Friend Recommendation Project

一、Project要求

Graph：social network
　Nodes：user
　Edges：potentially heterogeneous -- friend, follow, reply to, message, like, etc.
Tasks：
- Recommending/ranking new friends for user -- metrics: Hits@K, NDCG@K, MRR

Example model(s)：GraFRank
Datasets:Facebook, Google+, Twitter

二、Project说明概要

1、实验模型的选择：GraFRank（论文2021Graph Neural Networks for Friend Ranking in Large-scale Social Platforms）

2、数据集的选择：斯坦福ego-facebook

3、评价指标：Hits@K，NDCG@K，MRR

4、资料参考：GraFRank论文、Github开源代码

5、实验环境：PyCharm Community Edition 2022.2.2

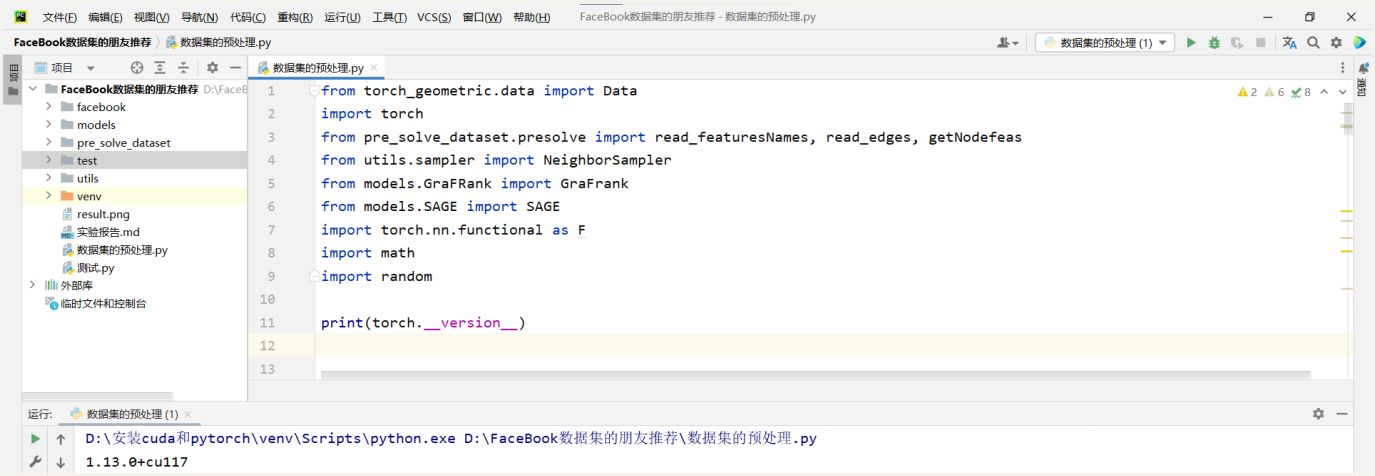
三、Project具体步骤

(一)环境配置

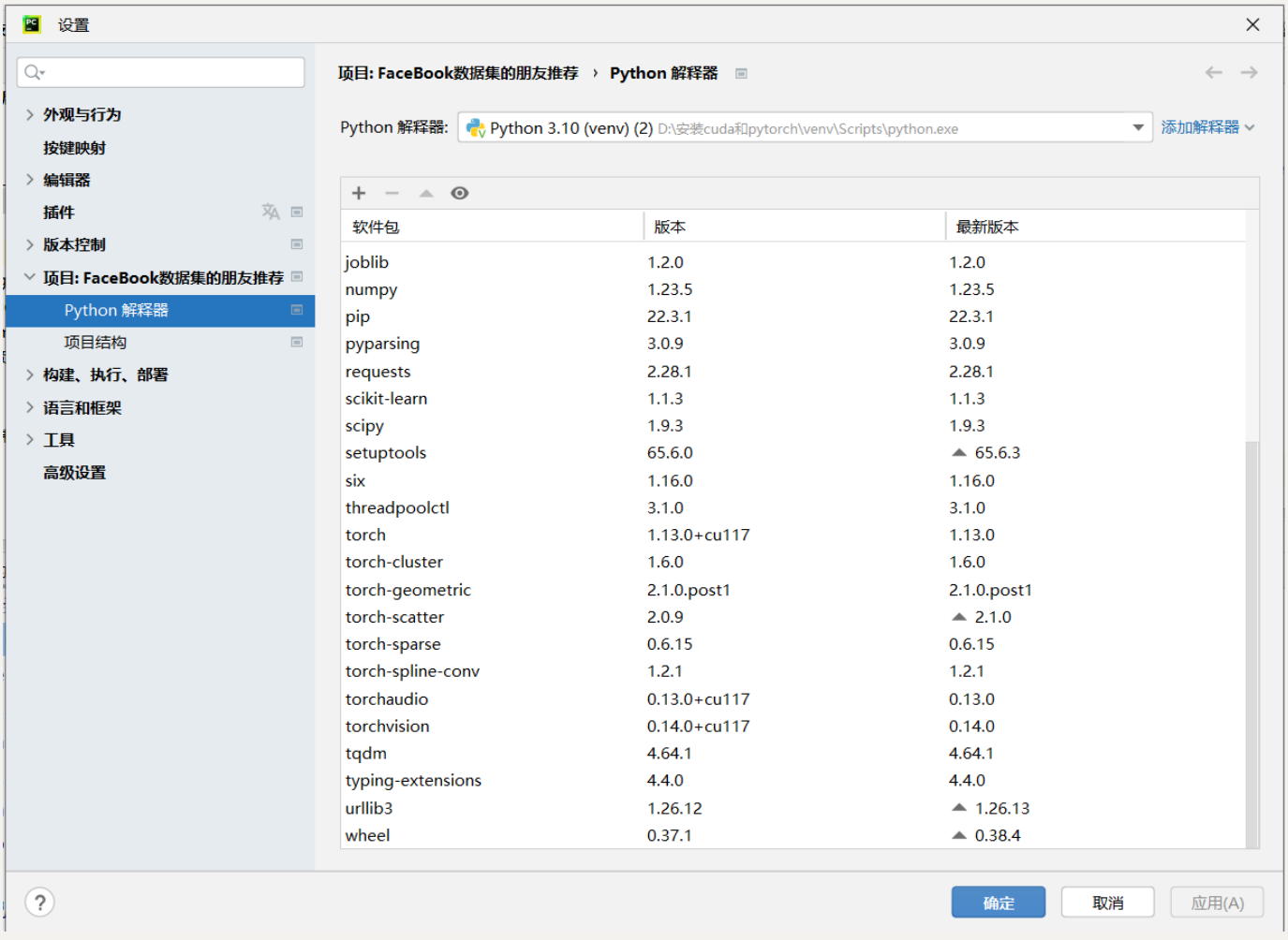
1、**IDE**：PyCharm Community Edition 2022.2.2

2、python的版本：3.10.0

3、torch和cuda版本：1.13.0+cu117



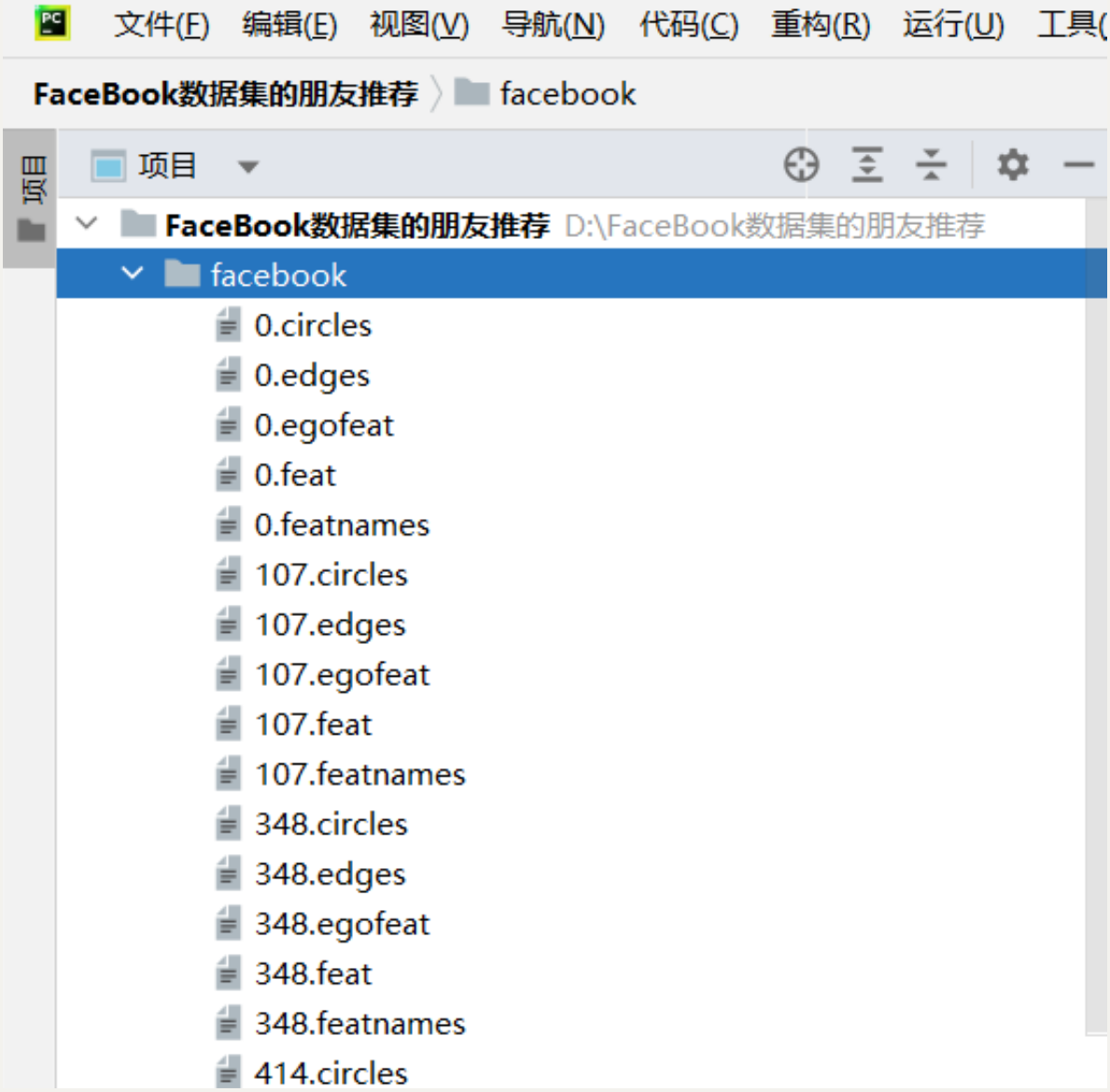
4、Pytorch-geometric的版本：通过官网然后在pycharm命令行pip下载。因为直接从python解释器添加会产生无法解析torch-sparse的问题。

