

数据挖掘实验报告

学院: 软件学院

专业: 软件工程

|  |  |
| --- | --- |
| *作者:*  周治平 | *指导教师:*  陶乾 |
|  |  |
| *学号:*  202030483178 | *班级:*  20级软件工程1班 |

2020-11-13

# 基于图神经网络的天气推荐实验

**摘要**

极端天气情况一直困扰着人们的工作和生活。部分企业或者工种对极端天气的要求不同，但是目前主流的天气推荐系统是直接将天气信息推送给全部用户。这意味着重要的天气信息在用户手上得不到筛选，降低用户的满意度，甚至导致用户的经济损失。我们计划开发一个基于图神经网络的天气靶向模型，根据用户的历史交互行为，判断不同天气对他的利害程度。如果有必要，则将该极端天气情况推送给该用户，让其有时间做好应对准备。该模型能够减少不必要的信息传递，提高用户的体验感。

## **介绍**

1. 理解GCN工作的原理

2. 熟悉Pytorch库的使用

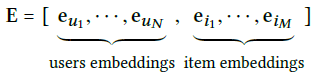
3. 掌握基于GCN的推荐系统的实现

## **方法和理论**

1.模型介绍

我们使用的模型包含三个部分：（1）嵌入层，提供用户嵌入和物品嵌入的嵌入层和初始化；（2）嵌入传播层，执行消息的传播和聚合操作；（3）预测层，聚合来自不同传播层的嵌入，并输出用户对物品的评分。

根据主流的推荐系统，一个用户（或物品）可以用一个d维的嵌入来表示。



接下来，我们构建GCN层。我们首先说明单层GNN的设计，然后将其推广到多个GCN层。

我们提供了单层嵌入传播聚合的矩阵形式：

其中是经过步嵌入传播得到的用户和项目的表示。设为初始消息传递迭代时的，即;表示单位矩阵。表示用户-物品图的拉普拉斯矩阵，公式为：

其中为用户-物品交互矩阵，为全零矩阵；是邻接矩阵，是对角度矩阵，其中第对角元素。因此，非零非对角项

通过实现矩阵形式的传播规则，我们可以以一种相当有效的方式同时更新所有用户和项目的表示。

通过层传播后，我们得到用户的多个表示，即。由于在不同的层中获得的表示反映了用户不同的偏好。因此，我们将它们连接起来，为用户构成最终的嵌入;对物品嵌入也如此。

其中为连接操作。需要注意的是，除了拼接之外，还可以应用其他聚合器，如加权平均、最大池化、平均池化等。这些聚合器隐含着不同的假设。使用拼接聚合的优点在于它的简单性，同学们可以自己探索其他聚合器的效果。

最后，我们通过内积来估计用户对目标物品的偏好:

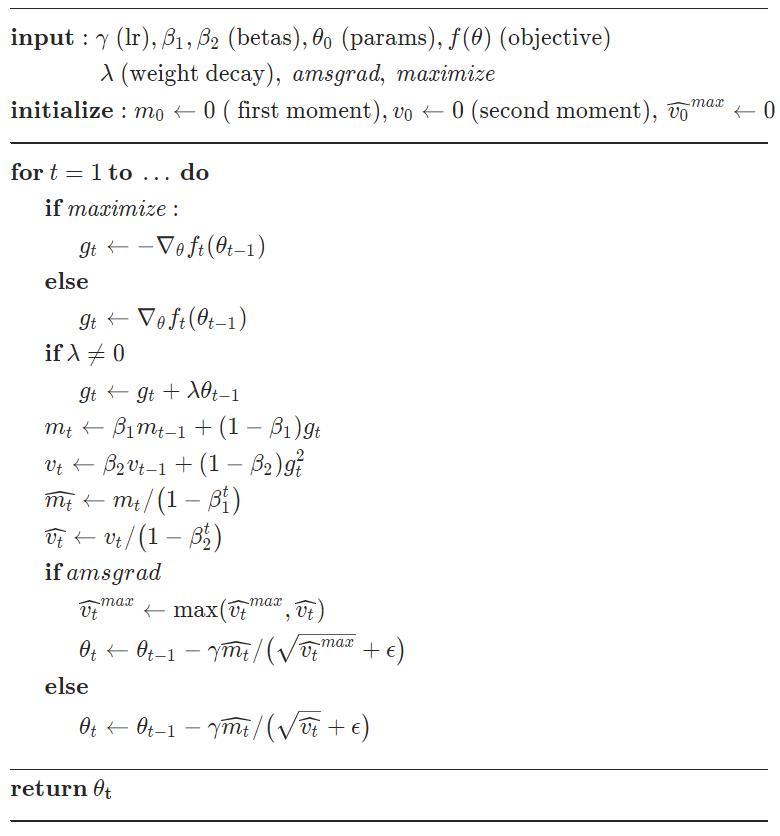
2.损失函数

在本次实验中，损失函数主要采用MSELoss函数来计算预测损失：

 在本次实验中，reduction项为空，因此默认为mean计算方法。

3.定义参数优化器

在本次实验中，GCN模型采用了Adam优化算法作为参数优化器，算法伪代码如下：



## **实验**

### 数据集

数据集共有三个txt文件，分别是user.txt，weather.txt，rating.txt。这些文件一共包含900名用户，1600个天气状况，95964条用户的历史交互记录。

