图嵌入的技术

矩阵分解

拉普拉斯特征映射

原理和推导

思想：两个相似的节点在降维后的目标子空间中应相互接近

相关数学符号：邻接矩阵W中存储的的两个节点之间的相似度，设数据实例的数目为n，则W的大小为$n \times n$。设目标子空间即最终的降维目标的维度为m，则定义$m \times n$的矩阵Y，其中每一个行向量y_i^T是数据实例i在目标m维子空间中的向量表示（即降维后的数据实例i）

目标函数：$min \sum_{i \neq j}(y_i - y_j)^2 W_{ij}$ ，可以理解为对每对节点的距离施加一个惩罚，一对节点之间的相似度越大（W_{ij}越大），则对于相同距离施加的惩罚越大（对于相同的$(y_i - y_j)^2$越敏感）。

下面对目标函数进行拉普拉斯变换：

$$
\begin{aligned}
&\sum_{ij}(y_i - y_j)^2 W_{ij} \\
&= \sum_{i=1}^n \sum_{j=1}^n (y_i - y_j)^2 W_{ij} \\
&= \sum_{i=1}^n \sum_{j=1}^n (y_i^T y_i - 2y_i^T y_j + y_j^T y_j) W_{ij} \\
&= \sum_{i=1}^n (\sum_{j=1}^n W_{ij}) y_i^T y_i + \sum_{j=1}^n (\sum_{i=1}^n W_{ij}) y_j^T y_j - 2\sum_{i=1}^n \sum_{j=1}^n y_i^T y_j W_{ij} \\
&= 2\sum_{i=1}^n D_{ii} y_i^T y_i - 2\sum_{i=1}^n \sum_{j=1}^n y_i^T y_j W_{ij}
\end{aligned}
$$

(D_{ii}是图的度矩阵，即$D_{ii} = \sum_{j=1}^n W_{ij}$，其中包含的信息是每一个节点的度数（入度+出度）

上式继续化简:$= 2\sum_{i=1}^n (\sqrt{D_{ii}} y_i)^T (\sqrt{D_{ii}} y_i) - 2\sum_{i=1}^n y_i^T (\sum_{j=1}^n y_j W_{ij})$

又由矩阵乘法的定义和迹运算可得：

$$
\begin{aligned}
&= 2trace(Y^T DY) - 2\sum_{i=1}^n y_i^T (YW)_i \\
&= 2trace(Y^T DY) - 2trace(Y^T WY) \\
&= 2trace[Y^T (D - W)Y] \\
&= 2trace(Y^T LY)
\end{aligned}
$$

则L = D - W称为图的拉普拉斯矩阵，则变换后的目标函数为:
$mintrace(Y^T LY), s.t. Y^T DY = I$
其中限制田间$s.t. Y^T DY = I$保证优化条件有解，然后运用拉格朗日乘子法对目标函数求解：
$f(Y) = tr(Y^T LY) + tr[\wedge(Y^T DY - I)]$

$\frac{\partial f(Y)}{\partial Y} = LY + L^T Y + D^T Y \wedge^T + DY\wedge = 2LY + 2DY\wedge = 0$
则$LY = -DY\wedge$

其中∧是一个对角矩阵，L，D都是实对称矩阵，其转置矩阵与其本身相同，则上式可以写成$Ly = \lambda Dy$，而这是一个广义特征值问题。通过求得m个最小非零特征值所对应的特征向量，便可以达到降维的目的。

或者更加简单：
$min_{Y^T DY=1} Y^T LY = min\frac{Y^T LY}{Y^T DY} = max\frac{Y^T WY}{Y^T DY}$
这也转化为一个广义特征值问题$WY = \lambda DY$

算法和实现

- 构建图。有两种主要的方式：
  - 用KNN算法将训练集中的点连接起来
  - 设定一个阈值ϵ，当两个点之间的距离小于或等于这个阈值，将两个点连接起来
- 确定邻接矩阵W，W_{ij}。当i和j不相连的时候，$W_{ij} = 0$，当i，j相连的时候，可以选用热核函数来确定：$W_{ij} = e^{-\frac{||x_i - x_j||^2}{t}}$，其中$t$是可选参数，或者也可以用$W_{ij} = 1$来简化。

```
def createGraphByKNN(Map, k):
    nums = Map.shape[0]
    dimension = Map.shape[1]
    W = np.zeros((nums, nums))
    for i in range(nums):
        CurrNode = Map[i]
        Diff = np.tile(CurrNode, (nums, 1)) - Map
        SqDiff = Diff ** 2
        SqDistance = SqDiff.sum(axis=1)
        SortedDistance = SqDistance.argsort()
        for j in range(k):
            idx = SortedDistance[j + 1]
            W[i][idx] = math.exp(-(SqDistance[idx]) / 3) #3是可调参数t的值
            # W[i][idx] = 1
    return W
```

