机器学习与数据挖掘 作业三

吴天 19334019

1 Abstract

本次实验实现了 Fast unfolding of communities in large networks 论文中的算法，并使用 Karate、Dolphins、Email-Eu-core 数据集验证了 louvain 算法在上述数据集上的性能。同时，还将实现的算法和 python-louvain 库算法进行比较，分析两者之间的差异。

2 Assignment

(1). 实现 Modularity 算法，采用 Fast unfolding of communities in large networks（查资料）实现 modularity 的优化。

(2). 在斯坦福大学网络数据集网站 https://snap.stanford.edu/data/ 或者其他网站，找到自己认为合适的 3 个数据集，进行如下分析：

(a) 自己算法的社区发现结果，用 Normalized Mutual Information 度量效果。

(b) python-louvain 库算法的社区发现结果。

(c) 分析两者的差异并发现自己代码的问题。

3 Method

3.1 Dataset

本次实验采用的数据集是 Karate、Dolphins Dataset 和 Email-Eu-core Dataset。

其中 Email-Eu-core Dataset 共有 1005 个节点、25571 条边、42 个社区。实验数据和社区标签可以在https://snap.stanford.edu/data/email-Eu-core.html获取。

Karate Dataset 共有 34 个节点，2 个社区。Dolphins Dataset 数据集共有 62 个节点，2 个社区。实验数据可以在http://www-personal.umich.edu/ mejn/netdata/获取。

Karate 和 Dolphins 的社区标签是根据该博客中的社区划分图片得到。

https://blog.csdn.net/qq_40587374/article/details/86597293

3.2 Modularity

大多数情况下，我们是不知道网络的真实划分的，尤其是对于大型网络来说更是如此，复杂网络的命名明确的说明了这种现实情况。但是我们依然有方法可以量化或评判我们的社区划分水平，也就是模块度（Modularity），也称 Q 值。

模块度可以表示为：

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

其中，节点彼此之间共有 m 个连接。v 和 w 是图的任意两个节点，当两个节点直接相连时 $A_{vw} = 1$，否则 $A_{vw} = 0$。k_v 代表的是节点 v 的度。$\delta(c_v, c_w)$ 是用来判断节点 v 和 w 是否在同一个社区内，在同一个社区内 $\delta(c_v, c_w) = 1$，否则 $\delta(c_v, c_w) = 0$。

模块度 Q 也可以表示为：

$$Q = \sum_c (\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2)$$

其中，m 表示图中边的数量，c 表示社区，$\sum_c$ 表示社区 c 内度数之和，$\sum_{tot}$ tot 表示社区 c 节点的度之和。

这两个表达式是等价的，在算法中用到第二个式子来计算 Q 的值。

3.3 Louvain

Louvain 算法是一种贪婪优化方法，其目的是将网络划分为密集连接的节点群，优化网络的模块化。模块化被定义为 “位于社区内的边数减去随机放置边的等价网络中的期望边数”。Louvain 方法在模块性和计算时间上都优于其他所有的社区检测方法。模块性得分高表明社区内部存在紧密的联系，但社区之间的联系较稀疏，表明找到了最优解。当获得高模块性分数时，社区具有显著的实际意义。Louvain 算法可分为两个阶段，并不断重复迭代。例如我们有 N 个节点的网络：

(1). 第一个 phase，使用 Modularity Optimization 给出一个划分:

(a) 为每一个节点都分配一个 community index，即此时网络有 N 个 community，此为初始状态。

(b) 对每个节点 i，我们考虑它的邻接节点 j；我们让 i 的 community 变成 j 的，看这个动作对 modularity 的值有怎样的作用。如果这个变动带来的 ΔQ 是正的，那我们就接纳这个变动，否则就保持原来的分配方式。

(c) 不断执行，直到 Q 值不再增加。

(2). 第二个 phase，将第一个 phase 得到的划分中同一个 community 进行折叠，折叠后形成一个新的网络:

(a) community 间的连接权重为连接两个 community 的节点之权重和。

(b) community 内部的连接形成一个自环，其权重为该 community 内部连接的和。

(3). 这两个 phase 做完一轮后，称作 pass；显然每次 pass 都会让 community 的数量变小，使 Q 值增大。不断执行 pass，直到 Q 值不再增加。

