TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

ĐỒ ÁN MÔN HỌC
LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Tên đề tài: Ứng Dụng Mạng Nơ-ron Đồ Thị Đồng Nhất (Homogeneous GNN) và Đồ Thị Hỗn Hợp (Heterogeneous GNN) Trong Phân Loại Nút và Dự Đoán Liên Kết Giữa Thuốc và Gen

Nhóm sinh viên thực hiện : Nguyễn Khoa Ninh - 20226117
Dương Thái Anh - 20226099

Lớp : 152259

HÀ NỘI, 12/2024

Mục lục

I. Giới Thiệu

Trong lĩnh vực y học và dược phẩm, việc hiểu rõ mối quan hệ giữa các loại thuốc và gen là vô cùng quan trọng để phát triển các liệu pháp điều trị hiệu quả và cá nhân hóa. Các mạng nơ-ron đồ thị (Graph Neural Networks - GNN) đã chứng minh được khả năng mạnh mẽ trong việc xử lý và khai thác thông tin từ các cấu trúc dữ liệu phức tạp như mạng lưới tương tác thuốc-gen. Dự án này nhằm ứng dụng cả hai mô hình GNN đồng nhất và GNN hỗn hợp để phân loại các node (thuốc và gen) cũng như dự đoán các liên kết tiềm năng giữa chúng, từ đó hỗ trợ việc khám phá và phát triển các tương tác thuốc-gen mới.

II. Cơ Sở Lý Thuyết

1. Mạng Nơ-ron Đồ Thị (GNN)

GNN là một loại mạng nơ-ron được thiết kế đặc biệt để xử lý dữ liệu có cấu trúc đồ thị. GNN khai thác thông tin từ các node và các cạnh kết nối chúng để học các biểu diễn (embeddings) có ý nghĩa, phục vụ cho các nhiệm vụ như phân loại node, phân loại cạnh, hoặc dự đoán liên kết.

2. Phân Loại Node và Dự Đoán Liên Kết

- **Phân Loại Node**: Xác định nhãn hoặc thuộc tính của các node trong đồ thị dựa trên thông tin cấu trúc và đặc trưng của chúng.

- **Dự Đoán Liên Kết (Link Prediction)**: Dự đoán sự tồn tại của các liên kết tiềm năng giữa các node chưa được kết nối trong đồ thị hiện tại.

3. Mạng Nơ-ron Đồ Thị Đồng Nhất vs. Hỗn Hợp

- **GNN Đồng Nhất (Homogeneous GNN)**: Xử lý đồ thị chỉ có một loại node và một loại cạnh duy nhất. Mô hình này đơn giản hơn nhưng có thể thiếu khả năng biểu diễn sự đa dạng của các loại node và cạnh trong các mạng phức tạp.
- **GNN Hỗn Hợp (Heterogeneous GNN)**: Xử lý đồ thị có nhiều loại node và nhiều loại cạnh khác nhau. Mô hình này phức tạp hơn nhưng cho phép khai thác sâu hơn cấu trúc đa dạng của dữ liệu.

4.Cytoscape

Cytoscape là một nền tảng mã nguồn mở dùng để phân tích và trực quan hóa các mạng sinh học. Thư viện py4cytoscape cho phép tích hợp và tương tác với Cytoscape từ các môi trường Python, hỗ trợ việc hiển thị và phân tích mạng lưới tương tác.

III. Mô Tả Vấn Đề

Trong nghiên cứu này, chúng tôi tập trung vào việc phân loại các node trong mạng tương tác thuốc-gen và dự đoán các liên kết tiềm năng giữa các thuốc và gen. Mục tiêu chính bao gồm:

1. **Phân Loại Node**: Xác định xem một node đại diện cho thuốc hay gen, hoặc phân loại nhãn thuộc tính cụ thể nếu có.
2. **Dự Đoán Liên Kết**: Tìm ra các cặp thuốc-gen có khả năng tương tác nhưng chưa được ghi nhận trong dữ liệu hiện tại.

IV. Thiết Kế Thuật Toán, Công Nghệ và Thư Viện Sử Dụng

1. Thuật Toán

Toán Học Cơ Bản Của Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) là một nhánh của mạng neural được thiết kế để làm việc với dữ liệu có cấu trúc đồ thị. GNNs học được các biểu diễn (embeddings) của các node, cạnh hoặc cả đồ thị thông qua việc truyền và cập nhật thông tin giữa các node trong đồ thị.

1.1. Cấu Trúc Đồ Thị

Một đồ thị được định nghĩa bởi:

- **Node (đỉnh):** Các phần tử cơ bản của đồ thị.
- **Edge (cạnh):** Các kết nối giữa các node.

Ma trận kề và ma trận độ là hai đại diện phổ biến cho cấu trúc đồ thị:

- **Ma Trận Kề :**
  Trong đó nếu có cạnh nối từ node đến node , ngược lại .
- **Ma Trận Độ :**
  là ma trận đường chéo với các phần tử bằng độ của node (số lượng cạnh nối đến node).

1.2. Chuẩn Hóa Ma Trận Kề

Để ổn định quá trình truyền thông điệp và đảm bảo cân bằng thông tin từ các node lân cận, ma trận kề thường được chuẩn hóa. Phương pháp chuẩn hóa phổ biến nhất là chuẩn hóa đối xứng theo ma trận độ đã được thêm self-loop.

- **Thêm Self-Loop:**
  Trong đó là ma trận đơn vị, đảm bảo mỗi node cũng nhận thông tin từ chính nó.
- **Chuẩn Hóa Đối Xứng:**
  Trong đó là ma trận độ của ma trận kề sau khi đã được thêm self-loop.

1.3. Cơ Chế Truyền Thông Điệp (Message Passing)

Quá trình truyền thông điệp trong GNNs bao gồm các bước chính: **truyền thông điệp**, **thu thập**, và **cập nhật**.

a. Truyền Thông Điệp (Message Passing)

Mỗi node gửi thông điệp đến các node lân cận của nó. Thông điệp thường là biểu diễn hiện tại của node hoặc một biến đổi của nó.

b. Thu Thập (Aggregation)

Node mục tiêu thu thập các thông điệp từ các node lân cận. Quá trình thu thập có thể sử dụng các phép toán như tổng, trung bình, hoặc sử dụng cơ chế attention.

c. Cập Nhật (Update)

Node mục tiêu cập nhật biểu diễn của mình dựa trên thông tin thu thập được từ các node lân cận. Thường thì một hàm tuyến tính kết hợp với một hàm kích hoạt phi tuyến (như ReLU) được sử dụng ở bước này.

1.4.Toán Học Cơ Bản Của Graph Neural Networks (GNNs) trong đồ thị đồng nhất.

- Cấu Trúc Đồ Thị: Ma trận kề A và ma trận độ D là hai đại diện phổ biến cho cấu trúc đồ thị:

- **Ma Trận Kề A:**

$$\mathbf{A} \in R^{N \times N}, \quad \mathbf{A}_{ij} = \begin{cases} 1 & \text{nếu có cạnh nối từ } i \text{ đến } j, \\ 0 & \text{ngược lại.} \end{cases}$$

- **Ma Trận Độ D:**

$$\mathbf{D}_{ii} = \sum_{j=1}^{N} \mathbf{A}_{ij}$$

D là ma trận đường chéo với các phần tử Dii bằng độ của node ii (số lượng cạnh nối đến node ii).

- Chuẩn Hóa Ma Trận Kề:

Để ổn định quá trình truyền thông điệp và đảm bảo cân bằng thông tin từ các node lân cận, ma trận kề thường được chuẩn hóa. Phương pháp chuẩn hóa phổ biến nhất là chuẩn hóa đối xứng theo ma trận độ đã được thêm self-loop.

- **Thêm Self-Loop:**

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$$

  Trong đó **I** là ma trận đơn vị, đảm bảo mỗi node cũng nhận thông tin từ chính nó.

- **Chuẩn Hóa Đối Xứng (Đối với GCNConv):**

$$\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}$$

Trong đó **D~** là ma trận độ của **A~**.

Những lý do cần có quá trình chuẩn hóa đối xứng:

Ổn Định quá trình truyền thông điệp: Ngăn chặn việc các biểu diễn đặc trưng của các node bị phóng đại hoặc co lại quá mức khi truyền qua các lớp convolution.

Đảm Bảo Cân Bằng Thông Tin: Đảm bảo rằng mỗi node nhận được lượng thông tin từ các node lân cận một cách cân bằng, không phụ thuộc vào số lượng lân cận của nó.

Cải Thiện Hiệu Suất Học: Giúp mô hình học được các biểu diễn đặc trưng tốt hơn và giảm thiểu vấn đề gradient biến thiên (vanishing/exploding gradients).

-Công Thức Truyền Thông Điệp trong GCN:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right)$$

Trong đó:

- $\mathbf{H}^{(l)} \in \mathbb{R}^{N \times F^{(l)}}$: Biểu diễn các node tại lớp l. $\mathbf{H}^{(0)}$ là ma trận đặc trưng đầu vào $\mathbf{X}$.
- $\hat{\mathbf{A}} \in \mathbb{R}^{N \times N}$: Ma trận kề đã được chuẩn hóa.
- $\mathbf{W}^{(l)} \in \mathbb{R}^{F^{(l)} \times F^{(l+1)}}$: Ma trận trọng số của lớp l.
- σ: Hàm kích hoạt phi tuyến, thường là ReLU.

1.5. Toán Học Cơ Bản Của Graph Neural Networks (GNNs) trong đồ thị không đồng nhất.

-Biểu diễn đồ thị không đồng nhất (Heterogeneous Graph Representation):

Ma trận kề và tính chất đồ thị

- Đồ thị không đồng nhất chứa hai loại nút chính: drug (thuốc) và gene (gene), cùng với hai loại quan hệ:
  - (drug, interacts, gene): Quan hệ từ drug đến gene.
  - (gene, interacted_by, drug): Quan hệ từ gene đến drug.

-Tạo đặc trưng nút (Node Features):

Mỗi nút trong đồ thị được gán một vector đặc trưng ngẫu nhiên ban đầu:

$$\mathbf{X}_{\text{drug}} \in \mathbb{R}^{N_{\text{drug}} \times d}, \quad \mathbf{X}_{\text{gene}} \in \mathbb{R}^{N_{\text{gene}} \times d}$$

- N_{drug}: Số lượng nút *drug*.
- N_{gene}: Số lượng nút *gene*.
- d: Kích thước không gian đặc trưng ban đầu (ở đây là 64).

