

数据挖掘课程实验

实验4 链接预测

计科210X 甘晴void 202108010XXX

数据挖掘课程实验
实验4 链接预测

实验背景

实验要求

数据集解析

实验建模

实验探索过程

失败的探索——DGL库

<0> DGL库简介

<1> 读取基因并构建图

<2> 构建GNN模型

<3> 训练模型

<4> 输出结果与可视化

<5> 模型评估

★<6> 失败总结

任务1

<1> 数据读取与构建图数据

<2> GAT 模型定义

<3> 训练模型

<4> 评估链接预测结果

<5> 创建并训练 GAT模型

<6> 链接预测和结果评估

<7> 图数据可视化部分

★<8> 结果展示

任务2

<1> 修改模型

★<2> 结果展示

<3> 总结

<4> 进一步探索，n通道

实验感悟

参考文献

实验背景

节点分类（Node Classification）是图/图谱数据上常被采用的一个学习任务，既是用模型预测图中每个节点的类别。链接预测（Link Prediction）一般指的是，对存在多对象的总体中，每个对象之间的相互作用和相互依赖关系的推断过程。

实验要求

1. 利用已经掌握的深度学习方法（比如图卷积网络、图注意力网络、对抗生成网络等），实现相关的半监督分类/预测任务。
2. 探索多图联合的深度学习方法（比如多通道卷积网络、多头注意力网络、异构图注意力网络等），实现相关的半监督分类/预测任务。
3. 面向上述方法，根据不同的training set和test set的比例，分析算法的性能指标（比如Accuracy、Precision、Recall、F1 Score等）。
4. 面向上述方法，根据不同的training set和test set的比例，分析算法性能（比如：ROC、AUC、AUPR等）。
5. 面向上述方法，根据不同的正负样本情况（比例），负样本随机选择（正样本除外），分析上述算法性能。

数据集解析

共有7个文件如下

- GeneList为基因列表，
- Positive_LinkSL为基因关系，
- feature1_go和feature2_ppi为两种基因原始特征
- Network1_SL.txt为节点之间的一个已知链接关系（这个与上面的Positive_LinkSL是一样的）
- Network2_CPDB.tsv为另一组节点之间的另一组已知链接关系
- Network3_string.tsv为另一组节点之间的另一组已知链接关系

具体描述如下：

- GeneList.txt：6375个基因，每一行为基因的英文名称。
- Positive_LinkSL.txt：总共有19667对基因关系，可以看作一种基因与基因之间的关联网络。该文件中第一列和第二列分别是基因的英文名称，第三列代表该两个基因的置信分数。（Network2_CPDB.tsv与Network3_string.tsv相仿）
- feature1_go.txt：共有6375行，128列，每一行代表一个基因，每一列代表该基因的一个维度的特征值。
- feature2_ppi.txt：共有6375行，128列，每一行代表一个基因，每一列代表该基因的一个维度的特征值。

可以较为简单地理解如下：

- 对于基因编号*i*，**GeneList**内保存了基因*i*对应的名称，
- 对于基因编号*i*，*j*。**Positive_LinkSL**内保存了基因*i*和*j*的联系，该文件内的每一行都是某两个基因之间的联系以及该联系的置信分数。（**Network2_CPDB.tsv**与**Network3_string.tsv**相仿）
- 对于基因编号*i*，剩下两个以“feature”开头的文件的每一行有128列，每一列是刻画该基因的一个维度的特征值。可以理解为对于基因的刻画有两个角度（**ppi**和**go**），每个角度有128个维度的特征。这两个文件都各自有6375行，对应6375个基因。

★但是由于**Network2_CPDB**和**Network3_string**并没有给出相应的节点特征信息，我认为给出的信息应该是不全的，故没有采用。

实验建模

对于上述信息可以概述如下：

任务1

- **GeneList**为节点列表，**feature1_go**和**feature2_ppi**为节点特征，**Positive_LinkSL**为边及边权。
- 先构建图，再使用图深度学习方法完成节点表示学习。
- 划分数据集和测试集，进行链接预测。
- 要求给出指标Accuracy、Precision、Recall、F1 Score，ROC、AUC、AUPR。

任务2

- **GeneList**为节点列表，**feature1_go**和**feature2_ppi**为节点特征，**Network1_SL**为边及边权。
- 先构建图，再使用多图联合的图深度学习方法完成节点表示学习。
- 划分数据集和测试集，进行链接预测。
- 要求给出指标Accuracy、Precision、Recall、F1 Score，ROC、AUC、AUPR。

实验探索过程

失败的探索——DGL库

<0> DGL库简介

DGL（Deep Graph Library）是一个用于图神经网络（GNN）的开源深度学习库。它为研究人员和开发者提供了在图结构数据上进行深度学习的工具和接口。DGL支持多种图神经网络模型，包括GCN（Graph Convolutional Network）、GraphSAGE（Graph Sample and Aggregation）、GAT（Graph Attention Network）等。

DGL的主要特点包括：

- 图抽象： DGL将图抽象为节点和边的集合，允许用户以一种直观的方式操作和处理图数据。
- 多后端支持： DGL支持多个深度学习框架，如PyTorch、TensorFlow和MXNet，使用户能够选择他们喜欢的框架进行图神经网络的开发。
- 灵活性： DGL提供了一系列用于创建、操作和分析图的API，使用户能够自定义模型和操作以满足不同的需求。
- 性能优化： DGL致力于提供高性能的图神经网络计算，通过优化底层实现，使得处理大规模图数据成为可能。

<1> 读取基因并构建图

读取基因数据和构建图：

- 通过`open`函数读取基因列表文件（'GeneList.txt'），将每行的基因名存储在`gene_list`列表中。
- 创建基因到索引的映射`gene_dict`，将基因名映射为索引。
- 读取基因关系和置信分数文件（'Positive_LinkSL.txt'），提取源节点、目标节点和置信分数。
- 通过`torch.tensor`创建包含边索引和置信分数的图数据结构`graph`。
- 从文件中读取两个特征矩阵（'feature1_go.txt'和'feature2_ppi.txt'）并用`torch.tensor`转换为PyTorch张量。
- 将特征数据添加到图的节点和边数据中。

该部分的代码如下

```
1 # 读取基因列表
2 with open('GeneList.txt', 'r') as f:
3     gene_list = [line.strip() for line in f]
4 # 构建基因到索引的映射
```

```

5 gene_dict = {gene: idx for idx, gene in enumerate(gene_list)}
6
7 # 读取基因关系和置信分数
8 with open('Positive_LinkSL.txt', 'r') as f:
9     edges = [line.strip().split() for line in f]
10 # 提取基因关系的源节点、目标节点和置信分数
11 src_nodes = [gene_dict[edge[0]] for edge in edges] +
    [gene_dict[edge[1]] for edge in edges]
12 dst_nodes = [gene_dict[edge[1]] for edge in edges] +
    [gene_dict[edge[0]] for edge in edges]
13 confidence_scores = [float(edge[2]) for edge in edges] +
    [float(edge[2]) for edge in edges]
14
15 # 读取特征
16 with open('feature1_go.txt', 'r') as file:
17     feature1_go = np.array([list(map(float, line.split())) for
    line in file])
18 with open('feature2_ppi.txt', 'r') as file:
19     feature2_ppi = np.array([list(map(float, line.split())) for
    line in file])
20
21 # 构建图
22 edges = torch.tensor(src_nodes), torch.tensor(dst_nodes)
23 graph = dgl.graph(edges)
24 graph.edata['confidence'] =
    torch.tensor(confidence_scores, dtype=torch.float32)
25 graph.ndata['feature1_go'] =
    torch.tensor(feature1_go, dtype=torch.float32)
26 graph.ndata['feature2_ppi'] =
    torch.tensor(feature2_ppi, dtype=torch.float32)
27
28 """print(graph)
29 # 输出边的权值
30 edge_weights = graph.edata['confidence'].squeeze().numpy()
31 print("Edge weights:")
32 print(edge_weights)
33 # 输出节点特征 'feature1_go'
34 feature1_go_values =
    graph.ndata['feature1_go'].squeeze().numpy()
35 print("Node Feature 'feature1_go':")
36 print(feature1_go_values)
37 # 输出节点特征 'feature2_ppi'

```

```

38 feature2_ppi_values =
    graph.ndata['feature2_ppi'].squeeze().numpy()
39 print("Node Feature 'feature2_ppi':")
40 print(feature2_ppi_values)"""
41
42 print(graph)

```

运行结果如下：

```

1 E:\anaconda\envs\python3-11\python.exe E:\python_files\数据挖掘
  \exp4\my.py
2 Graph(num_nodes=6375, num_edges=39334,
3       ndata_schemes={'feature1_go': Scheme(shape=(128,),
dtype=torch.float32), 'feature2_ppi': Scheme(shape=(128,),
dtype=torch.float32)})
4       edata_schemes={'confidence': Scheme(shape=(),
dtype=torch.float32)})

```

该部分是成功的，成功地将我们需要的所有信息加入到图中了。

<2> 构建GNN模型

预处理结束之后，需要构建图神经网络模型

- 导入DGL库和PyTorch库。
- 定义一个包含两层SAGE卷积的GNN模型 `SAGE`。
- 使用 `construct_negative_graph` 函数构建负样本图。
- 定义一个用于计算两节点之间得分的 `DotProductPredictor` 模型。
- 定义整体的模型 `Model`，包括SAGE卷积和得分计算模块。
- 初始化模型和Adam优化器。

