Assignment 4

Due: 16th **April 2020** **Total: 80 points**

Instructions

- Institute Plagiarism Policy applies to all HWs.
- It has been made explicit wherever usage of `networkx` or `snap.py` is required. In all other cases, these libraries can only be used for loading and representing the graphs. The implementations need to be done from scratch.
- **Submission Instructions:**
  - All the submissions must be inside a `zip` file named `a4_<your_roll_number>.zip` containing a report named `report.pdf` and a folder named `src` containing all your scripts.
  - All the code must be well documented. Failure to do so will result in a deduction of 2% of the total from the obtained marks in the respective question.
  - All the plots and analyses required should be uploaded in a `PDF` file named `report.pdf`.

Problem 1: Link Prediction (30 points)

1. Load the `fb-pages-food` dataset using NetworkX. The data file is shared with the assignment.
2. Create a copy of this graph by randomly removing 25% of the edges.
3. Now for each pair of nodes in the graph, which do not have an edge between them, compute the following Link Prediction heuristics:
   - Common Neighbors
   - Jaccard Index
   - Adamic-Adar Index
   - Preferential Attachment
4. Now, for each edge not in the graph, you have a probabilistic score of it being an actual edge. Compute the ROC-AUC score for each index.
5. Repeat the above steps across $k = 10$ folds. In each fold, remove 25% of the edges chosen randomly with replacement from the original graph. Report the average ROC-AUC score across the 10 folds.
6. Repeat this k-fold experiment while increasing the percentage of edges removed from the original graph. Try with percentages: 25%, 35%, 45%, 55%. Plot the average ROC-AUC score obtained in each case. What do you observe?

Problem 2: Node2Vec (20 points)

- In this question we'll be getting some hands on experience with Node2Vec in order to solidify our intuition behind the algorithm. For more details, you can find the research paper on Node2Vec here.
- To complete this problem you will need to clone or download the Node2Vec repository onto your computer. We recommend you develop in the `src` folder. Take a look at `main.py` and their `README` to get a grasp of how to run the Node2Vec code.
- You can use NetworkX for this problem.

Problem 2a: Visualizing the Graph (1 point)

Load the `karate.edgelist` into an undirected graph and create a visualization of the resultant graph (an easy way to do this is with NetworkX). Take a look at node 33. What sort of structural role does it seem to play in the Karate graph?

Problem 2b: DeepWalk (2 points)

- Recall that Node2Vec works by carrying out a number of random walks from each node in the graph, where the walks are parameterized by p and q. More precisely, after having just traversed the edge from node node t to node v, the unnormalized transition probability of travelling from node v to a neighboring node x is given by:

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p}, & \text{if } d_{tx} = 0 \\ 1, & \text{if } d_{tx} = 1 \\ \frac{1}{q}, & \text{if } d_{tx} = 2 \end{cases}$$

  where d_{tx} is the shortest path distance between nodes t and x.

- DeepWalk is a similar algorithm to Node2Vec that also learns representations for each node through random walks. However, unlike Node2Vec, at each node the next neighbor to walk to is chosen uniformly at random.
- You can read more about DeepWalk here.
- One can think of DeepWalk as a specific case of Node2Vec. Explain briefly but specifically (1-2 sentences) why this is true, and how one can use Node2Vec to simulate DeepWalk.

Problem 2c: Finding Clusters (6 points)

- The main advantage of Node2Vec is its flexibility in the type of similarities it can infer between nodes in a graph.
- Without calculating the shortest path distance between any nodes in the Karate graph, give a list of the five nodes with the lowest shortest path distances to node 33 (there may be more than five nodes that are direct neighbors of node 33 - any 5 will do). Explain clearly how you generated this list, how you set p and q, and the reasoning behind your methodology.
- *We won't be grading the correctness of your outputted nodes, but rather the correctness of the rationale behind your methodology – your approach to this question should be clearly and concisely justified.*

Problem 2d: Finding Structural Similarity (6 points)

Given the fact that the node with the highest degree in the Karate graph is node 34, and without calculating the degrees of any other nodes, give a list of the five nodes that are also likely to have moderately high degrees. Explain clearly how you generated this list, how you set p and q, and the reasoning behind your methodology.

Problem 2e: Neighborhood Similarity (5 points)

- Intuitively, Node2Vec works by creating a representation of a 'neighborhood' around each node through random walks, then generating vector representations of each node that preserve the neighborhood similarity.
- Setting $p = 100$ and $q = 0.01$, find, in the Karate graph, the node whose Node2Vec representation has the closest L2 (Euclidean) distance from node 33.

- What are the degrees of node 33 and this node?
- Is this what you would expect? Give a high level explanation of this result (it may help to refer to your visualization of the Karate graph from part 1).

