金庸的江湖——金庸武侠小说中的人物关系挖掘

组长：韩畅，组员：李展烁、王一之、闫旭芃

2020 年 7 月 30 日

1 实验规划与设计

1.1 任务分配

171860551, 韩畅：组长,算法设计与实验规划，任务一、任务六的可视化部分
171860550, 王一之：算法设计与实验规划,任务四
171860549, 闫旭芃：算法设计与实验规划,任务五,任务六的任务四五数据处理部分
171840565, 李展烁：算法设计与实验规划,任务二、任务三

1.2 任务要求

1.3 设计思路

2 实验实现

2.1 任务一

todo

2.2 任务二

todo

2.3 任务三

todo

2.4 任务四:基于人物关系图的PageRank计算

2.4.1 PageRank算法介绍

PageRank，又称网页排名，名字源于google创始人之一的Larry Page，是Google公司所使用的对与网页重要性排序的算法。
PageRank通过网页之间的超链接评价网页重要性，它的基本思想是：

1) 如果一个网页被多个网页所指向，则该网页比较重要

2) 如果一个重要的网页指向另一个网页，则另一个网页也比较重要

该算法模拟一个上网者，随机打开一个网页，之后随机点击该网页的链接，统计上网者分布在每个网页的概率。
最初，每个网页的概率均等，每次跳转时，网页X将其PR(PageRank)均分到所指向的所有页面，记链接数为L(X), 于是，经过一次跳转后：

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + ...$$

我们将每个网页抽象成一个节点，超链接抽象为有向边，共同构成一个图。则每次跳转可视为所有页面PR构成的特征向量R与该图的出度邻接矩阵M相乘，即：

$$R = \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_n) \end{bmatrix} \quad M = \begin{bmatrix} p_1 \to p_1 & p_2 \to p_1 & \cdots & p_n \to p_1 \\ p_1 \to p_2 & p_2 \to p_2 & \cdots & p_n \to p_2 \\ \vdots & \vdots & \ddots & \vdots \\ p_1 \to p_n & p_2 \to p_n & \cdots & p_n \to p_n \end{bmatrix}$$

$$R_1 = MR_0$$

多次迭代后，PR值趋于稳定，即为最终的PR值。

2.4.2 设计思路

任务四的输入为任务三的输出，格式如下：
人物[名字$_1$,影响$_1$—名字$_2$,影响$_2$—...—名字$_n$,影响$_n$]
影响$_i$ 为名字$_i$ 与该人物归一化后的同现次数，表示名字$_i$ 对该人物的影响权重。
每个人物视为图的一个节点，边权重为二人同现次数。
对于普通的PageRank计算，由于会存在自环边以及无出度的节点，为方式到达某一节点后陷入该点，会加入“随机浏览者”（random surfer）的概念，即到达某个节点后有一定概率直接跳转到任意一个节点，从而避免此情况。然而在此次任务中，首先没有自身与自身同现的情况，因此无自旋边；同时A与B同现，则B一定与A也同现，因此不考虑权重时所有边实际都为无向边，因此不存在出度为0的节点。所以此次任务无需引入“随机浏览者”。

MapReduce框架下，运算分布进行，因此不使用邻接矩阵，而采用邻接表的形式。算法大致分为三阶段：

阶段一：预处理首先要将输入格式化为供之后迭代处理的形式。采用如下格式：key：人物
value：PageRank#[名字$_1$,影响$_1$—名字$_2$,影响$_2$—...—名字$_n$,影响$_n$]
以概率为初始值，PageRank应设置为1/N，但N值较大，较小数字做乘法时误差较大，因此将初始PR设置为1来减小误差。

