Social Media Influence on Elections

University of St. Thomas
Center for Applied Mathematics

Madilin Herzog and Rweto Rwetoijera

Advisor: Natasa Dragovic

August 2024

1 Introduction

Social media is utilized all around us connecting people throughout the country and the world. This allows political campaigns, news, and advertisement to spread quickly and globally. During the 2020 political cycle roughly $9 billion was spent on political advertisement, according to AdImpact, a non-partisan firm that tracks ad campaign spending's. They are estimating in the 2024 election cycle roughly $10 billion will be spent on political advertisement (*4*). In AdImpact, political advertisement includes broadcasting, digital advertisement, radio, and more (*4*). With all this money being spent on advertisement and with the use of social media, this will allow political campaigns to spread throughout the country even more than in the past. With the substantial amounts of money being spent on elections, this raises questions like how does social media impact political candidates and elections?.

In this paper, we are going to further investigate the influence social media has on election with two different models. The first model we will be looking at is Adaptive DeGroot Model. Utilizing this model, we will see how social media impacts elections as a random event that occurs occasionally over time and discuss key results noticed. Additionally, we will discuss a second model; Opinion Dynamics Network Model. This model focuses on the way people talk to each other and interact via social media where geography is important. We will explore the results and explore the effects social media has for political candidates. We will also compare the models to find similarities in trends to help explain the effect social media has on elections.

2 Background

In political campaigns, candidates often need to decide where to position themselves on the political spectrum. In our case, it ranges from left to right as we consider two candidates. The idea is that their positioning will influence the results of the election. This decision is crucial because voters beliefs are spread out across this spectrum, mostly resembling a Gaussian mixture. This means that most people cluster at certain points, but smaller groups with different beliefs will also be formed. Imagine an election with two candidates, one on the left and one on the right. All voters must vote, and the candidates positions can change the elections outcome. If one candidate moves closer to the center (where the median voter is), they might attract more votes from being undecided or moderate voters. However, social media plays a dominant role in the digital world and heavily influences people's perspectives. The situation becomes complex. This can amplify both centrist and extremist messages, influencing voters unpredictably. This can be understood through mathematical models that analyze how these shifts occur (*3*).

3 Motivation and Goals

The goal of the project was to represent social media in a system to observe how individual's political views change with social media's influence as social media is a growing factor in modern day use in political campaigns. Since there is an increase in spending for campaigns in broadcasting and advertisement, we wanted to see how social media can impact a person's political opinion. With this potential change in a person's political opinion, we wanted to investigate how this impacts a political candidate by trying to understand the best way to optimize their position to get the most votes based on the impact social media can have on an election.

4 Adaptive DeGroot Model and Results

The DeGroot learning model, introduced by statistician Morris H. DeGroot in 1974, is a foundational framework for studying opinion dynamics within social networks. This model provides a mathematical approach to understanding how individuals update their beliefs based on the opinions of others over time. The DeGroot learning model examines how n agents update their opinions over time based on a stochastic trust matrix T. Each agents opinion vector evolves according to

$$p(t) = Tp(t-1)$$

eventually converging to

$$p(\infty) = \lim_{t \to \infty} T^t p(0)$$

under certain conditions (*1*).

In a strongly connected network, where every agent can reach every other agent easily, beliefs would likely converge to a common value. The final consensus is mostly influenced by the left eigenvector s of T, which reflects the weight of each agents initial opinion in the final consensus. Even if the network is not strongly connected, beliefs can still converge if every strongly connected and closed set of agents is aperiodic. Without these conditions, groups might reach internal consensus without achieving overall societal consensus.

The basic DeGroot model operates with the consideration of a static trust influence matrix, where the influence one person (agent) has on another remains constant over iterations. This assumption does not accurately reflect the dynamic nature of social media interactions, where trust and influence can fluctuate rapidly. For instance, trust and influence can change based on current events or shifts in personal relationships, making the static assumption less applicable in real-world scenarios.

We modified the Adaptive DeGroot model to capture the dynamic nature of trust and influence within a social network. Identifying how these factors evolve as individuals receive and process new information. The model starts by generating initial scores s_i for different candidates for each voter i using a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, representing each voter's starting opinions. The initial scores are generated using a Gaussian distribution with a mean (μ) of 5, a standard deviation (σ) of 2, and a variance (σ^2) of 4:

$$X \sim \mathcal{N}(\mu = 5, \sigma = 2, \sigma^2 = 4)$$

The influence matrix is generated using a Gaussian distribution with a mean (μ) of 0, a standard deviation (σ) of 1, and a variance (σ^2) of 1:

$$Y \sim \mathcal{N}(\mu = 0, \sigma = 1, \sigma^2 = 1)$$

The influence each voter i has on others j is captured in an influence matrix $\mathbf{W}$, with elements w_{ij}, also generated with Gaussian-distributed values. This matrix $\mathbf{W}$ is then normalized to ensure that each row sums to one maintaining appropriate weight.

$$\sum_{j=1}^{N} w_{ij} = 1.$$

In this adaptive model, the influence matrix $\mathbf{W}$ is not static, it dynamically adjusts its values over iterations (time) to reflect the evolving nature of trust and influence. The changes occurring in the influence matrix are closely tied to the changes happening in the perspectives or scores. This adaptation is key for representing and understanding how perspectives change as individuals interact and receive new information, particularly in the context of social media and politics.

The simulation iteratively updates the voters' scores s_i based on this influence matrix $\mathbf{W}$, allowing beliefs to evolve gradually. To introduce real-world situations, random events are introduced after every 10 iterations. These events randomly alter individual scores s_i, simulating external influences on voter opinions. After each event, scores s_i are normalized to preserve consistency across the simulation.

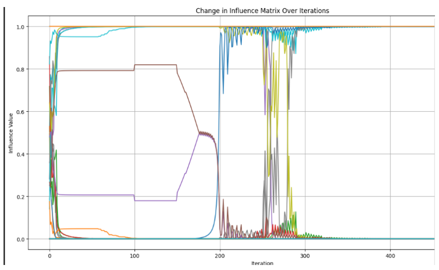
The simulation continues until the scores s_i reach consensus or until the maximum number of iterations is reached (1,000 in this case). The iteration limit can be adjusted as needed. This limit is established because, in some cases, the influence matrix $\mathbf{W}$ can fluctuate significantly, requiring more time to reach consensus. Throughout the simulation, various scenarios can be easily tested by generating values rather than inputting them manually. The average number of connections per voter is controlled to ensure a realistic network structure.

Key Observations:

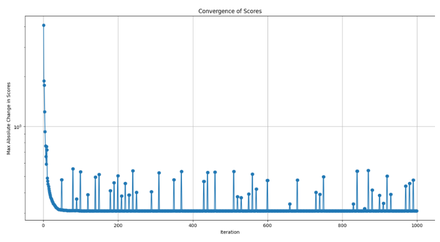
- The trust matrix $\mathbf{W}$ often displayed significant noise and shocks due to random events and fluctuations in the candidates' scores. This was particularly observed in the early stages of the simulation.As illustrated in Figure 1(a),these fluctuations gradually diminished as the system approached stability.

Figure 1: Adaptive DeGroot Model

(a) Changes in the influence matrix over iterations



(b) Convergence of scores showing max change between iterations

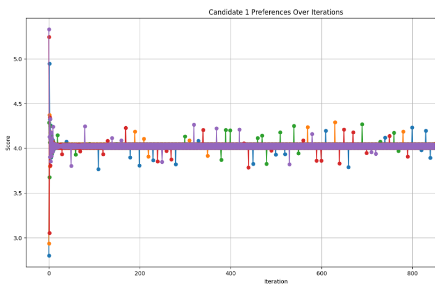


- Certain influence values w_{ij} peaked, and candidate scores evolved rapidly before undergoing significant fluctuations, particularly during random events. This can be observed in Figure 2, where changes in scores show noticeable spikes, resulting from these random events. This behavior was especially observed in cases where consensus was not easily reached

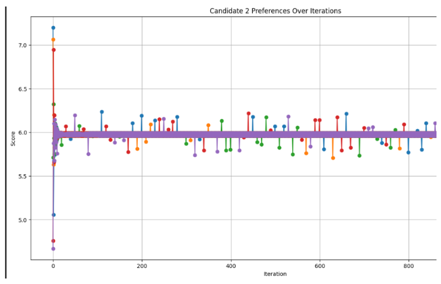
- The number of initial connections per voter i was a critical factor

Figure 2: Adaptive DeGroot Model

(a) Average scores for each candidate 1 over iterations



(b) Average scores for each candidate 2 over iterations



influencing the simulation's success and dynamics. The initial average number of connections significantly affected the evolution of the influence or trust matrix **W**, showing the importance of network structure in opinion formation and consensus-building.

5 Opinion Dynamics Network Model and Results

The second model we looked at was the Opinion Dynamics Network (ODyN) model (*2*). An ODyN model utilizes the concept of Opinion Dynamics to represent the different ways opinions are spread. Additionally, the ODyN model factors in geography, which makes it different from the previous Adaptive DeGroot model we looked at. The ODyN model has agents connected to each other to represent their interactions. This creates a social network allowing the connections between agents to be observed. In the ODyN model, once the initial model is created, it runs through a simulation; for each iteration it checks who the agents are connected to and sees if they can be influenced by the people they are connected to. In this particular code, it includes mega-influencers which are agents that are more influential than others. They don't have more power in converting others, just more reach. They can be seen as nodes that are larger than others. For example, you can see an example in ODyN model Figure 2 (a) below (*5*).

