main

2022 年 9 月 26 日

1 基于 GNN 的金融异常检测任务

1.1 1. 实验介绍

反欺诈是金融行业永恒的主题，在互联网金融信贷业务中，数字金融反欺诈技术已经得到广泛应用并取得良好效果，这其中包括了近几年迅速发展并在各个领域得到越来越广泛应用的图神经网络。本项目以互联网智能风控为背景，从用户相互关联和影响的视角，探索满足风控反欺诈领域需求的，可拓展、高效的图神经网络应用方案，从而帮助更好地识别欺诈用户。

本项目主要关于实现预测模型 (**不限于图神经网络**)，进行节点异常检测任务，并验证模型精度。而本项目基于的数据集DGraph，DGraph 是大规模动态图数据集的集合，由真实金融场景中随着时间演变事件和标签构成。

1.1.1 1.1 实验目的

- 了解如何使用 Pytorch 进行神经网络训练
- 了解如何使用 Pytorch-geometric 等图网络深度学习库进行简单图神经网络设计 (推荐使用 GAT, GraphSAGE 模型)。
- 了解如何利用 MO 平台进行模型性能评估。

1.1.2 1.2 预备知识

- 具备一定的深度学习理论知识，如卷积神经网络、损失函数、优化器，训练策略等。
- 了解并熟悉 Pytorch 计算框架。
- 学习 Pytorch-geometric，请前往：https://pytorch-geometric.readthedocs.io/en/latest/

1.1.3 1.3 实验环境

- numpy = 1.18.5
- pytorch = 1.4.0
- torch_geometric = 1.7.0
- torch_scatter = 2.0.3
- torch_sparse = 0.5.1

1.2 2. 实验内容

1.2.1 2.1 数据集信息

DGraph-Fin 是一个由数百万个节点和边组成的有向无边权的动态图。它代表了 Finvolution Group 用户之间的社交网络，其中一个节点对应一个 Finvolution 用户，从一个用户到另一个用户的边表示**该用户将另一个用户视为紧急联系人**。下面是位于 `dataset/DGraphFin` 目录的 DGraphFin 数据集的描述:

```
x: 20维节点特征向量
y: 节点对应标签，一共包含四类。其中类1代表欺诈用户而类0代表正常用户(实验中需要进行预测的两类标
edge_index: 图数据边集,每条边的形式(id_a,id_b)，其中ids是x中的索引
edge_type: 共11种类型的边
edge_timestamp: 脱敏后的时间戳
train_mask, valid_mask, test_mask: 训练集，验证集和测试集掩码
```

本预测任务为识别欺诈用户的节点预测任务，只需要将欺诈用户（Class 1）从正常用户（Class 0）中区分出来。需要注意的是，其中测试集中样本对应的 label **均被标记为-100**。

1.2.2 2.2 导入相关包

导入相应模块，设置数据集路径、设备等。

[1]:
```
from utils import DGraphFin
from utils.utils import prepare_folder
from utils.evaluator import Evaluator
```

```
import torch
import torch.nn.functional as F
import torch.nn as nn

import torch_geometric.transforms as T

import numpy as np
from torch_geometric.data import Data
import os

# 设置 gpu 设备
device = 0
device = f'cuda:{device}' if torch.cuda.is_available() else 'cpu'
device = torch.device(device)
```

1.2.3 2.3 数据处理

在使用数据集训练网络前，首先需要对数据进行归一化等预处理，如下：

[2]:
```
path='./datasets/632d74d4e2843a53167ee9a1-momodel/' # 数据保存路径
save_dir='./results/' # 模型保存路径
dataset_name='DGraph'
dataset = DGraphFin(root=path, name=dataset_name, transform=T.ToSparseTensor())

nlabels = dataset.num_classes
if dataset_name in ['DGraph']:
    nlabels = 2    # 本实验中仅需预测类 0 和类 1

data = dataset[0]
data.adj_t = data.adj_t.to_symmetric() # 将有向图转化为无向图



if dataset_name in ['DGraph']:
    x = data.x
    x = (x - x.mean(0)) / x.std(0)
```

```
    data.x = x
if data.y.dim() == 2:
    data.y = data.y.squeeze(1)

split_idx = {'train': data.train_mask, 'valid': data.valid_mask, 'test': data.
 ↪test_mask}  # 划分训练集，验证集

train_idx = split_idx['train']
result_dir = prepare_folder(dataset_name,'mlp')
```

这里我们可以查看数据各部分维度

[3]:
```
print(data)
print(data.x.shape)  #feature
print(data.y.shape)  #label
```

```
Data(adj_t=[3700550, 3700550, nnz=4], test_mask=[183840], train_mask=[857899],
valid_mask=[183862], x=[3700550, 20], y=[3700550])
torch.Size([3700550, 20])
torch.Size([3700550])
```

