金融异常检测任务 - 程序报告

吴天宇 12334125

1 实验概要

1.1 实验内容

本实验旨在利用图神经网络（Graph Neural Networks）和多层感知机（MLP）在金融领域进行异常检测，识别欺诈用户。我们将基于 DGraph-Fin 数据集，该数据集包含用户之间的社交网络关系和节点特征。

实验主要包括以下内容：

- 使用 PyTorch 和 PyTorch Geometric 进行图数据的加载和预处理。
- 定义并训练多层感知机（MLP）模型和 GraphSAGE 模型。
- 评估模型在节点分类任务中的性能，主要使用 AUC（Area Under the Curve）作为评估指标。
- 分析模型的训练过程和结果。

1.2 实验结果概要

在本实验中，我们分别训练了 MLP 模型和 GraphSAGE 模型。通过对比，我们发现：

- MLP 模型：只利用节点的特征信息，未考虑图结构，训练速度较快，但在验证集上的 AUC 表现有限。
- GraphSAGE 模型：结合了节点的特征和邻居信息，通过图卷积捕捉节点之间的关系，在验证集上取得了更高的 AUC。

最终，GraphSAGE 模型在验证集上取得了更优的性能，证明了利用图结构信息对于金融异常检测任务的重要性。

2 多层感知机 MLP 模型

2.1 导入必要的库

In [ ]:
```
import torch
import torch.nn.functional as F
import torch.nn as nn
import torch_geometric.transforms as T
from utils import DGraphFin
from utils.evaluator import Evaluator
import os

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

2.2 数据加载和预处理

In [ ]:
```
# ==================== 数据加载和预处理 ====================
# 路径和参数设置
path = './datasets/632d74d4e2843a53167ee9a1-momodel/'  # 数据保存路径
save_dir = './results/'  # 模型保存路径
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
dataset_name = 'DGraph'  # 数据集名称

dataset = DGraphFin(root=path, name=dataset_name, transform=T.ToSparseTensor())
data = dataset[0]

# 数据预处理
x = data.x
x = (x - x.mean(0)) / x.std(0)  # 标准化节点特征
data.x = x

# 划分训练集、验证集和测试集
split_idx = {
    'train': data.train_mask,
    'valid': data.valid_mask,
    'test': data.test_mask
}
train_idx = split_idx['train']
```

```
# 将数据移动到设备上（GPU 或 CPU）
data = data.to(device)
```

2.3 定义模型

In [ ]:
```
# ==================== 定义模型 ====================
class MLP(nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, num_layers, dropout, batchnorm=True):
        super(MLP, self).__init__()
        # 定义多层感知机结构
        self.lins = nn.ModuleList([nn.Linear(in_channels, hidden_channels)])
        self.bns = nn.ModuleList([nn.BatchNorm1d(hidden_channels)]) if batchnorm else None
        for _ in range(num_layers - 2):
            self.lins.append(nn.Linear(hidden_channels, hidden_channels))
            if batchnorm:
                self.bns.append(nn.BatchNorm1d(hidden_channels))
        self.lins.append(nn.Linear(hidden_channels, out_channels))
        self.dropout = dropout

    def reset_parameters(self):
        #重置模型参数
        for lin in self.lins:
            lin.reset_parameters()
        if self.bns:
            for bn in self.bns:
                bn.reset_parameters()

    def forward(self, x):
        #模型前向传播
        for i, lin in enumerate(self.lins[:-1]):
            x = lin(x)
            if self.bns:
                x = self.bns[i](x)
            x = F.relu(x)
            x = F.dropout(x, p=self.dropout, training=self.training)
        x = self.lins[-1](x)
        return F.log_softmax(x, dim=-1)
```

2.4 训练和评估函数

In [ ]:
```
# ==================== 训练和评估函数 ====================
# 训练超参数设置
num_layers = 5
hidden_channels = 128

mlp_parameters = {'num_layers': num_layers, 'hidden_channels':hidden_channels, 'dropout': 0.5, 'batchnorm':
in_channels, out_channels = data.x.size(-1), 2
model = MLP(in_channels, **mlp_parameters, out_channels=out_channels).to(device)

