
关联规则挖掘实验(Association Rule Mining)

姓名 查鹏 邮箱:1613541957@qq.com 联系方式: 15895987843

(南京大学 计算机科学与技术系, 南京 210046)

关键字: Association rule mining; Apriori; FP-growth;

1 实验数据和实验目的

1.1 实验数据说明

本次实验给的数据是一家超市的用户购买记录, 共有 9834 条购买记录。这些购买记录用 CSV 格式进行存储, 保存格式如下:

- 1 {商品 1, 商品 2, 商品 3,}
- 2 {商品 1, 商品 2,}

其中 1,2 这些为购买记录的表示符号, 花括号里的内容是该条购买记录购买的商品。下面是部分数据的截图:

9809	{sliced cheese, frozen meals, margarine, red/blush wine}					
9810	{beef, root vegetables, other vegetables, frozen vegetables, frozen dessert, domestic eggs}					
9811	{meat, hamburger meat, Instant food products, soda}					
9812	{citrus fruit, berries, other vegetables, whole milk, frozen meals, newspapers}					
9813	{tropical fruit, pip fruit, rolls/buns, pastry, bottled water, fruit/vegetable juice, newspapers}					
9814	{sausage, whole milk, sliced cheese, pastry}					
9815	{whole milk, yogurt, frozen meals, bottled water}					

用课本上的专业术语来说的话, 每条购买记录就是一个事务 T, 前面的序号就是事务标识符 TID, 花括号中的内容为项集(itemSet), 所有的事务构成一个事务集 D, 事务集的大小为 9834, 也就是 $|D|=9834$ 。

1.2 实验目的

- 1.2.1 使用 1.1 结束的数据集, 通过关联规则挖掘(Association rule mining)找出其所有的频繁项集(frequent itemSet)。并筛选出满足最小支持度(support)的频繁项集。
- 1.2.2 分别使用 Apriori 算法和 FP-Growth 算法进行关联数据挖掘, 并且比较两个算法的时间复杂度和空间复杂度。
- 1.2.3 尝试去探索一些在实验 Apriori 和 FP-Growth 算法时候的关联规则

2 实验原理

2.1 数据集的获取

2.1.1 数据的存储

使用 java 自带的集合类存储一个事务, 然后把集合作为元素存储在一个线性表中。比如用这样的格式:

```
private static List<Set<String>> dataset=new ArrayList<>();
```

每一个 Set<String>就是一个事务 T, 然后每一个 dataSet 就是一个事务集 D。

2.1.2 数据的读取

因为在事务集是存储为 CSV 格式, 我们读取 CSV 文件的时候只需要每次读取一行, 然后使用 java 自带

的 split 函数以逗号和花括号为分割符，把字符串分割为一个数组，然后去除数组的第一个和最后一个元素，剩下的就是我们需要的数据

```
br = new BufferedReader(new FileReader(csv));
while ((line = br.readLine()) != null) { //读取一行
    dset = new TreeSet<String>();
    data.add(line);
    String[] major = line.split("[},{]"); //分割字符串
    for (int i = 2; i < major.length-1; i++) //获取数据
        dset.add(major[i]);
    dataSet.add(dset); //将数据加入事务集中
}
```

2.2 Apriori算法

2.2.1 算法总体介绍

Apriori 算法使用一种逐层搜索的迭代方法，其中 k 项集用于探索 $k+1$ 项集。首先算法通过扫描数据库累计每个项的出现次数，并收集满足最小支持度的项找出频繁一项集，该集合记为 L_1 。然后通过 L_1 找出频繁 2 项集 L_2 ，使用 L_2 找出 L_3 ，如此下去，直到不能再找出频繁 K 项集为止。找出每个 L_k 都有扫描一次数据库。

为了提高频繁项集逐层产生的效率，算法中使用了先验知识：

频繁项集的所有非空子集也一定是频繁的！

2.2.2 获取频繁一项集(frequent_1_itemsets)

参数：以项 I 为主键，以项 I 的出现次数为 Value 的哈希表 dc

```
Function find_frequent_1_itemsets (D:事务集,min_sup:最小支持度){
    for each 项集  $L \in D$  {
        for each 项  $I \in L$  {
            If(dc 包含  $I$  ){
                dc 中主键  $I$  对应的出现次数 value 加 1;
            }
            Else{
                将  $I$  加入 dc 中，并将 value 记为 1;
            }
        }
    }
    取出 dc 中所有 value>min_sup 的主键
}
```