1. user.txt

用户的信息记录在user.txt中。格式如下：

用户ID\t年龄\t性别\t职业\t地理位置

1. weather.txt

天气的信息记录在weather.txt中。格式如下：

天气ID\t天气类型\t温度\t湿度\t风速

1. rating.txt

用户的历史交互记录在rating.txt中。格式如下：

用户ID\t天气ID\t评分

### 实现

1.定义GCN单层模型

1. class GCN\_Layer(Module):
2. *#初始化*
3. def \_\_init\_\_(self,inF,outF):
4. super(GCN\_Layer, self).\_\_init\_\_()
5. *#两个线性层*
6. self.W1 = torch.nn.Linear(in\_features=inF,out\_features=outF)*#第一层参数*
7. self.W2 = torch.nn.Linear(in\_features=inF, out\_features=outF)*#第二层参数*
8. *#模型单层预测*
9. def forward(self,graph,selfLoop,features):
10. part1=self.W1(torch.sparse.mm(graph+selfLoop,features))*#里面包含了参数矩阵W，graph对应L，(L+I)EW1*
11. part2=self.W2(torch.mul(torch.sparse.mm(graph,features),features))*#mm为矩阵乘法，mul为矩阵点乘，(LE\*E)W*
12. return nn.LeakyReLU()(part1+part2)*#激活函数*

2.定义GCN模型

1. class GCN(Module):
2. *#初始化*
3. def \_\_init\_\_(self, args, user\_feature, item\_feature, rating):
4. super(GCN, self).\_\_init\_\_()
5. *#参数集合*
6. self.args = args
7. *#设备*
8. self.device = args.device
9. *#用户特征*
10. self.user\_feature = user\_feature
11. *#物品特征*
12. self.item\_feature = item\_feature
13. *#评价特征*
14. self.rating = rating
15. *#用户数量与物品数量*
16. self.num\_user = rating['user\_id'].max() + 1*#用户*
17. self.num\_item = rating['item\_id'].max() + 1*#物品*
18. *#num\_embeddings(python: int) – 词典的大小尺寸，比如总共出现5000个词，那就输入5000。此时index为（0 - 4999）*
19. *#embedding\_dim(python: int) – 嵌入向量的维度，即用多少维来表示一个符号。*
20. *#padding\_idx(python: int, optional) – 填充id，比如，输入长度为100，但是每次的句子长度并不一样，后面就需要用统一的数字填充，而这里就是指定这个数字，这样，网络在遇到填充id时，就不会计算其与其它符号的相关性。（初始化为0）*
21. *# user embedding 32+4+2+8+18=64*
22. self.user\_id\_embedding = nn.Embedding(user\_feature['id'].max() + 1, 32)*#id，词典尺寸，嵌入维度大小*
23. self.user\_age\_embedding = nn.Embedding(user\_feature['age'].max() + 1, 4)*#年龄*
24. self.user\_gender\_embedding = nn.Embedding(user\_feature['gender'].max() + 1, 2)*#性别*
25. self.user\_occupation\_embedding = nn.Embedding(user\_feature['occupation'].max() + 1, 8)*#职业*
26. self.user\_location\_embedding = nn.Embedding(user\_feature['location'].max() + 1, 18)*#地理位置*
27. *# item embedding 32+8+8+8+8=64*
28. self.item\_id\_embedding = nn.Embedding(item\_feature['id'].max() + 1, 32)*#id*
29. self.item\_type\_embedding = nn.Embedding(item\_feature['type'].max() + 1, 8)*#类型*
30. self.item\_temperature\_embedding = nn.Embedding(item\_feature['temperature'].max() + 1, 8)*#温度*
31. self.item\_humidity\_embedding = nn.Embedding(item\_feature['humidity'].max() + 1, 8)*#湿度*
32. self.item\_windSpeed\_embedding = nn.Embedding(item\_feature['windSpeed'].max() + 1, 8)*#风速*
33. *# 自循环*
34. self.selfLoop = self.getSelfLoop(self.num\_user + self.num\_item)*#I矩阵*
35. *# 堆叠GCN层，本次实验只使用两层*
36. self.GCN\_Layers = torch.nn.ModuleList()
37. for \_ in range(self.args.gcn\_layers):
38. self.GCN\_Layers.append(GCN\_Layer(self.args.embedSize, self.args.embedSize))
39. self.graph = self.buildGraph()*#构建邻接矩阵*
40. self.transForm = nn.Linear(in\_features=self.args.embedSize \* (self.args.gcn\_layers + 1),
41. out\_features=self.args.embedSize)
42. *#构建单位矩阵I*
43. def getSelfLoop(self, num):
44. i = torch.LongTensor(
45. [[k for k in range(0, num)], [j for j in range(0, num)]])
46. val = torch.FloatTensor([1] \* num)
47. return torch.sparse.FloatTensor(i, val).to(self.device)
48. *#构建拉普拉斯邻接矩阵L*
49. def buildGraph(self):
50. *#构建邻接矩阵L*
51. rating=self.rating.values
52. *#根据数据集构造R矩阵*
53. graph=sp.coo\_matrix((rating[:,2],(rating[:,0],rating[:,1])),shape=(self.num\_user,self.num\_item)).tocsr()*#给特征按列标上坐标，即构建R*
54. *#sp.csr\_matrix((n,m))构建n\*m的0矩阵*
55. graph=sp.bmat([[sp.csr\_matrix((graph.shape[0],graph.shape[0])),graph],[graph.T,sp.csr\_matrix((graph.shape[1],graph.shape[1]))]])*#构建A：0，R；R.T，0*
56. *#拉普拉斯变换，一个[]里面相当于一列*
57. row\_sum\_sqrt= sp.diags(1/(np.sqrt(graph.sum(axis=1).A.ravel())+1e-8))*#按列求和得行*
58. col\_sum\_sqrt = sp.diags(1 / (np.sqrt(graph.sum(axis=0).A.ravel()) + 1e-8))*#按行求和得列*
59. *# @表示常规的数学上定义的矩阵相乘；*
60. graph=row\_sum\_sqrt@graph@col\_sum\_sqrt
61. *#使用torch的稀疏张量表示*
62. graph=graph.tocoo()
63. values=graph.data
64. indices=np.vstack((graph.row,graph.col))
65. graph=torch.sparse.FloatTensor(torch.LongTensor(indices),torch.FloatTensor(values),torch.Size(graph.shape))
66. return graph.to(self.device)
67. *#获取模型特征*
68. def getFeature(self):
69. *# 根据用户特征获取对应的embedding*
70. user\_id = self.user\_id\_embedding(torch.tensor(self.user\_feature['id']).to(self.device))
71. age = self.user\_age\_embedding(torch.tensor(self.user\_feature['age']).to(self.device))
72. gender = self.user\_gender\_embedding(torch.tensor(self.user\_feature['gender']).to(self.device))
73. occupation = self.user\_occupation\_embedding(torch.tensor(self.user\_feature['occupation']).to(self.device))
74. location = self.user\_location\_embedding(torch.tensor(self.user\_feature['location']).to(self.device))
75. user\_emb = torch.cat((user\_id, age, gender, occupation, location), dim=1)
76. *# 根据天气特征获取对应的embedding*
77. item\_id = self.item\_id\_embedding(torch.tensor(self.item\_feature['id']).to(self.device))
78. item\_type = self.item\_type\_embedding(torch.tensor(self.item\_feature['type']).to(self.device))
79. temperature = self.item\_temperature\_embedding(torch.tensor(self.item\_feature['temperature']).to(self.device))
80. humidity = self.item\_humidity\_embedding(torch.tensor(self.item\_feature['humidity']).to(self.device))
81. windSpeed = self.item\_windSpeed\_embedding(torch.tensor(self.item\_feature['windSpeed']).to(self.device))
82. item\_emb = torch.cat((item\_id, item\_type, temperature, humidity, windSpeed), dim=1)
83. *# 拼接到一起*
84. concat\_emb = torch.cat([user\_emb, item\_emb], dim=0)*#按行合并*
85. return concat\_emb.to(self.device)
86. *#模型预测*
87. def forward(self, users, items):
88. features=self.getFeature()
89. final\_emb=features.clone()
90. for GCN\_Layer in self.GCN\_Layers:
91. features=GCN\_Layer(self.graph,self.selfLoop,features)
92. final\_emb=torch.cat((final\_emb,features.clone()),dim=1)*#按列合并，前900行是用户特征样本，后1600行是物品特征样本，然后列是特征名称*
93. user\_emb,item\_emb=torch.split(final\_emb,[self.num\_user,self.num\_item])*#分离用户特征与物品特征*
94. user\_emb=user\_emb[users]*#取批处理大小*
95. item\_emb=item\_emb[items]*#取批处理大小*
96. user\_emb=self.transForm(user\_emb)*#最后一层线性层转换*
97. item\_emb=self.transForm(item\_emb)*#最后一层线性层转换*
98. prediction=torch.mul(user\_emb,item\_emb).sum(1)*#模型预测，按列求和*
99. return prediction

