Final-Project 项目报告

1120212952 陆思翰，1120210686 张乐阳，1120210857 胡彬

2023 年 12 月 22 日

摘要

好友推荐是现在主流的社交媒体中常用的算法，用以提高用户的连通性、保留率和参与度。然而传统的推荐算法对于低活跃度、低连通的用户的好友推荐存在盲区。而 GraFRank 算法从多种特征模式和用户-用户交互中学习表达性的用户表示，对低活跃度用户群体的好友推荐准确性有显著提高。

本项目旨在实现基于 GraFRank 算法的社交网络好友推荐功能。GraFRank 利用图神经网络，通过模态特定的邻居聚合和跨模态注意力机制，分析用户的多方面特征和社交链接。项目的实践部分在中小型 Facebook 数据集上展示了该算法在候选人检索和排名任务上的优势。总体来说，本项目不仅展示了 GNN 在社交网络分析中的潜力，还特别强调了在朋友排名方面解决交互稀疏和用户交互异质性的重要性，对于好友网络分析与推荐算法优化具有一定意义。

项目分工：

1. 陆思翰：算法学习、模型优化、项目代码实现、报告撰写
2. 张乐阳：算法学习、报告撰写
3. 胡彬：算法学习、报告撰写

关键字: 图神经网络，社交网络，好友推荐，GraFRank 算法, GraphSAGE 算法, Node2Vec 算法, 模态特定邻居聚合

1 项目介绍

本项目基于 GraFRank 算法针对 Facebook 用户数据集进行社交网络分析与好友推荐算法的实现。

GraFRank 算法（Graph Attentional Friend Ranker），是一个用于社交媒体平台中好友排名的图神经网络架构。该算法特别适用于处理具有多模态节点特征和成对链接特征的大型社交网络。GraFRank 利用图神经网络进行朋友排名，重点利用多模态用户特征和链接通信特征来增强社交平台上的朋友建议，从而将相似特征的用户聚类推荐。

在本实验中，我们使用的是来自 https://snap.stanford.edu/中的 ego-Facebook 数据集，主要分析使用到的是其中的 combined 数据集（将 egonet 的所有边组合的数据）。

该数据集包含来自 Facebook 的“圈子”（或“朋友列表”）。使用 Facebook 应用程序，通过对调查参与者进行调查收集了 Facebook 的数据。数据集包括节点特征（个人资料）、圈子和自我网络。在 Facebook 中，用户好友的添加需要双方的同意，所以我们得到的社交网络图谱为无向图。

为了保护隐私，Facebook 的数据已经进行了匿名处理，用新的值替换了每个用户的 Facebook 内部 ID。此外，虽然提供了数据集的特征向量，但这些特征的解释已经被模糊化。例如，原始数据集可能包含一个特征“political=Democratic Party”，而新数据集只包含“political=anonymized feature 1”。因此，使用匿名化的数据，可以确定两个用户是否具有相同的政治倾向，但无法确定他们个人的政治倾向代表什么。

常见的好友推荐算法主要聚焦于用户习惯的聚类，这也就和之前我们在课上进行的词向量实验有相似之处，通过统计用户的特征概率来进行好友推荐。所以我们在算法综合比较时，除了项目中本就用于对比效果的 GraphSAGE 算法，还添加了 Node2Vec 算法的好友推荐效果比较，综合比较三种算法的效果来得出最佳的好友推荐方案。

2 算法分析

GraFRank（Graph Friend Rank）算法：

1. 算法解释：

GraFrank 模型是一种用于处理多模态数据的模型，旨在捕捉用户之间的关联信息，主要包含以下四个关键概念/步骤：

1. 消息传播（Message Propagation）：这一部分涉及将消息从一个用户传播到其邻居。在 GraFrank 模型中，每个用户通过将自身的特征与邻居的特征进行聚合来更新表示。这样，用户的表示可以捕捉到邻居的信息，并且通过多次传播可以捕捉到更远的连接信息。

2. 消息聚合（Message Aggregation）：消息聚合是指将邻居的特征与自身的特征进行聚合以更新用户表示。GraFrank 模型使用邻居的特征来丰富用户的表示，并考虑到邻居的信息。同时，为了保留原始特征的信息，模型还引入了自连接，将自身特征与聚合后的特征进行融合。

3. 高阶传播（Higher-order Propagation）：高阶传播是指通过堆叠多个邻居聚合层来建模多跳邻居之间的连接信息。每个邻居聚合层都会更新用户表示，并将更新后的表示传递给下一个邻居聚合层。通过多次传递邻居的信息，模型可以捕捉到更远的连接信息，从而建模用户之间更复杂的关联关系。

4. 跨模态注意力（Cross Modality Attention）：这一部分涉及学习不同特征模态之间的复杂关系。GraFrank 模型引入了跨模态注意力机制，通过使用多层感知机来学习每个特征模态对用户表示的影响权重。这样，模型可以更好地捕捉不同模态之间的相关性，从而提高多模态数据处理的性能。

通过结合上述 4 个步骤，GraFrank 模型能够有效地处理多模态数据，并捕捉到用户之间的关联信息。这使得该模型在多模态推荐、社交网络分析等任务中具有广泛的应用潜力。

接下来，我们将详细解释 GraFRank 的两大模块：特定模态的邻居聚合以及跨模态注意力层。

（1）特定模态的邻居聚合层由于不同的模态在诱导同质性方面存在差异，我们将每个模态都独立处理。因此，我们为每个用户 $u \in v$，在时间 $t \in R^+$ 上学习一个模态特定的表示 $Z_u^k(t) \in R^D$。每个用户在它的时间邻域 $N_t(u)$ 中灵活地优先考虑不同的朋友，从而考虑到用户群体中同质性分布的差异。接下来，我们对每个模态 k 通过独立且唯一的消息传递函数计算每个用户的表示，即对于任意用户 $u \in v$，$t \in R^+$，计算其所有的 $Z_u^k(t) \in R^D$。

GraFrank 首先描述单层的结构，它由两个主要操作组成：消息传播和消息聚合。然后，GraFrank 再将这一结构拓展到了多个连续的层。

（2）交叉模态注意层这是一种跨空间性的注意机制，旨在学习不同特征模式之间复杂的非线性相关性。具体来讲，我们学习了模态注意的权值 $\beta_u^k(t)$，用该值来区分每个模态 k 的影响，在此基础上使用了一个两级多层感知器（two-layer Multi-Layer Perceptron）。

下面是其定义：

$$\beta_u^k(t) = \frac{\exp\left(a_m^T W_m z_{u,L}^k + b_m\right)}{\sum_{k=1}^K \exp\left(a_m^T W_m z_{u,L}^k + b_m\right)}$$

得到了权重为 $W_m \in R^{D\times D}, a_m \in R^D$，以及标量偏差 b_m 后，我们可以得知用户 u 的最终表示 $h_u(t) \in R^D$ 是在 L 层特定模态的用户表示的加权聚合计算得出 $\left\{z_{u,L}^1, ..., z_{u,L}^K\right\}$ 后，同时按照模态权值 $\beta_u^k(t)$ 来进行加权平均所得到的，定义如下：

$$h_u(t) = \sum_{k=1}^K \beta_u^k(t) W_m z_{u,L}^k$$

2. 算法实现：

(i) 构造函数 init

```
    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.
```

```
7           :param input_dim_list: list containing the cardinality of node features
                per modality.
8           """
9           super(GraFrank, self).__init__()
10          self.num_layers = num_layers
11          self.modality_convs = nn.ModuleList()
12          self.edge_channels = edge_channels
13          # we assume that the input features are first partitioned and then
                concatenated across the K modalities.
14          self.input_dim_list = input_dim_list
15
16          for inp_dim in self.input_dim_list:
17              modality_conv_list = nn.ModuleList()
18              for i in range(num_layers):
19                  in_channels = in_channels if i == 0 else hidden_channels
20                  modality_conv_list.append(GraFrankConv((inp_dim + edge_channels,
                        inp_dim), hidden_channels))
21
22              self.modality_convs.append(modality_conv_list)
23
24          self.cross_modality_attention = CrossModalityAttention(hidden_channels)
```

参数解释:

