DGL 2025 Coursework 1

Viyan Raj

CID: 02025734 vr121@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

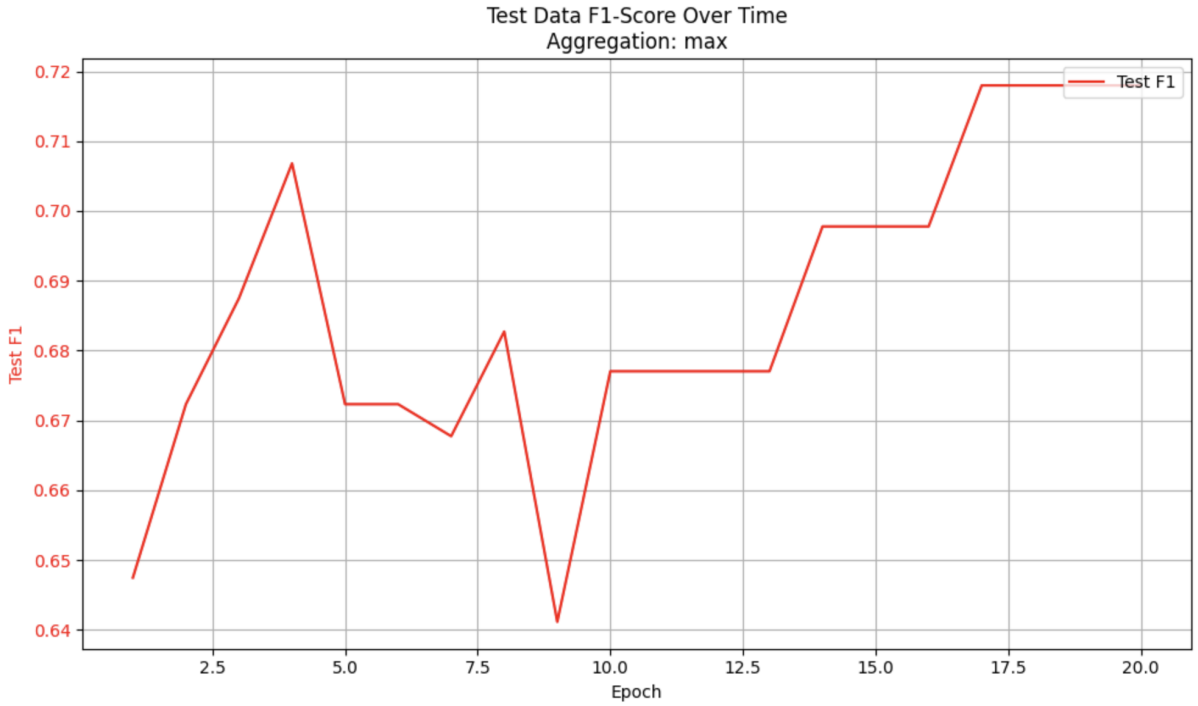
My implementations for the graph-level aggregation methods in the GCN can be found in the Q1.1.a cell. The model class is called MyGraphNeuralNetwork.

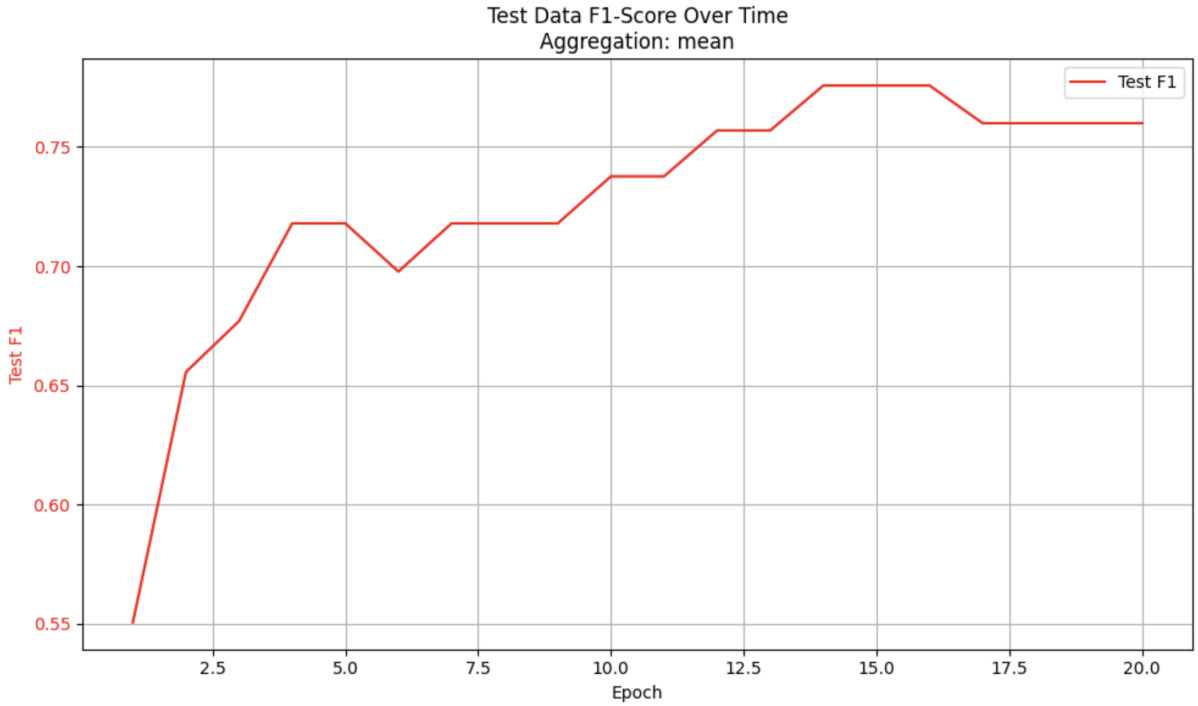
1.1.b Graph-Level Training

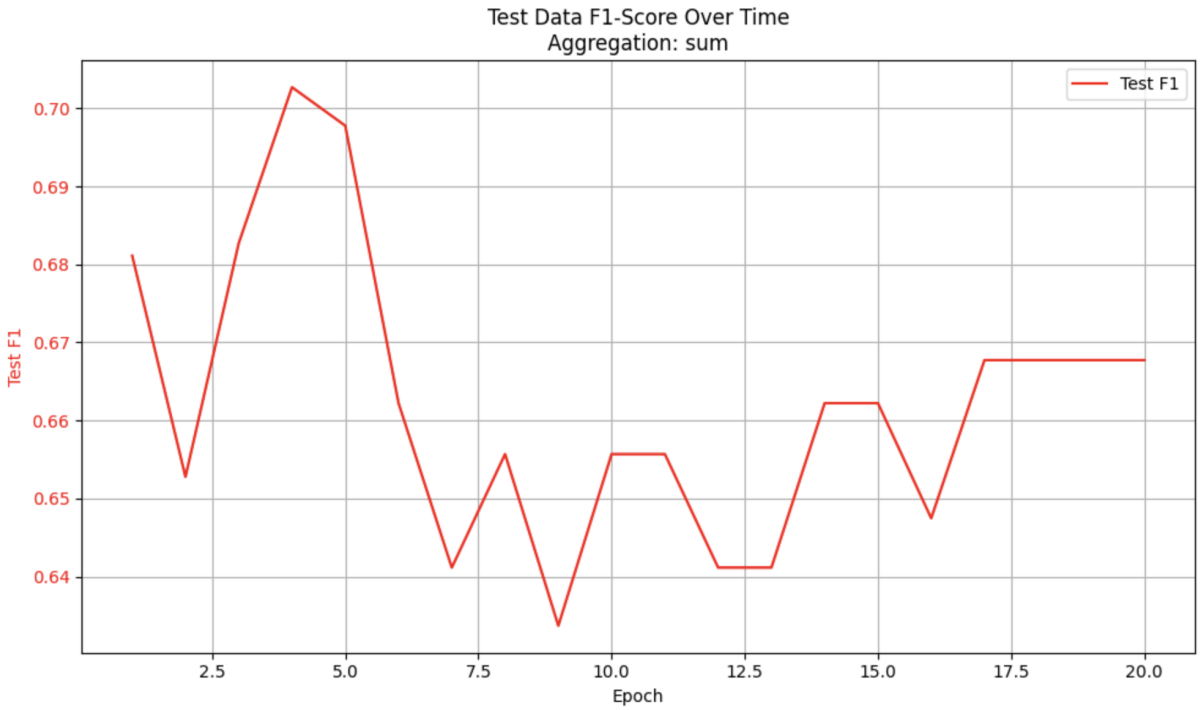
The solution is within the Q1.1.b cell. The train_model function uses train_epoch to train the GCN over a number of epochs, displaying the training loss, training accuracy and validation accuracy at each epoch. This log can be found in the notebook. The function returns 5 arrays: train_losses, train_accs, val_accs, train_f1s, val_f1s.

The 'plot_training_and_validation' function was altered to become more generic, allowing plotting of either one or two lines, on any metric (e.g. F1, accuracy etc.)

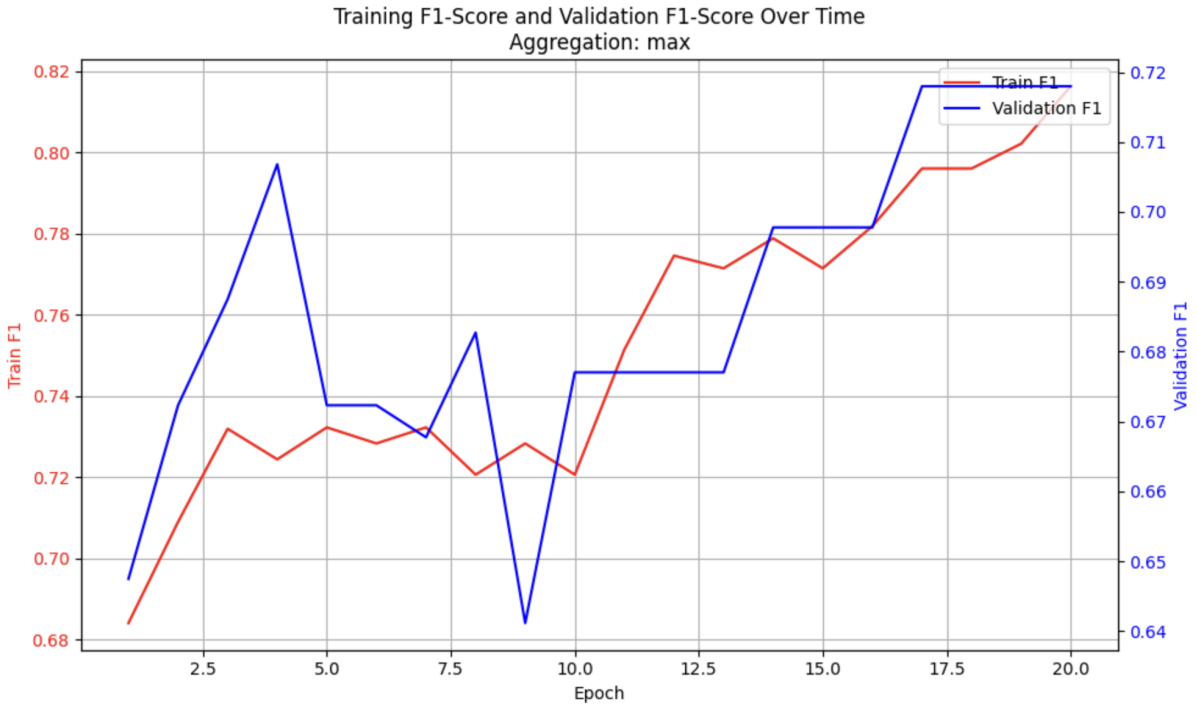
Combining these changes allows plotting of test F1 for all three aggregation methods, shown below.



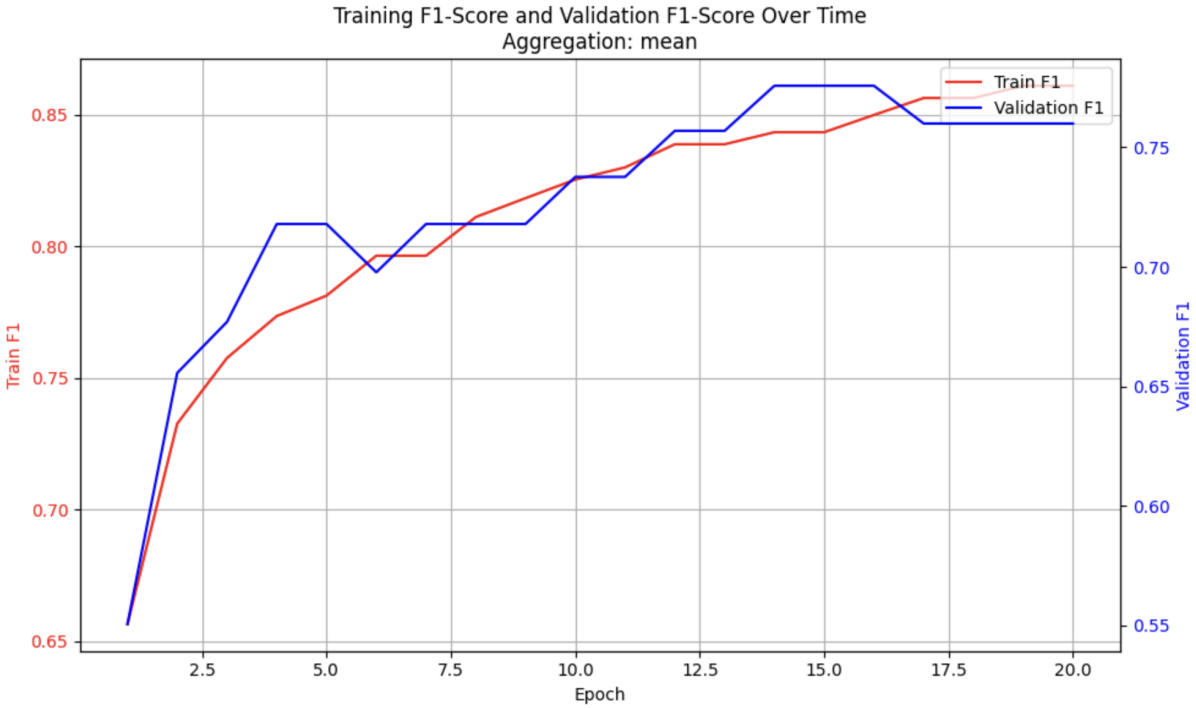

Test Data F1-Score Over Time
Aggregation: mean
Test F1
0.75
0.70
0.65
0.60
0.55
Test F1
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epoch


Test Data F1-Score Over Time
Aggregation: sum
Test F1
0.70
0.69
0.68
0.67
0.66
0.65
0.64
Test F1
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epoch

1.1.c Training vs. Evaluation F1



With max aggregation, the training F1 gradually increases, reaching 0.82 by epoch 20. The evaluation F1 fluctuates significantly at the start (indicating instability), and ends around 0.72. There is an observable gap between training and evaluation F1 scores, suggesting potential overfitting. Although the final evaluation F1 is good, the variance during training implies the model struggles with generalization when using max aggregation.



With mean aggregation, the training F1 steadily rises to 0.85 by epoch 20. The evaluation F1 increases consistently to 0.76, closely tracking the training F1. The small gap between training and evaluation F1 curves suggests less overfitting and good generalization. The convergence pattern is smooth and stable, showing that mean aggregation provides consistent learning signals.

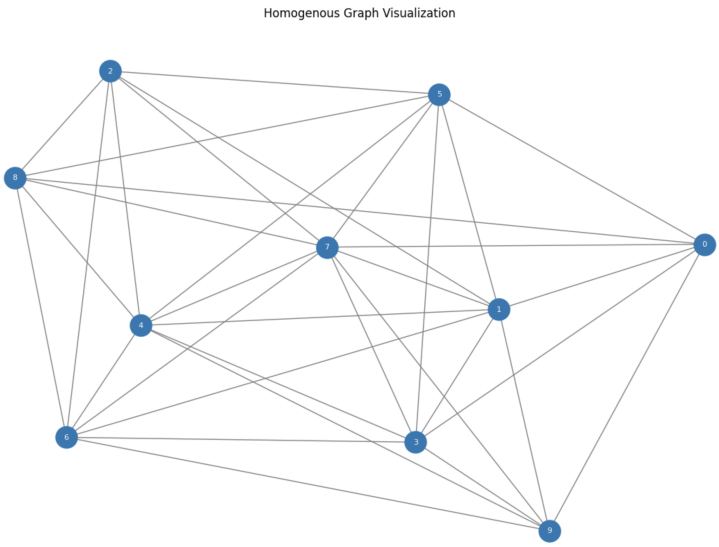
For sum aggregation, the training F1 reaches 0.78, but shows slight instability in the final epochs. The evaluation F1 starts at 0.74 then quickly drops to 0.64–0.67, remaining relatively flat. The large gap between training and evaluation F1 (even after a few epochs), and stagnation of evaluation F1 performance (despite training F1 improving), suggests overfitting. Sum aggregation may cause the model to focus on graph size, which can bias the learning process.

