

**课 程 实 验 报 告**

**课程名称： 大数据分析**

**专业班级： 大数据2101班**

**学 号： U202115652**

**姓 名： 李嘉鹏**

**指导教师： 王蔚**

**报告日期： 2023年5月11日**

**计算机科学与技术学院**

**目录**

[实验一 wordCount算法及其实现 1](#_Toc134825893)

[**1.1实验目的** 1](#_Toc134825894)

[**1.2 实验内容** 1](#_Toc134825895)

[**1.3 实验过程** 1](#_Toc134825896)

[1.3.1 编程思路 1](#_Toc134825897)

[1.3.2 函数模块定义 3](#_Toc134825898)

[1.3.3 数据结构与存储形式 4](#_Toc134825899)

[1.3.4 遇到的问题及解决方式 4](#_Toc134825900)

[1.3.5 实验测试与结果分析 4](#_Toc134825901)

[**1.4 实验总结** 8](#_Toc134825902)

[**1.5 参考文献** 8](#_Toc134825903)

[实验二 PageRank算法及其实现 9](#_Toc134825904)

[**2.1实验目的** 9](#_Toc134825905)

[**2.2 实验内容** 9](#_Toc134825906)

[**2.3 实验过程** 9](#_Toc134825907)

[2.3.1 编程思路 9](#_Toc134825908)

[2.3.2 数据结构与存储形式 11](#_Toc134825909)

[2.3.3 遇到的问题及解决方式 11](#_Toc134825910)

[2.3.4 实验测试与结果分析 11](#_Toc134825911)

[**2.4 实验总结** 13](#_Toc134825912)

[**2.5 参考文献** 13](#_Toc134825913)

[实验三 关系挖掘实验 14](#_Toc134825914)

[**3.1 实验内容** 14](#_Toc134825915)

[**3.2 实验过程** 14](#_Toc134825916)

[3.2.1 编程思路 14](#_Toc134825917)

[3.2.2 函数模块定义 19](#_Toc134825918)

[3.2.3 数据结构与存储形式 20](#_Toc134825919)

[3.2.4 遇到的问题及解决方式 21](#_Toc134825920)

[3.2.5 实验测试与结果分析 21](#_Toc134825921)

[**3.3 实验总结** 26](#_Toc134825922)

[**3.4 参考文献** 27](#_Toc134825923)

[实验四 kmeans算法及其实现 28](#_Toc134825924)

[**4.1实验目的** 28](#_Toc134825925)

[**4.2 实验内容** 28](#_Toc134825926)

[**4.3 实验过程** 29](#_Toc134825927)

[4.3.1 编程思路 29](#_Toc134825928)

[4.3.2 函数模块定义 31](#_Toc134825929)

[4.3.3 数据结构与存储形式 32](#_Toc134825930)

[4.3.4 对已归一化数据集的理解 32](#_Toc134825931)

[4.3.5 遇到的问题及解决方式 33](#_Toc134825932)

[4.3.6 实验测试与结果分析 33](#_Toc134825933)

[**4.4 实验总结** 34](#_Toc134825934)

[**4.5 参考文献** 34](#_Toc134825935)

# 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、（可选）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。

学有余力的同学可以在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。

提示：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

整体上，本次实验主要分为四个模块：map、combine、shuffle、reduce。

对于第一个模块map，编写了mapper函数实现了对源文件source01~09的键值对的分解，通过循环方式依次启动九个map进程。我利用time.perf\_counter()函数记录第一个进程开始的时间，并在九个进程结束时再分布记录一次系统时间，两个时间相减就能得到运行到某一个进程时所耗费的总时间。最后，依据字典序将map结果输出。map的核心代码如下。

|  |
| --- |
| **with** write **as** f:  **for** words **in** lines:  **for** word **in** words:  f.write(**"{},{}\n"**.format(word, 1)) *#每个词出现一次为一个键值对* |

Python的多线程功能十分强大，可以高效处理大量任务，因此本次实验中，我全部采用了多线程的方式。创建进程的代码如下。

|  |
| --- |
| map\_threads=[] **for** i **in** range(9):  *#循环9次，依次启动9个mapper进程* t=threading.Thread(target=mapper(**f"source0{**i+1**}"**,**f"map{**i+1**}"**))  map\_threads.append(t)  t.start() start = time.perf\_counter()count=1 *#循环计数器* **for** t **in** map\_threads:  t.join()  print(**f"map\_t{**count**}: %s s"** %(time.perf\_counter() - start)) *# 计算多线程运行时间* count+=1 |

对于第二个模块combine，编写了combiner函数，首先创建一个空字典用以记录源文件中的单词和相应出现的次数，接下来对map得到的文件进行遍历。同样地，我通过循环依次进行九个combine进程，并分别记录消耗的时间。combine的核心代码如下。

|  |
| --- |
| **for** line **in** read:  line = line.strip()  word, count = line.split(**','**, 1)  **try**:  count = int(count)  **except** ValueError:  **continue  if** word **in** count\_dict.keys(): *#已经查找到这个词，count+1* count\_dict[word] = count\_dict[word] + count  **else**: *#第一次找到这个词，count=1* count\_dict[word] = count |

对于第三个模块shuffle，编写了shuffle函数，利用哈希函数对上一步得到的combine结果进行划分。具体而言，我调用了库函数hash(word)得到每一个单词的hash值，由于最终有3个reduce节点，我在shuffle阶段需要将全部单词分配到三个“桶”中。因此我将hash值对3取余，使最终的单词分布趋于平衡（即每个reduce节点的负载近似一致）。shuffle的核心代码如下。

|  |
| --- |
| **for** line **in** file:  line = line.strip()  word, count = line.split(**','**, 1)  **if** (word[0] >= **'a' and** word[0]<=**'z' and** hash(word)%3==0) **or** (word[0] >= **'A' and** word[0]<=**'Z' and** hash(word)%3==0): *#利用哈希函数对word进行shuffle* write1.write(**"{},{}\n"**.format(word, count))  **elif** (word[0] >= **'a' and** word[0]<=**'z' and** hash(word)%3==1) **or** (word[0] >= **'A' and** word[0]<=**'Z' and** hash(word)%3==0):  write2.write(**"{},{}\n"**.format(word, count))  **else**:  write3.write(**"{},{}\n"**.format(word, count))  *#没有直接使用特定首字母mod3的方式（如下），避免了负载不均衡性  # if (word[0] >= 'a' and word[0]<='z' and ord(word[0])%3==0) or (word[0] >= 'A' and word[0]<='Z' and ord(word[0])%3==0)  # elif (word[0] >= 'a' and word[0]<='z' and ord(word[0])%3==1) or (word[0] >= 'A' and word[0]<='Z' and ord(word[0])%3==1)* |

最终，对于模块reduce，本质上和combine函数一致，但仅需要启动三个进程。编写完上述四个模块后，依次运行map、combine、shuffle、reduce，即可得到运行结果文件。

算法的整体流程图如图1-1所示。

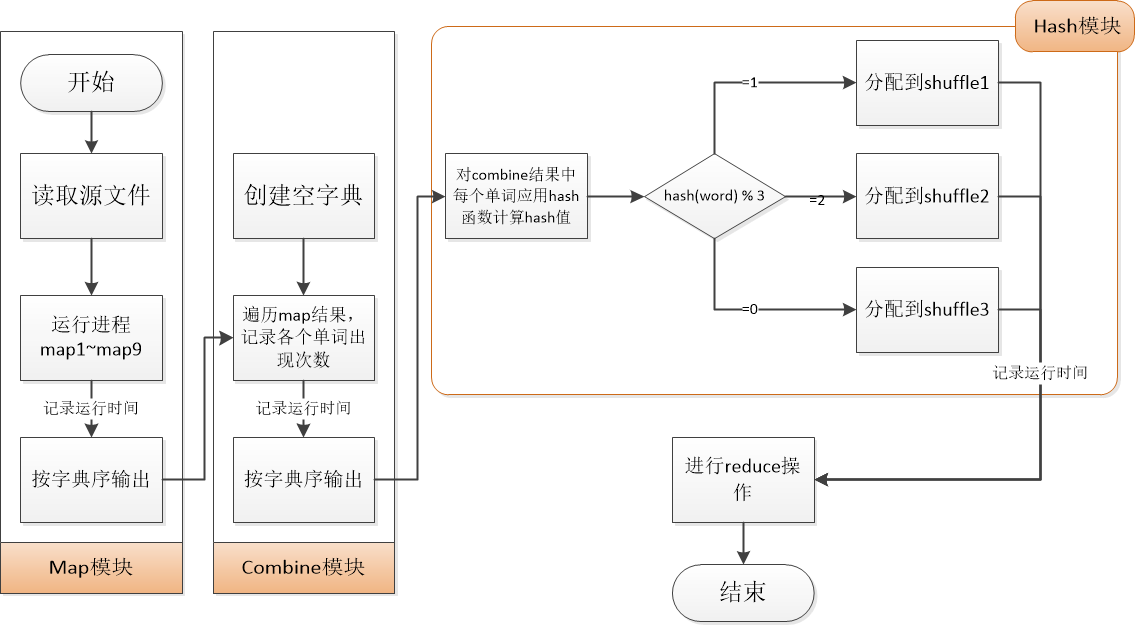


图1-1：算法流程图

### 1.3.2 函数模块定义

**（1）mapper**

①函数参数：readfile, writefile

②函数功能：完成map任务，逐个统计出现的单词。

**（2）combiner**

①函数参数：inputfile, outputfile

②函数功能：完成combine任务，并按照单词的字典序排序输出。

**（3）shuffle**

①函数参数：readfile

②函数功能：完成shuffle任务，依据哈希值将单词分到三个桶里。

**（4）reduce**

①函数参数：readfile, writefile

②函数功能：对shuffle后的结果进行处理。

### 1.3.3 数据结构与存储形式

**（1）map\_threads、combine\_threads、shuffle\_threads**

①类型：列表

②作用：储存map、combine、shuffle三个模块的多线程。

**（2）count\_dict**

①类型：字典

②作用：存放每个单词及其出现的次数。

### 1.3.4 遇到的问题及解决方式

本次实验最大的问题就是在写循环开始多个进程时不太会处理。在网上查阅资料发现，Python的进程也可以被视为数组元素，由此可采用xxx\_threads=[]来创建一个空的进程列表，每次循环中通过下面两条语句在进程列表中添加新的进程即可。

t=threading.Thread(target=mapper(f"source0{i+1}",f"xxx{i+1}"))

xxx\_threads.append(t)