阶段二：迭代计算迭代计算PR值，直到PR收敛。
在Mapper中，首先输出如下键值对：key：人物
value：#出度表

此对目的在于维护出度表，value前加#使reducer便于区分。
之后计算PR值，记A的出度表集合为N,则计算过程如下式：

$$NewPR(A) = \sum_{x \in N} OldPR(x) * weight(x \to A)$$

计算得到新的PR值，再输出一组键值对：key：人物
value：新PageRank值
在Reducer中，首先查看value前是否有#号以区分该键值对类型。由于是一个迭代过程，将输出格式化，与Mapper的输入格式相同。

阶段三：处理结果在Mapper阶段去除结果中的出度表，只保留PR值。
利用Partition类进行排序，由于默认为升序，结果需要降序，因此重写DoubleWritable。因为只有1000余数据，未采用采样排序，使用了简单的全排序。
Reducer阶段整理输出即可

2.4.3 代码讲解

程序可分为三个模块：PageRank、RageResultSort以及调度模块。模块一：PageRank
此模块包含了阶段一与阶段二。

```
class PageRankMapper extends Mapper<Text, Text, Text, Text> {
    private Text resKey = new Text();      // 要传入reduce的key和value
    private Text resValue = new Text();

    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();
        Double rank = 0.0;
        String relations = new String();
        if (value.toString().contains("#")) {// 不是第一次迭代
            rank = Double.parseDouble(value.toString().split("#")[0]); // 原本rank
            relations = value.toString().split("#")[1];       // 出度表 格式为[名字,影响|名字,影响|...]其中影响为对key中人物的影响
        } else {//第一次迭代，设置默认rank
            rank = 1.0;                // 初始化rank值为1
            relations = value.toString(); // 出度表
        }
        resKey.set(name);
        resValue.set("#" + relations);
        context.write(resKey, resValue);  // key为名字 value以#开头，后面为出度表，生成此键值对

        String[] slist = relations.split("\\[|\\]")[1].split("\\|");    // 分离出度表
        String target = new String();
        Double weight = 0.0;
        Double res = 0.0;
        for (String s : slist) {
            target = s.split(",")[0];   // 贡献影响的人名
            weight = Double.parseDouble(s.split(",")[1]);   // 影响值作为权重
            resKey.set(target);
            res = weight * rank;//得到此人对其造成的影响值
            resValue.set(res.toString());
            context.write(resKey, resValue);  // 发送键值对，key为受影响的人物名字，value为造成的影响值
        }

    }
}
```

图 1. PageRankMapper

2.5 任务五:在人物关系图上的标签传播

思路分析：在本实验里，我们拟采用LPA*LabelPropagationAlgorithm* 标签传播算法对金庸小

说中的人物进行聚类。LPA 是一种半监督的图分析算法，能够为图中的每一个节点打上标签，对图的顶点进行聚类，从而在一张类似社交网络的图中完成社区发现。

2.5.1 LPA算法介绍

算法思路：

对于网络中的每一个节点，在初始阶段，Label Propagation算法对于每一个节点都会初始化一个唯一的一个标签。每一次迭代都会根据与自己相连的节点所属的标签改变自己的标签，更改的原则是选择与其相连的节点中所属标签最多的社区标签为自己的社区标签，这就是标签传播的含义了。随着社区标签不断传播。最终，连接紧密的节点将有共同的标签。

算法过程：

第一步：先给每个节点分配对应标签，即节点1对应标签1，节点i对应标签i；

第二步：遍历N个节点（for i=1：N），找到对应节点邻居，获取此节点邻居标签，找到出现次数最大标签，若出现次数最多标签不止一个，则随机选择一个标签替换成此节点标签；

第三步：若本轮标签重标记后，节点标签不再变化（或者达到设定的最大迭代次数），则迭代停止，否则重复第二步

具体伪代码如下图：

Input: Network $G = (V, E)$
Output: Label C(x) for each node
1: 初始化网络中所有的顶点，对于顶点 x，其标签 $C_0(x) = x$
2: **repeat**
3: 每一个顶点 x 向邻接的顶点发送自己标签 $C_t(x)$
4: 每一个顶点 x 从邻接顶点 $\{x_{n1}, x_{n2}, ..., x_{nk}\}$ 选择出现次数最多的标签作为自己新的标签 $C_{t+1}(x)$
5: **until** 每一个顶点的标签就是其邻接顶点中出现次数最多的标签

