DGL 2025 Coursework 1

Asia Belfiore

CID: 02129867 ab6124@ic.ac.uk

Department of Computing
Imperial College London

February 20, 2025

Abstract

Instructions: This is a structured report template for your DGL 2025 coursework. Please insert your written answers, discussions, and figures in the designated sections. **Do not include any code** in this report. All code should remain in your Jupyter notebooks.
Note: We have **kept the structure the same as the Coursework Description PDF** to maintain consistency across your notebooks and this report template. Please keep your headings and subheadings aligned with those in the provided instructions. However, if a section primarily relates to code implementation, you may keep your answers concise (e.g., reference your notebook or provide brief clarifications).

1 Graph Classification

1.1 Graph-Level Aggregation and Training

1.1.a Graph-Level GCN

A function '*global_aggregation*' was created: based on the '*graph_aggregation_method*' (str) parameter, it computes the appropriate element-wise aggregation ('**mean**', '**sum**', '**max**', or '**none**') of all the node embeddings passed as parameter. The default behaviour (in case no valid aggregation name is passed as parameter) is the '*mean*' aggregation.
In the 'MyGraphNeuralNetwork' class, '*global_aggregation*' is called in the forward pass on the output of the last GCN layer of the Network, and returns the result of the aggregation of all the learned node embeddings of the graph.
Each aggregation method was implemented using the appropriate built-in **torch** functions. Additionally, self-connections were added to the adjacency matrix of the graph in the forward pass of the Network (in order to comply with the requirements of the forward pass of the GCN layer). Please refer to the Q1 **.ipynb** file for full solution and implementation details.

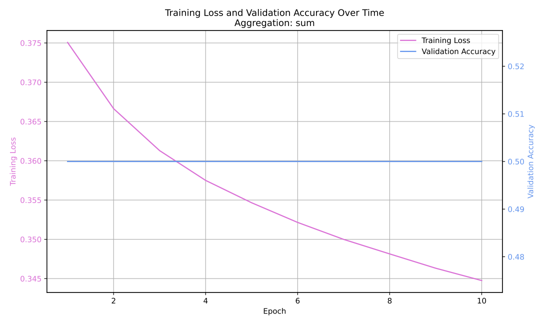
NOTE: The '*global_aggregation()*' function was created outside the 'MyGraphNeuralNetwork' class in order to be available anywhere in the file (and for any model), and the '*none*' aggregation method was added for testing purposes.

1.1.b Graph-Level Training

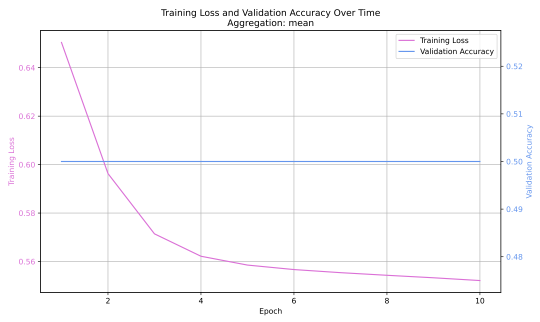
The given '*train_epoch*' and '*test*' functions were merged and adapted into a new function '*run_epoch*' which, based on a boolean parameter, runs the model's training OR evaluation pipeline for a single epoch. The function returns the average loss, accuracy, true and predicted labels for the current epoch. It optionally returns the learned embeddings (based on a boolean parameter).
A new function '*train_model*' was added to perform the entire training and testing pipeline for a given number of epochs. At each epoch, the model is first trained on the given training dataset and then validated on the validation dataset (by sequential calling of the aforementioned '*run_epoch*' function), storing the loss and accuracy for both steps in four separate lists.
An additional boolean parameter, '*f1*', is added for optional F1 score calculation for both training and validation, based on the true and predicted labels returned by the '*run_epoch*' function.
Loss, Accuracy and F1 scores for all aggregation methods are shown in Figure 1.
Please refer to the Q1 **.ipynb** file for full solution and implementation details.

1.1.c Training vs. Evaluation F1

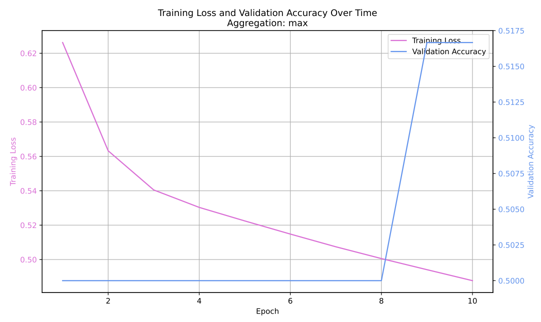
All aggregation methods lead to very poor performance, showing unsatisfactory accuracy and loss and as the model seems to not learn during training, as shown by the flat accuracy over epochs (Figure 1d) and high achieved loss around 0.5.
Furthermore, the model shows high variance and inconsistency between different runs, with either the *sum* or *max* aggregations leading to random F1 spikes at the end of training on occasional runs (Figure 2). In the example run shown in Figure 2, the **max**



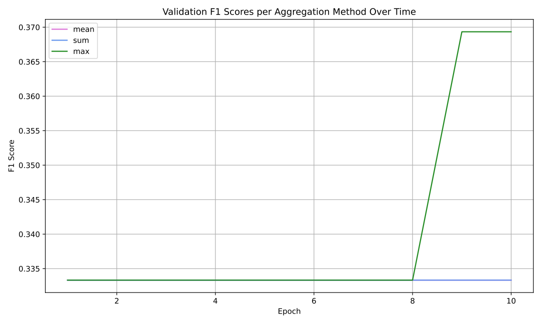
(a) Training Loss and Validation Accuracy per epoch with Sum Aggregation.



(b) Training Loss and Validation Accuracy per epoch with Mean Aggregation.



(c) Training Loss and Validation Accuracy per epoch with Max Aggregation.



(d) Validation F1 Score per epoch for every Aggregation method.

Figure 1: Global Sum, Max and Mean Aggregation Loss and Accuracy Comparison

aggregation slightly outperforms the other methods, as it shows lower loss and better convergence for both testing and validation out of all the aggregation methods, along with higher final F1 score on both testing and validation datasets.
The *mean* aggregation performs the worst out of all the methods on every run, possibly due to the fact that averaging node features leads to loss of specificity and does not retain the most important features across the graph nodes, as opposed to the other two methods.

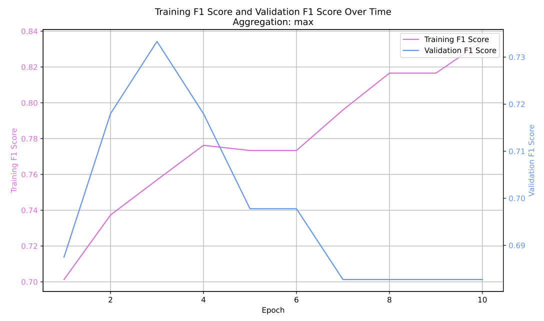
1.2 Analyzing the Dataset

1.2.a Plotting

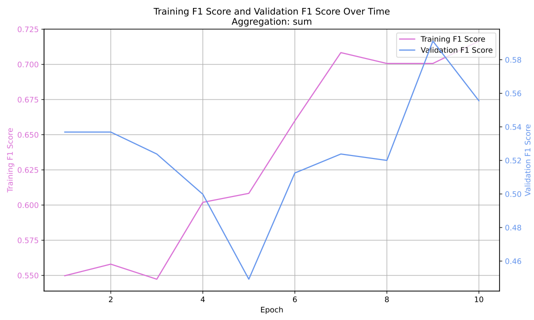
All plots can be found in Figure 3. Please refer to the Q1 **.ipynb** file for full solution and implementation details.

1.2.b Discussion

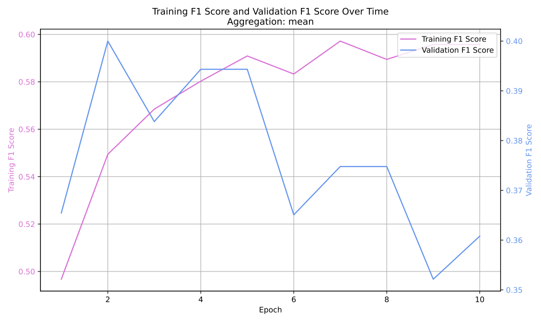
There is an evident class imbalance between the Training and Validation Datasets: while the two node classes (Class 0 and Class 1) are about equally present in the Validation Dataset, there is a clear overpowering of Class 1 in the Training dataset, having about three times the amount of nodes of Class 0 (Figure 3a,3b).
This can help explain the difference in accuracy between training and validation steps.



(a) Training and Validation F1 Score per epoch with Max Aggregation.



(b) Training and Validation F1 Score per epoch with Sum Aggregation.



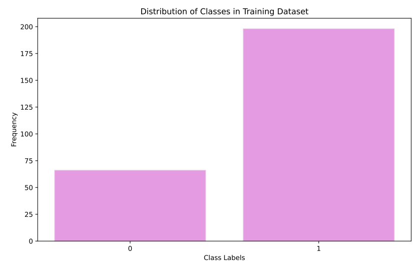
(c) Training and Validation F1 Score per epoch with Mean Aggregation.

Figure 2: Comparison of Training and Validation F1 scores for all Aggregation Methods

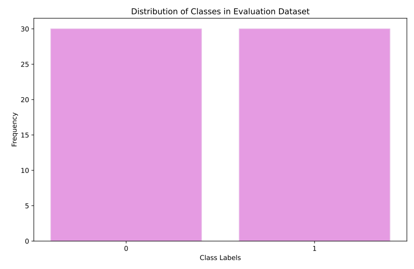
The distribution of features is about the same between the two datasets (Figure 3c,3d). Both classes have similar feature dimensions, with most nodes having both first and second feature dimensions centred around 0, forming a clear spherical cluster within the [-0.4,+0.4] range in both dimensions.
The Validation dataset is noticeably smaller in size compared to the Training, thus some sparsity is introduced in the feature distribution; however, the overall scattering of the feature dimensions is preserved.
Finally, due to the class imbalance issue between the two datasets, there is a visible difference in density of features between the two datasets, with Class 1 nodes of the Training dataset being much more densely present in the aforementioned central cluster.

The topologies of both classes are about preserved between the two datasets and so are the topologies of the nodes in the two classes (Figures 3e-3h).
Class 0 nodes resemble a fully connected graph/clique, with most nodes sharing edges.
Class 1 nodes resemble a more modular graph, with both datasets showing three main clusters (namely A,B,C), with one central cluster (B) connected to each of the two other clusters (i.e. A connected to B, C connected to B, A and C not connected).

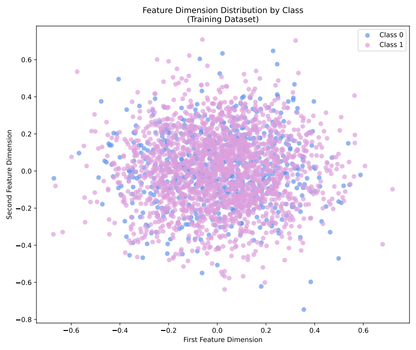
The preserved graph characteristics between the two datasets should lead to steady and similar model accuracies during training and testing, however the evident node class imbalance is undoubtedly the source of confusion in the model, responsible for the difference in performance between test and validation.