1.2.4 2.4 定义模型

这里我们使用简单的多层感知机作为例子:

[6]:
```
class MLP(torch.nn.Module):
    def __init__(self
                 , in_channels
                 , hidden_channels
                 , out_channels
                 , num_layers
                 , dropout
                 , batchnorm=True):
        super(MLP, self).__init__()
        self.lins = torch.nn.ModuleList()
        self.lins.append(torch.nn.Linear(in_channels, hidden_channels))
        self.batchnorm = batchnorm
```

```
        if self.batchnorm:
            self.bns = torch.nn.ModuleList()
            self.bns.append(torch.nn.BatchNorm1d(hidden_channels))
        for _ in range(num_layers - 2):
            self.lins.append(torch.nn.Linear(hidden_channels, hidden_channels))
            if self.batchnorm:
                self.bns.append(torch.nn.BatchNorm1d(hidden_channels))
        self.lins.append(torch.nn.Linear(hidden_channels, out_channels))

        self.dropout = dropout

    def reset_parameters(self):
        for lin in self.lins:
            lin.reset_parameters()
        if self.batchnorm:
            for bn in self.bns:
                bn.reset_parameters()

    def forward(self, x):
        for i, lin in enumerate(self.lins[:-1]):
            x = lin(x)
            if self.batchnorm:
                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)
```

配置后续训练、验证、推理用到的参数。可以调整以下超参以提高模型训练后的验证精度：

- `epochs`：在训练集上训练的代数；
- `lr`：学习率；
- `num_layers`：网络的层数；
- `hidden_channels`：隐藏层维数；
- `dropout`：dropout 比例；
- `weight_decay`：正则化项的系数。

[7]:
```
mlp_parameters = {
    'lr': 0.01
    , 'num_layers': 2
    , 'hidden_channels': 128
    , 'dropout': 0.0
    , 'batchnorm': False
    , 'weight_decay': 5e-7
                }
epochs = 200
log_steps =10 # log 记录周期
```

初始化模型，并使用 **Area Under the Curve (AUC)** 作为模型评价指标来衡量模型的表现。AUC 通过对 ROC 曲线下各部分的面积求和而得。

具体计算过程参见 https://github.com/scikit-learn/scikit-learn/blob/baf828ca1/sklearn/metrics/_ranking.py#L3

[8]:
```
para_dict = mlp_parameters
model_para = mlp_parameters.copy()
model_para.pop('lr')
model_para.pop('weight_decay')
model = MLP(in_channels=data.x.size(-1), out_channels=nlabels, **model_para).
 ↪to(device)
print(f'Model MLP initialized')



eval_metric = 'auc'  # 使用 AUC 衡量指标
evaluator = Evaluator(eval_metric)
```

```
Model MLP initialized
```

1.2.5 2.5 训练

使用训练集中的节点用于训练模型，并使用验证集进行挑选模型。

[64]:
```
def train(model, data, train_idx, optimizer):
     # data.y is labels of shape (N, )
    model.train()
```

```
    optimizer.zero_grad()

    out = model(data.x[train_idx])

    loss = F.nll_loss(out, data.y[train_idx])
    loss.backward()
    optimizer.step()

    return loss.item()
```

[65]:
```
def test(model, data, split_idx, evaluator):
    # data.y is labels of shape (N, )
    with torch.no_grad():
        model.eval()

        losses, eval_results = dict(), dict()
        for key in ['train', 'valid']:
            node_id = split_idx[key]

            out = model(data.x[node_id])
            y_pred = out.exp()  # (N,num_classes)

            losses[key] = F.nll_loss(out, data.y[node_id]).item()
            eval_results[key] = evaluator.eval(data.y[node_id],␣
 ↪y_pred)[eval_metric]

    return eval_results, losses, y_pred
```

[ ]:
```
print(sum(p.numel() for p in model.parameters()))  # 模型总参数量

model.reset_parameters()
optimizer = torch.optim.Adam(model.parameters(), lr=para_dict['lr'],␣
 ↪weight_decay=para_dict['weight_decay'])
best_valid = 0
min_valid_loss = 1e8
```

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

    if valid_loss < min_valid_loss:
        min_valid_loss = valid_loss
        torch.save(model.state_dict(), save_dir+'/model.pt') # 将表现最好的模型保存

    if epoch % log_steps == 0:
        print(f'Epoch: {epoch:02d}, '
              f'Loss: {loss:.4f}, '
              f'Train: {100 * train_eval:.3f}, ' # 我们将 AUC 值乘上 100，使其在 0-100 的区间内
              f'Valid: {100 * valid_eval:.3f} ')
```