# 优化器和损失函数
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-4)

# 评估器
evaluator = Evaluator('auc')

def train(model, data, train_idx, optimizer):
    """
    训练模型
    :param model: 模型对象
    :param data: 数据对象
    :param train_idx: 训练集索引
    :param optimizer: 优化器
    :return: 损失值
    """
    model.train()
    optimizer.zero_grad()
    out = model(data.x[train_idx])
    loss = F.nll_loss(out, data.y[train_idx].squeeze().long())
    loss.backward()
    optimizer.step()
    return loss.item()

def test(model, data, split_idx, evaluator):
    """
    测试模型性能
    :param model: 模型对象
    :param data: 数据对象
    :param split_idx: 数据集划分字典
```

```
    :param evaluator: 评估器
    :return: 评估结果、损失和预测值
    """
    model.eval()
    with torch.no_grad():
        losses, eval_results = {}, {}
        for key in ['train', 'valid']:
            node_id = split_idx[key]
            out = model(data.x[node_id])
            y_pred = out.exp()
            losses[key] = F.nll_loss(out, data.y[node_id].squeeze().long()).item()
            eval_results[key] = evaluator.eval(data.y[node_id].squeeze().long(), y_pred)['auc']
    return eval_results, losses, y_pred
```

2.5 训练模型

In [ ]:
```
# ==================== 训练模型 ====================
def train_model(model, data, split_idx, optimizer, evaluator, save_dir, epochs=1000, log_steps=10):
    """
    执行模型训练并保存最佳模型。
    :param model: 模型对象
    :param data: 数据对象
    :param split_idx: 数据集划分字典
    :param optimizer: 优化器
    :param evaluator: 评估器
    :param save_dir: 模型保存路径
    :param epochs: 训练轮数
    :param log_steps: 日志记录频率
    """
    best_valid_auc, min_valid_loss = 0, float('inf')
    train_idx = split_idx['train']

    for epoch in range(1, epochs + 1):
        loss = train(model, data, train_idx, optimizer)
        eval_results, losses, _ = test(model, data, split_idx, evaluator)
        train_auc, valid_auc = eval_results['train'], eval_results['valid']
        train_loss, valid_loss = losses['train'], losses['valid']

        # 保存最优模型
        if valid_loss < min_valid_loss:
            min_valid_loss = valid_loss
            torch.save(model.state_dict(), os.path.join(save_dir, f'best_mlp_model_layers{num_layers}_hidde

        if epoch % 10 == 0:
            print(f'第 {epoch:04d} 轮，损失值：{loss:.4f}，训练集 AUC：{train_auc * 100:.2f}% ，验证集 AUC：{val

train_model(model, data, split_idx, optimizer, evaluator, save_dir)
```

2.6 保存并加载最佳模型

In [ ]:
```
# ==================== 保存并加载最佳模型 ====================
def load_best_model(model, save_dir):
    """
    加载最佳模型权重。
    :param model: 模型对象
    :param save_dir: 模型保存路径
    :return: 加载权重后的模型
    """
    model.load_state_dict(torch.load(os.path.join(save_dir, 'best_mlp_model.pt')))
    return model
```

2.7 测试函数

In [ ]:
```
# ==================== 测试函数 ====================
def predict(model, data, node_id):
    """
    预测指定节点的标签概率。
    :param model: 训练好的模型
    :param data: 数据对象
    :param node_id: 节点索引
    :return: 节点的预测概率
    """
    model.eval()
    with torch.no_grad():
        out = model(data.x[node_id].unsqueeze(0))
        y_pred = out.exp()
    return y_pred
```

3 图神经网络 GraphSAGE 模型

3.1 导入必要的库

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import SAGEConv
from torch_geometric.data import Data
import torch_geometric.transforms as T
from utils import DGraphFin
from utils.evaluator import Evaluator
import numpy as np
import os

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

3.2 数据加载和预处理

In [ ]:
```
# ==================== 数据加载和预处理 ====================
# 数据路径设置
path = './datasets/632d74d4e2843a53167ee9a1-momodel/'  # 数据保存路径
save_dir = './results/'  # 模型保存路径
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
dataset_name = 'DGraph'  # 数据集名称

