Online Social Network Analysis - Final Project (CS 579)

Facebook Social Network Dataset - Analysis

Final Project

By

Shallum Israel Maryapanor (A20547274)
Sanjana Rayarala (A20548132)

Illinois Institute of Technology
Computer Science Department

DATE: 05/1/2024

Introduction

In the digital era, social networks like Facebook play a pivotal role in shaping human interactions and societal dynamics. The objective of this project is to delve into the architecture and characteristics of the Facebook social network using a dataset that encapsulates user interactions and connections. By constructing a network where nodes represent users and edges signify friendships, a variety of network science techniques are applied to uncover the underlying dynamics of the network. Our approach includes constructing a network graph of users and their connections, identifying key communities through community detection algorithms, and analyzing node influence using centrality measures. Additionally, we will apply link analysis methods like PageRank and HITS to further understand user roles in information dissemination. By visualizing these networks and simulating various scenarios, we aim to gain insights into the network's robustness and the implications of structural changes. This comprehensive analysis will help uncover the underlying patterns of social interactions on Facebook, offering valuable perspectives for targeted applications and enhancing our understanding of social network dynamics.

Data Description

The dataset used in this study, titled "Social circles: Facebook," was obtained from the Stanford Large Network Dataset Collection (SNAP) (https://snap.stanford.edu/data/egonets-Facebook.html). This dataset encompasses 'circles' or 'friends lists' derived from Facebook, collected through a Facebook app from survey participants. It is designed to facilitate social network analysis by providing a detailed yet anonymized snapshot of user interactions and network structure.

Dataset Composition:

The dataset has multiple edges, circles, and ego feat, feat, and feat names files where each file has information from different perspectives.
The edges files represent friendships or connections between users, forming the network's backbone. In specific they describe the edges between nodes, where each edge represents a friendship link between two users. Each line in the file contains a pair of user IDs, indicating a direct connection. For instance, "236 186"

signifies a friendship between the users with IDs 236 and 186. The edges are fundamental for constructing the graph structure of the network, enabling analysis on connectivity and network flow.
The .circles files indicate circles which are essentially clusters or groups indicating a user's various friend lists. These files enumerate the social circles (or friend lists) for each ego user. Each line represents a circle and lists the nodes (user IDs) included in that circle. For example, "circle0 71 215 54..." means that users identified by IDs 71, 215, 54, etc., are part of "circle0" for this particular ego. This information is critical for community detection algorithms, which can analyze how users group together within the network.
The .egofeat files shows the ego networks where each 'ego' network is centered around a single 'ego' user and includes all direct connections (friends), along with the connections those friends have to one another. These files contain feature vectors for the ego users themselves. Each row represents an ego user's features, encoded in a binary format. Each feature (e.g., "0 0 1 0...") corresponds to whether the ego user possesses certain attributes (anonymized), as indicated by a 1 (present) or 0
(absent). This data helps in profiling and understanding the characteristics of central users in each ego network.
The .feat files are similar to the .egofeat files, these contain feature vectors for all nodes in the ego network (not just the ego user). Each line corresponds to a user and their attributes, formatted in the same binary encoding. This comprehensive attribute data is useful for examining how features distribute across the network and can be used to predict node behavior or preferences.
Lastly, .featnames files describe the anonymized features, aiding in understanding the types of data collected without revealing specific, sensitive details. These files provide a mapping of indices in the feature vectors to the actual features, although the real meanings are anonymized. Each line maps a feature index to a description (e.g., "birthday; anonymized feature 0"). This mapping is crucial for interpreting the binary feature vectors in .egofeat and .feat files, allowing researchers to identify which types of attributes are included in the analysis even though the specific details remain obscured.

The selection of this dataset was driven by its comprehensive and pre-structured nature, ideal for in-depth network analysis. It provides a robust framework for understanding social dynamics through network science techniques such as community detection and centrality analysis. The anonymization of data ensures privacy compliance while retaining the dataset's utility for scientific research. By leveraging such a dataset, this project aims to uncover insights into the formation of social circles, influence patterns, and the overall network structure within the context of Facebook. The analysis could potentially reveal significant patterns relevant to social behaviors, information dissemination, and community formation on digital platforms.

Project process

Here's a detailed description of the project process, including the steps taken, challenges faced, and how they were dealt with:

1. <u>Loading and Building the Network:</u>

A base file was created called “network.py” which reads all the files from the dataset and works its way to build a network. This file has nothing to display/output, but is imported by every other file in the project, be it for visualization or for analysis. One difficulty faced while building the network was to ensure proper data parsing and to handle potential errors in the data format, and it was achieved by implementing error handling.

<u>network.py:</u>

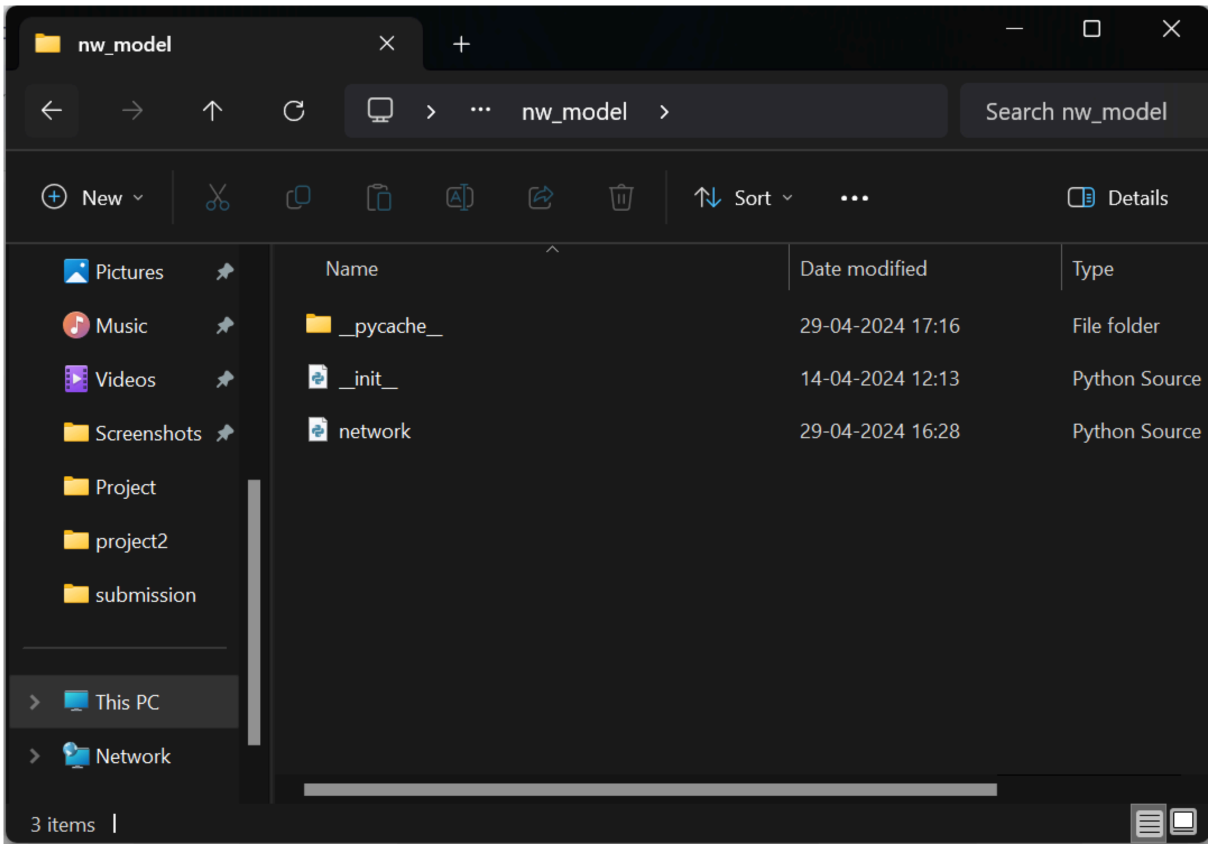
```python
import networkx as nx
import os
import matplotlib.pyplot as plt


def load_edges(file_path, graph):
    #print(f"Loading edges from {file_path}")
    with open(file_path, 'r') as file:
        for line in file:
            parts = line.strip().split()
            if len(parts) != 2:
                continue  # Skip any malformed lines

            node1, node2 = map(int, parts)
            graph.add_edge(node1, node2)  # Adds the edge to the graph



def load_features(file_path, graph, is_ego=False):
    feature_names = graph.graph.get('feature_names', {})
    #print(f"Loading features from {file_path}, is_ego={is_ego}")

    with open(file_path, 'r') as file:
        for line in file:
            parts = line.strip().split()
            if len(parts) < 2:  # Ensures there's at least one feature plus a node ID
                print("Warning: Malformed line skipped:", line)
                continue

            node_id = int(parts[0])
            features = list(map(int, parts[1:]))

            # Ensures every node mentioned has a node entry in the graph
            if node_id not in graph:
                graph.add_node(node_id)

            # Assigns features to the node
                        # Maps feature indices to their descriptive names or defaults to
'feature_index'
                named_features = {feature_names.get(i, f'feature_{i}'): feat for i, feat in
enumerate(features)}

            if is_ego:
                    # For ego features, combine with existing using max to preserve the
most significant features
                current_features = graph.nodes[node_id].get('features', {})
                                updated_features = {k: max(current_features.get(k, 0),
named_features[k]) for k in named_features}
                graph.nodes[node_id]['features'] = updated_features
            else:
                graph.nodes[node_id]['features'] = named_features
    '''
    # Debugging output to check the first few nodes
    for node_id in list(graph.nodes)[:5]:
        if 'features' in graph.nodes[node_id]:
            print(f"Features for node {node_id}: {graph.nodes[node_id]['features']}")
        else:
            print(f"No features found for node {node_id}")
    '''


def extract_ego_node_id(filename):
    # The ego node ID is assumed to be the part before the first dot ('.') in the
filename
    ego_node_id = filename.split('.')[0]
    return int(ego_node_id)  # Convert to integer if the ID is expected to be a
numeric value

def load_egofeat(file_path, graph, ego_node_id):
    feature_names = graph.graph.get('feature_names', {})
    #print(f"Loading ego features from {file_path} for node {ego_node_id}")

    with open(file_path, 'r') as file:
        line = file.readline().strip()  # Assuming only one line for ego features
        features = list(map(int, line.split()))

        if ego_node_id not in graph:
            graph.add_node(ego_node_id)  # Ensure the ego node exists

        # Map feature indices to their descriptive names or default to
'feature_index'
        named_features = {feature_names.get(i, f'feature_{i}'): feat for i, feat in
enumerate(features)}

        # For ego features, simply assign them as they are typically not merged with
other features
        graph.nodes[ego_node_id]['features'] = named_features

        #print(f"Ego Features for node {ego_node_id}:
{graph.nodes[ego_node_id]['features']}")



def load_featnames(file_path, graph):
    feature_names = {}
    with open(file_path, 'r') as file:
        for line in file:
            index, feature_description = line.strip().split(' ', 1)
            feature_names[int(index)] = feature_description

    # Attach feature names to the graph as an attribute (useful for referencing
later)
    graph.graph['feature_names'] = feature_names
    #print(f"Loaded feature names: {feature_names}")
```

```python
def load_circles(file_path, graph):
    node_to_circles = {}  # Maps node to the list of circles it belongs to
    circle_to_nodes = {}  # Maps circle name to the set of nodes it includes

    #print(f"Loading circles from {file_path}")
    with open(file_path, 'r') as file:
        for line in file:
            parts = line.strip().split()
            circle_name = parts[0]  # The first part is the circle name
            circle_members = set(map(int, parts[1:]))  # The rest are node IDs

            # If circle is not present in circle_to_nodes, initialize it
            circle_to_nodes[circle_name] = circle_to_nodes.get(circle_name, set())

            # Add all members to the circle
            circle_to_nodes[circle_name].update(circle_members)

            for node_id in circle_members:
                # Add node to graph if not present
                if node_id not in graph:
                    graph.add_node(node_id)

                # Add circle to the node's list of circles
                node_to_circles.setdefault(node_id, set()).add(circle_name)

                # Update graph's node attribute
                graph.nodes[node_id]['circles'] = list(node_to_circles[node_id])

    return node_to_circles, circle_to_nodes


def build_network():
    # Initialize an empty Graph
    G = nx.Graph()
    # Path to the directory with all the data files
    base_path = 'facebook'

    # List all files in the directory
    files = os.listdir(base_path)
    for file in files:
        file_path = os.path.join(base_path, file)
        if file.endswith('.edges'):
            load_edges(file_path, G)
        elif file.endswith('.feat'):
            load_features(file_path, G)
        elif file.endswith('.egofeat'):
               ego_node_id = extract_ego_node_id(file)  # Extract the ego node ID from
the filename
                  load_egofeat(file_path, G, ego_node_id)  # Pass the extracted ID to the
function
        elif file.endswith('.circles'):
            node_to_circles, circle_to_nodes= load_circles(file_path, G)
        elif file.endswith('.featnames'):
            load_featnames(file_path, G)


    print("Total number of nodes:", G.number_of_nodes())
    print("Total number of edges:", G.number_of_edges())
    return G
```

