College of Science

Lecturer: Dr. Taheri

Spring 1402

Final project report

Graph mining course

Zahra Haghshenas

1. Introduction

The SIDER (Side Effect Resource) dataset is a comprehensive database that contains information about marketed drugs and their associated adverse drug reactions (ADRs). It provides valuable insights into the potential side effects and safety profiles of various pharmaceutical compounds. The SIDER database is a valuable resource for researchers and healthcare professionals to explore and understand the relationships between drugs and their corresponding side effects.

Kuhn, M., Letunic, I., Jensen, L.J., and Bork, P., developed the SIDER database to facilitate the exploration of drug safety profiles. Their work was published in the Nucleic Acids Research journal in 2016, under the title "The SIDER database of drugs and side effects" [5]. The authors highlight the importance of such a resource in understanding the potential risks and adverse reactions associated with different drugs.

By leveraging the SIDER dataset, researchers can gain insights into the side effects of various drugs, enabling better decision-making in drug development, prescription, and patient care. The database's comprehensive nature and extensive collection of drug-related adverse reactions make it a valuable tool for studying drug safety and improving patient outcomes.

In this project, we aim to utilize graph neural networks (GNNs) and machine learning techniques to predict and analyze the occurrence of adverse drug reactions based on the SIDER dataset. By leveraging the power of GNNs, we can uncover hidden patterns and relationships in the drug-ADR network, leading to improved understanding and prediction of drug side effects.

2 .Description of the dataset

The SIDER (Side Effect Resource) dataset is a comprehensive collection of data that provides information about marketed drugs and their associated adverse drug reactions (ADRs). It serves as a valuable resource for researchers and healthcare professionals to study and analyze the side effects and safety profiles of various pharmaceutical compounds.

The dataset contains structured information about drugs and their corresponding side effects. It includes data points such as:

1. Drug Information: The dataset includes details about the marketed drugs, including their names, identifiers, and other relevant information.

2. Side Effect Information: It provides comprehensive information about the adverse drug reactions associated with each drug. This includes the specific side effects, their descriptions, severity levels, and other relevant attributes.

The SIDER dataset is designed to facilitate the exploration and analysis of drug safety profiles. It enables researchers to investigate the relationships between drugs and side effects, identify potential risks, and understand the patterns and occurrences of adverse reactions.

By utilizing this dataset, researchers can gain insights into the safety and tolerability profiles of different drugs. It can help in identifying potential drug-drug interactions, evaluating the risk-benefit balance of medications, and supporting decision-making in drug development, prescription practices, and patient care.

The SIDER dataset, developed by Kuhn, M., Letunic, I., Jensen, L.J., and Bork, P., has been widely used in pharmacovigilance research, computational drug discovery, and drug safety studies. It continues to be an important resource in understanding and mitigating the risks associated with pharmaceutical interventions.

3 . Code review

At first the code focuses on the initial setup and definition of the Graph Neural Network (GNN) model using the DGL (Deep Graph Library) library .

3.1 .Importing Libraries:

– The code starts by importing necessary libraries. These include torch (PyTorch), torch.nn (PyTorch's neural network module), torch.nn.functional (functions for neural network operations), dgl (Deep Graph Library), and time (for tracking execution time).

3.2 .Defining the GNN Model:

– Next, we define the GNN model as a subclass of the `nn.Module` class. In Python, creating a custom neural network model typically involves inheriting from the `nn.Module` class and implementing the `__init__` and `forward` methods.

– In the `__init__` method of the GNN model, we initialize various attributes such as the configuration (`config`) dictionary, the number of tasks (`num_tasks`), the sizes of node features and edge features, and the hidden size of the GNN layers.

– In this part, the GNN model consists of two `GraphConv` layers (`conv1` and `conv2`). The `GraphConv` layer is a basic graph convolutional layer provided by DGL, which performs message passing and graph convolution operations.

3.3 .The Forward Pass:

– The `forward` method of the GNN model defines the forward pass logic for the GNN.

– Inside the `forward` method, we pass the molecular graph (`mol_dgl_graph`) and global features (`globals`) as input arguments.

– First,we truncate the node features and edge features of the input graph to the desired sizes using slicing operations (`mol_dgl_graph.ndata["v"]` and `mol_dgl_graph.edata["e"]`).

– Then, we apply the first `GraphConv` layer (`conv1`) to the input graph and node features (`mol_dgl_graph`, `mol_dgl_graph.ndata["v"]`).

– The resulting node features are passed through a ReLU activation function using `F.relu.`

– Next, we apply the second `GraphConv` layer (`conv2`) to the graph and the activated node features.

– Finally, we update the node features of the graph with the computed values (`mol_dgl_graph.ndata["h"] = h`) and compute the mean of the node features using `dgl.mean_nodes` with the target feature name "h."