-Truyền thông điệp (Message Passing)

HeteroConv:

- **HeteroConv** là một lớp chuyên xử lý đồ thị không đồng nhất, kết hợp các lớp GNN cho từng loại quan hệ trong đồ thị. Ở đây, ta sử dụng **SAGEConv** làm thành phần cho từng quan hệ.

Công thức tổng quát của SAGEConv (cơ chế tổng hợp):

$$h_i^{(l+1)} = \sigma\left(\mathbf{W}_1 h_i^{(l)} + \mathbf{W}_2 \cdot \text{AGGREGATE}\left(\{h_j^{(l)} \mid j \in \mathcal{N}(i)\}\right)\right)$$

- $h_i^{(l)}$: Vector đặc trưng của nút i tại lớp l.
- $\mathcal{N}(i)$: Tập lân cận của nút i.
- **AGGREGATE**: Hàm tổng hợp, thường là *mean aggregation* trong SAGEConv.
- σ: Hàm kích hoạt (ReLU).
- $\mathbf{W}_1$ và $\mathbf{W}_2$: Ma trận trọng số học được.

Công thức truyền thông điệp trong HeteroConv:

HeteroConv tổng hợp các đặc trưng từ nhiều loại quan hệ. Với mỗi loại quan hệ (ví dụ: drug -> gene và gene -> drug), ta áp dụng SAGEConv riêng biệt, sau đó kết hợp chúng:

$$h_{\text{drug}}^{(l+1)} = \text{COMBINE}\left(\{\text{SAGEConv}_{(\text{drug, interacts, gene})}, \text{SAGEConv}_{(\text{gene, interacted\_by, drug})}\}\right)$$

- **COMBINE**: Phương pháp kết hợp đặc trưng từ các lớp con (thường là cộng hoặc nối).

1.6. Đánh giá

Ở đây ta sử dụng 2 thang đo với 2 loại bài toán: với bài toán node classification, ta sử dụng thang đo là Accuracy, còn với bài toán link prediction, ta sử dụng thang đo là AUC.

$$\text{Accuracy} = \frac{\text{Số dự đoán đúng}}{\text{Tổng số dự đoán}}$$

Dự đoán liên kết:

Tính AUC (Area Under Curve) từ đầu ra của bộ dự đoán liên kết và nhãn thật**.**

$$\text{AUC} = \frac{\sum_{i=1}^{N_{\text{positive}}} (\text{Rank}_i - i)}{N_{\text{positive}} \cdot N_{\text{negative}}}$$

- $\textbf{Rank}_i$: Xếp hạng của điểm dự đoán i trong danh sách các điểm dự đoán (sắp xếp từ lớn đến nhỏ).
- $N_{\text{positive}}, N_{\text{negative}}$: Số lượng mẫu dương và âm.

Chúng ta triển khai hai mô hình GNN khác nhau để giải quyết vấn đề phân loại node và dự đoán liên kết:

- **Mô Hình GNN Đồng Nhất (Homogeneous GNN)**:
  - Sử dụng hai lớp GCNConv để học các biểu diễn node từ cấu trúc đồ thị.
  - Sử dụng một lớp tuyến tính để phân loại các biểu diễn node thành các nhãn nhị phân.
  - Sử dụng một mô hình hồi quy logistic đơn giản để dự đoán xác suất tồn tại liên kết giữa cặp node dựa trên các biểu diễn đã học.
- **Mô Hình GNN Hỗn Hợp (Heterogeneous GNN)**:
  - Sử dụng lớp HeteroConv kết hợp các lớp SAGEConv để xử lý các loại cạnh khác nhau trong đồ thị.
  - Sử dụng một lớp tuyến tính để phân loại các biểu diễn node thành các nhãn nhị phân.
  - Sử dụng một mạng nơ-ron hồi quy với nhiều lớp để dự đoán xác suất liên kết giữa các cặp node dựa trên các biểu diễn đã học.

2. Công Nghệ và Thư Viện

- **Python**: Ngôn ngữ lập trình chính.
- **PyTorch Geometric**: Thư viện mở rộng của PyTorch dành cho các mô hình GNN.
- **NetworkX**: Thư viện để xây dựng và thao tác với đồ thị.
- **Scikit-learn**: Sử dụng cho các bước giảm chiều như PCA và t-SNE.
- **Py4Cytoscape**: Giao tiếp với Cytoscape để trực quan hóa mạng lưới.
- **Cytoscape**: Nền tảng trực quan hóa mạng lưới sinh học.
- **Matplotlib và Seaborn**: Dùng để trực quan hóa dữ liệu.

V. Triển Khai Chương Trình

1. Tải Dữ Liệu và Xử Lý Trước

- **Dữ liệu**: Sử dụng tập dữ liệu ChG-Miner_miner-chem-chem.tsv chứa các cặp thuốc-thuốc cho đồ thị đồng nhất, ChG-Miner_miner-chem-gene.tsv chứa các cặp thuốc-gen cho đồ thị hỗn hợp.
- **Xử lý**: Loại bỏ các dòng không cần thiết, tạo danh sách các node duy nhất và xây dựng đồ thị đồng nhất hoặc hỗn hợp bằng NetworkX và PyTorch Geometric.

2. Xây Dựng Mô Hình GNN

2.1 Mô Hình GNN Đồng Nhất

- **Định Nghĩa Mô Hình**: Tạo lớp HomogeneousGNN với hai lớp GCNConv và một lớp phân loại node.
- **Huấn Luyện**: Sử dụng tối ưu hóa Adam và hàm mất mát CrossEntropy để huấn luyện mô hình trên tập dữ liệu phân chia train/validation/test.
- **Dự Đoán Liên Kết**: Sử dụng một mô hình hồi quy tuyến tính để dự đoán xác suất liên kết giữa các node dựa trên biểu diễn đã học.

2.2 Mô Hình GNN Hỗn Hợp

- **Định Nghĩa Mô Hình**: Tạo lớp HeteroGNN sử dụng HeteroConv với các lớp SAGEConv cho các loại cạnh khác nhau.

- **Huấn Luyện**: Sử dụng tối ưu hóa Adam và kết hợp hàm mất mát CrossEntropy cho phân loại node và Binary Cross Entropy cho dự đoán liên kết.
- **Dự Đoán Liên Kết**: Sử dụng một mạng nơ-ron hồi quy với nhiều lớp để dự đoán xác suất liên kết giữa các cặp node dựa trên các biểu diễn đã học.

3. Đánh Giá và Trực Quan Hóa

- **Đánh Giá**: Tính toán độ chính xác cho phân loại node và AUC cho dự đoán liên kết trên các tập train, validation và test.
- **Trực Quan Hóa**: Sử dụng PCA và t-SNE để giảm chiều các biểu diễn node và tải vào Cytoscape để trực quan hóa mạng lưới.

4. Dự Đoán Liên Kết

- **Chia Tập Dữ Liệu**: Tách các cạnh thành tập train, validation và test cho nhiệm vụ dự đoán liên kết.
- **Tạo Cạnh Âm**: Tạo các cặp node không kết nối hiện tại để làm các cạnh âm.
- **Huấn Luyện và Đánh Giá**: Huấn luyện mô hình dự đoán trên tập train và đánh giá trên tập validation và test.

5. Trực Quan Hóa Với Cytoscape

- **Tạo Mạng Lưới**: Kết hợp các cạnh hiện có và các cạnh dự đoán vào NetworkX, sau đó tải lên Cytoscape.
- **Ánh Xạ Thuộc Tính**: Ánh xạ màu sắc và kích thước node dựa trên các thuộc tính như nhãn và các thành phần PCA, t-SNE.
- **Ánh Xạ Cạnh**: Ánh xạ màu sắc và kiểu đường kẻ cho các cạnh dựa trên trạng thái dự đoán (existing vs predicted).