(1) in-channels: 节点特征的总基数或总维度。

在图数据中，每个节点都可以有一个特征向量，这个特征向量可以包含多个维度或特征。在 GraFrank 中，每个节点的特征可以分为以下四个方面:

Profile Attributes（个人属性）: 这个特征集合包含了用户静态的人口统计特征，用于描述用户的个人信息，包括年龄、性别、最近的位置、语言等，这些信息可以从用户的个人资料中列出或推断出来。

Content Interests（内容兴趣）: 这是一个实数值的特征向量，用于描述用户在平台内与文本内容（例如帖子、故事）的交互，通过这个特征，模型可以了解用户对于不同主题的内容的兴趣程度。

Friending Activity（好友活动）: 这个特征表示用户在不同时间范围内的好友关系活动的聚合数量，包括发送/接收的好友请求的数量、相互成为好友的数量以及用户在不同时间范围内查看建议好友的次数。通过这个特征，模型可以了解用户在社交网络中的好友关系活动情况。

Engagement Activity（参与活动）: 这个特征表示用户在不同时间范围内与其他好友进行的应用内直接和间接互动的聚合数量，包括用户发送的文本消息、发送的短视频以及对帖子的评论等。通过这个特征，模型可以了解用户在平台上与好友之间的互动程度。

(2) hidden-channels: 潜在嵌入维度

潜在嵌入维度是用来表示数据特征的低维度空间。它是将高维度数据映射到一个更紧凑、更抽象、更易于处理的空间的方式之一。其目的在于捕捉数据中的关键特征和模式，同时尽可能地减少数据的维度，方便接下来更有效地处理数据。通过潜在嵌入，我们可以将原始数据转换成一个更易于理解和处理的形式，同时保留数据的关键信息。

(3) edge-channels: 链路特征数量

图 1: 链路特征

即 euv 中的格子数量，两个节点之间的边所蕴含的特征数量

(4) num-layers: 消息传递层的数量

在这个模型中，num-layers 的值用于循环创建多个消息传递层。在图神经网络中，消息传递是一种用于节点之间信息交互和更新的机制。每个消息传递层都执行一次消息传递操作，将节点的特征从相邻节点传递给目标节点，并更新目标节点的特征表示。通过增加消息传递层的数量，模型可以逐步地聚合更多的局部和全局信息，从而提高对图结构的理解和表征能力。

可以使用如下公式表示每经过一个消息传递层节点的更新机制:

$$h_{u,l} = F_{\theta,l}\left(h_{u,l-1}, \{h_{v,l-1}\}\right), v \in N\left(u\right)$$

(5) input-dim-list: 包含了每个模态节点特征基数的列表 (list containing the cardinality of node features per modality.)

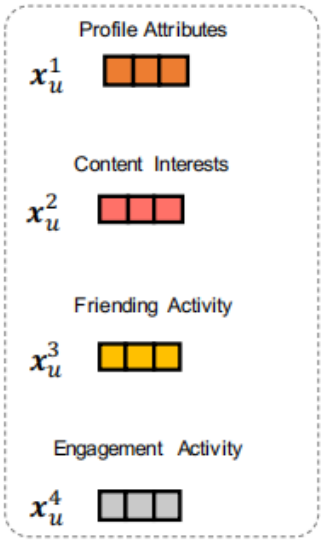


图 2: 各模态节点特征列表

整体解释:

该构造函数创建了一个多模态的消息传递网络。通过循环遍历每个输入维度，为每个模态创建多个消息传递层，并将其组织成一个模块列表的列表。然后，使用交叉模态注意力机制 CrossModalityAttention 来处理这些消息传递层的输出。该网络结构可以用于处理多模态数据，并在不同模态之间进行信息传递和特征更新。

(ii) forward 方法

```
1    def forward(self, x, adjs, edge_attrs):
2        """ 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)
```

参数解释：

(1) x: 节点特征，形如 [B', in-channels]，其中 B' 是小批量节点（和其邻居）的数量，in-channels 是输入特征的维度数。它表示了输入模型的节点特征。

(2) adjs: 每个层的采样边索引的列表，采用了 PyTorch Geometric 中的 EdgeIndex 格式。其中每个元素对应一个层的边索引。每个边索引是一个形如 [2, E] 的张量，其中 E 是采样边索引的数量。它描述了每个层中节点之间的连接关系。

(3) edge-attrs: 边属性，形状为 [E', edge-channels]，其中 E' 是每个层采样边索引的数量，edge-channels 是边属性的维度。它表示了每个层边的属性信息。

(4) 返回值: 节点嵌入表示，形状为 [B, hidden-channels]，其中 B 是小批量中目标节点的数量，hidden-channels 是潜在维度。它表示了模型计算得到的节点嵌入表示。

步骤简述：

首先，对每个“模态”（通过 self.modality-convs 表示）进行循环遍历。

然后，对于每个模态，根据输入特征的不同维度进行分割，然后进行递归的卷积操。这里使用了 torch.split 函数对输入特征进行分割，并获取当前模态对应的特征。在每一层循环中，将节点特征和目标节点特征传递给卷积层（$convs_k[i]$），并进行消息传递操作。在每一层的消息传递后，使用激活函数 ReLU 进行非线性变换，然后应用 Dropout 以防止过拟合。

最后，通过 self.cross-modality-attention 进行跨模态的注意力融合，将每个模态的结果整合成最终的节点嵌入表示。

总体来说，这个函数实现了一个基于递归消息传递和跨模态融合的图神经网络结构，用于计算节点的嵌入表示。

(iii) full-forward 方法

```
    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)
```

参数解释： (1) x: 节点的特征矩阵，形如 [N, in-channels]，其中 N 是图中节点的总数，in-channels 表示每个节点特征的维度。每一行代表一个节点，每列代表节点特征中的不同维度。

(2) edge-index: 边的索引矩阵，形如 [2, E]，其中 E 是图中边的总数。它描述了图中连接的节点对。每一列是一个边，两行分别表示这条边连接的两个节点的索引。

(3) edge-attr: 边的特征矩阵，形如 [E, edge-channels]，edge-channels 表示每条边特征的维度。对于每条边，这个矩阵存储了该边的特征信息。

(4) 返回值: 返回节点嵌入矩阵，形如 [N, hidden-channels]，其中 hidden-channels 是节点潜在的维度。这个矩阵包含了对于图中每个节点的潜在表示。

步骤简述：

首先，使用 torch.split 函数将节点特征 x 按照模态进行划分，并存储在列表 x-list 中。每个元素 x-list[k] 代表了对应模态 k 的节点特征。

然后对每个模态（通过 self.modality-convs 表示）进行循环遍历。对于每个模态，通过循环执行了模型中的卷积层操作。在每一层循环中，使用卷积层 conv 处理当前模态的特征 x_k，并进行消息传递操作。在每一层的消息传递后，使用激活函数 ReLU 进行非线性变换，然后应用 Dropout 以防止过拟合。将每个模态的最终结果作为节点嵌入的一部分，存储在 result 列表中。

最后，通过 self.cross-modality-attention 进行跨模态的注意力融合，将每个模态的结果整合成最终的节点嵌入表示。

这段代码与之前的 forward 函数相比，不再依赖于每层的采样边索引和边属性，而是直接使用整个图的边索引和边属性。这对于小图的处理更加高效，并且可以同时计算所有节点的嵌入表示。

GraphSAGE 算法：

1. 算法解释：

GraphSAGE 是一个能够利用顶点的属性信息高效产生未知顶点 embedding 的一种归纳式 (inductive) 学习的框架。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。GraphSAGE 是 Graph SAmple and aggreGatE 的缩写, 其中 SAmple 指如何对邻居个数进行采样。aggreGatE 指拿到邻居的 embedding 之后如何汇聚这些 embedding 以更新自己的 embedding 信息。