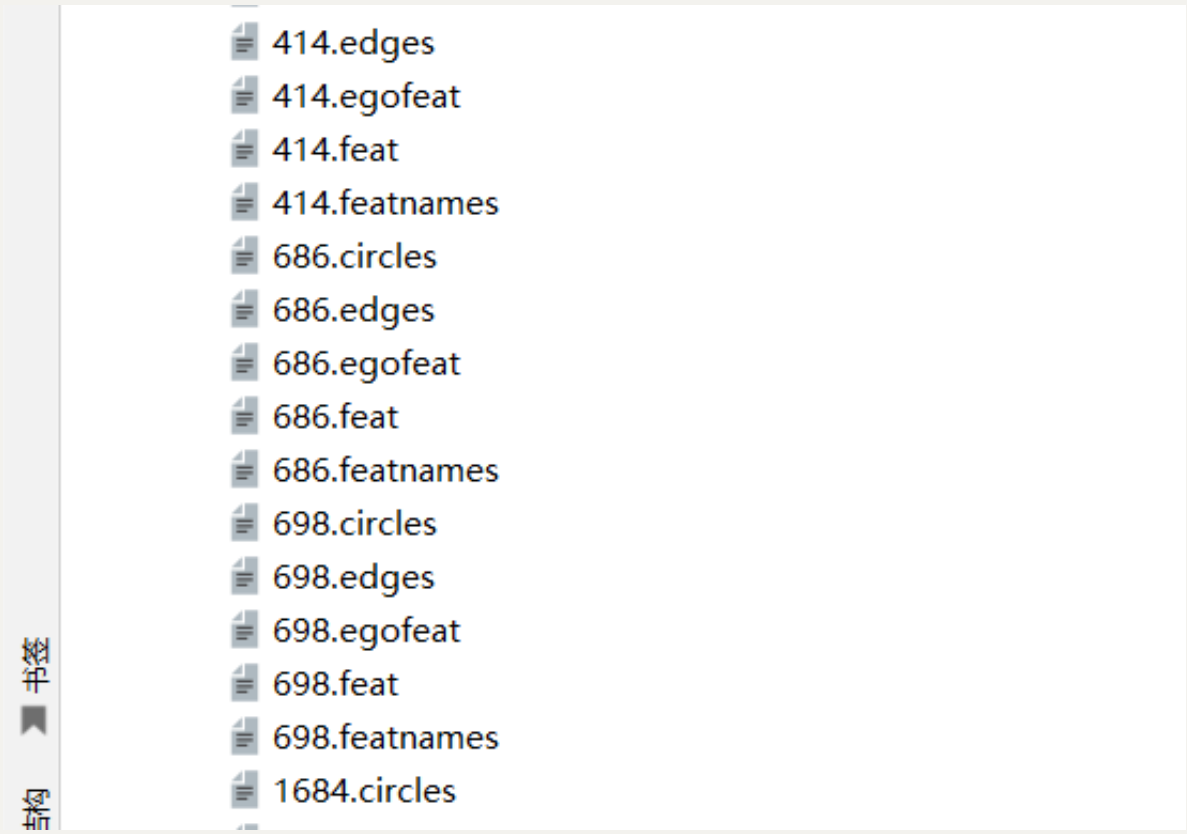


(二)数据集的预处理

1、原始数据集一览：ego-facebook。

Dataset statistics	
Nodes	4039
Edges	88234
Nodes in largest WCC	4039 (1.000)
Edges in largest WCC	88234 (1.000)
Nodes in largest SCC	4039 (1.000)
Edges in largest SCC	88234 (1.000)
Average clustering coefficient	0.6055
Number of triangles	1612010
Fraction of closed triangles	0.2647
Diameter (longest shortest path)	8
90-percentile effective diameter	4.7





id.circles代表是以id用户为中心的一个社交圈。
id.edges代表这个圈子内的无向边。
id.featnames代表id中心社交圈涉及的全部用户特征名。
id.feat代表id社交圈内用户具体的特征信息，以0代表不具有特征，1代表具有特征。
id.egofeat代表id自己的特征信息。
2、数据集处理：
（1）特征名称的统计：
由于每个社交圈涉及的特征信息有交叉也有不同，最后要得到具体的每个用户-特征矩阵，则需要统计特征的个数，也即矩阵的列数。

函数**read_featuresNames()**负责批量读取每个社交圈的特征属性（id.featnames）最后得到不重复的特征集合并返回。

```python
#指定读取文件夹和文件格式批量读取featnames并统一featnames
def read_featuresNames(path, suffix):
    files = os.listdir(path)
    result = []
    for file in files:
        pos = path + "\\" + file
        if os.path.splitext(file)[1] != suffix:
            continue
        f = open(pos, 'r')
        f_data = f.readlines()
        for row in f_data:
            tmp_list = row.split(' ')
            str_fea = tmp_list[1] + tmp_list[2] + tmp_list[3]
            result.append(str_fea)
    return list(set(result))
```

可以得出一共有**1406**个特征。

（2）边信息的读取：

由于是无向边，因此这里正向反向都要存放一次，这是后面Pytorch-geometric要使用的data类来存放一张图时所要求的。

函数**read_edges()**负责批量读取每个id.edges文件并存放进元组列表。

```python
# 读取边
def read_edges(path, suffix):
    files = os.listdir(path)
    result = []
    for file in files:
        pos = path + "\\" + file
        if os.path.splitext(file)[1] != suffix:
            continue
        f = open(pos, 'r')
        f_data = f.readlines()
        for row in f_data:
            tmp_list = row.split(' ')
            result.append((int(tmp_list[0]), int(tmp_list[1])))
            result.append((int(tmp_list[1]), int(tmp_list[0])))
    return list(set(result))
```

（3）读取一共从标号0~4038共4039个用户的特征信息。
这里要读取每个社交圈进行读取特征信息，然后将信息填入用户-特征矩阵。

函数getNodesfeas()负责得到shape=[4039, 1406]的用户特征矩阵信息(01矩阵)。主要解决的问题是如何将featname与用户特征矩阵的列对应起来的问题，找到对应关系后就可以填入信息。

```
def getNodefeas(feas):
    nodes_fea = torch.zeros([4039, 1406], dtype=torch.float32)
    circles_list = [0, 107, 348, 414, 686, 698, 1684, 1912, 3437, 3980]
    for ego in circles_list:
        path1 = './facebook/' + str(ego) + '.featnames'
        f1 = open(path1, 'r')
        fea_list = []
        fea_list.clear()
        f_data1 = f1.readlines()
        for row in f_data1:
            tmp_list = row.split(' ')
            y = feas.index(tmp_list[1] + tmp_list[2] + tmp_list[3])
            fea_list.append(y)

        path2 = './facebook/' + str(ego) + '.egofeat'
        f2 = open(path2, 'r')
        f_data2 = f2.readlines()
        for row in f_data2:
            tmp_list = row.split(' ')
            for i in range(0, len(tmp_list)):
                nodes_fea[ego][fea_list[i]] = int(tmp_list[i])

        path3 = './facebook/' + str(ego) + '.feat'
        f3 = open(path3, 'r')
        f_data3 = f3.readlines()
        for row in f_data3:
            tmp_list = row.split(' ')
            x = int(tmp_list[0])
            for i in range(1, len(tmp_list)):
                nodes_fea[x][fea_list[i - 1]] = int(tmp_list[i])
    return nodes_fea
```

（4）随机选择95%的用户为数据集，然后剩下%5为测试集。然后将图信息存入Data类处理。
这里x为用户特征矩阵，edges.t()为data类所要求格式的邻接边关系，然后设置训练集掩码和测试集掩码。边的特征维度参考github上开源代码对Cora的处理同样设置为5。

```python
def getRandomIndex(n, x):
    index = random.sample(range(n), x)
    return index


# 获取特征名称总览
feas = read_featuresNames('./facebook', '.featnames')
# 读取边
edges = read_edges('./facebook', '.edges')
edges = torch.tensor(edges, dtype=torch.long)
# 处理点特征矩阵
nodes_fea = getNodefeas(feas)
# 随机选择95%为训练集
num_nodes = 4039
train1 = torch.ones(num_nodes, dtype=bool)
test1 = torch.zeros(num_nodes, dtype=bool)
index = getRandomIndex(num_nodes, int(0.05 * num_nodes))
for i in range(0, len(index)):
    train1[i] = False
for i in range(0, len(index)):
    test1[index] = True
data = Data(x=nodes_fea, edge_index=edges.t(), train_mask=train1, test_mask=test1)
# 边特征维数
n_edge_channels = 5
# 边属性
data.edge_attr = torch.ones([data.edge_index.shape[1], n_edge_channels])
```

