机器学习与数据挖掘 ~ Homework3

19335084 黄梓浩

1 实验目的和实验要求

1.1 实验目的

实现 Modularity 算法，采用 **Fast unfolding of communities in large networks** 实现优化。

1.2 实验要求

- 用自己实现的社区发现算法得到预测结果，用NMI度量效果。
- 使用 GenLouvain算法 得到预测结果，用NMI度量效果。
- 分析两者代码的差异，并发现自己代码的问题。

2 算法原理

2.1 社区发现问题

在复杂网络中，如果节点可以很容易地分组成多个节点集，使得节点集的内部是紧密连接的，就称网络具有社区结构。这意味着网络可以自然地划分为具有 内部中节点紧密连接 和 外部其他节点连接稀疏的节点集。这能反映的含义是： 如果节点都是同一个社区的，则它们更有可能连接，如果它们不在同一社区，则不太可能连接。

这个结构在 一些共同位置、兴趣、职业等的社区团体发现的方面上有很大作用，有几种算法用于处理计算这种结构。

2.2 最大化模块度

模块度最大化 是 社区发现的一种最常用的方法。

模块度是一个衡量网络的社区划分质量的收益函数，越大表示划分的效果越好。其计算方式如下：

$$Q = \frac{1}{2m}\sum_{vw}[A_{vw} - \frac{k_v k_w}{2m}]\delta(c_v, c_w)$$

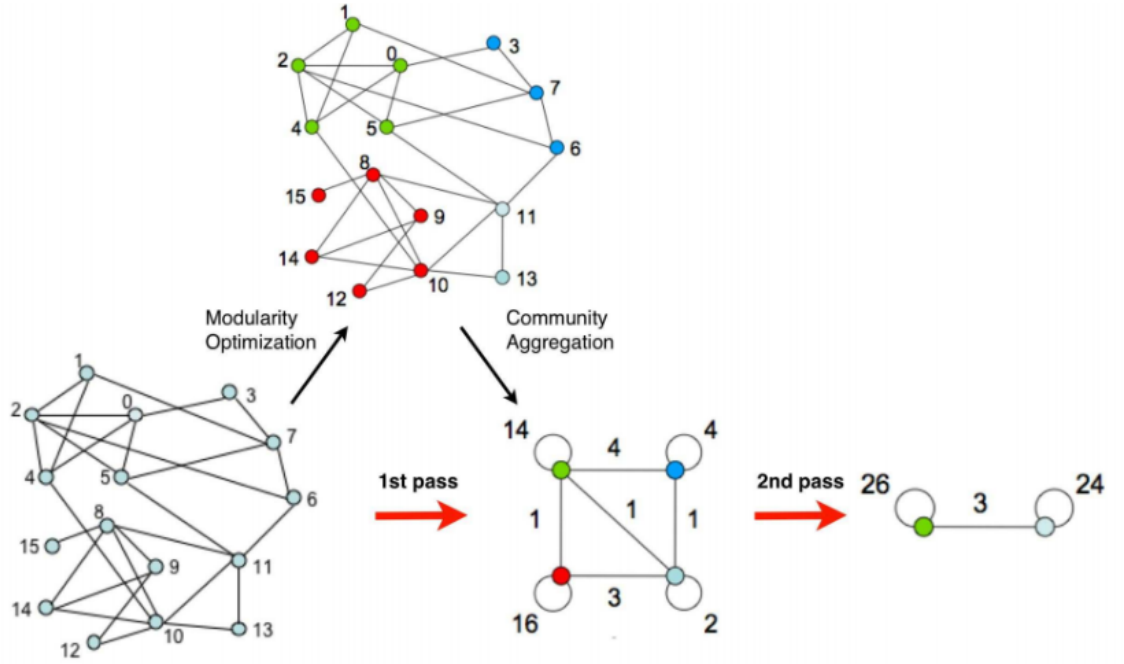
其中 A_{vw} 表示 节点v 和 节点w 间的边权值， k_v表示和 节点v 相连的权值之和，c_v表示 节点v 所属的社区，当 节点v 和 节点w 属于同一社区时，$\delta(c_v, c_w)$取为1，否则为0，m 表示边权重的和。

该方法便是基于最大化模块度的思想进行的。

2.3 Fast-unfolding (louvain) 算法

这是一个最大化模块度的社区发现算法，其基本思想是：

- 第一步： 遍历网络中的每个节点，根据模块度增益 来决定是否 将其加入到 邻居节点的社区，以及加入到哪一个邻居的社区。
- 第二步： 根据第一步的社区划分结果，把同一个社区的节点都合并为一个节点，并更新节点间的边。
- 重复迭代执行 第一步、第二步，直到 模块度增益 收敛 为止。