代码如下：

```

1 # 构建一个2层的GNN模型
2 import dgl.nn as dglnn
3 import torch.nn as nn
4 import torch.nn.functional as F
5 class SAGE(nn.Module):
6     def __init__(self, in_feats, hid_feats, out_feats):

```

```

7         super().__init__()
8         # 实例化SAGEConv, in_feats是输入特征的维度, out_feats是输出特
          征的维度, aggregator_type是聚合函数的类型
9         self.conv1 = dglnn.SAGEConv(
10             in_feats=in_feats, out_feats=hid_feats,
          aggregator_type='mean')
11         self.conv2 = dglnn.SAGEConv(
12             in_feats=hid_feats, out_feats=out_feats,
          aggregator_type='mean')
13
14     def forward(self, graph, inputs):
15         # 输入是节点的特征
16         h = self.conv1(graph, inputs)
17         h = F.relu(h)
18         h = self.conv2(graph, h)
19         return h
20
21 def construct_negative_graph(graph, k):
22     src, dst = graph.edges()
23
24     neg_src = src.repeat_interleave(k)
25     neg_dst = torch.randint(0, graph.num_nodes(), (len(src) *
          k,))
26     return dglnn.graph((neg_src, neg_dst),
          num_nodes=graph.num_nodes())
27
28 import dglnn.function as fn
29 class DotProductPredictor(nn.Module):
30     def forward(self, graph, h):
31         # h是从5.1节的GNN模型中计算出的节点表示
32         with graph.local_scope():
33             graph.ndata['h'] = h
34             graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
35             return graph.edata['score']
36
37 def compute_loss(pos_score, neg_score):
38     # 间隔损失
39     n_edges = pos_score.shape[0]
40     return (1 - pos_score.unsqueeze(1) + neg_score.view(n_edges,
          -1)).clamp(min=0).mean()
41
42 class Model(nn.Module):

```

```

43     def __init__(self, in_features, hidden_features,
out_features):
44         super().__init__()
45         self.sage = SAGE(in_features, hidden_features,
out_features)
46         self.pred = DotProductPredictor()
47     def forward(self, g, neg_g, x):
48         h = self.sage(g, x)
49         #return self.pred(g, h), self.pred(neg_g, h)
50         pos_score = self.pred(g, h)
51         neg_score = self.pred(neg_g, h)
52         return pos_score, neg_score

```

该步的图结构模型应该是没有问题的。

<3> 训练模型

完成模型定义之后，可以开始训练模型：

- 在每个训练周期中，使用 `construct_negative_graph` 生成负样本图。
- 通过前向传播计算正样本和负样本的得分，并计算间隔损失。
- 使用Adam优化器进行反向传播和参数更新。

代码如下：

```

1  node_features = graph.ndata['feature1_go']
2  n_features = node_features.shape[1]
3  k = 5
4  model = Model(n_features, 10, 5)
5  opt = torch.optim.Adam(model.parameters())
6  for epoch in range(1):
7      negative_graph = construct_negative_graph(graph, k)
8      pos_score, neg_score = model(graph, negative_graph,
node_features)
9      loss = compute_loss(pos_score, neg_score)
10     opt.zero_grad()
11     loss.backward()
12     opt.step()
13     print(f'Epoch {epoch + 1}, Loss: {loss.item()}')
14

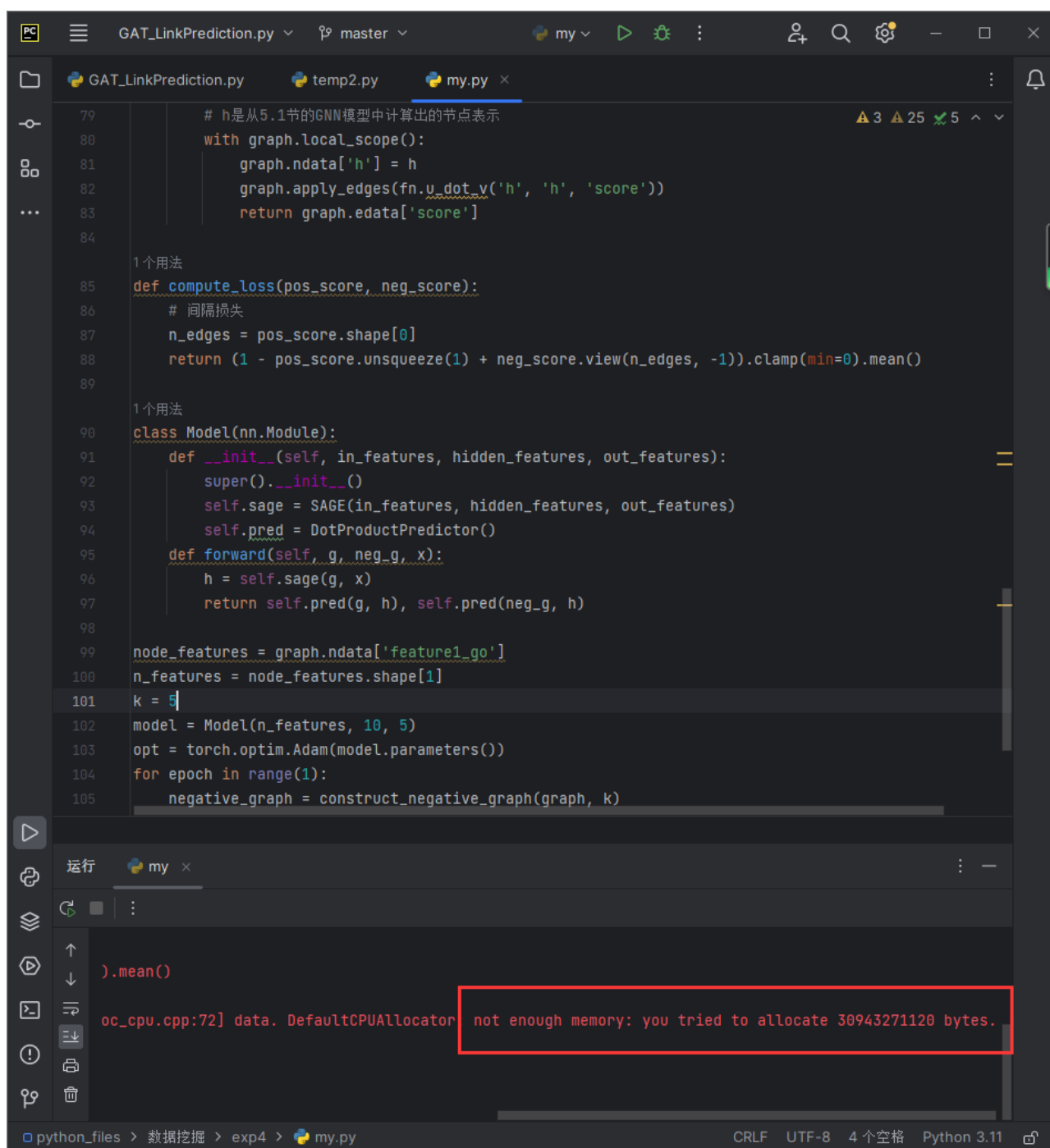
```


其中，**k** 是用于构建负样本图的参数。具体来说，对于每一对正样本边，会通过 `construct_negative_graph` 函数生成 **k** 个负样本边。构建负样本是为了训练图神经网络（GNN）模型，其中负样本边的目的是提供模型更多的信息，使其能够更好地区分正样本和负样本，从而提高模型的性能。