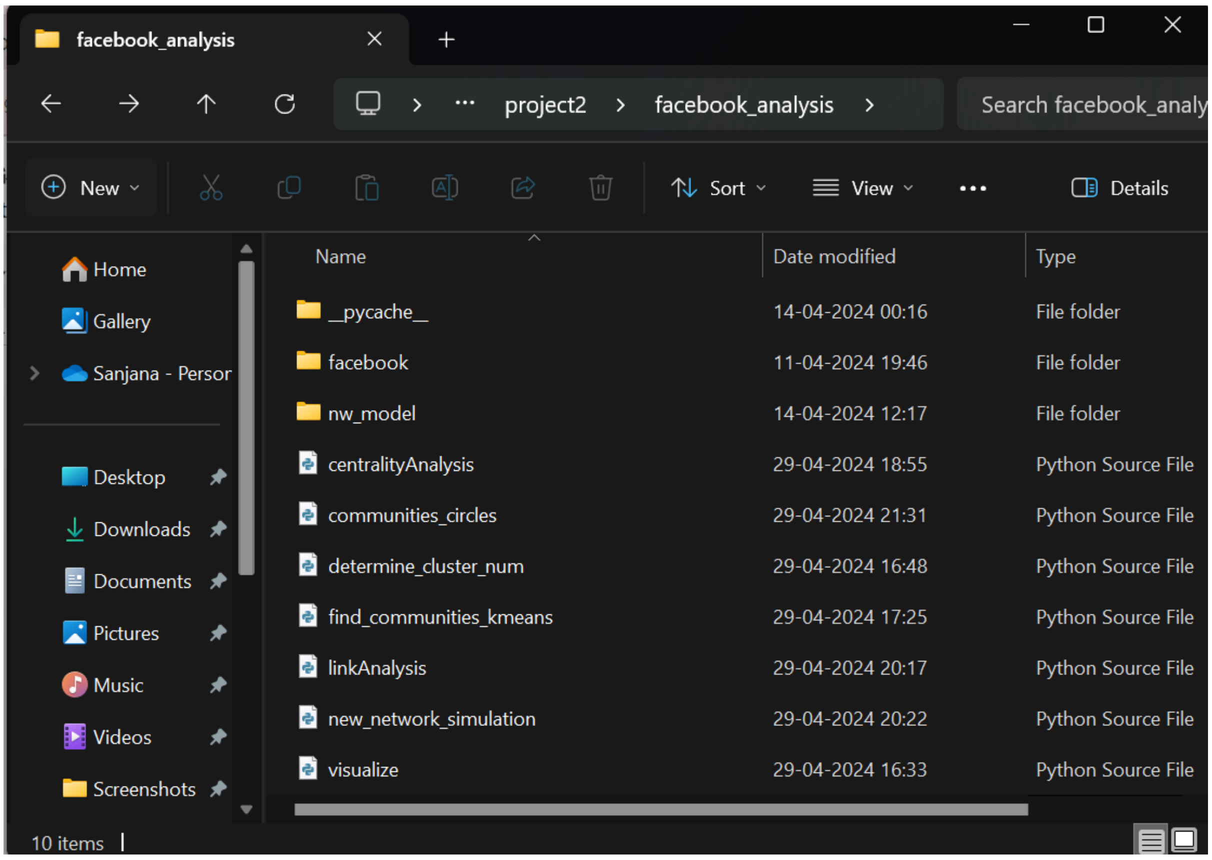
network.py is in “nw_model” which is a sub-folder of a folder “facebook_analysis” which consists of the rest of other files in addition to the "facebook" dataset folder.

<u>Files:</u>

- visualize.py
- determine_cluster_num.py
- find_communities_kmeans.py
- centralityAnalysis.py
- linkAnalysis.py
- new_network_simulation.py



Name	Date modified	Type
pycache	29-04-2024 17:16	File folder
init	14-04-2024 12:13	Python Source
network	29-04-2024 16:28	Python Source



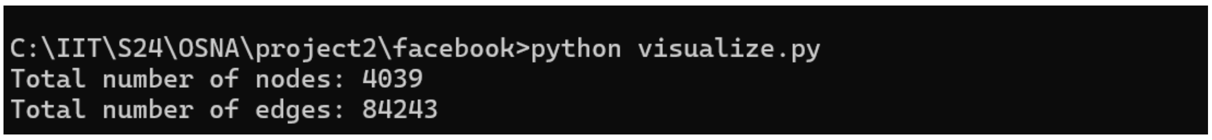
Name	Date modified	Type
pycache	14-04-2024 00:16	File folder
facebook	11-04-2024 19:46	File folder
nw_model	14-04-2024 12:17	File folder
centralityAnalysis	29-04-2024 18:55	Python Source File
communities_circles	29-04-2024 21:31	Python Source File
determine_cluster_num	29-04-2024 16:48	Python Source File
find_communities_kmeans	29-04-2024 17:25	Python Source File
linkAnalysis	29-04-2024 20:17	Python Source File
new_network_simulation	29-04-2024 20:22	Python Source File
visualize	29-04-2024 16:33	Python Source File

