Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

Exploring Graph Data

Part One: Think About The Data (Megan S.)

The data set we will be exploring is the Twitch Gamers Social Network. In the Twitch Gamers Social Network, the nodes are Twitch gamers and the edges represent a mutual follower relationship. The network has no missing attributes and forms a single strongly connected component. We are interested in this network because we are Twitch fans.

Before analysis, we have some predictions about the network. We predict that Twitch gamers with high centrality values are likely to have a large number of followers. This implies that gamers with high centrality values are expected to be highly engaged with their audience and maintain consistent content creation.

We hypothesize that the degree distribution of the Twitch social network graph will follow a power-law distribution. Given the platform's tendency for certain gamers to amass large followings while others have fewer connections, coupled with the dynamics of preferential attachment, it is likely that a small number of gamers will have significantly higher degrees than the majority, aligning with the characteristics of a power-law distribution.

We hypothesize that the Twitch social network graph will exhibit the small-world property. This expectation is based on the likelihood of short average path lengths between Twitch gamers due to the interconnected nature of the platform. Despite the potentially vast size of the network, the presence of mutual follower relationships can facilitate relatively quick connections between any two gamers within the network. Additionally, the clustering of followers around popular gamers may further decrease path lengths, contributing to the small-world phenomenon commonly observed in social networks.

Part Two: Write Functions For Graph Analysis (Michael B.)

To complete the data analysis using Python, we'll write functions to extract key metrics from the dataset. This includes determining the number of vertices to gauge network size, computing the degree of a vertex to understand node connectivity, assessing clustering coefficients to identify densely connected regions, and calculating betweenness centrality to pinpoint influential nodes. Additionally, we'll develop a function for the average shortest path length to evaluate network efficiency, and generate an adjacency matrix for visualizing connections between nodes. These functions (below) will provide comprehensive insights into the network's structure and dynamics, facilitating thorough data analysis.

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

```
file_path = r"C:\Users\akmik\OneDrive\Desktop\CSCI 347\Mini Project 2\large_twitch_edges.csv"
edges = read_csv(file_path)
```

```
def read_csv(file_path):
    """
    Read the CSV file containing edges and return a list of edges.

    Parameters:
        file_path (str): The path to the CSV file.

    Returns:
        list: A list of edges, where each edge is represented as a tuple (vertex1, vertex2).
    """
    edges = []
    with open(file_path, 'r') as file:
        csv_reader = csv.reader(file)
        next(csv_reader)  # Skip the header row
        for row in csv_reader:
            if row[0] != 'numeric_id_1' and row[1] != 'numeric_id_2':
                edges.append((int(row[0]), int(row[1])))
    return edges
```

I(Michael B.) created my own read_csv function to properly open the file for use in the following functions. All of the following functions use the code above in some way in terms of getting the data for further manipulation:

```
def num_vertices(edges):
    """
    Calculate the number of vertices in the graph.

    Parameters:
        edges (list): A list of edges, where each edge is represented as a tuple (vertex1, vertex2).

    Returns:
        int: The number of vertices in the graph.
    """
    vertices = set()
    for edge in edges:
        vertices.update(edge)
    return len(vertices)
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

```python
def degree_of_vertex(edges, vertex):
    """
    Calculate the degree of a given vertex in the graph.

    Parameters:
        edges (list): A list of edges, where each edge is represented as a tuple (vertex1, vertex2).
        vertex (int): The vertex for which degree needs to be calculated.

    Returns:
        int: The degree of the given vertex.
    """
    degree = 0
    for edge in edges:
        if vertex in edge:
            degree += 1
    return degree
```

```python
def clustering_coefficient(edges, vertex):
    """
    Calculate the clustering coefficient of a given vertex in the graph.

    Parameters:
        edges (list): A list of edges, where each edge is represented as a tuple (vertex1, vertex2).
        vertex (int): The vertex for which clustering coefficient needs to be calculated.

    Returns:
        float: The clustering coefficient of the given vertex.
    """
    neighbors = set()
    for edge in edges:
        if vertex in edge:
            neighbors.add(edge[0] if edge[0] != vertex else edge[1])
    num_neighbors = len(neighbors)
    if num_neighbors < 2:
        return 0.0