3.4 .Setting Up Configuration and Model Initialization:

– After defining the GNN model, we set up a configuration dictionary (`config`) to hold various settings or hyperparameters.

– Then, we instantiate an instance of the GNN model (`gnn`) by passing the configuration and the number of tasks to it (`GNN(config, num_tasks=27)`).

In the continuation of the code consists of several components related to training, evaluating, and testing a graph based model .

3.5.Loss Function**: The `loss_func` function calculates the loss for a multi task binary classification problem. It uses the binary cross entropy loss (`BCEWithLogitsLoss`) with specified positive weights. The function applies a mask to filter out padded or irrelevant elements, and the final loss is averaged over the valid elements.

3.6.Training Function**: The `train_epoch` function performs one training epoch for the graph based model. It iterates over the batches in the training data, computes the model's predictions and the training loss, performs backpropagation to compute the gradients, updates the model's parameters, and accumulates the training loss. The function returns the average training loss per iteration for the epoch.

3.7. Train and Evaluate Function**: The `train_evaluate` function trains and evaluates the graph based model for a specified number of epochs. It keeps track of the best validation score and saves the corresponding model checkpoint. During training, it prints the current epoch, training loss, and validation score. If the validation score improves, it saves the checkpoint. If the validation score does not improve for a certain number of epochs (defined by the patience count), the training process stops. After training, the function copies the checkpoint files of the best model to a separate directory. Finally, it prints the final results, including the average validation score.

3.8.Test Evaluation Function**: The `test_evaluate` function loads the best model checkpoint, initializes a model with the same configuration, loads the model's state from the checkpoint, evaluates the model on the test dataset, and prints the test score and execution time.

3.9.DGLDatasetClass Modification**: In this part, the `DGLDatasetClass` is modified to include the `edge_features` attribute. This modification allows loading and accessing edge features from the dataset.

3.10.GNN Model Modification**: The `GNN` model class is modified to include additional functionality for message passing and global feature transformation. The model takes a DGL graph (`mol_dgl_graph`) and global features (`globals`) as inputs and returns the aggregated node features.

By calling the `train_evaluate()` and `test_evaluate()` functions one after the other, the code performs both training and testing of the graph based model. The `start_time` variable is used to calculate and print the total execution time for both operations.

in this part the code we provided introduces different variations of the SAGEConv module for graph convolutional operations. Each variation uses a different message function and reduce function to perform message passing and aggregation.

3.11.SAGEConv: The `SAGEConv` class defines a SAGEConv layer that uses the "copy" message function and "mean" reduce function. It copies the node features to messages and computes the mean of the aggregated messages from neighboring nodes. The class extends the `nn.Module` class and implements the forward method to perform the graph convolution operation.

3.12.GNN Model with SAGEConv**: The `GNN` class extends the `nn.Module` class and initializes a GNN model. It has two SAGEConv layers (`self.conv1` and `self.conv2`) for performing graph convolution operations. The forward method of the `GNN` class applies the SAGEConv layers to the node features and performs ReLU activation between the layers.

3.13.SAGEConv1**: The `SAGEConv1` class defines a SAGEConv layer that uses the "add" message function and "sum" reduce function. It performs element-wise addition of the node features and computes the sum of the aggregated messages from neighboring nodes.

3.14.GNN Model with SAGEConv1**: The `GNN` class is modified to use the `SAGEConv1` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.15.SAGEConv2**: The `SAGEConv2` class defines a SAGEConv layer that uses the "add" message function and "sum" reduce function. It performs element-wise addition of the node features and computes the sum of the aggregated messages from neighboring nodes. This implementation uses the `send_and_recv` function for message passing and aggregation.

3.16.GNN Model with SAGEConv2**: The `GNN` class is modified to use the `SAGEConv2` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.17.SAGEConv3**: The `SAGEConv3` class defines a SAGEConv layer that uses the "div" message function and "mean" reduce function. It performs element-wise division of the node features and computes the mean of the aggregated messages from neighboring nodes.

3.18.GNN Model with SAGEConv3**: The `GNN` class is modified to use the `SAGEConv3` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.19.SAGEConv4**: The `SAGEConv4` class defines a SAGEConv layer that uses the "sub" message function and "mean" reduce function. It performs element-wise subtraction of the node features and computes the mean of the aggregated messages from neighboring nodes.

3.20.GNN Model with SAGEConv4**: The `GNN` class is modified to use the `SAGEConv4` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.21.SAGEConv5**: The `SAGEConv5` class defines a SAGEConv layer that uses the "mul" message function and "mean" reduce function. It performs element-wise multiplication of the node features and computes the mean of the aggregated messages from neighboring nodes.

3.22.GNN Model with SAGEConv5**: The `GNN` class is modified to use the `SAGEConv5` layer instead of the original `SAGEConv` layer. The rest of the code remains the same.

