```
# import libraries
import networkx as nx
import pandas as pd
import seaborn as sns
from networkx.algorithms.community import louvain_communities
from scipy.sparse import coo_matrix
import random
import itertools

#  Define functions

# def particpationCoeffcient() - finds how many groups a node is directly
connected to
def particpationCoeffcient(G,n):
    modularity(G)
        # Get the edge list of the nodes 1 degree away from the given node
    edgelist = list(nx.bfs_edges(G, n, depth_limit=1))

    # Extract the edges and create a subgraph
    subgraph = G.edge_subgraph(edgelist)

    # Get the communities of the nodes in the subgraph
    nearby_communities = set(nx.get_node_attributes(subgraph,
'Louvain').values())

     # Check if the node is part of a commmunity
    if nx.get_node_attributes(G, 'Louvain').get(n) is None:
        return 0

    # Calculate the participation coefficient
    if len(nearby_communities) == 1:
        return 0
    else:
        node_community = nx.get_node_attributes(G, 'Louvain')[n]
        within_community_degree = sum([d for n, d in
G.degree(subgraph.nodes()) if nx.get_node_attributes(G, 'Louvain')[n] ==
node_community])
        total_degree = sum(dict(G.degree(subgraph.nodes())).values())
        return 1 - (within_community_degree / total_degree)**2


# returns an aggregate of centrality measures

def aggCent(G, n):
    w1 = nx.degree_centrality(G)[n]
    betdict = nx.betweenness_centrality(G)
    w2 = betdict[n]
    w3 = particpationCoeffcient(G, n)
    aggCent = w1 + w2 + w3
    return aggCent


# adds all related attributes to nodes
def brainAttributes(G):
    a = nx.betweenness_centrality(G)
    b = {}
    c = {}
    d = {}
    if not G.is_directed():
        e = nx.rich_club_coefficient(G, normalized=False, seed=42)
        nx.set_node_attributes(G, e, "Rich_Club_Coefficient")
    f = modularity(G)
    nx.set_node_attributes(G, f, "Louvain")
    for n in G.nodes():
        b.update({n:particpationCoeffcient(G,n)})
        c.update({n:aggCent(G,n)})
        d.update({n:G.degree(n)})
    nx.set_node_attributes(G, a, "Betweenness")
    nx.set_node_attributes(G, b, "Particpation_Coeffcient")
    nx.set_node_attributes(G, c, "Centrality_Aggregate")
    nx.set_node_attributes(G, d, "Degree")





# adds Louvain groups to node attributes
def modularity(G):
    louvain = louvain_communities(G, seed=42)
    louvain_dict = {}
    for i,group in enumerate(louvain):
        for g in group:
            louvain_dict[g] = i
    communities ={}
    for node, community_id in louvain_dict.items():
        if(community_id not in communities):
            communities[community_id] = []
        communities[community_id].append(node)
    return louvain_dict


# Def explore() - function to return network measures that are relevant in
the context of our network
    #-

# isSmallWorld return a True if graph object G is a small world network
def isSmallWorld(G):
    print("Starting sigma calculation: ")
    a = nx.sigma(G,niter=25,nrand=3,seed=42)
    print("Done!")
    if (a >= 1):
        return True
    else:
        return False

# latticeOrRandom will return a printed string indicating what the omega
small world coefcient is indicating
def latticeOrRandom(G):
    print("Starting omega calculation: ")
    a = nx.omega(G)
    print("Done!")
    if(a > 0):
        print("Graph is has lattice-like properties")
    elif(a == 0):
        print("Graph has small world properties")
    else:
        print("Graph has random-like properties")

# selects 10% of random nodes from G
def select_random_nodes(G):
    num_nodes = len(G.nodes)
    num_to_select = round(num_nodes * 0.1)
    nodes = list(G.nodes)
    selected_nodes = random.sample(nodes, num_to_select)
    return selected_nodes


# returns a dictionary with cluster density
def clusterDensity(G):
    # Run the algorithm with a random seed, so it's the same each time
    louvain = louvain_communities(G, seed=42)
    louvain_dict = {}
    for i,group in enumerate(louvain):
        for g in group:
            louvain_dict[g] = i
    communities ={}
    for node, community_id in louvain_dict.items():
        if(community_id not in communities):
            communities[community_id] = []
        communities[community_id].append(node)

    community_subgraphs = {community_id: G.subgraph(nodes) for community_id,
nodes in communities.items()}

    cluster_density = {}
    for community_id, subgraph in community_subgraphs.items():
        density = nx.density(subgraph)
        cluster_density[community_id] = density

    return cluster_density



# def richClub(k) - returns the rich club subgraph for nodes above degree
return True



# def isScaleFree() - returns a True if there is a power law degree
distribution.

