Deep Learning Project

# Disease-gene interaction prediction with graph neural networks

Csibi Alexandra

GPVFEV

# Introduction

Graph Neural Networks (GNNs) are transforming data analysis by uncovering complex relationships across domains, with significant applications in bioinformatics for understanding disease-gene associations, which are crucial for advancing medical research and treatments. This project leverages GNNs and the DisGeNET database, a comprehensive source of disease-gene interactions, to predict these associations using advanced models, a curated dataset, and an interactive application. The solution includes tools for data acquisition, graph preparation, model training, and real-time prediction, offering a balanced approach to computational efficiency and accuracy. This documentation provides a detailed overview of the project's components, models, evaluation metrics, and interactive features, empowering users to explore the intersection of deep learning and bioinformatics while addressing real-world challenges.

# Technological background

GNNs are specialized neural networks designed to process graph-structured data, capturing complex relationships among entities. They are particularly effective in domains where data is naturally represented as graphs, such as social networks, molecular structures, and recommendation systems. By leveraging the connections between nodes, GNNs can model dependencies and interactions that traditional neural networks might overlook.

## GNN working and architecture

How GNNs Work Graph Neural Networks (GNNs) process data represented as graphs by enabling nodes to exchange information with their neighbors. This mechanism, known as message passing, allows GNNs to learn both node-level and graph-level representations effectively.

1. **Initialization**: Nodes are initialized with their feature vectors, such as a user's profile attributes in a social network or atomic properties in a molecule.
2. **Message Passing**: Each node shares its features with connected neighbors, facilitating the exchange of information. This iterative process helps nodes learn from their immediate and extended neighborhoods.
3. **Aggregation**: Nodes aggregate the received messages from neighbors using methods like summation, averaging, or maximum pooling, capturing neighborhood-level information.
4. **Update**: Aggregated data is combined with the node's current features to update its representation, enabling better contextual understanding.
5. **Prediction**: After multiple rounds of message passing and feature updates, the GNN uses the learned node representations to make predictions about nodes, edges, or the entire graph.

GNNs have a modular architecture tailored for graph-based tasks. Below are the main components:

## Input layer

- **Node features**: Attributes specific to each node, such as profile details or molecular properties.
- **Edge features (optional)**: Characteristics of connections between nodes, like bond types in molecules.
- **Graph representation**:
    - **Node feature matrix**: Contains features for all nodes.
    - **Adjacency matrix**: Encodes edges, showing how nodes are interconnected.

## Graph convolution layers

The core of GNNs, these layers enable information exchange between nodes.

- **Message passing**: Nodes send their feature data to neighbors.
- **Aggregation**: Neighbors' messages are aggregated, summarizing local neighborhood information.
- **Update**: Aggregated data is integrated with existing features to refine node representations.

## Stacking layers

Multiple graph convolution layers are stacked to propagate information further across the graph, enabling each node to learn from both immediate and distant neighbors.

## Pooling layers

Pooling reduces the graph size by summarizing node features, similar to pooling in CNNs. Common methods:

- **Global mean/max pooling**: Averages or selects the maximum of node features.
- **Graph attention pooling**: Uses attention mechanisms to focus on critical nodes.

## Readout layer

Aggregates information from all nodes to form a unified graph representation, particularly useful for graph-level predictions like classifying molecules.

## Output layer

Tailored to the task:

- **Node classification**: Predicts labels for individual nodes.
- **Edge prediction**: Determines connections or their types between nodes.
- **Graph classification**: Assigns labels to entire graphs.

