DGL 2025 Coursework 1

Steven Chen

CID: 01861172 shc20@ic.ac.uk

Department of Computing
Imperial College London

February 19, 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

Code for different Aggregation Method

```
# Graph aggregation
if graph_aggregation_method == 'sum':
    graph_output = H2.sum(dim=0)
elif graph_aggregation_method == 'max':
    graph_output = H2.max(dim=0).values
else: # Default to mean
    graph_output = H2.mean(dim=0)

graph_output = torch.sigmoid(graph_output.squeeze())
```

1.1.b Graph-Level Training

Mean: Epoch 20, Training Loss: 0.53, Train Accuracy: 0.75, Validation Accuracy: 0.50, Train F1-Score: 0.45, Validation F1-Score: 0.33

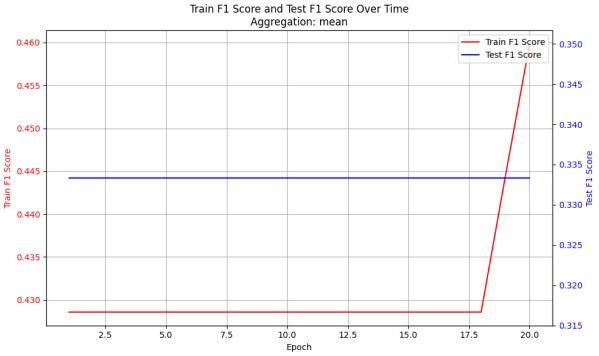

Figure 1: Mean

Sum: Epoch 20, Training Loss: 0.33, Train Accuracy: 0.77, Validation Accuracy: 0.50, Train F1-Score: 0.53, Validation F1-Score: 0.33

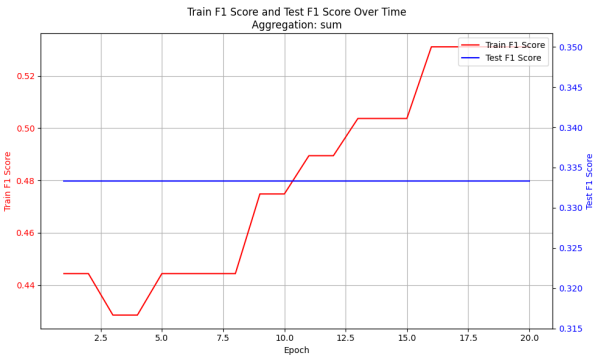

Figure 2: Sum

Max: Epoch 20, Training Loss: 0.43, Train Accuracy: 0.76, Validation Accuracy: 0.51, Train F1-Score: 0.65, Validation F1-Score: 0.49

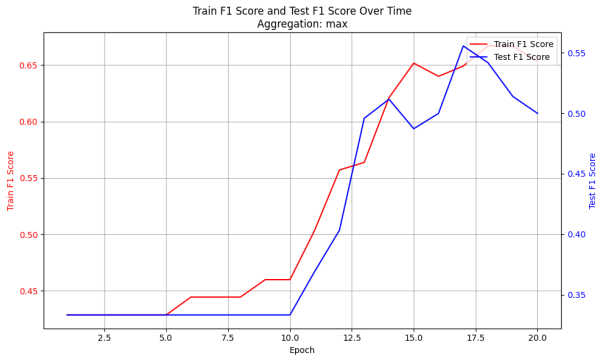

Figure 3: Max

1.1.c Training vs. Evaluation F1

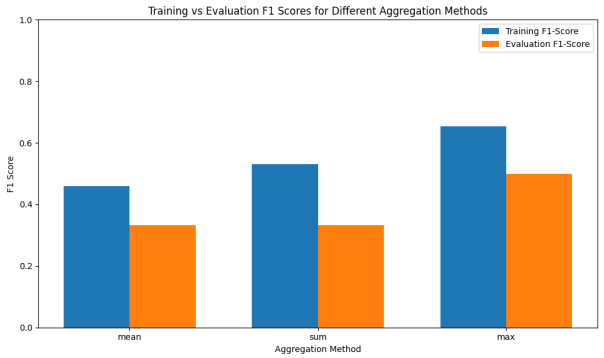

Figure 4: Compare Aggregation Method

Mean: Training F1-Score: 0.4598 Evaluation F1-Score: 0.3333

Sum: Training F1-Score: 0.5311 Evaluation F1-Score: 0.3333

Max: Training F1-Score: 0.6532 Evaluation F1-Score: 0.4999

The max aggregation method had the best training and evaluation F1 score among the three. One possible explanation of why max is the best function for aggregation is because it is scale-invariant, where the output would not change based on the number of inputs.