Problem 3: GCNs for Community Detection (30 marks)

1. We dig deeper into the well-known "Zachary's karate club" social network that we saw in Problem 2. The network captures 34 members of a karate club, documenting pairwise links between members who interacted outside the club. The club later splits into two communities led by the instructor (node 0) and club president (node 33). You could read more about the story in the wiki page. You can access the graph using `nx.karate_club_graph()` in NetworkX.

2. In this problem, we will try to leverage the power of GCNs for community detection in Zachary's Karate Club dataset, using the `dgl` library in conjunction with PyTorch. If you have difficulty training this on your system, please make use of Google Colab. Document each and every line of your model (this will be reviewed stringently) explaining why you have written each line.

3. Before you start attempting this question, it is highly recommended that you read through a great tutorial on using `dgl` for coding GCNs from here. You might need to read up previous tutorials too, but this link is where you will get the most help. Note that this is just to get you familiar with the `dgl` API only.

4. First, load the Karate Club network as a graph object that you can manipulate. (**Hint:** You might want to check out `dgl.DGLGraph`)

5. Nodes and edges in `DGLGraph` can have feature tensors. Features of multiple nodes/edges are batched on the first dimension. (**Hint:** Check out the `ndata` attribute for `DGLGraph`).

6. Initialise a random embedding for every node that is of a dimension of your choice. For starting, try keeping the size of the node embedding to 5? Play around from there!

7. Graph convolutional network (GCN) is a popular model proposed by Kipf & Welling to encode graph structure. The model consists of several layers, each perform convolution-like operation defined on graph:

$$Y = \hat{A}XW$$

, where X is the node embedding tensor (stacked along the first dimension), W is a projection matrix (the weight parameter) and $\hat{A}$ is the normalized adjacency matrix:

$$A = D^{-\frac{1}{2}}AD^{\frac{1}{2}}$$

8. The equations above involve a matrix multiplication between the normalized adjacency matrix and node features. And this can be expressed in terms of message passing paradigm:

   - **message phase:** all nodes first compute and send out messages along out-going edges.
   - **reduce phase:** all node then collect in-coming messages, aggregate them and update their own embedding.

   How to implement this using `dgl` has been shown in the tutorial shared. Note however that in the tutorial these steps are followed in the reverse order. You can follow whatever order you want, it does not make much of a difference.

9. You can choose how to define your **message functions** and **reduce functions** to construct variations of the GCN described in the original paper. However, for starting out, follow these definitions:

- **message function**: Suppose the current embedding of node i after the linear transformation (i.e. multiplying weight matrix W) is h_i. From the equation of GCN above, each node sends out the embedding after linear transformation to their neighbors. Then the message from node j to node i can be computed as:

  $$m_{j \to i} = h_j$$

  So the message function takes out node feature `h` as the message, and can be defined using DGL's built-in functions. (**Hint:** You must have already seen `dgl`'s internal functions in action in the tutorial. You can know more about them here)
- **reduce function**: Each node aggregates received messages by summation. So the aggregated messages on node i can be computed as:

  $$\tilde{h_i} = \sum_{j \in \mathcal{N}(i)} m_{j \to i}$$

  ,where $\mathcal{N}(i)$ refers to the set of neighbors of node i. (**Hint:** `dgl.function.sum('m', 'h')`)

10. Define a two-layer Graph Convolutional Network, with help on how to do it from the tutorial.
11. Now let's train this model to predict the club membership after the split. To train the model, we adopt Kipf's semi-supervised setting:
    - Only the instructor node (node 0) and the president node (node 33) are labeled.
    - The initial node feature is a one-hot encoding of the node id.
    - You can only use the labels of the two labelled nodes for training the model. (Calculating loss and back-propagating)
12. Since the final node embedding is a vector of length two (for predicting two classes), we can plot it as a point on a 2D plot and visualize how the final embeddings cluster towards each other. You can pick any 5 time steps in your training process where you can choose to plot the node embeddings.
13. Now, use the labels of all the nodes and report the accuracy of your model in predicting communities.
14. Repeat the whole process by defining a new message function and reduce function:
    - **new message function:**

      $$m_{j \to i} = \frac{1}{\sqrt{d_j}} h_j$$

    - **new reduce function:**

      $$\tilde{h_i} = \frac{1}{\sqrt{d_i}} \sum_{j \in \mathcal{N}(i)} m_{j \to i}$$

    This will help in incorporating normalization of the adjacency matrix as was shown in the matrix equation above.
    (**Hints:**
    - You can define your own `message_func` and `reduce_func` and use them as inputs to the `update_all` method in the forward pass inside the GCN module.
    - you can use `G.in_degrees(G.nodes())` to get a 1-D tensor containing the degrees of all the nodes.
    - you can try a user-defined message function to perform multiplication between source node feature and edge feature.)