In the model each dot is a node that represents an agent. The agents represent a person in a sense. The model also has lines connecting each node representing the connection/ interactions between each agent. The connection in the social network is representing individuals connected via social media. (See ODyN model Figure 1(a)) The agents are connected based on weights that are randomly assigned to them and their initial values. The connection of the nodes will be discussed further later in this section. The initial model takes the number of agents and randomly places them within a triangle. The number of agents are randomly assigned opinions and weights. The weights are assigned randomly based on a uniform distribution. The opinions for the agents are randomly assigned based on a bimodal distribution. The bimodal distribution is two normal distributions combined. For even number of agents 50% is assigned to each normal distribution respectively. Then each agent potentially gets connected to the other agents based on probabilities, weight, and distance. The probability of one agent influencing another can be written as (*2*).

$$p_{uv} = min(1, \frac{1}{(1 + \|X_u - X_v\|/\lambda)^{\sigma}} W_v^{\alpha} \mathbb{1}_{|H_u - H_v| < b}) \tag{1}$$

This equation represents the probability that node u influences node v. Each part of the equations has an impact on the chances the two nodes are connected. X_u and X_v are values determined by a location in R^2 which is a uniform distribution. Also, if an agent's weight is higher, they are "more important" meaning they have a higher chance at changing another agent's opinion. This can be seen by W_v^{α}. The last part of the equation that has an impact focuses on the agent's opinions at a certain time. If opinions are too different from each other they will not be connected. This can be seen in $|H_u - H_v|$. Lastly, geography is important in this model if the parameters are set where distance is important as nodes will not be connected if they are too far away from each other. Once the connection is made, this gets added to the initial model which the code was based around previous work (*5*). The simulation code is also from this previous work, slightly changed to view the code for setting up the model and to run the simulation, see appendix for additional information. After the initial model is created, the simulation will run, allowing for the opinions to change over time.

The simulation goes through multiple iterations, updating the beliefs of the agents each iteration until the rolling average is less than the stopping threshold. The opinions are changed based on who each node is connected to and how open the agent is to others along with connections to mega-influencers. Then the simulation for the iteration it is on, goes through and changes each agents beliefs based on Hegselmann-Krause Model, which takes the agents opinions and updates it based on taking the average of the opinions of the neighbors(*6*). Once the criterion are met, the simulation is stopped, and the opinions at each iteration are plotted on two types of graph to give a visual on how opinions of the agents changes over time based on who they were connected to.

In the code, the initial variables the user can set are γ, b, α, δ, and number of agents. Each of the variables change the system in some way. To start, γ is the power law exponent, which is used in multiple parts of setting up the system. Next, b is how open an agent is to their neighbors. The higher the value of b, the more open they are to other agent's opinions that they are connected to. α is the importance of weight which helps determine how important a person is. Next, δ determines how important distance is for the agents. The higher the value of δ, the more important distance is, meaning less connections with those farther in distance. Lastly, the number of agents can be changed so there are more or less people in the system. Depending on the values these are set at, it can change the overall model and change the average number of connections. This can change the overall outcome of the simulation affecting the results of where political candidate's place themselves for gaining the most votes.

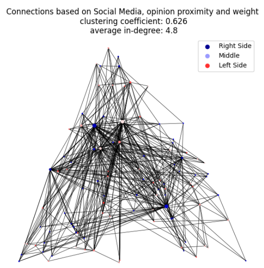
After a variety of testing with normal and uniform distribution, the model was changed to focus on bimodal distribution. This allows for the focus to be on having agents on both extremes of the political spectrum. The initial values we found that resulted in an unstable system was $\gamma = 1.5$, $b = 1$, $\alpha = 2$,$\delta = 3$, and $\lambda = \frac{1}{10}$ with number of agents being equal to 100. In the simulations, there were cases where the opinions in the network converged to a more neutral ideology towards the center of the spectrum. There were cases where the network diverged, meaning the agent's opinions stayed on the extreme ends of the political spectrum. To begin, we will look at a couple starting models of the ODyN model that were simulated and then look at the results.

Next, we are going to further examine two trends that occurred under these certain parameters. Both Figures represent an average case of what was seen while running the simulations.

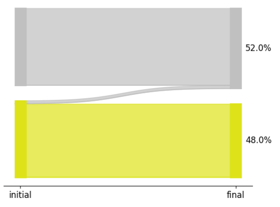
Figure 3: ODyN model (a) is an example of what an average initial model looked like without any mega-influencers. In the model there are no larger nodes than others that stand out, meaning in this system there is not an agent that has more social media power than others. Additionally, we can see that not every node is connected. In Figure 3: ODyN model (b) this graph represents the initial beliefs and shows the final beliefs of the agents. In this case we start with 50% of agents on each side of the spectrum; by the end it resulted with 52% of the agents in the left-side opinion and 48% of agents on the right-side opinion. This indicates that there was not a great shift in agent's opinions when being connected together in this setting. Looking at Figure 3: ODyN model (c) presents another way to view the change in agent's opinions. In this figure, we can see that having the agents on these extremes tend to have the agents stick to their initial opinions. This results in the agent's opinions diverging over time. In this case with no mega-influencer and people connecting via social media, the best place for candidates to position themselves is on their respected extreme. Placing themselves on the extremes would allow for the most votes and would give them the best spots to win the election.

Figure 3: ODyN Model

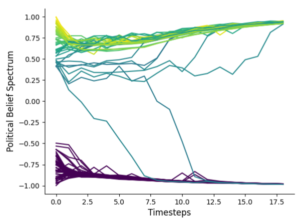
(a) Initial ODyN Model


Connections based on Social Media, opinion proximity and weight
clustering coefficient: 0.626
average in-degree: 4.8
Right Side
Middle
Left Side

(b) Initial Beliefs and Final Beliefs



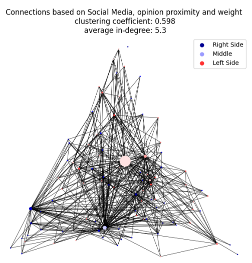
(c) Belief Change Over Time



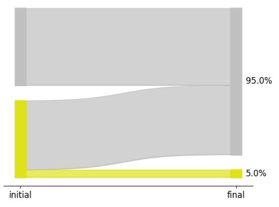
With the same initial values, we ran another case where there is the emergence of mega-influencers, which can be seen in the figures below. In this network there is a mega-influencer that is more towards the middle/ left side of the spectrum. This influencer has more power in persuading others as well as having more connections. This represents a social media celebrity or someone who has a larger following to share opinions with others. In this case looking at Figure 4: ODyN model (b) started with 50% of people on both sides of the spectrum, the final result does not remain equal percentages on both side of the spectrum like the previous figures. In this case, 95% of the agents end up on the left side of the spectrum and 5% of the agents end on the right side of the spectrum. Then, looking at Figure 4: ODyN model (c) it changes the perspective of figure 4: ODyN model (b). It shows us that the views of the agents change with the influence of a mega-influencer. It allows us to see that the agents did not switch from one extreme opinion to another, rather the mega-influencer helped sway the opinions towards the center of the political spectrum. Then, over time, the opinions of the agents end up towards the center. In this case, with having a mega-influencer, the effect of social media pulls agents opinions towards the center of the political scale. In this case, the best position for the political candidates to gain the most votes is to place their views and campaign plan to be more

Figure 4: ODyN Model

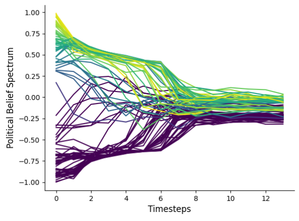
(a) Initial ODyN Model


Connections based on Social Media, opinion proximity and weight
clustering coefficient: 0.598
average in-degree: 5.3
Right Side
Middle
Left Side

(b) Initial Beliefs and Final Beliefs



(c) Belief Change Over Time



neutral towards the center view.

Overall, depending on the initial model in the ODyN model it will change the outcome with a result in either the opinions of all agents converging or diverging. In the case of the opinions diverging, social media kept those with similar ideas together and separated those who had different ideas. In this case, the best plan for political candidates would be to place themselves on the political extreme to potentially get the most votes. Meanwhile, when there is a mega-influencer or when the opinions converge, social media brought together agent's opinions, resulting in the best political position to be is towards the center of the political spectrum. In the end, social media has the potential to change the way a candidate may choose to position themselves for the best voting outcome for themselves.

6 Conclusion

In both models we can see a trend that there are a variety of cases where social media will change the potential positions of voters; therefore, a candidate may want to place themselves more towards the center or extreme for optimal amount of voters. In Figure 1 in the Adaptive DeGroot Model there are moments where the opinions diverge and converge. The moments where the graph diverges are similar to ODyN model Figure 3: ODyN model (c). In this case it is best for the political candidates to place themselves on the extremes of the political spectrum to gain the most votes. Meanwhile in the Adaptive DeGroot Model, when the model converges or the different views intercept, the best place for the political candidate to be is in the middle of the spectrum. This can be seen in a similar case in ODyN model Figure 4: ODyN model (c) where the opinions of the agents converge over time. Thus, indicating the best position for political candidates would be in the center. This would allow for optimal votes. In both models we can see similarities in trends with social media demonstrating when the best strategy is to be on the extreme part of the spectrum or the middle. This demonstrates that social media does have some potential to change people's opinions but since there are many other factors to include, since people's opinions are influenced by many additional factors other than social media, further testing with more parameters may be needed to further determine social media's impact to the fullest.

7 Future Work

Based on our findings and current work we think the next best steps for the Adaptive DeGroot Model would be to enhance the model's effectiveness by determining the optimal range for the number of average connections relative to the number of agents. Identifying this range will increase the effectiveness of the simulation performance and may enable the evaluation of the model using real-world data. Additionally, to understand how varying initial perspectives and trust matrix values influence consensus formation and belief dynamics over iterations. Analyzing why beliefs may converge, diverge, or show other patterns.Then interpreting these behaviors in real-life contexts will provide a better understanding of social influence mechanisms and societal consensus.

Future work for the ODyN model could include adding a variety of different variables to allow the network to become more realistic. Another step that could be taken is to test the model with a higher number of agents to see if any new trends appear with certain parameters. Additionally, it would be interesting to see if trends with 100 agents can be replicated with more agents. Another potential step could be to randomly place agents on the bimodal distribution rather than having them evenly distributed. This can allow for a more realistic test, as people's views are not evenly distributed along the political spectrum.