3.定义数据加载类

1. class MyDataset(Dataset):
2. def \_\_init\_\_(self, rating):
3. super(Dataset, self).\_\_init\_\_()
4. self.user = rating['user\_id']
5. self.weather = rating['item\_id']
6. self.rating = rating['rating']
7. def \_\_len\_\_(self):
8. return len(self.rating)
9. def \_\_getitem\_\_(self, item):
10. return self.user[item], self.weather[item], self.rating[item]

4.参数初始化

1. fix\_seed\_torch(seed=2021)
2. *# 设置训练的超参数*
3. parser = argparse.ArgumentParser()
4. parser.add\_argument('--gcn\_layers', type=int, default=2, help='the number of gcn layers')*#gcn层数，默认为两层*
5. parser.add\_argument('--n\_epochs', type=int, default=30, help='the number of epochs')*#训练论数*
6. parser.add\_argument('--embedSize', type=int, default=64, help='dimension of user and entity embeddings')*#用户与物品项维度*
7. parser.add\_argument('--batch\_size', type=int, default=1024, help='batch size')*#批处理大小*
8. parser.add\_argument('--lr', type=float, default=0.001, help='learning rate')*#学习率*
9. parser.add\_argument('--ratio', type=float, default=0.8, help='size of training dataset')*#训练集占比为0.8*
10. args = parser.parse\_args()
11. *# 设备是否支持cuda*
12. device = torch.device("cuda") if torch.cuda.is\_available() else torch.device("cpu")
13. args.device = device