2.2.3 Apriori_gen()算法

算法目的：由 L_{k-1} 得到 C_k

```
Function apriori_gen (  $L_{k-1}$ : 频繁  $k-1$  项集,min_sup:最小支持度){
    for each 项集  $l_1 \in L_{k-1}$  {
        for each 项集  $l_2 \in L_{k-1}$  {
            if(isCanLink ( $l_1, l_2$  ){
                 $c = l_1 \cup l_2$ ;
                if has_frequent_subset (c,  $L_{k-1}$  ) then delete c;
            }
        }
    }
}
```

```

else
    add c to  $C_k$ ;
}
}
}
去除  $C_k$  中不满足最小支持度  $\min\_sup$  的项 1 集;
return  $C_k$ ;
}

```

2.2.4 判断两个项集是否能连接

函数目的：判断两个项集是否可以连接

输入：项集 $l_1 \in L_{k-1}$

输入：项集 $l_2 \in L_{k-1}$

```

Function isCanLink ( $l_1 \in L_{k-1}, l_2 \in L_{k-1}$ ){
    if ( $l_1[1]==l_2[1]$ )&&( $l_1[2]==l_2[2]$ )...( $l_1[k-2]==l_2[k-2]$ )&&( $l_1[k-1]<l_2[k-2]$ )
        return true;
    return false;
}

```

2.2.5 判断 k 项集的某个 k-1 元项是否包含在 k-1 项集中

```

Function has_frequent_subsets (c:k 项集合;  $L_{k-1}$ :频繁 k-1 项集合){
    for c 的每一个 (k-1)子集 s{
        if  $s \notin L_{k-1}$ 
            return true;
    }
    return false;
}

```

2.2.6 Apriori 算法

```

输入：D：事务集；
输入：min_sup:最小支持度阈值
输出：L,D 中的满足要求的频繁项集
Function Apriori(D,min_sup){
     $L_1$ =find_frequent_1_itemSets(D,min_sup);
    for(k=2;  $L_{k-1} \neq null$ ; k++){
         $C_k = \text{apriori\_gen}(L_{k-1}, \min\_sup)$ 
        print( $C_k$ );
    }
}

```

2.3 FP-Growth算法

算法：FP-Growth。使用 FP 树，通过模式增长挖掘频繁模式。

输入：D:事务数据库

输入：min_sup:最小支持度

输出：频繁模式的完全集

2.3.1 FP 树构造方法

按照以下的步骤构造 FP 树：

- 1) 扫描事务数据集 D 一次。收集频繁的集合 F 和它们的支持度计数。对 F 按照支持度计数降序排序，结果为频繁项列表 L；
- 2) 创建 FP 树的根节点，以 null 标记，对于 D 中的每一个事务 T，执行：
选择 T 中的频繁项，并且按照 L 中的次序排序。设 T 排序后的频繁项列表为 [p|P]，其中 p 是第一个元素，而 P 是剩余元素的列表。调用 insert_tree([p|P], T)。
- 3) Insert_tree([p|P], T)
 - a) 如果 T 有子女 N，而且 N.item-name=p.item-name，则 N 的计数+1；
 - b) 否则，创建一个新节点 N，将其计数设置为 1，链接到它的父节点 T，并且通过节点链结构将其链接到具有相同 item-name 的节点。
 - c) 如果 p 非空，则递归调用 insert_tree([p|P], T)；

2.3.2 FP_growth 算法

```
Function FP_growth(Tree,  $\alpha$ ) {
  If (Tree 包含单个路径 P) {
    For(路径 P 中结点的每个组合  $\beta$ )
      产生模式  $\alpha \cup \beta$ ，其支持度计数 support_count 等于  $\beta$  中结点的最小支持度计数
    }
  else {
    for (Tree 的头表中的每个  $\alpha_i$ ) {
      产生一个模式  $\beta = \alpha \cup \alpha_i$ ，其支持度计数 support_count =  $\alpha_i$ .support_count；
      构造  $\beta$  的条件模式基；
      构造  $\beta$  的条件 FP 树  $Tree_\beta$ ；
      if(  $Tree_\beta$  非空)
        FP_growth( $Tree_\beta$ ,  $\beta$ )；
    }
  }
}
```

3 实验过程

3.1 实验数据的获取和存储

3.1.1 数据结构

在对事物数据集的存储中，我使用集合类来存储项集，然后用线性表来存储一个完整的事务：

```
private static List<Set<String>> dataSet;
```

其中线性表的元素是 `Set<String>` 类型，也就是由 `String` 类型构成的集合，然后用一个由集合作为元素构成的线性表作为一个事务集。通过这样的方式就可以把事务集保存下来。

然后就是频繁项集的存储，因为每一个频繁项集的存储都需要把其对应的支持度计数存储下来，所以我采用哈希存储，以项集为主键，以项集的支持度计数为 `value` 的一个 `HashMap`。