1.2 Analyzing the Dataset

1.2.a Plotting

1.2.b Training Set

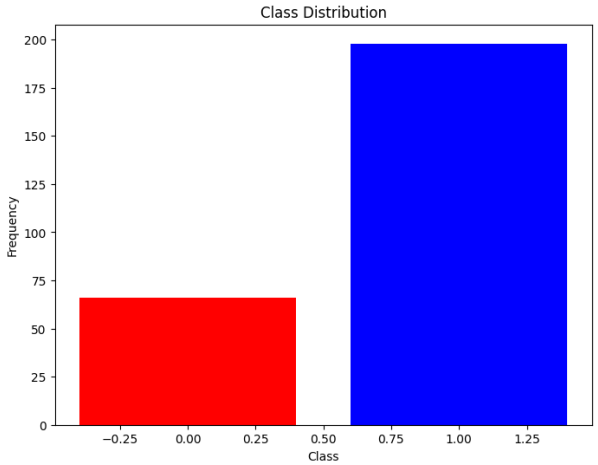

Figure 5: Train Set Class Distribution

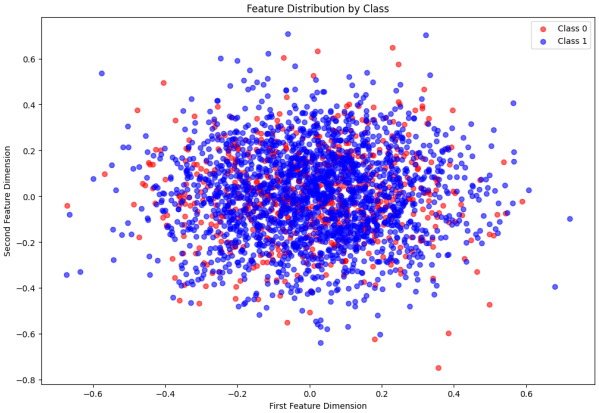

Figure 6: Training Set Feature Distribution

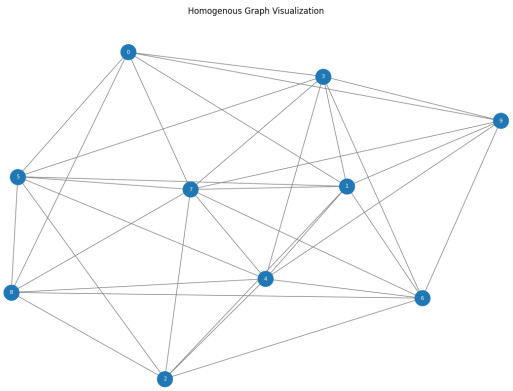

Figure 7: Training Set Label 0

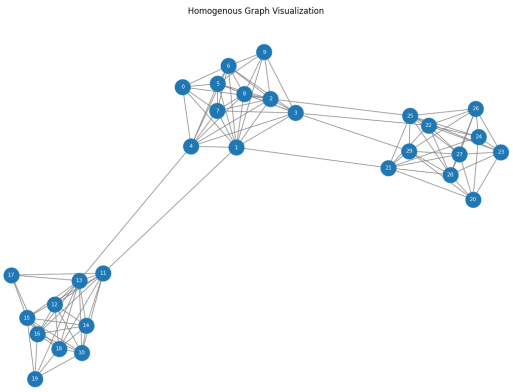

Figure 8: Training Set Label 1

1.2.c Evaluation Set

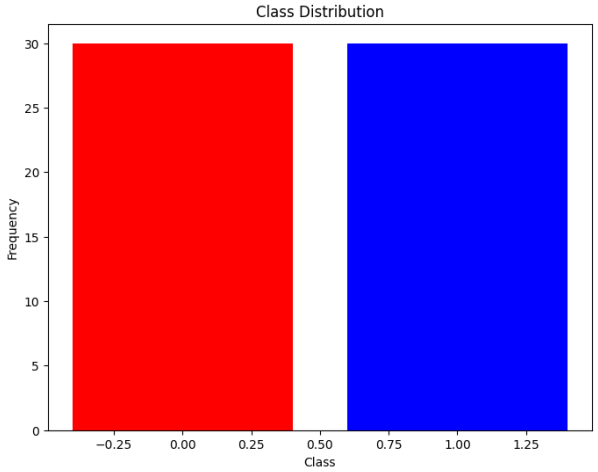

Figure 9: Evaluation Set Feature Distribution

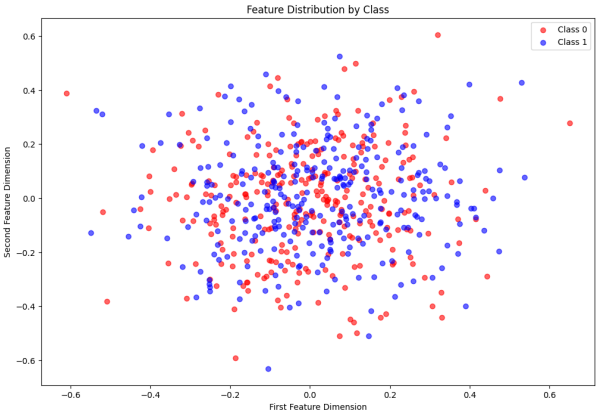

Figure 10: Evaluation Set Feature Distribution

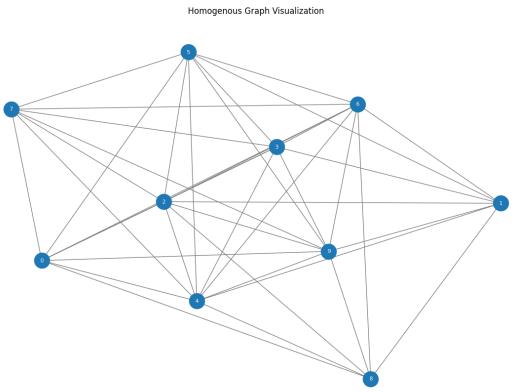

Figure 11: Evaluation Set Label 0

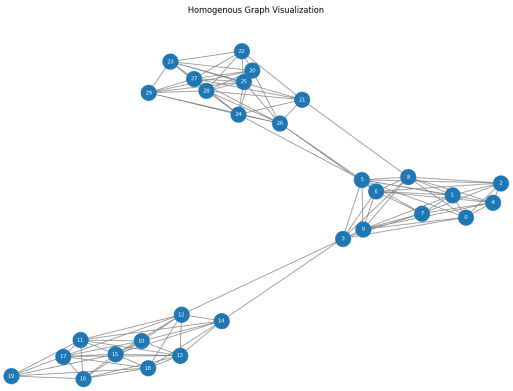

Figure 12: Evaluation Set Label 1

1.2.d Discussion

Label Imbalance: The Training set had the problem of class label imbalance, where there are many more samples of one label than the other.

Features Distribution: The samples in both the training and evaluation set had a binominal-like distribution for features 1 and 2, where the distribution appears symmetrical and the mean is centred at 0. Both classes heavily overlap meaning it is hard to separate the two classes using these two features. The two features seem to be independent as well, where there is no clear linear/non-linear pattern between the two features.

Graph Class: The graphs for label 0 seem to be a more fully connected graph while the graphs for label 1 seem to be a more modular graph.

1.3 Overcoming Dataset Challenges

1.3.a Adapting the GCN

I have added a ModuleList(), layers, to store the variable number of layers. I have also added a linear classification head for better result

```
# Input layer
self.layers.append(MyGCNLayer3(input_dim, hidden_dim))

# Hidden layers
for _ in range(num_layers - 2):
    self.layers.append(MyGCNLayer3(hidden_dim, hidden_dim))

# Final layer
self.layers.append(MyGCNLayer3(hidden_dim, output_dim, use_nonlinearity=False))

self.classification_head = nn.Linear(output_dim, 1)
```

In the forward pass, a for loop is used to apply all layers of convolution

```
for layer in self.layers:
    H = layer(H, A_normalized)
    graph_embedding.append(H)
```

Tuning Hyperparameters: I have tested the number of layers in the range of 2 to 5 and tested the hidden dimension size from 1 to 32. Here are the results:

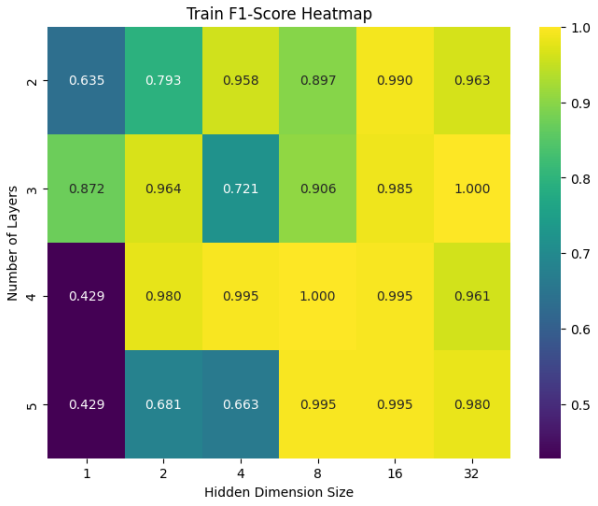

Figure 13: Training F1-score in parameter tuning

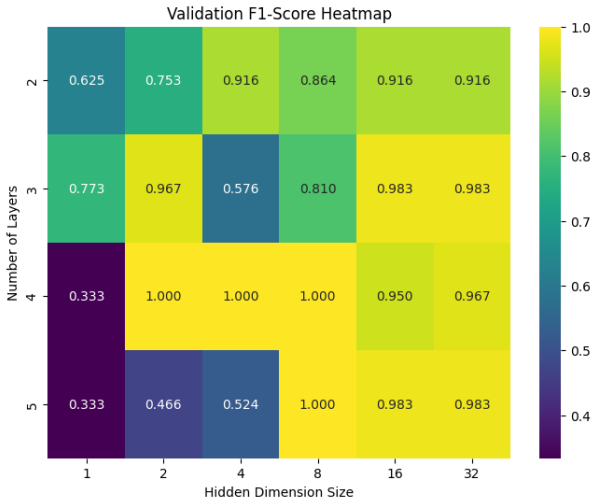

Figure 14: Evaluation F1-score in parameter tuning

I have also averaged both scores for analysis

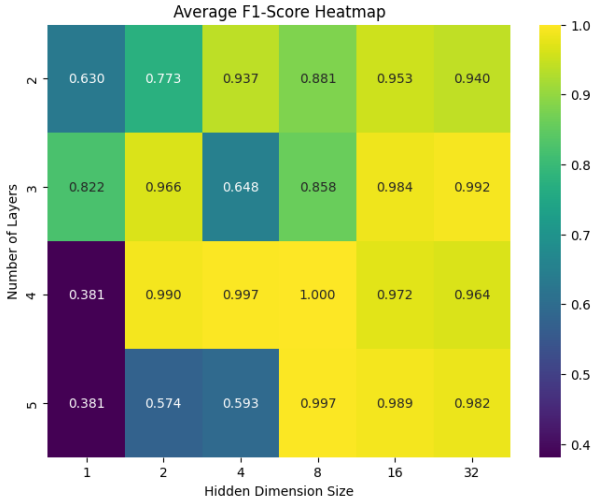

Figure 15: Averaged of both score in parameter tuning

When using the max aggregation method We can see that number of layers = 4 and hidden dimension size = 8 has the highest score in both evaluation and the averaged F1-score. Therefore it is our best parameter.

1.3.b Improving the Model

Data preprocessing (Oversampling): Since the samples in the training set are imbalanced, I have tried using oversampling to match the number of label 1 samples to label 0 samples.

```
diff = len(class_1_samples) - len(class_0_samples)

# Randomly duplicate samples from class 1 to balance the dataset
if diff > 0:
```

```
        class_0_samples.extend(random.choices(class_0_samples, k=diff))
    Else:
        class_1_samples.extend(random.choices(class_1_samples, k=-diff))

    # Combine the balanced samples
    train_data = class_0_samples + class_1_samples
    random.shuffle(train_data)
```

Data preprocessing (Feature Reduction): In the dataset analysis we identify that features 1 and 2 are independent and have a similar distribution, so we can remove one of the features to reduce noise and computational complexity.

```
    # Remove the first feature dimension
    train_data = [(X[:, 1:], A, y) for X,A,y in train_data]
    eval_data = [(X[:, 1:], A, y) for X,A,y in eval_data]
    input_dim = input_dim - 1
```

Loss Function (Weighted loss function): Since one class is significantly more frequent than another, a standard loss function might lead to biased predictions. I experimented with a weighted loss function so that the minority class contributes more to the overall loss.

```
    weight = torch.tensor(class_0_samples_count / class_1_samples_count, dtype=torch.
```

Hyperparameter Tuning: I have tested using different aggregation methods and output dimensions in the range of 1 to 32.