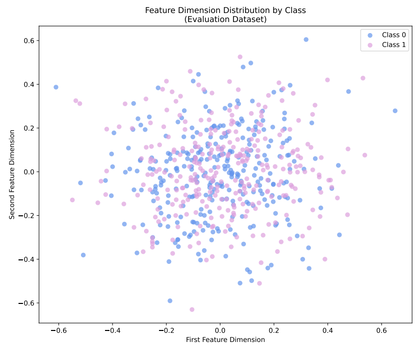
(a) Distribution of classes in the Training Dataset



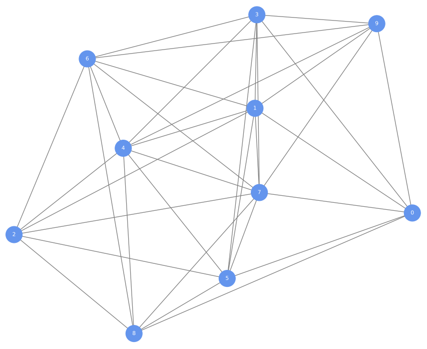
(b) Distribution of classes in the Validation Dataset



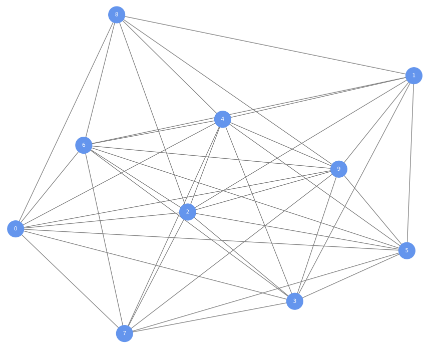
(c) Feature Dimensions distribution per class in the Training Dataset.



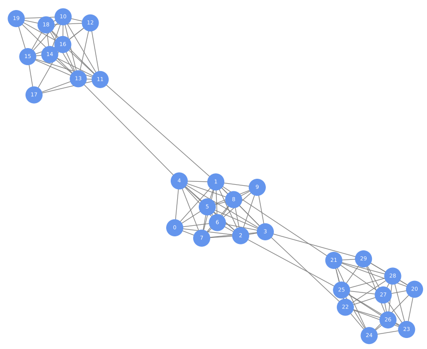
(d) Feature Dimensions distribution per class in the Training Dataset.



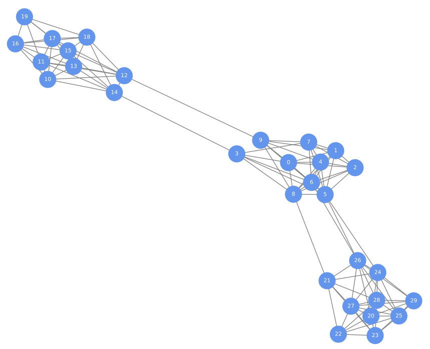
(e) Topology of class 0 nodes of Training Dataset.



(f) Topology of class 0 nodes of Validation Dataset.



(g) Topology of class 1 nodes of Training Dataset.



(h) Topology of class 1 nodes of Validation Dataset.

Figure 3: Topological Analysis of Tranining and Validation Datasets

1.3 Overcoming Dataset Challenges

1.3.a Adapting the GCN

A new class '**MyGraphNeuralNetwork2**' and a new class '**MyGCNLayer2**' were created, modelled on the basis of the original 'MyGraphNeuralNetwork' and 'MyGCNLayer' classes. '*MyGraphNeuralNetwork2*' represents the adapted GCN, with two regular GCN layers followed by a **Classification Head**, a Linear Layer that maps the D-dimensional final graph embedding learned by the GCN layers to a 1-dimensional output for classification. The network is defined by the following parameters:

- **Number of Layers** (L) to allow the model to dynamically create L GCN layers using a for loop and the torch *ModuleList()* class.
- **Input dimension** (D=10) of the graph node features, used as input dimension for the first GCN layer.
- Desired **hidden dimension** (H) of the GCN layers, used as output dimension of the embeddings of the first layer and as input-output dimension of the following hidden layers.
- Desired **output dimension** (D') of the last GCN layer (i.e. final dimension of the graph embeddings), used as output dimension of the last GCN layer and as input dimension for the classification head.

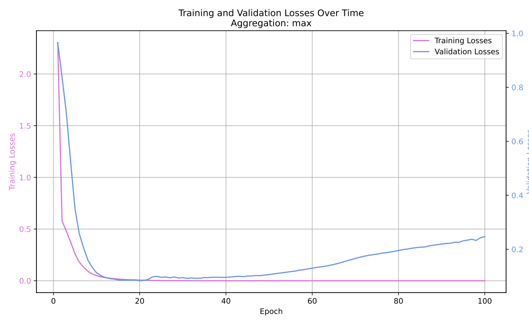
For an input graph of size (NxD) (i.e. N nodes with D features each) and model with hidden dimension H and L GCN layers, the model will:

1. (NxD $\rightarrow$ NxH) Pass all N nodes through the first GCN layer to get N nodes with feature size H.
2. (NxH $\rightarrow$ NxH) Pass the embeddings through L-2 GCN layers which will keep the dimensionality unchanged.
3. (NxH $\rightarrow$ NxD') Pass the embeddings through the last GCN layer which will output features of size (NxD').
4. (NxD' $\rightarrow$ 1xD') Perform global aggregation on all the graph nodes.
5. (1xD' $\rightarrow$ 1) Pass the aggregated embedding through the Classification Head to get a 1-D output for binary graph classification.

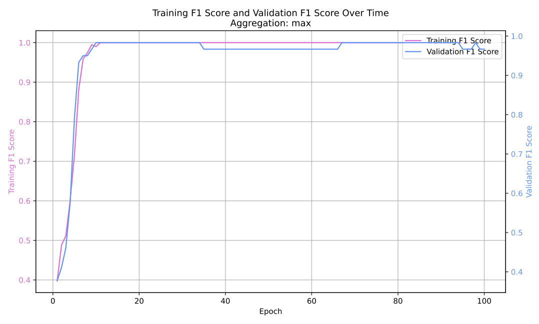
The class performs a chosen **global aggregation** following the methods described in Q1.1.a. during the forward pass, on the embeddings of the last GCN layer and before the classification head is applied. The model optionally returns ALL the N embeddings of every GCN layer as a list.
Full solutions can be found in the Q1 **.ipynb** file.

1.3.b Improving the Model

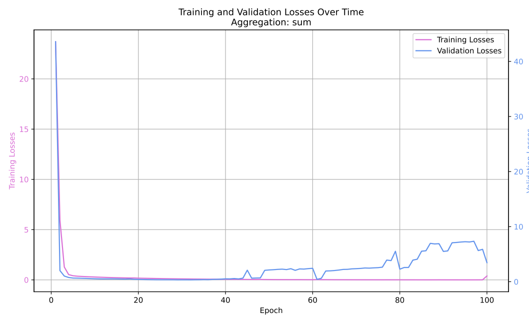
Performance Results of the Improved Model are shown in Figure 4, with the final model reaching **100%** Accuracy on both the Training and Validation Datasets (Figure 4c,4d). Improvement strategies are detailed below in Section **1.3.d 'Final Analysis and Explanation'** and results of experimentations are shown below in Figure 6.
Full implementations and explorations can be found in the Q1 **.ipynb** file.



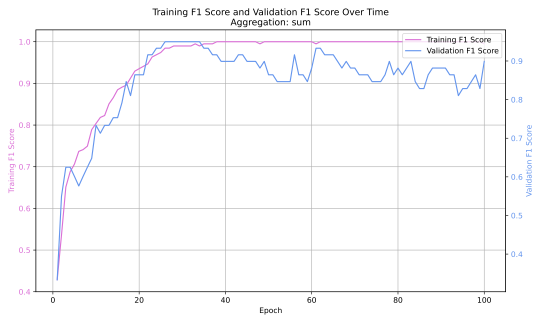
(a) Training and Validation Loss per epoch with Max Aggregation.



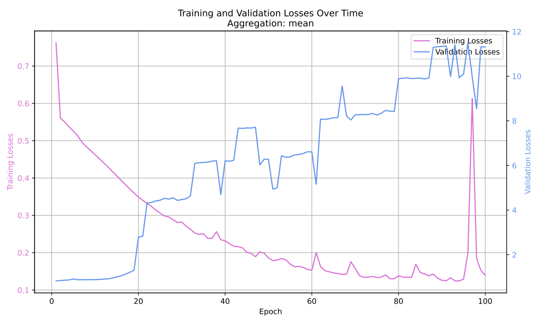
(b) Training and Validation F1 Score per epoch with Max Aggregation.



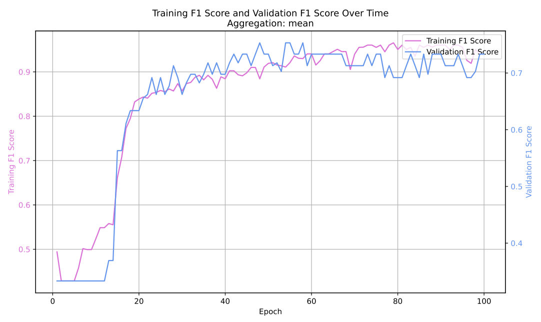
(c) Training and Validation Loss per epoch with Sum Aggregation.



(d) Training and Validation F1 Score per epoch with Sum Aggregation.



(e) Training and Validation Loss per epoch with Mean Aggregation.

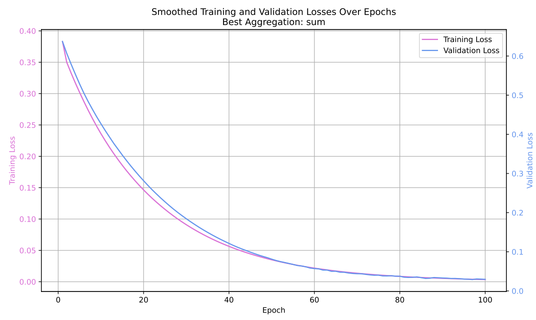


(f) Training and Validation F1 Score per epoch with Mean Aggregation.

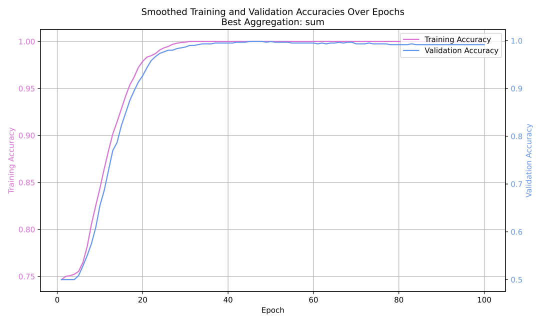
Figure 4: Comparison of Training and Validation Losses and F1 scores for all Aggregation Methods for the Improved Model

1.3.c Evaluating the Best Model

Performance of the best model is shown below in Figure 5, smoothed over 10 different training (and evaluation) runs of 100 epochs each.
The smoothed average was obtained by collecting all the losses and accuracies per epoch for each run in separate lists, and using the *np.mean(..., axis=0)* function on each list. Please refer to the Q1 **.ipynb** file for full and detailed implementation.



(a) Smoothed Training and Validation Loss per epoch of Best Model.



(b) Smoothed Training and Validation Accuracy per epoch of Best Model.

Figure 5: Smoothed Training and Validation Losses and Accuracy for the Best Model across 10 runs

1.3.d Final Analysis and Explanation

The following changes were made to improve model performance:

- **Model Architecture**:
  - Reduced dimension of hidden embeddings from 8 to **5**.
  - Increased the number of GCN layers from 2 to **3** to allow for various abstraction levels and to let the network learn more nuanced embeddings.
  - Changed the graph embedding dimension from 8 to **5**.
- **Epoch Number**: Increased the number of epochs from 10 to **100**.
- **Learning Rate**: Increased the learning rate from 0.001 to **0.0015**.
- **Global Aggregation**: Set the global aggregation method to **sum**, after training and evaluating the model on all methods (Figure 4).

