

华中科技大学
Huazhong University of Science and Technology

课程实验报告

课程名称： 大数据分析

专业班级： 校交 1901 班

学 号： U201910681

姓 名： 骆瑞霖

指导教师： 崔金华

报告日期： 2021-12-12

计算机科学与技术学院

目录

1 实验一 wordCount 算法及其实现	1
1.1 实验目的	1
1.2 实验内容	1
1.3 实验过程	2
1.3.1 编程思路	2
1.3.2 遇到的问题以及解决方式	4
1.3.3 实验测试与结果分析	4
1.4 实验总结	5
2 实验二 PageRank 算法及其实现	6
2.1 实验目的	6
2.2 实验内容	6
2.3 实验过程	6
2.3.1 编程思路	6
2.3.2 遇到的问题及解决方式	9
2.3.3 实验测试与结果分析	9
2.4 实验总结	9
3 实验三 关系挖掘实验	10
3.1 实验内容	10
3.2 实验过程	10
3.2.1 编程思路	10
3.2.2 遇到的问题及解决方式	12
3.2.3 实验测试与结果分析	12
3.3 实验总结	12
4 实验四 kmeans 算法及其实验	14
4.1 实验目的	14
4.2 实验内容	14
4.3 实验过程	14
4.3.1 编程思路	14
4.3.2 遇到的问题及解决方式	16
4.3.3 实验测试与结果分析	17
4.4 实验总结	17
5 实验五 推荐系统	18
5.1 实验目的	18
5.2 实验内容	18
5.3 实验过程	19

5.3.1	编程思路	19
5.3.2	遇到的问题及解决方式	24
5.3.3	实验测试与结果分析	25
5.4	实验总结	27

1 实验一 wordCount 算法及其实现

1.1 实验目的

1. 理解 map-reduce 算法思想与流程；
2. 应用 map-reduce 思想解决 wordCount 问题；
3. 掌握并应用 combine 与 shuffle 过程。

1.2 实验内容

提供 9 个预处理过的源文件（source01-09）模拟 9 个分布式节点，每个源文件中包含一百万个由英文、数字和字符（不包括逗号）构成的单词，单词由逗号与换行符分割。

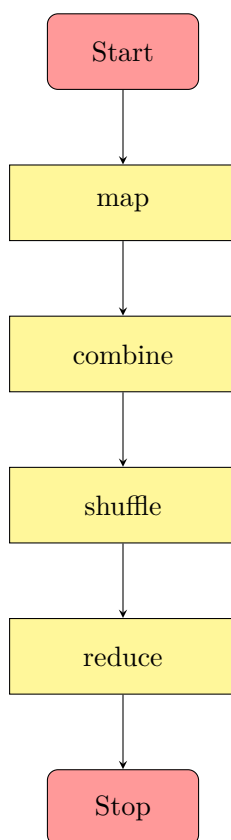
要求应用 map-reduce 思想，模拟 9 个 map 节点与 3 个 reduce 节点实现 wordCount 功能，输出对应的 map 文件和最终的 reduce 结果文件。由于源文件较大，要求使用多线程来模拟分布式节点。

在 map-reduce 的基础上添加 combine 与 shuffle 过程，并可以计算线程运行时间来考察这些过程对算法整体的影响。实现 shuffle 过程时应保证每个 reduce 节点的工作量尽量相当，来减少整体运行时间。

1.3 实验过程

1.3.1 编程思路

首先根据 map-reduce 的思想可以得到实验的编程流程图大致如下：



map, combine, reduce 等过程均可以使用多线程实现，下面分别进行编程思路的叙述：

- map

map 过程是将文本中的所有单词全部提取出来，并为每一个单词打上 1 的标签，这里使用多线程实现，定义一个数组 thread_list 来作为线程的列表，每一个线程的执行目标函数为 mapper，参数就是几号文件；

```
1 # Python code
2 thread_list = []
3 for i in range(1, 10):
4     thread_list.append(threading.Thread(target=mapper, args=(i, )))
```

- combine

combine 过程的目的是将 map 后的每一个文件中的结果，相同词汇的次数累计起来，得到新的词典；同样使用多线程完成，由于 combine 后词汇行数会减小很多，所以不用文件存储了，直接得到新的词汇到出现次数的哈希，存入一个列表中，由于需要得到线程执行函数的返回值，

这里定义类 `CombineThread` 继承 `threading.Thread` 类，在其中定义方法 `get_result` 来获取该线程任务执行完毕后的结果；使用 `join` 方法使得所有创造的子线程执行结束后主线程再将所有生成的 9 个新词典加入列表。

`CombineThread` 类定义如下：

```
1 # Python code
2 class CombineThread(threading.Thread):
3     def __init__(self, func, args=()):
4         super(CombineThread, self).__init__()
5         self.func = func
6         self.args = args
7
8     def run(self):
9         self.res = self.func(*self.args)
10
11     def get_result(self):
12         try:
13             return self.res
14         except Exception as e:
15             return None
```

程序执行完毕后得到大小为 9 的列表 `combine_dict_list`，其中每一个元素都是一个原文本的词汇字典，key 为词汇，value 为在文本中的出现次数。

- shuffle

在 shuffle 过程中，我们需要将 combine 后的 9 个字典中的键值对分配给 3 个 reduce 结点来完成最后的 reduce 过程，在 shuffle 过程中我们应该保证三个 reduce 结点的工作负载是差不多的，我们需要选取一个策略将词汇进行分配，考虑到最后 reduce 的结果应该是按照字典序排序的，不如就按照键值对中词汇键的首字符来分区，经过计数，大写和符号开头的词汇占比相比小写字母开头的词汇要少很多，所以我的策略是将大写开头、符号开头，小写字母 a c 开头的单词分进第一个 reduce 结点中，剩下的字母以首字母'o' 作为划分，shuffle 操作完毕后使用三个文件记录三个 reduce 结点应该处理的数据。

- reduce

在 reduce 过程中，每一个结点分别处理 shuffle 过程输出的三个文件中的数据，将不同分区中相同词汇的出现次数进行累计，然后三个结点分别输出按键升序的排列结果，由于三个分区的结果已经按照首字母顺序划分过了，所以直接按第 1 到第 3 个结点的顺序进行最终结果的写入就得到了 map-reduce 的最终结果，输出到文件。

其中 reduce 的子线程需要完成的目标函数为 `Reducer()`；

```
1 # Python code
2 def Reducer(order):
3     sourcefile = './shuffle/shuffle0' + (str)(order)
4     dict = {}
5     for line in open(sourcefile):
6         element = line.split('\t')
7         word, num = element[0], (int)(element[1])
8         if word in dict:
9             dict[word] += num
10        else:
11            dict.update({word : num})
12    dict = sorted(dict.items(), key=lambda d: d[0])
13    filename = './reducer/reducer0' + (str)(order)
14    w = open(filename, 'w')
15    for ind in range(len(dict)):
16        w.write(dict[ind][0] + '\t' + str(dict[ind][1]) + '\n')
17    w.close()
```

1.3.2 遇到的问题以及解决方式

- 对于 map_reduce 过程记忆不够清晰，通过查找博客的方式对这个过程有了更深的印象，并将其应用到了代码实现中。

- 对于 shuffle 过程，思考如何让所有的 reduce 结点所花处理时间得到合理的分配，通过在网寻找一些常见的 shuffle 策略以及针对单词数据类型本身特点，最后制定了上述的分配策略。

1.3.3 实验测试与结果分析

实验要求输出对应的 map 文件和最终的 reduce 结果文件，map 部分代码执行完毕后得到 9 个文件命名为 mapper01~mapper09 分别存储 9 个文件的 map 后的结果；

```
nonritualistic 1
freedoot 1
coleslaws 1
argalas 1
Essam 1
Vinnie 1
immaculateness 1
Listerised 1
moisturizer 1
dicrotic 1
dingeys 1
Ajanta 1
```