- 进行特征映射。先求出图的度数矩阵D，再求出拉普拉斯矩阵$L = D - W$，然后求解广义特征值问题$Ly = \lambda Dy$，也就是求解$D^{-1}Ly = \lambda y$的普通特征值问题，通得到过求解$D^{-1}L$矩阵的特征向量和特征值，使用最小的m个非零特征值对应的特征向量作为降维后的结果输出。

```
def LaplacianEigenmaps(W):
    print(W)
    nums = W.shape[0]
    D = np.zeros((nums, nums)) #求得图的度数矩阵
    for i in range(nums):
        CurrNodeEdges = W[i]
        degree = CurrNodeEdges.sum()
        D[i][i] = degree
    L = D - W #得到图的拉普拉斯矩阵
    # 进行广义特征值问题求解 Ly = aDy
    D_L = np.dot(np.linalg.inv(D), L)
    print(D_L)
    EigenValue, EigenVector = np.linalg.eig(D_L) #a是特征值，b是特征向量
    print(EigenValue, EigenVector)
    SortedEigenValue = EigenValue.argsort()
    Y = np.zeros((2, nums))
    counter = 0
    i = 0
    # 选择2个最小的特征值（非零）对应的特征向量组成嵌入矩阵
    while counter < 2:
        IdxOfEigenValue = SortedEigenValue[i]
        if EigenValue[IdxOfEigenValue] != 1:
            Y[counter] = EigenVector[IdxOfEigenValue]
            counter += 1
        i += 1
    TransposeOfY = Y.transpose()
    return TransposeOfY
```

实验结果

降维之前的图：

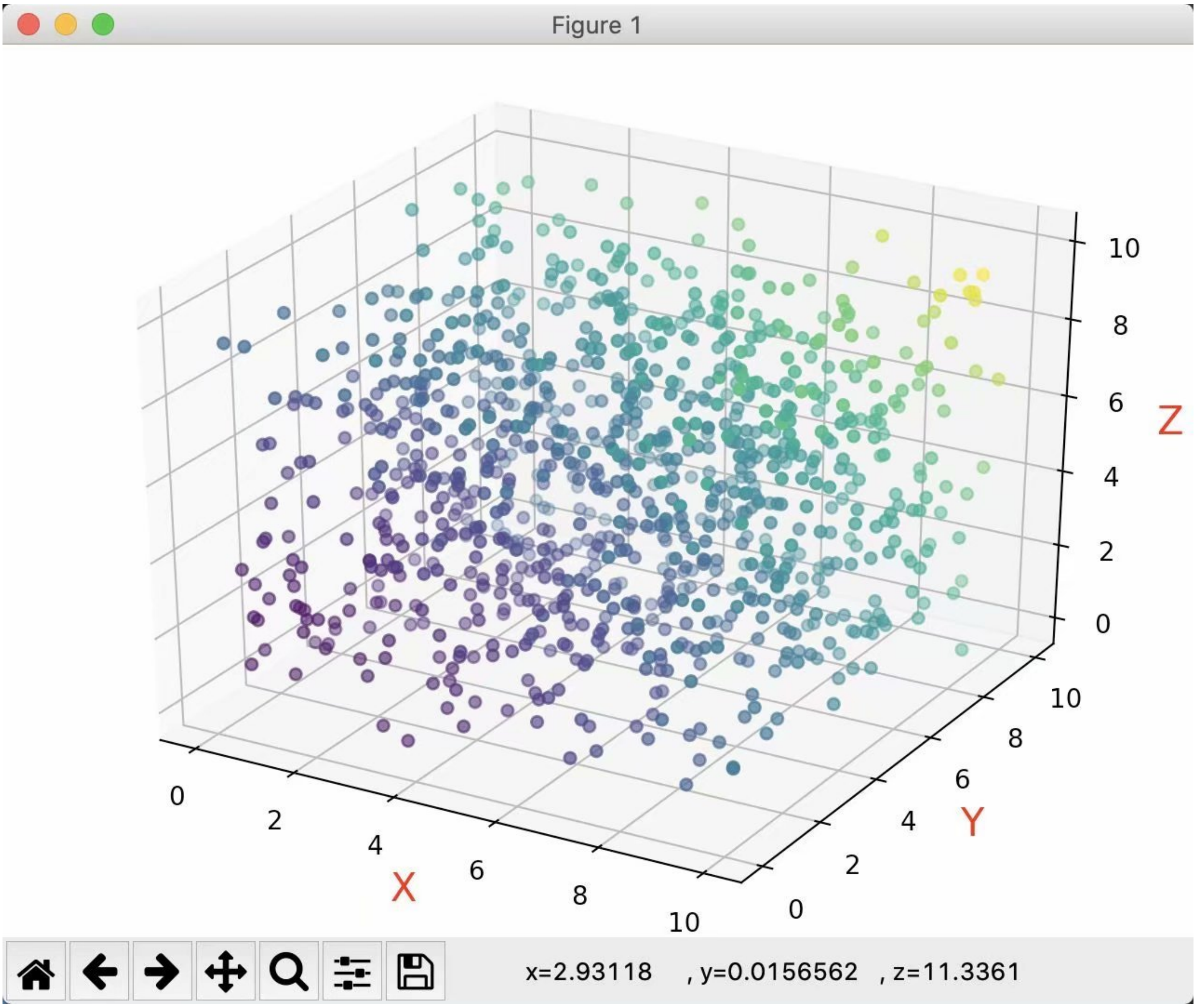
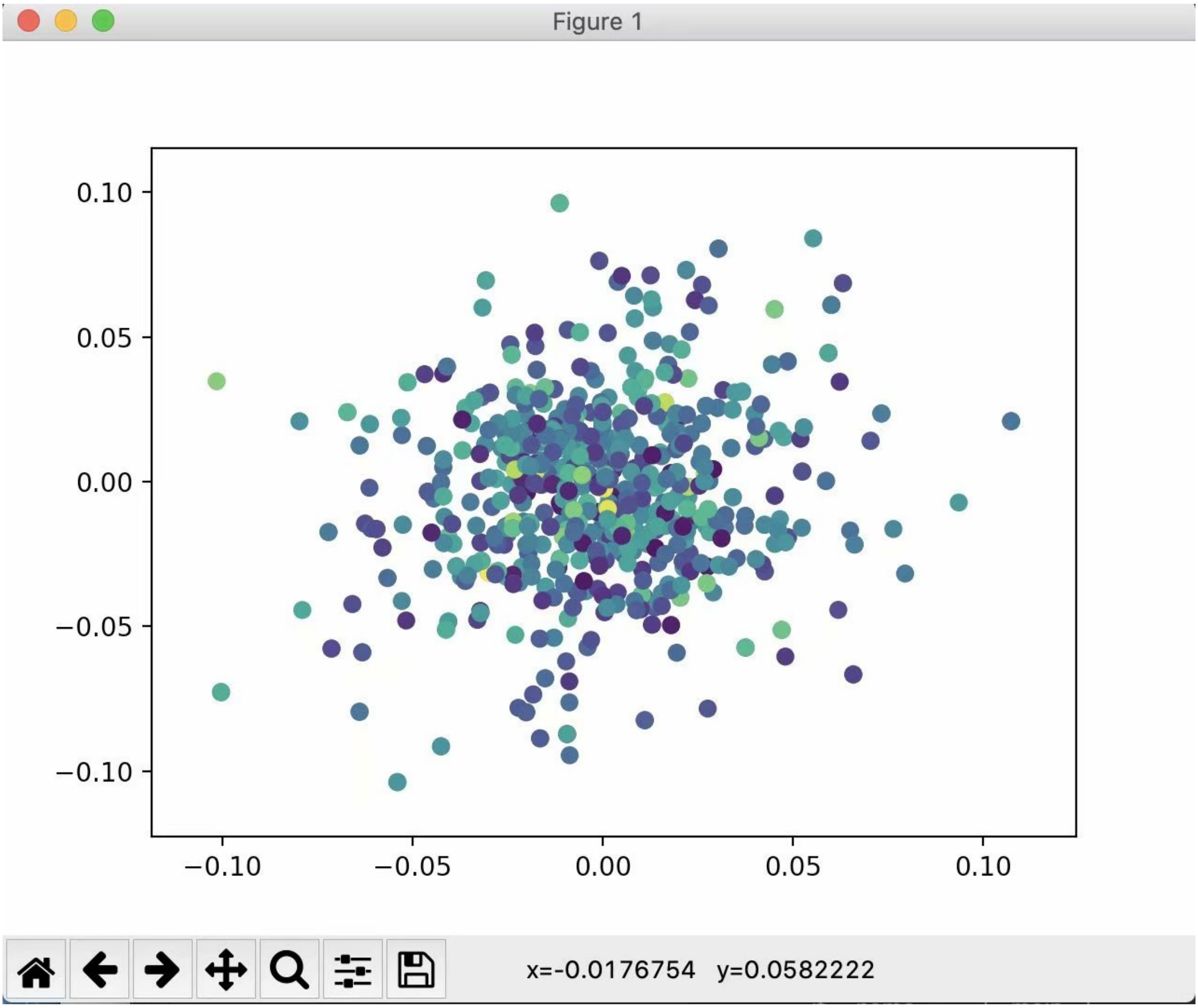