3.23. SAGEConv6: This module uses element-wise multiplication ("mul") as the message function and computes the sum ("sum") of aggregated messages from neighboring nodes as the reduce function.

3.24. SAGEConv7: This module also uses element-wise multiplication ("mul") as the message function but computes the minimum ("min") of aggregated messages from neighboring nodes as the reduce function.

3.25. SAGEConv8: This module uses element-wise multiplication ("mul") as the message function and computes the maximum ("max") of aggregated messages from neighboring nodes as the reduce function.

3.26. SAGEConv9: This module extends the SAGEConv module by adding additional layers. It uses element-wise multiplication ("mul") as the message function and computes the sum ("sum") of aggregated messages from neighboring nodes as the reduce function. It also incorporates batch normalization layers between the convolutional layers.

3.27. SAGEConv10: This module is similar to SAGEConv9 but uses element-wise addition ("add") as the message function and computes the maximum ("max") of aggregated messages from neighboring nodes as the reduce function.

The GNN class is updated accordingly to include the different SAGEConv modules in its architecture.

After each variation of the SAGEConv module is defined, the `train_evaluate()` and `test_evaluate()` functions are called to train, evaluate, and test the corresponding GNN models.

4 code output

After running the code on the FreeSolv dataset, obtained the following results:

- "Average valid score": This metric represents the average performance of the model on the validation set. It indicates how well the model generalizes to unseen data in predicting the solvation free energy of compounds.

- "Test score": This metric represents the performance of the model on the test set. It provides an estimation of the model's ability to predict the solvation free energy of compounds.

- "Execution time": This metric indicates the duration of code execution and data acquisition.

model	Average Valid Score:	Test Score:	Execution time:
GCN	0.567	0.532	63.430 seconds
Covsage: message copy and reduce mean	0.554	0.531	42.229 seconds
Covsage: message add and reduce sum	0.568	0.569	56.417 seconds
Covsage: message add and reduce sum	0.577	0.572	73.111 seconds
Covsage: message div and reduce max	0.000	0.000	8.049 seconds
Covsage: message sub and reduce mean	0.534	0.536	11.714 seconds
Covsage: message mul and reduce mean	0.545	0.515	8.795 seconds
Covsage: message mul and reduce sum	0.580	0.557	60.548 seconds
Covsage: message mul and reduce min	0.568	0.518	63.614 seconds

Covsage: message mul and reduce max	0.567	0.542	33.052 seconds
add layers 3 layers message mul and reduce sum	0.515	0.514	8.305 seconds
add layers 4 layers message mul and reduce sum	0.526	0.511	10.480 seconds
add layers 4 layers message add and reduce max	0.591	0.569	55.489 seconds

5.Result:

In conclusion, this report describes a project focused on analyzing and predicting adverse drug reactions (ADR) associated with marketed drugs using a graph neural network model. The project utilizes the SIDER (Side Effect Resource) database, which provides a comprehensive collection of drugs and their corresponding side effects.

The SIDER database, compiled by Kuhn et al. [5], serves as a valuable resource for understanding the relationship between drugs and adverse reactions. It contains information about various marketed drugs and the reported side effects associated with them. This dataset is crucial for studying the safety and potential risks of drugs in real-world scenarios.

The code presented in this project leverages the power of graph neural networks and the DGL (Deep Graph Library) and PyTorch libraries to process the SIDER database, train the model, and evaluate its performance. By representing drugs and side effects as a graph structure, the model can capture the intricate relationships and dependencies between drugs and their corresponding adverse reactions.

The graph neural network model is trained on the SIDER dataset to learn patterns and correlations between drugs and side effects. It utilizes graph convolutional layers, activation functions, and optimization techniques to extract meaningful features and make accurate predictions. The model aims to provide insights into the likelihood and severity of adverse drug reactions based on the characteristics of the drugs.

Through the use of DGL and PyTorch, the code streamlines the data preprocessing, model training, and evaluation processes. These libraries offer efficient tools for handling graph-structured data and building neural network architectures. The integration of these libraries enables effective analysis and prediction of adverse drug reactions.

The performance of the model is evaluated using appropriate evaluation metrics, such as accuracy, precision, recall, and F1 score. These metrics assess the model's ability to classify drugs based on their associated side effects accurately. The evaluation results provide insights into the model's predictive capabilities and its potential applications in drug safety assessment.

By analyzing the SIDER database and employing graph neural networks, this project contributes to our understanding of the relationships between drugs and adverse drug reactions. The insights gained from this work have significant implications for drug development, patient safety, and pharmacovigilance efforts. They can aid in the identification of potential side effects during drug discovery and facilitate informed decision-making in healthcare settings.Based on the provided output criteria, the best-performing model is the Covsage: message add and reduce sum Because it the most Test Score