图 2. LPA算法伪代码

对于金庸小说中的人物关系的标签分析来说，其中人物的网络可以看成一个每条边带权值的有向图，因此我们的算法也要在此基础上做一些改变。首先我们还是对每一个顶点进行初始化，然后每一个顶点根据自己的出边向邻接顶点发送自己的标签。最后每个顶点从自己收到的标签里取权值和最大的标签作为自己新的标签（原算法的出现次数之和需要改成权值之和），相当于原算法中。当每一个顶点的标签不再改变时，算法终止。

具体分为3个部分进行:

2.5.2 第一部分：LabelInitial 社区初始化

第一个阶段，初始化每一个顶点，将每一个节点的人物姓名作为该节点的标签。(人名，(标签#邻接表))

Map：一行一行分析实验三的输出结果，直接将每个顶点的人名作为每个顶点的标签，输出

Reduce：不做任何处理直接输出(人名，(标签#邻接表))

Mapper类：

一行行读入任务三输出的结果(人名，邻接表)，然后将人名作为该任务的标签以(人名，(标签，邻接表)) 的形式输出。初始时，每一个节点都是一个独立的社区，所以这里直接将标签赋值成该节点的人物姓名即可

```java
class LabeliniMapper extends Mapper<Text, Text, Text, Text> {
    private Text rKey = new Text();
    private Text rValue = new Text();
    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();
        String Lablelist = value.toString();
        Lablelist=name+"#"+Lablelist;
        rKey.set(name);
        rValue.set(Lablelist);
        context.write(rKey, rValue);
        //context.write(new Text(name), new Text(name + "#" + list));
    }
}
```

图 3. LabelInitial Mapper

Reduce类：

直接输出即可，同时在main函数中设置job.setNumReduceTasks=1输出一个文件即可

2.5.3 第二部分：LabelAnalyse LPA社区聚类迭代

第二个阶段，读入第一部分处理好的数据，分别发送邻接表(节点，邻接表)和每一条边（即(节点的邻居人名，节点标签:二者所连边的权重)），之后利用相同key将同一个节点和不同标签的边分在同一个reducer的同一个组里面，之后选择出现权重和最大的标签作为新的标签即可。之后累次循环迭代直到标签稳定。

同时，LPA当遇到二分图的时候，会出现标签震荡，这里当迭代轮数超过10 轮后就会开始震荡，因此我们这里可以选择迭代15 -20轮。

伪代码：

```
Input: 人物关系网络权重图
Output: 每个人物的标签和人物关系网络权重图
 1: function MAP(nid n, node N)
 2:     emit(nid n, N.ADJACENCYLIST)
 3:     for each nodeid m in N.ADJACENCYLIST do
 4:         emit(m,N.label and weight(n→m))
 5:     end for
 6: end function
 7:
 8: function REDUCE(nid m, [p1, p2, ...])
 9:     M ← ∅
10:     H ← HASHMAP
11:     for each s in [p1, p2, ...] do
12:         if ISNODE(s) then
13:             M ← s
14:         else
15:             if H.CONTAINS(s) then
16:                 Update (s.label, s.oldWeight + s.newWeight)
17:             else
18:                 Insert (s.label, s.weight)
19:             end if
20:         end if
21:     end for
22:     M.label ← The key which has the maximum value in H
23:     emit(nid m, node M)
24: end function
```

