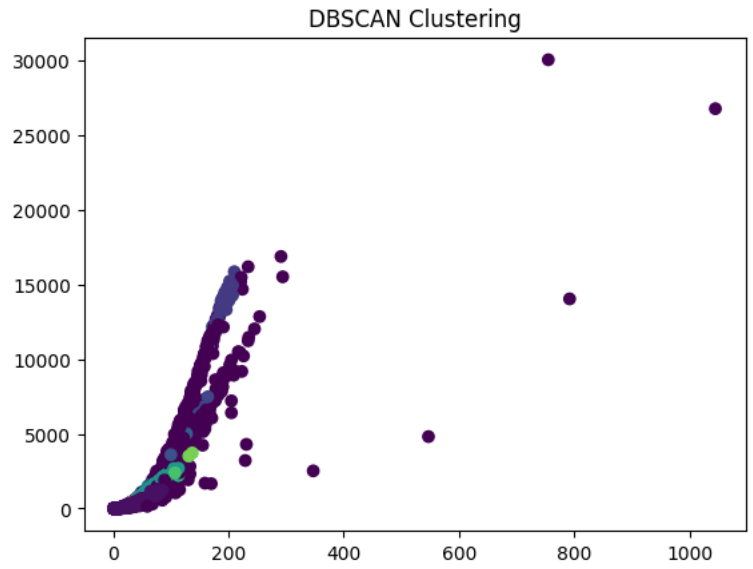
```
      eccentricity  No of triangles with Node  triangle for network  \
0                6                       2519                 39777
1                7                         57                   402
2                7                         40                   381
3                7                         86                   996
4                7                         39                   378
...            ...                        ...                   ...
4034             8                          1                     3
4035             8                          0                     0
4036             8                          1                     3
4037             8                          4                    15
4038             8                         20                   114

      k=4_Cluster  k = 6_Cluster  agg4Cluster  agg6Cluster  dbs1Cluster
0               1              2            1            1           -1
1               0              4            2            2           -1
2               0              4            2            2           -1
3               0              4            2            2           -1
4               0              4            2            2           -1
...           ...            ...          ...          ...          ...
4034            0              4            2            2           -1
4035            0              4            2            2           -1
4036            0              4            2            2           -1
4037            0              4            2            2           -1
4038            0              4            2            2           -1

[4039 rows x 262 columns]
```

```
min_samples_values = range(2, 4040)  # Starting from 2, as silhouette_score requires at least 2 clusters

# List to store silhouette scores for different min_samples values
silhouette_scores = []

# Loop over min_samples values
for min_samples in min_samples_values:
    # Apply DBSCAN for each min_samples value
    labels = DBSCAN(eps=11, min_samples=min_samples).fit_predict(dbminvaluedf)

    # Check if there is more than one unique label
    unique_labels = np.unique(labels)
    if len(unique_labels) > 1:
        # Calculate silhouette score
        silhouette_scores.append(silhouette_score(dbminvaluedf, labels))
    else:
        # If only one label, append a placeholder value (e.g., -1)
        silhouette_scores.append(-1)

# Plot silhouette scores
plt.plot(min_samples_values, silhouette_scores, marker='o')
plt.xlabel('Min Samples')
plt.ylabel('Silhouette Score')
plt.title('Silhouette Score for Different Min Samples in DBSCAN')
plt.show()
```

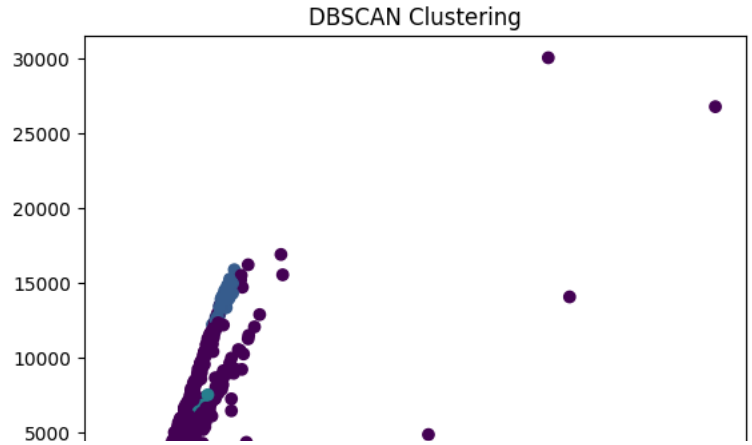
```
optimal_min_samples = min_samples_values[np.argmax(silhouette_scores)]
print(f'Optimal Min Samples: {optimal_min_samples}')
```

```
Optimal Min Samples: 2
```

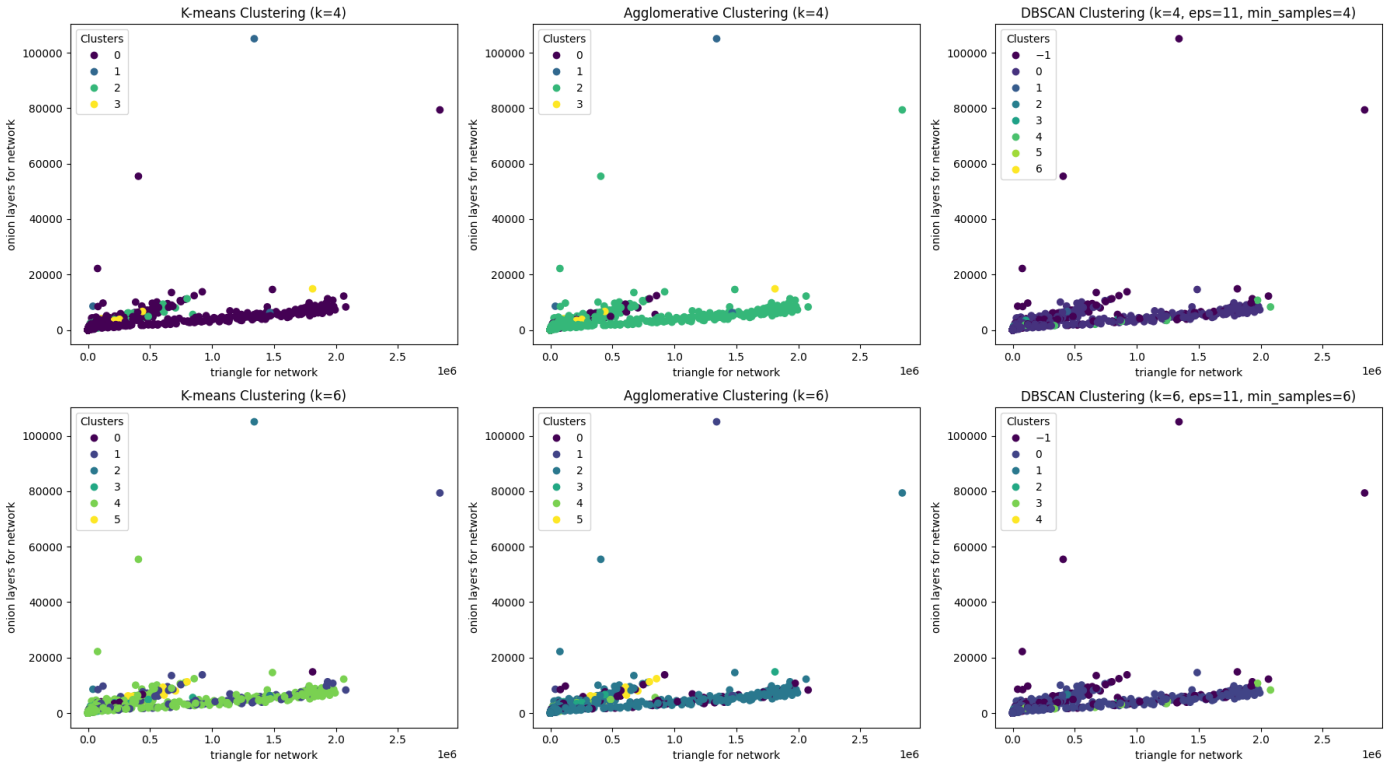
```
#dbscan algorithm
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Assuming df is your DataFrame

# Step 1: Standardize the data (required for DBSCAN)
scaler = StandardScaler()
data_standardized = scaler.fit_transform(onefinaldf)

# Step 2: Apply DBSCAN
# Choose appropriate values for `eps` (maximum distance between two samples for one to be considered as in the neighborhood of the other)
# and `min_samples` (the number of samples in a neighborhood for a point to be considered as a core point)
dbscan = DBSCAN(eps=11, min_samples=2)
onefinaldf['dbsCluster'] = dbscan.fit_predict(data_standardized)


# Step 3: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
plt.scatter(onefinaldf['NodeDegree'], onefinaldf['Common Connections'], c=onefinaldf['dbsCluster'], cmap='viridis')
plt.title('DBSCAN Clustering')
plt.show()
```



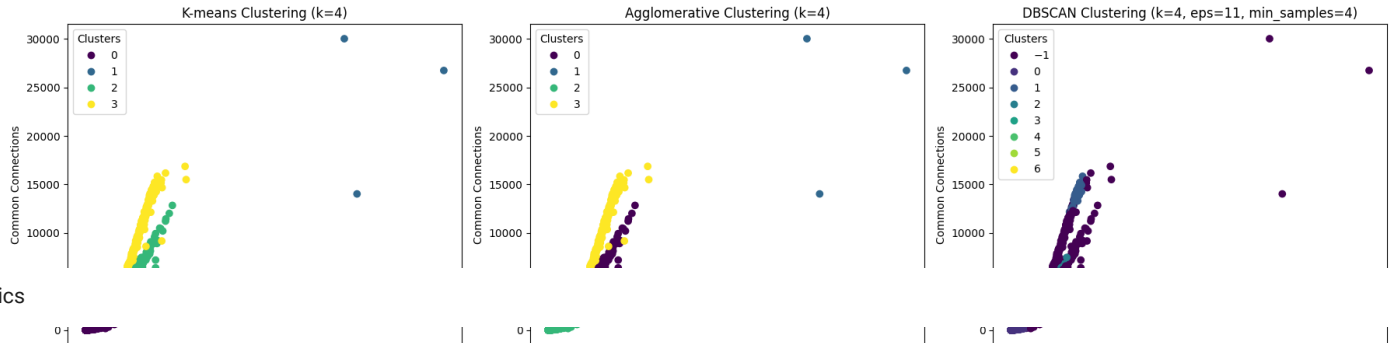

```
# Step 2: Apply DBSCAN
# Choose appropriate values for `eps` (maximum distance between two samples for one to be considered as in the neighborhood of the other)
# and `min_samples` (the number of samples in a neighborhood for a point to be considered as a core point)
dbscan = DBSCAN(eps=11, min_samples=4)
onefinaldf['dbs4Cluster'] = dbscan.fit_predict(data_standardized)


# Step 3: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
plt.scatter(onefinaldf['NodeDegree'], onefinaldf['Common Connections'], c=onefinaldf['dbs4Cluster'], cmap='viridis')
plt.title('DBSCAN Clustering')
plt.show()
```




```
# Step 2: Apply DBSCAN
# Choose appropriate values for `eps` (maximum distance between two samples for one to be considered as in the neighborhood of the other)
# and `min_samples` (the number of samples in a neighborhood for a point to be considered as a core point)
dbscan = DBSCAN(eps=11, min_samples=6)
onefinaldf['dbs6Cluster'] = dbscan.fit_predict(data_standardized)



# Visualize clusters for k-means, agglomerative clustering, and DBSCAN in a 2x3 grid
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(18, 10))

# Plot for k=4 clusters in k-means
scatter_4_kmeans = axes[0, 0].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['k=4_Cluster'
axes[0, 0].set_title('K-means Clustering (k=4)')
axes[0, 0].set_xlabel('triangle for network')
axes[0, 0].set_ylabel('onion layers for network')
axes[0, 0].legend(*scatter_4_kmeans.legend_elements(), title='Clusters')

# Plot for k=4 clusters in agglomerative clustering
scatter_4_agg = axes[0, 1].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['agg4Cluster'],
axes[0, 1].set_title('Agglomerative Clustering (k=4)')
axes[0, 1].set_xlabel('triangle for network')
axes[0, 1].set_ylabel('onion layers for network')
axes[0, 1].legend(*scatter_4_agg.legend_elements(), title='Clusters')

# Plot for k=6 clusters in k-means
scatter_6_kmeans = axes[1, 0].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['k = 6_Cluste
axes[1, 0].set_title('K-means Clustering (k=6)')
axes[1, 0].set_xlabel('triangle for network')
axes[1, 0].set_ylabel('onion layers for network')
axes[1, 0].legend(*scatter_6_kmeans.legend_elements(), title='Clusters')

# Plot for k=6 clusters in agglomerative clustering
scatter_6_agg = axes[1, 1].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['agg6Cluster'],
axes[1, 1].set_title('Agglomerative Clustering (k=6)')
axes[1, 1].set_xlabel('triangle for network')
axes[1, 1].set_ylabel('onion layers for network')
axes[1, 1].legend(*scatter_6_agg.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=4, eps=11, min_samples=5)
scatter_dbscan_4 = axes[0, 2].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['dbs4Cluster'
axes[0, 2].set_title('DBSCAN Clustering (k=4, eps=11, min_samples=4)')
axes[0, 2].set_xlabel('triangle for network')
axes[0, 2].set_ylabel('onion layers for network')
axes[0, 2].legend(*scatter_dbscan_4.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=6, eps=0.8, min_samples=5)
scatter_dbscan_6 = axes[1, 2].scatter(onefinaldf["triangle for network"], onefinaldf['onion layers for network'], c=onefinaldf['dbs6Cluster'
axes[1, 2].set_title('DBSCAN Clustering (k=6, eps=11, min_samples=6)')
axes[1, 2].set_xlabel('triangle for network')
axes[1, 2].set_ylabel('onion layers for network')
axes[1, 2].legend(*scatter_dbscan_6.legend_elements(), title='Clusters')

plt.tight_layout()
plt.show()
```