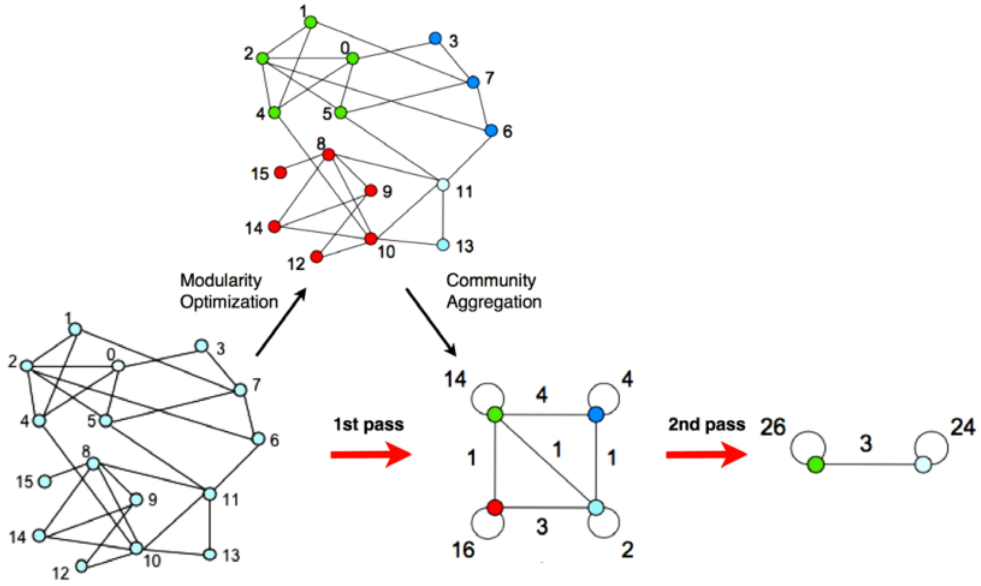


Figure 1. Visualization of the steps of our algorithm. Each pass is made of two phases: one where modularity is optimized by allowing only local changes of communities; one where the communities found are aggregated in order to build a new network of communities. The passes are repeated iteratively until no increase of modularity is possible.

图 1: Louvain 算法图示，来源论文

3.4 compute ΔQ

有 2 个版本的 ΔQ 计算公式，第一个是论文的初稿中的版本:

$$\Delta Q = [\frac{\sum_{in}+2k_{i,in}}{2m} - (\frac{\sum_{tot}+k_i}{2m})] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2]$$

第二个是论文之后发表的版本:

$$\Delta Q = [\frac{\sum_{in}+k_{i,in}}{2m} - (\frac{\sum_{tot}+k_i}{2m})] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2]$$

二者之间只有一个 $k_{i,in}$ 系数的区别。其中，m 表示图中所有边的权值和,$\sum_{in}$ 是社区 C 的内部的所有边的权值和，$k_{i,in}$ 是所有从节点 i 指向区域 C 的边的权值和,$\sum_{tot}$ 是所有指向区域 C 中的节点的边的权值和, k_i 是指向节点 i 的所有边的权值和。

将公式化简之后得到，

$$\Delta Q = \frac{1}{2m}(2k_{i,in} - \frac{\sum_{tot} * ki}{m})$$

以及

$$\Delta Q = \frac{1}{2m}(k_{i,in} - \frac{\sum_{tot} * ki}{m})$$

在上述公式中 $\frac{1}{2m}$ 在算法进行的过程中全程是一个常数，不会影响其大小的比较。因此在比较大小的过程中只需要比较 $(2k_{i,in} - \frac{\sum_{tot} * ki}{m})$ 或者 $(k_{i,in} - \frac{\sum_{tot} * ki}{m})$ 即可。

