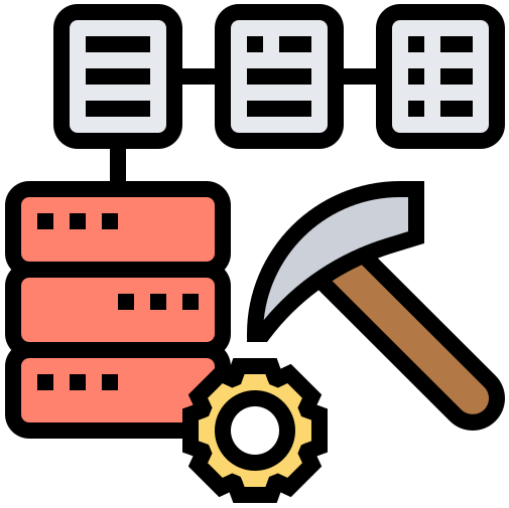

Data Mining

HW 4

Ashkan Ansarifard

1970082

A.Y. 2024/25

Table of Contents

1 Introduction

1.1 Project Structure

In this section, we show how the project is structured. In the following sections, we refect to this structure and explain what each module does.
For now, you can find the overall project structure as following:

```
project_root/
+-- main.py
|
+-- data/
|   +-- dataset_analyzer.py
|   +-- dataset_loader.py
|   +-- prepare_link_prediction_data.py
|
+-- embeddings/
|   +-- analyze_embeddings.py
|   +-- visualize_embeddings.py
|
+-- evaluation/
|   +-- evaluator.py
|   +-- cross_dataset_evaluator.py
|   +-- link_prediction_evaluator.py
|
+-- models/
|   +-- gnn_model.py
|   +-- generalized_gnn.py
|   +-- gnn_explainer.py
|   +-- node2vec_model.py
|   +-- link_prediction_gnn.py
|
+-- saved_models/
|   +-- generalized_<timestamp>/
|       +-- model.pth
|       +-- metadata.json
|
+-- training/
|   +-- trainer.py
|   +-- trainer_link_prediction.py
|
+-- utils/
|   +-- config.py
|   +-- dimensionality_handler.py
|   +-- device.py
|   +-- find_best_models.py
|   +-- load_generalized_gnn.py
|   +-- model_saver.py
|
+-- README.md
+-- requirements.txt
+-- setup.bat
+-- setup.sh
```

1.2 Environment Setup

To execute this project, follow these steps:

1. Ensure that Python 11.3 is installed on your system. You can download it from the official Python website.
2. Run the provided setup script to create and configure the virtual environment:
   - On Windows, run:

     ```
     setup.bat
     ```

   - On macOS/Linux, run:

     ```
     bash setup.sh
     ```

3. After the setup script completes, activate the virtual environment:
   - On Windows:

     ```
     <venv_name>\Scripts\activate
     ```

   - On macOS/Linux:

     ```
     source <venv_name>/bin/activate
     ```

4. Run the project using:

   ```
   python main.py
   ```

   If executed without providing the required arguments, the following message will be displayed:

```
usage:  main.py [-h] {single,
experiment,
find_best,
link_prediction,
analyze_embeddings,
node2vec,explain} ...
main.py:  error:  the following arguments are
required:  command
```

After completing these steps, your environment will be properly configured for executing the project.

2 Dataset Analyzer

The `DatasetMetricsAnalyzer` calculates various graph metrics to analyze the structure and properties of datasets.

Metrics and Their Interpretations

- **Degree Distribution:**
  Describes the distribution of node degrees in the graph. A similar degree distribution across datasets may indicate similar graph connectivity patterns, which could improve model transferability.
- **Average Clustering Coefficient:**
  Measures the likelihood of nodes forming local triangles. Higher values indicate dense local clusters. Comparing this across datasets helps understand differences in local graph density, affecting message-passing algorithms.
- **Graph Density:**
  The ratio of edges to possible edges in the graph. Densely connected graphs may allow models to propagate information more effectively, while sparse graphs may pose challenges for learning.
- **Connected Components:**
  Counts the number of isolated subgraphs in the dataset. Fewer components indicate better connectivity, which often correlates with improved model performance for tasks like node classification.
- **Community Detection (Louvain):**
  Identifies clusters or communities in the graph and measures modularity, which evaluates the strength of these communities. Higher modularity indicates well-defined clusters, which can aid in tasks like community detection or semi-supervised learning.
- **Diameter and Average Path Length:**
  The diameter is the longest shortest path between any two nodes, while the average path length indicates the typical distance between nodes. Smaller values suggest more compact graphs, potentially leading to faster information propagation.
- **Node Centrality Metrics:**
  Measures like degree centrality, betweenness centrality, closeness centrality, eigenvector centrality, and PageRank identify important nodes in the graph. Comparing centrality metrics across datasets highlights differences in node influence and structural roles, affecting task-specific outcomes.