This modular structure enables GNNs to generalize across diverse graph-based tasks, leveraging their ability to model both node features and structural relationships. [2]

## Key GNN models

Several GNN architectures have been developed to address specific challenges in graph representation, these are the ones that were implemented in this project:

### Graph Convolutional Networks (GCNs)

The research primarily focuses on graph neural networks for classification tasks and hence the most used GNN architecture viz. GCN is used as a baseline model. Comparing the fundamental structure of GCN with more advanced versions of GNNs would aid in determining whether enhanced GNN variants offer a substantial improvement in performance. GCN utilizes the convolutional mechanism to propagate information across a graph by aggregating features from neighboring nodes. GCN is highly effective in capturing local graph structures. [3]

### GraphSAGE (Graph Sample and Aggregate)

GraphSAGE involves node sampling followed by aggregation of the sampled node features using an aggregation function such as mean aggregation or pooling aggregation. With its localized sampling and aggregation approach, the model can efficiently handle large graphs. Moreover, GraphSAGE exhibits insensitivity to node ordering, making it a key contender as a model for tasks related to graph classification. GraphSAGE is investigated for its ability to capture more localized and diverse information from the graph. GraphSAGE is also the state-of-the-art model in efficiently adapting to diverse graph types. [3]

### Graph Isomorphism Networks (GINs)

GIN is permutation invariant and hence can address the over-smoothing problems encountered by GCN. GIN is most typical for processing isomorphic graphs and is researched for its flexibility in capturing complex graph patterns. GIN consists of isomorphism layers that are invariant to node ordering. The node features are aggregated using summation allowing the model to capture relationships within a graph. GIN, characterized by its isomorphic and permutation invariant properties, has considerable potential in effectively capturing intricate structural patterns present within graphs. Therefore, it is crucial to explore the model's capabilities in the context of graph classification rather than focusing primarily on node classification. [3]

## Link prediction with GNNs

Link prediction involves forecasting the existence of a connection between two nodes in a graph, a task vital in various applications like social network analysis, recommendation systems, and biological network modeling. Traditional methods relied on heuristic-based similarity measures, which often lacked flexibility and generalization capabilities. GNNs have revolutionized link prediction by learning from both the graph structure and node features. They aggregate information from neighboring nodes to capture local and global

structural patterns, enabling the prediction of missing or future links with higher accuracy. [4]

The prediction steps are described below:

1. An encoder creates node embeddings by processing the graph with two convolution layers.
2. We randomly add negative links to the original graph. This makes the model task binary classifications with the positive links from the original edges and the negative links from the added edges.
3. A decoder makes link predictions (i.e. binary classifications) on all the edges including the negative links using node embeddings. It calculates a dot product of the node embeddings from pair of nodes on each edge. Then, it aggregates the values across the embedding dimension and creates a single value on every edge that represents the probability of edge existence. [1]

