Social Graphs, Recommendation Systems

Yusuf Abdi

08/20/2023

Part 1: Recommender System using Collaborative Filtering

Implement a Movie Recommendation System and run it on the Movie Lens Dataset (Train vs Test). Mesure performance on test set using RMSE

- First you are required to compute first a user-user similarity based on ratings and movies in common
- Second, make rating predictions on the test set followoing the KNN idea: a prediction (user, movie) is the weighted average of other users' rating for the movie, weighted by user-similarity to the given user.

```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import mean_squared_error
from math import sqrt
```

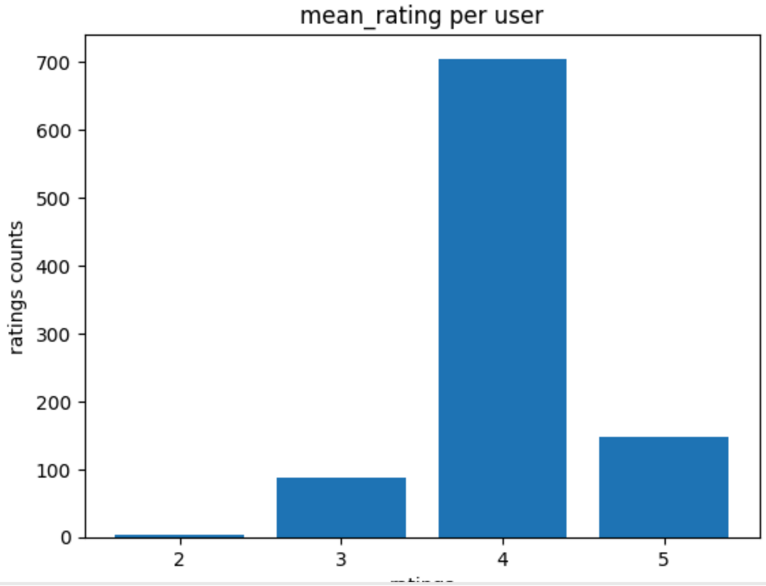
```
# Load the ratings data
# Columns: user_id, movie_id, rating, timestamp
ratings = pd.read_csv('/content/drive/MyDrive/jenkins/ml-100k/u.data', sep='\t', names=['user_id', 'movie_id', 'rating', 'times

# Load the movies data (for later use)
# Columns: movie_id, movie_title, release_date, ..., etc.
movies = pd.read_csv('/content/drive/MyDrive/jenkins/ml-100k/u.item',
                     sep='|',
                     encoding='latin-1',
                     names=['movie_id', 'movie_title',  'release_date', 'video_release_date', 'IMDb_URL',
                            'unknown', 'Action', 'Adventure', 'Animation', 'Childrens', 'Comedy', 'Crime',
                            'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'Musical', 'Mystery',
                            'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western'])
```

```
# mean ratings give by each user
mean_ratings = ratings.groupby('user_id').agg({'rating':'mean'})
mean_ratings['rating'] = np.ceil(mean_ratings['rating'])

# convert ratings to interger
mean_ratings['rating'] = mean_ratings['rating'].astype(int)

print(mean_ratings)
```

```
         rating
user_id
1             4
2             4
3             3
4             5
5             3
...         ...
939           5
940           4
941           5
942           5
943           4

[943 rows x 1 columns]
```

```
# write mean ratings to dict for easy plot
# plot number of use that gave a particular ratings
mean_ratings_dict = mean_ratings['rating'].value_counts().to_dict()
```

```
# plot bar chat
plt.figure()
plt.bar(mean_ratings_dict.keys(), mean_ratings_dict.values())
plt.title('mean_rating per user')
plt.xlabel('ratings')
plt.ylabel('ratings counts')
plt.xticks(np.arange(min(mean_ratings_dict.keys()), max(mean_ratings_dict.keys())+1, 1))
plt.gca().xaxis.set_major_locator(MultipleLocator(1))
plt.show()
```



```
# split data into test and train dataset
train_data, test_data = train_test_split(ratings, test_size=0.2, random_state=42)
```

```
# Create a user-item matrix
n_users = ratings['user_id'].nunique()
n_movies = ratings['movie_id'].nunique()

# create pivot table see similarities between users
user_item_matrix = pd.pivot_table(ratings, index='user_id', columns='movie_id', values='rating')
user_item_matrix.fillna(0, inplace=True)

# Compute the cosine similarity
user_similarity = cosine_similarity(user_item_matrix)
```

```
#  clear pivot table to see how users rate movies
user_item_matrix
```

movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1676	1677	1678	1679	1680	1681	1682
user_id																					
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	4.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
939	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
940	0.0	0.0	0.0	2.0	0.0	0.0	4.0	5.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
941	5.0	0.0	0.0	0.0	0.0	0.0	4.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
942	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
943	0.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
# write user similarity to dataframe for better readable form
user_similarity_df = pd.DataFrame(user_similarity)
user_similarity_df
```

	0	1	2	3	4	5	6	7	8	9	...	933	934	935
0	1.000000	0.166931	0.047460	0.064358	0.378475	0.430239	0.440367	0.319072	0.078138	0.376544	...	0.369527	0.119482	0.274876
1	0.166931	1.000000	0.110591	0.178121	0.072979	0.245843	0.107328	0.103344	0.161048	0.159862	...	0.156986	0.307942	0.358789
2	0.047460	0.110591	1.000000	0.344151	0.021245	0.072415	0.066137	0.083060	0.061040	0.065151	...	0.031875	0.042753	0.163829
3	0.064358	0.178121	0.344151	1.000000	0.031804	0.068044	0.091230	0.188060	0.101284	0.060859	...	0.052107	0.036784	0.133115
4	0.378475	0.072979	0.021245	0.031804	1.000000	0.237286	0.373600	0.248930	0.056847	0.201427	...	0.338794	0.080580	0.094924
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
938	0.118095	0.228583	0.026271	0.030138	0.071459	0.111852	0.107027	0.095898	0.039852	0.071460	...	0.066039	0.431154	0.258021
939	0.314072	0.226790	0.161890	0.196858	0.239955	0.352449	0.329925	0.246883	0.120495	0.342961	...	0.327153	0.107024	0.187536
940	0.148617	0.161485	0.101243	0.152041	0.139595	0.144446	0.059993	0.146145	0.143245	0.090305	...	0.046952	0.203301	0.288318
941	0.179508	0.172268	0.133416	0.170086	0.152497	0.317328	0.282003	0.175322	0.092497	0.212330	...	0.226440	0.073513	0.089588
942	0.398175	0.105798	0.026556	0.058752	0.313941	0.276042	0.394364	0.299809	0.075617	0.221860	...	0.263791	0.210763	0.143253

943 rows × 943 columns

```
# predict using k = 40 as intial test

def predict(ratings, similarity, k=40):
    pred = np.zeros(ratings.shape)
    for i in range(ratings.shape[0]):
        # Get top k similar users' indices
        top_k_users = np.argsort(similarity[:,i])[-1:-k-1:-1]
        for j in range(ratings.shape[1]):
            pred[i, j] = similarity[i, :][top_k_users].dot(ratings[top_k_users, j])
            sum_abs_similarities = np.sum(np.abs(similarity[i, :][top_k_users]))
            if sum_abs_similarities > 0:
                pred[i, j] /= sum_abs_similarities
            else:
                pred[i, j] = 0
    return pred

# Get the predictions
user_prediction = np.round(predict(user_item_matrix.values, user_similarity))
```

```
user_prediction
```

```
array([[4., 2., 1., ..., 0., 0., 0.],
       [2., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [4., 0., 1., ..., 0., 0., 0.],
       [2., 1., 0., ..., 0., 0., 0.],
       [3., 3., 2., ..., 0., 0., 0.]])
```

