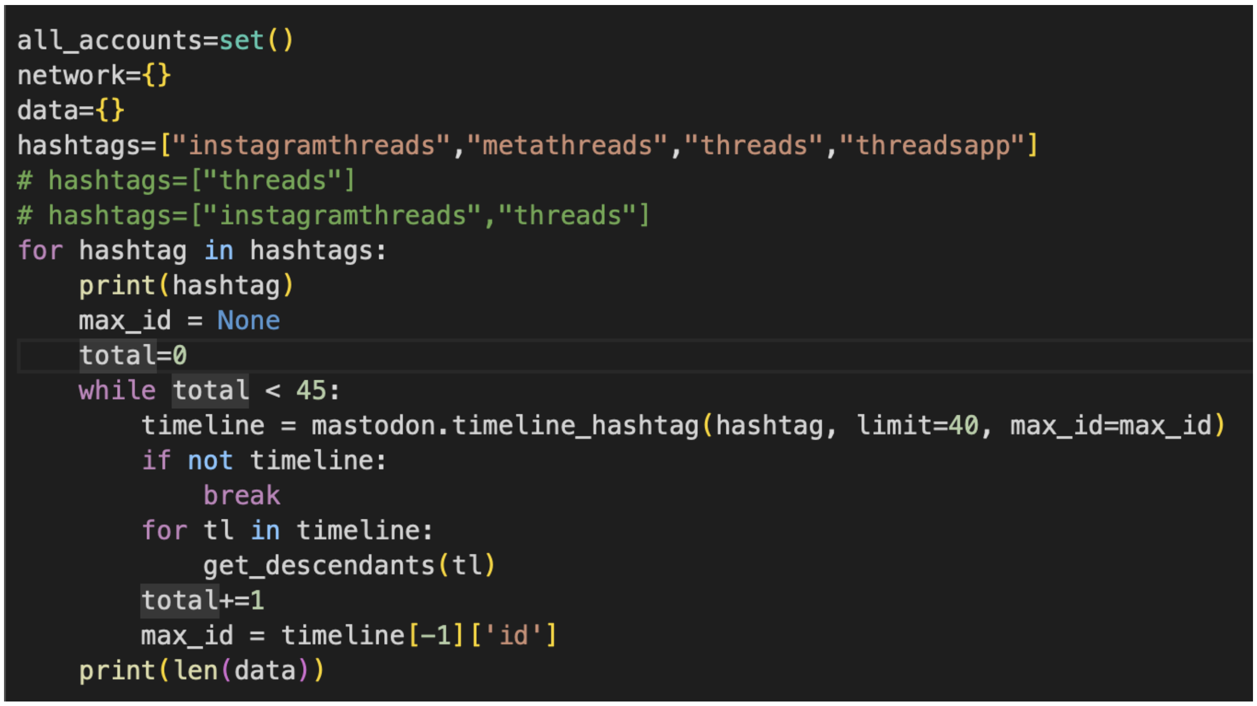
CSE 472: Social Media Mining
Project I - Social Media Data Analysis

Step 0:
I decided to focus on the topic of "Threads" because it's a subject that many people have strong opinions about. Some people really like threads, some don't, and some are in the middle and don't take a side. Threads get a lot of attention on Mastodon, and people talk about them a lot. To find toots about threads on Mastodon, I mainly used four hashtags. These hashtags helped me find and collect toots related to threads so I could study them.