一般来说，**k**取值不宜过低，但是，**k**取值增大会带来计算代价的增加和内存占用的增加。

仅仅对于**k=5**，我的本地计算机就出现了较大的问题。

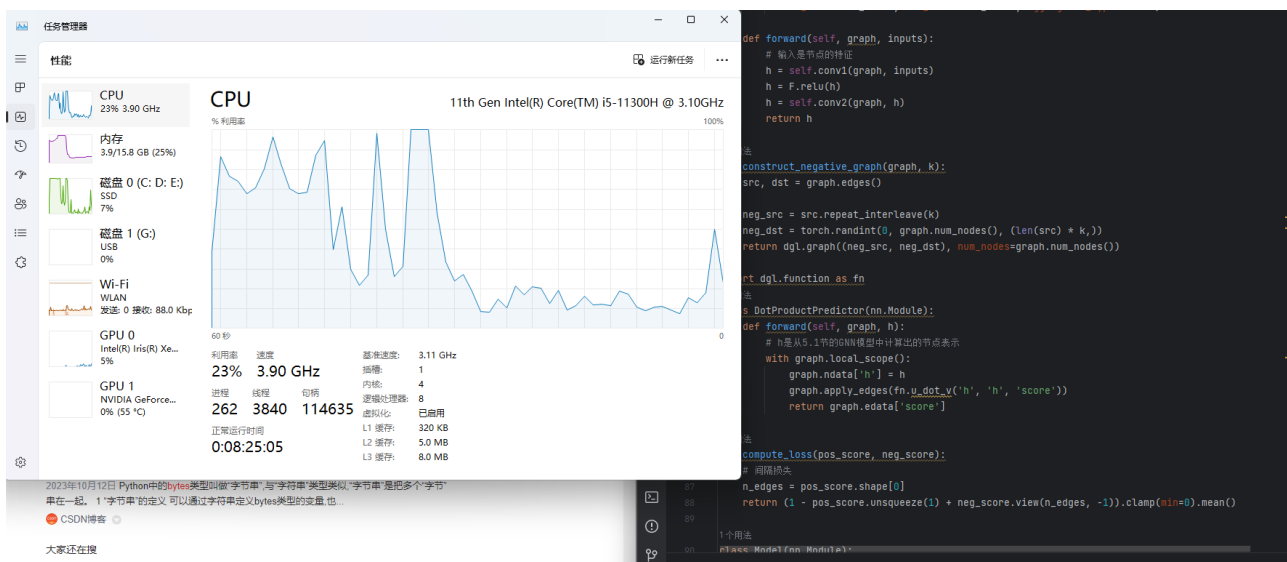
首先是内存代价的不可接受，这需要30943271120bytes内存空间，换算过后是大约28.81GB，对于本地计算机的16GB运行内存来说，这已经超出太多了。



```
79 # h是从5.1节的GNN模型中计算出的节点表示
80 with graph.local_scope():
81     graph.ndata['h'] = h
82     graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
83     return graph.edata['score']
84
85 1个用法
86 def compute_loss(pos_score, neg_score):
87     # 间隔损失
88     n_edges = pos_score.shape[0]
89     return (1 - pos_score.unsqueeze(1) + neg_score.view(n_edges, -1)).clamp(min=0).mean()
90
91 1个用法
92 class Model(nn.Module):
93     def __init__(self, in_features, hidden_features, out_features):
94         super().__init__()
95         self.sage = SAGE(in_features, hidden_features, out_features)
96         self.pred = DotProductPredictor()
97     def forward(self, g, neg_g, x):
98         h = self.sage(g, x)
99         return self.pred(g, h), self.pred(neg_g, h)
100
101 node_features = graph.ndata['feature1_go']
102 n_features = node_features.shape[1]
103 k = 5
104 model = Model(n_features, 10, 5)
105 opt = torch.optim.Adam(model.parameters())
106 for epoch in range(1):
107     negative_graph = construct_negative_graph(graph, k)
```

```
oc_cpu.cpp:72] data. DefaultCPUAllocator: not enough memory: you tried to allocate 30943271120 bytes.
```

我将**k**值调整为1，即使仅仅是这样，虽然可以运行，但是资源基本上已经被全部占用了。



此外，我还将深度学习的层数调整为了1，但

<4> 输出结果与可视化

假设上面的步骤都全部正确，接下来进行的是可视化输出。

- 打印每个训练周期的损失。
- 输出正样本的置信度分布。
- 生成随机标签 `true_labels`。
- 使用模型获取节点表示，并通过t-SNE降维到2D空间。
- 使用NetworkX库构建图结构，节点包括基因名和对应标签，边包括基因关系和得分。
- 绘制图的节点、边和标签，展示链接预测的可视化结果。

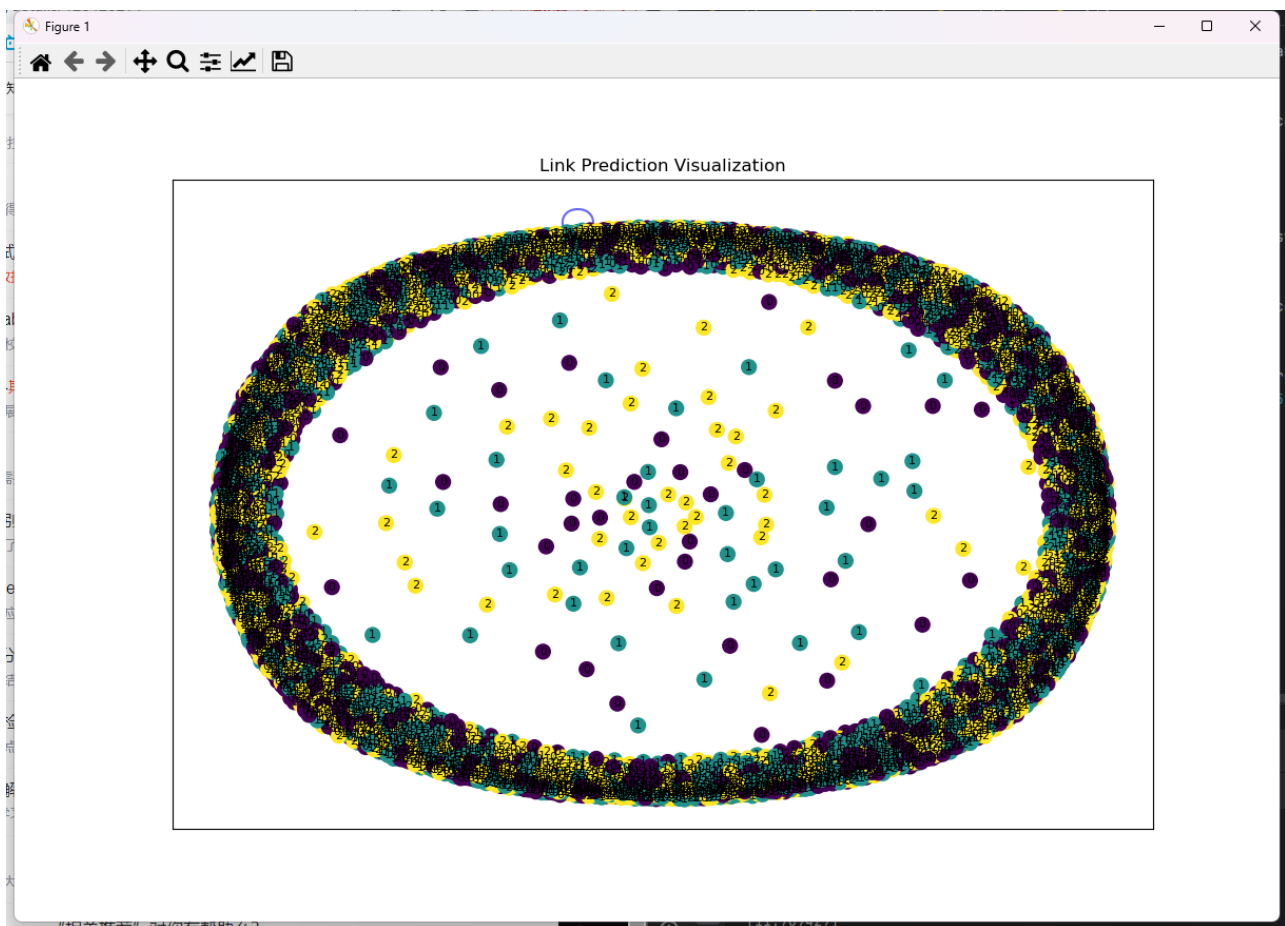
```
1 # 输出边的置信度分布
2 print("Edge Confidence Distribution:")
3 print(pos_score.detach().numpy())
4
5 import networkx as nx
6 import matplotlib.pyplot as plt
7 from sklearn.manifold import TSNE
8
9 true_labels = torch.randint(0, 3, (len(gene_list),)) # 0, 1, 2
   之间的随机标签
10
11 # 获取节点表示
12 with torch.no_grad():
13     node_embeddings = model.sage(graph, node_features).numpy()
14
```

```

15 # 将节点表示降维到二维空间进行可视化
16 tsne = TSNE(n_components=2, random_state=42)
17 node_embeddings_2d = tsne.fit_transform(node_embeddings)
18
19 # 构建 NetworkX 图
20 G = nx.Graph()
21 for i, gene in enumerate(gene_list):
22     G.add_node(gene, label=true_labels[i].item(),
23               color=true_labels[i].item())
24
25 for edge, score in zip(edges, pos_score.detach().numpy()):
26     G.add_edge(gene_list[edge[0]], gene_list[edge[1]],
27               score=score)
28
29 # 绘制图
30 plt.figure(figsize=(12, 8))
31 pos = nx.spring_layout(G, seed=42)
32 node_color = [true_labels[i].item() for i in
33               range(len(gene_list))]
34
35 # 绘制节点
36 nx.draw_networkx_nodes(G, pos, node_size=100,
37                       node_color=node_color, cmap='viridis')
38
39 # 绘制链接预测的边
40 edge_color = ['b' if score > 0.5 else 'r' for score in
41               nx.get_edge_attributes(G, 'score').values()]
42 nx.draw_networkx_edges(G, pos, edge_color=edge_color, width=1.5,
43                       alpha=0.6)
44
45 # 绘制节点标签
46 labels = nx.get_node_attributes(G, 'label')
47 nx.draw_networkx_labels(G, pos, labels=labels, font_size=8)
48
49 plt.title('Link Prediction Visualization')
50 plt.show()