图 1: map 运行结果

reduce 部分程序执行完毕后得到输出文件 ans，就是最终 map_reduce 的结果，部分输出如下图所示：

```
zygodont 25
zygogenesis 30
zygogenetic 14
zygoid 18
zygolabialis 21
zygoma 22
zygomas 9
zygomata 11
zygomatic 16
zygomaticeauricular 22
zygomaticeauricularis 20
zygomaticefacial 15
zygomaticefrontal 20
zygomaticeomaxillary 10
zygomaticeorbital 21
zygomaticesphenoid 23
zygomaticeotemporal 14
```

图 2: reduce 输出部分结果

1.4 实验总结

通过本次实验，我体会了一次搭建简易的 map_reduce 数据处理过程，对 map_reduce 的整个流程有了更深的印象，同时也完成了用 Python 语言进行大数据分析的第一步。

2 实验二 PageRank 算法及其实现

2.1 实验目的

1. 学习 pagerank 算法并熟悉其推导过程；
2. 实现 pagerank 算法, 理解阻尼系数的作用；
3. 将 pagerank 算法运用于实际, 并对结果进行分析。

2.2 实验内容

提供的数据集包含邮件内容 (emails.csv), 人名与 id 映射 (persons.csv), 别名信息 (aliases.csv), emails 文件中只考虑 MetadataTo 和 MetadataFrom 两列, 分别表示收件人和寄件人姓名, 但这些姓名包含许多别名, 思考如何对邮件中人名进行统一并映射到唯一 id.

完成这些后, 即可由寄件人和收件人为节点构造有向图, 不考虑重复边, 编写 pagerank 算法的代码, 根据每个节点的入度计算其 pagerank 值, 迭代直到误差小于 10^{-8} .

实验进阶版考虑加入 teleport, 用以对概率转移矩阵进行修正, 解决 dead ends 和 spider trap 的问题。

输出人名 id 及其对应的 pagerank 值。

2.3 实验过程

2.3.1 编程思路

本次实验最重要的莫过于对 pagerank 算法的理解, 首先 pagerank 是由 Google 提出的计算互联网网页重要度的算法, 在 pagerank 高时搜索相关内容时会出现在较前位置, 基础的 pagerank 公式可以用如下表示:

$$PR(a)_{i+1} = \sum_{i=0}^n \frac{PR(T_i)_i}{L(T_i)}$$

$PR(T_i)_i$ 表示的是其他节点的 (指向 a 结点) 的 PR 值, $L(T_i)$ 表示其他节点的出链数。将求 pagerank 的过程进行矩阵化表达就是使用转移概率矩阵, 又称为马尔可夫矩阵, 通过转移矩阵可以快速计算下一次的 pagerank, 例如对于下图: 那么会有转移矩阵

$$\begin{bmatrix} 0 & 0 & 1/2 & 1 \\ 1/2 & 0 & 0 & 0 \\ 1/2 & 1 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \end{bmatrix}$$

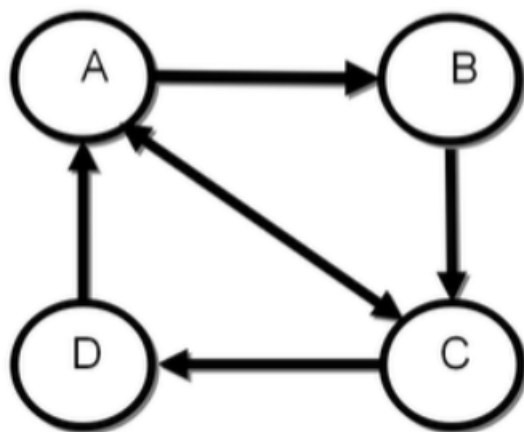


图 3: 示例图

使用均分的初始化方法，得到初始化的 PR 值序列为：

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

那么此时直接由：

$$M * PR = \begin{bmatrix} 3/8 \\ 1/8 \\ 3/8 \\ 1/8 \end{bmatrix}$$

此时就得到了新的 PR；再经过多次迭代使得结果与上一次结果的距离小于误差要求时，就作为最终结果的 rankpage 值。

在了解了基本的算法思路后，我们进一步地去解决 Dead Ends 问题，Dead Ends 问题可以抽象化为这样一种叙述，如果某结点没有任何出链，它会导致网站权重在迭代的过程中变为 0；对于 Dead Ends 问题，使用 Teleport 来解决，我们对转移矩阵进行修正：

$$M + a^T \left(\frac{e}{n} \right)$$

- $a = [a_0, a_1, \dots, a_n]$, 对于 M 中全为 0 的列 i , 有 $a_i = 1$
- e 为由 1 填满的列矩阵
- n 为矩阵 M 行列数

由此修正， M 中全为 0 的列可以由 $\frac{1}{n}$ 来填充，从而解决了没有出链的结点的问题。

然后解决 Spider Traps 问题，Spider Traps 是某结点与其他结点之间没有 out links，导致网站的权重会朝向一个结点进行偏移。

为了解决该问题，进一步对转移矩阵进行修正，Spider Traps 的结果就是某网站不会向打开其他网站这个行为进行跳转，那么就分配一个随机跳转到其他网页的概率即可，修正结果如下：

$$M = \beta M + (1 - \beta) \frac{ee^T}{n}$$

- β 表示跟随出链打开网页的概率；
- $1-\beta$ 就表示随机跳到其他网页的概率；
- ee^T 表示大小为 $n * n$ 的单位矩阵；

综上，可以得到最终的修正公式：

$$PR(a) = [\beta(M + a^T(\frac{e}{n})) + (1 - \beta)\frac{ee^T}{n}] * r$$

修正的关键过程代码如下：

```
1 # Python code
2 def update_mat(mat):
3     col_sum = np.sum(mat, axis=0)
4     n, beta = len(mat), 0.85
5     ind = [i for i in range(len(col_sum)) if col_sum[i] == 0]
6     for i in range(len(mat)):
7         for j in ind:
8             mat[i][j] = 1 / n
9     mat = np.multiply(mat, beta)
10    mat += np.multiply(np.full(mat.shape, 1 / n), 1 - beta)
11    return mat
```

在使用修正公式之前，首先将发送-接收邮件信息转化为转移矩阵，注意到 excel 中的序号并不是连续且不空缺，首先将发送接收方的所有序号向 $0, 1, \dots$ 进行映射转移，因为矩阵的行列 i, j 表示的是结点 i 和 j 之间的连接关系，构建转移矩阵后进行修正，然后对初始 rankpage 的 $n * 1$ 矩阵进行迭代更新，主要代码如下：

```
1 # 计算两个n*1列向量的距离
2 def cal_distance(r1, r2):
3     return np.sqrt(np.sum(np.square(r1 - r2)))
4 new_R = np.dot(M, R)
5 epsilon = math.pow(10, -8) # 误差规定为1e-8
6 while cal_distance(new_R, R) >= epsilon:
7     R = new_R
8     new_R = np.dot(M, R)
```