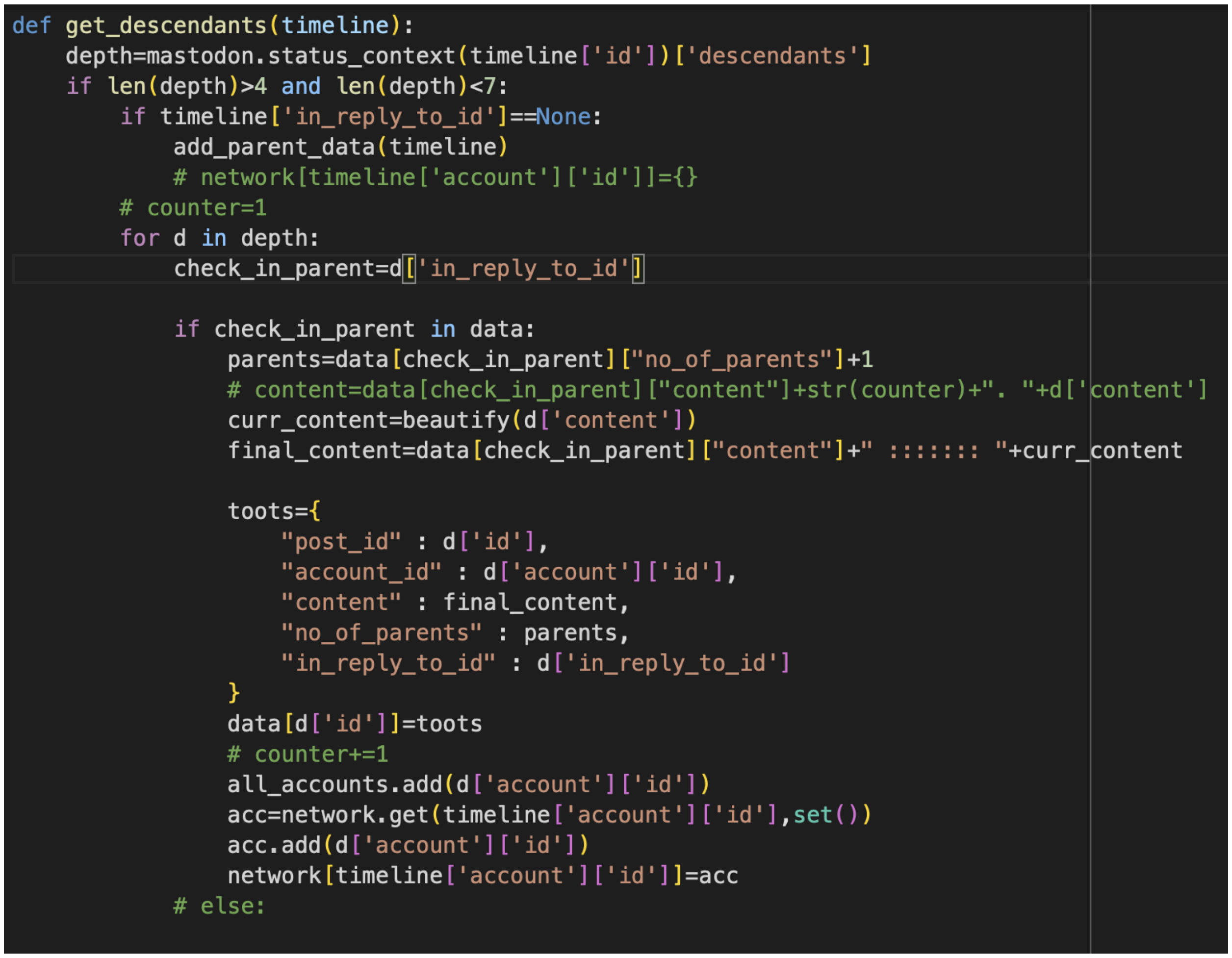
Step 1.1: Data Crawling
I used four hashtags to crawl the data from the Mastodon API. The hashtags chosen are "instagramthreads", "metathreads", "threads", "threadsapp". I crawled the data iteratively by increasing the timeline each time. I had to increase the timeline because Mastodon API fetches at most 40 toots at a time. Therefore, the timeline had to be updated after each iteration. When the next iteration starts it will start from where the previous iteration ended.



```
all_accounts=set()
network={}
data={}
hashtags=["instagramthreads","metathreads","threads","threadsapp"]
# hashtags=["threads"]
# hashtags=["instagramthreads","threads"]
for hashtag in hashtags:
    print(hashtag)
    max_id = None
    total=0
    while total < 45:
        timeline = mastodon.timeline_hashtag(hashtag, limit=40, max_id=max_id)
        if not timeline:
            break
        for tl in timeline:
            get_descendants(tl)
        total+=1
        max_id = timeline[-1]['id']
    print(len(data))
```