```
# write prediction to dataframe
prediction_df = pd.DataFrame(user_prediction)

# approximate preidction to the neariest whole number
prediction_df = prediction_df.applymap(math.ceil)

# name index
prediction_df.index.name = 'user_id'

# name column
prediction_df.columns.name = 'movie_id'

prediction_df.head()
```

movie_id	0	1	2	3	4	5	6	7	8	9	...	1672	1673	1674	1675	1676	1677	1678	1679	1680	1681
user_id																					
0	4	2	1	3	1	0	4	2	3	1	...	0	0	0	0	0	0	0	0	0	0
1	2	0	0	0	0	0	2	0	2	1	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	4	2	1	2	1	0	3	1	1	1	...	0	0	0	0	0	0	0	0	0	0

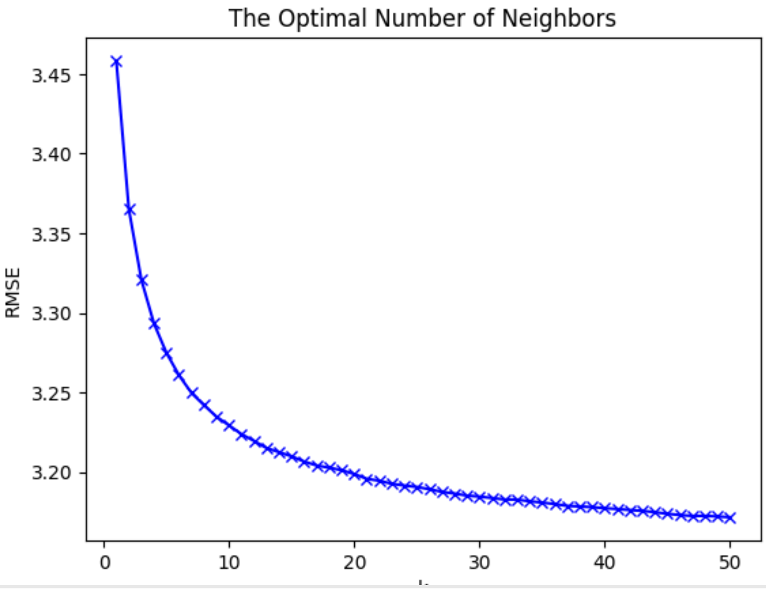
```
# get rmse score when k is 40

def get_rmse(pred, actual):
    # Ignore nonzero terms.
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return sqrt(mean_squared_error(pred, actual))

# Calculate the RMSE
test_user_item_matrix = pd.pivot_table(test_data, index='user_id', columns='movie_id', values='rating')
test_user_item_matrix.fillna(0, inplace=True)

# compare prediction_df and test item matrix
rmse = get_rmse(user_prediction, test_user_item_matrix.values)

print('User-based CF RMSE: ' + str(rmse))
```

```
User-based CF RMSE: 3.222530061923395
```

```
# hyper parameter tunning check for a better RMSE for k in range 1 to 50

# dict to store RMSE values for different k
rmse_val = {}

# Use a loop to compute RMSE for different k
for K in range(50):
    K = K + 1
    user_prediction = predict(user_item_matrix.values, user_similarity, k=K)

    # Calculate the RMSE
    rmse = get_rmse(user_prediction, test_user_item_matrix.values)

    # store rms in dictionary
    rmse_val[K] = rmse
    print('RMSE value for k= ' , K , 'is:', rmse)
```

```
# Plotting the RMSE values against k values
plt.plot(range(1, 51), rmse_val.values, 'bx-')
plt.xlabel('k')
plt.ylabel('RMSE')
plt.title('The Optimal Number of Neighbors')
plt.show()
```

```
# Plotting the RMSE values against k values
plt.plot(range(1, 100, 2), rmse_val.values(), 'bx-')
plt.xlabel('k')
plt.ylabel('RMSE')
plt.title('The Optimal Number of Neighbors')
plt.show()
```