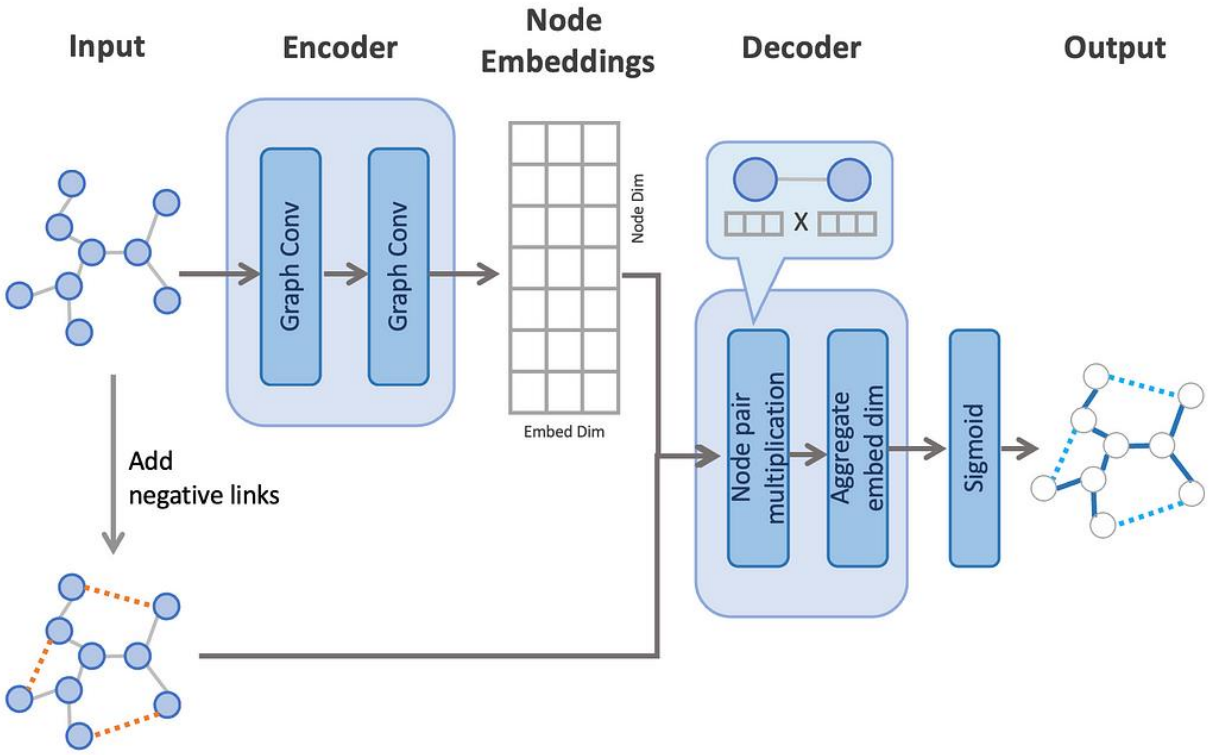


Figure 1: Link Prediction Model Architecture [1]

# Implementation

## Repository structure and components

The implementation of this project is organized into well-defined sections, ensuring clarity and modularity. Below, we detail the structure and functionality of the repository, describing how each component contributes to achieving the project's goals.

## Data

The repository includes preprocessed and serialized data essential for the graph neural network (GNN) models:

- **GDA_df.csv**: This file contains preprocessed gene-disease associations, serving as the foundation for training the models.
- **graph_data.pkl**: A serialized file containing the homogeneous graph data used by the models.

## Model

Four GNN architectures were implemented to handle link prediction for gene-disease associations. Each model offers distinct advantages:

- **GCN_DP**: This basic model uses two GCNConv layers for encoding and a simple dot product for classification. It is straightforward yet effective for baseline link prediction tasks.
- **GCN_MLP**: Enhancing the encoding with a more complex decoder, this model uses a Multilayer Perceptron (MLP) for classification, improving accuracy over the simpler GCN_DP.
- **GraphSAGE_MLP**: Featuring SAGEConv layers, this model is designed for large-scale graphs by sampling neighbors during training. An MLP decoder further refines predictions.
- **GIN_MLP**: With GINConv layers for encoding, this model captures fine-grained structural properties of the graph, making it the most effective for distinguishing associations.

Each model's best-performing configurations are saved in dedicated weight files (GCN_DP.pth, GCN_MLP.pth, GraphSAGE_MLP.pth, and GIN_MLP.pth), ensuring reproducibility and ease of deployment.

## Code components

The repository contains several scripts that handle data processing, graph preparation, model training, and application development:

### Data handling

- **data_acquisition_processing.py:** Automates data fetching and preprocessing, transforming raw data from DisGeNET into formats usable by the models. It includes the get_data function, which processes gene-disease associations based on user-specified disease types.
- **graph_preparation.py:** Prepares the graph structure, including node features (combining gene, disease, and type indicators) and edge indices, like assigning unique IDs to nodes and creates edges based on associations. Reverse edges are added to enhance embedding quality.

**Model training**

- **trainer.py:** Implements a Trainer class for training, evaluation, and testing workflows. It employs *BCEWithLogitsLoss* for classification and supports advanced features such as learning rate scheduling (*CosineAnnealingLR*) and early stopping for optimized training. Evaluation metrics include AUC, F1 score, confusion matrix, and a dynamically determined threshold for binary classification. The testing is implemented by loading the best model checkpoint to evaluate test performance.

**Interactive application**

- **gradio_app.py:** Provides a user-friendly Gradio-interface for gene-disease link prediction. Users input gene and disease IDs, and the app fetches their features from DisGeNET API. After that processes input to match the training graph format and updates the graph if necessary. Uses the best-performing model to predict associations and outputs binary classification based on the best threshold.

### Containerization

A Docker-based setup ensures that the project is portable and easy to run.