这样避免了复制多句重复语句并修改细节（例如循环次数）的繁琐，在进程数量变化时也能很方便地进行调节。

### 1.3.5 实验测试与结果分析

依次运行map、combine、shuffle、reduce，得到运行结果文件map1~9、combine1~9、shuffle1~3、reduce1~3，如图1-2所示。

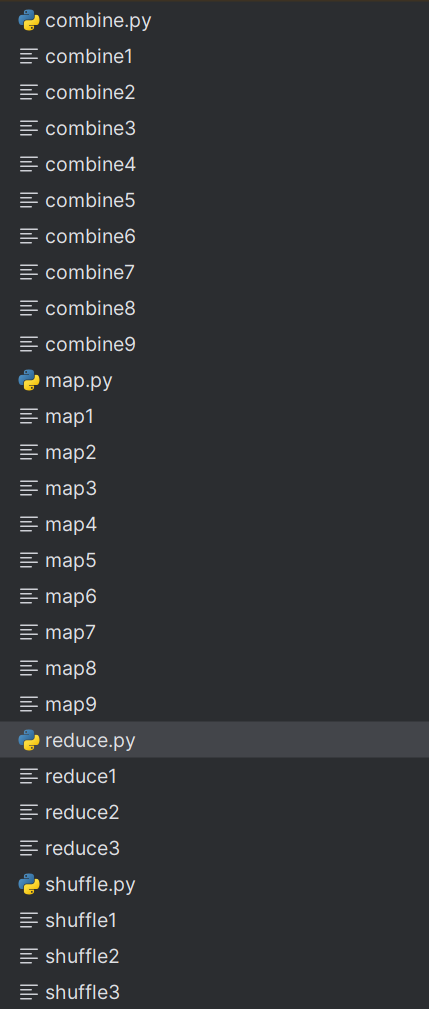
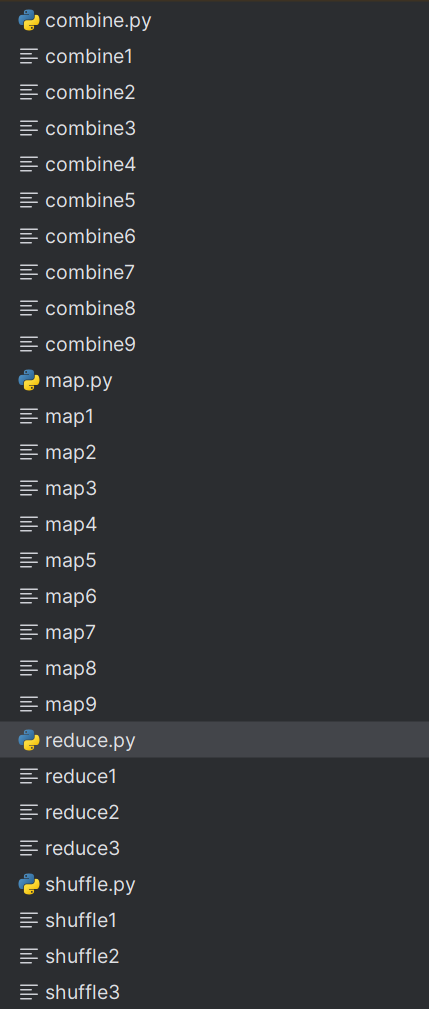
 

图1-2：结果文件列表

对于四个模块map、combine、shuffle、reduce，分别统计了相应的运行时间，如图1-3~图1-6所示。对其分别进行逐差即可得到每个进程的运行时间。

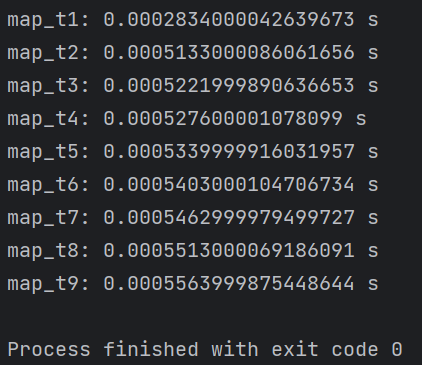
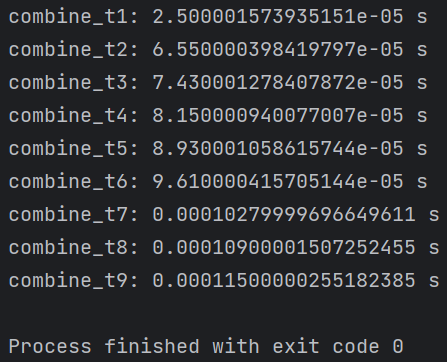
 

图1-3：map模块九个进程花费时间 图1-4：combine模块九个进程花费时间

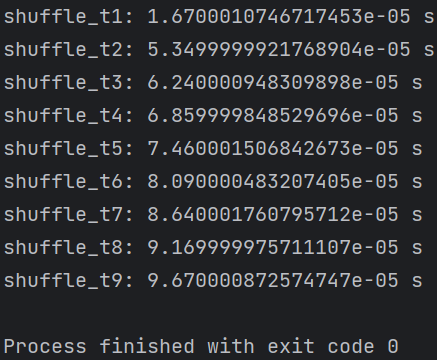


图1-5：shuffle模块九个进程花费时间

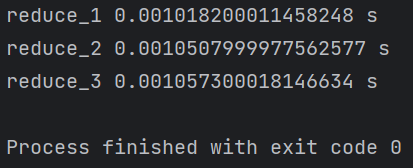


图1-6：reduce模块三个进程花费时间

从中可以看出，combine和shuffle两个模块运行速度很快，不会对算法的整体实现耗时产生影响。

接下来，对结果文件进行分析，在map1~9中，可以看到源文件的文本已经被按照“单词 空格 单词”的格式拆分开，并将每个单词的每一次出现进行了记录，如图1-7所示。

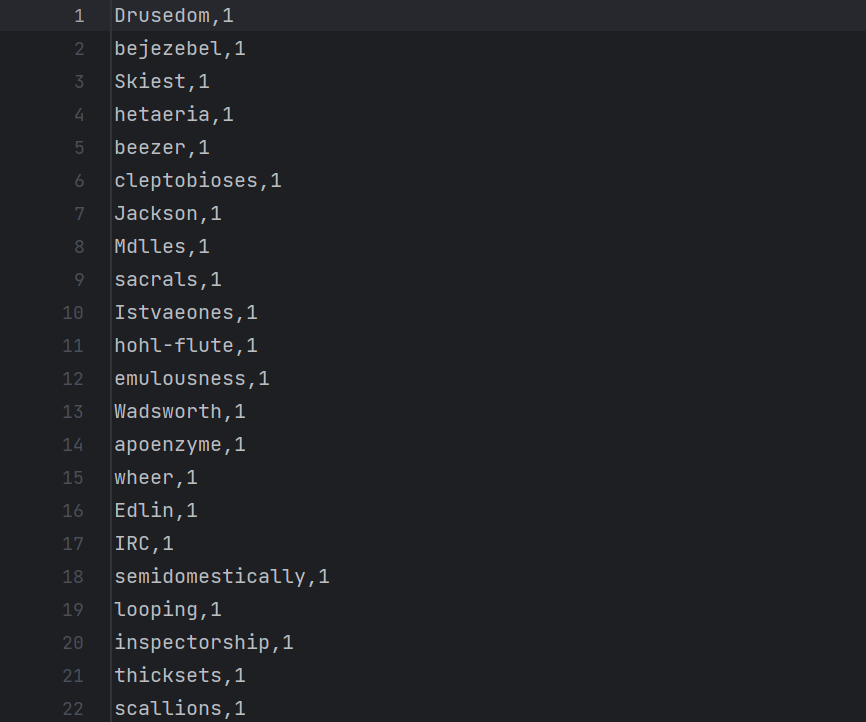
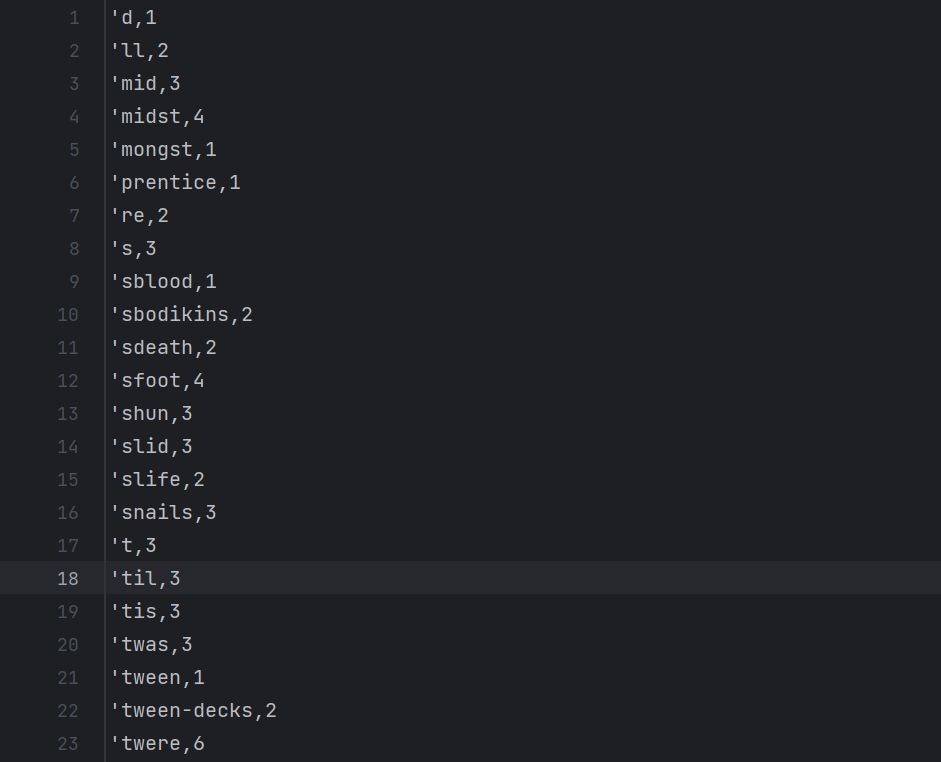
 

图1-7：map结果文件 图1-8：combine结果文件

在combine1~9中，可以看到已经将上述map文件进行了汇总，按照单词-出现次数的格式输出到文件中，并按字母序排序，如上图1-8所示。

在shuffle1~3中，根据前面用hash值模3的思路，可以在三个文件中找到相似的单词（即不是简单地只包含某几个首字母，或是只出现大写或小写单词等），且三个文件的单词数量相近，如图1-9、图1-10所示。可见，这个方法在很大程度上避免了工作负载不均衡的问题。