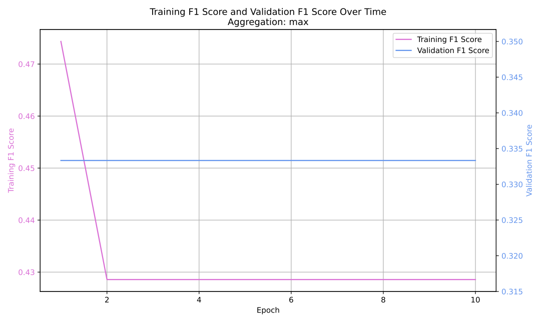
I experimented with a series of hyperparameter combinations by changing the size of the output and hidden dimensions, number of hidden layers, learning rate and aggregation methods, and found that:

- any learning rate less then *0.005* gave similar optimal performances, and settled on *0.0015* as it gave the smoothest and quickest loss decrease and accuracy peak.
- Mean aggregation gave the worst performer by far on any hyperparameter combination.

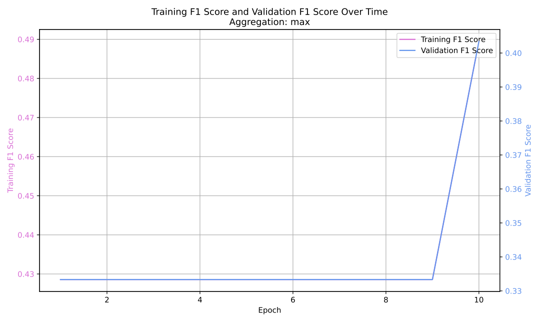
- Max and Sum aggregations gave very similar model performances on the same hyperparameters, however Sum aggregation had overall higher accuracies more consistently.
- More layers slowly led to worse performances (probably due to *oversmoothing*), and similarly so did very high hidden embedding sizes (≥ 15).
- I experimented with different Loss Functions (*Cross Entropy* and *L1 Loss*), but found the best to undoubtedly be the original **Binary Cross Entropy** Loss.
- I explored dataset **normalization**, but that led poorer model performance when compared to the original dataset, and thus discarded this modification.

It is worth mentioning that the new model without any hyperparameter tuning by default achieved a high accuracy, and thus it was not possible to change many details in the model or training pipeline in order to preserve the maximum accuracy.

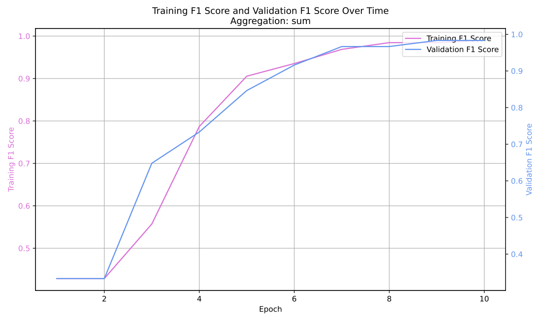
Results of experimentations are shown below in Figure 6.



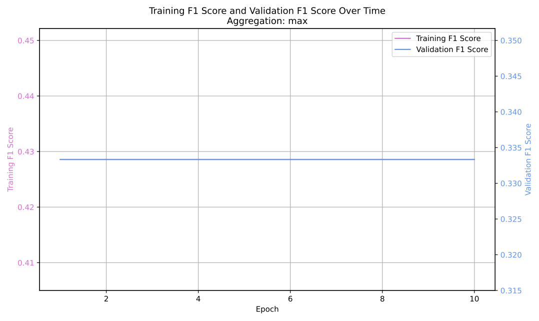
(a) Combo 1: Hidden Dim: 8, Layers: 2, Output Dim: 8, Aggregation: max, LR: 0.001



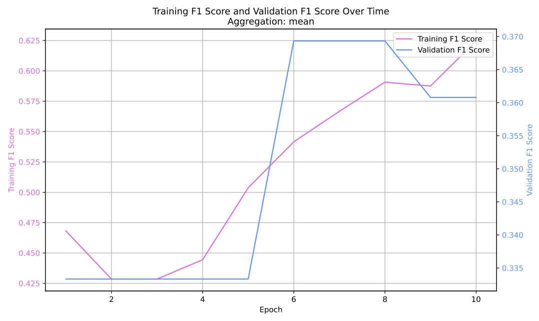
(b) Combo 2: Hidden Dim: 8, Layers: 2, Output Dim: 10, Aggregation: max, LR: 0.001



(c) Combo 3: Hidden Dim: 5, Layers: 3, Output Dim: 5, Aggregation: sum, LR: 0.005



(d) Combo 4: Hidden Dim: 3, Layers: 5, Output Dim: 2, Aggregation: max, LR: 0.0015



(e) Combo 5: Hidden Dim: 8, Layers: 2, Output Dim: 2, Aggregation: mean, LR: 0.01

Figure 6: Comparison of Training and Validation Accuracies for various hyperparameter combinations of the Improved Model

2 Node Classification in a Heterogeneous Graph

2.1 Dataset

2.1.a Problem Challenge

Because the feature dimensions of the nodes differ from each other, it is harder to understand what the determining trait of each node class is: if nodes A and B with different feature dimensions are of the same class, there must be some complex pattern and relationship between features of A and of B that is harder to find compared to nodes of

same-sized features.
Standard node classification relies on gathering information on each set of nodes from the same class to understand what traits they share, and this is not easy when comparing nodes of different natures. Think of it as comparing apples to oranges and trying to understand what classifies them both as fruit, as opposed to comparing red and green apples!

2.1.b Real-World Analogy

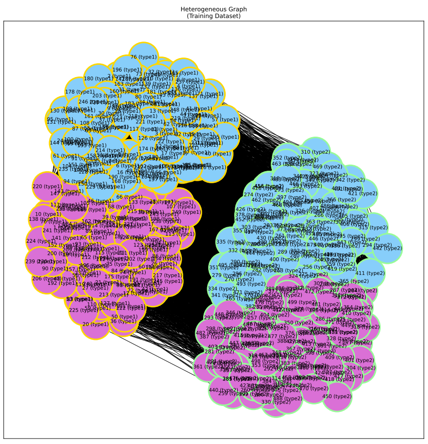
Take as an example a **Social Status Graph**, where each node represents an individual in a Society, and can be of two types based on their Social Status: **R** (Royalty) or **C** (Commoner). The graph shows the relationships between Royal People (Nodes of Class R) and Common People (Nodes of class C) within a (probably ancient) Society.
Now let's assume that each node can also be of different natures, for example based on the individual's biological sex, and have nodes of type **W** (woman), with N features, or **M** (man), with K features.
This scenario creates a dataset similar to the structure of the given one: each node can be of two types ('Type 1' or 'Type 2', like the nodes of nature 'W' and 'M' in the given scenario), and can be of one of two possible classes, Class 0 and Class 1 (like the two classes 'R' and 'C' in the scenario).
It is important here to understand relationships between nodes of different natures and classes in order to understand the underlying patterns that shape such Society.
For example, it can be important to understand the relationship between Royalty and Sex, investigating if nodes of either nature are less or more likely to be of Royal Status rather than Commoner or vice versa, or to understand how and if each Class relates to the other, or if they tend to isolate from each other. Relationships between such heterogeneous nodes are crucial to uncover complex social dynamics and presence of prejudices (like classism or sexism).

2.1.c Interpretation of the Dataset: Plotting the Graph

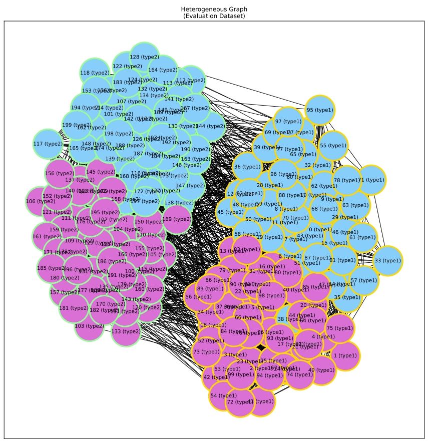
Figure 7 shows the topology of both the Training (Figure 7a,7c) and Validation (Figure 7b,7d) datasets. Graph nodes are differentiated by **class**, indicated by the node colour, and **type** indicated by node outline colour:

- pink nodes = class 0
- blue nodes = class 1
- yellow-contoured nodes = type 1
- lime-contoured nodes = type 2

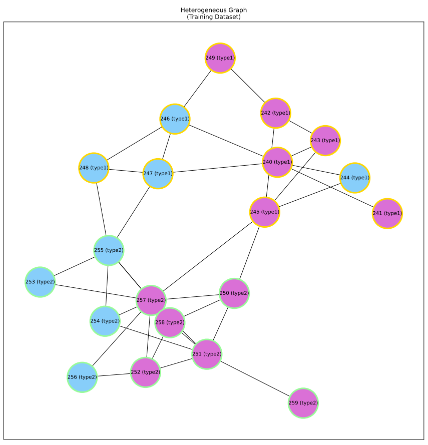
Sub-Graphs (Figure 7c,7d) are shown below in order to better visualize edges between heterogeneous nodes. Please refer to the Q2 **.ipynb** file for detailed implementation.
Both datasets retain similar topology, where the nodes are **clustered** by node *type*, with a clear spatial separation between nodes of type 1 and of type 2. Furthermore, the nodes are also clustered (within each type-cluster) by *class*, with, again, a clear spatial separation between nodes of class 0 (clustered on the bottom-half of each cluster) and of class 1 (clustered on the top-half of each cluster).



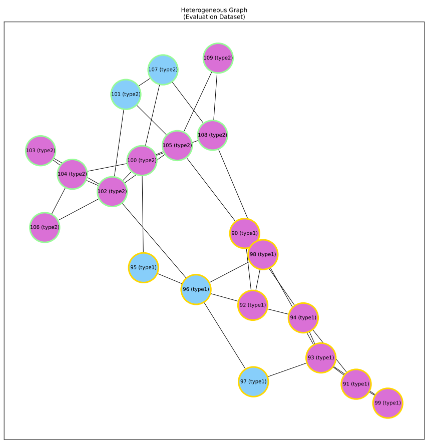
(a) Topology of entire training dataset



(b) Topology of entire Validation dataset



(c) Topology of a subgraph of the training dataset



(d) Topology of subgraph of the Validation dataset

Figure 7: Topology of The Training and Validation Graphs by Node type and class.

Thus there is no mixing of nodes of different type and class, and they tend to stick together to their most similar nodes. However, there are edges between nodes of different natures (as highlighted by the sub-graphs), and thus the nodes are not fully isolated within their own cluster.