(三)GraFRank模型

1、原理：

不同于简单的SOTA模型，这篇论文基于Snapchat的数据，他们观察到了两个现象。

(1)一个是heterogeneity in modality homophily，即用户在不同的模态特征中表现出的同质化程度不同，因此建模对四个模态每个单独学习了一个attention，而非将它们简单的concat成一个向量统一学习。

(2)模态特征的组合也能产生有效信息，即cross-modality correlations。

于是在设计GraFRank时就着重添加了两个模块：modality-specific neighbor aggregation用于在聚合邻居信息时对不同模态分别聚合邻居信息；cross-modality attention layer用于在组合模态特征时增加attention机制。

2、示意图：

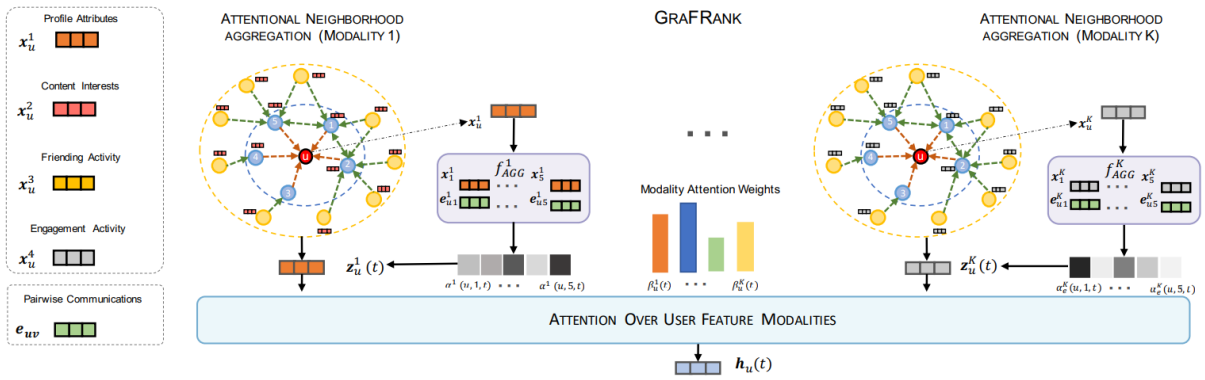


Figure 5: Overall framework of GRAFRANK: a set of K modality-specific neighbor aggregators (parameterized by individual modality-specific user features and link communication features) to compute K intermediate user representations; cross-modality attention layer to compute final user representations by capturing discriminative facets of each modality.

3、源代码:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from models.layers import GraFrankConv, CrossModalityAttention


class GraFrank(nn.Module):
    """
    GraFrank Model for Multi-Faceted Friend Ranking with multi-modal node features and pairwise link features.
    (a) Modality-specific neighbor aggregation: modality_convs
    (b) Cross-modality fusion layer: cross_modality_attention
    """

    def __init__(self, in_channels, hidden_channels, edge_channels, num_layers, input_dim_list):
        """
        :param in_channels: total cardinality of node features.
        :param hidden_channels: latent embedding dimensionality.
        :param edge_channels: number of link features.
        :param num_layers: number of message passing layers.
        :param input_dim_list: list containing the cardinality of node features per modality.
        """
        super(GraFrank, self).__init__()
        self.num_layers = num_layers
```

```
        self.modality_convs = nn.ModuleList()
        self.edge_channels = edge_channels
        # we assume that the input features are first partitioned and
then concatenated across the K modalities.
        self.input_dim_list = input_dim_list

        for inp_dim in self.input_dim_list:
            modality_conv_list = nn.ModuleList()
            for i in range(num_layers):
                in_channels = in_channels if i == 0 else
hidden_channels
                modality_conv_list.append(GraFrankConv((inp_dim +
edge_channels, inp_dim), hidden_channels))

            self.modality_convs.append(modality_conv_list)

        self.cross_modality_attention =
CrossModalityAttention(hidden_channels)

    def forward(self, x, adjs, edge_attrs):
        """ Compute node embeddings by recursive message passing,
followed by cross-modality fusion.
        :param x: node features [B', in_channels] where B' is the
number of nodes (and neighbors) in the mini-batch.
        :param adjs: list of sampled edge indices per layer
(EdgeIndex format in PyTorch Geometric) in the mini-batch.
        :param edge_attrs: [E', edge_channels] where E' is the number
of sampled edge indices per layer in the mini-batch.
        :return: node embeddings. [B, hidden_channels] where B is the
number of target nodes in the mini-batch.
        """
        result = []
        for k, convs_k in enumerate(self.modality_convs):
            emb_k = None
            for i, ((edge_index, _, size), edge_attr) in
enumerate(zip(adjs, edge_attrs)):
                x_target = x[:size[1]]  # Target nodes are always
placed first.
```

```
                x_list = torch.split(x,
split_size_or_sections=self.input_dim_list, dim=-1)  # modality
partition
                x_target_list = torch.split(x_target,
split_size_or_sections=self.input_dim_list, dim=-1)
                x_k, x_target_k = x_list[k], x_target_list[k]

                emb_k = convs_k[i]((x_k, x_target_k), edge_index,
edge_attr=edge_attr)

                if i != self.num_layers - 1:
                    emb_k = emb_k.relu()
                    emb_k = F.dropout(emb_k, p=0.5,
training=self.training)

            result.append(emb_k)
        return self.cross_modality_attention(result)

    def full_forward(self, x, edge_index, edge_attr):
        """ Auxiliary function to compute node embeddings for all
nodes at once for small graphs.
        :param x: node features [N, in_channels] where N is the total
number of nodes in the graph.
        :param edge_index: edge indices [2, E] where E is the total
number of edges in the graph.
        :param edge_attr: link features [E, edge_channels] across all
edges in the graph.
        :return: node embeddings. [N, hidden_channels] for all nodes
in the graph.
        """
        x_list = torch.split(x,
split_size_or_sections=self.input_dim_list, dim=-1)  # modality
partition
        result = []
        for k, convs_k in enumerate(self.modality_convs):
            x_k = x_list[k]
            emb_k = None
            for i, conv in enumerate(convs_k):
                emb_k = conv(x_k, edge_index, edge_attr=edge_attr)
```

```
                if i != self.num_layers - 1:
                    emb_k = emb_k.relu()
                    emb_k = F.dropout(emb_k, p=0.5,
training=self.training)

            result.append(emb_k)
        return self.cross_modality_attention(result)
```

(四)GNN SAGE模型

1、原理:

GraphSAGE是一个inductive框架，在具体实现中，训练时它仅仅保留训练样本到训练样本的边。inductive learning 的优点是可以利用已知节点的信息为未知节点生成Embedding. GraphSAGE 取自 Graph SAmple and aggreGate, SAmple指如何对邻居个数进行采样。aggreGate指拿到邻居的embedding之后汇聚这些embedding以更新自己的embedding信息。

首先对邻居采样，采样后的邻居embedding传到节点上来，并使用一个聚合函数聚合这些邻居信息以更新节点的embedding，根据更新后的embedding预测节点的标签，如原理图所示。

2、示意图:

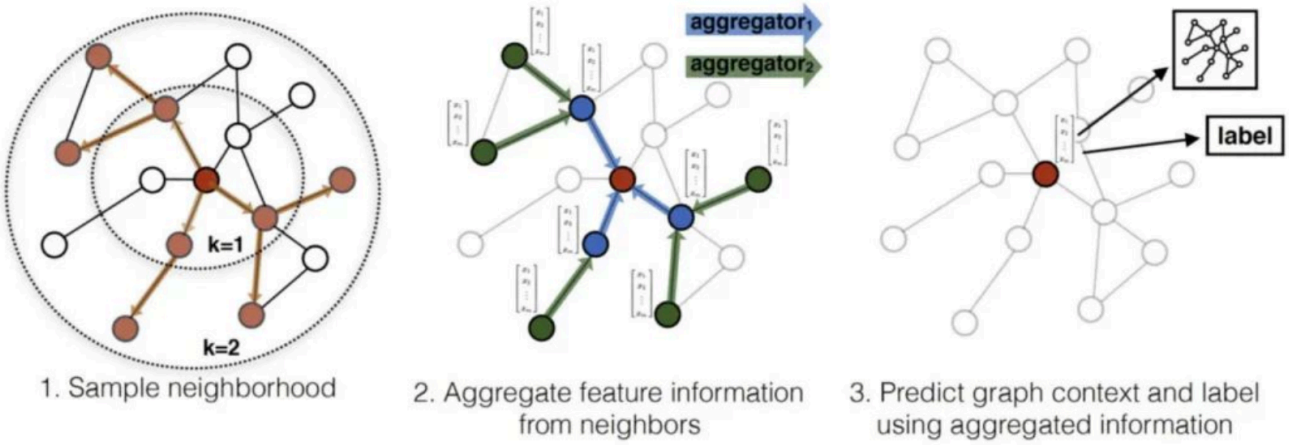


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

3、源代码:

```
def GraphSAGE(feature_dim, neighbor_num, n_hidden, n_classes,
use_bias=True, activation=tf.nn.relu,
              aggregator_type='mean', dropout_rate=0.0, l2_reg=0):

    features = Input(shape=(feature_dim,))
    node_input = Input(shape=(1,), dtype=tf.int32)
    neighbor_input = [Input(shape=(l,),dtype=tf.int32) for l in
neighbor_num]

    if aggregator_type == 'mean':
        aggregator = MeanAggregator
    else:
        aggregator = PoolingAggregator

    h = features
    for i in range(0, len(neighbor_num)):
        if i > 0:
            feature_dim = n_hidden
        if i == len(neighbor_num) - 1:
            activation = tf.nn.softmax
            n_hidden = n_classes
        h = aggregator(units=n_hidden, input_dim=feature_dim,
activation=activation, l2_reg=l2_reg, use_bias=use_bias,
                       dropout_rate=dropout_rate,
neigh_max=neighbor_num[i])(
            [h, node_input,neighbor_input[i]])#

    output = h
    input_list = [features, node_input] + neighbor_input
    model = Model(input_list, outputs=output)
    return model
```

(五)消息传递和聚合

1. 消息传递

在消息传递阶段，模型学习的是user u 在time t 得到的邻居信息 $z_u^k(t, N_u(t))$ 。对每个节点ego-net中的邻居（即一阶邻居），会通过注意力（friendship attention）来控制他对用户的影响程度（这里paper写的有点混乱，此处的邻居信息 $W_1^k x_v^{s,k}$ 等同于下文中出现的邻居信息 $m_{u\leftarrow v}^k$ ），

$$\alpha^k(u,v,t) = \text{LeakyRELU}(a_k^T(W_1^k x_u^{s,k} || W_1^k x_v^{s,k})),\ s = \mathcal{T}(t) \quad (1)$$

其中 $W_1^k \in \mathbb{R}^{D_k \times D}$ 是全量用户共用的linear transformation矩阵，通过 $||$ 将转换后的用户和邻居向量concat起来后，再用weight vector $a_k \in \mathbb{R}^{2D}$ 和LeadyReLU做single feed-forward。当然，不同邻居v的attention会通过softmax做归一化，

$$\alpha^k(u,v,t) = \frac{\exp(\alpha^k(u,v,t))}{\sum_{w\in N_t(u)} \exp(\alpha^k(u,v,t))} \quad (2)$$

那么，每个用户 u 在模态 k 中接收到的邻居信息 $z_u^k(t, N_u(t)) \in \mathbb{R}^{D_k}$ 可以通过邻居本身的信息 $m_{u\leftarrow v}^k \in \mathbb{R}^D$ （用户 v 在时间 t 传递给用户 u 的模态message）和邻居注意力加权得到，

$$z_u^k(t, N_u(t)) = \sum_{v\in N_u(t)} \alpha^k(u,v,t) m_{u\leftarrow v}^k \quad (3)$$

值得注意的是，snap对用户行为做过统计，发现大多数用户每个月只会与10%-20%的朋友发送至少以此信息，因此这个注意力机制对用户临近程度的建模非常重要。同时在邻居消息 $m_{u\leftarrow v}^k$ 上，他们也融合了边上的link communication features e_{uv}^s 作为消息，

$$m_{u\leftarrow v}^k = W_2^k x_v^{s,k} + W_e^k e_{uv}^s + b,\ s = \mathcal{T}(t) \quad (4)$$

其中， $W_2^k \in \mathbb{R}^{D_k \times D}$ ， $W_e^k \in \mathbb{R}^{E\times D}$ 和 b 是可学习权重和bias。那么，节点 u 在这一层的新embedding就可以通过聚合上一层的embedding $x_u^{s,k}$ 得到，得到了第一个公式

$$\alpha^k(u,v,t) = \text{LeakyReLU}(a_k^T(W_1^k x_u^{s,k} || m_{u\leftarrow v}^k)), s = \mathcal{T}(t) \quad (1')$$

2. 消息聚合：

在得到Eq(3)的邻居信息后，将其与上一层的自身embedding聚合，可以得到节点 u 在这一层的embedding，

$$z_u^k(t) = F_\theta^k(m_{u\leftarrow u}^k, z_u^k(t, N_u(t))) = \sigma\left(W_a^k(z_u^k(t, N_t(t)) || m_{u\leftarrow u}^k) + b_a\right)$$

其中 F_θ^k 是一个dense layer， σ 采用了ELU activation。他们在实验中也发现，这里增加的self-connections信息，也是Grafrank效果优于GCN和GAT的一个重要原因。

```
from typing import Optional
```

```
from typing import Union, Tuple
import torch

from torch import Tensor
import torch.nn as nn
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.typing import OptPairTensor, Adj, Size,
OptTensor
from torch_geometric.utils import softmax
from torch_sparse import SparseTensor
import torch.nn.functional as F


class GraFrankConv(MessagePassing):
    """
    Modality-specific neighbor aggregation in GraFrank implemented by
stacking message-passing layers that are
    parameterized by friendship attentions over individual node
features and pairwise link features.
    """

    def __init__(self, in_channels: Union[int, Tuple[int, int]],
                 out_channels: int, normalize: bool = False,
                 bias: bool = True, **kwargs):  # yapf: disable
        kwargs.setdefault('aggr', 'add')
        super(GraFrankConv, self).__init__(**kwargs)

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.normalize = normalize
        self.negative_slope = 0.2
        if isinstance(in_channels, int):
            in_channels = (in_channels, in_channels)

        self.self_linear = nn.Linear(in_channels[1], out_channels,
bias=bias)
        self.message_linear = nn.Linear(in_channels[0], out_channels,
bias=bias)
```

```
        self.attn = nn.Linear(out_channels, 1, bias=bias)
        self.attn_i = nn.Linear(out_channels, 1, bias=bias)

        self.lin_l = nn.Linear(out_channels, out_channels, bias=bias)
        self.lin_r = nn.Linear(out_channels, out_channels,
bias=False)

        self.reset_parameters()
        self.dropout = 0

    def reset_parameters(self):
        self.lin_l.reset_parameters()
        self.lin_r.reset_parameters()

    def forward(self, x: Union[Tensor, OptPairTensor], edge_index:
Adj, edge_attr: OptTensor = None,
                size: Size = None) -> Tensor:
        if isinstance(x, Tensor):
            x: OptPairTensor = (x, x)

        x_l, x_r = x[0], x[1]
        self_emb = self.self_linear(x_r)
        alpha_i = self.attn_i(self_emb)
        out = self.propagate(edge_index, x=(x_l, x_r), alpha=alpha_i,
edge_attr=edge_attr, size=size)
        out = self.lin_l(out) + self.lin_r(self_emb)  # dense layer.

        if self.normalize:
            out = F.normalize(out, p=2., dim=-1)

        return out

    def message(self, x_j: Tensor, alpha_i: Tensor, edge_attr:
Tensor, index: Tensor, ptr: OptTensor,
                size_i: Optional[int]) -> Tensor:
        message = torch.cat([x_j, edge_attr], dim=-1)
        out = self.message_linear(message)
        alpha = self.attn(out) + alpha_i
        alpha = F.leaky_relu(alpha, self.negative_slope)
```

```
        alpha = softmax(alpha, index, ptr, size_i)
        self._alpha = alpha
        alpha = F.dropout(alpha, p=self.dropout,
training=self.training)
        out = out * alpha
        return out

    def message_and_aggregate(self, adj_t: SparseTensor) -> Tensor:
        pass

    def __repr__(self):
        return '{}({}, {})'.format(self.__class__.__name__,
self.in_channels,
                                   self.out_channels)


class CrossModalityAttention(nn.Module):
    """
    Cross-Modality Fusion in GraFrank implemented by an attention
mechanism across the K modalities.
    """

    def __init__(self, hidden_channels):
        super(CrossModalityAttention, self).__init__()
        self.hidden_channels = hidden_channels
        self.multi_linear = nn.Linear(hidden_channels,
hidden_channels, bias=True)
        self.multi_attn = nn.Sequential(self.multi_linear, nn.Tanh(),
nn.Linear(hidden_channels, 1, bias=True))

    def forward(self, modality_x_list):
        """
        :param modality_x_list: list of modality-specific node
embeddings.
        :return: final node embedding after fusion.
        """
        result = torch.cat([x.relu().unsqueeze(-2) for x in
modality_x_list], -2)  # [..., K, hidden_channels]
```

```
        wts = torch.softmax(self.multi_attn(result).squeeze(-1),
dim=-1)
        return torch.sum(wts.unsqueeze(-1) *
self.multi_linear(result), dim=-2)

    def __repr__(self):
        return '{}({}, {})'.format(self.__class__.__name__,
self.hidden_channels,
                                   self.hidden_channels)
```

(六)小批量节点邻居采样

如何像graphsage中对mini-batch的节点进行邻居采样并训练模型，使得大规模全连接图的GNN模型训练成为可能，pyg是通过torch_geometric.loader.NeighborSampler实现的

```
import torch
from torch_cluster import random_walk
from torch_geometric.loader import NeighborSampler as
RawNeighborSampler


class NeighborSampler(RawNeighborSampler):
    def sample(self, batch):
        batch = torch.tensor(batch)
        row, col, _ = self.adj_t.coo()
        # For each node in `batch`, we sample a direct neighbor (as
positive
        # example) and a random node (as negative example). This can
be modified to include hard negatives.
        pos_batch = random_walk(row, col, batch, walk_length=1,
                                coalesced=False)[:, 1]

        neg_batch = torch.randint(0, self.adj_t.size(1),
(batch.numel(),),
                                  dtype=torch.long)

        batch = torch.cat([batch, pos_batch, neg_batch], dim=0)
```

```
        return super(NeighborSampler, self).sample(batch)
```

(七)train.py

```
from torch_geometric.data import Data
import torch
from pre_solve_dataset.presolve import read_featuresNames,
read_edges, getNodefeas
from utils.sampler import NeighborSampler
from models.GraFRank import GraFrank
from models.SAGE import SAGE
import torch.nn.functional as F
import math
import random


def getRandomIndex(n, x):
    index = random.sample(range(n), x)
    return index