1.3.c Evaluating the Best Model

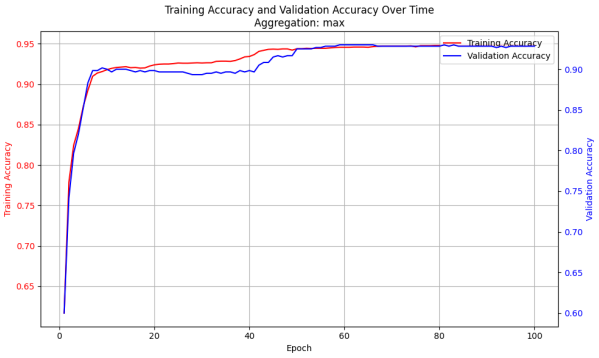

Figure 16: Best Model Accuracy over 100 epoch averaged over 10 runs

1.3.d Final Analysis and Explanation

Data preprocessing (Feature Reduction): In this dataset, features 1 and 2 exhibit very similar statistical distributions, suggesting that both features capture nearly the same underlying information. Since they are independent, removing one of these similar features reduces the overall noise in the data and decreases computational overhead during model training.

Oversampling and weighted Loss Function: Feature reduction and oversampling are experimented with. The result shows that it is best used together:

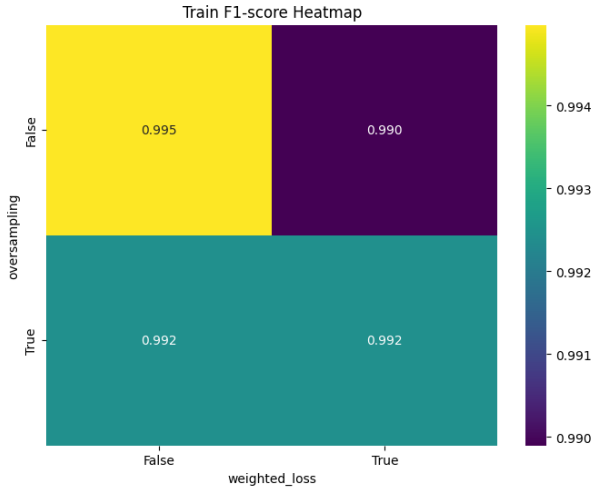

Figure 17: Train F1-score in oversample and weighted loss

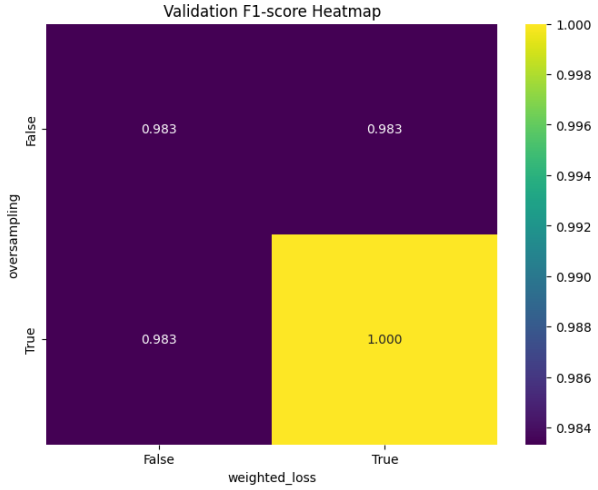

Figure 18: Evaluation F1-score in oversample and weighted loss

Explaination: Oversampling and weighted loss are both good techniques for handling imbalanced datasets. Oversampling increases the number of minority class samples ensuring that the model sees enough examples from that minority class while a weighted loss function assigns a higher penalty to misclassifications of minority class instances. Together, they help reduce the bias toward the majority class and can mitigate issues like overfitting.

Hyperparameter Tuning: I have tested pairs of related hyperparameters (e.g. (num of layers, hidden dimension size), (aggregation method, output dimension size), (oversampling, weighted loss function)), and analyse the train and eval F1-score. This allows parameter tuning without too much combination

This is the result of tuning the aggregation function and output dimension:

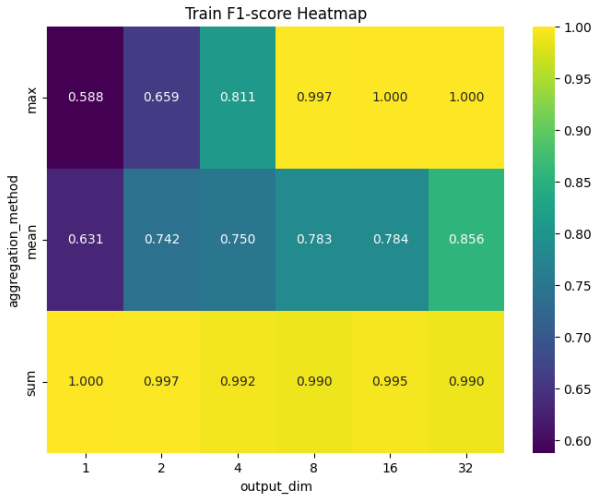

Figure 19: Train F1-score in aggregation function and output dimension

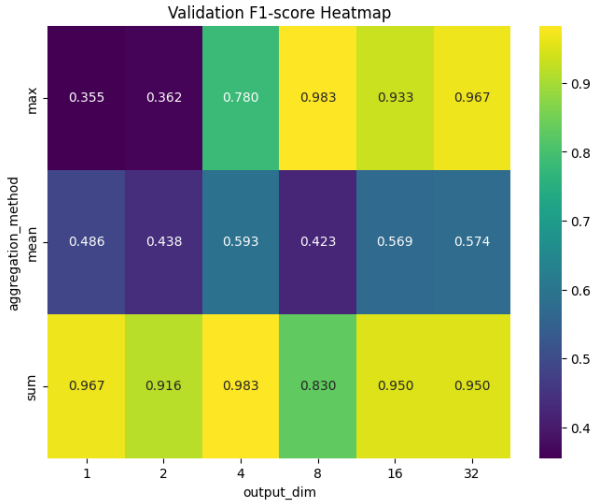

Figure 20: Evaluation F1-score in aggregation function and output dimension

The best hyperparameter I got is:

```
aggregation_method = "max"
hidden_dim = 8
num_layers = 4
output_dims = 8
oversampling = True
weighted_loss = True
feature_reduction = True
```

2 Node Classification in a Heterogeneous Graph

2.1 Dataset

2.1.a Problem Challenge

It is a challenging problem since each node type has a different feature dimension, thus it is not trivial to perform aggregation with the neighbourhood using the standard aggregation method (i.e. add up the features space in the neighbourhood).

2.1.b Real-World Analogy

We have 2 node types and 2 class labels.

A real-world example of this scenario could be a social network of individuals where:

Node Types: Male & Female

Edges (Relationships): A connection between two nodes means they are friends

The task is to identify whether a person is single or married.
Class Labels (Marital Status of Each Person): Single or Married

Gender of Friends on Marital Status: Marital status can be influenced by the gender of their friends. For example, if a male has many female friends, he might be less likely to be married.

2.1.c Interpretation of the Dataset: Plotting the Graph

The graph is dense and highly connected, with connections between different node types and class labels

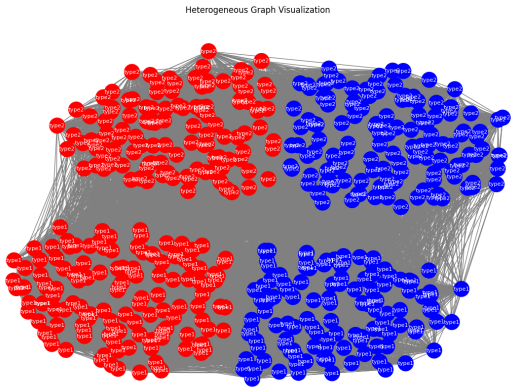

Figure 21: Heterogenous Graph

The class type and class distribution are fairly evenly spread, where the 4 categories have about 120 nodes.

