Université Paris cité
UFR des Sciences Fondamentales et Biomédicales

PPD Report 1

MSc : Machine Learning for Data Science (MLDS)

Prepared by :
Mr. MAHMAHI Anis
Mr. HATTABI Ilyes

Supervised by :
Mr. LABIOD Lazhar

Promotion : 2023/2024

Contents

1 Background

Graph Neural Networks (GNNs) are advanced neural network architectures designed to process graph-structured data, which are highly effective in various applications such as node classification, link prediction, and recommendation systems. GNNs can aggregate information from neighboring nodes and capture both local and global graph structures.

GNNs extend traditional neural networks to operate directly on graphs, a data structure capable of representing complex relationships.

A graph consists of sets of nodes or vertices connected by edges or links. In GNNs, nodes represent entities (like users or items), and edges signify their relationships or interactions.

GNNs follow an encoder-decoder architecture. In the encoder phase, the graph is fed into the GNN, which computes a representation or embedding for each node, capturing both its features and context within the graph.

The decoder phase involves making predictions or recommendations based on the learned node embeddings, such as computing similarity measures between node pairs or using embeddings in downstream models.

In the context of GNNs for recommender systems, the goal is to generate embeddings for the user and item nodes in the graph. The embeddings can then be used for tasks such as candidate generation, scoring, and ranking.

2 Key Terms

- **Node Embeddings**: are low-dimensional vector representations that capture the structural and relational information of nodes in a graph. GNNs are designed to learn these embeddings by iteratively aggregating information from neighboring nodes.

- **Aggregation Functions**: are used in GNNs to combine information from neighboring nodes. Common aggregation functions include summation, averaging, and max-pooling, among others. These functions determine how information is aggregated and propagated through the graph.

- **Message Passing**: is a fundamental operation in GNNs where nodes exchange information with their neighbors. During message passing, each node aggregates the information from its neighbors to update its own representation.

  Message Passing Layer mathematically is :

$$h_u^{(k+1)} = UPDATE^{(k)}\left(h_u^{(k)}, \text{AGGREGATE}^{(k)}\left(\left\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\right\}\right)\right)$$

  The state update for a node u is mainly performed using the two already introduced operations **aggregate** and **update**.

  AGGREGATE uses the states of all direct neighbors v of a node u and aggregates them in a specific way (min, max, mean ...) then the **UPDATE** operation uses the current state in time step k and combines it with the aggregated neighbor states.

This basic formula stays the same for all variants of message passing GNN (Graph Convolutional Networks (GCN), Graph Attention Networks (GAT) ...) the only thing in which they are different is how they perform the update and aggregate functions.

3 GNN Training Process

3.1 How it works

let's assume this is our input graph in the following :