```
# get lowwest rmse from loop
best_k = min(rmse_val, key=rmse_val.get)
print(f'best k = {best_k}')
```

```
best k = 50
```

```
# predict using k = 50

def predict(ratings, similarity, k=best_k):
    pred = np.zeros(ratings.shape)
    for i in range(ratings.shape[0]):
        # Get top k similar users' indices
        top_k_users = np.argsort(similarity[:,i])[-1:-k-1:-1]
        for j in range(ratings.shape[1]):
            pred[i, j] = similarity[i, :][top_k_users].dot(ratings[top_k_users, j])
            sum_abs_similarities = np.sum(np.abs(similarity[i, :][top_k_users]))
            if sum_abs_similarities > 0:
                pred[i, j] /= sum_abs_similarities
            else:
                pred[i, j] = 0
    return pred

# Get the predictions
user_prediction = np.round(predict(user_item_matrix.values, user_similarity))
```

```
# get rmse score best_k

def get_rmse(pred, actual):
    # Ignore nonzero terms.
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return sqrt(mean_squared_error(pred, actual))

# Calculate the RMSE
test_user_item_matrix = pd.pivot_table(test_data, index='user_id', columns='movie_id', values='rating')
test_user_item_matrix.fillna(0, inplace=True)

# compare prediction_df and test item matrix
rmse = get_rmse(user_prediction, test_user_item_matrix.values)

print('User-based CF RMSE: ' + str(rmse))
```

```
User-based CF RMSE: 3.2126468838015794
```

Part 3A: Social Community Detection

Implement edge-removal community detection algorithm on the Flicker Graph. Use the betweeness idea on edges and the Girvan−Newman Algorithm. The original dataset graph has more than 5M edges; in DM_resources there are 4 different sub-sampled graphs with edge counts from 2K to 600K; you can use these if the original is too big. You should use a library to support graph operations (edges, vertices, paths, degrees, etc). We used igraph in python which also have builtin community detection algorithms (not allowed); these are useful as a way to evaluate communities you obtain

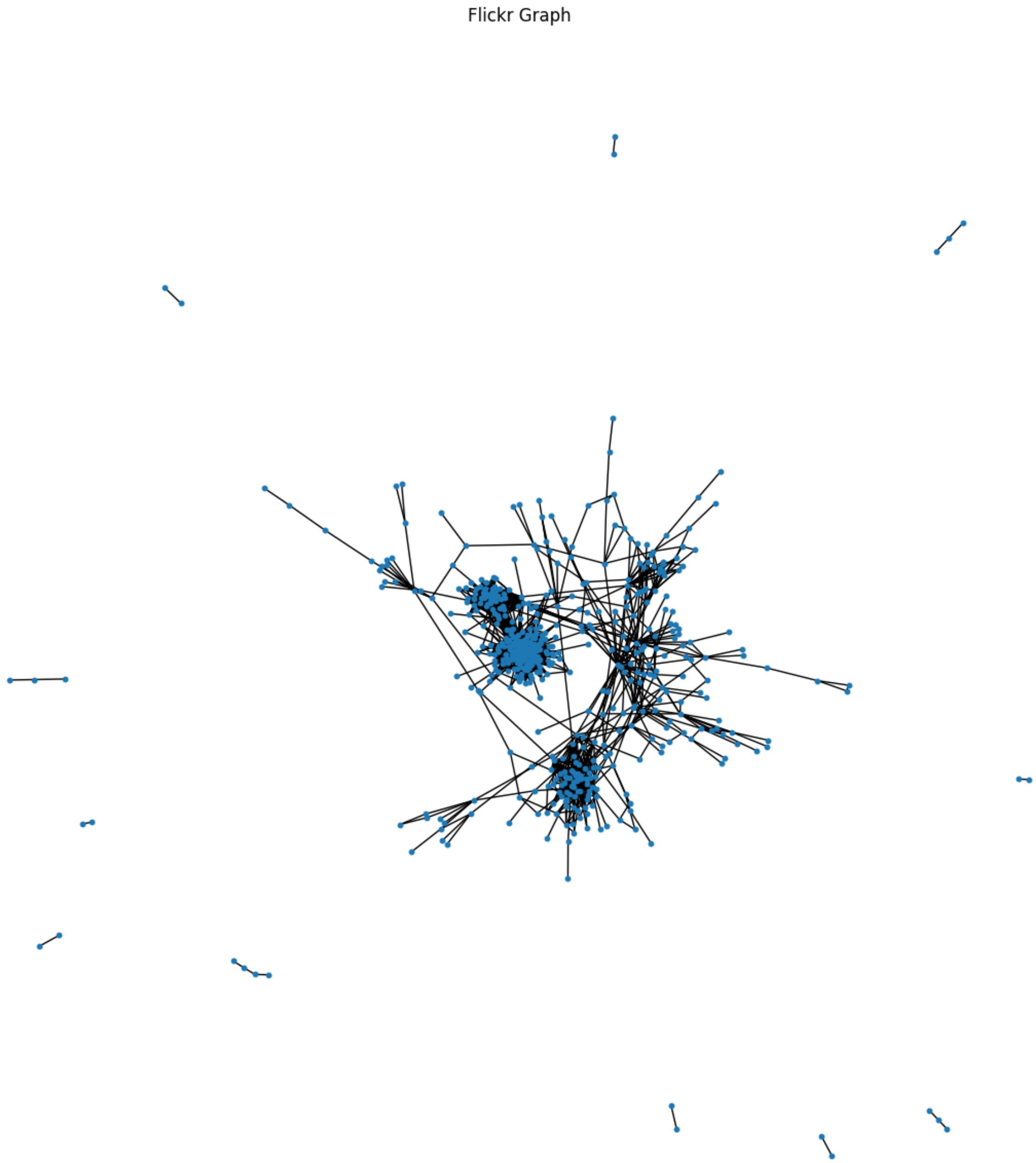
```
!pip install networkx
```

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