Thus, mean aggregation performs the best. This is because it balances information propagation without bias toward graph size (as seen in sum aggregation). It also provides stable gradients, resulting in smooth convergence and better generalization to unseen data. Overfitting is minimised, with validation performance tracking closely to training performance. The mean operation normalizes node feature contributions, preventing graphs with more nodes from dominating learning (as in sum aggregation). Unlike max aggregation, which only retains the strongest signal (leading to potential information loss), mean aggregation preserves overall structural information more robustly.

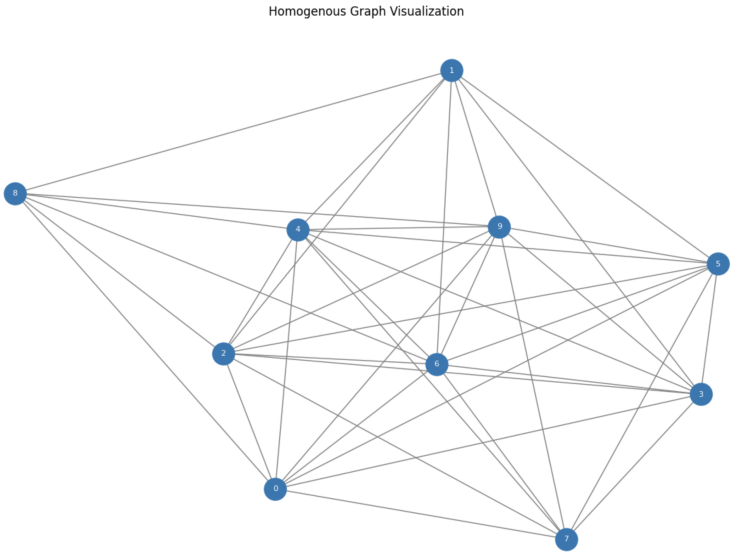
1.2 Analyzing the Dataset

1.2.a Plotting

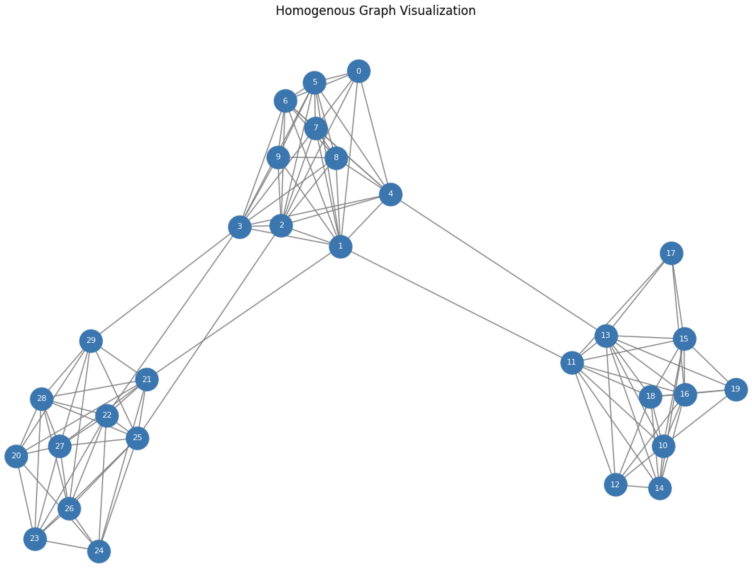
Graph Topologies:



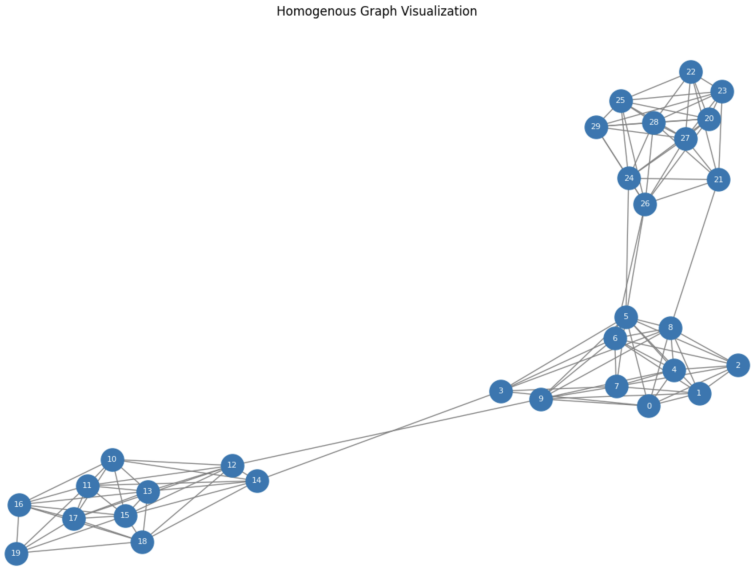
Graph Topology for a graph in training dataset with label 0



Graph Topology for a graph in evaluation dataset with label 0

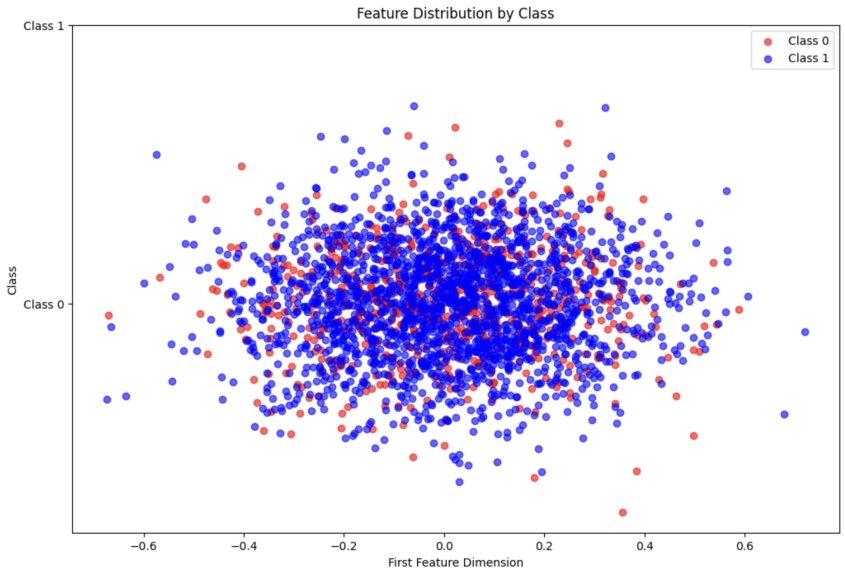


Graph Topology for a graph in training dataset with label 1

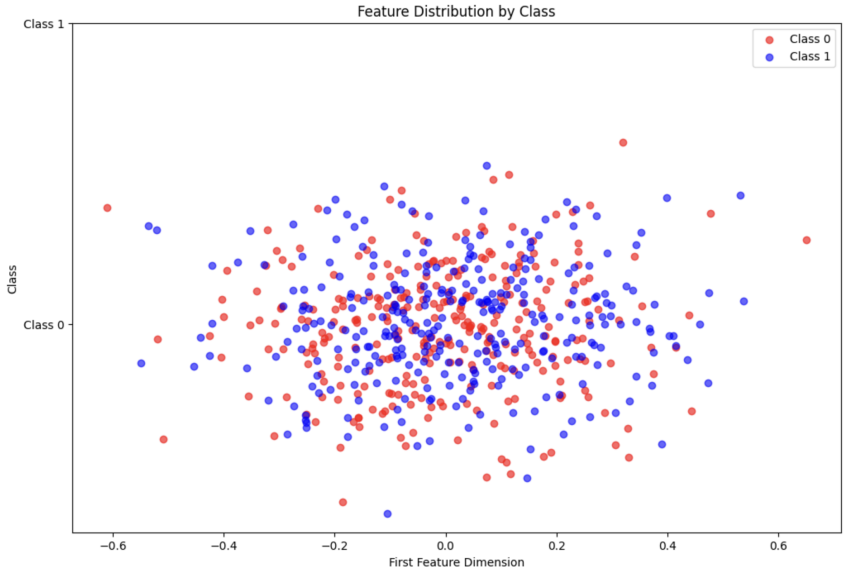


Graph Topology for a graph in evaluation dataset with label 1

Feature Distributions:

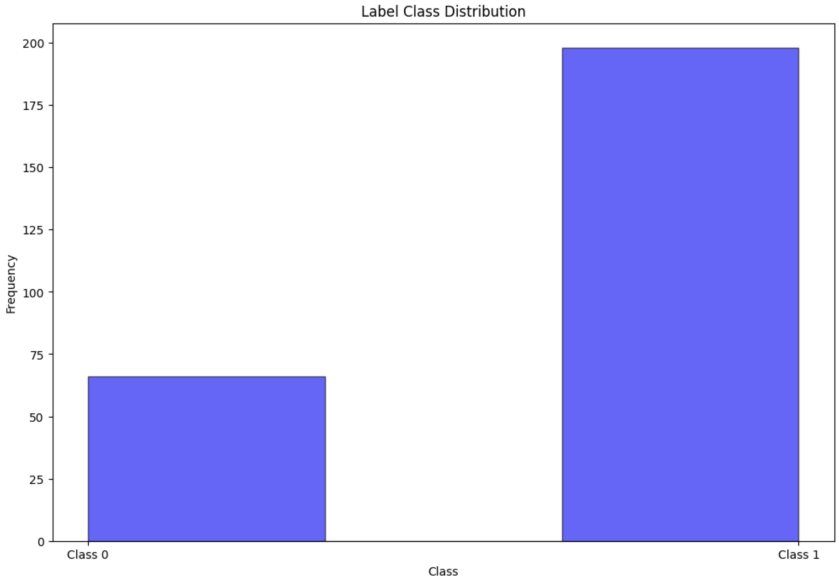


Feature Distribution of graphs in training dataset

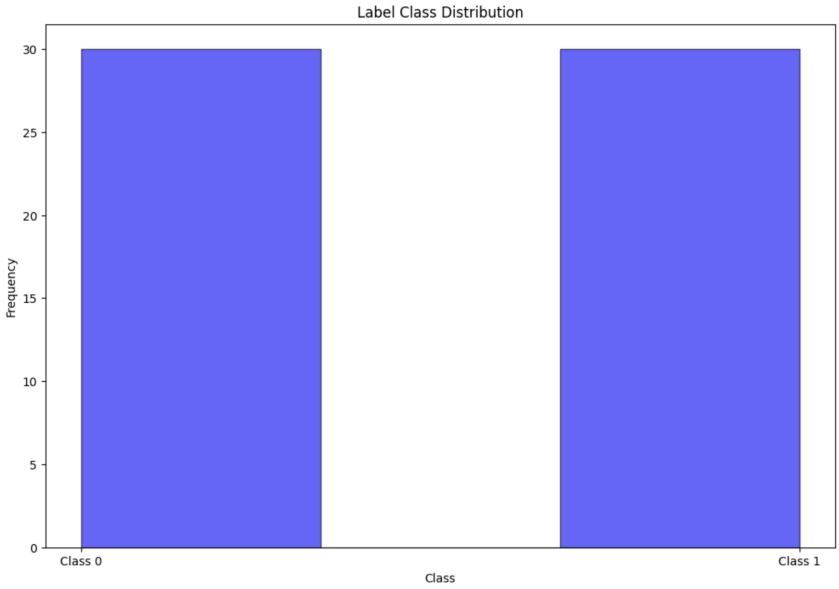


Feature Distribution of graphs in evaluation dataset

Label Distributions:



Label Class Distribution in training dataset



Label Class Distribution in evaluation dataset

1.2.b Discussion

From the graph topologies, the dataset consists of two structurally distinct classes. Class 0 contains dense, homogeneous graphs (uniform distribution of edges among nodes) with high global connectivity. This resembles near-complete graphs. Class 1 contains clustered, modular graphs requiring higher-order neighborhood aggregation. In addition, there seems to be low variance between training and evaluation sets for both classes, meaning the GCN should be able to generalise well (and not overfit) if it correctly captures the connectivity pattern. For class 1, the evaluation set shows shows slightly fewer inter-cluster edges, potentially increasing classification complexity if the model overly relies on

global connectivity patterns.

From the feature distributions, in both training and evaluation sets, there is significant overlap between Class 0 and Class 1. There is no clear boundary separating the two classes in the first feature dimension. The dense central cluster of points (around 0 on the x-axis) for both classes suggests high intra-class similarity in the feature space, meaning the GCN will have to rely heavily on graph structural information rather than node features alone to achieve effective classification. This makes feature aggregation across neighborhoods critical. The training set has a high density of points near the center (0 on x-axis), with a large number of samples. The features are more densely packed compared to the evaluation set, which has less samples. There is no significant feature shift between training and evaluation sets along the visualized dimensions. Both sets show a roughly symmetric distribution around the center, with similar variance and range. This indicates minimal covariate shift between training and evaluation sets, reducing the risk of distribution-related generalization failures.

From the label distributions, the training set is highly imbalanced towards Class 1, risking poor generalization for Class 0. Since the evaluation set is balanced, it will expose any bias, making robust training strategies critical.

1.3 Overcoming Dataset Challenges

1.3.a Adapting the GCN

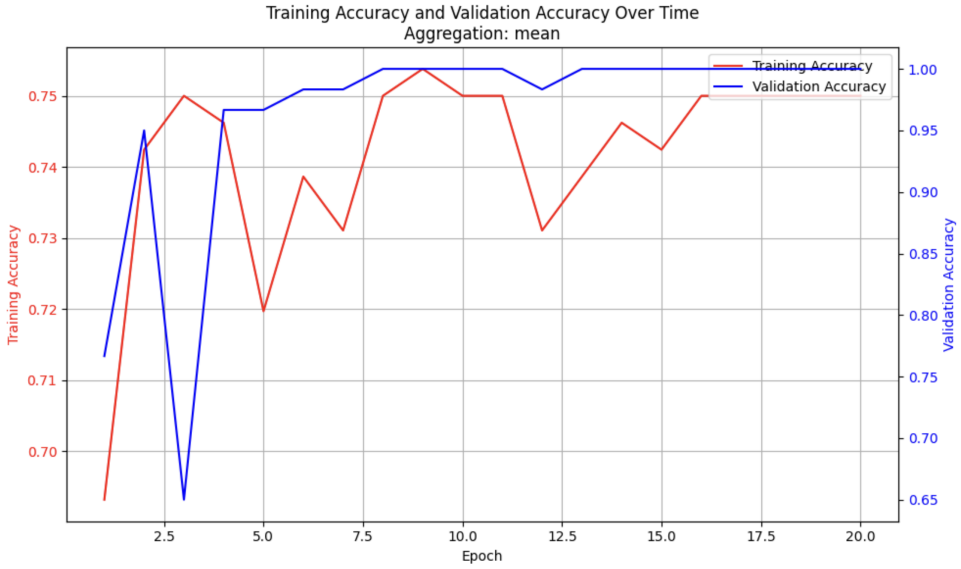
The code can be found in the cells under Q1.3a in notebook 1. The initial GCN was parameterised, forming the class 'MyGraphNeuralNetwork2' which accepts input dimension, output dimension, hidden layer dimension, and number of layers as parameters.

Then, using ParameterGrid from sklearn.model_selection, a hyperparameter grid-search was performed.

The hyperparameters were hidden layer dimension (8, 16, or 32), number of layers (2, 3, or 4), output dimension (1, 2, 4, or 8), learning rate (0.001, 0.01, or 0.1).

In each iteration, a different combination of hyper-parameters from the grid was selected to build a new model instance of 'MyGraphNeuralNetwork2', which was then trained and evaluated on the relevant datasets. The hyper-parameter combination with the highest test data F1-Score was then logged and printed.

The best-performing combination of hyperparameters was: hidden dimensions = 20, layers = 2, learning rate = 0.1, output dimensions: 2. This gave a test accuracy score of 1.0, and a test F1 of 0.333. This is the plot of training and evaluation F1 score over time for this combination:



1.3.b Improving the Model

Three different methods were identified to overcome the challenges faced by the model MyGraphNeuralNetwork2 in Q2. These were implemented in a new GCN model, MyGraphNeuralNetwork3.

First, dropout was added to the GCN architecture to improve generalization and prevent overfitting. Dropout randomly disables a proportion of neurons during training. With a dropout rate of 0.5, half of the activations from each graph convolutional layer are set to zero at every training step. This forces the network to learn more robust and distributed representations by preventing neurons from relying too heavily on specific connections. During inference, dropout is disabled, and outputs are scaled accordingly. In GCNs, dropout is particularly beneficial because it introduces randomness in the neighborhood aggregation process, ensuring the model does not overfit to local node-specific features or over-smooth node embeddings. This leads to improved validation performance.

