Miguel Ángel Sánchez Cortés
Master in Data Science
Sapienza University of Rome

June 9, 2024

HOMEWORK 1 — Social Networks

1 Problem 1

Consider the following modification of the Barabási–Albert preferential attachment model [1] that we did in class: When a new node arrives at time t again it comes with l edges. However, this time each edge selects a node v with probability proportional to the degree d_v plus a constant c, that is, the probability equals:

$$\frac{d_v + c}{(t-1)(2l+c)},$$

where $c \geq -l$, as we describe at the end of Chapter 4 in the notes (so for $c = 0$ we have the Barabási-Albert model). Show that the degree distribution that we obtain as $t \to \infty$ is approximately a power law with exponent $3 + \frac{c}{l}$.

Solution

First, we can make a refresher of the process with which we can build this generalized Barabási-Albert model. Let $t = 1, 2, 3, \dots$ represent discrete time steps, we will build a random network with the property that at time t the network has t nodes and tl edges using the following procedure:

1. We start initially with a graph of 2 nodes, v_1 and v_2, and $2l$ edges between them.
2. At each time step $t \geq 3$:
   (a) A new node v_t is added to the network.
   (b) l new edges are added. For each edge, one endpoint is v_t and the other one is selected with probability proportional to the degree at time $t-1$. In particular, node u is selected with probability:
   $$\frac{d_u + c}{\sum_{w \in V_{t-1}} (d_w + c)} = \frac{d_u + c}{(t-1)(2l+c)}, \tag{1-1}$$
   where $c \geq -l$ and V_t is the set of nodes at time t.

We will follow the same heuristic argument as in class, although there are more rigorous methods to show that the degree distribution that we obtain as $t \to \infty$ follows a power law with exponent $3 + cl$. To do this we first define:

- $n_k(t)$: the mean number of nodes at time t with degree k.
- $p_k(t) = \frac{n_k(t)}{t}$: the ratio of nodes at time t with degree k.

Our objective will be to write a recursive equation for the values of $p_k(t)$. We can notice that when a new node is created at step $t+1$, we will have l new edges, where the probability that a node u of degree k increases to a degree $k+1$ will be given by[1]:

$$l \frac{d_u + c}{\sum_{w \in V_{t-1}} (d_w + c)} = l \left[\frac{d_u + c}{t(2l+c)} \right] = \frac{l(k+c)}{t(2l+c)},$$

since there will be l cases where this can happen with probability of happening given by equation 1-1 and $d_u = k$.

[1]Here we are making the big assumption that no edge will end on the same node, when there is a $\frac{1+c}{t(2l+c)}$ probability of this happening. Nevertheless, we are interested in the limit $t \to \infty$ and on this limit we can see that such probability tends to 0.

Using this fact, we can therefore write a recursive equation for the values of $n_k(t)$ noticing that we have three different cases:

- When $k < l$: In this case, we can notice that $n_k(t+1)$, the average number of nodes at time $t+1$ with degree k is equal to zero since by construction every node has at least degree l.
- When $k > l$: In this case, we can notice that at time $t+1$, since l new edges arrive to the network, we have that the average number of nodes that will increase their degree from $k-1$ to k is given by:

  $$n_{k-1}(t)\left[\frac{l(k-1+c)}{t(2l+c)}\right].$$

  At the same time, the average number of nodes that will increase their degree from k to $k+1$ is given by:

  $$n_k(t)\left[\frac{l(k+c)}{t(2l+c)}\right].$$

  Therefore, we can write an expression for $n_k(t+1)$, the average number of nodes at time $t+1$ with degree k as:

  $$n_k(t+1) = n_k(t) + n_{k-1}(t)\left[\frac{l(k-1+c)}{t(2l+c)}\right] - n_k(t)\left[\frac{l(k+c)}{t(2l+c)}\right]$$

- When $k = l$: In this case, we can notice that at time $t+1$, the number of nodes of degree l increases by 1 since we are adding just one node to the network, and at the same time, the average number of nodes that will increase their degree from l to $l+1$ is given by:

  $$n_l(t)\left[\frac{l(l+c)}{t(2l+c)}\right].$$

  Therefore we have:

  $$n_l(t+1) = n_l(t) + 1 - n_l(t)\left[\frac{l(l+c)}{t(2l+c)}\right] \tag{1-2}$$

From equation 1-2 it is straightforward to find an expression for $p_l(t) = \frac{n_l(t)}{t}$ assuming that in the limit $t \to \infty$, $p_k(t)$ converges to a value that is independent of t. In this case, we have that $p_k(t+1) = p_k(t) = p_k$, and therefore we have:

$$\begin{aligned}
n_l(t+1) = n_l(t) + 1 - n_l(t)\left[\frac{l(l+c)}{t(2l+c)}\right] &\Longleftrightarrow (t+1)p_l(t+1) = tp_l(t) + 1 - p_l(t)\left[\frac{l(l+c)}{(2l+c)}\right] \\
&\Longleftrightarrow (t+1)p_l = tp_l + 1 - p_l\left[\frac{l(l+c)}{(2l+c)}\right] \\
&\Longleftrightarrow \left[\frac{l(l+c)}{(2l+c)} + (t+1) - t\right]p_l = 1 \\
&\Longleftrightarrow p_l = \left[\frac{2l+c}{l(l+2+c)+c}\right] \Longleftrightarrow \boxed{p_l = \left[\frac{2+c/l}{l+2+c+c/l}\right]}
\end{aligned} \tag{1-3}$$

From equation 1, in the same way, we can obtain a recursive expression for $p_k(t)$ when $k > l$ as follows:

$$\begin{aligned}
&n_k(t+1) = n_k(t) + n_{k-1}(t)\left[\frac{l(k-1+c)}{t(2l+c)}\right] - n_k(t)\left[\frac{l(k+c)}{t(2l+c)}\right] \\
\Longleftrightarrow &(t+1)p_k(t+1) = tp_k(t) + p_{k-1}(t)\left[\frac{l(k-1+c)}{(2l+c)}\right] - p_k(t)\left[\frac{l(k+c)}{(2l+c)}\right] \\
\Longleftrightarrow &(t+1)p_k = tp_k + p_{k-1}\left[\frac{l(k-1+c)}{(2l+c)}\right] - p_k\left[\frac{l(k+c)}{(2l+c)}\right]
\end{aligned}$$

$$\Longleftrightarrow \left[\frac{l(k+c)}{(2l+c)} + (t+1) - t\right] p_k = \left[\frac{l(k-1+c)}{(2l+c)}\right] p_{k-1}$$
$$\Longleftrightarrow p_k = \left[\frac{l(k-1+c)}{(2l+c)}\right] \left[\frac{l(k+c)+(2l+c)}{(2l+c)}\right]^{-1} p_{k-1}$$
$$\Longleftrightarrow p_k = \left[\frac{l(k-1+c)}{l(k+2+c)+c}\right] p_{k-1} \Longleftrightarrow \boxed{p_k = \left[\frac{k-1+c}{k+2+c+c/l}\right] p_{k-1}} \tag{1-4}$$

Now, we can notice that equation 1-4 is a recursive relation, and we can analyze its general form by performing recursive multiplications. For $k = l+1$ we have:

$$p_{l+1} = \left[\frac{l+1-1+c}{l+1+2+c+c/l}\right] p_l = \left[\frac{(l+c)}{(l+c+c/l)+3}\right] \left[\frac{2+c/l}{(l+c+c/l)+2}\right].$$

In the same way, for $k = l+2$ we have:

$$p_{l+2} = \left[\frac{l+2-1+c}{l+2+2+c+c/l}\right] p_{l+1} = \left[\frac{(l+c)+1}{(l+c+c/l)+4} \cdot \frac{(l+c)}{(l+c+c/l)+3}\right] \left[\frac{2+c/l}{(l+c+c/l)+2}\right].$$

And for $k = l+3$ we have:

$$p_{l+3} = \left[\frac{l+3-1+c}{l+3+2+c+c/l}\right] p_{l+2} = \left[\frac{(l+c)+2}{(l+c+c/l)+5} \cdot \frac{(l+c)+1}{(l+c+c/l)+4} \cdot \frac{(l+c)}{(l+c+c/l)+3}\right] \left[\frac{2+c/l}{(l+c+c/l)+2}\right].$$

In general we can see that the form of the equation is:

$$p_k = \frac{(k+c)-1}{(k+c+c/l)+2} \cdot \frac{(k+c)-2}{(k+c+c/l)+1} \cdots \frac{(l+c)}{(l+c+c/l)+3} \cdot \frac{(2+c/l)}{(l+c+c/l)+2}$$

In order to simplify the expression above we can notice a pattern that reminds a fundamental property of the Gamma function [2], that is:

$$\Gamma(x+n) = (x+n-1)(x+n-2)\cdots x\Gamma(x) \Longleftrightarrow \frac{\Gamma(x+n)}{\Gamma(x)} = (x+n-1)(x+n-2)\cdots x, \tag{1-5}$$

for $n \in \mathbb{N}$. Therefore, using the property in 1-5 we can re-write the expression for p_k in the following way:

$$p_k = \frac{\Gamma(k+c)}{\Gamma(c)} \cdot \frac{\Gamma(c+c/l+1)}{\Gamma(k+c+c/l+3)} = \left[\frac{\Gamma(c+c/l+1)}{\Gamma(c)}\right] \frac{\Gamma(k+c)}{\Gamma(k+c+c/l+3)}$$

In the equation above we can see that the term in brackets is constant with respect to the degree k so we can ignore it for our analysis. At the same time, we can make use of another asymptotic property of the Gamma function, since for $x \to \infty$, we have that $\Gamma(x+\alpha) \sim \Gamma(x)x^{\alpha}$. Finally, using this we obtain:

$$p_k \sim \frac{\Gamma(k)k^c}{\Gamma(k)k^{c+c/l+3}} = \frac{k^c}{k^{c+c/l+3}} = k^{-(c/l+3)} \Longrightarrow \boxed{p_k \sim k^{-\gamma} \text{ with } \gamma = 3 + \frac{c}{l}}$$

Thus concluding the proof that the degree distribution of this generalized Barabási-Albert model is approximately a power law with exponent $\gamma = 3 + \frac{c}{l}$ as $t \to \infty$.

[2]This idea was taken from the book *Networks: An Introduction*, by Mark Newman [2].

2 Problem 2

We are given the following graph, $G = (V, E)$:

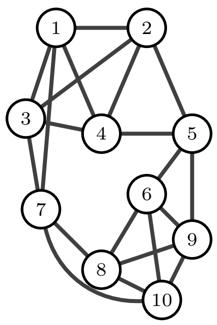


Figure 1: Undirected graph $G = (V, E)$.

1. Find the densest subgraph using the greedy algorithm we saw in class.

Solution

Let's remember that in class, given an undirected graph $G = (V, E)$ and a set of nodes $S \subseteq V$, we defined the concept of *sparsity* of the set S as:

$$f(S) = \frac{|E(S)|}{|S|}, \tag{2-1}$$

where $E(S) = \{(u, v) \in E : u \in S, v \in S\}$ is the set of edges of G with both endpoints in S. At the same time, we also defined $\deg_S(v) = |\{(v, u) \in E : u \in S\}|$ as the degree of node v restricted to the nodes in S.