下图展示了 GraphSAGE 学习的一个过程。

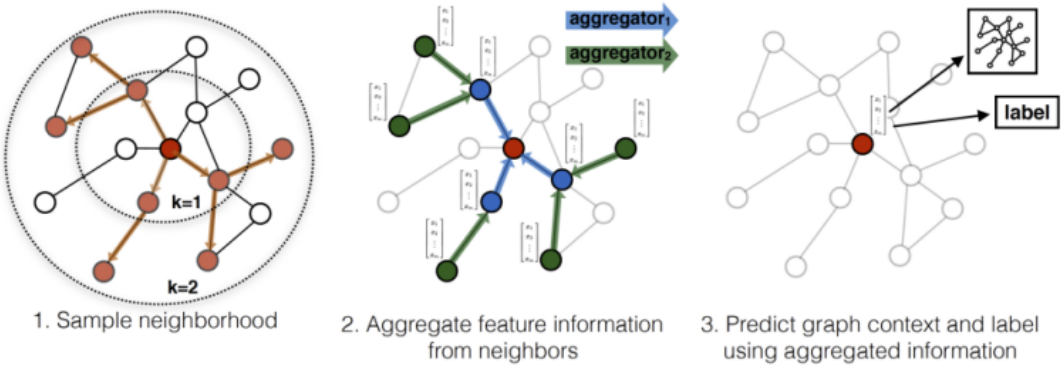


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

图 3: GraphSAGE 示意图

如图所示，GraphSAGE 的思想主要包含三个步骤

i. 对图中每个顶点邻居顶点进行采样

ii. 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

iii. 根据更新后的 embedding 预测节点的标签

该算法的详细过程如下：

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

图 4: GraphSAGE 算法过程

i. 对图上的每个结点 v，设置它的初始 embedding h_v 为它的输入特征 x_v；

ii. 之后进行 K 次迭代，在每次迭代中，对每个结点 v，聚合其邻居结点（采样后）的在上一轮迭代中生成的结点表示 h_u^{k-1} 生成当前结点的邻居结点表示 $h_{N(v)}^k$，之后连接 h_v^{k-1} 输入一个前馈神经网络得到结点的当前表示 h_v^k；

iii. 最后得到每个结点的表示 h_v^k

该算法有两个关键点：一是邻居结点采样，二是聚合邻居结点信息的聚合函数。在邻居结点采样方面：K 轮迭代中，每轮采样不同的样本，采样数量为 S_i; 在聚合函数方面：有三种聚合函数如下：

i.Mean aggregator

$$h_v^k \leftarrow \sigma\left(W \cdot MEAN\left(\left\{h_v^{k-1}\right\} \cup \left\{h_u^{k-1}, \forall u \in N(v)\right\}\right)\right)$$

ii.LSTM aggregator

使用 LSTM 对邻居结点信息进行聚合。值得注意地是，因为 LSTM 的序列性，这个聚合函数不具备对称性。在该方法具体应用时使用对邻居结点随机排列的方法来将其应用于无序集合。

iii.Pooling aggregator

$$AGGREGATE_k^{pool} = \max\left(\left\{\sigma\left(W_{pool} h_{u_i}^k + b\right), \forall u_i \in N(v)\right\}\right)$$

2. 算法实现：

```
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
```

```
13              self.convs.append(SAGEConv((in_channels, in_channels),
                    hidden_channels))
14
15      def forward(self, x, adjs, edge_attrs):
16          for i, ((edge_index, _, size), edge_attr) in enumerate(zip(adjs,
                edge_attrs)):
17              x_target = x[:size[1]]  # Target nodes are always placed first.
18              x = self.convs[i]((x, x_target), edge_index)
19              if i != self.num_layers - 1:
20                  x = x.relu()
21                  x = F.dropout(x, p=0.5, training=self.training)
22          return x
23
24      def full_forward(self, x, edge_index, edge_attr):
25          for i, conv in enumerate(self.convs):
26              x = conv(x, edge_index)
27              if i != self.num_layers - 1:
28                  x = x.relu()
29                  x = F.dropout(x, p=0.5, training=self.training)
30          return x
```

代码首先导入了必要的库，包括 torch.nn 和 torch-geometric.nn.conv 中的 SAGEConv。然后定义了一个名为 SAGE 的类，继承自 nn.Module。

在 SAGE 类的初始化方法中，定义了模型的参数：输入通道数（in-channels）、隐藏层通道数（hidden-channels）和层数（num-layers）。

然后创建了一个空的 nn.ModuleList，用于存储 SAGEConv 层。通过循环，根据层数创建了对应数量的 SAGEConv 层，并将其添加到 convs 列表中。

forward 方法是模型的前向传播函数。它接受输入 x、图的邻接矩阵 adjs 和边属性 edge-attrs 作为输入。在循环中，对每个邻接矩阵和边属性进行迭代。首先根据邻接矩阵和边属性获取目标节点的特征 x-target。然后将 x 和 x-target 作为输入，通过 SAGEConv 层进行图卷积操作。如果不是最后一层，则对输出进行 ReLU 激活函数处理，并进行一半的 dropout 操作。最后返回输出 x。full-forward 方法与 forward 方法类似，但是它接受的输入是图的边索引 edge-index 和边属性 edge-attr。在循环中，对每个 SAGEConv 层进行迭代，依次进行图卷积操作、ReLU 激活函数处理和 dropout 操作。最后返回输出 x。

Node2Vec 算法：

1. 算法解释：

Word2Vec 是根据词与词的共现关系学习向量的表示，而 Node2Vec 采用的 DeepWalk 策略受其启发，通过随机游走的方式提取顶点序列，再用 Word2Vec 模型根据顶点和顶点的共现关系，学习顶点的向量表示。可以理解为用文字把图的内容表达出来，如下图所示。

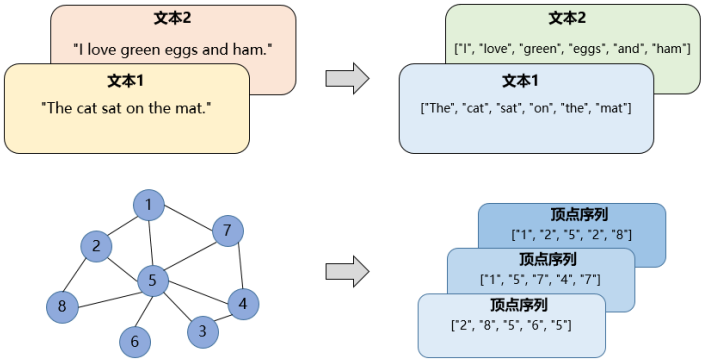


图 5: Word2Vec 与 Node2Vec

DeepWalk 不适用于有权图，它无法学习边上的权重信息。Node2Vec 可以看作 DeepWalk 的扩展，它学习嵌入的过程也可以分两步：

i. 二阶随机游走（2nd-order random walk）

ii. 使用 skip-gram 学习顶点嵌入

可以看到与 DeepWalk 的区别就在于游走的方式，在二阶随机游走中，转移概率 π_{VX} 受权值 W_{VX} 影响（无权图中 W_{VX} 为 1）：

$$\pi_{VX} = \alpha_{pq}(t,x) \cdot W_{VX}$$

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p}, if\ d_{tx} = 0 \\ 1, if\ d_{tx} = 1 \\ \frac{1}{q}, if\ d_{tx} = 2 \end{cases}$$

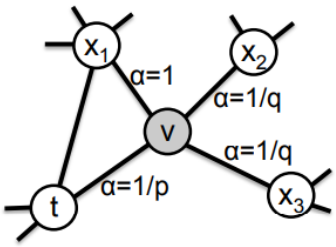


图 6: Enter Caption

在上式中，t 代表上一个节点，v 代表当前节点，x 代表下一个准备访问的节点。d_{tx} 代表上一个节点与待访问节点的距离。d_{tx} =0 代表从当前节点返回上一个访问节点，即“t->v->t”。