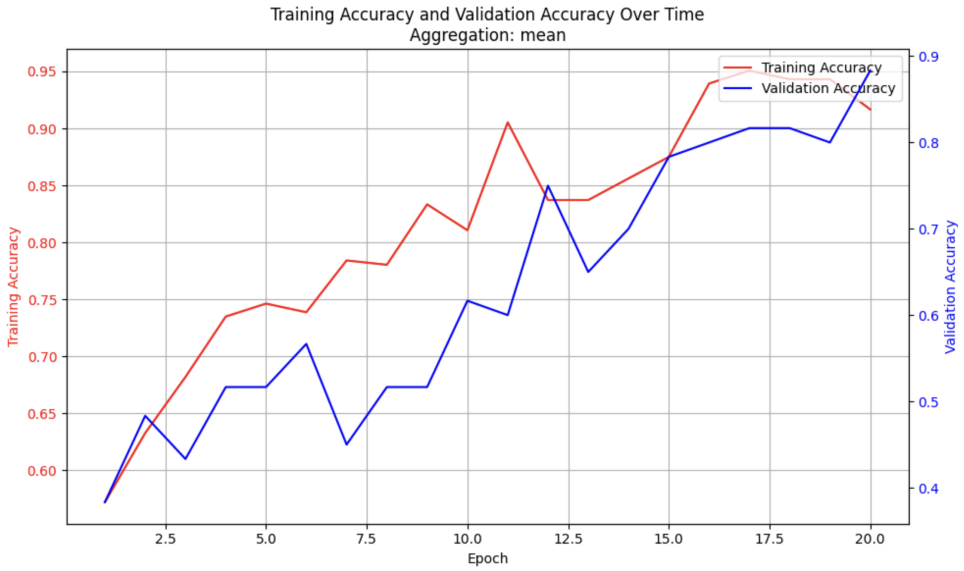
Next, a hyperparameter search was ran over several parameters, including hidden dimension, number of layers, learning rate and dropout rate/probability. Once run, the values producing the highest testing accuracy were selected and implemented in the model MyGraphNeuralNetwork3. These were: hidden dimension = 24, number of layers = 3, learning rate = 0.003, dropout rate = 0.5.

Hyperparameter grid search systematically explores all combinations of predefined hyperparameter values to find the best-performing configuration. For graph neural networks, parameters like hidden dimension, number of layers, learning rate, and dropout rate critically affect learning and generalization. Grid search ensures optimal performance by evaluating each combination and selecting the one with the highest validation or testing accuracy. This process balances model complexity and generalization, identifies stable learning rates, suitable hidden dimensions, and effective dropout rates to prevent overfitting, streamlining model optimization without manual guesswork.

The final addition was an adjustment of the loss function. Rather than standard binary cross-entropy, a class-weighted binary cross-entropy loss function was used during training.

This is important because the training dataset is imbalanced, so standard BCE may bias the model toward the majority class. Without weighting, the model can achieve high accuracy by favoring the dominant class, neglecting the minority class and resulting in poor recall and F1 scores. Class-weighted BCE solves this by penalizing misclassifications of the minority class more heavily, encouraging the model to learn its distinguishing features. This is particularly useful in GNNs where node embeddings rely on information from neighboring nodes. Weighted loss ensures that embeddings capture relevant patterns for all classes, improving generalization.

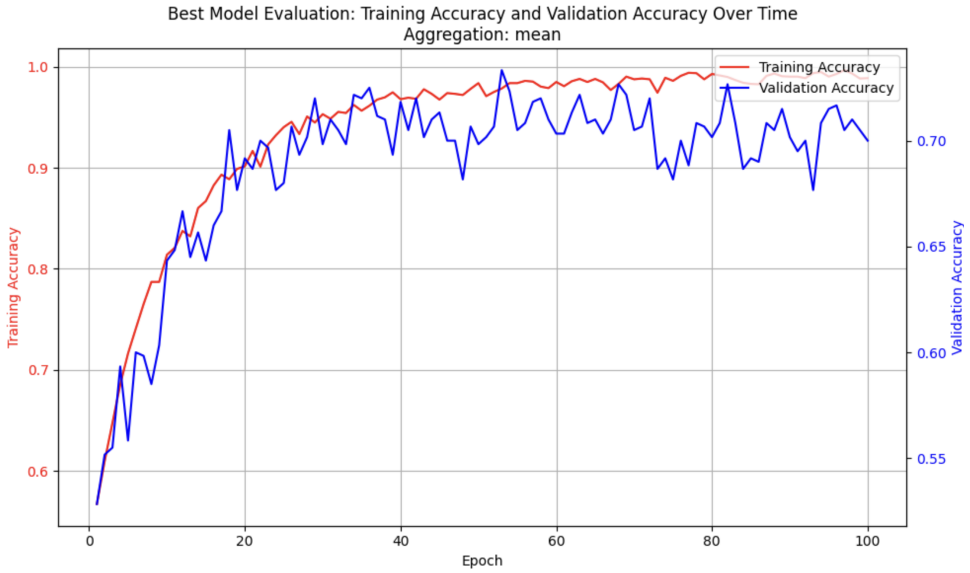
Below is the training and validation accuracy of the improved model, MyGraphNeuralNetwork3.



The training accuracy was 0.917, and validation accuracy was 0.8833.

1.3.c Evaluating the Best Model

Below is the plot of my best model, ran across 100 epochs and averaged over 10 i.i.d. runs.



1.3.d Final Analysis and Explanation

The first change to the model architecture was to parameterise several features, such as number of layers and hidden dimension size. This was the first step to an increase in customisability of the network

I then changed the model to have dropout in certain layers, with a rate of 0.5. This was implemented in all hidden layers (not input or output). This means that half the nodes (selected randomly) were de-activated during training. This forces the network to learn more distributed representations, which helps with generalisation. Dropout is disabled during inference, and outputs are scaled accordingly.

The loss function was altered to be a class-weighted binary cross-entropy loss function. This was weighted according to the frequency of each label (0 or 1) in the training dataset. This weighting meant that less common labels in the training dataset have a higher loss associated with them, ensuring the network learns patterns in these graphs. Without this, it is difficult for the node to learn as it's training will be biased towards the majority class.

I used grid search (from sklearn.model_selection import ParameterGrid) to tune hyper-parameters across a range of values. This would run the training and testing logic over every combination of hyperparameter values given, storing the parameters and metrics of each run in a dictionary. These hyperparameters were hidden dimension, number of layers, learning rate and dropout rate/probability. After being trained and tested across each combination, the parameter-set with the highest metrics were inspected and set as the model's parameters to achieve optimal performance.

2 Node Classification in a Heterogeneous Graph

2.1 Dataset

2.1.a Problem Challenge

The node classification problem described is more challenging than standard tasks due to the presence of heterogeneous nodes with different feature dimensions. The graph includes two node types, t_1 and t_2, with d_1=20 and d_2=30 features, respectively. Unlike standard GNNs that assume homogeneous nodes with a common feature space, this task requires handling multiple feature spaces. The model must perform type-specific transformations and align feature dimensions for effective message passing, increasing architectural complexity. Additionally, cross-type edges introduce semantic variability in relationships, requiring specialized aggregation mechanisms. In standard tasks, edges are uniformly interpreted, but here, the GNN must adapt aggregations based on node and edge types. This often requires advanced architectures like R-GCN, HAN, or HGT, capable of capturing complex, heterogeneous interactions. Furthermore, the distribution of class labels across different node types complicates generalization, as the model must learn consistent representations despite structural and feature heterogeneity. Lastly, the problem includes scalability challenges, with hundreds of nodes per type and complex adjacency structures. Efficiently managing multi-modal data fusion, cross-type neighborhood aggregation, and proper indexing of heterogeneous features makes this task significantly more complex than standard node classification.

2.1.b Real-World Analogy

A real-world example of this heterogeneous node classification problem is an academic collaboration network, where the goal is to predict the research field (class label) of entities within the network. In this scenario, the two node types represent authors (t1) and research papers (t2). The feature vectors for author nodes (t1) include metrics such as number of publications and areas of expertise (with d1 = 20 features). The feature vectors for research-paper nodes (t2) include features like citation count, keywords, and abstract embeddings (d2 = 30 features).

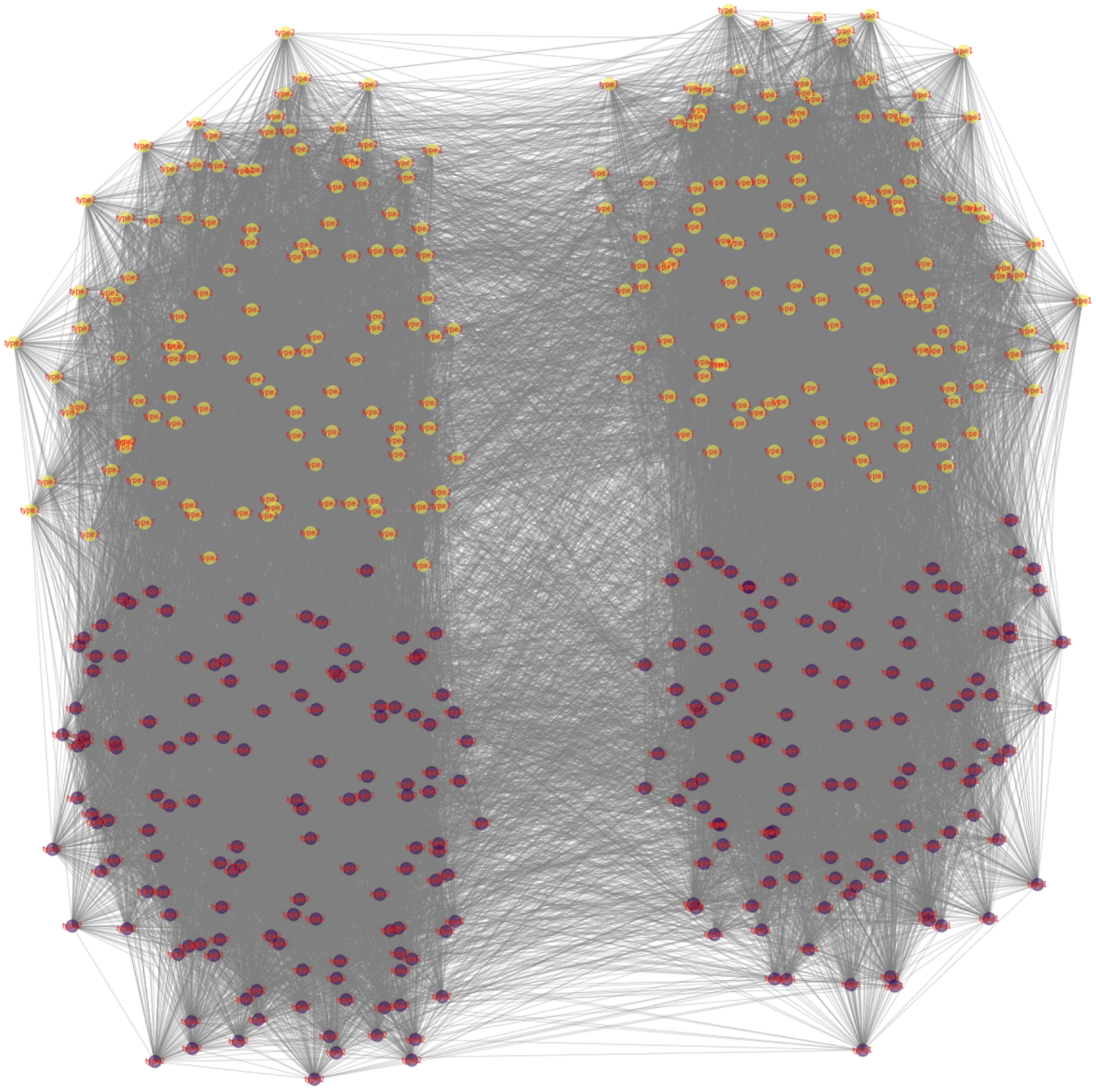
The two class labels, c1 and c2, represent different research domains, such as biology (c1) and physics (c2). Edges between nodes represent relationships such as an author writing a paper or co-authorship links between authors through shared publications.

The graph is heterogeneous because the relationships and feature representations differ between authors and papers, but both are required to accurately classify the research field.

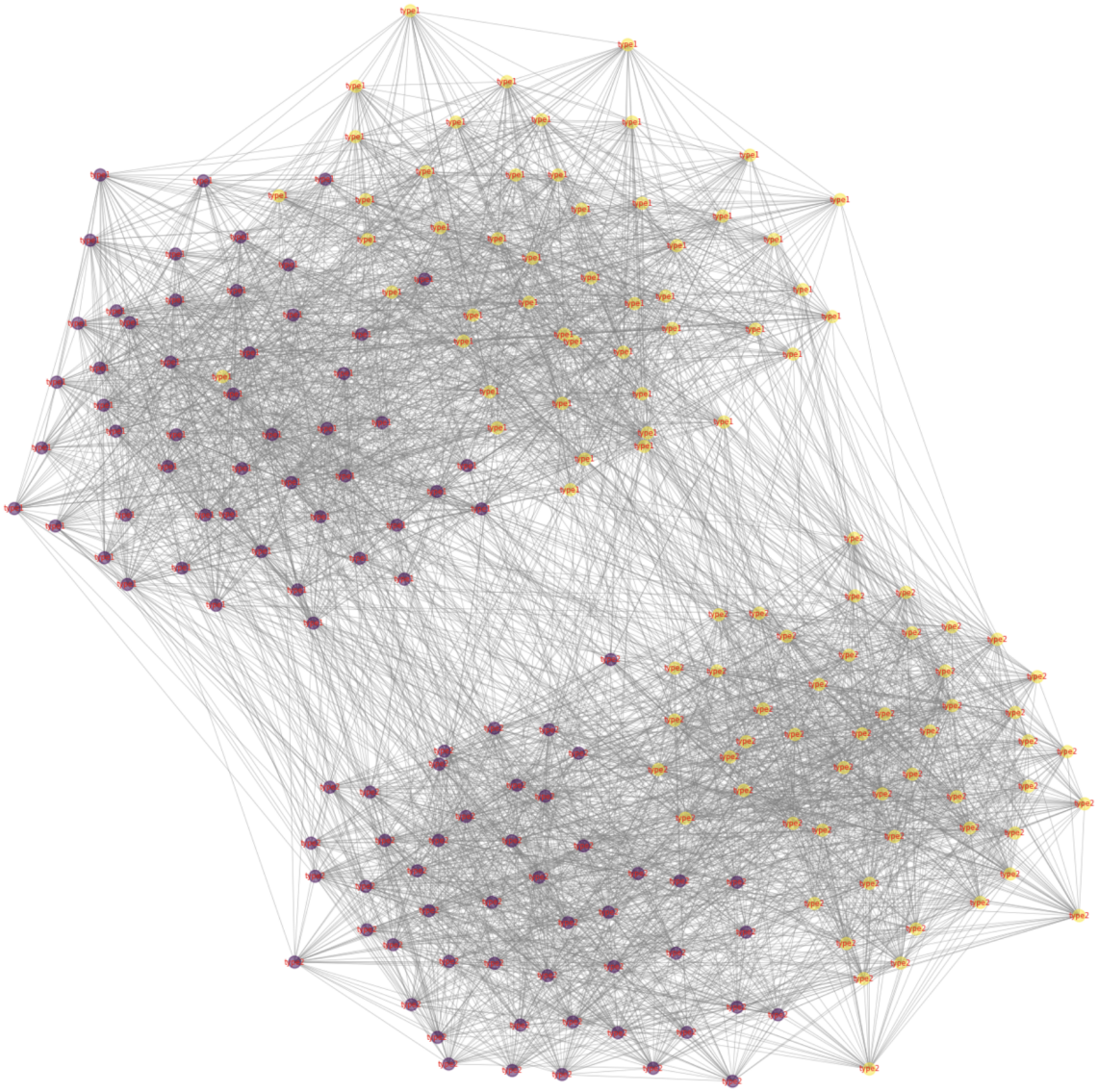
Understanding the relationships between node types and classes is important because the classification of one node type often depends on information from the other. For example, an author's research domain (class) can be inferred from the topics of their published papers, while the classification of a paper's research field (class) may depend on a different

feature vector, such as the expertise of its authors. If these cross-type relationships aren't recognized, a model can have poor generalization, as significant contextual information would be ignored.