We also noticed that these concepts are related, since $f(S)$ is half of the average degree among the nodes in S:

$$\frac{\sum_{v \in S} \deg_S(v)}{|S|} = \frac{2|E(S)|}{|S|}. \tag{2-2}$$

Finally, from this, we defined the problem of finding the *densest subgraph* as the problem of finding a set $S \subseteq V$ that maximizes $f(S)$. This problem can be approximately solved using the greedy algorithm 2 presented below:

Algorithm 1: *GreedyDensestSubgraph*(G)

```
Input  : $G = (V, E)$: Simple undirected graph
Output: A set of nodes $S \subseteq V$
1  $S \leftarrow V$
2  $S^G \leftarrow V$
3  while $|S| > 1$ do
4      $v = \arg\min_{v \in S} \deg_S(v)$
5      $S \leftarrow S \setminus \{v\}$
6      if $f(S) \geq f(S^G)$ then
7          $S^G \leftarrow S$
8      end
9  end
10 return $S^G$
```

This solution can be summarized as follows: We start with $S = V$ and we continue by removing the node with minimum degree from the graph induced by S until S remains with one node. At the end, we return the set S that during execution had the highest density. We can apply this algorithm to the graph in Figure 1 as follows:

1. Initially $S = V$, we also can observe that G_0 is a 4-regular graph and therefore we can remove any of the nodes since they all have the same degree $\deg_S(v) = 4$. We also have an initial value of sparsity $f(S) = \frac{|E(S)|}{|S|} = \frac{20}{10} = 2$
2. Let's assume we remove node 1, therefore our new induced graph by $S = V \setminus \{1\}$ is given by:

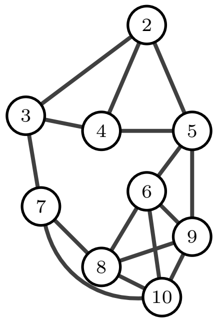


Figure 2: Undirected graph $G_1 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{16}{9} \approx 1.77$. We can also choose between nodes $\{2, 3, 4, 7\}$ to remove since they all have $\deg_S(v) = 3$ thanks to the previous removal of node 1.

3. Let's suppose we remove node 2, therefore our new induced graph by $S = V \setminus \{1, 2\}$ is given by:

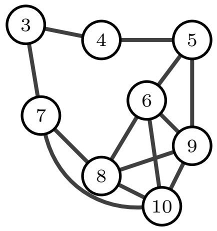


Figure 3: Undirected graph $G_2 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{13}{8} \approx 1.62$. We can also choose between nodes $\{3, 4\}$ to remove since they all have $\deg_S(v) = 2$ thanks to the previous removal of nodes $\{1, 2\}$.

4. Let's suppose we remove node 3, therefore our new induced graph by $S = V \setminus \{1, 2, 3\}$ is given by:

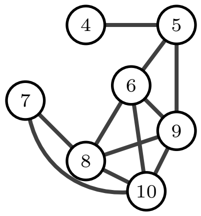


Figure 4: Undirected graph $G_3 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{11}{7} \approx 1.57$. Now, can choose node 4 to remove since it has the minimum $\deg_S(v) = 1$ thanks to the previous removal of nodes $\{1, 2, 3\}$.

5. Therefore our new induced graph by $S = V \setminus \{1, 2, 3, 4\}$ is given by:

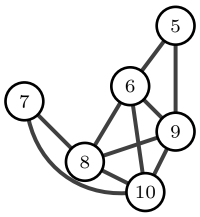


Figure 5: Undirected graph $G_4 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{10}{6} \approx 1.66$. We can also choose between nodes $\{5, 7\}$ to remove since they all have $\deg_S(v) = 2$ thanks to the previous removal of nodes $\{1, 2, 3, 4\}$.

6. Let's suppose we remove node 5, therefore our new induced graph by $S = V \setminus \{1, 2, 3, 4, 5\}$ is given by:

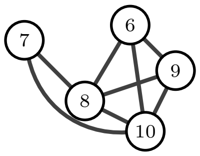


Figure 6: Undirected graph $G_5 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{8}{5} \approx 1.6$. Now, can choose node 7 to remove since it has the minimum $\deg_S(v) = 2$ thanks to the previous removal of nodes $\{1, 2, 3, 4, 5\}$.

7. Therefore our new induced graph by $S = V \setminus \{1, 2, 3, 4, 5, 7\}$ is given by:

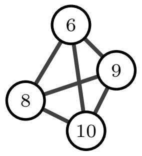


Figure 7: Undirected graph $G_6 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{6}{4} \approx 1.5$. We can also choose between nodes $\{6, 8, 9, 10\}$ to remove since they all have $\deg_S(v) = 3$ thanks to the previous removal of nodes $\{1, 2, 3, 4, 5, 7\}$.

8. Let's suppose we remove node 6, therefore our new induced graph by $S = V \setminus \{1, 2, 3, 4, 5, 6, 7\}$ is given by:

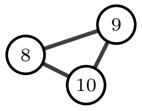


Figure 8: Undirected graph $G_7 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{3}{3} = 1$. We can also choose between nodes $\{8, 9, 10\}$ to remove since they all have $\deg_S(v) = 2$ thanks to the previous removal of nodes $\{1, 2, 3, 4, 5, 6, 7\}$.

9. Finally, let's suppose we remove node 8, therefore our new induced graph by $S = V \setminus \{1, 2, 3, 4, 5, 6, 7, 8\}$ is given by:

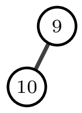


Figure 9: Undirected graph $G_8 = (S, E(S))$.

Now, we obtain a new value of sparsity given by $f(S) = \frac{|E(S)|}{|S|} = \frac{1}{2} = 0.5$. And the greedy algorithm comes to an end.

As we can see, the result of this algorithm is that the *densest subgraph* is the graph itself, since it is the one with maximum sparsity $\boxed{f(S) = 2}$.

2. Find a minimum cut.

Solution

Let's remember that by definition, a graph cut (C_1, C_2), is a *partition* of the nodes in two sets C_1, C_2, where: $C_1 \cap C_2 = \emptyset$ and $C_1 \cup C_2 = V$. At the same time, we define the size of the cut (C_1, C_2) to be the number of edges between C_1 and C_2:

$$|E(C_1, C_2)| = |E \cap (C_1 \times C_2)| = |\{(i, j) \in E : i \in C_1, j \in C_2\}| \tag{2-3}$$

Therefore, we can naturally define the problem of finding a *minimum* cut, as the problem of finding a cut (C_1, C_2) that minimizes $|E(C_1, C_2)|$. We can find a minimum cut using the randomized Karger's algorithm for connected graphs 5.

Algorithm 2: *KargersAlgorithm*(G)

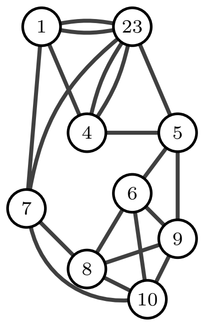
```
Input  : $G = (V, E)$: Simple undirected graph
Output: A cut $(C_1, C_2)$ from $G$
1 while $|V| > 2$ do
2     choose $e \in E$ uniformly at random
3     $G \leftarrow G/e$
4 end
5 return $(C_1, C_2)$
```

This algorithm can be summarized as follows: While the number of nodes in the graph are greater than 2, we continue by selecting an edge from E uniformly at random and *contracting it.* The result of contracting the edge $e = (u, v)$ is a new node uv, where every edge (w, u) or (w, v) for $w \notin \{u, v\}$ to the endpoints of the contracted edge is replaced by an edge (w, uv). This operation is represented by G/e.

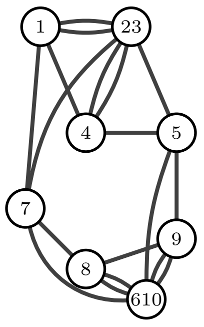
The algorithm ends when all nodes are contracted except two and the number of edges connecting both nodes are the number of edges of the minimum cut found by the algorithm. In the end, the nodes that were contracted to each resulting node form a cut (C_1, C_2) and this cut is the minimum cut with probability:

$$\mathbf{P} = \binom{n}{2}^{-1}, \tag{2-4}$$

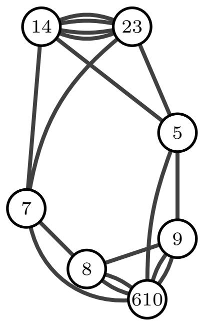
where $n = |V|$. In Figure 10 we present a successful result of a run of Karger's algorithm for the graph presented in Figure 1. As we can observe, we obtained a minimum cut $C_1 = \{1, 2, 3, 4, 5\}$ and $C_2 = \{6, 7, 8, 9, 10\}$, with $\boxed{|E(C_1, C_2)| = 4}$.



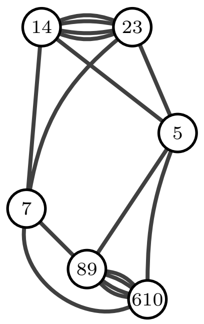
(a) Iteration 1: Contraction of nodes 2 and 3.



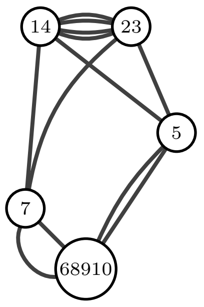
(b) Iteration 2: Contraction of nodes 6 and 10.



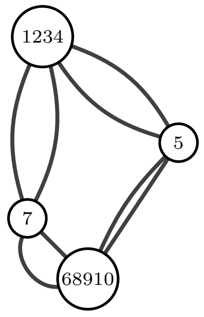
(c) Iteration 3: Contraction of nodes 1 and 4.



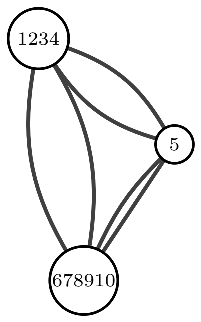
(d) Iteration 4: Contraction of nodes 8 and 9.



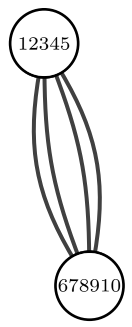
(e) Iteration 5: Contraction of nodes 89 and 610.



(f) Iteration 6: Contraction of nodes 14 and 23.



(g) Iteration 7: Contraction of nodes 7 and 68910.



(h) Iteration 8: Contraction of nodes 5 and 1234.

Figure 10: An example of a run of Karger's Algorithm. As we can observe, in the end we obtain a (minimum) cut of the graph, in this case $C_1 = \{1, 2, 3, 4, 5\}$ and $C_2 = \{6, 7, 8, 9, 10\}$, with $|E(C_1, C_2)| = 4$.

3. Demonstrate (by calculating λ_2, $\phi(G)$. etc.) that Cheeger's inequalities hold for this graph.

<u>**Solution**</u>

To calculate Cheeger's inequalities, we first have to remember an auxiliary concept we studied in class in order to make calculations easier. First, let's remember that the *conductance* of a d-regular graph G is given by:

$$\phi(G) = \min_{C_1 \subset V} \phi(C_1) = \min_{\substack{C_1 \subset V \\ C_2 = V \setminus C_1}} \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}}, \tag{2-5}$$

this is, the minimum conductance among all its partitions. As we can notice from Equation 2-5, the problem of minimizing the graph conductance is equivalent to the one of minimizing the sparsest cut.

Using this concept we therefore claimed the following result, known as Cheeger's inequalities:

$$\frac{\lambda_2}{2} \leq \phi(G) \leq \sqrt{2\lambda_2}, \tag{2-6}$$

where λ_2 is the second smallest eigenvalue of the normalized Laplacian matrix $\mathcal{L}$ given by:

$$\mathcal{L} = \mathbb{I} - \frac{1}{d}A, \tag{2-7}$$

where A is the adjacency matrix of a d-regular graph.

