BiNE

理论

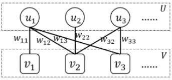
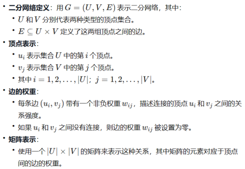


Figure 2: An example of the bipartite network structure

表示法



- **二分网络定义**：用 $G = (U, V, E)$ 表示二分网络，其中：
  - U 和 V 分别代表两种类型的顶点集合。
  - $E \subseteq U \times V$ 定义了这两组顶点之间的边。
- **顶点表示**：
  - u_i 表示集合 U 中的第 i 个顶点。
  - v_j 表示集合 V 中的第 j 个顶点。
  - 其中 $i = 1, 2, \ldots, |U|$； $j = 1, 2, \ldots, |V|$。
- **边的权重**：
  - 每条边 (u_i, v_j) 带有一个非负权重 w_{ij}，描述连接的顶点 u_i 和 v_j 之间的关系强度。
  - 如果 u_i 和 v_j 之间没有连接，则边的权重 w_{ij} 被设置为零。
- **矩阵表示**：
  - 使用一个 $|U| \times |V|$ 的矩阵来表示这种关系，其中矩阵的元素对应于顶点间的边的权重。

1. 模型改进

1-1 两个目标函数关系

1. 显示关系：显示关系捕捉的是上图中的U和V两个集合中的，简单理解就是捕捉二部图的两种不同类型节点之间的关系
2.隐式关系：隐式关系是捕捉上图中U和U集合中的节点关系，以及v和v集合中的关系，这种关系是捕捉同类型的节点之间的关系

1-2 随机游走策略

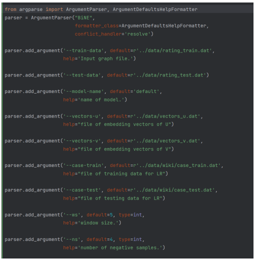
1. 自适应生成随机游走序列， 该自适应生成是依据与其节点的重要性来设定的，也就是重要的节点应当进行更多次次随机游走
2.随机游走的长度的不同，对比起原始的设定好的随机游走的长度，这部分是通过概率的方式来确定是否停止随机游走，通过这样的方式更加合理

2. 具体阐述模型改进点

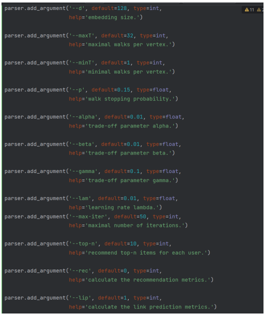
2-1 模型显示关系

代码实现

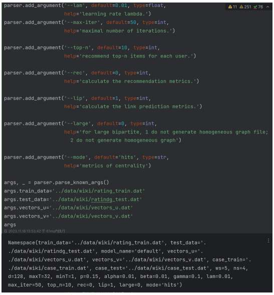
1. 形参设定



```python
from argparse import ArgumentParser, ArgumentDefaultsHelpFormatter
parser = ArgumentParser("BiNE",
                        formatter_class=ArgumentDefaultsHelpFormatter,
                        conflict_handler='resolve')

parser.add_argument('--train-data', default=r'../data/rating_train.dat',
                    help='Input graph file.')

parser.add_argument('--test-data', default=r'../data/rating_test.dat')

parser.add_argument('--model-name', default='default',
                    help='name of model.')

parser.add_argument('--vectors-u', default=r'../data/vectors_u.dat',
                    help="file of embedding vectors of U")

parser.add_argument('--vectors-v', default=r'../data/vectors_v.dat',
                    help="file of embedding vectors of V")

parser.add_argument('--case-train', default=r'../data/wiki/case_train.dat',
                    help="file of training data for LR")

parser.add_argument('--case-test', default=r'../data/wiki/case_test.dat',
                    help="file of testing data for LR")

parser.add_argument('--ws', default=5, type=int,
                    help='window size.')

parser.add_argument('--ns', default=4, type=int,
                    help='number of negative samples.')
```



```python
parser.add_argument('--d', default=128, type=int,
                    help='embedding size.')

parser.add_argument('--maxT', default=32, type=int,
                    help='maximal walks per vertex.')

parser.add_argument('--minT', default=1, type=int,
                    help='minimal walks per vertex.')

parser.add_argument('--p', default=0.15, type=float,
                    help='walk stopping probability.')

parser.add_argument('--alpha', default=0.01, type=float,
                    help='trade-off parameter alpha.')

parser.add_argument('--beta', default=0.01, type=float,
                    help='trade-off parameter beta.')

parser.add_argument('--gamma', default=0.1, type=float,
                    help='trade-off parameter gamma.')

parser.add_argument('--lam', default=0.01, type=float,
                    help='learning rate lambda.')
parser.add_argument('--max-iter', default=50, type=int,
                    help='maximal number of iterations.')

parser.add_argument('--top-n', default=10, type=int,
                    help='recommend top-n items for each user.')

parser.add_argument('--rec', default=0, type=int,
                    help='calculate the recommendation metrics.')

parser.add_argument('--lip', default=1, type=int,
                    help='calculate the link prediction metrics.')
```

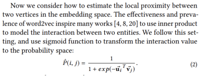


```python
parser.add_argument('--lam', default=0.01, type=float,
                    help='learning rate lambda.')
parser.add_argument('--max-iter', default=50, type=int,
                    help='maximal number of iterations.')

parser.add_argument('--top-n', default=10, type=int,
                    help='recommend top-n items for each user.')

parser.add_argument('--rec', default=0, type=int,
                    help='calculate the recommendation metrics.')

parser.add_argument('--lip', default=1, type=int,
                    help='calculate the link prediction metrics.')

parser.add_argument('--large', default=0, type=int,
                    help='for large bipartite, 1 do not generate homogeneous graph file;
                      2 do not generate homogeneous graph')

parser.add_argument('--mode', default='hits', type=str,
                    help='metrics of centrality')

args, _ = parser.parse_known_args()
args.train_data='../data/wiki/rating_train.dat'
args.test_data='../data/wiki/ratindg_test.dat'
args.vectors_u='../data/wiki/vectors_u.dat'
args.vectors_v='../data/wiki/vectors_v.dat'
args
```