```

这里为了让节点彼此区分开来，给不同的节点随机分配了颜色。



<5> 模型评估

若之前步骤正确，在这一步可以对于之前的模型进行评估。

对于Accuracy、Precision、Recall、F1 Score

```
1 # 模型评估
2 model.eval() # 切换模型为评估模式，这会影响某些层（如Dropout）
3 with torch.no_grad():
4     # 这里的 node_features 为测试集的特征
5     test_pos_score, test_neg_score = model(graph,
6     negative_graph, node_features)
7     test_predicted_labels = torch.where(test_pos_score > 0.5, 1,
8     0).numpy()
9 # 计算评估指标
10 test_true_labels = torch.randint(0, 3, (graph.num_nodes(),)) #
    替换为实际的测试集标签
11 accuracy = accuracy_score(test_true_labels.numpy(),
12 test_predicted_labels)
```

```

11 precision = precision_score(test_true_labels.numpy(),
    test_predicted_labels)
12 recall = recall_score(test_true_labels.numpy(),
    test_predicted_labels)
13 f1 = f1_score(test_true_labels.numpy(), test_predicted_labels)
14
15 print(f"Test Accuracy: {accuracy:.4f}")
16 print(f"Test Precision: {precision:.4f}")
17 print(f"Test Recall: {recall:.4f}")
18 print(f"Test F1 Score: {f1:.4f}")

```

对于ROC、AUC、AUPR

```

1 # 计算 ROC 和 AUC
2 fpr, tpr, _ = roc_curve(true_labels.numpy(),
    pos_score.detach().numpy())
3 roc_auc = roc_auc_score(true_labels.numpy(),
    pos_score.detach().numpy())
4
5 # 绘制 ROC 曲线
6 plt.figure(figsize=(8, 6))
7 plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve
    (AUC = {roc_auc:.2f})')
8 plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
9 plt.xlabel('False Positive Rate')
10 plt.ylabel('True Positive Rate')
11 plt.title('Receiver Operating Characteristic (ROC) Curve')
12 plt.legend(loc="lower right")
13 plt.show()
14
15 # 计算 AUPR
16 precision, recall, _ =
    precision_recall_curve(true_labels.numpy(),
    pos_score.detach().numpy())
17 auapr = average_precision_score(true_labels.numpy(),
    pos_score.detach().numpy())
18
19 # 绘制 Precision-Recall 曲线
20 plt.figure(figsize=(8, 6))
21 plt.step(recall, precision, color='b', alpha=0.2, where='post')

```

```

22 plt.fill_between(recall, precision, step='post', alpha=0.2,
    color='b')
23 plt.xlabel('Recall')
24 plt.ylabel('Precision')
25 plt.title('Precision-Recall Curve (AUPR =
    {0:.2f})'.format(aupr))
26 plt.show()

```

★<6> 失败总结

由于DGL对于资源的需求实在太大了，本地计算机的内存和算力都不能满足要求，故本实验使用该种方法似乎并不能得到满意的结果。

DGL是一个很好用的工具，但是确实不太适合本地计算机来运行。

以上的代码与推演，照理应该是正确的，在算力和内存等资源充足的地方应该能发挥效果。

任务1

<1> 数据读取与构建图数据

- `read_data(file_path)`: 读取文件中的数据，并返回每一行的列表。
- `build_graph_data(gene_list, link_list, feature1, feature2)`: 构建图数据，包括节点特征 (`feature1` 和 `feature2`)，边的索引 (`edge_index`) 和边的属性 (`edge_attr`)。同时，构建了一个基因字典 `gene_dict` 用于将基因名称映射到索引。

定义读取文件的函数如下

```

1 def read_data(file_path):
2     with open(file_path, 'r') as f:
3         data = f.read().splitlines()
4     return data

```

其中，对于图数据的构建如下：

```

1 # 构建图数据
2 def build_graph_data(gene_list, link_list, feature1, feature2):

```

```

3     edge_index = []
4     edge_attr = []
5     x1 = []
6     x2 = []
7
8     gene_dict = {gene: idx for idx, gene in
enumerate(gene_list)}
9
10    for link in link_list:
11        gene1, gene2, confidence = link.split('\t')
12        if gene1 in gene_dict and gene2 in gene_dict:
13            edge_index.append([gene_dict[gene1],
gene_dict[gene2]])
14            edge_attr.append(float(confidence))
15
16    edge_index = torch.tensor(edge_index,
dtype=torch.long).t().contiguous()
17    edge_attr = torch.tensor(edge_attr,
dtype=torch.float).view(-1, 1)
18
19    for gene in gene_list:
20        if gene in gene_dict:
21            x1.append(feature1[gene_dict[gene]])
22            x2.append(feature2[gene_dict[gene]])
23
24    x1 = torch.tensor(x1, dtype=torch.float)
25    x2 = torch.tensor(x2, dtype=torch.float)
26
27    data = Data(x1=x1, x2=x2, edge_index=edge_index,
edge_attr=edge_attr)
28    return data

```

读取基因列表 (`GeneList.txt`)、链接列表 (`Positive_LinkSL.txt`) 以及两个特征文件 (`feature1_go.txt` 和 `feature2_ppi.txt`)。然后划分数据集为训练集和测试集，并构建相应的图数据。在主函数中调用的读取代码如下：

```

1  # 读取数据
2  gene_list = read_data('GeneList.txt')
3  link_list = read_data('Positive_LinkSL.txt')
4  feature1 = np.loadtxt('feature1_go.txt')
5  feature2 = np.loadtxt('feature2_ppi.txt')
6
7  # 划分数据集和测试集
8  train_gene_list, test_gene_list = train_test_split(gene_list,
    test_size=0.2, random_state=42)
9
10 # 构建训练集和测试集的图数据
11 train_data = build_graph_data(train_gene_list, link_list,
    feature1, feature2)
12 test_data = build_graph_data(test_gene_list, link_list,
    feature1, feature2)

```

<2> GAT 模型定义

- `GATModel(nn.Module)`: 定义了一个简单的 GAT 模型，使用了 `GATConv` 层。

```

1  # GAT 模型定义
2  class GATModel(nn.Module):
3      def __init__(self, in_channels, out_channels, heads):
4          super(GATModel, self).__init__()
5          self.conv1 = GATConv(in_channels, out_channels,
    heads=heads)
6
7      def forward(self, x, edge_index, edge_attr):
8          x = self.conv1(x, edge_index, edge_attr)
9          return x

```


<3> 训练模型

- `train(model, data, optimizer, criterion, epochs)`: 训练 GAT 模型。在每个 epoch 中，计算模型的损失值，并将其记录在 `losses` 列表中。训练完成后，通过 Matplotlib 绘制损失曲线图。

```
1  # 训练模型
2  def train(model, data, optimizer, criterion, epochs):
3      model.train()
4      losses = [] # 用于记录每个 epoch 的损失值
5      for epoch in range(epochs):
6          optimizer.zero_grad()
7          out = model(data.x1, data.edge_index, data.edge_attr)
8          loss = criterion(out, data.x2)
9          loss.backward()
10         optimizer.step()
11         losses.append(loss.item()) # 记录当前 epoch 的损失值
12         print(f'Epoch {epoch + 1}/{epochs}, Loss:
13             {loss.item()}')
14
15     # 绘制损失曲线图
16     plt.plot(losses)
17     plt.title('Training Loss Over Epochs')
18     plt.xlabel('Epoch')
19     plt.ylabel('Loss')
20     plt.show()
```

<4> 评估链接预测结果

- `evaluate(y_true, y_pred)`: 使用 sklearn 库中的指标评估链接预测结果，包括准确率、精确度、召回率、F1 分数、ROC AUC 和平均精度 (AUPR)。

```

1 def evaluate(y_true, y_pred):
2     y_true = (y_true > 0.3).int().cpu().numpy()
3     y_pred = (y_pred > 0.3).int().cpu().numpy()
4
5     accuracy = accuracy_score(y_true, y_pred)
6     precision = precision_score(y_true, y_pred, average='micro')
7     recall = recall_score(y_true, y_pred, average='micro')
8     f1 = f1_score(y_true, y_pred, average='micro')
9     roc_auc = roc_auc_score(y_true, y_pred)
10    aupr = average_precision_score(y_true, y_pred)
11
12    return accuracy, precision, recall, f1, roc_auc, aupr

```

<5> 创建并训练 GAT模型

- 创建 GAT 模型，定义优化器和损失函数，然后调用 `train` 函数进行模型训练。

```

1 # 创建并训练 GAT 模型
2 model = GATModel(in_channels=128, out_channels=128, heads=1)
3 optimizer = optim.Adam(model.parameters(), lr=0.001)
4 criterion = nn.MSELoss()
5
6 train(model, train_data, optimizer, criterion, epochs=200)

```

指定训练次数为100次，学习率调为0.001。

<6> 链接预测和结果评估

- 使用训练好的模型对测试集进行链接预测，然后调用 `evaluate` 函数评估预测结果。

```

1  # 进行链接预测
2  pred_scores = model(test_data.x1, test_data.edge_index,
3                      test_data.edge_attr)
4
5  # 评估链接预测结果
6  accuracy, precision, recall, f1, roc_auc, aupr =
7  evaluate(test_data.x2, pred_scores)
8
9  print(f'Accuracy: {accuracy} \nPrecision: {precision} \nRecall:
10 {recall} \nF1 Score: {f1}')
11
12 print(f'ROC AUC: {roc_auc} \nAUPR: {aupr}')
```