算法通过 p、q 两个超参数来控制游走到不同顶点的概率。

q：被称作进出参数，控制“向内”还是“向外”游走。若 q>1，倾向于访问与 t 接近的顶点，若 q<1 则倾向于访问远离 t 的顶点。

p：被称为返回参数，控制重复访问刚刚访问过的顶点的概率。若设置的值较大，就不大会刚问刚刚访问过的顶点。若设置的值较小，那就可能回路返回一步。

2. 算法实现：

```
1        def __init__(self, graph: nx.Graph, dimensions: int = 128, walk_length:
             int = 80, num_walks: int = 10, p: float = 1,
2                q: float = 1, weight_key: str = 'weight', workers: int = 1,
                     sampling_strategy: dict = None,
```

```
                quiet: bool = False, temp_folder: str = None, seed: int = None):


        self.graph = graph
        self.dimensions = dimensions
        self.walk_length = walk_length
        self.num_walks = num_walks
        self.p = p
        self.q = q
        self.weight_key = weight_key
        self.workers = workers
        self.quiet = quiet
        self.d_graph = defaultdict(dict)

        if sampling_strategy is None:
            self.sampling_strategy = {}
        else:
            self.sampling_strategy = sampling_strategy

        self.temp_folder, self.require = None, None
        if temp_folder:
            if not os.path.isdir(temp_folder):
                raise NotADirectoryError("temp_folder does not exist or is not a
                    directory. ({})".format(temp_folder))

            self.temp_folder = temp_folder
            self.require = "sharedmem"

        if seed is not None:
            random.seed(seed)
            np.random.seed(seed)

        self._precompute_probabilities()
        self.walks = self._generate_walks()
```

初始化 Node2Vec 对象，预先计算行走概率并生成行走。

参数解释:

graph: 输入图

dimensions: 嵌入维度（默认值：128）

$walk_length$: 每次行走中的节点数（默认值：80）

num_walks: 每个节点的行走次数（默认值：10）

p: 返回超参数（默认值：1）

q: 输入参数（默认值：1）

$weight_key$: 在加权图中，这是权重属性的键（默认值：'weight'）

workers: 并行执行的工作线程数（默认值：1）

$sampling_strategy$: 节点特定的采样策略，支持设置节点特定的'q'、'p'、'num_walks' 和'$walk_length$'。

seed: 随机数生成器的种子。

$temp_folder$: 用于保存 $self.d_graph$ 内存映射的文件夹路径(用于大型图形);将传递给 joblib.Parallel.$temp_folder$。

```
    def _precompute_probabilities(self):
        d_graph = self.d_graph

        nodes_generator = self.graph.nodes() if self.quiet \
            else tqdm(self.graph.nodes(), desc='Computing transition probabilities')

        for source in nodes_generator:

            # 为第一次行走初始化概率字典
            if self.PROBABILITIES_KEY not in d_graph[source]:
                d_graph[source][self.PROBABILITIES_KEY] = dict()

            for current_node in self.graph.neighbors(source):

                # 初始化概率字典
                if self.PROBABILITIES_KEY not in d_graph[current_node]:
                    d_graph[current_node][self.PROBABILITIES_KEY] = dict()

                unnormalized_weights = list()
                d_neighbors = list()

                # 计算非归一化权重
                for destination in self.graph.neighbors(current_node):

                    p = self.sampling_strategy[current_node].get(self.P_KEY,
                                                                 self.p) if
                                                                     current_node
                                                                     in self.
                                                                     sampling_

                                                                 strategy else
                                                                     self.p
                    q = self.sampling_strategy[current_node].get(self.Q_KEY,
                                                                 self.q) if
                                                                     current_node
                                                                     in self.
                                                                     sampling_

                                                                 strategy else
                                                                     self.q

                    try:
                        if self.graph[current_node][destination].get(self.
                            weight_key):
                            weight = self.graph[current_node][destination].get(
                                self.weight_key, 1)
                        else:
```

```
                        ## Example : AtlasView({0: {'type': 1, 'weight
                            ':0.1}})- when we have edge weight
                        edge = list(self.graph[current_node][destination])
                            [-1]
                        weight = self.graph[current_node][destination][edge].
                            get(self.weight_key, 1)

                except:
                    weight = 1

                if destination == source:  # Backwards probability
                    ss_weight = weight * 1 / p
                elif destination in self.graph[source]:  # 若邻居相连
                    ss_weight = weight
                else:
                    ss_weight = weight * 1 / q

                # 分配未归一化的采样策略权重, 在随机游走过程中进行归一化
                unnormalized_weights.append(ss_weight)
                d_neighbors.append(destination)

            # 归一化
            unnormalized_weights = np.array(unnormalized_weights)
            d_graph[current_node][self.PROBABILITIES_KEY][
                source] = unnormalized_weights / unnormalized_weights.sum()

        #  计算源节点的首次行走权重
        first_travel_weights = []

        for destination in self.graph.neighbors(source):
            first_travel_weights.append(self.graph[source][destination].get(
                self.weight_key, 1))

        first_travel_weights = np.array(first_travel_weights)
        d_graph[source][self.FIRST_TRAVEL_KEY] = first_travel_weights /
            first_travel_weights.sum()

        # 保存邻居节点
        d_graph[source][self.NEIGHBORS_KEY] = list(self.graph.neighbors(
            source))
```

该函数用于为每个节点预先计算转移概率。实现效果如下:

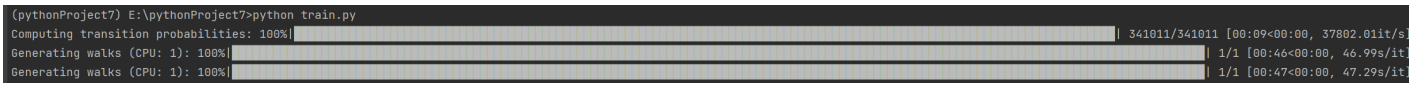


图 7: 预计算转移概率

```
    def _generate_walks(self) -> list:
```

```
        flatten = lambda l: [item for sublist in l for item in sublist]

        # 将num_walks分割给每个工作线程
        num_walks_lists = np.array_split(range(self.num_walks), self.workers)

        walk_results = Parallel(n_jobs=self.workers, temp_folder=self.temp_folder
            , require=self.require)(
            # 生成随机游走
            delayed(parallel_generate_walks)(self.d_graph,
                                             self.walk_length,
                                             len(num_walks),
                                             idx,
                                             self.sampling_strategy,
                                             self.NUM_WALKS_KEY,
                                             self.WALK_LENGTH_KEY,
                                             self.NEIGHBORS_KEY,
                                             self.PROBABILITIES_KEY,
                                             self.FIRST_TRAVEL_KEY,
                                             self.quiet) for
            idx, num_walks
            in enumerate(num_walks_lists, 1))

        # 将结果展平为一个列表
        walks = flatten(walk_results)

        # 返回游走列表，其中每个游走是一个节点列表
        return walks
```

该函数用于生成随机游走，这些游走将用作 skip-gram 模型的输入。

```
    def fit(self, **skip_gram_params) -> gensim.models.Word2Vec:
        if 'workers' not in skip_gram_params:
            skip_gram_params['workers'] = self.workers

        # 确定gensim库的版本，以及输出的命名方式（在版本4.0.0中，输出维度的命名从
            'size'更改为'vector_size'）
        gensim_version = pkg_resources.get_distribution("gensim").version
        size_key = 'size' if gensim_version < '4.0.0' else 'vector_size'
        if size_key not in skip_gram_params:
            skip_gram_params[size_key] = self.dimensions

        if 'sg' not in skip_gram_params:
            skip_gram_params['sg'] = 1

        # 创建 Word2Vec 模型实例，但不传递语料库
        model = gensim.models.Word2Vec(**skip_gram_params)

        # 构建词汇表
        model.build_vocab(self.walks)

```

```
        # 现在进行训练
        model.train(self.walks, epochs=model.epochs, total_examples=model.
            corpus_count)
        return model
```

该函数使用 gensim 的 Word2Vec 模型创建嵌入向量, 对序列采用 word2vec 训练，得到 node embedding。

3 实验过程

首先对于数据集中的数据进行读取和预处理操作,该数据集已经区分为了 circles、edges、egofeat、feat、featnames 五类数据，无需额外的清理工作，我们只需要分别读取并整合提取即可。