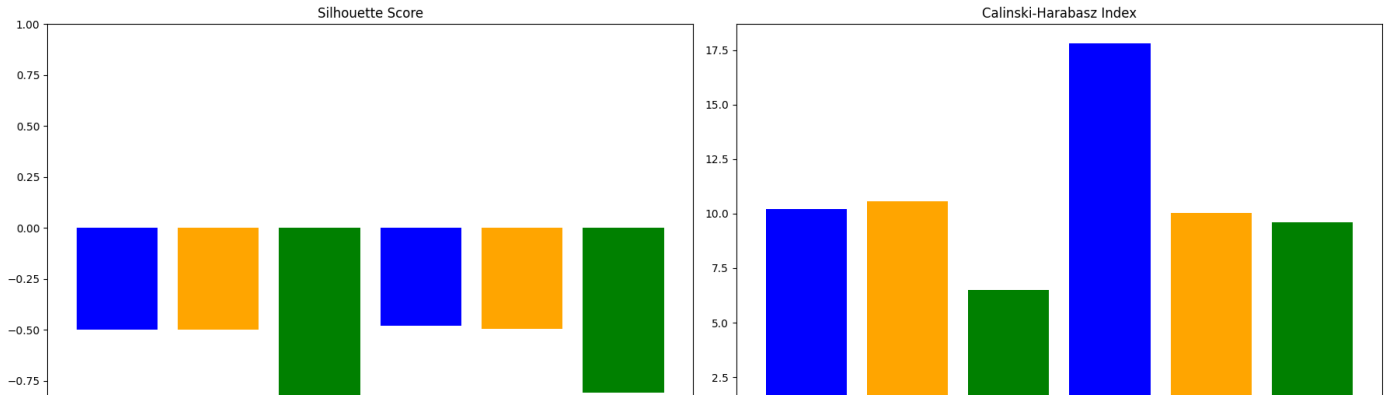

K-means Clustering (k=4)
Agglomerative Clustering (k=4)
DBSCAN Clustering (k=4, eps=11, min_samples=4)
K-means Clustering (k=6)
Agglomerative Clustering (k=6)
DBSCAN Clustering (k=6, eps=11, min_samples=6)
Clusters
onion layers for network
triangle for network
1e6

```
# Step 4: Visualize the clusters for k-means and agglomerative clustering
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(18, 10))

# Plot for k=4 clusters in k-means
scatter_4_kmeans = axes[0, 0].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k=4_Cluster'], cmap='viridis
axes[0, 0].set_title('K-means Clustering (k=4)')
axes[0, 0].set_xlabel('NodeDegree')
axes[0, 0].set_ylabel('Common Connections')
axes[0, 0].legend(*scatter_4_kmeans.legend_elements(), title='Clusters')

# Plot for k=4 clusters in agglomerative clustering
scatter_4_agg = axes[0, 1].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['agg4Cluster'], cmap='viridis')
axes[0, 1].set_title('Agglomerative Clustering (k=4)')
axes[0, 1].set_xlabel('NodeDegree')
axes[0, 1].set_ylabel('Common Connections')
axes[0, 1].legend(*scatter_4_agg.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=4, eps=11, min_samples=5)
scatter_dbscan_4 = axes[0, 2].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['dbs4Cluster'], cmap='viridis
axes[0, 2].set_title('DBSCAN Clustering (k=4, eps=11, min_samples=4)')
axes[0, 2].set_xlabel('NodeDegree')
axes[0, 2].set_ylabel('Common Connections')
axes[0, 2].legend(*scatter_dbscan_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters in k-means
scatter_6_kmeans = axes[1, 0].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['k = 6_Cluster'], cmap='virid
axes[1, 0].set_title('K-means Clustering (k=6)')
axes[1, 0].set_xlabel('NodeDegree')
axes[1, 0].set_ylabel('Common Connections')
axes[1, 0].legend(*scatter_6_kmeans.legend_elements(), title='Clusters')

# Plot for k=6 clusters in agglomerative clustering
scatter_6_agg = axes[1, 1].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['agg6Cluster'], cmap='viridis')
axes[1, 1].set_title('Agglomerative Clustering (k=6)')
axes[1, 1].set_xlabel('NodeDegree')
axes[1, 1].set_ylabel('Common Connections')
axes[1, 1].legend(*scatter_6_agg.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=6, eps=0.8, min_samples=5)
scatter_dbscan_6 = axes[1, 2].scatter(onefinaldf["NodeDegree"], onefinaldf['Common Connections'], c=onefinaldf['dbs6Cluster'], cmap='viridis
axes[1, 2].set_title('DBSCAN Clustering (k=6, eps=11, min_samples=6)')
axes[1, 2].set_xlabel('NodeDegree')
axes[1, 2].set_ylabel('Common Connections')
axes[1, 2].legend(*scatter_dbscan_6.legend_elements(), title='Clusters')

plt.tight_layout()
plt.show()
```



Metrics

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import silhouette_score, calinski_harabasz_score

# Extract features
X = onefinaldf[['triangle for network', 'onion layers for network']]

# Extract cluster labels
kmeans_labels_4 = onefinaldf['k=4_Cluster']
agg_labels_4 = onefinaldf['agg4Cluster']
dbscan_labels_4 = onefinaldf['dbs4Cluster']

kmeans_labels_6 = onefinaldf['k = 6_Cluster']
agg_labels_6 = onefinaldf['agg6Cluster']
dbscan_labels_6 = onefinaldf['dbs6Cluster']

# Calculate Silhouette Score
kmeans_silhouette_4 = silhouette_score(X, kmeans_labels_4)
agg_silhouette_4 = silhouette_score(X, agg_labels_4)
dbscan_silhouette_4 = silhouette_score(X, dbscan_labels_4)

kmeans_silhouette_6 = silhouette_score(X, kmeans_labels_6)
agg_silhouette_6 = silhouette_score(X, agg_labels_6)
dbscan_silhouette_6 = silhouette_score(X, dbscan_labels_6)

# Calculate Calinski-Harabasz Index
kmeans_calinski_harabasz_4 = calinski_harabasz_score(X, kmeans_labels_4)
agg_calinski_harabasz_4 = calinski_harabasz_score(X, agg_labels_4)
dbscan_calinski_harabasz_4 = calinski_harabasz_score(X, dbscan_labels_4)

kmeans_calinski_harabasz_6 = calinski_harabasz_score(X, kmeans_labels_6)
agg_calinski_harabasz_6 = calinski_harabasz_score(X, agg_labels_6)
dbscan_calinski_harabasz_6 = calinski_harabasz_score(X, dbscan_labels_6)

# Data for plotting
algorithms = ['K-means (k=4)', 'Agglomerative (k=4)', 'DBSCAN (k=4)', 'K-means (k=6)', 'Agglomerative (k=6)', 'DBSCAN (k=6)']
silhouette_scores = [kmeans_silhouette_4, agg_silhouette_4, dbscan_silhouette_4, kmeans_silhouette_6, agg_silhouette_6, dbscan_silhouette_6]
calinski_harabasz_scores = [kmeans_calinski_harabasz_4, agg_calinski_harabasz_4, dbscan_calinski_harabasz_4, kmeans_calinski_harabasz_6, agg

# Plot Silhouette Score and Calinski-Harabasz Index
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(18, 6))

# Plot Silhouette Score
axes[0].bar(algorithms, silhouette_scores, color=['blue', 'orange', 'green', 'blue', 'orange', 'green'])
axes[0].set_title('Silhouette Score')
axes[0].set_ylim(-1, 1)

# Plot Calinski-Harabasz Index
axes[1].bar(algorithms, calinski_harabasz_scores, color=['blue', 'orange', 'green', 'blue', 'orange', 'green'])
axes[1].set_title('Calinski-Harabasz Index')

plt.tight_layout()
plt.show()
```




```
#Clustering via networkx Library
Clus = nx.clustering(G)

Clust = [round(value, 2) for value in Clus.values()]

# Create a DataFrame with a single column
cludf = pd.DataFrame({'library Clustering': Clust})

# Display the DataFrame
print(cludf)
```

```
      library Clustering
0                   0.04
1                   0.42
2                   0.89
3                   0.63
4                   0.87
...                  ...
4034                1.00
4035                0.00
4036                1.00
4037                0.67
4038                0.56

[4039 rows x 1 columns]
```

```
onefinaldf = pd.concat([onefinaldf, cludf], axis = 1)
print(onefinaldf)
```

```
      Node  NodeDegree  0_count  1_count  2_count  3_count  4_count  5_count  \
0        0         347        0       14       27       22       17       12
1        1          17        0        0        0        0        0        0
2        2          10        0        0        0        0        0        0
3        3          17        0        0        0        1        0        1
4        4          10        0        0        0        0        0        1
...    ...         ...      ...      ...      ...      ...      ...      ...
4034  4034           2        0        0        0        0        0        0
4035  4035           1        0        0        0        0        0        0
4036  4036           2        0        0        0        0        1        0
4037  4037           4        0        0        0        0        0        0
4038  4038           9        0        0        0        0        0        0

      6_count  7_count  ...  triangle for network  k=4_Cluster  k = 6_Cluster  \
0          15       18  ...                 39777            1              2
1           0        1  ...                   402            0              4
2           0        0  ...                   381            0              4
3           0        0  ...                   996            0              4
4           0        0  ...                   378            0              4
...       ...      ...  ...                   ...          ...            ...
4034        0        0  ...                     3            0              4
4035        0        0  ...                     0            0              4
4036        0        0  ...                     3            0              4
4037        0        0  ...                    15            0              4
4038        2        1  ...                   114            0              4

      agg4Cluster  agg6Cluster  dbs1Cluster  dbsCluster  dbs4Cluster  \
0               1            1           -1          -1           -1
1               2            2           -1           0            0
2               2            2           -1           0            0
3               2            2           -1           0            0
4               2            2           -1           0            0
...           ...          ...          ...         ...          ...
4034            2            2           -1           0            0
```