<7> 图数据可视化部分

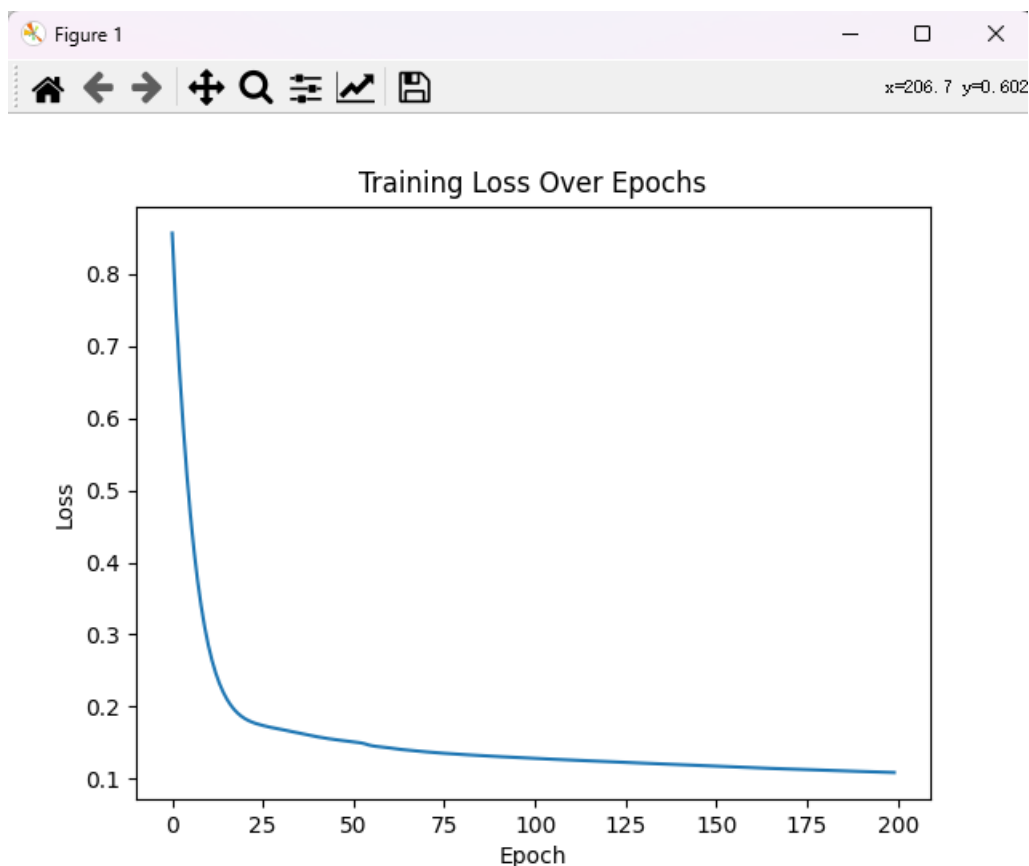
- 将 PyTorch Geometric 图数据转换为 NetworkX 图，使用 NetworkX 绘制图的布局，并通过 Matplotlib 进行绘制。

```

1  import networkx as nx
2  import torch
3  from torch_geometric.data import Data
4
5
6  # 将 PyTorch Geometric 图数据转换为 NetworkX 图
7  G = nx.Graph()
8  G.add_nodes_from(range(test_data.num_nodes))
9  G.add_edges_from(test_data.edge_index.t().tolist())
10
11 # 使用 NetworkX 绘制图
12 pos = nx.spring_layout(G)
13 nx.draw(G, pos, with_labels=True, font_weight='bold',
14         node_color='lightblue', node_size=1000, font_size=8,
15         edge_color='gray')
16 plt.show()
```

★<8> 结果展示

使用上述模型进行运行，损失曲线如下。



部分损失值如下：

```
1 Epoch 1/200, Loss: 0.8566558361053467
2 Epoch 2/200, Loss: 0.7528260350227356
3 Epoch 3/200, Loss: 0.6675369143486023
4 Epoch 4/200, Loss: 0.5916386842727661
5 Epoch 5/200, Loss: 0.5249260067939758
6 Epoch 6/200, Loss: 0.46694767475128174
7 Epoch 7/200, Loss: 0.41712379455566406
8 Epoch 8/200, Loss: 0.37475287914276123
9 Epoch 9/200, Loss: 0.3390277028083801
10 Epoch 10/200, Loss: 0.309112012386322
11 Epoch 11/200, Loss: 0.284216046333313
12 Epoch 12/200, Loss: 0.2636083960533142
13 Epoch 13/200, Loss: 0.2465600073337555
14 Epoch 14/200, Loss: 0.23244094848632812
15 .....
16 Epoch 195/200, Loss: 0.10945269465446472
17 Epoch 196/200, Loss: 0.10929632186889648
```

```
18 Epoch 197/200, Loss: 0.10914068669080734
19 Epoch 198/200, Loss: 0.1089857891201973
20 Epoch 199/200, Loss: 0.10883160680532455
21 Epoch 200/200, Loss: 0.10867814719676971
```

进行200次之后，大概在0.1左右。

模型评估结果如下

```
1 Accuracy: 0.4549019607843137
2 Precision: 0.8565955895528382
3 Recall: 0.9963490534849291
4 F1 Score: 0.9212020532584679
5 ROC AUC: 0.5012495279165683
6 AUPR: 0.8531546660454162
```

解释如下：

- 准确率 (**Accuracy**): 0.45，表示正确预测的链接占总链接的比例。
- 精确度 (**Precision**): 0.86，表示在所有模型预测为正的链接中，有 86% 是正确的。
- 召回率 (**Recall**): 0.996，表示在所有实际为正的链接中，模型成功预测了 99.6%。
- **F1 分数 (F1 Score)**: 0.92，是精确度和召回率的调和平均值，提供了模型在正类别上的综合性能指标。
- **ROC AUC**: 0.50，表示模型在正例和负例之间的区分能力，ROC AUC 约接近 0.5，说明模型的性能接近随机猜测。
- **AUPR (平均精度)**: 0.85，表示模型在正例上的精度，AUPR 越接近 1 表示性能越好。