The only difference between the two dataset (aside from the graph size) is the positioning of the different clusters, which seem to be mirrored:

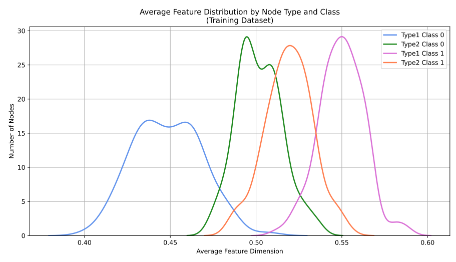
- in the Training Graph:
  - nodes of *type 1* are clustered on the **left**
  - nodes of *type 2* are clustered on the **right**
- in the Validation Graph:

    – nodes of *type 1* are clustered on the **right**
    – nodes of *type 2* are clustered on the **left**

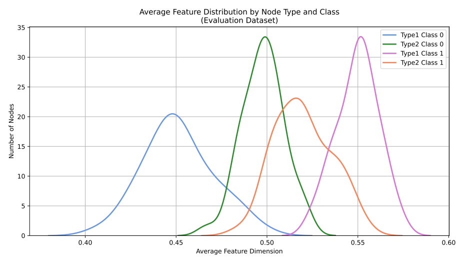
- nodes of *class 0* are clustered on the bottom of each type cluster in both graphs
- nodes of *class 1* are clustered on the top of each type cluster in both graphs

2.1.d Interpretation of the Dataset: Plotting the Node Feature Distributions

Figure 8 shows the distribution of average values of node features based on the nodes' type and class, for both datasets.



(a) Distribution of node features in Training dataset



(b) Distribution of node features in Validation dataset

Figure 8: Distribution of Mean Node Feature Values by node type and class for both Training and Validation Datasets.

Full and detailed implementations can be found in the Q2 **.ipynb** file.

2.1.e Interpretation of the Dataset: Discussion

The feature values are distributed almost identically between the two datasets for the same node types and classes, and all node classes and types have a clear normal distribution of feature values, with different means and (somewhat) similar variances.
However, the values distributions are quite different when comparing nodes of different natures:

- type 2 nodes of either class tend to have tighter values in the range [0.47, 0.57], and have a significant overlap around 0.5
- type 1 nodes of either class tend to have more extreme values, respectively in the lower values (below 0.5 for class 0 nodes) and in the higher values (above 0.5 for class 1 nodes), and have no overlap
- nodes of class 0 tend to have lower feature values (below 0.5), while class 1 nodes have higher feature values (above 0.5)

2.2 Naive Solution: Padding

2.2.a Limitations of Naive Solution

The use of naive padding is a limiting approach which can lead to:

- **Loss of Feature Meaning**: the *presence* alone of each feature for a given node type may have a specific meaning that crucially characterizes that particular node type, and so is the *absence* of such feature. Adding the extra features with a value of 0 to the smaller-sized nodes may change and distort the intrinsic nature of the node, leading to confusion (and lower accuracy) in prediction and classification.

- **Computational and Memory Inefficiency**: adding extra values for smaller-sized inputs leads to an increase in computational overhead, especially when the difference between the feature dimensions becomes large, mainly due to the numerous expensive matrix multiplication operations that characterize GCNs. Furthermore, this extra computation (and extra memory allocation) is '*useless*', as it doesn't carry any information about the node that is being analysed (and may lead to slower loss convergence and training).

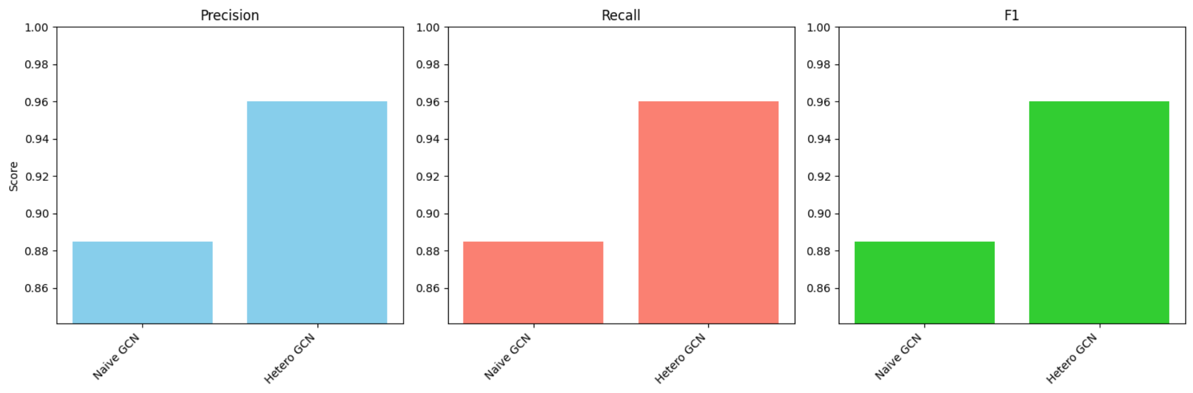
2.3 Node-Type Aware GCN

2.3.a Implementation

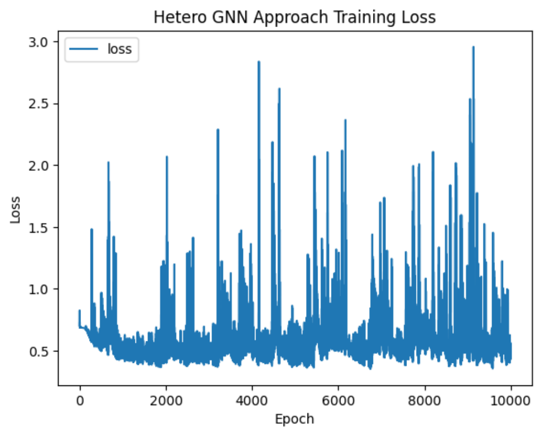
I tested out multiple architectures to solve this problem, by creating two GCN layers that treated multiple node types differently.

- **Separate Weight Matrices**: the first **TypeAwareGCNLayer** was adapted from the basic GCN Layer by accepting, as input dimension, a list of dimensions for each of the node types, then creates two separate weight matrices with the respective input dimensions. In the forward pass, it takes as input a list of two features matrices (and the full graph adjacency matrix) and applies the respective weight matrices to each one, to create embeddings of the same size. The embeddings are then concatenated, locally aggregated using the adjacency matrix and returned (after optional application of non-linearity with Relu Activation). The **HeteroGCN2** Network Model is made up of two *TypeAwareGCNLayer* layers, the first acting as a node-aware layer with separate weight matrices and non linearity, and the second acting like a regular GCN Layer with a uni-dimensional set of features.
  This approach successfully handles different-sized input features without loss of information of padding and utilizes spatial aggregation during separate feature transformation.
  Performance Result of this model is shown in Figure 9.

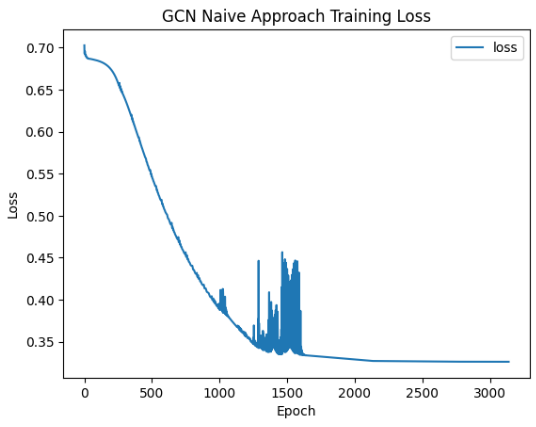
- **Separate Linear Interpolations**: the final **TypeAwareLinearLayer** is a 'composite' Linear Layer. It uses two separate linear layers with different input sizes (corresponding to the two node type feature dimensions) and produces embeddings of the same dimension. In the forward pass, it only takes the two feature matrices as inputs and passes them through the corresponding-sized linear layer, then concatenates the outputs and returns them after adding non-linearity (Relu Activation). The **HeteroGCN** Network Model is made up of one *TypeAwareLinearLayer* layer, followed by two vanilla GCN layers, one with non linearity and the final one without.
  This approach successfully handles different-sized input features without loss of information of padding but doesn't use neighbourhood information during the transformation of different-dimensional features.
  Performance Result of this model is shown in Figure 10.



(a) Comparison of F1 Scores of HeteroGCN2 and Naive GCN



(b) HeteroGCN2 Training Loss



(c) GCN with padding Training Loss

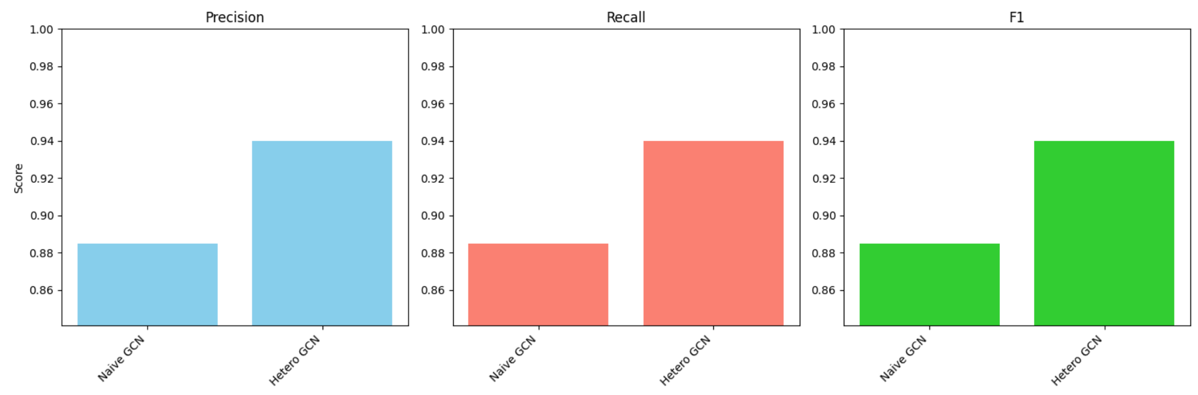
Figure 9: Training Loss and F1 Score comparison of Vanilla GCN with Padding and Implemented Hetero GCN with Node-Aware GCN Layers.

Please refer to the the Q2 **.ipynb** file for full and detailed implementation.

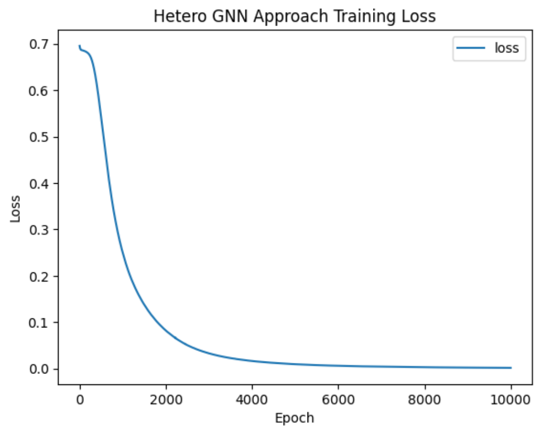
2.3.b Discussion

Both implemented solutions outperformed the Naive Model (Figure 9a,10a).