在 2023.11.18 13:53:42 于 61ms内执行

```
Namespace(train_data='../data/wiki/rating_train.dat', test_data='.
./data/wiki/ratindg_test.dat', model_name='default', vectors_u='.
./data/wiki/vectors_u.dat', vectors_v='../data/wiki/vectors_v.dat', case_train='.
./data/wiki/case_train.dat', case_test='../data/wiki/case_test.dat', ws=5, ns=4,
d=128, maxT=32, minT=1, p=0.15, alpha=0.01, beta=0.01, gamma=0.1, lam=0.01,
max_iter=50, top_n=10, rec=0, lip=1, large=0, mode='hits')
```

2-1-1连接概率计算

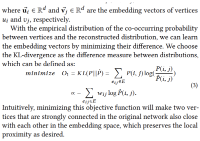


Now we consider how to estimate the local proximity between two vertices in the embedding space. The effectiveness and prevalence of word2vec inspire many works [4, 8, 20] to use inner product to model the interaction between two entities. We follow this setting, and use sigmoid function to transform the interaction value to the probability space:

$$\hat{P}(i,j) = \frac{1}{1 + exp(-\vec{\mathbf{u}}_i{}^T \vec{\mathbf{v}}_j)}. \tag{2}$$

通过两个嵌入后的两个向量之间的转置相乘并且通过sigmod函数来得到两个节点之间连接的概率

2-1-2 KL-Divergence



where $\vec{\mathbf{u}}_i \in \mathbb{R}^d$ and $\vec{\mathbf{v}}_j \in \mathbb{R}^d$ are the embedding vectors of vertices u_i and v_j, respectively.

With the empirical distribution of the co-occurring probability between vertices and the reconstructed distribution, we can learn the embedding vectors by minimizing their difference. We choose the KL-divergence as the difference measure between distributions, which can be defined as:

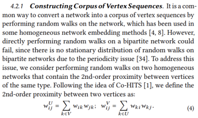
$$\begin{aligned} minimize \quad O_1 = KL(P\|\hat{P}) &= \sum_{e_{ij} \in E} P(i,j) \log\left(\frac{P(i,j)}{\hat{P}(i,j)}\right) \\ &\propto - \sum_{e_{ij} \in E} w_{ij} \log \hat{P}(i,j). \end{aligned} \tag{3}$$

Intuitively, minimizing this objective function will make two vertices that are strongly connected in the original network also close with each other in the embedding space, which preserves the local proximity as desired.

通过最小化顶点共现概率的实证分布和重构分布之间的差异来学习嵌入向量

2-2 模型隐式关系

2-2-1 隐式关系的构建

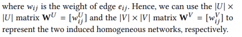


4.2.1 ***Constructing Corpus of Vertex Sequences.*** It is a common way to convert a network into a corpus of vertex sequences by performing random walks on the network, which has been used in some homogeneous network embedding methods [4, 8]. However, directly performing random walks on a bipartite network could fail, since there is no stationary distribution of random walks on bipartite networks due to the periodicity issue [34]. To address this issue, we consider performing random walks on two homogeneous networks that contain the 2nd-order proximity between vertices of the same type. Following the idea of Co-HITS [1], we define the 2nd-order proximity between two vertices as:

$$w_{ij}^U = \sum_{k \in V} w_{ik} w_{jk}; \quad w_{ij}^V = \sum_{k \in U} w_{ki} w_{kj}. \tag{4}$$

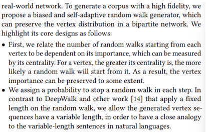
这里使用的方法不是直接在二部图上进行随机游走而去获取训练语料而是通过转化为两张同构图去进行随机游走的，这种方式主要是捕捉二阶相似性。

2-2-2 两张同构图的表示法



where w_{ij} is the weight of edge e_{ij}. Hence, we can use the $|U| \times |U|$ matrix $\mathbf{W}^U = [w_{ij}^U]$ and the $|V| \times |V|$ matrix $\mathbf{W}^V = [w_{ij}^V]$ to represent the two induced homogeneous networks, respectively.

2-2-3 进行有策略的随机游走

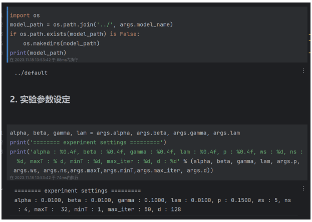


real-world network. To generate a corpus with a high fidelity, we propose a biased and self-adaptive random walk generator, which can preserve the vertex distribution in a bipartite network. We highlight its core designs as follows:

- First, we relate the number of random walks starting from each vertex to be dependent on its importance, which can be measured by its centrality. For a vertex, the greater its centrality is, the more likely a random walk will start from it. As a result, the vertex importance can be preserved to some extent.
- We assign a probability to stop a random walk in each step. In contrast to DeepWalk and other work [14] that apply a fixed length on the random walk, we allow the generated vertex sequences have a variable length, in order to have a close analogy to the variable-length sentences in natural languages.

这里直接指定了使用的训练数据是用wiki的数据集，已经处理好的

2. 实验参数设定



```
import os
model_path = os.path.join('../', args.model_name)
if os.path.exists(model_path) is False:
    os.makedirs(model_path)
print(model_path)
```

在 2023.11.18 13:53:42 于 88ms内执行

```
../default
```

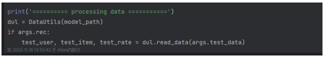
2. 实验参数设定

```
alpha, beta, gamma, lam = args.alpha, args.beta, args.gamma, args.lam
print('======== experiment settings =========')
print('alpha : %0.4f, beta : %0.4f, gamma : %0.4f, lam : %0.4f, p : %0.4f, ws : %d, ns :
 %d, maxT : % d, minT : %d, max_iter : %d, d : %d' % (alpha, beta, gamma, lam, args.p,
 args.ws, args.ns,args.maxT,args.minT,args.max_iter, args.d))
```

在 2023.11.18 13:53:42 于 74ms内执行

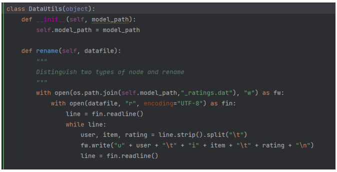
```
======== experiment settings =========
alpha : 0.0100, beta : 0.0100, gamma : 0.1000, lam : 0.0100, p : 0.1500, ws : 5, ns
 : 4, maxT :  32, minT : 1, max_iter : 50, d : 128
```