```
# Specify the correct path to the edge list file
file_path = "/content/drive/MyDrive/jenkins/Flickr-dataset/Flickr-dataset/data/Flickr_sampled_edges/edges_sampled_3K.csv"

# Load the graph from the edge list file
G = nx.read_edgelist(file_path, delimiter=',')

# Print the number of nodes in the loaded graph
print("Number of nodes:", G.number_of_nodes())
```

```
Number of nodes: 540
```

```
# plot graph
plt.figure(figsize=(12, 12))
nx.draw(G, with_labels=False, node_size=10)
plt.title("Flickr Graph")
plt.show()
```



```
import networkx as nx

def girvan_newman(G):
    # Function to compute the communities in a graph using the Girvan–Newman algorithm

    # Copy the graph to avoid modifying the original one
    G_copy = G.copy()

    # This list will contain the resulting communities
    communities = []

    # The algorithm continues until we have more than one community
    while nx.is_connected(G_copy):

        # Compute the edge betweenness centrality for all edges in the graph
        edge_betweenness = nx.edge_betweenness_centrality(G_copy)

        # Find the edge with the highest betweenness centrality
        max_edge = max(edge_betweenness, key=edge_betweenness.get)

        # Remove the edge with the highest betweenness centrality
        G_copy.remove_edge(*max_edge)
    # After the graph is split, we consider each connected component as a community
    communities = list(nx.connected_components(G_copy))

    return communities


# Run the Girvan-Newman algorithm
communities = girvan_newman(G)

# Print the resulting communities
for i, community in enumerate(communities):
    print(f"Community {i + 1}: {community}")
```