    total_possible_edges = num_neighbors * (num_neighbors - 1) / 2
    actual_edges = 0
    for u in neighbors:
        for v in neighbors:
            if u < v and (u, v) in edges:
                actual_edges += 1
    return actual_edges / total_possible_edges
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

```python
def betweenness_centrality(edges, vertex):
    """
    Calculate the betweenness centrality of a given vertex in the graph.
    Parameters:
        edges (list): A list of edges, where each edge is represented as a tuple (vertex1, vertex2).
        vertex (int): The vertex for which betweenness centrality needs to be calculated.
    Returns:
        float: The betweenness centrality of the given vertex.
    """
    def get_shortest_paths(edges, vertex):
        paths = defaultdict(int)
        queue = [vertex]
        visited = set(queue)
        distance = {vertex: 0}
        while queue:
            current_vertex = queue.pop(0)
            for edge in edges:
                if current_vertex in edge:
                    neighbor = edge[0] if edge[0] != current_vertex else edge[1]
                    if neighbor not in visited:
                        queue.append(neighbor)
                        visited.add(neighbor)
                        distance[neighbor] = distance[current_vertex] + 1
                        paths[neighbor] += 1
                    elif distance[neighbor] == distance[current_vertex] + 1:
                        paths[neighbor] += 1
        return paths
    total_paths = defaultdict(int)
    for vertex in range(num_vertices(edges)):
        paths = get_shortest_paths(edges, vertex)
        for key, value in paths.items():
            total_paths[key] += value

    vertex_paths = total_paths[vertex]
    total_possible_paths = (num_vertices(edges) - 1) * (num_vertices(edges) - 2) / 2
    return vertex_paths / total_possible_paths
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

```python
def average_shortest_path_length(edges):
    """
    Calculate the average shortest path length in the graph.

    Parameters:
        edges (list): A list of edges, where each edge is represented as a tuple (vertex1, vertex2).

    Returns:
        float: The average shortest path length in the graph.
    """
    def bfs_shortest_path(edges, start):
        visited = {start}
        queue = [(start, 0)]
        while queue:
            node, depth = queue.pop(0)
            for edge in edges:
                if node in edge:
                    neighbor = edge[0] if edge[0] != node else edge[1]
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, depth + 1))
        return sum(depth for _, depth in queue) / len(queue)

    total_lengths = 0
    for vertex in range(num_vertices(edges)):
        total_lengths += bfs_shortest_path(edges, vertex)
    return total_lengths / num_vertices(edges)
```

```python
def adjacency_matrix(edges):
    """
    Generate the adjacency matrix of the graph.

    Parameters:
        edges (list): A list of edges, where each edge is represented as a tuple (vertex1, vertex2).

    Returns:
        list: The dense adjacency matrix of the graph.
    """
    num_verts = num_vertices(edges)
    matrix = [[0] * num_verts for _ in range(num_verts)]
    for edge in edges:
        matrix[edge[0]][edge[1]] = 1
        matrix[edge[1]][edge[0]] = 1
    return matrix
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

```python
def eigenvector_centrality(adjacency_matrix):
    """
    Calculate the eigenvector centrality using power iteration.

    Parameters:
        adjacency_matrix (list): The dense adjacency matrix of the graph.

    Returns:
        list: The eigenvector corresponding to the dominant eigenvector of the adjacency matrix.
    """
    def normalize(vector):
        norm = sum(vector)
        return [x / norm for x in vector]

    n = len(adjacency_matrix)
    x = [1] * n
    prev_x = [0] * n
    epsilon = 1e-8
    while sum((x[i] - prev_x[i]) ** 2 for i in range(n)) > epsilon:
        prev_x = x.copy()
        x = [0] * n
        for i in range(n):
            for j in range(n):
                x[i] += adjacency_matrix[i][j] * prev_x[j]
        x = normalize(x)
    return x
```

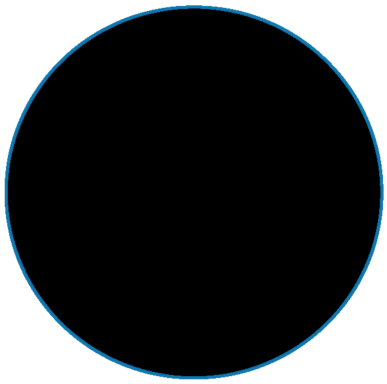
Part Three: Analyze The Graph Data (Megan S., Michael B, and Rohan K.)

(Megan S.) To produce a **visualization** of the graph, we used the largest connected components because the whole network is too large to graph. Based on the circular visualization of the largest connected component in the image below, the data is so dense that the circle is completely filled in.

```python
import csv
import networkx as nx

# Read CSV file and create a networkx graph
def read_csv(filename):
    G = nx.Graph()
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # Skip the header row
        for row in reader:
            user1, user2 = map(int, row)
            G.add_edge(user1, user2)
    return G

# Find the largest connected component
def largest_connected_component(G):
    connected_components = nx.connected_components(G)
    largest_component = max(connected_components, key=len)
    return largest_component

if __name__ == "__main__":
    filename = "/Users/megansteinmasel/Desktop/twitch_gamers/large_twitch_edges.csv"
    G = read_csv(filename)
    largest_component = largest_connected_component(G)
    print("Number of nodes in the largest connected component:", len(largest_component))
```

```
Number of nodes in the largest connected component: 168114
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat



(Megan S.) We used the whole data network to find the 10 nodes with the **highest degree**. The code and output is shown below.