实施代码如下:

```
import os
import torch

def read_features_names(path, suffix):
    """
    批量读取特征名称并去重。
    :param path: 特征文件所在目录。
    :param suffix: 文件后缀。
    :return: 唯一的特征名称列表。
    """
    files = os.listdir(path)
    unique_features = set()
    for file in files:
        if os.path.splitext(file)[1] == suffix:
            with open(os.path.join(path, file), 'r') as f:
                for row in f:
                    feature_name = '␣'.join(row.split()[1:4])
                    unique_features.add(feature_name)
    return list(unique_features)

def read_edges(path, suffix):
    """
    读取边。
    :param path: 边文件所在目录。
    :param suffix: 文件后缀。
    :return: 唯一边的集合。
    """
    files = os.listdir(path)
    edges = set()
    for file in files:
        if os.path.splitext(file)[1] == suffix:
            with open(os.path.join(path, file), 'r') as f:
                for row in f:
                    nodes = tuple(map(int, row.split()[:2]))
                    edges.add(nodes)
```

```
                    edges.add(nodes[::-1])  # 双向边
    return list(edges)

def get_node_features(feas, num_nodes=4039, num_features=1406):
    """
    获取节点特征。
    :param feas: 特征名称列表。
    :param num_nodes: 节点数量。
    :param num_features: 特征维数。
    :return: 节点特征张量。
    """
    nodes_fea = torch.zeros([num_nodes, num_features], dtype=torch.float32)
    circles_list = [0, 107, 348, 414, 686, 698, 1684, 1912, 3437, 3980]
    for ego in circles_list:
        featnames_path = os.path.join('./facebook', f'{ego}.featnames')
        with open(featnames_path, 'r') as f1:
            fea_list = [feas.index('␣'.join(row.split()[1:4])) for row in f1]

        egofeat_path = os.path.join('./facebook', f'{ego}.egofeat')
        with open(egofeat_path, 'r') as f2:
            for row in f2:
                nodes_fea[ego, fea_list] = torch.tensor(list(map(int, row.split()
                    )), dtype=torch.float32)

        feat_path = os.path.join('./facebook', f'{ego}.feat')
        with open(feat_path, 'r') as f3:
            for row in f3:
                data = list(map(int, row.split()))
                x = data[0]
                nodes_fea[x, fea_list] = torch.tensor(data[1:], dtype=torch.
                    float32)
    return nodes_fea
```

通过数据集预处理，我们读取目录中所有具有指定后缀的文件，对于每个文件，按行读取内容，并将每行的第 2 到第 4 个元素（使用空格分割）合并为一个特征名称。将所有特征名称添加到一个集合中，自动去除重复项，最后返回一个包含所有唯一特征名称的列表，即是我们所需要的用于建模的数据集。

在完成了数据预处理后，进行训练模型的初始化过程。通过设置不同的 MODEL-TYPE 选择预先导入的三个模型进行训练，其中 Node2Vec 模型需要先基于数据集创建节点连接图（基于 NetworkX 库实现），并提初始化模型实例。

在先前的作业中，我们实现了网络连通性的可视化图像绘制，通过 NetworkX 库的 spring-layout 函数计算节点的布局位置，通过 draw-networkx 函数绘制网络图，将用于 Node2Vec 的节点连接图预先构建并可视化。

输出网络图效果如下：

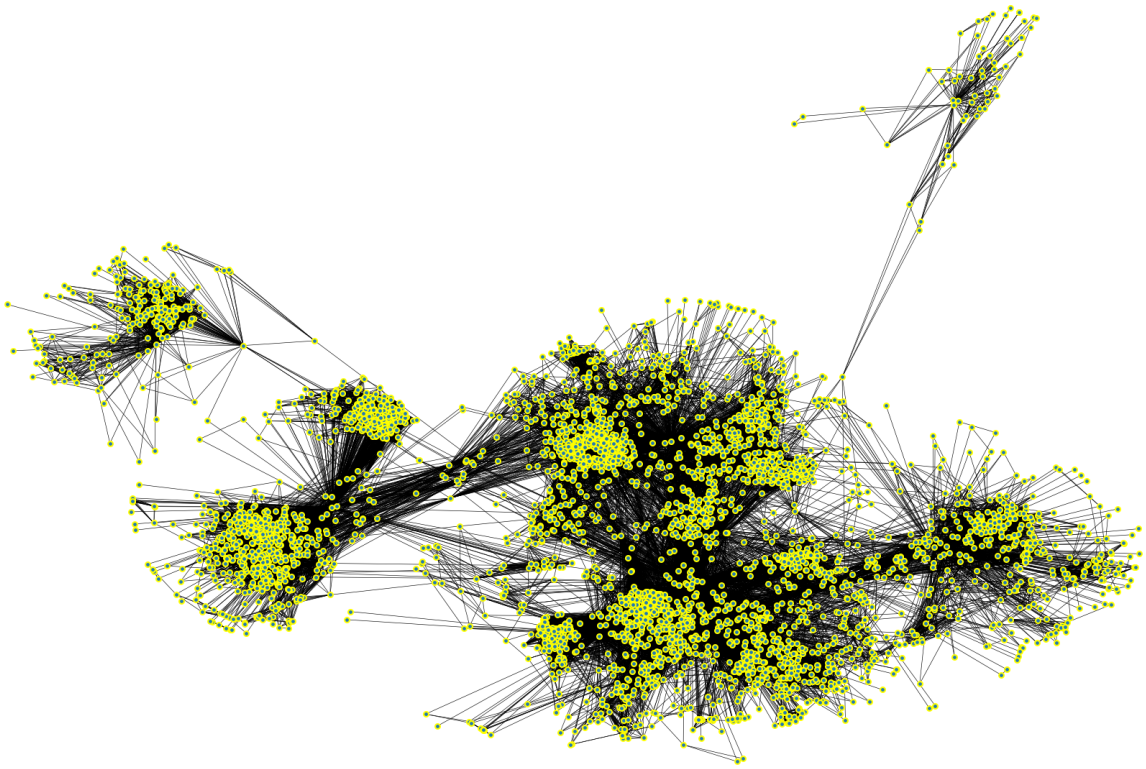


图 8: 网络连接图

整体模型训练

实现代码如下:

```
from matplotlib import pyplot as plt
from torch_geometric.data import Data
import torch
from pre_solve_dataset.presolve import read_features_names, read_edges,
    get_node_features
from utils.sampler import NeighborSampler
from models.GraFRank import GraFrank
from models.SAGE import SAGE
from models.Node2Vec import Node2Vec
import torch.nn.functional as F
import math
import random
import networkx as nx

# 定义全局变量
NUM_NODES = 4039
# 选择模型
MODEL_TYPE = 'GraFrank'
# MODEL_TYPE = 'SAGE'
```

```
# MODEL_TYPE = 'Node2Vec'

# 获取特征名称总览
feas = read_features_names('./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 = get_node_features(feas)
# 随机选择95%的数据集为训练集
train1 = torch.ones(NUM_NODES, dtype=bool)
test1 = torch.zeros(NUM_NODES, dtype=bool)
index = get_random_index(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, tesk_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])

# 创建 NetworkX 图
G = nx.Graph()
# 添加节点到图中
for node in range(NUM_NODES):
    G.add_node(node)
# 添加边到图中
for edge in edges:
    G.add_edge(edge[0], edge[1])
# 初始化 Node2Vec 实例
node2vec = Node2Vec(graph=G, dimensions=128, walk_length=3, num_walks=1, p=1, q
    =1)
walks = node2vec._generate_walks()

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')
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])
    # 输入维度列表假设节点特征首先被分割，然后在K种模态之间进行合并
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    x = data.x.to(device)
elif MODEL_TYPE == 'SAGE':
```

```
    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)
elif MODEL_TYPE == 'Node2Vec':
    model = node2vec.fit()


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}')

# 当使用 Node2Vec 时，直接从 Word2Vec 模型中提取嵌入向量
if MODEL_TYPE == 'Node2Vec':
    embeddings = {word: model.wv[word] for word in model.wv.key_to_index}
    embeddings_tensor = {word: torch.tensor(embeddings[word], device=device) for
        word in embeddings}
    # embeddings_tensor 现在包含了节点嵌入向量，可以用于后续任务
```

好友推荐效果评估

这里我们对于结果的评估标准选择了 Precision（准确度）、Recall（回归率）、f1 值、NDCG（归一化折损累计增益）、hits（命中率）和 MRR（平均倒数排名）。这里我们设计推荐的好友为模型训练结果中得到分值最大的前 50 个用户，所以评估则计算这 50 个用户的平均指标。各项评估指标分别的意义与计算方式如下：