8 Appendix

8.1 Appendix A: ODyN Model Code

- The code below is the initial code with the parameters that create an unstable result in the agents opinions over time. When run in python or a python translator (in our case we used Visual Studio Code) the initial parameters are run through a simulation file code along with running the data stored from the simulation file through the visualization file code to create figures with the results of the simulations. The code was initially created initially created by Anna Haensch for another project and alter for our project (*5*).

```
import pandas as pd
import numpy as np
from matplotlib.patches import Rectangle

import sys
sys.path.append('..')

import src as odyn

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.cm import ScalarMappable
proportion_right = .5
proportion_left = .5
p = [proportion_right, proportion_left]

gamma = 1.5
b = 1
alpha = 2
delta = 3
lam = 1/10
model = odyn.OpinionNetworkModel(
                probabilities = p,
                power_law_exponent = gamma,
                openness_to_neighbors = b,
                openness_to_influencers = 1.5,
                distance_scaling_factor = lam,
                importance_of_weight = alpha,
                importance_of_distance = delta,
                include_opinion = True,
                include_weight = True,
                include_distance = True
                           )

num_agents = 100
model.populate_model(num_agents = num_agents, show_plot
                                    = True)
sim = odyn.NetworkSimulation()
sim.run_simulation(model = model, stopping_thresh = .01,
                                show_plot = False,
                               store_results = False)

fig, ax = plt.subplots()
odyn.get_alluvial_plot_axis(ax = ax,
                dynamic_belief_df = sim.
                                                dynamic_belief_df
                                                ,
                right_leaning_threshold = -10, #we
                                                ignore
                                                this for
                                                now
                left_leaning_threshold = 0)
plt.show()

fig, ax = plt.subplots()
odyn.get_line_plot_axis(ax = ax, dynamic_belief_df = sim
                               .dynamic_belief_df)
plt.show()
sim.plot_simulation_results()
```