最后，reduce文件将三个shuffle结果文件进行合并，如图1-11所示。

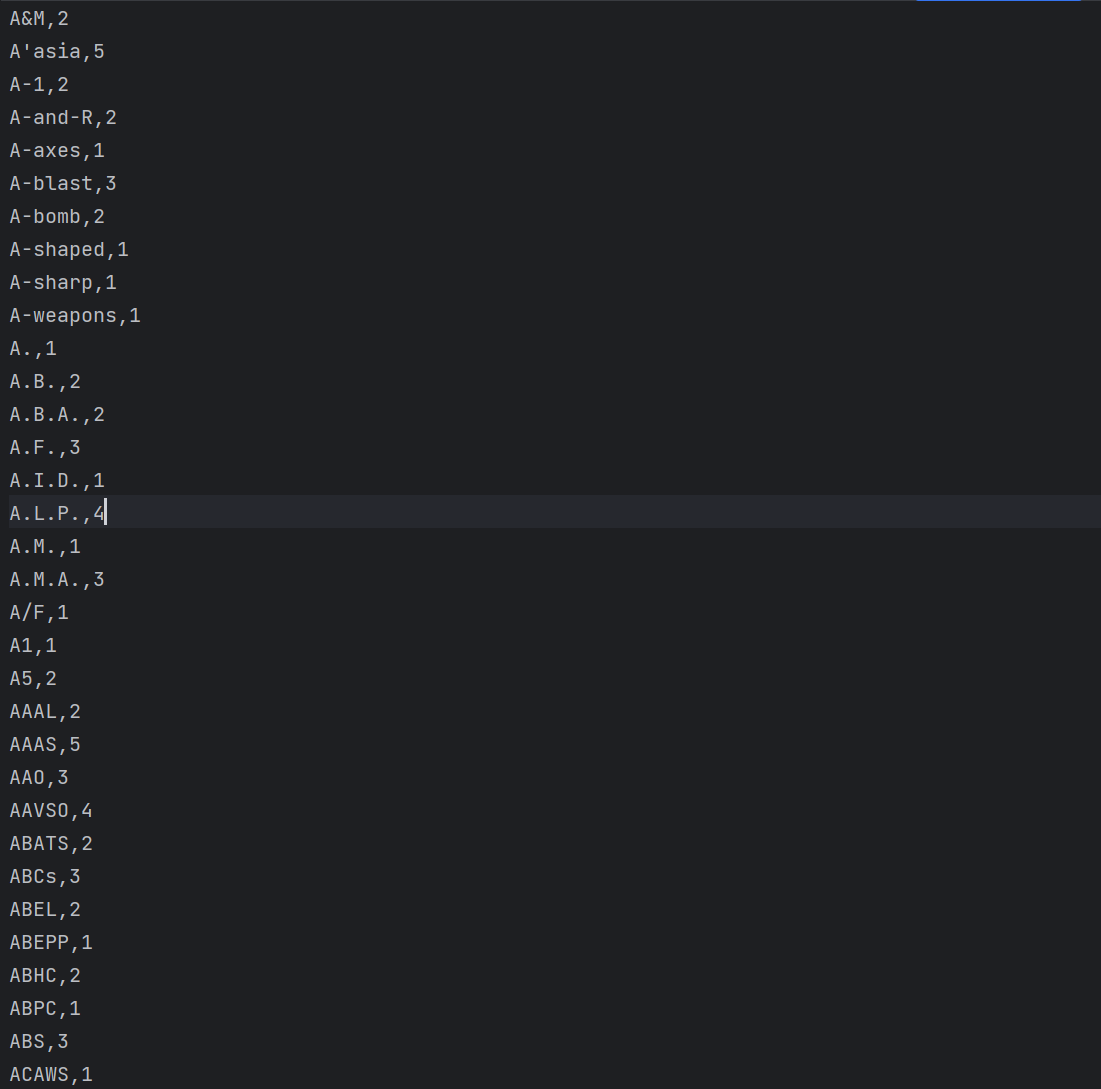


图1-9：shuffle结果文件

图1-10：shuffle1、shuffle2、shuffle3的行数

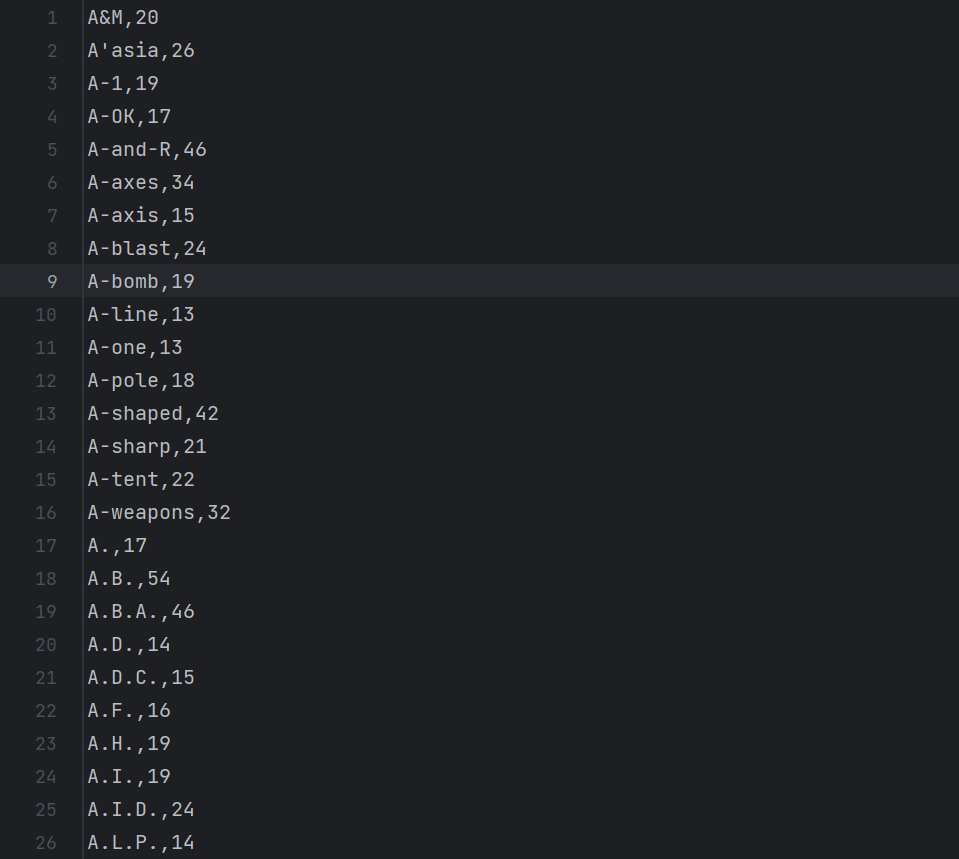


图1-11：reduce结果文件

## **1.4 实验总结**

本次实验是对大数据分析课上MapReduce理论的实现，让我通过具体的数据集深入理解了map、combine、shuffle、reduce这四个模块的实现方法以及它们各自需要注意的细节。同时，之前的很多实验都是基于C++完成的，这次使用Python语言实现让我感受到了Python语言的灵活性以及其读入、写入文件的高效性。

MapReduce是一种面向大规模数据集的并行计算算法。通过对本次实验中九个大文件的操作，我进一步理解了map、combine、reduce的基本思想，并通过尝试得到了一种表现较好的shuffle方法（利用哈希函数hash(word)完成）。

## **1.5 参考文献**

[1] MapReduce详解 <https://blog.csdn.net/woaini886353/article/details/124687084>

[2] Map函数和Reduce函数 <https://zhuanlan.zhihu.com/p/172532077>

[3] MapReduce编程实例：单词计数 <https://zhuanlan.zhihu.com/p/138993195>

# 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？（提供预处理代码preprocess.py以及处理后的sent\_receive.csv数据以供参考）。

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 β，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

本实验主要是对Pagerank算法的具体实现。压缩包里已经给定了preprocess.py文件，可以直接读入别名和人名id信息，并对邮件中出现的人名进行统一。接下来，根据邮件中的发送和接收方构造有向图的边。

在核心的pagerank算法中，首先遍历收件人和发件人，找到图中全部不重复的结点。接下来构建一个邻接矩阵M并对其初始化，使其在每一对（发件人，收件人）的位置上的值都为1，其余位置上的值为0，其边长为所有结点的总数N。

此后，对M的每一列进行标准化，具体操作是使每一列的和等于1。实现标准化的代码如下。

|  |
| --- |
| **for** j **in** range(N):  sum\_of\_col = sum(M[:, j])*#取得每一列的矩阵之和* **for** i **in** range(N):  **if** M[i, j]:  M[i, j] /= sum\_of\_col |

做好上述准备工作后，需要实现算法的核心模块——迭代。定义误差e、迭代次数k、阻尼系数β，并分别将其初始化为1、0、0.85（阻尼系数β的取值一般在0.8到0.9之间，后续会进行调参并观察迭代次数随β的变化）。每次迭代中，用next\_r记录新的r矩阵，根据公式可得到next\_r的值。注意在任何一次迭代中，都要再次将r标准化。若某时刻计算得到的误差e小于10-8，则停止迭代。迭代部分的代码如下。

|  |
| --- |
| **while** e > 0.00000001: *# 迭代到误差小于10e-8停止* next\_r = np.dot(M, r) \* b + (1-b) / N \* np.ones(N)sum\_of\_col = sum(next\_r)  next\_r = next\_r / sum\_of\_col *#确保整个矩阵值的和保持为1* e = next\_r - r  e = max(map(abs, e)) *# 计算误差* r = next\_r  k += 1 |