5.构建数据集

1. *# 读取用户特征、天气特征、评分*
2. user\_feature = pd.read\_csv('./data/user.txt', encoding='utf-8', sep='\t')
3. item\_feature = pd.read\_csv('./data/weather.txt', encoding='utf-8', sep='\t')
4. rating = pd.read\_csv('./data/rating.txt', encoding='utf-8', sep='\t')*#评价，也就是关联矩阵*
5. *# 构建数据集*
6. dataset = MyDataset(rating)
7. trainLen = int(args.ratio \* len(dataset))
8. train, test = random\_split(dataset, [trainLen, len(dataset) - trainLen])
9. train\_loader = DataLoader(train, batch\_size=args.batch\_size, shuffle=True, pin\_memory=True)*#转换成loader*
10. test\_loader = DataLoader(test, batch\_size=len(test))*#转换成loader*

6.定义模型、损失函数、参数优化器

1. *# 定义模型*
2. model = GCN(args, user\_feature, item\_feature, rating)*#定义模型*
3. model.to(device)
4. *# 定义优化器*
5. optimizer = Adam(model.parameters(), lr=args.lr, weight\_decay=0.001)
6. *# 定义损失函数*
7. loss\_function = MSELoss()
8. train\_result = []
9. test\_result = []
10. *# 最好的epoch*
11. best\_loss = sys.float\_info.max

7.模型训练与评估

1. *# 训练*
2. for i in range(args.n\_epochs):
3. *#模型训练*
4. model.train()
5. for batch in train\_loader:
6. *#梯度归0*
7. optimizer.zero\_grad()
8. *#模型预测*
9. prediction=model(batch[0].to(device),batch[1].to(device))
10. *#获取损失*
11. train\_loss=torch.sqrt(loss\_function(batch[2].float().to(device),prediction))
12. *#计算梯度*
13. train\_loss.backward()
14. *#参数优化*
15. optimizer.step()
16. train\_result.append(train\_loss.item())
17. *#模型测试*
18. model.eval()
19. for data in test\_loader:
20. *#模型评估*
21. prediction=model(data[0].to(device),data[1].to(device))
22. *#测试损失*
23. test\_loss = torch.sqrt(loss\_function(data[2].float().to(device), prediction))
24. test\_loss=test\_loss.item()
25. *#记录最好的结果*
26. if best\_loss>test\_loss:
27. best\_loss=test\_loss
28. torch.save(model.state\_dict(),'./model/bestModelParms-{}.pth'.format(start\_time))
29. test\_result.append(test\_loss)
30. logger.info("Epoch {:d}: trainLoss {:.4f},testLoss {:.4f}".format(i,train\_loss,test\_loss))

8.结果获取与绘图

1. best\_epoch, RMSE = sorted(list(enumerate(train\_result)), key=lambda x: x[1])[0]
2. logger.info("Epoch {:d}: bestTrainLoss {:.4f}".format(best\_epoch, RMSE))
3. best\_epoch, RMSE = sorted(list(enumerate(test\_result)), key=lambda x: x[1])[0]
4. logger.info("Epoch {:d}: bestTestLoss {:.4f}".format(best\_epoch, RMSE))
5. *# 画图*
6. draw\_loss\_pic(train\_result, test\_result)

9. 定义工具类函数

1. *#绘制损失函数变化图像*
2. def draw\_loss\_pic(train\_loss, test\_loss):
3. plt.plot([i for i in range(len(train\_loss))], [i for i in train\_loss], 'b-', label="Train Loss")
4. plt.plot([i for i in range(len(test\_loss))], [i for i in test\_loss], 'r-', label="Test Loss")
5. plt.xlabel('epoch')
6. plt.ylabel('RMSE')
7. plt.legend()
8. plt.show()
9. *#固定随机种子*
10. def fix\_seed\_torch(seed=2021):
11. random.seed(seed)
12. os.environ['PYTHONHASHSEED'] = str(seed)
13. np.random.seed(seed)
14. torch.manual\_seed(seed)
15. torch.cuda.manual\_seed(seed)
16. torch.cuda.manual\_seed\_all(seed)  *# if you are using multi-GPU.*
17. torch.backends.cudnn.benchmark = False
18. torch.backends.cudnn.deterministic = True

10.结果分析

TABLE I

运行参数与结果参数

|  |  |
| --- | --- |
| 批处理大小 | batch\_size = 1024 |
| 设备 | device = cpu |
| 特征维度 | embedSize = 64 |
| GCN层数 | gcn\_layers=2 |
| 学习率 | lr= 0.001 |
| 训练轮数 | n\_epochs=30 |
| 训练数据集占比 | ratio=0.8 |
| 最好训练损失 | bestTrainLoss=0.8655 |
| 最好评估损失 | bestTestLoss=0.9277 |