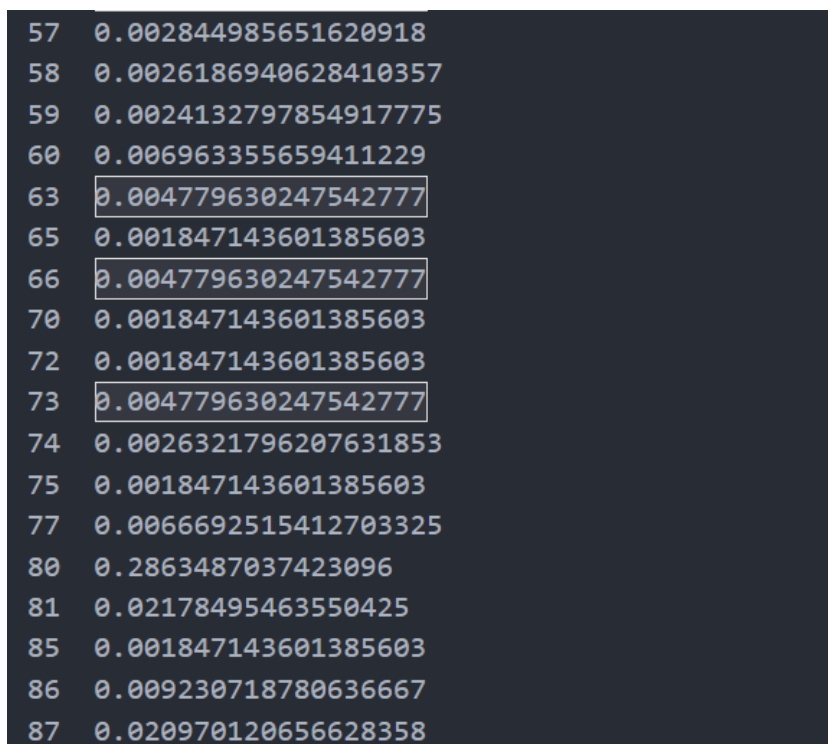
将最终结果写入 ans 文件中。

2.3.2 遇到的问题及解决方式

- 首先遇到的问题就是对修正公式已经遗忘并且对于原理也不够熟悉，这些通过知乎和csdn上的教程进行了查缺补漏；
- 一开始没有正确地使用转移矩阵，应该是使用初始连接矩阵进行标准化后再进行转置后的结果来进行运算；
- 对 teleport β 的合理性和有效性有一定的怀疑，因为使用 β 进行修正之后不一定满足转移矩阵列之和为 1 的性质，所以我认为仅仅使用 β 并不能解决 Dead End 的问题，事实上这应该正是阻尼系数的缺点之一。

2.3.3 实验测试与结果分析

以序号-pagerank 值的形式输出结果到文件 ans 中，部分结果如下图所示：



```
57 0.002844985651620918
58 0.0026186940628410357
59 0.0024132797854917775
60 0.006963355659411229
63 0.004779630247542777
65 0.001847143601385603
66 0.004779630247542777
70 0.001847143601385603
72 0.001847143601385603
73 0.004779630247542777
74 0.0026321796207631853
75 0.001847143601385603
77 0.0066692515412703325
80 0.2863487037423096
81 0.02178495463550425
85 0.001847143601385603
86 0.009230718780636667
87 0.020970120656628358
```

图 4: 部分运行结果

2.4 实验总结

在本次实验中我复习了 pagerank 算法的大致流程，了解了 pagerank 算法在网页访问这样一个我们日常生活中十分频繁的行为背后运行的机理，同时也在实验手册的指示下，在博客网站的查阅和理解下成功完成了实验。

3 实验三 关系挖掘实验

3.1 实验内容

1. 实验内容

编程实现 Apriori 算法，要求使用给定的数据文件进行实验，获得频繁项集以及关联规则。

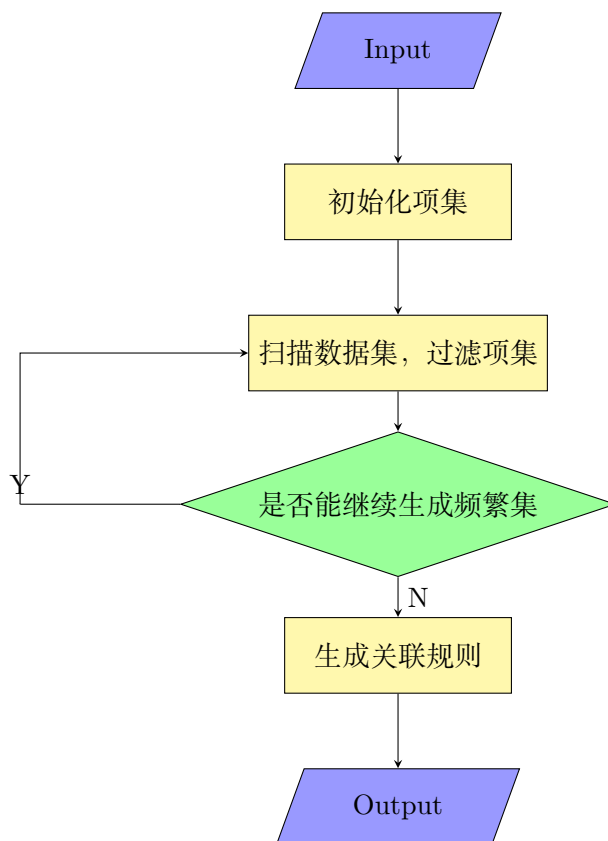
2. 实验要求

以 Groceries.csv 作为输入文件，输出 1 3 阶频繁项集与关联规则，各个频繁项的支持度，各个规则的置信度，各阶频繁项集的数量以及关联规则的总数，固定参数以方便检查，频繁项集的最小支持度为 0.005，关联规则的最小置信度为 0.5。

3.2 实验过程

3.2.1 编程思路

实验的整体流程如下：



针对不同功能函数进行说明：

- createInit

初始化时所有物品单个可以作为项集列表中的一员；

- scanDataset

扫描数据集过程中,对于每一个数据集中的项集,将当前的项集中每一个商品组合与其进行对比,若为子集,则该组合的出现次数增加 1,最后将所有项集出现次数除以数据集的规格,即为支持度大小,对于支持度大于设定阈值的项集,保存之,并同时在项集到支持度的映射中保存其对应的支持度数据。

- generateNewSet

由旧的频繁集生成物品数量 +1 的新频繁集,这里为了优化程序的效率,不采用依次取两个集合取并集后检查是否符合条件的方式,这样不仅要检查长度是否合适,在程序执行的过程中会产生许多重复的项;取而代之的是进行一个二重循环,首先比较两个集合的前 $Len - 2$ 个单词是否相同,若相同再取两者的并集,这样能够保证只更新一个单词,达到长度 +1 的目标同时不重复,例如假设将 $\{0,1\},\{1,2\},\{0,2\}$ 来生成三元素项集,如果是简单地进行排列组合,那么会产生 3 次 $\{0,1,2\}$,但是采取上述优化方式就只会对 $\{0,1\},\{0,2\}$ 进行合并,得到一个 $\{0,1,2\}$.

- apriori

apriori 主程序,完成初始化项集,扫描和生成新项集的循环以及 L 和 support_data 的更新过程。该重要步骤如下:

```
1 # Python code
2 def apriori(data, min_support):
3     init_set = createInit(data)
4     D = list(map(set, data))
5     eligible_set, support_data = scanDataset(data, init_set, min_support)
6     L = [eligible_set]
7     k = 2
8     while len(L[k - 2]) > 0:
9         nxt = generateNewSet(L[k - 2], k)
10        lk, supportData = scanDataset(data, nxt, min_support)
11        support_data.update(supportData)
12        L.append(lk)
13        k += 1
14    return L, support_data
```

- generaterules

对于 2 3 阶的频繁项集,通过拆分的方法,每次将其中一个单词作为关联关系的右端,然后计算可信度,对于可信度超过阈值的集合对,将关联规则左端,右端以及可信度作为三元组进行保存,方便最后打印结果到文件。

生成关联规则的代码如下:

```

1 # Python code
2 def generaterules(L, support_data, min_conf=0.5):
3     total_rules = []
4     for i in range(1, 3):
5         for freq_set in L[i]:
6             for item in freq_set:
7                 right = frozenset([item])
8                 confidence = support_data[freq_set] / support_data[freq_set -
9                                     right]
10                if confidence >= min_conf:
11                    total_rules.append((freq_set - right, right, confidence))
12     return total_rules

