How to do Deep Learning on Graphs with Graph Convolutional Networks

翻译: How to do Deep Learning on Graphs with Graph Convolutional Networks

什么是图卷积网络

What is a Graph Convolutional Network?

More formally, a *graph convolutional network (GCN)* is a neural network that operates on graphs. Given a graph $G = (E, V)$, a GCN takes as input

- an input feature matrix $N \times F^0$ feature matrix, $\boldsymbol{X}$, where N is the number of nodes and F^0 is the number of input features for each node, and
- an $N \times N$ matrix representation of the graph structure such as the adjacency matrix $\boldsymbol{A}$ of G.[1]

A hidden layer in the GCN can thus be written as $\boldsymbol{H^i} = f(\boldsymbol{H^{i-1}}, \boldsymbol{A}))$ where $\boldsymbol{H^0} = \boldsymbol{X}$ and f is a propagation [1]. Each layer $\boldsymbol{H^i}$ corresponds to an $N \times F^i$ feature matrix where each row is a feature representation of a node. At each layer, these features are aggregated to form the next layer's features using the propagation rule f. In this way, features become increasingly more abstract at each consecutive layer. In this framework, variants of GCN differ only in the choice of propagation rule f [1].

图卷积网络是一个在图上进行操作的神经网络。给定一个图 $G=(E,V)$,一个 GCN 的输入包括:

- 一个输入特征矩阵 X，其维度是 $N \times Fo$,其中 N 是节点的数目，Fo 是每个节点输入特征的数目
- 一个 $N \times N$ 的对于图结构的表示的矩阵，例如 G 的邻接矩阵 A

GCN 的一个隐藏层可以写成 $H^i=f(H^{i-1},A)$,其中 $H^0=X$ 并且 f 是一个 propagation。每层 H^i 对应一个 $N\times F^i$ 的特征矩阵，矩阵的每行是一个节点的特征表示。在每层，这些特征通过 propagation rule f 聚合起来形成下一层的特征。通过这种方式，特征变得越来越抽象在每一层。在这个框架中，GCN 的各种变体仅仅是在 propagation rule f 的选择上有不同。

一个简单的 Propagation Rule

一个可能最简单的传播规则是：

$$f(H^i,A)=\sigma(AH^iW^i)$$

其中 W^i 是第 i 层的权重并且σ是一个非线性激活函数例如 ReLU 函数。权重矩阵的维度是 $F^i\times F^{i+1}$；也就是说权重矩阵的第二个维度决定了在下一层的特征的数目。如果你对卷积神经网络熟悉，这个操作类似于 filtering operation 因为这些权重被图上节点共享。

可以简单表示为

$$H^{i+1}=f(H^i,A)=\sigma(AH^iW^i)$$

A：(N,N)邻接矩阵

F^i：一维数值，第i层的特征树目

H^i：(N,F^i)，每一行都是一个结点的特征表示

$X=H^0$：(N,F^0)，输入向量

W^i：（F^i,F^{i+1}）第i层的权值矩阵

来自 <https://aistudio.baidu.com/aistudio/projectdetail/340225>

简化

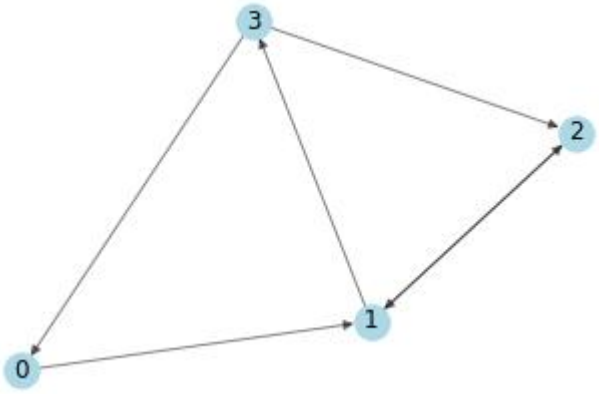
让我们在简单的水平上研究 propagation rule 。令：

- $i=1$, $s.t.$ f 是一个输入特征矩阵的函数，
- Σ 是恒等函数（Identity function），并且
- 选择权重矩阵满足 $AH^0W^0=AXW^0=AX$