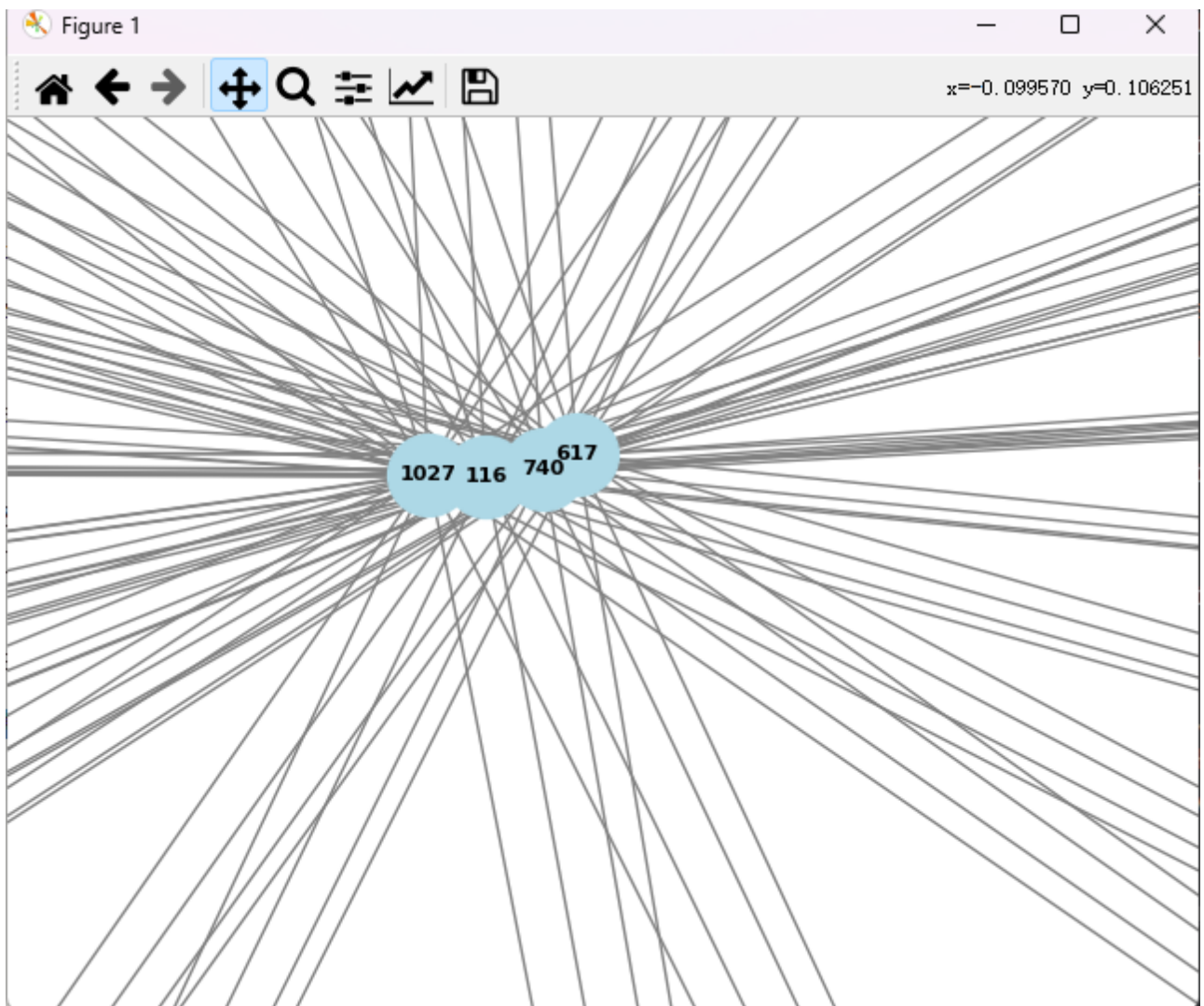
构建基因链接预测图如下

（选取预测分数大于指定阈值的链接作为预测有关的链接）



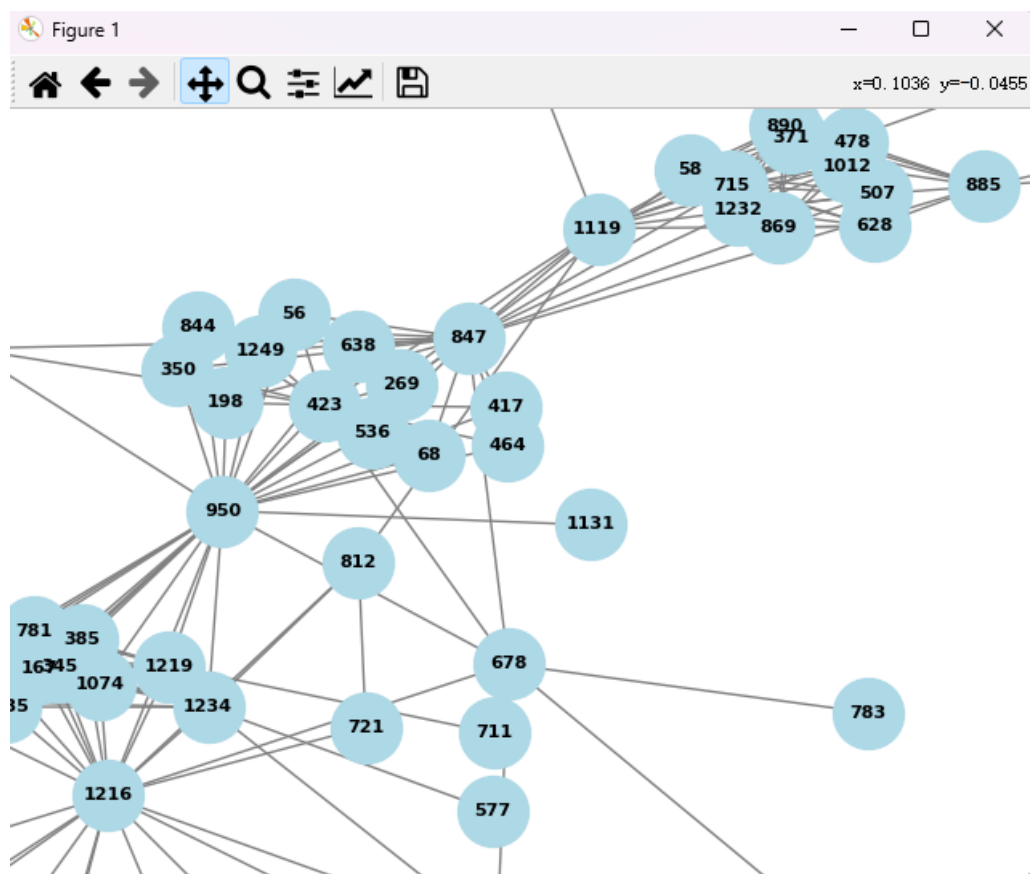
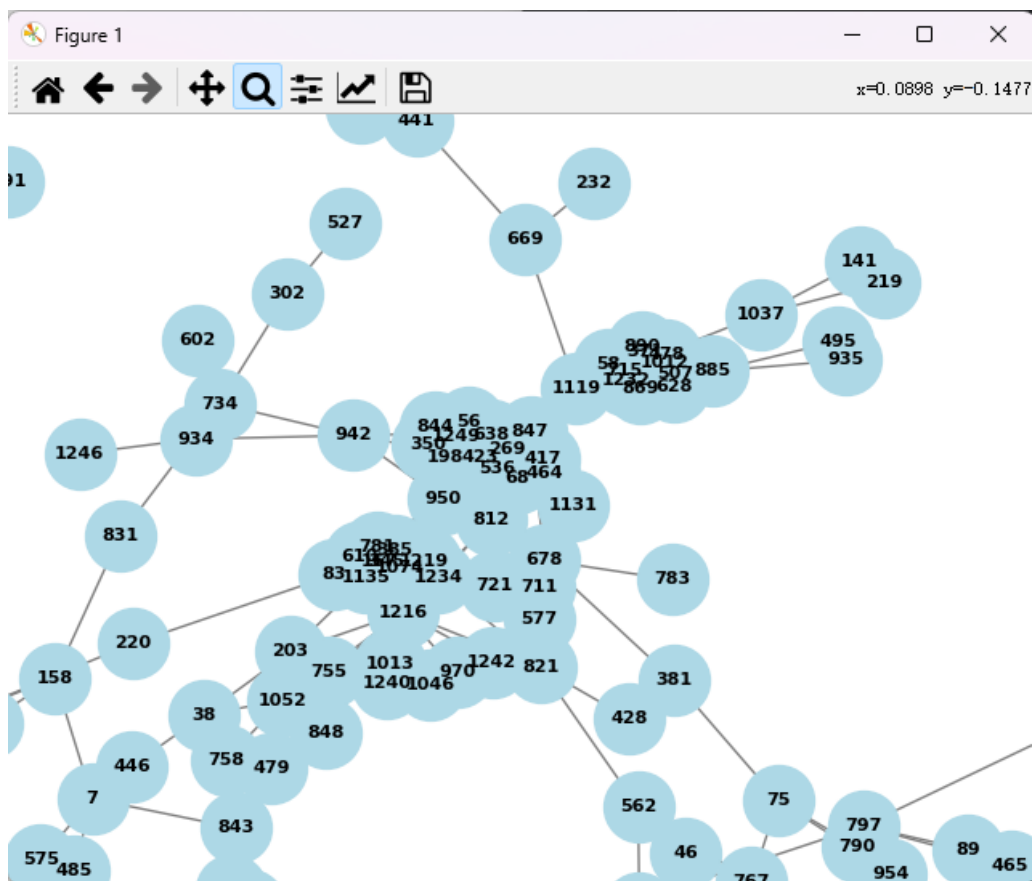
这是整体的趋势图，对于其中的局部放大可以看到目标基因之间的联系。

对于中间部分，与周围联系较多的节点，可以通过节点编号查到基因名



- 1 1027 CLDN23
- 2 116 ADRB1
- 3 740 CBR3
- 4 617 C1QBP

下面是一些其它的局部结构



任务2

使用多通道在刚刚的基础上对模型和训练做修改

<1> 修改模型

这里我们使用的多通道卷积网络。所以对于模型的定义需要修改，把原本的单通道扩展成多个，并在适当的地方进行合并。

```
1  # Multi-Channel Graph Convolutional Network 模型定义
2  class MultiChannelGCN(nn.Module):
3      def __init__(self, in_channels, out_channels):
4          super(MultiChannelGCN, self).__init__()
5          self.conv1 = GCNConv(in_channels, out_channels)
6          self.conv2 = GCNConv(in_channels, out_channels)
7
8      def forward(self, x1, x2, edge_index, edge_attr):
9          x1 = self.conv1(x1, edge_index, edge_attr)
10         x2 = self.conv2(x2, edge_index, edge_attr)
11         return x1, x2
```

除了要在模型定义的地方进行修改，在训练函数以及调用函数也要进行修改。

修改训练函数：

```
1  # 训练模型
2  def train(model, data, optimizer, criterion, epochs):
3      model.train()
4      losses = [] # 用于记录每个 epoch 的损失值
5      for epoch in range(epochs):
6          optimizer.zero_grad()
7          out1, out2 = model(data.x1, data.x2, data.edge_index,
8                              data.edge_attr)
9          loss1 = criterion(out1, data.x1)
10         loss2 = criterion(out2, data.x2)
11         loss = loss1 + loss2
12         loss.backward()
13         optimizer.step()
14         losses.append(loss.item()) # 记录当前 epoch 的损失值
15         print(f'Epoch {epoch + 1}/{epochs}, Loss:
16               {loss.item()}')
```