3. 数据加载

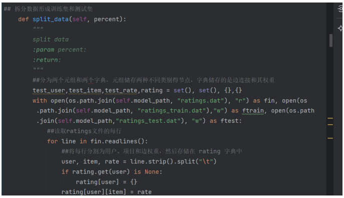


```
print('========== processing data ===========')
dul = DataUtils(model_path)
if args.rec:
    test_user, test_item, test_rate = dul.read_data(args.test_data)
```

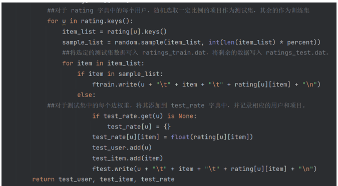
在 2023.11.18 13:53:42 于 45ms内执行



```
class DataUtils(object):
    def __init__(self, model_path):
        self.model_path = model_path

    def rename(self, datafile):
        """
        Distinguish two types of node and rename
        """
        with open(os.path.join(self.model_path,"_ratings.dat"), "w") as fw:
            with open(datafile, "r", encoding="UTF-8") as fin:
                line = fin.readline()
                while line:
                    user, item, rating = line.strip().split("\t")
                    fw.write("u" + user + "\t" + "i" + item + "\t" + rating + "\n")
                    line = fin.readline()
```



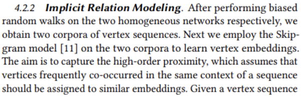
```
## 拆分数据形成训练集和测试集
    def split_data(self, percent):
        """
        split data
        :param percent:
        :return:
        """
        ##分为两个元组和两个字典，元组储存两种不同类别得节点，字典储存的是边连接和其权重
        test_user,test_item,test_rate,rating = set(), set(), {},{}
        with open(os.path.join(self.model_path, "ratings.dat"), "r") as fin, open(os
         .path.join(self.model_path, "ratings_train.dat"),"w") as ftrain, open(os.path
         .join(self.model_path,"ratings_test.dat"), "w") as ftest:
            ##读取ratings文件的每行
            for line in fin.readlines():
                ##将每行分割为用户、项目和边权重，然后存储在 rating 字典中
                user, item, rate = line.strip().split("\t")
                if rating.get(user) is None:
                    rating[user] = {}
                rating[user][item] = rate
```



```
            ##对于 rating 字典中的每个用户，随机选取一定比例的项目作为测试集，其余的作为训练集
            for u in rating.keys():
                item_list = rating[u].keys()
                sample_list = random.sample(item_list, int(len(item_list) * percent))
                ##将选定的测试集数据写入 ratings_train.dat，将剩余的数据写入 ratings_test.dat。
                for item in item_list:
                    if item in sample_list:
                        ftrain.write(u + "\t" + item + "\t" + rating[u][item] + "\n")
                    else:
            ##对于测试集中的每个边权重，将其添加到 test_rate 字典中，并记录相应的用户和项目。
                        if test_rate.get(u) is None:
                            test_rate[u] = {}
                        test_rate[u][item] = float(rating[u][item])
                        test_user.add(u)
                        test_item.add(item)
                        ftest.write(u + "\t" + item + "\t" + rating[u][item] + "\n")
        return test_user, test_item, test_rate
```

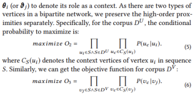
1. 随机游走次数设定：每个节点发生多少次随机游走是由它的重要性所决定的，该重要性也就是它的中心性。
2.随机游走长度设定：这个随机游走的长度是可变的是通过一个概率值设定随机游走停止是否停止。

2-2-4通过skip-gram来学习顶点嵌入方式


4.2.2 ***Implicit Relation Modeling***. After performing biased random walks on the two homogeneous networks respectively, we obtain two corpora of vertex sequences. Next we employ the Skip-gram model [11] on the two corpora to learn vertex embeddings. The aim is to capture the high-order proximity, which assumes that vertices frequently co-occurred in the same context of a sequence should be assigned to similar embeddings. Given a vertex sequence

使用Skip-gram模型对这两个顶点序列的语料库进行处理，目的是学习顶点嵌入。这个过程的目标是捕捉高阶邻近性，假设在同一个序列上下文中频繁共现的顶点应该被赋予相似的嵌入。

2-2-5 保持高阶邻近性


$\boldsymbol{\theta}_i$ (or $\boldsymbol{\vartheta}_j$) to denote its role as a context. As there are two types of vertices in a bipartite network, we preserve the high-order proximities separately. Specifically, for the corpus D^U, the conditional probability to maximize is:

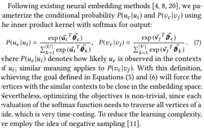
$$maximize\ O_2 = \prod_{u_i \in S \wedge S \in D^U} \prod_{u_c \in C_S(u_i)} P(u_c|u_i). \tag{5}$$

where $C_S(u_i)$ denotes the context vertices of vertex u_i in sequence S. Similarly, we can get the objective function for corpus D^V:

$$maximize\ O_3 = \prod_{v_j \in S \wedge S \in D^V} \prod_{v_c \in C_S(v_j)} P(v_c|v_j). \tag{6}$$

在二部网络中，由于存在两种类型的顶点U和V，分别为这两种顶点保持高阶邻近性，因此需要最大化语料库Du的条件概率P（uc|ui）同理对于Dv也要这样进行

2-2-6 条件概率参数化

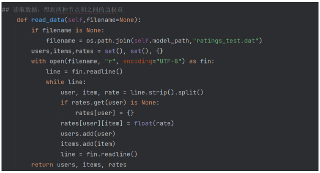

Following existing neural embedding methods [4, 8, 20], we parameterize the conditional probability $P(u_c|u_i)$ and $P(v_c|v_j)$ using he inner product kernel with softmax for output:

$$P(u_c|u_i) = \frac{\exp(\vec{\mathbf{u}}_i{}^T \vec{\boldsymbol{\theta}}_c)}{\sum_{k=1}^{|U|} \exp(\vec{\mathbf{u}}_i{}^T \vec{\boldsymbol{\theta}}_k)}, \quad P(v_c|v_j) = \frac{\exp(\vec{\mathbf{v}}_j{}^T \vec{\boldsymbol{\vartheta}}_c)}{\sum_{k=1}^{|V|} \exp(\vec{\mathbf{v}}_j{}^T \vec{\boldsymbol{\vartheta}}_k)}. \tag{7}$$

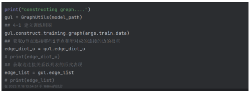
where $P(u_c|u_i)$ denotes how likely u_c is observed in the contexts of u_i; similar meaning applies to $P(v_c|v_j)$. With this definition, achieving the goal defined in Equations (5) and (6) will force the vertices with the similar contexts to be close in the embedding space. Nevertheless, optimizing the objectives is non-trivial, since each evaluation of the softmax function needs to traverse all vertices of a side, which is very time-costing. To reduce the learning complexity, we employ the idea of negative sampling [11].