# 加载数据集
dataset = DGraphFin(root=path, name=dataset_name, transform=T.ToSparseTensor())
nlabels = 2  # 仅需预测类别 0 和类别 1
data = dataset[0]
data.adj_t = data.adj_t.to_symmetric()  # 将有向图转换为无向图

# 数据预处理
x = data.x
x = (x - x.mean(0)) / x.std(0)  # 标准化节点特征
data.x = x
if data.y.dim() == 2:
    data.y = data.y.squeeze(1)  # 如果标签维度为 2，则压缩为 1 维

# 划分训练集、验证集和测试集
split_idx = {
    'train': data.train_mask,
    'valid': data.valid_mask,
    'test': data.test_mask
}
train_idx = split_idx['train']

# 将数据移动到设备上（GPU 或 CPU）
data = data.to(device)

# 将稀疏邻接矩阵 adj_t 转换为 edge_index（适用于 SAGEConv）
row, col, _ = data.adj_t.coo()  # 获取 COO 格式的行、列索引
data.edge_index = torch.stack([row, col], dim=0)  # 构建 edge_index 矩阵，形状为 [2, num_edges]
```

3.3 定义模型

In [ ]:
```
# ==================== 定义模型 ====================
class GraphSAGE(nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GraphSAGE, self).__init__()
        # 定义三个 SAGEConv 层
        self.conv1 = SAGEConv(in_channels, hidden_channels)
        self.conv2 = SAGEConv(hidden_channels, hidden_channels)
        self.conv3 = SAGEConv(hidden_channels, out_channels)

        # 定义用于残差连接的线性层
        self.res1 = nn.Linear(in_channels, hidden_channels) if in_channels != hidden_channels else None
        self.res2 = nn.Linear(hidden_channels, hidden_channels)

    def reset_parameters(self):
        # 重置模型参数
        self.conv1.reset_parameters()
        self.conv2.reset_parameters()
        self.conv3.reset_parameters()
        if self.res1:
            self.res1.reset_parameters()
```

```
        self.res2.reset_parameters()

    def forward(self, x, edge_index):
        # 第一层卷积 + 残差连接
        identity = x  # 保存输入以用于残差连接
        x = F.relu(self.conv1(x, edge_index))  # 图卷积和激活函数
        if self.res1:
            identity = self.res1(identity)  # 如果维度不同，调整维度
        x1 = x + identity  # 残差连接

        # 第二层卷积 + 残差连接
        identity = x1
        x = F.relu(self.conv2(x1, edge_index))
        x2 = x + self.res2(identity)  # 残差连接

        # 第三层卷积（输出层）
        x3 = self.conv3(x2, edge_index)

        # 使用 Log Softmax 获取类别概率
        return F.log_softmax(x3, dim=-1)

# 实例化模型并移动到设备上
in_channels = data.x.size(-1)  # 输入特征维度
hidden_channels = 2            # 隐藏层维度
out_channels = nlabels         # 输出类别数
model = GraphSAGE(
    in_channels=in_channels,
    hidden_channels=hidden_channels,
    out_channels=out_channels
).to(device)
```

3.4 训练和评估函数

In [ ]:

```
# ==================== 训练和评估函数 ====================
# 训练超参数设置
epochs = 20            # 训练轮数
lr = 0.005               # 学习率
weight_decay = 2e-4      # 权重衰减（L2 正则化系数）

# 优化器和损失函数
optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)

# 评估器
eval_metric = 'auc'  # 使用 AUC 作为评估指标
evaluator = Evaluator(eval_metric)

# 定义训练函数
def train(model, data, train_idx, optimizer):
    model.train()  # 设置模型为训练模式
    optimizer.zero_grad()  # 清空梯度
    out = model(data.x, data.edge_index)  # 前向传播
    loss = F.nll_loss(out[train_idx], data.y[train_idx])  # 计算损失（负对数似然损失）
    loss.backward()  # 反向传播
    optimizer.step()  # 更新参数
    return loss.item()  # 返回损失值