```
4035            2            2          -1          0           0
4036            2            2          -1          0           0
4037            2            2          -1          0           0
4038            2            2          -1          0           0

      dbs6Cluster  library Clustering
0              -1                0.04
1               0                0.42
2               0                0.89
3               0                0.63
4               0                0.87
...           ...                 ...
4034            0                1.00
4035            0                0.00
4036            0                1.00
4037            0                0.67
4038            0                0.56

[4039 rows x 266 columns]
```

```
#comparing scores with networkx library clustering
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import homogeneity_score, completeness_score, v_measure_score

# Extract cluster labels
kmeans_labels_4 = onefinaldf['k=4_Cluster']
agg_labels_4 = onefinaldf['agg4Cluster']
dbscan_labels_4 = onefinaldf['dbs4Cluster']

kmeans_labels_6 = onefinaldf['k = 6_Cluster']
agg_labels_6 = onefinaldf['agg6Cluster']
dbscan_labels_6 = onefinaldf['dbs6Cluster']

# Calculate scores for k=4 clusters
homogeneity_kmeans_4 = homogeneity_score(onefinaldf['library Clustering'], kmeans_labels_4)
completeness_kmeans_4 = completeness_score(onefinaldf['library Clustering'], kmeans_labels_4)
v_measure_kmeans_4 = v_measure_score(onefinaldf['library Clustering'], kmeans_labels_4)

homogeneity_agg_4 = homogeneity_score(onefinaldf['library Clustering'], agg_labels_4)
completeness_agg_4 = completeness_score(onefinaldf['library Clustering'], agg_labels_4)
v_measure_agg_4 = v_measure_score(onefinaldf['library Clustering'], agg_labels_4)

homogeneity_dbscan_4 = homogeneity_score(onefinaldf['library Clustering'], dbscan_labels_4)
completeness_dbscan_4 = completeness_score(onefinaldf['library Clustering'], dbscan_labels_4)
v_measure_dbscan_4 = v_measure_score(onefinaldf['library Clustering'], dbscan_labels_4)

# Calculate scores for k=6 clusters
homogeneity_kmeans_6 = homogeneity_score(onefinaldf['library Clustering'], kmeans_labels_6)
completeness_kmeans_6 = completeness_score(onefinaldf['library Clustering'], kmeans_labels_6)
v_measure_kmeans_6 = v_measure_score(onefinaldf['library Clustering'], kmeans_labels_6)

homogeneity_agg_6 = homogeneity_score(onefinaldf['library Clustering'], agg_labels_6)
completeness_agg_6 = completeness_score(onefinaldf['library Clustering'], agg_labels_6)
v_measure_agg_6 = v_measure_score(onefinaldf['library Clustering'], agg_labels_6)

homogeneity_dbscan_6 = homogeneity_score(onefinaldf['library Clustering'], dbscan_labels_6)
completeness_dbscan_6 = completeness_score(onefinaldf['library Clustering'], dbscan_labels_6)
v_measure_dbscan_6 = v_measure_score(onefinaldf['library Clustering'], dbscan_labels_6)

# Data for plotting
algorithms = ['K-means (k=4)', 'Agglomerative (k=4)', 'DBSCAN (k=4)', 'K-means (k=6)', 'Agglomerative (k=6)', 'DBSCAN (k=6)']
homogeneity_scores = [homogeneity_kmeans_4, homogeneity_agg_4, homogeneity_dbscan_4, homogeneity_kmeans_6, homogeneity_agg_6, homogeneity_db
completeness_scores = [completeness_kmeans_4, completeness_agg_4, completeness_dbscan_4, completeness_kmeans_6, completeness_agg_6, complete
v_measure_scores = [v_measure_kmeans_4, v_measure_agg_4, v_measure_dbscan_4, v_measure_kmeans_6, v_measure_agg_6, v_measure_dbscan_6]

# Plot scores
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))

# Plot Homogeneity Score
axes[0].bar(algorithms, homogeneity_scores, color=['blue', 'orange', 'green', 'blue', 'orange', 'green'])
axes[0].set_title('Homogeneity Score')

# Plot Completeness Score
axes[1].bar(algorithms, completeness_scores, color=['blue', 'orange', 'green', 'blue', 'orange', 'green'])
axes[1].set_title('Completeness Score')

# Plot V-Measure Score
axes[2].bar(algorithms, v_measure_scores, color=['blue', 'orange', 'green', 'blue', 'orange', 'green'])
axes[2].set_title('V-Measure Score')

plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/cluster/_supervised.py:64: UserWarning: Clustering metrics expects discrete valu
  warnings.warn(msg, UserWarning)
```


```
onefinaldf.to_csv('/content/drive/MyDrive/Colab Notebooks/onefinaldata.csv', index=False)
```

TWO HOP DATA

```
import pandas as pd
import networkx as nx
from itertools import combinations

twohopcommon_data = []

for node in range(500):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommond1f = pd.DataFrame(twohopcommon_data, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommond1f)
```

```
     Common Connections
0                 33343
1                  2849
2                  2856
3                  2849
4                  2856
..                  ...
495                3811
496                8264
497                8377
498                3780
499                3438

[500 rows x 1 columns]
```

```
twohopcommond1f.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp1.csv', index=False)



path1 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp1.csv'
twohopcommon1 = pd.read_csv(path1)
print(twohopcommon1)
```

```
     Common Connections
0                 33343
1                  2849
2                  2856
3                  2849
4                  2856
..                  ...
495                3811
496                8264
497                8377
498                3780
499                3438

[500 rows x 1 columns]
```

```
twohopcommon_data2 = []

for node in range(501, 1001):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data2.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommondf2 = pd.DataFrame(twohopcommon_data2, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommondf2)
```

```
     Common Connections
0                  3440
1                  4331
2                  7810
3                  4053
4                  3660
..                  ...
495               27916
496               27699
497               27786
498               27718
499               27779

[500 rows x 1 columns]
```

```
twohopcommondf2.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp2.csv', index=False)


path2 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp2.csv'
twohopcommon2 = pd.read_csv(path2)
print(twohopcommon2)
```

```
     Common Connections
0                  3440
1                  4331
2                  7810
3                  4053
4                  3660
..                  ...
495               27916
496               27699
497               27786
498               27718
499               27779

[500 rows x 1 columns]
```

```
import pandas as pd
import networkx as nx
from itertools import combinations

twohopcommon_data3 = []

for node in range(1001, 1501):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data3.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommondf3 = pd.DataFrame(twohopcommon_data3, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommondf3)
```

```
     Common Connections
0                 29826
1                 27791
2                 27700
3                 27675
4                 27781
..                  ...
495               27932
496               27766
497               27789
498               27783
499               27766

[500 rows x 1 columns]
```

```
twohopcommondf3.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp3.csv', index=False)



path3 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp3.csv'
twohopcommon3 = pd.read_csv(path3)
print(twohopcommon3)
```

```
     Common Connections
0                 29826
1                 27791
2                 27700
3                 27675
4                 27781
..                  ...
495               27932
496               27766
497               27789
498               27783
499               27766

[500 rows x 1 columns]
```

```
import pandas as pd
import networkx as nx
from itertools import combinations
twohopcommon_data4 = []

for node in range(1501, 2001):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data4.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommondf4 = pd.DataFrame(twohopcommon_data4, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommondf4)


twohopcommondf4.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp4.csv', index=False)



twohopcommondf4.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp4.csv', index=False)



path4 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp4.csv'
twohopcommon4 = pd.read_csv(path4)
print(twohopcommon4)
```

```
          Common Connections
     0                 27777
     1                 27725
     2                 27767
     3                 27737
     4                 42496
     ..                  ...
     495               30776
     496               30645
     497               30851
     498               30746
     499               30747

     [500 rows x 1 columns]
```

```
twohopcommon_data5 = []

for node in range(2001, 2501):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data5.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommondf5 = pd.DataFrame(twohopcommon_data5, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommondf5)

twohopcommondf5.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp5.csv', index=False)



twohopcommondf5.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp5.csv', index=False)



path5 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp5.csv'
twohopcommon5 = pd.read_csv(path5)
print(twohopcommon5)
```

```
     Common Connections
0                 30845
1                 30729
2                 31117
3                 30855
4                 30625
..                  ...
495               30796
496               30755
497               31239
498               30687
499               30599

[500 rows x 1 columns]
```

```
twohopcommon_data6 = []

for node in range(2501, 3001):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data6.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommondf6 = pd.DataFrame(twohopcommon_data6, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommondf6)



twohopcommondf6.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp6.csv', index=False)



path6 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp6.csv'
twohopcommon6 = pd.read_csv(path6)
print(twohopcommon6)
```

```
     Common Connections
0                 30742
1                 30897
2                 30764
3                 30629
4                 30775
..                  ...
495               14778
496               14803
497               14814
498               14826
499               14726

[500 rows x 1 columns]
```

```
twohopcommon_data7 = []

for node in range(3001, 3501):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data7.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommondf7 = pd.DataFrame(twohopcommon_data7, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommondf7)



twohopcommondf7.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp7.csv', index=False)



path7 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp7.csv'
twohopcommon7 = pd.read_csv(path7)
print(twohopcommon7)


twohopcommon_data8 = []

for node in range(3501, len(G)):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)


    # Use itertools.combinations to iterate over unique pairs of elements in acData
    for neighbor1, neighbor2 in combinations(acData, 2):
        # Check if the common neighbors have 1 unit distance with the original nodes

        N = nx.shortest_path_length(G, source=neighbor1, target=neighbor2)



        # Count only when the shortest distance is 1
        if N == 1:
            abData.append(1)

    twohopcommon_data8.append(sum(abData))  # Append the sum of abData for the current node

# Create a DataFrame from the list of sums
twohopcommondf8 = pd.DataFrame(twohopcommon_data8, columns=["Common Connections"])

# Display the DataFrame
print(twohopcommondf8)

twohopcommondf8.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp8.csv', index=False)
```

```
twohopcommondf8.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopcommonp8.csv', index=False)



path8 = '/content/drive/MyDrive/Colab Notebooks/twohopcommonp8.csv'
twohopcommon8 = pd.read_csv(path8)
print(twohopcommon8)

          Common Connections
     0                  5588
     1                  5352
     2                  5347
     3                  5352
     4                  5335
     ..                  ...
     533                 203
     534                 204
     535                 203
     536                 201
     537                 196

     [538 rows x 1 columns]


twohopcommondf = pd.concat([twohopcommon1, twohopcommon2, twohopcommon3, twohopcommon4, twohopcommon5, twohopcommon6, twohopcommon7, twohopc
print(twohopcommondf)

           Common Connections
     0                 33343
     1                  2849
     2                  2856
     3                  2849
     4                  2856
     ...                 ...
     4033                203
     4034                204
     4035                203
     4036                201
     4037                196

     [4038 rows x 1 columns]


import pandas as pd
#near degree data
twodegree_data = []

for node in range(len(G)):
    abData = []
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    if node in acData:
      acData.remove(node)

    for j in acData:
        M =G.degree(j)
        abData.append(M)

    twodegree_data.append(abData)  # Append a new list as a row to all_data

# Create a DataFrame from the list of lists
twodegreedf = pd.DataFrame(twodegree_data, columns= [f'{i} degree connections' for i in range(1, 2916)])

# Display the DataFrame
print(twodegreedf)

          1 degree connections  2 degree connections  3 degree connections  \
     0                      17                   10                    17
     1                     347                   10                    17
     2                     347                   17                    17
     3                     347                   17                    10
     4                     347                   17                    10
     ...                   ...                  ...                   ...
     4034                    8                   59                     8
     4035                    8                   59                     8
     4036                    8                   59                     8
     4037                    8                   59                     8
     4038                    8                   59                     8

          4 degree connections  5 degree connections  6 degree connections  \
     0                      10                   13                     6
```

```
      1                     10                    13                     6
      2                     10                    13                     6
      3                     10                    13                     6
      4                     17                    13                     6
      ...                  ...                   ...                   ...
      4034                  14                     2                     1
      4035                  14                     2                     1
      4036                  14                     2                     1
      4037                  14                     2                     1
      4038                  14                     2                     1