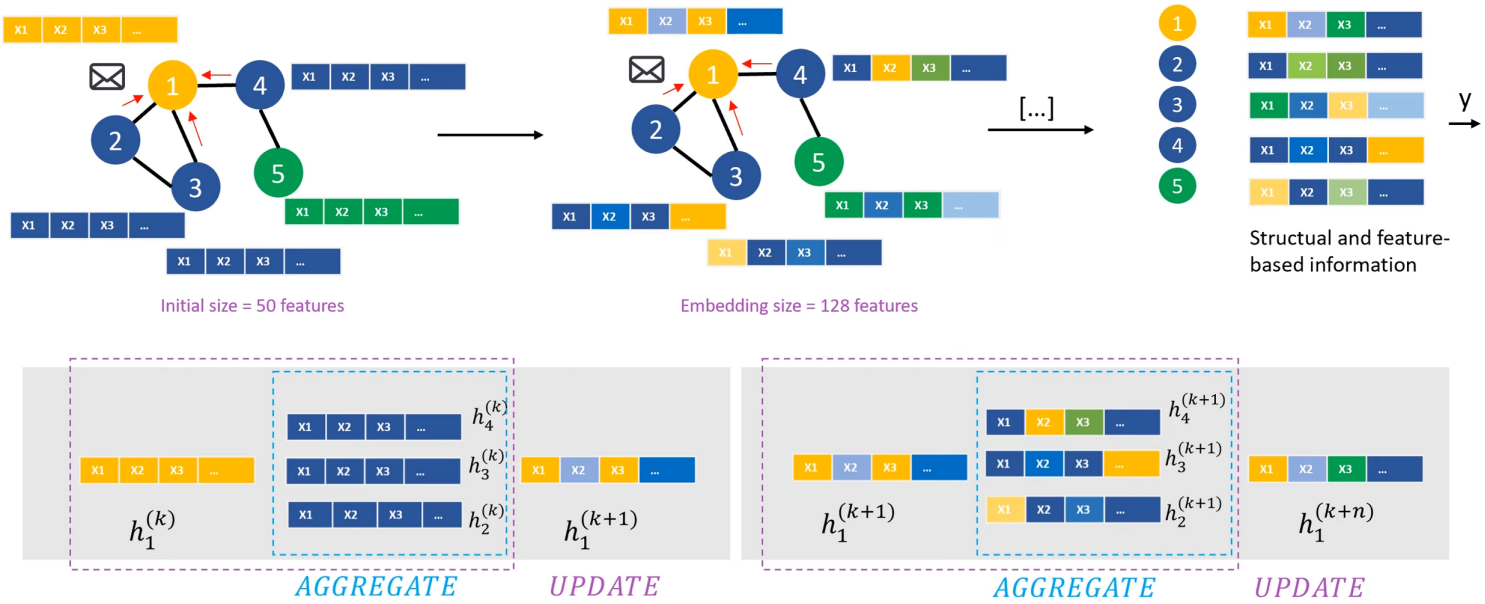


Figure 1: How GNN works

We see in Fig.1 that there is one yellow node with the number 1 with a yellow node state vector $h_1^{(k)}$ at time step k. This is the node we will focus on in this example, to update the note state $h_1^{(k)}$ we collect the information of the direct neighbors, which means we perform a message passing.

What we end up with is the information about our current node state $h_1^{(k)}$ and the information about our neighbors node states as well $h_2^{(k)}$, $h_3^{(k)}$, $h_4^{(k)}$, $h_5^{(k)}$.

Then we perform an aggregation on the neighbor states to combine their information, finally we put our current state and the combined neighbor information together to get a new state or embedding called $h_1^{(k+1)}$. Note how some of the feature information of the blue nodes enters the state of the yellow node.

This message passing is done by all nodes and therefore we have new embeddings for every node in our graph by the end of k step.

The size of these new embeddings is a hyper-parameter and depends on the graph data we use. As we can see the node (5) holds only information about the blue node and itself because it's green and blue in $k+1$. This node doesn't know about our yellow node (1) but this will change.

Let's perform another message passing step to see what happens (actually we can

perform several of these message passing steps which corresponds to the **number of layers** in GNN).

Again we use the current node embedding of our yellow node, collect the state messages of its neighbors and aggregate them in some way and update it. by $k + 2$, we see that some information about the green node passed into it.

This means that node (1) knows something about node (5), but additionally in our example every single node in the graph knows something about all other nodes, this knowledge is stored in each of our node embeddings and contains the feature-based as well as structural information about the nodes. Eventually we can use the embeddings to perform predictions through adding MLP or sklearn-head on top of it.

3.2 How many GNN layers do we need ?

From Fig.1, we see that after two layers the yellow node (1) already contains the information about all nodes in the graph (represented in its embedding vector).

The number of layers in a gnn defines how many neighborhood hops we perform, this number is a hyper-parameter and depends on the graph size, if you have small graphs you can quickly learn all the information after only a few layers. The number of layers also depends on the learning task sometimes only a local area of the graph might be relevant for your predictions. But stacking too many message passing layers in a gnn can also lead to a phenomenon called **over smoothing** which means basically make all node states indistinguishable from each other as illustrated in Fig 2.

In our previous example if we keep combining these states over many more layers we will not learn anything new but instead make all node states indistinguishable from each other.