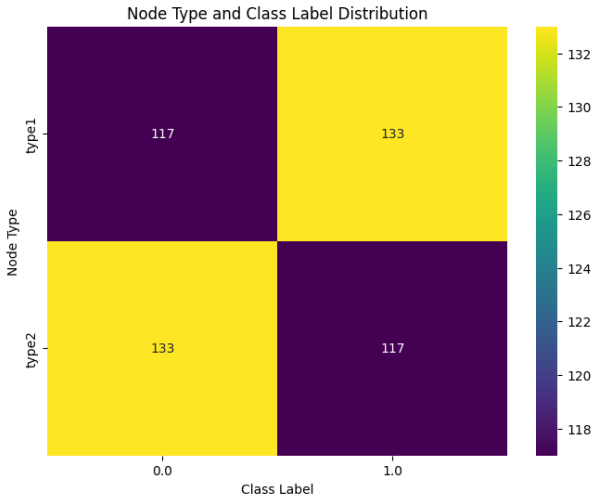

Figure 22: Node Type and Class Distribution

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

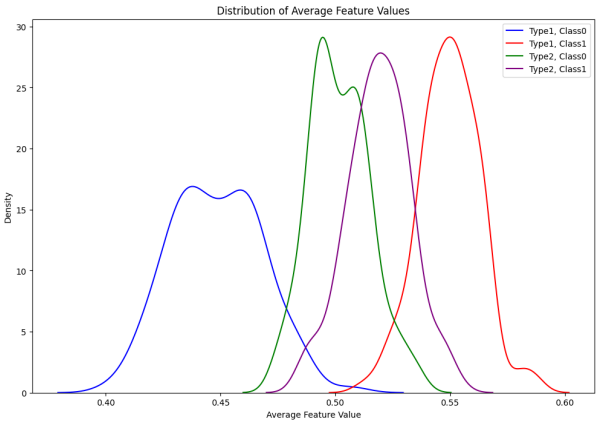

Figure 23: Node Feature Distribution

2.1.e Interpretation of the Dataset: Discussion

Distribution: The average feature values for Class 1 nodes exhibit a smooth distribution around their mean but display greater variability further from the mean.

In contrast, the average feature values for Class 0 nodes show two peaks near the mean, with a smooth distribution in the regions further from the mean.

Values: Type 1 class 0 nodes have a much lower average feature and larger variability than the other 3. Additionally, all the categories have different average feature mean, but Type 1 class 1, Type 2 class 0 and Type 2 class 1 have more overlapping.

2.2 Naive Solution: Padding

2.2.a Limitations of Naive Solution

Loss of semantic meanings: Different node types may have distinct semantic meanings for their features, the semantic meaning of the first feature vector of one node type may not match the same semantic meaning with the other. A GNN trained on such misaligned features may fail to learn meaningful representations. This also causes GNN to struggle to learn relationships between different node types.

Wasted Memory: The zero-padded values take up memory space but don't contribute to meaningful computations or learning. This can be especially problematic if there is a larger difference in the feature dimension size of the node types in a larger dataset.

2.3 Node-Type Aware GCN

2.3.a Implementation

I have reordered the adjacency matrix so that it is sorted by node type, It will then be useful to split the adjacency matrix in the forward pass.

```
def sort_A_by_type(A, mapping):
    """
    Sorts the adjacency matrix A such that all type1 nodes come before type2 nodes.
    """
    type1_indices = [i for i in range(len(mapping))
    if mapping[i]["node_type"] == "type1"]

    type2_indices = [i for i in range(len(mapping))
    if mapping[i]["node_type"] == "type2"]

    sorted_indices = type1_indices + type2_indices

    A_sorted = A[sorted_indices, :][:, sorted_indices]
```

In the forward pass, I apply the GCN layer according to the node type separately using the split adjacency matrix. Now both node types are mapped to the same hidden dimension, we can apply the second layer together using the full adjacency matrix.

```
    # Initialize hidden representations
    H1_type1 = torch.zeros(X1.shape[0], self.gcn1_type1.Omega.shape[1])
    H1_type2 = torch.zeros(X2.shape[0], self.gcn1_type2.Omega.shape[1])

    # Apply first GCN layer separately for type1 and type2 nodes
    H1_type1 = self.gcn1_type1(X1, A[:X1.shape[0], :X1.shape[0]])
    H1_type2 = self.gcn1_type2(X2, A[X1.shape[0]:, X1.shape[0]:])

    # Stack hidden representations
    H1 = torch.cat((H1_type1, H1_type2), dim=0)

    # Apply second GCN layer
```

```
H2 = self.gcn2(H1, A)

# Output layer with sigmoid activation
output = torch.sigmoid(self.linear(H2))
```

2.3.b Discussion

In this approach, I split the adjacency matrix into two separate matrices, each representing the relationships between one of the node types. This allows each node type to independently map to the same dimensionality, fully utilizing its own feature space.

The key reasoning behind this decision is to avoid the potential loss of semantic meaning that can occur when node types are treated together in a padded feature vector. By mapping each node type separately through its own set of parameters, the model ensures that each node type's semantics are preserved effectively.

Additionally, this approach avoids computational waste associated with zero-padding, as no unnecessary padding vectors are introduced.

Advantages and Limitations: The main advantage of this implementation is the ability to handle both node types separately, allowing them to be interpreted in their own context. By having distinct layers and parameters for each node type, the model can better capture the individual features and relationships of each type. Ultimately, the full adjacency matrix is used for message passing between the node types, ensuring that information can be exchanged while still respecting the differences between node types.

However, the drawback of this approach lies in the increased model complexity. With separate parameters for each node type, there may be a larger number of parameters to optimize, potentially increasing the training time and the risk of overfitting.

Naive Approach: Average F1 over 10 runs: 0.6405

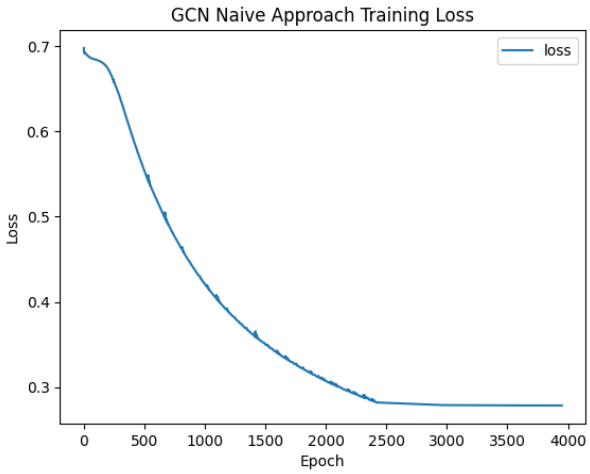

Figure 24: Naive Approach Training Loss

HeteroGNN Approach: Average F1 over 10 runs: 0.820

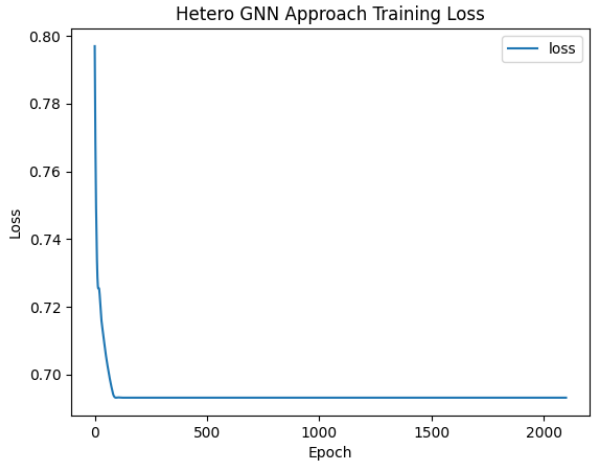

Figure 25: HeteroGNN Training Loss

By exhausively testing the combination of learning rate in rate 0.0005 to 0.01 and hidden dimension size from 2 to 512 over 10 runs. The experiment shows that the optimal values are learning rate = 0.01 and hidden dimension = 512.