```
Community 1: {'24203', '72820', '23006', '62504', '16803', '76086', '37432', '35041', '14465', '79943', '1602', '69970', '
Community 2: {'39171', '8187'}
Community 3: {'42792', '15759', '4748'}
Community 4: {'2957', '22626', '27570'}
Community 5: {'11862', '26314'}
Community 6: {'17790', '10069', '14123'}
Community 7: {'6628', '25610'}
Community 8: {'27429', '38827'}
Community 9: {'31786', '10228', '18244', '24188'}
Community 10: {'63994', '60242'}
Community 11: {'42383', '45915'}
Community 12: {'18760', '9790'}
```

```
## to check using built in algorithm
# Evaluate detected communities using builtin algorithm
from networkx.algorithms.community import girvan_newman as gn_builtin

# Run the built-in Girvan-Newman algorithm
communities_builtin = list(gn_builtin(G))

# Print the resulting communities
for i, community in enumerate(communities_builtin):
    print(f"Community {i + 1}: {community}")
```

Part 3B: Social Community Detection

Implement the modularity detection algorithm on the Flicker Graph in previous problem. You will need to compute the modularity matrix B and its highest-val eigenvector V1. The split vector S (+1 / -1) aligns by sign with V1; follow this paper. Split the graph repeatedly in two parts until the desired number of components is obtained.

```
import networkx as nx

# Specify the correct path to the edge list file
file_path = "/content/drive/MyDrive/jenkins/Flickr-dataset/Flickr-dataset/data/Flickr_sampled_edges/edges_sampled_3K.csv"

# Load the graph from the edge list file
G = nx.read_edgelist(file_path, delimiter=',')

# Print the number of nodes in the loaded graph
print("Number of nodes:", G.number_of_nodes())
```

```
Number of nodes: 540
```

```
print("Number of nodes:", G.number_of_nodes())
print("Number of edges:", G.number_of_edges())
```

```
Number of nodes: 540
Number of edges: 2694
```

```
# Compute the modularity matrix for a given graph G
def modularity_matrix(G):
    # Convert the adjacency matrix of the graph G into a numpy array
```

```python
    A = nx.to_numpy_array(G)
    # Calculate the sum of rows (or columns) which represents the degree of each node
    degrees = A.sum(axis=1)
    # Get the total number of edges in the graph
    m = G.number_of_edges()
    # Compute the modularity matrix B
    B = A - (degrees @ degrees.T) / (2 * m)
    return B


# Find the leading eigenvector of the modularity matrix B
def leading_eigenvector(B):
    # Get eigenvalues and eigenvectors of the matrix B
    eigenvalues, eigenvectors = np.linalg.eigh(B)
    # Identify the largest eigenvalue's index
    largest = np.argmax(eigenvalues)
    # Return the eigenvector corresponding to the largest eigenvalue
    return eigenvectors[:, largest]

# Split nodes based on the sign of their values in the leading eigenvector
def split_vector(V1):
    S = np.sign(V1)
    return S



# Generate subgraphs based on positive and negative values in the leading eigenvector
def split_graph(G, S):
    # Nodes with positive values
    positive_nodes = [node for node, sign in zip(G.nodes(), S) if sign > 0]
    # Nodes with non-positive values
    negative_nodes = [node for node, sign in zip(G.nodes(), S) if sign <= 0]
    # Create subgraphs for the separated node sets
    return G.subgraph(positive_nodes), G.subgraph(negative_nodes)



# Main function to detect communities within the graph G
def detect_communities(G, desired_num_communities):
    # If the graph has no nodes or we've reached the desired number of communities, stop further splitting
    if G.number_of_nodes() == 0 or desired_num_communities <= 1:
        return [G]

    # Compute the modularity matrix for G
    B = modularity_matrix(G)
    # Get the leading eigenvector of B
    V1 = leading_eigenvector(B)
    # Determine the split based on the leading eigenvector
    S = split_vector(V1)
    # Split the graph into two subgraphs based on the split vector S
    G1, G2 = split_graph(G, S)

    communities = []

