

**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： U202115612**

**姓 名： 俞若鹏**

**指导教师： 王 蔚**

**报告日期： 2023/5/8**

**计算机科学与技术学院**

**目录**

[实验三 关系挖掘实验 1](#_Toc58793746)

**[3.1 实验内容](#_Toc58793747)** [1](#_Toc58793747)

**[3.2 实验过程](#_Toc58793748)** [1](#_Toc58793748)

[3.2.1 编程思路 1](#_Toc58793749)

[3.2.2 遇到的问题及解决方式 1](#_Toc58793750)

[3.2.3 实验测试与结果分析 1](#_Toc58793751)

**[3.3 实验总结](#_Toc58793752)** [2](#_Toc58793752)

# 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1~4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

本次实验包含两个部分，其一是基于Apriori算法对给定的数据进行关联关系挖掘；其二是在第一部分的基础上对Apriori算法进行改进，使用PCY算法，运用其中的bitmap映射提高内存的使用效率，提高算法的运算效率。

对数据集的清洗，清洗后的数据采取列表结构存储，每个列表元素对应一个子列表，表示一行数据中购物车所包含的商品集合。

Apriori算法是对数据集筛选出所需大小的频繁项集，生成大小为n的频繁项集需要进行n轮Apriori算法，每一轮的算法执行流程大致一直，具体执行流程为对上一轮筛选出的n-1大小的频繁项Ln-1集排列组合成大小为n的候选集Cn，

随后对Cn在原始数据集中扫描计算其支持度及其关联规则的置信度，筛选出超出阈值的候选频繁项，最后组成第n轮的频繁项集Ln。

在本次实验的算法实现当中，对于第一轮候选集的选出是单独编写init\_candidate函数，遍历数据集列表中所有的购物车，并统计所有商品的种类集合以及商品在数据集中出现的次数，随后进行过滤筛选出频繁项集。对于后续轮次的算法执行，实验采用一个工具类ClusterUtil封装算法所需的全部函数，其中包括：

**函数名：**count

**函数功能：**对依据排列组合选出的候选集在数据集中统计其中候选项出现的次数。

**函数名：**filter

**函数功能：**对于count过后的候选集计算其中每一个候选项的支持度以及其关联规则的置信度，符合要求的候选项添加到频繁项集当中，最后返回频繁项集。

**函数名：**construct

**函数功能：**基于上一轮的频繁项组合出这一轮的候选集，组合规则是对Ln-1中两个频繁项如果其[:n-1]是相同的就将它们两个组合，例如AB和AC，都包含A，则组合成ABC加入候选集。

PCY算法相对Apriori算法所做的改进是在本轮算法的执行过程中计算下一轮频繁项集的bitmap映射，在每一轮的construct函数的候选集组合阶段需要借鉴上一轮生成的bitmap以减少候选集的大小，从而减少计算量；每一轮中会额外生成一个HashTable，对于当前轮次的频繁项集中每一个频繁项进行hash映射到HashTable中并进行计数，最后将HashTable中计数大于阈值的序列号映射要bitmap中相应的bit位中，将对应的bit位置1，生成的bitmap相对于HashTable在内存占用上有极大的压缩，充分利用本轮内存的同时也不会影响下一轮的内存使用。

实验中在ClusterUtil类中删除了filter函数，添加了hash\_with\_bitmap函数，将filter功能合并到了count函数当中，在construct函数中对每一个组合出的候选项hash判断其在bitmap中对应的bit位是否为1，不为1则不降其加入候选集中，从而达到减少候选集大小的目的，从而减少后续函数功能执行的计算量。

**函数名：**construct

**函数功能：**基于Apriori算法的construct，不同之处在于，对于候选集的选出是直接对原始数据集进行排列组合，并将组合结果hash到对应的bitmap上的bit位，判断是否是在频繁位上，满足要求才会加入候选集。

**函数名：**count

**函数功能：**对于construct初步过滤后的候选集再次进行候选过滤，统计每一个候选项在数据集中出现的次数，并过滤掉支持度小于阈值的候选项。

**函数名：**hash\_with\_bitmap

**函数功能：**对初始数据集内每一个购物车中的商品进行排列组合，对组合出的候选项调用hash函数获取其哈希值，映射到HashTable中，并对对应的值计数加一；在全部组合完后对HashTable进行压缩，计算每个item中值的支持度，符合要求的item映射到Bitmap对应的bit位中。

### 3.2.2 遇到的问题及解决方式

* **问题描述：**在Apriori算法中如何针对上一轮的频繁项集选出下一轮的候选集不知道从何入手。

**解决方式：**一开始对于候选项和频繁项都采用python中set的数据结构，组合候选项是，对频繁项采取集合运算，如果并集大小为n-1就将两个频繁项进行并集运算生成候选项，但是在程序最终运行结果却远大于预测值，在debug过程中发现对于AB、AC、BC这样的频繁项集，两两组合判断的过程中会生成三个ABC候选项，从而大大增大了候选集大小；在查阅资料过后，发现只需要判断前n-1个元素是否一致即可，但是在实现的过程中忽略了python中set结构内部元素的存储是无序的，但set非常方便count和filter阶段的的算法运算，略作取舍后，还是将set替换成了tuple，保证了候选集和频繁集内部数据元素的有序性，方便下一轮候选集的选出，在count和filter函数内部对tuple转化成set再进行运算。