- **Assortativity:**
  Quantifies the tendency of nodes to connect with similar-degree nodes. Datasets with high assortativity may require models that leverage this homophily for better performance.
- **Transitivity:**
  Measures the ratio of triangles to possible triangles in the graph. High transitivity indicates a tightly interconnected graph, which can enhance tasks like link prediction.
- **Radius and Core Numbers:**
  The radius measures the minimum eccentricity of any node, while core numbers represent the maximum subgraph where all nodes have at least a certain degree. These metrics reveal the robustness and central structure of the graph.
- **Edge Betweenness:**
  Measures the importance of edges in connecting different parts of the graph. Datasets with high edge betweenness may be more sensitive to edge removal, impacting model generalization.

Benefits of Comparing Metrics Across Datasets

Comparing these metrics provides insights into:

- **Structural Similarity:**
  Helps identify datasets with similar graph structures, improving model transferability.
- **Feature Distribution:**
  Certain metrics, like clustering coefficient or transitivity, reflect underlying feature distributions and alignment.
- **Task Feasibility:**
  Datasets with high connectivity, well-defined communities, or uniform centrality may favor tasks like node classification or link prediction.
- **Performance Bottlenecks:**
  Metrics such as high diameter or low density can highlight challenges in propagating information through the graph, identifying areas where models may struggle.
- **Dataset Diversity:**
  Understanding the variations between datasets ensures that models are tested under diverse conditions, improving their robustness and generalization.

3 Node Classification on Cora Dataset

3.1 Simple GNN Model

For this task, my first attempt was to create a ”simple” GNN model, which loads the datasets from ‘data/datasets‘ (the pipeline will download them if it cannot find them). Then it creates an instance of the ‘GNNModel‘ with fixed characteristics.
The architecture of the model consists of two graph convolution layers. Specifically:

- The first layer is a GCNConv layer that maps the input node features (`input_dim`) to a hidden representation (`hidden_dim`).
- A ReLU activation is applied to introduce non-linearity, followed by a dropout layer to prevent overfitting.
- The second layer is another GCNConv layer that maps the hidden representation to the output space (`output_dim`), corresponding to the number of target classes.
- A log-softmax activation function is applied to the final layer’s output to produce normalized probabilities over the node classes.

The model takes as input:

- `x`: The feature matrix of the nodes.
- `edge_index`: The edge list that represents the graph structure.

3.2 Generalized GNN Model

The next step, was to make a more ”Generalized” model which can be initiated with different variabless. The Generalized GNN model is a flexible and modular implementation that supports multiple GNN variants, including GCN, GraphSAGE, and GAT. The architecture can be customized in terms of the number of layers, the use of residual connections, layer normalization, and dropout. Below, the key design features and characteristics of the model are outlined:

- **GNN Variants**: The model supports three types of graph convolution layers:

  - `GCNConv` for Graph Convolutional Networks (GCN).
  - `SAGEConv` for GraphSAGE.
  - `GATConv` for Graph Attention Networks (GAT).

- **Architecture**:
  - The first layer maps the input node features (`input_dim`) to a hidden representation (`hidden_dim`).
  - Intermediate layers (optional) maintain the hidden dimension and apply the selected convolution operator.
  - The final layer maps the hidden representation to the output space (`output_dim`), producing class logits for each node.

- **Features and Enhancements**:
  - **Residual Connections**: If enabled, adds a residual skip connection between the input and hidden layers, facilitating gradient flow and stabilizing training.
  - **Layer Normalization**: Optionally applies layer normalization after each hidden layer to improve training dynamics.
  - **Dropout**: Introduces dropout regularization after ReLU activations to prevent overfitting.

- **Input Flexibility**:
  - Supports either a full `data` object containing node features (`x`) and edge indices (`edge_index`) or separate `x` and `edge_index` tensors.

- **Output**: The model outputs log-softmax probabilities across node classes, making it suitable for node classification tasks.