```python
import csv
import networkx as nx

# Read CSV file and create a networkx graph
def read_csv(filename):
    G = nx.Graph()
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # Skip the header row
        for row in reader:
            user1, user2 = map(int, row)
            G.add_edge(user1, user2)
    return G

# Find the top N nodes with the highest degree
def top_nodes_by_degree(G, n=10):
    degrees = dict(G.degree())
    top_nodes = sorted(degrees, key=degrees.get, reverse=True)[:n]
    return top_nodes

if __name__ == "__main__":
    filename = "/Users/megansteinmasel/Desktop/twitch_gamers/large_twitch_edges.csv"
    G = read_csv(filename)
    top_nodes = top_nodes_by_degree(G, n=10)
    print("Top 10 nodes with the highest degree:")
    for node in top_nodes:
        print(node)
```

```
Top 10 nodes with the highest degree:
61862
125642
32338
71050
110345
64605
52703
155127
60588
123018
```

(Rohan K.) We used the whole data network to find the top 10 nodes with the highest **betweenness centrality**. The code and output is shown below.

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

```python
import csv
import networkx as nx

def read_csv(filename):
    G = nx.Graph()
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            user1, user2 = map(int, row)
            G.add_edge(user1, user2)
    return G

# Load the graph from the CSV file
graph = read_csv("large_twitch_edges.csv")

# Compute betweenness centrality using approximate algorithm
betweenness_centrality = nx.betweenness_centrality(graph, k=1000, endpoints=False)

# Sort nodes based on betweenness centrality values
sorted_nodes = sorted(betweenness_centrality.items(), key=lambda x: x[1], reverse=True)

# Print the top 10 nodes with highest betweenness centrality
print("Top 10 nodes with highest betweenness centrality:")
for node, centrality in sorted_nodes[:10]:
    print("Node:", node, "Betweenness Centrality:", centrality)
```

✓ 13m 25.3s Python

```
Top 10 nodes with highest betweenness centrality:
Node: 61862 Betweenness Centrality: 0.07939312009816589
Node: 32338 Betweenness Centrality: 0.05619559842388517
Node: 71050 Betweenness Centrality: 0.05291538197094897
Node: 125642 Betweenness Centrality: 0.046107190953200455
Node: 155127 Betweenness Centrality: 0.0347161148008913 6
Node: 110345 Betweenness Centrality: 0.03352050934511893
Node: 52703 Betweenness Centrality: 0.023641931129752523
Node: 64605 Betweenness Centrality: 0.022784476686421862
Node: 60588 Betweenness Centrality: 0.014010719795910087
Node: 69330 Betweenness Centrality: 0.012078826835340416
```

(Rohan K.) We used the whole data network to find the 10 nodes with the **highest clustering coefficient**. The code and output is shown below.

```python
import csv
import networkx as nx

def read_csv(filename):
    G = nx.Graph()
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            user1, user2 = map(int, row)
            G.add_edge(user1, user2)
    return G

# Load the graph from the CSV file
graph = read_csv("large_twitch_edges.csv")

# Compute clustering coefficient for each node
clustering_coefficients = nx.clustering(graph)

# Sort nodes based on clustering coefficient values
sorted_nodes = sorted(clustering_coefficients.items(), key=lambda x: x[1], reverse=True)

# Extract the top 10 nodes with highest clustering coefficients
top_10_nodes = [node for node, _ in sorted_nodes[:10]]

# Report the top 10 nodes
print("Top 10 nodes with highest clustering coefficients:")
for node in top_10_nodes:
    print("Node:", node, "Clustering Coefficient:", clustering_coefficients[node])
```

```
Top 10 nodes with highest clustering coefficients:
Node: 101382 Clustering Coefficient: 1.0
Node: 130634 Clustering Coefficient: 1.0
Node: 25994 Clustering Coefficient: 1.0
Node: 66624 Clustering Coefficient: 1.0
Node: 132454 Clustering Coefficient: 1.0
Node: 43761 Clustering Coefficient: 1.0
Node: 144973 Clustering Coefficient: 1.0
Node: 87904 Clustering Coefficient: 1.0
Node: 120862 Clustering Coefficient: 1.0
Node: 66224 Clustering Coefficient: 1.0
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