2.1.c Interpretation of the Dataset: Plotting the Graph



Graph Structure of Training Graph, coloured by class and labelled by type



Graph Structure of Evaluation Graph, coloured by class and labelled by type

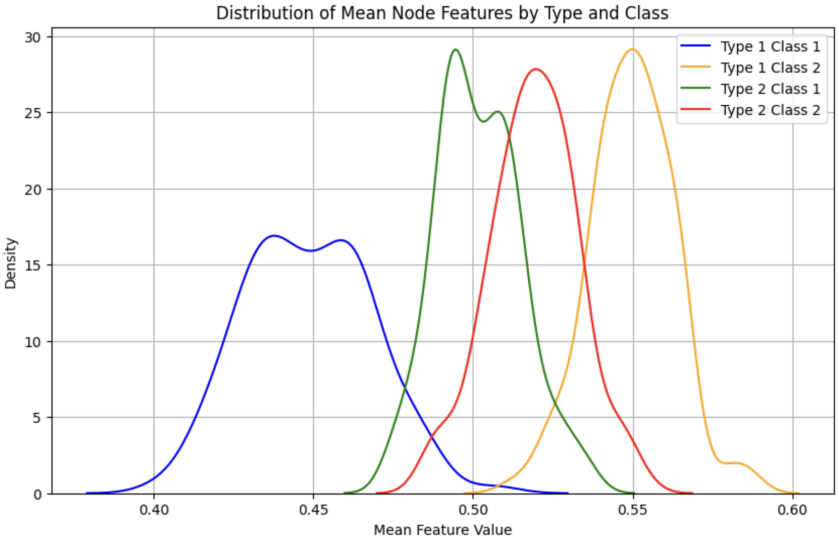
Several patterns of node distribution, among both class and type, can be observed. There is a clear separation of nodes by class labels (yellow and purple), with nodes of the same class forming distinct and densely connected clusters. This strong class-based clustering indicates that nodes are more likely to connect with others of the same class. The separation is consistent across both the training and evaluation graphs, with minimal overlap between classes, suggesting that graph structure alone provides strong signals for classification.

Both node types are distributed throughout each class cluster, rather than being segregated by type. This indicates class membership is not solely determined by node type, but instead by a combination of node features and structural position.

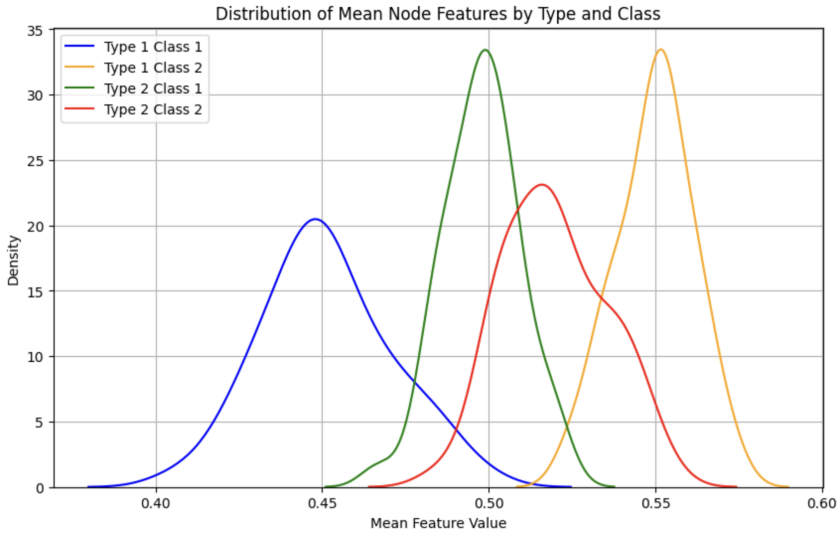
The training graph is denser than the evaluation graph, with more edges connecting nodes across and within classes. The evaluation graph shows clearer modularity, with well-separated class clusters and fewer cross-type edges. This difference in density means the

training graph provides richer structural information for learning inter-type relationships, and the evaluation graph tests the model to generalize these relationships with less direct connectivity.

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



Training Graph: distribution of mean node features



Evaluation Graph: distribution of mean node features

2.1.e Interpretation of the Dataset: Discussion

The distribution of mean node features plots show a visible separation between classes based on mean feature values for both node types. Class 1 nodes (blue for type 1 and green for type 2) consistently have lower mean feature values than Class 2 nodes (orange

for type 1 and red for type 2). This separation suggests that the node features provide strong discriminative signals for distinguishing between classes, potentially allowing the model to leverage feature information effectively for classification.

Type 1 nodes have a broader spread in mean feature values compared to type 2 nodes, which have narrower, more concentrated distributions. This indicates that type 1 nodes possess more diverse feature representations, while type 2 nodes are more homogeneous. This implies that the model may need to aggregate information differently across node types, as the variability in type 1 features could offer richer signals, while type 2 nodes may rely more heavily on structural context for accurate classification.

The relative positions and shapes of the distributions are consistent between the training and evaluation graphs. In both cases, type 2 Class 1 and type 2 Class 2 distributions are closer together, suggesting that type 2 nodes may be more challenging to classify based on features. The low variance between training and evaluation graphs imply that the feature characteristics generalize well.

2.2 Naive Solution: Padding

2.2.a Limitations of Naive Solution

Two key limitations of the naive solution are reduced model expressiveness and lower computational efficiency.

The naive solution causes reduced model expressiveness because zero-padding introduces sections of uninformative features for nodes with smaller feature dimensions. For example, type 1 nodes with 20 features are padded with 10 zeros to match the 30-dimensional feature space of type 2 nodes. This can dilute the impact of meaningful features/signals during training, making it harder for the model to learn relevant patterns. Also, the model may mistakenly assign importance to these padded dimensions, leading to suboptimal feature transformations. This means the GNN would struggle to capture subtle relationships between node features and their corresponding classes, especially in heterogeneous graphs where different node types have distinct semantic meanings.

Zero-padding also leads to unnecessary computational and memory overhead, decreasing the model's efficiency. It causes the GNN to perform operations on zero-valued features that do not contribute to learning. This is a particular issue in large-scale graphs, where computational resources are a concern. For instance, the extra 10 zero-valued features added to every type 1 node increase the computational cost of each network layer without providing any useful information. This overhead affects both training time and inference speed, making the model less scalable. Memory consumption also rises unnecessarily, limiting the model's practicality for large heterogeneous graphs.

2.3 Node-Type Aware GCN

2.3.a Implementation

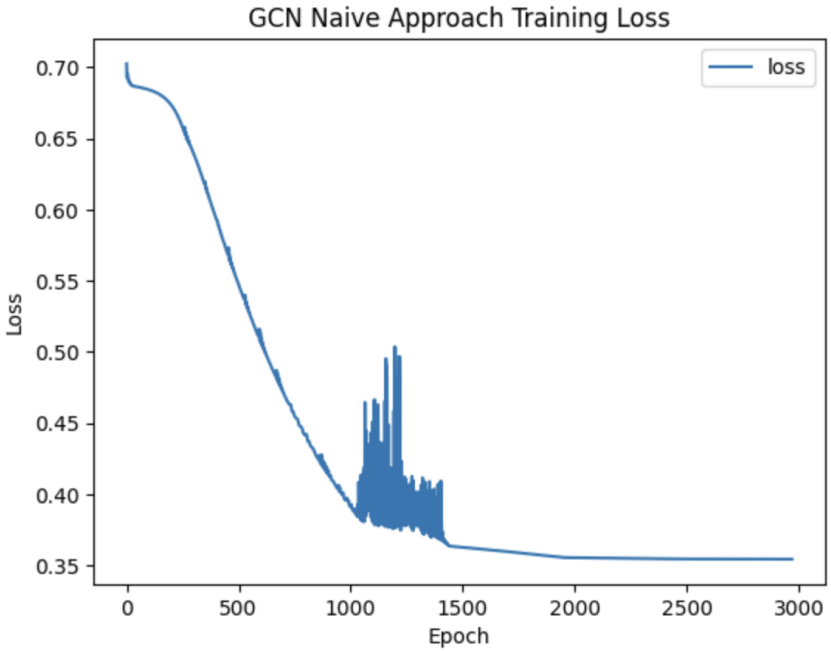
The implementation of node-type aware GCN can be found in the 'Question 2.3' cell of the q2 notebook.

2.3.b Discussion

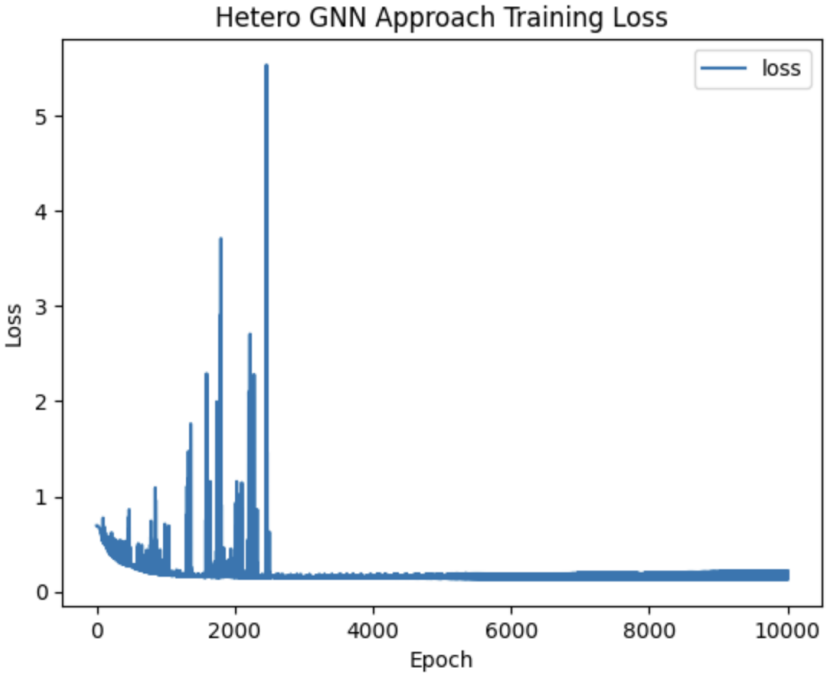
My HeteroGCN model employs a node-aware 2-layer GCN architecture, tailored for heterogeneous graphs with distinct node types. Unlike the original naive padding implementation, where all nodes were treated uniformly, HeteroGCN separately processes each node type through independent GCN layers (gcn1_type1 and gcn1_type2). This design preserves the unique structural and feature characteristics of each node type. The processed embeddings are then merged and passed through a shared second GCN layer and a classifier for final predictions. The mapping parameter passed at runtime ensures correct alignment between nodes in the adjacency matrix and their corresponding features.

The limitation of the naive implementation was that it relied on padding feature matrices to ensure uniform dimensions across node types. This caused information loss (padding adds redundant information), inefficiency (computations are wasted on padded entries), and lack of type-specific processing (the naive model could not differentiate between node types). HeteroGCN overcomes these issues by using node-type specific GCN layers (independently process each node type), dynamic mapping between adjacency indices and feature indices at runtime (eliminates the need for padding), and better generalization (due to type-aware representation learning).

My approach has several advantages. One is type-aware learning - specialized layers for each node type lead to richer embeddings and improved classification performance. Another is dynamic mapping, by passing mapping at runtime, the model can adapt to varying graph structures during validation or testing without re-initialization. The final advantage is reduced redundancy - eliminating padding reduces memory consumption and speeds up training. The approach also has some potential drawbacks. For example, the model introduces additional complexity by maintaining separate GCN layers and dynamic indexing. In addition, the model is sensitive to hyperparameters as oerformance heavily depends on tuning parameters (hidden_dim, learning rate, and patience). Finally, processing multiple adjacency submatrices could introduce computational overhead for very large graphs.



Loss Curve for Naive (Padding) Approach



Loss Curve for Node-Type Aware GCN (HeteroGCN)

For hyperparameter tuning, a grid search strategy was used (sklearn.model selection's ParameterGrid). A set of possible hyperparameter values was defined, and the model was repeatedly trained and evaluated with each possible combination of hyperparameters. Then the model with the highest F1 score was selected.

The search focused on three parameters: hidden dimension, learning rate, and scheduler patience. For hidden Dimension (hidden_dim), the values tested were: 2, 4, 8, 9, 13, 14

16, 23, 26, 27, 31, 32. The chosen value was 26. The reason for the range selected was that lower dimensions (2, 4) led to underfitting, while higher dimensions (> 32) risked overfitting. 26 balanced performance and generalization, giving the best validation F1-score. For learning rate (lr), the values tested were: 0.001, 0.005, 0.01, 0.02, 0.05. The chosen value was 0.01. The reason was that 0.001 and 0.005 slowed convergence, whilst 0.02 and 0.05 caused instability in training. 0.01 provided a balance between convergence speed and training stability. For scheduler patience (patience), the values tested were 50, 100, 200, 500, 1000. The chosen value was 1000. Short patience values led to premature learning rate reductions, stopping proper convergence. A patience of 1000 allowed the model sufficient time to explore the parameter space before adjusting the learning rate, leading to better validation performance.

Despite showing promise, the final HeteroGCN achieved a validation precision of 0.8350, falling just short of the naive F1 of 0.8850.

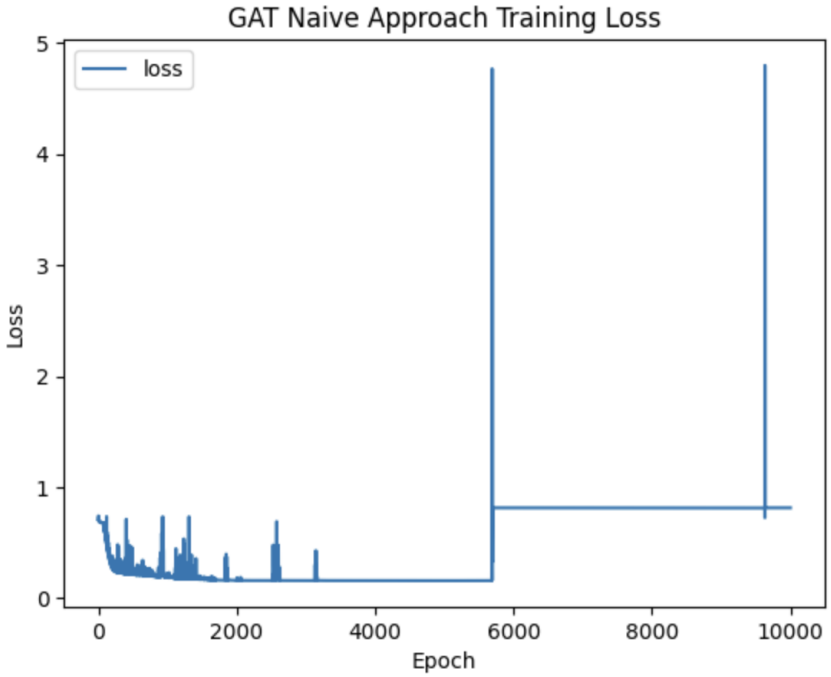
2.4 Exploring Attention

2.4.a Implementation

The implementation of the GCN with an attention-based aggregation mechanism can be found in the 'Question 2.4' cell of the q2 notebook.

2.4.b Discussion

The attention-based aggregation in the GraphAttentionNetwork (GAT) allows each node to assign learnable importance weights to its neighbors during feature aggregation. Instead of treating all neighboring nodes equally (as in standard GCNs), GAT uses a self-attention mechanism. First, each node computes an attention score for its neighbors based on their feature similarity. Then, the scores are then normalized using a softmax function, ensuring they sum to one. Then, each node's updated representation is the weighted sum of its neighbors' features, where higher weights indicate more influential neighbors. This approach allows the model to focus on more relevant neighbors dynamically, improving representation learning, especially in heterogeneous graphs where neighbor significance can vary a lot.

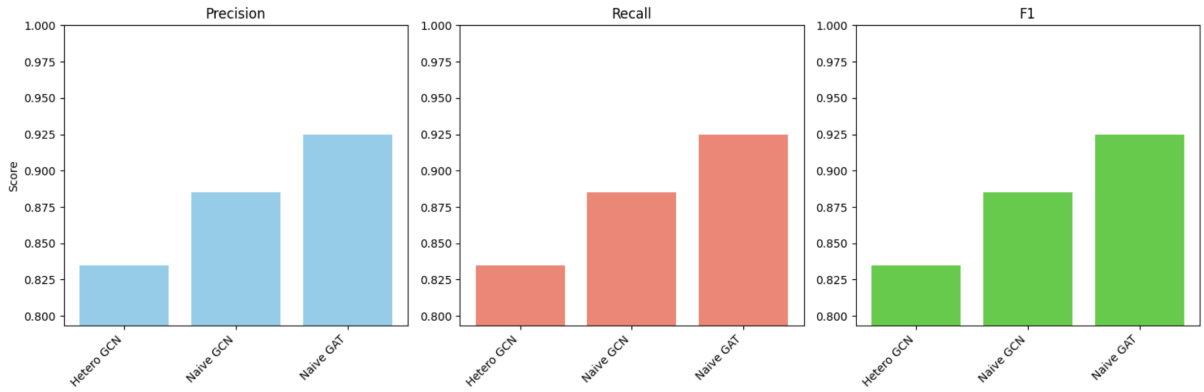


Loss Curve for Naive Graph Attention Network

To identify optimal hyperparameters, the grid search strategy was used again. This focused on four parameters: hidden dimension, number of attention heads, learning rate, and scheduler patience.