2. Visualization of the network

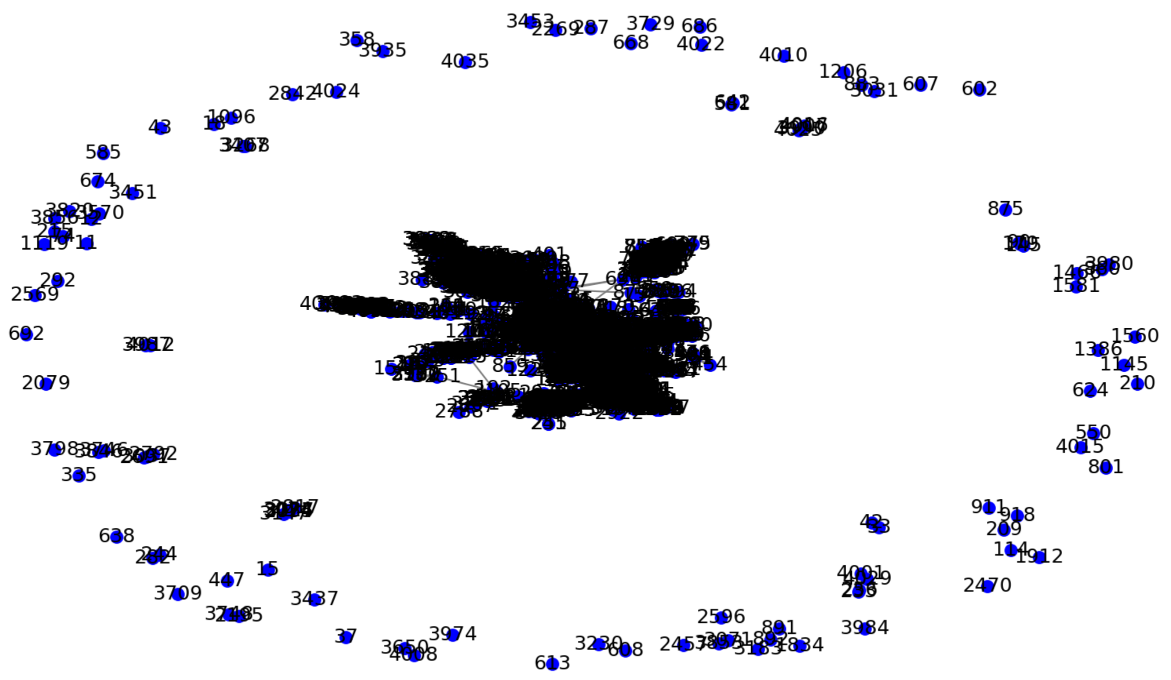
The visualize.py script is designed to visualize a network graph using the NetworkX library, supported by matplotlib for rendering. From the network.py file, build_network() function is imported whose task is to build the network and return the Graph. The returned Graph is then used by the plot() function to visualize the network.

visualize.py:

```
import matplotlib.pyplot as plt
import networkx as nx
from nw_model.network import build_network

def plot(G):
    plt.figure(figsize=(6, 6))
    pos = nx.spring_layout(G, scale=2)
    nx.draw(G, pos, with_labels=True, node_size=50, node_color='blue',
edge_color='gray')
    plt.show()

G= build_network()
plot(G)
```



```
C:\IIT\S24\OSNA\project2\facebook>python visualize.py
Total number of nodes: 4039
Total number of edges: 84243
```



3. Community Detection:

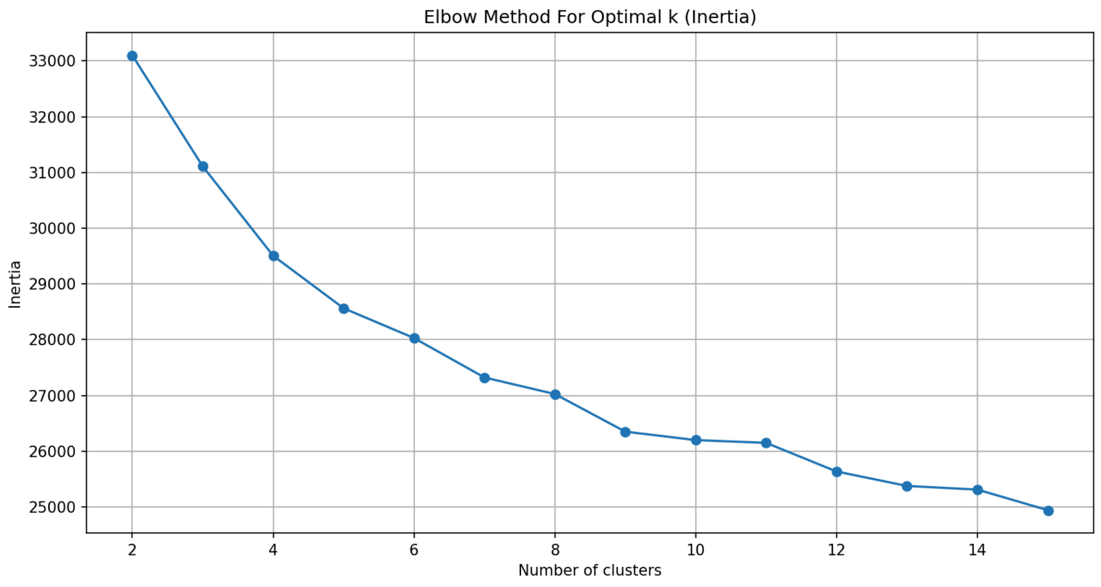
To find the communities two files were involved: determine_cluster_num.py and find_communities_kmeans.py. The determine_cluster_num.py results in a graph through which we'll be able to consider the number of clusters to pass it as a parameter for k-means algorithm which is implemented in find_communities_kmeans.py.

determine_cluster_num.py:

```
#TO find suitable no. of clusters.
import networkx as nx
import os
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import numpy as np
from nw_model.network import build_network

def optimal_clusters(graph, max_clusters=10):
    # Gather feature lists
    feature_list = [list(data['features'].values()) for node, data in
graph.nodes(data=True) if 'features' in data]
```

```python
    #maximum length of any feature list
    max_length = max(len(features) for features in feature_list)

    #Ensure all features are of the same length
    padded_features = [features + [0] * (max_length - len(features)) for features in
feature_list]

    features = np.array(padded_features)

    if features.size == 0:
        print("No features available for clustering.")
        return

    inertias = []

    for k in range(2, max_clusters + 1):
        kmeans = KMeans(n_clusters=k, random_state=42)
        kmeans.fit(features)
        inertias.append(kmeans.inertia_)

    #Plotting the elbow curve for inertia
    plt.figure(figsize=(12, 6))
    plt.plot(range(2, max_clusters + 1), inertias, marker='o')
    plt.title('Elbow Method For Optimal k (Inertia)')
    plt.xlabel('Number of clusters')
    plt.ylabel('Inertia')
    plt.grid(True)
    plt.show()


G=build_network()
optimal_clusters(G, max_clusters=15) #plot resulted in 6
```

<u>Output:</u>



The plot is generated on the basis of "Elbow Method". It's a graph used to find the optimal number of clusters for k-means clustering. The x-axis shows the number of clusters, and the y-axis shows the inertia for each k-means solution with a different number of clusters. Inertia is a measure of how internally coherent clusters are. It's calculated as the sum of the squared distances between each data point and the centroid of its assigned cluster. Lower inertia values are better, indicating that the data points in each cluster are closer to their centroids. To determine the optimal number of clusters, we look for the "elbow" in the graph; this is a point where the inertia begins to decrease more slowly. It indicates that adding more clusters doesn't provide as much benefit in reducing the inertia.

In the resulted graph, the inertia rapidly decreases as the number of clusters increases from 2 to around 6, and after that, the rate of decrease slows down. The "elbow" seems to be around 6 clusters. This is a sign that increasing the number of clusters beyond 6 doesn't result in significantly better modeling of the data. Therefore, based on this method, we consider choosing 6 as the optimal number of clusters for the k-means clustering algorithm.

find_communities_kmeans.py script uses network analysis and machine learning techniques to detect and visualize communities within a network graph constructed from data loaded by the `build_network()` function from the `nw_model.network` module. Initially, the script prepares feature data for each node, ensuring uniformity by padding shorter feature lists. It then applies the

KMeans clustering algorithm, defined to identify a elbow specified number of communities by setting k=6, as identified from elbow plot, assigning each node to a community based on its features. Post-clustering, the script computes average values of the most prominent features for each community, enhancing interpretability. For visualization, it identifies unique communities, extracts corresponding subgraphs, and visualizes each community separately using NetworkX and matplotlib. The nodes in each community are displayed with annotations that highlight the top three average features, providing insights into the characteristics that define each community. This allows for an in-depth exploration of the network’s structure, facilitating a better understanding of how nodes group based on shared attributes, which is critical for tasks such as targeted analysis or network optimization.