            7 degree connections  8 degree connections  9 degree connections  \
      0                     20                     8                    57
      1                     20                     8                    57
      2                     20                     8                    57
      3                     20                     8                    57
      4                     20                     8                    57
      ...                  ...                   ...                   ...
      4034                   6                     9                     2
      4035                   6                     9                     2
      4036                   6                     9                     2
      4037                   6                     9                     2
      4038                   6                     9                     2

            10 degree connections  ...  2906 degree connections  \
      0                      10  ...                      NaN
      1                      10  ...                      NaN
      2                      10  ...                      NaN
      3                      10  ...                      NaN
      4                      10  ...                      NaN
      ...                   ...  ...                      ...
      4034                    7  ...                      NaN
      4035                    7  ...                      NaN
      4036                    7  ...                      NaN
      4037                    7  ...                      NaN
      4038                    7  ...                      NaN

            2907 degree connections  2908 degree connections  \
      0                       NaN                      NaN
      1                       NaN                      NaN
      2                       NaN                      NaN
      3                       NaN                      NaN
      4                       NaN                      NaN
```

```
#transforming degree data by adding all the degree and counting for a specific degree

# Assuming you have a list of 100 values named values_to_count
values_to_count = range(1046)  # Replace ... with the actual values

# Add columns for each value in values_to_count
for value in values_to_count:
    column_name = f'{value}_count'
    twodegreedf[column_name] = twodegreedf.apply(lambda row: row.eq(value).sum(), axis=1)

print(twodegreedf)
```

```
      4035                  14                     2                     1
      4036                  14                     2                     1
      4037                  14                     2                     1
      4038                  14                     2                     1

            7 degree connections  8 degree connections  9 degree connections  \
```

```
4036                    7  ...           0           0           0
4037                    7  ...           0           0           0
4038                    7  ...           0           0           0

      1039_count  1040_count  1041_count  1042_count  1043_count  1044_count  \
0              0           0           0           0           0           0
1              0           0           0           0           0           0
2              0           0           0           0           0           0
3              0           0           0           0           0           0
4              0           0           0           0           0           0
...          ...         ...         ...         ...         ...         ...
4034           0           0           0           0           0           0
4035           0           0           0           0           0           0
4036           0           0           0           0           0           0
4037           0           0           0           0           0           0
4038           0           0           0           0           0           0

      1045_count
0              1
1              1
2              1
3              1
4              1
...          ...
4034           0
4035           0
4036           0
4037           0
4038           0

[4039 rows x 3961 columns]
```

```
twohopdegreedf = twodegreedf.drop(twodegreedf.columns[:2915], axis=1)
#dropping columns in degreedf
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[793:1045], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[756:792], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[548:755], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[348:547], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[295:347], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[292:294], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[255:291], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[246:254], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[236:245], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[232:234], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[230:231], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[227:229], axis=1)
twohopdegreedf = twohopdegreedf.drop(twohopdegreedf.columns[225:226], axis=1)
# Display the DataFrame after dropping columns
print(twohopdegreedf)
```

```
      0_count  1_count  2_count  3_count  4_count  5_count  6_count  7_count  \
0           0       25       45       38       35       38       32       30
1           0       14       27       22       17       12       15       18
2           0       14       27       22       17       12       15       18
3           0       14       27       22       17       12       15       18
4           0       14       27       22       17       12       15       18
...       ...      ...      ...      ...      ...      ...      ...      ...
4034        0        7        9        4        9        1        4        5
4035        0        6       10        4        9        1        5        4
4036        0        7        9        4        9        1        4        5
4037        0        7       10        4        8        1        4        5
4038        0        7       10        4        9        1        4        5

      8_count  9_count  ...  235_count  245_count  254_count  291_count  \
0          33       39  ...          1          1          1          0
1          15       15  ...          0          0          0          0
2          15       15  ...          0          0          0          0
3          15       15  ...          0          0          0          0
4          15       15  ...          0          0          0          0
...       ...      ...  ...        ...        ...        ...        ...
4034        4        8  ...          0          0          0          0
4035        4        7  ...          0          0          0          0
4036        4        8  ...          0          0          0          0
4037        5        6  ...          0          0          0          0
4038        4        6  ...          0          0          0          0

      294_count  347_count  547_count  755_count  792_count  1045_count
0             1          0          0          1          1           1
1             0          1          0          0          0           1
2             0          1          0          0          0           1
3             0          1          0          0          0           1
4             0          1          0          0          0           1
...         ...        ...        ...        ...        ...         ...
```

```
4034          0          0          0          0          0          0
4035          0          0          0          0          0          0
4036          0          0          0          0          0          0
4037          0          0          0          0          0          0
4038          0          0          0          0          0          0

[4039 rows x 239 columns]
```

```
path8 = '/content/drive/MyDrive/Colab Notebooks/twohopdegreedf.csv'
twohopdegreedf = pd.read_csv(path8)
print(twohopdegreedf)
```

```
import pandas as pd
import networkx as nx

twodcData = []

for node in G.nodes():
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)

    # Calculate the average neighbor degree for the subgraph
    average_neighbor_degree = nx.average_neighbor_degree(subgraph)

    # Round the values to 2 decimal places
    average_degrees = {k: round(v, 2) for k, v in average_neighbor_degree.items()}

    # Sum the average degrees
    B = sum(average_degrees.values())

    twodcData.append(B)

# Create a DataFrame with a single column
twoadcdf = pd.DataFrame({'Average Degree Connectivity for network': twodcData})

# Display the DataFrame
print(twoadcdf)
```

```
twohopdegreedf.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopdegreedf.csv', index=False)
twoadcdf.to_csv('/content/drive/MyDrive/Colab Notebooks/twoadcdf.csv', index=False)
```

```
path8 = '/content/drive/MyDrive/Colab Notebooks/twoadcdf.csv'
twoadcdf = pd.read_csv(path8)
print(twoadcdf)
```

```
twonode_bridge_counts = {}

# Iterate over nodes from 0 to (number of nodes - 1)
for node in range(len(G)):
    # Create a subgraph containing only the current node and its neighbors
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)

    # Calculate local bridges for the subgraph
    local_bridges = list(nx.local_bridges(subgraph, with_span=False))

    # Count the number of bridges for the current node
    bridge_count = len(local_bridges)

    # Store the result in the dictionary
    twonode_bridge_counts[node] = bridge_count



# Convert the dictionary to a DataFrame
twohopbrdf = pd.DataFrame(list(twonode_bridge_counts.values()), columns=['Bridge_Count'])

# Display the DataFrame
print(twohopbrdf)
```

```
      Bridge_Count
0               26
1               14
2               14
3               14
4               14
...            ...
4034             7
4035             7
4036             7
4037             7
4038             7

[4039 rows x 1 columns]
```

```
twohopbrdf.to_csv('/content/drive/MyDrive/Colab Notebooks/twohopbrdf.csv', index=False)



path8 = '/content/drive/MyDrive/Colab Notebooks/twohopbrdf.csv'
twohopbrdf = pd.read_csv(path8)
print(twohopbrdf)
```

```
      Bridge_Count
0               26
1               14
2               14
3               14
4               14
...            ...
4034             7
4035             7
4036             7
4037             7
4038             7

[4039 rows x 1 columns]
```

```
#Returns the boundary expansion of the set S.

#The boundary expansion is the quotient of the size of the node boundary and the cardinality of S. [1]
twoboundary_expansion_list = []

# Iterate over all nodes in the graph
for node in G.nodes():
    # Create a subgraph with the current node and its neighbors
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)
    # Calculate the boundary expansion for the current node
    boundary_expansion = nx.boundary_expansion(G, subgraph)

    # Append the boundary expansion to the list
    twoboundary_expansion_list.append(boundary_expansion)

# Create a DataFrame from the list
twobounddf = pd.DataFrame({'Boundary_Expansion': twoboundary_expansion_list})

# Display the DataFrame
print(twobounddf)
twobounddf.to_csv('/content/drive/MyDrive/Colab Notebooks/twobounddf.csv', index=False)
```

```
      Boundary_Expansion
0               1.146807
1               3.364943
2               3.364943
3               3.364943
4               3.364943
...                  ...
4034            0.066667
4035            0.066667
4036            0.066667
4037            0.066667
4038            0.066667

[4039 rows x 1 columns]
```

```
path8 = '/content/drive/MyDrive/Colab Notebooks/twobounddf.csv'
twobounddf = pd.read_csv(path8)
print(twobounddf)
```

```
      Boundary_Expansion
0               1.146807
1               3.364943
2               3.364943
3               3.364943
4               3.364943
...                  ...
4034            0.066667
4035            0.066667
4036            0.066667
4037            0.066667
4038            0.066667

[4039 rows x 1 columns]
```

```
#The average global efficiency of a graph is the average efficiency of all pairs of nodes
import pandas as pd
import networkx as nx
from itertools import combinations

twoglobaleff_list = []
twoweight_matching_list = []
tworandom_list = []
panther_list = []
# Iterate over all nodes in the graph
for node in G.nodes():
    # Create a subgraph with the current node and its neighbors
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)

    # Calculate the boundary expansion for the current node
    globaleff = nx.global_efficiency(subgraph)

    weighteff = len(list(nx.max_weight_matching(subgraph, maxcardinality=False, weight='weight')))

    result = nx.non_randomness(subgraph, k=None, weight='weight')
    second_value = result[1]

    #panther = len(nx.panther_similarity(subgraph, 0, k=5, path_length=5, c=0.5, delta=0.1, eps=None, weight='weight'))

    # Append the boundary expansion to the list
    twoglobaleff_list.append(globaleff)
    twoweight_matching_list.append(weighteff)
    tworandom_list.append(second_value)
    #panther_list.append(panther)
# Create a DataFrame from the list
twoglobaldf = pd.DataFrame({
    'global_efficiency': twoglobaleff_list,
    'max_weight_matching': twoweight_matching_list,
    'random_probability': tworandom_list,
    # 'panther_similarity': panther_list  # Uncomment if needed
})
# Display the DataFrame
print(twoglobaldf)

twoglobaldf.to_csv('/content/drive/MyDrive/Colab Notebooks/twoglobaldf.csv', index=False)
```

```
      global_efficiency  max_weight_matching  random_probability
0              0.435346                  744          -66.852262
1              0.523734                  164           75.626117
2              0.523734                  164           75.626117
3              0.523734                  164           75.626117
4              0.523734                  164           75.626117
...                 ...                  ...                 ...
4034           0.557910                   26           10.139742
4035           0.557910                   26           10.139742
4036           0.557910                   26           10.139742
4037           0.557910                   26           10.139742
4038           0.557910                   26           10.139742

[4039 rows x 3 columns]
```

```
path8 = '/content/drive/MyDrive/Colab Notebooks/twoglobaldf.csv'
twoglobaldf = pd.read_csv(path8)
print(twoglobaldf)
```

```
      global_efficiency  max_weight_matching  random_probability
0              0.435346                  744          -66.852262
1              0.523734                  164           75.626117
2              0.523734                  164           75.626117
3              0.523734                  164           75.626117
4              0.523734                  164           75.626117
...                 ...                  ...                 ...
4034           0.557910                   26           10.139742
4035           0.557910                   26           10.139742
4036           0.557910                   26           10.139742
4037           0.557910                   26           10.139742
4038           0.557910                   26           10.139742

[4039 rows x 3 columns]
```

```
#The Kemeny constant measures the time needed for spreading across a graph. Low values indicate a closely connected graph whereas high value
twokemeny_list = []

# Iterate over all nodes in the graph
for node in G.nodes():
    # Create a subgraph with the current node and its neighbors
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)

    # Calculate the boundary expansion for the current node
    kemeny = nx.kemeny_constant(subgraph)

    # Append the boundary expansion to the list
    twokemeny_list.append(kemeny)

# Create a DataFrame from the list
twokemenydf = pd.DataFrame({'kemeny constant for network': twokemeny_list})

# Display the DataFrame
print(twokemenydf)

twokemenydf.to_csv('/content/drive/MyDrive/Colab Notebooks/twokemenydf.csv', index=False)
```