也就是说，$f(X,A)=AX$。这个传播规则或许太过简单，但是后面我们将会对缺失的部分进行补充。做个旁注，AX 现在等于一个多层感知机的输入层。

一个简单的例子

我们将使用下面的图作为一个简单的例子：



这个图的邻接矩阵表示是

```
A = np.matrix([
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype=float
)
```

A 为该图的邻接矩阵

```
import numpy as np
A = np.matrix([  ##邻接矩阵
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype=float
)
```

X 为输入的特征向量，我们直接取值，维度为(N,F^0),其中 N 为结点个数，F^0 为输入向量的特征维数

接下来，我们需要特征！我们根据每个节点的索引来生成两个整数特征。这会使得后面手动验证矩阵的计算过程更容易。

```
X = np.matrix([
    [i, -i]
    for i in range(A.shape[0])
], dtype=float)
```

```
X = np.matrix([
            [i, -i]
            for i in range(A.shape[0])
        ], dtype=float)
X
```

```
matrix([[ 0.,  0.],
        [ 1., -1.],
        [ 2., -2.],
        [ 3., -3.]])
```

```
A*X
```

```
matrix([[ 1., -1.],
        [ 5., -5.],
        [ 1., -1.],
        [ 2., -2.]])
```

好了，现在我们有了一个图，以及它的邻接矩阵 A 和一个输入特征的集合 X。让我们看看当使用传播规则时会发生什么：

```
In [6]: A * X
Out[6]: matrix([
                [ 1., -1.],
                [ 5., -5.],
                [ 1., -1.],
                [ 2., -2.]]
```

发生了什么，每个节点的表示（每一行）现在等于他的邻居的特征的和（对于结点来说是出度的邻居的特征和-方观 2020-4-30 17:08:50）！换句话说，图卷积层将每个节点表示成它的邻居的总数。在手动验证的时候注意，一个节点 n 是一个节点 v 的邻居只有在存在一个从 v 到 n 的边时。

使用 Propagation Rule

即将出现的问题

你可能已经注意到了问题：

- A*X 的结点表示中，并没有加自己的特征值。一个节点的聚集表示不包括它自己的特征！这个表示只是它的邻居节点特征的聚集，所以只有有一个自循环的节点才会包括它自己的特征在聚集中。
- 邻接结点多的结点的特征值会大，少的特征值就小。具有很大度数的节点将会有很大的值在它们的特征表示中，而具有很小度数的节点将会有很小的值。这些可能造

成梯度消失或者梯度爆炸。这对于随机梯度下降也可能是有问题的，随机梯度下降通常被用来训练这样的网络，而且对于每个输入特征的值的范围是敏感的。

下面，我将分别讨论这些问题

加入 Self-Loops

为了处理第一个问题，我们可以通过简单地给每个节点加入一个自循环（self-loops）。在实践中，这可以通过把单位矩阵 I 加到邻接矩阵 A （在应用传播规则之前）上来实现。

```
I = np.matrix(np.eye(A.shape[0]))
A_hat = A + I
In [8]: A_hat = A + I
       A_hat * X
Out[8]: matrix([
          [ 1., -1.],
          [ 6., -6.],
          [ 3., -3.],
          [ 5., -5.]])
```

因为现在每个节点是自己的邻居，所以节点在计算邻居节点的特征之和的过程中也把自己的特征加进去了。

```
I = np.matrix(np.eye(A.shape[0]))
I
```

```
matrix([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

```
A_hat = A + I
A_hat * X
```

```
matrix([[ 1., -1.],
        [ 6., -6.],
        [ 3., -3.],
        [ 5., -5.]])
```

Normalizing the Feature Representations

The feature representations can be normalized by node degree by transforming the adjacency matrix `A` by multiplying it with the inverse degree matrix `D` [1]. Thus our simplified propagation rule looks like this [1]:

特征表示可以通过节点的度来进行归一化，方法是将邻接矩阵 A 转换为 A 和度矩阵 D 的逆的乘积。因此我们简化的传播规则看起来向这样：

$$f(X,A)=D^{-1}AX$$

【原因见 https://tkipf.github.io/graph-convolutional-networks/
GRAPH CONVOLUTIONAL NETWORKS
THOMAS KIPF, 30 SEPTEMBER 2016】

让我们看看发生了什么。首先我们计算度矩阵