最后要输出人名id及其对应的pagerank值，直接采用数组循环输出即可。

算法的整体流程图如图2-1所示。

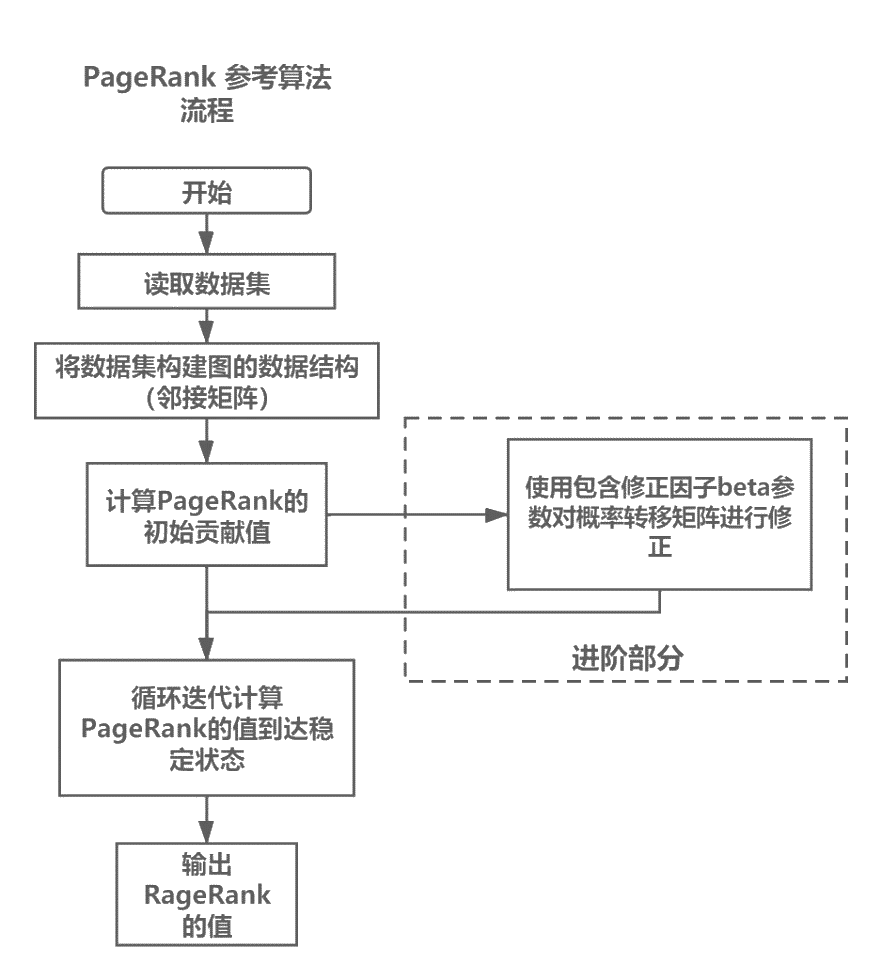


图2-1：算法流程图

### 2.3.2 数据结构与存储形式

**（1）edges**

①类型：二维数组

②作用：存储每条边的起点和终点（即发件人和收件人）。

**（2）nodes**

①类型：一维数组

②作用：存储当前所有的结点（发件人和收件人）。

**（3）M**

①类型：大小为N\*N的矩阵（N为nodes的大小）

②作用：邻接矩阵，记录每一对用户之间的(sent, receive)，若发件人a给收件人b发邮件，则M[a][b] = 1，否则为0。

**（4）next\_r**

①类型：大小为1\*N的矩阵，即一维数组

②作用：存储每次迭代后所有人的pagerank值。

### 2.3.3 遇到的问题及解决方式

实验中主要遇到了两个问题，其一是Python解释器没有配置好，导致一些库函数调用的时候报错，实际上只需要在Project Settings中设置好相应的环境即可解决。

其二是在迭代的过程中，忘记将r标准化了，导致每次r中元素的总和都不等于1，最后收敛时的pagerank总和也不等于1。这里当时调bug找了好半天，实际上只需要进行标准化就能解决，经验证，最后输出的pagerank结果的和为1，完美解决了这一问题。

### 2.3.4 实验测试与结果分析

在β=0.85下运行pagerank.py，输出的人名id与pagerank值如图2-2所示（节选一部分）。可以看到总迭代次数为55次。

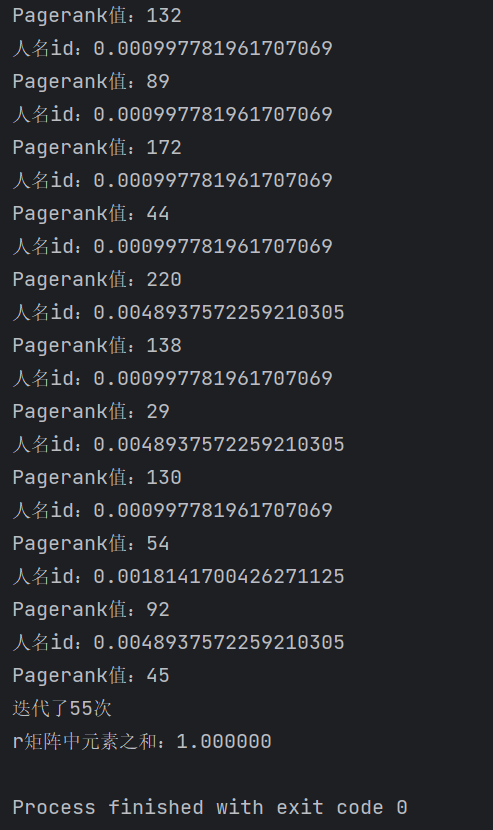
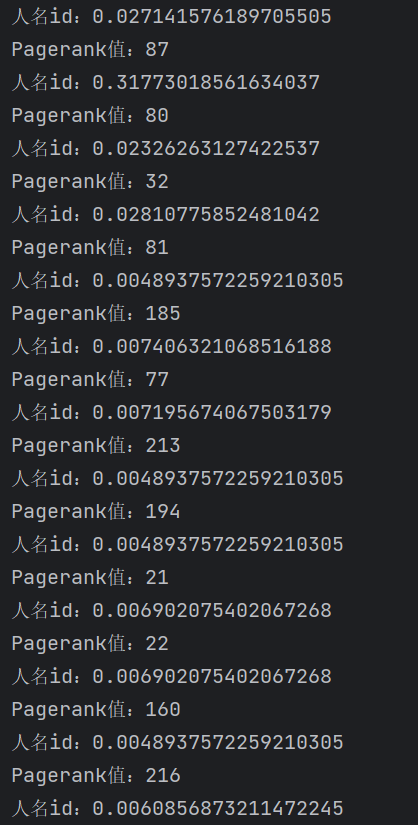


图2-2：β=0.85时的人名id、pagerank值与迭代次数

为了进一步观察所有人的Pagerank值，直接输出整个r矩阵，如图2-3所示。

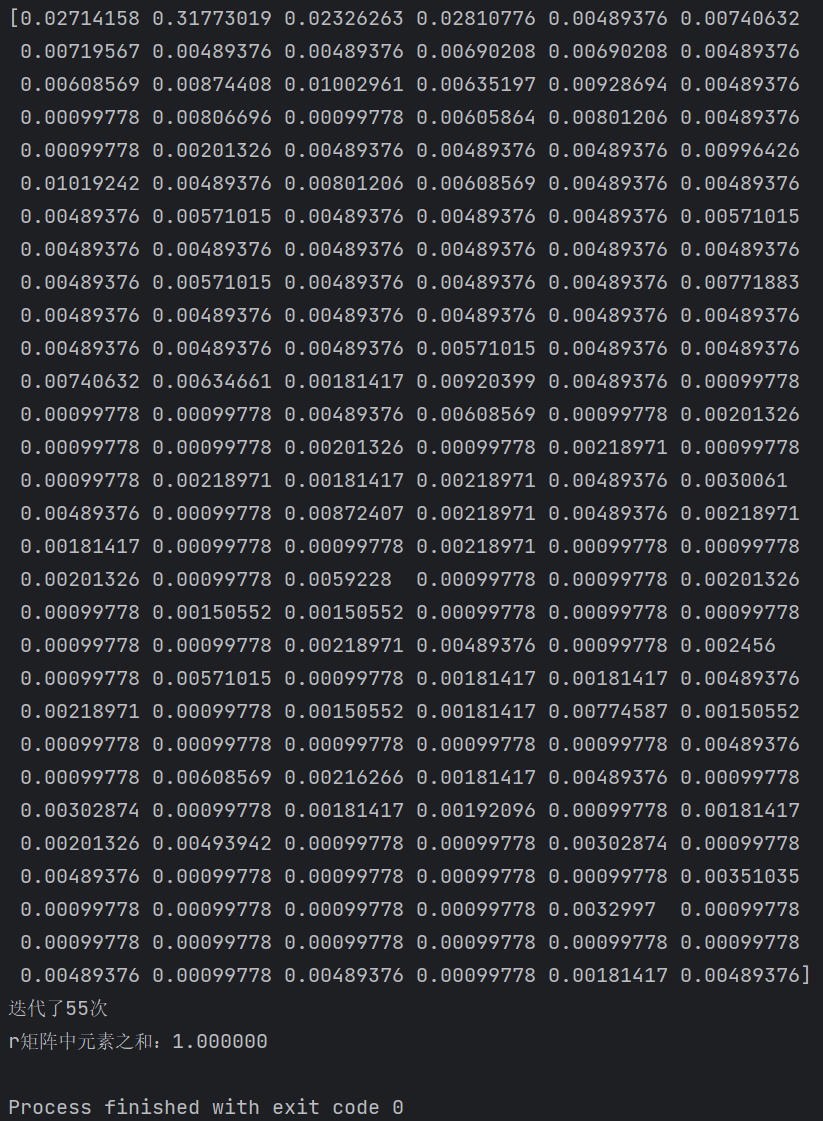
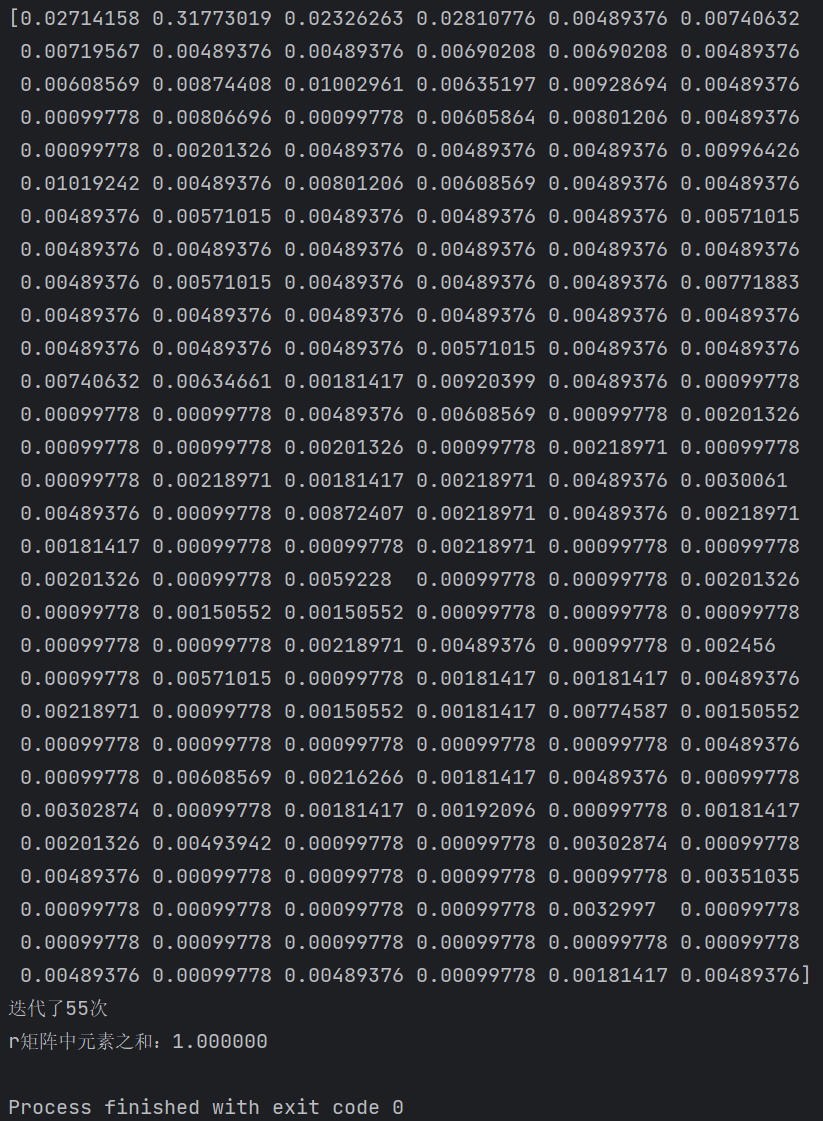