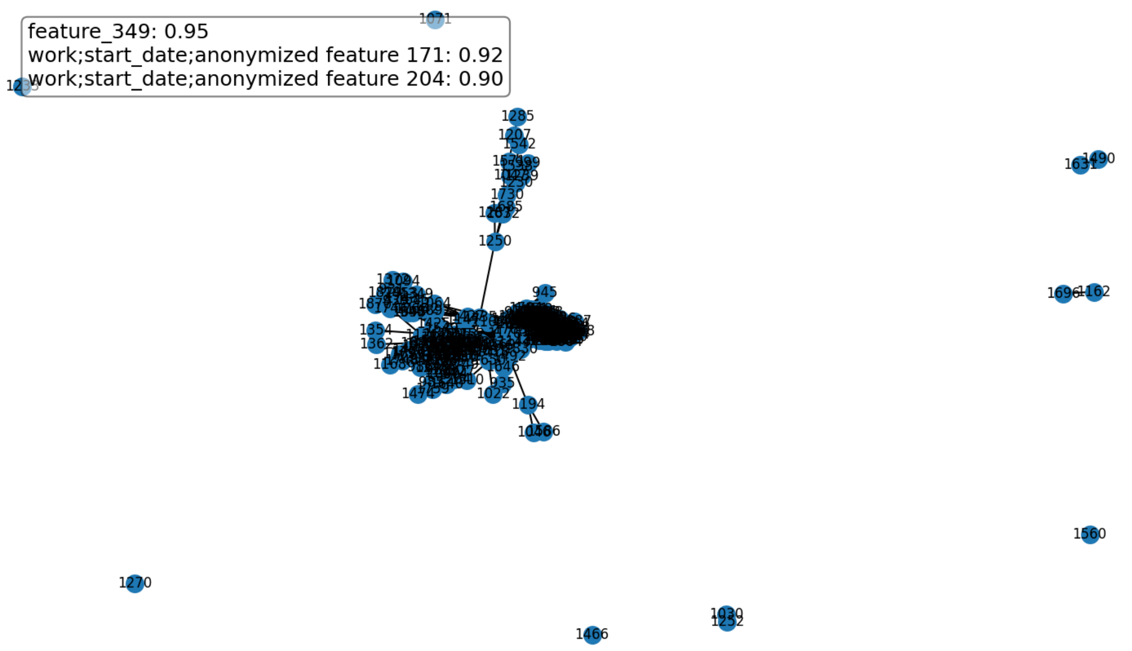
find_communities_kmeans.py:

```
import networkx as nx
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import numpy as np
from nw_model.network import build_network

def detect_communities_kmeans(graph, num_clusters=3):
    #Ensure nodes have features and that they are all of same length
    feature_list = []
    nodes = []
    for node, data in graph.nodes(data=True):
        if 'features' in data:
            nodes.append(node)
            feature_list.append(list(data['features'].values()))

    #Find the max length of any feature list
    max_length = max(len(features) for features in feature_list)
    padded_features = [features + [0] * (max_length - len(features)) for features in
feature_list]

    features = np.array(padded_features)

    #fit KMeans
    kmeans = KMeans(n_clusters=num_clusters, random_state=42)
    kmeans.fit(features)
    labels = kmeans.labels_

    # Compute the most common features for each community
    community_features = {}
    for node, label in zip(nodes, labels):
        graph.nodes[node]['community'] = label
        features = graph.nodes[node]['features']
        if label not in community_features:
            community_features[label] = {}
        for feature, value in features.items():
            if feature not in community_features[label]:
                community_features[label][feature] = []
            community_features[label][feature].append(value)

    # For each community, calculate the average of each feature
    for label, features in community_features.items():
        community_features[label] = {feature: np.mean(values) for feature, values in
features.items()}

    return community_features

def annotate_subgraphs_with_features(graph, community_features,
layout=nx.spring_layout):
    #Identify unique communities
    communities = set(nx.get_node_attributes(graph, 'community').values())
    for community in communities:
        #Extract subgraph
        nodes_in_community = [node for node in graph if
graph.nodes[node]['community'] == community]
        subgraph = graph.subgraph(nodes_in_community)

        #Plot  subgraph
        plt.figure(figsize=(10, 10))
        pos = layout(subgraph)
        nx.draw(subgraph, pos, with_labels=True, node_size=100, font_size=8)

        #Annotate with the most common features in this community
        common_features = community_features[community]
        most_common_features = sorted(common_features, 
key=common_features.get, reverse=True)[:3]  # top 3 features
        text_str = '\n'.join(f'{feature}: {common_features[feature]:.2f}' for feature in 
most_common_features)
        plt.text(0.05, 0.95, text_str, transform=plt.gca().transAxes, fontsize=12,
                 verticalalignment='top', bbox=dict(boxstyle='round', facecolor='white', 
alpha=0.5))

        plt.title(f'Community {community} - Top 3 Features')
        plt.show()


G=build_network()
community_features= detect_communities_kmeans(G, num_clusters=6)
annotate_subgraphs_with_features(G, community_features)
```

The code outputs 6 different graphs, representing different communities within a larger social network. In these visualizations, nodes correspond to individuals or entities, while the edges illustrate relationships or connections between them. Each image also features a floating text box detailing the most significant features common to nodes within that particular community.

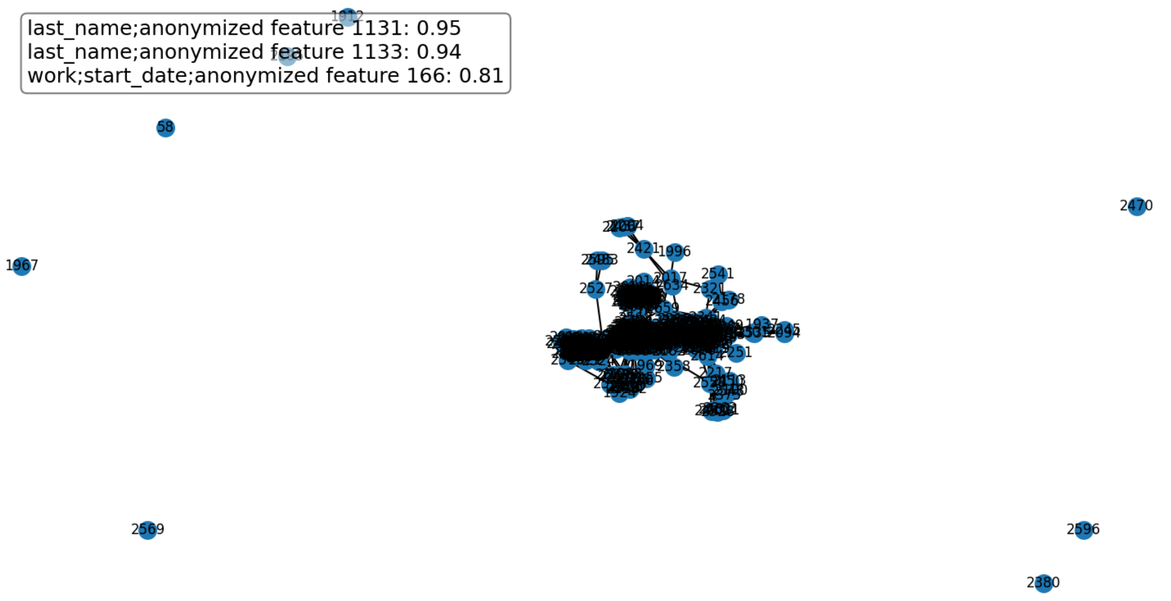
The features within the text boxes, accompanied by numerical values, are the results of some form of attribute analysis(the top three features of the nodes within that community and mean of each feature). Given the anonymized labels (e.g., "feature 349," "anonymized feature 171"), specific attributes (like occupation, location, or other demographic or behavioral traits) have been obscured for privacy.

Community 1:



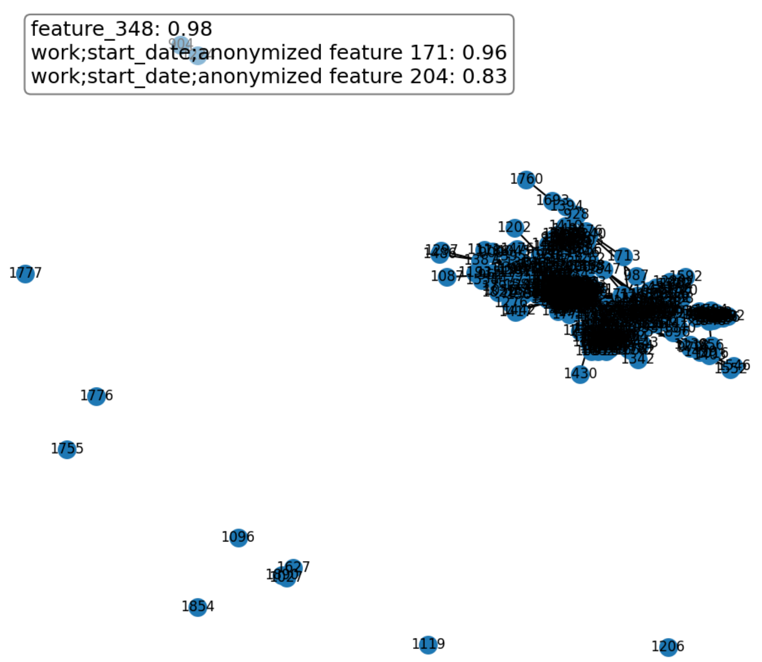
This community might be characterized by commonalities in work history or professional background, suggested by the significant presence of work-related features. The network layout, with one central dense cluster and several outlying nodes, may indicate a core group with a few loosely connected members.

Community 2:



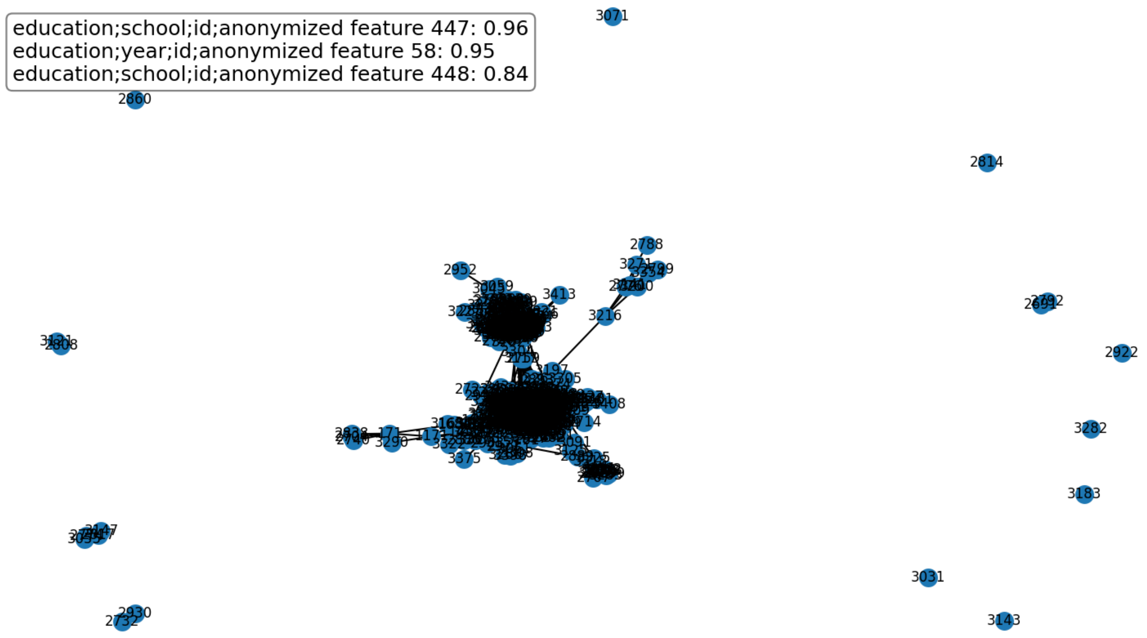
This community might be similar to the first one, with a potential focus on professional attributes or other common characteristics that can be inferred from the work start date. It's a smaller and more cohesive cluster, possibly indicating a tighter-knit group or a more niche community.