# 定义测试函数
def test(model, data, split_idx, evaluator):
    model.eval()  # 设置模型为评估模式
    with torch.no_grad():
        out = model(data.x, data.edge_index)  # 前向传播
        y_pred = out.exp()  # 将 Log Softmax 输出转换为概率
        eval_results = {}
        losses = {}
        for key in ['train', 'valid']:
            node_id = split_idx[key]
            losses[key] = F.nll_loss(out[node_id], data.y[node_id]).item()  # 计算损失
            # 计算评估指标（AUC）
            eval_results[key] = evaluator.eval(data.y[node_id], y_pred[node_id])[eval_metric]
    return eval_results, losses  # 返回评估结果和损失
```

3.5 训练、保存并加载最佳模型

In [ ]:

```
# ==================== 训练、保存并加载最佳模型 ====================
best_valid_auc = 0  # 初始化最佳验证集 AUC
best_model_state = None  # 用于保存最佳模型状态

for epoch in range(1, epochs + 1):
    loss = train(model, data, train_idx, optimizer)  # 训练一步
```

```
    eval_results, losses = test(model, data, split_idx, evaluator)  # 在训练集和验证集上测试
    train_auc = eval_results['train']
    valid_auc = eval_results['valid']

    if valid_auc > best_valid_auc:
        best_valid_auc = valid_auc
        best_model_state = model.state_dict()  # 保存当前最佳模型状态
        # 保存最佳模型
        model_filename = f'best_sage_model_conv3_hidden{hidden_channels}_lr{lr}_wd{weight_decay}.pt'
        torch.save(best_model_state, os.path.join(save_dir, model_filename))

    if epoch % 10 == 0:
        print(f'第 {epoch:04d} 轮，损失值：{loss:.4f}，训练集 AUC：{train_auc * 100:.2f}% ，验证集 AUC：{valid_au

print("训练完成。")
print(f"最佳验证集 AUC：{best_valid_auc * 100:.2f}%")
print(f"最佳模型已保存至 {os.path.join(save_dir, model_filename)}")

model.load_state_dict(torch.load(os.path.join(save_dir, model_filename), map_location=device))
```

3.6 测试并保存预测结果函数

In [ ]:

```
# ==================== 测试并保存预测结果函数 ====================
def test_and_save_predictions(model, data, save_path):
    """
    运行模型的前向传播，并保存所有节点的预测结果
    :param model: 训练好的模型
    :param data: 包含节点特征和边的图数据
    :param save_path: 保存预测结果的文件路径
    """
    model.eval()  # 设置模型为评估模式
    with torch.no_grad():
        # 对所有节点进行前向传播
        out = model(data.x, data.edge_index)
        y_pred = out.exp()  # 将 Log Softmax 输出转换为概率

    # 保存预测结果
    torch.save(y_pred.cpu(), save_path)
    print(f"预测结果已保存至 {save_path}")

# 运行模型并保存预测结果
predictions_save_path = os.path.join(
    save_dir,
    f'best_sage_model_conv3_hidden{hidden_channels}_lr{lr}_wd{weight_decay}_predictions.pt'
)
test_and_save_predictions(model, data, predictions_save_path)
```

3.7 测试-预测函数

In [ ]:

```
# ==================== 测试-预测函数 ====================
def predict(data, node_id):
    """
    加载模型并在 MoAI 平台进行预测
    :param data: 数据对象，包含 x 和 edge_index 等属性
    :param node_id: int，需要进行预测的节点索引
    :return: tensor，类别 0 和类别 1 的概率
    """
    out = model
    y_pred = out[node_id].exp() # 获取指定节点的预测概率，并增加一个维度

    return y_pred  # 返回预测概率

model = torch.load('./results/best_sage_model_conv3_hidden128_lr0.002_wd0.0002_predictions.pt')
```