For the hidden dimension, the values tested were: 4, 8, 9, 16, and 32. The chosen value was 9. Lower values such as 4 and 8 led to underfitting as they failed to capture complex relationships, while higher values like 16 and 32 risked overfitting. The value 9 provided the best balance between model complexity and generalization. For the number of attention heads, the values tested were: 1, 2, 3, 4, and 8. Increasing the number of heads allows the model to capture diverse aspects of neighborhood information. A model with 3 heads was selected because it offered richer feature representations without too much computational overhead, outperforming both single-head and higher-head alternatives. For learning rate, the values tested were: 0.001, 0.005, 0.01, 0.02, and 0.05. The value chosen was 0.02 as it enabled faster convergence without introducing instability, whilst lower rates slowed training and higher rates caused oscillations in validation performance. For scheduler patience, the values tested were: 50, 100, 500, and 1000. A patience value of 1000 was selected as attention-based models often benefit from longer patience periods, allowing enough time for the attention coefficients to stabilize, leading to higher final validation F1-scores.

2.5 Overall Discussion



Bar plot comparing precision, recall and F1 scores of the naive model, HeteroGCN, and Graph Attention Network

The bar plot shows that the Naive GAT model achieves the highest precision, recall, and F1 score among the three models, followed by the Naive GCN, with the Hetero GCN performing the worst.

The Node-Type Aware GCN (Hetero GCN), although theoretically expected to outperform the Naive GCN, underperformed in practice. The Hetero GCN should achieve better performance because it processes node types separately using type-aware GCN layers before combining them. This approach allows the model to preserve and exploit node-type-specific patterns rather than losing them through naive padding. By maintaining the distinct characteristics of each node type, the model should provide more meaningful embeddings and better classification performance. The reason for underperformance could be due to the imbalance in the training dataset, where one of the labels had far fewer graph samples, meaning one of the specialised node processor was trained a smaller dataset.

The Naive GAT outperforms the others due to its attention-based aggregation mechanism, which assigns dynamic weights to neighboring nodes during the message-passing process. This allows the model to focus on the most informative neighbors, capturing more relevant structural and feature information, resulting in better performance across all evaluation metrics.

3 Investigating Topology in Node-Based Classification Using GNNs

3.1 Analyzing the Graphs

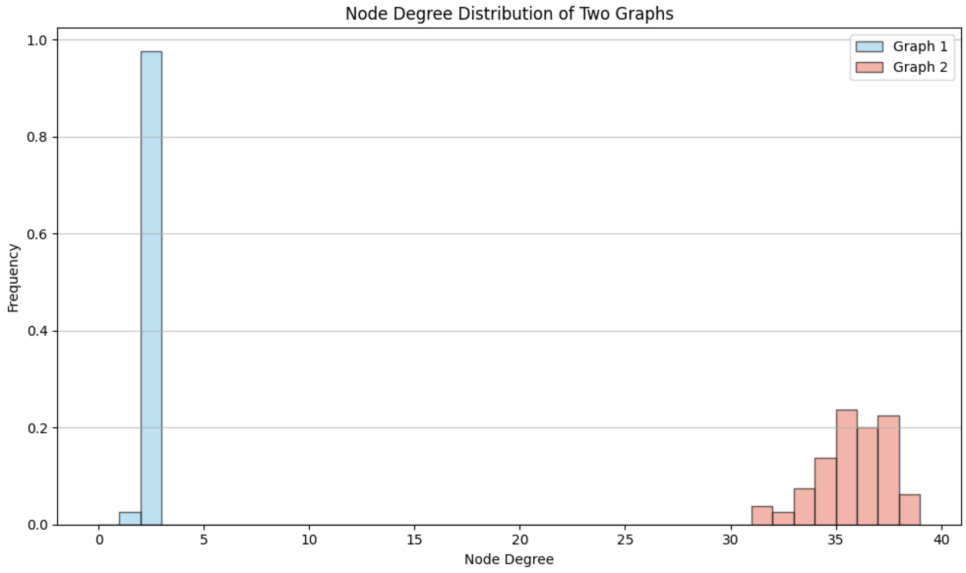
3.1.a Topological and Geometric Measures

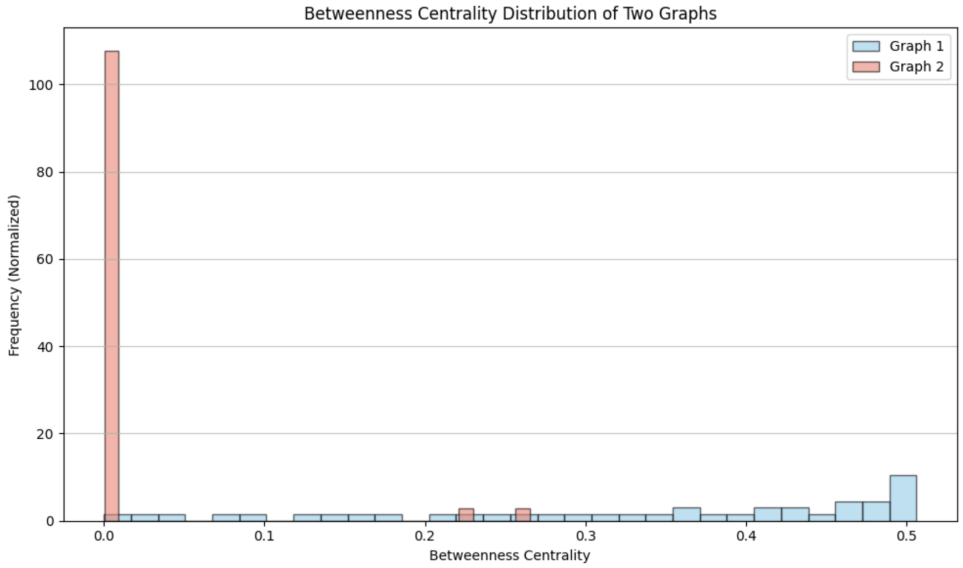
Node degree is a topological measure that represents the number of edges connected to a node. It quantifies how connected a particular node is within the graph, distinguishing between highly connected hub nodes and peripheral nodes with fewer connections. Node degree contributes to understanding the structure of a graph by identifying influential nodes that may play critical roles in communication, information flow, or robustness of the network.

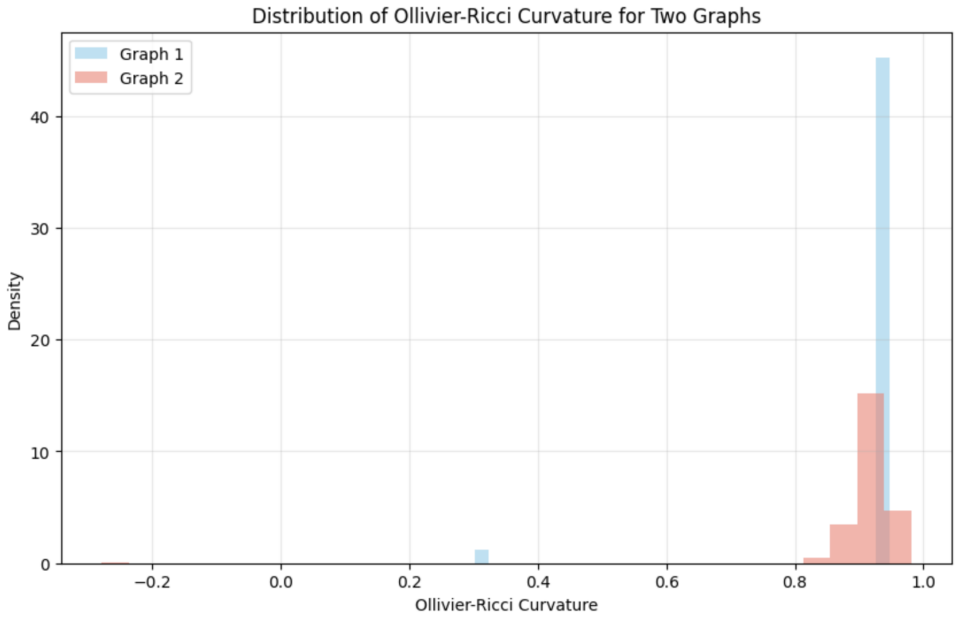
Betweenness centrality is another topological measure that quantifies the extent to which a node lies on the shortest paths between other nodes. It measures the importance of a node as a bridge within the network, indicating its role in controlling the flow of information between different parts of the graph. Nodes with high betweenness centrality are crucial for maintaining connectivity and can influence how efficiently information or resources travel across the network. Betweenness centrality contributes to understanding the structure of a graph by highlighting nodes that, if removed, could disrupt communication between otherwise disconnected subgraphs.

Ollivier-Ricci curvature is a geometric measure that assesses how much the local neighborhood of a node differs from a flat geometric space. It quantifies the robustness and cohesiveness of connections between nodes by analyzing the curvature along edges. Positive curvature indicates tightly connected communities with strong local cohesion, while negative curvature suggests edges acting as bridges between different communities. Ollivier-Ricci curvature contributes to understanding the structure of a graph by revealing hierarchical relationships, detecting community boundaries, and identifying vulnerable connections that, if disrupted, could significantly alter the graph's global structure.

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







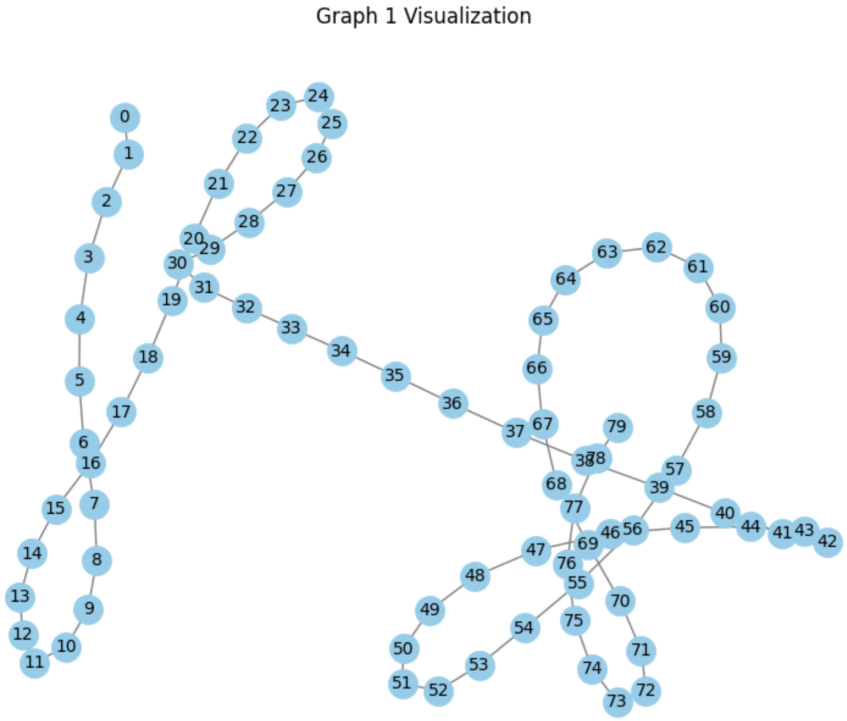
These plots show significant topological and geometric differences between the two graphs. The node degree distribution plot shows that graph 1 has low degrees (2–3), indicating a sparse, tree-like structure with limited connectivity. In contrast, graph 2 has higher degrees (30–38), indicating a denser, well-connected structure with multiple edges per node. This implies Graph 2 is more interconnected, while Graph 1 has fewer redundant paths.

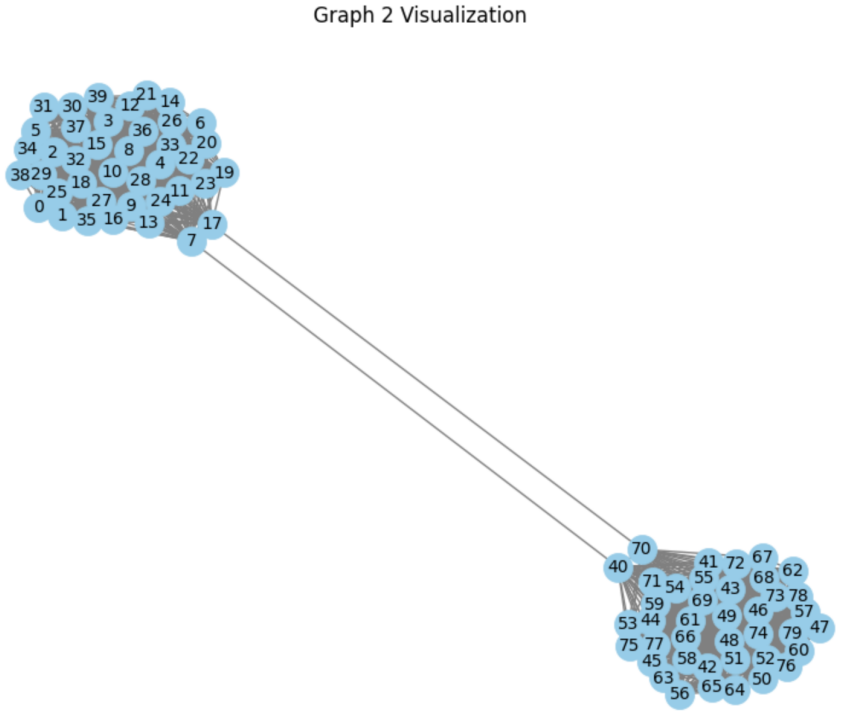
The betweenness centrality distribution shows that graph 1 has nodes with high centrality (up to 0.5), indicating reliance on key nodes as bottlenecks for shortest paths. Graph 2 has centrality values clustered near zero, indicating uniform connectivity with no dominant nodes, leading to greater robustness.

The Ollivier-Ricci curvature distribution shows Graph 1 has curvature values tightly clustered around 0.95, suggesting strong local cohesion but weak global connectivity. Graph 2 exhibits a broader distribution (0.85–0.95), reflecting globally cohesive neighborhoods with diverse connectivity patterns.

Thus, graph 1 is sparse, with critical nodes and local clustering, whilst graph 2 is dense, robust, and globally connected.

3.1.c Visualizing the Graphs

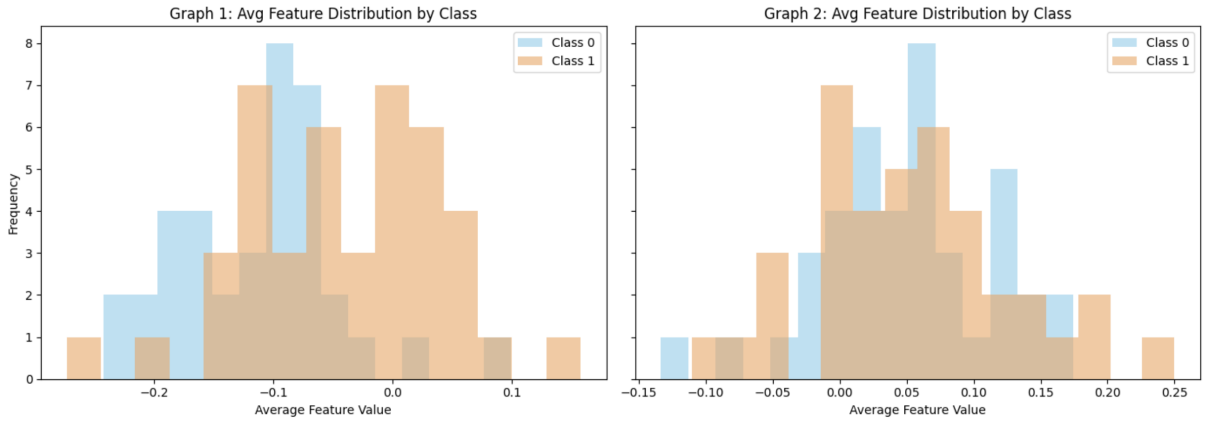




Graph 1 has a sparse, tree-like topology with linear chains and small cycles. The nodes are loosely connected, forming multiple branching paths. This is a decentralized structure where no single node dominates connectivity, leading to longer paths between distant nodes. This indicates high average shortest path lengths and minimal redundancy.

In contrast, Graph 2 shows a highly clustered topology, composed of two densely connected clusters connected by only a few inter-cluster edges. This implies to high redundancy within each community, enabling multiple alternative paths between nodes. The inter-cluster edges, however, act as critical bridges. Thus, Graph 2 likely exhibits lower average shortest path lengths within communities but relies heavily on a few nodes for cross-community connectivity.

3.1.d Visualizing Node Feature Distributions



The mean node feature distribution plots for Graph 1 and Graph 2 both show visible overlap between the distributions of Class 0 and Class 1 in both cases.

In Graph 1, the overlap is substantial, with both classes sharing a wide range of average feature values centered around zero. This significant overlap indicates that the features are not well-separated, making it harder for models to distinguish between the two classes purely based on node features. However, Class 1 shows a slightly broader distribution with a tail extending toward positive values, which may offer some discriminative potential.