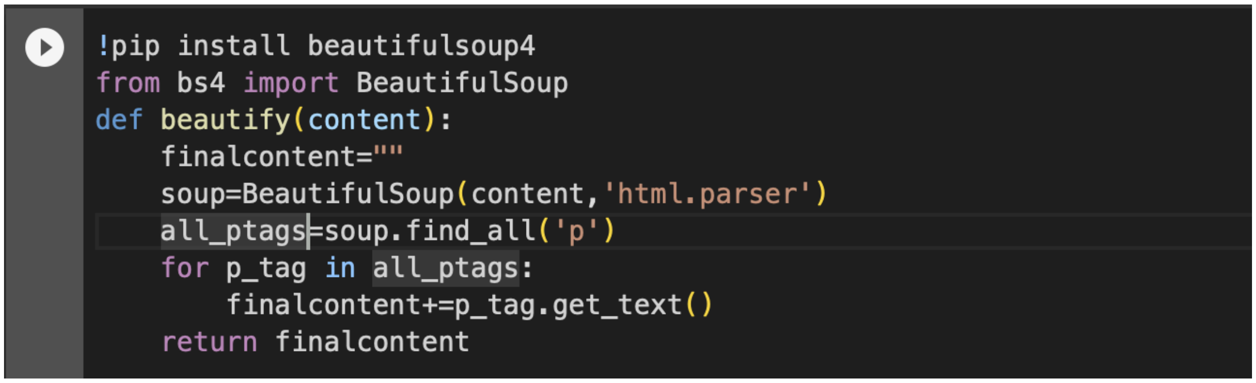
Step 1.2: Data Preprocessing
When I am fetching the timeline for all the hashtags, I am only considering the toots having at least 4 replies and at most 6 replies. After getting the toots, which I will be exploring, I am traversing the descendants of the source toot (Parent toot). Later, I am

maintaining the number of parents of the toots and then combining the content of the ancestors with this toot content with the delimiter “::::::”. I am storing the content of the ancestors because during classification(next steps), I need the whole conversation of toot so that the classification of the user can be more precise.



```python
def get_descendants(timeline):
    depth=mastodon.status_context(timeline['id'])['descendants']
    if len(depth)>4 and len(depth)<7:
        if timeline['in_reply_to_id']==None:
            add_parent_data(timeline)
            # network[timeline['account']['id']]={}
        # counter=1
        for d in depth:
            check_in_parent=d['in_reply_to_id']

            if check_in_parent in data:
                parents=data[check_in_parent]["no_of_parents"]+1
                # content=data[check_in_parent]["content"]+str(counter)+". "+d['content']
                curr_content=beautify(d['content'])
                final_content=data[check_in_parent]["content"]+" ::::::: "+curr_content

                toots={
                    "post_id" : d['id'],
                    "account_id" : d['account']['id'],
                    "content" : final_content,
                    "no_of_parents" : parents,
                    "in_reply_to_id" : d['in_reply_to_id']
                }
                data[d['id']]=toots
                # counter+=1
                all_accounts.add(d['account']['id'])
                acc=network.get(timeline['account']['id'],set())
                acc.add(d['account']['id'])
                network[timeline['account']['id']]=acc
            # else:
```

When the content is fetched, I used BeautifulSoup to retract all the <p> tag content from the toots.



```python
!pip install beautifulsoup4
from bs4 import BeautifulSoup
def beautify(content):
    finalcontent=""
    soup=BeautifulSoup(content,'html.parser')
    all_ptags=soup.find_all('p')
    for p_tag in all_ptags:
        finalcontent+=p_tag.get_text()
    return finalcontent
```