- The code below creates the visualizations for the the results of simulation. It plots the initial model, plotting agents, and their connections in a triangle. Additionally, this plots a line plot and an alluvial plot. This allows us to view the change in agents beliefs of time. This code is run through the building the environment code above and the initial parameters are run through simulation code and then through this code to create visualization. The code was initially created initially created for another project and alter for our project(*5*).

```
import pandas as pd
import numpy as np
from datetime import timedelta
from matplotlib import cm
from matplotlib.colors import ListedColormap,
                               LinearSegmentedColormap
from matplotlib import colormaps
from matplotlib.patches import Rectangle
from pySankey.sankey import sankey
from scipy import stats
import geopandas as gpd
import matplotlib.pyplot as plt
import matplotlib.gridspec as grid_spec
from .geolocations import *

COLORS = {"light_orange":"#E69F00",
             "light_blue":"#56B4E9",
             "teal":"#009E73",
             "yellow":"#F0E442",
             "dark_blue":"#0072B2",
             "dark_orange":"#D55E00",
             "pink":"#CC79A7",
             "purple":"#9370DB",
             "black":"#000000",
             "silver":"#DCDCDC"}
BELIEF_CMAP = LinearSegmentedColormap.from_list(
        'trunc({n},{a:.2f},{b:.2f})'.format(n=cm.seismic
                                       , a=.1, b=0.7),
        cm.seismic(np.linspace(0.1, 0.7, 100))) #seismic
SIM_CMAP = LinearSegmentedColormap.from_list(
        'trunc({n},{a:.2f},{b:.2f})'.format(n=cm.BuGn.
                                       name, a=0.1, b=0.7
                                       ),
        cm.BuGn(np.linspace(0.1, 0.7, 100)))

def plot_triangulated_county(geo_df, bounding_box = None
                             , restricted = False,
                             aspect_ratio = 1):
    """ Plots county with triangular regions.

    Inputs:
        geo_df: (dataframe) geographic datatframe
                                       including county
                                       geometry.
        bounding_box: (list) list of 4 vertices
                                       determining a
                                       bounding box
                where agents are to be added.  If no box
                                           is given,
                                           then the
                bounding box is taken as the envelope of
                                           the
                                          county.
        restricted: (bool) if True then region is
                                       restrict to
                                       bounding box.
        aspect_ratio: (float) aspect ratio of final plot
                                       .

    Returns:
        Boundary of county and triangulation of region.
    """
    tri_dict = make_triangulation(geo_df)
    tri_df = gpd.GeoDataFrame({"geometry":[Polygon(t)
                                  for t in tri_dict["
                                  geometry"]["
                                  coordinates"]]})

    # Establish initial CRS
    tri_df.crs = "EPSG:3857"

    # Set CRS to lat/lon
    tri_df = tri_df.to_crs(epsg=4326)

    fig, ax = plt.subplots(figsize = (10,10))
    linewidth = 1
```

```
# Get bounding box geometry.
if bounding_box is not None:
    sq_df = gpd.GeoDataFrame({"geometry":[Polygon(
                               bounding_box)]})

    # Get bounded triangles.
    if restricted == True:
        inset = [i for i in tri_df.index if tri_df.
                                  loc[i,"
                                  geometry"].
                                  within(sq_df.
                                  loc[0,"
                                  geometry"])]
        tri_df = tri_df.loc[inset,:].copy()

        # Set plot limits.
        minx = np.array(bounding_box)[:,0].min()
        miny = np.array(bounding_box)[:,1].min()
        maxx = np.array(bounding_box)[:,0].max()
        maxy = np.array(bounding_box)[:,1].max()

        ax.set_xlim(minx - .0005,maxx + .0005)
        ax.set_ylim(miny - .0005,maxy + .0005)

        linewidth = 4

# Plot triangles
tri_df.boundary.plot(ax = ax,
                    alpha=1,
                    linewidth = linewidth,
                    edgecolor = COLORS["light_blue"]
                                                   )

# Plot county boundary.
geo_df.crs = "EPSG:3857"
geo_df = geo_df.to_crs(epsg=4326)
geo_df.boundary.plot(ax = ax,edgecolor = "black",
                               linewidth = linewidth)

# Plot bounding box.
if bounding_box is not None:
    if restricted == False:
        sq_df.boundary.plot(ax = ax,
                            alpha = 1,
                            linestyle = "--",
                            linewidth = 2,
                            color =
                            COLORS["dark_orange"])

ax.set_axis_off()
ax.set_aspect(aspect_ratio)
plt.show()

return None
```

```
def plot_agents_on_triangle(triangle_object, agent_df):
    """ Returns triangle filled with agents.

    Inputs:
        triangle_object : (polygon) shapely triangle
                                    object.
        agent_df: (dataframe) x,y coordinates for agents
                                    .

    Outputs:
        Plot of points on triangle.
    """
    fig, ax = plt.subplots(figsize = (8,8))
    df = gpd.GeoDataFrame({"geometry": triangle_object},
                              index = [0])
    df.boundary.plot(ax = ax, alpha=1, edgecolor =
                             COLORS["light_blue"])
    ax.scatter(agent_df["x"], agent_df["y"], color =
                             COLORS["dark_blue"],
        zorder = 0)
    ax.set_axis_off()
    plt.show()
    return None

def plot_agents_with_belief_and_weight(belief_df):
    """ Returns triangle filled with agents.

    Inputs:
        triangle_object : (polygon) shapely triangle
                                    object.
        agents: (dataframe) x,y coordinates for agents.

    Outputs:
        Plot of points on triangle.
    """
    fig, ax = plt.subplots(figsize = (8,8))
    ax.scatter(x = belief_df["x"],
               y = belief_df["y"],
               s = [int(w) for w in belief_df["weight"]
                                         .values],
               c = belief_df["belief"],
               cmap = BELIEF_CMAP)

    ax.scatter([],[],color = BELIEF_CMAP(0), label = "
                             Left side")
    ax.scatter([],[],color = BELIEF_CMAP(128), label = "
                             Middle")
    ax.scatter([],[],color = BELIEF_CMAP(256), label = "
                             Right Side")
    ax.set_axis_off()
    plt.legend(loc = "best", prop={'size': 15})

    plt.show()
    return None


def plot_network(model):
    """ OpinionNetworkModel instance
    """
    # Load point df.
    belief_df = model.belief_df

    op = model.include_opinion
    wt = model.include_weight

    adjacency_df = model.adjacency_df
    adjacency_df.columns = [int(i) for i in adjacency_df
                                  .columns]

    cc = model.clustering_coefficient
    md = model.mean_degree
    # Plot people.
    fig, ax = plt.subplots(figsize = (8,8))
    ax.scatter(x = belief_df["x"],
               y = belief_df["y"],
               s = [int(w) for w in belief_df["weight"]
                                        .values],
               c = belief_df["belief"],
               cmap = BELIEF_CMAP)

    for j in belief_df.index:
        for k in np.where(adjacency_df.loc[j,:] == 1)[0]
                                       :
            ax.plot((belief_df.loc[j,"x"], belief_df.loc
                                         [k,"x"]),
                        (belief_df.loc[j,"y"],
                        belief_df.loc[k,"y"]),
                        color = "k", lw = .5,
                        zorder = 0)

    # Turn off axes
    ax.set_axis_off()

    # Add title
    title = "Connections based on Social Media"
    if op == True:
        if wt == True:
            title = "Connections based on Social Media,
                                         opinion
                                         proximity and
                                         weight"
        if wt == False:
            title = "Connections based on Social Media
                                         and opinion
                                         proximity"
    if op == False:
        if wt == True:
            title = "Connections based on Social Media
                                            and weight"
        if wt == False:
            title = "Connections based on Social Media"



    title = title + "\n clustering coefficient: " + str(
        np.around(cc, decimals = 3)) + "\n average in-
                                        degree: " + str(
        np.around(md, decimals = 1))
    ax.set_title(title)

    # Add legend
    ax.scatter([],[],color = BELIEF_CMAP(0), label = "
                                    Right Side")
    ax.scatter([],[],color = BELIEF_CMAP(128), label = "
                                    Middle")
    ax.scatter([],[],color = BELIEF_CMAP(256), label = "
                                    Left Side")
    plt.legend(loc = "best")
    plt.axis()

    plt.show()

    None


def get_ridge_plot(dynamic_belief_df,
                   phases = [],
                   reach_dict = None,
                   show_subplot_labels = True,
                   show_title = True):
    """ Ridgeplot of updating beliefs.

    Inputs:
        dynamic_belief_df: (dataframe) updating beliefs
                                        across multiple
                                        phases
        phases: (list) phases to show in plot.
        reach_dict: (dictionary) value is propotional
                                        reach of key.
        show_subplot_labels: (bool) if True show subplot
                                         labels.
        show_title: (bool) if True show plot title.

    Ouputs:
        Ridgeplot of updating belief distributions over
                                        phases.
    """
    if phases == []:
        phases = dynamic_belief_df.shape[1] - 1
        if phases < 5:
            phases = [t for t in range(phases +1)]
        else:
            t = phases // 5
            phases = [0] + [t * (i+1) for i in range(1,5
                                                   )]

    xx = np.linspace(-2,2,1000)
    gs = grid_spec.GridSpec(len(phases),1)
    fig = plt.figure(figsize=(8,4))

    i = 0

    ax_objs = []

    for p in range(len(phases)):
        ax_objs.append(fig.add_subplot(gs[i:i+1, 0:]))
        x = dynamic_belief_df[phases[p]].values
        kde = stats.gaussian_kde(x)
        ax_objs[-1].plot(xx, kde(xx), color = SIM_CMAP(0
                                               ))
        c = int((p * 256) / (len(phases) -1))
        ax_objs[-1].fill_between(xx,kde(xx), color=
                                               SIM_CMAP(c), alpha
                                                = 0.8)

        ax_objs[-1].set_yticks([])
        ax_objs[-1].set_yticklabels([])
        ax_objs[-1].set_ylabel('')

        #ax_objs[-1].set_axis_off()
        if show_subplot_labels == True:
            ax_objs[-1].text(2.1,0,"{} time steps".
                                                   format(phases[
                                                   p]),
                fontweight = "bold",
                fontsize=10,ha="left")

        # make background transparent
        rect = ax_objs[-1].patch
        rect.set_alpha(0)

        ax_objs[-1].set_xlim(-2,2)
        if i == len(phases)-1:
            ax_objs[-1].set_xticks([-1,0,1])
            ax_objs[-1].set_xticklabels([r"$\leftarrow$
                                                   Left Side",
                                                "Middle",
                                                r"Right Side
                                                $\rightarrow$"])
            ax_objs[-1].tick_params(axis='both', which='
                                                   both', length=
                                                   0)
        else:
            ax_objs[-1].set_xticks([])
            ax_objs[-1].set_xticklabels([])
        spines = ["top","right","left","bottom"]
        for s in spines:
            ax_objs[-1].spines[s].set_visible(False)

        i += 1


    gs.update(hspace=-0.7)
    left = int(reach_dict[-1] * 100)
    right = int(reach_dict[1] *100)
    if show_title == True:
        plt.title("Left Reach: {}%    Right Reach: {}%".
                                      format(left, right
                                      ),
            y=-.4, fontweight = "bold")

    return None


def get_line_plot_axis(ax, dynamic_belief_df):
    """ Get line plot on axis object.

    Input:
        ax: (axis object) on which to place alluvial
                                      plot.
        dynamic_belief_df: (dataframe) showing changing
                                      beliefs over time

    Return:
        Returns line plot of all agents evolving over
                                      time.
    """
    hfont = {'fontname':'DejaVu Sans'}
    m = dynamic_belief_df.iloc[:,0].min()
    M = dynamic_belief_df.iloc[:,0].max()
    cmap_linspace = np.linspace(m, M, 100)
    colors = plt.cm.viridis(cmap_linspace)

    for i in dynamic_belief_df.index:
        c = np.where(cmap_linspace <= dynamic_belief_df.
                                      iloc[i,0])[0][-1]
        ax.plot(dynamic_belief_df.loc[i,:], color =
                                      colors[c])
        ax.set_ylabel("Political Belief Spectrum",
                                      fontsize = 12, **
                                      hfont)
        ax.set_xlabel("Timesteps", fontsize = 12, **
                                      hfont)

    ax.spines.top.set_visible(False)
    ax.spines.right.set_visible(False)

    return ax
```

```
def _sigmoid(x):
    return 1 / (1 + np.exp(-x))

def _calc_sigmoid_line(width, y_start, y_end):
    xs = np.linspace(-5, 5, num=50)
    ys = np.array([_sigmoid(x) for x in xs])
    xs = xs / 10 + 0.5
    xs *= width
    ys = y_start + (y_end - y_start) * ys
    return xs, ys

def _get_wide_df(dynamic_belief_df,
                              right_leaning_threshold =
                              -1, left_leaning_threshold
                               = 0):
    df = pd.DataFrame(index = dynamic_belief_df.index)
    for c in dynamic_belief_df.columns:
        if c == 0:
            df["initial belief"] =  np.where(
                                        dynamic_belief_df
                                        .loc[:,c] >
                                        left_leaning_threshold
                                        , "
                                        left_leaning",
                                         "
                                        right_leaning"
                                        )
        if c == dynamic_belief_df.columns[-1]:
            df["final belief"] = "right_leaning"
            hes_idx = np.where(dynamic_belief_df.loc[:,c
                                        ] >
                                        left_leaning_threshold
                                        )[0]
            df.loc[hes_idx,"final belief"] =  "
                                        left_leaning"
            ##

    df["freq"] = 1

    wide_df = df[["initial belief", "final belief","freq
                                  "]].groupby(["initial
                                  belief", "final belief
                                  "], as_index=False).
                                  sum()
    wide_df["freq"] = wide_df["freq"] / wide_df["freq"].
                                  sum()

    return wide_df

def get_alluvial_plot_axis(ax, dynamic_belief_df,
                              right_leaning_threshold,
                         left_leaning_threshold):
    """ Return alluvial plot on axis.
```

```
Input:
    ax: (axis object) on which to place alluvial
                                 plot.
    dynamic_belief_df: (dataframe) showing changing
                                 beliefs over time
    start_right: (int) percentage of population that
                                  starts right
                                 leaning.
    right_leaning_threshold: (int) value below which
                                  individuals are
                                 considered
        left leaning; this is typically, model.
                                     threshold.
    hesitant_threshold: (int) this is the value
                                 above which
                                 individuals are
        considered right leaning.
    figure_name: (str) filepath to save figure.

Output:
    Alluvial plot showing right,middle and left
                                 leaning beliefs
                                 over time

"""
wide_df = _get_wide_df(dynamic_belief_df,
                             right_leaning_threshold
                             ,
                             left_leaning_threshold
                             )
hfont = {'fontname':'DejaVu Sans'}
cmap = {"left_leaning":"#DDE318","right_leaning":"
                             silver"}
y_end = 0
y_left_start = 0
y_right_start = wide_df[wide_df["initial belief"] ==
                              "left_leaning"]["freq
                             "].sum() + .1
for final in ["left_leaning","right_leaning"]:

    for initial in ["left_leaning","right_leaning"]:
        idx = wide_df[(wide_df["final belief"] ==
                                     final)&(
            wide_df["initial belief"] == initial)].
                                         index
        if len(idx) > 0:
            idx = idx[0]
            height = wide_df.loc[idx,"freq"]
            if initial == "left_leaning":

                xs, ys_under = _calc_sigmoid_line(1,
                                              y_start
```

```
                                             y_left_start


                                             y_end =
                                             y_end)
                y_left_start += wide_df.loc[idx,"
                                              freq"]

            if initial == "right_leaning":

                xs, ys_under = _calc_sigmoid_line(1,
                                            y_start
                                                                                                    =

                                            y_right_start
                                                                                                    ,

                                            y_end =
                                            y_end)
                y_right_start +=  wide_df.loc[idx,"
                                              freq"]

            ax.plot(xs, ys_under, color = cmap[final
                                        ], alpha =
                                         .7,
                                        zorder = 0
                                        )
            ax.fill_between(xs, ys_under, ys_under +
                                         height,
                                        color =
                                        cmap[final
                                        ],
                            alpha = .7, zorder = 0)

            y_end += height

        y_end += .1

    ## Right hand annotations
    left_rect_height = wide_df[wide_df["final belief"] =
                                  = "left_leaning"
                                ]["freq"].sum()
    ax.add_patch(Rectangle((1, 0),
                            .05,
                            left_rect_height,
                            edgecolor =
                            cmap["left_leaning"],
                            facecolor =
                            cmap["left_leaning"],
                            lw = 2,
                            zorder = 1))
    text = f"{np.around(100 * left_rect_height, decimals
                                          = 1)}%"
    ax.annotate(text = text,
                xy = (1.07,.5* left_rect_height),
                fontsize = 12,
                ha = "left",
                va = "center",
                **hfont)

    right_rect_height = wide_df[wide_df["final belief"]
                                    == "right_leaning"
                                     ]["freq"].sum()
    ax.add_patch(Rectangle((1, left_rect_height + .1),
                            .05,
                            right_rect_height,
                            edgecolor =
                            cmap["right_leaning"],
                            facecolor =
                            cmap["right_leaning"],
                            lw = 2,
                            zorder = 1))
    text = f"{np.around(100 * right_rect_height,
                                    decimals = 1)}%"
    ax.annotate(text = text,
                xy = (1.07,left_rect_height + .1 + .5*
                                                    right_rect_height
                                                    ),
                fontsize = 12,
                ha = "left",
                va = "center",
                **hfont)

   ##

    right_rect_height = wide_df[wide_df["initial belief"
                                    ] == "right_leaning"][
                                    "freq"].sum()
    ax.add_patch(Rectangle((0, left_rect_height + .1),
                            .05,
                            right_rect_height,
                            edgecolor =
                            cmap["right_leaning"],
                            facecolor =
                            cmap["right_leaning"],
                            lw = 2,
                            zorder = 1))

    ##

    ax.spines.top.set_visible(False)
    ax.spines.left.set_visible(False)
    ax.spines.right.set_visible(False)
    ax.set_yticks([])
    ax.set_xticks([0 + 0.025,1 + 0.025])
    ax.set_xticklabels(["initial","final"], fontsize =
                                                12, **hfont)

    return ax
```