- **Dockerfile**: Defines the environment for running either JupyterLab (for development) or the Gradio application (for interaction). Includes all dependencies for data processing, model training, and serving.
- **start.sh**: Simplifies starting the Docker container, allowing users to choose between JupyterLab and the Gradio app.

## Installation and usage instructions

The project can be set up via two methods:

1. **Clone and build locally**: Clone the repository and build a lightweight Docker image.
2. **Pull prebuilt image**: Use the publicly available Docker image (alexandracsibi/deeplearning-project).

### Running the Project

After obtaining the Docker image, you can either run JupyterLab or the Gradio app. If you built the Docker image locally, replace *alexandracsibi/deeplearning-project:latest* with your local image name.

**Run JupyterLab**:

Use JupyterLab to explore the project by testing data acquisition, graph data preparation, and training or evaluating models. Run the following command:

```
docker run -p 8888:8888 -it alexandracsibi/deeplearning-project:latest
```

After starting the container, connect to JupyterLab at http://localhost:8888. You will be prompted to enter a password ("kicsikutya").

**Run the Gradio App**:

Use the Gradio app to interact with the pre-trained models for gene-disease link prediction. Run the following command:

```
docker run -p 7860:7860 -it alexandracsibi/deeplearning-project:latest
```

Access the app at http://localhost:7860.

# Evaluation and performance

The project evaluated four GNN models using standard metrics (F1 score, AUC, and confusion matrices). The performance metrics for the models trained and evaluated in this project are summarized below:

| Metric | GCN_DP | GCN_MLP | GraphSAGE_MLP | GIN_MLP |
|---|---|---|---|---|
| F1 | 0.8835 | 0.9020 | 0.9217 | 0.9252 |
| AUC | 0.9414 | 0.9640 | 0.9272 | 0.9716 |

## Confusion Matrices

### GCN_DP model

| | Predicted Positive | Predicted Negative |
|---|---|---|
| Actual Positive | 2890 | 422 |
| Actual Negative | 357 | 2955 |

### GCN_MLP model

| | Predicted Positive | Predicted Negative |
|---|---|---|
| Actual Positive | 3044 | 268 |
| Actual Negative | 371 | 2941 |

**GraphSAGE_MLP model**

| | Predicted Positive | Predicted Negative |
|---|---|---|
| Actual Positive | 2781 | 531 |
| Actual Negative | 27 | 3285 |

**GIN_MLP model**

| | Predicted Positive | Predicted Negative |
|---|---|---|
| Actual Positive | 3089 | 223 |
| Actual Negative | 269 | 3043 |

The GIN model demonstrated the best performance in separating positive and negative edges, as evidenced by its high F1 score, AUC, and confusion matrix on the test dataset. This exceptional edge classification capability translates directly to its effectiveness in the Gradio app for individual gene-disease predictions. Its precision in distinguishing associations makes it the most reliable model for practical applications in real-world scenarios.

# Contribution note

In this project:

- The data fetching process was a collaborative effort between me and my former teammate (Nagypál Márton Péter, Q88P3E).
- All subsequent steps, including data processing, graph preparation, model training, evaluation, and application development, were completed individually by me.

# Bibliography

[1] *Graph Neural Networks with PyG on Node Classification, Link Prediction, and Anomaly Detection*. Towards Data Science. Retrieved from https://towardsdatascience.com/graph-neural-networks-with-pyg-on-node-classification-link-prediction-and-anomaly-detection-14aa38fe1275

[2] *Graph Neural Networks Demystified: How They Work, Their Applications, and Essential Tools*. Ignitarium. Retrieved from https://ignitarium.com/graph-neural-networks-demystified-how-they-work-their-applications-and-essential-tools/?utm_source=chatgpt.com

[3] *Towards Causal Classification: A Comprehensive Study on Graph Neural Networks*. arXiv preprint arXiv:2401.15444. Retrieved from https://arxiv.org/html/2401.15444v1

[4] *Graph Neural Networks: Link Prediction*. In L. Wu, P. Cui, & J. Pei (Eds.), *Graph Neural Networks: Foundations, Frontiers, and Applications* (Chapter 10). Retrieved from https://graph-neural-networks.github.io/static/file/chapter10.pdf