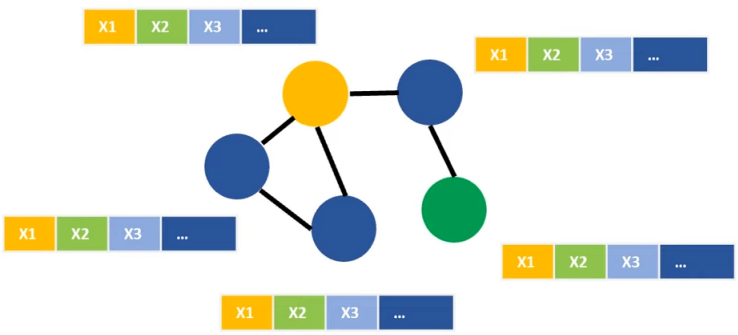


Figure 2: Over-Smoothing phenomenon

4 Overview of Graph Convolutional Networks (GCN)

GCNs [2] are a popular type of GNN architecture. They adapt the concept of convolutions from CNN to the graph domain, enabling information propagation and feature extraction across the graph.

The general idea of GCN :

For each node, we get the feature information from all its neighbors and of course, the feature of itself. Assume we use the average() function as aggregation function (for simplification purposes only). We will do the same for all the nodes. Finally, we feed these average values into a neural network which correspond to an update function.

Let's consider the green node in Figure 3. First off, we get all the feature values of its neighbors, including itself, then take the average. The result will be passed through a neural network to return a resulting vector.

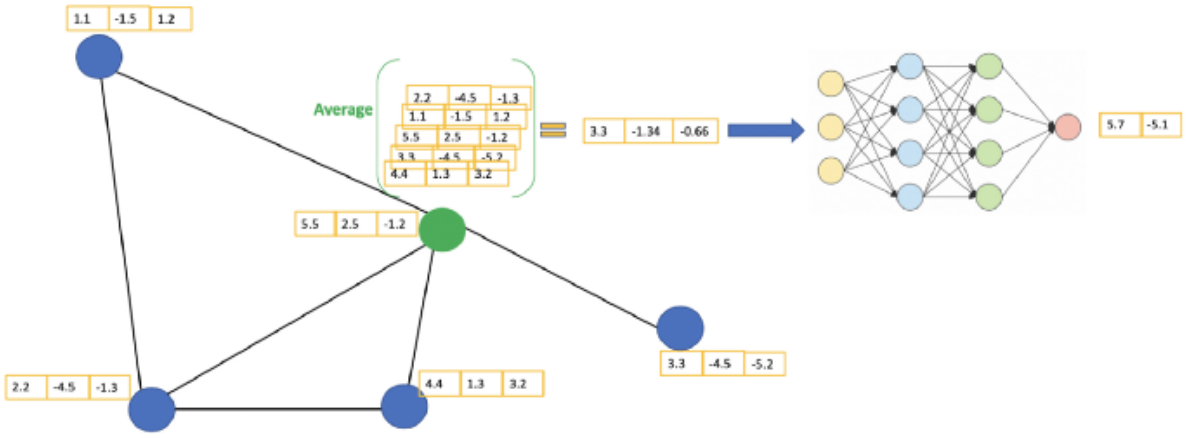


Figure 3: GCN Overview

In practice, we use more sophisticated aggregate functions rather than the average function. We can also stack more layers on top of each other to get a deeper GCN. The output of a layer will be treated as the input for the next layer.