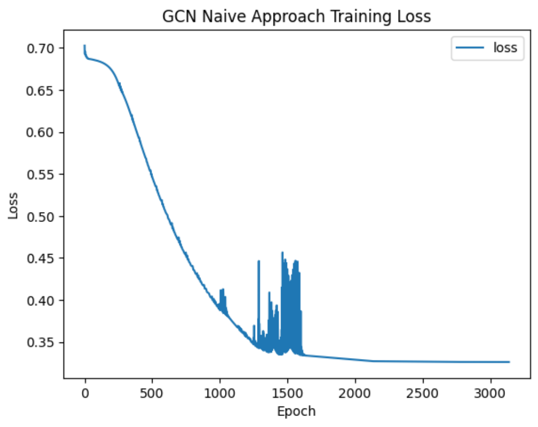
The *TypeAwareGCNLayer* was implemented following the logic of a regular GCN layer, using transformation and aggregation of node features but handling different-sized inputs by using separate weight matrices of the corresponding dimensions. This introduced an extra level of complexity and increased the computational and memory requirements, as the double weight matrices introduces extra learnable model parameters and additional matrix multiplications.
The best version of this model was optimized for the given default seed, with high learning rate (of *0.05*) and high patience (equal to the number of epochs) and hidden dimension (of *10*). This hyperparameter combination and architecture gave best final accuracy of 0.96 (Figure 9a).
This model, however, produced very variable results in between separate runs and when setting different seeds, with accuracy varying from 0.6 to 1.0 with different seeds. Furthermore, the model showed unsatisfactory loss convergence, ending with over 0.5 loss and very high variance during training (Figure 9b). Furthermore, it doesn't address the



(a) Comparison of F1 Scores of Final HeteroGCN and Naive GCN



(b) HeteroGCN Training Loss



(c) GCN with padding Training Loss

Figure 10: Training Loss and F1 Score comparison of Vanilla GCN with Padding and Implemented Heterogeneous GCN with Linear Node-Aware Layer.

increased complexity and memory requirements that the Naive Model has.
Due to these faults, this model was scrapped (but training and implementation is available in the Q2 **.ipynb** file).

The *TypeAwareLinearLayer* was implemented with the idea that a linear layer by definition is able to transform an input from one dimension to another, all while extracting important information about the input features with learnable parameters that aid the model learning process. Creating a ad-hoc layer with two separate linear layers for each node type allows the model to learn type-specific information about each node, analysing them separately and then using the following GCN layers to learn the relationship of said nodes in relation to the entire graph. This architecture, as opposed to the first one, doesn't account for neighbourhood aggregation during the transformation of each type-specific feature vector, but treats them separately first.
The final **HeteroGCN** model using such layer produced very consistent results even with different seeds. The model achieved a very smooth loss convergence to 0 (Figure 10a) and achieved over 0.94 accuracy in every training and validation run (Figure 10a). The model was trained on a learning rate of *0.001*, same hidden size as the HeteroGCN2 model and lower patience (of *1000*). During training it showed steady and continuous loss reduction and accuracy increase (Figure 10b).

Due to its superior performance, this model was chosen as the final one (training and implementation is available in the Q2 **.ipynb** file).

For both models, the hyperparameter tuning process was a set of trial-and-error stages, trying out different combinations of layer size and number and learning rates. At each step, the changes that positively impacted the performance were retained, while the other parameters were changed and tested again.

2.4 Exploring Attention

2.4.a Implementation

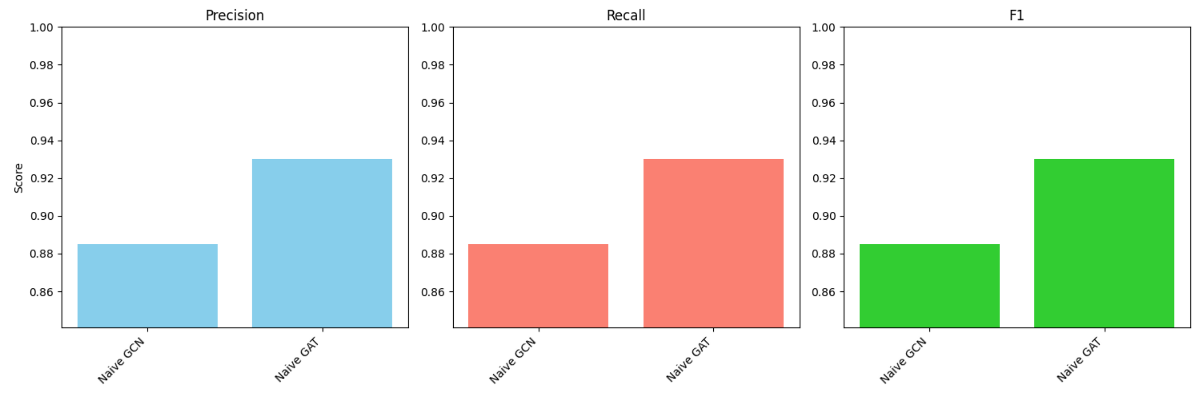
The final **GraphAttentionNetwork** Model is made up of two *AttentionBasedGCNLayer* layers, with the first having non-linearity in the output. The *AttentionBasedGCNLayer* was implemented with the attention mechanism descibed in the next section. The idea is to aggregate node embeddings based on which other graph nodes are the most similar to the current one instead of aggregating over all node neighbours. Performance Result of this model is shown in Figure 11. The optimized model hyperparameters and the full implementation details are available in the Q2 **.ipynb** file.

2.4.b Discussion

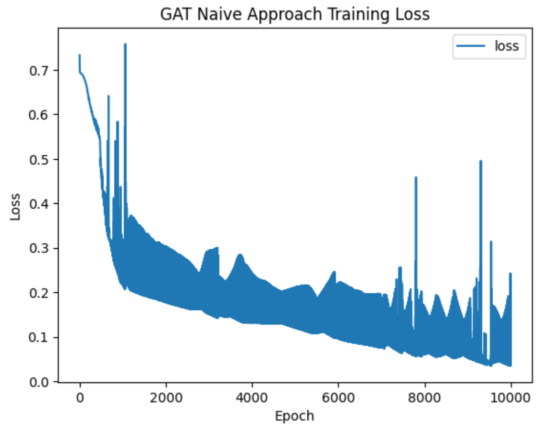
The **AttentionBasedGCNLayer** layer was implemented on the basis of the vanilla GCN layer. It takes the same parameters as a regular GCN layer with the addition of a new learnable set of parameters, **Phi** (Φ), for the similarity matrix computation. Furthermore, during the forward pass, instead of aggregating locally based on the adjacency matrix, it performs aggregation based on a **Attention Matrix** which is re-calculated at every layer and at every forward pass by the (implemented) *compute_attention_matrix* layer function using the adjacency matrix and the last embedding layer to calculate a **Similarity Matrix** by following the process described in the lectures [4]:

1. Reshape every node embedding from (NxD) to (NxNxD) to create a square matrix
2. Stack every pair of (NxNxD) embedding together (by concatenating the original feature matrix with its transpose) to create a matrix of shape (NxNx2xD)
3. Reshape the stacked matrix into a (NxNx2D) final matrix, which represents the feature matrix of stacked node-pair embeddings for all graph nodes
    - The above process is equivalent to nested looping through all the entries of the feature matrix and manually stacking all node-pair embeddings, but it is much faster (i.e. doesn't have O(N^2) complexity)
4. Multiply the final stacked matrix by the learnable Φ to compute similarity scores for each node pair
5. Apply the **SoftMask** to the similarity matrix using the Adjacency matrix with self loops

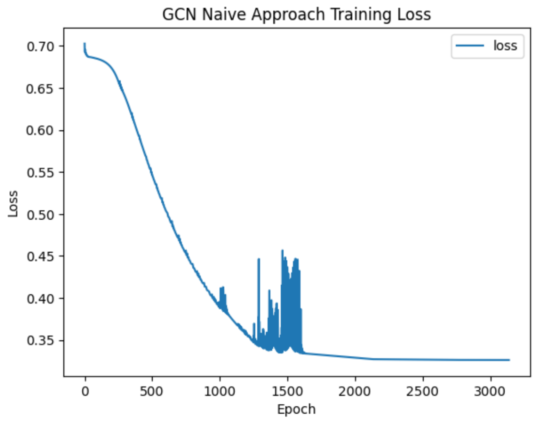
The final **GraphAttentionNetwork** model using such layers achieved a final accuracy of over 0.93 (Figure 11a), but produced very inconsistent results ($\pm$0.4 F1) with



(a) Comparison of F1 Scores of Attention-based GCN and Naive GCN with Padding



(b) Implemented Attention-Based GCN Training Loss



(c) GCN with padding Training Loss

Figure 11: Training Loss and F1 Score comparison of Vanilla GCN with Padding and the Attention-based GCN.

different seeds. Furthermore, although the model achieved a good loss convergence to 0, it showed very high variance and unstable training (Figure 10b). As for the previous models, the hyperparameter tuning process was a set of trial-and-error stages, trying out different combinations of layer size and number and learning rates. At each step, the changes that positively impacted the performance were retained, while the other parameters were changed and tested again. The final model was trained on a learning rate of *0.005*, smaller hidden size compared to the vanilla GCN (of *5*) and lower patience (of *1000*).

2.5 Overall Discussion

The **Node-Type Aware CGN** is the best performing model out of all the implemented ones. Not only does it achieve a smoother loss convergence to 0 (Figure 10b, 10c, 11b), but it also reaches the highest F1 Score (Figure 12) with most stability and almost no variance between different seeds and runs.
Since the Node-Type aware GCN handles nodes of different natures individually first, it can better capture type-specific characteristics, creating more meaningful and informative embeddings for each node-type. This contrasts the padding approach of the naive GCN,

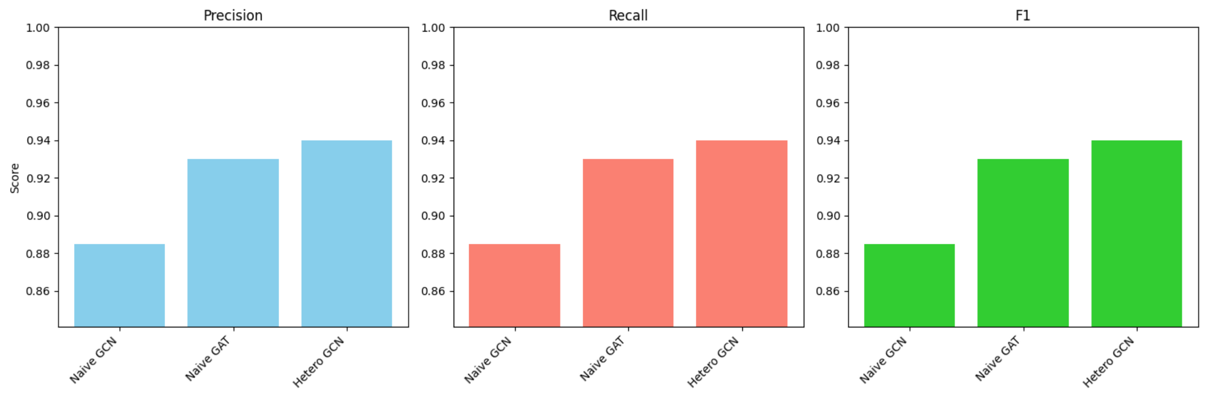


Figure 12: Precision, Recall and F1 Score comparison of all three models

which results in loss of node-specific information. The heterogeneity in the embeddings of the Hetero GCN more accurately reflects the heterogeneity of the graph itself, leading to an enhanced graph classification ability.

The **Attention-Based CGN** is the second-best model (Figure 12), outperforming the Naive GCN with Padding. The naive GCN, in fact, treats all neighbours equally during the aggregation phase, potentially ignoring the distinct roles that different node-types may play. The attention-based aggregation mechanism, instead, allows the model to create more meaningful embeddings by learning how to weight each neighbour differently based on their importance with respect to the current node. This allows the node embeddings to account for the heterogeneity of the node-types, creating more informative embeddings that, again, better reflect the real graph structure.