Figure 1
10
8
6
4
2
0
Z
0
2
4
6
8
10
X
0
2
4
6
8
10
Y
x=2.93118 , y=0.0156562 , z=11.3361

降维之后的图:



直接分解节点的邻接矩阵

思想：用矩阵分解将节点的相似性近似为低维向量

相关数学符号：W为节点的邻接矩阵，Y是节点的嵌入，Y^c是上下文节点的嵌入。

目标函数：$min||W - YY^{cT}||$，运用奇异值分解(SVD)，则

$W = \sum_{i=1}^{|V|} \sigma_i u_i u_i^{cT} \approx \sum_{i=1}^{d} \sigma_i u_i u_i^{cT}$，

其中d是嵌入后的维度，则使目标函数最小的Y和Y^T分别是：

$$Y = [\sqrt{\sigma_1} u_1, \ldots, \sqrt{\sigma_d} u_d],$$
$$Y^c = [\sqrt{\sigma_1} u_1^c, \ldots, \sqrt{\sigma_d} u_d^c]$$

其中u_i和u_i^c是σ_i的奇异向量，则对于节点i，它对应的嵌入便是Y_i

深度学习

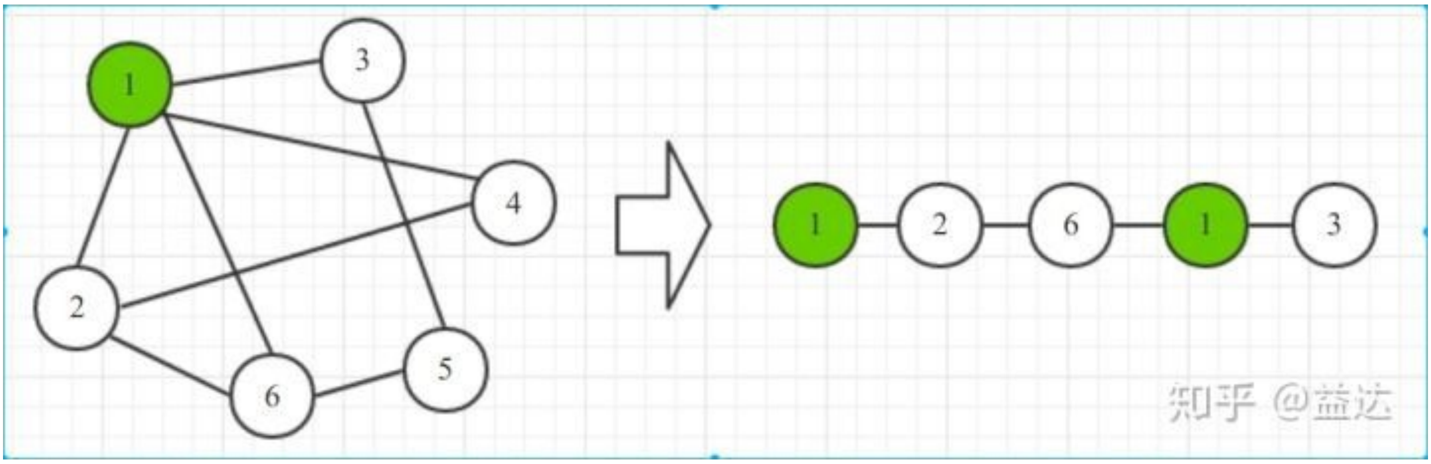
DeepWalk

原理

上述两种方法需要用到节点的邻接矩阵，但是如果遇到相对大一点的图（包含1000+个顶点），尽管可以用矩阵分解的方式获得每个节点长度相对较短的向量表示，但总的来说复杂度还是比较高。Deepwalk是另外一种更有效的解决方法。

基本思想：如果两个节点有着非常相似的邻接节点，那么通过模型训练，这两个节点的嵌入向量将非常相似。

Deepwalk从某一点出发，给定一个游走长度k，得到一条随机的游走路径：



然后应用word2vec模型，将每个节点看作一个单词，将随机采样得到的路径看作一个句子。假设我们采样得到的路径为$v_0, v_1, ..., v_n$,则我们需要优化的目标函数为：

$$P_r(v_n|v_0, v_1, ..., v_{n-1})$$

意思就是当知道$(v_0, v_1, ..., v_{n-1})$游走路径后，游走的下一个节点是v_i的概率。由于这里的v_i没法直接计算，于是引入一个映射函数Φ，将顶点映射为向量（这其实就是我们要求的），转化成向量后就可以对顶点v_i进行计算：

$$\Phi : v \in V- > R^{|V| \times d}$$

映射函数Φ对图中的每一个节点映射成d维向量，Φ实际上是一个矩阵，总共有$|V| \times d$个参数，这些参数就是需要学习的。

因此需要优化的目标函数可以写成：

$$P_r(v_n|(\Phi(v_0), \Phi(v_1), ..., \Phi(v_{n-1})))$$

但由于$(\Phi(v_0), \Phi(v_1), ..., \Phi(v_{n-1}))$这部分的概率太过难求，因此运用一个skip-gram模型，将优化目标转