def small_world_subgraph(G, nodes):
    #Creates a subgraph for each node in a list of nodes, each subgraph
having that specific node removed.
    #Runs isSmallWorld function on the subgraph that will return a true if
the graph is still a small world.
    #Returns a dictionary with the node name of the node removed for the
subgraph as the key and the boolean isSmallworld for said subgraph as the
value.
    result = {}
    for node in nodes:
        subgraph = G.subgraph(set(G.nodes()) - {node})
        result[node] = isSmallWorld(subgraph)
    return result


def will_infect_entire_network(G, initial_excited_node, time_steps=50):

    #Function to predict whether an "epidemic" will spread to all nodes in
the network.

    #Parameters:
    #G (nx.Graph): The network graph
    #initial_excited_node (int): The node to start the "epidemic"
    #time_steps (int, optional): The number of time steps for the simulation.
Defaults to 50.

    #Returns:
   # bool: True if all nodes become excited, False otherwise



    # Reset the states of all nodes
    for node in G.nodes:
        G.nodes[node]['excited'] = False  # No nodes are initially excited
        G.nodes[node]['state'] = 'S'  # All nodes are initially Susceptible
        G.nodes[node]['refractory_period'] = 0  # No refractory period 
initially

    # Set the initial excited node
    G.nodes[initial_excited_node]['excited'] = True

    # Perform the simulation
    for t in range(time_steps):
        new_excited_nodes = []  # List to hold newly excited nodes
        new_node_states = {}  # Dictionary to hold new states of nodes

        # Loop through all nodes
        for node in G.nodes:
            # If node is Susceptible and not in refractory period
            if G.nodes[node]['state'] == 'S' and 
G.nodes[node]['refractory_period'] <= 0:
                # If the node is excited
                if G.nodes[node]['excited']:
                    new_node_states[node] = 'I'  # Node becomes Infected
                    G.nodes[node]['refractory_period'] = 10  # Set refractory 
period

            # If node is Infected
            elif G.nodes[node]['state'] == 'I':
                new_node_states[node] = 'R'  # Node becomes Resistant

            # If node is Resistant
            elif G.nodes[node]['state'] == 'R':
                new_node_states[node] = 'S'  # Node becomes Susceptible again

            # Decrease refractory period count
            if G.nodes[node]['refractory_period'] > 0:
                G.nodes[node]['refractory_period'] -= 1

            # Spread excitement to neighbors if the node is excited
            if G.nodes[node]['excited']:
                for neighbor in G.neighbors(node):
                    # If the neighbor node is not excited, Susceptible, and 
not in refractory period
                    if not G.nodes[neighbor]['excited'] and 
G.nodes[neighbor]['state'] == 'S' and G.nodes[neighbor]['refractory_period'] 
<= 0:
                        rate = infection_rate(G.edges[node, 
neighbor]['strength'], G.nodes[neighbor]['state'])
                        # If the random number is less than the infection 
rate
                        if random.random() < rate:
                            new_excited_nodes.append(neighbor)  # Add 
neighbor to the list of newly excited nodes
```

```
        # Update node states and excited nodes
        for node in new_node_states:
            G.nodes[node]['state'] = new_node_states[node]  # Update the 
state of the node
            G.nodes[node]['excited'] = False  # Set excited status to False

        for node in new_excited_nodes:
            G.nodes[node]['excited'] = True  # Set excited status to True for 
newly excited nodes

    # Check if all nodes have been excited at some point
    for node in G.nodes:
        # If any node was not excited, return False
        if not G.nodes[node]['excited']:
            return False

    # If all nodes were excited at some point, return True
    return True


def will_infect_entire_network_rewired(G, initial_excited_node, 
time_steps=50):

    #Function to predict whether an "epidemic" will spread to all nodes in 
the network
    #while randomizing the connections of edges at each stage of the 
epidemic.