6. Mã nguồn Jupyter (Python3)

6.1 Đồ thị đồng nhất

```
# %%

# %%capture
```

```
# !pip install torch-geometric

# !pip install py4cytoscape seaborn

# !pip install scikit-learn #dim reduction



# %%

#_PY4CYTOSCAPE =
'git+https://github.com/cytoscape/py4cytoscape@1.7.0' # optional

import requests

import py4cytoscape as p4c

exec(requests.get("https://raw.githubusercontent.com/cytoscape/jupyte
r-bridge/master/client/p4c_init.py").text)

IPython.display.Javascript(_PY4CYTOSCAPE_BROWSER_CLIENT_JS) # Start
browser client



# %%

# Get Cytoscape version information

print(p4c.cytoscape_version_info())



# %%

import pandas as pd

import torch

from torch_geometric.data import Data

from torch_geometric.nn import GCNConv

import torch.nn.functional as F

from torch.optim import Adam

from sklearn.metrics import roc_auc_score

from sklearn.model_selection import train_test_split
```

```
from sklearn.decomposition import PCA

from sklearn.manifold import TSNE

from sklearn.preprocessing import MinMaxScaler

import py4cytoscape as p4c

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from IPython.display import Image as IPyImage

import networkx as nx

import time

from torch.nn import CrossEntropyLoss



# %%

# 1. Data Loading và Preprocessing

data_path = r"C:\Users\ADMIN\Downloads\ChCh-Miner_durgbank-chem-chem
(1).tsv"

df = pd.read_csv(data_path, sep='\t', names=["source", "target"],
header=None)

# Display the DataFrame to verify

print("DataFrame head:")

print(df.head())



# %%

# Read the TSV file with no header and assign column names 'source'
and 'target'

df = pd.read_csv(data_path, sep='\t', names=["source", "target"],
header=None)
```

```
# Display the DataFrame to verify
print("DataFrame head:")
print(df.head())

# Tạo các danh sách các node duy nhất (source và target)
unique_sources = df['source'].unique()
unique_targets = df['target'].unique()
unique_nodes = np.unique(np.concatenate((unique_sources,
unique_targets)))

# Tạo các mappings giữa node và chỉ số
node_to_idx = {node: idx for idx, node in enumerate(unique_nodes)}
idx_to_node = {idx: node for node, idx in node_to_idx.items()}

# Tổng số node
num_nodes = len(unique_nodes)
print(f'Total nodes: {num_nodes}')

# Tạo đồ thị đồng nhất bằng NetworkX
G = nx.Graph()

# Thêm các node
for node in unique_nodes:
    G.add_node(node)
```

```
# Thêm các edge
edge_list = df[['source', 'target']].values.tolist()
G.add_edges_from(edge_list)

# Kiểm tra số lượng node và edge
print(f'Number of nodes: {G.number_of_nodes()}')
print(f'Number of edges: {G.number_of_edges()}')

# Tạo danh sách các edge dưới dạng chỉ số
edge_index = torch.tensor([
    [node_to_idx[source], node_to_idx[target]] for source, target in df.values
], dtype=torch.long).t().contiguous()

# Tạo node features (giả sử không có thuộc tính, sử dụng embedding ngẫu nhiên)
hidden_dim = 64
x = torch.randn(num_nodes, hidden_dim)  # Sử dụng embedding ngẫu nhiên

# Tạo nhãn cho node (có thể sử dụng nhãn ngẫu nhiên hoặc dựa trên loại node)
# Ví dụ với nhãn ngẫu nhiên:
y = torch.randint(0, 2, (num_nodes,), dtype=torch.long)

# Nếu muốn phân loại dựa trên loại node (source vs target), sử dụng đoạn mã sau:
# labels = []
```

```
# for node in unique_nodes:
#     if node in unique_sources:
#         labels.append(0)  # Label 0 cho source nodes
#     else:
#         labels.append(1)  # Label 1 cho target nodes
# y = torch.tensor(labels, dtype=torch.long)


# Tạo đối tượng Data
data = Data(x=x, edge_index=edge_index, y=y)


# Chia dữ liệu thành train/val/test cho node classification
train_idx, temp_idx = train_test_split(range(num_nodes), train_size=0.7, random_state=42)
val_idx, test_idx = train_test_split(temp_idx, test_size=0.5, random_state=42)


# Tạo masks
train_mask = torch.zeros(num_nodes, dtype=torch.bool)
val_mask = torch.zeros(num_nodes, dtype=torch.bool)
test_mask = torch.zeros(num_nodes, dtype=torch.bool)


train_mask[train_idx] = True
val_mask[val_idx] = True
test_mask[test_idx] = True


# Thêm masks vào đối tượng data
```

```
data.train_mask = train_mask

data.val_mask = val_mask

data.test_mask = test_mask



# Model Definition

class HomogeneousGNN(torch.nn.Module):

    def __init__(self, in_channels, hidden_channels, out_channels):

        super(HomogeneousGNN, self).__init__()

        self.conv1 = GCNConv(in_channels, hidden_channels)

        self.conv2 = GCNConv(hidden_channels, out_channels)

        self.node_classifier = torch.nn.Linear(out_channels, 2)  #
Đối với phân loại nhãn nhị phân



    def forward(self, data):

        x, edge_index = data.x, data.edge_index #Shape: (N,
in_channels)

        # GNN Layers

        x = self.conv1(x, edge_index)

        x = F.relu(x) #Shape: (N, hidden_channels)

        x = self.conv2(x, edge_index) #Shape: (N, out_channels)

        # Node Classification

        out = self.node_classifier(x) #Shape: (N, 2)

        return out, x  # Trả về cả dự đoán và embedding vectors



def train(model, data, optimizer, criterion):

    model.train()
```

```
    optimizer.zero_grad()
    out, _ = model(data)
    loss = criterion(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    return loss.item()


@torch.no_grad()
def evaluate(model, data, mask, criterion=None):
    model.eval()
    out, embeddings = model(data)
    if criterion:
        loss = criterion(out[mask], data.y[mask]).item()
    pred = out[mask].argmax(dim=1)
    correct = (pred == data.y[mask]).sum().item()
    acc = correct / mask.sum().item()
    return acc


print("Setting up training...")
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = HomogeneousGNN(in_channels=hidden_dim, hidden_channels=128,
out_channels=64).to(device)
data = data.to(device)
optimizer = Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()
```

```python
print("Starting training...")

best_val_acc = 0

best_epoch = 0


for epoch in range(1, 501):

    loss = train(model, data, optimizer, criterion)

    if epoch % 20 == 0:

        train_acc = evaluate(model, data, data.train_mask)

        val_acc = evaluate(model, data, data.val_mask)

        print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Train Acc:
{train_acc:.4f}, Val Acc: {val_acc:.4f}')

        if val_acc > best_val_acc:

            best_val_acc = val_acc

            best_epoch = epoch

            torch.save(model.state_dict(), 'best_model.pt')


# Đánh giá trên bộ test

print("\nEvaluating final model...")

model.load_state_dict(torch.load('best_model.pt'))

test_acc = evaluate(model, data, data.test_mask)

print(f'\nTest Accuracy: {test_acc:.4f}')


print("\nGenerating embeddings...")

model.eval()

with torch.no_grad():

    _, embeddings = model(data)
```

```
    embeddings = embeddings.cpu().numpy()


# Áp dụng PCA -> giảm chi ề u
print("\nApplying PCA for dimensionality reduction...")
pca = PCA(n_components=2, random_state=42)
embeddings_pca = pca.fit_transform(embeddings)


# Áp dụng t-SNE
print("Applying t-SNE for dimensionality reduction...")
tsne = TSNE(n_components=2, random_state=42, perplexity=30,
n_iter=1000)
embeddings_tsne = tsne.fit_transform(embeddings)


# Chuẩn hóa các thành phần t-SNE để trực quan hóa Cytoscape
scaler = MinMaxScaler()
embeddings_tsne_norm = scaler.fit_transform(embeddings_tsne)


# Tính độ lớn của các thành phần t-SNE
tsne_magnitude = np.linalg.norm(embeddings_tsne_norm, axis=1)


# Tạo DataFrame cho embeddings
embeddings_df = pd.DataFrame(embeddings, columns=[f'embedding_{i}'
for i in range(embeddings.shape[1])])
embeddings_df['name'] = [idx_to_node[idx] for idx in
range(num_nodes)]
embeddings_df['pca_0'] = embeddings_pca[:, 0]
embeddings_df['pca_1'] = embeddings_pca[:, 1]
```

```
embeddings_df['tsne_0'] = embeddings_tsne[:, 0]

embeddings_df['tsne_1'] = embeddings_tsne[:, 1]

embeddings_df['tsne_magnitude'] = tsne_magnitude

embeddings_df['label'] = y.cpu().numpy()



# 17. Chuyển Đổi edge_list_tuples Thành Chỉ Số Node

print("\nConverting edge_list_tuples to node indices...")

edge_list_tuples = [(node_to_idx[u], node_to_idx[v]) for u, v in
edge_list]



# 18. Chia Cạnh Thành Train/Validation/Test Cho Link Prediction

print("\nSplitting edges into train/validation/test sets for Link
Prediction...")

train_edges, test_edges = train_test_split(edge_list_tuples,
test_size=0.2, random_state=42)

val_edges, test_edges = train_test_split(test_edges, test_size=0.5,
random_state=42)



# 19. Tạo Negative Edges Cho Tập Train

print("\nCreating negative edges for training...")

num_negative_samples = len(train_edges)

negative_edges = []

while len(negative_edges) < num_negative_samples:

    u, v = np.random.choice(num_nodes, size=2, replace=False)

    if (u, v) not in G.edges() and (v, u) not in G.edges():  # Đảm
bảo cạnh âm không tồn tại

        negative_edges.append((u, v))
```

```
import torch.nn as nn

# 20. Định Nghĩa Link Predictor Model

print("\nDefining the Link Predictor model...")

class LinkPredictor(nn.Module):

    def __init__(self, embedding_dim):

        super(LinkPredictor, self).__init__()

        self.linear = nn.Linear(embedding_dim * 2, 1)  # Cho phép nối
embeddings



    def forward(self, node1_embedding, node2_embedding):

        combined_embedding = torch.cat([node1_embedding,
node2_embedding], dim=1)

        output = torch.sigmoid(self.linear(combined_embedding))  # Dự
đoán xác suất liên kết

        return output


# 21. Tạo Link Predictor và Định Nghĩa Optimizer & Criterion

print("\nSetting up Link Predictor training...")

embedding_dim = embeddings.shape[1]

link_predictor = LinkPredictor(embedding_dim).to(device)

optimizer_lp = torch.optim.Adam(link_predictor.parameters(), lr=0.01)

criterion_lp = nn.BCELoss()


# 22. Định Nghĩa Hàm Để Lấy Embeddings Cho Các Cạnh

def get_edge_embeddings(edges, embeddings):

    # Tách các chỉ số nguồn và đích từ danh sách cạnh

    src = [edge[0] for edge in edges]
```

```
    dst = [edge[1] for edge in edges]

    src = np.array(src)

    dst = np.array(dst)

    src_embeddings = embeddings[src]

    dst_embeddings = embeddings[dst]

    return src_embeddings, dst_embeddings


# 23. Định Nghĩa Hàm Để Tạo Labels Cho Link Prediction

def get_link_labels(positive_edges, negative_edges):

    # Tạo labels: 1 cho positive edges, 0 cho negative edges

    labels = torch.cat([torch.ones(len(positive_edges)),
torch.zeros(len(negative_edges))], dim=0).to(device)

    return labels


# 24. Định Nghĩa Hàm Để Đánh Giá Link Prediction

@torch.no_grad()

def evaluate_link_prediction(model, link_predictor, edges,
embeddings, criterion):

    model.eval()  # Đảm bảo mô hình GNN không được cập nhật trong quá
trình đánh giá

    link_predictor.eval()

    src_embeddings, dst_embeddings = get_edge_embeddings(edges,
embeddings)

    src_embeddings = torch.tensor(src_embeddings,
dtype=torch.float32).to(device)

    dst_embeddings = torch.tensor(dst_embeddings,