3.5 Evalution Metric

我们使用 Normalized Mutual Information(NMI) 即归一化互信息来衡量算法划分结果和真实结果的重合程度，在整个评估过程中，原始数据集提供的社区标签作为基本事实。互信息指的是两个随机变量之间的关联程度如下公式计算:

$$I(X,Y) = \sum_{x,y} p(x,y) log \frac{p(x,y)}{p(x)p(y)}$$

标准互信息是将互信息归一化到 0 1:

$$U(X,Y) = 2\frac{I(X;Y)}{H(X) + H(Y)}$$

其中:

$$H(X) = \sum_{i=1}^{n} p(x_i)I(x_i) = \sum_{i=1}^{n} p(x_i) log_b \frac{1}{p(x_i)} = -\sum_{i=1}^{n} p(x_i) log_b p(x_i)$$

NMI 值越大，代表聚类的效果越好。

4 Result

4.1 Nonlinear Classfier and Regularize

本次实验采用的数据集是 Karate、Dolphins Dataset 和 Email-Eu-core Dataset。在每个数据集上，分别应用两个不同的 ΔQ 版本的算法 ($k_{i,in}$ 系数为 2 和系数为 1)，以及 python-louvain 库，绘制社区划分结果的 NMI 图和 Q 值的图（由于 python-louvain 的

结果是随机的，这里绘制的是两次运行过程中效果最好的结果），如图 2、3 所示：

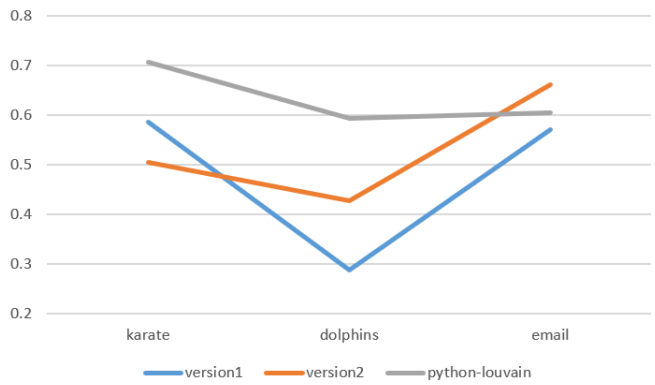


图 2: Karate、Dolphins Dataset 和 Email-Eu-core Dataset 数据集下，应用两个不同的 ΔQ 版本的算法，以及 python-louvain，得到的结果的 NMI 图

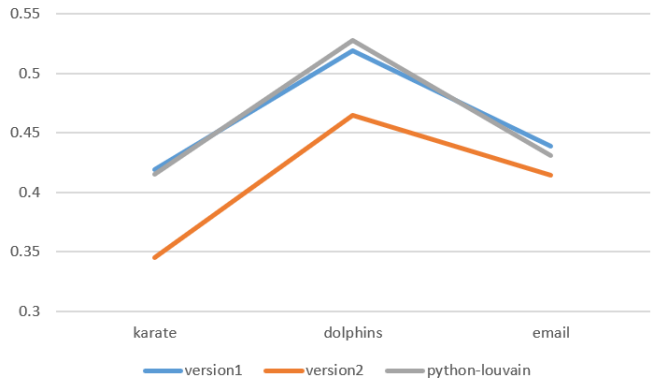


图 3: Karate、Dolphins Dataset 和 Email-Eu-core Dataset 数据集下，应用两个不同的 ΔQ 版本的算法，以及 python-louvain，得到的结果的 Q 值图

从实验结果中，我们可以看到，在 Karate、Dolphins 两个较小的数据集上，python-louvain 的社区划分效果最好，而在 Email-Eu-core Dataset 中，ΔQ 的第二个版本的方法的效果最好（图 2）。这可能是不同的方法在规模不同的数据集上表现不一，也可能是因为 Louvain 算法是贪心算法，只能收敛到局部最优而不是全局最优，所以结果存在一定的差异性。

ΔQ 的第一个版本的方法和 python-louvain 的社区划分结果的模块度相近，而 ΔQ 的第二个版本的方法的划分结果模块度是最小的（图 3）。Q 值的大小和结果的 NMI 表现不一致，是数据和 Louvain 算法存在一定随机性的问题。

在运行时间上，实现的算法运行时间在小规模数据 Karate、Dolphins 数据集上和 python-louvain 的运行时间差不多，但一旦数据的规模变大，实现的算法运行效率显著低于 python-louvain（图 4），这是需程序要改进的地方。