1. Precision：精确度是推荐的项目中与用户相关的项目的比例，是衡量推荐准确性的指标。

$$Precision = \frac{TP}{TP + FP}$$

(TP 为正确预测的正样本，FP 为错误预测的负样本)

2. Recall：召回率是推荐给用户的相关项目的比例，是衡量推荐全面性的指标。

$$Recall = \frac{TP}{TP + FN}$$

(TP 为正确预测的正样本，FN 为错误预测的正样本)

3. F1：是精度和召回率的调和平均值，表示推荐系统整体性能，F1 值越高意味着推荐结果既准确又全面。

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

4. NDCG：DCG 是每个推荐用户的分数（训练结果赋值），除以他所在的位置，即一个项目（item）推荐的排名（按照预测顺序）越是靠后，其折损越严重。

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

通过 DCG 得到单一用户的折损累计增益，然后除以最大累计增益 $iDCG_k$，进行归一化操作得到 n 个用户的折损累计增益，即 nDCG。

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

这里的 $iDCG_k$ 是在计算得到 DCG 后对标注的分数进行降序重排再次计算 DCG 得到的最大累计增益值。

5. MRR：（Mean Reciprocal Rank）为推荐的平均倒数排名，通过用户的排名来强调推荐结果的顺序性。

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

P_i 表示用户的真实访问值在推荐列表中的位置（排名），用于标识推荐结果的顺序的准确性

6. Hits：表示目标用推荐的准确性（是否推荐到了相关性高的用户）。

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

hits(i) 表示第 i 个用户是否出现在推荐列表中，返回布尔值。

用户推荐结果评估实施代码：

```
# 初始化评估指标
total_precision = 0
total_recall = 0
total_f1 = 0

# 遍历每个用户进行评估
for i in range(len(index)):
    user_id = index[i]
    N = 50  # Top-N推荐
    actual_friends = get_adjacency_list(edges, user_id)
    predicted_friends = [int(indices[user_id][j]) for j in range(N)]

    precision, recall, f1 = calculate_precision_recall_f1(actual_friends,
        predicted_friends, N)
    total_precision += precision
    total_recall += recall
    total_f1 += f1

# 计算平均评估指标
avg_precision = total_precision / len(index)
avg_recall = total_recall / len(index)
avg_f1 = total_f1 / len(index)

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

```
37      test_friends = set(test_friends)
38      actual_friends = set(actual_friends)
39      hits = len(list(test_friends & actual_friends))
40      test_friends = list(test_friends)
41      actual_friends = list(actual_friends)
42      for k in range(0, len(actual_friends)):
43          IDCG += 1.0 / math.log2(k + 2)
44      for j in range(0, len(test_friends)):
45          if test_friends[j] in actual_friends:
46              rank += float(1.0 / (j + 1))
47              break
48      cnt = 0
49      for j in range(0, len(test_friends)):
50          if test_friends[j] in actual_friends:
51              DCG += 1.0 / math.log2(cnt + 2)
52              cnt += 1
53      # 最大累计增益IDCG不为0
54      if IDCG != 0:
55          NDCG.append(DCG / IDCG)
56      if len(actual_friends) == 0:
57          continue
58      hitsN_user = hits * 1.0 / len(actual_friends)
59      hits_list.append(hitsN_user)
60
61  print(f"Average␣Precision@{N}:␣")
62  print(avg_precision)
63  print(f"Average␣Recall@{N}:␣")
64  print(avg_recall)
65  print(f"Average␣F1@{N}:␣")
66  print(avg_f1)
67  print("hits@50:␣")
68  print(sum(hits_list) / len(hits_list))
69  print("MRR:␣")
70  print(rank / len(index))
71  print("NDCG:␣")
72  print(sum(NDCG) / len(NDCG))
```

运行得到输出结果（以 GraFRank 模型为例）如下：

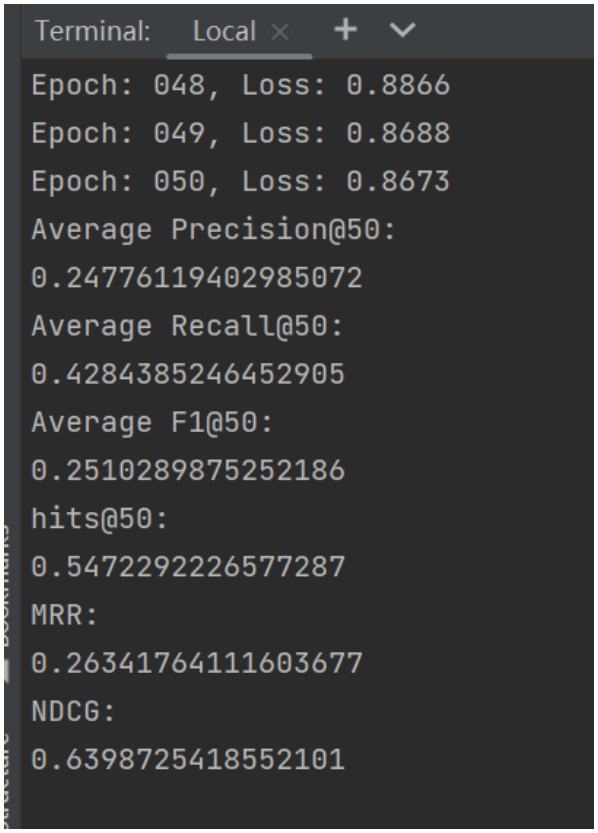


图 9: GraFRank 模型评估结果

通过运行对应模型下的训练代码，我们得到了三个模型的推荐列表的评估结果，如下表所示：

模型 \ 评估指标	Average Precision	Average Recall	Average f1	Hits	MRR	NDCG
GraFRank	0.2477612	0.4284385	0.2510290	0.5472292	0.2634176	0.6398725
GraphSAGE	0.2971144	0.4450813	0.2766158	0.5108754	0.2613458	0.6152907
Node2Vec	0.2234851	0.4354652	0.2235981	0.4892859	0.2198786	0.5879206

图 10: 模型评估对比

综合以上的评估结果，我们可以发现，在该数据集的推荐情况中，基于 GraFRank 模型的综合推荐效果较好。当然，由于该数据集较小且无向，所以各个模型之间的差距并不显著。

在完成了模型训练及推荐之后，我们通过一下代码将训练结果进行保存

```
1    torch.save(result, 'result.pt')
2    torch.save(index, 'index.pt')
```

接下来，我们需要通过训练结果，基于训练得到的用户分数评价分析用户的相似性，从而构建相似性矩阵。就我们的分析结果对测试集中的用户推荐好友（这里每位用户推荐 50 为相似度较高的好友）并输出推荐的好友列表 recommendations.xlsx。

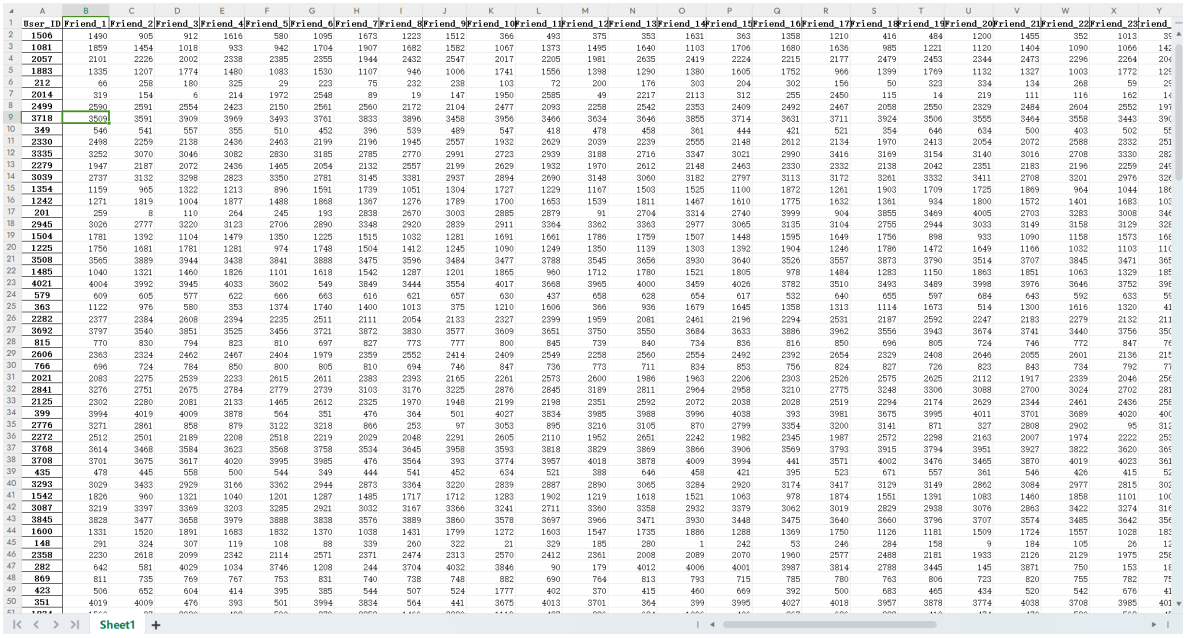
实施代码如下：