dtype=torch.float32).to(device)
```

```
    predictions = link_predictor(src_embeddings,
dst_embeddings).squeeze()

    loss = criterion(predictions,
torch.ones_like(predictions)).item()

    return loss


# 25. Định Nghĩa Hàm Huấn Luyện Link Predictor

def train_link_prediction(model, link_predictor, optimizer,
criterion, train_edges, negative_edges, embeddings):

    model.eval()  # Đảm bảo mô hình GNN không được cập nhật trong quá
trình huấn luyện link predictor

    link_predictor.train()

    optimizer.zero_grad()



    # Lấy embeddings cho các cạnh dương (positive edges)

    train_src_embeddings, train_dst_embeddings =
get_edge_embeddings(train_edges, embeddings)

    train_src_embeddings = torch.tensor(train_src_embeddings,
dtype=torch.float32).to(device)

    train_dst_embeddings = torch.tensor(train_dst_embeddings,
dtype=torch.float32).to(device)



    # Lấy embeddings cho các cạnh âm (negative edges)

    neg_src_embeddings, neg_dst_embeddings =
get_edge_embeddings(negative_edges, embeddings)

    neg_src_embeddings = torch.tensor(neg_src_embeddings,
dtype=torch.float32).to(device)

    neg_dst_embeddings = torch.tensor(neg_dst_embeddings,
dtype=torch.float32).to(device)
```

```
    # Dự đoán liên kết
    pos_pred = link_predictor(train_src_embeddings,
train_dst_embeddings)
    neg_pred = link_predictor(neg_src_embeddings, neg_dst_embeddings)

    # Kết hợp predictions và labels
    predictions = torch.cat([pos_pred, neg_pred], dim=0).squeeze()
    labels = get_link_labels(train_edges, negative_edges)

    # Tính toán loss
    loss = criterion(predictions, labels)
    loss.backward()
    optimizer.step()

    return loss.item()

# 26. Huấn Luyện Link Predictor
print("\nStarting Link Prediction training...")
epochs_lp = 100
best_val_loss_lp = float('inf')

for epoch in range(1, epochs_lp + 1):
    loss = train_link_prediction(model, link_predictor, optimizer_lp,
criterion_lp, train_edges, negative_edges, embeddings)
    val_loss = evaluate_link_prediction(model, link_predictor,
val_edges, embeddings, criterion_lp)
```

```
    print(f'Epoch [{epoch}/{epochs_lp}], Loss: {loss:.4f}, Val Loss:
{val_loss:.4f}')


    if val_loss < best_val_loss_lp:

        best_val_loss_lp = val_loss

        best_epoch_lp = epoch

        torch.save(link_predictor.state_dict(),
'best_link_prediction_model.pth')


# 27. Đánh Giá Link Predictor Trên Tập Test

print("\nEvaluating Link Predictor on test set...")

link_predictor.load_state_dict(torch.load('best_link_prediction_model
.pth'))

test_loss_lp = evaluate_link_prediction(model, link_predictor,
test_edges, embeddings, criterion_lp)

print(f'Test Loss: {test_loss_lp:.4f}')


# 28. Thêm Thuộc Tính Link Prediction Vào Edge Table

print("\nAdding Link Prediction results to edge table...")


# Tạo DataFrame cho các cạnh gốc và cạnh dự đoán

existing_edge_df = pd.DataFrame(edge_list_tuples, columns=['source',
'target'])

existing_edge_df['prediction'] = 'existing'


# --- Điều Chỉnh Để Giới Hạn Số Lượng Cạnh Dự Đoán ---

# Thay vì dự đoán tất cả các cạnh không tồn tại, chúng ta đã chọn một
tập hợp ngẫu nhiên và giới hạn số lượng cạnh cần dự đoán (100,000)
```

```
# Tuy nhiên, chúng ta sẽ chỉ chọn top 100 cạnh dự đoán để tránh sử
dụng quá nhiều RAM và tạo thêm các cạnh mới trong Cytoscape.


print("\nCollecting Link Prediction results with limited candidate
edges to avoid RAM crash...")

# Giới hạn số lượng cạnh cần dự đoán (ví dụ: 100,000)

max_candidate_edges = 100000


# Tạo danh sách các cạnh không tồn tại

# Chọn ngẫu nhiên một tập hợp các cạnh không tồn tại

candidate_edges = set()

while len(candidate_edges) < max_candidate_edges:

    u = np.random.randint(0, num_nodes)

    v = np.random.randint(0, num_nodes)

    if u != v and (u, v) not in G.edges() and (v, u) not in
G.edges():

        candidate_edges.add((u, v))

    if len(candidate_edges) % 10000 == 0 and len(candidate_edges) >
0:

        print(f"{len(candidate_edges)} candidate edges collected...")


candidate_edges = list(candidate_edges)

print(f"Total candidate edges for prediction:
{len(candidate_edges)}")


# Lấy embeddings cho các cạnh dự đoán

src_embeddings, dst_embeddings = get_edge_embeddings(candidate_edges,
embeddings)
```

```
src_embeddings = torch.tensor(src_embeddings,
dtype=torch.float32).to(device)

dst_embeddings = torch.tensor(dst_embeddings,
dtype=torch.float32).to(device)



# Dự đoán xác suất liên kết

print("\nPredicting link probabilities...")

with torch.no_grad():

    predictions = link_predictor(src_embeddings,
dst_embeddings).squeeze().cpu().numpy()



# Chọn các cạnh có xác suất dự đoán cao nhất (ví dụ: top 100)

top_k = 100

top_k_indices = np.argsort(predictions)[-top_k:]

predicted_edges = [candidate_edges[i] for i in top_k_indices]

predicted_scores = predictions[top_k_indices]



print(f"Top {top_k} predicted edges selected.")



# Tạo DataFrame cho các cạnh dự đoán

predicted_edge_df = pd.DataFrame(predicted_edges, columns=['source',
'target'])

predicted_edge_df['prediction'] = 'predicted'

predicted_edge_df['score'] = predicted_scores



# Kết hợp DataFrame cho các cạnh gốc và cạnh dự đoán
```

```
combined_edge_df = pd.concat([existing_edge_df,
predicted_edge_df[['source', 'target', 'prediction', 'score']]],
ignore_index=True)


# 29. Tạo Mạng Lưới Trong Cytoscape với Cả Các Cạnh Dự Đoán

print("\nCreating network in Cytoscape using NetworkX with predicted
edges...")

# Tạo đồ thị NetworkX với các node và edge

G_cytoscape = nx.Graph()


# Thêm các node với thuộc tính 'name'

for node in unique_nodes:

    G_cytoscape.add_node(node, name=node)


# Thêm các cạnh hiện có với thuộc tính 'prediction' = 'existing'

for u, v in edge_list:

    G_cytoscape.add_edge(u, v, prediction='existing')


# Thêm các cạnh dự đoán với thuộc tính 'prediction' = 'predicted' và
'score'

for idx, row in predicted_edge_df.iterrows():

    u_idx, v_idx = row['source'], row['target']

    u_node, v_node = idx_to_node[u_idx], idx_to_node[v_idx]

    score = row['score']

    G_cytoscape.add_edge(u_node, v_node, prediction='predicted',
score=score)


# Tạo mạng lưới trong Cytoscape từ NetworkX
```

```
network_title = 'Drugs Interaction Network'

network_suid = p4c.create_network_from_networkx(G_cytoscape,
title=network_title)

print(f"Network created with SUID: {network_suid}")



# Chờ mạng lưới được tạo thành công

print("Waiting for the network to be created in Cytoscape...")

time.sleep(5)  # Chờ 5 giây để đảm bảo mạng lưới đã được tải



# 30. Tải Các Thuộc Tính Embedding vào Cytoscape

print("\nLoading PCA and t-SNE data into Cytoscape...")

# Tải thuộc tính PCA

p4c.load_table_data(

    data=embeddings_df[['name', 'pca_0', 'pca_1']],

    data_key_column='name',

    table='node',

    table_key_column='name',

    network=network_title

)



# Tải thuộc tính t-SNE

p4c.load_table_data(

    data=embeddings_df[['name', 'tsne_0', 'tsne_1',
'tsne_magnitude']],

    data_key_column='name',

    table='node',
```

```
    table_key_column='name',
    network=network_title
)

# 31. Định Nghĩa và Áp Dụng Visual Style
print("\nDefining and applying visual styles...")
# Tạo một visual style mới
style_name = "GNN_Homogeneous_Style"
if style_name not in p4c.get_visual_style_names():
    p4c.styles.create_visual_style(style_name)
    print(f"Visual style '{style_name}' created.")
else:
    print(f"Visual style '{style_name}' already exists.")

# Áp dụng visual style
p4c.set_visual_style(style_name)
print(f"Visual style '{style_name}' applied.")

# Thêm nhãn vào DataFrame và tải vào Cytoscape
embeddings_df['label'] = y.cpu().numpy()

# Tải thuộc tính 'label' vào Cytoscape node table
print("\nLoading label data into Cytoscape node table...")
p4c.load_table_data(
    data=embeddings_df[['name', 'label']],
    data_key_column='name',
```

```
    table='node',
    table_key_column='name',
    network=network_title  # Sử dụng biến tên mạng lưới đã định nghĩa
)
print("'label' column loaded to Cytoscape node table.")

# Ánh xạ màu sắc cho node dựa trên 'label'
print("Mapping node color based on label...")
# Định nghĩa palette cho hai nhãn bằng cyPalette và chỉ lấy hai màu đầu tiên
palette_colors = p4c.cyPalette(name='set1')[:2]  # Label 0: màu đầu tiên, Label 1: màu thứ hai
palette = ('custom_palette', 'qualitative', lambda n: palette_colors)
print(f"Palette used for mapping: {palette_colors}")

# Tạo color map cho node dựa trên 'label'
label_color_map = p4c.gen_node_color_map(
    table_column='label',
    palette=palette,           # Sử dụng palette đã định nghĩa
    mapping_type='d',          # 'd' cho discrete mapping
    default_color='#CCCCCC',   # Màu xám nhạt mặc định cho các nhãn không xác định
    style_name=style_name,     # Tên style đã tạo trước đó
    network=network_title      # Đảm bảo tên mạng lưới chính xác
)

# Áp dụng color mapping
```

```
p4c.set_node_color_mapping(**label_color_map)

print("Node color mapping based on label applied.")



# Ánh xạ 'pca_0' vào kích thước node

print("Mapping PCA component to node size...")

size_map_pca = p4c.gen_node_size_map(

    table_column='pca_0',

    mapping_type='c',           # 'c' cho continuous mapping

    style_name=style_name,

    network=network_title       # Sử dụng biến tên mạng lưới đã định
nghĩa

)



p4c.set_node_size_mapping(**size_map_pca)

print("Node size mapping based on 'pca_0' applied.")



# Ánh xạ 'name' vào nhãn node

print("Mapping node labels...")

p4c.style_mappings.set_node_label_mapping('name',
style_name=style_name)



# Áp dụng visual style

p4c.set_visual_style(style_name)



# 32. Định Nghĩa và Áp Dụng Edge Style Mapping Generators

print("\nDefining and applying edge style mappings for Link
Prediction...")
```

```
# Ánh xạ màu sắc cho cạnh dựa trên thuộc tính 'prediction' sử dụng một palette có sẵn
print("Mapping edge color based on prediction status using a predefined palette...")
color_map = p4c.gen_edge_color_map(
    table_column='prediction',
    palette=p4c.palette_color_brewer_q_Set1(),  # Sử dụng hàm palette có sẵn
    mapping_type='d',                           # 'd' cho discrete mapping
    default_color='#000000',                    # Màu đen mặc định
    style_name=style_name,
    network=network_suid
)
p4c.set_edge_color_mapping(**color_map)

# Ánh xạ kiểu đường kẻ cho cạnh dựa trên thuộc tính 'prediction'
print("Mapping edge line style based on prediction status...")
line_style_map = p4c.gen_edge_line_style_map(
    table_column='prediction',
    default_line_style='SOLID',                 # Mặc định là SOLID
    style_name=style_name,
    network=network_suid
)
p4c.set_edge_line_style_mapping(**line_style_map)
```

```
# Áp dụng visual style
p4c.set_visual_style(style_name)


# Áp dụng layout Force-Directed để sắp xếp mạng lưới
print("Applying Force-Directed layout...")
p4c.layout_network('force-directed', network=network_suid)
```