This generalized design allows for experimentation with various GNN architectures, making it adaptable for different datasets and graph-based learning tasks.

3.3 Trainer Class

The `GNNTrainer` class is designed to manage the training process for various types of GNN models, including both the simple GNN model and the Generalized GNN model. Its purpose is to streamline the process of loading data, initializing models, optimizing parameters, and saving trained models along with metadata.

3.3.1 Key Features

- **Dataset Handling**:
  - Automatically loads the dataset (default: `Cora`) using the `DatasetLoader` utility.
- **Model Initialization**:
  - Supports two types of models:
    1. `GNNModel`: A simpler two-layer GNN.
    2. `GeneralizedGNN`: A more customizable model supporting GCN, GraphSAGE, and GAT layers, with options for residual connections, layer normalization, and dropout.
  - Model parameters such as the number of layers, hidden dimensions, and variant type can be configured during initialization.
- **Training Process**:
  - Utilizes the Adam optimizer with configurable learning rate and weight decay.
  - Performs node classification by optimizing the negative log-likelihood loss function (`nll_loss`).
  - Displays training progress per epoch, including the loss value.
- **Model and Metadata Saving**:
  - Saves the trained model and its metadata (e.g., architecture details, dataset statistics, and training hyperparameters) for reproducibility and future use.

- Creates a dedicated directory to store models and metadata.

3.3.2 Workflow

The `GNNTrainer` class follows a systematic workflow:

1. Load the dataset.
2. Initialize the model based on the specified type and configuration.
3. Train the model for a fixed number of epochs, tracking the loss.
4. Save the trained model and its metadata for evaluation or reuse.

3.4 GNN Evaluator

The `GNNEvaluator` class is designed to evaluate the performance of trained GNN models on node classification tasks. It loads a pre-trained model, applies it to a dataset, and computes various evaluation metrics to quantify its performance. Below are the key features and workflow of the evaluator:

3.4.1 Key Features

- **Model Loading and Initialization**:
  - Supports loading both `GNNModel` (simple GNN) and `GeneralizedGNN`.
  - Automatically restores the model's weights from a specified directory (`save_dir`).
  - Configurable model parameters such as hidden dimensions, number of layers, and variant type (e.g., GCN, GraphSAGE, GAT).
- **Dataset Compatibility**:
  - Supports evaluation on various datasets (default: `Cora`).
- **Evaluation Metrics**:
  - Computes a range of standard metrics, including:
    1. **Accuracy**: Proportion of correctly classified nodes.

2. **Precision**: Measure of true positive predictions relative to all positive predictions.
3. **Recall**: Measure of true positive predictions relative to all actual positives.
4. **F1 Score**: Harmonic mean of precision and recall, representing a balance between the two.

- **Metadata Updates**:
  - Updates the saved model's metadata with evaluation results for reproducibility and tracking.

3.4.2 Workflow

The evaluation process is as follows:

1. Load the pre-trained model and dataset.
2. Move the data and model to the appropriate device.
3. Perform inference on the test nodes (`test_mask`) to obtain predicted class labels.
4. Compute evaluation metrics by comparing predictions against true labels.
5. Print the metrics and save them as part of the model's metadata.

3.5 Cross-Dataset Evaluator

The `CrossDatasetEvaluator` class provides a comprehensive framework for evaluating the performance of a pre-trained GNN model across multiple datasets. This process assesses the model's generalizability and adaptability to datasets with varying characteristics, such as differing feature dimensions or class distributions.

3.5.1 Key Features

- **Support for Multiple Datasets**:

  - The class can evaluate the model on a predefined set of datasets (default: `CiteSeer` and `PubMed`).

- **Feature Dimension Handling**:
  - Automatically detects mismatches between the feature dimensions of the dataset and the model's expected input.
  - Supports multiple techniques for resolving these mismatches:
    1. **Zero Padding**: Adds zero-valued features to match the target dimension.
    2. **Replication**: Duplicates existing features to fill the required dimensions.
    3. **Dimensionality Reduction (PCA)**: Reduces features to the target dimension for datasets with higher dimensionalities.

- **Evaluation Metrics**:
  - Computes standard metrics such as accuracy, precision, recall, and F1 score for node classification tasks.
  - Collects and stores dataset-specific information, including the number of nodes, features, and classes.

- **Metadata Integration**:
  - Stores cross-dataset evaluation results, including metrics and dimension-handling techniques, in the model's metadata for future reference.