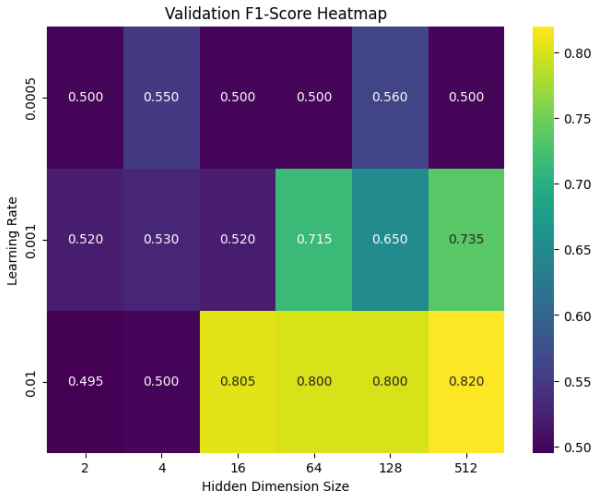

Figure 26: HeteroGNN Hyperparameter Tunning

2.4 Exploring Attention

2.4.a Implementation

I have stacked every two nodes to compute the similarity matrix using the learnable similarity weights. Then use the softmask function to compute the attention score. We get the output by multiplying the attention score and the embeddings.

```
h = torch.matmul(x, self.Omega) + self.beta

N = h.shape[0]

# Compute similarity scores
h_i = h.unsqueeze(1).expand(-1, N, -1)  # (N, N, output_dim)
```

```
h_j = h.unsqueeze(0).expand(N, -1, -1)  # (N, N, output_dim)

# Stack node feature pairs and compute similarity
sim_input = torch.cat([h_i, h_j], dim=-1)  # (N, N, 2 * output_dim)
similarity = torch.matmul(sim_input, self.similarity_weight).squeeze(-1) # (N, N)

# Apply softmask function
A_hat = adj + torch.eye(N, device=x.device)  # Add self-connections
mask = (A_hat > 0).float()  # Create mask for adjacency matrix
S = similarity.masked_fill(mask == 0, float('-inf'))  # Set masked values to -inf

# Softmax over each column
S_softmax = F.softmax(S, dim=1)

# Apply attention weights to node features
output = torch.matmul(S_softmax, h)  # (N, output_dim)
```

2.4.b Discussion

Attention-based aggregation allows selectively weighing the importance of neighbouring nodes when updating a node's representation.

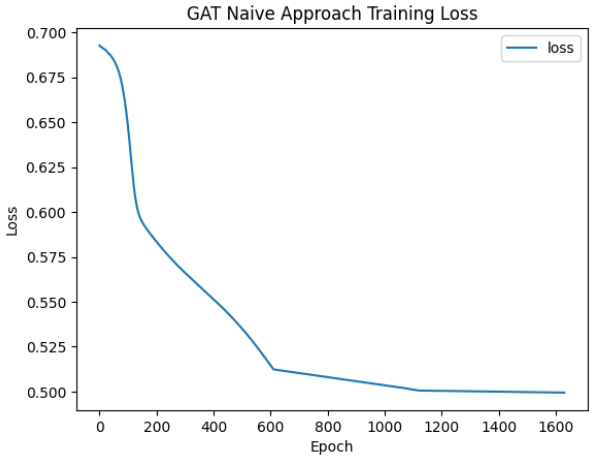

Figure 27: Attention Model Loss Curve

I have tested the learning rate in the range of 0.01 to 0.0005 and hidden dimension size in the range of 2 to 512 over 10 runs. The best parameter I got is learning rate = 0.001 and hidden dimension size = 128

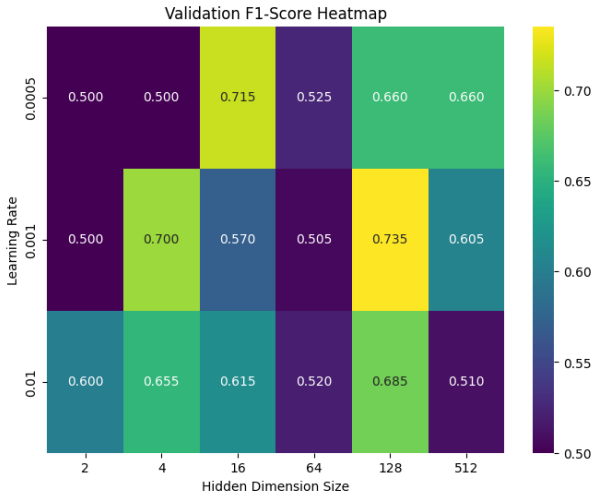

Figure 28: Hyperparameter Tuning for Attention Model

2.5 Overall Discussion

Naive GCN has the worst results among the three (F1 = 0.64). Naive GAT performs better (F1 = 0.74) than Naive GCN but is still worse than HeteroGCN. HeteroGCN performs best out of the three (F1 = 0.80).

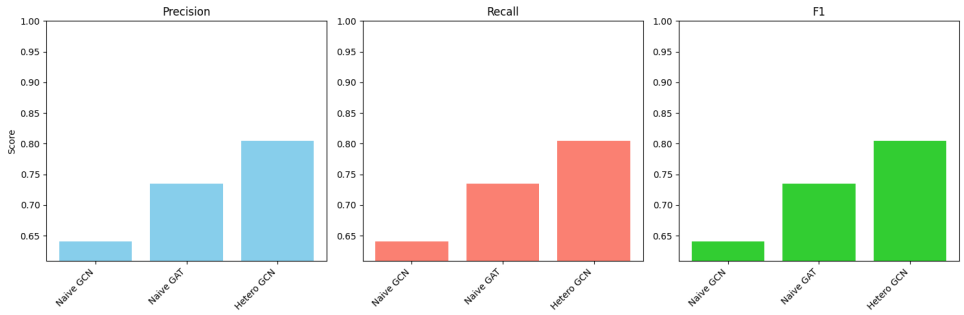

Figure 29: Compare Three Models in Precision, Recall and F1

Node-Type Aware GCN Outperform naive GCN: Naive GCN treats all node types the same way. This can lead to suboptimal learning because different node types often have distinct feature distributions. Node-Type Aware GCN applies separate layers to different node types, ensuring that each type of node has a dedicated transformation function suited to its properties. This enhances expressiveness and allows for better feature extraction.

Attention-based aggregation Outperform naive GCN: Naive GCN uniformly aggregates information from neighbouring nodes, which means all neighbours contribute equally to the node's updated representation. Attention-based GNN introduces learnable attention weights, allowing the model to focus more on influential neighbours while down-weighting less relevant ones. By prioritizing significant relationships, it leads to better results.

3 Investigating Topology in Node-Based Classification Using GNNs

3.1 Analyzing the Graphs

3.1.a Topological and Geometric Measures

Node Degree
Definition: The degree of a node in a graph is the number of edges connected to it. In the case of an undirected graph, it is the count of edges. In the case of a directed graph, it is divided into the number of incoming edges and the number of outgoing edges.

Quantifies: It can quantify the local connectivity of a node.

Understanding of Graph Structure: It can identify hub nodes in networks, networks with high-degree and clustering tendencies

Betweenness Centrality:
Definition: Betweenness centrality is given by

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v.

Quantifies: How a node acts as a bridge within the network. A node with high betweenness centrality has a crucial role in communicating between different parts of the network.

Understanding of Graph Structure: It identifies the nodes that connect different parts of the network, which can be useful for understanding how the information spreads in the network and identifying the bottlenecks of a network.

Ollivier-Ricci Curvature:
Definition: The coarse Ollivier-Ricci curvature for a pair of nodes x, y is given by

$$\kappa(x, y) = 1 - \frac{W(m_x, m_y)}{d(x, y)}$$

where $W(m_x, m_y)$ is the Wasserstein distance (is the minimum average travelling distance that can be achieved by any transportation plan) between the ea probability distributions of m_x to m_y centered at nodes x and y respectively. And $d(x, y)$ is the shortest path distance between nodes x and y.

Quantifies: It quantifies how connected or disconnected the graph is at different points. In general, negative Ricci curvature means the edge behaves locally as a shortcut or bridge. Positive Ricci curvature of xy indicates that locally there are more triangles in the neighbourhood of x, y. [1]

Understanding of Graph Structure: It identifies bottlenecks, bridges, and edges crucial for global connectivity, which can help analyze network robustness.

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

Nodes in graph 1 have less node degree (mean = 2) while nodes in graph 2 have much higher node degree (mean = 36). This suggested that graph 2 is a more fully connected graph while graph 1 is sparse.

Most of the nodes in graph 1 have 2 node degrees and a few with 1 node degree, which might suggest that the graph is a long-chained structure

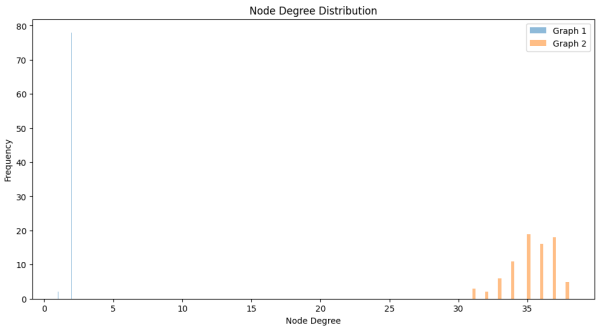

Figure 30: Node Degree of Two Graphs

Graph 1 has a more diverse betweenness centrality distribution, where the values are spread across a wide range from near 0 up to 0.5. This suggests that some nodes play a crucial intermediary role, while others have little influence. A high maximum value of 0.5 indicates that Graph 1 contains more key nodes that serve as important connectors. Overall, the spread suggested that the graph has a decentralized structure, where some nodes act as crucial intermediaries.