对于模块度增益的计算：

如果记 Σin 为 社区c 内所有边权重和的2倍， Σtot 为连到社区c中的节点 的所有连边的权重和，

则对于单个社区c，其模块度可表示为： $Q_c = \frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2$

在此基础上，若节点 i **加入**该社区，模块度为： $\frac{\sum_{in}+2k_{i,in}}{2m} - (\frac{\sum_{tot+k_i}}{2m})^2$

$k_{i,in}$表示的是 节点i 连到社区c中的边的权重和（即社区c中有连到节点i 的所有边，权重和）

在此基础上，若节点 i **离开**该社区，模块度为： $\frac{\sum_{in}-2k_{i,in}}{2m} - \left(\frac{\sum_{tot}-k_i}{2m}\right)^2$

得到以上公式后，回到该算法的第一步，我们需要根据模块度的增益决定一个节点要到哪一个社区去，

那对于节点 i，计算其 **离开原来社区的模块度增益 + 加入新的社区的模块度增益**，即可得到**总的模块度增益**：

实际计算时可以分两步进行，

离开原社区的模块度增益： $\left(\frac{\sum_{in}-2k_{i,in}}{2m} - (\frac{\sum_{tot+k_i}}{2m})^2\right) - \left(\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2\right)$

加入新社区的模块度增益：$\left(\frac{\sum_{in}+2k_{i,in}}{2m}-(\frac{\sum_{tot+k_i}}{2m})^2\right)-\left(\frac{\sum_{in}}{2m}-(\frac{\sum_{tot}}{2m})^2\right)$

化简该式子分别可得：

$$\frac{(-2k_{i,in}+\frac{\sum_{tot}\times k_i}{m}-\frac{k_i^2}{2m})}{2m} \quad 和 \quad \frac{(2k_{i,in}+\frac{\sum_{tot}\times k_i}{m}-\frac{k_i^2}{2m})}{2m}$$

在实际计算时可以忽略分母 2m，把这两个式子相加就是这一步下来 ΔQ 的值。

然后在代码实现的时候，我们只要按边权和从大到小按顺序枚举节点进入其他的社区，用这个式子来计算 ΔQ，找出最大的 ΔQ的情况，把节点加入新社区就行了。

3 代码 （使用 Python实现）

3.1 导入数据类型

我们需要把数据处理成只有 从0开始的有序的节点 、边的格式。

```
nodes = [0, 1, 2]
edges = [ [0 , 2 , 1], [0, 1, 1], [1, 2, 1] ]
# 表示有 3个节点，3条边，0和2有权值为1的边，0和1有权值为1的边……
```

3.2 Fast - unfolding 类

我们希望实现一个这样的类，导入节点和边的数据后，执行类方法可以完成社区发现任务，并输出 模块度Q，以及输出 原本节点所划分的社区序号（方便用于NMI比较）。

按照算模块度增益的式子来定义变量和方法：

$$\frac{(-2k_{i,in}+\frac{\sum_{tot}\times k_i}{m}-\frac{k_i^2}{2m})}{2m} \quad 和 \quad \frac{(2k_{i,in}+\frac{\sum_{tot}\times k_i}{m}-\frac{k_i^2}{2m})}{2m}$$

类变量如下，其中 node_in 和 node_tot 是用于计算 Σin 和 Σtot的。

```
[self.nodes, self.edges, self.node_num] # 节点，边，节点的初始数量
self.node_com    # 节点所在的社区,初始节点的社区是自己编号
self.node_in     # 社区节点的内部边的权重和,初始社区内没有边，为0
self.node_tot    # 节点的连边的权重和
self.com_dete    # 初始节点的社区划分
....
self.m             # m（网络的边权值和）
self.node_edges    # 节点所相连的边
```

类方法如下：

```
def __init__(self, nodes, edges, m):

# 第一步 根据模块度增益决定节点要加入到哪个社区中
def step1_fun(self):

# 第二步 合并同一个社区中的节点变为一个节点
def step2_fun(self, coms):

# 迭代进行第一步和第二步，进行社区发现,输出
def run(self):
```

3.3 初始化：

开始时初始化类变量，导入节点和边。

一开始每个节点各自是一个社区，而这时社区内没有边，所以 node_in 都为0.