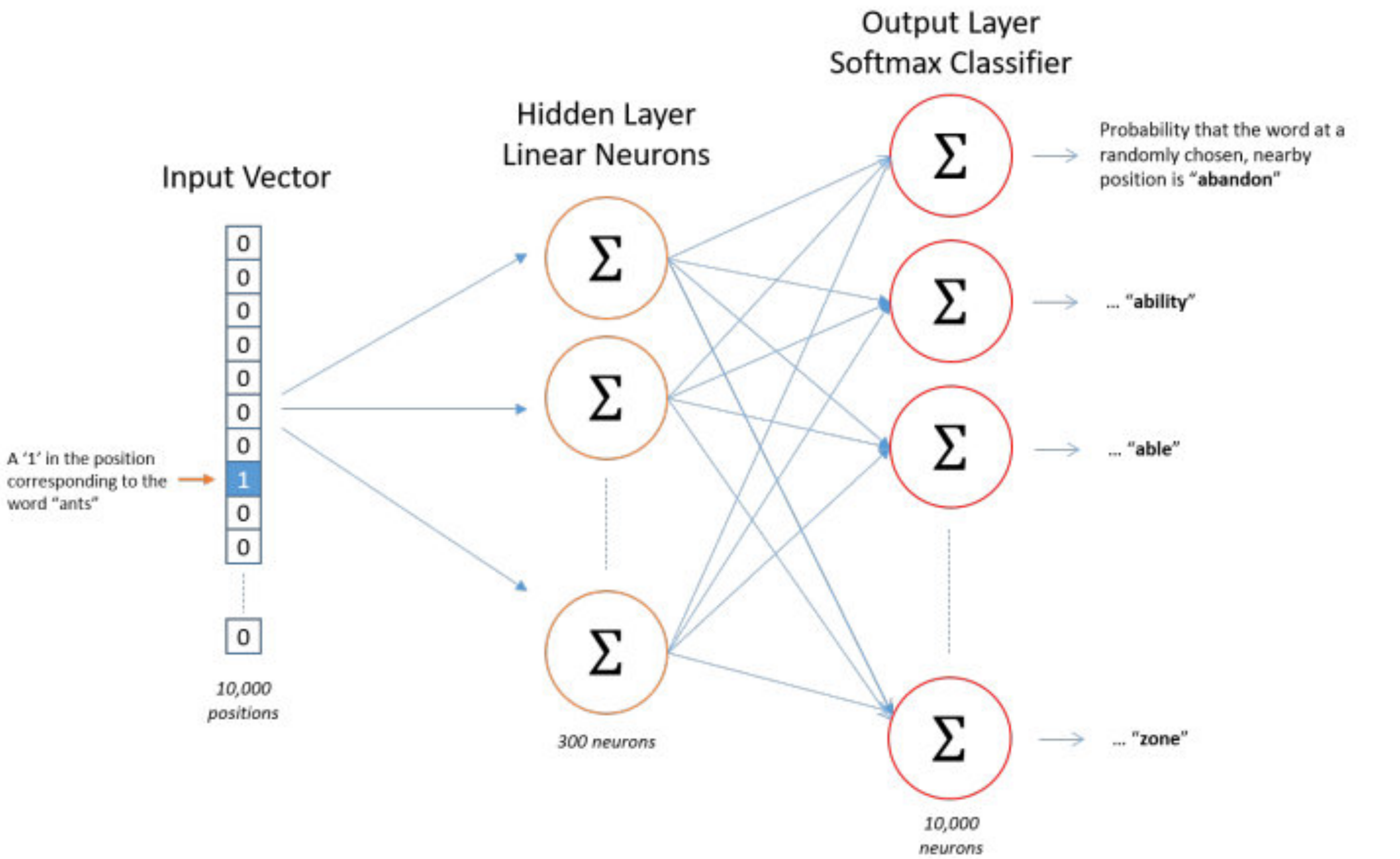
化为：

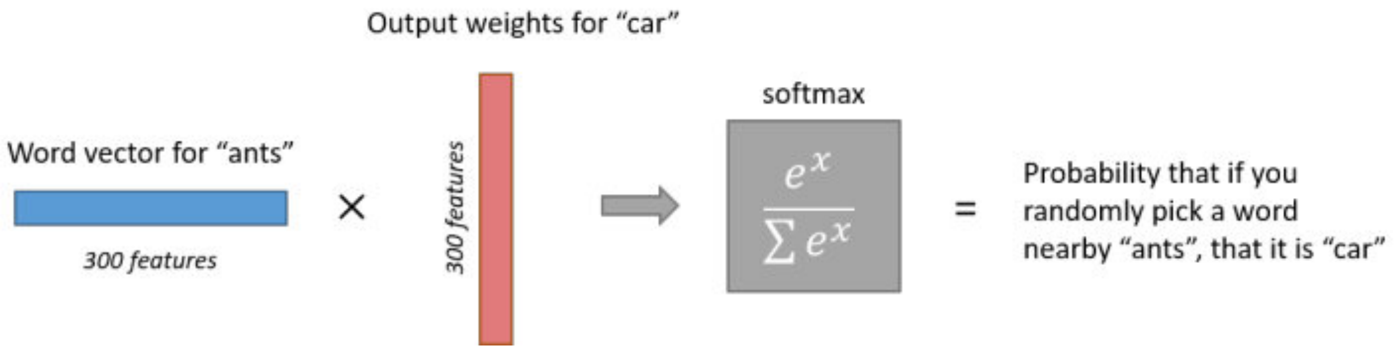
$$min_{\Phi}\ -logP_r(\{v_{i-w},...,v_{i-1},v_{i+1},...,v_{i+w}\}|\Phi(v_i))$$

也便是：

$$min_{\Phi}\ -log\sum_{-w\le j\le w}P(\Phi(v_{i+j})|\Phi(v_i))$$

其中$P(\Phi(v_{i+j})|\Phi(v_i))=\frac{exp(\Phi(v_{i+j})\Phi(v_i))}{\sum_{k=1}^{|V|}exp(\Phi(v_k)\Phi(v_i)))}$，对相关概率进行归一化。





但每次都用上式进行归一化计算代价太过昂贵，有两种解决方法：

- 层次softmax：构建一棵根据词频构建的哈夫曼树，其中每一片叶子是一个节点，其余每个子节点是一个二分类器，则每一个节点的概率为$P(\Phi(v_{i+j})|\Phi(v_i))=\prod_{t=1}^{log|V|}P(b_t|\Phi(v_i))$ 其中$P(b_t|\Phi(v_i))$是二分类器，表示在树节点b_t选择到树叶v_i的概率。所有树叶的概率加起来等于1，把复杂度从$O(|V|^2)$减到