Almost all nodes in graph 2 have a betweenness centrality of zero or very close to zero. And a few nodes around 0.2 to 0.3. This suggests that there are very few or no "bridge" nodes, meaning the network could be highly modular with little interconnectivity between different groups.

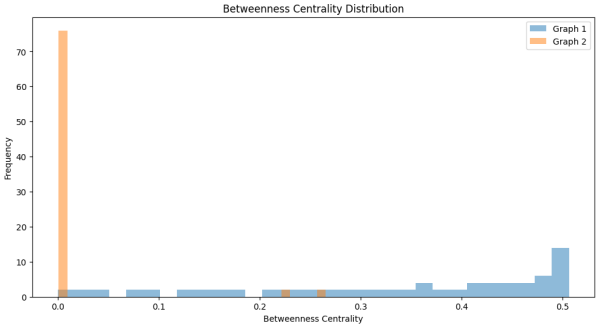

Figure 31: Betweenness Centrality Distribution of Two Graphs

Graph 1 has two low peaks, one between 0.8 and 1.0 and one small one between 0.2 and 0.4. The low Ollivier-Ricci curvature nodes suggested that the graph is not highly connected, implying the graph is sparse.

Graph 2 has two peaks, one with a wider and a larger-value Ollivier-Ricci curvature distribution, and another with at few negative values. The wider and larger-value peak suggested that a lot of nodes are highly connected and indicates that there are more triangles in the local neighbourhood of those nodes, which implies it is highly modular. The few nodes with negative Ollivier-Ricci curvatures suggest there are only a few nodes that act as a bridge, which might graph 2 is a highly modular graph with only a limited number of nodes as bridges.

The difference in height between the higher peaks of the two graphs suggested that graph 2 has much more highly connected nodes than graph 1.

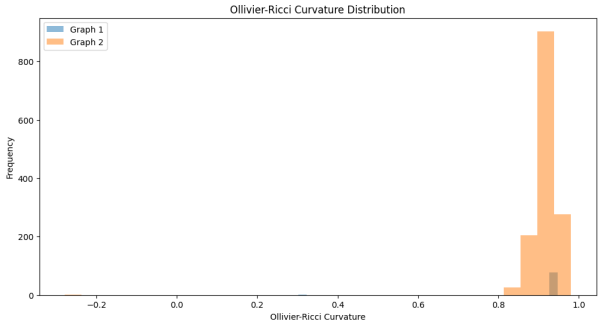

Figure 32: Ollivier-Ricci Curvature Distribution of Two Graphs

3.1.c Visualizing the Graphs

Graph 1 exhibits a long-chained structure:

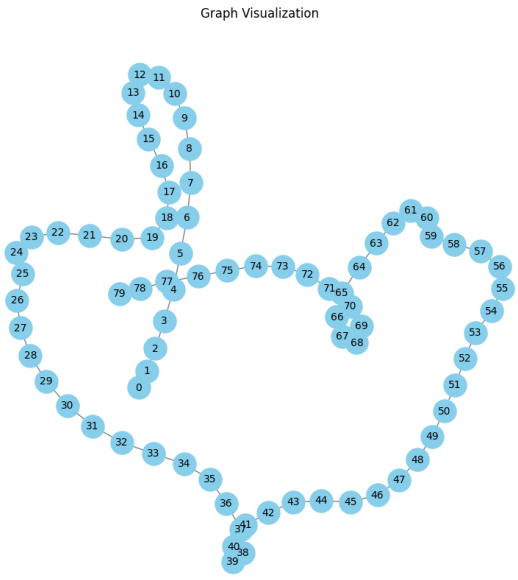

Figure 33: Graph 1 Visualisation

Graph 2 exhibits a highly modular structure with few inter-modular connections:

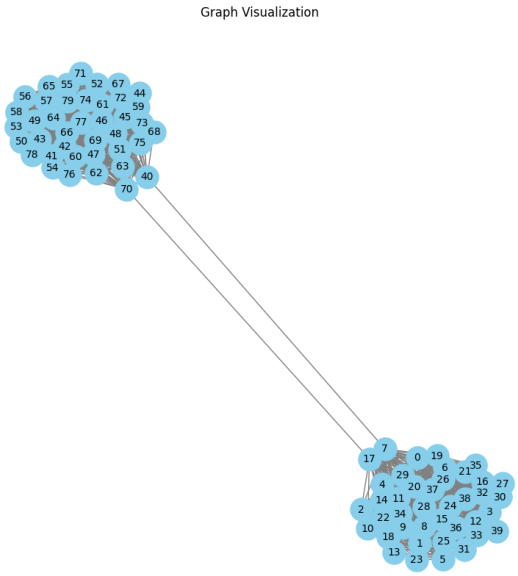

Figure 34: Graph 2 Visualisation

3.1.d Visualizing Node Feature Distributions

For graph 1, there are a few overlaps between -0.15 to -0.05, but class 0 tends to have more negative values while class 1 tends to have more positive values. There is much less overlap out of the -0.15 to -0.05 range.

For graph 1, there are a lot of overlaps between 0 to 0.15, and there are no clear separations between the two classes.

The average feature distribution of the two classes in graph 1 have less overlap than the feature distribution of the two classes in graph 2, which suggests that classification for the classes might be easier in graph 1 than in graph 2

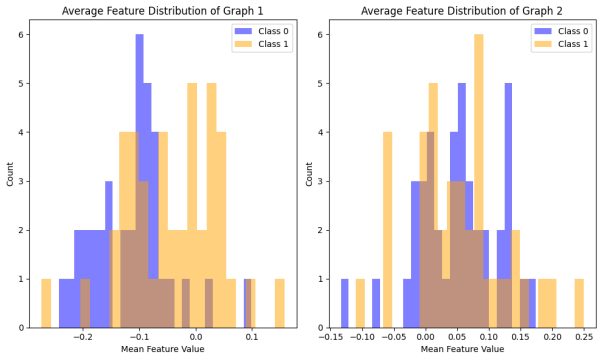

Figure 35: Average Feature Distribution of Two Graphs

3.2 Evaluating GCN Performance on Different Graph Structures

3.2.a Implementation of Layered GCN

3 layers seems to work the best in this case with a relatively high F1 score for both graphs

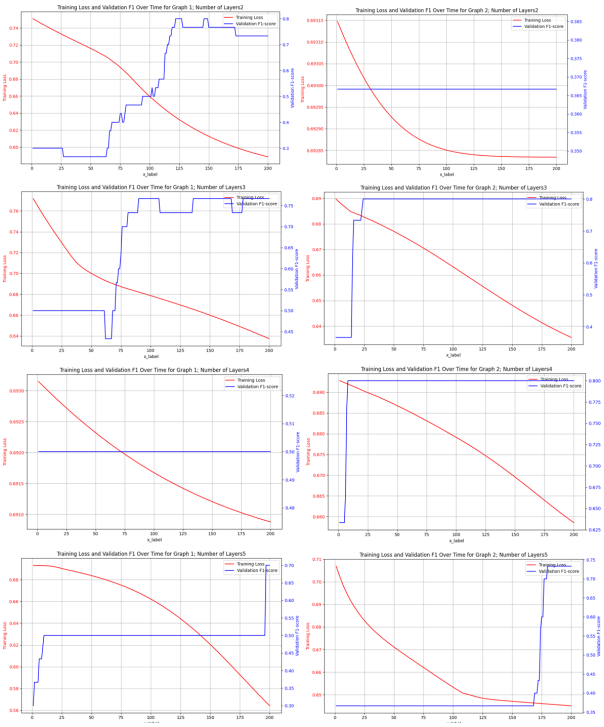

Figure 36: Training Loss and Validation F1 over time for both graphs using different number of layers

3.2.b Plotting of t-SNE Embeddings

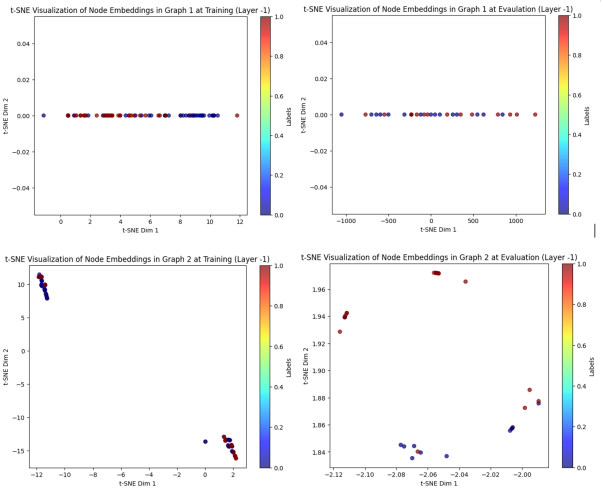

Figure 37: Node embeddings at final layer for each graph for both training and evaluation