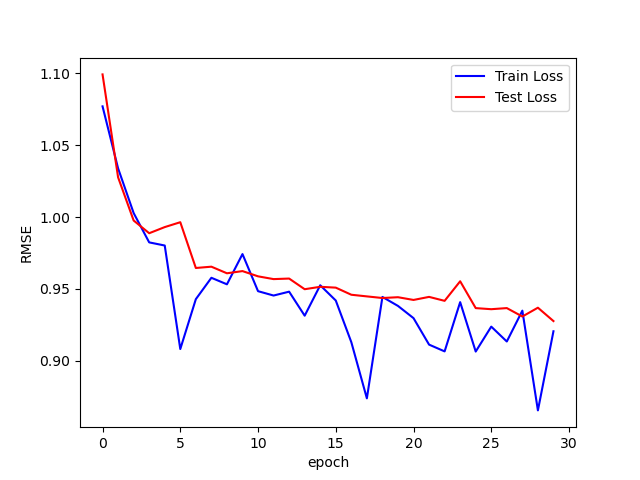


Figure. 1. 训练损失与评估损失随训练轮数变化图

## **结论**

由实验结果可知，使用GCN模型对数据进行训练时，随着迭代次数增加，模型的预测损失整体越来越小，预测准确率整体越来越高，但训练集的损失以的波动较大，而测试集的训练损失波动较小，整体而言GCN模型在天气系统的推荐中的应用是较为合适的。

在本次实验中，主要是实现了基于GCN的推荐算法。

在GCN推荐算法中，其主要步骤是先导入数据，提取用户特征和物品特征，但在GCN中，因为其主要面向互相具有依赖关系的数据集，因此其还需提取一个依赖矩阵，即用户-物品交互矩阵，也就是邻接矩阵，保存着数据之间的依赖关系，然后经过一系列预处理以及拉普拉斯变换之后，划分训练集与验证集和测试集，最终确定模型所需要的数据，然后定义GCN模型，在本次实验中GCN模型主要由两个GCN层组成，单个GCN层由两个线性层以及一个LeakyReLU激活层组成，其经过两个线性层的相加进入LeakyReLU层，获取单层的预测，在整体GCN模型中通过经过多层GCN正向传播，然后将多个特征拼接矩阵经过一层线性层后最终转换成单个特征矩阵，该矩阵即为交互预测值，然后定义MSELOSS损失函数，然后使用Adam优化算法进行训练，进行参数优化，最后训练结果输出就完成了。

对于本次实验而言，主要有代码参考，与GCN分类算法相比，GCN推荐算法还是有所不同的，其单层结构比GCN分类算法更为复杂，但两者都需要提取依赖矩阵。虽然在公式上比较好理解，但更多是了解如何去进行代码实现，这与在课堂上仅学习到的公式是相差很大的。此外还学习到了许多机器学习的库，让自己对这些库中的函数有了更进一步的了解与运用。总体而言本次实验的收获是挺大的。

## **拓展实验**

1研究超参数，例如图卷积层数对模型的影响。

batch\_size：影响着每一轮训练样本数量。

device：影响训练所需要的显卡类型。

embedSize：影响着用户特征与物品特征能否匹配。

GCN层数：一般而言，GCN层数越高，训练效果越好，但如果GCN层数过高，容易引起过拟合现象。

lr：影响程序学习的快慢。

n\_epochs：影响着最终训练的效果。

ratio：影响着训练与评估效果。

1.1 GCN层数对模型的影响

下面将GCN层分别从2,3,4,5层进行测试：

TABLE II

运行参数与结果参数

|  |  |
| --- | --- |
| GCN层数 | gcn\_layers=2 |
| 最好训练损失 | bestTrainLoss=0.8655 |
| 最好评估损失 | bestTestLoss=0.9277 |

TABLE III

运行参数与结果参数

|  |  |
| --- | --- |
| GCN层数 | gcn\_layers=3 |
| 最好训练损失 | bestTrainLoss= 0.8853 |
| 最好评估损失 | bestTestLoss= 0.9207 |

TABLE IV

运行参数与结果参数

|  |  |
| --- | --- |
| GCN层数 | gcn\_layers=4 |
| 最好训练损失 | bestTrainLoss= 0.8743 |
| 最好评估损失 | bestTestLoss= 0.9206 |

TABLE V

运行参数与结果参数

|  |  |
| --- | --- |
| GCN层数 | gcn\_layers=5 |
| 最好训练损失 | bestTrainLoss= 0.8762 |
| 最好评估损失 | bestTestLoss= 0.9187 |