    # Recursively detect communities in the split subgraphs
    communities += detect_communities(G1, desired_num_communities - 1)
    communities += detect_communities(G2, desired_num_communities - 1)

    return communities



# Define the desired number of communities
desired_num_communities = 4

# Run the algorithm
communities = detect_communities(G, desired_num_communities)

# Print the resulting communities
for i, community in enumerate(communities):
    print(f"Community {i + 1}: {set(community.nodes())}\n")
```

```
Community 1: {'36203', '38019', '14492', '7361', '4038', '5727', '30267', '7502', '8880', '16803', '30146', '49376', '2541

Community 2: {'33583', '21462', '35099', '38786', '7375', '56467', '24203', '43736', '34375', '23006', '48921', '54740', '
Community 3: {'64260', '74661', '69903', '70343', '66972', '80023', '79111', '22782', '70706', '76086', '67027', '26208',

Community 4: {'63989', '72820', '67306', '79257', '73799', '68987', '73782', '44287', '79943', '67207', '73785', '73787',

Community 5: {'55689', '20410', '46519', '62498', '62247', '46808', '62053', '19959', '57360', '61984', '32614', '62279',

Community 6: {'65041', '46512', '57345', '62482', '62504', '61934', '46479', '46497', '46515', '62475', '46476', '57354',

Community 7: {'79670', '77993', '69883', '48161', '67114', '67012', '77917', '67194', '67063', '72788', '46355', '69832',

Community 8: {'79745', '69041', '76727', '63448', '57392', '66897', '41073', '67851', '67266', '75053', '67070', '41056',
```

Part 4: Knowledge Base Question Answering

Given is knowledge graph with entities and relations, questions with starting entity and answers, and their word embedding . For each question, navigate the graph from the start entity outwards until you find appropriate answer entities.

- there might be multiple answers correct, use F1 to evaluate
- utils functions (similarity, load_graphs) are given, but you can choose not to use them
- answers are given to be used for evaluation only
- your strategy should be a graph traversal augmented with scoring of paths; you might discard paths not promising along the way. This is similar to a focused crawl strategy.
- for simplicity, the questions are picked so that the answer is always at the end of the relevant path (not intermediary)

```
import gensim
from nltk import word_tokenize
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from collections import defaultdict
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import networkx as nx
import csv
from collections import deque
```

```
# load graph
def load_knowledge_graph(filename):
    G = nx.DiGraph()  # A directed graph, since knowledge graphs typically have direction (from source to target through a re

    with open(filename, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # Skip header row

        for row in reader:
            source, relation, target = row
            G.add_edge(source, target, relation=relation)

    return G

# Load the knowledge graph
graph = load_knowledge_graph('/content/drive/MyDrive/jenkins/graph.txt')
```

```
import networkx as nx
import matplotlib.pyplot as plt
# function to plot graph

def display_graph(G):
    # Draw the graph
    pos = nx.spring_layout(G)  # positions for all nodes; you can try other layout algorithms too
    nx.draw(G, pos, with_labels=True, font_weight='bold', node_color='skyblue', node_size=700, width=1.0, edge_color='gray')

    # Draw edge labels
    edge_labels = nx.get_edge_attributes(G, 'relation')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8)

    plt.show()
```

```
display_graph(graph)
```

```
import networkx as nx

knowledge_graph = []
annotations = []

with open('/content/drive/MyDrive/jenkins/graph.txt', 'r') as file:
    for line in file:
        triple = line.strip().split(', ')
        knowledge_graph.append(tuple(triple))

import ast

with open('/content/drive/MyDrive/jenkins/annotations.txt', 'r') as file:
    for line in file:
        annotation = ast.literal_eval(line.strip())
        annotations.append(annotation)


# Create a directed graph from the knowledge graph
G = nx.Graph()
for triple in knowledge_graph:
    start, relation, end = triple

    # Remove square brackets from start and end node names
    start = start.replace('[', '').replace(']', '')
    end = end.replace('[', '').replace(']', '')