```

3.2.2 遇到的问题及解决方式

本次实验前期工作一切顺利，主要问题在于生成规则处花了过多时间，因为一开始并没有注意到只需要针对 1 3 阶频繁项集进行操作，所以在思考生成关联规则的非暴力拆分方法，我的想法是从一个个的项集进行合并，在满足独一无二的生成频繁项集时，尝试通过分配子集给关联规则右端的方式来进行递归式的搜索，将存储答案的列表作为参数进行传递，在生成新规则和依靠可信度检验的同时对其进行更新，但是最后没有实现成功，如果是按照实验要求的话只需要对于不同阶数的情况，将频繁项集中的单词逐个地换至规则右端进行可信度的计算然后更新结果即可。

其实对本实验还有一个疑问，就是不明白为什么关联规则的右端只会是单个元素组成的集合。

3.2.3 实验测试与结果分析

经过与参考答案的对比，顺利得到如下结果：

Value 1	Value 2	Value 3	Value4
Total single	Total double	Total triple	Sum of rules
120	605	264	99

表 1: Result Table

与所给参考结果完全相同，支持度和可信度结果也基本一致。

3.3 实验总结

在本次实验中我掌握了关系型数据挖掘的方法 Apriori, 在数据集上成功完成了物品的关联规则的挖掘，在自己动手实现的过程中也体会到了 Apriori 算法的局限性和缺点：

- 在每一步产生候选项目集时循环产生的组合过多，没有排除不应该参与组合的元素；

- 每次计算项集的支持度时，都对数据库 D 中的全部记录进行了一遍扫描比较，如果是一个大型的数据库的话，这种扫描比较会大大增加计算机系统的 I/O 开销。而这种代价是随着数据库的记录的增加呈现出几何级数的增加。

我认为可以将每个项集通过相应的 hash 函数映射到 hash 表中的不同的桶中，可以通过将桶中的项集技术跟最小支持计数相比较先淘汰一部分项集，在逐层搜索循环过程的第 k 步中，根据 k-1 步生成的 k-1 维频繁项目集来产生 k 维候选项目集，由于在产生 k-1 维频繁项目集时，我们可以实现对该集中出现元素的个数进行计数处理，因此对某元素而言，若它的计数个数不到 k-1 的话，可以事先删除该元素，从而排除由该元素将引起的大规格所有组合。

此外还有 Jiawei Han 等人提出的 FP-growth 算法可以用于高效发现频繁项集，可以在后续过程中进行学习和应用。

4 实验四 kmeans 算法及其实验

4.1 实验目的

1. 加深对聚类算法的理解, 进一步认识聚类算法的实现;
2. 分析 kmeans 流程, 探究聚类算法原理;
3. 掌握 kmeans 算法核心要点;
4. 将 kmeans 算法运用于实际, 并掌握其度量好坏方式。

4.2 实验内容

提供葡萄酒识别数据集 (WineData.csv), 数据集已经被归一化 (normalizedwinedata.csv)。同学可以思考数据集为什么被归一化, 如果没有被归一化, 实验结果是怎么样的, 以及为什么这样。

同时葡萄酒数据集中已经按照类别给出了 1、2、3 种葡萄酒数据, 在 csv 文件中的第一列标注了出来, 大家可以将聚类好的数据与标的数据做对比。

编写 kmeans 算法, 算法的输入是葡萄酒数据集, 葡萄酒数据集一共 13 维数据, 代表着葡萄酒的 13 维特征, 请在欧式距离下对葡萄酒的所有数据进行聚类, 聚类的数量 K 值为 3。

在本次实验中, 最终评价 kmean 算法的精准度有两种, 第一是葡萄酒数据集已经给出的三个聚类, 和自己运行的三个聚类做准确度判断。第二个是计算所有数据点到各自质心距离的平方和。请各位同学在实验中计算出这两个值。

实验进阶部分: 在聚类之后, 任选两个维度, 以三种不同的颜色对自己聚类的结果进行标注, 最终以二维平面中点图的形式来展示三个质心和所有的样本点。

4.3 实验过程

4.3.1 编程思路

首先对于实验提出的问题, 我认为坐标数据的归一化是非常必要的步骤, 因为 K-means 的本质是基于欧式距离的数据划分算法, 均值和方差大的维度将对数据的聚类产生决定性影响, 这些维度的数据将主导着所有维度所共同决定的结果。所以未做归一化处理和统一单位的数据是无法直接参与运算和比较的。常见的数据预处理方式有: 数据归一化, 数据标准化。

★ kmeans++ 选择初始聚类中心

在编程思路, 首先对于 kmeans 算法做一个改进, 称为 kmeans++ 算法, 主要的改变在于选取初始聚类中心这一步上, 初始聚类中心的选择非常重要, 它不仅影响着算法迭代结束的次, 也会直接对结果产生影响, 如果只是随机选择所有初始聚类中心, 很可能使得最终的聚类界限不够明显, 聚类准确率也较低, kmeans++ 的思想就是初始化过程中使得聚类中心间隔远, 遵循以下规则:

- 首先随机选择一个初始聚类中心，放入聚类中心集合 C 中；
- 对于所有数据点，计算到集合 C 中所有聚类中心的距离平方，选取最小值作为距离属性 D_i ，那么每一个点被选择为下一个聚类中心的可能就是

$$p_i = \frac{D_i}{\sum_{j=1}^n D_j}$$

按照这个概率分布随机选择下一个聚类中心，选择完毕后初始聚类中心总数增加 1

- 比较是否达到规定初始聚类上限，若是，则退出，否则返回第二步继续选择。

kmeans++ 选择初始聚类中心的代码如下：

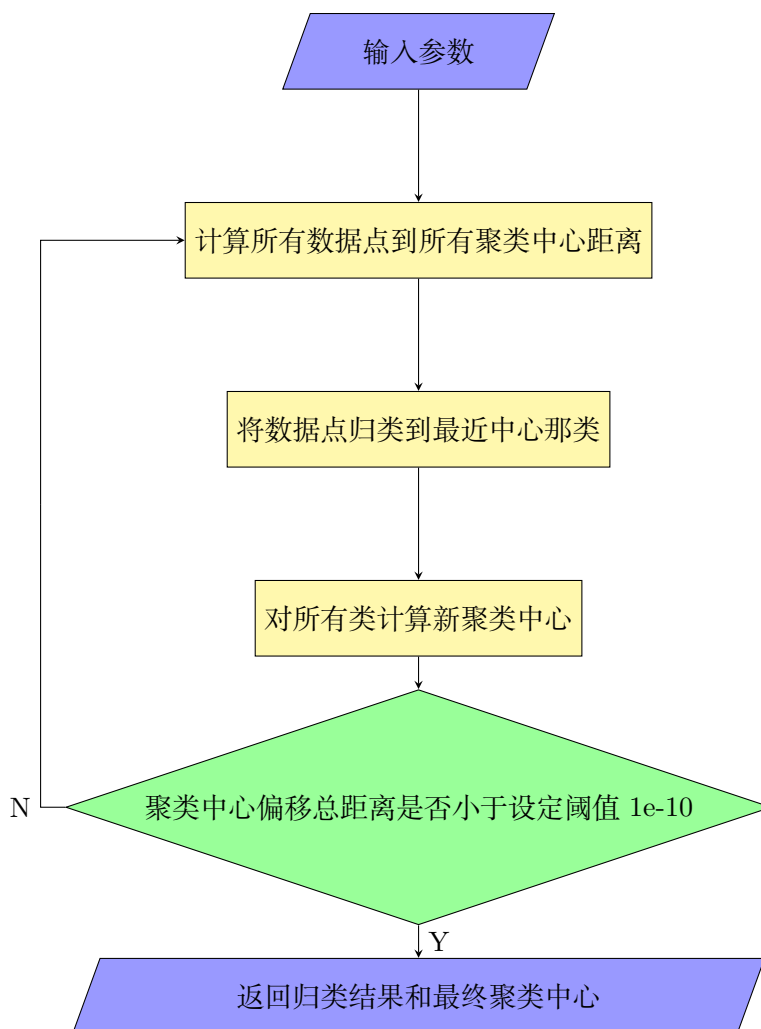
```

1  # Python code
2  # x和k分别是所有数据点的坐标，初始聚类中心数
3  def initialCentroid(x, k):
4      c_id = [np.random.choice(np.arange(len(x)))] # 第一个簇中心序号
5      k -= 1
6      while k > 0:
7          dis = np.zeros(shape=(len(c_id), len(x)))
8          for i in range(len(c_id)):
9              dis[i] = np.sum(np.square(x - x[c_id[i]]), axis=1)
10         min_dis = np.min(dis, axis=0)
11         min_dis = min_dis / np.sum(min_dis)
12         ind = np.random.choice(np.arange(len(x)), p=min_dis.ravel())
13         k -= 1
14         c_id.append(ind)
15     return c_id