# 获取特征名称总览
feas = read_featuresNames('./facebook', '.featnames')
# 读取边
edges = read_edges('./facebook', '.edges')
edges = torch.tensor(edges, dtype=torch.long)
# 处理点特征矩阵
nodes_fea = getNodefeas(feas)
# 随机选择95%为训练集
num_nodes = 4039
train1 = torch.ones(num_nodes, dtype=bool)
test1 = torch.zeros(num_nodes, dtype=bool)
index = getRandomIndex(num_nodes, int(0.05 * num_nodes))
for i in range(0, len(index)):
    train1[i] = False
for i in range(0, len(index)):
    test1[index] = True
```

```
data = Data(x=nodes_fea, edge_index=edges.t(), train_mask=train1,
test_mask=test1)
# 边特征维数
n_edge_channels = 5
# 边属性
data.edge_attr = torch.ones([data.edge_index.shape[1],
n_edge_channels])

train_loader = NeighborSampler(data.edge_index, sizes=[10, 10],
batch_size=256, shuffle=True, num_nodes=data.num_nodes)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# model_type = 'GraFrank'
model_type = 'SAGE'
if model_type == 'GraFrank':
    model = GraFrank(data.num_node_features, hidden_channels=64,
edge_channels=n_edge_channels, num_layers=2,
                     input_dim_list=[350, 350, 350, 356])  # input
dim list assumes that the node features are first
    # partitioned and then concatenated across the K modalities.
else:
    model = SAGE(data.num_node_features, hidden_channels=64,
num_layers=2)

model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
x = data.x.to(device)


def train(loader):
    model.train()

    total_loss = 0
    it = 0
    for batch_size, n_id, adjs in loader:
        it += 1
        edge_attrs = [data.edge_attr[e_id] for (edge_index, e_id,
size) in adjs]
        adjs = [adj.to(device) for adj in adjs]
```

```
        edge_attrs = [edge_attr.to(device) for edge_attr in
edge_attrs]

        optimizer.zero_grad()
        out = model(x[n_id], adjs, edge_attrs)
        out, pos_out, neg_out = out.split(out.size(0) // 3, dim=0)

        # binary skipgram loss can be replaced with margin-based
pairwise ranking loss.
        pos_loss = F.logsigmoid((out * pos_out).sum(-1)).mean()
        neg_loss = F.logsigmoid(-(out * neg_out).sum(-1)).mean()
        loss = -pos_loss - neg_loss
        loss.backward()
        optimizer.step()

        total_loss += float(loss) * out.size(0)

    return total_loss / data.num_nodes


@torch.no_grad()
def test():
    x, edge_index, edge_attr = data.x.to(device),
data.edge_index.to(device), data.edge_attr.to(device)
    model.eval()
    out = model.full_forward(x, edge_index, edge_attr).cpu()
    return out


result = torch.tensor((num_nodes, num_nodes))
for epoch in range(1, 51):
    loss = train(train_loader)
    test()
    if epoch == 50:
        result = test()
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}')

ans = torch.zeros((num_nodes, num_nodes), dtype=float)
# 计算任意两个用户之间的相似度
```

```
for i in range(0, num_nodes):
    temp1 = result[i]
    temp1_v = temp1 * temp1
    s1 = math.sqrt(temp1_v.sum())
    for j in range(0, num_nodes):
        temp2 = result[j]
        r = float(torch.matmul(temp1, temp2))
        temp2_v = temp2 * temp2
        s2 = math.sqrt(temp2_v.sum())
        ans[i][j] = r / (s1 * s2)

sort_ed, indices = torch.sort(ans, dim=1, descending=True)


def getAdj(edge, user):
    friends = []
    for i in range(0, len(edge)):
        if edge[i][0] == user:
            friends.append(int(edge[i][1]))
        if edge[i][1] == user:
            friends.append(int(edge[i][0]))
    return list(set(friends))


NDCG = []
hits_list = []
rank = 0
for i in range(0, len(index)):
    DCG = 0
    IDCG = 0
    user_id = i
    N = 50
    # 推荐的朋友列表
    test_friends = []
    test_friends.clear()
    actual_friends = getAdj(edges, user_id)
    for j in range(0, N):
        test_friends.append(int(indices[i][j]))
    test_friends = set(test_friends)
```

```
        actual_friends = set(actual_friends)
        hits = len(list(test_friends & actual_friends))
        test_friends = list(test_friends)
        actual_friends = list(actual_friends)
        for k in range(0, len(actual_friends)):
            IDCG += 1.0 / math.log2(k + 2)
        for j in range(0, len(test_friends)):
            if test_friends[j] in actual_friends:
                rank += float(1.0 / (j + 1))
                break
        cnt = 0
        for j in range(0, len(test_friends)):
            if test_friends[j] in actual_friends:
                DCG += 1.0 / math.log2(cnt + 2)
                cnt += 1
        if IDCG != 0:
            NDCG.append(DCG / IDCG)
        if len(actual_friends) == 0:
            continue
        hitsN_user = hits * 1.0 / len(actual_friends)
        hits_list.append(hitsN_user)