图2-3：记录Pagerank值的r矩阵的最终状态

分别改变β=0.8和β=0.9，得到不同的迭代次数分别为46次和69次，如图2-4、图2-5所示。

与β=0.85时迭代55次相比较，上述现象出现的原因是β越大，每次迭代中teleport的概率越低，越容易陷入spider trap和dead ends的问题，导致收敛越慢。若取β=1，则Pagerank算法回归到最原始的版本，如果图中存在spider trap和dead ends，则算法有可能无法收敛。这也很好的解释了为什么目前网页重要度排序中一般都会用β来调节teleport的效果。

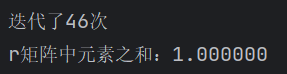
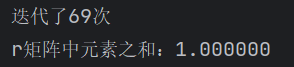
 

图2-4：β=0.8时的迭代次数 图2-5：β=0.9时的迭代次数

## **2.4 实验总结**

本次实验中，我具体实现了Pagerank算法，进一步理解了一个重要的互联网网页重要度排序方法。

它的亮点主要是在有向图上定义一个随机游走模型，通过一定概率跳出，避免了死循环等问题。通过这种算法，可以很快速地对网页等级进行标识，同时还能应用于社会影响力分析、文本摘要等多个领域。

本次实验还进一步让我掌握了利用Python实现迭代算法的能力，在代码熟练度上有很大提升。

## **2.5 参考文献**

[1] 图机器学习——3.2 PageRank 算法问题改进与延伸 <https://blog.csdn.net/weixin_41929524/article/details/121736652>

[2] cs224w（图机器学习）2021冬季课程学习笔记4 Link Analysis: PageRank (Graph as Matrix) <https://developer.aliyun.com/article/1053533>

# 实验三 关系挖掘实验

## **3.1 实验内容**

**必做：**

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

**加分项：**

1. 实验内容

在Apriori算法的基础上，要求使用pcy或pcy的几种变式multiHash、multiStage等算法对二阶频繁项集的计算阶段进行优化。

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

输出pcy或pcy变式算法中的vector的值，以bit位的形式输出

 参数不变，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

本次实验要求实现Apriori算法，以及其进阶的PCY算法（利用Multihash、Multistage完成）。两种算法的基本流程都是不断迭代生成候选频繁项集和频繁项集，最终生成关联规则，但在构造频繁项集的过程中，使用PCY算法可以优化计算效率。

**（一）Apriori算法编程思路**

首先，根据Apriori算法，我需要通过候选频繁一项集C1构造频繁一项集L1，并根据频繁一项集L1构造出候选频繁二项集C2，依次类推逐步构造出L2、C3、L3。最后，需要在L1、L2、L3中找出全部的关联规则。整体算法流程图如图3-1所示。

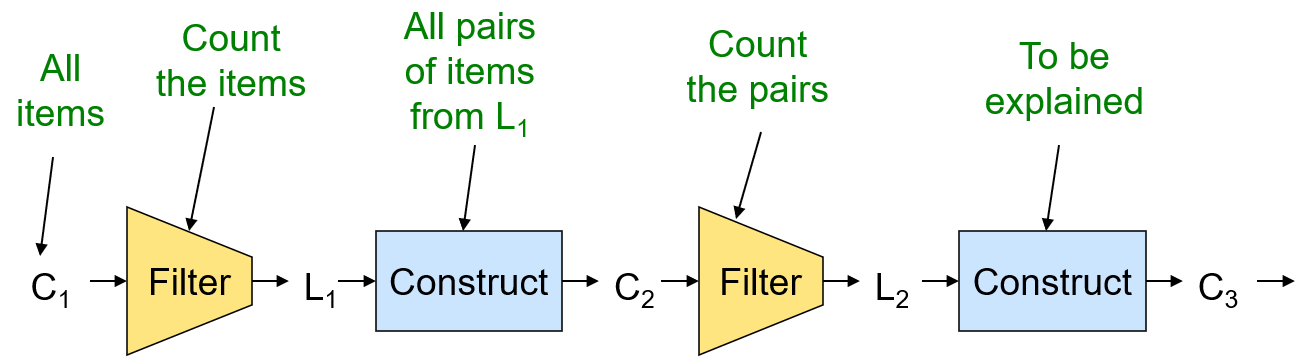


图3-1：Apriori算法流程图

在具体实现上，首先通过读取原始的Groceries.csv数据集（如图3-2所示）得到全部的C1，然后分别统计一项集中每个元素的标准化支持度，若support≥0.005（本次实验中规定的最小支持度），则将其视为频繁项集L1中的元素。对C1中每个元素遍历一次，并统计其support值即可。标准化支持度的计算方法是：

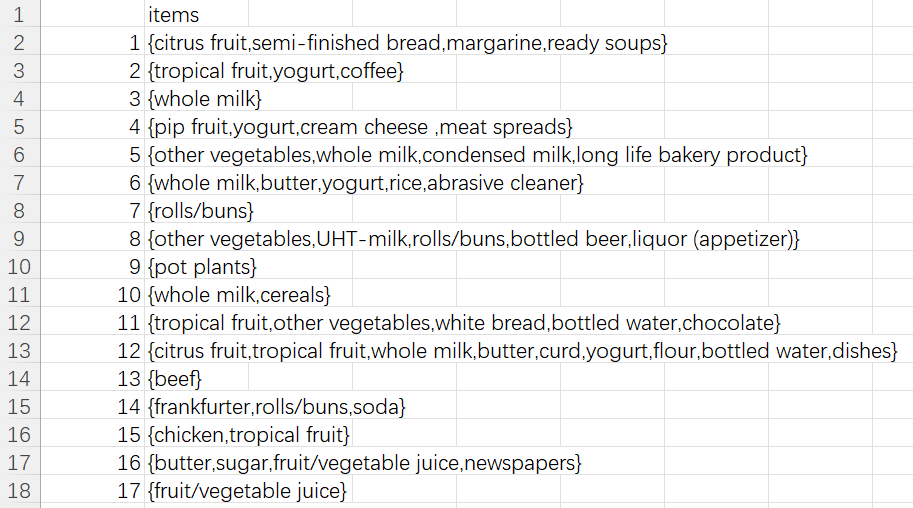


图3-2：原始数据集部分预览

接下来，要根据L1构造C2。我对L1中的元素进行两两组合（例如由得到新的集合），并对组合后的大量可能集合进行了检验，以确保进入C2的集合都是合法的。同理，对C2中的每个元素（集合）遍历统计其support值，并判断能否进入L2。

在根据L2构造C3的过程中，采用了比较直接的方式，即两两比较L2中的集合，若两个集合中只有一个元素不同，那么取这两个集合的并集，就能得到含有三个元素的集合。这个过程中需要采取剪枝策略提高算法的性能，主要是依据Ck+1中每一个集合的子集都在Lk中的原理，对每个三元组依次去除其中任意一个元素，并判断剩下的二元组是否在L2中。若出现一个二元组不在L2中，那么说明这个三元组不能加入C3。实现构造功能的代码如下。

|  |
| --- |
| **def** is\_apriori(ck\_item, Lk): *#在从Lk构造CK+1的过程中，判定一个项集ck\_item是否在Ck+1中* **for** item **in** ck\_item:  sub\_item = ck\_item - frozenset([item])  **if** sub\_item **not in** Lk:  **return False  return True**  **def** Create\_Ck(Lk, k): *# 通过合并Lk-1中的项，获得Ck中的项* Ck = set()  len\_Lk = len(Lk)  list\_Lk = list(Lk)  **for** i **in** range(len\_Lk):  **for** j **in** range(i + 1, len\_Lk):  l1 = list(list\_Lk[i])[0:k - 2]  l2 = list(list\_Lk[j])[0:k - 2]  l1.sort()  l2.sort()  **if** l1 == l2:  Ck\_item = list\_Lk[i] | list\_Lk[j]  **if** is\_apriori(Ck\_item, Lk): *#如果只含k-1个元素或含有大于k+1个元素，则会被直接筛掉* Ck.add(Ck\_item)  **return** Ck |

最后，利用关联规则的置信度公式，得出全部的有效规则。其中，置信度的计算方法是：



其中I是n元频繁项集，j是与I有关系的单个货物。

若某条规则满足conf≥0.5（本次实验规定的最小置信度），则将其视为有效规则并加入关联规则集合中。实现这部分功能的代码如下。

|  |
| --- |
| **def** get\_Rule(L, support\_data, min\_confidence):*#获取关系规则  # 参数：所有的频繁项目集（L1~L3），项目集-支持度dic，最小置信度* rule\_list = []  sub\_set\_list = []  **for** i **in** range(len(L)):  **for** frequent\_set **in** L[i]:  **for** sub\_set **in** sub\_set\_list:  **if** sub\_set.issubset(frequent\_set): *# 寻找上一层循环中出现的frequent\_set的子集* conf = support\_data[frequent\_set] / support\_data[sub\_set] *# conf(rule)=S(J)/S(J-j)* rule = (sub\_set, frequent\_set - sub\_set, conf)  **if** conf >= min\_confidence **and** rule **not in** rule\_list: *#找出大于最小置信度的规则* rule\_list.append(rule)  sub\_set\_list.append(frequent\_set)  **return** rule\_list |