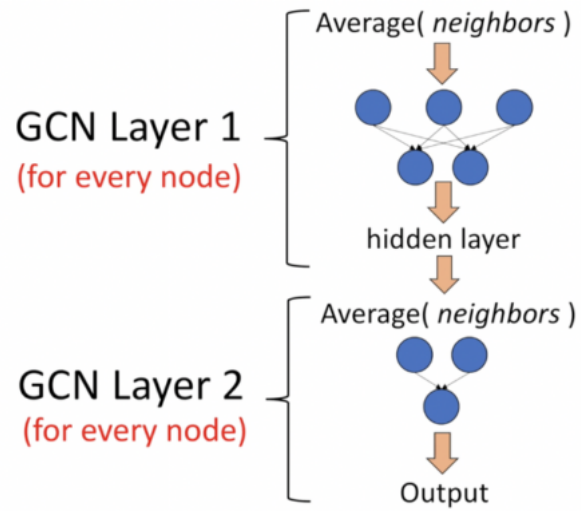
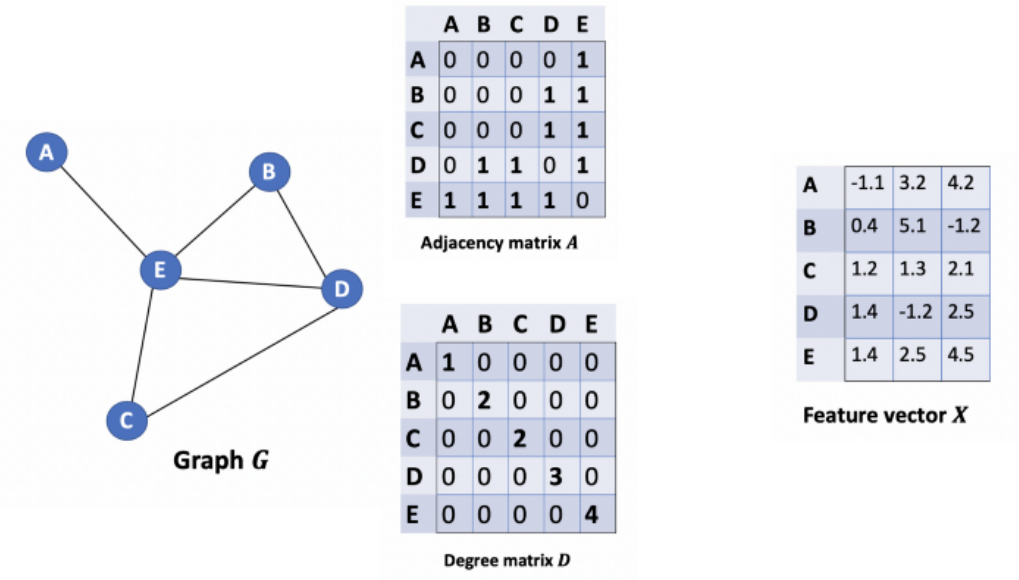


Figure 4: Multiple GCNs Layers

Lets start by some definitions :



How can we get all the feature values from neighbors for each node? The solution lies in the multiplication of A and X.

In the adjacency matrix, we see that node A has a connection to E. The first row of the resulting matrix (A*X) is the feature vector of E, which A connects to. Similarly, the second row of the resulting matrix is the sum of feature vectors of D and E. By doing this, we can get the sum of all neighbors' vectors.

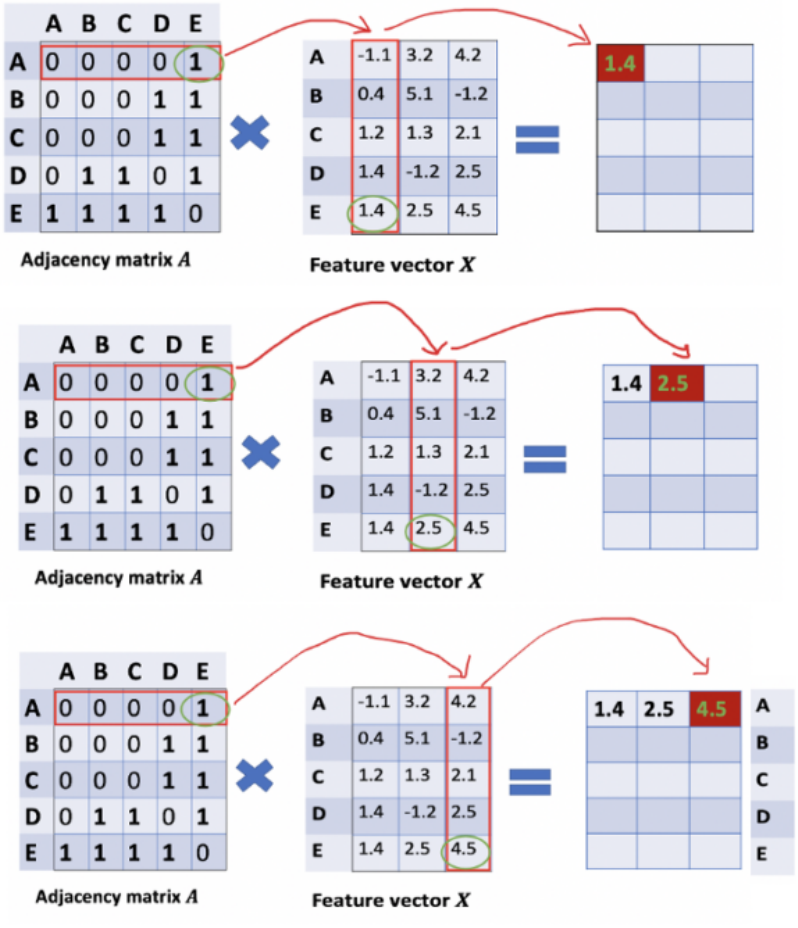


Figure 5: A*X

There are still some things that need to improve here.