For the problem of the graph in Figure 1, we can obtain the value of λ_2 (and its corresponding eigenvector) using equation 2-7 and *Python* as shown on Appendix A. The resulting values for the eigenvalue problem are:

$$\lambda_2 \approx 0.25, \quad \vec{v_2} \approx \begin{pmatrix} -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ 1.88 \times 10^{-16} \\ 1.70 \times 10^{-16} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \end{pmatrix}. \tag{2-8}$$

For the graph conductance $\phi(G)$, we can notice from Figure 10 that for all the cuts where $|E(C_1, C_2)| = 4$, i.e. minimum cuts, the one that optimizes the quantity given by equation 2-5 is given by $C_1 = \{1, 2, 3, 4, 5\}$ and $C_2 = \{6, 7, 8, 9, 10\}$. This follows since, if we contract nodes without mantaining the same values for $|C_1|$ and $|C_2|$, the quantity $\min\{|C_1|, |C_2|\}$ will always be less than for the balanced case ($|C_1| = |C_2|$) and therefore, the conductance will be greater.

Therefore, we can obtain the conductance $\phi(G)$ from this cut and see that Cheeger's inequalities hold for graph G:

$$\phi(G) = \frac{4}{4 \cdot 5} = \frac{1}{5} = 0.2 \Longrightarrow \boxed{0.125 \approx \frac{\lambda_2}{2} \leq \phi(G) \leq \sqrt{2\lambda_2} \approx 0.71} \tag{2-9}$$

4. Find the cut that satisfies Part 3 (and show that it does).

<u>Solution</u>

The cut that satisfies Part 3 is known as the *sparsest cut* of the graph i.e. the cut that minimizes the conductance:

$$\phi(G) = \min_{C_1 \subset V} \phi(C_1) = \min_{\substack{C_1 \subset V \\ C_2 = V \setminus C_1}} \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}}, \tag{2-10}$$

In class we saw the Sweeping algorithm used to obtain this cut based on the proof of Cheeger's inequalities (see Algorithm 3). We have already performed the first step of this algorithm since we obtained λ_2 and its corresponding eigenvector given by equation 2-8.

```
Algorithm 3: Sweeping(G)
  Input  : $G = (V, E)$: Simple undirected $d$-regular graph
  Output: A cut $(C_1, C_2)$ from $G$
1 Compute the second smallest eigenvalue $\lambda_2$ of $\mathcal{L}$ and the corresponding eigenvector $\vec{x} = \vec{v_2}$.
2 Sort the vertices of $V$ in increasing order of $x_i$. Assume that $x_1 \leq x_2 \leq \cdots \leq x_n$.
3 For each $i \in [n-1]$ consider the cut $(C_1, C_2)$ with $C_1 = [i]$ and $C_2 = V \setminus C_1$. Compute the value:
      $\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}}$
4 From all the $n-1$ cuts computed in the last step, return the one that minimizes $\phi(C_1)$.
```

To perform the second step, we can sort the nodes of G in increasing order of the values of v_{2_i}. As we can see, these entries are already sorted in increasing order and therefore the nodes are sorted from 1 to 10:

$$\vec{v_2} = \begin{array}{c} \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \\ \mathbf{6} \\ \mathbf{7} \\ \mathbf{8} \\ \mathbf{9} \\ \mathbf{10} \end{array} \begin{pmatrix} -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ 1.88 \times 10^{-16} \\ 1.70 \times 10^{-16} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \end{pmatrix} \implies \vec{v_2}_{\text{sorted}} = \begin{array}{c} \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \\ \mathbf{6} \\ \mathbf{7} \\ \mathbf{8} \\ \mathbf{9} \\ \mathbf{10} \end{array} \begin{pmatrix} -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ -3.53 \times 10^{-1} \\ 1.88 \times 10^{-16} \\ 1.70 \times 10^{-16} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \\ 3.53 \times 10^{-1} \end{pmatrix} \tag{2-11}$$

For the third step, we can iteratively sweep through all the cuts in order of the entries of the eigenvector of the second smallest eigenvalue:

1. Now, if we consider the cut $C_1 = \{1\}$ and $C_2 = V \setminus \{1\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{4}{4 \cdot \min\{1, 9\}} = 1$$

2. In the same way, if we consider the cut $C_1 = \{1, 2\}$ and $C_2 = V \setminus \{1, 2\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{6}{4 \cdot \min\{2, 8\}} = \frac{3}{4}$$

3. Now, if we consider the cut $C_1 = \{1, 2, 3\}$ and $C_2 = V \setminus \{1, 2, 3\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{6}{4 \cdot \min\{3, 7\}} = \frac{1}{2}$$

4. In the same way, if we consider the cut $C_1 = \{1, 2, 3, 4\}$ and $C_2 = V \setminus \{1, 2, 3, 4\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{4}{4 \cdot \min\{4, 6\}} = \frac{1}{4}$$

5. Now, if we consider the cut $C_1 = \{1,2,3,4,5\}$ and $C_2 = V \setminus \{1,2,3,4,5\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{4}{4 \cdot \min\{5,5\}} = \frac{1}{5}$$

6. In the same way, if we consider the cut $C_1 = \{1,2,3,4,5,6\}$ and $C_2 = V \setminus \{1,2,3,4,5,6\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{6}{4 \cdot \min\{4,6\}} = \frac{3}{8}$$

7. Now, if we consider the cut $C_1 = \{1,2,3,4,5,6,7\}$ and $C_2 = V \setminus \{1,2,3,4,5,6,7\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{6}{4 \cdot \min\{3,7\}} = \frac{1}{2}$$

8. In the same way, if we consider the cut $C_1 = \{1,2,3,4,5,6,7,8\}$ and $C_2 = V \setminus \{1,2,3,4,5,6,7,8\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{6}{4 \cdot \min\{2,8\}} = \frac{3}{4}$$

9. Finally, if we consider the cut $C_1 = \{1,2,3,4,5,6,7,8,9\}$ and $C_2 = V \setminus \{1,2,3,4,5,6,7,8,9\}$ of the graph on Figure 1, we obtain a conductance value of:

$$\phi(C_1) = \frac{|E(C_1, C_2)|}{d \cdot \min\{|C_1|, |C_2|\}} = \frac{4}{4 \cdot \min\{1,9\}} = 1$$

In the end of this algorithm, we can see that the cut that minimized the conductance $\phi(C_1)$ value is:

$$\boxed{C_1 = \{1,2,3,4,5\} \text{ and } C_2 = \{6,7,8,9,10\}, \text{ with } \phi(C_1) = \frac{1}{5}}. \tag{2-12}$$

As we can see, in this case, the sparsest cut coincides with the minimum cut and as we've seen in equation 2-9, this cut satisfies Cheeger's inequalities.

3 Problem 3

As Aris Gionis mentioned in class, an interesting phenomenon in social networks is that a random person's expected degree is smaller than the degree of her peers: "Your friends are more popular than you are!" Given an undirected graph $G = (V, E)$, select a random node and let X be the random variable that equals to its degree and Y to be the random variable that equals the average degree of the node's neighbors.

1. Prove that $\mathbb{E}[X] \leq \mathbb{E}[Y]$.

<u>Proof</u>

Let $G = (V, E)$ be an undirected graph and let X be the random variable that represents the degree of a randomly selected node of this graph. We first can notice that. by definition, the expected degree of a randomly selected node in G is given by the average degree of the nodes in the graph:

$$\mathbb{E}[X] = \frac{1}{n}\sum_{v\in V}\deg(v) = \frac{1}{n}\cdot 2m = \frac{2m}{n}, \tag{3-1}$$

where $n = |V|$, $m = |E|$, and the second equality is given by the fact that $\sum_v \deg(v) = 2|E|$ for undirected graphs. At the same time, let Y represent the random variable that equals the average degree of the node's neighbors. Y can be modeled by randomly selecting a node and then computing the average degree among all the neighboring nodes. We also can notice that this process is equivalent to the process of randomly selecting an edge from E, choosing one of the nodes connected to that edge and obtaining the degree of the selected node.

Since the probability that a random edge is connected to a node is proportional to its degree, we have that the probability that a random edge is connected to a node u with degree $\deg(u)$ is equal to:

$$\frac{\deg(u)}{2m} = \frac{\deg(u)}{\sum_{v\in V}\deg(v)}, \tag{3-2}$$

where $m = |E|$, and the second equality is given by the fact that $\sum_v \deg(v) = 2|E|$ for undirected graphs. Therefore, the average degree of a node's neighbor is simply given by computing the expectation over all the nodes:

$$\mathbb{E}[Y] = \sum_{u\in V}\deg(u)\cdot\frac{\deg(u)}{\sum_{v\in V}\deg(v)} = \frac{\sum_{u\in V}\deg(u)^2}{\sum_{v\in V}\deg(v)} = \frac{\mathbb{E}[X^2]}{\mathbb{E}[X]} \tag{3-3}$$

Now, if we denote the mean and variance of the degree distribution of G as: $\mu = \mathbb{E}[X]$ and $\sigma^2 = \mathbb{E}[X^2] - \mathbb{E}[X]^2$, we can rewrite the expression on equation 3-3 as:

$$\mathbb{E}[Y] = \frac{\mathbb{E}[X^2]}{\mathbb{E}[X]} = \frac{\sigma^2 + \mathbb{E}[X]^2}{\mathbb{E}[X]} = \frac{\sigma^2 + \mu^2}{\mu} = \frac{\sigma^2}{\mu} + \mu \tag{3-4}$$

From probability theory we know that $\sigma^2 \geq 0$ since the variance of a random variable is always non-negative. At the same time, we can notice that X is a non-negative random variable (there's never a negative degree) and therefore its expected value is also non-negative $\mu \geq 0$. This means, from elementary algebra that $\frac{\sigma^2}{\mu} \geq 0$, and therefore:

$$\frac{\sigma^2}{\mu} + \mu \geq \mu \iff \boxed{\mathbb{E}[Y] \geq \mathbb{E}[X]}, \tag{3-5}$$

by equations 3-4 and 3-1, and from the fact that the sum of a positive number b with another positive number a is always greater than the number b alone. This concludes the proof.

2. When do we have that $\mathbb{E}[X] = \mathbb{E}[Y]$?

<u>Solution</u>

From equation 3-4, assuming that $\mu > 0$, we can see that:

$$\mathbb{E}[Y] = \mathbb{E}[X] \iff \frac{\sigma^2}{\mu} + \mu = \mu \iff \frac{\sigma^2}{\mu} = 0 \iff \sigma^2 = 0, \tag{3-6}$$

this is, whenever the variance of the degree distribution of G is equal to zero. This can only happen if the degrees of all the nodes in V have the same value, i.e,. if $\boxed{G \text{ is a } d\text{-regular graph}}$.

4 Problem 4

We are monitoring a graph, which arrives as a stream of edges $\mathcal{E} = e_1, e_2, \ldots$. We assume that exactly one edge arrives at a time, with edge e_i arriving at time i, and the stream is starting at time 1. Each edge e_i is a pair of vertices (u_i, v_i), and we use V to denote the set of all vertices that we have seen so far.