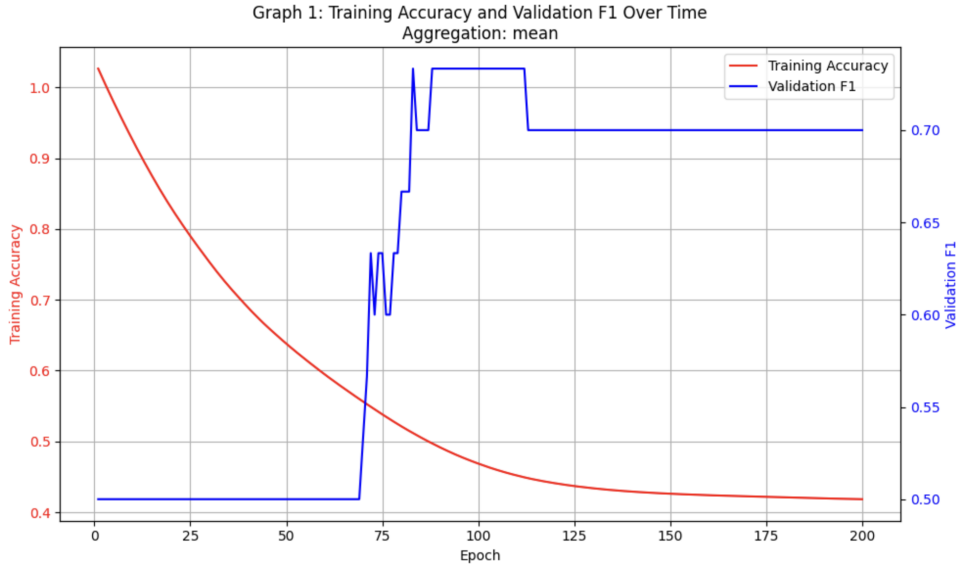
In Graph 2, the distributions are much more tightly clustered and exhibit a narrower range of feature values. Although overlap between classes still exists, the peaks of the distributions are more distinct compared to Graph 1. This indicates that Graph 2's features are more discriminative, potentially allowing for better classification performance. The reduced spread in Graph 2 could indicate that nodes are more homogeneously represented, due to a more modular graph structure.

Comparing both graphs, Graph 1 displays greater feature diversity but with more class overlap, meaning that while the graph structure may be more complex, the feature representation does not provide clear separation between classes. Graph 2 shows less overlap and more compact feature distributions, implying an easier classification task due to better feature alignment with class labels. Graph 2 shows better class separation in terms of node feature distributions.

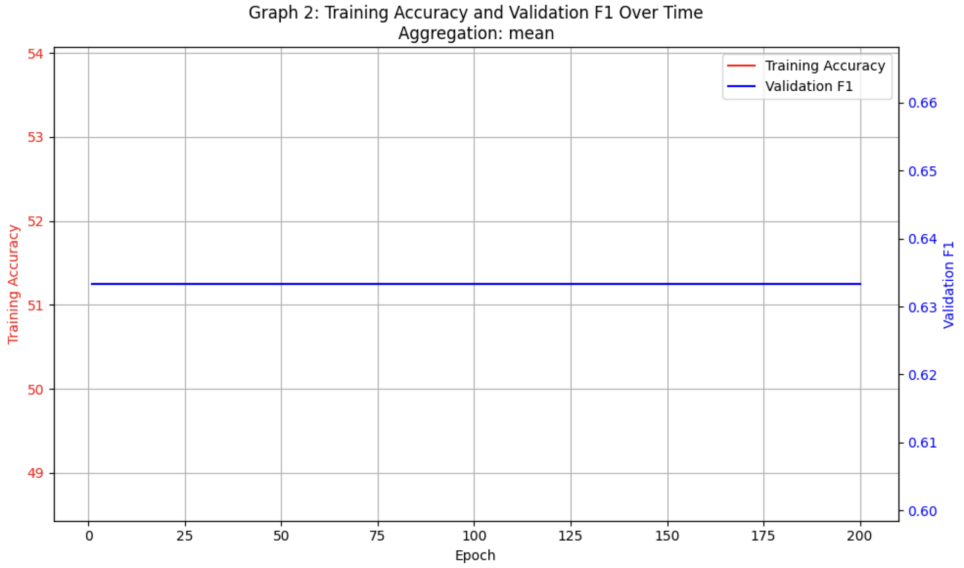
3.2 Evaluating GCN Performance on Different Graph Structures

3.2.a Implementation of Layered GCN

The implementation of LayeredGCN can be found in the Q3.2a cell in the Q3 notebook. LayeredGCN was trained on G1 and G2 independently. These are the plots:

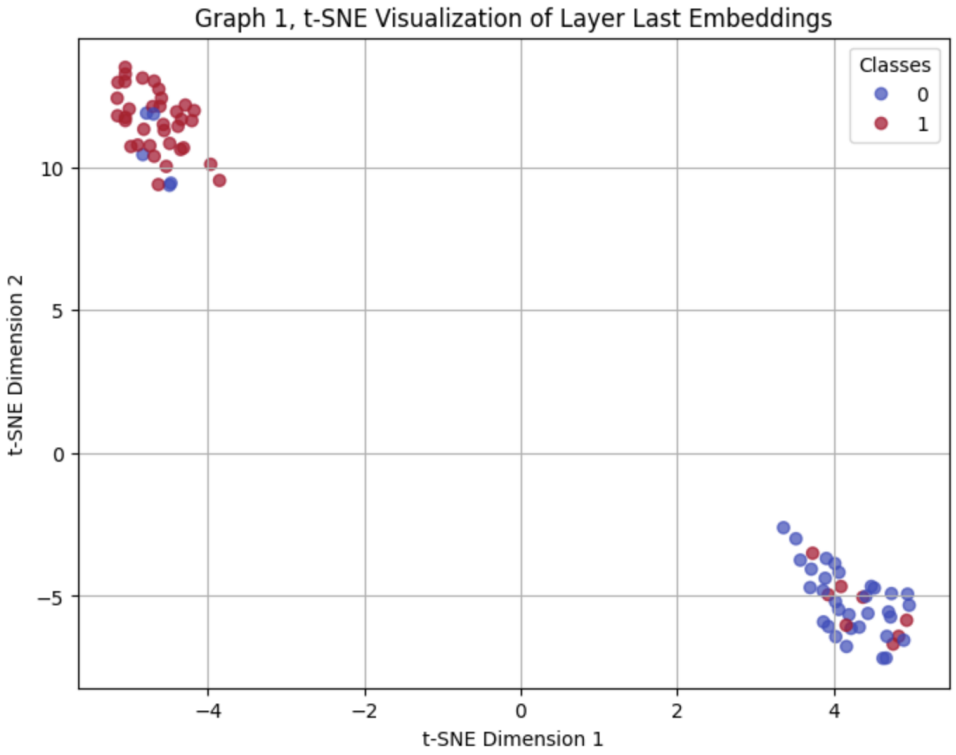


On Graph 1, the model achieves a training F1 of 0.8375 and a validation F1 of 0.7333.

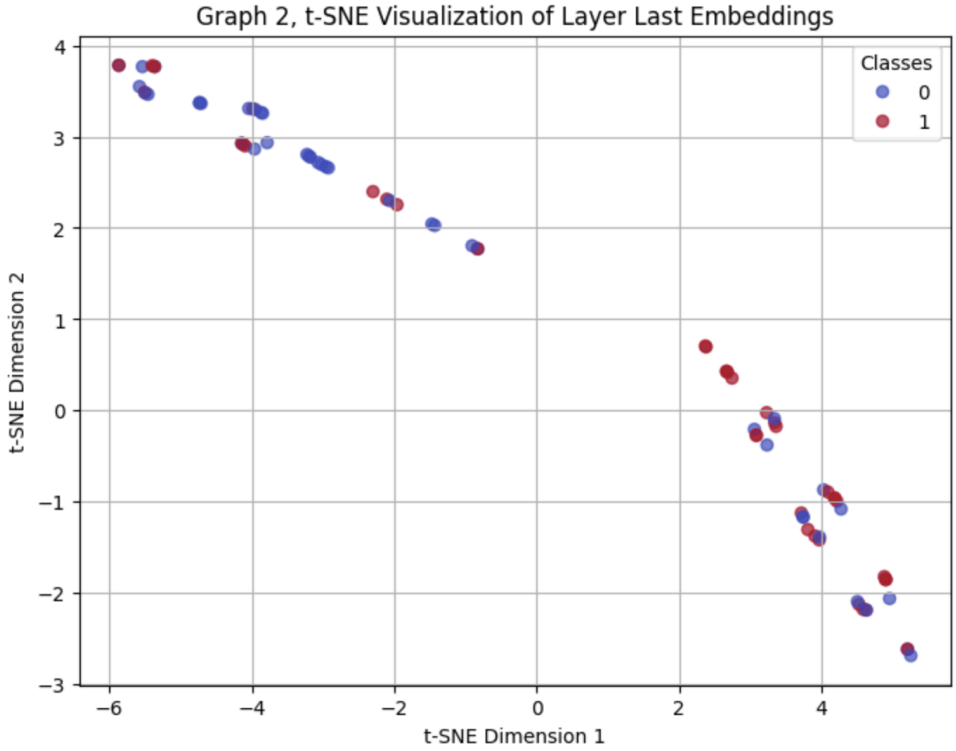


On Graph 2, the model achieves a training F1 of 0.5125 and a validation F1 of 0.3667.

3.2.b Plotting of t-SNE Embeddings



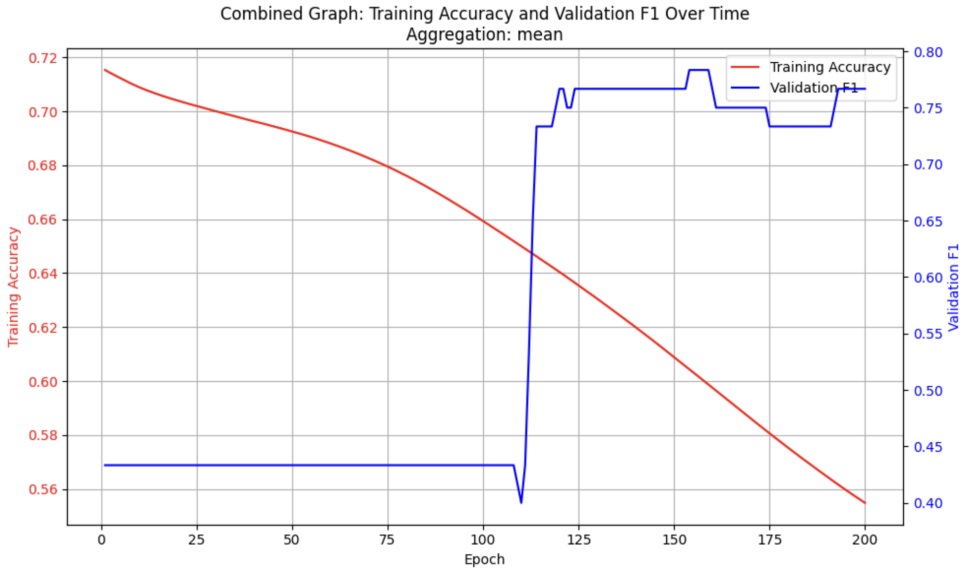
Graph 1: t-SNE visualisation of final layer node embeddings, labelled by class



Graph 2: t-SNE visualisation of final layer node embeddings, labelled by class

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

The GCN implementation which trains on both graphs at once is in the Q3.2c cell of the Q3 notebook.



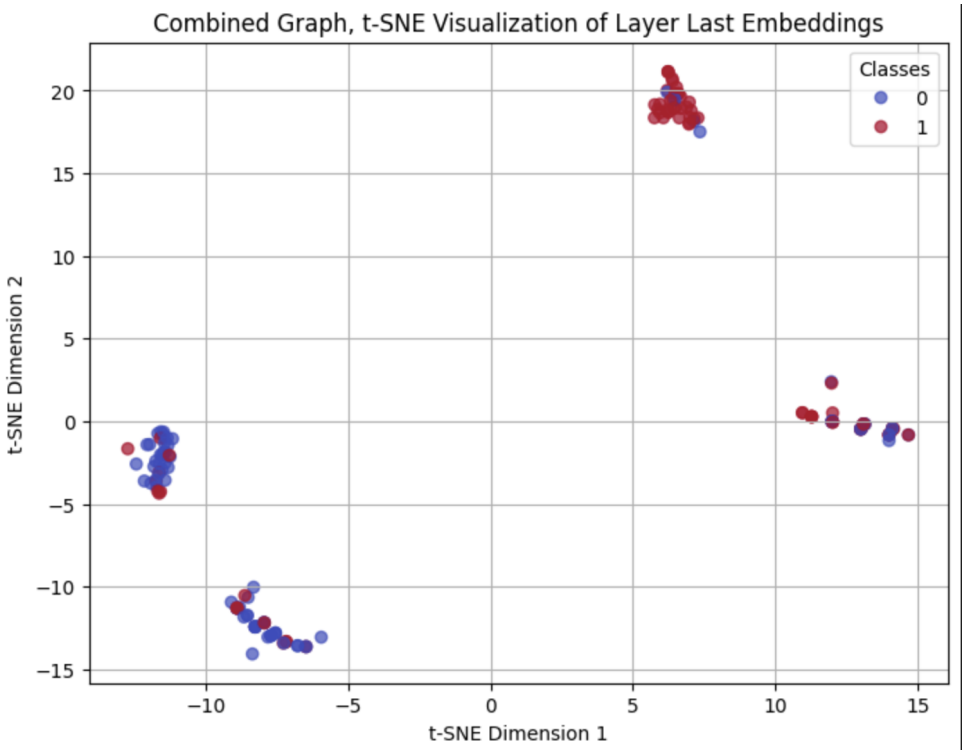
The performance of the model, when trained on $G_1 \cup G_2$, slightly outperforms training on G_1, and significantly outperforms training on G_2.

Training on G_1 alone results in a training F1 of 0.8375, and a validation F1 of 0.7333.

Training on G_2 alone results in a training F1 of 0.5125, and a validation F1 of 0.3667.

When trained on $G_1 \cup G_2$, the model has a training F1 of 0.7625, and a validation F1 of 0.7667.

3.2.d Joined vs. Independent Training



The observed performance differences when training on the combined graph $G_1 \cup G_2$ compared to training on G_1 or G_2 alone can be explained by the complementary information and structural diversity provided by combining the two graphs. Training on G_1 alone results in a relatively high validation F1 score of 0.7333, indicating that G_1 has informative topology and feature distributions that generalize well. In contrast, training on G_2 alone leads to a much lower validation F1 score of 0.3667, despite a high training F1 of 0.8375. This significant discrepancy suggests that the model overfits when trained only on G_2. This is due to the homogeneous structure in G_2, which encourages the model to memorize patterns instead of generalizing them, and feature redundancy /overlap that reduces the model's ability to distinguish between classes during validation.

When training on the combined graph $G_1 \cup G_2$, the validation F1 score improves to 0.7667, surpassing the performance of training on either graph individually. Incorporating G_2 introduces additional structural variations or complementary feature patterns that prevent the model from overfitting to the simpler patterns in G_1. The combined dataset acts as a form of regularization, encouraging the model to learn more generalizable embeddings. This effect helps the model overcome the challenges presented by the less separable feature distributions in G_2 by leveraging the richer and more distinguishable patterns present in G_1.

Additionally, the lower training F1 score of 0.7625 observed when training on the combined graph, compared to 0.8375 when training only on G_1, indicates that the model is balancing optimization objectives. It sacrifices a perfect fit on the more learnable G_1 to achieve better generalization across both datasets. This trade-off reflects the model's need to accommodate the complexity introduced by G_2, forcing it to learn patterns that

generalize well rather than memorizing those specific to G_1.

When training on two graphs simultaneously, several implementation options exist. My chosen approach in the provided code combined the adjacency matrices using torch.block_diag, creating a block-diagonal adjacency matrix. This treats the two graphs as disconnected components within a single larger graph, ensuring no cross-graph edges exist. Features and labels are concatenated along the node dimension. This allows the model to process both graphs during each training epoch, capturing patterns that generalize across them. The benefit of this is its' simplicity and ability to capture shared patterns across graphs. However, it assumes no beneficial inter-graph node relationships.

Another approach is to modify the model architecture. For example, a multi-branch GCN could be implemented where each branch processes a different graph independently. The embeddings from both branches could then be combined, either via concatenation or attention-based mechanisms, before passing them through a final classifier. This design would allow the model to learn graph-specific representations first, as well as shared knowledge for the final prediction.