Community 3:


feature_348: 0.98
work;start_date;anonymized feature 171: 0.96
work;start_date;anonymized feature 204: 0.83
918
1145
1760
1202
1713
1777
1087
1430
1776
1755
1701
1834
1096
1854
1119
1206

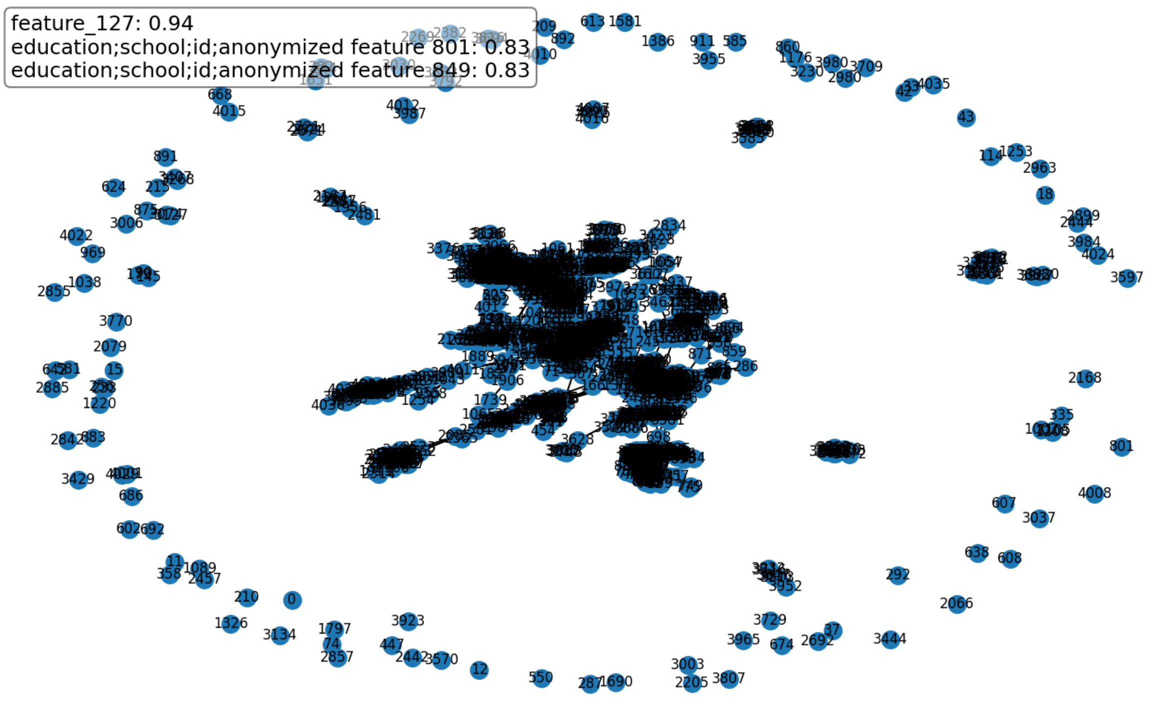
This community shares features with the first one but seems to be more dispersed, which might suggest a broader or more diverse set of connections among the members based on their work start dates and other common features.

Community 4:


education;school;id;anonymized feature 447: 0.96
education;year;id;anonymized feature 58: 0.95
education;school;id;anonymized feature 448: 0.84

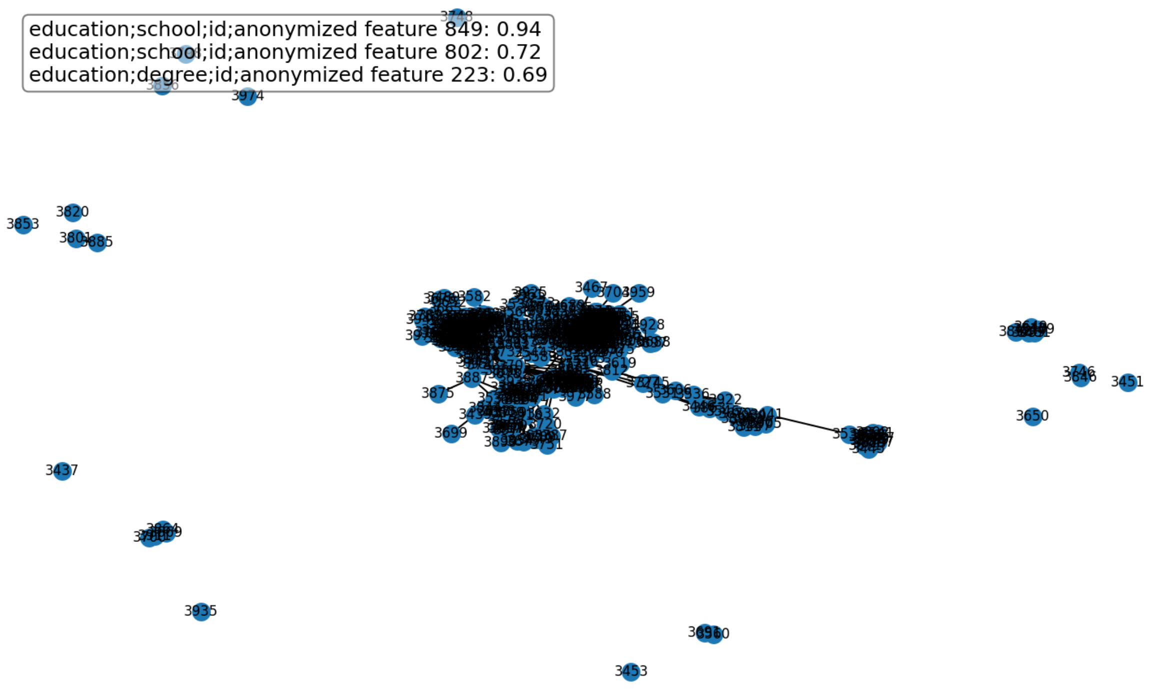
These features could relate to educational background, suggesting this community is formed around common educational experiences or affiliations. The cluster is centralized, with distinct satellite nodes, which could indicate subgroups or peripheral members with less interaction with the core.

Community 5:


feature_127: 0.94
education;school;id;anonymized feature 801: 0.83
education;school;id;anonymized feature 849: 0.83

This community is highly interconnected with dense clusters, possibly indicative of a group with strong common educational or professional backgrounds. The presence of outlying nodes could signify individuals with connections to the core group but who are less central to the community dynamics.

Community 6:


education;school;id;anonymized feature 849: 0.94
education;school;id;anonymized feature 802: 0.72
education;degree;id;anonymized feature 223: 0.69

Here, the network structure appears to have a 'hub-and-spoke' layout with one or more central nodes that are highly connected and many peripheral nodes with fewer connections. The central nodes could be influential individuals within the community, possibly with significant educational or professional status.