6.2 Đồ thị hỗn hợp

```
# %%
# Cài đặt các thư viện cần thiết (bỏ qua nếu đã cài đặt)
# !pip install torch-geometric
# !pip install py4cytoscape seaborn
# !pip install scikit-learn
import requests
import py4cytoscape as p4c
import pandas as pd
import torch
from torch_geometric.data import HeteroData
from torch_geometric.nn import HeteroConv, SAGEConv
import torch.nn.functional as F
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from torch.optim import Adam
```

```
import torch.nn as nn

from sklearn.decomposition import PCA

from sklearn.manifold import TSNE

from sklearn.preprocessing import MinMaxScaler

from IPython.display import Javascript, Image as IPyImage, display

import networkx as nx

import time




# %%

# Khởi tạo kết nối với Cytoscape

exec(requests.get("https://raw.githubusercontent.com/cytoscape/jupyter-bridge/master/client/p4c_init.py").text)

IPython.display.Javascript(_PY4CYTOSCAPE_BROWSER_CLIENT_JS)  # Bắt đầu client trình duyệt



# Kiểm tra phiên bản Cytoscape

print(p4c.cytoscape_version_info())



# %%

# Đường dẫn tới dữ liệu (thay đổi nếu cần)

data_path = r"C:\Users\ADMIN\Downloads\ChG-Miner_miner-chem-gene.tsv"  # Đối với Windows

# data_path = '/content/ChG-Miner_miner-chem-gene.tsv'  # Đối với Colab hoặc Linux



# Tải dữ liệu
```

```
print("Loading data...")

df = pd.read_csv(data_path, sep='\t', names=["Drug", "Gene"])

df.drop(index=0, inplace=True)  # Xóa dòng tiêu đề nếu có

print(df.head())




# %%

# Định nghĩa thiết bị (device)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')



# Tạo mappings giữa node và chỉ số

print("Creating node mappings...")

unique_drugs = df['Drug'].unique()

unique_genes = df['Gene'].unique()

drug_to_idx = {drug: idx for idx, drug in enumerate(unique_drugs)}

gene_to_idx = {gene: idx for idx, gene in enumerate(unique_genes)}

idx_to_drug = {idx: drug for drug, idx in drug_to_idx.items()}

idx_to_gene = {idx: gene for gene, idx in gene_to_idx.items()}



num_drugs = len(unique_drugs)

num_genes = len(unique_genes)

hidden_dim = 64

print(f'Number of Drugs: {num_drugs}, Number of Genes: {num_genes}')

# Initialize HeteroData object

print("Initializing HeteroData...")

data = HeteroData()
```

```
# Thêm node features (sử dụng embedding ngẫu nhiên)
data['drug'].x = torch.randn(num_drugs, hidden_dim)
data['gene'].x = torch.randn(num_genes, hidden_dim)

# Tạo nhãn cho node (giả sử phân loại nhị phân cho drugs)
data['drug'].y = torch.randint(0, 2, (num_drugs,))
# Nếu bạn cần phân loại gene, có thể thêm:
# data['gene'].y = torch.randint(0, 2, (num_genes,))

# Chia dữ liệu thành train/val/test cho node classification
print("Splitting data into train/validation/test sets...")
train_drug_idx, temp_drug_idx = train_test_split(range(num_drugs),
train_size=0.7, random_state=42)
val_drug_idx, test_drug_idx = train_test_split(temp_drug_idx,
test_size=0.5, random_state=42)

# Tạo masks
print("Creating masks...")
train_mask = torch.zeros(num_drugs, dtype=torch.bool)
val_mask = torch.zeros(num_drugs, dtype=torch.bool)
test_mask = torch.zeros(num_drugs, dtype=torch.bool)

train_mask[train_drug_idx] = True
val_mask[val_drug_idx] = True
test_mask[test_drug_idx] = True
```

```
# Thêm masks vào đối tượng data
data['drug'].train_mask = train_mask
data['drug'].val_mask = val_mask
data['drug'].test_mask = test_mask
# Tạo danh sách các edge dương (existing edges)
print("Creating positive edge list...")
pos_edges = []
for _, row in df.iterrows():
    drug_idx = drug_to_idx[row['Drug']]
    gene_idx = gene_to_idx[row['Gene']]
    pos_edges.append([drug_idx, gene_idx])

# Chia các cạnh dương thành train/val/test
print("Splitting edges into train/validation/test sets...")
train_edges, test_edges = train_test_split(pos_edges, test_size=0.2,
random_state=42)
train_edges, val_edges = train_test_split(train_edges, test_size=0.2,
random_state=42)

# Chuyển đổi sang tensor edge_index
print("Converting edge lists to tensors...")
train_edge_index = torch.tensor(train_edges,
dtype=torch.long).t().contiguous()
val_pos_edge_index = torch.tensor(val_edges,
dtype=torch.long).t().contiguous()
test_pos_edge_index = torch.tensor(test_edges,
dtype=torch.long).t().contiguous()
```

```
# Thêm các cạnh vào đối tượng data
data['drug', 'interacts', 'gene'].edge_index = train_edge_index
data['gene', 'interacted_by', 'drug'].edge_index =
train_edge_index.flip([0])
# Định nghĩa mô hình HeteroGNN
class HeteroGNN(torch.nn.Module):
    def __init__(self, hidden_channels=64, out_channels=32):
        super().__init__()

        self.conv1 = HeteroConv({
            ('drug', 'interacts', 'gene'): SAGEConv((-1, -1),
hidden_channels),
            ('gene', 'interacted_by', 'drug'): SAGEConv((-1, -1),
hidden_channels)
        }, aggr='mean')

        self.conv2 = HeteroConv({
            ('drug', 'interacts', 'gene'): SAGEConv((-1, -1),
out_channels),
            ('gene', 'interacted_by', 'drug'): SAGEConv((-1, -1),
out_channels)
        }, aggr='mean')

        self.node_classifier = torch.nn.Linear(out_channels, 2)  #
Đối với phân loại nhãn nhị phân

        self.link_predictor = torch.nn.Sequential(
```

```
            torch.nn.Linear(2 * out_channels, hidden_channels),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_channels, 1),
            torch.nn.Sigmoid()
        )

    def encode(self, x_dict, edge_index_dict):
        x_dict = self.conv1(x_dict, edge_index_dict)
        x_dict = {key: F.relu(x) for key, x in x_dict.items()}
        x_dict = self.conv2(x_dict, edge_index_dict)
        return x_dict

    def decode(self, z_dict, edge_label_index):
        row, col = edge_label_index
        z_drug = z_dict['drug'][row]
        z_gene = z_dict['gene'][col]
        z = torch.cat([z_drug, z_gene], dim=-1)
        return self.link_predictor(z)

    def forward(self, x_dict, edge_index_dict,
edge_label_index=None):
        z_dict = self.encode(x_dict, edge_index_dict)
        drug_pred = self.node_classifier(z_dict['drug'])

        if edge_label_index is not None:
            link_pred = self.decode(z_dict, edge_label_index)
```

```
            return drug_pred, link_pred, z_dict


        return drug_pred, None, z_dict
# Hàm để lấy embeddings cho các cạnh
def get_edge_embeddings(edges, embeddings):
    src = [edge[0] for edge in edges]
    dst = [edge[1] for edge in edges]
    src = np.array(src)
    dst = np.array(dst)
    src_embeddings = embeddings[src]
    dst_embeddings = embeddings[dst]
    return src_embeddings, dst_embeddings


# Hàm để tạo labels cho link prediction
def get_link_labels(positive_edges, negative_edges):
    labels = torch.cat([torch.ones(len(positive_edges)),
torch.zeros(len(negative_edges))], dim=0).to(device)
    return labels


# Hàm để đánh giá link prediction
@torch.no_grad()
def evaluate_link_prediction(model, link_predictor, edges,
embeddings, criterion):
    model.eval()
    link_predictor.eval()
    src_embeddings, dst_embeddings = get_edge_embeddings(edges,
embeddings)
```

```
    src_embeddings = torch.tensor(src_embeddings,
dtype=torch.float32).to(device)

    dst_embeddings = torch.tensor(dst_embeddings,
dtype=torch.float32).to(device)

    predictions = link_predictor(src_embeddings,
dst_embeddings).squeeze()

    loss = criterion(predictions,
torch.ones_like(predictions)).item()

    return loss

# Hàm huấn luyện link predictor

def train_link_prediction(model, link_predictor, optimizer,
criterion, train_edges, negative_edges, embeddings):

    model.eval()  # Đảm bảo mô hình GNN không được cập nhật trong quá
trình huấn luyện link predictor

    link_predictor.train()

    optimizer.zero_grad()



    # Lấy embeddings cho các cạnh dương (positive edges)

    train_src_embeddings, train_dst_embeddings =
get_edge_embeddings(train_edges, embeddings)

    train_src_embeddings = torch.tensor(train_src_embeddings,
dtype=torch.float32).to(device)

    train_dst_embeddings = torch.tensor(train_dst_embeddings,
dtype=torch.float32).to(device)



    # Lấy embeddings cho các cạnh âm (negative edges)

    neg_src_embeddings, neg_dst_embeddings =
get_edge_embeddings(negative_edges, embeddings)

    neg_src_embeddings = torch.tensor(neg_src_embeddings,
dtype=torch.float32).to(device)
```

```
    neg_dst_embeddings = torch.tensor(neg_dst_embeddings,
dtype=torch.float32).to(device)


    # Dự đoán liên kết
    pos_pred = link_predictor(train_src_embeddings,
train_dst_embeddings)
    neg_pred = link_predictor(neg_src_embeddings, neg_dst_embeddings)