3.5.2 Workflow

The evaluation process follows these steps:

1. Load the pre-trained model and move it to the appropriate computation device.
2. Iterate over the specified datasets and preprocess each one.
3. Detect and handle feature dimension mismatches using the configured technique (`auto`, `zero_pad`, `replicate`, or `pca`).
4. Perform inference on the test set and compute evaluation metrics.

5. Save the results, including dataset-specific metrics and dimension-handling details, as part of the model's metadata.

3.6 Execution Example

3.6.1 Train, Evaluate, and Cross-Test Using a Single Model Type

To train, evaluate, and cross-test a single GNN model, use the following command:

- Run the command:

```
python main.py single train evaluate cross_test
```

This command derives key parameters from the dataset loader:

- `input_dim = self.dataset.num_features = 1433`:
  Number of input features from the `Cora` dataset.
- `hidden_dim = 256`:
  Hidden layer dimension (default configuration).
- `num_classes = 7`:
  Number of classes, derived from the dataset loader (`DatasetLoader("Cora")`).

After execution, a folder will be created in the `saved_models` directory with a name in the format `single_timestamp`. This folder contains:

- `model.pth`: The trained model's weights.
- `metadata.json`: A file containing detailed metadata about the model, dataset, training configuration, and evaluation metrics.

3.6.2 Perform an Experiment with Different Model Architectures

To explore different configurations of the Generalized GNN Model, we can run experiments with varying parameters. This allows us to train and evaluate multiple models and select the one that performs best.

- Run the command:

```
python main.py experiment
```

This command trains, evaluates, and cross-tests models with different configurations, saving the results in the `saved_models` directory. The experiments are conducted over a range of parameter values:

- `hidden_dim_values`: [64, 128, 256, 512]
- `num_layers_values`: [2, 4, 8, 12]
- `variant_values`: [”gcn”, ”sage”, ”gat”]
- `dropout_values`: [0.3, 0.5, 0.7]
- `use_residual_values`: [True, False]
- `use_layer_norm_values`: [True, False]

For each combination of parameters, the experiment performs the following steps:

1. Train the model on the specified dataset.
2. Evaluate the model’s performance on the test set.
3. Cross-test the model on other datasets (e.g., CiteSeer and PubMed).

The results, including model weights and evaluation metrics, are saved in the `saved_models` directory under subfolders named with the timestamp and corresponding parameter configuration. This allows for systematic comparison and selection of the best-performing model.

After running the `experiment` command, you can execute `find_best` command:

```
python main.py find_best
```

It output lists the best-performing models for each dataset along with their respective accuracies. Below is an example output from an experiment:

- **Best Model for PubMed Dataset:**
  - `Model Name: generalized_20241216_095352`
  - `Accuracy: 0.3900`
- **Best Model for CiteSeer Dataset:**
  - `Model Name: generalized_20241216_122132`
  - `Accuracy: 0.2680`
- **Best Model for Cora Dataset:**
  - `Model Name: generalized_20241216_114950`
  - `Accuracy: 0.8220`

4 Link Prediction

For performing link prediction, you can execute `find_best` command:

```
python main.py link_prediction
```

4.1 Link Prediction Implementation

This section describes the implementation of the link prediction task, which involves training and evaluating a Graph Neural Network (GNN) to predict the existence of edges (links) in a graph. The implementation uses the `EdgePredictionTrainer` class and the `GeneralizedGNNForEdgePrediction` model.

4.1.1 Workflow for Link Prediction

The link prediction task is executed using the `run_link_prediction` function, which follows these steps:

1. **Dataset Loading:**
    - The dataset is loaded using the `DatasetLoader`, providing node features and edges.
    - Example datasets include Cora, CiteSeer, and PubMed.
2. **Model Configuration:**
    - A dictionary of hyperparameters is defined, including:
        - Input dimensions (`input_dim`) based on the number of node features.
        - Hidden and output dimensions (`hidden_dim` and `output_dim`).
        - Number of layers (`num_layers`) and additional settings (e.g., residual connections, dropout).
3. **Training and Evaluation:**
    - The `EdgePredictionTrainer` is used to train the GNN for edge prediction.

- Actions include:
  - **Training:** Optimizing the model on positive and negative edges.
  - **Evaluation:** Calculating metrics such as accuracy, precision, recall, F1-score, and ROC-AUC.