print("hits@50: ")
print(sum(hits_list) / len(hits_list))
print("MRR: ")
print(rank / len(index))
print("NDCG: ")
print(sum(NDCG) / len(NDCG))
```

(八)评估指标的原理

1、hits@N

命中率HR(Hits Ratio)

意义：关心用户想要的，我有没有推荐到，强调预测的“准确性”

$$HR = \frac{1}{N}\sum_{i=1}^{N} hits(i)$$

参数说明:
N:用户的总数量
hits(i):第i个用户访问的值是否在推荐列表中，是则为1，否则为0

这里具体的处理方法为：对每个user，生成N个潜在对象组成的推荐列表，然后在测试集中与实际推荐列表取得交集，然后再除以列表长度。总的hits@N为所有测试集用户的hits@N的平均值。

2、MRR

平均倒数排名(Mean Reciprocal Rank,MRR)

意义：关心找到的这些项目，是否放在用户更显眼的位置里，即强调“顺序性“

$$MRR = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{p_i}$$

参数说明:
N:用户的总数量

p_i：第i个用户的真实访问值在推荐列表的位置，若推荐列表不存在该值，则$p_i \to \infty$

这里具体的处理方法为：对每个user，遍历生成的推荐列表中的对象，如果对象在实际列表中出现，那么mrr += 1.0/rank(推荐列表)。然后总的MRR为所有测试集用户此项数据的平均值。

3、NDCG

折损累计增益 DCG

$$DCG_k = \sum_{i=1}^{k} \frac{rel_i}{\log(i+1)}$$

和CG相同，如果按照论文风格来写，可以表示为 $DCG@k$ 。DCG是每个推荐项目的分数，除以它所在的位置，也就是说一个项目（item）推荐的排名越靠后，它折损的越严重。这里的顺序，是预测的顺序，而用到的分数，是数据集中的真实分数。

所以DCG的计算方法是：我们的模型计算出某个用户对所有候选项目的相似度后，根据相似度对项目进行排序，返回前k个最相似的项目作为推荐结果。这k个项目维持我们推荐的顺序，为它们标注上它们在数据集中真实的打分，然后拿来计算DCG。

DCG还有另一种计算方式，是用指数计算的，公式稍有差别，但是计算思想相同，具体可以参照我一开始列出的那几个教程。

归一化折损累计增益 nDCG

$$nDCG_k = \frac{DCG_k}{iDCG_k}$$

nDCG在论文里通常写作 $nDCG@k$ 。

$nDCG_k$ 就是折损累计增益 DCG_k 除以最大累计增益 $iDCG_k$ 。

最大累计增益iDCG的计算方法是：模型返回了k个推荐的项目，我们将这k个项目标注上它们在原始数据集上的分数，然后再根据分数进行重排序，然后再计算DCG，得到的就是iDCG。

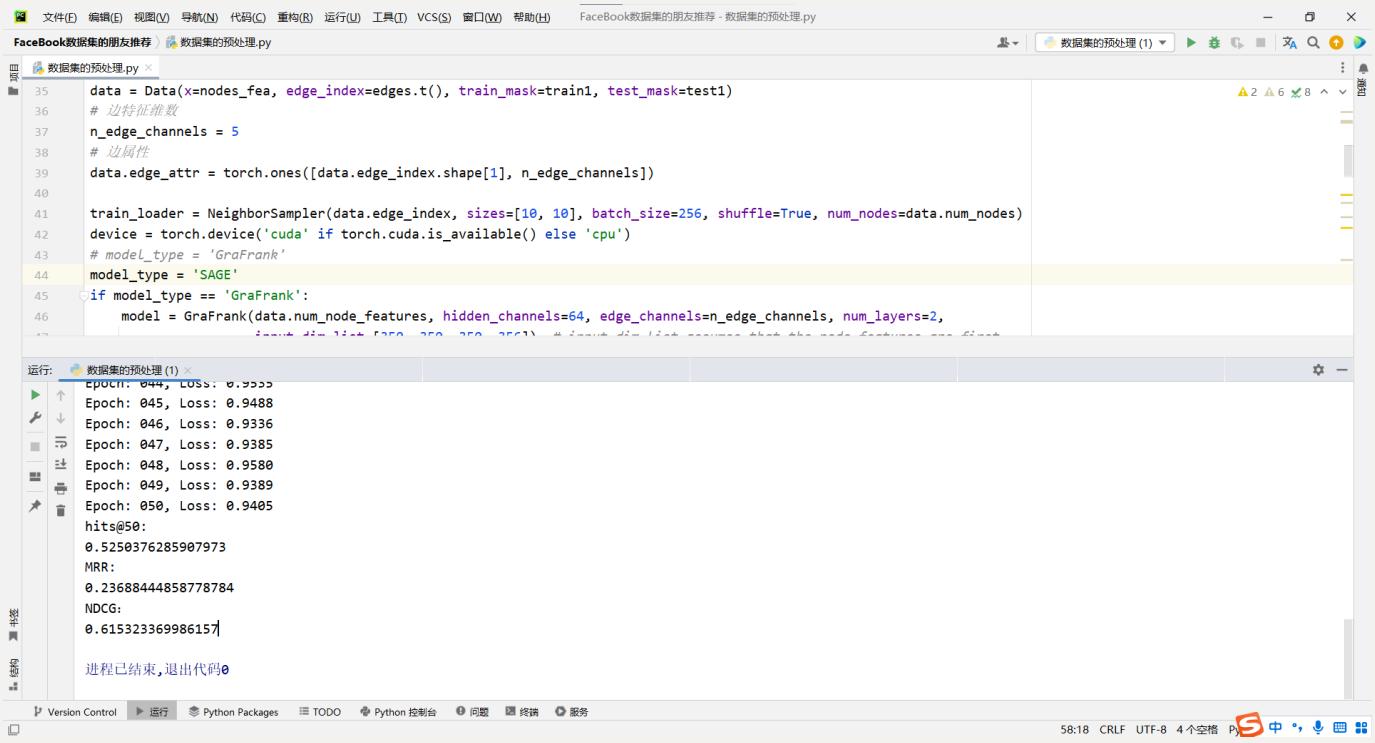
那么nDCG整体的计算过程就是：模型根据用户和候选物品的相似度，对候选项目进行排序，返回k个最相似的项目作为推荐结果。保持模型的推荐顺序，给每个项目标注它在原始数据集中的分数，计算DCG。然后再对标注的分数从大到小重排，再计算一次DCG，这一次计算出的DCG就是iDCG，让二者相除得到的就是nDCG。

这里具体的处理方法为：对每个user，DCG以推荐列表为计算对象，rel=1代表推荐命中，rel=0代表推荐没命中。IDCG以真实列表为计算对象。

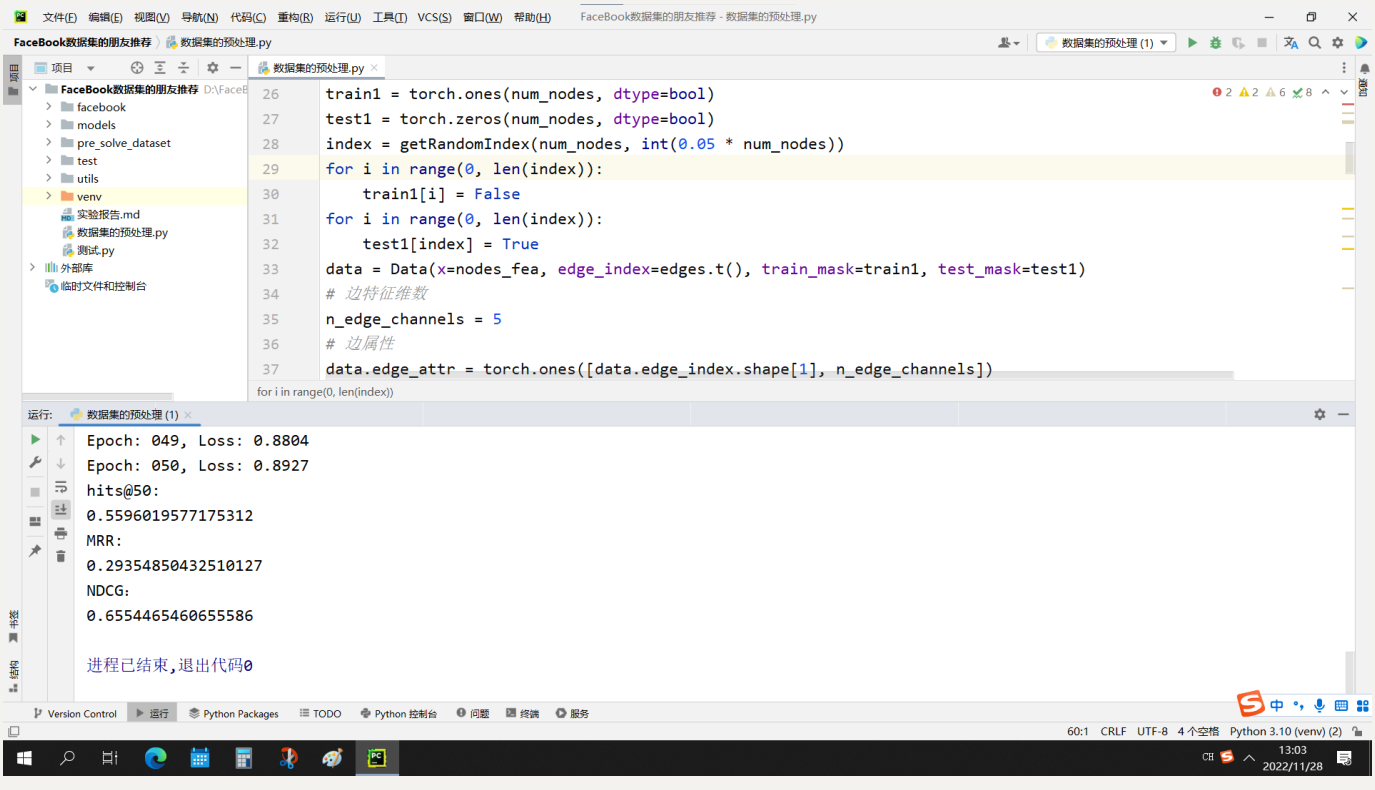
(九)Result

模型	HITS@50	MRR	NDCG
SAGE	0.53	0.24	0.62
GraFRank	0.56	0.29	0.66

SAGE:

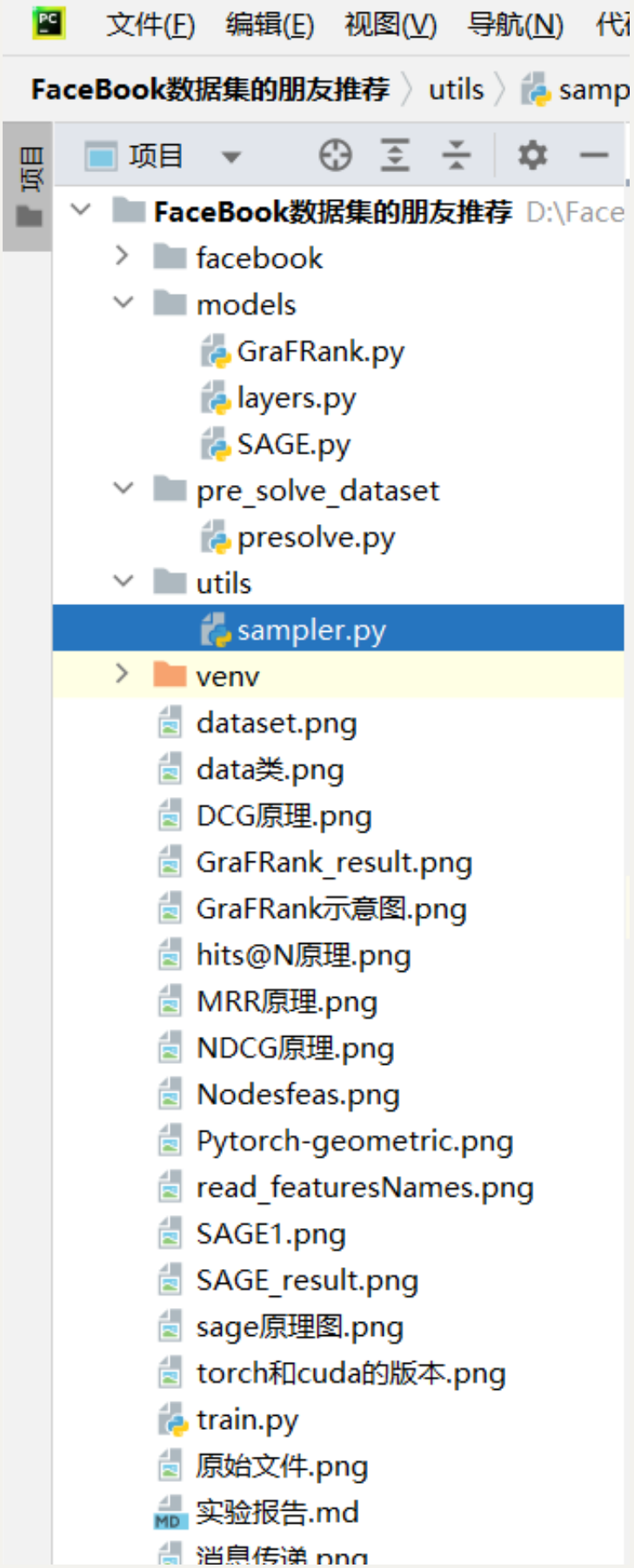


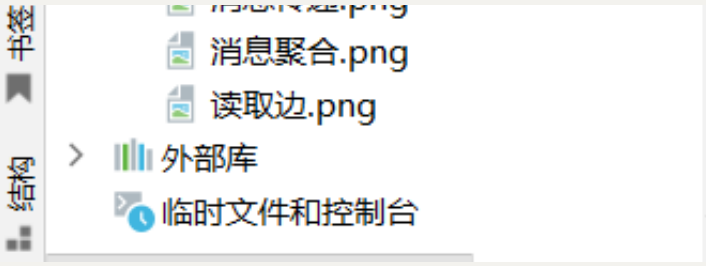
GraFRank:



(十)全部源代码（用IDE打开方式打开py文件也能得到源码）

项目所有文件：





GraFRank.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from models.layers import GraFrankConv, CrossModalityAttention


class GraFrank(nn.Module):
    """
    GraFrank Model for Multi-Faceted Friend Ranking with multi-modal node features and pairwise link features.
    (a) Modality-specific neighbor aggregation: modality_convs
    (b) Cross-modality fusion layer: cross_modality_attention
    """

    def __init__(self, in_channels, hidden_channels, edge_channels, num_layers, input_dim_list):
        """
        :param in_channels: total cardinality of node features.
        :param hidden_channels: latent embedding dimensionality.
        :param edge_channels: number of link features.
        :param num_layers: number of message passing layers.
        :param input_dim_list: list containing the cardinality of node features per modality.
        """
        super(GraFrank, self).__init__()
        self.num_layers = num_layers
        self.modality_convs = nn.ModuleList()
        self.edge_channels = edge_channels
```

```
        # we assume that the input features are first partitioned and
then concatenated across the K modalities.
        self.input_dim_list = input_dim_list

        for inp_dim in self.input_dim_list:
            modality_conv_list = nn.ModuleList()
            for i in range(num_layers):
                in_channels = in_channels if i == 0 else
hidden_channels
                modality_conv_list.append(GraFrankConv((inp_dim +
edge_channels, inp_dim), hidden_channels))

            self.modality_convs.append(modality_conv_list)

        self.cross_modality_attention =
CrossModalityAttention(hidden_channels)

    def forward(self, x, adjs, edge_attrs):
        """ Compute node embeddings by recursive message passing,
followed by cross-modality fusion.
        :param x: node features [B', in_channels] where B' is the
number of nodes (and neighbors) in the mini-batch.
        :param adjs: list of sampled edge indices per layer
(EdgeIndex format in PyTorch Geometric) in the mini-batch.
        :param edge_attrs: [E', edge_channels] where E' is the number
of sampled edge indices per layer in the mini-batch.
        :return: node embeddings. [B, hidden_channels] where B is the
number of target nodes in the mini-batch.
        """
        result = []
        for k, convs_k in enumerate(self.modality_convs):
            emb_k = None
            for i, ((edge_index, _, size), edge_attr) in
enumerate(zip(adjs, edge_attrs)):
                x_target = x[:size[1]]  # Target nodes are always
placed first.
                x_list = torch.split(x,
split_size_or_sections=self.input_dim_list, dim=-1)  # modality
partition
```

```
                x_target_list = torch.split(x_target,
split_size_or_sections=self.input_dim_list, dim=-1)
                x_k, x_target_k = x_list[k], x_target_list[k]

                emb_k = convs_k[i]((x_k, x_target_k), edge_index,
edge_attr=edge_attr)

                if i != self.num_layers - 1:
                    emb_k = emb_k.relu()
                    emb_k = F.dropout(emb_k, p=0.5,
training=self.training)

            result.append(emb_k)
        return self.cross_modality_attention(result)

    def full_forward(self, x, edge_index, edge_attr):
        """ Auxiliary function to compute node embeddings for all
nodes at once for small graphs.
        :param x: node features [N, in_channels] where N is the total
number of nodes in the graph.
        :param edge_index: edge indices [2, E] where E is the total
number of edges in the graph.
        :param edge_attr: link features [E, edge_channels] across all
edges in the graph.
        :return: node embeddings. [N, hidden_channels] for all nodes
in the graph.
        """
        x_list = torch.split(x,
split_size_or_sections=self.input_dim_list, dim=-1)  # modality
partition
        result = []
        for k, convs_k in enumerate(self.modality_convs):
            x_k = x_list[k]
            emb_k = None
            for i, conv in enumerate(convs_k):
                emb_k = conv(x_k, edge_index, edge_attr=edge_attr)

                if i != self.num_layers - 1:
                    emb_k = emb_k.relu()
```

```
                    emb_k = F.dropout(emb_k, p=0.5,
training=self.training)

            result.append(emb_k)
        return self.cross_modality_attention(result)
```

