Zaitsev_1

September 25, 2024

1 Zaitsev Anton | Exercise 1

1.1 General Information

This exercise focuses on constructing and analyzing social networks (*Facebook* and *Twitter*) using data from *.edges* files. These networks are constructed using the *networkx* library in Python, and the analysis includes calculating the number of nodes and edges, finding the maximum and average degrees and extracting partial subgraphs.

1.2 Network construction

The *.edges* file is used to add both the nodes and the edges the between nodes. We do not consider *ego* node and *.feat* files (based on the feedback).

[1]:
```
import networkx as nx
import numpy as np
import pandas as pd

# global variables
DATA_FOLDER = "data/"

def construct_graph(edges_path: str, directed: bool = False) -> nx.Graph:
    """
    Construct a graph from an edge list.

    Parameters:
        - edges_path (str): The path to the .edges file, which contains the␣
 ↪edges of the graph.
        - directed (bool): If True, the graph will be treated as directed,␣
 ↪creating a NetworkX DiGraph.

    Returns:
        - g (nx.Graph or nx.DiGraph): A NetworkX graph containing all the nodes␣
 ↪and edges from the .edges file.
                                      Additionally, nodes present in the .feat␣
 ↪file but missing in the .edges
                                      file will be added to the graph as␣
 ↪disconnected nodes (without any edges).
```

```
    """
    # read graph from .edges file
    if directed:
        g = nx.read_edgelist(edges_path, create_using=nx.DiGraph(),␣
 ↪nodetype=int)
    else:
        g = nx.read_edgelist(edges_path, create_using=nx.Graph(), nodetype=int)
    return g

# load facebook network, indirected graph
G_fb = construct_graph(edges_path=f"{DATA_FOLDER}3437.edges", directed=False)
# load twitter network, directed graph
G_tw = construct_graph(edges_path=f"{DATA_FOLDER}3253671.edges", directed=True)
```

1.3 Question a

[2]:
```
# --------- a ---------

def get_nodes_edges(g: nx.Graph) -> tuple[int, int]:
    """
    Function to get the number of nodes and edges in a graph.

    Parameters:
        - g (networkx.Graph or networkx.DiGraph): The input graph.

    Returns:
        - n_nodes (int): The number of nodes in the graph.
        - n_edges (int): The number of edges in the graph.
    """
    # get the number of nodes
    n_nodes = g.number_of_nodes()
    # get the number of edges
    n_edges = g.number_of_edges()
    return n_nodes, n_edges

# get the number of nodes, edges for facebook network
n_nodes_fb, n_edges_fb = get_nodes_edges(G_fb)
# get the number of nodes, edges for twitter network
n_nodes_tw, n_edges_tw = get_nodes_edges(G_tw)
print(f"Facebook: number of nodes: {n_nodes_fb}, number of edges:␣
 ↪{n_edges_fb}\n"
      f"Twitter: number of nodes: {n_nodes_tw}, number of edges: {n_edges_tw}")
```

```
Facebook: number of nodes: 140, number of edges: 407
Twitter: number of nodes: 147, number of edges: 3942
```