    #Parameters:
    #G (nx.Graph): The network graph
    #initial_excited_node (int): The node to start the "epidemic"
    #time_steps (int, optional): The number of time steps for the simulation. 
Defaults to 50.

    #Returns:
    #bool: True if all nodes become excited, False otherwise



    # Reset the states of all nodes
    for node in G.nodes:
        G.nodes[node]['excited'] = False  # No nodes are initially excited
        G.nodes[node]['state'] = 'S'  # All nodes are initially Susceptible
        G.nodes[node]['refractory_period'] = 0  # No refractory period 
initially

    # Set the initial excited node
    G.nodes[initial_excited_node]['excited'] = True
    # Perform the simulation
    for t in range(time_steps):
        # Rewire the network at each time step while preserving degree
distribution
        G = nx.double_edge_swap(G, nswap=100, max_tries=200,seed=42)  #
Adjust 'nswap' and 'max_tries' as needed

        new_excited_nodes = []  # List to hold newly excited nodes
        new_node_states = {}  # Dictionary to hold new states of nodes

        # Loop through all nodes
        for node in G.nodes:
            # If node is Susceptible and not in refractory period
            if G.nodes[node]['state'] == 'S' and
G.nodes[node]['refractory_period'] <= 0:
                # If the node is excited
                if G.nodes[node]['excited']:
                    new_node_states[node] = 'I'  # Node becomes Infected
                    G.nodes[node]['refractory_period'] = 10  # Set refractory
period

            # If node is Infected
            elif G.nodes[node]['state'] == 'I':
                new_node_states[node] = 'R'  # Node becomes Resistant

            # If node is Resistant
            elif G.nodes[node]['state'] == 'R':
                new_node_states[node] = 'S'  # Node becomes Susceptible again

            # Decrease refractory period count
            if G.nodes[node]['refractory_period'] > 0:
                G.nodes[node]['refractory_period'] -= 1

            # Spread excitement to neighbors if the node is excited
            if G.nodes[node]['excited']:
                for neighbor in G.neighbors(node):
                    # If the neighbor node is not excited, Susceptible, and
not in refractory period
                    if not G.nodes[neighbor]['excited'] and
G.nodes[neighbor]['state'] == 'S' and G.nodes[neighbor]['refractory_period']
<= 0:
                        rate = infection_rate(G.edges[node,
neighbor]['strength'], G.nodes[neighbor]['state'])
                        # If the random number is less than the infection
rate
                        if random.random() < rate:
                            new_excited_nodes.append(neighbor)  # Add
neighbor to the list of newly excited nodes
```

```
        # Update node states and excited nodes
        for node in new_node_states:
            G.nodes[node]['state'] = new_node_states[node]  # Update the
state of the node
            G.nodes[node]['excited'] = False  # Set excited status to False

        for node in new_excited_nodes:
            G.nodes[node]['excited'] = True  # Set excited status to True for
newly excited nodes

    # Check if all nodes have been excited at some point
    for node in G.nodes:
        # If any node was not excited, return False
        if not G.nodes[node]['excited']:
            return False

    # If all nodes were excited at some point, return True