(Megan S.) We used the whole data network to find the top 10 nodes as ranked by **eigenvector centrality**. The code and output is shown below.

```python
import csv
import networkx as nx

# Read CSV file and create a networkx graph
def read_csv(filename):
    G = nx.Graph()
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # Skip the header row
        for row in reader:
            user1, user2 = map(int, row)
            G.add_edge(user1, user2)
    return G

# Find the top N nodes with the highest eigenvector centrality
def top_nodes_by_eigenvector_centrality(G, n=10):
    eigenvector_centrality = nx.eigenvector_centrality(G)
    top_nodes = sorted(eigenvector_centrality, key=eigenvector_centrality.get, reverse=True)[:n]
    return top_nodes

if __name__ == "__main__":
    filename = "/Users/megansteinmasel/Desktop/twitch_gamers/large_twitch_edges.csv"
    G = read_csv(filename)
    top_nodes = top_nodes_by_eigenvector_centrality(G, n=10)
    print("Top 10 nodes with the highest eigenvector centrality:")
    for node in top_nodes:
        print(node)
```

```
Top 10 nodes with the highest eigenvector centrality:
125642
32338
61862
110345
71050
64605
52703
60588
155127
152296
```

(Rohan K.) We used the whole data network to find the top 10 nodes as ranked by **Pagerank**. Below is the code and output.

```python
import csv
import networkx as nx

def read_csv(filename):
    G = nx.Graph()
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            user1, user2 = map(int, row)
            G.add_edge(user1, user2)
    return G

# Load the graph from the CSV file
graph = read_csv("large_twitch_edges.csv")

# Compute PageRank
pagerank_scores = nx.pagerank(graph)

# Sort nodes based on PageRank scores
sorted_nodes = sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True)

# Extract the top 10 nodes with highest PageRank scores
top_10_nodes = [node for node, _ in sorted_nodes[:10]]

# Report the top 10 nodes
print("Top 10 nodes with highest PageRank scores:")
for node in top_10_nodes:
    print("Node:", node, "PageRank Score:", pagerank_scores[node])
```

✓ 1m 2.3s Python

```
Top 10 nodes with highest PageRank scores:
Node: 61862 PageRank Score: 0.0027776746826347265
Node: 71050 PageRank Score: 0.0022569924321814 3
Node: 125642 PageRank Score: 0.0020806435325073 8
Node: 32338 PageRank Score: 0.0018744205340927036
Node: 110345 PageRank Score: 0.001435768147815589
Node: 64605 PageRank Score: 0.0013158565101092114
Node: 155127 PageRank Score: 0.0011585026264481573
Node: 52703 PageRank Score: 0.0011074833843317431
Node: 6250 PageRank Score: 0.0009268748754864182
Node: 60588 PageRank Score: 0.0008967905169496635
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

(Megan S.) Based on the data findings, it's evident that certain nodes consistently appear among the top ranks across different centrality measures, while others vary. Node 61862 consistently ranks highly, appearing in the top three for degree, betweenness centrality, eigenvector centrality, and Pagerank. Node 32338 also maintains a significant presence, being among the top three for degree, betweenness centrality, and eigenvector centrality. However, there are notable differences as well. For instance, while node 125642 is prominent in betweenness centrality, eigenvector centrality, and Pagerank, it doesn't feature prominently in degree centrality. Similarly, node 71050 ranks high in betweenness centrality and Pagerank but doesn't appear in the top three for degree or eigenvector centrality. Additionally, nodes like 101382, 130634, and 25994 (etc.) stand out in clustering coefficients but don't feature prominently in other centrality measures. These variations suggest that different nodes play unique roles within the network, with some being crucial hubs across multiple measures while others excel in specific aspects.

(Michael B.) We used the largest connected component to find the **average shortest path length,** below is the code and its output:

```python
import pandas as pd
import networkx as nx

def get_largest_connected_component(file_path):
    # Read the CSV file into a pandas DataFrame
    df = pd.read_csv(file_path)

    # Create a graph from the DataFrame
    graph = nx.from_pandas_edgelist(df, 'numeric_id_1', 'numeric_id_2')

    # Find all connected components
    connected_components = nx.connected_components(graph)

    # Get the largest connected component
    largest_cc = max(connected_components, key=len)

    # Extract the largest connected component
    graph_lcc = graph.subgraph(largest_cc).copy()

    return graph_lcc

def average_shortest_path_length_lcc(file_path):
    # Get the largest connected component
    graph_lcc = get_largest_connected_component(file_path)