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

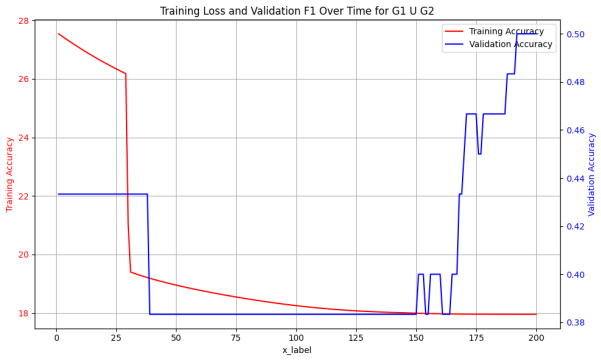

Figure 38: Training Loss and Validation F1 over time using G1 U G2

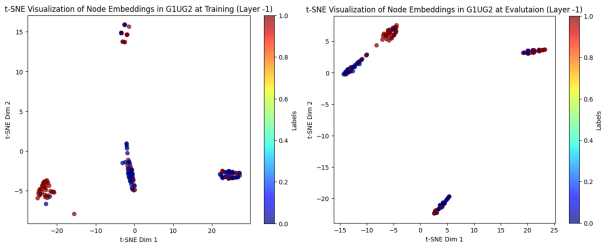

Figure 39: Node embeddings at final layer for G1 U G2 for both training and evaluation

3.2.d Joined vs. Independent Training

Performance: The average results of 10 runs of training using both joined and independent approach, it shows that independent training has a higher average F1-score (around 0.7) than joined training (around 0.6).

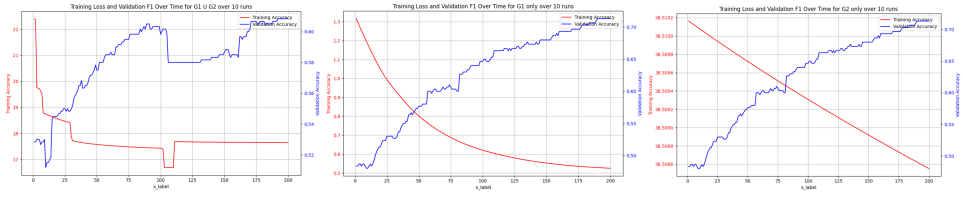

Figure 40: Joined vs. Independent Average Training Loss and Validation F1 over time over 10 runs

Embeddings: By comparing the final layer of the joined and independent training, we can see that the embeddings between the two classes in independent training could be more easily separated with a linear line, have less overlapping and have less number of clusters. This suggests that the independent training approach results in more distinct feature representations for each class and allows the model to create a linear decision boundary between classes.

In contrast, the embeddings from the joined training approach exhibit greater overlap between classes and have more than 2 clusters, making it more challenging to draw

a linear boundary that separates them. This overlap suggests that the shared training process may have introduced noise, potentially reducing its classification performance.

Hypothesis: The poorer performance of joined training is likely due to the additional noise and variability when training on two very topologically different graphs. In contrast, the better performance in independent training is likely due to the model's ability to focus on the unique structure of each graph without interference from the other, allowing the model to create clear class boundaries.

Model modifications: We could introduce graph-specific embedding layers to capture the unique structural properties of each graph. Then feed both embeddings to the next layer.

Training modifications: Use separate mini-batches for G1 and G2 during each training epoch, reducing interference and allowing the model to learn graph-specific patterns.

3.3 Topological Changes to Improve Training

3.3.a Plot the Ricci Curvature for Each Edge

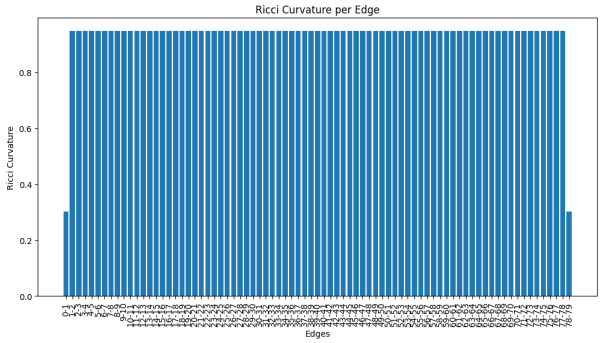

Figure 41: Ricci Curvature for Each Edge in G1

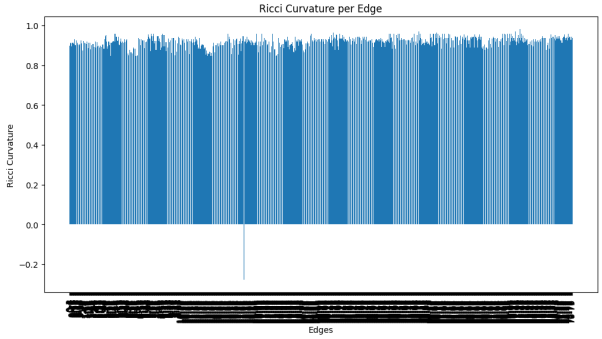

Figure 42: Ricci Curvature for Each Edge in G2

3.3.b Investigate Extreme Case Topologies

Making GNN Behave Like an MLP: To make a Graph Neural Network (GNN) behave like a Multi-Layer Perceptron (MLP), you can remove the influence of the graph's

topology by setting the adjacency matrix to an identity matrix to disable message passing and ignoring neighbourhood information.

Results: The average F1-score (0.55) is slightly worse than without doing augmentation, the embeddings have high overlapping and there are no significant benefits in making the adjacency matrix disable message passing to make it behave like MLP. This suggested that the connection between nodes is important in the classification task.

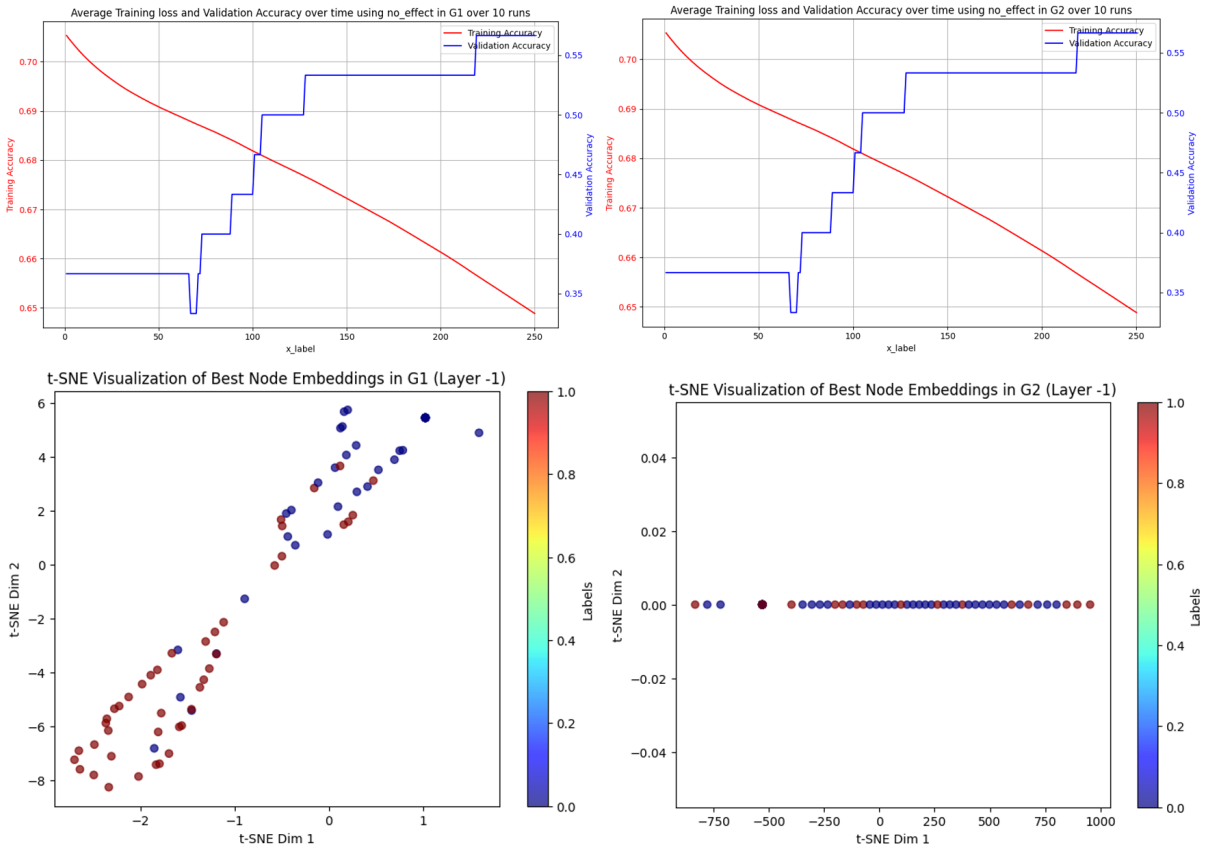

Figure 43: Training loss and F1-score over 10 runs and best final layer embeddings using no_effect

Optimal training and testing if labels were available: Edges connecting nodes of the same class should have high connectivity, while edges connection nodes of the same class should have low connectivity. Therefore, I set the adjacency weight for all edges with the same class to 1 and 0 otherwise.