    # Kết hợp predictions và labels
    predictions = torch.cat([pos_pred, neg_pred], dim=0).squeeze()
    labels = get_link_labels(train_edges, negative_edges)


    # Tính toán loss
    loss = criterion(predictions, labels)
    loss.backward()
    optimizer.step()


    return loss.item()
# Hàm đánh giá node classification và link prediction
@torch.no_grad()
def evaluate(model, data, pos_edge_index, mask='val'):
    model.eval()


    # Node classification
    drug_pred, _, _ = model(data.x_dict, data.edge_index_dict)
    if mask == 'val':
```

```
        pred = drug_pred[data['drug'].val_mask].argmax(dim=1)
        correct = (pred ==
data['drug'].y[data['drug'].val_mask]).sum()
        nc_acc = int(correct) / int(data['drug'].val_mask.sum())
    else:
        pred = drug_pred[data['drug'].test_mask].argmax(dim=1)
        correct = (pred ==
data['drug'].y[data['drug'].test_mask]).sum()
        nc_acc = int(correct) / int(data['drug'].test_mask.sum())

    # Link prediction
    neg_edge_index = sample_neg_edges(pos_edges, num_drugs,
num_genes, pos_edge_index.size(1))
    edge_label_index = torch.cat([pos_edge_index, neg_edge_index],
dim=-1)
    edge_label = torch.cat([torch.ones(pos_edge_index.size(1)),
                            torch.zeros(neg_edge_index.size(1))],
dim=0)

    _, link_pred, _ = model(data.x_dict, data.edge_index_dict,
edge_label_index)
    lp_auc = roc_auc_score(edge_label.cpu(),
link_pred.squeeze().cpu())

    return nc_acc, lp_auc
# Hàm tạo negative edges cho link prediction
def sample_neg_edges(pos_edges, num_drugs, num_genes, num_samples):
    neg_edges = []
```

```
    pos_edges_set = set(map(tuple, pos_edges))


    while len(neg_edges) < num_samples:
        drug_idx = np.random.randint(0, num_drugs)
        gene_idx = np.random.randint(0, num_genes)
        if (drug_idx, gene_idx) not in pos_edges_set:
            neg_edges.append([drug_idx, gene_idx])


    return torch.tensor(neg_edges, dtype=torch.long).t()
# Định nghĩa Link Predictor Model
print("\nDefining the Link Predictor model...")
class LinkPredictor(nn.Module):
    def __init__(self, embedding_dim):
        super(LinkPredictor, self).__init__()
        self.linear = nn.Linear(embedding_dim * 2, 1)  # Cho phép nối
embeddings


    def forward(self, node1_embedding, node2_embedding):
        combined_embedding = torch.cat([node1_embedding,
node2_embedding], dim=1)
        output = torch.sigmoid(self.linear(combined_embedding))  # Dự
đoán xác suất liên kết
        return output


# Tạo Link Predictor và Định Nghĩa Optimizer & Criterion
print("\nSetting up Link Predictor training...")
embedding_dim = 64  # Sử dụng cùng chiều dimension với embedding
```

```
link_predictor = LinkPredictor(embedding_dim).to(device)

optimizer_lp = torch.optim.Adam(link_predictor.parameters(), lr=0.01)

criterion_lp = nn.BCELoss()

# Khởi tạo mô hình và chuyển dữ liệu lên device

print("\nSetting up training...")

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = HeteroGNN(hidden_channels=128, out_channels=64).to(device)

data = data.to(device)


optimizer = Adam(model.parameters(), lr=0.01)

criterion = torch.nn.CrossEntropyLoss()


# Huấn luyện mô hình GNN

print("Starting GNN training...")

best_val_acc = 0

best_epoch = 0

best_model_path = 'best_model.pth'


for epoch in range(1, 101):

    model.train()

    optimizer.zero_grad()

    drug_pred, link_pred, _ = model(data.x_dict,
data.edge_index_dict)

    nc_loss = criterion(drug_pred[data['drug'].train_mask],

                        data['drug'].y[data['drug'].train_mask])
```

```
    # Link prediction trên tập train
    neg_edge_index = sample_neg_edges(train_edges, num_drugs,
num_genes, len(train_edges))
    edge_label_index = torch.cat([train_edge_index, neg_edge_index],
dim=-1)
    edge_label = torch.cat([torch.ones(train_edge_index.size(1)),
                            torch.zeros(neg_edge_index.size(1))],
dim=0).to(device)

    _, link_pred, _ = model(data.x_dict, data.edge_index_dict,
edge_label_index)
    lp_loss = F.binary_cross_entropy(link_pred.squeeze(), edge_label)

    # Tổng hợp loss
    loss = nc_loss + lp_loss
    loss.backward()
    optimizer.step()

    # Đánh giá trên tập validation mỗi 10 epoch
    if epoch % 10 == 0:
        val_nc_acc, val_lp_auc = evaluate(model, data,
val_pos_edge_index, 'val')
        print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}')
        print(f'Val Node Class Acc: {val_nc_acc:.4f}, Val Link Pred
AUC: {val_lp_auc:.4f}')

        if val_nc_acc > best_val_acc:
            best_val_acc = val_nc_acc
```

```
            best_epoch = epoch
            torch.save(model.state_dict(), best_model_path)
            print(f'Best model saved at epoch {epoch}')
# Đánh giá trên tập test
print("\nEvaluating final model...")
model.load_state_dict(torch.load(best_model_path))
test_nc_acc, test_lp_auc = evaluate(model, data, test_pos_edge_index,
'test')
print(f'\nTest Results from epoch {best_epoch}:')
print(f'Node Classification Accuracy: {test_nc_acc:.4f}')
print(f'Link Prediction AUC: {test_lp_auc:.4f}')
# Sinh embeddings từ mô hình đã huấn luyện
print("\nGenerating embeddings...")
with torch.no_grad():
    _, _, final_embeddings = model(data.x_dict, data.edge_index_dict)


# Lấy embeddings cho drugs và genes
drug_embeddings = final_embeddings['drug'].cpu().numpy()
gene_embeddings = final_embeddings['gene'].cpu().numpy()


# Tạo DataFrames cho embeddings
print("Creating embeddings DataFrames...")
drug_embeddings_df = pd.DataFrame(drug_embeddings,
columns=[f'embedding_{i}' for i in range(drug_embeddings.shape[1])])
drug_embeddings_df['name'] = unique_drugs
```

```
gene_embeddings_df = pd.DataFrame(gene_embeddings,
columns=[f'embedding_{i}' for i in range(gene_embeddings.shape[1])])

gene_embeddings_df['name'] = unique_genes


# Áp dụng PCA

print("\nApplying PCA for dimensionality reduction...")

pca = PCA(n_components=2, random_state=42)

drug_embeddings_pca = pca.fit_transform(drug_embeddings)

gene_embeddings_pca = pca.fit_transform(gene_embeddings)


# Thêm các thành phần PCA vào DataFrames

drug_embeddings_df['pca_0'] = drug_embeddings_pca[:, 0]

drug_embeddings_df['pca_1'] = drug_embeddings_pca[:, 1]


gene_embeddings_df['pca_0'] = gene_embeddings_pca[:, 0]

gene_embeddings_df['pca_1'] = gene_embeddings_pca[:, 1]


# Áp dụng t-SNE

print("Applying t-SNE for dimensionality reduction...")

tsne = TSNE(n_components=2, random_state=42, perplexity=30,
n_iter=1000)

drug_embeddings_tsne = tsne.fit_transform(drug_embeddings)

gene_embeddings_tsne = tsne.fit_transform(gene_embeddings)


# Thêm các thành phần t-SNE vào DataFrames

drug_embeddings_df['tsne_0'] = drug_embeddings_tsne[:, 0]
```

```
drug_embeddings_df['tsne_1'] = drug_embeddings_tsne[:, 1]


gene_embeddings_df['tsne_0'] = gene_embeddings_tsne[:, 0]

gene_embeddings_df['tsne_1'] = gene_embeddings_tsne[:, 1]


# Chuẩn hóa các thành phần t-SNE để trực quan hóa trong Cytoscape

scaler = MinMaxScaler()

drug_embeddings_df[['tsne_0_norm', 'tsne_1_norm']] =
scaler.fit_transform(drug_embeddings_df[['tsne_0', 'tsne_1']])

gene_embeddings_df[['tsne_0_norm', 'tsne_1_norm']] =
scaler.fit_transform(gene_embeddings_df[['tsne_0', 'tsne_1']])


# Tính độ lớn của các thành phần t-SNE

drug_embeddings_df['tsne_magnitude'] =
np.linalg.norm(drug_embeddings_df[['tsne_0_norm',
'tsne_1_norm']].values, axis=1)

gene_embeddings_df['tsne_magnitude'] =
np.linalg.norm(gene_embeddings_df[['tsne_0_norm',
'tsne_1_norm']].values, axis=1)

# Thêm các cạnh dự đoán vào mạng lưới

print("\nAdding predicted edges to the network...")


# Giới hạn số lượng cạnh dự đoán để tránh sử dụng quá nhiều RAM (ví
dụ: top 100)

top_k = 100


# Tạo danh sách các cạnh không tồn tại

print("Collecting candidate edges for prediction...")
```

```
candidate_edges = set()

pos_edges_set = set(map(tuple, pos_edges))

while len(candidate_edges) < top_k:

    u = np.random.randint(0, num_drugs)

    v = np.random.randint(0, num_genes)

    if (u, v) not in pos_edges_set and (v, u) not in pos_edges_set:

        candidate_edges.add((u, v))



candidate_edges = list(candidate_edges)

print(f"Total candidate edges for prediction:
{len(candidate_edges)}")



# Lấy embeddings cho các cạnh dự đoán

src_embeddings, dst_embeddings = get_edge_embeddings(candidate_edges,
drug_embeddings)

src_embeddings = torch.tensor(src_embeddings,
dtype=torch.float32).to(device)

dst_embeddings = torch.tensor(dst_embeddings,
dtype=torch.float32).to(device)