```
      kemeny constant for network
0                     2030.583502
1                      406.232390
2                      406.232390
3                      406.232390
4                      406.232390
...                           ...
4034                    68.707931
4035                    68.707931
4036                    68.707931
4037                    68.707931
4038                    68.707931

[4039 rows x 1 columns]
```

```
path8 = '/content/drive/MyDrive/Colab Notebooks/twokemenydf.csv'
twokemenydf = pd.read_csv(path8)
print(twokemenydf)
```

```
      kemeny constant for network
0                     2030.583502
1                      406.232390
2                      406.232390
3                      406.232390
4                      406.232390
...                           ...
4034                    68.707931
4035                    68.707931
4036                    68.707931
4037                    68.707931
4038                    68.707931

[4039 rows x 1 columns]
```

```
twokcore_counts = []

# Iterate over nodes from 0 to (number of nodes - 1)
for node in G.nodes():
    # Create a subgraph containing only the current node and its neighbors
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)

    # Calculate local bridges for the subgraph
    k_core = nx.core_number(subgraph)
    B = sum(list(k_core.values()))


    twokcore_counts.append(B)

# Convert the list to a DataFrame
twohopkkcoredf = pd.DataFrame({'K-core for network': twokcore_counts})

# Display the DataFrame
print(twohopkkcoredf)
```

```
      K-core for network
0                  40653
1                   3400
2                   3400
3                   3400
4                   3400
...                  ...
4034                 255
4035                 255
4036                 255
4037                 255
4038                 255

[4039 rows x 1 columns]
```

```
twotriangle_counts = []

# Iterate over nodes from 0 to (number of nodes - 1)
for node in G.nodes():
    # Create a subgraph containing only the current node and its neighbors
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)

    # Calculate local bridges for the subgraph
    triangle = nx.triangles(subgraph, nodes=None)
    B = sum(list(triangle.values()))

    twotriangle_counts.append(B)

# Convert the list to a DataFrame
twohopktriangledf = pd.DataFrame({'triangle for network': twotriangle_counts})

# Display the DataFrame
print(twohopktriangledf)
```

```
      triangle for network
0                  1514841
1                    39777
2                    39777
3                    39777
4                    39777
...                    ...
4034                   942
4035                   942
4036                   942
4037                   942
4038                   942

[4039 rows x 1 columns]
```

```
#getting all the necessary data from one hop for the Node Data
selected_columns = ['Node', 'NodeDegree', 'Average Degree Connectivity for Node', 'Core_Number', 'onion layers', 'eccentricity', 'No of tria

# Creating a new DataFrame with the selected columns
commonhopdf = onefinaldf[selected_columns]

# Display the new DataFrame
print(commonhopdf)
```

```
      Node  NodeDegree  Average Degree Connectivity for Node  Core_Number  \
0        0         347                                18.96           21
1        1          17                                48.24           13
2        2          10                                49.90            9
3        3          17                                59.76           13
4        4          10                                42.60            9
...    ...         ...                                  ...          ...
4034  4034           2                                38.50            2
4035  4035           1                                59.00            1
4036  4036           2                                31.50            2
4037  4037           4                                23.25            4
4038  4038           9                                15.22            5

      onion layers  eccentricity  No of triangles with Node
0              108             6                       2519
1               51             7                         57
2               30             7                         40
3               49             7                         86
4               30             7                         39
...            ...           ...                        ...
4034             2             8                          1
4035             1             8                          0
4036             2             8                          1
4037             8             8                          4
4038            14             8                         20

[4039 rows x 7 columns]
```

```
twoonion_counts = []

# Iterate over nodes from 0 to (number of nodes - 1)
for node in G.nodes():
    # Create a subgraph containing only the current node and its neighbors
    length = nx.single_source_shortest_path_length(G, node, cutoff=2)
    acData = sorted(list(length.keys()))

    # Include the node itself in the subgraph
    subgraph = G.subgraph(acData)

    # Calculate local bridges for the subgraph
    k_core = nx.onion_layers(subgraph)
    sorted_keys = sorted(k_core.values())

    B = sum(list(sorted_keys))

    twoonion_counts.append(B)

# Convert the list to a DataFrame
twooniondf = pd.DataFrame({'onion layers for network': twoonion_counts})

# Display the DataFrame
print(twooniondf)
```

```
      onion layers for network
0                       143440
1                         8567
2                         8567
3                         8567
4                         8567
...                        ...
4034                       447
4035                       447
4036                       447
4037                       447
4038                       447

[4039 rows x 1 columns]
```

```
#Adding the twohop data
twohopdatadf = pd.concat([commonhopdf, twohopcommondf, twohopdegreedf, twoadcdf, twooniondf, twohopbrdf, twobounddf, twoglobaldf, twokemenyd
print(twohopdatadf)
```

```
      Node  NodeDegree  Average Degree Connectivity for Node  Core_Number  \
0        0         347                                18.96           21
1        1          17                                48.24           13
2        2          10                                49.90            9
3        3          17                                59.76           13
4        4          10                                42.60            9
...    ...         ...                                  ...          ...
4034  4034           2                                38.50            2
4035  4035           1                                59.00            1
4036  4036           2                                31.50            2
4037  4037           4                                23.25            4
4038  4038           9                                15.22            5

      onion layers  eccentricity  No of triangles with Node  \
0              108             6                       2519
1               51             7                         57
2               30             7                         40
3               49             7                         86
4               30             7                         39
...            ...           ...                        ...
4034             2             8                          1
4035             1             8                          0
4036             2             8                          1
4037             8             8                          4
4038            14             8                         20

      Common Connections  0_count  1_count  ...  \
0                33343.0        0       25  ...
1                 2849.0        0       14  ...
2                 2856.0        0       14  ...
3                 2849.0        0       14  ...
4                 2856.0        0       14  ...
...                  ...      ...      ...  ...
4034              204.0        0        7  ...
4035              203.0        0        6  ...
4036              201.0        0        7  ...
4037              196.0        0        7  ...
4038                NaN        0        7  ...

      Average Degree Connectivity for network  onion layers for network  \
0                                  174898.08                    143440
1                                   28959.92                      8567
2                                   28959.92                      8567
3                                   28959.92                      8567
4                                   28959.92                      8567
...                                      ...                       ...
4034                                1481.20                       447
4035                                1481.20                       447
4036                                1481.20                       447
4037                                1481.20                       447
4038                                1481.20                       447

      Bridge_Count  Boundary_Expansion  global_efficiency  \
0               26            1.146807           0.435346
1               14            3.364943           0.523734
2               14            3.364943           0.523734
3               14            3.364943           0.523734
4               14            3.364943           0.523734
```

```
#finalizing two hop data by adding networkx library clustering coefficient data to check with the metrics
twofinaldf = pd.concat([twohopdatadf, cludf], axis = 1)
print(twofinaldf)
```

```
      Node  NodeDegree  Average Degree Connectivity for Node  Core_Number  \
0        0         347                                18.96           21
1        1          17                                48.24           13
2        2          10                                49.90            9
3        3          17                                59.76           13
4        4          10                                42.60            9
...    ...         ...                                  ...          ...
4034  4034           2                                38.50            2
4035  4035           1                                59.00            1
4036  4036           2                                31.50            2
4037  4037           4                                23.25            4
4038  4038           9                                15.22            5

      onion layers  eccentricity  No of triangles with Node  \
0              108             6                       2519
1               51             7                         57
```

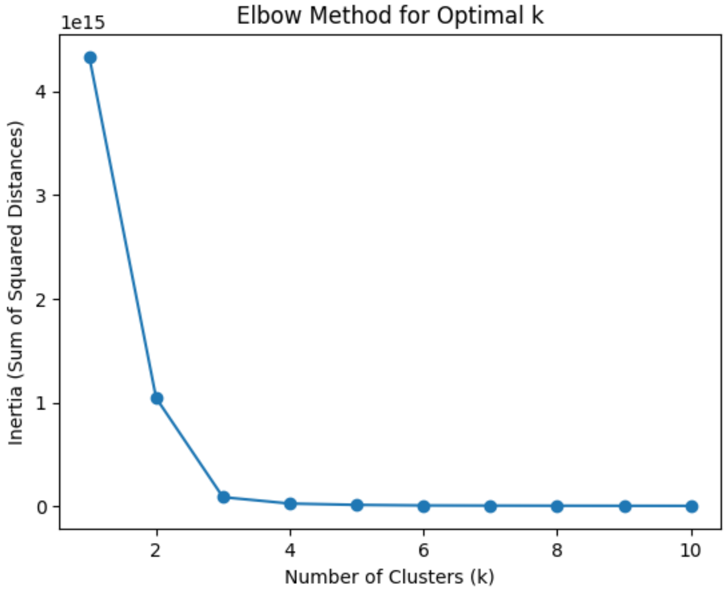
```
2              30            7                      40
3              49            7                      86
4              30            7                      39
...           ...          ...                     ...
4034            2            8                       1
4035            1            8                       0
4036            2            8                       1
4037            8            8                       4
4038           14            8                      20

      Common Connections  0_count  1_count  ...  onion layers for network  \
0                33343.0        0       25  ...                    143440
1                 2849.0        0       14  ...                      8567
2                 2856.0        0       14  ...                      8567
3                 2849.0        0       14  ...                      8567
4                 2856.0        0       14  ...                      8567
...                  ...      ...      ...  ...                       ...
4034               204.0        0        7  ...                       447
4035               203.0        0        6  ...                       447
4036               201.0        0        7  ...                       447
4037               196.0        0        7  ...                       447
4038                 NaN        0        7  ...                       447

      Bridge_Count  Boundary_Expansion  global_efficiency  \
0               26            1.146807           0.435346
1               14            3.364943           0.523734
2               14            3.364943           0.523734
3               14            3.364943           0.523734
4               14            3.364943           0.523734
...            ...                 ...                ...
4034             7            0.066667           0.557910
4035             7            0.066667           0.557910
4036             7            0.066667           0.557910
4037             7            0.066667           0.557910
4038             7            0.066667           0.557910

      max_weight_matching  random_probability  kemeny constant for network  \
0                     744          -66.852262                   2030.583502
1                     164           75.626117                    406.232390
2                     164           75.626117                    406.232390
3                     164           75.626117                    406.232390
4                     164           75.626117                    406.232390
```

```
twofinaldf.replace([np.inf, -np.inf], 0, inplace=True)
```

```
twofinaldata = twofinaldf.fillna(0)
```

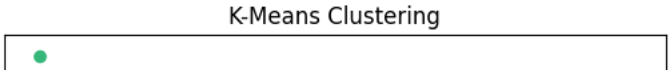
```
#determining the number of Steps for K means
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Assuming df is your DataFrame with standardized features
# Step 1: Standardize the data
scaler = StandardScaler()
features_standardized = scaler.fit_transform(twofinaldata)

# Store the sum of squared distances for different values of k
inertia_values = []

# Define a range of k values to try
k_values = range(1, 11)  # You can adjust the range based on your problem

# Apply k-means clustering for each k and store the inertia
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(twofinaldata)
    inertia_values.append(kmeans.inertia_)

# Plot the elbow curve
plt.plot(k_values, inertia_values, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia (Sum of Squared Distances)')
plt.show()
```

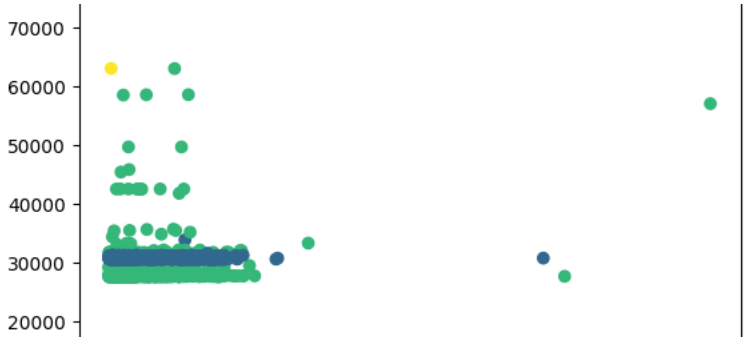
```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
```