4.1.2 EdgePredictionTrainer Class

The `EdgePredictionTrainer` class handles the following:

- **Edge Splitting:**
  - The graph's edges are split into training and validation sets using a random permutation.
  - Negative edges (non-existent links) are dynamically sampled during training and validation.
- **Training Process:**
  - Positive and negative edges are concatenated to create a balanced dataset.
  - The model predicts scores for edges, and the loss is calculated using binary cross-entropy.
  - Validation is performed after each epoch, tracking the best model based on the validation ROC-AUC score.
- **Evaluation Process:**
  - The model's predictions on the validation/test set are converted to probabilities using the sigmoid function.
  - Metrics such as accuracy, precision, recall, F1-score, and ROC-AUC are computed to evaluate performance.

4.1.3 Generalized GNN for Edge Prediction

The `GeneralizedGNNForEdgePrediction` class defines the architecture of the GNN. Key features include:

- Support for different GNN variants (e.g., GCN, GraphSAGE, GAT).
- Residual connections and layer normalization for better training stability.
- An edge predictor layer to calculate edge scores based on node embeddings.

4.1.4 Console Output and Metrics

Running the link prediction task provides detailed outputs for training and evaluation. Below is an excerpt from the console output:

```
 Starting link prediction...
Starting training...
Epoch 1/100, Loss:  0.6934, Val AUC: 0.5492
Epoch 2/100, Loss:  0.6929, Val AUC: 0.5969
...
Epoch 100/100, Loss:  0.6181, Val AUC: 0.6160
Starting evaluation...
Evaluation Metrics:  {'accuracy':  0.564, 'precision':
0.650, 'recall':  0.279, 'f1_score':  0.390,
'roc_auc':  0.589}
```

Key Observations:

- **Training Metrics:**
  The loss decreases during training, and validation ROC-AUC improves, indicating model learning.
- **Evaluation Metrics:**
  Metrics like accuracy, precision, recall, F1-score, and ROC-AUC are returned at the end.

4.2 Analysis of Link Prediction Model

Model's Ability to Predict Links:

The link prediction model effectively identifies existing and non-existing edges by leveraging graph structures and node features. Metrics like ROC-AUC, precision, recall, and F1-score highlight its capability to detect true connections, though trade-offs between precision and recall reflect challenges with sparse or noisy graphs.

Practical Applications:

Link prediction has diverse applications, including:

- **Recommender Systems:** Suggesting user-item connections or friends in social networks.
- **Knowledge Graph Completion:** Filling in missing relationships in domains like healthcare or search engines.
- **Social Network Analysis:** Discovering communities or improving targeted marketing.
- **Fraud Detection:** Identifying suspicious activities in financial networks.
- **Drug Discovery:** Predicting molecular or protein interactions for new drug targets.

Challenges and Considerations:

Model performance depends on factors such as:

- **Data Sparsity:** Sparse graphs hinder generalization.
- **Feature Relevance:** Noisy features degrade accuracy.
- **Graph Variability:** Diverse structures may limit transferability.
- **Scalability:** Large graphs require efficient computation.

5 Node Embeddings Extraction and Visualization

5.1 Process of Extracting and Visualizing Node Embeddings

The process of extracting and visualizing node embeddings is outlined below:

Step 1: Embedding Extraction

The `extract_embeddings` function extracts node embeddings from a specified layer of a trained Generalized GNN (Graph Neural Network). The steps are:

- Metadata is loaded from the model directory to retrieve information about the dataset.
- The corresponding dataset is loaded using the `DatasetLoader`.
- A forward hook is registered on the specified layer (default: last layer) to capture its output during forward propagation.
- The model processes the graph, and the embeddings are collected.

Step 2: Embedding Visualization

The `analyze_and_visualize_embeddings` function applies dimensionality reduction techniques (e.g., PCA, t-SNE) to the high-dimensional embeddings and visualizes them in a 2D space. Each point in the plot corresponds to a node, and colors represent different node classes. This step aids in understanding how well the GNN separates nodes of different classes.