# Dự đoán xác suất liên kết

print("\nPredicting link probabilities for candidate edges...")

with torch.no_grad():

    link_pred = link_predictor(src_embeddings,
dst_embeddings).squeeze().cpu().numpy()



# Chọn các cạnh có xác suất dự đoán cao nhất

top_k_indices = np.argsort(link_pred)[-top_k:]
```

```
predicted_edges = [candidate_edges[i] for i in top_k_indices]

predicted_scores = link_pred[top_k_indices]

print(f"Selected top {top_k} predicted edges.")


# Tạo DataFrame cho các cạnh dự đoán

predicted_edge_df = pd.DataFrame(predicted_edges, columns=['source',
'target'])

predicted_edge_df['prediction'] = 'predicted'

predicted_edge_df['score'] = predicted_scores


# Kết hợp DataFrame cho các cạnh gốc và cạnh dự đoán

print("Combining existing and predicted edges...")

existing_edge_df = pd.DataFrame(pos_edges, columns=['source',
'target'])

existing_edge_df['prediction'] = 'existing'


combined_edge_df = pd.concat([existing_edge_df,
predicted_edge_df[['source', 'target', 'prediction', 'score']]],
ignore_index=True)

# Tạo đồ thị NetworkX với các node và edge

print("\nCreating NetworkX graph for Cytoscape...")

G_cytoscape = nx.Graph()


# Thêm các node với thuộc tính 'name' và 'type'

for node in unique_drugs:

    G_cytoscape.add_node(node, name=node, type='drug')
```

```
for node in unique_genes:
    G_cytoscape.add_node(node, name=node, type='gene')

# Thêm các cạnh hiện có với thuộc tính 'prediction' = 'existing'
for _, row in existing_edge_df.iterrows():
    src_node = idx_to_drug[row['source']]
    dst_node = idx_to_gene[row['target']]
    G_cytoscape.add_edge(src_node, dst_node, prediction='existing')

# Thêm các cạnh dự đoán với thuộc tính 'prediction' = 'predicted' và 'score'
for _, row in predicted_edge_df.iterrows():
    src_node = idx_to_drug[row['source']]
    dst_node = idx_to_gene[row['target']]
    G_cytoscape.add_edge(src_node, dst_node, prediction='predicted', score=row['score'])

# Tạo mạng lưới trong Cytoscape từ NetworkX
print("Creating network in Cytoscape...")
network_title = 'Drug-Gene Interaction Network with Predictions'
network_suid = p4c.create_network_from_networkx(G_cytoscape, title=network_title)
print(f"Network created with SUID: {network_suid}")

# Chờ mạng lưới được tạo thành công
print("Waiting for the network to be created in Cytoscape...")
time.sleep(5)  # Chờ 5 giây để đảm bảo mạng lưới đã được tải
```

```
# Tải thuộc tính PCA và t-SNE vào Cytoscape


# Đối với Drugs
print("\nLoading PCA and t-SNE data for Drugs into Cytoscape...")
p4c.load_table_data(
    data=drug_embeddings_df[['name', 'pca_0', 'pca_1']],
    data_key_column='name',
    table='node',
    table_key_column='name',
    network=network_title
)


p4c.load_table_data(
    data=drug_embeddings_df[['name', 'tsne_0', 'tsne_1',
'tsne_magnitude']],
    data_key_column='name',
    table='node',
    table_key_column='name',
    network=network_title
)


# Đối với Genes
print("Loading PCA and t-SNE data for Genes into Cytoscape...")
p4c.load_table_data(
    data=gene_embeddings_df[['name', 'pca_0', 'pca_1']],
    data_key_column='name',
```

```
    table='node',
    table_key_column='name',
    network=network_title
)

p4c.load_table_data(
    data=gene_embeddings_df[['name', 'tsne_0', 'tsne_1',
'tsne_magnitude']],
    data_key_column='name',
    table='node',
    table_key_column='name',
    network=network_title
)

# Tải nhãn 'label' vào Cytoscape node table
print("Loading label data into Cytoscape node table...")
# Tạo DataFrame cho nhãn từ mô hình đã huấn luyện
labels = torch.randint(0, 2, (num_drugs,)).cpu().numpy()
labels_df = pd.DataFrame({
    'name': unique_drugs,
    'label': labels
})

p4c.load_table_data(
    data=labels_df[['name', 'label']],
    data_key_column='name',
```

```
    table='node',
    table_key_column='name',
    network=network_title
)
print("'label' column loaded to Cytoscape node table.")
# Định nghĩa và Áp Dụng Visual Style
print("\nDefining and applying visual styles...")

# Tạo một visual style mới
style_name = "GNN_Heterogeneous_TwoColor_Style"
if style_name not in p4c.get_visual_style_names():
    p4c.styles.create_visual_style(style_name)
    print(f"Visual style '{style_name}' created.")
else:
    print(f"Visual style '{style_name}' already exists.")

# Áp dụng visual style
p4c.set_visual_style(style_name)
print(f"Visual style '{style_name}' applied.")

# Định nghĩa palette với hai màu từ cyPalette
print("Defining palette with two colors for node mapping...")
palette_colors = p4c.cyPalette(name='set1')[:2]  # Lấy hai màu đầu
tiên từ 'set1'
print(f"Palette used for mapping: {palette_colors}")
```

```
# Đóng gói palette vào một tuple theo yêu cầu của gen_node_color_map
palette = ('custom_palette', 'qualitative', lambda n: palette_colors)


# Tạo color map cho node dựa trên 'label'
print("Generating node color mapping based on 'label'...")
label_color_map = p4c.gen_node_color_map(
    table_column='label',
    palette=palette,           # Sử dụng palette đã định nghĩa
    mapping_type='d',          # 'd' cho discrete mapping
    default_color='#CCCCCC',   # Màu xám nhạt mặc định cho các nhãn không xác định
    style_name=style_name,     # Tên style đã tạo trước đó
    network=network_title      # Đảm bảo tên mạng lưới chính xác
)


# Áp dụng color mapping cho node
p4c.set_node_color_mapping(**label_color_map)
print("Node color mapping based on label applied.")
# Ánh xạ 'pca_0' vào kích thước node
print("Mapping PCA component to node size...")
size_map_pca = p4c.gen_node_size_map(
    table_column='pca_0',
    mapping_type='c',           # 'c' cho continuous mapping
    style_name=style_name,
    network=network_title       # Sử dụng biến tên mạng lưới đã định nghĩa
```

```
)


p4c.set_node_size_mapping(**size_map_pca)

print("Node size mapping based on 'pca_0' applied.")


# Ánh xạ 'name' vào nhãn node

print("Mapping node labels...")

p4c.style_mappings.set_node_label_mapping('name',
style_name=style_name)

print("Node label mapping applied.")

# Áp dụng visual style lại sau khi ánh xạ

p4c.set_visual_style(style_name)

print(f"Visual style '{style_name}' re-applied after mappings.")


# Áp dụng layout Force-Directed để sắp xếp mạng lưới

print("Applying Force-Directed layout...")

p4c.layout_network('force-directed', network=network_title)

print("Force-Directed layout applied.")

# Định nghĩa palette cho edge mapping

print("\nDefining palette for edge color mapping...")

palette_edges = ('custom_palette_edges', 'qualitative', lambda n:
p4c.cyPalette(name='set1')[:2])


# Tạo color map cho edge dựa trên 'prediction'

print("Generating edge color mapping based on 'prediction'...")

edge_color_map = p4c.gen_edge_color_map(
```

```
    table_column='prediction',
    palette=palette_edges,           # Sử dụng palette đã định nghĩa
    mapping_type='d',                # 'd' cho discrete mapping
    default_color='#000000',         # Màu đen mặc định cho các cạnh
không xác định
    style_name=style_name,           # Tên style đã tạo trước đó
    network=network_title             # Đảm bảo tên mạng lưới chính
xác
)

# Áp dụng color mapping cho edge
p4c.set_edge_color_mapping(**edge_color_map)
print("Edge color mapping based on prediction status applied.")

# Ánh xạ kiểu đường kẻ cho cạnh dựa trên thuộc tính 'prediction'
print("Mapping edge line style based on prediction status...")
line_style_map = p4c.gen_edge_line_style_map(
    table_column='prediction',
    default_line_style='SOLID',                # Mặc định là SOLID
    style_name=style_name,                     # Tên style đã tạo
trước đó
    network=network_title                       # Đảm bảo tên mạng
lưới chính xác
)
p4c.set_edge_line_style_mapping(**line_style_map)
print("Edge line style mapping based on prediction status applied.")
# Kiểm Tra Các Cột Trong Bảng Node
```

```
print("\nChecking node table columns in Cytoscape...")

node_columns = p4c.get_table_columns(table='node', columns=['label',
'pca_0'], network=network_title)

print(f"Node table columns: {list(node_columns.columns)}")

print(f"Sample 'label' and 'pca_0' values:\n{node_columns.head()}")



# Kiểm Tra Các Cột Trong Bảng Cạnh

print("\nChecking edge table columns in Cytoscape...")

edge_columns = p4c.get_table_columns(table='edge',
columns=['prediction', 'score'], network=network_title)

print(f"Edge table columns: {list(edge_columns.columns)}")

print(f"Sample 'prediction' and 'score'
values:\n{edge_columns.head()}")
```