```
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Step 1: Standardize the data
scaler = StandardScaler()
features_standardized = scaler.fit_transform(twofinaldata)

# Step 3: Apply k-means clustering
k = 4  # Replace with your optimal k
kmeans = KMeans(n_clusters=k, random_state=42)
twofinaldata['k=4_Cluster'] = kmeans.fit_predict(features_standardized)

# Display the DataFrame with cluster assignments
print(twofinaldata)

# Step 4: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
# Visualization is more challenging with a large number of features
# You may need to choose specific features for plotting or use advanced visualization techniques
plt.scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['k=4_Cluster'], cmap='viridis')
plt.title('K-Means Clustering')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fr
  warnings.warn(
      Node  NodeDegree  Average Degree Connectivity for Node  Core_Number  \
0        0         347                                18.96           21
1        1          17                                48.24           13
2        2          10                                49.90            9
3        3          17                                59.76           13
4        4          10                                42.60            9
...    ...         ...                                  ...          ...
4034  4034           2                                38.50            2
4035  4035           1                                59.00            1
4036  4036           2                                31.50            2
4037  4037           4                                23.25            4
4038  4038           9                                15.22            5

      onion layers  eccentricity  No of triangles with Node  \
0              108             6                       2519
1               51             7                         57
2               30             7                         40
3               49             7                         86
4               30             7                         39
...            ...           ...                        ...
4034             2             8                          1
4035             1             8                          0
4036             2             8                          1
4037             8             8                          4
4038            14             8                         20

      Common Connections  0_count  1_count  ...  Bridge_Count  \
0                33343.0        0       25  ...            26
1                 2849.0        0       14  ...            14
2                 2856.0        0       14  ...            14
3                 2849.0        0       14  ...            14
4                 2856.0        0       14  ...            14
...                  ...      ...      ...  ...           ...
4034               204.0        0        7  ...             7
4035               203.0        0        6  ...             7
4036               201.0        0        7  ...             7
4037               196.0        0        7  ...             7
4038                 0.0        0        7  ...             7

      Boundary_Expansion  global_efficiency  max_weight_matching  \
0               1.146807           0.435346                  744
1               3.364943           0.523734                  164
2               3.364943           0.523734                  164
3               3.364943           0.523734                  164
4               3.364943           0.523734                  164
...                  ...                ...                  ...
4034            0.066667           0.557910                   26
4035            0.066667           0.557910                   26
4036            0.066667           0.557910                   26
4037            0.066667           0.557910                   26
4038            0.066667           0.557910                   26

      random_probability  kemeny constant for network  K-core for network  \
0             -66.852262                  2030.583502               40653
1              75.626117                   406.232390                3400
2              75.626117                   406.232390                3400
3              75.626117                   406.232390                3400
4              75.626117                   406.232390                3400
...                  ...                          ...                 ...
4034           10.139742                    68.707931                 255
4035           10.139742                    68.707931                 255
4036           10.139742                    68.707931                 255
4037           10.139742                    68.707931                 255
4038           10.139742                    68.707931                 255

      triangle for network  library Clustering  k=4_Cluster
0                  1514841               0.04            2
1                    39777               0.42            3
2                    39777               0.89            3
3                    39777               0.63            3
4                    39777               0.87            3
...                    ...                ...          ...
4034                   942               1.00            3
4035                   942               0.00            3
4036                   942               1.00            3
4037                   942               0.67            3
4038                   942               0.56            3

[4039 rows x 259 columns]
```

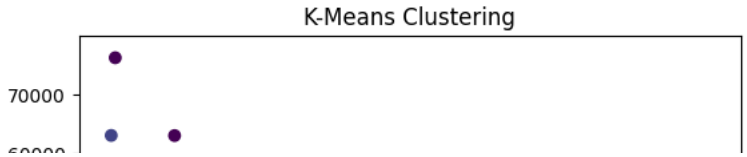





```
#Step 3: Apply k-means clustering
N = 6  # Replace with your optimal k
kmeans = KMeans(n_clusters=N, random_state=42)
twofinaldata['k = 6_Cluster'] = kmeans.fit_predict(features_standardized)


# Step 4: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
# Visualization is more challenging with a large number of features
# You may need to choose specific features for plotting or use advanced visualization techniques
plt.scatter(twofinaldata["Common Connections"], twofinaldata['NodeDegree'], c=onefinaldf['k = 6_Cluster'], cmap='viridis')
plt.title('K-Means Clustering')
plt.show()
```

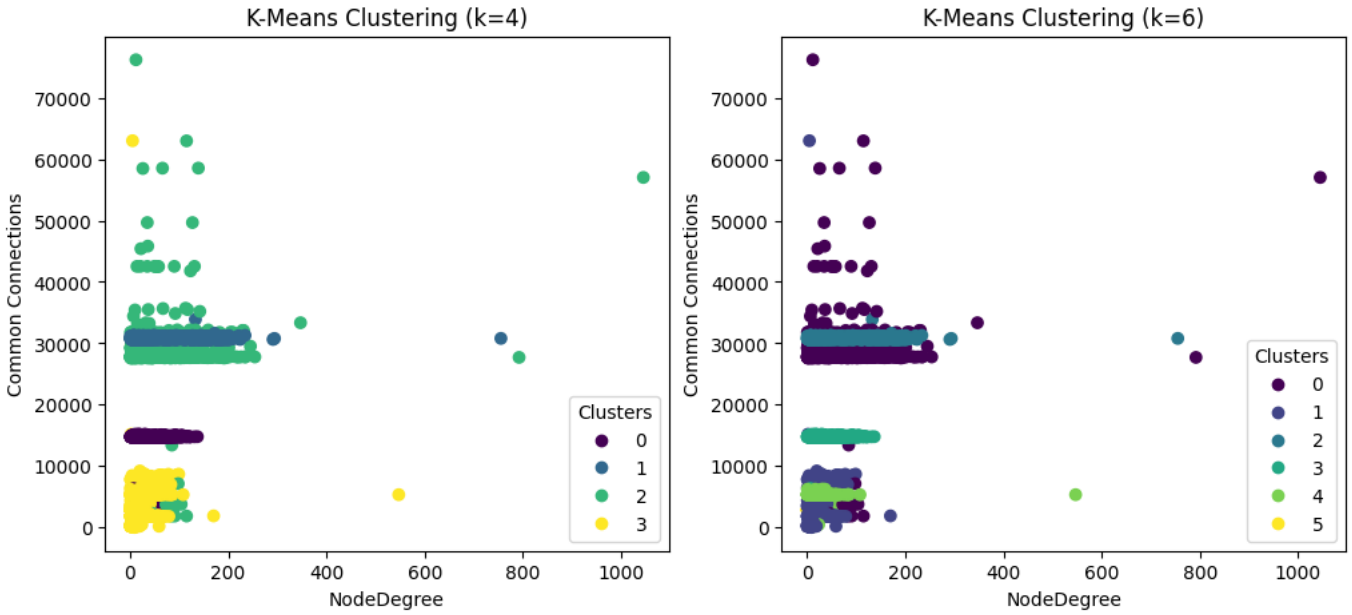
```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change from 10
  warnings.warn(
```


```
# You may need to choose specific features for plotting or use advanced visualization techniques
plt.scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['k = 6_Cluster'], cmap='viridis')
plt.title('K-Means Clustering')
plt.show()
```



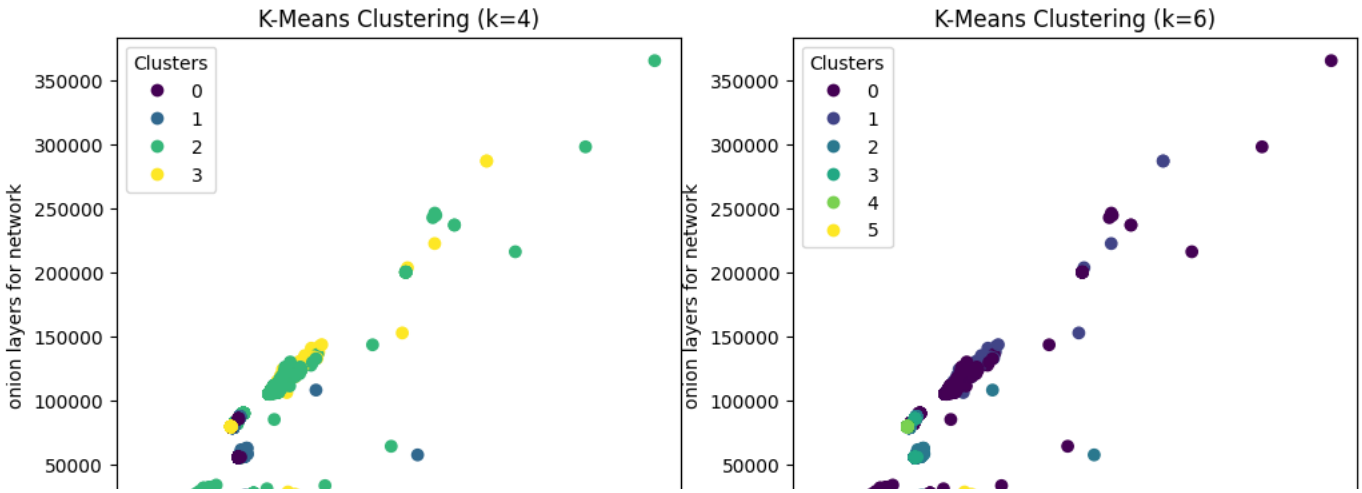

```
# Step 4: Visualize the clusters
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot for k=4 clusters
scatter_4 = axes[0].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['k=4_Cluster'], cmap='viridis')
axes[0].set_title(f'K-Means Clustering (k=4)')
axes[0].set_xlabel('NodeDegree')
axes[0].set_ylabel('Common Connections')
axes[0].legend(*scatter_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters
scatter_6 = axes[1].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['k = 6_Cluster'], cmap='viridis')
axes[1].set_title(f'K-Means Clustering (k=6)')
axes[1].set_xlabel('NodeDegree')
axes[1].set_ylabel('Common Connections')
axes[1].legend(*scatter_6.legend_elements(), title='Clusters')
```

```
<matplotlib.legend.Legend at 0x7a199aae6c20>
```

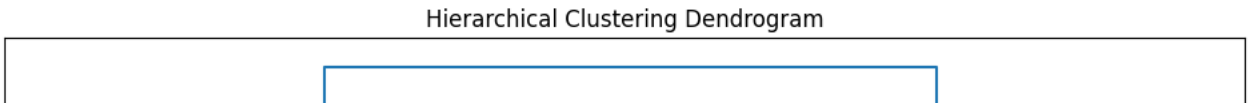