- The code below runs the simulation along with creating the initial Opinion Dynamics Network. Initially the first half of the code gets run through creating the ODyN model. It adds agents connections, beliefs, and more to the model. Once the initial model is created, the second half of the code runs the simulation aspect where the agents are changing their beliefs based on who they are connected to. These beliefs get stored and then run through the visualization code to be graphed to see the change in opinions over time. The code was initially created initially created by Anna Haensch for another project and alter for our project(*5*).

```
import pandas as pd
import numpy as np
import logging
import os
import geojson
import random
import decimal
import math
import itertools

import geopandas as gpd
from geopy.geocoders import Nominatim
from shapely.geometry import Point, Polygon,
                              MultiPolygon, shape
import shapely.ops
from pyproj import Proj

from bs4 import BeautifulSoup
import requests
from abc import ABC, abstractmethod

from .geolocations import *
from .visualizations import *

logging.basicConfig(level=logging.WARNING)

class OpinionNetworkModel(ABC):
    """ Abstract base class for network model """

    def __init__(self,
                probabilities = [.45,.1,.45],
                power_law_exponent = 1.5,
                openness_to_neighbors = 1.5,
                openness_to_influencers = 1.5,
                distance_scaling_factor = 1/10,
                importance_of_weight = 1.6,
                importance_of_distance = 8.5,
        include_opinion = True,
        include_weight = True,
        include_distance = True,
        left_reach = 0.8,
        right_reach = 0.8,
        threshold = -10
        ):
"""Returns initialized OpinionNetworkModel.

Inputs:
    probabilities: (list) probabilities of each
                              mode; these
                              are the
        values "p_0,p_1,p_2" from [1].
    density: (int) number of agents per unit
                              square.
    model: (str) either "triangular" or "
                              spherical".
    power_law_exponent: (float) exponent of
                              power law,
                              must be > 0;
        this is "gamma" from [1].
    openness_to_neighbors: (float) maximum inter
                              -mode distance
                               that agents
        can influence; this is "b" from [1].
    openness_to_influencers: (float) distance in
                               opinion space
                               that
        mega-influencers can reach; this is "
                                  epsilon"
                                  from [1].
    distance_scaling_factor: (float) Scale
                              distancy by
                              this amount,
                              must
        be >0; this is "lambda" from [1].
    importance_of_weight: (float) Raise weights
                              to this power,
                               must be > 0;
        this is "alpha" from [1].
    importance_of_distance: (float) Raise
                              adjusted
                              distance to
                              this power,
        must be > 0; this is "delta" from [1].
    include_opinion: (boolean) If True, include
                              distance in
                              opinion space
        in the probability measure.
    include_weight: (boolean) If True, include
                              influencer
                              weight in the
        probability measure.
        include_distance: (boolean) If True, include
                                    pairwise
                                    distance in
                                    the
            probability measure.
        left_reach: (float) this is the proportion
                                    of the
                                    susceptible
                                    population
            that the left mega-influencers will
                                        actually
                                        reach,
                                        must be
                                        between
            0 and 1; this is p_L from [1]
        right_reach: (float) this is the proportion
                                    of the
                                    susceptible
                                    population
            that the right mega-influencers will
                                        actually
                                        reach,
                                        must be
                                        between
            0 and 1; this is p_R from [1]
        threshold: (int) value below which opinions
                                    no longer
                                    change.

    Outputs:
        Fully initialized OpinionNetwork instance.
    """
    self.probabilities = probabilities
    self.power_law_exponent = power_law_exponent
    self.openness_to_neighbors =
                              openness_to_neighbors

    self.openness_to_influencers =
                              openness_to_influencers

    self.distance_scaling_factor =
                              distance_scaling_factor

    self.importance_of_weight = importance_of_weight
    self.importance_of_distance =
                              importance_of_distance

    self.include_opinion = include_opinion
    self.include_weight = include_weight
    self.include_distance = include_distance
    self.left_reach = left_reach
    self.right_reach = right_reach
    self.threshold = threshold
```

```
        self.agent_df = None
        self.belief_df = None
        self.prob_df = None
        self.adjacency_df = None
        self.mega_influencer_df = None
        self.clustering_coefficient = 0
        self.mean_degree = 0

    def populate_model(self, num_agents = None,
                                  show_plot = False):
        """ Fully initialized but untrained
                                      OpinionNetworkModel
                                       instance.

        Input:
            num_agents: (int) number of agents to plot.
            show_plot: (bool) if true then plot is shown
                                             .

        Output:
            OpinionNetworkModel instance.
        """
        agent_df = self.add_random_agents_to_triangle(
                                      num_agents =
                                      num_agents,
                                            show_plot =
                                                                                False
                                                                                )


        logging.info("\n {} agents added.".format(
                                      agent_df.shape[0])
                                      )

        belief_df = self.assign_weights_and_beliefs(
                                      agent_df)
        logging.info("\n Weights and beliefs assigned.")

        prob_df = self.compute_probability_array(
                                      belief_df)
        adjacency_df = self.compute_adjacency(prob_df)
        logging.info("\n Adjacencies computed.")

        # Connect mega-influencers
        mega_influencer_df = self.
                                      connect_mega_influencers
                                      (belief_df)

        # Compute network statistics.
        logging.info("\n Computing network statistics...
                                      ")
        cc, md = self.compute_network_stats(adjacency_df
                                      )
        logging.info("\n Clustering Coefficient: {}".
                                          format(cc))
        logging.info("\n Mean Degree: {}".format(md))

        self.agent_df = agent_df
        self.belief_df = belief_df
        self.prob_df = prob_df
        self.adjacency_df = adjacency_df
        self.mega_influencer_df = mega_influencer_df
        self.clustering_coefficient = cc
        self.mean_degree = md

        if show_plot == True:
            self.plot_initial_network()

        return None

    def populate_model_with_geography(self, num_agents =
                                       None, geo_df = None,
                                      bounding_box = None,
                                      show_plot = False):
        """ Fully initialized but untrained
                                          OpinionNetworkModel
                                           instance.

        Input:
            num_agents: (int) number of agents to plot.
            geo_df: (dataframe) geographic datatframe
                                              including
                                              county
                                              geometry.
            bounding_box: (list) list of 4 vertices
                                              determining a
                                              bounding box
                where agents are to be added.  If no box
                                                    is given,
                                                    agents
                                                   are added
                to a random triangle.
            show_plot: (bool) if true then plot is shown
                                              .

        Output:
            OpinionNetworkModel instance where underying
                                               population
                                              has
            geopspatial connections.
        """
        if bounding_box is None:
            agent_df =
    self.add_random_agents_to_triangle_with_geography(
                        num_agents =
                        num_agents,
                        geo_df = geo_df,
                        show_plot = False)
        else:
            if geo_df is None:
                raise ValueError("If a bounding box is
                                               specified,
                                                then a "
                    "geo_df must also be given.")
            agent_df =
    self.add_random_agents_to_triangles_with_geography(
                            geo_df = geo_df,
                            bounding_box =
                            bounding_box,
                            show_plot = False)

        logging.info("\n {} agents added.".format(
                                       agent_df.shape[0])
                                       )

        belief_df = self.assign_weights_and_beliefs(
                                       agent_df)
        logging.info("\n Weights and beliefs assigned.")

        prob_df = self.compute_probability_array(
                                       belief_df)
        adjacency_df = self.compute_adjacency(prob_df)
        logging.info("\n Adjacencies computed.")

        # Connect mega-influencers
        mega_influencer_df = self.
                                       connect_mega_influencers
                                       (belief_df)

        # Compute network statistics.
        logging.info("\n Computing network statistics...
                                       ")
        cc, md = self.compute_network_stats(adjacency_df
                                       )
        logging.info("\n Clustering Coefficient: {}".
                                       format(cc))
        logging.info("\n Mean Degree: {}".format(md))

        self.agent_df = agent_df
        self.belief_df = belief_df
        self.prob_df = prob_df
        self.adjacency_df = adjacency_df
        self.mega_influencer_df = mega_influencer_df
        self.clustering_coefficient = cc
        self.mean_degree = md

        if show_plot == True:
            self.plot_initial_network()

        return None

    def plot_initial_network(self):
        plot_network(self)
        return None

    def add_random_agents_to_triangle(self, num_agents,
                                      show_plot = False):
        """ Assign N points on a triangle using Poisson
                                              point process.

        Input:
            num_agents: (int) number of agents to add to
                                              the triangle.
                                               If None,
                then agents are added according to
                                                 density.
            show_plot: (bool) if true then plot is shown
                                             .

        Returns:
            An num_agents x 2 dataframe of point
                                             coordinates.
        """
        v1 = [0,0]
        v2 = [1000,0]
        v3 = [.5 * 1000, .5 * np.sqrt(3) * 1000]

        triangle_object = Polygon([v1, v2, v3, v1])
        bnd = list(triangle_object.boundary.coords)

        # Get Vertices
        V1 = np.array(bnd[0])
        V2 = np.array(bnd[1])
        V3 = np.array(bnd[2])

        # Sample from uniform distribution on [0,1]
        U = np.random.uniform(0,1,num_agents)
        V = np.random.uniform(0,1,num_agents)

        UU = np.where(U + V > 1, 1-U, U)
        VV = np.where(U + V > 1, 1-V, V)

        # Shift triangle into origin and and place
                                              points.
        agents = (UU.reshape(len(UU),-1) * (V2 - V1).
                                              reshape(-1,2))
        + (VV.reshape(len(VV),-1) * (V3 -
                V1).reshape(-1,2))

        # Shift points back to original position.
        agents = agents + V1.reshape(-1,2)
        agent_df = pd.DataFrame(agents, columns = ["x","
                                              y"])

        if show_plot == True:
            plot_agents_on_triangle(triangle_object,
                                                    agent_df)

        return agent_df

    def add_random_agents_to_triangle_with_geography(
                                 self, num_agents,
                                 geo_df = None,
                                 triangle_object = None
                                 ,
        show_plot = False):
        """ Assign N points on a triangle using Poisson
                                       point process.

        Input:
            num_agents: (int) number of agents to add to