Step 2: Network data Collection and Network Construction

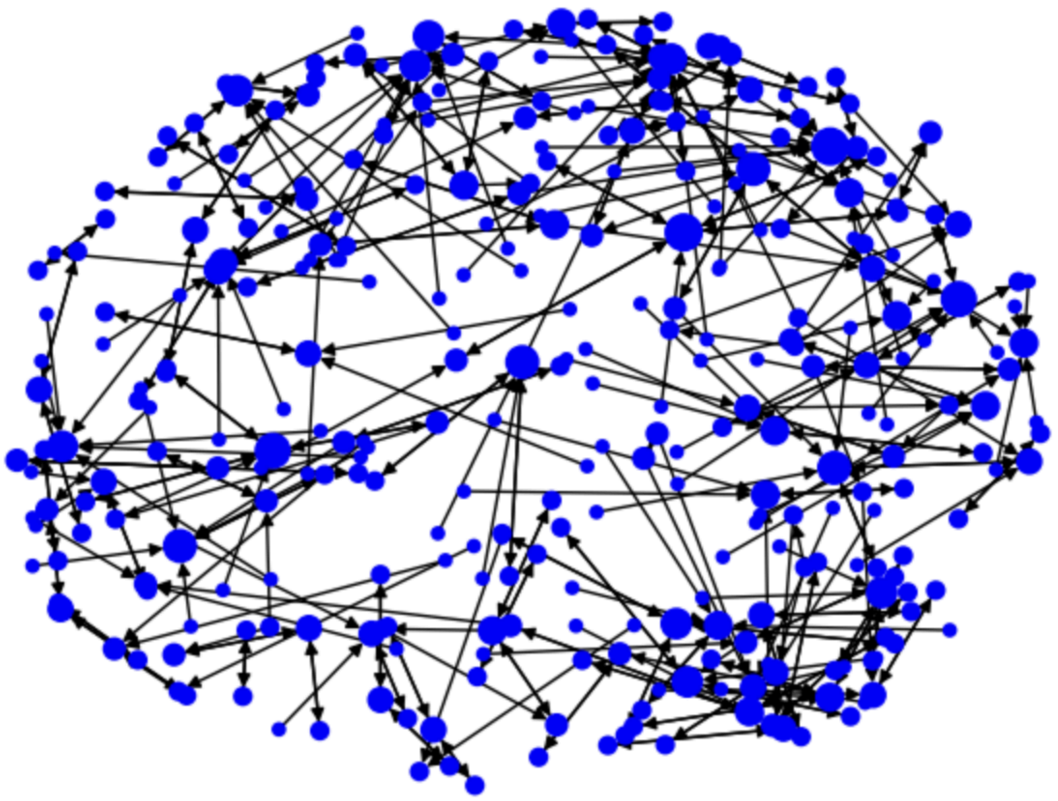
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

G = nx.DiGraph()

no_nodes=0
for node in dist_users:
    G.add_node(node)
for e in edge:
    if e[0]!=e[1]:
        G.add_edge(e[0],e[1])
# G=G.reverse()
d = dict(G.degree)