```

★kmeans 主体部分

kmeans 函数实现流程如下：



值得注意的是在计算距离时可以利用 Numpy 矩阵的特性进行计算速度的优化,同时本程序采用的迭代更新停止的条件是没有设置次数上限,以每一个中心点偏移距离之和是否小于规定大小作为停止条件,即:

$$\sum_{i=1}^k \Delta D_i < \epsilon$$

★ kmeans 可视化

使用 matplotlib.pyplot 进行 kmeans 结果的可视化,本例选取 13 个坐标维度中的第一维和第二维作为投射到二维坐标轴的横纵轴,用不同颜色标注三个簇并特别显示聚类中心。

4.3.2 遇到的问题及解决方式

本次实验遇到的问题较少,发生的一个小问题是一开始发现聚类准确率很低,然后发现是标签没有对上,因为没有设定聚类中心的顺序,是 kmeans 算法过程中自行决定的,最后得到结果后按照 1,2,3 的区域进行 belong 标签的调整,得到正确的结果。

4.3.3 实验测试与结果分析

实验得到的最终准确率为 95.50561797752809%, 另一个指标是计算所有数据点到其各自聚类中心距离平方和, 其值约为 48.99938177317381;

图形可视化的结果如下:

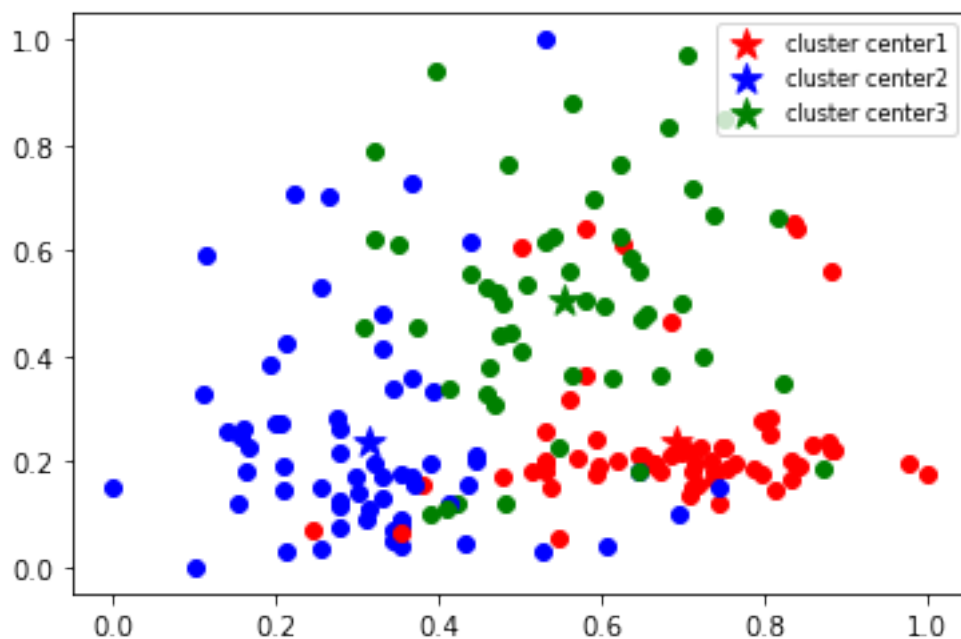


图 5: k-means 聚类结果二维显示

4.4 实验总结

本次实验完成了 kmeans 算法在葡萄酒识别聚类问题中的应用, 并根据 kmeans++ 方法对初始聚类中心选择进行了策略上的优化, 同时使用 pyplot 完成了结果可视化的实现, 根据结果图来看聚类结果还行。

5 实验五 推荐系统

5.1 实验目的

1. 了解推荐系统的多种推荐算法并理解其原理。
2. 实现 User-User 的协同过滤算法并对用户进行推荐。
3. 实现基于内容的推荐算法并对用户进行推荐。
4. 对两个算法进行电影预测评分对比。
5. 在学有余力的情况下，加入 minhash 算法对效用矩阵进行降维处理。

5.2 实验内容

给定 MovieLens 数据集，包含电影评分，电影标签等文件，其中电影评分文件分为训练集 train_set 和测试集 test_set 两部分；

- 基础版必做一：基于用户的协同过滤推荐算法

对训练集中的评分数据构造用户-电影效用矩阵，使用 pearson 相似度计算方法计算用户之间的相似度，也即相似度矩阵。对单个用户进行推荐时，找到与其最相似的 k 个用户，用这 k 个用户的评分情况对当前用户的所有未评分电影进行评分预测，选取评分最高的 n 个电影进行推荐。

在测试集中包含 100 条用户-电影评分记录，用于计算推荐算法中预测评分的准确性，对测试集中的每个用户-电影需要计算其预测评分，再和真实评分进行对比，误差计算使用 SSE 误差平方和。

选做部分提示：此算法的进阶版采用 minhash 算法对效用矩阵进行降维处理，从而得到相似度矩阵，注意 minhash 采用 jaccard 方法计算相似度，需要对效用矩阵进行 01 处理，也即将 0.5-2.5 的评分置为 0，3.0-5.0 的评分置为 1。

- 基础版必做二：基于内容的推荐算法

将数据集 movies.csv 中的电影类别作为特征值，计算这些特征值的 tf-idf 值，得到关于电影与特征值的 n（电影个数）*m（特征值个数）的 tf-idf 特征矩阵。根据得到的 tf-idf 特征矩阵，用余弦相似度的计算方法，得到电影之间的相似度矩阵。

对某个用户-电影进行预测评分时，获取当前用户的已经完成的所有电影的打分，通过电影相似度矩阵获得已打分电影与当前预测电影的相似度，按照下列方式进行打分计算：

$$score = \frac{\sum_{i=1}^n score'(i) * sim(i)}{\sum_{i=1}^n sim(i)}$$

选取相似度大于零的值进行计算，如果已打分电影与当前预测用户-电影相似度大于零，加入计算集合，否则丢弃。（相似度为负数的，强制设置为 0，表示无相关）假设计算集合中一共有 n 个电影，score 为我们预测的计算结果，score' (i) 为计算集合中第 i 个电影的分数，sim(i) 为第 i 个电影与当前用户-电影的相似度。如果 n 为零，则 score 为该用户所有已打分电影的平均值。

选取相似度大于零的值进行计算，如果已打分电影与当前预测用户-电影相似度大于零，加入计算集合，否则丢弃。（相似度为负数的，强制设置为 0，表示无相关）假设计算集合中一共有 n 个电影， $score$ 为我们预测的计算结果， $score'(i)$ 为计算集合中第 i 个电影的分数， $sim(i)$ 为第 i 个电影与当前用户-电影的相似度。如果 n 为零，则 $score$ 为该用户所有已打分电影的平均值。