                                             the triangle.
                                              If None,
                then agents are added according to
                                               density.
            geo_df: (dataframe) geographic datatframe
                                            including
                                            county
                                            geometry.
            triangle_object: (Polygon) bounded
                                            triangular
                                            region to be
                                            populated.
            show_plot: (bool) if true then plot is shown
                                            .

        Returns:
            An num_agents x 2 dataframe of point
                                            coordinates
                                            where point
                                            values
            have an underlying geospatial meaning.
        """
        if triangle_object is None:
            # If no triangle is given, initialize
                                            triangle with
                                            area 1 km^2.
            triangle_object = Polygon([[0,0],[1419,0], [
                                            1419/2,1419],[
                                            0,0]])

            # If density is specified, adjust triangle
                                            size.
            if geo_df is not None:
                density = geo_df.loc[0,"density"]
                b = 1419 * (num_agents/density) ** (1/2)
                triangle_object = Polygon([[0,0],[b,0],
                                               [b/2,b],[0
                                               ,0]])
        bnd = list(triangle_object.boundary.coords)
        gdf = gpd.GeoDataFrame(geometry = [
                                       triangle_object])

        # Establish initial CRS
        gdf.crs = "EPSG:3857"

        # Set CRS to lat/lon
        gdf = gdf.to_crs(epsg=4326)

        # Extract coordinates
        co = list(gdf.loc[0,"geometry"].exterior.coords)
        lon, lat = zip(*co)
        pa = Proj(
            "+proj=aea +lat_1=37.0 +lat_2=41.0 +lat_0=39
                                            .0 +lon_0=-106
                                            .55")
        x, y = pa(lon, lat)
        coord_proj = {"type": "Polygon", "coordinates":
                                       [zip(x, y)]}
        area = shape(coord_proj).area / (10 ** 6) # area
                                   in km^2

        # Get Vertices
        V1 = np.array(bnd[0])
        V2 = np.array(bnd[1])
        V3 = np.array(bnd[2])

        # Sample from uniform distribution on [0,1]
        U = np.random.uniform(0,1,num_agents)
        V = np.random.uniform(0,1,num_agents)

        UU = np.where(U + V > 1, 1-U, U)
        VV = np.where(U + V > 1, 1-V, V)

        # Shift triangle into origin and and place
                                       points.
        agents = (UU.reshape(len(UU),-1) * (V2 - V1).
                                       reshape(-1,2))
        + (VV.reshape(len(VV),-1) * (V3 -
                V1).reshape(-1,2))

        # Shift points back to original position.
        agents = agents + V1.reshape(-1,2)
        agent_df = pd.DataFrame(agents, columns = ["x","
                                       y"])

        if show_plot == True:
            plot_agents_on_triangle(triangle_object,
                                            agent_df)

        return agent_df

    def add_random_agents_to_triangles_with_geography(
                                       self, geo_df,
                                       bounding_box = None,
                                       show_plot = False):
        """ Plots county with triangular regions with
                                            correspding
                                            geospatial info.

        Inputs:
            geo_df: (dataframe) geographic datatframe
                                            including
                                            county
                                            geometry.
            bounding_box: (list) list of 4 vertices
                                            determining a
                                            bounding box
                where agents are to be added.  If no box
                                                 is given,
                                                 then the
                bounding box is taken as the envelope of
                                                 the
                                                county.
            show_plot: (bool) if true then plot is shown
                                            .

        Returns:
            Populated triangles in specified county
                                            enclosed in
                                            the given
            bounding box where regions are filled with
                                            proper density
                                             using a
                                            Poisson
            point process.
        """
        tri_dict = make_triangulation(geo_df)
        tri_df = gpd.GeoDataFrame({"geometry":[Polygon(t
                                       ) for t in
                                       tri_dict["geometry
                                       "]["coordinates"]]
                                       })

        # Establish initial CRS
        tri_df.crs = "EPSG:3857"

        # Set CRS to lat/lon.
        tri_df = tri_df.to_crs(epsg=4326)

        # Get triangles within bounding box.
        if bounding_box is None:
            geo_df.crs = "EPSG:3857"
            geo_df = geo_df.to_crs(epsg=4326)
            sq_df = gpd.GeoDataFrame(geo_df["geometry"])
        else:
            sq_df = gpd.GeoDataFrame({"geometry":[
                                                       Polygon(
                                                       bounding_box)]
                                                       })
              inset = [i for i in tri_df.index if tri_df.loc[i
                                                  ,"geometry"].
                                                  within(sq_df.loc[0
                                                  ,"geometry"])]

              # Load triangle area.
              agent_df = pd.DataFrame()
              for i in inset:
                  co = list(tri_df.loc[i,"geometry"].exterior.
                                                       coords)
                  lon, lat = zip(*co)
                  pa = Proj(
                      "+proj=aea +lat_1=37.0 +lat_2=41.0 +
                                                            lat_0=39.0
                                                             +lon_0=-
                                                            106.55")
                  x, y = pa(lon, lat)
                  coord_proj = {"type": "Polygon", "
                                                       coordinates":
                                                       [zip(x, y)]}
                  area = shape(coord_proj).area / (10 ** 6) #
                                                       area in km^2
                  num_agents = int(area * geo_df.loc[0,"
                                                       density"])
                  df = pd.DataFrame(columns = ["x","y"])
                  if num_agents > 0:
                      df =
    self.add_random_agents_to_triangle_with_geography(
                                       num_agents,
                               geo_df = geo_df,
                               triangle_object =
                               tri_df.loc[i,"geometry"],
                               show_plot = False)
                  agent_df = pd.concat([agent_df,df])

              agent_df.reset_index(drop = True, inplace = True
                                                  )

              # Plot triangles.
              if show_plot == True:
                  fig, ax = plt.subplots(figsize = (10,10))
                  tri_df.loc[inset,:].boundary.plot(ax = ax,
                                                       alpha=1,
                               linewidth = 3,
                               edgecolor = COLORS["light_blue"]
                                                              )

                  ax.scatter(agent_df["x"], agent_df["y"], s =
                                                        3)
                  ax.set_axis_off()
                  ax.set_aspect(.9)
```

```
            plt.show()

        return agent_df

    def assign_weights_and_beliefs(self, agent_df,
                                   show_plot = False):
        """ Assign weights and beliefs (i.e. modes)
                                    according to
                                    probabilities.

        Inputs:
            agent_df: (dataframe) xy-coordinates for
                                        agents.
            show_plot: (bool) if true then plot is shown
                                        .

        Returns:
            Dataframe with xy-coordinates, beliefs, and
                                        weights for
                                        each point.
        """
        belief_df = agent_df.copy()
        power_law_exponent = self.power_law_exponent
        k = -1/(power_law_exponent)


        assert np.sum(np.array(self.probabilities)) == 1
                                    , "Probabilities
                                    must sum to 1."

        belief_df["weight"] = np.random.uniform(0,1,
                                    belief_df.shape[0]
                                    ) ** (k)

        means = np.linspace(-1,1,len(self.probabilities)
                                    )

        components = [i for i in range(len(means))]
        cov = 0.5#np.abs((means[1] - means[0])/2)
        # Choose gmm component
        gmm_component = np.random.choice(components,
                                    belief_df.shape[0]
                                    , p = self.
                                    probabilities)
        x1 = np.random.normal(-.9,.25,len(agent_df)//2)
                                    #.9,.25
        for i in range(len(agent_df)//2):
            if x1[i] < -1:
                while x1[i] < -1:
                    x1[i] = np.random.normal(-.9, .25)
        x2 = np.random.normal(.9, .25, len(agent_df)//2)
        for j in range(len(agent_df)//2):
            if x2[j] > 1:
                while x2[j] > 1:
```

```
                x2[j]=np.random.normal(0.9, .25)
    X = np.concatenate([x1, x2])
    plt.hist(X)
    plt.show()
    # Sample from gaussians by component
    belief = X
    #[np.random.normal(loc = means[c], scale = cov)
                                for c in
                                gmm_component]
    #[np.random.uniform(-1,1) for c in gmm_component
                                ]#This can be used
                                 for uniform
                                distribution
    #[np.random.normal(loc = means[c], scale = cov)
                                for c in
                                gmm_component]#
                                This can be used
                                for normal
                                distribution
                                        belief_df
                                ["belief"] =
                                belief
    belief_df["decile"] = pd.qcut(belief_df["weight"
                                ], q = 100, labels
                                 = [
                    i for i in range(1,101)])

    if show_plot == True:
        plot_agents_with_belief_and_weight(belief_df
                                    )
    return belief_df


def compute_probability_array(self, belief_df):
    """ Return dataframe of probability that row n
                                influences column
                                m.

    Inputs:
        belief_df: (dataframe) xy-coordinats,
                                    beliefs and
                                    weights of
            agents.

    Returns:
        Dataframe with the probability that agent n
                                    influces agent
                                     m
        in row n column m.
    """
    n = belief_df.index.shape[0]
    prob_array = np.ones((n,n))

    # Prioritize connections to people close in
```

```
                                         physical space.
if self.include_distance == True:
    dist_array = np.zeros((n,n))
    for i in range(n):
        point_i = Point(belief_df.loc[i,"x"],
                                       belief_df.
                                       loc[i,"y"]
                                       )

        # Get distances from i to each other
                                       point.
        dist_array[i,:] = [point_i.distance(
                                       Point(
                                       belief_df.
                                       loc[j,"x"]
                                       ,
            belief_df.loc[j,"y"])) for j in
                                           belief_df
                                           .index
                                           ]

    # Compute the dimensionless distance metric.
    diam = dist_array.max().max()
    lam = (self.distance_scaling_factor * diam)
    delta = -1 * self.importance_of_distance

    dist_array = np.where(dist_array == 0,np.nan
                                     , dist_array)
    dist_array = (1 + (dist_array/lam)) ** delta

    prob_array = prob_array * dist_array

# Only allow connections to people close in
                               opinion space.
if self.include_opinion == True:
    op_array = np.zeros((n,n))
    # If row i connects to column j, that means
                                   person i is
    # influencing person j.  This can only
                                   happen if j is
    # already sufficiently close to person i in
                                   opinion space.
    for i in range(n):
        i_current_belief = belief_df.loc[i,"
                                       belief"]
        opinion_diff = np.where(
            np.abs(belief_df["belief"] -
                                           i_current_belief