```
D = np.array(np.sum(A,axis=0))[0]
D = np.matrix(np.diag(D))
In [9]: D
Out[9]: matrix([
           [1., 0., 0., 0.],
           [0., 2., 0., 0.],
           [0., 0., 2., 0.],
           [0., 0., 0., 1.]
       ])
```

```
D = np.array(np.sum(A, axis=0))[0]
D = np.matrix(np.diag(D))
D
```

```
matrix([[1., 0., 0., 0.],
        [0., 2., 0., 0.],
        [0., 0., 2., 0.],
        [0., 0., 0., 1.]])
```

在我们应用规则之前，让我们看看在转换了邻接矩阵后，其上发生了什么。

转换前

```
A = np.matrix([
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype=float
)
```

```
A
```

```
matrix([[0., 1., 0., 0.],
        [0., 0., 1., 1.],
        [0., 1., 0., 0.],
        [1., 0., 1., 0.]])
```

转换后

```
In [10]: D**-1 * A
Out[10]: matrix([
           [0. , 1. , 0. , 0. ],
           [0. , 0. , 0.5, 0.5],
           [0. , 0.5, 0. , 0. ],
           [1. , 0. , 1. , 0. ]
])
```

```
D**-1 * A
```

```
matrix([[0. , 1. , 0. , 0. ],
        [0. , 0. , 0.5, 0.5],
        [0. , 0.5, 0. , 0. ],
        [1. , 0. , 1. , 0. ]])
```

```
D**-1 * A * X
```

```
matrix([[ 1. , -1. ],
        [ 2.5, -2.5],
        [ 0.5, -0.5],
        [ 2. , -2. ]])
```

注意到邻接矩阵每行的权重在原来权重的基础上除以对应行的节点的度数。我们在变化后的

邻接矩阵上应用传播规则：

```
In [11]: D**-1 * A * X
Out[11]: matrix([
            [ 1. , -1. ],
            [ 2.5, -2.5],
            [ 0.5, -0.5],
            [ 2. , -2. ]
        ])
```

我们将邻居节点特征均值作为节点的表示。

总结

我们现在把自循环和归一化技巧结合起来。另外，我们将再次引入权重和激活函数（这在前

面为了简化讨论而省略了）。

加入权重

首先第一件事是应用权重。注意到这里 D_hat 是矩阵 A_hat = A + I 的度矩阵。

```
In [45]: W = np.matrix([
            [1, -1],
            [-1, 1]
        ])
        D_hat**-1 * A_hat * X * W
Out[45]: matrix([
           [ 1., -1.],
           [ 4., -4.],
           [ 2., -2.],
```

```
           [ 5., -5.]
       ])
```

```
A_hat  = A + I
D_hat = np.array(np.sum(A_hat, axis=0))[0]
D_hat = np.matrix(np.diag(D_hat))
D_hat
```

```
matrix([[2., 0., 0., 0.],
        [0., 3., 0., 0.],
        [0., 0., 3., 0.],
        [0., 0., 0., 2.]])
```

```
W = np.matrix([
            [1, -1],
            [-1, 1]
        ])
```

```
X
```

```
matrix([[ 0.,  0.],
        [ 1., -1.],
        [ 2., -2.],
        [ 3., -3.]])
```

```
D_hat**-1 * A_hat*X*W
```

```
matrix([[ 1., -1.],
        [ 4., -4.],
        [ 2., -2.],
        [ 5., -5.]])
```

如果我们想要减少输出特征表示的维度，我们可以减少权重矩阵的规模：

```
In [46]: W = np.matrix([
            [1],
            [-1]
        ])
```

```
        D_hat**-1 * A_hat * X * W
Out[46]: matrix([[1.],
       [4.],
       [2.],
       [5.]]
)
```

加入激活函数

我们选择保持特征表示的维度并且应用 ReLU 激活函数。

