Applying Graph Machine Learning with PyG to Real-World Problem Domain: Paper Classification in Citation Network

Tutorial by Valiullina Chulpan, Pichugina Anastasia, Saparov Yakov

1. Introduction

Tutorial Purpose.

This tutorial is aimed to show how you can use modern graph machine learning (Graph ML) techniques to solve real-world problems. Graph ML extends traditional ML approaches by accounting for the relationships and structures inherent in graph data, where entities are nodes and relationships are edges. Graphs are everywhere: from social networks like Facebook to biological networks where nodes represent proteins and edges represent interactions. The purpose of a graph is to present data that are too numerous or complicated to be described adequately in the text and in less space.

PyTorch Geometric.

PyG python library is a special library for working with graphs (is built upon PyTorch). It contains ‘data’ module to manage different graphs’ types, ‘utils’ for methods like softmax, index_sort, or is_sparse, ‘datasets’ module containing the most famous graph datasets, ‘nn’ module with convolution, normalization layers, etc.

The tutorial will start with problem definition, datasets overview and Exploratory Data Analysis (EDA), then introduction to Graph Neural Networks (GNNs), models implementing, results comparison, and conclusion.

In case you want to see our code we provided colab notebooks:

- Data Overview
- GCN and GAT models
- SSP model

2. Problem Definition

Our topic is Paper Classification in Citation Networks. We have a network where each node is a publication and has its label (theme considered in the publication). Each node has its own binary feature vector derived from a dictionary of unique words. Each edge describes a citation from one paper to another.

There are 3 tasks in Graph ML:

- Node-level: label prediction (classification)
- Edge-level: link prediction
- Graph-level: graph clustering

In this tutorial we will consider node classification: predict the class label of each node in the graph based on its features and the structure of the graph.

As evaluation metrics we will consider accuracy, precision, recall, f1-score, and ROC-AUC. In the further analysis we will compare them also.

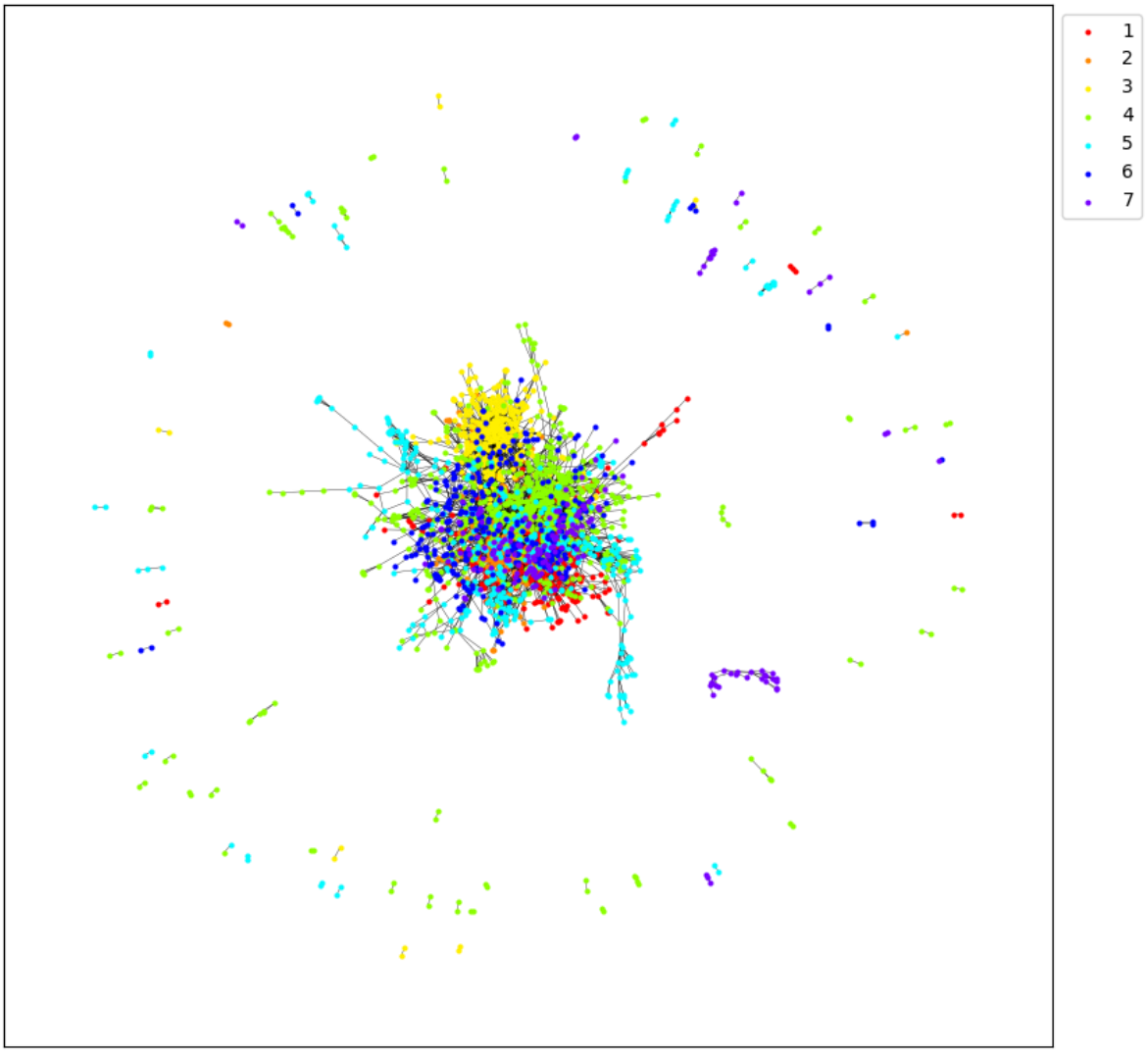
3. Dataset Overview

In this tutorial we decided to introduce 2 datasets: Cora and Citeseer. Both of them are popular and widely used citation network datasets. In each dataset all nodes are represented by research papers and all edges are citations. Nodes' feature vectors are binary vectors where each feature represents absence or presence of a specific word (from a predefined corpora) in paper.