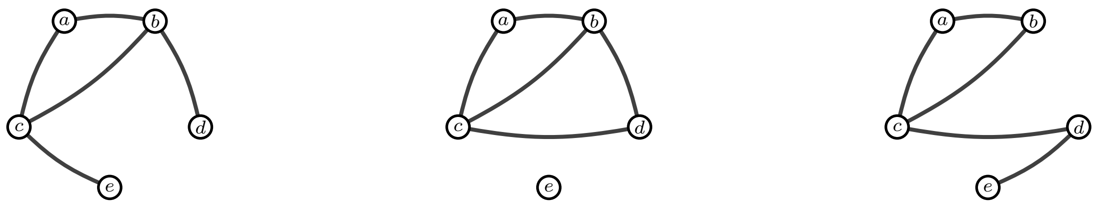
We assume that we are working in the sliding window model. According to that model, at each time t only the w most recent edges are considered active. Thus, the set of active edges $E(t, w)$ at time t and for window length w is:

$$E(t,w) = \begin{cases} e_{t-w+1}, \ldots, e_t, & \text{if } t > w, \\ e_1, \ldots, e_t, & \text{if } t \leq w, \end{cases}$$

We then write $G(t, w) = (V, E(t, w))$ to denote the graph that consists of the active edges at time t, given a window length w. As an example, given the stream of edges:

$$e_1 = (c, e), e_2 = (b, d), e_3 = (a, c), e_4 = (c, b), e_5 = (a, b), e_6 = (c, d), e_7 = (d, e)$$

the graphs $G(5, 5)$, $G(6, 5)$, and $G(7, 5)$ are shown below:



(a) $G(5, 5)$ (b) $G(6, 5)$ (c) $G(7, 5)$

Figure 11: Example of $G(t, w)$ with $w = 5$.

Notice that for all $t \geq w$ the graph $G(t+1, w)$ results from $G(t, w)$ by adding one edge and deleting one edge.

We want to monitor the connectivity of the graph $G(t, w)$. In other words, we want to design an algorithm that quickly decides, at any time t, if the graph $G(t, w)$ is connected. In the previous example, the graphs $G(5, 5)$ and $G(7, 5)$ are connected, while the graph $G(6, 5)$ is not connected.

1. Propose a streaming algorithm for deciding the connectivity of $G(t, w)$.

Solution

Notice that we have essentially two cases for the algorithm:

1. If $t < w$: $E(t + 1, w)$ will be simply $E(t, w)$ with an added edge $e_t + 1$ since the window doesn't have a role for this case. Therefore, checking for connectivity for $E(t+1, w)$ could be made simply by checking how $E(t, w)$'s connectivity changes when adding an edge e_{t+1}.
2. If $t \geq w$: $E(t + 1, w)$ will be simply $E(t, w)$ with an added edge $e_t + 1$ and a deleted edge e_{t+1-w}, since we only take into account the edges within the window w. Therefore, checking for connectivity for $E(t + 1, w)$ could be made by checking how $E(t, w)$'s connectivity changes when adding an edge e_{t+1} and deleting an edge e_{t+1-w}.

The first case can be efficiently solved by using a Disjoint-Set Data Structure [3] to mantain the connected components of the graph $G(t, w)$. A Disjoint-Set Data Structure in this context is a data structure that efficiently allows to add edges and check if two nodes belong to the same connected component. In general, the Disjoint-Set Data Structure supports three main operations:

- *MAKE SET*: An operation that creates new disjoint sets where each node belongs to its own set. This operation has time complexity $O(n)$.
- *FIND*: Finds the set where a node belongs to. This operation has also a time complexity $O(n)$.
- *UNION*: An operation that adds an edge between nodes and merges their connected components if they do not belong to the same one. This operation has also a time complexity $O(n)$.

Therefore updating this structure for $G(t + 1, w)$ can be done by simply using the arriving edge $e_{t+1} = (a, b)$ to merge their components if they belong to different connected components. Checking for connectivity would mean to check if there is only one connected component, this would have a total time complexity of $O(n)$ as well, assuming we have the previous number of connected components of $G(t + 1, w)$, since then it is enough to use *FIND* to check if nodes a and b previously didn't belonged to the same connected component, and if after updating they do, then update the number of connected components: $n_{\text{conncomp}} \leftarrow n_{\text{conncomp}} - 1$.

The second case is a little more tricky, since now we have to also account for an edge deletion. The simplest approach after deleting an edge $e_{t+1-w} = (c, d)$ (although maybe not the best one) would be to perform a BFS or DFS to check if c and d still belong to the same connected component. Then, split this case into two subcases:

1. If c and d indeed belong to the same component, then $n_{\text{conncomp}} \leftarrow n_{\text{conncomp}}$ and we have finished.
2. If c and d don't belong to the same component anymore, then, we could simply rebuild our Disjoint Set Data Structure from scratch and update $n_{\text{conncomp}} \leftarrow n_{\text{conncomp}} + 1$

Assuming that we have the set of edges $E(t, w)$ of a graph at time t and its number of connected components (this can be easily obtained by initializing the data structure at time 0). Therefore, for $t + 1$ our algorithm would be given by the pseudocode in 4.

2. How much space does your algorithm use?

Solution

Notice that for algorithm 4 to work at each time step t, we need to mantain the set of edges $E(t, w)$ of the graph $G(t, w)$, the Disjoint Set Data Structure of the same graph, and the number of connected components of the graph. In the worst case scenario, we have the following:

- We will have to mantain an edge set with w edges at every time t, therefore the space that will be required to mantain the edge set will be $O(w)$.
- We always will have to mantain a Disjoint Set Data Structure that contains the connected component data of each node. The cost to mantain the information of all the n nodes of the graph will be $O(w)$.
- The cost of mantaining n_{conncomp} the number of connected components at each step along with any temporal variables that will be created along the way, will always require space of $O(c)$ where c is a constant.

Therefore, the overall space used by this algorithm given in big-O notation is equivalent to the sum of the cases we've mentioned before, so:

$$S(w, n) = O(w + n), \tag{4-1}$$

where w is the size of the time window and n is the number of nodes of the temporal graph.

[3] See here for more information

```
Algorithm 4: $StreamingAlgorithm(t+1, w)$
  Input  : Edges $e_1 = (a, b)$ and $e_2 = (c, d)$ to be added and deleted from network $G(t, w)$ respectively
           at time $t + 1$.
  Output: Connectivity status of the network $G(t + 1, w)$.
1  $E(t+1, w) \leftarrow E(t, w) \cup \{e_1\} \setminus \{e_2\}$
2  sameComponent ← $FIND$(a) == $FIND$(b)
3  $UNION$(a,b)
4  sameComponentUpdated ← $FIND$(a) == $FIND$(b)
5  if $sameComponent \neq sameComponentUpdated$ then
6      $n_{\text{conncomp}} \leftarrow n_{\text{conncomp}} - 1$
7  end
8  if $t \geq w$ then
9      areConnected → $BFS$(E(t+1,w),c,d)
10     if $areConnected == FALSE$ then
11         $n_{\text{conncomp}} \leftarrow n_{\text{conncomp}} - 1$
12         $MAKESET$(n)
13         for $edge \in E(t+1, w)$ do
14             $UNION$(edge)
15         end
16     end
17 end
18 If $n_{\text{conncomp}} = 1$ return $TRUE$, otherwise return $FALSE$.
```

3. What is the update time of your algorithm?

Solution

The update time of our algorithm is the time needed to compute the output at time $t+1$, given the state of the input mentioned before at time t. Let's analyze this line-by-line in the worst case scenario (this is when $t \geq w$ and the graph is made by one component).

- Assuming we are mantaining the set of edges $E(t, w)$ of the temporal graph in an array. Line 1 of the code in algorithm 4 would have time complexity $O(w)$ since in the worst case scenario adding or deleting an item would result in traversing the whole array (of length w).
- Performing a *UNION* operation on a Disjoint Set has time complexity $O(n)$ therefore line 3 has the same time complexity.
- Running lines 2, 4, 5, 6, and 7 has a constant time complexity since they are simple operations.
- In the worst case scenario, performing *BFS* has an $O(n + w)$ time complexity and this dominates over lines 10-15 in the worst case scenario, that is when the graph is connected. Therefore lines 8-17 have an overall time complexity of $O(n + w)$.

Checking the number of connected components in line 18 has a constant time complexity since we are only checking the equality of a variable. Therefore, we can see that what dominates the whole time complexity is the BFS algorithm and our overall time complexity is:

$$T(w, n) = O(w + n). \tag{4-2}$$

Notice that in the worst case scenario, this algorithm is not better than simply running BFS at every time step t to check for connectivity in the graph, since this would have space $O(n)$ (we need to mantain only an adjacency matrix) and time complexity $O(n + w)$. Nevertheless, in other scenarios, our algorithm in 4 would be practically quicker, specially for the case $t < w$ where we don't have to delete edges since this would mean we only need to perform union and find operations with space and $O(n)$ time complexity $O(n)$.

5 Problem 5

In most opinion-formation models it is assumed that the difference of opinions of two individuals who interact in the social graph decreases during the opinion-formation process. In other words, people's interaction leads to improving agreement. However, in real life, the opposite phenomenon is observed: individuals whose initial opinions are sufficiently far apart, tend to disagree more when they interact with each other. In other words, the difference of opinions of two individuals who disagree "sufficiently enough" tends to increase even more, during the opinion formation process. This is known as the *back-fire phenomenon.*

Propose an opinion-formation model that incorporates the idea of back-fire. State your assumptions and motivate your choices. Argue why the model would lead to back-fire.

Solution

First, let's remember one of the opinion formation models studied in class, known as the *Bounded Confidence Model* (BCM) [3]. Let G be a social network $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. For each node $i \in V$, we associate a continuous opinion $x_i \in [0, 1]$. The idea behind the BCM is that nodes re-adjust their opinion when the difference in their opinions is smaller than a threshold d. Mathematically, suppose two nodes $i, j \in V$ have opinions x_i and x_j respectively. If $|x_i - x_j| < d$, we adjust their opinions as follows:

$$x_i \leftarrow x_i + \mu(x_j - x_i), \tag{5-1}$$
$$x_j \leftarrow x_j + \mu(x_i - x_j), \tag{5-2}$$

where μ is called a *convergence* parameter that quantifies how quickly the opinions of i and j converge to one another. Notice that the threshold d can be interpreted as the *openness* each agent (or node) in the graph has towards changing its opinion. Indeed, large values of d indicate that agents are willing to change their opinion even when interacting with other agents with radically different opinions. On the other side, small values of d indicate very closed minded agents that only interact with others that have almost the same opinions as them.

There are two important limitations we can observe of this model:

1. The assumption that d is constant for all the nodes considered in the population. d clearly can vary depending on each agent since people tend to have different kinds of open-mindedness depending also on the topic of which the opinion is being formed.
2. When $|x_i - x_j| \geq d$, agents simply don't interact and don't change their opinions. This could be thought as if agents have a kind of *apathy* towards other agents with different opinions. Clearly this model doesn't account for the back-fire effect.