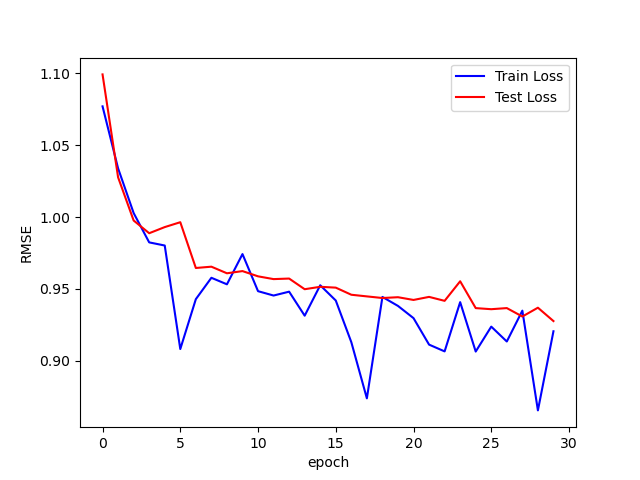


Figure. 2. gcn\_layers=2时训练损失与评估损失随训练轮数变化图

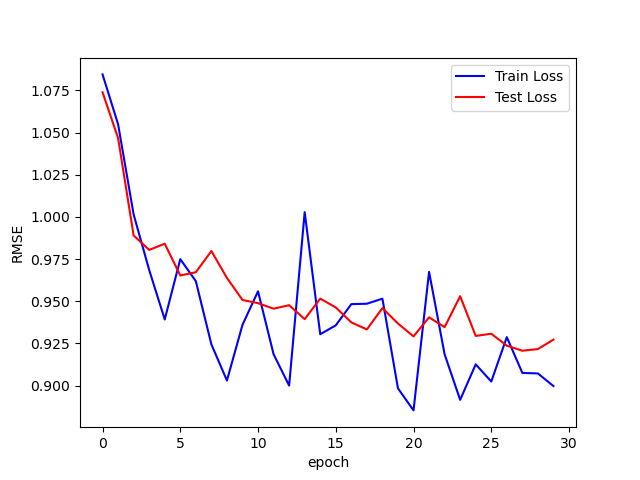


Figure. 3. gcn\_layers=3时训练损失与评估损失随训练轮数变化图

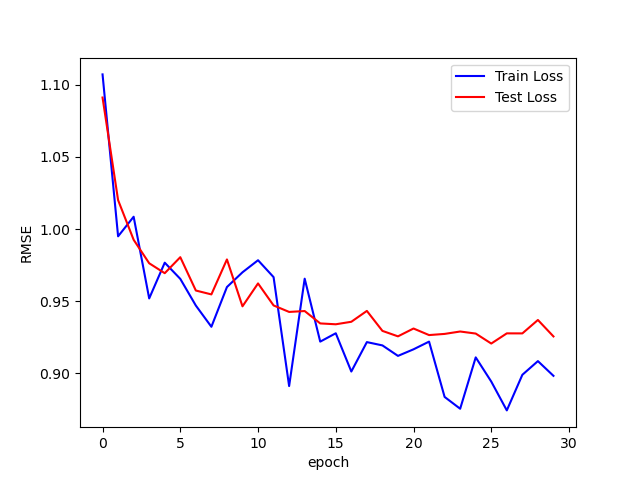


Figure. 4. gcn\_layers=4时训练损失与评估损失随训练轮数变化图

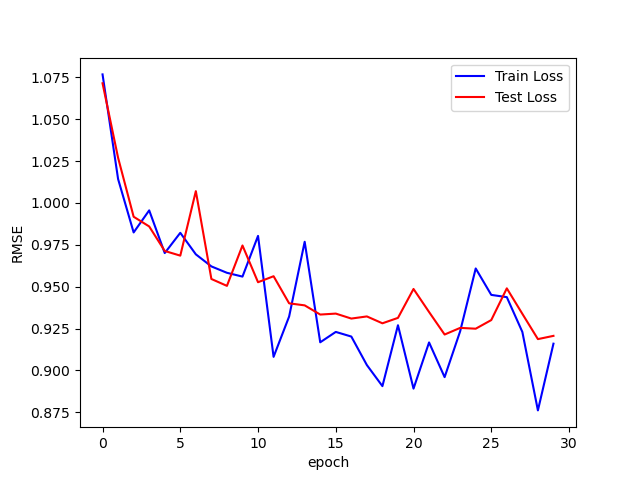


Figure. 5. gcn\_layers=5时训练损失与评估损失随训练轮数变化图

1.2学习率对模型的影响

下面将lr分别从0.001,0.01,0.1, 0.0001进行测试：

TABLE VI

运行参数与结果参数

|  |  |
| --- | --- |
| 学习率 | lr= 0.001 |
| 最好训练损失 | bestTrainLoss=0.8655 |
| 最好评估损失 | bestTestLoss=0.9277 |

TABLE VII

运行参数与结果参数

|  |  |
| --- | --- |
| 学习率 | lr= 0.01 |
| 最好训练损失 | bestTrainLoss= 0.8651 |
| 最好评估损失 | bestTestLoss= 0.9135 |

TABLE VIII

运行参数与结果参数

|  |  |
| --- | --- |
| 学习率 | lr= 0.1 |
| 最好训练损失 | bestTrainLoss= 2.2860 |
| 最好评估损失 | bestTestLoss= 2.2271 |

TABLE IX

运行参数与结果参数

|  |  |
| --- | --- |
| 学习率 | lr= 0.0001 |
| 最好训练损失 | bestTrainLoss= 0.9056 |
| 最好评估损失 | bestTestLoss= 0.9742 |