1.4 Question b

[3]:
```
# --------- b ---------

def find_max_degree(g: nx.Graph) -> int:
    """
    Function to find the maximum degree of nodes in the graph.

    Parameters:
        - g (networkx.Graph or networkx.DiGraph): The input graph.

    Returns:
        - max_degree (int): The maximum degree of any node in the graph.
    """
    max_degree = max(dict(g.degree()).values())
    return max_degree

def find_avg_degree(g: nx.Graph, n_nodes: int, n_edges: int) -> float:
    """
    Function to find the average degree of the graph.
    For undirected graph, the formula is: avg_degree = 2*E/N, where
        - E = number of edges
        - N = number of nodes
    For directed graph, the formula is: avg_degree = E/N

    Parameters:
        - n_nodes (int): The number of nodes in the graph.
        - n_edges (int): The number of edges in the graph.

    Returns:
        - avg_degree (float): The average degree of the graph.
    """
    avg_degree = n_edges / n_nodes
    if not g.is_directed():
        avg_degree *= 2
    return avg_degree

# find the maximum degree for facebook, twitter networks
max_degree_fb, max_degree_tw = find_max_degree(G_fb), find_max_degree(G_tw)
# find the average degree for facebook, twitter networks
avg_degree_fb, avg_degree_tw = find_avg_degree(G_fb, n_nodes_fb, n_edges_fb),␣
 ↪find_avg_degree(G_tw, n_nodes_tw, n_edges_tw)
print(f"Facebook: max degree: {max_degree_fb}, average degree:␣
 ↪{round(avg_degree_fb, 2)}, {int(avg_degree_fb)} nodes\n"
      f"Twitter: max degree: {max_degree_tw}, average degree:␣
 ↪{round(avg_degree_tw, 2)}, {int(avg_degree_tw)} nodes")
```

```
Facebook: max degree: 25, average degree: 5.81, 5 nodes
```

```
Twitter: max degree: 174, average degree: 26.82, 26 nodes
```

1.5 Question c

[4]:
```python
# --------- c ---------

def extract_partial_network(g: nx.Graph, n_nodes: int = 5) -> tuple[np.ndarray,␣
↪list]:
    """
    Extracts a subgraph with n_nodes from graph g and returns the nodes of the␣
↪subgraph and adjacency matrix.
    To make it more interesting, subgraph_nodes are picked randomly until the␣
↪number of edges in the new subgraph is more than zero.

    Parameters:
        - g (networkx graph): The original graph.
        - n_nodes (int): Number of nodes to extract. Default is 5.

    Returns:
        - adj_matrix (numpy array): The adjacency matrix of the partial network.
        - subgraph_nodes (list): The list of nodes in the partial network.
    """
    # ensure we don't exceed the number of nodes in the graph
    if n_nodes > g.number_of_nodes():
        raise ValueError("The requested number of nodes exceeds the total␣
↪number of nodes in the graph.")
    n_subgraph_edges = 0
    n_tries = 1
    while n_subgraph_edges == 0:
        # randomly sample n_nodes from the original graph's nodes
        subgraph_nodes = np.random.choice(g.nodes(), n_nodes, replace=False)
        # extract the subgraph induced by the selected nodes
        subgraph = g.subgraph(subgraph_nodes)
        # get number of edges
        n_subgraph_edges = subgraph.number_of_edges()
        print(f"Try {n_tries}")
        n_tries += 1
    # get the adjacency matrix of the subgraph
    adj_matrix = nx.adjacency_matrix(subgraph).todense()
    return adj_matrix, subgraph_nodes

# get adjacency matricies for subgraphs of facebook, twitter networks
subgraph_matrix_fb, subgraph_nodes_fb = extract_partial_network(G_fb)
subgraph_matrix_tw, subgraph_nodes_tw = extract_partial_network(G_tw)
print(f"Facebook:\n{subgraph_matrix_fb}\n"
      f"Twitter:\n{subgraph_matrix_tw}")
```

```
Try 1
```

```
Try 2
Try 1
Facebook:
[[0 0 0 1 0]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [1 0 0 0 0]
 [0 0 0 0 0]]
Twitter:
[[0 0 0 0 0]
 [0 0 0 1 1]
 [0 0 0 0 1]
 [0 1 0 0 0]
 [0 0 0 0 0]]
```

Adjacency matrix provides a straightforward way to represent the connections in a network. With the adjacency matrix of a given graph we can reconstruct the original graph, since we have a direct access to the edges of the graph (there is an edge between nodes i and j if $A[i, j] \neq 0$, with A - adjacency matrix). Besides, we can use adjacency matrix to perform matrix operations to analyze the graph, e.g. find shortest path between nodes, perform clustering or to find isolated nodes, i.e. tell if the graph is connected or not. Generally, if the network is not big, adjacency matrix is a efficient way to perform matrix-based calculations to analyze networks.