We therefore propose a *modified* Bounded Confidence Model that changes conditions given by equations (5-1)-(5-2) in order to account for back-fire. Specifically, this model addresses the second limitation we discussed earlier. For the *modified* BCM we define two thresholds, an agreement threshold d_a and a disagreement threshold d_d. The idea behind this model is that nodes can re-adjust their opinions if any of the following things happen:

- If for two nodes $i, j \in V$ with opinions x_i and x_j respectively we have that $|x_i - x_j| < d_a$, we adjust their opinions in the traditional way of a Bounded Confidence model (see equations (5-1)-(5-2)).
- If for two nodes $i, j \in V$ with opinions x_i and x_j respectively we have that $|x_i - x_j| > d_d$, we adjust their opinions in the following way:

$$x_i \leftarrow x_i - \mu(x_j - x_i), \tag{5-3}$$
$$x_j \leftarrow x_j - \mu(x_i - x_j), \tag{5-4}$$

Notice that in the last case, assuming that $x_i > x_j$ (it can be also the contrary without loss of generality), the quantity $-\mu(x_j - x_i) > 0$ and the quantity $-\mu(x_i - x_j) < 0$, therefore, instead of the opinions of nodes i and j getting closer with the interaction, they diverge far apart from each other, with the divergence being proportional to the difference in their opinions. this could be interpreted as back-fire, since assuming that nodes with very different opinions interact, they would radicalize their opinions even more instead of converging to a common agreement.

We can make the following observations of this model:

- Notice that we can interpret the disagreement threshold as the *close-mindedness* of an agent, since for low values of d_d, agents get quickly radicalized even if opinions are not that far apart (this could model agents that fight about everything if it is not exactly what they think). On the other side, for high values of d_d agents get radicalized only when opinions are very far apart, modeling a more mild assumption that people don't agree with opinions that are very far away from the ones they have.
- Notice that still, we can still have an *apathy* region $d_a \leq |x_i - x_j| \leq d_d$ where agents still don't interact with each other. We could interpret this as the region where agents don't care enough about the others opinions in order to interact.
- The updates of opinions don't have a closed bound, meaning that if an agent continuously interacts with agents that have very different opinions, it may happen that $x_i \notin [0, 1]$ so we need to make a slight modification of the updating equations in (5-1) and (5-2). Therefore, assuming that $x_i > x_j$:

$$x_i \leftarrow \min\left[1, x_i - \mu(x_j - x_i)\right], \tag{5-5}$$
$$x_j \leftarrow \max\left[0, x_j - \mu(x_i - x_j)\right], \tag{5-6}$$

  this modification doesn't change the overall behavior of the model, since it just means that people don't get radicalized ad infinitum.

Now, we can observe an example made by simulating the modified Bounded Confidence Model. Let's assume that G is a Barabási-Albert graph with parameters $n = 20$ and $l = 2$ and that our modified BCM has parameters $\mu = 0.5$ $d_a = 0.3$ and $d_d = 0.7$. Therefore we have the following dynamics:

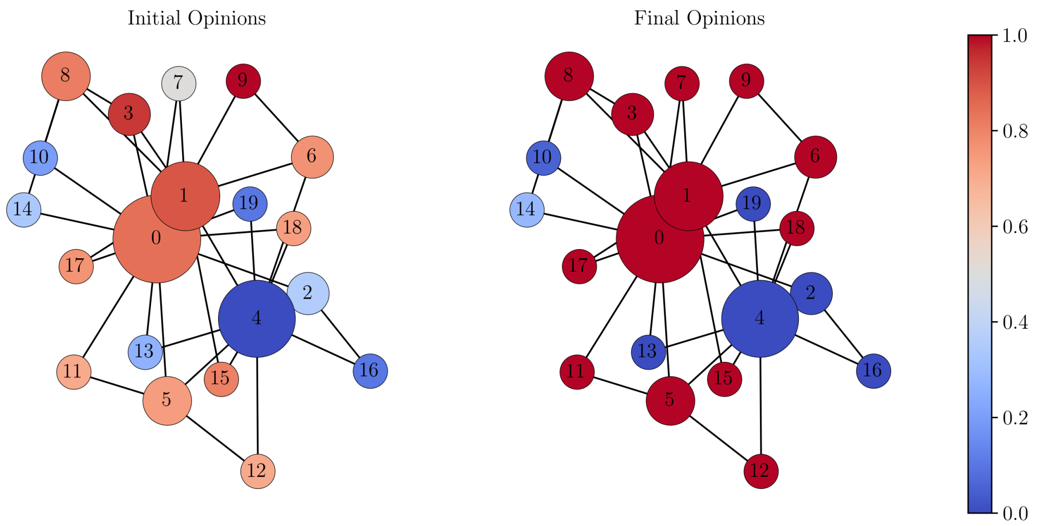


Figure 12: Simulation of the Modified Bounded Confidence Model for 100 time steps, where in each time step 1000 edges are sampled and the opinions of their attached nodes are modified (or not) accordingly. Initial opinions are initialized uniformly from $[0, 1]$.

As we can observe in Figure 5-1 the behavior of the model strongly depends on the initialization of the opinions. In this particular case observe how nodes 0 and 2 start with mildly different opinions but at the end of the simulation are radicalized towards $x_0 = 1$ and $x_2 = 0$ respectively due to their interaction. This is clearly a case of back-fire. Notice that thanks to the component of the original BCM, nodes with similar opinions converge to a common opinion, with the particularity that if some of the neighbors of a node with similar opinions are radicalized, the node is radicalized too. This is an interesting effect, that we would interpret as a sort of *echo chamber* effect.

Finally, it is of interest to observe the overall behavior of the model for different combinations of parameters d_a and d_d. Keeping $\mu = 0.5$ fixed and for G is a Barabási-Albert graph with parameters $n = 1000$ and $l = 2$, we observe the following behavior:

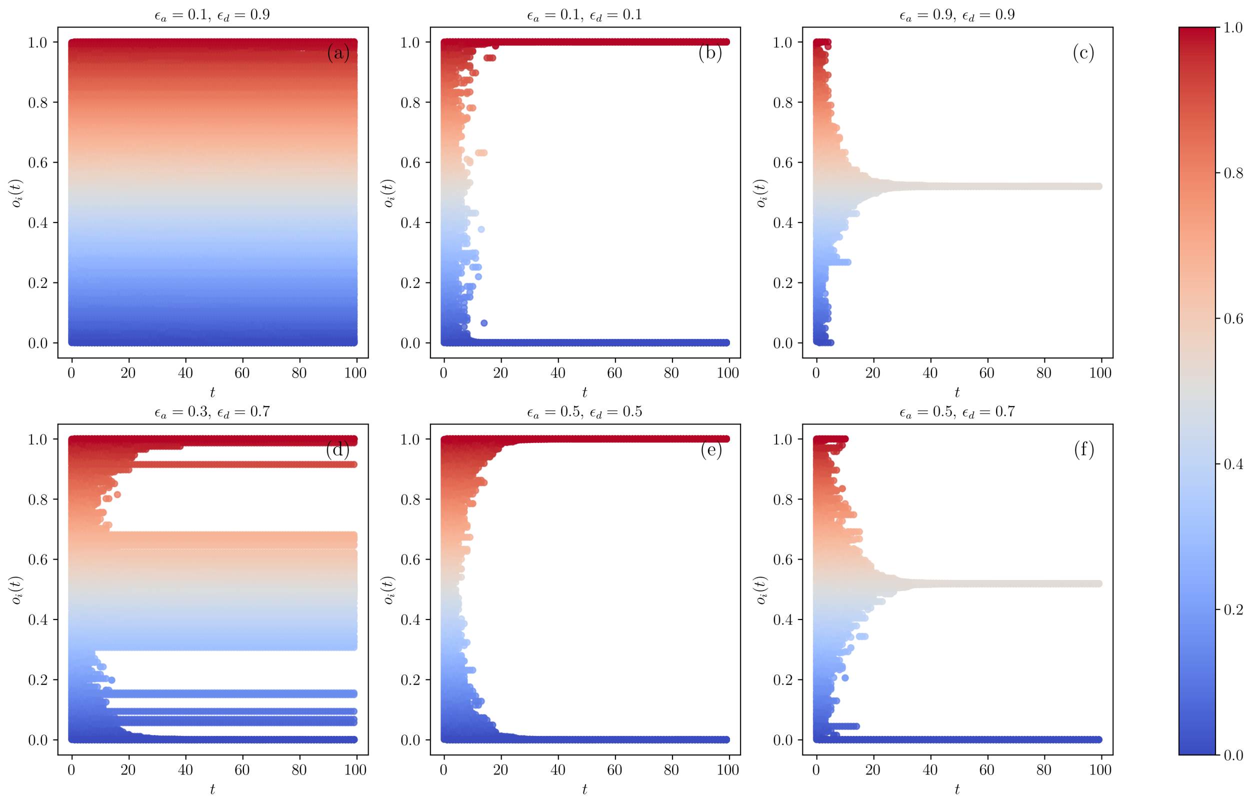


Figure 13: Different simulations of the Modified Bounded Confidence Model for 100 time steps, where in each time step 1000 edges are sampled and the opinions of their attached nodes are modified (or not) accordingly. Initial opinions are initialized uniformly from $[0, 1]$.

As we can see in Figure 13 we have some interesting results for extreme cases of the parameters d_a and d_d[4]. We can see that for a very small value of d_a and a very high value of d_d (see (a)). agents will not even bother to interact, resulting on opinions spread throughout the set $[0, 1]$ (assuming uniform initialization). On the other side, for very small values of d_a and d_d (see (b)), agents will quickly get radicalized, resulting on *polarized* opinions within nodes. Finally, if d_a and d_d have high values (see (c)), opinions will converge to a common value. There exist other interesting cases, for example on (d), with a smaller value of d_a and a higher value of d_d, highly polarized opinions will get radicalized but close enough opinions will be mantained over time, modeling for example an apathic set of agents with a few radical nodes. In general, we conclude that our model effectively accounts for the back-fire effect and that interesting emergent phenomena is observed due to the interaction of radicalized and non-radicalized nodes.

[4]In the case of the Figures, ϵ_a and ϵ_d

6 Applying Graph Neural Networks for Node Classification and Link Prediction Using the PubMed Dataset

In this last part of the homework, we will make a short report of the application of Graph Neural Networks (GNN's) for Node Classification and Link Prediction making a case study for the PubMed dataset. This dataset represents a citation network with nodes representing scientific publications from the PubMed database and edges indicate citations between these publications. Each node features a TF-IDF weighted word vector from the publication's abstract and a class label that denotes the publication's subject category.

6.1 Exploratory Data Analysis

In this section we will perform a basic Exploratory Data Analysis in order to analyze the main features of the PubMed dataset. As a first observation of the dataset we obtained the following characteristics:

Characteristic	**Value**
Number of Features	500
Number of Classes	3
Number of Nodes	19717
Number of Edges	88648
Average Degree	4.5

Table 1: Main characteristics of the PubMed dataset.

In summary, the Pubmed dataset is an undirected graph, without self loops and isolated nodes that includes 3 classes (categories of publications), 19717 nodes (articles), 88648 edges (total citations) and an average degree (average citation number) of 4.5. Moreover, the dataset already contains a separation of the nodes in different training, testing and validation sets, where the training set contains 60 nodes, the testing set contains 1000 nodes and the validation set contains 500 nodes.

6.1.1 Degree Distribution & Clustering Coefficient

It is of intertest to analyze the degree distribution of the network as a whole. This degree distribution can give us an overall picture of probability of finding an article with high, low or intermediate value of citations. In Figure 14 we can observe the degree distribution for the PubMed graph.
A high degree means that the articles were cited by many papers. Intuitively, nodes with high degrees are likely to be important, at least from the number of citations point of view. In Figure 14 we can see that the majority of articles in the PubMed graph have a small number of citations and we would deem them as *unimportant*, whereas there is a non-neglible number of articles with a very significant number of citations, we call this nodes *hubs*. We can also observe that the degree distribution seems to follow a Power Law distribution, a characteristic of social networks in general, specifically $P(k) \sim x^{-\alpha}$, with $\alpha = 1.56$.