```
1 import os
2 import pandas as pd
```

```python
import torch
from train import NUM_NODES, index  # 使用训练集中定义的num_nodes

# 基于训练结果，对于节点进行分数评价用于实现好友推荐
result = torch.load('result.pt')


# 基于训练得到的结果，计算所有用户之间的相似度
def compute_similarity(result):
    similarity = torch.zeros((NUM_NODES, NUM_NODES), dtype=torch.float)
    for i in range(NUM_NODES):
        for j in range(NUM_NODES):
            dot_product = torch.dot(result[i], result[j])
            norm_i = result[i].norm(p=2)
            norm_j = result[j].norm(p=2)
            similarity[i][j] = dot_product / (norm_i * norm_j)
    return similarity


# 根据相似度矩阵推荐好友
def recommend_friends(similarity_matrix, user_id, top_k):
    _, top_indices = torch.topk(similarity_matrix[user_id], top_k + 1)
    # 排除自身，只推荐其他用户
    top_indices = top_indices[1:]
    return top_indices.tolist()


# 对每个用户进行好友推荐
def make_recommendations(test_indices, similarity_matrix, top_k):
    recommendations = {}
    for user_id in test_indices:
        recommended_friends = recommend_friends(similarity_matrix, user_id, top_k
            )
        recommendations[user_id] = recommended_friends
    return recommendations


# 计算推荐结果的相似度矩阵
similarity_matrix = compute_similarity(result)

# 为测试集中的用户推荐好友
top_k = 50  # 推荐列表中的好友数量
recommendations = make_recommendations(index, similarity_matrix, top_k)

# 打印推荐结果
# 将推荐结果存入Excel文件
df = pd.DataFrame.from_dict(recommendations, orient='index', columns=[f"Friend_{i
    +1}" for i in range(top_k)])
```

```
df.index.name = 'User_ID'
# 判断文件是否存在
filename = 'recommendations.xlsx'
if os.path.exists(filename):
    # 文件存在，则在文件名后添加数字存储
    filename = filename + '_' + str(len(os.listdir('.'))) + '.xlsx'

