Final Project Report:
Node Classification with Graph Neural Networks

Team Name: CONMEN
Alex Chen, Dao Chi Lam, Dae Young Kim, Moe Elarbi
CSE 881: Data Mining

Problem Statement

Graph data structures are essential for modeling complex relationships in data mining tasks, representing interconnected entities in a structured way. Nodes correspond to individual entities, and edges capture their interactions or dependencies. For example, in social networks like Facebook or LinkedIn, users are nodes, and edges represent friendships or follows. This representation helps uncover patterns in real-world systems, such as community structures in social networks, fraud detection in financial systems, or disease spread in epidemiology.

Objective

Our goal is to develop a highly accurate node classification model for a given graph dataset, predicting each node's class label based on its features and relationships. We conducted a comparative study of modern graph neural network (GNN) architectures to identify which models best utilize node features and graph structure. The final performance was evaluated on a hidden test set.

Dataset Overview

The dataset contains 2,480 nodes connected via an adjacency matrix with 10,100 edges across 7 classes. Each node has 1,390 features, with only 496 nodes having known labels (used for training) and 1,984 unlabeled nodes for testing.

Nodes	Edges	Classes	Features	Train / Test
2,480	10,100	7	1,390	496 / 1,984

Table 1: Dataset summary

Files: `adj.npz` (Adjacency matrix, 2480×2480), `features.npy` (Feature matrix, 2480×1390), `labels.npy` (496 labels), `split.json` (Train/test node indices)

Methodology

Our team utilized a comprehensive, multifaceted exploration strategy. Each week, team members met to share findings, identify new promising approaches, and distribute research responsibilities. This methodology allowed us to explore a diverse range of models and strategies. The following section details these exploration paths, which contributed invaluable insight in informing final choices in implementation.

Data Preprocessing

To prepare the dataset for our graph neural network models, we applied several preprocessing techniques to enhance data quality and address challenges like redundant features and class imbalance. These steps ensured the data was suitable for effective model training.

- **Feature Selection**: Feature selection is the process of identifying and retaining the most informative variables in a dataset while discarding redundant or irrelevant ones. We removed features with zero standard deviation, as they provide no discriminative information and could hinder model performance.
- **Node Feature Normalization**: Node feature normalization rescales each node's attributes so that differing feature magnitudes do not dominate learning, leading to faster convergence and more stable GNN training. We tested differing static scaling methods, 'StandardScaler' and 'RobustScaler'. 'RobustScaler' is robust to outliers, scaling data based on the interquartile range. Later, we also experimented with skipping static scaling in favor of normalization layers build into model architecture.
- **Half-Hop**: Half-Hop averages a node's own feature vector with the mean of its one-hop neighbors' features, infusing each embedding with immediate local context while still keeping the node's unique signature (Azabou et al. (2023)). We used it to feed the model with denser, noise-resistant inputs without the over-smoothing inherent in deeper neighborhood aggregations, hence enhancing classification accuracy.
- **Two-Hop:** Two-Hop aggregation extends a node's receptive field by collecting and summarizing information not only from its immediate neighbors but also from their neighbors, capturing patterns that span two steps in the graph. This broader context helps distinguish nodes that look similar locally yet belong to different larger structures. We applied it to discover community-level signals that one-hop features missed, improving classification on classes defined by wider connectivity.
- **Virtual Node:** A virtual node is an extra, artificial node connected to every real node in the graph, acting as a central hub that gathers global information in one round of message passing and redistributes it on the next (Gilmer et al. (2017)). This hub lets distant nodes exchange signals without multiple hops, effectively injecting whole-graph context into each embedding. We used the virtual node to speed up far-away information propagation and improve classification when neighborhoods locally were not enough to reflect global patterns prevalent across the graph.
- **RandomWalk (NodeWalk):** RandomWalk samples short, stochastic paths starting from each node and records the frequency or order in which other nodes are visited, capturing higher-order proximity and community structure that single-step methods miss (Dwivedi et al. (2021)). The resulting walk statistics feed into the model as enriched features that encode both connectivity and distance. We inte-

grated NodeWalk to give the classifier rich structural cues from multi-step interactions, improving accuracy where local features were insufficient on their own.