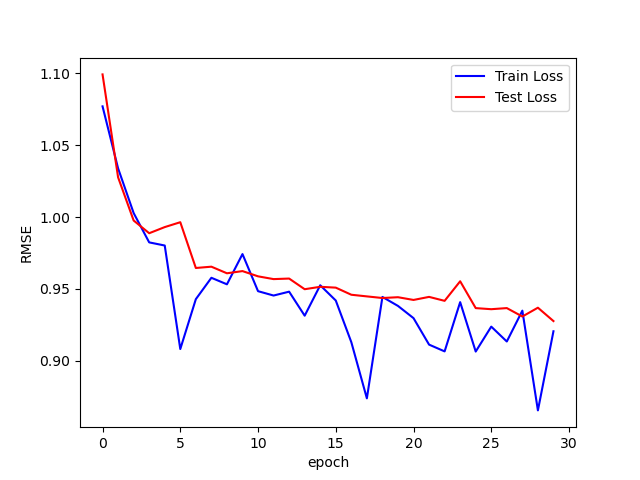


Figure. 6. lr=0.001时训练损失与评估损失随训练轮数变化图

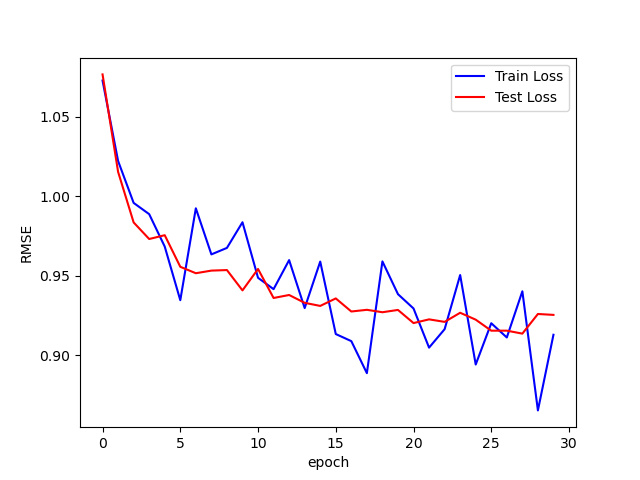


Figure. 7. lr=0.01时训练损失与评估损失随训练轮数变化图

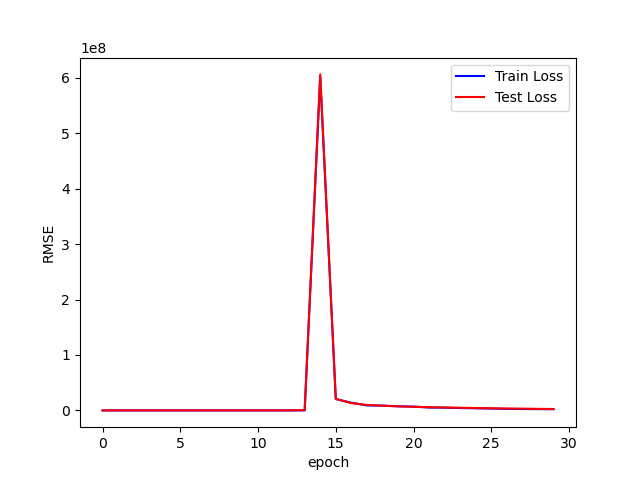


Figure. 8. lr=0. 1时训练损失与评估损失随训练轮数变化图

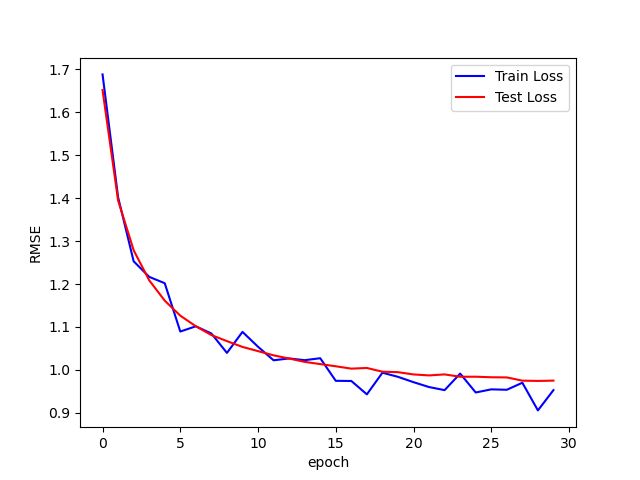


Figure. 9. lr=0. 0001时训练损失与评估损失随训练轮数变化图

1.3实验结论

对于GCN层数，GCN层数越高，其拟合效果越好，但如果GCN层数过高，容易造成模型的过拟合，鲁棒性较差，会导致评估误差变大。

对于学习率，学习率越低时，模型的拟合误差波动会越小，但不容易达到局部最优结果，当学习率越高时，模型的拟合误差波动越大，但可以快速达到局部最优处，但如果学习率过高，会导致无法进行参数优化，例如lr=0.1时，测试误差能达到非常高的数量级，因此学习率的设置需要适中，由实验可知lr=0.001时效果最好。

2我们的模型把所有GCN层的输出拼接在一起进行预测，你可以试试取平均、或者按权重相加吗？

2.1平均相加

2.1.1实现

具体实现位于model2.py

2.1.2实验结果

TABLE X

运行参数与结果参数

|  |  |
| --- | --- |
| 批处理大小 | batch\_size = 1024 |
| 设备 | device = cpu |
| 特征维度 | embedSize = 64 |
| GCN层数 | gcn\_layers=2 |
| 学习率 | lr= 0.001 |
| 训练轮数 | n\_epochs=30 |
| 训练数据集占比 | ratio=0.8 |
| 最好训练损失 | bestTrainLoss= 0.8960 |
| 最好评估损失 | bestTestLoss= 0.9281 |