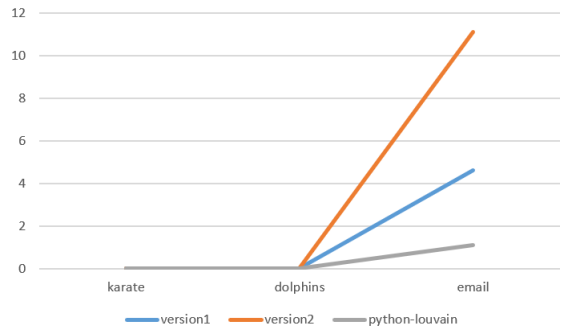


图 4: Karate、Dolphins Dataset 和 Email-Eu-core Dataset 数据集下，应用两个不同的 ΔQ 版本的算法，以及 python-louvain 的运行时间（单位:s）

5 code

代码实现中，将 Louvain 算法作为一个类，初始的参数有图的点、边、ΔQ 的计算版本，如图 5 所示：

```
class Louvain:
    '''
        Initializes the method.
        _nodes: a list of ints
        _edges: a list of ((int, int), weight) pairs
    '''
    def __init__(self, nodes, edges, option):
        self.option = option
        self.nodes = nodes
        self.edges = edges
        # precompute m (sum of the weights of all links in network)
        # k_i (sum of the weights of the links incident to node i)
        self.m = 0
        self.k_i = [0 for n in nodes]
        self.edges_of_node = {}
        self.w = [0 for n in nodes]
        for e in edges:
            self.m += e[1]
            self.k_i[e[0][0]] += e[1]
            self.k_i[e[0][1]] += e[1]
            # save edges by node
            if e[0][0] not in self.edges_of_node:
                self.edges_of_node[e[0][0]] = [e]
            else:
                self.edges_of_node[e[0][0]].append(e)
            if e[0][1] not in self.edges_of_node:
                self.edges_of_node[e[0][1]] = [e]
            elif e[0][0] != e[0][1]:
                self.edges_of_node[e[0][1]].append(e)
        # access community of a node in O(1) time
        self.communities = [n for n in nodes]
        self.actual_partition = []
```

图 5: Louvain 类

5.1 应用 Louvain 方法

Louvain 方法包含了 pass 的全部过程，并输出一些参数的值，方便观察社区划分的收敛情况。一个 pass 包含 first phase 和 second phase，每经历一个 pass，模块度都会增加，不断循环 pass，直到社区的模块度不再改变，如图 6 所示：

```
def apply_method(self):
    network = (self.nodes, self.edges)
    best_q = -1
    i = 0
    while True:
        print("第 %d 轮有 %d 个节点，%d 条边" % (i, len(network[0]), len(network[1])))
        i += 1
        partition = self.first_phase(network)
        q = self.compute_modularity(partition)
        partition = [c for c in partition if c]
        # clustering initial nodes with partition
        if self.actual_partition:
            actual = []
            for p in partition:
                part = []
                for n in p:
                    part.extend(self.actual_partition[n])
                actual.append(part)
            self.actual_partition = actual
        else:
            self.actual_partition = partition
        if q == best_q:
            break
        network = self.second_phase(network, partition)
        best_q = q
    return (self.actual_partition, best_q)
```

图 6: 应用 Louvain 方法

计算模块度，采用 Method 中的第二个 Q 值的表达式，实现如图 7 所示：

```
def compute_modularity(self, partition):
    q = 0
    m2 = self.m * 2
    for i in range(len(partition)):
        q += self.s_in[i] / m2 - (self.s_tot[i] / m2) ** 2
    return q
```

图 7: modularity 计算

5.2 first phase

一开始，每个点是一个社区，在 first phase 中，不断遍历图中的点，在所有相邻的社区中，选择把该点划入相邻社区能够最大增加模块度的社区，将该点并入该社区，不断遍历，直到社区划分不再改变，如图 8 所示：