Another important quantity to take into account, is the average clustering coefficient for a node. Formally, for undirected graphs this quantity is defined as:

$$C_i = \frac{2L_i}{k_i(k_i-1)}, \tag{6-1}$$

for any node i, where L_i is the number of links between neighbors of node i. Notice that $0 \leq C_i \leq 1$ and that the average of this measure over all the nodes in the graph can be interpreted as the probability that two neighbors of a node selected randomly are connected. In the case of the PubMed graph, we found that the average clustering coefficient is $C \approx 0.06$. This is a very small value, almost zero, meaning that the graph is very sparse with very few edges between neighboring nodes.

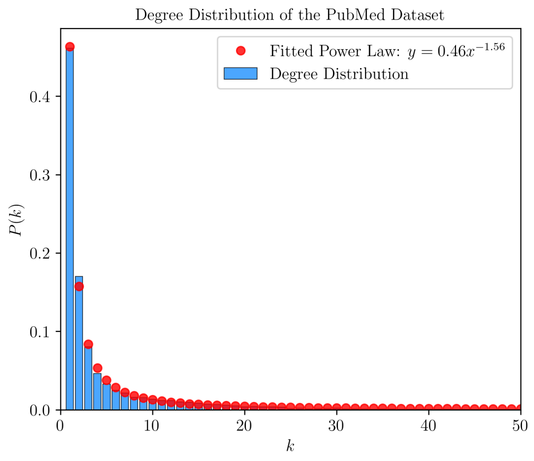


Figure 14: Degree Distribution of the PubMed graph. Notice that $P(k)$ follows a power law distribution.

6.1.2 Homophily

Although we've seen that the clustering is very low for the PubMed graph, it is of interest to see if nodes with the same class are more connected to each other than to others in other classes. This property is called *homophily*. We can quantify homophily by quantify the probability that nodes of the same class are connected. In Figure 15 we can observe this probability:

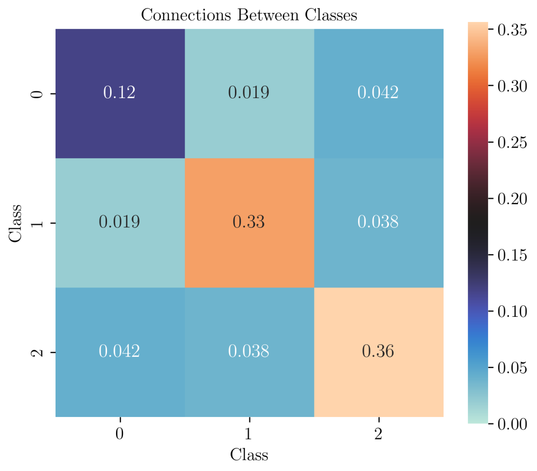


Figure 15: Probability for nodes of class k for being connected with nodes of every other class, represented as a Heatmap.

As we can observe, indeed there is indeed more probability of having a connection with a node of the same class between nodes in the PubMed dataset, although this probability is still low, roughly 30% for classes 2 and 3 and only 10% for the first class.

6.1.3 Node Features

As we've mentioned before, the node features of the PubMed graph are 500-sized vectors that contain the normalized TF-IDF values for a vocabulary of 500 words of the publications abstract. Lets remember that TF-IDF values are a way to represent the importance of a word in a document within a corpus. TF stands for Term Frequency, which measures how frequently a term occurs in a document. IDF stands for Inverse Document Frequency, which measures how unique or rare a term is across the entire corpus. By combining these two metrics, TF-IDF assigns higher weights to terms that are frequent within a document but rare across the corpus, thus capturing their importance in conveying the meaning of that document.

It is of interest to see how these values are spread for articles of each class. This could give us a broad insight on the topics of each class and the difference between them. In Figure 16 we plot the mean and standard deviation of the TF-IDF values for word in the vocabulary and for each class.

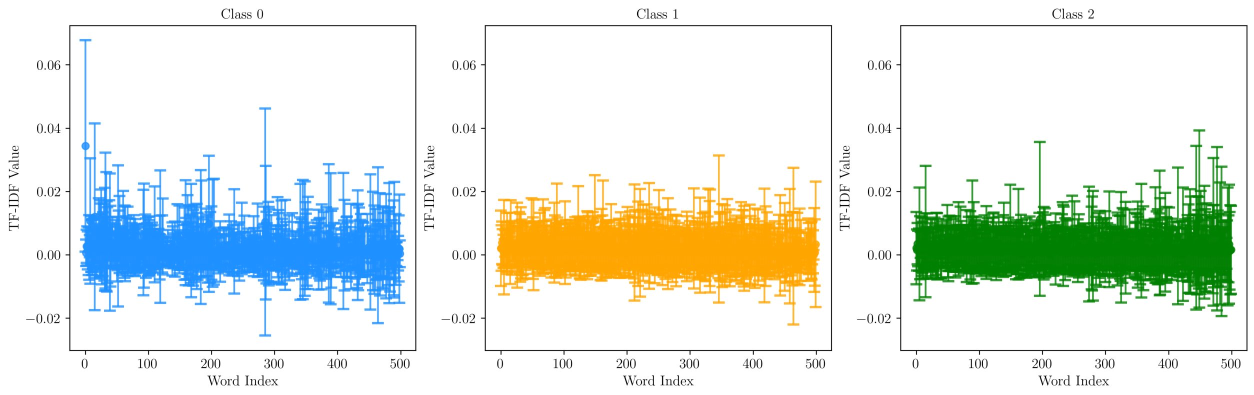


Figure 16: Mean and standard deviation of the TF-IDF values for word in the vocabulary and for each class in the PubMed dataset.

As we can see, the means value of the TF-IDF of each word in the index are roughly the same for all of the classes (indeed they must be to a certain extent, since these publications are of very similar topics). Nevertheless we can see some differences in the standard deviation, meaning that there are some words are particular of the class we are analyzing.

6.1.4 Class Distribution

Finally, we can analyze the distribution of the node classes in the PubMed graph. This is important since the distribution of node classes can significantly impact the performance and evaluation of machine learning models, particularly in tasks such as node classification. If certain classes are underrepresented compared to others, it can lead to biased models that perform well on majority classes but poorly on minority classes. In Figure 17 we plot the class distribution of the overall graph.

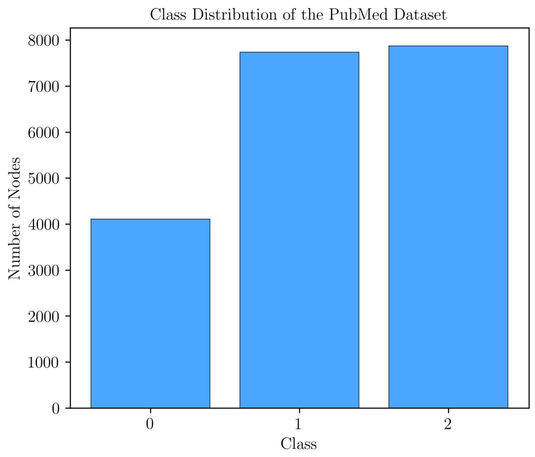


Figure 17: Node class distribution for the overall PubMed graph.

As we can see, class 0 is underrepresented in comparison with the other two classes of the dataset. In order to have a correct training of the GNN models we must ensure that this underrepresentation doesn't exist in the training set. Indeed, if we plot the class distribution of the training, testing, and validation set nodes predetermined by PyTorch (see Figure 18), we can observe that for the training class set all classes are equally represented in order to ensure that no bias is introduced when training our GNN models.

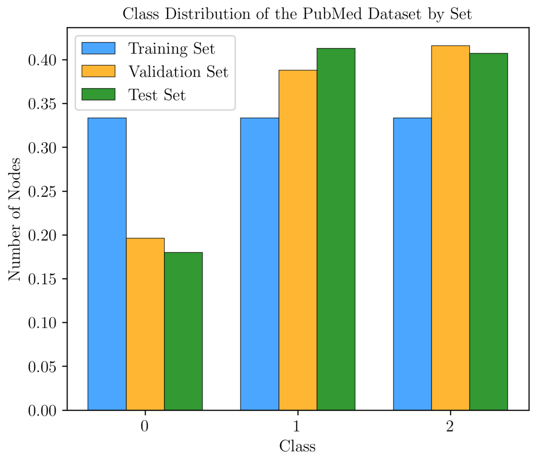


Figure 18: Node class distribution for the training, testing, and validation set of nodes for the PubMed graph predetermined by PyTorch.

On the other side, the testing and validation sets are imbalanced but with a similar distribution between them.

6.2 Graph Neural Networks (GNN)

Before delving fully into using GNN's to perform classification or prediction tasks in our chosen dataset, let's make a quick introduction of what they are and how are they useful. Graph Neural Networks (GNNs) are a class of deep learning models that operate on graph inputs. These networks have gained immense popularity in recent years because of their wide applicability in domains such as knowledge graphs, social networks and molecular biology. They can be used to learn embeddings for graph entities (nodes, edges, or entire graphs) and have shown remarkable performance on tasks like graph classification, node classification, link prediction and so on.

The common theme to any type of GNN is that it implements some form of neural message passing, whereby messages (in the form of vectors) are exchanged between the nodes of the graph to iteratively update the internal representations of the graph's nodes. Formally, given a graph $G = (V, E)$ along with any relevant node features $\vec{X} \in \mathbf{R}^{d_0 \times n}$, a GNN can be used to generate node embeddings $\vec{p}_v, \forall v \in V$. This is done iteratively as follows:

1. Consider hidden embedding vectors $\{\vec{h}_v^k\}$ representing each node $v \in V$. In each iteration k, every embedding vector $\vec{h}_v^k$ is updated based on information inferred from the corresponding local neighborhood, denoted as $\mathbf{N}_v = \{u \in V | (u, v) \in E\}$.

2. At layer (iteration) $k = 0$, the initial representations $\vec{h}_v^0 \in \mathbf{R}^{d_0}$ are usually derived from the node's labels or given input features of dimensionality d_0. This single layer update can then be formalized as:

$$\vec{m}_v^k = \text{AGGREGATE}_\theta^k \left(\vec{h}_u^{k-1} | u \in \mathbf{N}_v \right) \tag{6-2}$$

$$\vec{h}_v^k = \text{UPDATE}_\theta^k \left(\vec{h}_v^{k-1}, \vec{m}_v^k \right) \tag{6-3}$$

   for the GNN layers (iterations) $k = 1, \ldots, K$ with AGGREGATE$(\cdot)$ and UPDATE$(\cdot)$ referring to some differentiable functions.

The vector $\vec{m}_v^k$ represents the k-th layer message for node $v = 1, \ldots, n$ as aggregated from the corresponding local graph neighborhood $\mathbf{N}_v$. At each iteration k (see Figure 19), every node aggregates information from its local neighborhood, and as these iterations progress each node embedding encapsulates a larger receptive field within the graph.

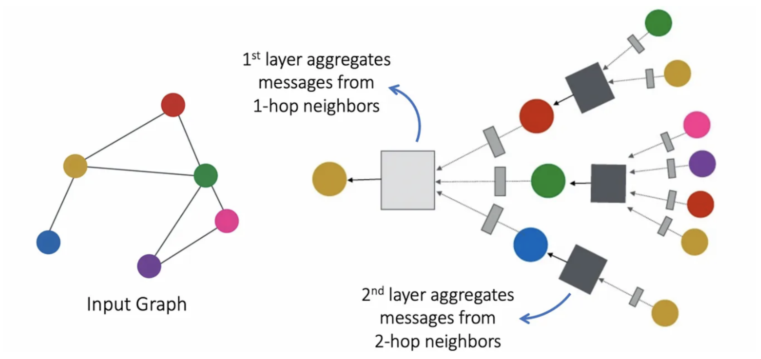

1st layer aggregates messages from 1-hop neighbors
Input Graph
2nd layer aggregates messages from 2-hop neighbors

Figure 19: A stack of GNN layers operating.