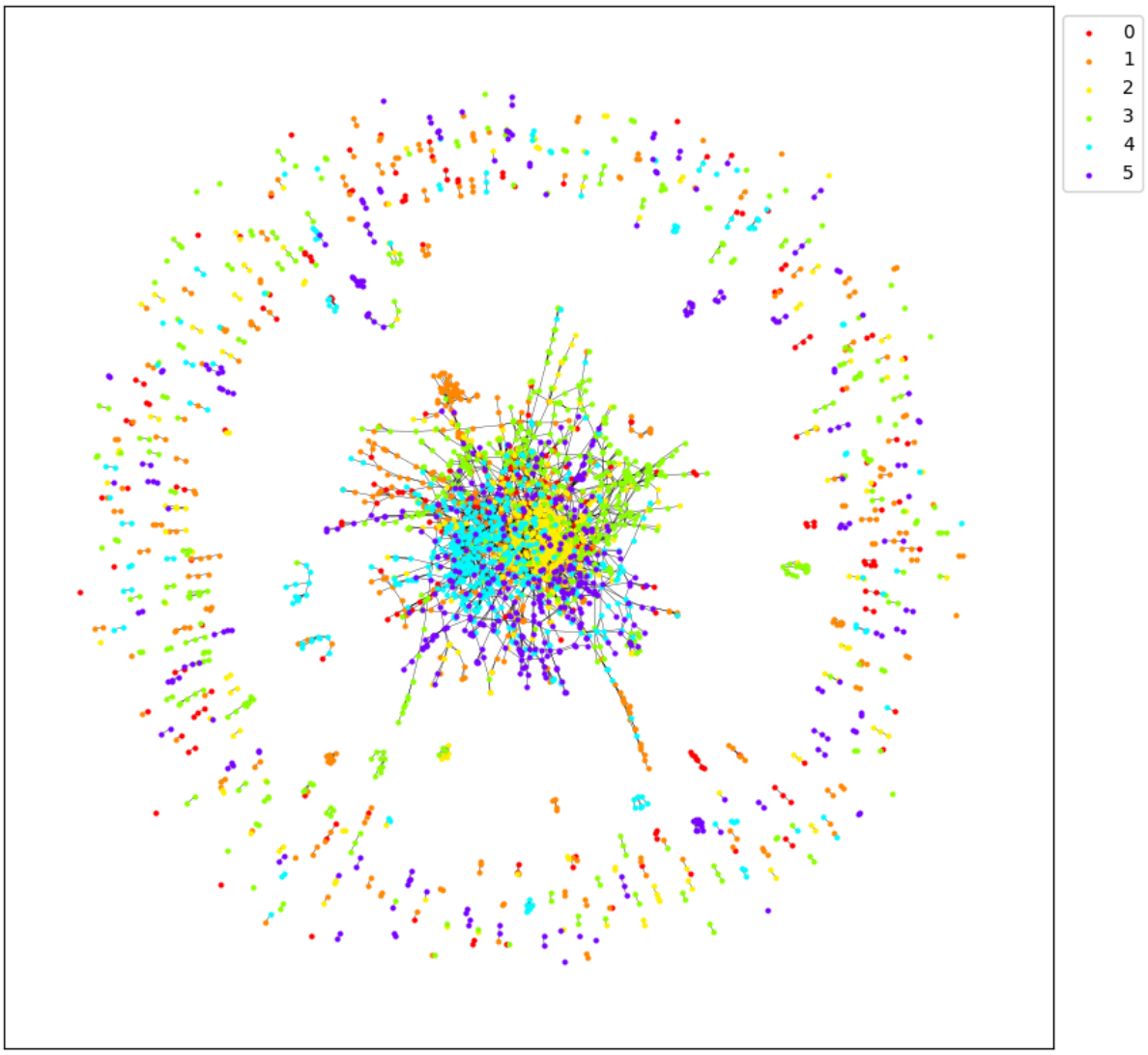
Characteristic / Dataset name	**Cora**	**Citeseer**
Graph size	2708 nodes and 5429 edges	3327 nodes and 4732 edges
Feature vector size	1433	3703
Classes	7 classes: Neural Networks, Genetic Algorithms, Reinforcement Learning, and more.	6 classes: Agents, AI, ML, DB, IR, HSI
Imbalance	Exists	Exists

Here are their visualizations:

Cora:

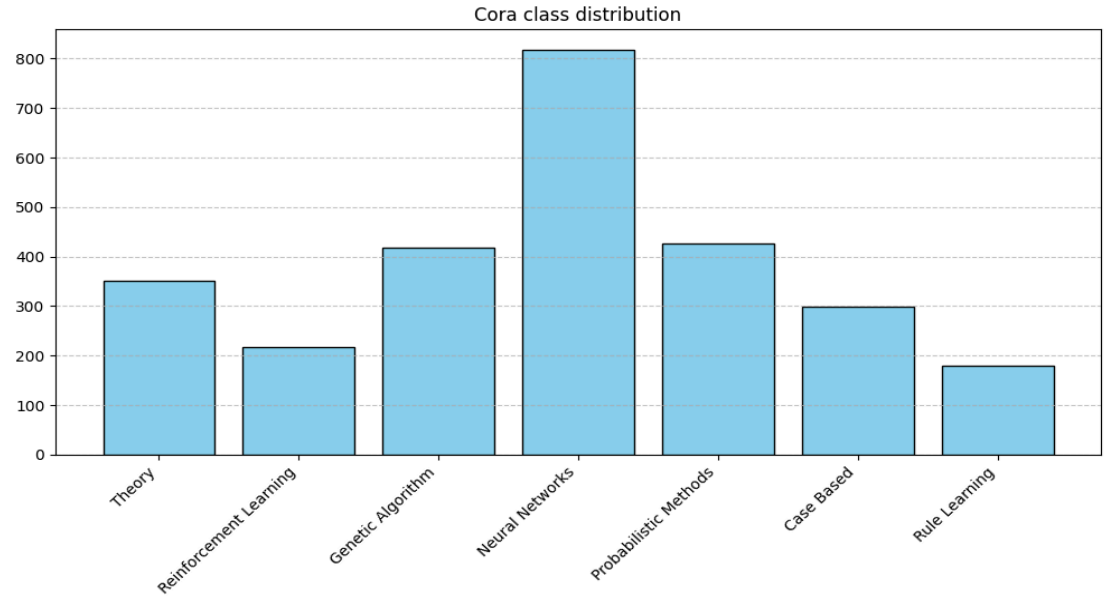


Citeseer:



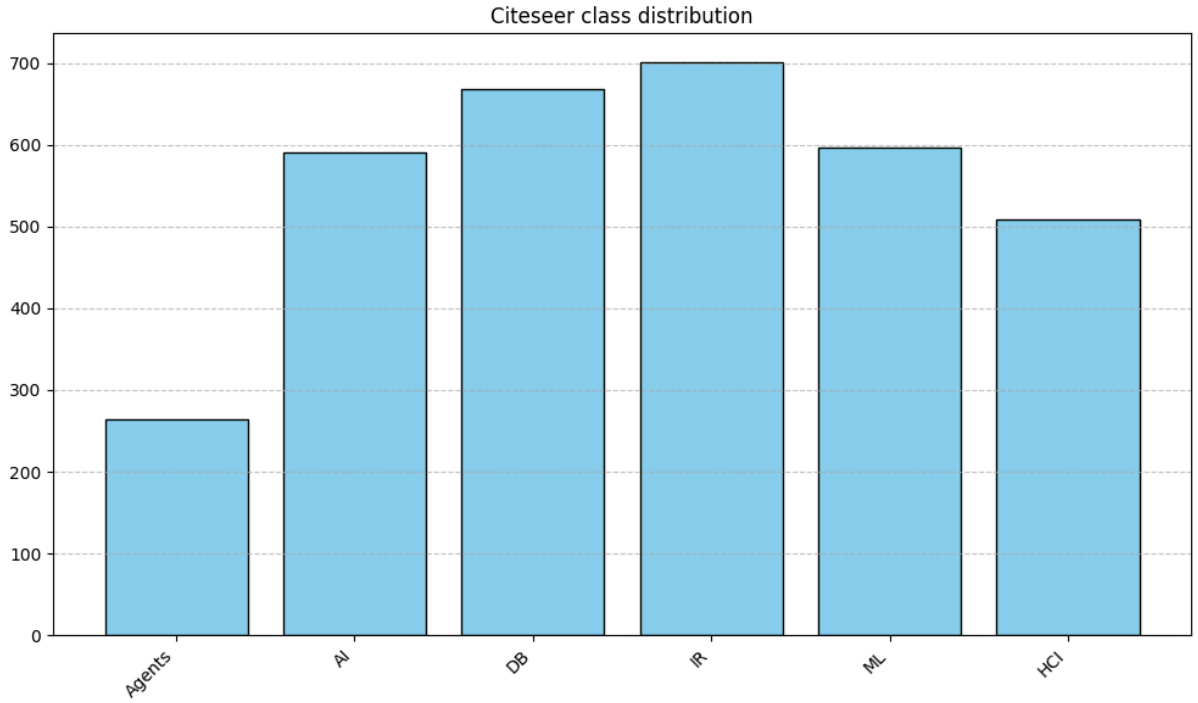
Both datasets are imbalanced (have different classes distributions).

Cora:



Here we see that ‘Neural Networks’ has about 800 entries while ‘Rule Learning’ has only 180.

Citeseer:



Citeseer has about the same situation when class imbalance occurs.

As an outcome we can conclude that usage of Accuracy as the main metric isn't really significant. We should use f1-score, combining precision and recall, and ROC-AUC.

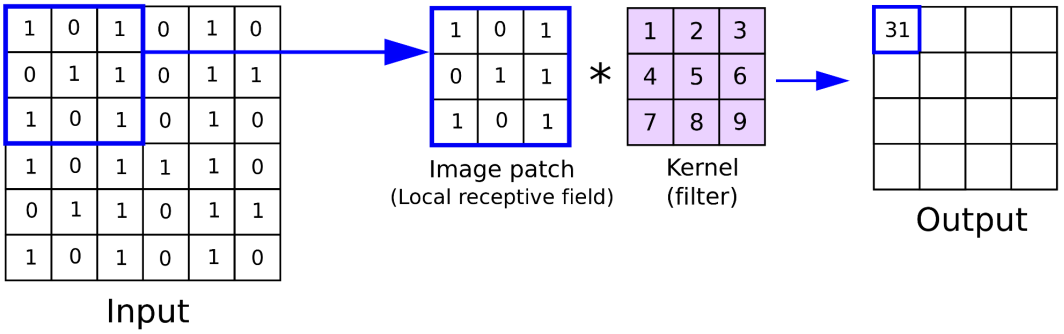
If you would like to see the EDA, check our colab with data overview.