    return True



# def select randomnode
# selcet random aggCen
# selct random richclub
```

1) Explore the network properties of the human connectome

```
# Create Graph Object
#G = nx.read_edgelist("bn-human-Jung2015_small_clean.edges")
Gdi = nx.read_graphml("rhesus_brain_1.graphml")
G = Gdi.to_undirected()

# Create Directed Graph Object
#Gdi = nx.read_edgelist("bn-human-
Jung2015_small_clean.edges",create_using=nx.DiGraph)

# add attributes to graph
# identify communities

brainAttributes(G)
print("done")
#brainAttributes(Gdi)
#print("done")

# identify hubs

# list bridges
#bridges = list(nx.bridges(G))
```

```
#if(nx.has_bridges(G)):
    #print(bridges)

# Convert nodes to a dataframe
nodes = pd.DataFrame.from_dict(G.nodes, orient='index')

# identify connection density
#print(clusterDensity(G))

# average shortest path
#nx.average_shortest_path_length(G)

#print(nodes)

louvain = louvain_communities(G, seed=42)
louvain_dict = {}
for i,group in enumerate(louvain):
    for g in group:
        louvain_dict[g] = i
    communities ={}
for node, community_id in louvain_dict.items():
    if(community_id not in communities):
        communities[community_id] = []
        communities[community_id].append(node)
print(louvain_dict)
nx.set_node_attributes(G, louvain_dict, "Louvain")
```

```
{'n3': 0, 'n30': 0, 'n7': 0, 'n15': 0, 'n13': 0, 'n8': 0, 'n108': 0, 'n26':
0, 'n9': 0, 'n1': 0, 'n25': 0, 'n18': 0, 'n189': 0, 'n2': 0, 'n20': 0, 'n0':
0, 'n5': 0, 'n12': 0, 'n14': 0, 'n10': 0, 'n163': 0, 'n29': 0, 'n19': 0,
'n11': 0, 'n24': 0, 'n142': 0, 'n4': 0, 'n6': 0, 'n28': 0, 'n22': 0, 'n21':
0, 'n232': 0, 'n72': 1, 'n17': 1, 'n45': 1, 'n57': 1, 'n40': 1, 'n107': 1,
'n73': 1, 'n27': 1, 'n52': 1, 'n34': 1, 'n42': 1, 'n38': 1, 'n49': 1, 'n104':
1, 'n65': 1, 'n39': 1, 'n47': 1, 'n60': 1, 'n70': 1, 'n48': 1, 'n33': 1,
'n68': 1, 'n23': 1, 'n75': 1, 'n53': 1, 'n61': 1, 'n63': 1, 'n41': 1, 'n54':
1, 'n62': 1, 'n81': 1, 'n80': 1, 'n59': 1, 'n79': 1, 'n77': 1, 'n69': 1,
'n32': 1, 'n66': 1, 'n37': 1, 'n92': 1, 'n97': 1, 'n111': 1, 'n78': 1,
'n110': 1, 'n36': 1, 'n67': 1, 'n44': 1, 'n35': 1, 'n56': 1, 'n55': 1, 'n46':
1, 'n71': 1, 'n64': 1, 'n58': 1, 'n50': 1, 'n43': 1, 'n31': 1, 'n51': 1,
'n131': 2, 'n175': 2, 'n183': 2, 'n141': 2, 'n140': 2, 'n143': 2, 'n156': 2,
'n154': 2, 'n153': 2, 'n145': 2, 'n166': 2, 'n174': 2, 'n127': 2, 'n176': 2,
'n170': 2, 'n147': 2, 'n130': 2, 'n161': 2, 'n167': 2, 'n137': 2, 'n119': 2,
'n122': 2, 'n173': 2, 'n120': 2, 'n115': 2, 'n168': 2, 'n114': 2, 'n117': 2,
'n180': 2, 'n135': 2, 'n157': 2, 'n164': 2, 'n138': 2, 'n148': 2, 'n177': 2,
'n158': 2, 'n144': 2, 'n155': 2, 'n150': 2, 'n162': 2, 'n159': 2, 'n171': 2,
'n146': 2, 'n160': 2, 'n169': 2, 'n116': 2, 'n133': 2, 'n125': 2, 'n126': 2,
'n136': 2, 'n151': 2, 'n124': 2, 'n132': 2, 'n165': 2, 'n149': 2, 'n129': 2,
'n87': 3, 'n152': 3, 'n187': 3, 'n123': 3, 'n96': 3, 'n112': 3, 'n118': 3,
'n211': 3, 'n128': 3, 'n235': 3, 'n209': 3, 'n86': 3, 'n106': 3, 'n210': 3,
'n85': 3, 'n74': 3, 'n236': 3, 'n134': 3, 'n93': 3, 'n101': 3, 'n88': 3,
'n95': 3, 'n178': 3, 'n109': 3, 'n84': 3, 'n102': 3, 'n100': 3, 'n89': 3,
'n121': 3, 'n91': 3, 'n237': 3, 'n172': 3, 'n199': 3, 'n105': 3, 'n139': 3,
'n90': 3, 'n16': 3, 'n113': 3, 'n94': 3, 'n103': 3, 'n76': 3, 'n179': 3,
'n181': 3, 'n82': 3, 'n98': 3, 'n99': 3, 'n83': 3, 'n213': 4, 'n233': 4,
'n182': 4, 'n221': 4, 'n240': 4, 'n203': 4, 'n228': 4, 'n225': 4, 'n193': 4,