<u>4. Centrality Analysis of the network:</u>

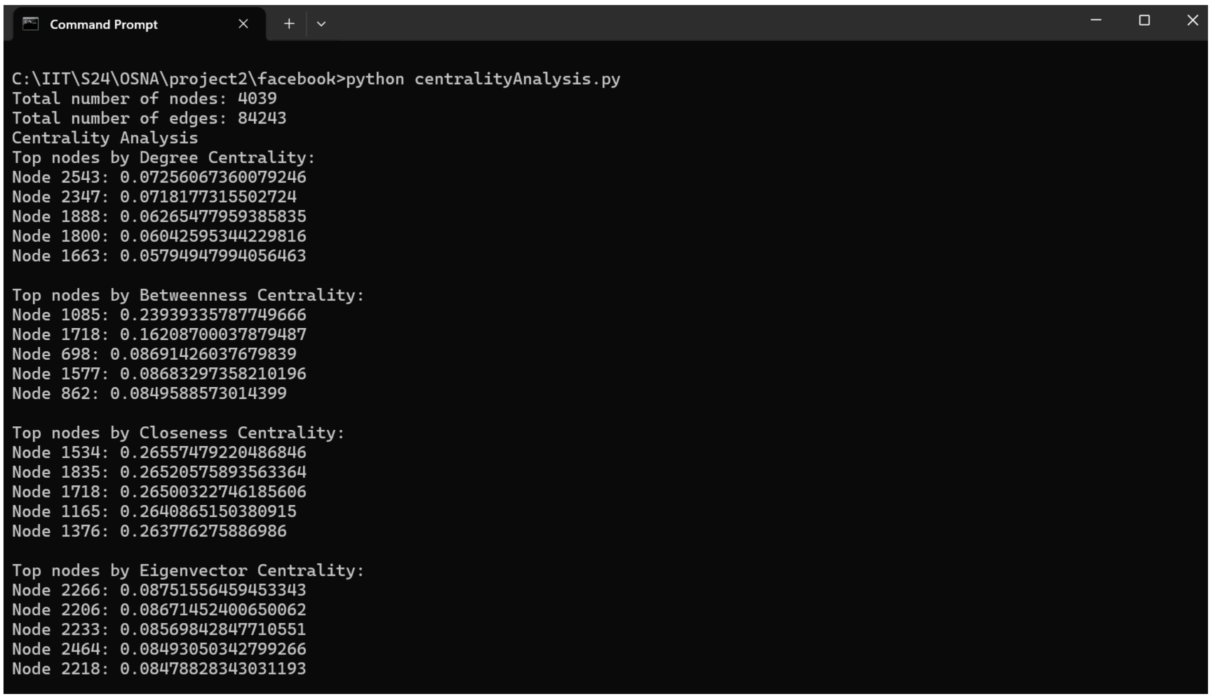
The centralityAnalysis.py script conducts a centrality analysis on the network graph. Centrality measures are crucial in network analysis as they help identify the most important nodes within a network based on different criteria. The code computes four types of centrality metrics: degree, betweenness, closeness, and eigenvector centrality. Each type of centrality provides different insights:
Degree centrality measures the number of connections a node has, identifying nodes that are highly connected and potentially influential in spreading information through the network.

Betweenness centrality assesses the frequency at which a node appears on the shortest paths between other nodes, highlighting nodes that act as bridges within the network.
Closeness centrality gauges how close a node is to all other nodes in the network, offering a view of how quickly a node can interact with others.
Eigenvector centrality takes into account the centrality of a node's neighbors, identifying nodes that are connected to other highly connected nodes, suggesting influence within highly interconnected parts of the network.
This analysis is helpful for understanding the roles of specific nodes in a network, which can be particularly relevant in applications like social network analysis, identifying key influencers, understanding the robustness of networks to the removal of certain nodes, and optimizing communication or flow within a network. The output provides a quick reference to those nodes that are central according to different measures of importance or strategic positioning within the graph.

centralityAnalysis.py:
```python
import networkx as nx
from nw_model.network import build_network

def centrality_analysis(graph, top_n=5):
    print("Centrality Analysis")

    #Calculating centralities
    degree = nx.degree_centrality(graph)
    betweenness = nx.betweenness_centrality(graph)
    closeness = nx.closeness_centrality(graph)
    try:
        eigenvector_centrality = nx.eigenvector_centrality(G, max_iter=1000)
    except nx.PowerIterationFailedConvergence:
        print("Eigenvector centrality failed to converge, increasing max_iter.")
        eigenvector_centrality = nx.eigenvector_centrality(G, max_iter=5000)  #Try
larger number of iterations

    #Get top n nodes i.e. top 5 nodes, for each centrality measure
    top_degree = sorted(degree.items(), key=lambda item: item[1],
reverse=True)[:top_n]
    top_betweenness = sorted(betweenness.items(), key=lambda item: item[1],
reverse=True)[:top_n]
```

```
    top_closeness = sorted(closeness.items(), key=lambda item: item[1],
reverse=True)[:top_n]
    top_eigenvector = sorted(eigenvector_centrality.items(), key=lambda item:
item[1], reverse=True)[:top_n]

    print("Top nodes by Degree Centrality:")
    for node, cent in top_degree:
        print(f"Node {node}: {cent}")

    print("\nTop nodes by Betweenness Centrality:")
    for node, cent in top_betweenness:
        print(f"Node {node}: {cent}")

    print("\nTop nodes by Closeness Centrality:")
    for node, cent in top_closeness:
        print(f"Node {node}: {cent}")

    print("\nTop nodes by Eigenvector Centrality:")
    for node, cent in top_eigenvector:
        print(f"Node {node}: {cent}")


G=build_network()
centrality_analysis(G)
```

Output:



```
C:\IIT\S24\OSNA\project2\facebook>python centralityAnalysis.py
Total number of nodes: 4039
Total number of edges: 84243
Centrality Analysis
Top nodes by Degree Centrality:
Node 2543: 0.07256067360079246
Node 2347: 0.0718177315502724
Node 1888: 0.06265477959385835
Node 1800: 0.06042595344229816
Node 1663: 0.05794947994056463

Top nodes by Betweenness Centrality:
Node 1085: 0.23939335787749666
Node 1718: 0.16208700037879487
Node 698: 0.08691426037679839
Node 1577: 0.08683297358210196
Node 862: 0.0849588573014399

Top nodes by Closeness Centrality:
Node 1534: 0.26557479220486846
Node 1835: 0.26520575893563364
Node 1718: 0.26500322746185606
Node 1165: 0.2640865150380915
Node 1376: 0.263776275886986

Top nodes by Eigenvector Centrality:
Node 2266: 0.08751556459453343
Node 2206: 0.08671452400650062
Node 2233: 0.08569842847710551
Node 2464: 0.08493050342799266
Node 2218: 0.08478828343031193
```

The analysis based on the given output offers valuable insights into the structure and influence patterns within the network:
Nodes 2543, 2347, 1888, 1800, and 1663 have the highest number of direct connections- Degree Centrality. This suggests they are prominent members of the network, perhaps acting as the most active participants or popular individuals/entities. They might be seen as influential due to their extensive direct reach.

Node 1085 stands out with the highest betweenness centrality, indicating it acts as a significant intermediary in the network. This node could be crucial for the flow of information or resources, as it connects different parts of the network. The removal of such a node might significantly disrupt communication within the network. Nodes 1718, 698, 1577, and 862 also play important roles as connectors or bridges between different community clusters.

Node 1534 is the most central in terms of closeness, suggesting it has the shortest average path to all other nodes in the network, positioning it well for efficient communication or rapid spread of information. Nodes 1835, 1718, 1165, and 1376 also exhibit high closeness centrality, indicating they are well-placed to quickly interact with other nodes across the entire network.

Node 2266, followed by nodes 2206, 2233, 2464, and 2218, have high eigenvector centrality scores. This implies these nodes are not only well-connected but are also linked to other well-connected nodes. They could be considered influential within influential circles or hubs within core clusters of the network.

From this analysis, we can deduce several points about the network's social structure:
- Influential Nodes: Node 1718 is notable as it appears in the top five for both betweenness and closeness centrality, marking it as both a connector and a rapid disseminator. This dual significance suggests that it holds an important strategic position within the network.

- Central vs. Peripheral Influence: Nodes with high degree centrality might not necessarily be the most crucial for the cohesion of the network, as indicated by the betweenness centrality results. For example, Node 2543 has the highest degree centrality but does not appear in the top five for betweenness centrality, suggesting its direct connections might not significantly impact the overall connectivity of the network.

- Network Resilience and Vulnerability: Nodes with high betweenness centrality, like Node 1085, might indicate points of vulnerability; their removal could fragment the network. Conversely, nodes with high closeness centrality could be targeted for information dissemination to ensure rapid coverage of the network.

- Influence Through Association: The eigenvector centrality results suggest that nodes like 2266 are influential within the network not just because of their direct connections but also because they are connected to other influential nodes.

Link Analysis of the network:

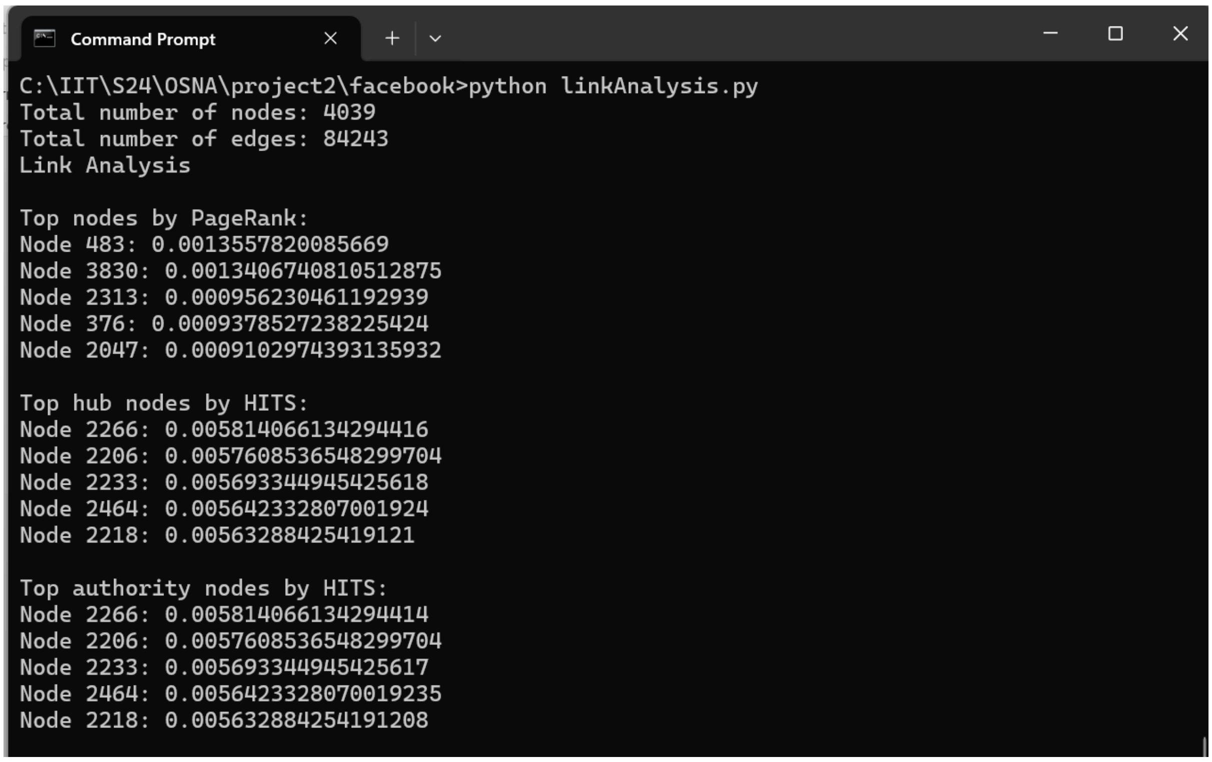
linkAnalysis.py script conducts link analysis on the network. The link_analysis() function initiates the analysis by first calculating the PageRank of each node. PageRank is an algorithm used to measure the relative importance or influence of each node within the network. The importance of a node is determined by the quantity and the quality of the edges linking to it. In other words, a node receives a higher PageRank if it is connected to many nodes or if it is connected to other nodes that are themselves important.