图 4. LabelAnalyse伪代码

具体程序：

1. Mapper类： 首先读入第一部初始化好的数据，首先发送(原节点人名，#邻接表)用于维护每一个节点的网络结构，以便下一次迭代使用

然后拆开邻接表中每一个邻居节点，将邻居节点姓名作为key，将该节点标签和权值作为value，发送(节点的邻居人名，节点标签:二者所连边的权重)

```
class LabelAnaMapper extends Mapper<Text, Text, Text, Text> {
    @Override
    protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        String name = key.toString();//名字
        String list = value.toString();//value （标签+邻接表）
        String label = list.split("#")[0];//标签
        String relationlist = "#"+list.split("#")[1];//#+邻接表
        context.write(new Text(name), new Text(relationlist));//发送邻接表维护网络结构
        String[] slist = relationlist.split("\\[|\\]")[1].split("\\|");//将每个关系分离

        for (String s : slist) {
            String target = s.split(",")[0];//和名字主人产生影响的人名
            String weight = s.split(",")[1];//权值
            String LabelWeight=label+":"+weight;
            context.write(new Text(target), new Text(LabelWeight));
        }
    }
}
```

图 5. LabelAnalyse Mapper

1. Reducer类： 当读入(人名, #邻接表)时，直接保存邻接表维护网络结构，以便下一次迭代使用。

当读入（节点的邻居人名，节点标签:二者所连边的权重）时，首先创建一个HashMap 数据结构，维护每一个（标签，权重），循环读入所有的这种输入。如果收到的该标签还没有出现过，那么以标签为key，权值为value插入哈希表中。如果标签已经出现过，则标签权值在原来的权值基础上累加上去。

然后循环比较，取权值和最大的标签作为新的标签，还是以原来的格式(节点名称，标签#邻接表) 输出作为结果

```
protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
        InterruptedException {
    HashMap<String, Double> neighborMap = new HashMap<>();
    String list = "";
    for (Text value : values) {
        String tempstr = value.toString();
        if (tempstr.startsWith("#")) {
            list = tempstr.substring(1);//从#后面截取整个邻接表
        } else {
            String l = tempstr.split(":")[0];//标签
            Double w = Double.parseDouble(tempstr.split(":")[1]);//对应权值
            if (!neighborMap.containsKey(l)) {//如果哈希表中没有这个标签
                neighborMap.put(l, w);//将标签，权值加入哈希表中
            } else {
                Double m = neighborMap.get(l) + w;//将权值加上
                neighborMap.put(l, m);
            }
        }
    }
    String label = "";
    Double maxWeight = 0.0;
    for (Map.Entry<String, Double> entry : neighborMap.entrySet()) {
        String keylable = entry.getKey();
        Double veight = entry.getValue();
        if (veight > maxWeight) {//循环比较找到权值最大的标签
            maxWeight = veight;
            label = keylable;
        }
    }
    String reslist = label + "#" + list;
    context.write(key, new Text(reslist));

}
```

图 6. LabelAnalyse Reducer

1. Main函数： 设置一个循环即可，循环20次，每次的输出变成下一次的输入

```java
public class LabelAnalyse {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("PageRank neet 2 paras as input and output");
            System.exit(2);
        }
        String input = otherArgs[0];                // 输入目录
        String output = new String();
        int maxIteration = 15;
        for (Integer i = 0; i < maxIteration; i++) {
            if (i == maxIteration-1) {
                output = otherArgs[1] + "final";    // 最终输出
            } else {
                output = otherArgs[1] + i.toString();    // 中间文件输出目录
            }
            Job job = Job.getInstance(conf, "LabelAnalyse");
            job.setJarByClass(LabelAnalyse.class);
            job.setMapperClass(LabelAnaMapper.class);
            job.setReducerClass(LabelAnaReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            job.setInputFormatClass(KeyValueTextInputFormat.class);// 以/t切割key与value
            job.setNumReduceTasks(5);//5个reducer
            FileInputFormat.addInputPath(job, new Path(input));
            Path path = new Path(output);
            FileSystem fileSystem = path.getFileSystem(conf);// 根据path找到这个文件
            if (fileSystem.exists(path)) {
                fileSystem.delete(path, true);// 设置true
            }
            FileOutputFormat.setOutputPath(job, new Path(output));
            job.waitForCompletion(true);
            input = output;
```