3 Investigating Topology in Node-Based Classification Using GNNs

3.1 Analyzing the Graphs

3.1.a Topological and Geometric Measures

The *Topological* Measures help define the role of each node in the graph. They allow the identification of which nodes are the most important (based on how much information flows through them) and most vulnerable to attacks:

- **Node Degree**: indicates the number of edges connected to a given node [2]. It essentially represents how 'popular' a node is and it helps in defining which nodes are the most connected and highly important (as a lot of information flows through them) in the graph.
- **Betweenness Centrality**: captures the 'importance' of nodes in a graph. The 'centrality' of a node in a graph is given by the number of shortest paths between node pairs in the graph that that node lies on. That is, how many of the shortest paths in the graph pass through that node, identifying the so called 'bridge-spanning' nodes [3].

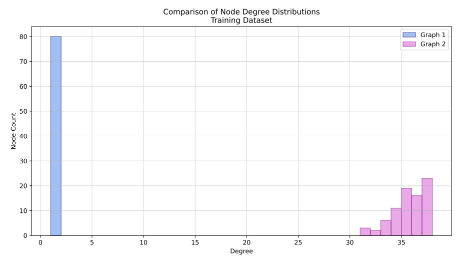
The *Geometric* Measures help to understand the topological (and geometrical) structure and behaviour of the graph as a whole:

- **Oliver-Ricci Curvature** (ORC): captures the 'curvature' of graphs and the shortest path characteristics of graph edges by using the idea of 'transport distance' between probability distributions of random walks between neighbouring nodes [1]. ORC helps understand how much 'work' is needed in order to flow information between neighbouring nodes. It helps understand the structural weaknesses ans strengths of the graph.

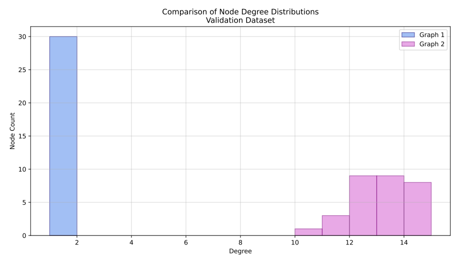
3.1.b Visualizing and Comparing Topological and Geometric Measures of Two Graphs

Visualizations of the topological measures described above for both the Training and Validation Graphs are shown in Figure 13. In general, Graph 1 and Graph 2 show very important topological and geometric differences, but each graph's characteristics are preserved between the two datasets.
For both datasets, Graph 1 contains all nodes with 0-2 edges each, while Graph 2 contains mostly nodes with a much higher number of edges (between 30 and 35 for the Training Graph, and 10 to 15 for the Validation Graph) (Figure 13a,13b).
Similarly, in both datasets Graph 1 nodes have an equal distribution of 'betweenness', meaning that most nodes lie on the graph's shortest paths and i.e. are likely to be highly involved in the flow of information (under the shortest walk assumption). On the other hand, the vast majority of nodes in Graph 2 have very low (0) betweenness centrality, and only few nodes have (slightly) higher betweenness values (Figure 13c,13d)), identifying few very central graph nodes.

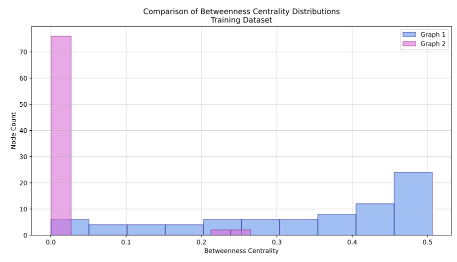
In both graphs, the ORC is high (close to 1) for most of the graph edges (Figure 13e,13f), meaning that both graph have some sort of structural regularity: this means that most graph nodes have similar connections and that these nodes are located in areas of the graph with tightly and similarly connected neighbours. There are few nodes in both Graphs with low ORC, meaning that there are some graph areas with poorly connected nodes. At the same time, there is a visible difference in edge counts between the two graphs: in Graph 2, there is a much higher number of edges, with the grand majority of them having high ORC, while in Graph 2, there is a much smaller number of total edges, again mostly with high ORC. This means that Graph 2 has many more connections between nodes, signalling a very different topological structure between the two graphs.



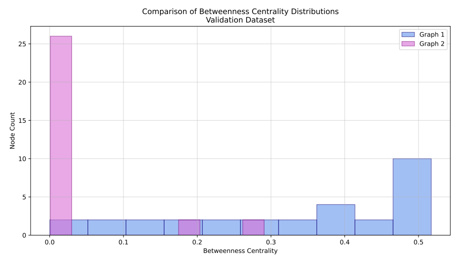
(a) Node Degree Distribution of Training Dataset



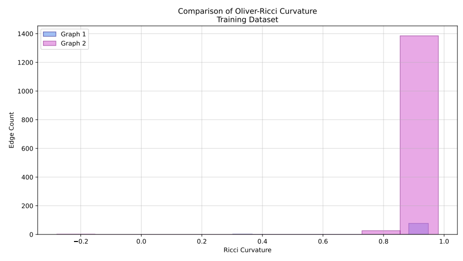
(b) Node Degree Distribution of Validation Dataset



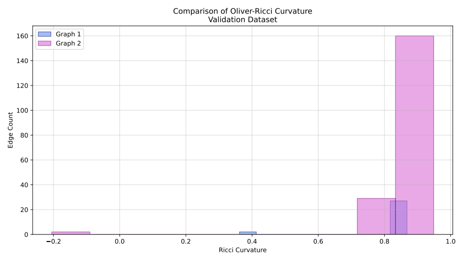
(c) Betweenness Centrality Distribution of Training Dataset



(d) Betweenness Centrality Distribution of Validation Dataset



(e) Ricci Curvature Distribution of Training Dataset



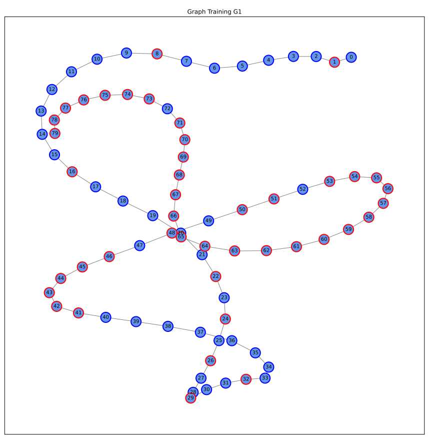
(f) Ricci Curvature Distribution of Validation Dataset

Figure 13: Comparison of Topological Measures of the Training and Validation Graphs.

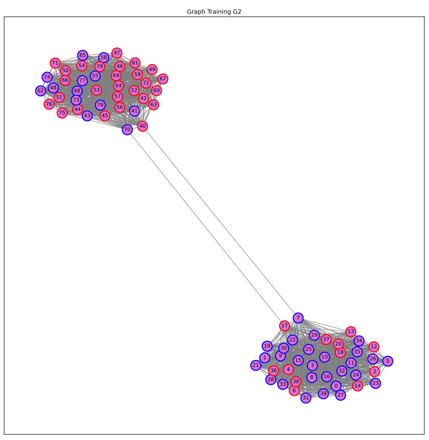
3.1.c Visualizing the Graphs

The *plot_graph* function was created using the *networkx* python library, by creating the graph from the adjacency matrix and using the *draw_networkx_nodes, draw_networkx_nodes* and *draw_networkx_labels* functions to draw out the created graph. An option to plot the graph by differentiating between nodes of different classes (with different edge colors) is given based on a parameter which optionally takes the labels of the graph. Please Refer to the Q3 **.ipynb** file for full and detailed implementations.

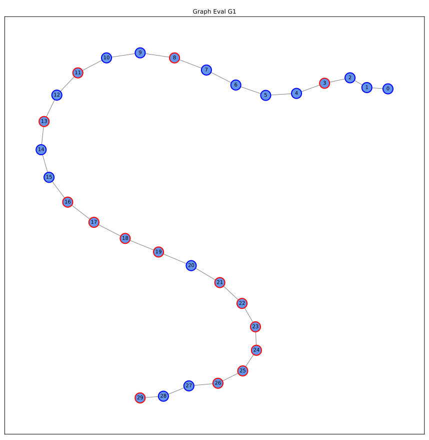
Graph topologies are shown in Figure 14. Graph 1 and Graph 2 retain their own structure in both datasets. Graph 1 has a **chain-like** structure, where every node is connected only to the previous and next node in the chain. Graph 2 has a **modular** structure, with two main clusters connected to each other via two edges and four nodes (two per cluster). This correctly reflects the values of the topological and geometric measures described in the above Section.



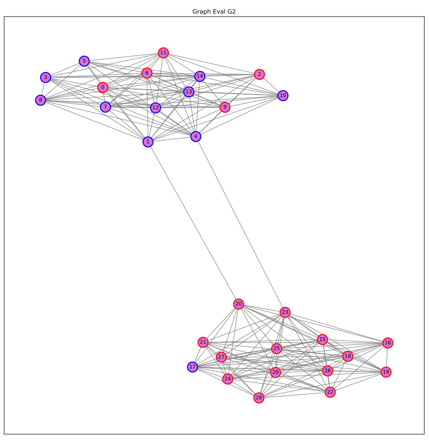
(a) Type 1 Graph of Training Dataset



(b) Type 2 Graph of Training Dataset



(c) Type 1 Graph of Validation Dataset



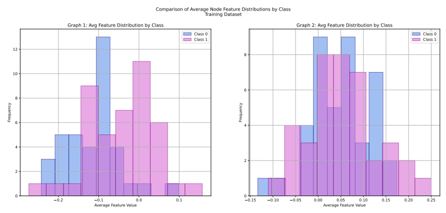
(d) Type 2 Graph of Validation Dataset

Figure 14: Comparison of Topologies of the Training and Validation Graphs, with distinction between nodes of class 0 (blue egde) and class 1 (red edge).

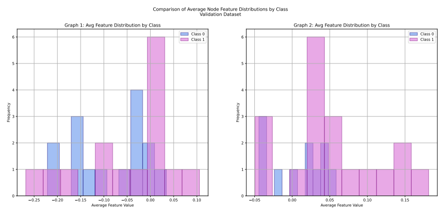
Graph 2 shows a much more efficient structure, thanks to the higher average node degree and higher number of high ORC edges. This means that it is easier for information to flow between spatially distant nodes. Graph 1, on the other hand, has a very poor information flow ability, as it requires traversing most of the chain in order for information to flow from one end to the other.

3.1.d Visualizing Node Feature Distributions

The *plot_node_feature_dist_by_class* function was used, which calculates the distribution of average node feature of all nodes in a given graph. The plots of the distribution of features based on their node class for each dataset is shown in Figure 15. Please Refer to the Q3 **.ipynb** file for full and detailed implementations.



(a) Feature Distribution of Graph 1 and Graph 2 of Training Dataset



(b) Feature Distribution of Graph 1 and Graph 2 of Validation Dataset

Figure 15: Comparison of Feature Distribution of Graph 1 and Graph 2 of each Dataset.

The feature distribution is quite different across graphs and datasets. In the Training Dataset, the features of both node classes have a normal distribution:

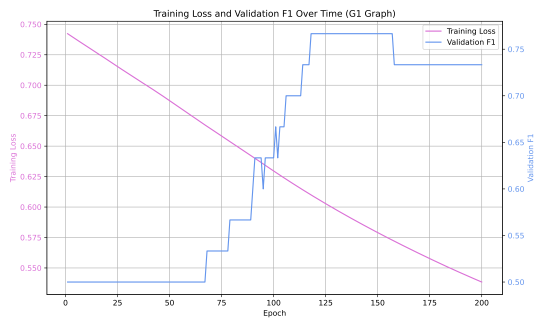
- In Graph 1 the two classes have visibly different means and similar variances. Class 0 nodes tend to have lower (negative) average features, while class 1 nodes tend to have higher average values.
- In Graph 2 the distribution of feature values for both classes has similar (almost same) mean and variance, with the distributions almost overlapping completely.
- There is a visible overlap between the average features of the nodes of both classes in both graphs. It is much more emphasized in Graph 2 (see point above) while it is smaller in Graph 1 (where only the tails of the two distributions overlap).