'n186': 4, 'n206': 4, 'n223': 4, 'n184': 4, 'n230': 4, 'n192': 4, 'n218': 4,
'n207': 4, 'n205': 4, 'n194': 4, 'n201': 4, 'n215': 4, 'n222': 4, 'n185': 4,
'n198': 4, 'n220': 4, 'n214': 4, 'n239': 4, 'n204': 4, 'n190': 4, 'n200': 4,
'n196': 4, 'n234': 4, 'n241': 4, 'n197': 4, 'n219': 4, 'n208': 4, 'n188': 4,
'n212': 4, 'n238': 4, 'n191': 4, 'n227': 4, 'n224': 4, 'n229': 4, 'n216': 4,
'n195': 4, 'n202': 4, 'n226': 4, 'n231': 4, 'n217': 4}



#nodes2 = pd.DataFrame.from_dict(G.nodes, orient='index')

#print(nodes2)

# sort nodes by Centrality Aggregate attribute
#node_centrality = nx.get_node_attributes(G, 'Centrality_Aggregate')
#sorted_nodes = sorted(node_centrality, key=node_centrality.get,
reverse=True)

# create subgraph of top 500 nodes  -- ended up not using this feature due to
having a smaller graph
#top_nodes = sorted_nodes[:500]
#subG = G.subgraph(top_nodes)

#pr = nx.pagerank(Gdi)
#nx.set_node_attributes(Gdi, pr, "pageRank")

# write subgraph to GML file
#nx.write_gml(G, 'rhesus_brain_1.gml')
#nx.write_gml(Gdi, 'rhesus_brain_1_directed.gml' )
#print("done")

#print(subG.number_of_nodes())
#print(subG.number_of_edges())
#print(top_nodes)

# histogram of degree
#sns.displot(x="Degree",data=nodes)

# Prove network is scale free, if not this implies contraint by physical cost
```

2) Explore which Neurons are critical in the prevention of Alzheimer's Disease or Schitzaphernia

```
# Prove Small world
    # high clustering
    # short avg path length
# Prove complex graph
Prtop = ('n7', 'n122', 'n4', 'n0', 'n1', 'n2', 'n75', 'n73', 'n88', 'n114',
'n6', 'n36', 'n78', 'n74', 'n115', 'n45', 'n31', 'n113', 'n123', 'n132',
'n128', 'n117', 'n185', 'n184')
centagtop =('n186', 'n32', 'n0', 'n77', 'n189', 'n2', 'n1', 'n75', 'n73',
'n34', 'n74', 'n113', 'n125', 'n194', 'n191', 'n184', 'n185', 'n120' ,'n187',
'n118', 'n76', 'n31', 'n131', 'n115')
random = select_random_nodes(G)
print(small_world_subgraph(G,itertools.islice(Prtop,2)))
print(small_world_subgraph(G,itertools.islice(centagtop,2)))
print(small_world_subgraph(G,itertools.islice(random,2)))

Starting sigma calculation:

# Lesion Hub and re-prove small world (AD markers include loss of small
world, increased path length)

# re-prove complex, (Schitzophernia is more simlar to a random than complex
graph/long physical distance between hubs)
latticeOrRandom(G)
```

3) Which theory better enables a complete infection of the network with action potential model as a SIRS epidemic, constant remodeling of synaptic edges vs stable edges

```
# run stable epidemic random start
will_infect_entire_network(G, itertools.islice(random,1), time_steps=50)

# run remodel epidemic random start
will_infect_entire_network_rewired(G, itertools.islice(random,1),
time_steps=50)

# run stable epidemic my hub start
will_infect_entire_network(G, itertools.islice(centagtop,1), time_steps=50)

# run remodel epidemic my hub start
will_infect_entire_network_rewired(G, itertools.islice(centagtop,1),
time_steps=50)

# run stable epidemic PageRank hub start
will_infect_entire_network(G, itertools.islice(Prtop,1), time_steps=50)

# run remodel epicemic PageRank hub start
will_infect_entire_network_rewired(G, itertools.islice(Prtop,1),
time_steps=50)
```

4) Does PageRank accurately describe hubbness?

```
# run pageRank() on Gdi
#pr = nx.pagerank(Gdi)
#nx.set_node_attributes(G, pr, "pageRank")

# compare and contrast hubs
```