通过内积核与softmax输出来参数化，这里条件概率是在ui上下文观察到uc的可能性，这种区其最大从而保持同类型系欸但之间的高阶邻近性

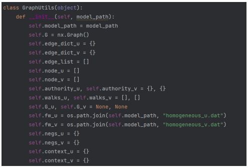
2-2-7 负采样LSH


```
## 读取数据，得到两种节点和之间的边权重
    def read_data(self,filename=None):
        if filename is None:
            filename = os.path.join(self.model_path,"ratings_test.dat")
        users,items,rates = set(), set(), {}
        with open(filename, "r", encoding="UTF-8") as fin:
            line = fin.readline()
            while line:
                user, item, rate = line.strip().split()
                if rates.get(user) is None:
                    rates[user] = {}
                rates[user][item] = float(rate)
                users.add(user)
                items.add(item)
                line = fin.readline()
        return users, items, rates
```

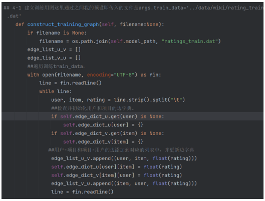
4. 图加载类


```
print("constructing graph....")
gul = GraphUtils(model_path)
## 4-1 建立训练用图
gul.construct_training_graph(args.train_data)
## 获取u节点连接哪些i节点和所对应的连接的边的权重
edge_dict_u = gul.edge_dict_u
# print(edge_dict_u)
## 获取边连接关系以列表的形式表现
edge_list = gul.edge_list
# print(edge_list)
在 2023.11.18 13:54:57 于 168ms内执行
```

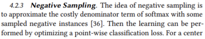
初始化


```
class GraphUtils(object):
    def __init__(self, model_path):
        self.model_path = model_path
        self.G = nx.Graph()
        self.edge_dict_u = {}
        self.edge_dict_v = {}
        self.edge_list = []
        self.node_u = []
        self.node_v = []
        self.authority_u, self.authority_v = {}, {}
        self.walks_u, self.walks_v = [], []
        self.G_u, self.G_v = None, None
        self.fw_u = os.path.join(self.model_path, "homogeneous_u.dat")
        self.fw_v = os.path.join(self.model_path, "homogeneous_v.dat")
        self.negs_u = {}
        self.negs_v = {}
        self.context_u = {}
        self.context_v = {}
```

4-1 construct_training_graph建立训练用图


```
## 4-1 建立训练用图这里通过之间我的预设即传入的文件是args.train_data='../data/wiki/rating_train
 .dat'
    def construct_training_graph(self, filename=None):
        if filename is None:
            filename = os.path.join(self.model_path, "ratings_train.dat")
        edge_list_u_v = []
        edge_list_v_u = []
        ##遍历训练train_data.
        with open(filename, encoding="UTF-8") as fin:
            line = fin.readline()
            while line:
                user, item, rating = line.strip().split("\t")
                ##检查并初始化用户和项目的边字典.
                if self.edge_dict_u.get(user) is None:
                    self.edge_dict_u[user] = {}
                if self.edge_dict_v.get(item) is None:
                    self.edge_dict_v[item] = {}
               ##用户-项目和项目-用户的边添加到对应的列表中，并更新边字典
                edge_list_u_v.append((user, item, float(rating)))
                self.edge_dict_u[user][item] = float(rating)
                self.edge_dict_v[item][user] = float(rating)
                edge_list_v_u.append((item, user, float(rating)))
                line = fin.readline()
```

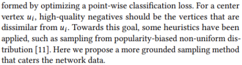
2-2-7-1 该部分使用负采样的意义



4.2.3 ***Negative Sampling***. The idea of negative sampling is to approximate the costly denominator term of softmax with some sampled negative instances [36]. Then the learning can be performed by optimizing a point-wise classification loss. For a center

负采样的目的是通过采样一些负实例来近似softmax的代价高昂的分母项。通过这种方式，学习过程可以转化为优化点分类损失

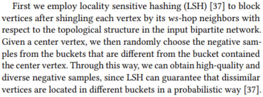
2-2-7-2 如何选择高质量的负样本



formed by optimizing a point-wise classification loss. For a center vertex u_i, high-quality negatives should be the vertices that are dissimilar from u_i. Towards this goal, some heuristics have been applied, such as sampling from popularity-biased non-uniform distribution [11]. Here we propose a more grounded sampling method that caters the network data.

对于中心顶点ui，高质量的负样本应该是与ui不相似的顶点。为了达到这个目标，采用了一些启发式方法，比如从受欢迎程度偏差的非均匀分布中采样。

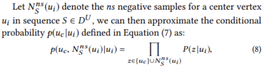
2-2-7-3 采样的负采样方式（LSH）



First we employ locality sensitive hashing (LSH) [37] to block vertices after shingling each vertex by its *ws*-hop neighbors with respect to the topological structure in the input bipartite network. Given a center vertex, we then randomly choose the negative samples from the buckets that are different from the bucket contained the center vertex. Through this way, we can obtain high-quality and diverse negative samples, since LSH can guarantee that dissimilar vertices are located in different buckets in a probabilistic way [37].