```
In [51]: W = np.matrix([
            [1, -1],
            [-1, 1]
        ])
        relu(D_hat**-1 * A_hat * X * W)
Out[51]: matrix([[1., 0.],
       [4., 0.],
       [2., 0.],
       [5., 0.]])
```

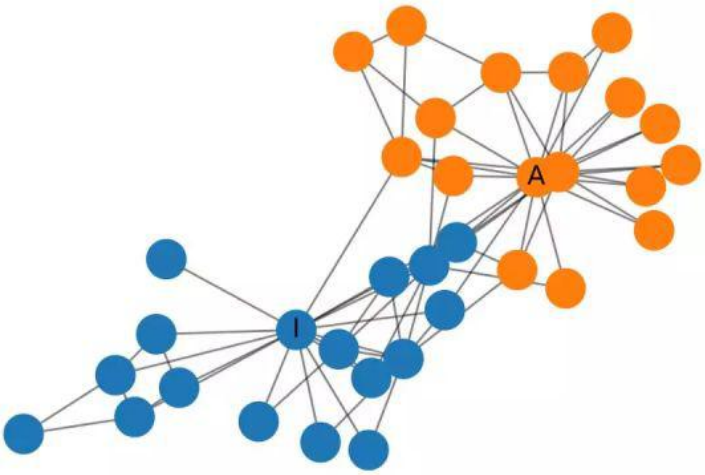
这就是一个带有邻接矩阵，输入特征，权重和激活函数的完整的隐藏层。

回到现实

最后，我们在一个真实的图上应用一个图卷积网络。我将会告诉你怎样去产生特征表示就像我们前面讲的。

Zachary's Karate Club

Zachary 空手道俱乐部是一个被广泛使用的社交网络，其中的节点代表空手道俱乐部的成员，边代表成员之间的相互关系。当年，Zachary 在研究空手道俱乐部的时候，管理员和教员发生了冲突，导致俱乐部一分为二。下图显示了该网络的图表征，其中的节点标注是根据节点属于俱乐部的哪个部分而得到的，「A」和「I」分别表示属于管理员和教员阵营的节点。



Zachary 空手道俱乐部图网络

构建 GCN

现在我们用一个实际的网络来进行实验，将图数据嵌入到二位空间中，看看嵌入的效果如何。

- 选取的网络为空手道俱乐部数据集(karate_club_graph)
- 嵌入分为两层，初始层的X设置为单位阵,两层W值为随机数
- 可以形式化表述为 $D^{-1}A\sigma(D^{-1}AXW^{1})W^{2}$ 其中，激活函数为RELU

构建 GCN 之 2 解释

接下来，我们将构建一个图卷积网络。我们并不会真正训练该网络，但是会对其进行简单的随机初始化，从而生成我们在本文开头看到的特征表征。我们将使用 networkx，它有一个可以很容易实现的 Zachary 空手道俱乐部的图表征。然后，我们将计算 A_hat 和 D_hat 矩阵。

```
from networkx import to_numpy_matrix
zkc = karate_club_graph()
order = sorted(list(zkc.nodes()))
A = to_numpy_matrix(zkc, nodelist=order)
I = np.eye(zkc.number_of_nodes())
A_hat = A + I
D_hat = np.array(np.sum(A_hat, axis=0))[0]
D_hat = np.matrix(np.diag(D_hat))
```

接下来，我们将随机初始化权重。

```
W_1 = np.random.normal(
    loc=0, scale=1, size=(zkc.number_of_nodes(), 4))
W_2 = np.random.normal(
    loc=0, size=(W_1.shape[1], 2))
```

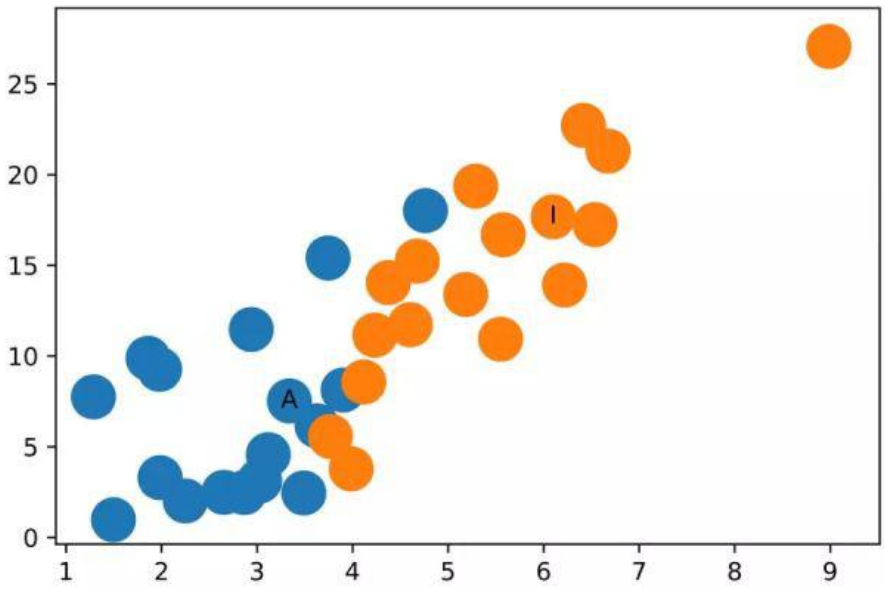
接着，我们会堆叠 *GCN* 层。这里，我们只使用单位矩阵作为特征表征，即每个节点被表示为一个 *one-hot* 编码的类别变量。