```
# visualize with kemeny constant - time and Onion layers of the network for clustering
# Step 4: Visualize the clusters
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot for k=4 clusters
scatter_4 = axes[0].scatter(twofinaldata["kemeny constant for network"], twofinaldata['onion layers for network'], c=twofinaldata['k=4_Clust
axes[0].set_title(f'K-Means Clustering (k=4)')
axes[0].set_xlabel('kemeny constant for network')
axes[0].set_ylabel('onion layers for network')
axes[0].legend(*scatter_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters
scatter_6 = axes[1].scatter(twofinaldata["kemeny constant for network"], twofinaldata['onion layers for network'], c=twofinaldata['k = 6_Clu
axes[1].set_title(f'K-Means Clustering (k=6)')
axes[1].set_xlabel('kemeny constant for network')
axes[1].set_ylabel('onion layers for network')
axes[1].legend(*scatter_6.legend_elements(), title='Clusters')
```

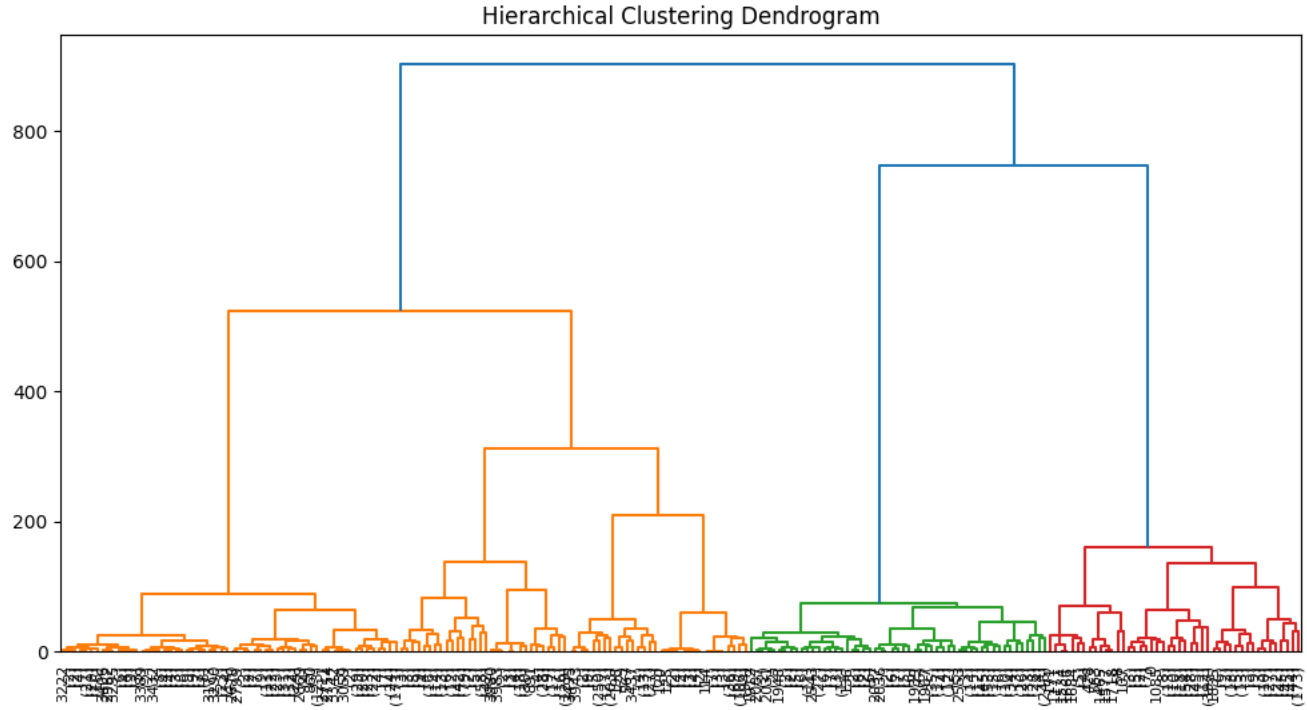
```
<matplotlib.legend.Legend at 0x7a198d744a30>
```



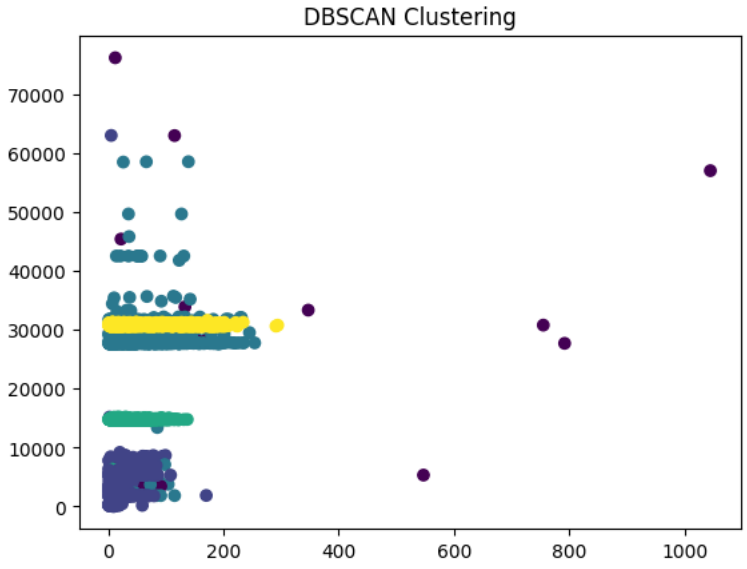

```
# Agglomerative Clustering
import pandas as pd
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage



# Step 1: Standardize the data (optional but recommended for hierarchical clustering)
scaler = StandardScaler()
data_standardized = scaler.fit_transform(twofinaldata)

# Step 2: Determine the linkage matrix using the ward method
linkage_matrix = linkage(data_standardized, method='ward')

# Step 3: Plot the dendrogram (optional but useful for visualizing hierarchical clustering)
plt.figure(figsize=(12, 6))
dendrogram(linkage_matrix, leaf_rotation=90., leaf_font_size=8., truncate_mode='level', p=5)
plt.title('Hierarchical Clustering Dendrogram')
plt.show()

# Step 4: Apply Agglomerative Hierarchical Clustering
# Choose the appropriate number of clusters (n_clusters) based on the dendrogram
n_clusters = 4  # Replace with your desired number of clusters
hierarchical_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward')
twofinaldata['agg4Cluster'] = hierarchical_clustering.fit_predict(data_standardized)
```




```
linkage_matrix = linkage(data_standardized, method='ward')

# Step 3: Plot the dendrogram (optional but useful for visualizing hierarchical clustering)
plt.figure(figsize=(12, 6))
dendrogram(linkage_matrix, leaf_rotation=90., leaf_font_size=8., truncate_mode='level', p=7)
plt.title('Hierarchical Clustering Dendrogram')
plt.show()

# Step 4: Apply Agglomerative Hierarchical Clustering
# Choose the appropriate number of clusters (n_clusters) based on the dendrogram
n_clusters = 6  # Replace with your desired number of clusters
hierarchical_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward')
twofinaldata['agg6Cluster'] = hierarchical_clustering.fit_predict(data_standardized)
```

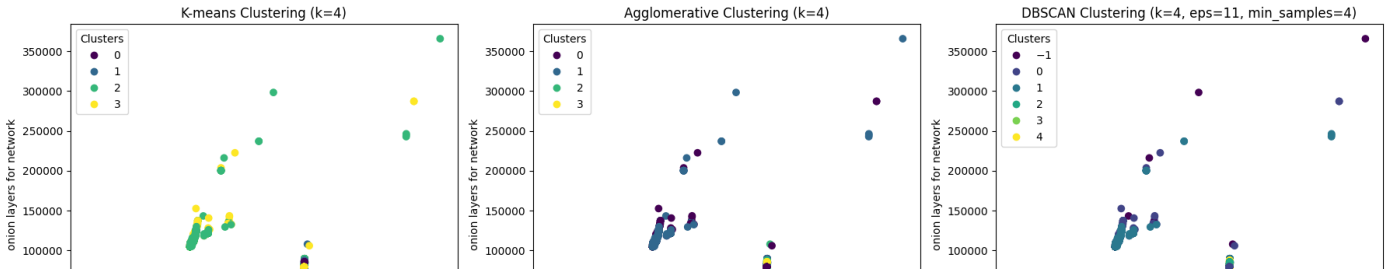



```
#DBSCAN
#dbscan algorithm
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Assuming df is your DataFrame

# Step 1: Standardize the data (required for DBSCAN)
scaler = StandardScaler()
data_standardized = scaler.fit_transform(twofinaldata)

# Step 2: Apply DBSCAN
# Choose appropriate values for `eps` (maximum distance between two samples for one to be considered as in the neighborhood of the other)
# and `min_samples` (the number of samples in a neighborhood for a point to be considered as a core point)
dbscan = DBSCAN(eps=11, min_samples=4)
twofinaldata['dbs4Cluster'] = dbscan.fit_predict(data_standardized)


# Step 3: Visualize the clusters (optional)
# This is just an example, adjust it based on your requirements
plt.scatter(twofinaldata['NodeDegree'], twofinaldata['Common Connections'], c=twofinaldata['dbs4Cluster'], cmap='viridis')
plt.title('DBSCAN Clustering')
plt.show()
```

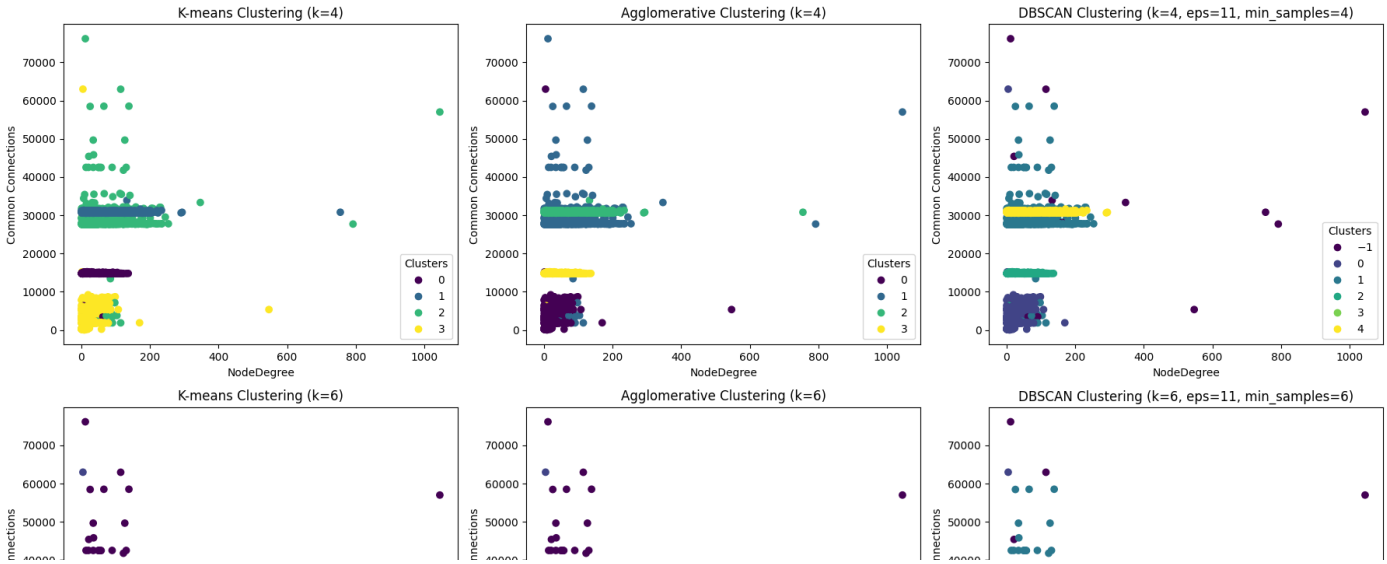



```
# Step 2: Apply DBSCAN
# Choose appropriate values for `eps` (maximum distance between two samples for one to be considered as in the neighborhood of the other)
# and `min_samples` (the number of samples in a neighborhood for a point to be considered as a core point)
dbscan = DBSCAN(eps=11, min_samples=6)
twofinaldata['dbs6Cluster'] = dbscan.fit_predict(data_standardized)
```

```
# Visualize clusters for k-means, agglomerative clustering, and DBSCAN in a 2x3 grid
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(18, 10))

# Plot for k=4 clusters in k-means
scatter_4_kmeans = axes[0, 0].scatter(twofinaldata["triangle for network"], twofinaldata['onion layers for network'], c=twofinaldata['k=4_Cl
axes[0, 0].set_title('K-means Clustering (k=4)')
axes[0, 0].set_xlabel('triangle for network')
axes[0, 0].set_ylabel('onion layers for network')
axes[0, 0].legend(*scatter_4_kmeans.legend_elements(), title='Clusters')

# Plot for k=4 clusters in agglomerative clustering
scatter_4_agg = axes[0, 1].scatter(twofinaldata["triangle for network"], twofinaldata['onion layers for network'], c=twofinaldata['agg4Clust
axes[0, 1].set_title('Agglomerative Clustering (k=4)')
axes[0, 1].set_xlabel('triangle for network')
axes[0, 1].set_ylabel('onion layers for network')
axes[0, 1].legend(*scatter_4_agg.legend_elements(), title='Clusters')