要求能够对指定的 `userID` 用户进行电影推荐，推荐电影为预测评分排名前 k 的电影。`userID` 与 k 值可以根据需求做更改。

推荐算法准确值的判断：对给出的测试集中对应的用户-电影进行预测评分，输出每一条预测评分，并与真实评分进行对比，误差计算使用 SSE 误差平方和。

选做部分提示：进阶版采用 minhash 算法对特征矩阵进行降维处理，从而得到相似度矩阵，注意 minhash 采用 jaccard 方法计算相似度，特征矩阵应为 01 矩阵。因此进阶版的特征矩阵选取采用方式为，如果该电影存在某特征值，则特征值为 1，不存在则为 0，从而得到 01 特征矩阵。

• 选做（进阶）部分

本次大作业的进阶部分是在基础版本完成的基础上大家可以尝试做的部分。进阶部分的主要内容是使用迷你哈希（MinHash）算法对协同过滤算法和基于内容推荐算法的相似度计算进行降维。同学可以把迷你哈希的模块作为一种近似度的计算方式。

协同过滤算法和基于内容推荐算法都会涉及到相似度的计算，迷你哈希算法在牺牲一定准确度的情况下对相似度进行计算，其能够有效的降低维数，尤其是对大规模稀疏 01 矩阵。同学们可以使用哈希函数或者随机数映射来计算哈希签名。哈希签名可以计算物品之间的相似度。

最终降维后的维数等于我们定义映射函数的数量，我们设置的映射函数越少，整体计算量就越少，但是准确率就越低。大家可以分析不同映射函数数量下，最终结果的准确率有什么差别。

对基于用户的协同过滤推荐算法和基于内容的推荐算法进行推荐效果对比和分析，选做的完成后再进行一次对比分析。

5.3 实验过程

5.3.1 编程思路

编程思路分为基础部分和选做部分两个大部分，二者又分别分为基于用户的协同过滤方法以及基于内容的推荐算法；

★ 基于用户的协同过滤推荐算法（基础版）

采用 *pearson* 相似度对用户相似度进行评价，*pearson* 相似度的计算方法如下：

$$\rho_{X,Y} = \frac{\sum XY - \frac{\sum X \cdot \sum Y}{N}}{(\sum X^2 - \frac{(\sum X)^2}{N}) * (\sum Y^2 - \frac{(\sum Y)^2}{N})}$$

利用该公式计算不同用户间的 *pearson* 相似度矩阵。

对于基于用户的推荐方法，编写类 `recommendByUsers`，其实现的主要功能如下所示：



其中`recommendSingle`中完成对特定的单个用户的评分预测以及电影推荐，着重描述一下该函数的完成逻辑；

首先需要先选出与受推荐用户的 pearson 相似度最大的 k 个用户 ID，对于每个 id，检查相似度是否 ≤ 0 ，因为相似度没达到 0 也可能是 pearson 相似度最大的前 k 个用户，若合格则累计 $score * similarity$ 以及 $similarity$ ，最后得到的评分中删去已评分电影，在新增电影评分中选出 score 前 n 大的电影进行推荐，同时将预测后的评分也作为返回值返回，以用于实验任务要求中对于测试集文件特定用户的特定电影的预测准确度。

对于测试集的预测，由于数据中有对同一个用户的不同电影进行推荐，所以可以设置一个字典命名为`cache`用于缓存已经被分数预测过的用户的预测结果，这样对于一个用户只用调用`recommendSingle`方法一次，该方法在 k 较大时的时间开销还是会比较大的，所以要尽量减少调用；

最后对得到的结果与标准结果计算 SSE。

★ 基于内容的推荐算法 (基础版)

基于内容的推荐只考虑了对象本身性质，将对象按照标签形成集合，本实验中预处理过程考虑使用电影的类别标签作为考察的特征对象，计算 tf-idf 特征矩阵，然后得到电影间的相似度矩阵。

TF-IDF 是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。对于一组语料，或者对于本实验而言，每一个电影的种类标签就是其文本语料，

对于一个词语（本实验中是一个种类标签） t 有平滑后的 tf-idf 计算公式为：

$$\begin{aligned} tf-idf(t) &= tf(t, d) * (idf(t, d) + 1) \\ &= tf(t, d) * \left(\log \frac{n_d + 1}{1 + df(t, d)}\right) \end{aligned}$$

其中 $tf(t, d)$ 是 t 在文本 d 中出现次数即词频，在本实验中由于一个标签在一个电影中最多只会出现一次所以一定出现词频一定为 1， n_d 是文本总数，在本实验中就是给出的电影总数， $df(t, d)$ 是共有多少文本出现了 t ；最后得到的 tf-idf 矩阵对于每一行进行 L2 归一化，即：

$$L_i = \frac{L_i}{\sqrt{\sum_{j=1}^n L_j^2}}, i \in [1, n]$$

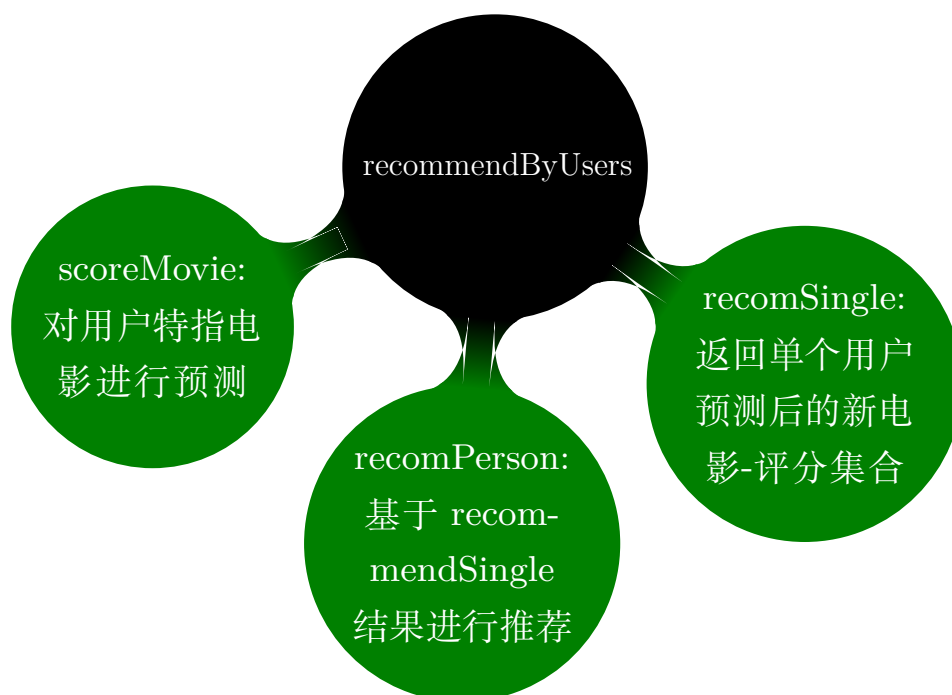
在得到 tf-idf 矩阵后对所有电影求余弦相似度，即：

$$\cos(X, Y) = \frac{X \bullet Y}{|X| \bullet |Y|}$$

利用 numpy 库对矩阵计算的底层加速，可以设计最快的计算方式，python 代码实现如下：

```
1 # Python Code
2 # 计算两个矩阵中行向量对的余弦相似度矩阵
3 def getCosSimilarBetweenTwoMat(self, v1, v2):
4     return np.dot(v1, np.array(v2).T) / (np.linalg.norm(v1, axis=1).reshape(-1, 1) * np.linalg.norm(v2, axis=1))
```

在实现了 tf-idf 相似度矩阵的获取之后，实现基于内容的电影推荐类，命名为 `recommendByContent`，其主要实现逻辑如下：



在计分过程中, 仍然采取根据相似度权重计算得分总和的方式, 对于某个待预测电影, 若枚举电影的相似度 ≤ 0 则不加入评分集合, 如果没有电影能够为其打分, 那么分数设置为已打分电影的得分平均值;