1. We miss the feature of the node itself. For example, the first row of the result matrix should contain features of node A too.

2. Instead of sum() function, we need to take the average, or even better, the weighted average of neighbors' feature vectors. **Why don't we use the sum() function?** The reason is that when using the sum() function, high-degree nodes are likely to have huge v vectors, while low-degree nodes tend to get small aggregate vectors, which may later cause **exploding or vanishing gradients** (e.g., when using sigmoid). Besides, Neural networks seem to be **sensitive to the scale of input data**. Thus, we need to normalize these vectors to get rid of the potential issues.

In Problem (1), we can fix by adding an Identity matrix I to A to get a new adjacency matrix $\tilde{A}$.

$$\tilde{A} = A + \lambda I_N$$

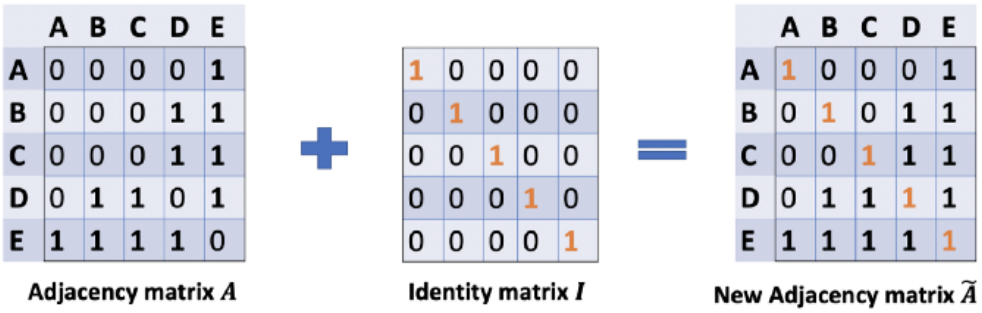


Figure 6: Adding self loops

In Problem (2), to scale the resulting matrix (A*X) (take the average), a solution is to multiply by the inverse of new D matrix

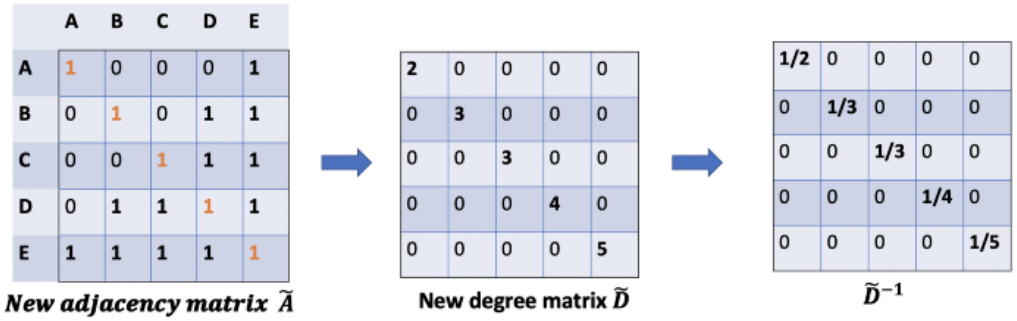


Figure 7: new D matrix

For example, node A has a degree of 2, so we multiple the sum vectors of node A by 1/2, while node E has a degree of 5, we should multiple the sum vector of E by 1/5, and so on.