使用局部敏感哈希（LSH）技术来分块顶点，每个顶点根据其在输入二部网络中的ws-跳邻居进行分块。然后，从包含中心顶点的块中随机选择负样本。这种方法可以获得高质量且多样化的负样本，因为LSH能够以概率方式保证不相似的顶点位于不同的块中。

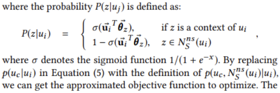
2-2-7-4 替换条件概率即对2-2-6去进行替换



Let $N_S^{ns}(u_i)$ denote the *ns* negative samples for a center vertex u_i in sequence $S \in D^U$, we can then approximate the conditional probability $p(u_c|u_i)$ defined in Equation (7) as:

$$p(u_c, N_S^{ns}(u_i)|u_i) = \prod_{z \in \{u_c\} \cup N_S^{ns}(u_i)} P(z|u_i), \tag{8}$$

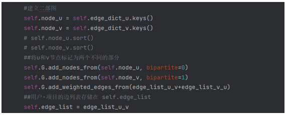
2-2-8 根据正采样和负采样的方式去替换2-2-5



where the probability $P(z|u_j)$ is defined as:

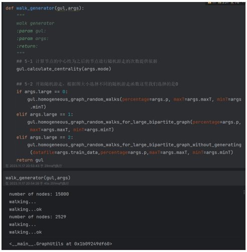
$$P(z|u_i) \quad = \quad \begin{cases} \sigma(\vec{\mathbf{u}}_i{}^T \vec{\boldsymbol{\theta}}_z), & \text{if } z \text{ is a context of } u_i \\ 1 - \sigma(\vec{\mathbf{u}}_i{}^T \vec{\boldsymbol{\theta}}_z), & z \in N_S^{ns}(u_i) \end{cases},$$

where σ denotes the sigmoid function $1/(1 + e^{-x})$. By replacing $p(u_c|u_i)$ in Equation (5) with the definition of $p(u_c, N_S^{ns}(u_i)|u_i)$, we can get the approximated objective function to optimize. The



```
#建立二部图
self.node_u = self.edge_dict_u.keys()
self.node_v = self.edge_dict_v.keys()
# self.node_u.sort()
# self.node_v.sort()
##将u和v节点标记为两个不同的部分
self.G.add_nodes_from(self.node_u, bipartite=0)
self.G.add_nodes_from(self.node_v, bipartite=1)
self.G.add_weighted_edges_from(edge_list_u_v+edge_list_v_u)
##用户-项目的边列表存储在 self.edge_list
self.edge_list = edge_list_u_v
```

5 随机游走生成器



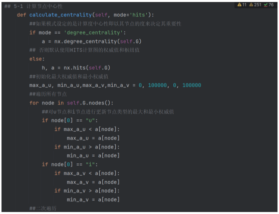
```
def walk_generator(gul,args):
    """
    walk generator
    :param gul:
    :param args:
    :return:
    """
    ## 5-1 计算节点的中心性为之后的节点进行随机游走的次数提供依据
    gul.calculate_centrality(args.mode)

    ## 5-2 开始随机游走，根据图大小选择不同的随机游走函数这里我们选择的是0
    if args.large == 0:
        gul.homogeneous_graph_random_walks(percentage=args.p, maxT=args.maxT, minT=args
         .minT)
    elif args.large == 1:
        gul.homogeneous_graph_random_walks_for_large_bipartite_graph(percentage=args.p,
         maxT=args.maxT, minT=args.minT)
    elif args.large == 2:
        gul.homogeneous_graph_random_walks_for_large_bipartite_graph_without_generating
         (datafile=args.train_data,percentage=args.p,maxT=args.maxT, minT=args.minT)
    return gul
在 2023.11.17 20:53:43 于 29ms内执行
```

```
walk_generator(gul,args)
在 2023.11.17 20:54:28 于 45s 251ms内执行
```

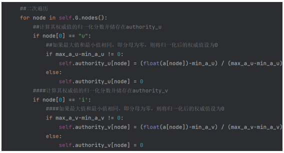
```
number of nodes: 15000
walking...
walking...ok
number of nodes: 2529
walking...
walking...ok

<__main__.GraphUtils at 0x1b09249df60>
```

之后调用的函数都是在GraphUtils这个类中的，因为是在生成的图上进行随机游走

5-1 计算节点中心性



```
## 5-1 计算节点中心性
    def calculate_centrality(self, mode='hits'):
        ##如果模式设定的是计算度中心性即以其节点的度来决定其重要性
        if mode == 'degree_centrality':
            a = nx.degree_centrality(self.G)
        ## 否则默认使用HITS计算图的权威值和枢纽值
        else:
            h, a = nx.hits(self.G)
        ##初始化最大权威值和最小权威值
        max_a_u, min_a_u,max_a_v,min_a_v = 0, 100000, 0, 100000
        ##遍历所有节点
        for node in self.G.nodes():
            ##对u节点和i节点进行更新节点类型的最大和最小权威值
            if node[0] == "u":
                if max_a_u < a[node]:
                    max_a_u = a[node]
                if min_a_u > a[node]:
                    min_a_u = a[node]
            if node[0] == "i":
                if max_a_v < a[node]:
                    max_a_v = a[node]
                if min_a_v > a[node]:
                    min_a_v = a[node]
        ##二次遍历
```

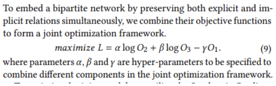


```
        ##二次遍历
        for node in self.G.nodes():
            ##计算其权威值的归一化分数并储存在authority_u
            if node[0] == "u":
                ##如果最大值和最小值相同，即分母为零，则将归一化后的权威值设为0
                if max_a_u-min_a_u != 0:
                    self.authority_u[node] = (float(a[node])-min_a_u) / (max_a_u-min_a_u)
                else:
                    self.authority_u[node] = 0
            ####计算其权威值的归一化分数并储存在authority_v
            if node[0] == 'i':
                ####如果最大值和最小值相同，即分母为零，则将归一化后的权威值设为0
                if max_a_v-min_a_v != 0:
                    self.authority_v[node] = (float(a[node])-min_a_v) / (max_a_v-min_a_v)
                else:
                    self.authority_v[node] = 0
```

5-2 开始随机游走

最大化上下文顶点之间的接近度，同时最小化负样本之间的接近度

3 联合优化

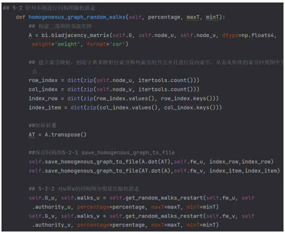


To embed a bipartite network by preserving both explicit and implicit relations simultaneously, we combine their objective functions to form a joint optimization framework.

$$maximize\ L = \alpha \log O_2 + \beta \log O_3 - \gamma O_1. \qquad (9)$$

where parameters α, β and γ are hyper-parameters to be specified to combine different components in the joint optimization framework.