pos = nx.spring_layout(G,scale=70, k=5/np.sqrt(G.order()))
nx.draw(G, pos, with_labels=False, node_size=[d[k]*20 for k in d], node_color='blue', font_color='black')
plt.title('Graph of all the users')
plt.show()
```

I have maintained one dictionary to store the network connections. The source toot's account user id is my key and the value of the dictionary is the list of all the reply's account ids. The dictionary is later stored into JSON format, for the sake of this project. I have visualized my directed graph using the networkx library. I am maintaining the edge list of the distinct users and getting their connection node from the in_reply_to_account_id value and storing it as a tuple. Tuples will be required for the direction of the edge. Later, the nodes and the edges are added in networkx graph component.

I have constructed a Diffusion Network graph in which each node represents the user who initiated the original toot, and the connected nodes represent users who responded to that original post.

In certain cases, there may exist nodes with fewer than four connected users. This occurrence can be attributed to instances where same users who created original toot responded later to the toot, resulting in descendants that include the same users. To avoid self-loops in such cases, I have excluded these users in this step.

Graph of all the users



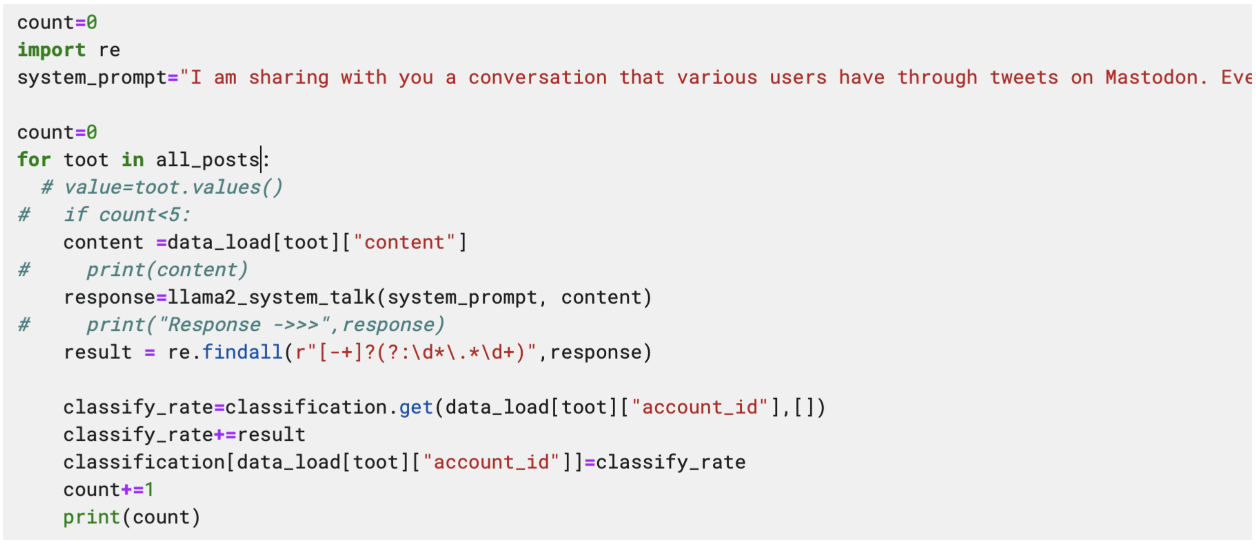
Step 3: Data Classification

I have decided to choose Llama2-7B as my LLM classification model. The prompt which I created for this project is given below.

