Assignment 2: Wikipedia Graph Analysis

Anupam Kumar (11940160)

1. Scraping and Labelling of Articles:

- This was done and submitted beforehand, we labeled the topics of Mathematics as mentioned and assigned labels 1,2,3, and 0 corresponding to BEGINNER, INTERMEDIATE, ADVANCED, and IRRELEVANT although we didn't need to use the label 0 for any label.

2. Building the Wikipedia Graph:

- The dataset(csv file) prepared in Part 1 was used to construct the graph.
- The columns of the dataset which contain the link and the label of the topics mostly concern us, so we collect that and store it in the dataset.
- Later, we also remove the rows in the dataset which contain NaN values.
- Data Cleaning:
  i. The data is cleaned to get rid of the duplicate links and those links which are the same but have been assigned different labels.
  ii. In case of the same topics being assigned different labels, we have taken the label which has the highest frequency.

```
for j in range(len(dataset)):
    labels = [] # list to store all labels

    for i in range(len(dataset)): # iterate over all rows
        try:
            try:
                t1 = dataset['Link'][j] # get the link
            except:
                pass
            try:
                t2 = dataset['Link'][i]
            except:
                pass

            if t1 == t2: # if the link is same
                try: # try to get the label
                    labels.append(dataset['Label'][i])
                except: # if label is not present
                    pass

        except ValueError: # if the link is not same
            pass
```

- Creation of root nodes in the Graph:
  i. For the creation of root nodes, we have taken the first 10 entries in the dataset and considered them as roots.

```
# As of now, we assign 10 as the number of root nodes
n = 10
root = [] # root nodes
difficulty = {} # dictionary to store difficulty of each node

for i in range(n):
    cur_node = dataset['Link'][i] # get the current node
    label = dataset['Label'][i] # get the label of the current node

    difficulty[cur_node] = label # add the difficulty of the current node to the dictionary
    root.append(cur_node) # add the current node to the root nodes

print(root)
```

- Deciding the child nodes:
  i. For each root, the child nodes are decided by parsing the link of the root node using Beautiful Soup and fetching all the links which can be found on that page.
  ii. After this, we check all the entries which match the links obtained and then assign those nodes as the child node for that root node and hence, create an edge between them.
- Further, the adjacency list for the Graph is made for traversal.
- Now, the constructed graph is visualized and displayed.

3. Creation of Additional features using Centrality Metrics and Clustering coefficient:

- The centrality metrics - degree centrality, closeness centrality, betweenness centrality, and Page Rank are calculated for the graph using their definition.
- Similarly, the clustering coefficient is also calculated.

4. Development of Node Classification Model:

- Initially, the nodes are converted to vectors using the node2vec library.
- Node to Vector conversion:
  i. The networkx graph is created using the initial graph obtained above.
  ii. The embedding dimension is kept at 16.
  iii. The number of nodes in each walk is kept at 80.
  iv. The number of walks per node is 10 and,
  v. The number of workers for parallel execution is 1.

- Now, the embedded nodes(embeddings) are created with window_size as 10, minimum count as 1, and batch_words as 4.

```
# conversion of Node to vector
import networkx as nx
from node2vec import Node2Vec
# define the graph to be used in the node2vec
ncmGraph = G_vis.getGraph()
# use the Node2vec to generate graph embeddings
node2vec = Node2Vec(ncmGraph, dimensions=16, walk_length=80, num_walks=10, workers=1)
# get the embeddings
embeddings = node2vec.fit(window=10, min_count=1, batch_words=4)
```

- After that, we create a dataframe for the embeddings of the nodes.

- Further, we create a dictionary to store the link to label mapping and this mapping is used to label embedding dataframe corresponding to the labeled embeddings and unlabeled embedding dataframe corresponding to the unlabelled embeddings.

- Consequently, we train the model using the Gradient Boosting Classifier where the number of boosting stages is 100 by default and the learning rate is kept at 0.1.

```
col = labelled_embDF.drop(columns = ['Label']).columns.tolist() # list of the columns of the labelled

# Splitiing the labelled embeddings dataframe into train and test dataframes
X = labelled_embDF[col].values
Y = labelled_embDF['Label'].values

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)

# Using the Gradient Boosting Classifier
gb_clf = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=1, random_state=0)
```

- After the training is completed, we print the testing accuracy, the confusion matrix, and the classification report at the end.

5. Graph Traversal and Article Ordering Algorithm:

- The graph traversal is done before as well since we needed to traverse the graph for calculating the different centrality metrics and the clustering coefficient.
- The traversal is done using the basic Graph Traversal Algorithm: BFS.
- We create a class for the visualization of the graph where matplotlib is used to create the figure of the graph.

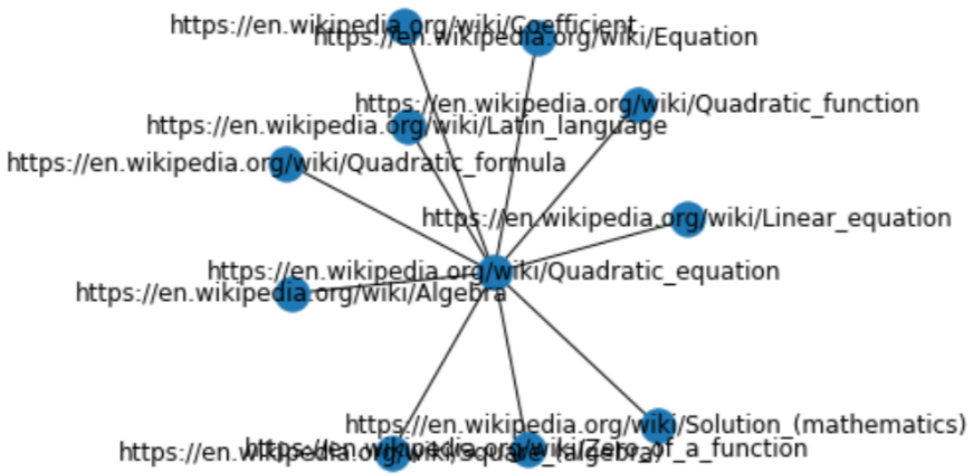
```
# Visualise the class to visulaise the graph
class GraphVisualization:
    def __init__(self):
        self.visual = [] # list to store the visualisation

    # addEdge - adds an edge to the visualisation
    def addEdge(self,u,v):
        self.visual.append((u,v)) # add the edge to the visualisation

    # visualize - visualises the graph
    def visualize(self):
        G = nx.Graph() # create a graph
        G.add_edges_from(self.visual) # add the edges to the graph
        nx.draw_networkx(G) # draw the graph
        plt.rcParams['figure.figsize'] = [50, 50] # set the size of the graph to be displayed
        plt.show() # display the graph

    # getGraph: returns the graph
    def getGraph(self):
        G = nx.Graph() # create a graph
        G.add_edges_from(self.visual) # add the edges to the graph
        return G # return the graph
```

- The visualize function generates the figure of the graph. One of the connected components of the graph formed is shown below:



- Article Ordering Algorithm:
  i. This algorithm takes in a topic link and prints the topics in order of easy to difficult based on the model prediction.
  ii. First, the neighbors of the topic are found out.
  iii. Now, for all the topics in the neighbors, if the topic is unlabeled then, it assigned a label using the prediction model else the initial label is used.
  iv. Finally, the topics obtained after this are ordered from beginner(1) to advanced(3) in increasing order and displayed to the user.