计算 ΔQ，实现如图 9 所示：

初始化，每个点都是一个社区，并初始化权重和计算的中间值，如图 10：

用 yield 获取某节点的相邻节点，如图 11：

```python
def first_phase(self, network):
    # make initial partition
    best_partition = self.make_initial_partition(network)
    while True:
        improvement = False
        for node in network[0]:
            node_community = self.communities[node]
            # default best community is its own
            best_community = node_community
            best_gain = 0
            # remove _node from its community
            best_partition[node_community].remove(node)
            best_shared_links = 0
            for e in self.edges_of_node[node]:
                if e[0][0] == e[0][1]:
                    continue
                if e[0][0] == node and self.communities[e[0][1]] == node_community \
                    or e[0][1] == node and self.communities[e[0][0]] == node_community:
                    best_shared_links += e[1]
            self.s_in[node_community] -= 2 * (best_shared_links + self.w[node])
            self.s_tot[node_community] -= self.k_i[node]
            self.communities[node] = -1
            communities = {} # only consider neighbors of different communities
            for neighbor in self.get_neighbors(node):
                community = self.communities[neighbor]
                if community in communities:
                    continue
                communities[community] = 1
                shared_links = 0
                for e in self.edges_of_node[node]:
                    if e[0][0] == e[0][1]:
                        continue
                    if e[0][0] == node and self.communities[e[0][1]] == community \
                        or e[0][1] == node and self.communities[e[0][0]] == community:
                        shared_links += e[1]
                # compute modularity gain obtained by moving _node to the community of _neighbor
                gain = self.compute_modularity_gain(node, community, shared_links)
                if gain > best_gain:
                    best_community = community
                    best_gain = gain
                    best_shared_links = shared_links
            # insert _node into the community maximizing the modularity gain
            best_partition[best_community].append(node)
            self.communities[node] = best_community
            self.s_in[best_community] += 2 * (best_shared_links + self.w[node])
            self.s_tot[best_community] += self.k_i[node]
            if node_community != best_community:
                improvement = True
        if not improvement:
            break
    return best_partition
```

图 8: first phase

```python
def compute_modularity_gain(self, node, c, k_i_in):
    if(self.option == 1):
        return  2*k_i_in - self.s_tot[c] * self.k_i[node] / self.m
    else:
        return  k_i_in - self.s_tot[c] * self.k_i[node] / self.m
```

图 9: ΔQ 计算

```python
def make_initial_partition(self, network):
    partition = [[node] for node in network[0]]
    self.s_in = [0 for node in network[0]]
    self.s_tot = [self.k_i[node] for node in network[0]]
    for e in network[1]:
        if e[0][0] == e[0][1]: # only self-loops
            self.s_in[e[0][0]] += e[1]
            self.s_in[e[0][1]] += e[1]
    return partition
```

图 10: 初始化

```python
def get_neighbors(self, node):
    for e in self.edges_of_node[node]:
        if e[0][0] == e[0][1]: # a node is not neighbor with itself
            continue
        if e[0][0] == node:
            yield e[0][1]
        if e[0][1] == node:
            yield e[0][0]
```

图 11: 获取相邻节点

5.3 second phase

将所有处于同一社区的节点合并成一个节点，社区之间的所有边变成一条边，将图简化，更新它们的权重，同时也更新中间计算值，如图 12 所示：

```python
def second_phase(self, network, partition):
    nodes_ = [i for i in range(len(partition))]
    # relabelling communities
    communities_ = []
    d = {}
    i = 0
    for community in self.communities:
        if community in d:
            communities_.append(d[community])
        else:
            d[community] = i
            communities_.append(i)
            i += 1
    self.communities = communities_
    # building relabelled edges
    edges_ = {}
    for e in network[1]:
        ci = self.communities[e[0][0]]
        cj = self.communities[e[0][1]]
        try:
            edges_[(ci, cj)] += e[1]
        except KeyError:
            edges_[(ci, cj)] = e[1]
    edges_ = [(k, v) for k, v in edges_.items()]
    # recomputing k_i vector and storing edges by node
    self.k_i = [0 for n in nodes_]
    self.edges_of_node = {}
    self.w = [0 for n in nodes_]
    for e in edges_:
        self.k_i[e[0][0]] += e[1]
        self.k_i[e[0][1]] += e[1]
        if e[0][0] == e[0][1]:
            self.w[e[0][0]] += e[1]
        if e[0][0] not in self.edges_of_node:
            self.edges_of_node[e[0][0]] = [e]
        else:
            self.edges_of_node[e[0][0]].append(e)
        if e[0][1] not in self.edges_of_node:
            self.edges_of_node[e[0][1]] = [e]
        elif e[0][0] != e[0][1]:
            self.edges_of_node[e[0][1]].append(e)
    # resetting communities
    self.communities = [n for n in nodes_]
    return (nodes_, edges_)
```