"I am sharing with you a conversation that various users have through tweets on Mastodon. Every tweet is separated by "::::::" . First tweet is the parent tweet, following tweets separated by "::::::" are the replies to the tweet before it. Based on this conversation, Can you rate the opinions of the last person on the conversation which i provided, on a scale of -1 to 1. Here -1 means that the user is Anti-Thread. 1 rating means that the user is Pro-Thread and 0 rating means that the user is Neutral or unbiased about threads. These ratings should be deduced by considering the entire conversation and what the last person has to say about it. Also if the last user's opinion is Neutral, then classify that user based on what the first user tweets. If the

user feels positive towards Threads, classify it as 1. If the user feels negative towards Threads, classify it as -1. Do not provide any further explanation. I am only concerned with the actual Rating. Do not provide any other textual response."

I have asked the model to give the classification based on the ratings ranging from -1 to 1. For the toots who are replies to the original toot. I am providing the whole conversation of the toot's ancestors, separated by the delimiter for accurate classification. I am requesting the model to categorize the final segment of the toot because that part represents the user's response to the toot. I used the rating system because it is easier to fetch the classification from the verbose LLM response.

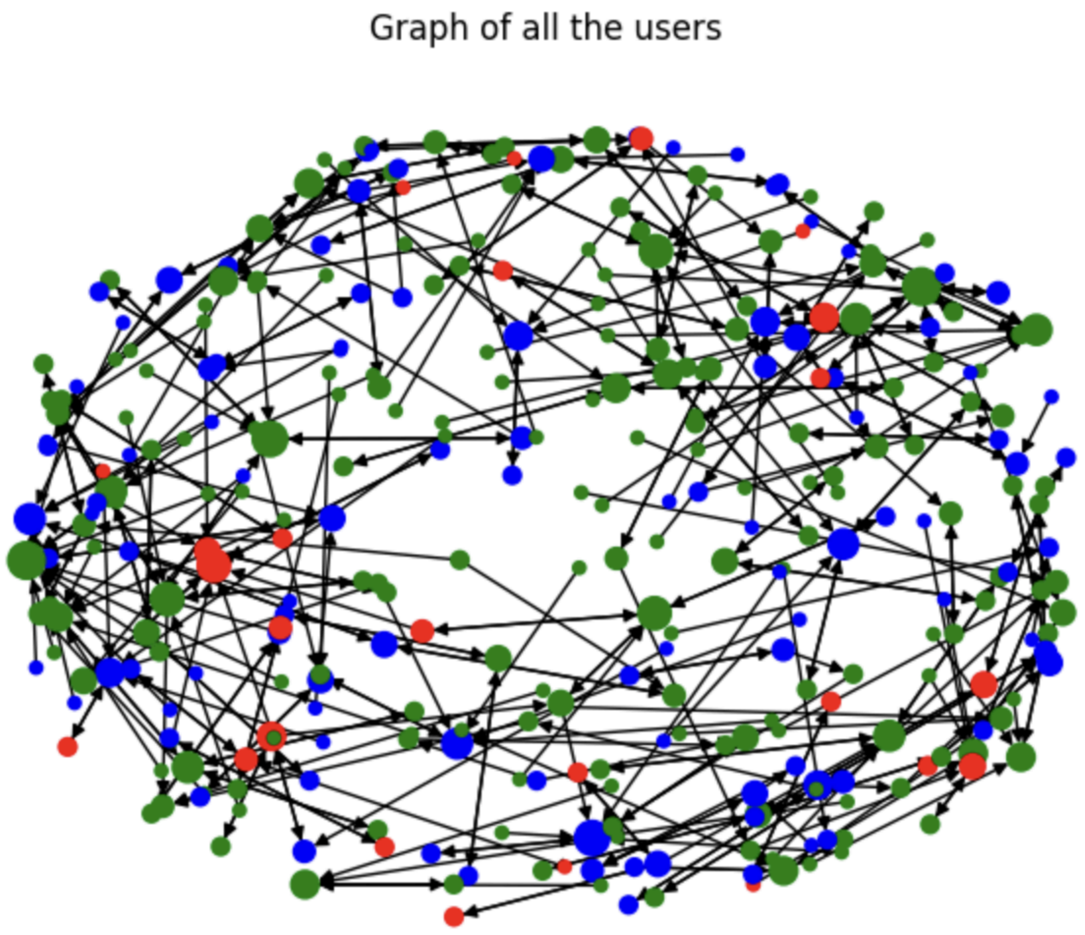