5.2 Visualization of Node Embeddings for the Cora Dataset

The visualization generated for the Cora dataset using PCA is shown below:

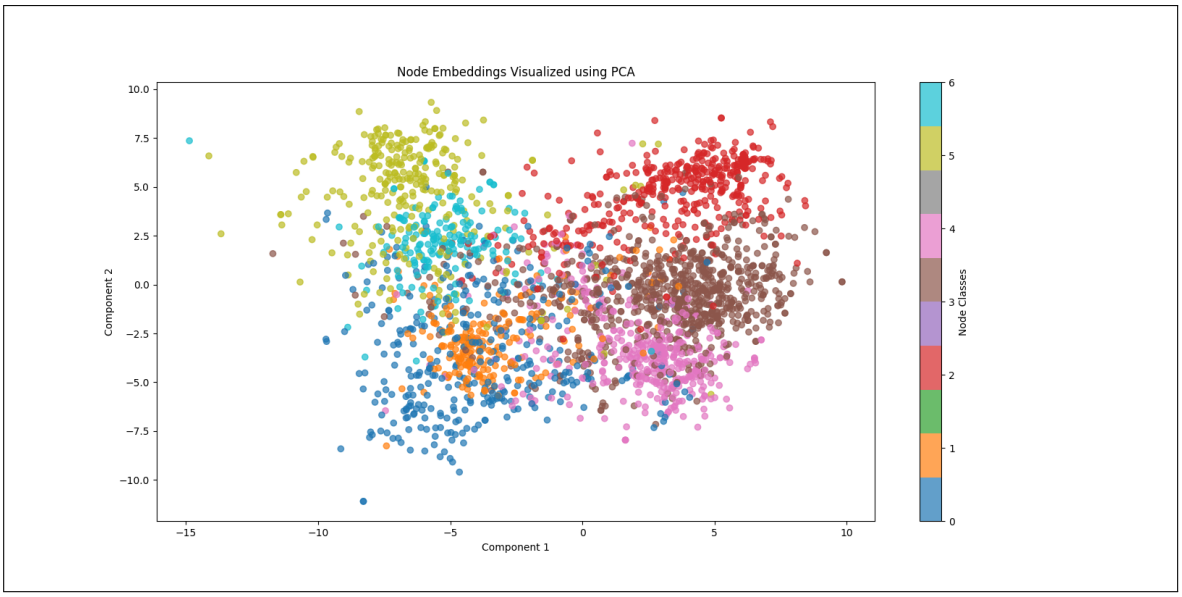

Figure 1: Visualization of Node Embeddings using PCA

5.3 Interpretation of Results

The visualization provides the following insights:

- **Cluster Formation:** Nodes of the same class (indicated by color) tend to form clusters. This indicates that the GNN successfully learns class-specific embeddings, separating nodes with similar properties in the embedding space.
- **Inter-class Separation:** Some overlap is visible between certain clusters, suggesting that the model struggles to fully distinguish between similar classes. This could be due to overlapping feature distributions or noisy edges in the graph.

6 Node2Vec Implementation and Results

6.1 Overview

Node2Vec is a powerful framework for generating node embeddings from graph data. It employs a biased random walk strategy to explore the graph and captures both local and global structural information through skip-gram modeling. These embeddings are then utilized for downstream tasks such as node classification and link prediction.

6.2 Execution

The Node2Vec process can be executed using the following command:

```
python main.py node2vec --dataset Cora --tasks
node_classification link_prediction
```

This command runs Node2Vec on the Cora dataset and evaluates its performance on node classification and link prediction tasks.

6.3 Training Process

During training, Node2Vec optimizes embeddings through random walks and skip-gram modeling. Key steps include:

- Generating random walks of specified length for each node.
- Using these walks to create positive and negative examples for skip-gram training.
- Optimizing the embeddings via stochastic gradient descent using a logistic loss function.

The training process outputs loss values over multiple epochs, indicating how well the model learns meaningful embeddings.

Example Output:

```
Epoch 1/100, Loss: 6.6019
Epoch 2/100, Loss: 6.4174
...
Epoch 100/100, Loss: 1.5908
Training complete!
```

6.4 Evaluation Results

Once embeddings are generated, Node2Vec evaluates its effectiveness on the following tasks:

6.4.1 Node Classification

Node classification uses the generated embeddings as input features to a logistic regression model. Metrics for classification performance include accuracy, precision, recall, and F1-score. The results for the Cora dataset are as follows:

- **Accuracy:** 35%
- **Precision:** 34.48%
- **Recall:** 35.71%
- **F1-Score:** 33.94%

These results highlight the model's ability to capture class-specific information within the embeddings.

6.4.2 Link Prediction

Link prediction evaluates the embeddings' capability to predict the existence of edges in the graph. Metrics include:

- **Accuracy:** 75.12%
- **Precision:** 66.78%
- **Recall:** 99.98%
- **F1-Score:** 80.07%

The high recall indicates that the model is effective in identifying true connections, while the slightly lower precision suggests occasional false positives.