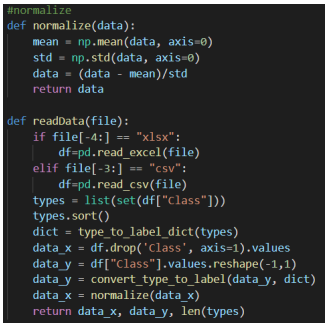
图 12: second phase

5.4 读取数据

数据集均被保存成.gml 或.txt 文件格式，读取数据，获得节点和边，并且将节点的序号改变成从 0 开始计算，读取数据的实现代码见图 13：

```python
def read_data(path):
    f = open(path, 'r')
    lines = f.readlines()
    f.close()
    nodes = {}
    edges = []
    if (path[-3:] == "txt"):
        for line in lines:
            n = line.split()
            if not n:
                break
            start = int(n[0])
            end = int(n[1])
            nodes[start] = 1
            nodes[end] = 1
            w = 1
            if len(n) == 3:
                w = int(n[2])
            edges.append(((start, end), w))
    elif (path[-3:] == "gml"):
        current_edge = (-1, -1, 1)
        in_edge = 0
        for line in lines:
            words = line.split()
            if not words:
                break
            if words[0] == 'id':
                nodes[int(words[1])] = 1
            elif words[0] == 'source':
                in_edge = 1
                current_edge = (int(words[1]), current_edge[1], current_edge[2])
            elif words[0] == 'target' and in_edge:
                current_edge = (current_edge[0], int(words[1]), current_edge[2])
            elif words[0] == 'value' and in_edge:
                current_edge = (current_edge[0], current_edge[1], int(words[1]))
            elif words[0] == ']' and in_edge:
                edges.append((current_edge[0], current_edge[1]), 1))
                current_edge = (-1, -1, 1)
                in_edge = 0
```

(a) 读取数据



```python
#normalize
def normalize(data):
    mean = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    data = (data - mean)/std
    return data

def readData(file):
    if file[-4:] == "xlsx":
        df=pd.read_excel(file)
    elif file[-3:] == "csv":
        df=pd.read_csv(file)
    types = list(set(df["Class"]))
    types.sort()
    dict = type_to_label_dict(types)
    data_x = df.drop('Class', axis=1).values
    data_y = df["Class"].values.reshape(-1,1)
    data_y = convert_type_to_label(data_y, dict)
    data_x = normalize(data_x)
    return data_x, data_y, len(types)
```

(b) 下标转换

图 13: gml 或 txt 数据读取

5.5 main 函数

main 函数的主要作用是使用实现的方法和 python-louvain 库对数据集进行社区划分，获取二者的划分结果，得到 NMI 和最终的 Q 值。(不同数据通过 python-louvain 得到的划分的节点的数据类型不太相同，比如 gml 最后是 string 类型，karate 数据的下标是从 0 开始，而 dolphins 是从 1 开始，因此转换成可比较的标签时，部分代码有差异)，main 函数实现代码见图 14:

6 code distinction

通过查看 python-louvain 的源码，发现实现的算法与 python-louvain 的方法的区别有如下三点:

(1). 数据结构的差异: 实现的算法中使用的数据结构都是 python 原生的数据结构来存储图和各种计算中间值，非常繁琐。而 python-louvain 使用的是 networkx 库来建图，获取各种图的属性也更加便捷高效。我分析实现的代码在大规模网络中耗时过长原因之一是图的遍历和各种参数的更新太慢。

(2). first phase 中的随机性: 作者在论文中叙述，这个阶段中，节点遍历的顺序并不影响结果，但会影响运行时间，python-louvain 做的是把节点先打乱，那么在遍历的时候就是随机遍历了一遍，这样可以在一定程度上避免出现结果运行时间过长的情况。同时由于 Louvain 是贪心算法，只能收敛到局部最优而不是全局最优，增加随机性也可以产生出不同的划分结果。python-louvain 有随机，而在这次作业中的算法实现没有加入随机。