了$O(|V|log|V|)$

- 负采样：关键思想是将原始softmax输出层的N个节点变为N个二分类器，就不存在softmax求和项。则可以得到:$P(\Phi(v_{i+j})|\Phi(v_i)) = log\sigma(\Phi(v_{i+j})\Phi(v_i)) + \sum_{t=1}^{K} E_{vt\ Pn}[log\sigma(-\Phi(v_t)\Phi(v_i))]$，其中K是通过采样得到的k个负样例得到的集合，因为输出层的二分类器需要同时接受“正例”和“负例”才能正确建模共现关系。复杂度为$O(k|V|)$

算法和实现

输入：顶点集V和边集E 输出：顶点的二维向量表示$embedding_{|V|\times 2}$

- 构建图。设定一个阈值ϵ,如果两个顶点之间的距离小于这个阈值就将这两个顶点连接起来。

```
def createGraph():
    V = np.random.random((100, 3)) * 10 #100个顶点
    # E = np.random.randint(10, size = (50, 2)) #50条边
    E = []
    nums = V.shape[0]
    for i in range(nums):
        for j in range(nums):
            if i != j:
                dif = (V[i][0] - V[j][0]) ** 2 + (V[i][1] - V[j][1]) ** 2 + (V[i][2]
                if dif <= 5:
                    E.append([i, j])


    fig = plt.figure()
    ax = Axes3D(fig)
    x = V[:, 0]
    y = V[:, 1]
    z = V[:, 2]
    color = x**2 + y**2 + z**2
    ax.scatter(x, y, z, c=color)
    for e in E:
        t_x = [x[e[0]], x[e[1]]]
        t_y = [y[e[0]], y[e[1]]]
        t_z = [z[e[0]], z[e[1]]]
        ax.plot(t_x, t_y, t_z, c="yellow")
    plt.show()
    return V, E, color
```

- 对图进行随机游走得到游走路径的集合。

```
for i in range(times_randomwalk):
        random.shuffle(nodes)
        for node in nodes:
            t_path = RandomWalk(V, Nbr, node, 10)
            # if len(t_path) >= 3:
            walk_path.append([str(i) for i in t_path])

def RandomWalk(V, Neighbors, start_node, walk_length):
    walk = [start_node]
    while len(walk) < walk_length:
        cur = walk[-1]
        cur_nbrs = Neighbors[cur]
        if len(cur_nbrs) > 0:
            walk.append(random.choice(cur_nbrs))
        else:
            break
    return walk
```

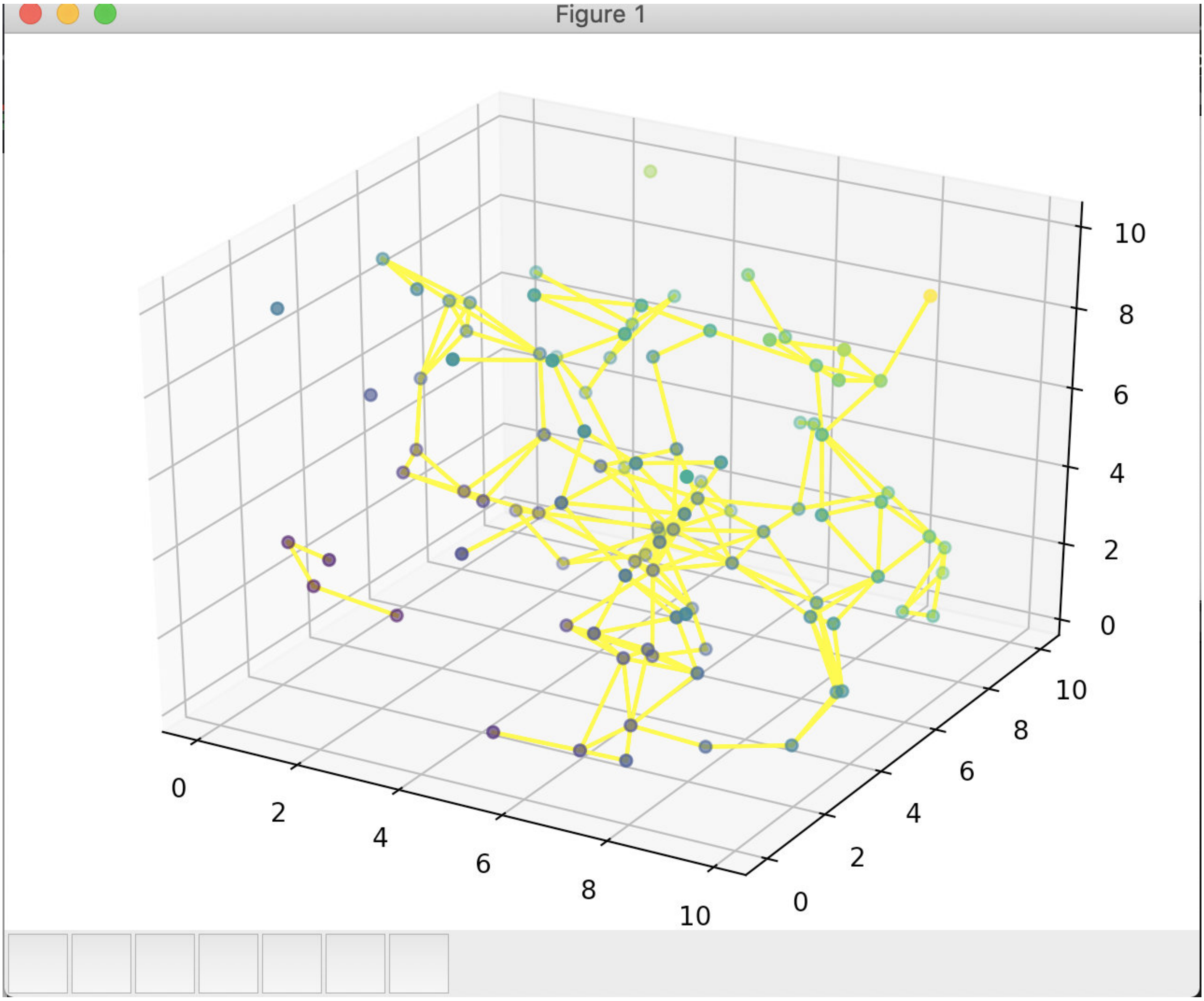
- 用得到的游走路径集合训练word2vec中的skip-gram模型，学习顶点的二维向量表示。

```
model = Word2Vec(walk_path, size=2, min_count=1, sg=1, window=5)
model.save('./Data/myModel')
```