通过导入的 边来更新 node_tot 和 node_edges（节点i 相连的边，用来找邻居）变量。

```
def __init__(self, nodes, edges):
    # 节点，边，节点的初始数量
    [self.nodes, self.edges, self.node_num] = [nodes, edges, len(nodes)]
    self.node_com = [n for n in nodes]    # 节点所在的社区,初始节点的社区是自己编号
    self.node_in = [0 for n in nodes]     # 社区节点的内部边的权重和,初始社区内没有边,
为0
    self.node_tot = [0 for n in nodes]    # 节点的连边的权重和
    self.com_dete = []  # 初始节点的社区划分
    self.m = 0  # m（网络的边权值和）
    n_edges = [[] for n in nodes]
    for e in edges:
        self.m += e[2]
        self.node_tot[e[0]] += e[2]
        self.node_tot[e[1]] += e[2]
        n_edges[e[0]].append((e[1], e[2]))
        n_edges[e[1]].append((e[0], e[2]))
    self.node_edges = n_edges    # 节点对应的边
```

3.4 算法第一步：

实现算法第一步，根据模块度增益决定节点要加入到哪个社区中：

按该式子进行：$\frac{(-2k_{i,in}+\frac{\Sigma_{tot}\times k_i}{m}-\frac{k_i^2}{2m})}{2m}$ 和 $\frac{(2k_{i,in}+\frac{\Sigma_{tot}\times k_i}{m}-\frac{k_i^2}{2m})}{2m}$

第一步分为几个步骤：

- 按边的权重和大小来遍历节点：
  - 计算该节点 **移出当前社区** 的模块度增益：
    - 因为节点可能是合并过的，所以 $k_{i,in}$ 它包括该节点的 node_in 和 该节点和要离开的社区的连边的权值和 remove_in，所以 $k_{i,in}$ = (remove_in + self.node_in[node])。

        - 我们要计算 remove_in ， 就需要通过 node_edges变量来找节点的邻居
        - 最后的模块度增益式子就是：

          ΔQ1 = `-2 * (remove_in + node_in[node])`

          `+ sig_tot[node_c] * node_tot[node] / m`

          `node_tot[node]**2 / (2*m)`
    - 遍历还没有访问过的社区
        - 计算该节点 **加入新的社区** 的模块度增益，计算方法和上面是一样的。
        - ΔQ2 = `+2 * (enter_in + node_in[node])`

          `+ sig_tot[node_c] * node_tot[node] / m`

          `node_tot[node]**2 / (2*m)`
        - 计算 ΔQ = ΔQ1 +ΔQ2，更新最大的ΔQ情况，记录将要加入的新社区。
    - 找出最大的ΔQ后，如果值没有收敛，就更新社区网络，把节点离开原社区，加入到新社区，并更新 sigma_in 和 sigma_tot 的值，然后继续遍历其他节点，重复过程；

      如果收敛的话就退出整个社区发现算法的运行过程。

```
# 第一步 根据模块度增益决定节点要加入到哪个社区中
def step1_fun(self):
    # 算出每个社区节点的 Σin和 Σtot
    self.sig_in = [self.node_in[node]*2 for node in self.nodes]
    self.sig_tot = [self.node_tot[node] for node in self.nodes]
    # 现在的社区划分
    now_com = [[node] for node in self.nodes]
    stop_flag = True
    while True:
        sp1_stop_flag = True
        # 按连边的权重和大小来遍历节点
        temp = [[node, self.node_tot[node]] for node in self.nodes]
        temp = sorted(temp, key=itemgetter(1), reverse=False)
        seq = [i[0] for i in temp]

        for node in seq:
            node_c = self.node_com[node]     # 节点所在的社区
            max_dq = 0
            remove_in = 0
            join_c = node_c      # 节点将要加入的社区
            fin_enter_in = 0    # 最后加入的时候的度
            # 找和这个节点相邻的节点
            neigh_node_set = set()  # 相邻节点的集合
            try:
                nei_edge = self.node_edges[node]
            except KeyError:
                continue
            for e in nei_edge:
                neigh_node_set.add(e[0])
                if self.node_com[e[0]] == node_c:    # 如果在同一社区
                    remove_in += e[1]

            # 把这个节点移出社区的模块度增益
            dq1 = -2 * (remove_in + self.node_in[node]) +self.sig_tot[node_c] *
\
                self.node_tot[node]/self.m - self.node_tot[node]**2/(2*self.m)
```

```
            visited_com = set() # 已经尝试过把节点加入的社区集合
            for neigh in neigh_node_set:
                neigh_c = self.node_com[neigh]
                if (neigh_c in visited_com) or (neigh_c == node_c):
                    continue
                visited_com.add(neigh_c)
                enter_in = 0
                for e in self.node_edges[node]:
                    if self.node_com[e[0]] == neigh_c: # 如果在同一社区
                        enter_in += e[1]
                # 把这个节点 加入 这个社区的模块度增益
                dq2 = 2 * enter_in + 2 * self.node_in[node]-
self.sig_tot[neigh_c] * \
                    self.node_tot[node]/self.m -
self.node_tot[node]**2/(2*self.m)

                dq = dq1 + dq2 # 总的模块度增益
                if dq > max_dq:
                    max_dq = dq
                    fin_enter_in = enter_in
                    join_c = neigh_c
            # ΔQ没有收敛的话
            if max_dq > 1e-8:
                # 把节点加入到新社区，更新相关值
                stop_flag = sp1_stop_flag = False
                now_com[node_c].remove(node)
                now_com[join_c].append(node)
                self.node_com[node] = join_c
                self.sig_in[node_c] -= 2 * (remove_in + self.node_in[node])
                self.sig_tot[node_c] -= self.node_tot[node]
                self.sig_in[join_c] += 2 * (fin_enter_in + self.node_in[node])
                self.sig_tot[join_c] += self.node_tot[node]
        if sp1_stop_flag:
            return stop_flag, now_com
```