4. Why Graph ML

When working with graphs in machine learning, the typical approach is to first create a useful representation of the things that matter (like nodes, edges, or entire graphs) depending on what you're trying to do. Then, you use these representations to train a model to make predictions for your specific task.

The thing is, most machine learning and deep learning (DL) tools are designed to work with simple data types. For example, images are grids with the same structure and size (height x weight x channel number), and text and speech are like lines with a clear order. But graphs are different - they don't have a fixed shape or size, and the nodes can have different numbers of neighbors. Moreover, in ML and DL we introduce feature independence, while graph nodes don't follow this assumption. Here we use **Homophily** assumption ("like attracts like" - papers with similar topics tend to cluster together in the graph): principle by *propagating* features and *aggregating* them within various graph neighborhoods via different mechanisms. Each node is connected to others in different ways, so we can't just look at them one by one, so we use a node's neighbourhood.

Revision of convolution:



But it's really hard to use convolutions on graphs because they can be any size and have a complicated structure. Unlike images, where pixels are arranged in a neat grid, graphs don't have a clear layout, so we can't use the same techniques. This makes it tough to apply CNNs to graphs. However, GNNs take the *idea* of convolution and make it work for more complex data, not just simple 2D pictures.

Node Embeddings is the required step in each Neural Network. Similarly to classical DL, they must be representative and similar to each other in case the nodes in the graph themself are similar to each other: “similarity in the embedding space approximates similarity in the network”. To achieve this, embeddings must capture not only a node’s information (feature vector), but structure of it’s neighbourhood too.

5. Models Overview

Graph Convolutional Network (GCN) is a foundational graph neural network model that uses convolutional operations to aggregate information from neighboring nodes. It processes the graph structure and node features to produce meaningful embeddings for tasks such as node classification. It is suitable for small graphs with limited neighborhood influence.

Graph Attention Network (GAT) extends GCN by using attention mechanisms to learn the importance of neighboring nodes, allowing the model to focus on relevant parts of the graph while ignoring noise. It uses attention mechanisms to weigh the importance of neighbors and handles complex and noisy graphs effectively.

If you want to try really powerful model, you should stick with SSP:

Semi-Supervised Propagation (SSP) is a graph-based learning technique that leverages the structure of the graph and a small set of labeled nodes to propagate labels or information to the unlabeled nodes. By iteratively updating node labels based on their neighbors and a predefined similarity measure, SSP can effectively utilize both labeled and unlabeled data. This approach is particularly useful for tasks like node classification in scenarios with limited labeled data. SSP assumes that connected nodes are likely to share similar labels, making it suitable for graphs with strong homophily.

Feature	GCN	GAT	SSP
Aggregation	Uniform (weighted by degree)	Attention-weighted	Selective propagation with preconditioning
Neighbor Influence	Equal contribution	Weighted by importance	Dynamically adjusted during training
Complexity	Linear in edges	Linear (higher constants due to attention)	Linear but computationally efficient
Use Case	General-purpose GNN	Noisy/complex graphs	Large graphs with training optimization

6. Model via code

In GAT node features and edge indices are passed through the first attention layer. Attention scores highlight influential neighbors, and features are aggregated. The second layer outputs the logits, processed with softmax for classification.

In GCN node features and edge indices are processed through the first GCNConv layer. Features are aggregated across all neighbors with uniform weighting. The second layer produces logits, normalized with log-softmax for classification.

SSP module computes node embeddings by selectively propagating features through graph neighborhoods using advanced convolutional operations. These embeddings capture both node-specific and neighborhood-specific information. The **CLS (Classification Layer)** module refines these embeddings and classifies nodes by applying log-softmax, ensuring accurate predictions while leveraging the graph structure.

For more information follow the link:
https://colab.research.google.com/drive/1bDHglH0RxRZpNl3qGCEjJYpkUpr2RWsa?usp=sharing

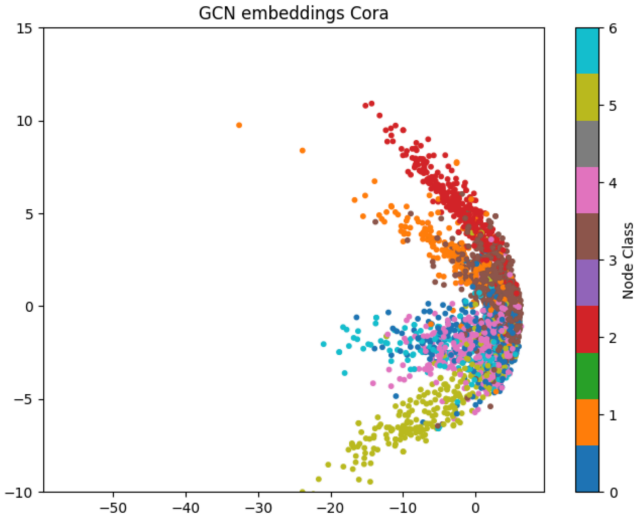
```
    class GCN(torch.nn.Module):
        def __init__(self, input_dim, hidden_dim, output_dim, dropout=0.5,
activation='relu'):
            super(GCN, self).__init__()
            self.conv1 = GCNConv(input_dim, hidden_dim)
            self.conv2 = GCNConv(hidden_dim, output_dim)
            self.dropout = torch.nn.Dropout(dropout)

            # GCNConv already contains Linear layers, so there is no need to
add them manually
            if activation == 'relu':
```