Specifically, after k iterations every node embedding contains information about its k-hop neighborhood, with the final output (after K iterations of message passing) defined as $\vec{p}_v = \vec{h}_v^K$. This output can then be used for prediction tasks, such as node classification or link prediction. To optimize the predictive power of this approach, the final node embeddings $\vec{p}_v = \vec{h}_v^K$ are fed into a problem-specific loss function, with some form of stochastic gradient descent optimizing the weight parameters of the network.
In this project, we will explore three different GNN architectures based on the three most popular GNN models: GCN, GraphSAGE and GAT.

6.2.1 Graph Convolutional Networks (GCN)

Most existing GNN algorithms can be written in the format given on equations (6-2) and (6-3), and different choices for aggregate and combine functions yield different GNN models. The first of the ones used for this project is called Graph Convolutional Network (or GCN) [4]. In this model, the aggregate and combine functions are integrated into the following process:

$$\vec{h}_v^k = \sigma\left(W_k \sum_{u \in \mathbf{N}_v \cup \{v\}} \frac{\vec{h}_u^{k-1}}{\sqrt{d_u d_v}}\right), \tag{6-4}$$

with W_k is a shared trainable weight matrix, the denominator serves as a normalization factor, where d_v is the degree of node v and $\sigma(\cdot)$ being some nonlinear activation function, like eLU or ReLU.

6.2.2 GraphSAGE

The GraphSAGE model [5] was one of the first models to propose the ability to generalize to unseen nodes during inference. While the original paper presents various options for the aggregate function, the one we'll use in this project is the mean aggregator, where we simply take the elementwise mean of the vectors in $\{\vec{h}_u^{k-1}, \forall u \in \mathbf{N}_v\}$. The mean aggregator is nearly equivalent to the convolutional propagation rule used in the transductive GCN framework. In particular, we use a variant of the GCN approach defined as follows:

$$\vec{h}_v^k = \sigma\left(W_k \cdot \text{CONCAT}\left(\vec{h}_v^{k-l}, \text{MEAN}\left(\{\vec{h}_u^{k-l}, \forall u \in \mathbf{N}_v\}\right)\right)\right), \tag{6-5}$$

with W_k is a shared trainable weight matrix and $\sigma(\cdot)$ being some nonlinear activation function, like eLU or ReLU.

6.2.3 Graph Attention Networks (GAT)

Graph Attention Networks (GATs) [6] are a variant of Graph Neural Networks (GNNs) that leverage attention mechanisms for feature learning on graphs. Standard GNNs, such as Graph Convolutional Networks (GCNs), do not differentiate between the contributions of different neighbors. GATs, on the other hand, assign an attention coefficient to each neighbor, indicating the importance of that neighbor's features for the feature update of the node. The aggregation and update functions are modified to include attention coefficients, which are calculated as follows:

1. First, for each node v, we compute a set of unnormalized attention coefficients with each of its neighbors $u \in \mathbf{N}_v$:

$$e_{vu} = \text{LeakyReLU}\left(\vec{a}^\top \left[W\vec{h}_v \parallel W\vec{h}_u\right]\right)$$

   where $\vec{a}$ is a learnable weight vector, W is a shared weight matrix, and $\parallel$ denotes concatenation.

2. These coefficients are then normalized using the softmax function:

$$\alpha_{vu} = \text{softmax}_u(e_{vu}) = \frac{\exp(e_{vu})}{\sum_{k \in \mathbf{N}_v} \exp(e_{vk})}$$

   The normalized attention coefficients α_{vu} indicate the importance of node u's features to node v.

3. Next, the node embeddings are updated by computing a weighted sum of the features of neighboring nodes:

$$\vec{h}_v^k = \sigma\left(\sum_{u \in \mathbf{N}_v} \alpha_{vu} W \vec{h}_u^{k-1}\right)$$

   where $\sigma(\cdot)$ is a nonlinear activation function, such as eLU or ReLU.

6.3 Node Classification with GNN's

In this project, we use a simple two-layer GNN architecture based on PyTorch GCN, GraphSAGE and GAT units. For the whole three models we followed the same steps of construction:

1. The first GNN layer is fed the initial node embeddings $\vec{h}_v^0$ of dimension d_0 (these embeddings contain the features of the graph, in our case, the TF-IDF values) and outputs a representation of size d_1.
2. Next, we apply a component-wise, non-linear ReLU transformation (In order to prevent overfitting, we add a probability of dropout as well).
3. The second GCN(GraphSAGE) layer is then fed a intermediate representation vector $\vec{h}_v^1$ and outputs a layer of size $d_2 = q$, where q is the number of classes in which the nodes of the graph will be classified.
4. Finally, the output is fed through a component-wise softmax transformation to provide one-hot encoded q-dimensional probabilistic node assignments $\vec{p}_v \in [0, 1]^q$ that represent the probabilities that each node belongs to a given class.

In order to make comparisons and tuning the loss obtained when training our GNN models, we performed two approaches when training each one of our GNN models:

- In a first approach we trained our GNN models by fully considering all the training nodes in our training set. This potentially would overfit the model since we are considering a big quantity of nodes that may be irrelevant when obtaining the embedding of another node.
- In a second approach we performed *Neighbor Sampling.* The basic idea of this is that for every node, instead of using the entire neighborhood information (i.e., all connected k-hop neighbors), we select or sample a subset of them to perform the aggregation needed in graph embedding. This could be benefitial to avoid overfitting and high computational cost.

The loss function used to train the GNN models was the traditional Cross Entropy Loss and the optimizer for gradient descent was the traditional ADAM. The hyperparameters used for training are reported in the following table:

Hyperparameter	**Value**
Initial Dimension (Number of Features)	500
Hidden Dimension	16
Output Dimension (Number of Classes)	3
Dropout Probability	0.5
Learning Rate	0.01
Weight Decay	5×10^{-4}
Epochs	300

Table 2: Hyperparameters for Training the GNN Models

6.3.1 GCN Results

For the GCN model with the architecture presented in last section, we can observe the training and validation loss functions for Full Batch Training and Mini-Batch Training (using Neighbor Sampling) as a function of the epochs in Figure 20:

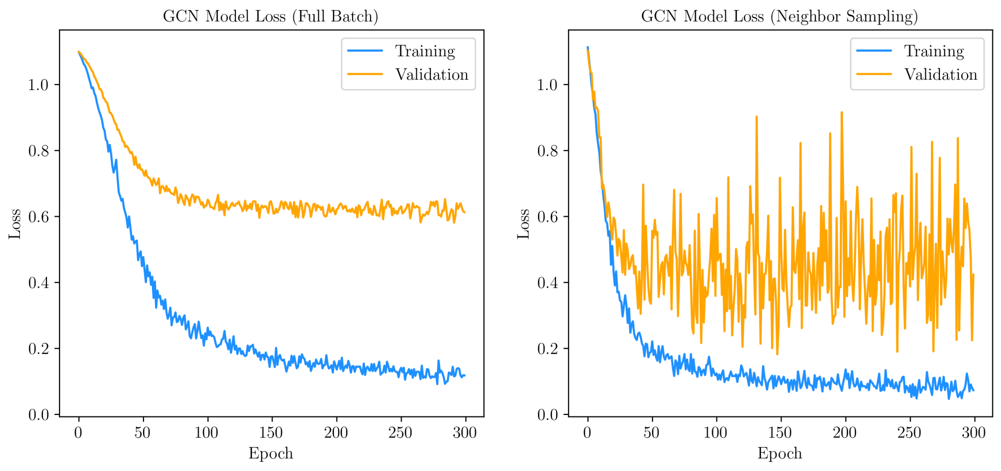


Figure 20: Training and Validation Loss Functions for Full Batch and Mini Batch training of a node classification model using GCN layers.

As we can observe, the validation loss function of the training of the GCN model using Full-Batch training stops decreasing and is relatively constant after a number of epochs. This could be a sign of overfitting. This means that the GCN model might be starting to memorize the training data instead of generalizing from it. This is why we opted out for a second approach using Mini-Batch training via Neighbor Sampling, where we can observe the same pattern but with much more variation around the mean, since we are training in subsets of the whole graph. The metrics obtained for both of these approaches are reported Table 3.

As we can see, the model achieved a classification accuracy of 79.6%, a precision of 79.3%, a recall of 78.9% and an F1-Score of 79.0%. In general we can conclude that the GCN model achieved a strong overall ability to correctly classify nodes into their categories for the PubMed dataset. On the other side, we can observe that we obtained lower accuracy values for the GCN model trained using Mini-Batch training, thus indicating that Mini-Batch training might not be as effective as Full-Batch training for this specific task and dataset even though it helped reduce overfitting as we mentioned before.

Metric	Full Batch Training	Mini Batch Training
Accuracy	79.6%	78.5%
Precision	79.3%	77.5%
Recall	78.9%	78.8%
F1-Score	79.0%	78.1%

Table 3: Precision Metrics for the GCN Model.

6.3.2 GraphSAGE Results

For the GraphSAGE model with the architecture presented in last section, we can observe the training and validation loss functions for Full Batch Training and Mini-Batch Training (using Neighbor Sampling) as a function of the epochs in Figure 21:

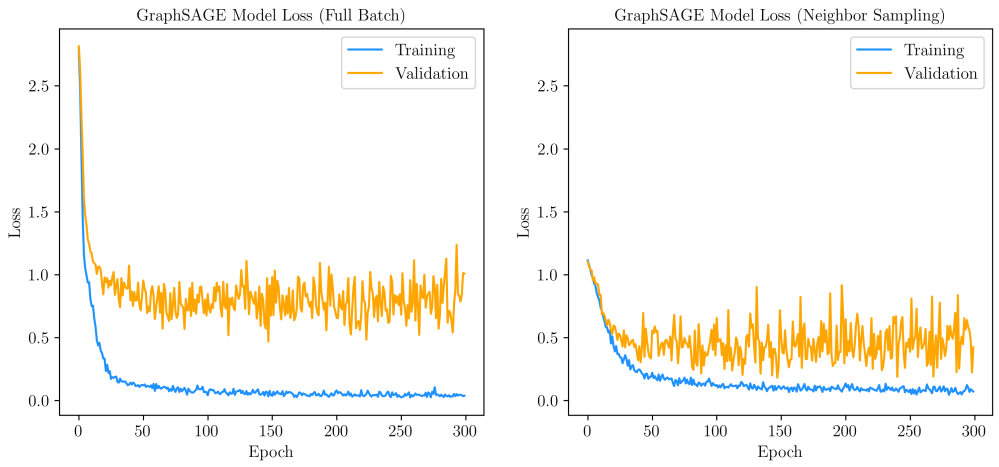


Figure 21: Training and Validation Loss Functions for Full Batch and Mini Batch training of a node classification model using GraphSAGE layers.

As we can observe, the validation loss function of the training of the GCN model using Full-Batch training stops decreasing and is relatively constant after a number of epochs as in the case of the GCN Model. This could be a sign of overfitting of this model as well. This is why we opted out for a second approach using Mini-Batch training via Neighbor Sampling, where we can observe that we were able to significantly reduce the overall values of loss and we were able to make the validation loss function descend with the training loss indicating that the model does not overfit as before. The metrics obtained for both of these approaches are reported Table 4.