6.5 Conclusion

Node2Vec generates versatile node embeddings that effectively represent graph structures. While performance on node classification tasks highlights room for improvement, link prediction metrics demonstrate its capability to capture meaningful graph relationships. These embeddings are valuable for tasks requiring structural and contextual understanding of graphs.

7 Explainability

7.1 Overview

Explainability in Graph Neural Networks (GNNs) is crucial for understanding the rationale behind model predictions. By identifying the most important features, nodes, and edges that influence a prediction, we can gain insights into the decision-making process of GNNs. This section outlines the implementation and results of explainability using the GNNExplainer algorithm.

7.2 Implementation

The explainability module uses the GNNExplainer to highlight significant graph components for a given node prediction. The process is executed via the following command:

```
python main.py explain --dataset Cora
```

Key steps in the implementation include:

- **Node Explanation**: The GNNExplainer identifies important edges and node features influencing the prediction for a specific node.
- **Edge Masking**: An edge importance mask is generated to indicate which edges were critical for the model's decision.
- **Visualization**: A subgraph centered on the explained node is visualized with highlighted edges and nodes, offering an intuitive understanding of the explanation.

7.3 Example Results

The visualization output for a selected node (Figure **??**) shows:

- Red nodes represent the class to which the node belongs.
- Gray nodes indicate other nodes in the subgraph.
- Blue edges denote significant edges identified by the explainer as critical to the prediction.

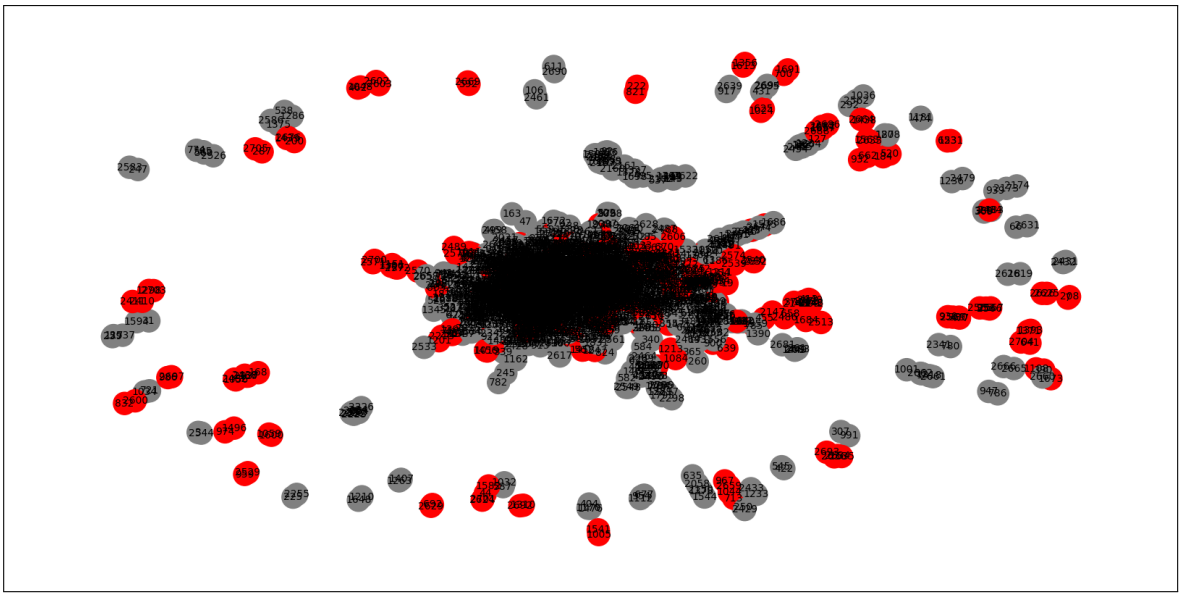

Figure 2: Visualization of

7.4 Analysis

The visualization highlights the most critical parts of the graph structure influencing the model's decision. This information can be used for:

- **Model Debugging**: Understanding why the model made a specific prediction helps in diagnosing issues such as overfitting or bias.
- **Trust and Interpretability**: Providing insights into the model's decision-making process increases user trust in AI systems.
- **Feature Engineering**: Identifying key features and relationships can guide feature selection or graph preprocessing for improved performance.