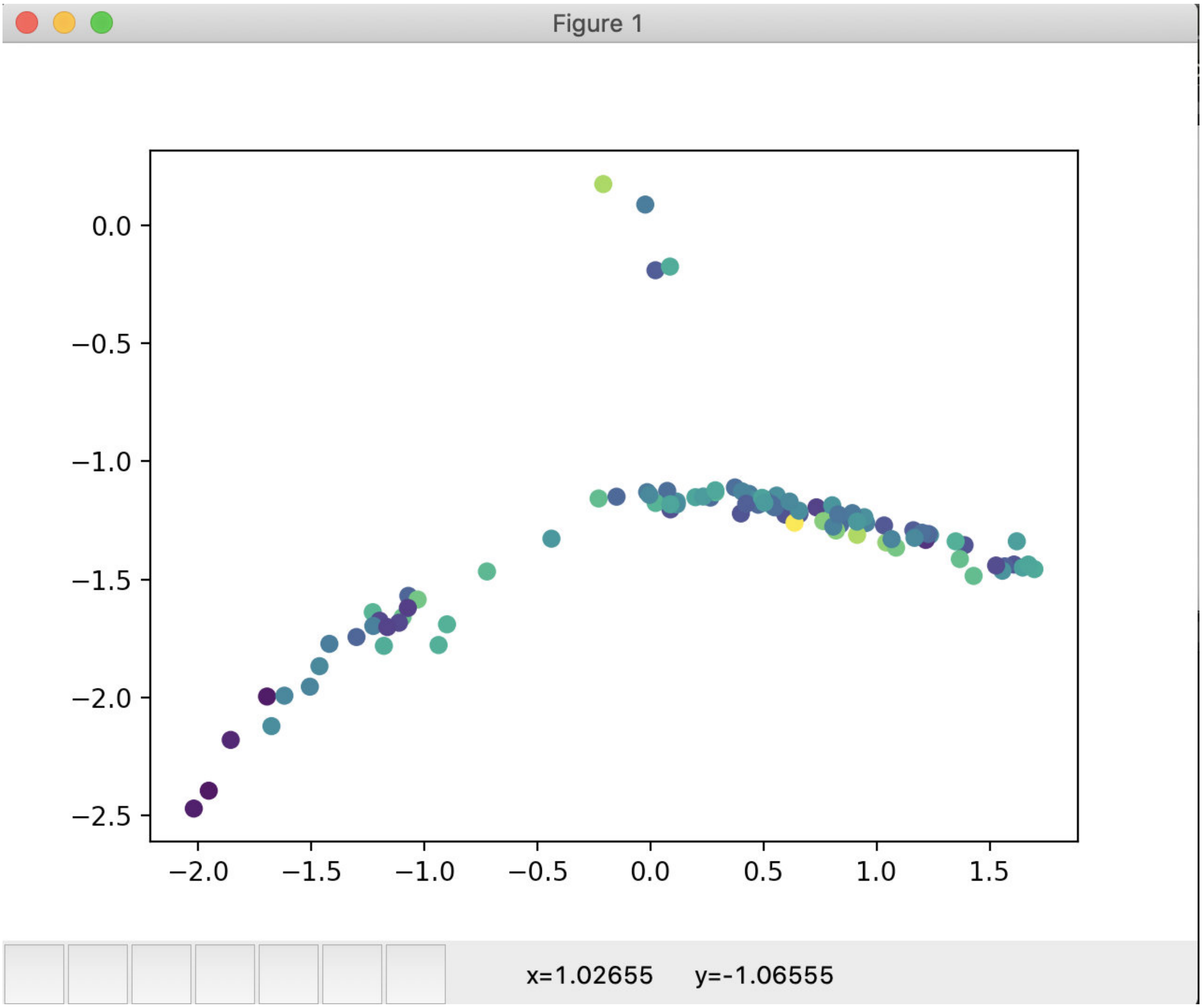
- 输出顶点的二维向量表示

实验结果

输入图：



输出的向量：



word2vec中skip-gram伪代码实现

由于skip-gram中部分细节难以用python代码实现，以上代码中调用了gensim中的Word2Vec包来实现。下面是是关于skip-gram的伪代码实现：