In the Validation Dataset, the features of both node classes have a less distinctive distribution, as most nodes (of both classes in both Graphs) seem to have a unique average value (with no distinctive average value among most nodes of any class). This is, however, also due to the much smaller number of nodes of the Validation dataset (compared to the traiing one).

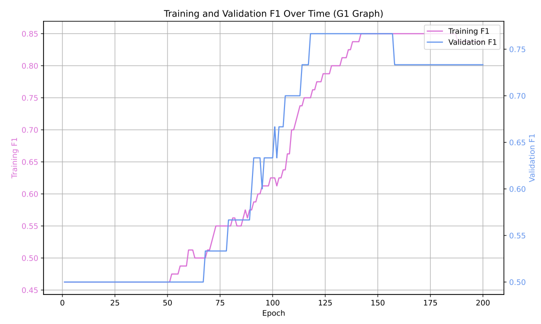
3.2 Evaluating GCN Performance on Different Graph Structures

3.2.a Implementation of Layered GCN

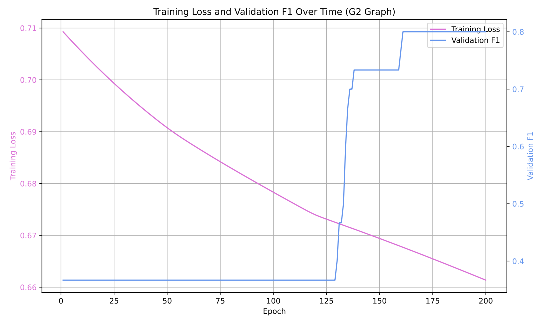
A new class *LayeredGraphNeuralNetwork* was created, which takes as input parameters the number of layers desired for the Network. This allows dynamic creation of different-sized Networks with arbitrary numbers of layers. In the forward pass, it stores the embeddings of each layer in a list and (optionally) returns them. Please Refer to the Q3 **.ipynb** file for full and detailed implementations.
Figure 16 shows the obtained training and validation losses and F1 scores for a custom LayeredGraphNeuralNetwork with 2 layers.



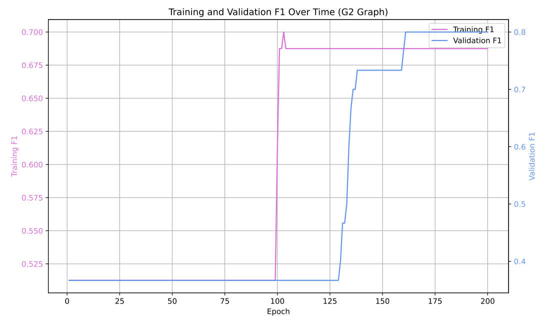
(a) Training Loss and Validation Accuracy on G1 Graph



(b) Training and Validation F1 Scores on G1 Graph



(c) Training Loss and Validation Accuracy on G2 Graph



(d) Training and Validation F1 Scores on G2 Graph

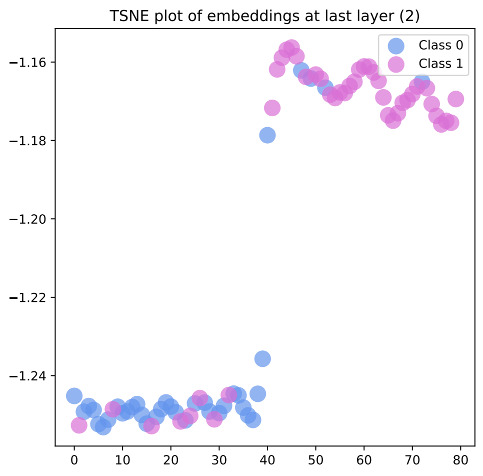
Figure 16: Comparison of Training Loss and F1 Scores of model trained separately on the two graphs.

3.2.b Plotting of t-SNE Embeddings

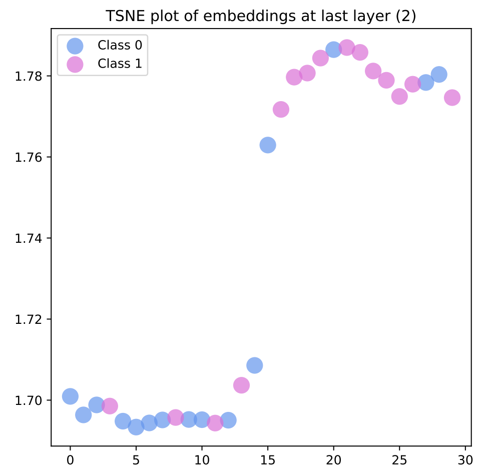
A new funtion *plot_tsne* was created, which takes as input parameters the list of leant embeddings and plots them using the *sklearn.manifold TSNE* function, distinguishing between nodes of different classes. Learned embeddings from the aforementioned model in Section3.2.a are shown in Figure 17. Please refer to the Q3 **.ipynb** file for full and detailed implementations.

3.2.c Training the Model on Merged Graphs $G_1 \cup G_2$

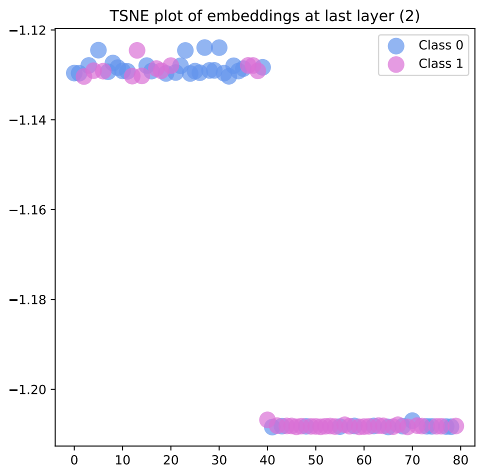
To train the model on both graphs I implemented a new training function, *train_G1UG2*, which, before the training starts merges both the training and validation graphs using an implemented *merge_graphs* function. This function patches the adjacency matrix using the provided *patch_A_for_dataset* function and creates a merged Adjacency matrix with the original A matrix of G1 on the top-left quadrant and the original A matrix of G2 on the bottom-right quadrant, and zeroes everywhere else. It then concatenates the features and the labels of G1 and G2. It returns a merged graph in the form of a tuple (X,A,y), emulating the format of the original datasets. The model is then trained on the merged training graph and evaluated on the merged validation graph. Please refer to the Q3 **.ipynb** file for full and detailed implementations.
Training and Validation over epochs and t-SNE of embeddings are shown in Figure 18.



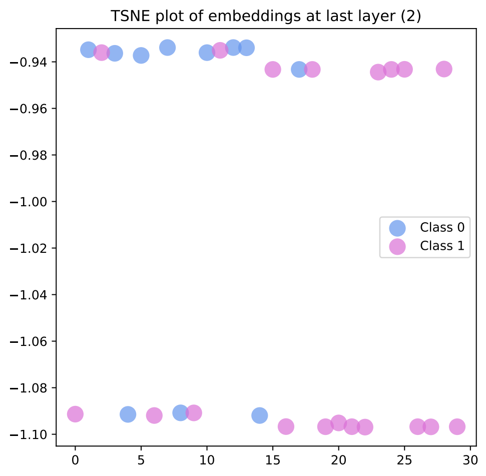
(a) t-SNE of Last Layer Embeddings of Training G1 Graph



(b) t-SNE of Last Layer Embeddings of Validation G1 Graph



(c) t-SNE of Last Layer Embeddings of Training G2 Graph

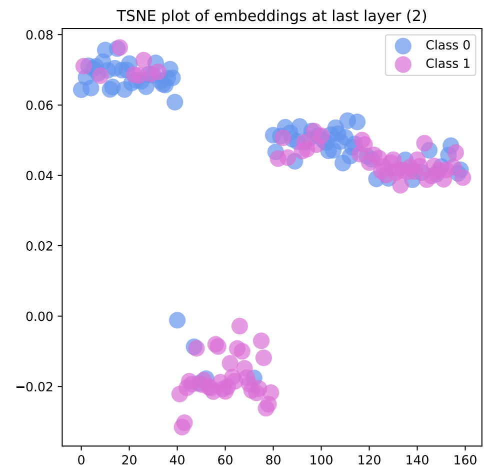


(d) t-SNE of Last Layer Embeddings of Validation G2 Graph

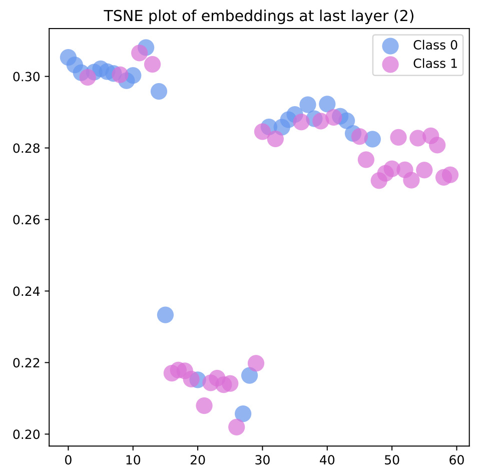
Figure 17: Comparison of t-SNE of learned last layer embeddings on model trained and evaluated separately on G1 and G2.

3.2.d Joined vs. Independent Training

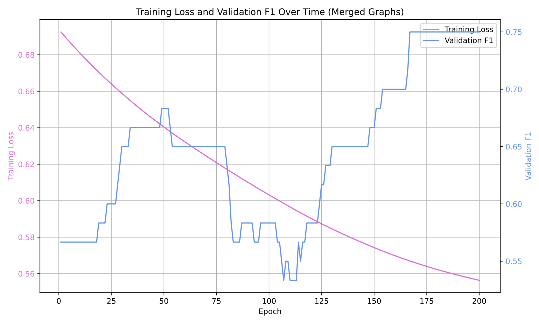
There was surprisingly not much difference between the accuracies obtained on the joined and independent training (Figure 16b,16d,18d), with the model achieving a score higher than 0.75 in both. However in the independent training there is a slightly higher accuracy (reaching 0.8) and there is a clear difference in the rate of growth of the accuracy between the two graphs and higher final loss, with G1 showing a similar and rapid accuracy increase in the training and validation graphs, while G2 shows a much less steady and less smooth curve and a higher final loss (Figure 18).
Furthermore, the learnt embeddings show a much different picture when comparing individual vs separate training. With independent training , the model learns much better to distinguish between different class nodes, with them having very different final embed-



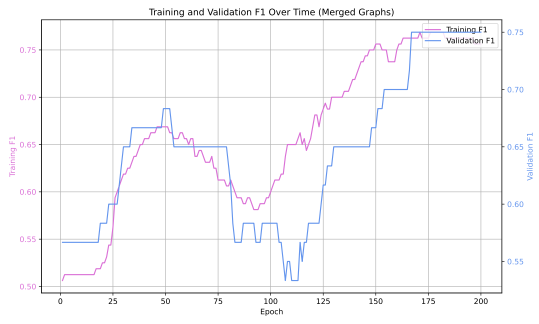
(a) t-SNE of Last Layer Embeddings of Training Merged Graph



(b) t-SNE of Last Layer Embeddings of Validation Merged Graph



(c) Training Loss and Validation F1 Score of model on Merged Graph



(d) Training and Validation F1 Score of model on Merged Graph

Figure 18: Comparison of Loss, F1 Scores and t-SNE of learned last layer embeddings on model trained and evaluated on the merged graphs.

dings (Figure 17a-d), while with joined training there is much more confusion between the two node classes (Figure 18a,18b), as the model shows clear difficulties in clustering nodes of different classes apart.