* **问题描述：**编写PCY算法时，仍然采用Apriori算法候选集的选取方式，在实际运行时发现结果不正确

**解决方式：**由于PCY算法需要生成bitmap，所以每生成一个候选项都需要在原始数据集中遍历得出该项出现的次数，并加入对应键的HashTable当中，这样的做法会导致count和construct功能的高度耦合，同时由于每生成一个候选项都需要遍历一次数据集，会导致程序性能大大下降。

在分析危害后，舍弃了Apriori算法依据上一轮频繁集筛选候选集的选取方式，改为依次对数据集的每一个购物车调用combinations函数进行排列组合，再对组合出的候选项hash映射，生成候选项的同时也在对HashTable进行计数，改进后仅需要遍历一次数据集即可，也保证了count函数和construct函数功能的耦合度。

* **问题描述：**由于PCY算法的一轮运算中bitmap的生成和频繁项集的生成二者是相互独立的，为了提高效率，我采用两个进程执行异步执行两个函数，但在结果输出时却发现频繁项集为0

**解决方式：**分析后发现，由于某些特殊原因，进程没有成功阻塞，导致下一轮计算时使用的仍然是未更新的bitmap，导致错误的剔除了候选项，与此同时也发现程序的执行效率并没有较大地提升，就摒弃了多进程的使用，改回同步执行。

### 3.2.3 实验测试与结果分析

对Apriori算法进行测试：



图1. Apriori算法测试

对PCY算法进行测试：



图2.PCY算法测试

可以看出PCY算法相较于Apriori算法效率有较大的提升，从每个轮次的运行时间也能看出Apriori算法运行时长集中在二项式和三项式，PCY则是呈递增的趋势，这也正好印证了各自的算法思路，Apriori是在上一个频繁集的基础上筛选候选集，算法的效率就涉及到上一个候选集的大小以及组合出的候选集的大小，二项和三项的候选集规模都巨大，通过debug可以看到二项式候选集的大小：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 轮次 | 1 | 2 | 3 | 4 |
| 候选集大小 | 169 | 7140 | 9374 | 1368 |

候选集的大小也符合算法执行时长的规律。

而PCY算法是对数据集直接进行排列组合产生候选集的，在debug下观察 候选集的大小：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 轮次 | 1 | 2 | 3 | 4 |
| Bitmap过滤前 | 43367 | 137278 | 399316 | 1098835 |
| Bitmap过滤后 | 43367 | 73267 | 36092 | 2045 |

观察表格不难发现过滤前候选集的大小是随着项数的增大而增大的，这较好地吻合程序运行时间的分布规律，同时也能看出bitmap对于候选集筛选的巨大作用。

同时也能看出Apriori候选集的大小都是远小于PCY算法的，但是Apriori算法执行的时间要长很多。在程序内部分别对每个步骤进行计时观察：

|  |  |  |  |
| --- | --- | --- | --- |
| 轮次 | 2 | 3 | 4 |
| Construct / s | 0.0010 | 0.0264 | 0.0045 |
| Count / s | 8.3199 | 11.2915 | 1.7132 |
| Filter / s | 0.0010 | 0.0022 | 0.0010 |
| Total / s | 8.3219 | 11.3201 | 1.7187 |

观察表格可以看出Count阶段占据了绝大多数的运行时长。

经过分析，推测原因是Apriori算法在筛选候选集时首先需要对频繁项进行切割，切割后仍要判断切割部分是否一致以达到合并要求，尤其在计数时需要涉及数据结构的切换（元组转换为集合），且需要对候选集中的每一个元素在初始数据集内进行计数查询，循环次数为候选集大小\*数据集大小，达到了百万的级别，这大大降低了算法的执行效率。

而PCY算法每次生成数据集只需要遍历一次数据集即可，生成的初始候选集虽然很大但是能有效降低循环次数。

程序运行的结果保存在result.csv文件当中。

## **3.3 实验总结**

通过本次实验，我学会了关联关系挖掘算法的相关知识，了解到如何通过循环往复的迭代操作获取预期的关系组合，回过头看整个算法的流程，其实Apriori和PCY算法的实现并不算复杂，但值得我学习的是二者背后的精妙思想。

Apriori算法的核心思想是基于上一轮频繁项集推出下一轮要筛选的候选集，是很典型的一个迭代算法，一轮轮的操作会将候选集的数据逐渐缩减，这个操作即所谓的剪枝，能够高效地找出所有的频繁项集，而PCY算法则在Apriori算法的基础之上进行改善，又充分利用了每一轮程序的内存空间，在实验结果的分析中也能看出Bitmap对于数据量缩减的巨大作用，算法效率得到了巨大的提升。

虽然Bitmap能过滤大部分的组合，但是未过滤前的组合数据量可以看出是非常大，这也是PCY算法实现可以进一步优化的突破口，在hash\_with\_bitmap和construct中都有运用到combinations函数，这也是在进行重复的数据操作，

可以对hash\_with\_bitmap中组合的数据进行保存，这样在construct时只需要依据Bitmap进行筛选过滤即可。但为了体现轮次之间的相对独立性，我没有在实际代码实现中这么做。

学习和实现两个算法的过程中，我也意识到了如何揭示数据中的潜在关系，帮助我们做出更好的决策。同时，我也认识到了算法的优化和改进的重要性，只有不断探索和实践，才能不断提高算法的效率和性能。