3.5 算法第二步：

实现算法第一步，合并同一个社区中的节点变为一个节点：

没有特别的地方，主要就是通过第一步的划分结果，来更新类的变量。

```
# 第二步 合并同一个社区中的节点变为一个节点
def step2_fun(self, coms):
    nodes_n = [i for i in range(len(coms))]          # 社区变为节点的序号
    nodes_c = [0 for i in range(self.node_num)]      # 原始节点对应的社区
    self.node_com = [n for n in nodes_n]
    l = len(coms)
    for i in range(l):
        for item in coms[i]:
            nodes_c[item] = i
    new_node_in = [0 for n in nodes_n]
    for n in self.nodes:
        new_node_in[ nodes_c[n] ] += self.node_in[n]
    # 更新边
```

```
    edge_set = {}
    for e in self.edges:
        c1 = nodes_c[e[0]]
        c2 = nodes_c[e[1]]
        if c1 == c2:   # 这条边的两个点在同一社区
            new_node_in[c1] += e[2]
        else:   # 不在同一社区，添边
            tup = ( (c1, c2) if(c1<c2) else (c2,c1) )
            try:
                edge_set[tup] += e[2]
            except KeyError:
                edge_set[tup] = e[2]
    # 从集合中取边为默认格式
    new_edges = [[x[0], x[1], y] for x, y in edge_set.items()]
    new_node_edges = [ [] for i in range(self.node_num)]
    self.node_tot = [2*new_node_in[node] for node in nodes_n]
    for e in new_edges:
        self.node_tot[e[0]] += e[2]
        self.node_tot[e[1]] += e[2]
        new_node_edges[e[0]].append((e[1], e[2]))
        new_node_edges[e[1]].append((e[0], e[2]))
    # 更新类的各个参数
    self.nodes = nodes_n
    self.edges = new_edges
    self.node_in = new_node_in
    self.node_edges = new_node_edges
```

3.6 算法运行：

算法运行过程就是不停地迭代第一步和第二步：

设置一个停止标记，在第一步的过程中，如果发现求得的 ΔQ收敛就可以停止 while循环了。

最后公式计算输出模块度，输出预测的社区标号

```
# 迭代进行第一步和第二步，进行社区发现，输出
def run(self):
    while True:
        stop_flag, coms = self.step1_fun()
        if stop_flag:
            break
        coms = [c for c in coms if c]
        if self.com_dete:
            com_dete = []
            for c in coms:
                temp = []
                for node in c:
                    temp.extend(self.com_dete[node])
                com_dete.append(temp)
            self.com_dete = com_dete
        else:
            self.com_dete = coms
        self.step2_fun(coms)

    # 计算并输出模块度
    q = 0
```

```
    for i in range(len(self.sig_tot)):
        q += (self.sig_in[i]-self.sig_tot[i]**2/(2*self.m))/(2*self.m)
    print('Q: ', q)
    # 输出预测出的各个节点的社区 labels
    prelabel = np.zeros(self.node_num, dtype=int)
    l = len(self.com_dete)
    for i in range(l):
        for item in self.com_dete[i]:
            prelabel[item] = i+1
    print(prelabel)
    return prelabel
```

4 结果分析

数据集使用（在data文件夹下）：

```
https://snap.stanford.edu/data/email-Eu-core.html
https://snap.stanford.edu/data/com-Amazon.html
https://snap.stanford.edu/data/com-DBLP.html
```