O2和O3是对于隐式关系的代表，而O1是代表的显示关系。通过设定超参数来对其进行权重调节该部分用的是SGA是随机梯度上升，因为我们需要优化的是目标函数因此选择SGA而不是SGD

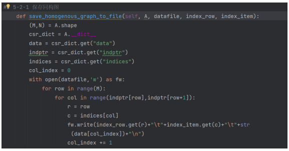


```
## 5-2 针对小图进行同构图随机游走
    def homogeneous_graph_random_walks(self, percentage, maxT, minT):
        ## 构建二部图的邻接矩阵
        A = bi.biadjacency_matrix(self.G, self.node_u, self.node_v, dtype=np.float64,
         weight='weight', format='csr')

        ## 建立索引映射，创造字典来映射行索引和列索引的节点并且进行反向索引，从而从矩阵的索引回到图中
         点
        row_index = dict(zip(self.node_u, itertools.count()))
        col_index = dict(zip(self.node_v, itertools.count()))
        index_row = dict(zip(row_index.values(), row_index.keys()))
        index_item = dict(zip(col_index.values(), col_index.keys()))

        ##矩阵转置
        AT = A.transpose()

        ##保存同构图5-2-1 save_homogenous_graph_to_file
        self.save_homogenous_graph_to_file(A.dot(AT),self.fw_u, index_row,index_row)
        self.save_homogenous_graph_to_file(AT.dot(A),self.fw_v, index_item,index_item)

        ## 5-2-2 对u和v的同构图分别进行随机游走
        self.G_u, self.walks_u = self.get_random_walks_restart(self.fw_u, self
         .authority_u, percentage=percentage, maxT=maxT, minT=minT)
        self.G_v, self.walks_v = self.get_random_walks_restart(self.fw_v, self
         .authority_v, percentage=percentage, maxT=maxT, minT=minT)
```

首先先会建立同构图（5-2-1），随后根据同构图进行随机游走（5-2-2）从而为后续隐形关系的捕捉提供基础。

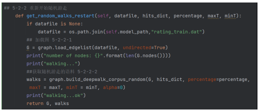
这部分的u同构图的构成是通过A.dot(AT)。v的同构图是通过AT.dot(A)形成的

5-2-1 保存同构图



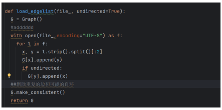
```
# 5-2-1 保存同构图
    def save_homogenous_graph_to_file(self, A, datafile, index_row, index_item):
        (M,N) = A.shape
        csr_dict = A.__dict__
        data = csr_dict.get("data")
        indptr = csr_dict.get("indptr")
        indices = csr_dict.get("indices")
        col_index = 0
        with open(datafile,'w') as fw:
            for row in range(M):
                for col in range(indptr[row],indptr[row+1]):
                    r = row
                    c = indices[col]
                    fw.write(index_row.get(r)+"\t"+index_item.get(c)+"\t"+str
                     (data[col_index])+"\n")
                    col_index += 1
```

5-2-2 开始随机游走



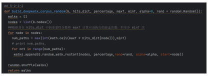
```
## 5-2-2 重新开始随机游走
    def get_random_walks_restart(self, datafile, hits_dict, percentage, maxT, minT):
        if datafile is None:
            datafile = os.path.join(self.model_path,"rating_train.dat")
        ## 加载图 5-2-2-1
        G = graph.load_edgelist(datafile, undirected=True)
        print("number of nodes: {}".format(len(G.nodes())))
        print("walking...")
        ##获取随机游走的语料 5-2-2-2
        walks = graph.build_deepwalk_corpus_random(G, hits_dict, percentage=percentage,
         maxT = maxT, minT = minT, alpha=0)
        print("walking...ok")
        return G, walks
```

5-2-2-1 加载图中连接的节点信息



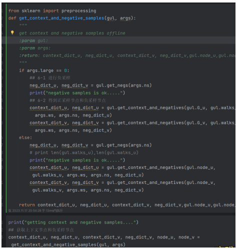
```
def load_edgelist(file_, undirected=True):
  G = Graph()
  #addddddd
  with open(file_,encoding="UTF-8") as f:
    for l in f:
      x, y = l.strip().split()[:2]
      G[x].append(y)
      if undirected:
        G[y].append(x)
  ##删除重复的边和可能的自环
  G.make_consistent()
  return G
```

5-2-2-2 获取随机游走的语料



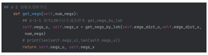
```
## 5-2-2-2
def build_deepwalk_corpus_random(G, hits_dict, percentage, maxT, minT, alpha=0, rand = random.Random()):
  walks = []
  nodes = list(G.nodes())
  ##根据其在 hits_dict 中的重要性分数和 maxT 计算应该执行的游走次数，但至少 minT 次
  for node in nodes:
    num_paths = max(int(math.ceil(maxT * hits_dict[node])),minT)
    # print num_paths,
    for cnt in range(num_paths):
     walks.append(G.random_walk_restart(nodes, percentage,rand=rand, alpha=alpha, start=node))

  random.shuffle(walks)
  return walks
```