                ) > self.openness_to_neighbors,0
                                              ,
                                              1)
        op_array[i,:] = opinion_diff
```

```
            prob_array = prob_array * op_array


        # Incentivize connections with heavily weighted
                                    people.
        if self.include_weight == True:
            wt_array = (belief_df["weight"] - belief_df[
                                        "weight"].min
                                        ()) ** self.
                                        importance_of_weight

            wt_array = wt_array.values.reshape(-1,1)

            prob_array = prob_array * wt_array

        prob_df = pd.DataFrame(prob_array)
        prob_df = prob_df.clip(upper = 1)
        prob_df = prob_df.replace(np.nan,0)

        return prob_df

    def compute_adjacency(self, prob_df):
        """ Compute NxN adjacency dataframe.

        Inputs:
            prob_df: (dataframe) num_agents x num_agents
                                        dataframe
                                        giving
                probabiltiy of influce row n on column m
                                        .

        Outputs:
            Dataframe where row n and column m is a 1 if
                                        n influences
                                        m,
            and a 0 otherwise.
        """
        adjacency_df = pd.DataFrame(0,index = [int(i)
                                    for i in prob_df.
                                    index],
            columns = [int(i) for i in prob_df.columns])
        U = np.random.uniform(0,1,(prob_df.shape[0],
                                    prob_df.shape[1]))
        adjacency_df = pd.DataFrame(np.where(U< prob_df.
                                    values,1,0))

        return adjacency_df

    def compute_network_stats(self, adjacency_df):
        """ Return clustering coefficient and mean
                                    degree.

        Inputs:
            adjacency_df: (dataframe) num_agents x
```

```
                                          num_agents
                                          dataframe,
                                          where
              a 1 in row n column m indicates that
                                              agent n
                                              influences
                                               agent m.

        Returns:
            Tuple of clustering coefficient and mean
                                          degree of
            the network.
        """
        cc = 0
        degrees = []
        for i in adjacency_df.index:
            nbhd = np.where(adjacency_df.loc[:,i] != 0)[
                                          0]
            deg = len(nbhd)
            degrees.append(deg)
            cc_i = 0
            if deg > 2:
                C = list(itertools.permutations(nbhd,2))
                cc_i = np.sum([int(adjacency_df.loc[c[0]
                                              ,c[1]] > 0
                                              ) for c in
                                               C]
                          ) / math.perm(deg,2)


            cc += cc_i
        clustering_coefficient = cc / adjacency_df.shape
                                      [0]

        return clustering_coefficient, np.mean(degrees)

    def connect_mega_influencers(self, belief_df):
        """ Returns mega_influencer reach dataframe

        Inputs:
            belief_df: (dataframe) xy-coordinats,
                                          beliefs and
                                          weights of
                agents.

        Outputs:
            Dataframe where rows are the keys of reach
                                          dict and
                                          columns
            are the index of belief_df, where a 1 in row
                                           n column m
                                          indicates
            that influencer n reaches agent m.
        """
        reach_dict = {-1:self.left_reach, 1:self.
                                    right_reach}
        mega_influencer_df = pd.DataFrame(0, index = [-1
                                    ,1], columns =
                                    list(
            belief_df.index))

        for i in mega_influencer_df.index:
            ind = np.where(np.abs(belief_df["belief"] -
                                        i) <= self.
                                        openness_to_influencers
                                        )[0]
            U = np.random.uniform(0,1,len(ind))
            mega_influencer_df.loc[i,ind[np.where(U <
                                        reach_dict[i])
                                        [0]]] = 1

        return mega_influencer_df


class NetworkSimulation(ABC):
    """ Abstract base class for learning algorithm.
    """
    def __init__(self):
        self.model = None
        self.stopping_thresh = None
        self.results = []
        self.dynamic_belief_df = None

    def run_simulation(self, model, stopping_thresh = .
                                01, show_plot = False,
                                 store_results = False
                                ):
        """ Carry out simulation.

        Inputs:
            model: OpinionNetworkModel instance.
            stopping_thresh: (float) threshold for
                                        stopping
                                        criterion.
            show_plot: (bool) if true, shows ridge plot.
            store_results: (bool) if True, stores
                                        results for
                                        all phases, if
                                         False
                then only updating beliefs are stored.

        Outputs:
            Complete simulation.
        """

        results = []
        self.model = model
        new_belief_df = model.belief_df.copy()
```

```
new_adjacency_df = model.adjacency_df.copy()
new_mega_influencer_df = model.
                              mega_influencer_df
                              .copy()

df = pd.DataFrame()
df[0] = new_belief_df["belief"].values

self.stopping_thresh = stopping_thresh
stopping_df = pd.DataFrame()

i = 0
while True:
    phase_dict = {}
    new_belief_df, new_mega_influencer_df = self
                                  .
                                  one_dynamics_iteration
                                  (
            belief_df = new_belief_df,
            adjacency_df = new_adjacency_df,
            openness_to_influencers =
            model.openness_to_influencers,
            mega_influencer_df =
            new_mega_influencer_df,
            threshold = model.threshold,
            iteration_number = i)

    phase_dict["belief_df"] = new_belief_df
    prob_df = model.compute_probability_array(
                                  new_belief_df)
    new_adjacency_df = model.compute_adjacency(
                                  prob_df)
    new_adjacency_df.columns = [int(i) for i in
                                  new_adjacency_df
                                  .columns]
    phase_dict["adjacency_df"] =
                                  new_adjacency_df

    clust_coeff, mean_degree = model.
                                  compute_network_stats
                                  (
                                  new_adjacency_df
                                  )
    phase_dict["clust_coeff"] = clust_coeff
    phase_dict["mean_degree"] = mean_degree

    results.append(phase_dict)
    df[i+1] = new_belief_df["belief"].values

    stopping_df[i] = df.iloc[:,i+1] - df.iloc[:,
                                  i]
    i = i+1
    # Check that at least 5 iterations have been
                                   carried out.
            if stopping_df.shape[1] > 5:
            # If rolling average change is less than
                                                  stopping_thresh
                                                  , break.
                if stopping_df.rolling(window = 5, axis =
                                                      1).mean().
                                                     iloc[:,-1
                                           ].abs().mean() <
                                           stopping_thresh:
                    break

        self.dynamic_belief_df = df

        if show_plot == True:
            self.plot_simulation_results()
        if store_results == True:
            self.results = results

        return None

    def plot_simulation_results(self):
        """ Return ridgeplot of simulation.
        """
        df = self.dynamic_belief_df
        phases = df.shape[1] - 1
        if df is None:
            raise ValueError("It looks like the
                                                  simulation
                                                  hasn't been
                                                  run yet.")
        if phases < 5:
            plot_phases = [t for t in range(phases +1)]
        else:
            t = phases // 5
            plot_phases = [0] + [t * (i+1) for i in
                                                  range(1,5)]
        get_ridge_plot(df, plot_phases, reach_dict = {-1
                                              :self.model.
                                              left_reach,
                                   1:self.model.right_reach})
        return None


    def one_dynamics_iteration(self,
                               belief_df,
                               adjacency_df,
                               openness_to_influencers,
                               mega_influencer_df,
                               threshold,
                               iteration_number):
        """ Returns updated belief_df.
        Inputs:
            belief_df: (dataframe)
            adjacency_df: (dataframe)
        openness_to_influencers: (float) distance in
                                         opinion space
                                         that
            mega-influencers can reach; this is "
                                            epsilon"
                                            from [1].
        mega_influencer_df: (dataframe) rows show
                                        connection to
                                        mega-
                                        influencers
            as binary values.
        threshold: (int) value below which opinions
                                        no longer
                                        change.
        iteration_number: (int) counter

    Returns:
        Updated belief_df after one round of
                                        Hegselmann-
                                        Krause.
    """
    df_new = belief_df.copy()
    new_mega_influencer_df = mega_influencer_df.copy
                                    ()

    for i in belief_df.index:
        current_belief = belief_df.loc[belief_df.
                                        index[i], "
                                        belief"]

        # After 10 iterations, only update agents
                                        with beliefs
                                        above
                                        threshold.
        if (current_belief < threshold) & (
                                        iteration_number
                                         >= 10):
            pass

        else:
         # Sum over column
         edges = np.where(adjacency_df[i] == 1)[0]
         n_edges = len(edges)

         new_belief = np.sum(belief_df.loc[edges,"
                                         belief"]) +
                                         current_belief


         if openness_to_influencers > 0:
         # Am  I connected to the right mega-
                                         influencer?
         if mega_influencer_df.loc[1,i] == 1:
         # Do I listen to them?
```

```
              if np.abs(current_belief - 1) <=
                                           openness_to_influencers
                                           :
                 n_edges = n_edges + 1
                 new_belief = new_belief + 1
                 # If not, flip a biased coin to decide
                                              if my
                                              influencer

                                              connection
                 # changes affiliation.
                  else:
                       u = np.random.uniform(0,1)
                       if u < self.model.left_reach:
                         new_mega_influencer_df.loc[1,i]
                                                        =
                                                        0
                         new_mega_influencer_df.loc[-1,i]
                                                         =
                                                         1

                 # Am  I connected to the left mega-
                                              influencer
                                              ?
                 if mega_influencer_df.loc[-1,i] == 1:
                     # Do I listen to them?
                     if np.abs(current_belief  - (-1)) <=

                                                  openness_to_influencers
                                                  :
                           n_edges = n_edges + 1
                           new_belief =
                           new_belief + (-1)
                     # If not, flip a biased coin to
                                                  decide
                                                   if my
                     influencer connection
                     # changes affiliation.
                     else:
                       u = np.random.uniform(0,1)
                       if u < self.model.right_reach:
                         new_mega_influencer_df.loc[-1,i]
                                                        =0
                         new_mega_influencer_df.loc[1,i]
                                                        =
                                                        1

                 new_belief = new_belief / (n_edges + 1)

                 df_new.loc[i,"belief"] = new_belief
         print(mega_influencer_df)
         return df_new, new_mega_influencer_df
```