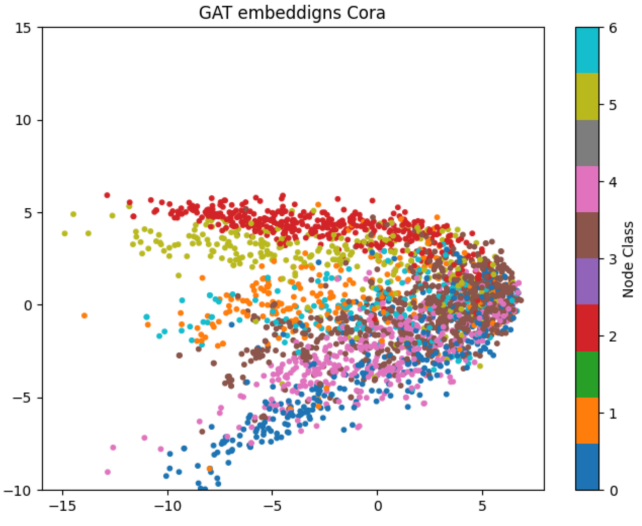
```
                self.activation = torch.nn.ReLU()
            else:
                raise Exception("Unsupportable type of activation function.")

        def forward(self, data):
            # usually data is fully passed to the forward
            # extract the required elements:
            x, edge_index = data.x, data.edge_index

            # first convolution
            x = self.conv1(x, edge_index)
            x = self.activation(x)
            x = self.dropout(x)

            # second convolution
            x = self.conv2(x, edge_index)
            x = F.log_softmax(x, dim=1)
            return x
```

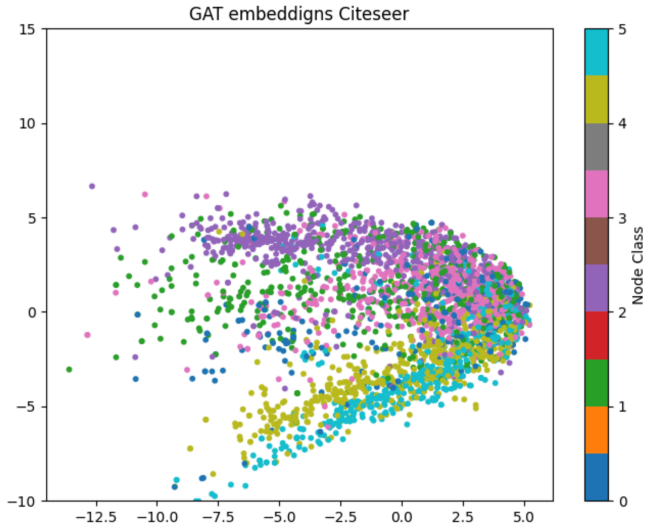
```
    class GAT(torch.nn.Module):
        def __init__(self, input_dim, hidden_dim, output_dim, heads=8,
dropout=0.5):
            super(GAT, self).__init__()
            self.gat1 = GATConv(input_dim, hidden_dim, heads=heads)
            self.gat2 = GATConv(hidden_dim * heads, output_dim, heads=1)
            self.dropout = torch.nn.Dropout(dropout)
            self.activation = torch.nn.ELU()

        def forward(self, data):
            # in the same way extract elements
            x, edge_index = data.x, data.edge_index

            # 1st pass
            x = self.gat1(x, edge_index)
            x = self.activation(x)
            x = self.dropout(x)

            # 2nd pass
            x = self.gat2(x, edge_index)
            x = F.log_softmax(x, dim=1)
            return x
```

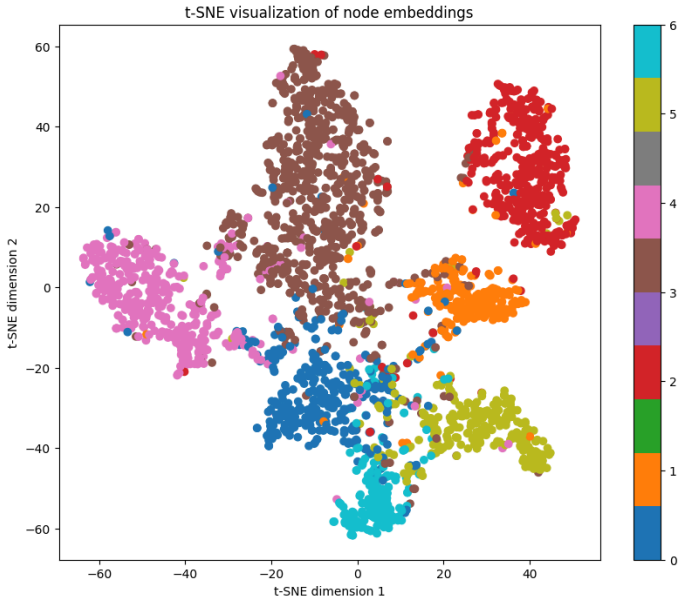
7. Visualizations

We plotted embeddings and it's visible that the cora dataset is better to differentiate classes and GCN is a better model for that purpose:

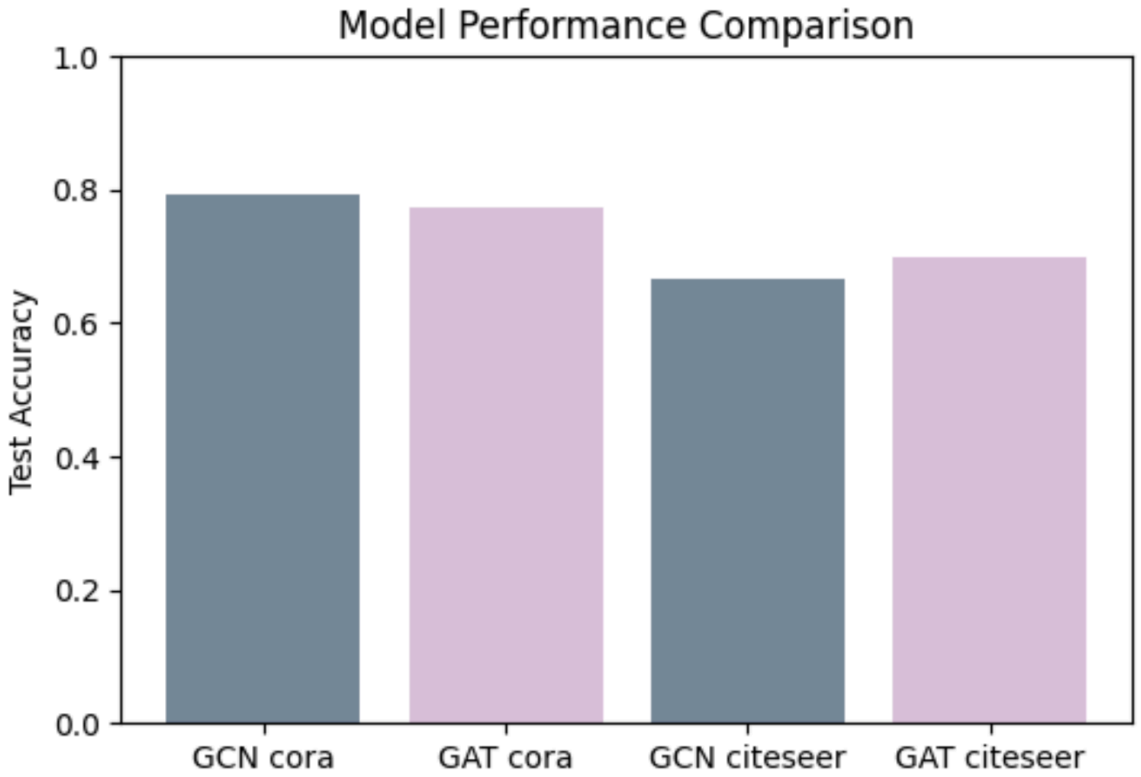




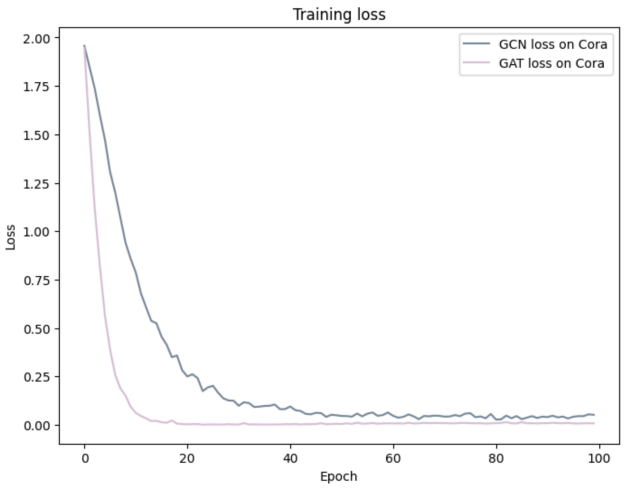


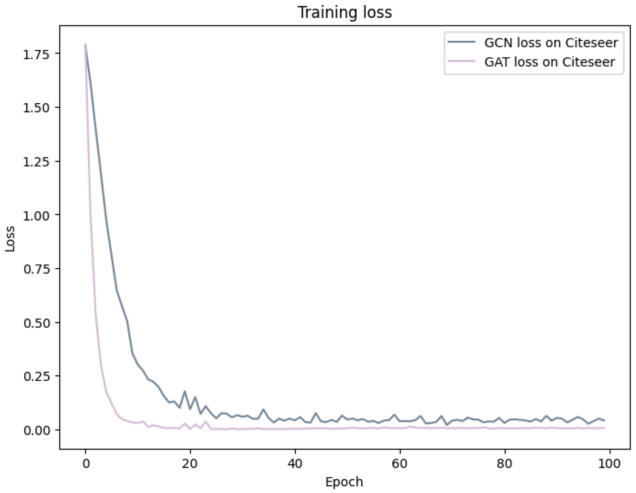


Models performance comparison on datasets again shows, that Cora dataset is better to try first and easier to interpret results:

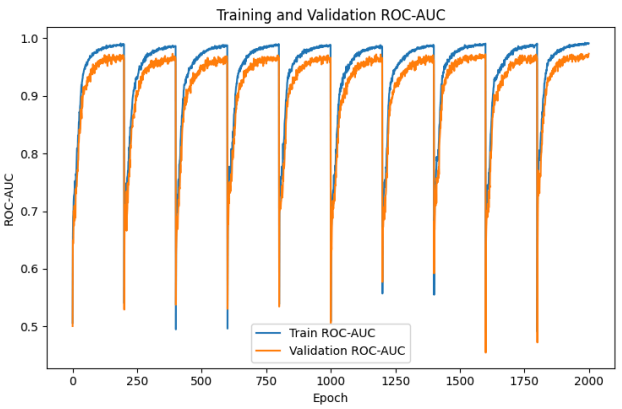


Losses are pretty same:

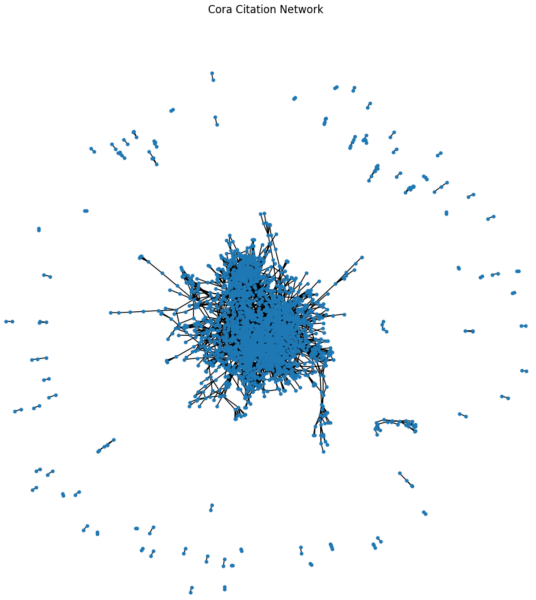


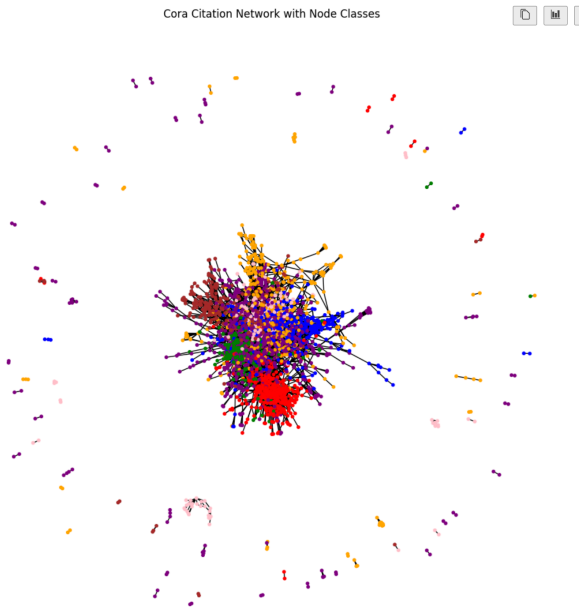


SSP:

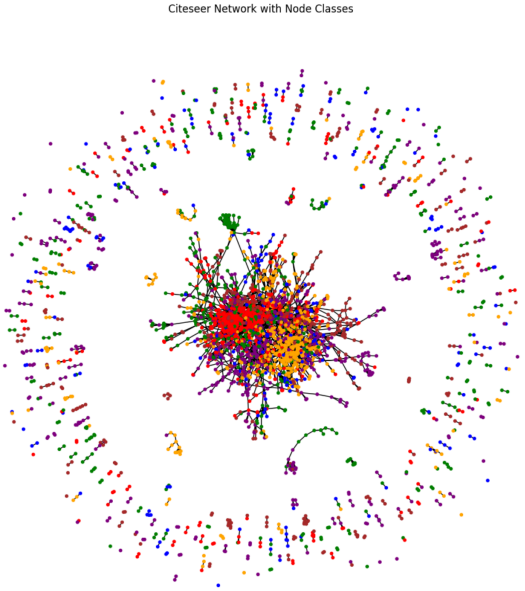


Cora dataset before and after node colorization:

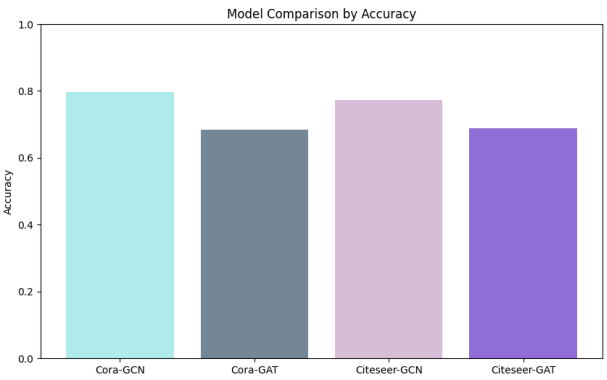


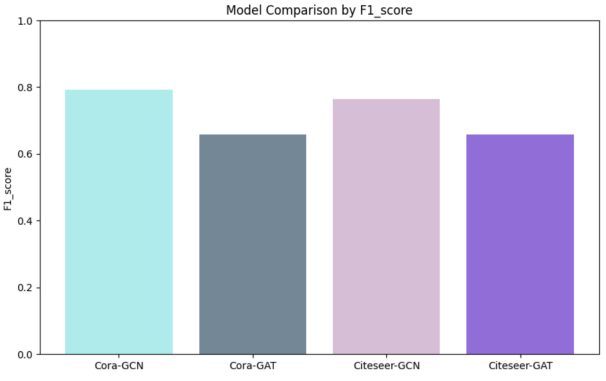


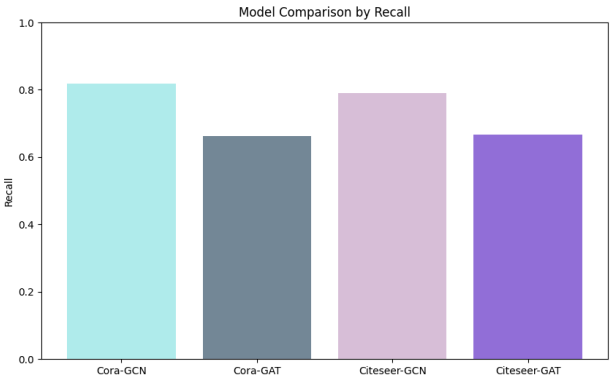
Citeseer dataset before and after node colorization:

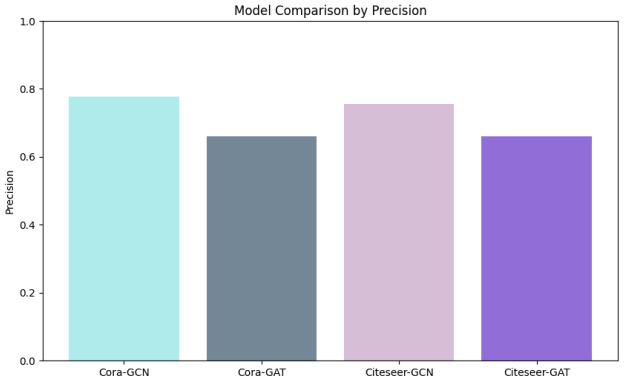


Metrics showed almost same results on these datasets with models, the best was Cora dataset on GCN:

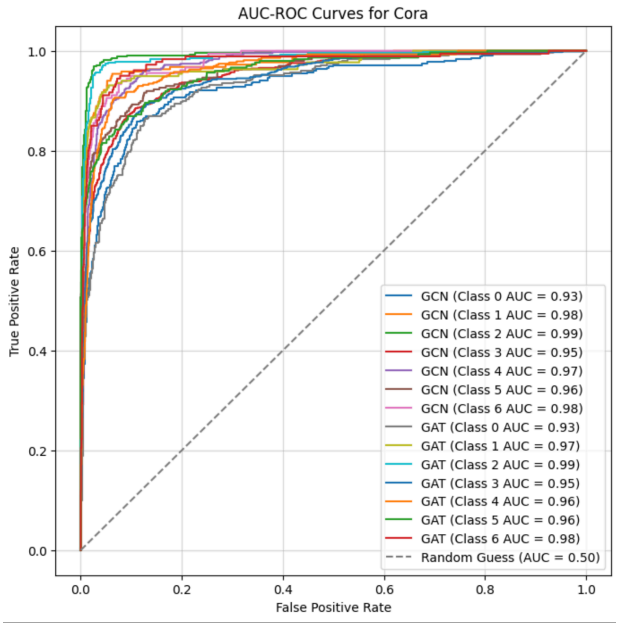


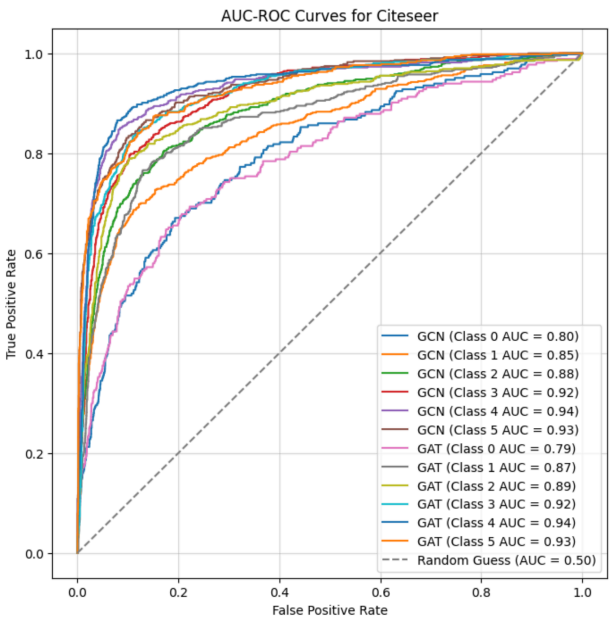


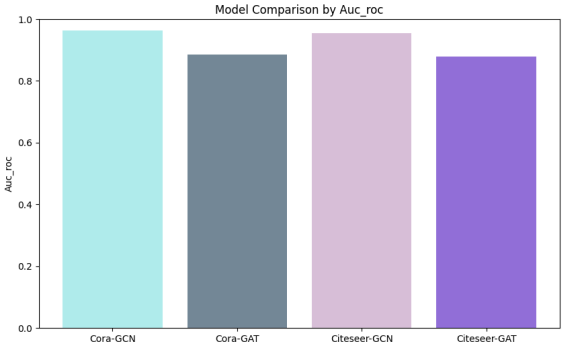




Here we can see that same models show better performance with cora dataset than citeseer:







If you need more accurate values:

```
Results for Cora Dataset:
  GCN -> Accuracy: 0.7970, Precision: 0.7825, Recall: 0.8152, F1-Score: 0.7940
  GAT -> Accuracy: 0.7920, Precision: 0.7669, Recall: 0.8034, F1-Score: 0.7786

Results for Citeseer Dataset:
  GCN -> Accuracy: 0.6820, Precision: 0.6600, Recall: 0.6621, F1-Score: 0.6563
  GAT -> Accuracy: 0.6880, Precision: 0.6618, Recall: 0.6634, F1-Score: 0.6590
```

8. Conclusion

This tutorial showcased the application of **Graph Neural Networks** to classify academic papers in citation networks using the **Cora** and **Citeseer** datasets.

Both datasets exhibit class imbalance, requiring metrics like **F1-score** and **ROC-AUC** for comprehensive evaluation. Visualizations revealed better class separation in Cora, making it more interpretable.

GCN is Efficient for general-purpose tasks, performs best on smaller, less noisy datasets like Cora. **GAT** excels in noisy graphs with attention-based aggregation but with higher complexity. **SSP** is robust for larger, complex graphs due to its selective propagation mechanism and optimization techniques.

Cora consistently outperformed Citeseer across all models, with **GCN** showing the best performance for class differentiation. SSP demonstrated scalability and stability, making it ideal for large-scale tasks.

Small recommendation from authors:

Use **GCN** for simplicity and efficiency on balanced or small datasets and **GAT** or **SSP** for noisy or complex graphs requiring selective focus or scalability.

9. References

https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications
https://medium.com/@koki_noda/ultimate-guide-to-graph-neural-networks-1-cora-dataset-37338c04fe6f
https://distill.pub/2021/gnn-intro/
https://tkipf.github.io/graph-convolutional-networks/