6 基于随机游走采样的语料进行上下文节点（正样本）负采样获取



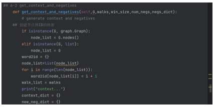
```
from sklearn import preprocessing
def get_context_and_negative_samples(gul, args):
    """
    get context and negative samples offline
    :param gul:
    :param args:
    :return: context_dict_u, neg_dict_u, context_dict_v, neg_dict_v,gul.node_u,gul.no
    """
    if args.large == 0:
        ## 6-1 进行负采样
        neg_dict_u, neg_dict_v = gul.get_negs(args.ns)
        print("negative samples is ok.....")
        ## 6-2 得到正采样节点和负采样节点
        context_dict_u, neg_dict_u = gul.get_context_and_negatives(gul.G_u, gul.walks
          args.ws, args.ns, neg_dict_u)
        context_dict_v, neg_dict_v = gul.get_context_and_negatives(gul.G_v, gul.walks
          args.ws, args.ns, neg_dict_v)
    else:
        neg_dict_u, neg_dict_v = gul.get_negs(args.ns)
        # print len(gul.walks_u),len(gul.walks_u)
        print("negative samples is ok.....")
        context_dict_u, neg_dict_u = gul.get_context_and_negatives(gul.node_u,
         gul.walks_u, args.ws, args.ns, neg_dict_u)
        context_dict_v, neg_dict_v = gul.get_context_and_negatives(gul.node_v,
         gul.walks_v, args.ws, args.ns, neg_dict_v)

    return context_dict_u, neg_dict_u, context_dict_v, neg_dict_v,gul.node_u,gul.node
在 2023.11.17 20:54:28 于 13ms内执行

print("getting context and negative samples....")
## 获取上下文节点和负采样节点
context_dict_u, neg_dict_u, context_dict_v, neg_dict_v, node_u, node_v =
 get_context_and_negative_samples(gul, args)
```

6-1 进行负采样

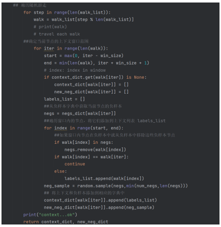


```
## 6-1 获取负采样节点
    def get_negs(self,num_negs):
        ## 6-1-1 使用LSH方法生成负样本 get_negs_by_lsh
        self.negs_u, self.negs_v = get_negs_by_lsh(self.edge_dict_u,self.edge_dict_v,
         num_negs)
        # print(len(self.negs_u),len(self.negs_v))
        return self.negs_u, self.negs_v
```

6-2 得到上下文节点和负采样节点

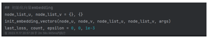


```
## 6-2 get_context_and_negatives
    def get_context_and_negatives(self,G,walks,win_size,num_negs,negs_dict):
        # generate context and negatives
    ## 创建节点到ID的映射
        if isinstance(G, graph.Graph):
            node_list = G.nodes()
        elif isinstance(G, list):
            node_list = G
        word2id = {}
        node_list=list(node_list)
        for i in range(len(node_list)):
            word2id[node_list[i]] = i + 1
        walk_list = walks
        print("context...")
        context_dict = {}
        new_neg_dict = {}
```

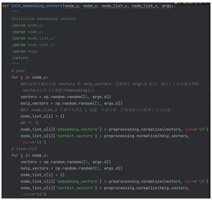


```
    ## 遍历随机游走
        for step in range(len(walk_list)):
            walk = walk_list[step % len(walk_list)]
            # print(walk)
            # travel each walk
        ##确定当前节点的上下文窗口范围
            for iter in range(len(walk)):
                start = max(0, iter - win_size)
                end = min(len(walk), iter + win_size + 1)
                # index: index in window
                if context_dict.get(walk[iter]) is None:
                    context_dict[walk[iter]] = []
                    new_neg_dict[walk[iter]] = []
                labels_list = []
                ##从负样本字典中获取当前节点的负样本
                negs = negs_dict[walk[iter]]
                ##遍历窗口内的节点，将它们添加到上下文列表 labels_list
                for index in range(start, end):
                    ##如果窗口内节点在负样本中就从负样本中移除这些负样本节点
                    if walk[index] in negs:
                        negs.remove(walk[index])
                    if walk[index] == walk[iter]:
                        continue
                    else:
                        labels_list.append(walk[index])
                neg_sample = random.sample(negs,min(num_negs,len(negs)))
                ## 将上下文和负样本添加到相应的字典中
                context_dict[walk[iter]].append(labels_list)
                new_neg_dict[walk[iter]].append(neg_sample)
        print("context...ok")
        return context_dict, new_neg_dict
```

7. 对节点embedding进行初始化

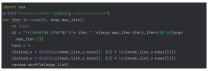


```
## 初始化向量embedding
node_list_u, node_list_v = {}, {}
init_embedding_vectors(node_u, node_v, node_list_u, node_list_v, args)
last_loss, count, epsilon = 0, 0, 1e-3
在 2023.11.17 20:57:28 于 2m 59s 965ms内执行
```

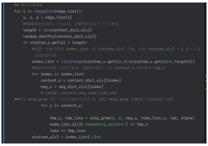


```
def init_embedding_vectors(node_u, node_v, node_list_u, node_list_v, args):
    """
    initialize embedding vectors
    :param node_u:
    :param node_v:
    :param node_list_u:
    :param node_list_v:
    :param args:
    :return:
    """
    # user
    for i in node_u:
        ##创建两个随机向量 vectors 和 help_vectors，其维度由 args.d 指定，通过上下文向量去帮助
         vectors去学习并调整其embedding表示
        vectors = np.random.random([1, args.d])
        help_vectors = np.random.random([1, args.d])
        ##在 node_list_u 字典中为节点 i 创建一个新字典，存储其嵌入向量和上下文向量
        node_list_u[i] = {}
        ## 归一化
        node_list_u[i]['embedding_vectors'] = preprocessing.normalize(vectors, norm='l2')
        node_list_u[i]['context_vectors'] = preprocessing.normalize(help_vectors,
         norm='l2')
    # item——同理
    for i in node_v:
        vectors = np.random.random([1, args.d])
        help_vectors = np.random.random([1, args.d])
        node_list_v[i] = {}
        node_list_v[i]['embedding_vectors'] = preprocessing.normalize(vectors, norm='l2')
        node_list_v[i]['context_vectors'] = preprocessing.normalize(help_vectors,
         norm='l2')
```