- **Laplacian Eigenvector:** Laplacian eigenvectors are the characteristic vectors of a graph's Laplacian matrix whose low-frequency components vary smoothly across densely connected regions and change sharply across sparse cuts, thereby encoding the graph's global geometry and community structure (Dwivedi et al. (2020)). These eigenvectors serve as positional encodings that complement local feature aggregations by revealing how nodes are situated within the overall topology. We combined them to offer our model explicit spectrum-based global information to help it discriminate nodes that share the same local neighborhoods but different positions within the entire graph.
- **DropEdge:** DropEdge randomly removes a subset of edges from the graph during each training epoch, forcing the GNN to rely on multiple, diverse subgraph views and thus alleviating over-dependence on any single connection. This stochastic thinning acts as both regularization and data augmentation, curbing over-smoothing in deep layers. We employed DropEdge to avoid overfitting and enable deeper models to generalize better on the unseen test nodes (Rong, Huang, and Huang (2019)).

Training Enhancements This section details some particular training augmentations that were given careful attention to prevent data leakage across validation folds during training. Specifically, data augmentation techniques that increased our training size were strictly confined to the training partition of each fold to ensure no information from the validation set was used in training.

- **Oversampling**: Random oversampling and GraphSMOTE balances an imbalanced dataset by either duplicating existing minority-class nodes (random oversampling) or creating new, synthetic minority nodes along edges in feature space that respect graph structure (GraphSMOTE: (Zhao, Zhang, and Wang (2021))), both of which enlarge the rare-class sample size. We employed these techniques to prevent model bias towards majority classes and improve accuracy on under-represented node types.
- **Pseudo-labeling:** Pseudo-labeling is a semi-supervised technique in which the model assigns provisional labels to unlabeled nodes when its prediction confidence exceeds a chosen threshold and then retrains using these high-confidence predictions as additional training data (Li, Yin, and Chen (2022)). By iteratively enlarging the labeled set, the model leverages information hidden in the unlabeled portion of the graph. We utilized pseudo-labeling to leverage the 1,984 unlabeled nodes, essentially expanding our training set and improving classification performance without introducing new human-labeled examples.

Model Explorations

We tested several GNN architectures to find the best fit for our dataset, as each model uses the graph structure and node features in unique ways to optimize performance.

- **GCN / GCNII**: These models use graph convolutions to gather neighbor information and capture local patterns effectively. GCNII adds deeper layers and residual connections for better results. We chose them for their simplicity and fit for our medium-sized graph, setting a solid baseline (Kipf and Welling (2016); Chen et al. (2020)).
- **GraphSAGE**: GraphSAGE samples nearby nodes to aggregate features efficiently, keeping computation light while maintaining structural information. We used it to manage our dataset's 10,100 edges, ensuring fast and strong feature learning (Hamilton, Ying, and Leskovec (2017)).
- **GAT / GATv2**: These models use attention mechanisms to prioritize key neighbors, with GATv2 offering improved stability over GAT. We used them to capture critical connections in our imbalanced dataset and enhance classification performance (Petar Veličković (2017); Brody, Alon, and Yahav (2022)).
- **APPNP**: APPNP integrates neural networks with personalized PageRank to propagate information across the graph, balancing local and global features. We used it to make the most of our sparse training labels, effectively connecting both nearby and distant nodes (Gasteiger, Bojchevski, and Günnemann (2018)).
- **MultiviewGNN**: MultiviewGNN combines three GNN architectures (GATv2, GCN with residual connections, and GraphSAGE) to capture diverse graph perspectives. This approach creates comprehensive node representations from different viewpoints for strong classification performance.

Throughout our experimentation, we used PyTorch as our primary framework due to its flexibility and ease of implementation for GNN. However, we also conducted experiments using alternative frameworks to ensure comprehensive evaluation of available tools.

- **TensorFlow / Keras**: We tried TensorFlow B., E., and N. (2019) and Keras Bhalerao et al. (2021) to see if they could improve our results. After testing several GNN models with both frameworks, we found that they gave similar accuracy to PyTorch. However, TensorFlow was harder to work with when building complex GNN architectures. Therefore, we decided to stick with PyTorch for our main experiments since it was easier to use while giving equally good performance.

Training and Evaluation Methods

We used key techniques to train and evaluate our graph neural network models, tackling challenges like class imbalance and limited training data in our dataset.