    # Remove '' from start and end node names
    start = start.replace("'", '').replace("'", '')
    end = end.replace("'", '').replace("'", '')

    G.add_edge(start, end, relation=relation)



print("Number of nodes:", G.number_of_nodes())
print("Number of edges:", G.number_of_edges())
```

```
Number of nodes: 297
Number of edges: 979
```

```
# Define a function to compute the similarity between a relation and a query.
def similarity(relation, query):
    # Vectorize the relation and query using TF-IDF.
    vectorizer = TfidfVectorizer().fit_transform([relation, query])

    # Convert the vectorized data to arrays.
    vectors = vectorizer.toarray()

    # Compute cosine similarity between the relation and query.
    cosine_sim = cosine_similarity(vectors)

    # Return the similarity score.
    return cosine_sim[0][1]


# Define BFS traversal to explore the graph and find answer nodes.
def bfs_traversal(G, start, query, max_depth=3, threshold=0.03):
    # Check if the starting node exists in the graph.
    if start not in G:
        print(f"Start node {start} is not in the graph.")
        return []

    # Initialize the answers list and visited set.
    answers = []
    visited = set()

    # Initialize the current layer with the starting node and an empty next layer.
    current_layer = deque([start])
    next_layer = deque()

    # Initialize the depth.
    depth = 0

    # Traverse the graph while there are nodes in the current layer and the depth is within limits.
    while current_layer and depth < max_depth:
        # Remove and return a node from the left side of the current layer.
        node = current_layer.popleft()

        # Add the current node to the visited set.
        visited.add(node)

        # Get neighbors of the current node.
        try:
            neighbors = list(G.neighbors(node))
        except NetworkXError as e:
            print(f"Error: {e}")
            continue

        # For each neighbor of the current node.
        for neighbor in neighbors:
            # If the neighbor has not been visited.
            if neighbor not in visited:
                # Get the relation associated with the edge to the neighbor.
                relation = G[node][neighbor]['relation']

                # Replace '.' and '_' with spaces in the relation for better text matching.
                relation = relation.replace('.', ' ').replace('_', ' ')

                # Compute the similarity score of the relation and the query.
                score = similarity(relation, query)

                # If the similarity score is above the threshold.
                if score >= threshold:
                    # Add the neighbor to the next layer and to the answers list.
                    next_layer.append(neighbor)
                    answers.append(neighbor)
                #elif node not in answers:
                    #answers.append(node)

        # If there are no nodes left in the current layer, move to the next depth.
        if not current_layer:
            depth += 1
            # Swap the current layer and the next layer.
            current_layer, next_layer = next_layer, current_layer

    # Return the list of answer nodes.
    return answers
```

```
# This function computes the F1 score based on predicted and true sets of answers.
def compute_f1(predicted, true):
    # Calculate the number of true positives: items that are both in the predicted and true sets.
    tp = len(set(predicted) & set(true))

    # Calculate the number of false positives: items that are in the predicted set but not in the true set.
    fp = len(set(predicted) - set(true))

    # Calculate the number of false negatives: items that are in the true set but not in the predicted set.
    fn = len(set(true) - set(predicted))

    # If there are no true positives, the F1 score is 0.
    if tp == 0:
        return 0

    # Calculate the precision: the ratio of true positives to the sum of true positives and false positives.
    precision = tp / (tp + fp)

    # Calculate the recall: the ratio of true positives to the sum of true positives and false negatives.
    recall = tp / (tp + fn)

    # Calculate the F1 score: the harmonic mean of precision and recall.
    f1 = (2 * precision * recall) / (precision + recall)

    return f1



# Lists to store the F1 scores, predicted answers, and true answers for each question.
f1_scores = []
prediction = []
true = []

# For each annotation (or question), retrieve the query, start node, and true answers.
# Then, predict answers using BFS traversal and compute the F1 score.
for annotation in annotations:
    query = annotation[1]
    start_node = annotation[2]
```