**（二）PCY算法（Multihash）编程思路**

PCY算法相比Apriori算法最大的不同就是利用哈希桶的方式，快速过滤了大量不可能成为频繁项集的候选项集（它们被哈希到不频繁的桶中），同时也不会漏掉真正的频繁项集（它们被哈希到频繁的桶中）。在算法实现上，只需要对二阶频繁项集的计算阶段进行优化，此时需要建立两个哈希表（分别对应两个不同的哈希函数）。接下来分别利用两个哈希函数mmh3.hash()和hash()计算各个候选二项集的哈希值，并将这个哈希值所对应的vector矢量加1。其中，mmh3.hash()属于一种非加密型哈希算法MurmurHash3，它一般用于哈希检索操作，当前版本是MurmurHash3，能够产生出32-bit或128-bit的哈希值。这种哈希算法对于随机分布特征表现更加优良，因此这次我将其作为PCY算法中的一个哈希函数。在判断每个二项集能否进入L2时，只需要判断其所在的哈希桶的support值是否大于等于最小支持度即可。哈希桶的支持度计算方法是：

若某个二项集对应的两个哈希桶的support均大于最小支持度，则将其放入candidate集合中，否则该二项集不能放入candidate集合中。这是因为若该二项集要成为频繁二项集，那么必须满足三个条件：第一，二项集中的两个元素都是频繁的，这一点由L1可以保证；第二，二项集在第一个哈希函数下被分到的哈希桶是“频繁”的；第三，二项集在第二个哈希函数下被分到的哈希桶是“频繁”的。只要有一个哈希桶不是“频繁”的，则其中的所有二项集都不可能是频繁的。利用哈希桶更新candidate集合的代码如下（以哈希函数mmh3.hash()为例）。

|  |
| --- |
| **if** k == 2: *#用Multihash对二阶频繁项集的计算阶段进行优化  # 第一次哈希，使用mmh3.hash函数* hash\_table1 = [0] \* hash\_size  **for** transaction **in** data:  **for** itemset **in** candidates:  **if** set(itemset).issubset(set(transaction)):  hash\_value = abs(mmh3.hash(str(tuple(itemset)))) % hash\_size  *#将itemset转变成字符串并获取哈希值* hash\_table1[hash\_value] += 1  filtered\_candidates1 = [] *#记录可能为频繁项集的candidates* **for** itemset **in** candidates:  hash\_value = abs(mmh3.hash(str(tuple(itemset)),seed=12345)) % hash\_size  *#加入seed，但hash的效果没有区别* **if** hash\_table1[hash\_value]/len(data) >= min\_support: *#标准化* filtered\_candidates1.append(itemset)  candidates = filtered\_candidates1 *#更新candidates* **else**:  candidates = sorted(candidates) *#保持不变* |

最后，只需要再从candidate中选出L2，即可降低算法运行时间。整体的算法流程如图3-3所示。

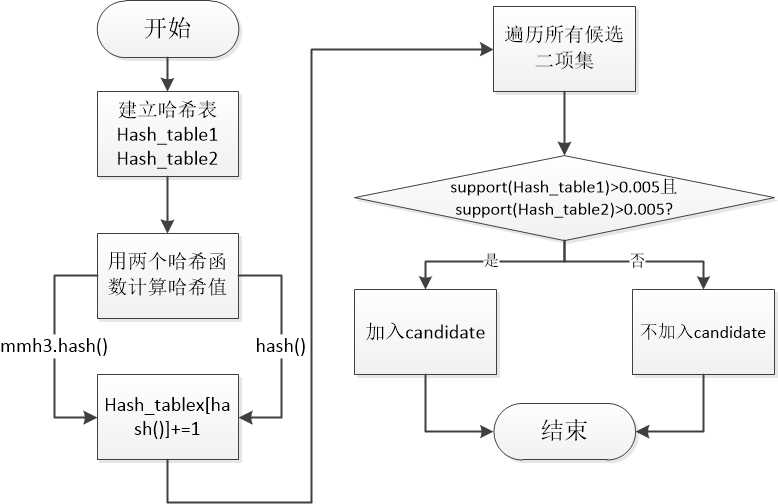


图3-3：PCY（Multihash）算法流程图

采用Multihash可以大幅度减少存储空间的需求，只需要保存两个vector矢量即可，如图3-4所示。在Pass 1中，包含了所有元素的总数以及两个哈希表。在Pass 2中，仅需存储所有的频繁项集、candidate集合中元素的总数以及两个哈希表对应的两个vector矢量，节省了大量内存空间。

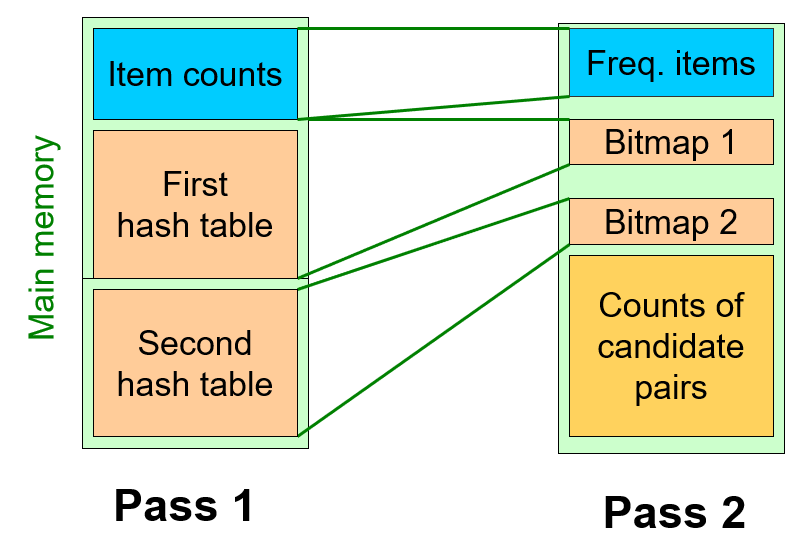


图3-4：Multihash算法的存储结构示意图

### 3.2.2 函数模块定义

**（一）Apriori算法函数模块**

**（1）data\_set**

①函数参数：无

②函数功能：读入Groceries.csv中的数据并存储。

**（2）Create\_C1**

①函数参数：data

②函数功能：根据data构造初始的候选频繁一项集C1。

**（3）is\_apriori**

①函数参数：ck\_item, Lk

②函数功能：在从Lk构造C(k+1)的过程中，判定一个项集ck\_item是否在C(k+1)中。这里依据的原理是C(k+1)每一个集合的子集都在Lk中。

**（4）Create\_Ck**

①函数参数：Lk, k

②函数功能：通过合并L(k-1)中的项，构造产生Ck中的项。

**（5）get\_Lk**

①函数参数：data\_set, Ck, min\_support, support\_data

②函数功能：从候选频繁k项集Ck中筛选出频繁k项集Lk。

**（6）get\_Rule**

①函数参数：L, support\_data, min\_confidence

②函数功能：获取最终的全部关系规则。

**（二）PCY算法（Multihash）函数模块**

**（1）generate\_candidates**

①函数参数：items, k

②函数功能：与前面的Create\_Ck函数一致，产生candidate集合。

**（2）get\_frequent\_itemsets\_pcy**

①函数参数：data, min\_support, hash\_size

②函数功能：利用两次哈希产生频繁项集，对二阶频繁项集的计算进行优化。

### 3.2.3 数据结构与存储形式

**（1）data**

①类型：二维列表，如[[item1, item2, …], [item3, item4, …], … ]

②作用：保存从Groceries.csv中读入的数据，每个一维列表保存初始数据集中一行中第二列的每个item。

**（2）C1、L1、C2、L2、C3、L3**

①类型：集合（frozenset）

②作用：Ck保存候选频繁k项集，用于筛选下一级的频繁多项集；Lk保存频繁k项集。实验中采用frozenset集合的形式存储C1、L1、C2、L2、C3、L3，其特点是一旦集合中的元素被确定后，就不能再对其进行增加或删除操作，从而防止在操作过程中对集合元素造成无意的修改。

**（3）L**

①类型：列表

②作用：存储全部的频繁k项集，即L1、L2、L3。

**（4）item\_count**

①类型：列表

②作用：存储各个item出现的次数，为后续计算支持度做好记录。

**（5）support\_data**

①类型：字典

②作用：保存每个k阶项的支持度，其中键为k阶项，值为支持度。

**（6）rule\_list**

①类型：列表

②作用：存储最终的关联规则，每一项包含关联规则左侧的item、右侧的item以及置信度。

**（7）candidate**

①类型：列表

②作用：PCY算法中经过两次哈希后得到的候选频繁项集。

**（8）hash\_table1、hash\_table2**

①类型：列表

②作用：PCY算法中存储两个哈希表的vector矢量，便于后续计算每个桶的支持度。

### 3.2.4 遇到的问题及解决方式

本次实验中主要遇到了两个问题。第一是在基础的Apriori算法中，在计算support值时没有除以购物车的总数，导致计算出来的L1、L2、L3三个频繁项集的元素总数都和标准答案不一致，这是比较粗心的一个问题。

第二是在PCY算法中，我使用了两个哈希函数，分别是mmh3.hash()以及hash()，然而这两个函数得到的哈希表的vector矢量均显得非常不均匀，大量的hash值出现了0次，而只有很少量的hash值出现了很多次。后来我发现，我是将每个项集中的集合转变成字符串并获取其hash值，但是整个Groceries文件中的货物种类总共只有169种，在单词上有高度的重叠，因此最后得到的hash值也是分布不均的，不过这不会影响最终的结果。

### 3.2.5 实验测试与结果分析

**（一）Apriori算法实验结果分析**

运行apriori.py，分别输出了C1、L1、C2、L2、C3、L3的总项数，以及L1、L2、L3三个项集所含的元素以及相对应的支持度support。最后输出了关联规则的总数目、具体条目以及每条规则的置信度confidence。如图3-5~图3-8所示。