- **Cross Validation**: For reliable generalization with our small, imbalanced dataset, we used 5-fold StratifiedKFold to assess model performance. It splits our 496 training nodes into balanced folds, keeping class distribution.
- **Loss Function**: To ensure fair predictions despite uneven class sizes, we used Negative Log Likelihood Loss (NLLLoss), testing with and without class weights. Class weights gave rare classes more influence, which enhanced balanced learning across all seven classes.
- **Optimizer**: Adam and AdamW optimizers were used to update weights efficiently, balancing speed and stability with weight decay to avoid overfitting. We used them for their strength in handling complex GNNs.
- **Scheduler**: We used ReduceLROnPlateau, reducing the learning rate by 0.5 if validation accuracy did not improve for 20 epochs. This fine-tuned training for better results across our GNN models.
- **Early Stopping**: We applied early stopping with a patience of 50 epochs, saving the best model based on validation accuracy. Training stopped if no improvement occurred, preventing overfitting and saving computational resources.

Other Miscellaneous Explorations

Some of our techniques do not fit squarely into the prior categories. Here, we describe a set of supplementary techniques, including architectural enhancements, such as residual / skip connection and normalization layers, and systematic hyperparameter optimization, such as manual grid search and automated Optuna.

- **Residual Connection**: This method adds a neural network layer's input to its output, preserving original information while learning new patterns (Scholkemper et al. (2024)). For example, in our MultiviewGNN, residual connections in GCN layers helped retain initial node features during complex transformations. We used this to stabilize training in deeper networks and avoid issues like vanishing gradients and information loss.
- **Normalization Layers**: We experimented with addition of normalization layering for models (Scholkemper et al.). Instead of static normalization, layering in normalization directly into the model architecture allowed for differing normalization scopes and exploration of newer ideas in addressing over-smoothing, and training instability. Of all currently proposed methods, we tried namely: BatchNorm, InstanceNorm, GraphNorm, and PairNorm.
- **Grid Search / Optuna Hyperparameter Tuning**: We utilized two different approaches to hyperparameter tuning. Grid search manual hyperparameter combination testing was used for simpler hyperparameter spaces as a quick method to check model sensitivity. Later, for more granular fine tuning, we adopted Optuna (Akiba et al. (2019)), a more advanced hyperparameter optimization framework that can search higher-dimensional spaces more efficiently. Optuna utilizes dynamic under-the-hood pruning for unpromising trials based on intermediate validation performance to allow for faster and more effective convergence.

Model Selection

To ensure fair and controlled comparison across model architectures, we adopted standard evaluation protocols. Each model was trained using the same 5-fold stratified cross-validation scheme, identical training parameters, and consistent optimization methods. Holding these factors constant and differentiating on model architecture allowed us to isolate that as the primary source of variation. Final model candidates for further tuning were based on highest mean cross validated accuracy across the folds.

Parameter	Value
Hidden dimension	64
Dropout rate	0.50
Learning rate	10^{-3}
Weight decay	10^{-4}

Table 2: Control parameters used for model selection

Additional notes:

- Model specific parameters were informed using their associated paper, or using default values from PyTorch.
- The Adam optimizer and ReduceLROnPlateau scheduler were used for controlling learning. A patience value of 10 was used, with a maximum epoch without improvement value of 20.

In addition to baseline architecture comparison, we conducted similarly controlled experiments to evaluate our explored augmentation methods. Each augmentation technique would be tried on a baseline model using two trials: one model using the augmentation, and one without as a control. We then compared mean cross validated accuracy across multiple training runs, quantifying average performance difference attributed to the augmentation. This allowed us to check raw improvements, ultimately influencing our decisions on their inclusion or exclusion from the final pipeline.

Implementation

The previous section has outlined all methods tried in our exploration phase, this section will cover a concise description of final implemented methods used to create, train, and evaluate the model which showcased our best testing performance.

Our implementation starts with simple z-score scaling standardization of the feature matrix (mean = 0, standard deviation = 1).

Our model architecture utilizes a two-layer linear message passing backbone followed by an APPNP type propogation layer.

- Each linear layer uses 64 hidden channels, and utilized a ReLU activation function.
- Node dropout regularization with a rate of 0.507 is applied after each ReLU activation.
- After our two linear layer blocks (linear layer, ReLU, dropout), the APPNP propagation layer is applied. The APPNP specific parameters was configured with 72 hops (K=72), and a teleport parameter of 4.940×10^{-2} (alpha = 4.940×10^{-2}).

The final output layer applies a `log softmax(dim=1)` activation function over the seven classes, transforming logits into log probabilities, pairing naturally with our loss criterion, `NLLLoss`.

Our training protocol incorporated a 10% stratified sample holdout set for validation purposes. We optimized the model using Adam with learning rate $7.754 \times 10^{-}3$ and weight decay $9.785 \times 10^{-}7$. We utilize ReduceLROnPlateau with a patience parameter of 50 for scheduling of our learning rate with Adam. Upon reaching optimal performance, we conducted a final fine-tuning stage consisting of 10 additional epochs using all available labeled data (including the previously held-out validation set) with the best learning rate.