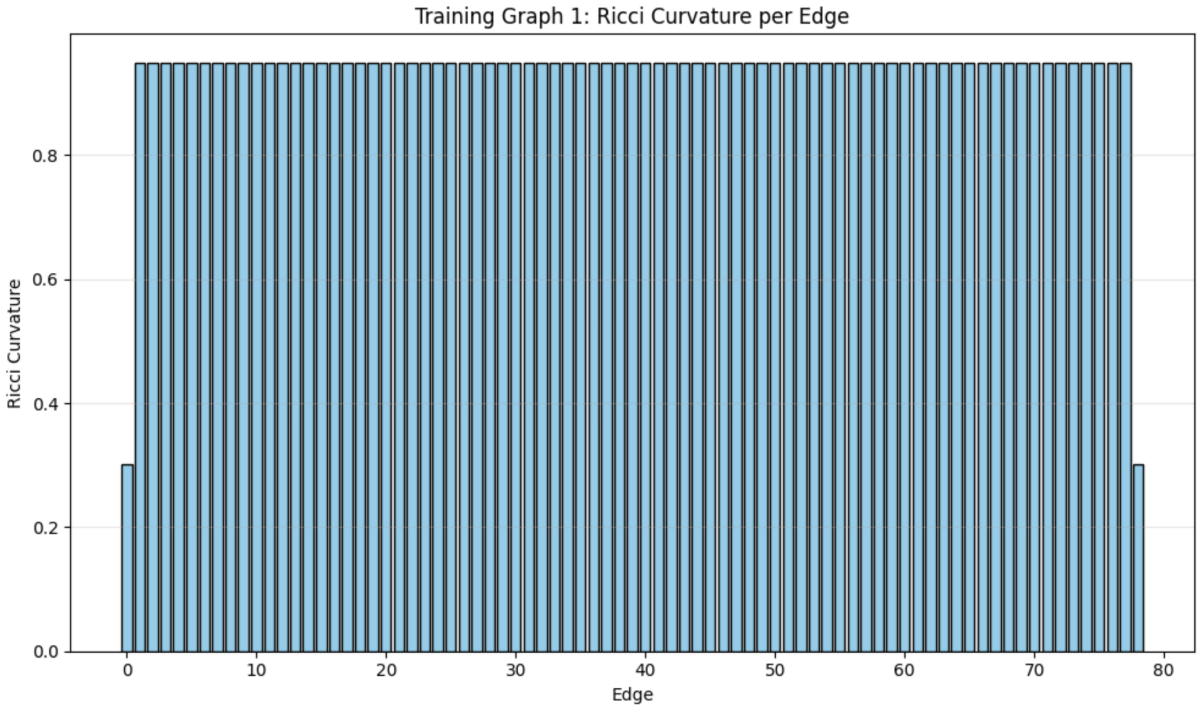
Another model modification is introducing graph-level embeddings that capture global properties of each graph. This involves pooling layers that summarize node embeddings, are useful if the graphs exhibit distinct structural properties.

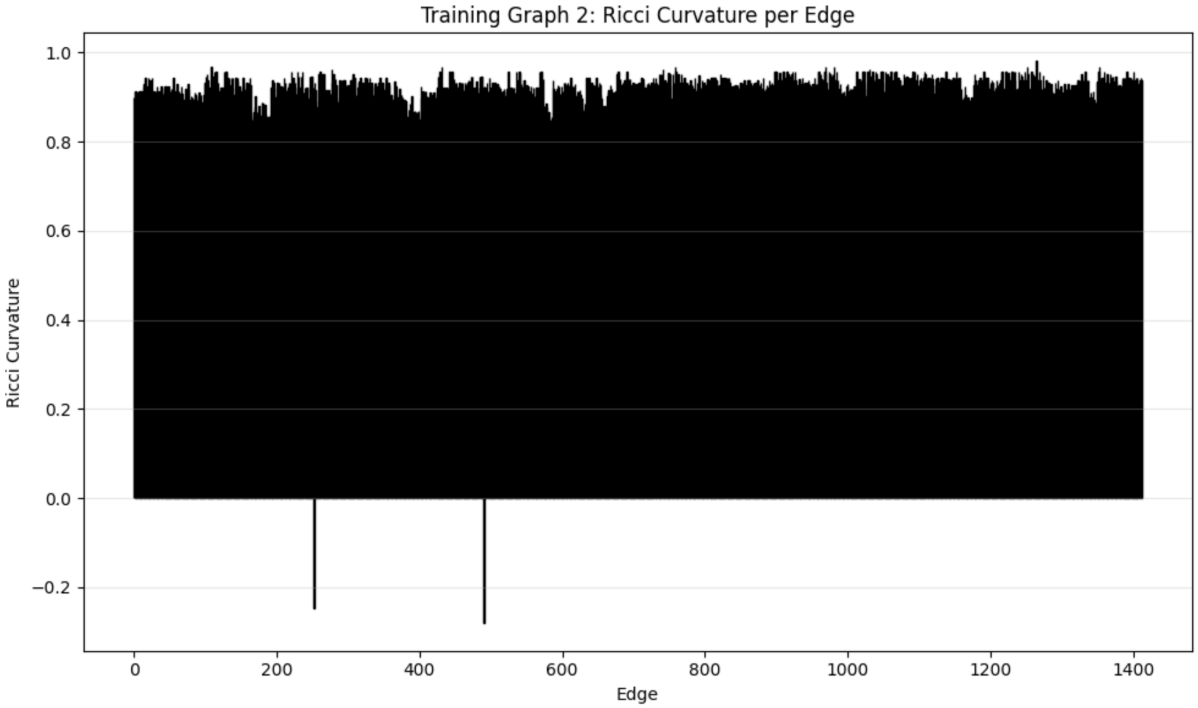
Another option is to alternate training on each graph per epoch (multi-task learning), sharing weights across the graphs but updating them separately. This strategy prevents the graph structure from dominating the learning process.

Finally, data augmentation strategies can be used. For example, adding synthetic edges based on feature similarity can introduce beneficial inductive biases. Augmentation could also include node feature perturbations or edge dropout, helping the model generalize better across different graph structures.

3.3 Topological Changes to Improve Training

3.3.a Plot the Ricci Curvature for Each Edge

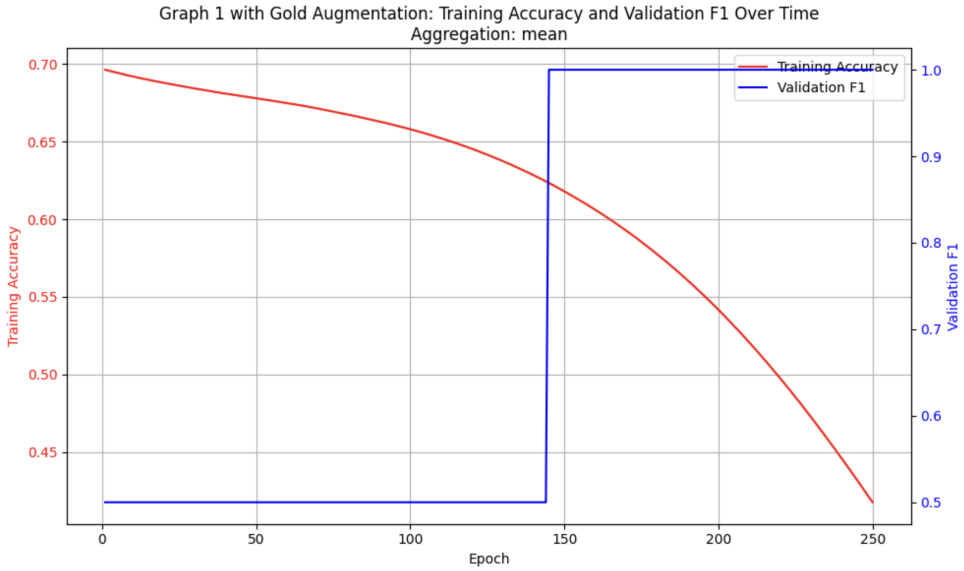


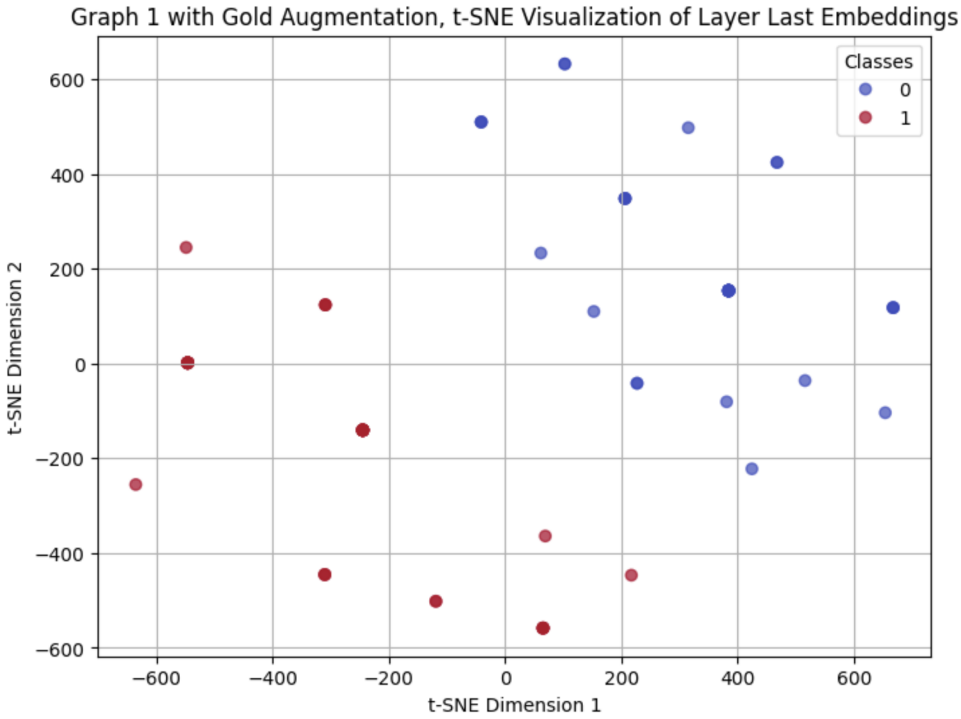


3.3.b Investigate Extreme Case Topologies

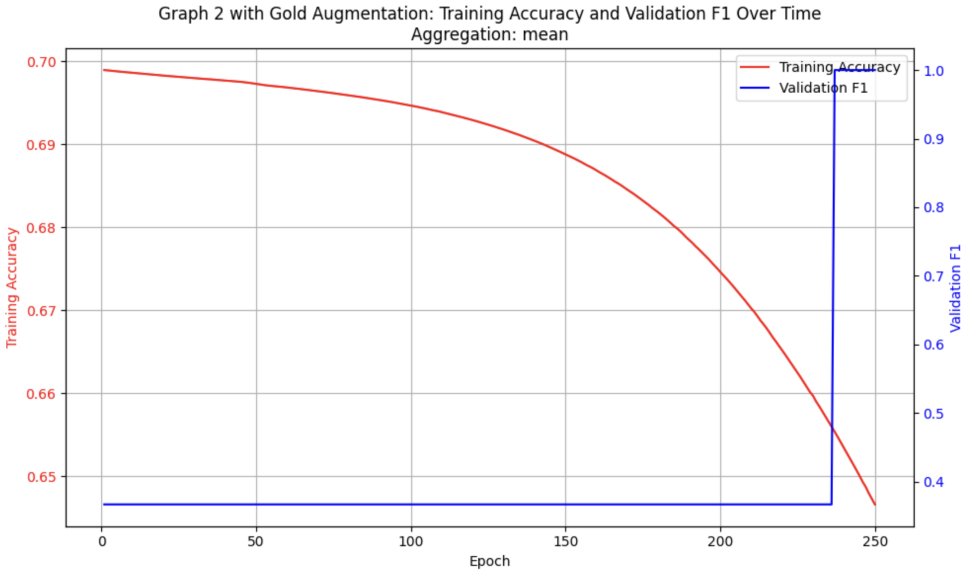
'No-Effect Topology': To modify the topology and make the GNN act like an MLP, the graph structure must be completely ignored. To implement this, set the adjacency matrix to be the identity matrix (of correct size). This ensures nodes have no neighbours so only propagate their own information to the next layer, like an MLP.

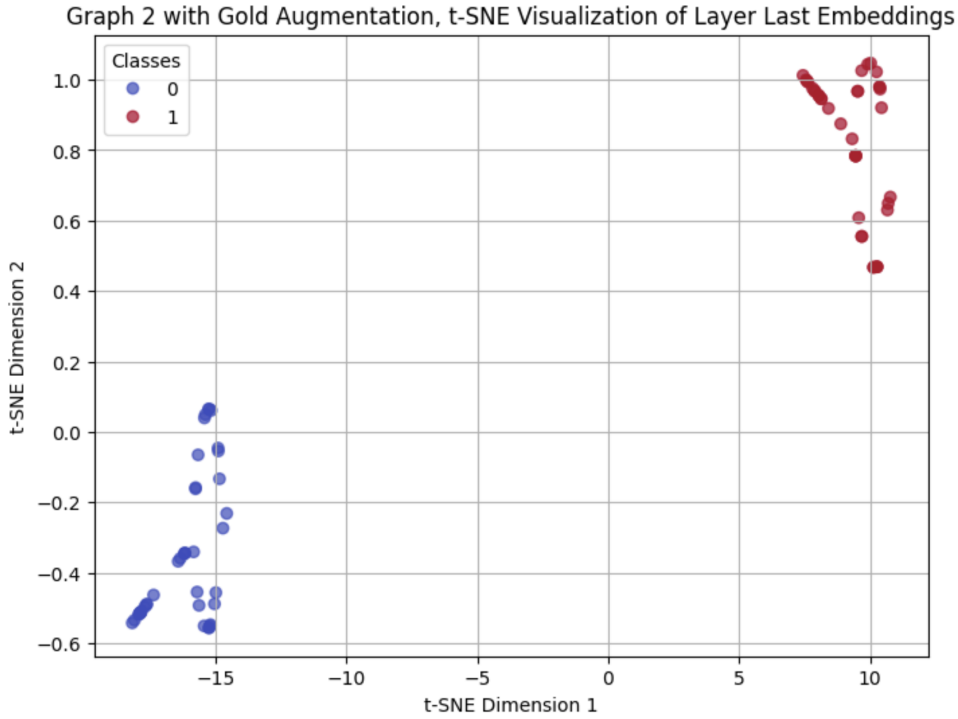
'Gold Topology': If labels are available, the ideal graph structure for optimal training and testing would be the following adjacency matrix. The adjacency matrix (A) is set as follows: A[i][j] = 1 if i and j are of the same class, and A[i][j] = 0 otherwise. This removes all inter-class links and forgets the graph structure entirely. The graph structure now directly encodes the target output, making the network optimal. This causes all nodes of the same class to be fully connected, forming complete subgraphs per class. This enhances the GNN's ability to distinguish classes by making label information directly available through the topology.





With this 'gold' topology augmentation, on Graph 1, the model achieves a perfect training F1 of 1.0 and a validation F1 of 1.0. This is in contrast to the non-augmented training F1 of 0.8375 and validation F1 of 0.7333, on Graph 1. From the t-SME, it is evident that the model is able to perfectly separate the two classes, using it's learned embeddings.

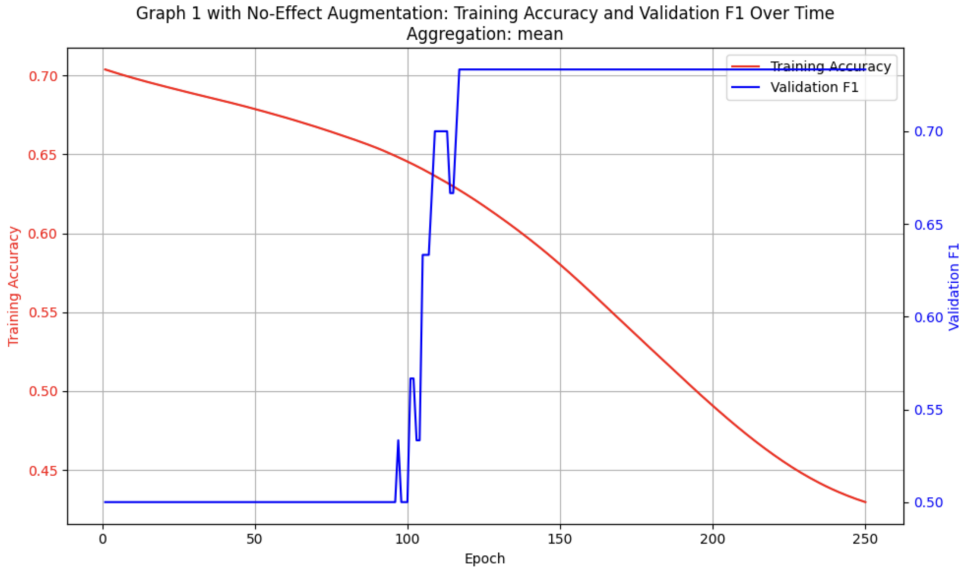


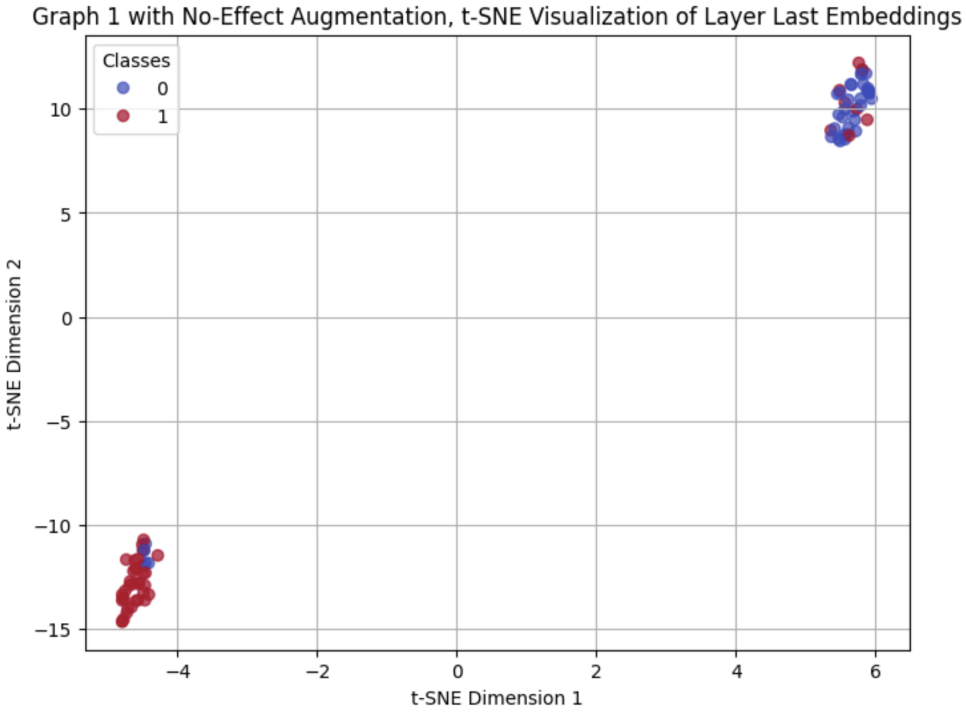


With this 'gold' topology augmentation, on Graph 2, the model achieves a perfect training F1 of 1.0 and a validation F1 of 1.0. This is in contrast to the non-augmented training F1 of 0.5125 and validation F1 of 0.3667, on Graph 2. From the t-SME, it is again evident that the model is able to perfectly separate the two classes, using it's learned embeddings.