    # Calculate the average shortest path length using unweighted Dijkstra's algorithm
    avg_shortest_path_length = nx.average_shortest_path_length(graph_lcc, method='unweighted')

    return avg_shortest_path_length

# Path to the CSV file
file_path = r"C:\Users\akmik\OneDrive\Desktop\CSCI 347\Mini Project 2\large_twitch_edges.csv"

# Calculate and print the average shortest path length of the largest connected component
avg_length_lcc = average_shortest_path_length_lcc(file_path)
print("Average Shortest Path Length (largest connected component):", avg_length_lcc)
```

```
Average Shortest Path Length (largest connected component): 5.15508797387992
```

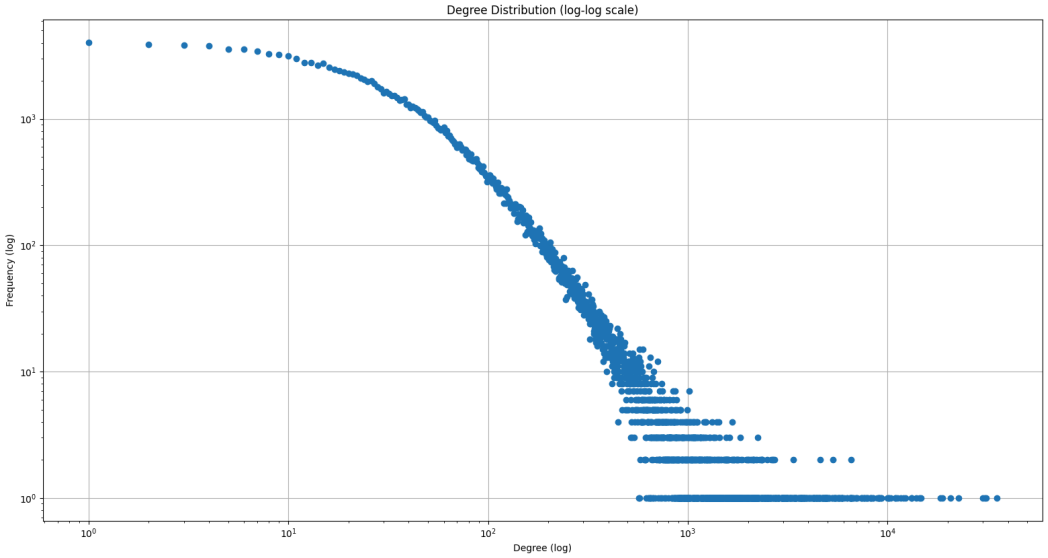
Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat

Based on our findings, such as the Twitch network's relatively high clustering coefficients and the small average shortest path length considering the sampled data size, **there is moderate evidence supporting the network exhibiting small-world behavior.** The network exhibits dense local connections and short average distances between nodes, characteristic of small-world networks. Further data and time to process could provide a more accurate analysis, but these findings suggest the Twitch network combines both local clustering and global connectivity features typical of small-world networks.

(Michael B.) Below is the code to generate the **Degree Distribution of the graph on a log-log scale plot** and its output**:**

```python
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

def plot_degree_distribution(file_path):
    # Read the CSV file into a pandas DataFrame
    df = pd.read_csv(file_path)

    # Create a graph from the DataFrame
    graph = nx.from_pandas_edgelist(df, 'numeric_id_1', 'numeric_id_2')

    # Get the degree of each node in the graph
    degrees = dict(graph.degree())

    # Plot the degree distribution on a log-log scale
    plt.figure(figsize=(10, 6))
    plt.title("Degree Distribution (log-log scale)")
    plt.xlabel("Degree (log)")
    plt.ylabel("Frequency (log)")

    # Compute degree distribution
    degree_values = list(degrees.values())
    degree_counts = pd.Series(degree_values).value_counts().sort_index()
    degree_counts.plot(marker='o', linestyle='None')

    plt.xscale('log')
    plt.yscale('log')
    plt.grid(True)

    plt.show()

# Path to the CSV file
file_path = r"C:\Users\akmik\OneDrive\Desktop\CSCI 347\Mini Project 2\large_twitch_edges.csv"

# Plot the degree distribution
plot_degree_distribution(file_path)
```

Mini Project Two: Data Mining
Megan Steinmasel, Michael Belmear, Rohan Kamat



The Twitch network's degree distribution demonstrates **power-law behavior,** indicating a scale-free structure where a small fraction of highly connected users or channels, termed hubs, significantly influence network dynamics. This distribution highlights the network's resilience to random failures and susceptibility to targeted attacks on hubs. It suggests a dynamic growth process where new connections preferentially attach to existing hubs, shaping the network's organization and resilience.