可以看出，L1有120项，L2有605项，L3有264项，与课上给出的参考答案一致，说明算法运行正确。



图3-5：C1、L1具体信息



图3-6：C2、L2具体信息



图3-7：C3、L3具体信息

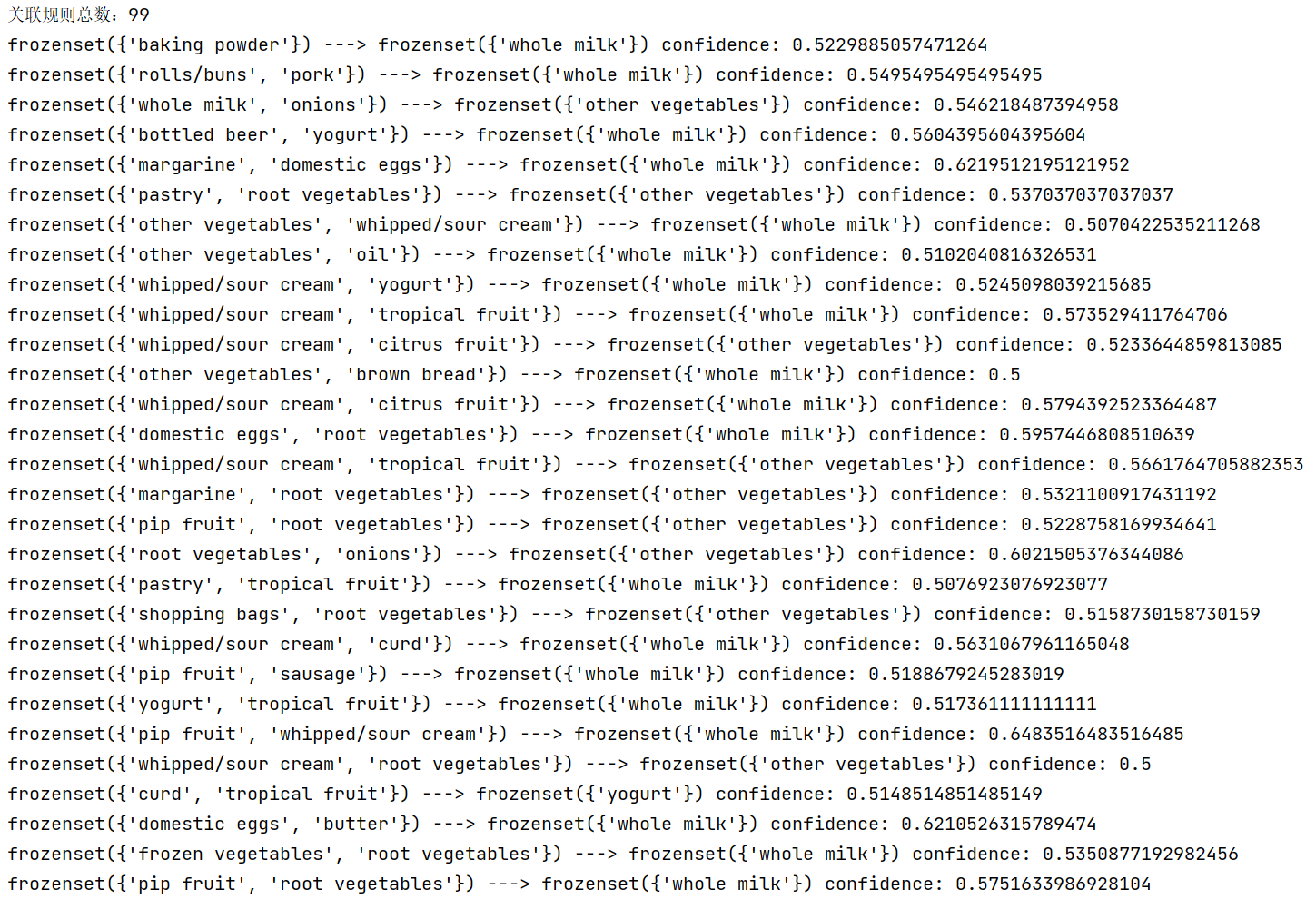


图3-8：关联规则信息

**（二）PCY算法实验结果分析**

运行pcy.py，输出了两个哈希表各自的vector矢量（以bit位的形式表示），以及L1、L2、L3三个项集所含的元素以及相对应的支持度support。最后输出了关联规则的总数目、具体条目以及每条规则的置信度confidence。如图3-9~图3-13所示。

可以看出，L1有120项，L2有605项，L3有264项，说明算法运行正确。

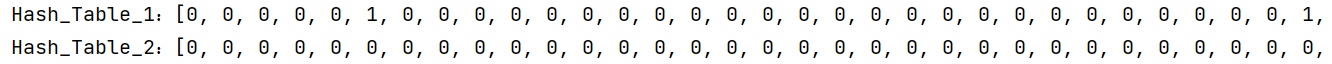


图3-9：Hash\_Table\_1和Hash\_Table\_2的bit vector信息（节选）

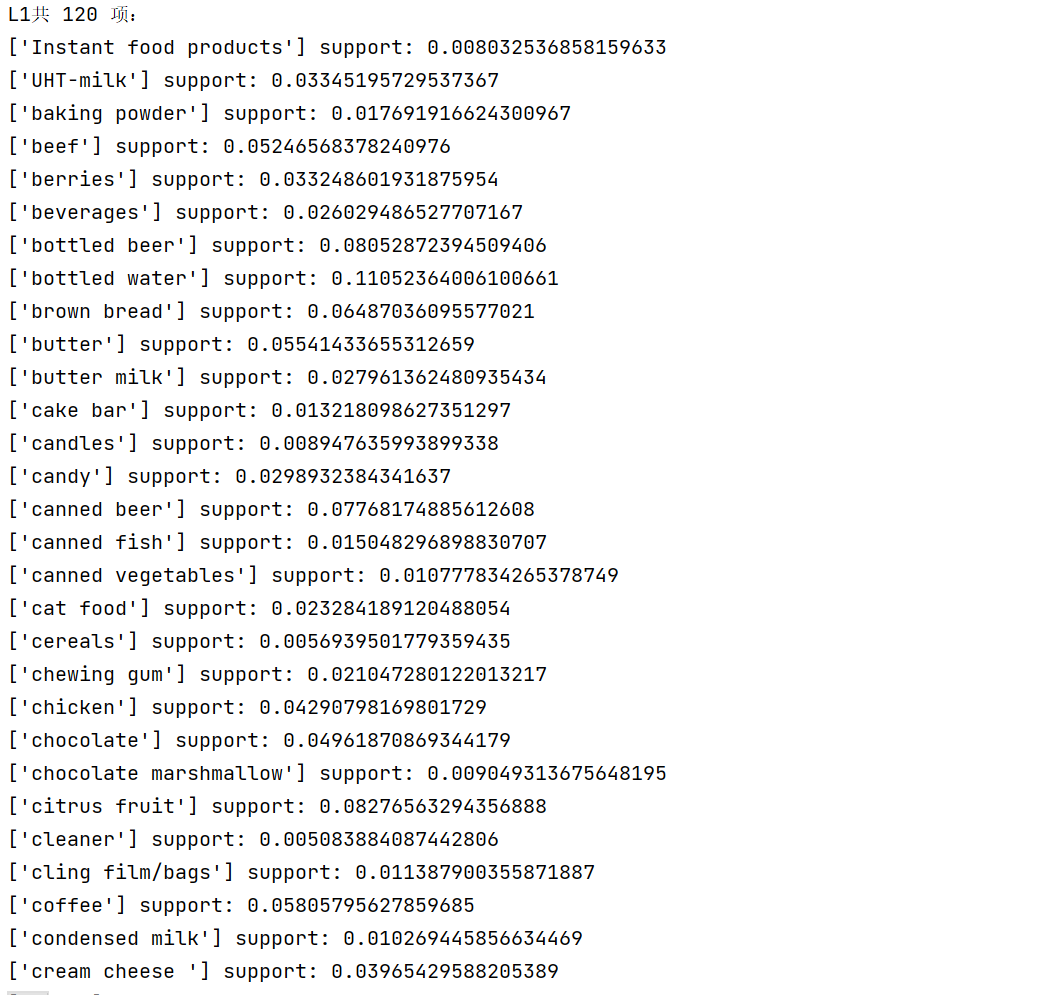


图3-10：L1具体信息



图3-11：L2具体信息



图3-12：L3具体信息



图3-13：关联规则信息

## **3.3 实验总结**

通过对Apriori算法的实现，我进一步掌握了迭代生成频繁n项集的方法，理解了项集的支持度、关联规则的置信度的计算方法，并得到了频繁的一至三项集以及全部的关联规则。这个算法主要是利用逐层搜索的迭代过程，找出各阶的频繁项集，其过程由连接与剪枝组成。实际上，这个算法在现实生活中有着十分重要的作用，例如分析消费者购买行为模式，并使商家制定更合理的销售策略。

除了完成基础的Apriori算法之外，我还使用基于Multihash的PCY算法进行了一定的改进。通过使用两个哈希函数、建立两个哈希表，我将全部的候选二项集分配到了不同的哈希桶中，并根据候选二项集成为频繁二项集的必要条件将其有选择性地放入candidate集合中，优化了二阶频繁项集的计算效率。无论是哪种算法，最后得到的结果均与参考结果一致。

值得一提的是，无论是Apriori还是PCY算法中都涉及到了剪枝的重要思想，在数据量很大的时候，全部遍历并获得每一种可能的情况是十分繁杂且没有必要的。在计算过程中，只需要时刻检查当前情形是否合法，若不合法则直接停止后续构造，即可大幅度减少可能性的总数，提高计算效率。

## **3.4 参考文献**

[1] 关联规则挖掘 <https://blog.csdn.net/yinlili2010/article/details/40193457>

[2] Apriori算法详解 <https://zhuanlan.zhihu.com/p/341882260?utm_id=0>

[3] 频繁项集与关联规则Frequent Itemset Mining and Association Rules、A-Priori算法、PCY算法 <https://blog.csdn.net/qq_39852142/article/details/111586236>

# 实验四 kmeans算法及其实现

## **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，数据集已经被归一化。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。效果展示图可如图4.1所示。



图4.1 葡萄酒数据集在黄酮和总酚维度下聚类图像（SSE为距离平方和，Acc为准确率）

## **4.3 实验过程**

### 4.3.1 编程思路

本次实验主要是通过kmeans算法对给定的葡萄酒数据进行分类（3种类别）。Kmeans算法会将给定数据集分成k个簇，且每个簇的中心采用簇中所有点在对应维度上数据的平均值。

具体的编程思路如下：

1. 使用csv.reader()函数从“归一化数据.csv”中读取各项数据，并以float的形式存储。这样做的原因是数据集中的归一化数据均为小数类型；
2. 迭代准备：使用rd.random()函数随机生成三个13维的初始中心点；
3. 迭代开始：每轮迭代中，首先按照距离最近原则，计算每一项输入数据到三个中心点的距离，并选取距离最小的中心点作为该点所在簇的中心点，将其划分至这个簇中。每轮遍历结束后，重新按照三个簇中所有点在13维上数据的平均值计算出新的中心点。重复这一过程，循环迭代。
4. 迭代结束：若某轮迭代最后，数据集某一项所在簇的中心点相对前一轮迭代发生变化，或者是它与中心点的距离发生变化，说明迭代没有收敛。反之，从这一轮开始之后的所有迭代结果均不会发生变化，因此迭代结束。
5. 计算三个聚类的SSE（误差平方和）。它的计算公式是：