SAGE.py

```
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn.conv import SAGEConv


class SAGE(nn.Module):
    def __init__(self, in_channels, hidden_channels, num_layers):
        super(SAGE, self).__init__()
        self.num_layers = num_layers
        self.convs = nn.ModuleList()
        for i in range(num_layers):
            in_channels = in_channels if i == 0 else hidden_channels
            self.convs.append(SAGEConv((in_channels, in_channels),
hidden_channels))

    def forward(self, x, adjs, edge_attrs):
        for i, ((edge_index, _, size), edge_attr) in
enumerate(zip(adjs, edge_attrs)):
            x_target = x[:size[1]]  # Target nodes are always placed
first.
            x = self.convs[i]((x, x_target), edge_index)
            if i != self.num_layers - 1:
                x = x.relu()
                x = F.dropout(x, p=0.5, training=self.training)
        return x

    def full_forward(self, x, edge_index, edge_attr):
        for i, conv in enumerate(self.convs):
            x = conv(x, edge_index)
```

```
            if i != self.num_layers - 1:
                x = x.relu()
                x = F.dropout(x, p=0.5, training=self.training)
        return x
```

layer.py

```
from typing import Optional
from typing import Union, Tuple
import torch

from torch import Tensor
import torch.nn as nn
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.typing import OptPairTensor, Adj, Size,
OptTensor
from torch_geometric.utils import softmax
from torch_sparse import SparseTensor
import torch.nn.functional as F


class GraFrankConv(MessagePassing):
    """
    Modality-specific neighbor aggregation in GraFrank implemented by
stacking message-passing layers that are
    parameterized by friendship attentions over individual node
features and pairwise link features.
    """

    def __init__(self, in_channels: Union[int, Tuple[int, int]],
                 out_channels: int, normalize: bool = False,
                 bias: bool = True, **kwargs):  # yapf: disable
        kwargs.setdefault('aggr', 'add')
        super(GraFrankConv, self).__init__(**kwargs)

        self.in_channels = in_channels
        self.out_channels = out_channels
```

```
        self.normalize = normalize
        self.negative_slope = 0.2
        if isinstance(in_channels, int):
            in_channels = (in_channels, in_channels)

        self.self_linear = nn.Linear(in_channels[1], out_channels,
bias=bias)
        self.message_linear = nn.Linear(in_channels[0], out_channels,
bias=bias)

        self.attn = nn.Linear(out_channels, 1, bias=bias)
        self.attn_i = nn.Linear(out_channels, 1, bias=bias)

        self.lin_l = nn.Linear(out_channels, out_channels, bias=bias)
        self.lin_r = nn.Linear(out_channels, out_channels,
bias=False)

        self.reset_parameters()
        self.dropout = 0

    def reset_parameters(self):
        self.lin_l.reset_parameters()
        self.lin_r.reset_parameters()

    def forward(self, x: Union[Tensor, OptPairTensor], edge_index:
Adj, edge_attr: OptTensor = None,
                size: Size = None) -> Tensor:
        if isinstance(x, Tensor):
            x: OptPairTensor = (x, x)

        x_l, x_r = x[0], x[1]
        self_emb = self.self_linear(x_r)
        alpha_i = self.attn_i(self_emb)
        out = self.propagate(edge_index, x=(x_l, x_r), alpha=alpha_i,
edge_attr=edge_attr, size=size)
        out = self.lin_l(out) + self.lin_r(self_emb)  # dense layer.

        if self.normalize:
            out = F.normalize(out, p=2., dim=-1)
```

```
        return out

    def message(self, x_j: Tensor, alpha_i: Tensor, edge_attr:
Tensor, index: Tensor, ptr: OptTensor,
                size_i: Optional[int]) -> Tensor:
        message = torch.cat([x_j, edge_attr], dim=-1)
        out = self.message_linear(message)
        alpha = self.attn(out) + alpha_i
        alpha = F.leaky_relu(alpha, self.negative_slope)
        alpha = softmax(alpha, index, ptr, size_i)
        self._alpha = alpha
        alpha = F.dropout(alpha, p=self.dropout,
training=self.training)
        out = out * alpha
        return out

    def message_and_aggregate(self, adj_t: SparseTensor) -> Tensor:
        pass

    def __repr__(self):
        return '{}({}, {})'.format(self.__class__.__name__,
self.in_channels,
                                   self.out_channels)


class CrossModalityAttention(nn.Module):
    """
    Cross-Modality Fusion in GraFrank implemented by an attention
mechanism across the K modalities.
    """

    def __init__(self, hidden_channels):
        super(CrossModalityAttention, self).__init__()
        self.hidden_channels = hidden_channels
        self.multi_linear = nn.Linear(hidden_channels,
hidden_channels, bias=True)
        self.multi_attn = nn.Sequential(self.multi_linear, nn.Tanh(),
nn.Linear(hidden_channels, 1, bias=True))
```

```
    def forward(self, modality_x_list):
        """
        :param modality_x_list: list of modality-specific node
embeddings.
        :return: final node embedding after fusion.
        """
        result = torch.cat([x.relu().unsqueeze(-2) for x in
modality_x_list], -2)  # [..., K, hidden_channels]
        wts = torch.softmax(self.multi_attn(result).squeeze(-1),
dim=-1)
        return torch.sum(wts.unsqueeze(-1) *
self.multi_linear(result), dim=-2)

    def __repr__(self):
        return '{}({}, {})'.format(self.__class__.__name__,
self.hidden_channels,
                                   self.hidden_channels)
```