Results: The average F1-score is at max (1.0), the embeddings are very separable and only connecting the nodes of the same class improved the performance significantly. This suggested that the connection between nodes with the same class has high significance in the classification task.

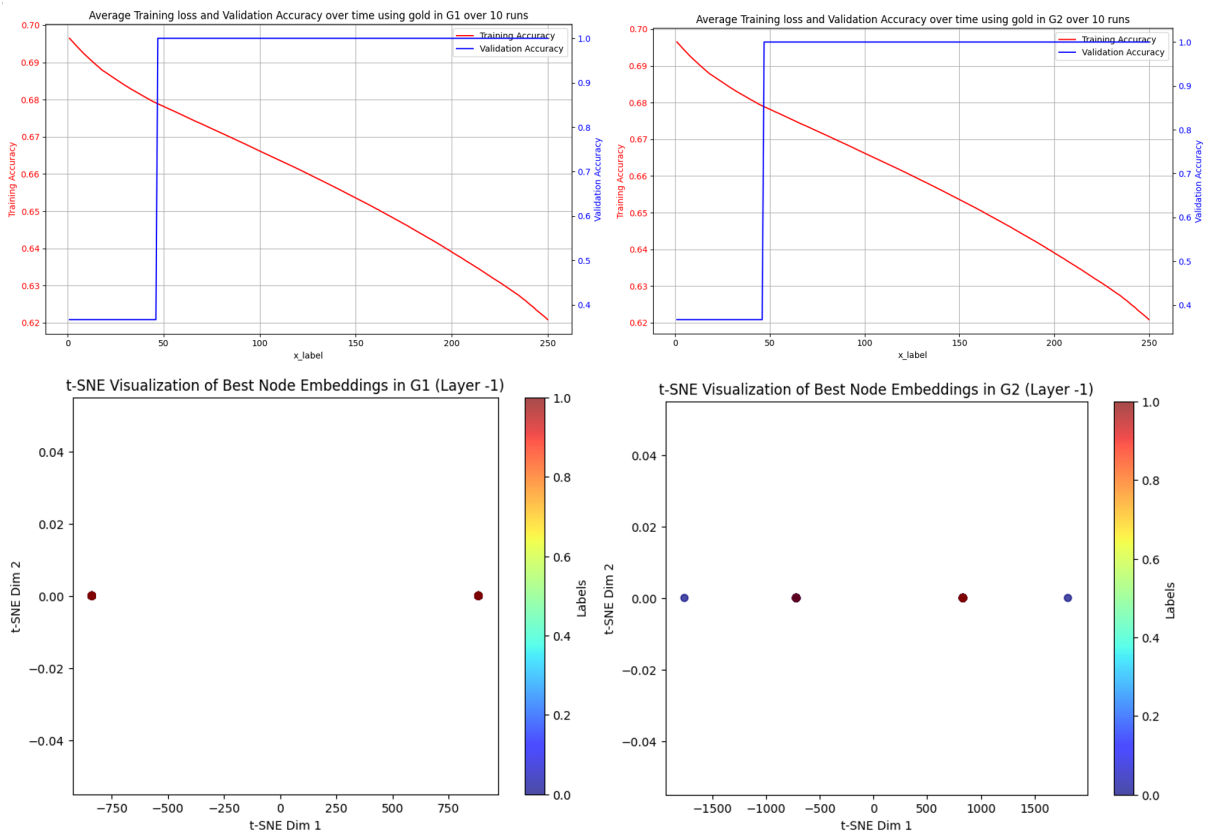

Figure 44: Training loss and F1-score over 10 runs and best final layer embeddings using *gold_label*

3.3.c Improving Graph Topology for Better Learning

Modifying connections based on class: I have strengthened intra-class connectivity and weakened inter-class connectivity by multiplying the weights of intra-class connectivity and half the weights of inter-class connectivity, the introduce class-aware edges weights in light of the results above (where we set 1 to the same class and 0 to a different class).

Introduce feature similarity in edges: I have computed pairwise similarity between node features using cosine similarity and merged the similarity-based matrix with the original adjacency matrix.

By doing both augmentations, I hope to enhance the connection between nodes with similar features and the same class.

Results: The average F1-score (0.80) is still better than without augmentation but worse than forcefully setting all edges with the same class to 1. This suggested that the classification almost only depends on the connections between the same class.

4 References

References

[1] Chien-Chun Ni et al. "Ricci curvature of the internet topology". In: *2015 IEEE conference on computer communications (INFOCOM)*. IEEE. 2015, pp. 2758–2766.

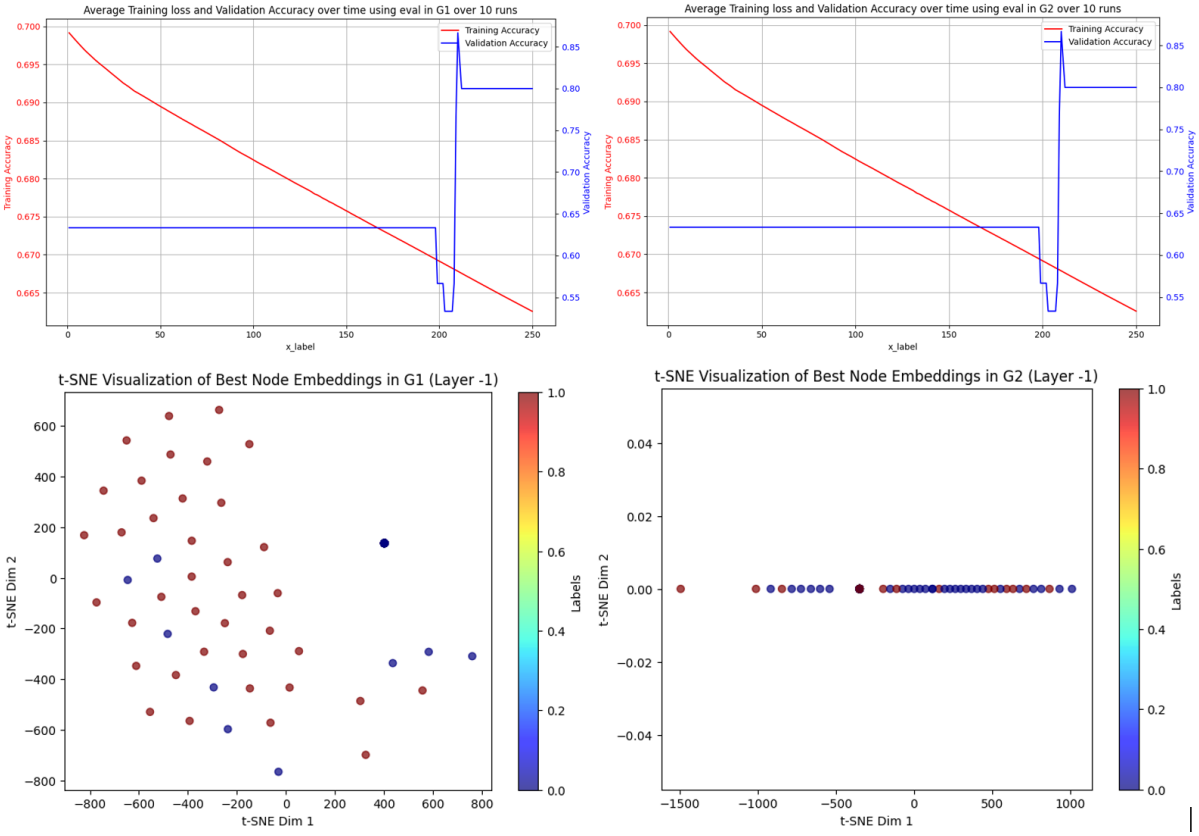

Figure 45: Training loss and F1-score over 10 runs and best final layer embeddings using *eval*