数据集1：**email-Eu-core** 的对比结果（上图为GenLouvain，下图为我的）：

```
节点数 1005
边数  25571
```

```
D:\graphyhao\doc\machine_learning\3>python test.py
standard louvain:
Q: 0.4388842236357771
[1 1 2 ... 1 1 2]
[ 2  2 22 ...  2  7 23]
NMI:  0.5287655167580243
```

```
D:\graphyhao\doc\machine_learning\3>python mycode.py
Q:  0.4364978086114031
[16 16 26 ... 16 16 26]
[ 2  2 22 ...  2  7 23]
NMI:  0.5628642903241156

D:\graphyhao\doc\machine_learning\3>
```

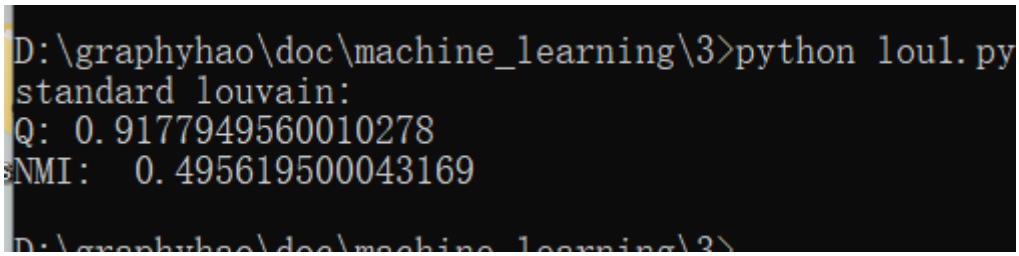
样例使用的是Python上的 Python-Louvain库 代替了Matlab 的GenLouvain。

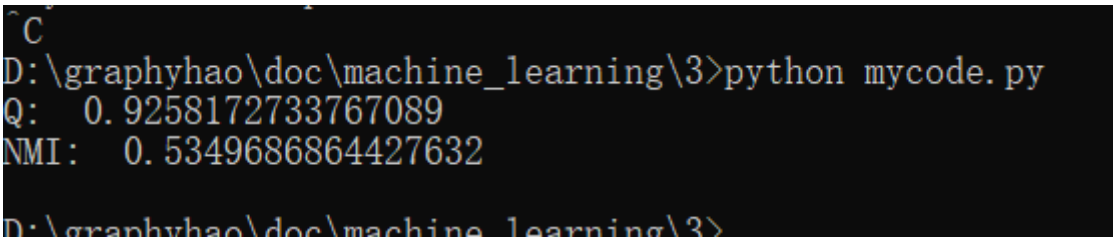
该数据集量比较小，运行非常快。

模块度Q 和NMI 如图，可以看出我实现的效果较好。

数据集2：**com-Amazon** 的对比结果 （上图为GenLouvain，下图为我的）：

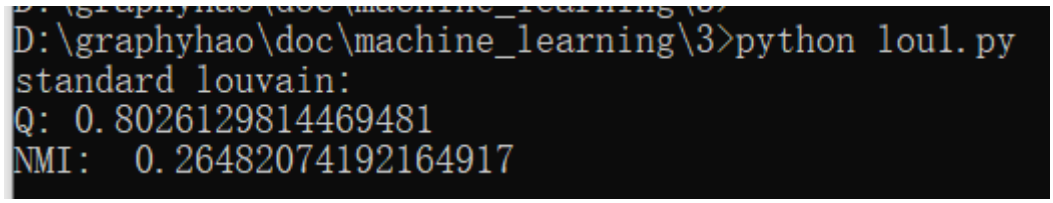
```
节点数 334863
边数  925872
```

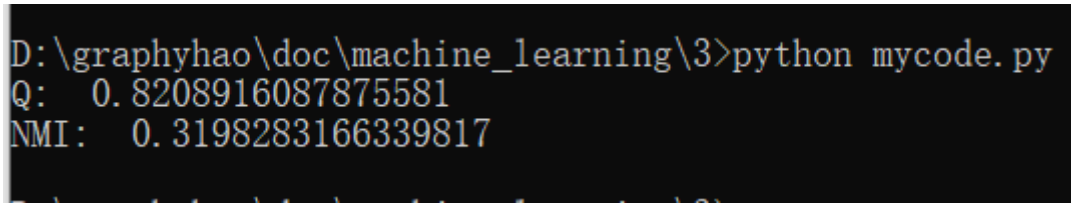




数据集3：**com-DBLP** 的对比结果 （上图为GenLouvain，下图为我的）：

```
节点数 317080
边数  1049866
```





后两个数据集的数据量较大，运行时长非常久，对最后一个数据两个程序的NMI表现都不是很好，但还是我的程序表现更加好。