```
def gcn_layer(A_hat, D_hat, X, W):
    return relu(D_hat**-1 * A_hat * X * W)
H_1 = gcn_layer(A_hat, D_hat, I, W_1)
H_2 = gcn_layer(A_hat, D_hat, H_1, W_2)
output = H_2
```

我们进一步抽取出特征表征。

```
feature_representations = {
    node: np.array(output)[node]
    for node in zkc.nodes()}
```

你看，这样的特征表征可以很好地将 *Zachary* 空手道俱乐部的两个社区划分开来。至此，我们甚至都没有开始训练模型！



Zachary 空手道俱乐部图网络中节点的特征表征

我们应该注意到，在该示例中由于 ReLU 函数的作用，在 x 轴或 y 轴上随机初始化的权重很可能为 0，因此需要反复进行几次随机初始化才能生成上面的图。

结语

本文中对图卷积网络进行了高屋建瓴的介绍，并说明了 GCN 中每一层节点的特征表征是如何基于其相邻节点的聚合构建的。读者可以从中了解到如何使用 numpy 构建这些网络，以及它们的强大：即使是随机初始化的 GCN 也可以将 Zachary 空手道俱乐部网络中的社区分离开来。

Back to Reality

现在我们在现实中的网络中运用图卷集网络(graph convolutional network 技术。选取的网络为空手道俱乐部数据集(karate_club_graph)。

```
from networkx import to_numpy_matrix
import networkx as nx
zkc = nx.karate_club_graph()
order = sorted(list(zkc.nodes()))
A = to_numpy_matrix(zkc, nodelist=order)
I = np.eye(zkc.number_of_nodes())
A_hat = A + I
D_hat = np.array(np.sum(A_hat, axis=0))[0]
D_hat = np.matrix(np.diag(D_hat))
```

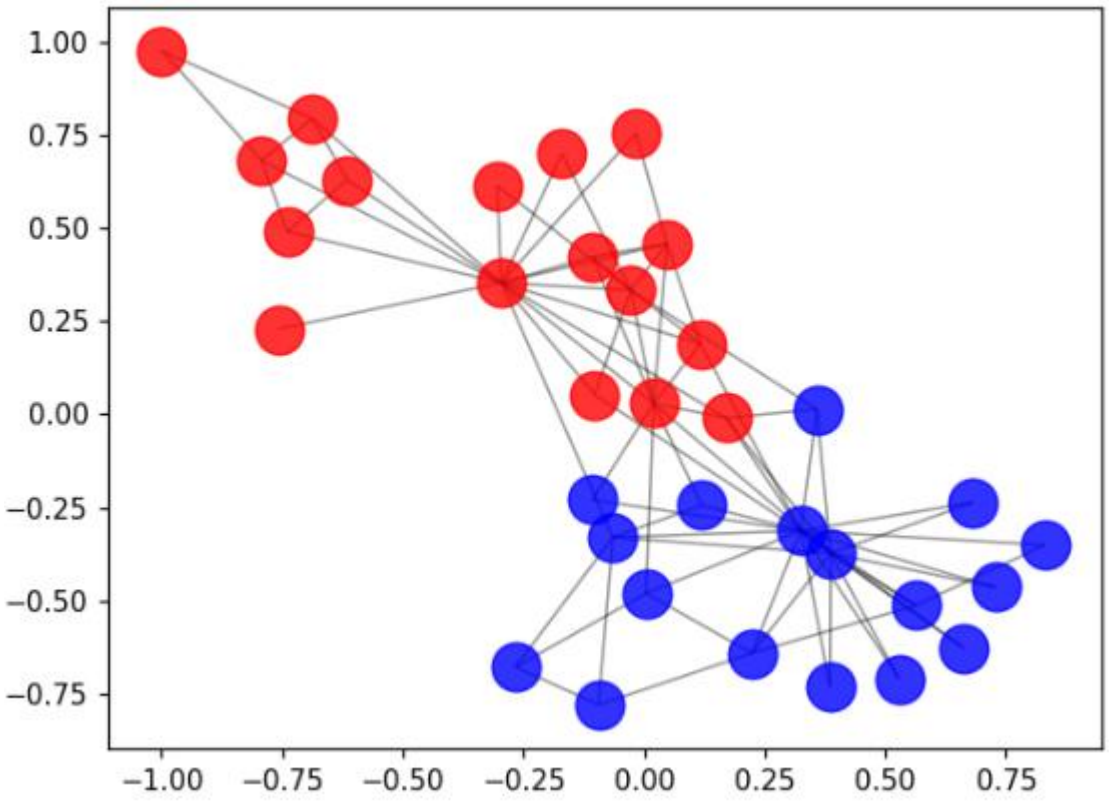
```
def plot_graph(G, weight_name=None):
    '''
    G: a networkx G
    weight_name: name of the attribute for plotting edge weights (if G is weighted)
    '''
    %matplotlib notebook
    import matplotlib.pyplot as plt