```
count=0
import re
system_prompt="I am sharing with you a conversation that various users have through tweets on Mastodon. Eve

count=0
for toot in all_posts:
  # value=toot.values()
#   if count<5:
    content =data_load[toot]["content"]
#     print(content)
    response=llama2_system_talk(system_prompt, content)
#     print("Response ->>>",response)
    result = re.findall(r"[-+]?(?:\d*\.*\d+)",response)

    classify_rate=classification.get(data_load[toot]["account_id"],[])
    classify_rate+=result
    classification[data_load[toot]["account_id"]]=classify_rate
    count+=1
    print(count)
```

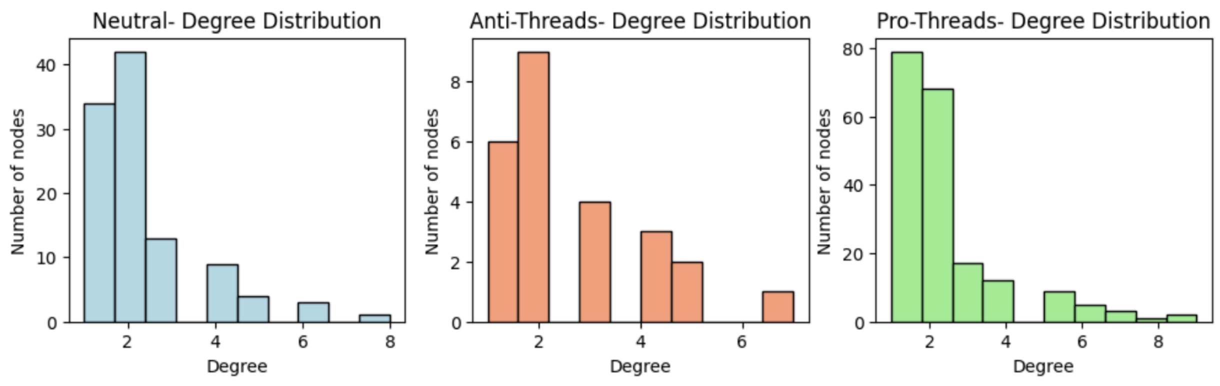
Once the response is fetched, I used regex to fetch the rating for the post's content, as the llama's response had some textual data. For the node classification, I created a classification dictionary, in which I am storing the llama's response rating.

Finally, I iterated over all the classifications and computed the mode for all the responses. I chose to calculate the mode because a user might have replied to multiple toots. Therefore, the overall rating is determined by the most common response among all the replies a user made to various toots.

When the mode of all the user's responses is positive, it is represented as green, indicating a pro-Thread user. If the mode is zero and represented as blue, the user is considered neutral. Conversely, if the mode is negative, it signifies an anti-Thread user.

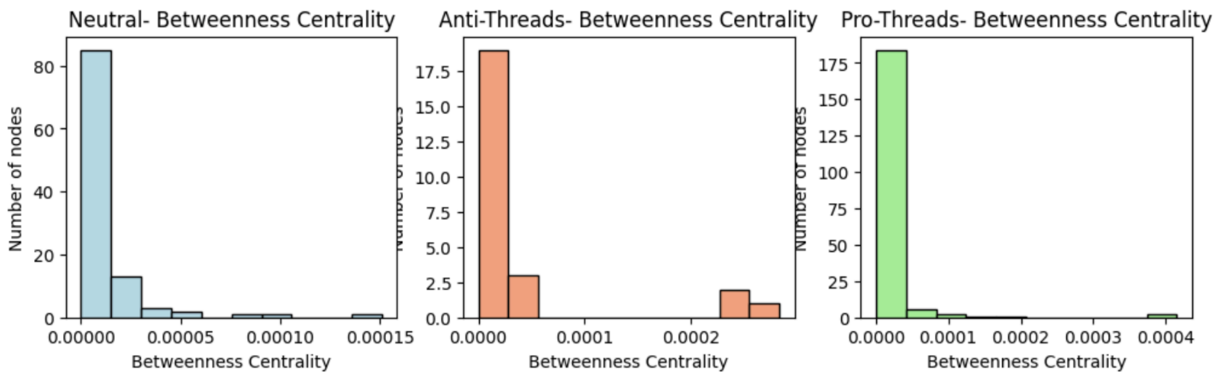


Step 4: Metrics

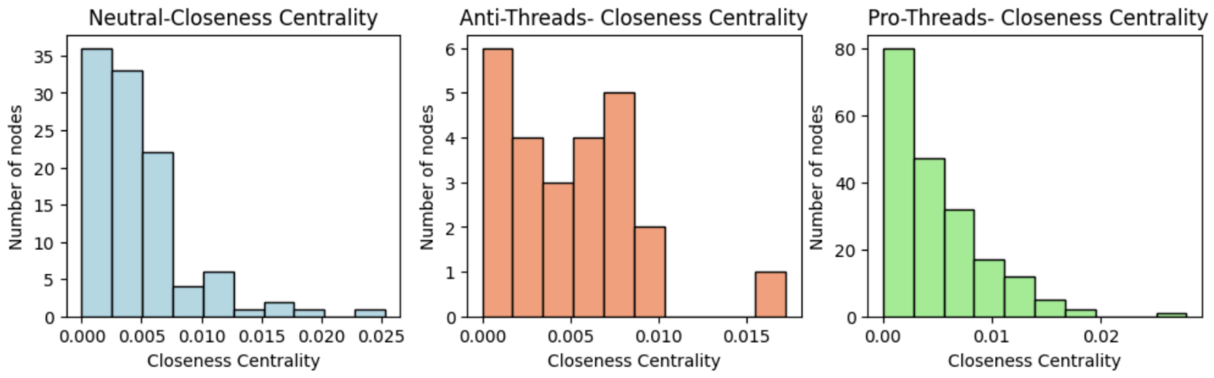


There is a high number of nodes having low degree distribution. This happens because I didn't count cases where users replied to their own toots. So, for most toots, there are only a few people replying to them.

It's a common pattern in networks like this, where most toots have a small number of replies, while a few have a lot. This tells us how conversations or interactions happen in the network.



I am observing a higher number of nodes with very low betweenness centrality in the diffusion network because many nodes within the network are not acting as crucial bridges in connecting different parts of the network. Betweenness centrality measures how often a node serves as a critical pathway for information flow between other nodes. This can be validated as there are very few toots which are common in the network. In other words, most nodes are not central in terms of controlling the flow of information between different parts of the network.



Closeness centrality measures how closely connected a node is to other nodes. There are many isolated nodes in my graph, where it is required to hop multiple times to reach other nodes. These nodes tend to have lower closeness centrality scores because they are not closely connected to the majority of other nodes. In other words, there are very few toots which are highly connected.

Appendix(Submitted Files):

threads_data_new.json : This serves as my primary JSON file, encompassing all the toots with more than four descendants. It includes information such as account ID, the number of parent toots, post ID, post content, and the in_reply_to_id.

network.json : This file contains all the information of the nodes and their edges based on the replies on the user's toots.

node_classification.json : This file comprises all the distinct users and their classified color(by LLM model). In this context, the color red indicates that the user holds Anti-Thread opinions, while green represents Pro-Thread viewpoints, and blue signifies a neutral stance.

smm_data_crawling.py : This is the executable code for crawling the data from Mastodon API.

network_classification.py : This python code contains the code for network visualization and the network classification.

visualization.py : This file contains all the visualizations done in this project.