These hyperparameter values were determined through trial experimentation using the Optuna framework. Our search space included hidden dimensions [64, 128], dropout rates [0.4, 0.7], learning rates $[3 \times 10^{-3}, 3 \times 10^{-2}]$, and weight decay values $[10^{-7}, 10^{-3}]$. For APPNP specific parameters, we explored propagation steps (K) ranging from 10 to 80 and teleport probabilities (alpha) between 0.01 and 0.2. After evaluating 100 trials, we selected the configuration that maximized mean stratified cross validation accuracy (5 fold), resulting in the parameters described below.

Hyperparameter	Final Value
Hidden dimension	64
Dropout rate	0.507
Learning rate	7.754×10^{-3}
Weight decay	9.785×10^{-7}
Propagation hops (K)	72
Teleport probability (alpha)	4.940×10^{-2}

Table 3: Hyperparameters of the final APPNP model.

Final Results With these parameters, our model was able to reach a top internal cross-validated accuracy of 90.32% and a testing accuracy of 85.23%.

Team Contributions

In alphabetical order:

- **Alex Chen**: Focused on APPNP. Contributed to graph data visualizations, augmentation exploration, class balancing, cross-validation and final model training methodology, and hyperparameter tuning.
- **Dao Chi Lam**: Focused on baseline models (GCNs). Contributed to data augmentation, class balancing, and cross-validation evaluation.
- **Dae Young Kim**: Developed the MultiviewGNN model, integrating GAT, GCN, and GraphSAGE. Implemented pseudo-labeling, ensemble methods, and hyperparameter tuning.
- **Moe Elarbi**: Implemented Tensorflow-based GNN models and experimented with GCN and GraphSAGE models. Assisted in debugging and training optimization.

References

Akiba, T.; Sano, S.; Yanase, T.; Ohta, T.; and Koyama, M. 2019. Optuna: A next-generation hyperparameter optimization framework. *arXiv preprint*.

Azabou, M.; Ganesh, V.; Thakoor, S.; Lin, C.-H.; Sathidevi, L.; Liu, R.; Valko, M.; Veličković, P.; and Dyer, E. L. 2023. Halfhop: A graph upsampling approach for slowing down message passing. *arXiv preprint*.

B., P.; E., N.; and N., W. Y. 2019. Deep learning with tensorflow: A review. journal of educational and behavioral statistics. *Journal of Educational and Behavioral Statistics,*.

Bhalerao; Shilpa; ; and Ingle, M. 2021. Convolutional neural network: A systemic review and its application using keras. *Computing Technologies and Applications* 199–217.

Brody, S.; Alon, U.; and Yahav, E. 2022. How attentive are graph attention networks? *arXiv preprint*.

Chen, M.; Wei, Z.; Huang, Z.; Ding, B.; and Li, Y. 2020. Simple and deep graph convolutional networks. *arXiv preprint*.

Dwivedi, V. P.; Joshi, C. K.; Luu, A. T.; Laurent, T.; Bengio, Y.; and Bresson, X. 2020. Benchmarking graph neural networks. *arXiv preprint*.

Dwivedi, V. P.; Luu, A. T.; Laurent, T.; Bengio, Y.; and Bresson, X. 2021. Graph neural networks with learnable structural and positional representations. *arXiv preprint*.

Gasteiger, J.; Bojchevski, A.; and Günnemann, S. 2018. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint*.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. *arXiv preprint*.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Inductive representation learning on large graphs. *arXiv preprint*.

Kipf, T. N., and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. In *arXiv preprint*.

Li, Y.; Yin, J.; and Chen, L. 2022. Informative pseudo-labeling for graph neural networks with few labels. *arXiv preprint*.

Petar Veličković, e. a. 2017. Graph attention networks. *arXiv preprint*.

Rong, Y.; Huang, W.; and Huang, T. X. J. 2019. Dropedge: Towards deep graph convolutional networks on node classification. *arXiv preprint*.

Scholkemper, M.; Wu, X.; Jadbabaie, A.; and Schaub, M. T. 2024. Residual connections and normalization can provably prevent oversmoothing in gnns. *arXiv preprint*.

Zhao, T.; Zhang, X.; and Wang, S. 2021. Graphsmote: Imbalanced node classification on graphs with graph neural networks. *arXiv preprint*.