(3). ΔQ 的计算: python-louvain 中实现的 ΔQ 的计算采用的是 $k_{i,in}$ 系数为 1 的版本，

而在这次作业中算法的实现，ΔQ 的计算是一个选项，两种计算方法都可以尝试。

7 Summary

通过这次实验，我对老师课堂上讲到社区分类理论知识的掌握更加深入了，并且完成 Louvain 算法实现，感觉收获很大。

同时，由于代码效率太低，一旦网络规模变大，运行时间就变得格外漫长。并且，受限于自己电脑的配置，没有在超大网络的数据集进行运行代码，这也是这次实验不足的一点。

8 Supplementary material

本实验的代码已上传至 GitHub，可以在我的 Github 获得。

GitHub 仓库是https://github.com/PointerA/ML_DM_course

```
#数据和标签的输入如下：
#data_paths = ['data/karate.gml', 'data/dolphins.gml', 'data/email-Eu-core.txt']
#label_paths = ['data/karate_label.txt', 'data/dolphins_label.txt','data/email-Eu-core-department-labels.txt']

################ arg ########################
parser = argparse.ArgumentParser()
parser.add_argument("--data_path", type=str, default="data/dolphins.gml", help="data_path is the path of the net")
parser.add_argument("--label_path", type=str, default="data/dolphins_label.txt", help="data_path is the path of the net")
parser.add_argument("--delta_Q", type=int, default=2, help="delta_Q version")
conf = parser.parse_args()
```

(a) 获取参数

```
################## 读入label ##########################
label_path = conf.label_path
f = open(label_path, 'r')
lines = f.readlines()
f.close()
true_label = []
for line in lines:
    n = line.split()
    if not n:
        break
    true_label.append(n[1])
```

(b) 读入真实标签

```
################ 读数据并调用实现的算法 ######################
data_path = conf.data_path
net_G_nodes,net_G_edges = read_data(data_path)
new_G = Louvain(net_G_nodes, net_G_edges, conf.delta_Q)  #构造成一个类，传入的参数依次是点集和边集
time_start = time.time()  # 记录开始时间
partition1,Q1 = new_G.apply_method() #应用其中的方法，对社区进行分割，返回值分别是最佳的社区划分，以及对应的Q值.
time_end = time.time()  # 记录结束时间
fastunfolding_label = [0]*len(net_G_nodes)
for i,part in enumerate(partition1):
    for node in part:
        fastunfolding_label[node] = i
fastunfolding_nmi = normalized_mutual_info_score(fastunfolding_label, true_label)
print("fastunfolding_nmi:",fastunfolding_nmi)
print("fastunfolding_Q:",Q1)
print("fastunfolding time:%.3fs"  % (time_end - time_start))# 计算的时间差为程序的执行时间，单位为秒/s
```

(c) 实现的方法

```
################ 调用python-louvain的算法 ######################
if (data_path[-3:] == "txt"):  #如果文件是txt格式
    net_G = nx.read_edgelist(data_path)
elif (data_path[-3:] == "gml"):  #如果文件是gml格式
    net_G  = nx.read_gml(data_path,label="id")
time_start = time.time()  # 记录开始时间
partition2 = community.best_partition(net_G)
Q2 = community.modularity(partition2,net_G)
time_end = time.time()  # 记录结束时间
pylouvain_label = []
for i in range(len(net_G_nodes)):
    if(data_path[-3:] == "txt"):
        pylouvain_label.append(partition2[str(i)])      #email-Eu-core.txt
    elif(data_path[-5:] == "e.gml"):
        pylouvain_label.append(partition2[i+1])      #karate.gml
    else:
        pylouvain_label.append(partition2[i])      #dolphins.gml

pylouvain_nmi = normalized_mutual_info_score(pylouvain_label, true_label)
print("pylouvain_nmi:",pylouvain_nmi)
print("pylouvain_Q:",Q2)
print("pylouvain time:%.3fs" % (time_end - time_start))# 计算的时间差为程序的执行时间，单位为秒/s
```

(d) python-louvain 的方法

图 14: main 函数