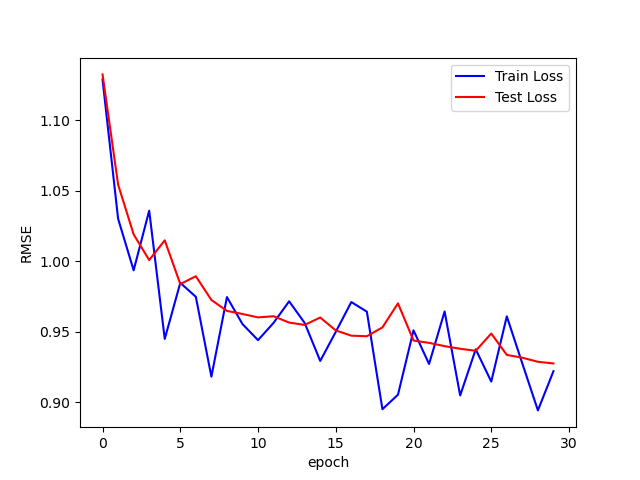


Figure. 10.平均相加时训练损失与评估损失随训练轮数变化图

2.1.3实验结论

当采用平均相加方法时，模型的训练效果与评估效果均比原先模型的差，训练误差波动也十分大。

2.2权重相加

2.2.1实现（权重由随机生成）

具体实现位于model4.py

2.2.2实验结果

TABLE XI

运行参数与结果参数

|  |  |
| --- | --- |
| 批处理大小 | batch\_size = 1024 |
| 设备 | device = cpu |
| 特征维度 | embedSize = 64 |
| GCN层数 | gcn\_layers=2 |
| 学习率 | lr= 0.001 |
| 分配权重 | [0.3052, -0.0486, 1.0106] |
| 训练轮数 | n\_epochs=30 |
| 训练数据集占比 | ratio=0.8 |
| 最好训练损失 | bestTrainLoss= 0.8890 |
| 最好评估损失 | bestTestLoss= 0.9280 |

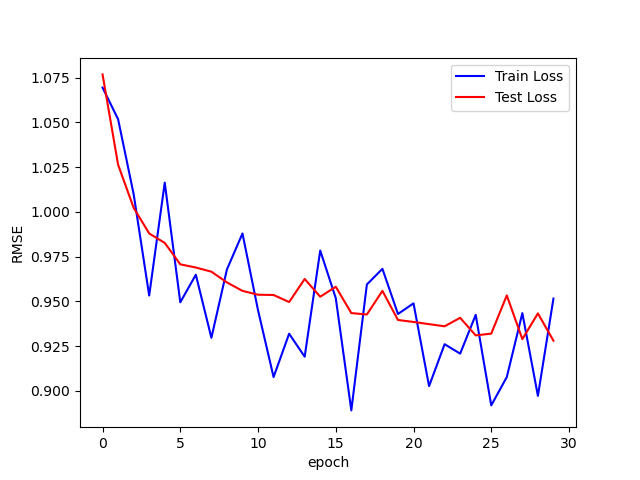


Figure. 11.权重相加时训练损失与评估损失随训练轮数变化图

2.2.3实验结论

当采用加权相加方法时，模型的训练效果与评估效果均比原先模型的略差，但比平均相加方法好，然而波动比两个模型都要大，其拟合效果与权重系数相关性较大，因此可以考虑将此权重作为训练参数的一部分进行训练，从而使得模型训练效果达到最优。

3 LightGCN（SIGIR 2021）指出，去掉自循环、W矩阵和非线性激活函数能提高模型的效果，你能动手试试吗？

3.1实现

具体实现位于model3.py

3.2实验结果

TABLE XII

运行参数与结果参数

|  |  |
| --- | --- |
| 批处理大小 | batch\_size = 1024 |
| 设备 | device = cpu |
| 特征维度 | embedSize = 64 |
| GCN层数 | gcn\_layers=2 |
| 学习率 | lr= 0.001 |
| 训练轮数 | n\_epochs=30 |
| 训练数据集占比 | ratio=0.8 |
| 最好训练损失 | bestTrainLoss= 0.8836 |
| 最好评估损失 | bestTestLoss= 0.9351 |

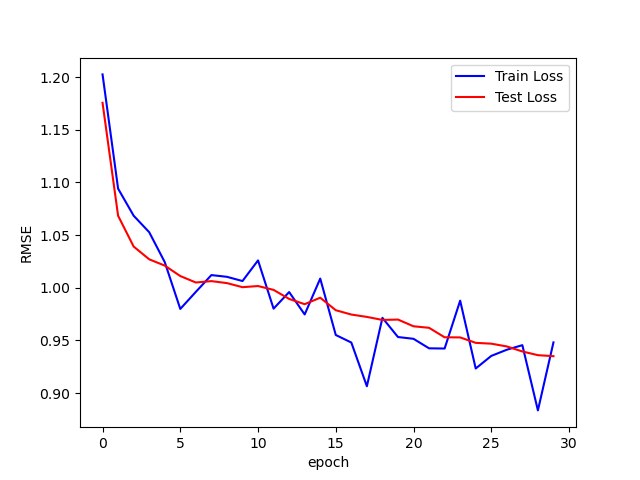


Figure. 12. LightGCN模型训练损失与评估损失随训练轮数变化图

3.3实验结论

通过建立LightGCN模型，模型训练之后，可得最好训练损失为0.8836，最好评估损失为0.9351，与原先模型相比，原先模型的拟合效果更优，因此在本实验中LightGCN并不能提高模型的效果。