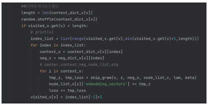
8. 正式开始训练



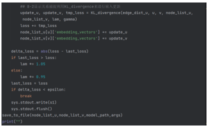
```
import sys
print("============== training ==============")
for iter in range(0, args.max_iter):
    ## 初始化
    s1 = "\r[%s%s]%0.2f%%"%("*"* iter," "*(args.max_iter-iter),iter*100.0/(args
     .max_iter-1))
    loss = 0
    visited_u = dict(zip(node_list_u.keys(), [0] * len(node_list_u.keys())))
    visited_v = dict(zip(node_list_v.keys(), [0] * len(node_list_v.keys())))
    random.shuffle(edge_list)
```



```
    ## 隐式关系捕捉
    for i in range(len(edge_list)):
        u, v, w = edge_list[i]
        ##获取用u节点的上下文长度，并随机打乱这个上下文列表
        length = len(context_dict_u[u])
        random.shuffle(context_dict_u[u])
        if visited_u.get(u) < length:
            #创建一个索引列表 index_list，从 visited_u[u] 开始，直到 visited_u[u] + 1 或上下文
             长度的最小值
            index_list = list(range(visited_u.get(u),min(visited_u.get(u)+1,length)))
            #遍历索引列表，对每个索引，获取对应的上下文 context_u 和负样本 neg_u
            for index in index_list:
                context_u = context_dict_u[u][index]
                neg_u = neg_dict_u[u][index]
                # center,context,neg,node_list,eta
    ##8-1 skip_gram 对于上下文中的每个节点 z，调用 skip_gram 函数进行节点u的嵌入更新
                for z in context_u:

                    tmp_z, tmp_loss = skip_gram(u, z, neg_u, node_list_u, lam, alpha)
                    node_list_u[z]['embedding_vectors'] += tmp_z
                    loss += tmp_loss
            visited_u[u] = index_list[-1]+3
```

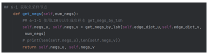


```
        ##同理在节点v上进行
        length = len(context_dict_v[v])
        random.shuffle(context_dict_v[v])
        if visited_v.get(v) < length:
            # print(v)
            index_list = list(range(visited_v.get(v),min(visited_v.get(v)+1,length)))
            for index in index_list:
                context_v = context_dict_v[v][index]
                neg_v = neg_dict_v[v][index]
                # center,context,neg,node_list,eta
                for z in context_v:
                    tmp_z, tmp_loss = skip_gram(v, z, neg_v, node_list_v, lam, beta)
                    node_list_v[z]['embedding_vectors'] += tmp_z
                    loss += tmp_loss
            visited_v[v] = index_list[-1]+3
```

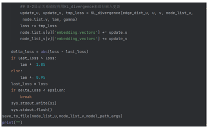


```
        ## 8-2显示关系捕捉利用KL_divergence来进行嵌入更新
        update_u, update_v, tmp_loss = KL_divergence(edge_dict_u, u, v, node_list_u,
         node_list_v, lam, gamma)
        loss += tmp_loss
        node_list_u[u]['embedding_vectors'] += update_u
        node_list_v[v]['embedding_vectors'] += update_v

    delta_loss = abs(loss - last_loss)
    if last_loss > loss:
        lam *= 1.05
    else:
        lam *= 0.95
    last_loss = loss
    if delta_loss < epsilon:
        break
    sys.stdout.write(s1)
    sys.stdout.flush()
save_to_file(node_list_u,node_list_v,model_path,args)
print("")
```

8-1 skip_gram(更新两种隐式关系)



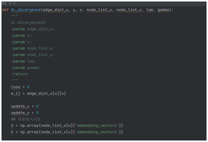
```
## 6-1 获取负采样节点
    def get_negs(self,num_negs):
        ## 6-1-1 使用LSH方法生成负样本 get_negs_by_lsh
        self.negs_u, self.negs_v = get_negs_by_lsh(self.edge_dict_u,self.edge_dict_v,
         num_negs)
        # print(len(self.negs_u),len(self.negs_v))
        return self.negs_u, self.negs_v
```



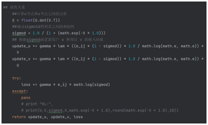
```
        ## 8-2显示关系捕捉利用KL_divergence来进行嵌入更新
        update_u, update_v, tmp_loss = KL_divergence(edge_dict_u, u, v, node_list_u,
         node_list_v, lam, gamma)
        loss += tmp_loss
        node_list_u[u]['embedding_vectors'] += update_u
        node_list_v[v]['embedding_vectors'] += update_v

    delta_loss = abs(loss - last_loss)
    if last_loss > loss:
        lam *= 1.05
    else:
        lam *= 0.95
    last_loss = loss
    if delta_loss < epsilon:
        break
    sys.stdout.write(s1)
    sys.stdout.flush()
save_to_file(node_list_u,node_list_v,model_path,args)
print("")
```

用于更新词嵌入和上下文向量，同时计算损失。它使用负采样方法优化损失函数，通过调整嵌入向量和上下文向量来最大化中心词和上下文词的联合概率，同时最小化中心词和负样本词的联合概率。这种方法在自然语言处理和词嵌入领域非常有效。

8-2 KL_divergence （在两种隐式关系更新后更新两种节点间的显式关系）



```
## 8-2
def KL_divergence(edge_dict_u, u, v, node_list_u, node_list_v, lam, gamma):
    """
    KL-divergence01
    :param edge_dict_u:
    :param u:
    :param v:
    :param node_list_u:
    :param node_list_v:
    :param lam:
    :param gamma:
    :return:
    """
    loss = 0
    e_ij = edge_dict_u[u][v]

    update_u = 0
    update_v = 0
    ## 获取嵌入向量
    U = np.array(node_list_u[u]['embedding_vectors'])
    V = np.array(node_list_v[v]['embedding_vectors'])
```



```
## 显性关系
    ##计算u节点和v节点之间的点积
    X = float(U.dot(V.T))
    ##通过sigmoid得到其之间的相似性
    sigmod = 1.0 / (1 + (math.exp(-X * 1.0)))
    ## 根据sigmod函更新用户 u 和项目 v 的嵌入向量
    update_u += gamma * lam * ((e_ij * (1 - sigmod)) * 1.0 / math.log(math.e, math.e)) *
      V
    update_v += gamma * lam * ((e_ij * (1 - sigmod)) * 1.0 / math.log(math.e, math.e)) *
      U

    try:
        loss += gamma * e_ij * math.log(sigmod)
    except:
        pass
        # print "KL:",
        # print(U,V,sigmod,X,math.exp(-X * 1.0),round(math.exp(-X * 1.0),10))
    return update_u, update_v, loss
```

这个函数用于在推荐系统或类似的应用中更新用户和项目的嵌入向量。它根据两个节点间的相互作用（通过边权重 e_ij 表示）以及它们当前的嵌入状态，使用 KL 散度作为损失函数来优化嵌入向量。