Metric	Full Batch Training	Mini Batch Training
Accuracy	76.9%	76.9%
Precision	76.3%	77.0%
Recall	76.7%	76.5%
F1-Score	76.4%	76.7%

Table 4: Precision Metrics for the GraphSAGE Model.

As we can see, both approaches achieved very similar results, with the difference of the overall pattern of the loss functions mentioned before. This model achieved worse results than our GCN architecture, with the main difference that training was a little bit faster than before.

6.3.3 GAT Results

Finally, for the GAT model with the architecture presented in last section, we can observe the training and validation loss functions for Full Batch Training and Mini-Batch Training (using Neighbor Sampling) as a function of the epochs in Figure 22:

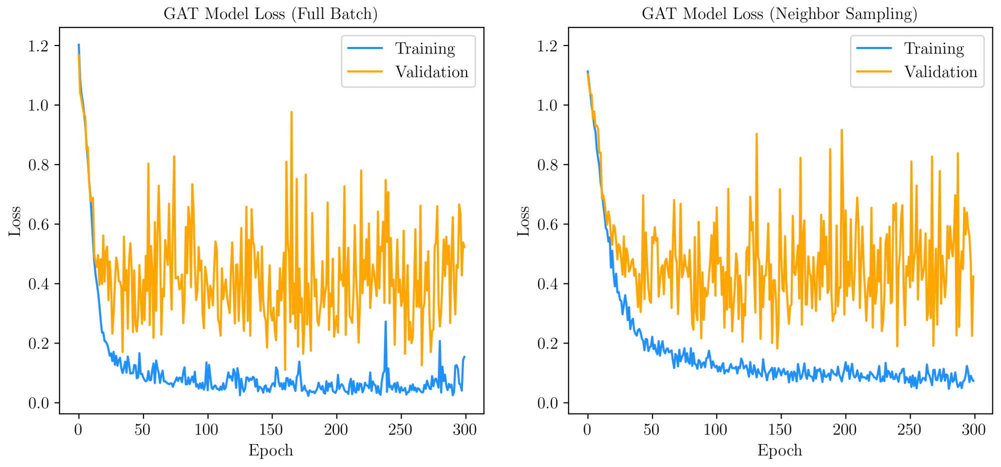


Figure 22: Training and Validation Loss Functions for Full Batch and Mini Batch training of a node classification model using GAT layers.

As we can observe, overall, the validation and training loss functions for both Full-Batch and Mini-Batch training have a similar behavior, both exhibiting high variance in the validation loss. The metrics obtained for both of these approaches are reported Table 5.

Metric	Full Batch Training	Mini Batch Training
Accuracy	77.8%	68.7%
Precision	78.3%	69.38%
Recall	76.0%	72.4%
F1-Score	76.9%	67.9%

Table 5: Precision Metrics for the GraphSAGE Model.

As we can see, in this case, Full-Batch training achieved significantly better results even though that the overall pattern of the loss functions mentioned before was the same. This model achieved better results than our GraphSAGE architecture, but worse results than our GCN architecture, making us conclude that using GAT for this particular task might not be the optimal choice. Overall, all GNN architectures achieved similar results with some differences, and it seems that for this classification task choosing between them is basically a trade-off between computational complexity and performance accuracy. For example, while GCN is easy to implement, it might not capture complex node relationships as effectively as GAT¿ even though this last architecture is more computationally intensive.

6.4 Link Prediction with GNN's

Link prediction on GNN's is trickier than node classification since we need a more complex architecture to make predictions on edges using node embeddings. The steps that are normally followed during link prediction steps are described below:

1. We create a GCN Model *encoder* that creates node embeddings by processing the graph with two GCN layers.

2. After doing this we randomly add *negative* links to the original graph. This induces a model task: binary classification, since we need to classify the positive links from the original edges and the negative links from the added edges.

3. We create a *decoder* which makes link predictions (i.e. binary classifications) on all the edges including the negative links using node embeddings. The decoder normally calculates a dot product of the node embeddings from pair of nodes on each edge. Then, it aggregates the values across the embedding dimension and creates a single value on every edge that represents the probability of edge existence.

An architecture diagram of this process can be seen in Figure 23.

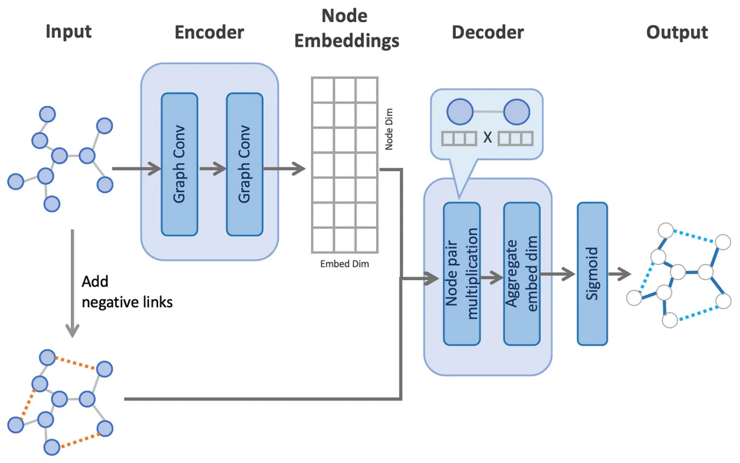


Figure 23: Architecture diagram for performing Link Prediction using GNN's.

Notice that in this case we will needed to create our own Training, Testing and Validation Data. To do this we performed Random Link Sampling, this process performs random sampling of links (edges) from the original graph to create training, testing, and validation datasets. This method ensures that the subsets of the data are representative of the overall structure and characteristics of the graph. The loss function used to train the link prediction model was the Binary Cross Entropy Loss since we were training for a binary classification class and the optimizer for gradient descent was the traditional ADAM. The hyperparameters used for training are reported in the following table:

Hyperparameter	**Value**
Initial Dimension (Number of Features)	500
Hidden Dimension	250
Output Dimension (Number of Classes)	125
Dropout Probability	0.5
Learning Rate	0.01
Weight Decay	5×10^{-4}
Epochs	100

Table 6: Hyperparameters for Training the GCN Architecture for Link Prediction.

6.4.1 Results

In Figure 24 we present the loss function for the training of the Link Prediction Model. For this model we obtained a ROC-AUC value of roughly 86%. A ROC-AUC score of around 86% signifies that our model had a robust performance of the in discerning between existing and non-existing links within the network.

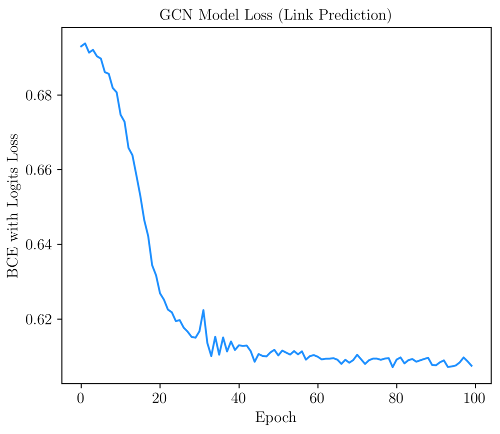


Figure 24: Training Loss Function for training of a link classification model using a GCN encoder.

Essentially, the ROC-AUC metric quantifies the model's ability to rank positive instances (i.e., actual links) higher than negative instances (i.e., non-existent links) across various threshold values. To study the performance of our metric, we also obtained the Hits@K metric defined as the proportion of correctly predicted links among the top K ranked predictions. Hits@K metric provides insight into the model's effectiveness in identifying true links within the top K predictions. For this model we considered Hits@10 and Hits@3 where we obtained a value of 1 for both cases, indicating that all of the true links were correctly predicted within the top 10 and top 3 ranked predictions, respectively[5].

6.5 Final Remarks and Conclusions

Even though we were not able to achieve the benchmark results for the node classification task on the pubMed dataset, we observed that all three GNN architectures exhibited strong performance in distinguishing between different node categories. As we've seen, GCN model achieved the highest classification accuracy of 79.6%, closely followed by GraphSAGE (76.9%) and GAT (77.8%). However, it is noteworthy that the performance of the GCN model deteriorated slightly when trained using mini-batch training via neighbor sampling, indicating that full-batch training might be more effective for this specific task and dataset. In contrast, GraphSAGE and GAT models demonstrated consistent performance regardless of the training approach.

Furthermore, even though we weren't able to achieve the benckmark resuslts for the link classification task either, our evaluation of link prediction using a GCN encoder revealed promising results, with a ROC-AUC score of approximately 86%. This also indicates the model's ability to effectively discriminate between existing and non-existing links within the network. In general, we can conclude that the devil is in the details, since a slight change in one layer or hyperparameter can significantly affect training in a positive or a negative way. Maybe with more time to work on this problem we could've tried more exhaustively different approaches and performed parameter tuning in order to achieve better results.

[5]There was probably an error in code but I was not able to find it in time.

References

[1] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[2] Mark Newman. *Networks*. Oxford University Press, 07 2018.

[3] Guillaume Deffuant, David Neau, Frédéric Amblard, and Gérard Weisbuch. Mixing beliefs among interacting agents. *Advances in Complex Systems*, 3:87–98, 01 2000.

[4] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[6] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

A Code Appendix

Source Code 1: Python code to obtain the second smallest eigenvalue and corresponding eigenvector of the normalized Laplacian matrix $\mathcal{L}$ given by equation 2-7 of the graph given by Figure 1.

```python
import numpy as np
import networkx as nx
from typing import Tuple
from numpy import linalg as LA

def second_smallest_eigenvalue(A: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    #We first obtain the eigenvalues and eigenvectors of the matrix
    w, v = LA.eig(A)
    #We sort the eigenvalues
    sorted_w = np.sort(w)
    #We obtain the index of the second smallest eigenvalue
    index = np.where(w == sorted_w[1])
    #We obtain the second smallest eigenvalue and its corresponding eigenvector
    second_smallest_eigenvalue = w[index]
    second_smallest_eigenvector = v[:,index]
    #We return the second smallest eigenvalue and its corresponding eigenvector
    return second_smallest_eigenvalue[0], second_smallest_eigenvector[:, 0]

def normalized_laplacian(A: np.ndarray, d: int) -> np.ndarray:
    #Here we define the identity matrix
    I = np.eye(A.shape[0])
    #Here we return the normalized laplacian matrix
    return I - (1/d)*A

if __name__ == "__main__":
    #Here we create the edge list of the graph
    edge_list = [(1,2), (1,3), (1,4), (1,7), (2,3), (2,4), (2,5), (3,4), (3,7), (4,5),
    (5,6), (5,9), (6,8), (6,9), (6,10), (7,8), (7,10), (8,9), (8,10), (9,10)]

    #Here we create the graph
    G = nx.from_edgelist(edge_list)

    #Here we obtain the adjacency matrix of the graph
    A = nx.adjacency_matrix(G).todense()
    #Here we obtain the normalized laplacian matrix of the graph
    L = normalized_laplacian(A, 4)
    #Here we obtain the second smallest eigenvalue and its corresponding eigenvector
    #of the normalized laplacian matrix
    second_smallest_value, second_smallest_vector = second_smallest_eigenvalue(L)
```

Submitted by Miguel Ángel Sánchez Cortés on June 9, 2024.