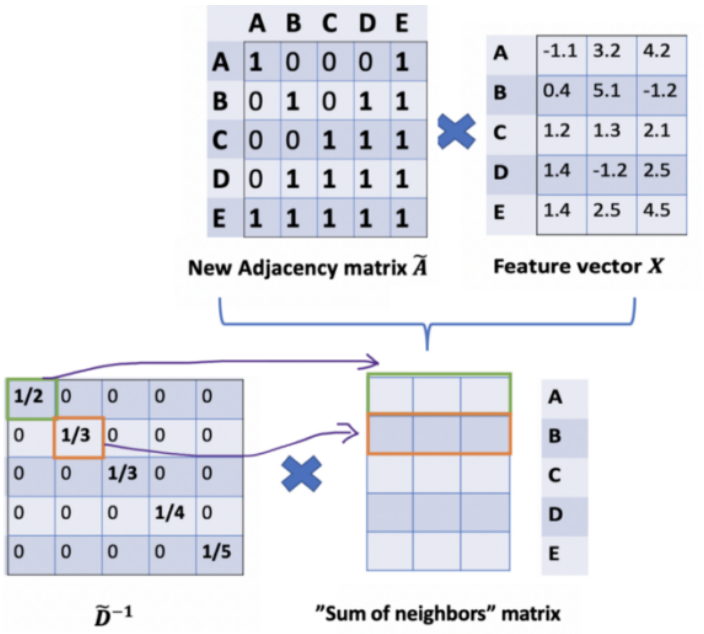


Figure 8: Scaling the resulting matrix

So what we did until now can be translated mathematically to :

$$\tilde{D}^{-1}\tilde{A}X$$

What if we apply **weighted average**, which means scaling not only by rows but by columns as well, and this can be achieved by $\tilde{D}^{-1}\tilde{A}\tilde{D}^{-1}X$:

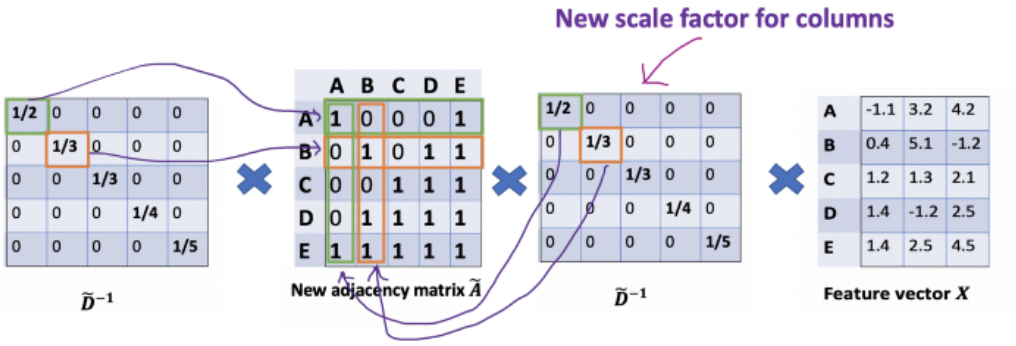


Figure 9: weighted average

What are we doing here is to put more weights on the nodes that have low-degree and reduce the impact of high-degree nodes. The idea of this weighted average is that we assume low-degree nodes would have bigger impacts on their neighbors, whereas high-degree nodes generate lower impacts as they scatter their influence at too many neighbors.