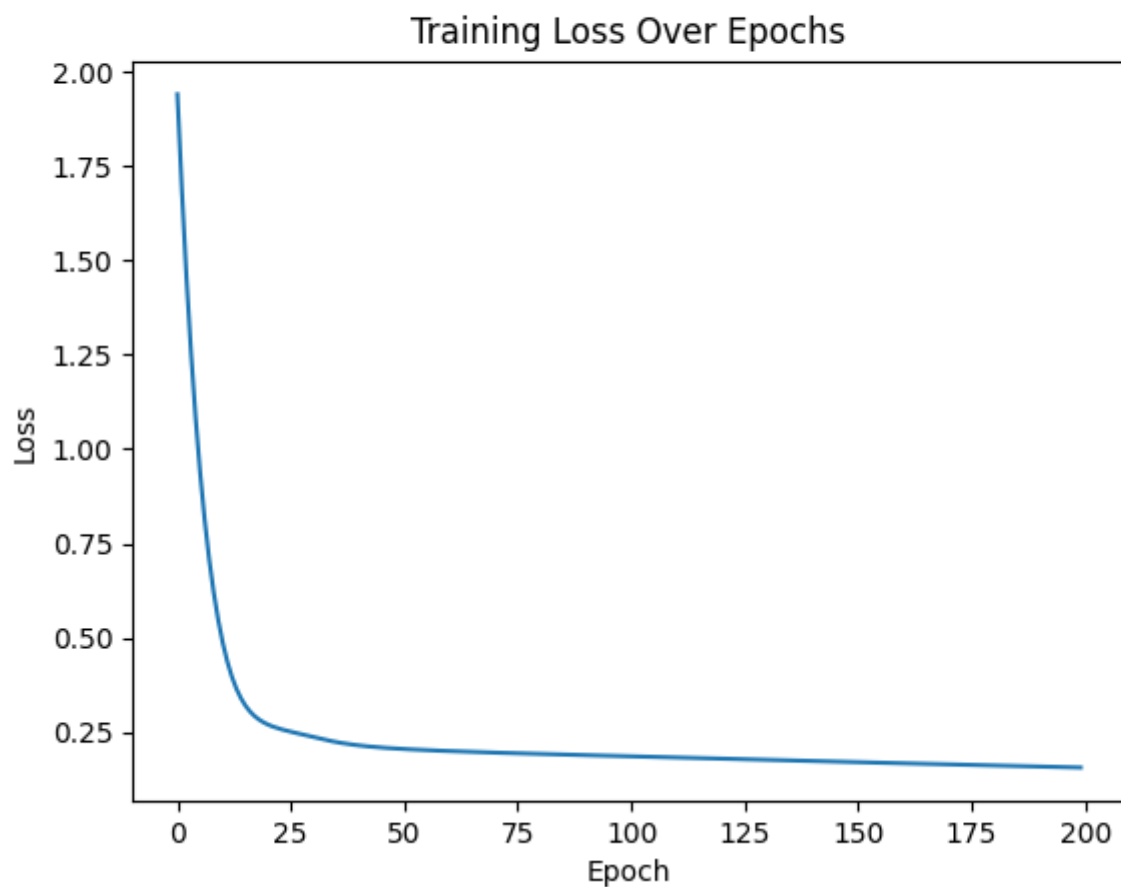
```
15
16     # 绘制损失曲线图
17     plt.plot(losses)
18     plt.title('Training Loss Over Epochs')
19     plt.xlabel('Epoch')
20     plt.ylabel('Loss')
21     plt.show()
```

修改调用部分：

```
1  # 进行链接预测
2  pred_scores1, pred_scores2 = model(test_data.x1, test_data.x2,
3  test_data.edge_index, test_data.edge_attr)
3  pred_scores = (pred_scores1 + pred_scores2) / 2 # 取两个通道的平均值
```

这样就将其转化为了一个使用双通道的图卷积网络模型。

★<2> 结果展示



损失率如下：

```

1 Epoch 1/200, Loss: 1.9401469230651855
2 Epoch 2/200, Loss: 1.682145357131958
3 Epoch 3/200, Loss: 1.4546871185302734
4 Epoch 4/200, Loss: 1.2563203573226929
5 Epoch 5/200, Loss: 1.084963083267212
6 Epoch 6/200, Loss: 0.9381833076477051
7 Epoch 7/200, Loss: 0.8134356737136841
8 Epoch 8/200, Loss: 0.708167552947998
9 Epoch 9/200, Loss: 0.6199674606323242
10 Epoch 10/200, Loss: 0.5466182827949524
11 Epoch 11/200, Loss: 0.48613178730010986
12 Epoch 12/200, Loss: 0.4367343485355377
13 Epoch 13/200, Loss: 0.39682072401046753
14 Epoch 14/200, Loss: 0.36491310596466064
15 .....
16 Epoch 195/200, Loss: 0.15746958553791046
17 Epoch 196/200, Loss: 0.1571885496377945

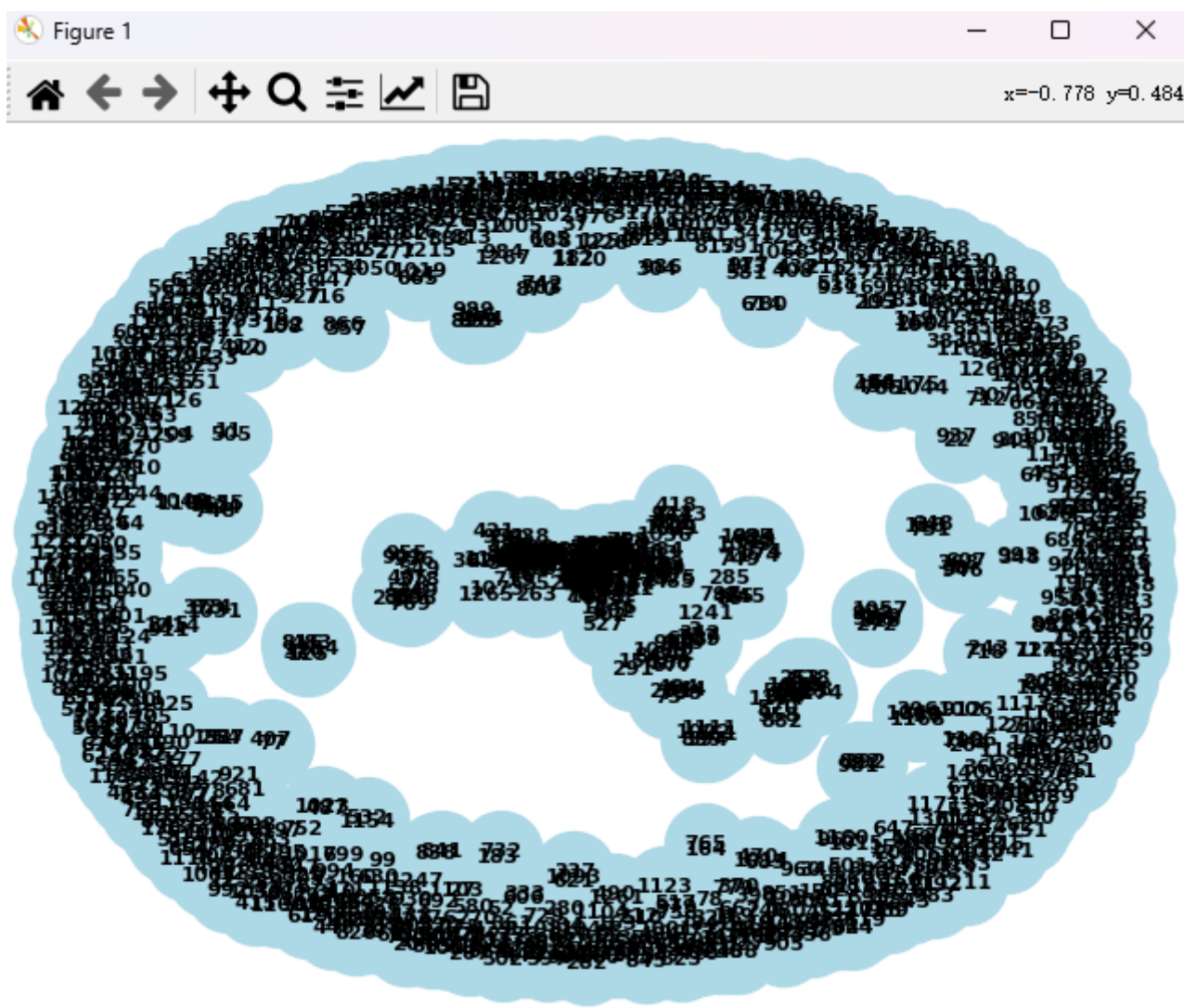
```

```
18 Epoch 197/200, Loss: 0.15690799057483673
19 Epoch 198/200, Loss: 0.15662789344787598
20 Epoch 199/200, Loss: 0.15634828805923462
21 Epoch 200/200, Loss: 0.15606917440891266
```