图 7. LabelAnalyse Main函数

2.5.4 第三部分：LabelResult 结果整理（包括任务六的一部分）

这里因为第二部分输出的结果中还有邻接表的存在，而且结果比较杂乱，所以直接将任务六数据整理整合到这一部分。在mapper中读入(节点名，标签#邻接表)，去除邻接表，并且将标签和节点名反转，将标签设置成key，输出(标签，节点名)以便之后归类

其中重写了partition类，将首字母拼音在a-h，h-p，p-z分为三个区域，分三个文件存储，这个功能可加可不加，如果不加的话那么就输出一个文件，文件中相同标签会在一起，比较直观。

reducer部分直接输出((标签,人名),空)

```java
class LabelResultMapper extends Mapper<Text, Text, Text, Text> {
    // private Text resKey = new Text(); // 要传入reduce的key和value
    private Text rKey = new Text();
    private Text rValue = new Text();

    @Override
    protected void map(Text key, Text value, Context context) throws IOException, Interr
        String name = key.toString();
        String lable = value.toString().split("#")[0];// 去除后面的关系表，只保留前面的标签
        lable = lable.trim();// 去掉前后空白符
        rKey.set(lable);// 将一个标签的人输入到一起
        rValue.set(name);
        context.write(rKey, rValue);
    }
}
```

(a) LabelResult Mapper

```java
class LabelResultReducer extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        for (Text name : values) {
            context.write(new Text(name.toString()+","+key.toString()),new Text(""));//中间加一个逗号
        }
    }
}
```

(b) LabelResult Reducer

```java
class ThreePartition extends Partitioner<Text,Text>{
    @Override
    public int getPartition(Text value1, Text value2, int i) {
        String part1="好";//首字母h
        String part2="跑";//首字母p
        Collator collator = Collator.getInstance(Locale.CHINA);
        CollationKey key1 = collator.getCollationKey(part1);
        CollationKey key2 = collator.getCollationKey(part2);
        CollationKey key = collator.getCollationKey(value1.toString());
        if(key.compareTo(key1)<=0){//首字母比较，分三个区
            return 0;
        }else if(key.compareTo(key2)<=0){
            return 1;
        }
        return 2;
    }
}
```

(c) LabelResult Partitioner

图 8. LabelResult 部分代码截图

2.5.5 总结：

LPA算法优点： LPA算法的最大的优点就是算法的逻辑非常简单，相对于优化模块度算法的过程是非常快的，不用pylouvain那样的多次迭代优化过程。

LPA算法利用自身的网络的结构指导标签传播，这个过程是无需任何的任何的优化函数，而且算法初始化之前是不需要知道社区的个数的，随着算法迭代最后可以自己知道最终有多少个社区利用MapReduce可以很好的利用其特性来进行迭代分析

LPA算法缺点： 划分结果不稳定，随机性强是这个算法致命的缺点。具体体现在：

1. 更新顺序： 节点标签更新顺序随机，但是很明显，越重要的节点越早更新会加速收敛过程；

2. 随机选择： 如果一个节点的出现次数最大的邻居标签不止一个时，随机选择一个标签作为自己标签。这种随机性可能会带来一个雪崩效应，即刚开始一个小小的聚类错误会不断被放大。不过如果相似邻居节点出现多个，可能是权重weight计算的逻辑有问题，需要回过头去优化weight抽象和计算逻辑；

2.6 任务六

todo

3 优化与改进

3.1 任务一

todo

3.2 任务二

todo

3.3 任务三

todo

3.4 任务四

原本迭代次数为20次，在检查中间结果时发现在14次之后，结果变化不大，基本收敛，因此将迭代次数改为15次，节省开销。

3.5 任务五

LPA当遇到二分图的时候，会出现标签震荡，这里当迭代轮数超过10 轮后就会开始震荡，因此我们这里可以选择迭代15 -20轮。

3.6 任务六

todo

4 实验经验总结与改进方向

1) todo

2) todo