After PageRank, the script computes the HITS scores, distinguishing between 'hub' and 'authority' scores. Hubs are nodes that point to several other nodes, indicating they are good at disseminating information. Authorities, on the other hand, are nodes that are pointed at by many hubs, signifying they are considered reliable or valuable by other nodes. The HITS algorithm thus provides a way of identifying not just who is important, but also the role they play within the network's information ecosystem.
Then we sort the nodes based on their PageRank, hub, and authority scores and prints out the top five nodes for each category. This output helps identify which nodes stand out according to each algorithm, offering a multifaceted perspective on influence within the network. For instance, a node might not have the highest PageRank but could be a critical hub, playing a significant role in connecting different parts of the network.

linkAnalysis.py:

```
import networkx as nx
from nw_model.network import build_network

def link_analysis(graph, top_n=5):
    print("Link Analysis")

    #calculate PageRank
    pagerank_scores = nx.pagerank(graph)

    #Calculate HITS scores
    hits_scores = nx.hits(graph, max_iter=100)  # max_iter may need to be
adjusted based on convergence

    #Get top n, i.e. 5 nodes for PageRank and HITS measures
    top_pagerank = sorted(pagerank_scores.items(), key=lambda item: item[1],
reverse=True)[:top_n]
    top_hubs = sorted(hits_scores[0].items(), key=lambda item: item[1],
reverse=True)[:top_n]
    top_authorities = sorted(hits_scores[1].items(), key=lambda item: item[1],
reverse=True)[:top_n]

    print("\nTop nodes by PageRank:")
    for node, score in top_pagerank:
        print(f"Node {node}: {score}")
    print("\nTop hub nodes by HITS:")
    for node, score in top_hubs:
        print(f"Node {node}: {score}")

    print("\nTop authority nodes by HITS:")
    for node, score in top_authorities:
        print(f"Node {node}: {score}")



G= build_network()
link_analysis(G)
```

<u>Output:</u>



```
C:\IIT\S24\OSNA\project2\facebook>python linkAnalysis.py
Total number of nodes: 4039
Total number of edges: 84243
Link Analysis

Top nodes by PageRank:
Node 483: 0.0013557820085669
Node 3830: 0.0013406740810512875
Node 2313: 0.000956230461192939
Node 376: 0.0009378527238225424
Node 2047: 0.0009102974393135932

Top hub nodes by HITS:
Node 2266: 0.005814066134294416
Node 2206: 0.0057608536548299704
Node 2233: 0.005693344945425618
Node 2464: 0.005642332807001924
Node 2218: 0.00563288425419121

Top authority nodes by HITS:
Node 2266: 0.005814066134294414
Node 2206: 0.0057608536548299704
Node 2233: 0.005693344945425617
Node 2464: 0.0056423328070019235
Node 2218: 0.005632884254191208
```

PageRank Analysis: The PageRank values indicate that nodes 483, 3830, 2313, 376, and 2047 are the most prominent according to Google's PageRank algorithm. Although their scores might appear small, it is the relative difference between them that matters. A higher PageRank score implies that a node is more significant within the network, potentially acting as an important point of information relay or influence. In this context, node 483 has the highest rank, which suggests that it might be a very influential node in the network, receiving a significant amount of 'endorsement' through links from other nodes.

HITS Analysis: The HITS algorithm distinguishes between hubs and authorities. The top hub nodes—2266, 2206, 2233, 2464, and 2218—are nodes that link to many other nodes; they act as broadcasters of information. Their role within the network is to disseminate information to other parts of the graph effectively. These nodes might not be the most popular or the most authoritative, but they serve as the backbone for the flow of information.

Conversely, the top authority nodes—2266, 2206, 2233, 2464, and 2218—have the same IDs as the top hub nodes, indicating a strong correlation between hubs and authorities in this particular network. This often happens in networks where some nodes are central both as distributors and receivers of information. They are considered reliable sources and are frequently referenced by other hubs. This kind of correlation can signify a cohesive community where the flow of information is somewhat circular, or it could highlight a hierarchical structure where certain nodes are central to both disseminating and validating information.

Implications of the Analysis: The fact that the same nodes appear as both top hubs and top authorities could imply that this network has a few very central nodes that are crucial to its structure and function. They are not just passing along information; they are also the destination of many pathways within the network. This could be characteristic of a network with a strong leadership or central figures, such as in a corporate hierarchy or a tightly-knit online community.

5. Network Simulation and Analysis:

This code conducts a simulation to analyze how certain changes might affect the structure and properties of a large network.
It starts by creating a duplicate of the original graph using the copy() method. This allows the simulation to make changes without altering the original network structure, preserving it for comparison or further analysis.
The script then calculates betweenness centrality for each node to identify strategic or highly connected nodes in the network. The ten nodes with the highest betweenness centrality are then removed from the simulated graph. This step simulates a scenario where key individuals or nodes are removed from a network, which could mimic, for example, the deletion of key user accounts from a social network or the shutdown of critical network servers.

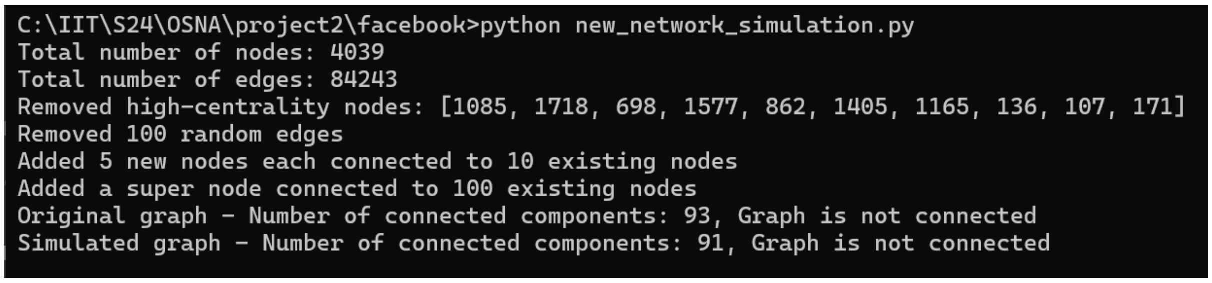
Next, the simulation randomly removes 100 edges from the network, provided that the graph has more than 100 edges. This models the potential impact of disruptions in relationships or communications channels within the network, such as breaking social ties or severing communication links.
Then we add five new nodes to the network, each connected to ten existing random nodes. This could represent new individuals joining a social network and starting to form connections or new websites linking to existing ones. The addition of new nodes with several connections may affect the network's connectivity and its ability to disseminate information.
A "super node" is then introduced, connected to 100 existing random nodes. The concept of a super node could apply to a highly influential social media influencer joining and connecting with many users or a major hub in a transportation network. This node's introduction is likely to have a significant impact on the network dynamics due to its many connections.
The analyze_and_plot() function is called to visualize and compare the original and simulated graphs side-by-side. It uses a spring layout to position the nodes based on their connections, aiming to spatially separate unconnected nodes while drawing connected nodes closer together. The original and simulated networks' visual comparison can provide intuitive insights into how the network's structure has evolved due to the simulation.
Finally, the print_network_stats function reports the number of connected components and, if the graph is fully connected, the average shortest path length for both the original and simulated graphs. This statistical data provides a quantitative measure of the network's connectivity and integration, indicating how easily information can flow through the network.
In summary, the code is a simulation tool to explore hypothetical scenarios and their impact on network structure.

new_network_simulation.py:

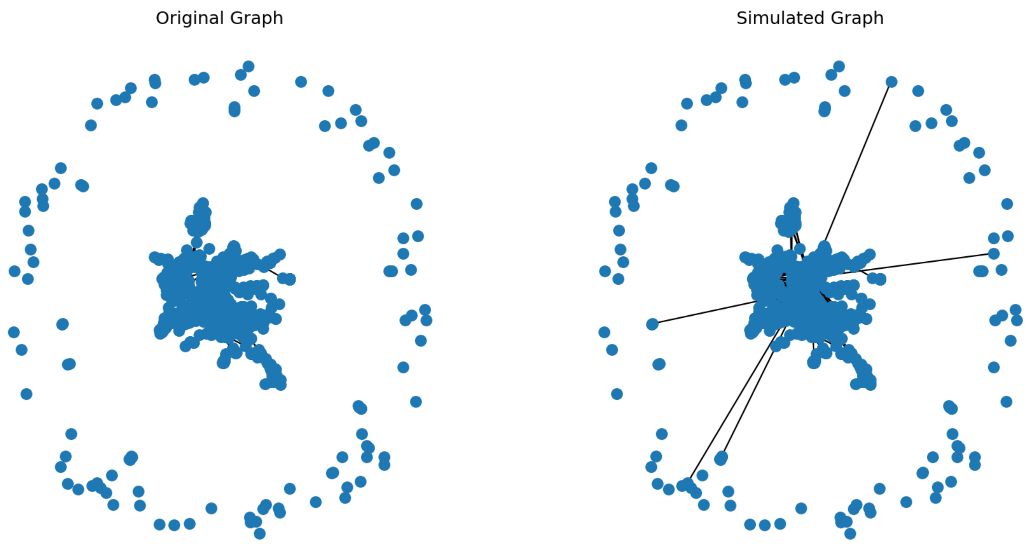
```python
import networkx as nx
import matplotlib.pyplot as plt
import random
from nw_model.network import build_network

def network_simulation_analysis_large(graph):
   #Create a copy of the graph to preserve the original
   sim_graph = graph.copy()

   #removing nodes with high betweenness centrality
```

```python
    centrality = nx.betweenness_centrality(graph)
    high_centrality_nodes = sorted(centrality, key=centrality.get,
reverse=True)[:10]
    sim_graph.remove_nodes_from(high_centrality_nodes)
    print(f"Removed high-centrality nodes: {high_centrality_nodes}")

    #Removing multiple random edges
    if len(sim_graph.edges()) > 100:
        edges_to_remove = random.sample(list(sim_graph.edges()), 100)
        sim_graph.remove_edges_from(edges_to_remove)
        print(f"Removed 100 random edges")

    #Adding multiple new nodes with multiple connections
    new_nodes = []
    for i in range(5):
        new_node = max(sim_graph.nodes()) + 1
        new_nodes.append(new_node)
        nodes_to_connect = random.sample(list(sim_graph.nodes()), min(10,
len(sim_graph.nodes())))
        sim_graph.add_node(new_node)
        sim_graph.add_edges_from((new_node, node) for node in
nodes_to_connect)
    print(f"Added 5 new nodes each connected to 10 existing nodes")

    #Creating a 'super node' connected to many nodes
    super_node = max(sim_graph.nodes()) + 1
    super_connections = random.sample(list(sim_graph.nodes()), min(100,
len(sim_graph.nodes())))
    sim_graph.add_node(super_node)
    sim_graph.add_edges_from((super_node, node) for node in
super_connections)
    print(f"Added a super node connected to 100 existing nodes")

    #Analyze and plot the original and simulated graphs
    analyze_and_plot(graph, sim_graph)

def analyze_and_plot(original_graph, simulated_graph):
    pos = nx.spring_layout(original_graph)
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14, 7))
    plt.subplots_adjust(wspace=0.3)

    # Plot the original graph
    nx.draw(original_graph, pos, ax=axes[0], node_size=50, with_labels=False, 
font_size=8)
    axes[0].set_title("Original Graph")

    #Update positions for new nodes in the simulated graph for consistency
    pos.update(nx.spring_layout(simulated_graph, pos=pos, fixed=pos.keys()))  # 
Fixed positions for original nodes

    #Plot the simulated graph
    nx.draw(simulated_graph, pos, ax=axes[1], node_size=50, with_labels=False, 
font_size=8)
    axes[1].set_title("Simulated Graph")

    plt.show()

    print_network_stats(original_graph, "Original graph")
    print_network_stats(simulated_graph, "Simulated graph")

def print_network_stats(graph, name):
    components = nx.number_connected_components(graph)
    if nx.is_connected(graph):
        avg_path_len = nx.average_shortest_path_length(graph)
        print(f"{name} - Number of connected components: {components}, Average 
path length: {avg_path_len}")
    else:
        print(f"{name} - Number of connected components: {components}, Graph is 
not connected")


G= build_network()
network_simulation_analysis_large(G)
```

Output:



```
C:\IIT\S24\OSNA\project2\facebook>python new_network_simulation.py
Total number of nodes: 4039
Total number of edges: 84243
Removed high-centrality nodes: [1085, 1718, 698, 1577, 862, 1405, 1165, 136, 107, 171]
Removed 100 random edges
Added 5 new nodes each connected to 10 existing nodes
Added a super node connected to 100 existing nodes
Original graph - Number of connected components: 93, Graph is not connected
Simulated graph - Number of connected components: 91, Graph is not connected
```



Original Graph Analysis: The original graph appears to have a dense core where many nodes are closely interconnected, surrounded by a periphery of less connected, more isolated nodes. This could indicate a typical social network with a tight-knit community or core area where most of the interactions take place. The outlying nodes may represent less active or peripheral members of the network.

Simulated Graph Analysis: The simulated graph shows some notable differences. There are lines drawn out to individual nodes, which represent the new nodes added during the simulation, particularly the "super node" connected to 100 existing nodes. These long connections reaching out from the central cluster can have several interpretations:

The creation of the super node and additional new nodes with multiple connections has introduced new pathways for interaction, potentially changing the network's dynamics.

The removal of nodes with high betweenness centrality might have led to a less centralized structure, although the core still seems intact, possibly indicating that the network has some resilience to the removal of key nodes.

The random removal of edges may have contributed to the slight dispersal of nodes around the central cluster, representing a less dense networking core.

Project Results

The primary goal of this project was to analyze the structure and dynamics of the Facebook social network using various network science techniques, including community detection, centrality analysis, and link analysis. We built the network from a dataset obtained from Stanford's Large Network Dataset Collection (SNAP), focusing on 'circles' or 'friends lists' derived from Facebook interactions. By representing users as nodes and friendships as edges, we constructed the graph, visualized the network, and applied different algorithms to analyze it.

The project delivered a comprehensive analysis of the Facebook social network's structure and dynamics. We were able to:

Visualize the network and identify communities within it.

Determine the optimal number of clusters for community detection using the Elbow Method.

Identify central nodes through various centrality measures such as degree, betweenness, closeness, and eigenvector centrality.

Conduct link analysis to understand the significance of nodes in terms of PageRank, hub, and authority scores.

Simulate changes to the network by removing key nodes, adding new ones, and randomly disrupting connections.

Overall, these results appear to align with your goals, offering insights into the network's robustness, key influencers, community structure, and information flow.

The success of this project can be attributed to several factors:

A comprehensive dataset provided a solid foundation for building the network and performing a thorough analysis.

A structured approach, including visualization, community detection, centrality analysis, link analysis, and simulation, allowed for diverse insights into the network's dynamics.

The use of robust algorithms and established network science techniques ensured reliable results.

Discussion of Project

Overall, the project seems to have gone well. The comprehensive analysis provided detailed insights into the network's structure and characteristics. Key nodes and communities were identified, and the simulation gave a glimpse into the potential impact of changes to the network. The successful use of various network science techniques and the ability to visualize the network added to the project's value.

While the project was successful, there are always areas for improvement:

More Robust Data Handling: We encountered some challenges with data parsing and error handling. In future projects, incorporating additional data validation and error correction techniques could ensure smoother data processing.

Deeper Analysis: Given more time, we could explore deeper analyses, such as examining the network's temporal dynamics, exploring other community detection methods, or conducting a detailed sentiment analysis based on user interactions.

Improved Visualizations: Although the visualizations were useful, exploring more sophisticated visualization tools or methods could enhance clarity and offer more interactive insights into the network structure.

This project provided valuable lessons in several areas:

Understanding Network Structure: The analysis highlighted the significance of centrality measures, community detection, and link analysis in understanding a social network's structure.

Importance of Key Nodes: The centrality analysis emphasized the impact that certain key nodes have on the overall network, demonstrating the importance of identifying and understanding their roles.

Robustness and Resilience: Through simulation, we learned how the removal of key nodes and the introduction of new connections can affect the network's structure and resilience.

Effective Use of Tools: The project reinforced the value of using established libraries like NetworkX and matplotlib for network analysis and visualization.

Future Work

If we had additional time to work on the project, we would:

Dynamic Analysis: Explore how the network structure changes over time to identify trends and shifts in user interactions.

Advanced Community Detection: Experiment with other community detection methods or refine existing ones to gain deeper insights into the network's clustering patterns.

Influence Propagation: Analyze how information or trends propagate through the network, using techniques like diffusion models to understand the network's influence dynamics.

Sentiment Analysis: Integrate sentiment analysis to examine the tone and mood of user interactions, linking this with network dynamics.

Detailed User Profiling: Use the feature data to create more detailed user profiles, allowing for a more nuanced understanding of the network's demographics and behaviors.

References:

Dataset Reference: Leskovec, J., & Mcauley, J. J. (2012). Social circles: Facebook. Retrieved from Stanford Large Network Dataset Collection (SNAP): http://snap.stanford.edu/data/egonets-Facebook.html

Network Analysis Library: Hagberg, A., Schult, D., & Swart, P. (2008). NetworkX: Python software for complex networks. Retrieved from https://networkx.org/

Data Visualization Library: Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), 90–95. doi:10.1109/MCSE.2007.55

Machine Learning Library: Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825-2830. Retrieved from http://www.jmlr.org/papers/v12/pedregosa11a.html

Elbow method reference
https://www.geeksforgeeks.org/elbow-method-for-optimal-value-of-k-in-kmeans

KMeans algorithm reference:
https://stanford.edu/~cpiech/cs221/handouts/kmeans.html