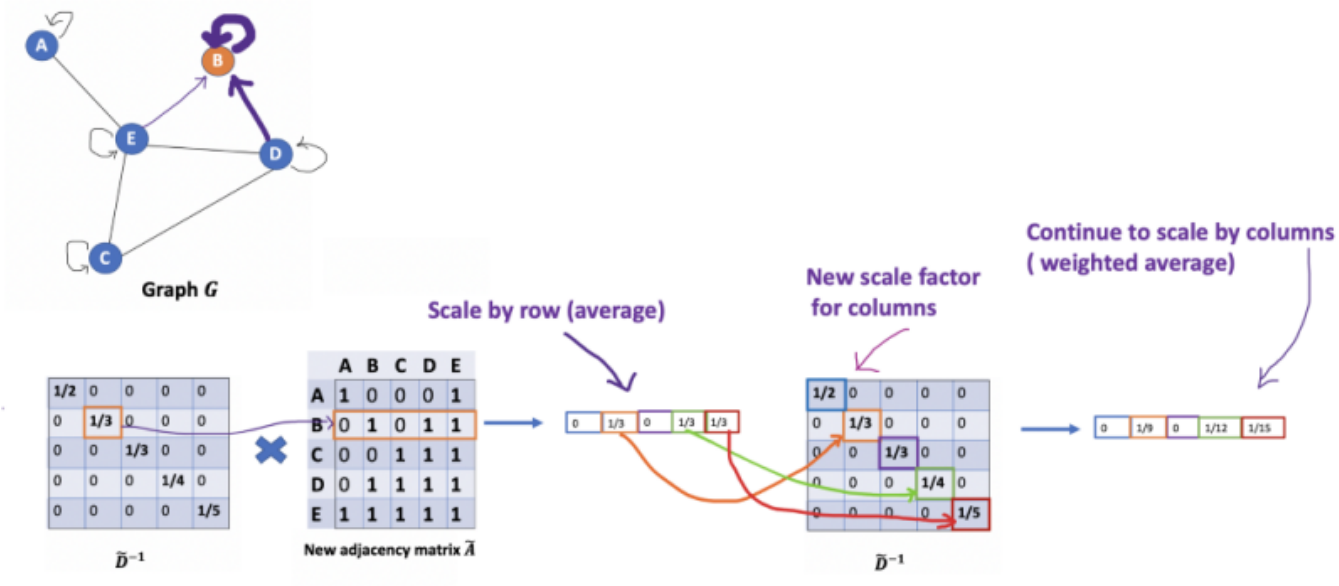


Figure 10: When aggregating feature at node B, we assign the biggest weight for node B itself (degree of 3), and the lowest weight for node E (degree of 5)

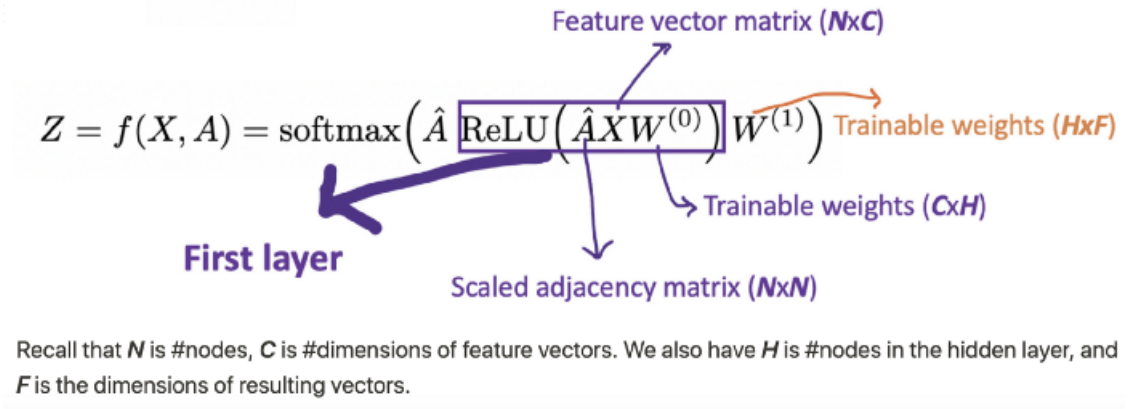
One minor note : in the official paper, they used $\tilde{D}^{-1/2}$ over $\tilde{D}^{-1}$ because of the twice normalization.

If we put all things together, the resulting matrix that will go throught the neural network is :

$$\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X$$

Example :

Let's call $\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ just for a clear view. With 2-layer GCN, we have the form of our forward model as below :



Recall that N is #nodes, C is #dimensions of feature vectors. We also have H is #nodes in the hidden layer, and F is the dimensions of resulting vectors.

Bibliography

[1] Aman Chadha and Vinija Jain. Graph neural networks. *Distilled AI*, 2020. https://aman.ai.

[2] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[3] Chau Pham. Graph convolutional networks (gcn). https://www.topbots.com/graph-convolutional-networks/, 2022.

[4] Jure Leskovec (Stanford). Excellent slides on graph representation learning. https://drive.google.com/file/d/1By3udbOt10moIcSEgUQOTR9twQX9AqOG/.

[5] WelcomeAIOverlords. Video graph convolutional networks (gcns) made simple. https://www.youtube.com/watch?v=2KRAOZIULzw.