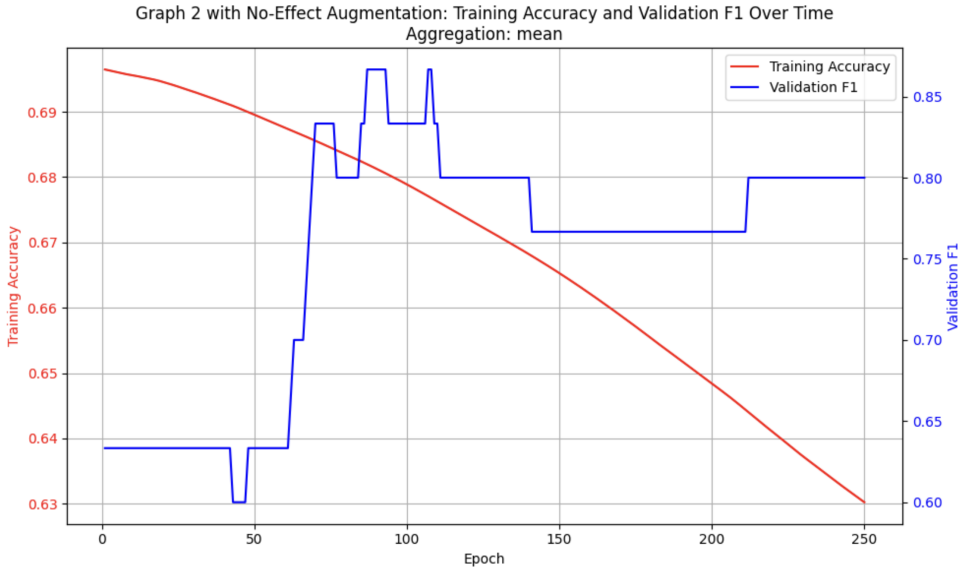
The 'gold' implementation achieves a perfect training and validation F1 score of 1.0 because the modified adjacency matrix leaks the label information into the graph topology. An edge between two nodes only exists if they are part of the same class, so during message passing, each node aggregates information exclusively from nodes of the same class, making it trivial for the model to classify them correctly. This means all nodes within a class share identical feature representations, so the classification task becomes equivalent to identifying clusters that already correspond perfectly to the ground-truth labels.

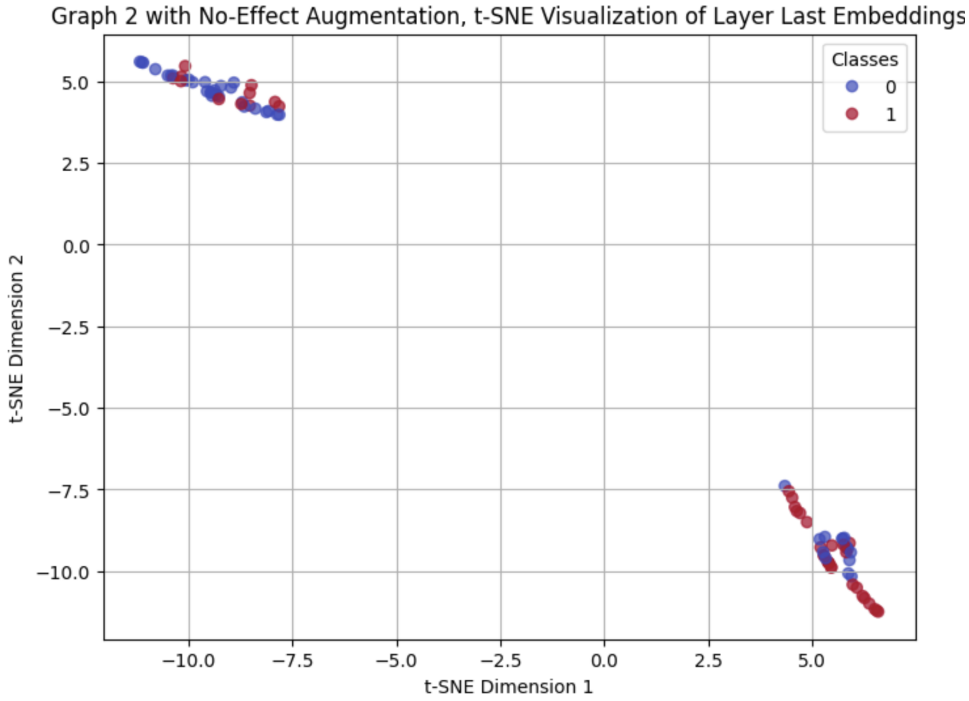
The t-SNE visualization for both graph 1 and graph 2 models shows a perfect separation between the two classes with no overlap, which occurs because the modified adjacency matrix used in the model encodes label information directly into the graph topology. t-SNE, which reduces high-dimensional embeddings into a 2D space while preserving relative distances, projects the perfectly clustered embeddings into distinct, well-separated groups.





With this 'no-effect' topology augmentation, on Graph 1, the model achieves a training F1 of 0.8375 and a validation F1 of 0.7333. This is identical to the non-augmented (control) training F1 of 0.8375 and validation F1 of 0.7333, on Graph 1. This indicates that graph 1 is a path graph (simple path), where each node is only connected to one-other node, forming a similar structure to an MLP. Nodes in graph 1 don't have neighbours, so ignoring the graph structure doesn't impact model performance at all.





With this 'no-effect' topology augmentation, on Graph 2, the model achieves a training F1 of 0.6875 and a validation F1 of 0.8. This is an improvement on the non-augmented (control) training F1 of 0.5125 and validation F1 of 0.3667, on Graph 2. This improvement indicates that graph 2 is a highly-dense graph, with nodes sharing features (much of which is noise) between neighbours. The model here has fitted to it's training data well.

3.3.c Improving Graph Topology for Better Learning

My first topology modification was Edge Dropout. This randomly removes a small percentage of edges between nodes, by setting the relevant entry in the adjacency matrix to

0.

My second topology modification was Edge Addition (Top-K Similar Nodes) This adds edges between nodes with similar feature vectors (using cosine similarity). The metric used to decide 'similar features' was cosine similarity of node features. This was calculated by normalizing each row to have unit L2 norm, and taking the dot product of all pairs. This metric (for each pair) gives the cosine similarity between the nodes.

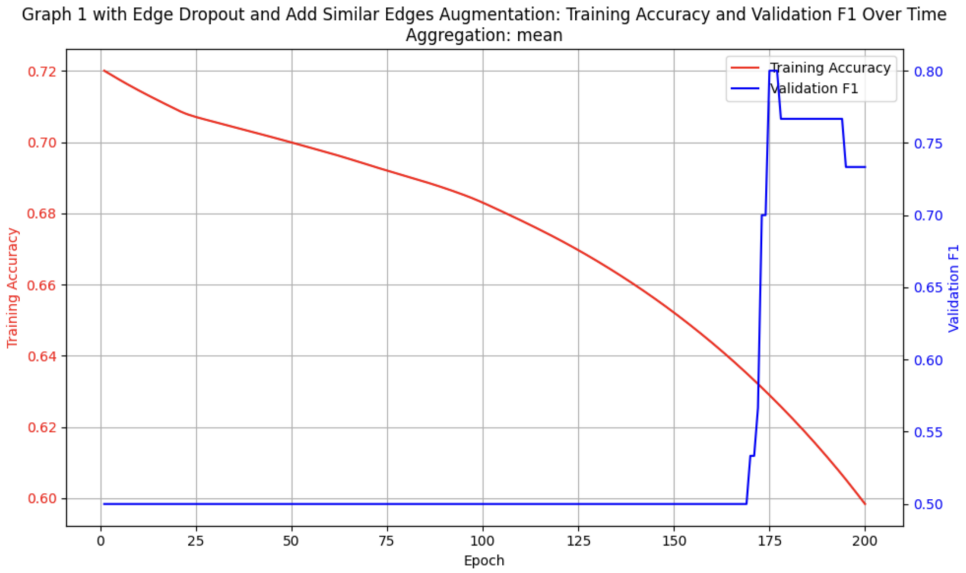
Edge Dropout helps the model generalize by preventing overfitting to specific connections. The model is exposed to multiple topological variations, thus improving adaptability.

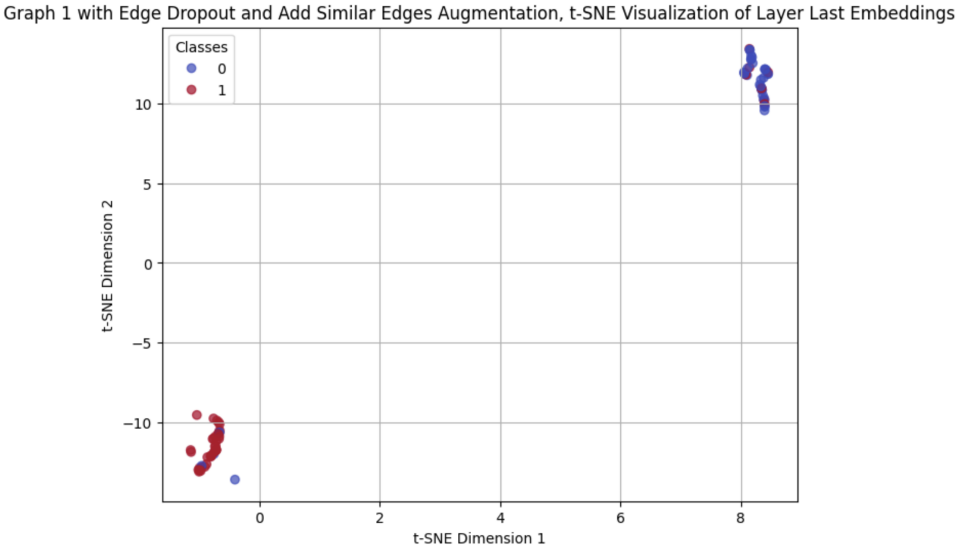
Edge Addition (adding edges between similar nodes) introduces new relational patterns, aiding classification. It provides additional connectivity based on feature similarity, enriching the graph structure and improves message-passing.

Both augmentations align with graph-based regularization, making predictions more robust.

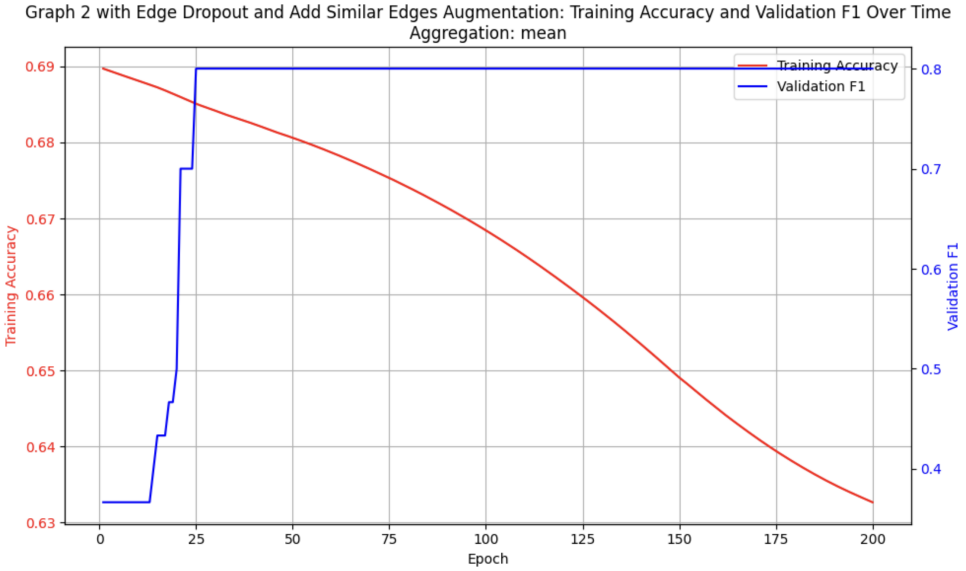
Both improvements were motivated by the previous part's "Extreme Case Topology", where only nodes of the same class had an edge between them. I aimed to implement a less extreme version of this, by removing some edges and adding edges between similar nodes.

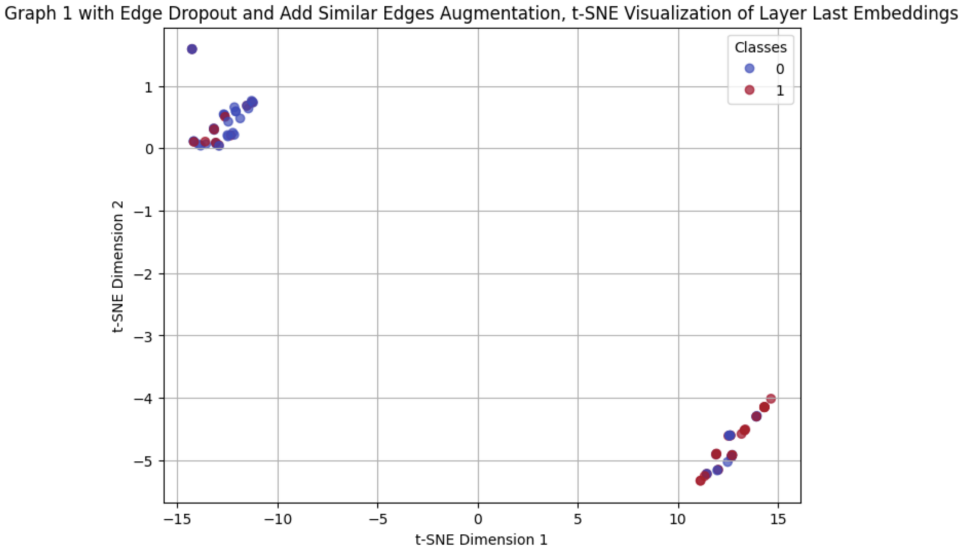
When training/testing the model, I applied both augmentations together. The results are below.





With the 'edge dropout' and 'edge addition' topology augmentations, on Graph 1, the model achieves a training F1 of 0.8500 and a validation F1 of 0.73333. This is similar to the non-augmented (control) training F1 of 0.8375 and validation F1 of 0.7333, on Graph 1. This is because that graph 1 is a path graph (simple path), where each node is only connected to one-other node. When nodes are removed and then added, the structure is still similar to a path graph (with perhaps a few small cycles). Thus, these augmentations make little difference to model performance in graph 1.





With the 'edge dropout' and 'edge addition' topology augmentations, on Graph 2, the model achieves a training F1 of 0.6875 and a validation F1 of 0.8. This is an improvement on the non-augmented (control) training F1 of 0.5125 and validation F1 of 0.3667, on Graph 2. This improvement indicates that graph 2 is a highly-dense graph, with nodes sharing features (much of which is noise) between neighbours. By removing random edges and instead adding edges between similar-feature nodes, some of this noise is removed. This allows the model to fit it's training data well.