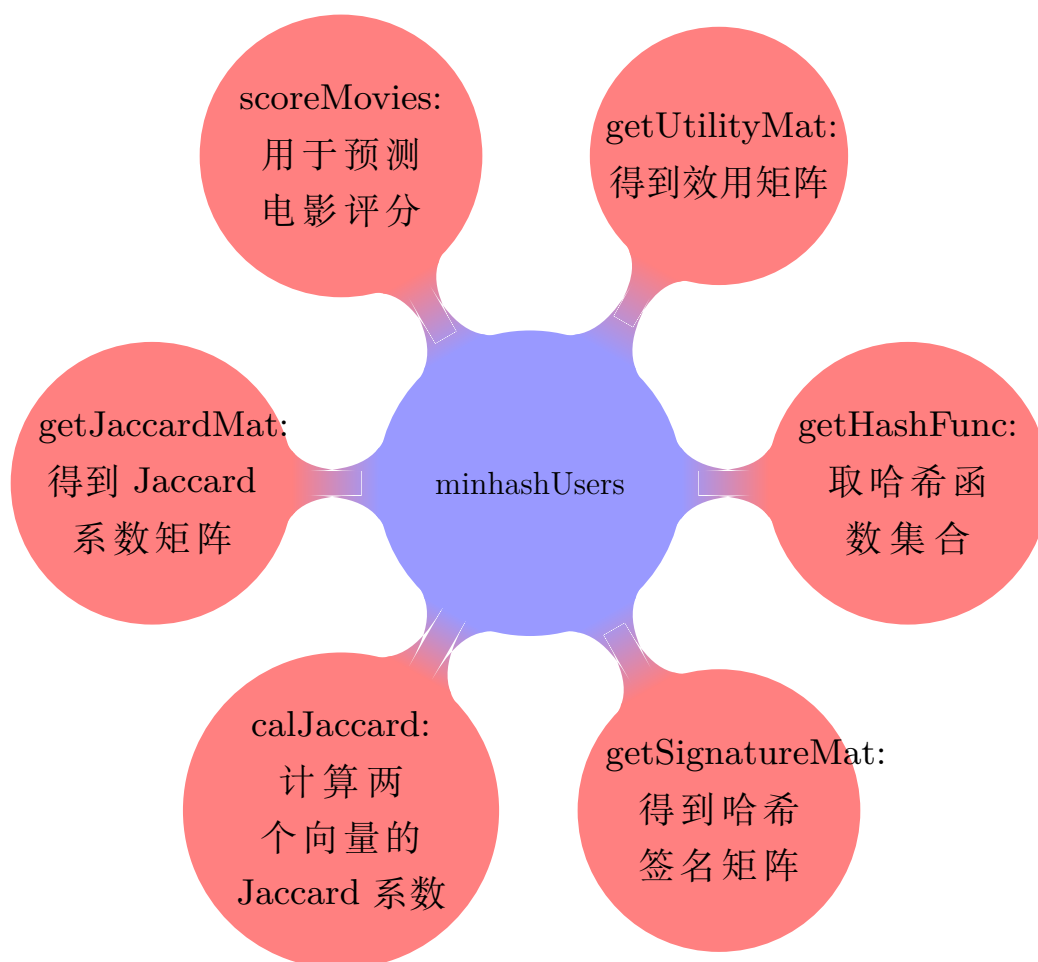
对于实验要求对指定 `userId` 进行电影推荐只需要调用 `recommender.recommendSpecificPerson` 即可。

计算算法准确值, 只需要对于 `test` 文件中每一个特定的用户和电影调用 `scoreSpecificMovie` 即可得到预测得分, 将预测得分集合与标准的 `rating` 计算 `SSE` 即可。

★ 基于用户的协同过滤推荐算法 (进阶版)

在数据挖掘的问题中, 一个常见且基本的问题就是比较两个集合的相似度, 当集合中的元素非常多且需要两两间计算相似度时, 传统的计算方法会有非常大的时间开销, 于是在进阶版中采用 `minhash` 算法来对效用矩阵进行降维。

设计类 `minhashUsers`, 主要功能如下所示:



► `getUtilityMat`

计算效用矩阵时, 根据评分规则, 将 `[0.5,2.5]` 评分划分为 0, 将 `[2.5,5.0]` 评分划分为 1, 返回效用矩阵的转置, 将待计算相似度的属性即用户作为列属性;

► `getHashFunc`

在 miniHash 算法过程中又可以通过一些手段进行优化,有随机数映射,使用哈希函数和局部敏感哈希算法 (LSH) 等等,最基础的 minHash 算法是将特征矩阵的行进行多次随机打乱,然后对于每一列,找到第一个标签 1 出现的位置作为这一次搜索的哈希签名,这种方法的缺点就是在特征矩阵非常大时对其进行打乱非常耗时,本实验中选择通过定义多个哈希函数的方式来进行优化,哈希函数的输入输出长度空间与特征数相等,因此通过两个集合的最小哈希值是否相等的概率来代表两集合的 Jaccard 相似度;

函数设计可以调整哈希函数个数,传入参数为个数 n ,生成哈希函数为:

$$h_1(x) = (x + 1) \bmod m \quad (1)$$

$$h_2(x) = (2 * x + 1) \bmod m \quad (2)$$

$$\vdots$$

$$h_n(x) = (n * x + 1) \bmod m \quad (3)$$

其中 m 表示特征对象长度;

►getSignatureMat

签名矩阵初始化为一个很大的值,对于效用矩阵中的每一行取出标签为 1 的坐标,对于这些位置,比较行对应哈希函数值与签名函数中对应一个单元格的大小,取较小者替换之,得到最小哈希组合作为哈希签名矩阵;

该核心实现方法的代码如下:

```

1  # Python Code
2  def getSignatureMat(self):
3      self.signatureMat = np.full(shape=(self.hashFuncSum, self.usersSum),
4                                   fill_value=1000000) # 初始化为很大的数
5      for i in range(len(self.utilityMat)):
6          oneIndex = np.array(np.where(self.utilityMat[i] == 1)) # 找到1的位置,
7                                   即可替换的位置
8          for j in range(self.hashFuncSum):
9              for k in oneIndex[0]:
10                 if self.hash[j, i] < self.signatureMat[j, k]:
11                     self.signatureMat[j, k] = self.hash[j, i]
12
13  return None

```

►calJaccard

计算两个向量的 Jaccard 相似度,为后面的函数提供方法接口;

►getJaccardMat

用getSignature方法得到的签名矩阵来计算 Jaccard 相似度矩阵,由于对称性可以减小一半计算开销;

►scoreMovies

用于为测试集数据进行电影评分的预测,由于是面向测试的函数,所以不计算所有未评

分电影的分数，取 Jaccard 相似度与待比较用户最接近的 k 个用户，将待预测电影与这些用户的评价库逐一比较使用基础版中的分数计算方法得到预测结果，返回预测结果的 numpy 数组用于计算 SSE。

★ 基于内容的推荐算法 (进阶版)

在本模块中依然使用 minHash 方法对特征矩阵进行降维，在基于内容的推荐算法中特征矩阵比较直白，就是某电影是否具有某个种类标签，若有则为 1，否则为 0；编写类 minhashUsers 实现，其基本结构与基于用户的协同过滤中的结构基本一致，只是挑选的特征不同，获得特征矩阵的方法也不相同；

挑选相异之处进行说明，相同流程不再赘述；

1. 获得特征矩阵时创建种类特征到列序号的映射，电影 ID 到行序号的映射，对于一个电影，拥有某种类时对应 1，否则对应 0，最后返回矩阵的转置，依然以待计算相似度的对象作为列属性；