输入：语句集合sentences，表示一个节点的特征数量（维度）d，窗口大小w*size，选词数量num*skip
输出：词向量矩阵

- 建立词汇表。

```
function create_vocabulary(sentences):
    word_set = {}
    for word in sentences:#得到一个没有重复单词的集合
        word_set.add(word)

    dic = {}
    for i from 0 to word_set.size:#得到每个单词在单词表中的位置
        dic[word_set[i]] = i

    O = eye(word_set.size)#得到每个单词的one-hot向量表示，O[i]是第i个单词的one-hot向量
    return dic, O
```

- 得到训练数据。

```
function get_train_data(input_word, sentences, w_size, num_skip):
    train_words = []
    for sentence in sentences:#得到训练集，每一个训练实例是一个以input_word为中心词的单词对
        if input_word in sentence:
            double_word = find_double_word(input_word, sentence, w_size, num_skip)
            for data in double_word:
                train_words.add(data)
    return train_words
```

- 建立神经网络

```
function create_network(dic, O, input_word, train_words):
    #输入层便是单词的one-hot向量矩阵
    word_input_vec = O[dic[input_word]]

    #隐藏层是是一个权重矩阵W（单词的向量表示），大小是|V| * d，建立神经网络时将这个权重矩阵初始化
    W = random_matrix((word_set.size, dimension))
    #此外便于运算，有一个权重矩阵的转置矩阵W_（其他单词的向量表示），大小是d * |V|
    W_ = transpose(W)

    #输出层是一个概率分布，用矩阵P表示，P[dic[word]]表示在窗口中出现word的概率
    v_c = matrix.dot(word_input_vec, W)#得到中心词的向量表示
    P = matrix.dot(v_c, W_)#得到其余单词跟中心词的相似度
    softmax(P)#用softmax函数对P进行归一化，得到概率分布，相似度越大的单词概率越大

    train(train_words)#对神经网络进行训练（细节尚未清楚）
    return W#返回单词的向量表示
```

LINE

LINE with First-order Proximity

定义两个节点之间的联合概率为：

$$p_1(v_i, v_j) = \frac{1}{1+exp(-u_i^T u_j)}$$

u_i, u_j分别是节点i和j的低维嵌入表示，对应需要拟合的经验概率为$\hat{p}_1(i,j) = \frac{w_{ij}}{W}$，即全部权重的归一化的占比（$W = \sum_{(i,j)\in E} w_{i,j}$），则对应的优化目标是：

$$O_1 = d(\hat{p}_1(\cdot,\cdot), p_1(\cdot,\cdot))$$

目的是使预定义的两个点之间的联合概率尽量靠近经验概率，这里作者用一个KL散度来度量这个距离，则要优化的目标函数为：

$$O_1 = max - \sum_{(i,j)\in E} w_{ij} log p_1(v_i, v_j)$$

LINE with Second-order Proximity

对于有向边(i, j)，在给定顶点v_i的条件下，产生上下文（邻居）节点为v_j的概率是：

$$p_2(v_j|v_i) = \frac{exp(u_j^T u_i)}{\sum_{k=1}^{|V|} exp(u_k^T u_i)}$$

其中$|V|$是上下文节点的个数。则优化目标是：

$$O_2 = \sum_{i\in V} \lambda_i d(\hat{p}_2(\cdot|v_i), p_2(\cdot|v_i))$$

其中λ_i是控制节点重要性的因子，可以通过节点的度数得到。这里的经验分布则定义为$\hat{p}_2(v_j|v_i) = \frac{w_{ij}}{d_i}$，其中$w_{ij}$是边(i, j)的权重，$d_i$是顶点$v_i$的出度，即$d_i = \sum_{k\in N(I)} W_{ik}$

同样使用KL散度可以将目标函数转化为：

$$O_2 = -\sum_{(i,j)\in E} w_{ij} log p_2(v_j|v_i)$$

Graph kernel

一种有效的图结构相似度的近似度量方式，具体方法是：

给定两个图$G_1(V_1, E_1), G_2(V_2, E_2)$，一种图的分解方式F，分解后的子图结构为：

$$F(G_1) = \{S_{1,1}, S_{1,2}, \ldots, S_{1,N_1}\} F(G_2) = \{S_{2,1}, S_{2,2}, \ldots, S_{2,N_2}\}$$

则G_1和G_2的kernel value可以表示为：

$$k_R(G_1, G_2) = \sum_{n_1=1}^{N_1} \sum_{n_2=1}^{N_2} \delta(S_{1,n_1}, S_{2,n_2})$$

其中$\delta(S_{1,n_1}, S_{2,n_2})$在$S_{1,n_1}$和$S_{2,n_2}$同构时为1，不同时为0.

这里可以用Weisfeiler-Lehman算法来判断两个子图是否同构：Weisfeiler-Lehman算法