# Plot for k=6 clusters in k-means
scatter_6_kmeans = axes[1, 0].scatter(twofinaldata["triangle for network"], twofinaldata['onion layers for network'], c=twofinaldata['k = 6_
axes[1, 0].set_title('K-means Clustering (k=6)')
axes[1, 0].set_xlabel('triangle for network')
axes[1, 0].set_ylabel('onion layers for network')
axes[1, 0].legend(*scatter_6_kmeans.legend_elements(), title='Clusters')

# Plot for k=6 clusters in agglomerative clustering
scatter_6_agg = axes[1, 1].scatter(twofinaldata["triangle for network"], twofinaldata['onion layers for network'], c=twofinaldata['agg6Clust
axes[1, 1].set_title('Agglomerative Clustering (k=6)')
axes[1, 1].set_xlabel('triangle for network')
axes[1, 1].set_ylabel('onion layers for network')
axes[1, 1].legend(*scatter_6_agg.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=4, eps=11, min_samples=5)
scatter_dbscan_4 = axes[0, 2].scatter(twofinaldata["triangle for network"], twofinaldata['onion layers for network'], c=twofinaldata['dbs4Cl
axes[0, 2].set_title('DBSCAN Clustering (k=4, eps=11, min_samples=4)')
axes[0, 2].set_xlabel('triangle for network')
axes[0, 2].set_ylabel('onion layers for network')
axes[0, 2].legend(*scatter_dbscan_4.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=6, eps=0.8, min_samples=5)
scatter_dbscan_6 = axes[1, 2].scatter(twofinaldata["triangle for network"], twofinaldata['onion layers for network'], c=twofinaldata['dbs6Cl
axes[1, 2].set_title('DBSCAN Clustering (k=6, eps=11, min_samples=6)')
axes[1, 2].set_xlabel('triangle for network')
axes[1, 2].set_ylabel('onion layers for network')
axes[1, 2].legend(*scatter_dbscan_6.legend_elements(), title='Clusters')

plt.tight_layout()
plt.show()
```

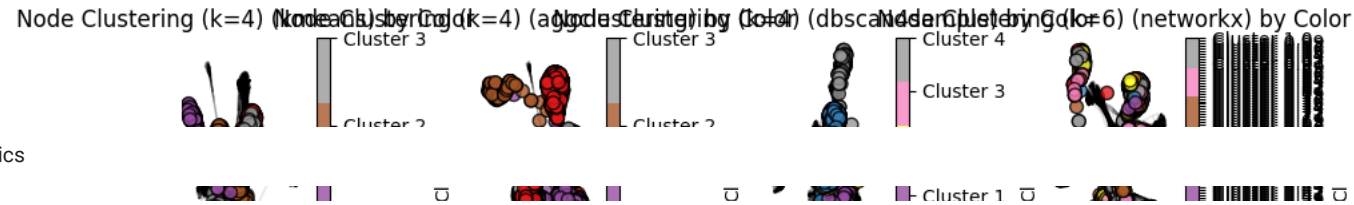



```
# Step 4: Visualize the clusters for k-means and agglomerative clustering
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(18, 10))

# Plot for k=4 clusters in k-means
scatter_4_kmeans = axes[0, 0].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['k=4_Cluster'], cmap='v
axes[0, 0].set_title('K-means Clustering (k=4)')
axes[0, 0].set_xlabel('NodeDegree')
axes[0, 0].set_ylabel('Common Connections')
axes[0, 0].legend(*scatter_4_kmeans.legend_elements(), title='Clusters')

# Plot for k=4 clusters in agglomerative clustering
scatter_4_agg = axes[0, 1].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['agg4Cluster'], cmap='viri
axes[0, 1].set_title('Agglomerative Clustering (k=4)')
axes[0, 1].set_xlabel('NodeDegree')
axes[0, 1].set_ylabel('Common Connections')
axes[0, 1].legend(*scatter_4_agg.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=4, eps=11, min_samples=5)
scatter_dbscan_4 = axes[0, 2].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['dbs4Cluster'], cmap='v
axes[0, 2].set_title('DBSCAN Clustering (k=4, eps=11, min_samples=4)')
axes[0, 2].set_xlabel('NodeDegree')
axes[0, 2].set_ylabel('Common Connections')
axes[0, 2].legend(*scatter_dbscan_4.legend_elements(), title='Clusters')

# Plot for k=6 clusters in k-means
scatter_6_kmeans = axes[1, 0].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['k = 6_Cluster'], cmap=
axes[1, 0].set_title('K-means Clustering (k=6)')
axes[1, 0].set_xlabel('NodeDegree')
axes[1, 0].set_ylabel('Common Connections')
axes[1, 0].legend(*scatter_6_kmeans.legend_elements(), title='Clusters')

# Plot for k=6 clusters in agglomerative clustering
scatter_6_agg = axes[1, 1].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['agg6Cluster'], cmap='viri
axes[1, 1].set_title('Agglomerative Clustering (k=6)')
axes[1, 1].set_xlabel('NodeDegree')
axes[1, 1].set_ylabel('Common Connections')
axes[1, 1].legend(*scatter_6_agg.legend_elements(), title='Clusters')

# Plot for DBSCAN clusters (k=6, eps=0.8, min_samples=5)
scatter_dbscan_6 = axes[1, 2].scatter(twofinaldata["NodeDegree"], twofinaldata['Common Connections'], c=twofinaldata['dbs6Cluster'], cmap='v
axes[1, 2].set_title('DBSCAN Clustering (k=6, eps=11, min_samples=6)')
axes[1, 2].set_xlabel('NodeDegree')
axes[1, 2].set_ylabel('Common Connections')
axes[1, 2].legend(*scatter_dbscan_6.legend_elements(), title='Clusters')

plt.tight_layout()
plt.show()
```



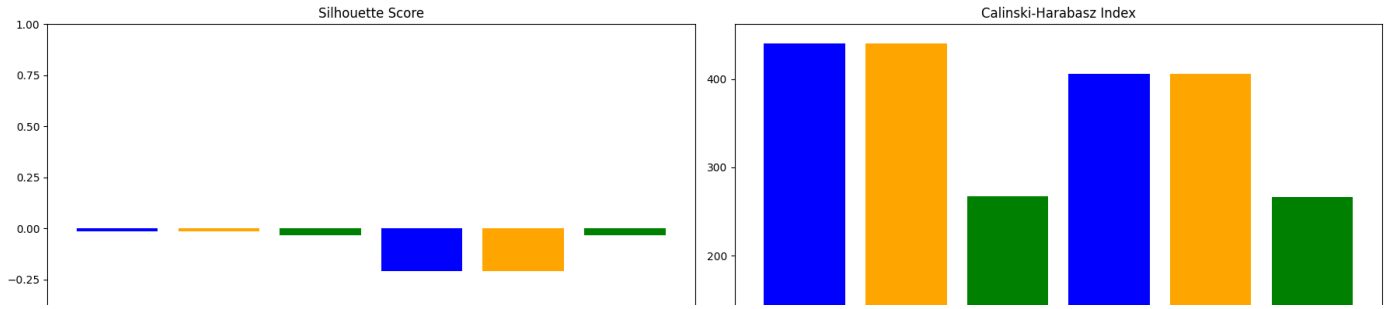

```
#generate graphs
# Create a 2x4 grid for subplots
fig, axes = plt.subplots(2, 4, figsize=(9, 5))

# Plot for k=4
plot_clustersname(G, twofinaldata['k=4_Cluster'], 4, 'kmeans', axes[0, 0])

# Plot for k=4 - agg
plot_clustersname(G, twofinaldata['agg4Cluster'], 4, 'aggclustering', axes[0, 1])

# Plot for k=4 - dbscan
plot_clustersname(G, twofinaldata['dbs4Cluster'], 4, 'dbscan4sample', axes[0, 2])

#plot for networkx - library
plot_clustersname(G, twofinaldata['library Clustering'], 6, 'networkx', axes[0,3])

# Plot for k=6
plot_clustersname(G, twofinaldata['k = 6_Cluster'], 6, 'kmeans', axes[1, 0])

# Plot for k=6 - agg
plot_clustersname(G, twofinaldata['agg6Cluster'], 6, 'aggclustering', axes[1, 1])

# Plot for k=6 - dbscan
plot_clustersname(G, twofinaldata['dbs6Cluster'], 4, 'dbscan6sample', axes[1, 2])

#plot for networkx - library
plot_clustersname(G, twofinaldata['library Clustering'], 6, 'networkx', axes[1,3])

# Adjust layout
plt.tight_layout()
plt.show()
```

```
<ipython-input-117-83f07bba3dce>:8: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be rem
  nodes = nx.draw_networkx_nodes(graph, pos=nx.spring_layout(graph), node_color=clusters, cmap=plt.cm.get_cmap('Set1', len(clusters.uniq
```



Metrics

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import silhouette_score, calinski_harabasz_score

# Extract features
X = twofinaldata[['triangle for network', 'onion layers for network']]

# Extract cluster labels
kmeans_labels_4 = twofinaldata['k=4_Cluster']
agg_labels_4 = twofinaldata['agg4Cluster']
dbscan_labels_4 = twofinaldata['dbs4Cluster']

kmeans_labels_6 = twofinaldata['k = 6_Cluster']
agg_labels_6 = twofinaldata['agg6Cluster']
dbscan_labels_6 = twofinaldata['dbs6Cluster']

# Calculate Silhouette Score
kmeans_silhouette_4 = silhouette_score(X, kmeans_labels_4)
agg_silhouette_4 = silhouette_score(X, agg_labels_4)
dbscan_silhouette_4 = silhouette_score(X, dbscan_labels_4)

kmeans_silhouette_6 = silhouette_score(X, kmeans_labels_6)
agg_silhouette_6 = silhouette_score(X, agg_labels_6)
dbscan_silhouette_6 = silhouette_score(X, dbscan_labels_6)

# Calculate Calinski-Harabasz Index
kmeans_calinski_harabasz_4 = calinski_harabasz_score(X, kmeans_labels_4)
agg_calinski_harabasz_4 = calinski_harabasz_score(X, agg_labels_4)
dbscan_calinski_harabasz_4 = calinski_harabasz_score(X, dbscan_labels_4)

kmeans_calinski_harabasz_6 = calinski_harabasz_score(X, kmeans_labels_6)
agg_calinski_harabasz_6 = calinski_harabasz_score(X, agg_labels_6)
dbscan_calinski_harabasz_6 = calinski_harabasz_score(X, dbscan_labels_6)

# Data for plotting
algorithms = ['K-means (k=4)', 'Agglomerative (k=4)', 'DBSCAN (k=4)', 'K-means (k=6)', 'Agglomerative (k=6)', 'DBSCAN (k=6)']
silhouette_scores = [kmeans_silhouette_4, agg_silhouette_4, dbscan_silhouette_4, kmeans_silhouette_6, agg_silhouette_6, dbscan_silhouette_6]
calinski_harabasz_scores = [kmeans_calinski_harabasz_4, agg_calinski_harabasz_4, dbscan_calinski_harabasz_4, kmeans_calinski_harabasz_6, agg

# Plot Silhouette Score and Calinski-Harabasz Index
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(18, 6))

# Plot Silhouette Score
axes[0].bar(algorithms, silhouette_scores, color=['blue', 'orange', 'green', 'blue', 'orange', 'green'])
axes[0].set_title('Silhouette Score')
axes[0].set_ylim(-1, 1)

# Plot Calinski-Harabasz Index
axes[1].bar(algorithms, calinski_harabasz_scores, color=['blue', 'orange', 'green', 'blue', 'orange', 'green'])
axes[1].set_title('Calinski-Harabasz Index')

plt.tight_layout()
plt.show()
```



as calinski score is high for no of triangles in network to onion layers, i am trying to check the score for no of common connections and kemeny constant (time to spread in a network)