presolve.py 数据预处理

```
import os
import torch


# 指定读取文件夹和文件格式批量读取featnames并统一featnames
def read_featuresNames(path, suffix):
    files = os.listdir(path)
    result = []
    for file in files:
        pos = path + "\\" + file
        if os.path.splitext(file)[1] != suffix:
            continue
        f = open(pos, 'r')
        f_data = f.readlines()
        for row in f_data:
```

```
            tmp_list = row.split(' ')
            str_fea = tmp_list[1] + tmp_list[2] + tmp_list[3]
            result.append(str_fea)
    return list(set(result))


# 一共4039个用户88234条边，每个用户有1406个特征维度
# nodes = []
# for i in range(0, 4039):
#     nodes.append(i)
# 读取边
def read_edges(path, suffix):
    files = os.listdir(path)
    result = []
    for file in files:
        pos = path + "\\" + file
        if os.path.splitext(file)[1] != suffix:
            continue
        f = open(pos, 'r')
        f_data = f.readlines()
        for row in f_data:
            tmp_list = row.split(' ')
            result.append((int(tmp_list[0]), int(tmp_list[1])))
            result.append((int(tmp_list[1]), int(tmp_list[0])))
    return list(set(result))


def getNodefeas(feas):
    nodes_fea = torch.zeros([4039, 1406], dtype=torch.float32)
    circles_list = [0, 107, 348, 414, 686, 698, 1684, 1912, 3437,
3980]
    for ego in circles_list:
        path1 = './facebook/' + str(ego) + '.featnames'
        f1 = open(path1, 'r')
        fea_list = []
        fea_list.clear()
        f_data1 = f1.readlines()
        for row in f_data1:
            tmp_list = row.split(' ')
```

```
            y = feas.index(tmp_list[1] + tmp_list[2] + tmp_list[3])
            fea_list.append(y)

        path2 = './facebook/' + str(ego) + '.egofeat'
        f2 = open(path2, 'r')
        f_data2 = f2.readlines()
        for row in f_data2:
            tmp_list = row.split(' ')
            for i in range(0, len(tmp_list)):
                nodes_fea[ego][fea_list[i]] = int(tmp_list[i])

        path3 = './facebook/' + str(ego) + '.feat'
        f3 = open(path3, 'r')
        f_data3 = f3.readlines()
        for row in f_data3:
            tmp_list = row.split(' ')
            x = int(tmp_list[0])
            for i in range(1, len(tmp_list)):
                nodes_fea[x][fea_list[i - 1]] = int(tmp_list[i])
    return nodes_fea
```

sampler.py

```
import torch
from torch_cluster import random_walk
from torch_geometric.loader import NeighborSampler as
RawNeighborSampler


class NeighborSampler(RawNeighborSampler):
    def sample(self, batch):
        batch = torch.tensor(batch)
        row, col, _ = self.adj_t.coo()
        # For each node in `batch`, we sample a direct neighbor (as
positive
```

```
        # example) and a random node (as negative example). This can
be modified to include hard negatives.
        pos_batch = random_walk(row, col, batch, walk_length=1,
                                coalesced=False)[:, 1]

        neg_batch = torch.randint(0, self.adj_t.size(1),
(batch.numel(),),
                                  dtype=torch.long)

        batch = torch.cat([batch, pos_batch, neg_batch], dim=0)
        return super(NeighborSampler, self).sample(batch)
```

train.py

```
from torch_geometric.data import Data
import torch
from pre_solve_dataset.presolve import read_featuresNames,
read_edges, getNodefeas
from utils.sampler import NeighborSampler
from models.GraFRank import GraFrank
from models.SAGE import SAGE
import torch.nn.functional as F
import math
import random


def getRandomIndex(n, x):
    index = random.sample(range(n), x)
    return index


# 获取特征名称总览
feas = read_featuresNames('./facebook', '.featnames')
# 读取边
edges = read_edges('./facebook', '.edges')
edges = torch.tensor(edges, dtype=torch.long)
```

```
# 处理点特征矩阵
nodes_fea = getNodefeas(feas)
# 随机选择95%为训练集
num_nodes = 4039
train1 = torch.ones(num_nodes, dtype=bool)
test1 = torch.zeros(num_nodes, dtype=bool)
index = getRandomIndex(num_nodes, int(0.05 * num_nodes))
for i in range(0, len(index)):
    train1[i] = False
for i in range(0, len(index)):
    test1[index] = True
data = Data(x=nodes_fea, edge_index=edges.t(), train_mask=train1,
test_mask=test1)
# 边特征维数
n_edge_channels = 5
# 边属性
data.edge_attr = torch.ones([data.edge_index.shape[1],
n_edge_channels])

train_loader = NeighborSampler(data.edge_index, sizes=[10, 10],
batch_size=256, shuffle=True, num_nodes=data.num_nodes)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# model_type = 'GraFrank'
model_type = 'SAGE'
if model_type == 'GraFrank':
    model = GraFrank(data.num_node_features, hidden_channels=64,
edge_channels=n_edge_channels, num_layers=2,
                     input_dim_list=[350, 350, 350, 356])  # input
dim list assumes that the node features are first
    # partitioned and then concatenated across the K modalities.
else:
    model = SAGE(data.num_node_features, hidden_channels=64,
num_layers=2)

model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
x = data.x.to(device)
```

```
def train(loader):
    model.train()

    total_loss = 0
    it = 0
    for batch_size, n_id, adjs in loader:
        it += 1
        edge_attrs = [data.edge_attr[e_id] for (edge_index, e_id,
size) in adjs]
        adjs = [adj.to(device) for adj in adjs]
        edge_attrs = [edge_attr.to(device) for edge_attr in
edge_attrs]

        optimizer.zero_grad()
        out = model(x[n_id], adjs, edge_attrs)
        out, pos_out, neg_out = out.split(out.size(0) // 3, dim=0)

        # binary skipgram loss can be replaced with margin-based
pairwise ranking loss.
        pos_loss = F.logsigmoid((out * pos_out).sum(-1)).mean()
        neg_loss = F.logsigmoid(-(out * neg_out).sum(-1)).mean()
        loss = -pos_loss - neg_loss
        loss.backward()
        optimizer.step()

        total_loss += float(loss) * out.size(0)

    return total_loss / data.num_nodes


@torch.no_grad()
def test():
    x, edge_index, edge_attr = data.x.to(device),
data.edge_index.to(device), data.edge_attr.to(device)
    model.eval()
    out = model.full_forward(x, edge_index, edge_attr).cpu()
    return out
```

```
result = torch.tensor((num_nodes, num_nodes))
for epoch in range(1, 51):
    loss = train(train_loader)
    test()
    if epoch == 50:
        result = test()
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}')

ans = torch.zeros((num_nodes, num_nodes), dtype=float)
# 计算任意两个用户之间的相似度
for i in range(0, num_nodes):
    temp1 = result[i]
    temp1_v = temp1 * temp1
    s1 = math.sqrt(temp1_v.sum())
    for j in range(0, num_nodes):
        temp2 = result[j]
        r = float(torch.matmul(temp1, temp2))
        temp2_v = temp2 * temp2
        s2 = math.sqrt(temp2_v.sum())
        ans[i][j] = r / (s1 * s2)

sort_ed, indices = torch.sort(ans, dim=1, descending=True)


def getAdj(edge, user):
    friends = []
    for i in range(0, len(edge)):
        if edge[i][0] == user:
            friends.append(int(edge[i][1]))
        if edge[i][1] == user:
            friends.append(int(edge[i][0]))
    return list(set(friends))


NDCG = []
hits_list = []
rank = 0
for i in range(0, len(index)):
    DCG = 0
```

```
    IDCG = 0
    user_id = i
    N = 50
    # 推荐的朋友列表
    test_friends = []
    test_friends.clear()
    actual_friends = getAdj(edges, user_id)
    for j in range(0, N):
        test_friends.append(int(indices[i][j]))
    test_friends = set(test_friends)
    actual_friends = set(actual_friends)
    hits = len(list(test_friends & actual_friends))
    test_friends = list(test_friends)
    actual_friends = list(actual_friends)
    for k in range(0, len(actual_friends)):
        IDCG += 1.0 / math.log2(k + 2)
    for j in range(0, len(test_friends)):
        if test_friends[j] in actual_friends:
            rank += float(1.0 / (j + 1))
            break
    cnt = 0
    for j in range(0, len(test_friends)):
        if test_friends[j] in actual_friends:
            DCG += 1.0 / math.log2(cnt + 2)
            cnt += 1
    if IDCG != 0:
        NDCG.append(DCG / IDCG)
    if len(actual_friends) == 0:
        continue
    hitsN_user = hits * 1.0 / len(actual_friends)
    hits_list.append(hitsN_user)

print("hits@50: ")
print(sum(hits_list) / len(hits_list))
print("MRR: ")
print(rank / len(index))
print("NDCG: ")
print(sum(NDCG) / len(NDCG))
```