```
2946
Epoch: 10, Loss: 0.0913, Train: 64.998%, Valid: 65.393%
Epoch: 20, Loss: 0.0859, Train: 69.117%, Valid: 68.921%
Epoch: 30, Loss: 0.0693, Train: 70.410%, Valid: 69.595%
Epoch: 40, Loss: 0.0654, Train: 67.230%, Valid: 66.464%
Epoch: 50, Loss: 0.0650, Train: 70.252%, Valid: 69.475%
Epoch: 60, Loss: 0.0645, Train: 70.727%, Valid: 69.863%
Epoch: 70, Loss: 0.0642, Train: 70.896%, Valid: 69.997%
Epoch: 80, Loss: 0.0641, Train: 71.070%, Valid: 70.177%
Epoch: 90, Loss: 0.0640, Train: 71.296%, Valid: 70.390%
Epoch: 100, Loss: 0.0640, Train: 71.462%, Valid: 70.515%
Epoch: 110, Loss: 0.0639, Train: 71.585%, Valid: 70.622%
Epoch: 120, Loss: 0.0639, Train: 71.703%, Valid: 70.730%
Epoch: 130, Loss: 0.0638, Train: 71.791%, Valid: 70.799%
Epoch: 140, Loss: 0.0638, Train: 71.866%, Valid: 70.863%
Epoch: 150, Loss: 0.0638, Train: 71.930%, Valid: 70.921%
Epoch: 160, Loss: 0.0637, Train: 71.990%, Valid: 70.979%
Epoch: 170, Loss: 0.0637, Train: 72.047%, Valid: 71.034%
```

```
Epoch: 180, Loss: 0.0637, Train: 72.100%, Valid: 71.083%
Epoch: 190, Loss: 0.0637, Train: 72.148%, Valid: 71.124%
```

1.2.6 2.6 模型预测

[9]:
```
model.load_state_dict(torch.load(save_dir+'/model.pt')) # 载入验证集上表现最好的模型
def predict(data,node_id):
    """
    加载模型和模型预测
    :param node_id: int, 需要进行预测节点的下标
    :return: tensor, 类 0 以及类 1 的概率, torch.size[1,2]
    """
    # ---------------------------- 实现模型预测部分的代码␣
 ↪----------------------------
    with torch.no_grad():
        model.eval()
        out = model(data.x[node_id])
        y_pred = out.exp()  # (N,num_classes)

    return y_pred
```

[10]:
```
dic={0:"正常用户",1:"欺诈用户"}
node_idx = 0
y_pred = predict(data, node_idx)
print(y_pred)
print(f'节点 {node_idx} 预测对应的标签为:{torch.argmax(y_pred)}, 为{dic[torch.
 ↪argmax(y_pred).item()]}。')

node_idx = 1
y_pred = predict(data, node_idx)
print(y_pred)
print(f'节点 {node_idx} 预测对应的标签为:{torch.argmax(y_pred)}, 为{dic[torch.
 ↪argmax(y_pred).item()]}。')
```

```
tensor([0.9947, 0.0053])
节点 0 预测对应的标签为:0, 为正常用户。
```

```
tensor([0.9943, 0.0057])
```

节点 1 预测对应的标签为:0，为正常用户。

1.3 3. 作业评分

作业要求:

1. 请加载你认为训练最佳的模型（不限于图神经网络）
2. 提交的作业包括【程序报告.pdf】和代码文件。

注意:

1. 在训练模型等过程中如果需要**保存数据**、**模型**等请写到 **results** 文件夹，如果采用 离线任务 请务必将模型保存在 **results** 文件夹下。
2. 训练出自己最好的模型后，先按照下列 cell 操作方式实现 NoteBook 加载模型测试；请测试通过在进行【系统测试】。
3. 点击左侧栏提交作业后点击生成文件则只需勾选 `predict()` 函数的 cell，即**【模型预测代码答题区域】**的 cell。
4. 请导入必要的包和第三方库 (包括此文件中曾经导入过的)。
5. 请加载你认为训练最佳的模型，即请按要求填写**模型路径**。
6. `predict()` 函数的输入和输出请不要改动。

========== **模型预测代码答题区域** ==========

在下方的代码块中编写 **模型预测**部分的代码，请勿在别的位置作答

[17]:
```python
## 生成 main.py 时请勾选此 cell
from utils import DGraphFin
from utils.evaluator import Evaluator
import torch
import torch.nn.functional as F
import torch.nn as nn
import torch_geometric.transforms as T
from torch_geometric.data import Data
import numpy as np
import os

def predict(data,node_id):
```

```
    """
    加载模型和模型预测
    :param node_id: int, 需要进行预测节点的下标
    :return: tensor, 类 0 以及类 1 的概率, torch.size[1,2]
    """
    # 这里可以加载你的模型
    model =
    model.load_state_dict(torch.load('./results/model.pt'))
    # 模型预测时, 测试数据已经进行了归一化处理
    # -------------------------- 实现模型预测部分的代码␣
↪--------------------------



    return y_pred
```