    plt.figure()
    pos = nx.spring_layout(G)
    edges = G.edges()
    weights = None

    if weight_name:
        weights = [int(G[u][v][weight_name]) for u,v in edges]
        labels = nx.get_edge_attributes(G,weight_name)
        nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)
        nx.draw_networkx(G, pos, edges=edges, width=weights);
    else:
        nodelist1 = []
        nodelist2 = []
        for i in range (34):
            if zkc.nodes[i]['club'] == 'Mr. Hi':
                nodelist1.append(i)
            else:
                nodelist2.append(i)
        #nx.draw_networkx(G, pos, edges=edges);
        nx.draw_networkx_nodes(G, pos, nodelist=nodelist1, node_size=300, node_color='r',alpha = 0.8)
        nx.draw_networkx_nodes(G, pos, nodelist=nodelist2, node_size=300, node_color='b',alpha = 0.8)
        nx.draw_networkx_edges(G, pos, edgelist=edges,alpha =0.4)
```

```
plot_graph(zkc)
```

```
<IPython.core.display.Javascript object>
```




```
W_1 = np.random.normal(
    loc=0, scale=1, size=(zkc.number_of_nodes(), 4))
W_2 = np.random.normal(
    loc=0, size=(W_1.shape[1], 2))
```

```
def relu(x):
    return (abs(x) + x) / 2

```

```
def gcn_layer(A_hat, D_hat, X, W):
    return relu(D_hat**-1 * A_hat * X * W)
H_1 = gcn_layer(A_hat, D_hat, I, W_1)
H_2 = gcn_layer(A_hat, D_hat, H_1, W_2)
output = H_2
```

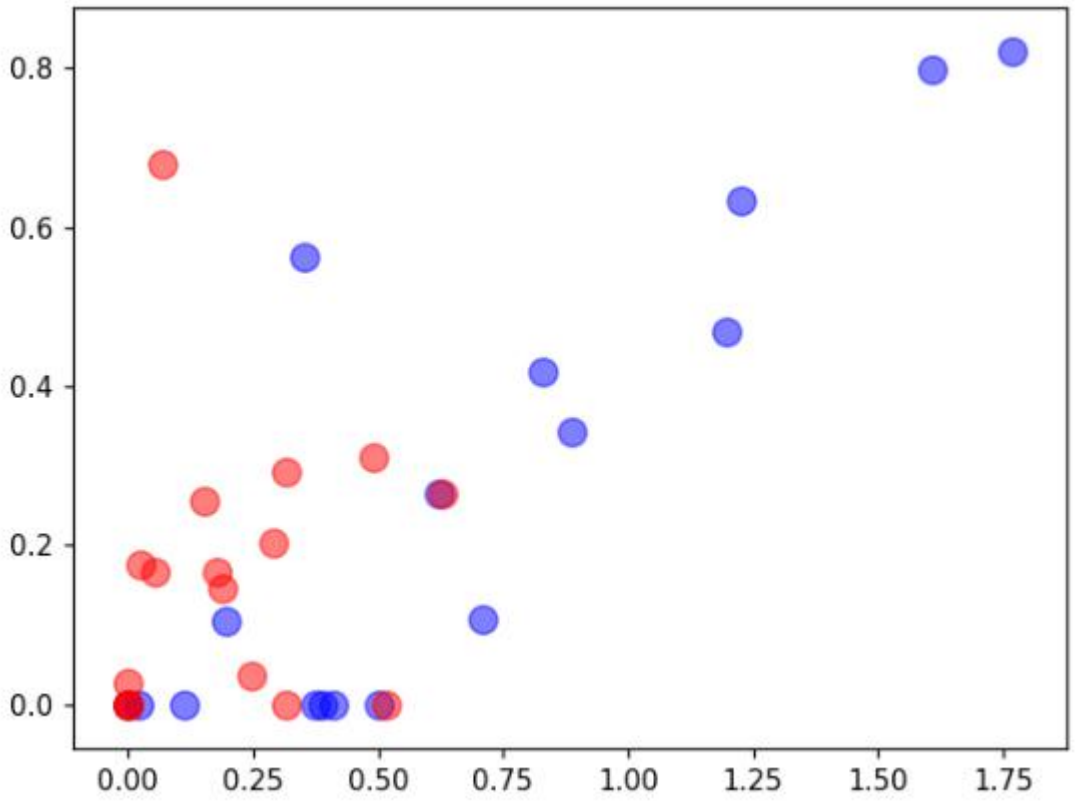
```
feature_representations = {
    node: np.array(output)[node]
    for node in zkc.nodes()}
```

```
feature_representations
```

```
{0: array([0.88602091, 0.34237275]),
 1: array([0.40862582, 0.        ]),
 2: array([0.38693926, 0.        ]),