# 保存文件
df.to_excel(filename)
```

运行 recommend.py 将首先运行 train.py，通过选择的模型进行训练，并根据训练结果输出推荐好友列表，好友列表为相似度较高的前 50 名用户的 ID，如下所示（部分）：



User_ID	Friend_1	Friend_2	Friend_3	Friend_4	Friend_5	Friend_6	Friend_7	Friend_8	Friend_9	Friend_10
1506	1490	905	912	1616	580	1095	1673	1223	1512	366
1081	1859	1454	1018	933	942	1704	1907	1682	1582	1067
2057	2101	2226	2002	2338	2385	2355	1944	2432	2547	2017
1883	1335	1207	1774	1480	1083	1530	1107	946	1006	1741
212	66	258	180	325	29	223	75	232	238	103
2014	319	154	6	214	1972	2548	89	19	147	1950
2499	2590	2591	2554	2423	2150	2561	2560	2172	2104	2477
3718	3590	3591	3909	3969	3493	3761	3833	3896	3458	3956
349	546	541	557	355	510	452	396	539	489	547
2330	2498	2259	2138	2436	2463	2199	2196	1945	2557	1932
3335	3252	3070	3046	3082	2830	3185	2785	2770	2991	2723
2279	1947	2187	2072	2436	1465	2054	2132	2557	2199	2629
3039	2737	3132	3298	2823	3350	2781	3145	3381	2937	2894
1354	1159	965	1322	1213	896	1591	1739	1051	1304	1727
1242	1271	1819	1004	1877	1488	1868	1367	1276	1789	1700
201	259	8	110	264	245	193	2838	2670	3003	2885
2945	3026	2777	3220	3123	2706	2890	3348	2920	2839	2911
1504	1781	1392	1104	1479	1350	1225	1515	1032	1281	1691
1225	1756	1681	1781	1281	974	1748	1504	1412	1245	1090
3508	3565	3889	3944	3438	3841	3888	3475	3596	3484	3477
1485	1040	1321	1460	1826	1101	1618	1542	1287	1201	1865
4021	4004	3992	3945	4033	3602	549	3849	3444	3554	4017
579	609	605	577	622	666	663	616	621	657	630
363	1122	976	580	353	1374	1740	1400	1013	375	1210
2282	2377	2384	2608	2394	2235	2511	2111	2054	2133	2327
3692	3797	3540	3851	3525	3456	3721	3872	3830	3577	3609
815	770	830	794	823	810	697	827	773	777	800
2606	2363	2324	2462	2467	2404	1979	2359	2552	2414	2409
766	696	724	784	850	800	805	810	694	746	847
2021	2083	2275	2539	2233	2615	2611	2383	2393	2165	2261
2841	3276	2751	2675	2784	2779	2739	3103	3176	3225	2876
2125	2302	2280	2081	2133	1465	2612	2325	1970	1948	2199
399	3994	4019	4009	3878	564	351	476	364	501	4027
2776	3271	2861	858	879	3122	3218	866	253	97	3053
2272	2512	2501	2189	2208	2518	2219	2029	2048	2291	2605
3768	3614	3468	3584	3623	3568	3758	3534	3645	3958	3593
3708	3701	3675	3617	4020	3995	3985	476	3564	393	3774
435	478	445	558	500	544	349	444	541	452	634
3293	3029	3433	2929	3166	3362	2944	2873	3364	3220	2839
1542	1826	960	1321	1040	1201	1287	1485	1717	1712	1283
3087	3219	3397	3369	3203	3285	2921	3032	3167	3366	3241
3845	3828	3477	3658	3979	3888	3838	3576	3889	3860	3578
1600	1331	1520	1891	1683	1832	1370	1038	1431	1799	1272
148	291	324	307	119	108	88	339	260	322	21
2358	2230	2618	2099	2342	2114	2571	2371	2474	2313	2570
282	642	581	4029	1034	3746	1208	244	3704	4032	3846
869	811	735	769	767	753	831	740	738	748	882
423	506	652	604	414	395	385	544	507	524	1777
351	4019	4009	476	393	501	3994	3834	564	441	3675

User_ID	Friend_11	Friend_12	Friend_13	Friend_14	Friend_15	Friend_16	Friend_17	Friend_18	Friend_19	Friend_20	Friend_21	Friend_22	Friend_23	Friend_
1506	375	353	1631	363	1358	1210	416	484	1200	1455	352	1013	3	
1081	1495	1640	1103	1706	1680	1636	985	1221	1120	1404	1090	1066	14	
2057	1981	2635	2419	2224	2215	2177	2479	2453	2344	2473	2296	2264	20	
1883	1398	1290	1380	1605	1752	966	1399	1769	1132	1327	1003	1772	12	
212	200	176	393	204	302	156	50	323	334	134	268	59	2	
2014	49	2217	2113	312	255	2450	115	14	219	111	116	162	1	
2499	2258	2542	2353	2409	2492	2467	2058	2550	2329	2484	2604	2552	19	
3956	3466	3634	3646	3855	3714	3631	3711	3924	3506	3555	3464	3558	3443	39
489	547	418	478	458	361	444	421	521	354	646	634	500	403	502	5
1932	2629	2039	2239	2555	2148	2612	2134	1970	2413	2054	2072	2588	2332	25
2723	2939	3188	2716	3347	3021	2990	3416	3169	3154	3140	3016	2708	3330	28
2199	2629	1932	1970	2612	2148	2463	2330	2332	2138	2042	2351	2183	2196	2259	24
2894	2690	3148	3060	3182	2797	3113	3172	3261	3332	3411	2708	3201	2976	32
1727	1229	1167	1503	1525	1100	1872	1261	1903	1709	1725	1869	964	1044	18
1700	1653	1539	1811	1467	1610	1775	1632	1361	934	1800	1572	1401	1683	10
2885	2879	91	2704	3314	2740	3999	904	3855	3469	4005	2703	3283	3008	34
2911	3364	3362	3363	2977	3065	3135	3104	2755	2944	3033	3149	3158	3129	32
1691	1661	1786	1759	1507	1448	1595	1649	1756	898	933	1090	1158	1573	16
1249	1350	1139	1303	1392	1904	1246	1796	1472	1649	1166	1032	1103	11
3477	3788	3545	3656	3930	3640	3526	3557	3873	3790	3514	3707	3845	3471	36
1865	960	1712	1780	1521	1805	978	1484	1283	1150	1863	1851	1063	1329	18
4017	3668	3965	4000	3459	4026	3782	3510	3493	3489	3998	3976	3646	3752	39
630	437	658	628	654	617	532	640	655	597	684	643	592	633	5
1210	1606	366	936	1679	1645	1358	1313	1114	1673	514	1300	1616	1320	4
2327	2399	1959	2081	2461	2196	2294	2531	2187	2592	2247	2183	2279	2132	21
3609	3651	3750	3550	3684	3633	3886	3962	3556	3943	3674	3741	3440	3756	35
777	800	845	739	840	734	836	816	850	696	805	724	746	772	847	7
2414	2409	2549	2258	2560	2554	2492	2392	2654	2329	2408	2646	2055	2601	2136	21
694	746	847	736	773	711	834	853	756	824	827	726	823	843	734	792	7
2393	2165	2261	2573	2600	1986	1963	2206	2303	2526	2575	2625	2112	1917	2339	2046	25
3176	3225	2876	2845	3189	2811	2964	2958	3210	2775	3248	3306	3088	2700	3024	2702	28
1970	1948	2199	2198	2351	2592	2072	2038	2028	2519	2294	2174	2629	2344	2461	2436	25
364	501	4027	3834	3985	3988	3996	4038	393	3981	3675	3995	4011	3701	3689	4020	40
253	97	3053	895	3216	3105	870	2799	3354	3200	3141	871	327	2808	2902	95	31
2048	2291	2605	2110	1952	2651	2242	1982	2345	1987	2572	2298	2163	2007	1974	2222	25
3958	3593	3818	3829	3869	3866	3906	3569	3793	3915	3794	3951	3927	3822	3620	36
393	3774	3957	4018	3878	4009	3994	441	3571	4002	3476	3465	3870	4019	4023	36
541	452	634	521	388	646	458	421	395	523	671	557	361	546	426	415	5
3364	3220	2839	2887	2890	3065	3284	2920	3174	3417	3129	3149	2862	3084	2977	2815	30
1717	1712	1283	1902	1219	1618	1521	1063	978	1874	1551	1391	1083	1460	1858	1101	10
3167	3366	3241	2711	3360	3358	2932	3379	3062	3019	2829	2938	3076	2863	3422	3274	31
3889	3860	3578	3697	3966	3471	3930	3448	3475	3640	3660	3796	3707	3574	3485	3642	35
1431	1799	1272	1603	1547	1735	1886	1288	1369	1750	1126	1181	1509	1724	1557	1028	18
260	322	21	329	185	280	1	242	53	246	284	158	9	184	105	26	1
2474	2313	2570	2412	2361	2008	2089	2089	2070	1960	2577	2488	2181	1933	2126	2129	1975	25
3704	4032	3846	90	179	4012	4006	4001	3987	3814	2788	3445	145	3871	750	153	1
738	748	882	690	764	813	793	715	785	780	763	806	723	820	755	782	7
507	524	1777	402	370	415	460	669	392	500	683	465	434	520	542	676	4
564	441	3675	4013	3701	364	399	3995	4027	4018	3957	3878	3774	4038	3708	3985	40

图 11: 好友推荐表

4 项目使用

这里我们将各个函数的调用都集成在了 recommend.py 中，也就是说，该文件即为 main 程序，用户需要首先在 train.py 中选择使用的模型，再通过在 terminal 运行 python recommend.py 命令，等待训练和测试评估完成（输出所有的评估指标结果），会输出 recommend(n).xlsx，即为好友推荐结果。

5 总结

在社交网络好友推荐的场景下，比较 GraFRank 算法与 SAGE（GraphSAGE）和 Word2Vec 算法，可以从以下几个方面进行分析：

1. GraFRank 算法核心特性：GraFRank 是一种专为社交网络设计的图神经网络（GNN），特别强调处理多模态用户特征和链接特征。优势：能够有效处理社交网络的复杂结构和动态性，特别是在处理用户间的直接和间

接关系方面。同时，GraFRank 通过跨模态注意力机制，能够更好地理解不同类型的用户交互。适用场景：特别适用于大规模的社交网络，其中用户交互丰富且多样。

2. GraphSAGE（SAGE）核心特性：GraphSAGE 是一种基于采样的图神经网络方法，用于生成节点嵌入。优势：能够有效地学习大型图中节点的低维表示，特别是在缺乏全局图结构信息的情况下。适用于动态图，能够处理不断增加的节点和边。适用场景：适合在大规模且动态变化的网络中进行节点分类和预测，但在处理用户间复杂的社交关系上可能不如 GraFRank 精确。

3. Word2Vec 核心特性：Word2Vec 是一种自然语言处理中使用的词嵌入方法，通常用于将词语转换为向量形式。优势：虽然主要用于文本数据，但 Word2Vec 也可以应用于社交网络分析，将用户或者用户的行为模式转化为嵌入向量，用于捕捉用户间的相似性。适用场景：适用于需要分析和处理大量文本数据的社交网络场景，如基于用户发布内容的推荐系统。综合比较复杂关系处理：GraFRank 在处理复杂的社交关系和多模态特征方面优于 SAGE 和 Word2Vec。动态图适应性：SAGE 在处理动态图方面表现较好，适合快速变化的社交网络。用户行为分析：Word2Vec 在将用户行为转化为向量表示，用于分析用户间相似性方面有其独特优势。场景特异性：GraFRank 更适合社交网络中的复杂交互和好友推荐，SAGE 和 Word2Vec 虽然可用于社交网络分析，但在某些方面可能不如 GraFRank 专业。

总的来说，GraFRank 在社交网络好友推荐方面具有显著优势，尤其是在处理复杂的用户关系和多模态数据方面。而 SAGE 和 Word2Vec 虽然也可用于类似场景，但它们在处理社交网络特定问题方面可能不如 GraFRank 专业和精确。

参考文献

[1] Sankar, Aravind and Liu, Yozen and Yu, Jun and Shah, Neil. *Graph Neural Networks for Friend Ranking in Large-scale Social Platforms*. Proceedings of the Web Conference 2021 (WWW '21), 2021.

[2] Hamilton, William L. and Ying, Rex and Leskovec, Jure. *Inductive Representation Learning on Large Graphs*. Proceedings of the 31st Conference on Neural Information Processing Systems, 2017.

[3] W Hamilton, Z Ying, J Leskovec. *Inductive representation learning on large graphs*. Advances in neural information processing systems, 2017.

[4] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, Jie Tang. *Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec*. WSDM '18: Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining:Pages 459–467, 2018.

[5] Aditya Grover, Jure Leskovec. *node2vec: Scalable Feature Learning for Networks*. KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining:Pages 855–864, 2016.

[6] Aravind Sankar, Yozen Liu, Jun Yu, Neil Shah. *Graph Neural Networks for Friend Ranking in Large-scale Social Platforms*. WWW '21: Proceedings of the Web Conference 2021:Pages 2535–2546, 2021.