2. 用于给测试集数据做预测时，对于待预测用户，用其所有已评分电影数据进行特定未评分电影的预测，权重为两个电影的 Jaccard 相似度。

5.3.2 遇到的问题及解决方式

★ 首先是一个编程上的问题，在进行 minHash 算法挑选每一行的 1 时使用 `np.where` 函数，但是在随后的语句中出现

```
The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

的报错，原因是 `np.where` 返回的并不是一维的 numpy 数组，只有增加一维索引才能取出一维数组，否则会因为 numpy 数组特性一次性为 signatureMat 代入 oneIndex 的所有值从而出现“多值对一值”的错误；

★ 在进阶版基于用户的协同过滤算法中，由于任务书中只给出了已评分的电影的 01 化方法并未提到未评价电影，我起初认为就当做 0 来做即可，但又觉得当做 0 会有增加了“0”特征的嫌疑，因为 0 在此情景下应该是代表了 $[0.5, 2.5]$ 的评分区间，后来在学习通上询问老师得到了用 0 处理的肯定答复，同时自己深入思考，在 minHash 过程中，我们是要找 1 出现的最小对应哈希值来作为哈希签名，也就是完全用标签 1 来区分特征，那么事实上未评价的电影本来就不称得上是“特征”，也就不至于对结果产生多大影响，所以按照空缺值就用效用矩阵中的 0 来使用继续编程。

★ 在进阶版基于内容推荐方法中，由于电影集合达到了 9000+ 条，所以 jaccardMat 的规模非常大，计算过程 jupyter 单元格计时显示为 6min，所以为了节约时间，便于下次使用，找到了 joblib 包对矩阵进行非常方便的存储和读取。

5.3.3 实验测试与结果分析

★ 基础版对比分析

对于基于用户的协同过滤推荐方法，由于选择的 k 值，即选取前多少个最相似的用户进行电影评分，会很明显地影响到最终预测结果的 SSE，选取不同 k 值进行预测结果比较，首先实验得到较适合区间约为 $[100,300]$ 得到表格如下：

k	SSE
100	84.60929280797903
110	85.1312935620827
120	85.11390486120229
130	82.95234047089075
140	82.58000471264087
150	81.24304320711192
160	82.74501739669861
170	81.22938048701586
180	79.86828786127539
190	79.2054612321952
200	77.54783577875114
210	76.48473679670569
220	76.19617415123156
230	77.5168343035015
240	78.32879269160995
250	80.578743997901
260	81.11767894128013
270	81.52892577874152
280	81.82100328228033
290	81.78954137299066
300	80.89906199790795

表 2: k 与 SSE 对应结果

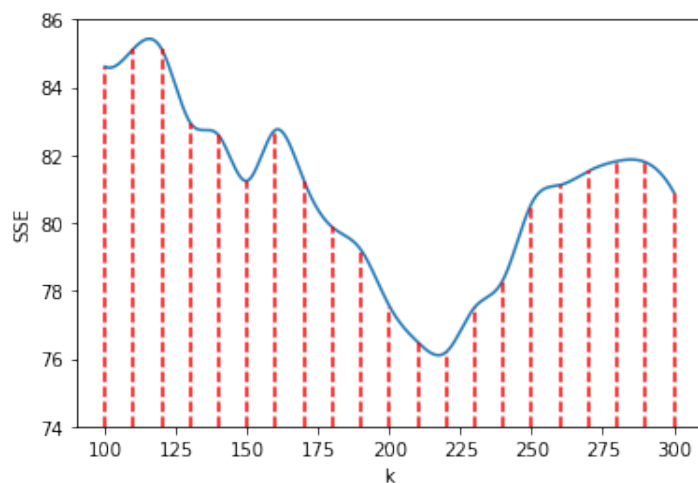


图 6: SSE 随 k 变化趋势图

根据图表可以比较清楚地看到 SSE 随自变量 k 的变化趋势, k 值挑选的太少是不合理的, 因为人数选取过少很可能相当一部分电影因为没有遇到评分过的人而直接变为 0 分, 而且人数过少具备更多的随机性; 而选取人数过多也不行, 因为会从某一个模糊的阈值开始, 预测结果受到不那么相似的用户的评分结果的结果影响而造成较大的偏差, 且最低 SSE 在 210 230 间, 当 k 取 220 时 SSE 为 76.19617415123156。

基于内容推荐的结果表现较基于用户的协同过滤较好, 因此也不能完全说基于用户的协同过滤方法就比基于内容推荐更好, 我认为在本数据集中, 由于电影数量较多, 而且每个用户评分电影数量大多数都在十几二十条, 所以在基于用户相似度推荐时会有很多未评分电影缺少用户样本去提供评分样本, 所以造成许多评分搁置, 带来较大误差。而且我认为对于电影这个评价对象, 拥有电影种类的情况下, 基于内容推荐可能反而是更好的方法, 因为对我自己而言, 我也总是倾向于找某个特别喜欢的电影种类看, 比如悬疑片, 那么就会对这类电影具有较高的兴趣和评价, 所以按照类型标签来捕获关键信息进行推荐也是行之有效的。

★ 进阶版对比分析

从结果上看, 进阶版的基于用户的协同过滤算法表现并不好, 选取 200 个以上最相似用户, SSE 维持在 199 200 间, 且变化非常细微, 首先基于 minHash 进行特征矩阵的降维损失了非常多的关键信息, 而且由于将评分 0.5 2.5 直接置 0 处理, 将这些特征与未评分电影混合化, 进一步掩埋了信息, 我认为这会导致预测评分比实际评分低不少, 最终得到较大的误差 SSE。

而进阶版的基于内容推荐方法则依然表现良好, 在基础版上获得了更低的 SSE, 为 65.98285 75811148, 我认为这是因为事实上是否应用 minHash 在对于电影与其种类的特征对象中, 仅仅只是改变了相似度的计算方法而已, 本质上特征矩阵并没有改变, 拥有某个特征就标识 1, 否则标识 0, 因此并没有损失什么关键信息, 而且还有一个关键点造成两种方法之间预测结果的差异之大, 就是特征数量, 电影种类只有 20 来种, 但是基于用户的协同过滤方法中是需要计算用户的相似度, 电影评分 01 化后实际上每一个电影都是一个特征, 而电影总数达到了 9000+ 个, 这差距是非常悬殊的, 换句话说, 它的特征矩阵要稀疏得多, 所以造成更大的误差也不足为奇了。

★ 最后是建立哈希函数个数对于预测结果 SSE 的影响, 这里直接选取表现最好的进阶版基于内容推荐算法, 选取不同的哈希函数个数 num 进行测试, 得到结果表格如下:

num	SSE
2	63.315543681475006
4	64.693131578845
6	66.55207838378904
8	65.57966841043574
10	65.61872688678098
12	65.68960588579287
14	66.01158800340758
16	66.52134546397016
18	66.63631574550122

表 3: num 与 SSE 对应结果

补充测试只选择一个哈希函数的结果, 是 63.44754044520913, 也非常不错, 这里有一点疑

问就是并没有出现任务书中所说的映射函数越少结果越差的情况，可能存在一定的偶然性，或者还有自己没有注意到的点，但是随着选取函数的不断增加，SSE 会稍稍增加，分析原因应该是取最小哈希值的变化可能性增加了，导致评分结果有更大的偏移。

5.4 实验总结

通过本次实验完成了基于用户的协同过滤推荐算法和基于内容的推荐算法，并在此基础上使用了 minHash 算法对特征矩阵进行降维从而简化了计算过程，并分析了不同算法在测试集上的不同预测结果造成的误差大小差别以及可能的原因，在这次大作业中遇到了不少问题，包括知识点上的，编程上的，但都通过查找网上资料，英文原版书籍文献，询问老师和自己花时间思考和 Debug 完成了这次实验，因此收获还是非常多的，通过这次实验也能够进一步体会到数据挖掘算法的强大，对数据挖掘，数据分析以及推荐系统有了更加深入的认识。