 3: array([0.19478099, 0.10516756]),
 4: array([0.82815959, 0.41738152]),
 5: array([1.1971192 , 0.46978126]),
 6: array([1.2271154 , 0.63378424]),
 7: array([0., 0.]),
 8: array([0.11110005, 0.        ]),
 9: array([0., 0.]),
 10: array([0.6209274 , 0.26495055]),
 11: array([1.60869786, 0.79829349]),
 12: array([0.35029305, 0.56226336]),
 13: array([0.02171053, 0.        ]),
 14: array([0.        , 0.02638456]),
 15: array([0.06979159, 0.68002892]),
 16: array([1.7675629 , 0.82039984]),
 17: array([0.50286326, 0.        ]),
 18: array([0.31509428, 0.29327311]),
 19: array([0.37260057, 0.        ]),
 20: array([0., 0.]),
 21: array([0.70826438, 0.10767323]),
 22: array([0.15022781, 0.25590783]),
 23: array([0.17645064, 0.16650816]),
 24: array([0.29110197, 0.20382017]),
 25: array([0.18688296, 0.14564473]),
 26: array([0.02367803, 0.17550985]),
 27: array([0., 0.]),
 28: array([0.51547931, 0.        ]),
 29: array([0.05318727, 0.16647217]),
 30: array([0.31639705, 0.        ]),
 31: array([0.24761528, 0.03619812]),
 32: array([0.48872535, 0.31039692]),
 33: array([0.62804696, 0.26496685])}
```

```
import matplotlib.pyplot as plt
%matplotlib notebook
for i in range (34):
    if zkc.nodes[i]['club'] == 'Mr. Hi':
        plt.scatter(np.array(output)[i,0],np.array(output)[i,1] ,color = 'b',alpha=0
    else:
        plt.scatter(np.array(output)[i,0],np.array(output)[i,1] ,color = 'r',alpha=0
#plt.scatter(np.array(output)[:,0],np.array(output)[:,1])
```

```
<IPython.core.display.Javascript object>
```



目前来看，这个映射分类效果并不好，待我后续分析补充吧。

来自 <https://blog.csdn.net/qq_36793545/article/details/84844867>