8.2 Appendix B: Adaptive DeGroot Code

- The simulation is structured into three key sections:Initialization and Setup, Score Updates and Influence Matrix Adjustment, and Simulation Execution and Analysis. Below is the explanation along with the corresponding code for each section.
- a).**Initialization and Setup**
  In this section, the initial conditions for the simulation are established.The `generate_gaussian_scores` function is used to create initial scores for each voter based on a Gaussian distribution. These scores are clipped to fall within the range $[0, 10]$ and are then normalized so that the total score for each voter sums to 10.The `generate_influence_matrix_gaussian` function creates an influence matrix, which represents how much each voter influences others. This matrix is also based on a Gaussian distribution, with connections between voters determined randomly.The matrix is normalized so that the influence of each voter on others sums to 1.These initial scores and the influence matrix set the stage for the simulation, defining the initial voter preferences and the dynamics of influence.

```
python
def generate_gaussian_scores(num_voters,
                              num_candidates, mean=5
                              , std_dev=2):
    scores = np.random.normal(loc=mean, scale=
                                  std_dev, size=(
                                  num_voters,
                                  num_candidates))
    # Clip values to be between 0 and 10
    scores = np.clip(scores, 0, 10)
    # Normalize to ensure the sum is 10
    return 10 * scores / scores.sum(axis=1, keepdims
                                  =True)

def generate_influence_matrix_gaussian(num_voters,
                              average_connections,
                              mean=0, std_dev=1):
    if average_connections > num_voters:
        raise ValueError("Average connections cannot
                                   be greater
                                  than the
                                  number of
                                  voters.")

    influence_matrix = np.zeros((num_voters,
                                  num_voters))   for
                                   i in range(
                                  num_voters):
                                        connections
                                  = random.sample(
                                          range(num_voters),

                                          average_connections
                                          )
                                          influence_values =
                                           np.random.normal(
                                          loc=mean, scale=
                                          std_dev, size=
                                          average_connections
                                          )
         # Clip values to be between 0 and 1 and avoid
                                             division by zero
    influence_values = np.clip(influence_values, 0,
                                          1)
        influence_matrix[i, connections] =
                                              influence_values

    return influence_matrix / influence_matrix.sum(
                                          axis=1, keepdims=
                                          True)
```

– b).**Score Updates and Influence Matrix Adjustment**
This section focuses on the iterative process of updating the scores of voters and adjusting the influence matrix based on these updates.The `update_scores` function recalculates the scores of each voter by applying the influence matrix, simulating the impact of peer influence. To introduce a realistic environment, the `apply_random_event` function occasionally alters a voter's score for a particular candidate, introducing random events that might affect voter preferences.The `calculate_score_changes` function then measures the changes in scores between iterations, which informs how the influence matrix should be updated. The `update_influence_matrix` function modifies the influence matrix to reflect these changes, making it adaptive to the evolving voter dynamics. Lastly, the `adjust_influence_matrix` function tunes the matrix by emphasizing or reducing the most significant changes, ensuring stability in the simulation over time.

```
python
def update_scores(scores, influence_matrix):
    return np.dot(influence_matrix, scores)

def apply_random_event(scores):
    random_voter = random.randint(0, scores.shape[0]
                                           - 1)
    random_candidate = random.randint(0, scores.
                                          shape[1] - 1)
    scores[random_voter, random_candidate] *= (1 +
                                          random.uniform(-0.
                                          1, 0.1))
    scores[random_voter] = normalize_scores(scores[
                              random_voter])
    return scores

def calculate_score_changes(initial_scores,
                           new_scores):
    return np.abs(new_scores - initial_scores).sum(
                              axis=1)

def update_influence_matrix(influence_matrix,
                           score_changes):
    new_influence_matrix = np.zeros_like(
                              influence_matrix)
    for i in range(influence_matrix.shape[0]):
  for j in range(influence_matrix.shape[1]):
     new_influence_matrix[i, j] = influence_matrix[i
                               , j] *
                               score_changes[j]
                                   # Normalize
                               the row to sum to
                                1
    new_influence_matrix[i, :] /= np.sum(
                              new_influence_matrix
                              [i, :])
    return new_influence_matrix

def adjust_influence_matrix(influence_matrix,
                           new_influence_matrix):
    change = new_influence_matrix - influence_matrix
    max_change = np.max(change)
    min_change = np.min(change)

    for i in range(new_influence_matrix.shape[0]):
    for j in range(new_influence_matrix.shape[1]):
                                      if change[
                              i, j] ==
                              max_change:
     new_influence_matrix[i, j] += 0.1 *
                               new_influence_matrix
                               [i, j]
  elif change[i, j] == min_change:
                            new_influence_matrix
                            [i, j] -= 0.05 *
                            new_influence_matrix
                            [i, j]

    # Re-normalize the row to sum to 1
                              new_influence_matrix
                              [i, :] /= np.sum(
                              new_influence_matrix
                              [i, :])

    return new_influence_matrix
```

– c).**Simulation Execution and Analysis**

The final section involves running the entire simulation and analyzing the results. The `run_simulation` function manages the overall process, beginning with the initialization of scores and the influence matrix. The iterative process is conducted by the `iterative_update` function, which repeatedly updates scores and the influence matrix until the scores converge or the maximum number of iterations is reached. During this process, random events are occasionally applied to introduce variability. The function tracks the history of scores and the influence matrix throughout the iterations.After the simulation concludes, the results are analyzed and visualized, including the final scores for each candidate, the progression of voter preferences over time, and the evolution of the influence matrix. This comprehensive analysis offers insights into how voter preferences stabilize and how influence dynamics change throughout the simulation.

```
‘‘‘python
def iterative_update(scores, influence_matrix,
                              tolerance=1e-6,
                              max_iterations=1000):
    previous_scores = scores.copy()
    history = [previous_scores]  # List to store
                                   scores at each
                                   iteration
    influence_history = [influence_matrix.copy()]  #
                                    List to store
                                   influence matrix
                                   at each iteration
    for iteration in range(max_iterations):
new_scores = update_scores(previous_scores,
       influence_matrix)
        history.append(new_scores)
        # Check for convergence
  if np.max(np.abs(new_scores - previous_scores)) <
                                tolerance:
          print(f"Converged after {iteration + 1}
                                        iterations")

                                        return
                                        new_scores,
                                        iteration +
                                        1, history,
                                        influence_history

    # Apply random event impact every 10 iterations
 if (iteration + 1) % 10 == 0:
     print(f"Applying random event at iteration {
                                  iteration + 1}")
```

```
                                        previous_scores =

                                        apply_random_event
                                        (previous_scores)
        # Calculate score changes and update
                                          influence
                                          matrix
         score_changes = calculate_score_changes(
                                           previous_scores
                                           , new_scores)
         new_influence_matrix =
                                           update_influence_matrix
                                           (
                                           influence_matrix
                                           ,
                                           score_changes)
         # Adjust the influence matrix based on
                                           observed
                                           changes
         influence_matrix = adjust_influence_matrix(
                                           influence_matrix
                                           ,
                                           new_influence_matrix
                                           )
     influence_history.append(influence_matrix.copy()
                                       )
         previous_scores = new_scores
     print("Reached maximum iterations")
     return previous_scores, max_iterations, history,
                                        influence_history

def run_simulation(num_voters, num_candidates,
                                  average_connections):
    initial_scores = generate_gaussian_scores(
                                      num_voters,
                                      num_candidates)
influence_matrix
= generate_influence_matrix_gaussian(num_voters,
                                  average_connections)

    print("Initial Scores:")
    print(initial_scores)
    print("\nInitial Influence Matrix:")
    print(influence_matrix)

    final_scores, num_iterations, history,
                                      influence_history
                                      = iterative_update
                                      (initial_scores,
                                      influence_matrix)

    # Convert history list to a NumPy array for
                                      easier
                                      manipulation
```

```
  history = np.array(history)
  influence_history = np.array(influence_history)

  # Display final scores
  print("\nFinal scores after convergence:")
  for candidate in range(num_candidates):
    print(f"Candidate {candidate + 1} Scores:",
                                final_scores[:,
                                candidate])

  # Plotting the data from history
  for candidate in range(num_candidates):
    plt.figure(figsize=(16, 8))
    for voter in range(num_voters):
   plt.plot(range(history.shape[0]), history[:,
                               voter, candidate]
                               , label=f'Voter {
                               voter + 1} -
                               Candidate {
                               candidate + 1}',
                               marker='o')
   plt.title(f"Candidate {candidate + 1}
                               Preferences Over
                               Iterations")
plt.xlabel('Iteration')
      plt.ylabel('Score')
      plt.legend()
      plt.grid(True)
      plt.show()

  # Plotting average of scores for each candidate
  plt.figure(figsize=(16, 8))
  for candidate in range(num_candidates):
  avg_scores = np.mean(history[:, :, candidate],
                              axis=1)
    plt.plot(range(history.shape[0]), avg_scores,
                                label=f'Average
                                - Candidate {
                                candidate + 1}',
                                 marker='o')
  plt.title("Average Scores for Each Candidate
                              Over Iterations")
  plt.xlabel('Iteration')
  plt.ylabel('Average Score')
  plt.legend()
  plt.grid(True)
  plt.show()

  # Plotting change in influence matrix
  plt.figure(figsize=(16, 8))
  for i in range(influence_matrix.shape[0]):
      for j in range(influence_matrix.shape[1]):
  plt.plot(range(len(influence_history)),
                              influence_history[
                                          :, i, j], label=f'
                                          Influence {i+1}-{j
                                          +1}')
    plt.title("Change in Influence Matrix Over
                                          Iterations")
    plt.xlabel('Iteration')
    plt.ylabel('Influence Value')
    plt.legend()
    plt.grid(True)
```

9

References

1. A. Wolitzky, *Lecture 5: The Degroot Learning Model (PDF): Networks: Economics*, (`ocw.mit.edu/courses/14-15-networks-spring-2022/resources/mit14_15s22_lec5/.`).
2. A. Haensch, N. Dragovic, C. Borgers, B. Boghosian, *Journal of Artificial Societies and Social Simulation* **26**, 8, ISSN: 1460-7425, (`http://jasss.soc.surrey.ac.uk/26/1/8.html`) (2023).
3. C. Borgers, N. Dragovic, A. Haensch, *Society for Industrial and Applied Mathematics* (2024).
4. *Download Adimpact's 2024 Political Spending Projections Report* (`adimpact.com/2024-political-spending-projections-report/.`).
5. A. Haensch, *ODyN*, en, (`https://github.com/annahaensch/ODyN/blob/main/LICENSE`).
6. D. Natasa, *Opinion Dynamics- Overview*, en.