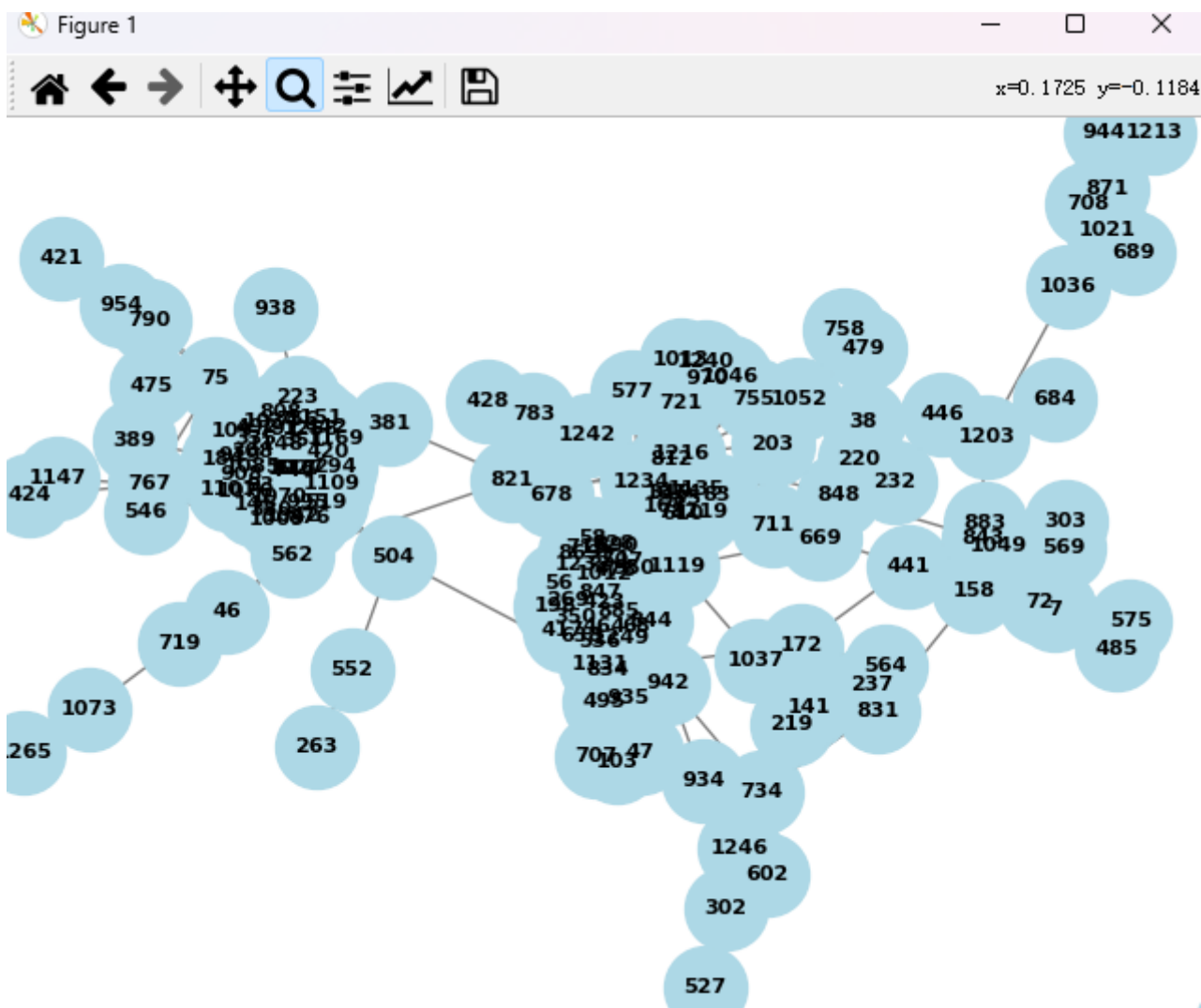
指标评估如下:

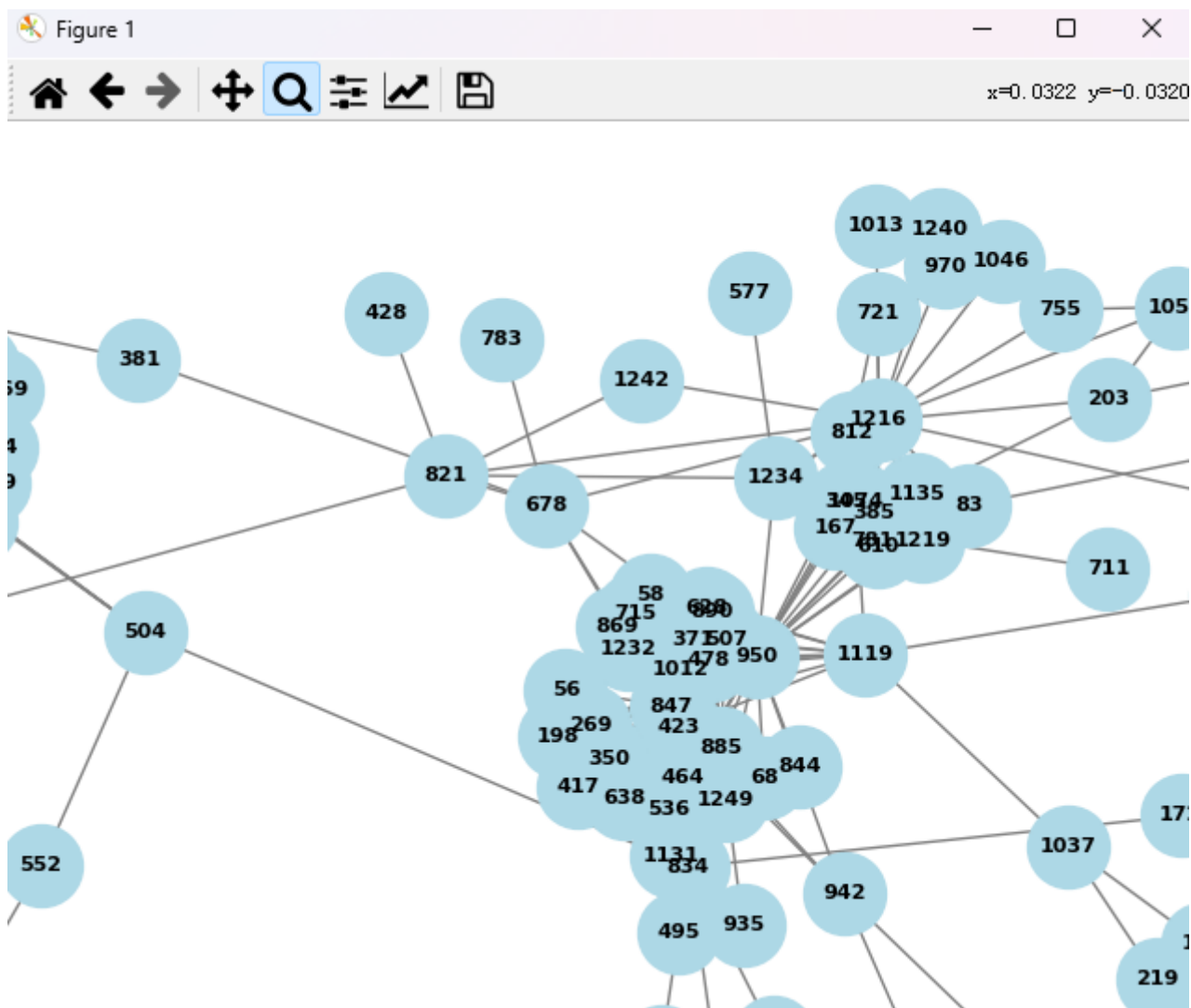
```
1 Accuracy: 0.5427450980392157
2 Precision: 0.8652827615217433
3 Recall: 0.9757082692501186
4 F1 Score: 0.9171837684645032
5 ROC AUC: 0.5324953459502417
6 AUPR: 0.8606581811658711
```

整体展示如下:



部分局部展示如下:





<3> 总结

使用双通道后，由于对于原来的特征彼此之间区分信息的保留变多了，所以链接的预测正确率有明显的上升。所以双通道以及多通道的图神经网络还是有好处的。

<4> 进一步探索，**n**通道

继续修改刚刚的代码，使用数组替换模型中的x1和x2，达到**n**通道的效果，如下：

```

1 # Multi-Channel Graph Convolutional Network 模型定义
2 class MultiChannelGCN(nn.Module):
3     def __init__(self, in_channels, out_channels, num_channels):
4         super(MultiChannelGCN, self).__init__()
5         self.channels = nn.ModuleList([GCNConv(in_channels,
6 out_channels) for _ in range(num_channels)])
7
8     def forward(self, *inputs):
9         output_channels = [channel(x, inputs[-2], inputs[-1]) for
10 channel, x in zip(self.channels, inputs[:-2])]
11         return output_channels

```

详细代码附在后面，修改代码中的 `num_channels =`，调整为想要的通道数即可。

发现将通道从1上调至2后，正确率上升效果明显，继续上调后，正确率上升效果不明显。

这是通道数目为10时的结果：

```

1 Accuracy: 0.5435294117647059
2 Precision: 0.8650597497897928
3 Recall: 0.976010119158845
4 F1 Score: 0.9171917738830919
5 ROC AUC: 0.5317604380292384
6 AUPR: 0.8605590887908858

```

上升不显著，基本还是在0.54，其余指标基本都略微有变化，但变化不是很多。故认为2通道基本已经能满足要求。

实验感悟

由于老师将收作业的时间延后了，我确实有更多的时间来进行探究，感觉对于图神经网络有了一个更为直观的感悟。但是我还是没有从一个更底层的角度去深究其原理，仅仅停留在代码层面，还是不够的，还有很多需要学习的地方。

本学期在数据挖掘上确实学习到了很多。

参考文献

使用图神经网络进行链接预测

- https://docs.dgl.ai/tutorials/blitz/4_link_predict.html
- https://docs.dgl.ai/en/0.8.x/guide_cn/training-link.html
- <https://github.com/Giantjc/LinkPrediction>
- https://zhuanlan.zhihu.com/p/599510610?utm_id=0
- https://docs.dgl.ai/en/latest/guide_cn/training-node.html