其中是13维数据中的一项。SSE值越小，代表数据项越接近其各自的中心点，聚类效果越好。因此计算SSE可以评估最终聚类效果。

1. 使用get\_acc()函数计算最终分类的准确度。准确度的计算公式是：

其中“正确分类的项目数”是指三个类别中正确分类的数量之和，每个类别中，以出现次数最多的种类作为这个类的种类。

1. 使用matplotlib中的pyplot.scatter()函数绘制聚类示意图，图中三个簇分别对应红、蓝、绿三种不同的颜色。用户可以输入任意两个维度，得到的图形也具有多样性。

整体的程序流程图如图4-1所示。

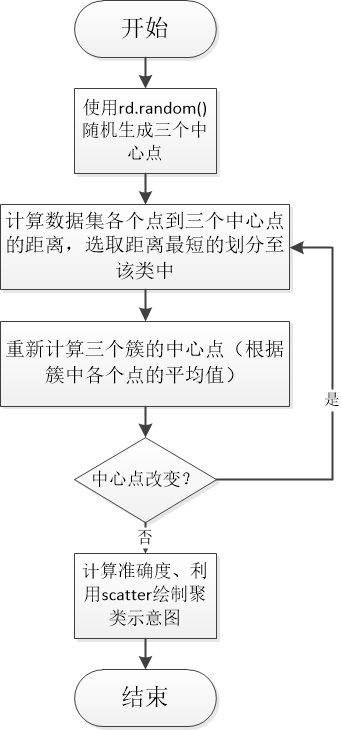


图4-1：程序流程图

上述迭代过程的代码实现如下。

|  |
| --- |
| **while** change:  count += 1  change = **False  for** i **in** range(m):  min\_s = 999999.0  min\_center = -1  **for** j **in** range(k):  distance = len\_s(centers[j], data[i])  **if** distance < min\_s:  min\_center = j + 1  min\_s = distance *# 计算每个点到中心点的距离并得到最小值* **if** data\_min[i, 0] != min\_center **or** data\_min[i, 1] != min\_s: *#中心点变化，或者是到中心点的距离发生变化* data\_min[i, :] = min\_center, min\_s  change = **True** *# 更新每个点的最近中心点和距离信息* **for** j **in** range(k):  point = get\_points(m, data, j, data\_min) *# 根据求得的每个点的信息分类* centers = np.array(centers)  centers[j, :] = np.mean(point, axis=0) *# 计算出新的中心点* |

计算聚类准确度的代码实现如下。

|  |
| --- |
| **def** get\_acc(m, data, k, data\_min):  hit = 0  **for** j **in** range(k):  cluster\_tmp = [0, 0, 0]  point = get\_points(m, data, j, data\_min)  **for** item **in** point:  cluster\_tmp[int(item[0])-1] += 1  hit += max(cluster\_tmp)  acc = hit / len(data)  print(**'准确度(acc)：'**, acc) *# 准确度为正确分类数与总数之比* |

### 4.3.2 函数模块定义

**（1）len\_s**

①函数参数：center, data

②函数功能：计算数据集中某个点到中心点的距离。

**（2）ssenum**

①函数参数：m, data\_min

②函数功能：分别计算三种聚类的误差平方和SSE。

**（3）get\_points**

①函数参数：m, data, j, data\_min

②函数功能：根据求得的数据集中每个点的信息（与其最近的中心点）将其分配到三个聚类中。

**（4）get\_acc**

①函数参数：m, data, k, data\_min

②函数功能：迭代结束后，计算最终的聚类准确度。

### 4.3.3 数据结构与存储形式

**（1）data**

①类型：二维数组

②作用：存储归一化数据集中每一项的信息。

**（2）data\_min**

①类型：二维数组

②作用：分别存储某个数据所在簇的中心点及其与中心点的距离。

**（3）centers**

①类型：列表

②作用：存储三个中心点的13维坐标。

**（4）clusters\_tmp**

①类型：列表

②作用：计算聚类准确度时，存储0、1、2三种类别的分类数并与真实值进行比较。

**（5）sse\_num**

①类型：列表

②作用：存储三个聚类各自的误差平方和SSE。

### 4.3.4 对已归一化数据集的理解

任务书前面提出“数据集已经被归一化”，并让我思考归一化的作用以及不进行归一化带来的影响。

我认为此处对原始数据集进行归一化可以使数据处理更加方便，它将13维的葡萄酒数据限制在0~1的区间范围内，使各个维度的数据在决定数据点的特征上具有同等的地位。

如果没有进行归一化，原始数据集会放大某些因素对聚类的影响，同时会在很大程度上忽略某些因素。如图4-2所示，可以看到第一维数据范围大约在13~15之间，但第二维数据范围大约在1~3之间，二者的尺度差别太大，需要在放缩后才能使用。

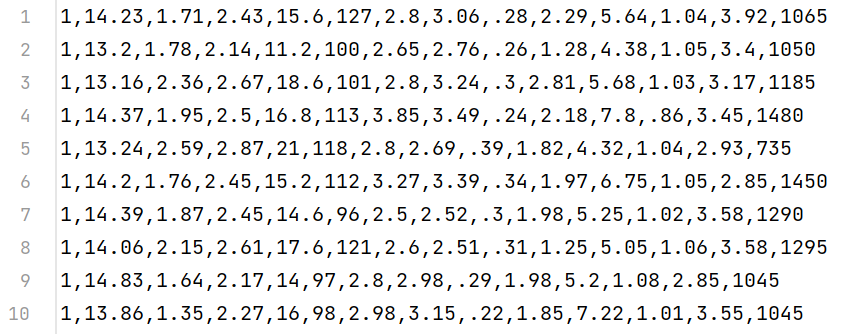


图4-2：原始数据集的13维数据示意图

### 4.3.5 遇到的问题及解决方式

实验中主要遇到了两个问题。第一是最开始用pyplot.scatter()函数绘制图像时，没有弹出聚类结果图，后来查阅资料发现，要再加一句plt.show语句才能把绘制好的图形展示出来。第二是在判断迭代终止条件时，一开始只认为数据集中每个点对应的中心点不变就能说明聚类完成，实际上还存在一种可能的情形就是到中心点的距离发生变化，这时该点需要再次计算到其它中心点的距离，因此可能会划分到另一个簇中。

### 4.3.6 实验测试与结果分析

运行kmans.py，程序输出了循环次数、三个聚类各自的SSE、聚类准确度，如图4-3所示。可以看出三个聚类的SSE分别为19.998、15.056、13.935，最终的准确度为0.9438，拟合效果较好，因此程序完成了应有的功能。

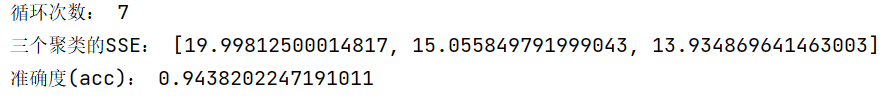


图4-3：聚类结束后输出的信息

此后程序提示输入两个需要绘制聚类关系图的属性，以1~13表示。输入“1 2”并回车，程序自动绘制出属性1和属性2的聚类关系图，如图4-4、图4-5所示。

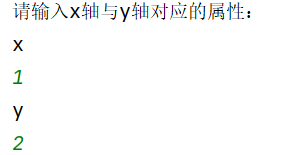


图4-4：输入要绘制的属性

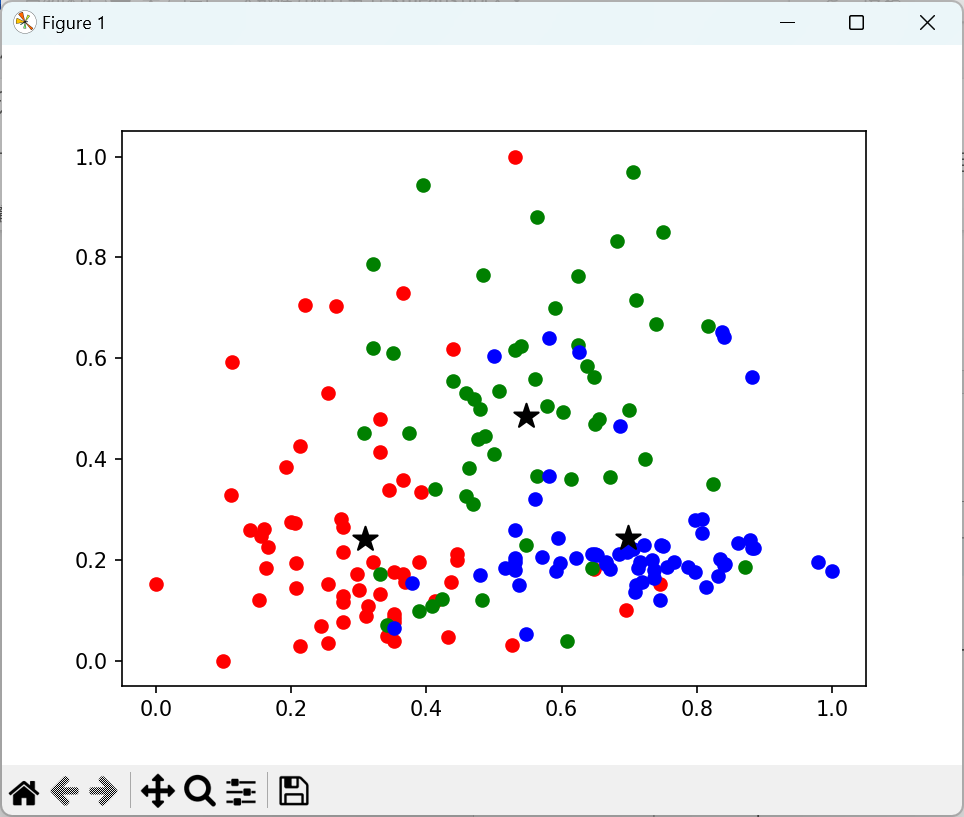


图4-5：属性1和属性2的聚类关系图

上图中，三个聚类的中心点用五角星“★”表示。

## **4.4 实验总结**

本次实验中，我通过对kmeans聚类算法的实现，掌握了一种重要的聚类处理算法。在面对大规模数据集时，聚类可以更快地掌握样本数据集的大体分布与特征。使用聚类算法时，要特别注意对所给数据进行归一化，以便于更好地保证迭代快速收敛。

## **4.5 参考文献**

[1] K-Means（K-均值）聚类算法<https://blog.csdn.net/sikh_0529/article/details/126806720>

[2] 聚类算法数据预处理——数据归一化 <https://it.cha138.com/jingpin/show-32640.html>