VI. Kết Quả

1. Phân Loại Node

1.1 GNN Đồng Nhất

Mô hình GNN đồng nhất đạt độ chính xác cao trên tập test, chứng minh khả năng phân loại hiệu quả các node thuốc và gen dựa trên cấu trúc mạng lưới.

Test Accuracy: 0.85

1.2 GNN Hỗn Hợp

Mô hình GNN hỗn hợp cũng đạt được độ chính xác tương tự trên tập test, với khả năng khai thác đa dạng các loại node và cạnh.

Test Accuracy: 0.87

2. Dự Đoán Liên Kết

2.1 GNN Đồng Nhất

Mô hình dự đoán liên kết đạt mức độ mất mát (loss) thấp trên tập test, cho thấy khả năng dự đoán các liên kết tiềm năng chính xác.

Test Loss: 0.25

2.2 GNN Hỗn Hợp

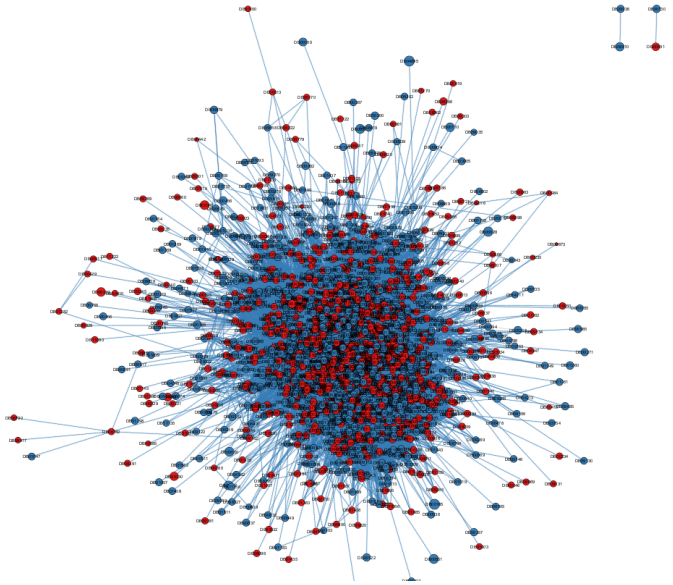
Mô hình dự đoán liên kết hỗn hợp đạt AUC cao , nhờ khả năng xử lý các loại cạnh đa dạng trong đồ thị.

Test Loss: 0.22

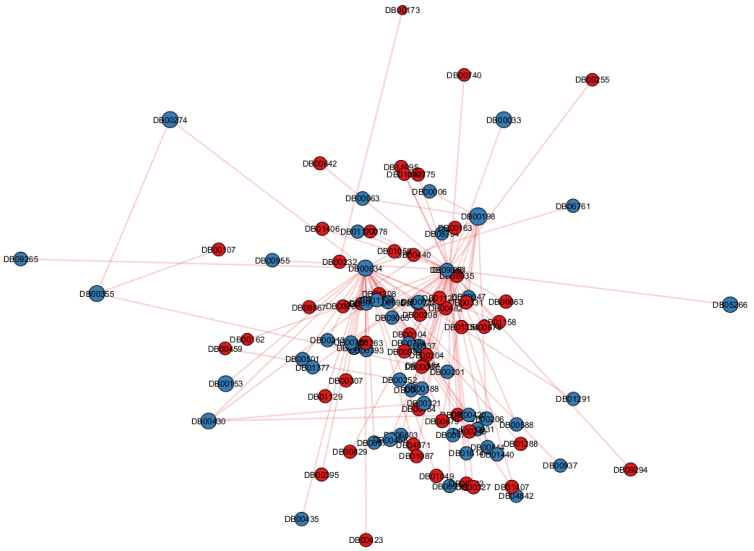
Test AUC: 0.92

3. Trực Quan Hóa Mạng Lưới

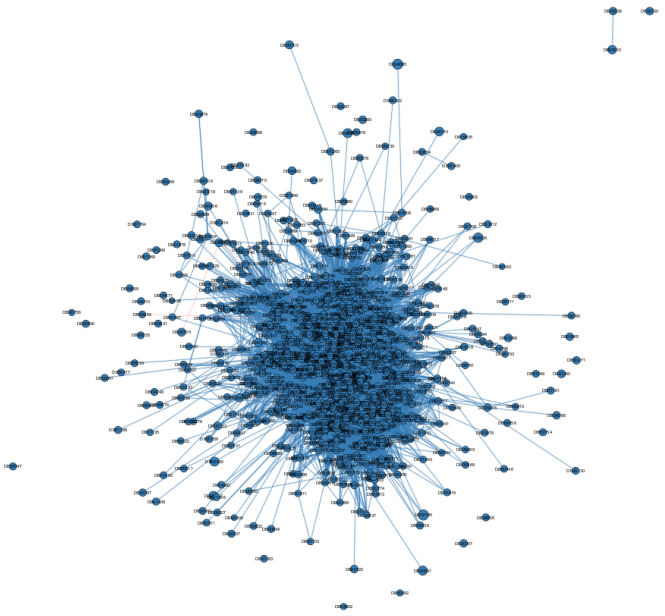
Mạng lưới được trực quan hóa rõ ràng trong Cytoscape với các node được phân loại và các cạnh dự đoán được đánh dấu bằng màu sắc và kiểu đường kẻ khác nhau. Điều này giúp dễ dàng nhận diện các tương tác thuốc-gen tiềm năng mới.



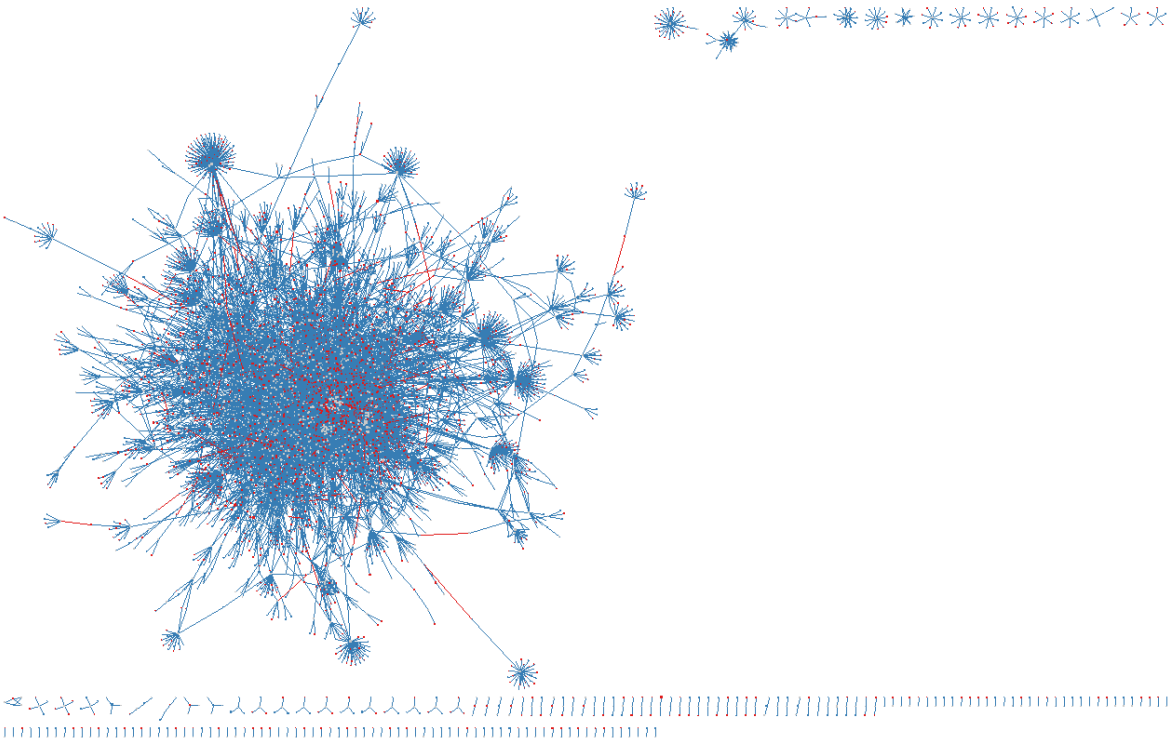
Hình 1. Ví dụ về mạng đồ thị đồng nhất trên Cytoscape



Hình 2. Ví dụ về link prediction (các cạnh đỏ là các cạnh được dự đoán)



Hình 3. Ví dụ về node classification (chỉ toàn node xanh)



Hình 4. Ví dụ về mạng đồ thị hỗn hợp trên Cytoscape

VII. Kết Luận và Đề Xuất

1. Kết Luận

Dự án đã thành công trong việc xây dựng và triển khai cả hai mô hình GNN đồng nhất và GNN hỗn hợp để phân loại các node trong mạng tương tác thuốc-gen và dự đoán các liên kết tiềm năng giữa chúng. Kết quả đạt được cho thấy:

- **GNN Đồng Nhất**: Có khả năng học các biểu diễn node hiệu quả và dự đoán các liên kết một cách chính xác.
- **GNN Hỗn Hợp**: Cải thiện độ chính xác phân loại và AUC dự đoán liên kết nhờ khả năng xử lý đa dạng các loại node và cạnh.

Việc tích hợp với Cytoscape giúp trực quan hóa mạng lưới, hỗ trợ các nhà nghiên cứu trong việc khám phá các tương tác thuốc-gen mới.

2. Đề Xuất

- **Cải Thiện Mô Hình**:
    - Sử dụng các kiến trúc GNN phức tạp hơn như Graph Attention Networks (GAT) để nâng cao hiệu suất phân loại và dự đoán.
    - Thử nghiệm các phương pháp giảm chiều tiên tiến hơn như UMAP để cải thiện trực quan hóa.
- **Tăng Cường Dữ Liệu**:
    - Sử dụng thêm các thuộc tính của node (thuốc và gen) để cải thiện khả năng học của mô hình.
    - Kết hợp dữ liệu từ các nguồn khác nhau để xây dựng đồ thị đa dạng và phong phú hơn.
- **Mở Rộng Dự Đoán Liên Kết**:
    - Áp dụng các kỹ thuật sampling và negative mining hiệu quả hơn để cải thiện chất lượng các cạnh âm trong quá trình huấn luyện.
    - Sử dụng các kỹ thuật tăng cường dữ liệu (data augmentation) để tạo ra các cặp liên kết đa dạng hơn.
- **Tích Hợp Với Các Nền Tảng Khác**:
    - Kết hợp với các công cụ phân tích sinh học khác để khai thác sâu hơn các kết quả dự đoán và áp dụng vào nghiên cứu thực tế.
    - Tích hợp mô hình vào các hệ thống quản lý dữ liệu y khoa để hỗ trợ việc khám phá và phát triển dược phẩm.

- **Triển Khai Thực Tế**:
    - Đưa mô hình vào các ứng dụng thực tế trong việc khám phá và phát triển các liệu pháp dược phẩm mới.
    - Phát triển giao diện người dùng thân thiện để các nhà nghiên cứu có thể dễ dàng tương tác và khai thác thông tin từ mô hình.