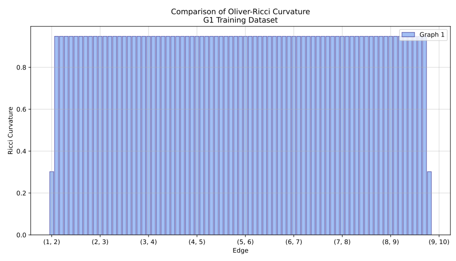
Independent training may have reduced model confusion, by allowing the model to see each graph features independently instead of all together. With joined training, however, the model may better understand the context between the two graphs, learning from a unified set of features how they interact with each other. However, in this case the model performance is directly determined by the merging strategy used. Here, since the graphs were 'patched' together, no checks were implemented to find shared nodes between the graphs, which may have led to the slightly poorer performance given the 'padding' of the joint adjacency matrix.

An alternative approach would be to instead implement a custom model that takes both graphs as input during the training, by first performing local aggregation on each graph separately, then using (for example) Linear layers to jointly transform the embeddings into a single embedding, before feeding them into the model's GCN layers.

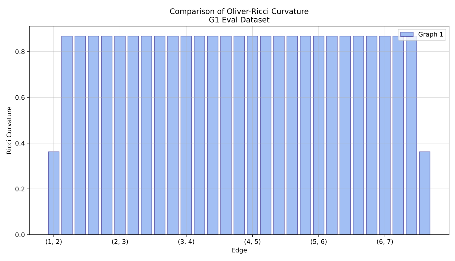
3.3 Topological Changes to Improve Training

3.3.a Plot the Ricci Curvature for Each Edge

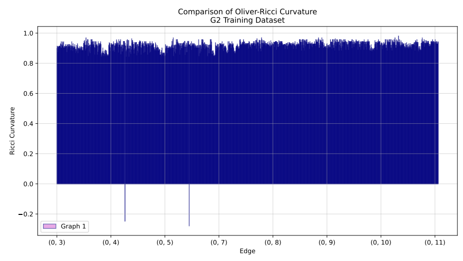
Plots on the Oliver-Ricci Curvature (ORC) distribution of all edges of all the Graphs are shown in Figure 19. Please refer to the Q3 **.ipynb** file for full and detailed implementations.



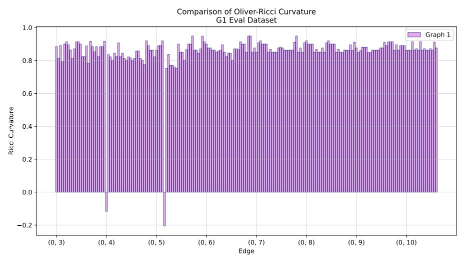
(a) ORC per edge in G1 Training Graph



(b) ORC per edge in G1 Validation Graph



(c) ORC per edge in G2 Training Graph



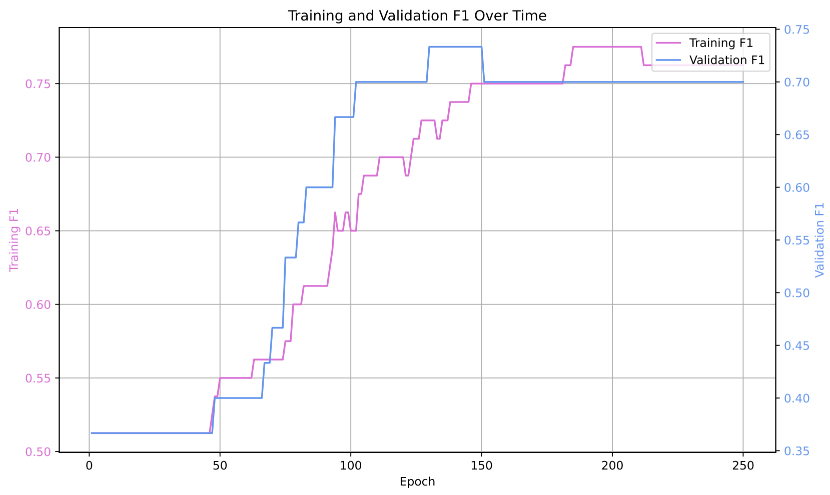
(d) ORC per edge in G2 Validation Graph

Figure 19: Comparison of Oliver-Ricci Curvature (ORC) distribution on edges of the Training and Validation G1 and G2 Graphs

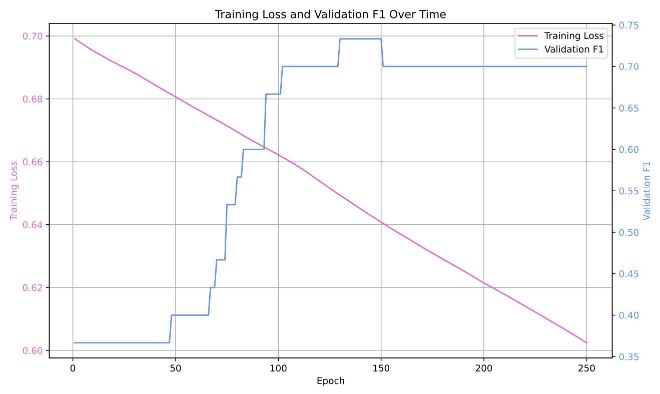
3.3.b Investigate Extreme Case Topologies

To induce the GCN to behave like a regular MLP, any influence of the adjacency matrix should be removed during the forward pass, i.e. nullifying the aggregation process. This can be done by, for example, setting the adjacency matrix to be the identity matrix, allowing only self-connections for every node (Figure 20).

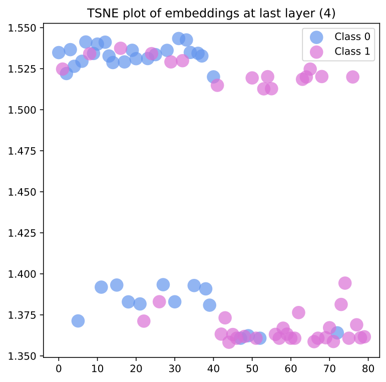
On the other hand, if labels were available both during training and testing, a graph with connections between (non-self) node pairs of the same class would have the structure for optimal training. This is because such structure allows optimal information flow between any two nodes of the same class in the graph, with shortest path length of 1 available between any node pair of the same class (Figure 21), allowing the model to differentiate between nodes of each class optimally and create characteristic embeddings for each. As expected, the graph with optimal structure quickly reaches 100% accuracy and zero loss (Figure 21a,21b) and successfully clusters nodes by class (Figure 21c).



(a) Training and Validation F1 Scores with No-Effect Changes



(b) Training Loss and Validation Accuracy with No-Effect Changes

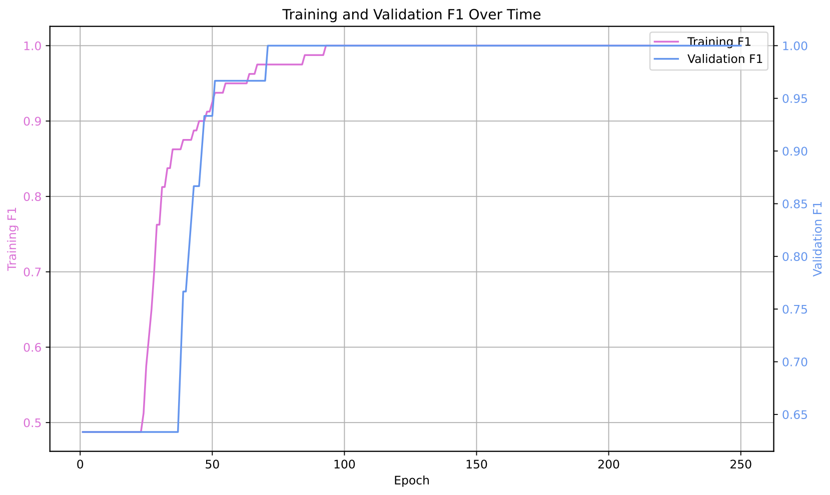


(c) Final Learned embeddings with No-Effect Changes (Training Dataset)

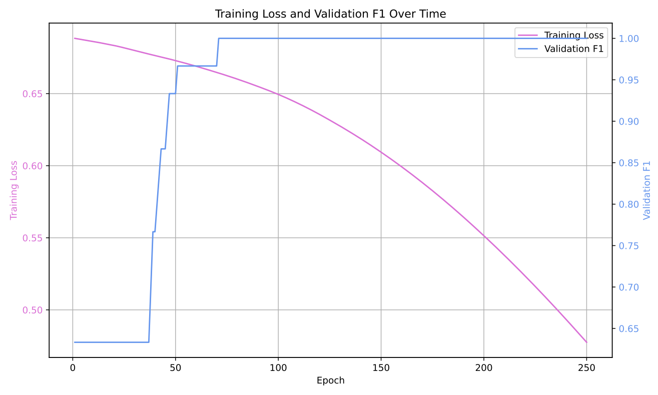
Figure 20: Comparison of F1 Scores, Loss and Leaned embeddings on model trained with No-Effect Changed Graph

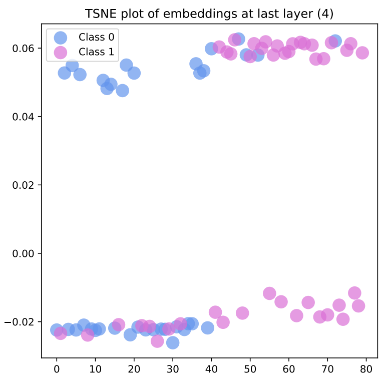
3.3.c Improving Graph Topology for Better Learning

A first strategy would be to change the edge weight between different node pairs based on their *similarity*. This can help the GCN more effectively learn patterns within local node neighbourhoods that are similar to each other and better differentiate between nodes with different characteristics (similar to an attention-based mechanism). Alternatively, *edge dropout* could be explored in order to enhance generalization capabilities of the model and reduce variance between testing and training.



(a) Training and Validation F1 Scores with Optimal Structural Changes





(b) Training Loss and Validation Accuracy with Optimal Structural Changes

(c) Final Learned embeddings with Optimal Structural Changes (Training Dataset)

Figure 21: Comparison of F1 Scores, Loss and Leaned embeddings on model trained with Optimally-Structured Graph

References

[1] Corinna Coupette, Sebastian Dalleiger, and Bastian Rieck. Ollivier-ricci curvature for hypergraphs: A unified framework, 2023.

[2] Jennifer Golbeck. Chapter 21 - analyzing networks. In Jennifer Golbeck, editor, *Introduction to Social Media Investigation*, pages 221–235. Syngress, Boston, 2015.

[3] Derek L. Hansen, Ben Shneiderman, Marc A. Smith, and Itai Himelboim. Chapter 6 - calculating and visualizing network metrics. In Derek L. Hansen, Ben Shneiderman, Marc A. Smith, and Itai Himelboim, editors, *Analyzing Social Media Networks with NodeXL (Second Edition)*, pages 79–94. Morgan Kaufmann, second edition edition, 2020.

[4] BASIRA Lab. Deep graph-based learning - 3.5 global and local aggregation methods. https://www.youtube.com/watch?v=zRmzVkidkqA. [Accessed 19-02-2025].