3.1.2 文件读取

由于数据集是用 CSV 格式存储的，项集是用花括号包围的，而且项与项之间还有逗号分割。而且 CSV 的每一行开头都是事务标识符，所以我们在读文件时候每次读一行，然后再用 `String` 的 `split` 函数以 “{, }” 为分割符进行分割得到一个数组，然后数组中除了第一个和最后一个之外的元素，就是一个项集：

```
br = new BufferedReader(new FileReader(csv));
while ((line = br.readLine()) != null) { // 读取一行
    dset = new TreeSet<String>();
    data.add(line);
    String[] major = line.split("[},{]"); // 分割字符串
    for (int i = 2; i < major.length - 1; i++) // 获取数据
        dset.add(major[i]);
    dataSet.add(dset); // 将数据加入事务集中
}
```

3.2 算法比较方式

3.2.1 时间复杂度

时间复杂度的比较比较简单，只需要在函数的开头和结尾分别计算时间，然后相减，就知道程序运行了多久时间了。

3.2.2 空间复杂度

空间复杂度的比较只需要在运行时候打开任务管理器看看 Eclipse 占用了多少内存就可以了。

3.2.3 设置不同的支持度和置信度

我把支持度设置为 5,10,50,100 四个等级，然后每个支持度都设置不同等级的置信度，置信度为 50%,60%,70%,80%,90% 还有完全信任。

4 实验结果

由于在关联规则挖掘过程中，时间和空间的开销主要是在寻找频繁项集的过程中，而在关联规则的产生过程中的开销远远小于寻找频繁项集的过程。所有时间复杂度和空间复杂度的比较我主要用在第一步上。



4.1 时间复杂度和空间复杂度的比较

4.1.1 Sup=10

FP-Growth butter,whipped/sour cream null null white bread 10 fruit/vegetable juice null null null whipped/sour cream 36 meat hard cheese 14 newspapers null null berries 22 misc. beverages null null pickled vegetables 10 找出频繁项完毕-----! 共用时: 24586ms	Apriori 频繁4项集: 共3137项: {[UHT-milk,bottled water,other vegetables,s 频繁5项集: 共3761项: {[beef,butter,other vegetables,root vegetables 频繁6项集: 共10项: {[beef,other vegetables,rol 频繁7项集: 共0项: {} 共用时: 151509ms
 Java(TM) Platform SE binary 61.6% 148.2 MB	 Java(TM) Platform SE binary 28.3% 292.5 MB

4.1.2 Sup=50

FP-Growth	Apriori
-----------	---------

null brown bread 101 pastry margarine 65 margarine beef 59 newspapers null napkins 52 找出频繁项完毕-----! 共用时: 1406ms	频繁2项集: 共605项: {[UHT-milk,bottled water.], [UHT-milk, 频繁3项集: 共264项: {[beef,other vegetables,rolls/buns.], [be 频繁4项集: 共12项: {[citrus fruit,other ve 频繁5项集: 共0项: {} 共用时: 12322ms
 Java(TM) Platform SE binary 18.8% 38.7 MB	 Java(TM) Platform SE binary 38.0% 153.7 MB

4.1.3 Sup=70

FP-Growth pastry null whipped/sour cream 72 whole milk frozen meals 97 yogurt waffles 74 null long life bakery product 89 找出频繁项完毕-----! 共用时: 1617ms	Apriori 频繁3项集: 共103项: {[beef,other vegetables, 频繁4项集: 共2项: {[other vegetables,root ve 频繁5项集: 共0项: {} 共用时: 11484ms
 Java(TM) Platform SE binary 31.5% 313.6 MB	 Java(TM) Platform SE binary 41.6% 64.8 MB

4.1.4 Sup=5

FP-Growth java.lang.OutOfMemoryError:	Apriori 频繁5项集: 共5988项: {[UHT-milk,bottled water,other vegetables,rolls/buns,soda.], 频繁6项集: 共785项: {[beef,bottled beer,other vegetables,rolls/buns,root vegetable 频繁7项集: 共38项: {[beef,citrus fruit,other vegetables, 频繁8项集: 共0项: {} 共用时: 940084ms
	 Java(TM) Platform SE binary 29.6% 305.2 MB

4.2 实验结果分析

从 4.1 的实验结果上来看, 我可以得出这样结论:

- 1) FP-Growth 算法确实能够极大程度的提升时间复杂度, 而且时间复杂度是和 Apriori 算法不是一个数量级的;
- 2) 但是 FP-Growth 算法时间复杂度的提升是靠空间换取时间, 比如在 4.1 中, 当最小支持度设置为 5 时, FP-Growth 算法在我电脑上会提示内存不足, 在最小支持度的设置的小的时候 FP-growth 占用的内存是 Apriori 的好几倍。但是随着支持度阈值的提升, FP-growth 算法的空间复杂度的下降也会变得十分明显。
- 3) 对于挖掘长的频繁模式和短的频繁模式, FP-Growth 都是有效的和可伸缩的, 并且比 Apriori 快了差不多一个数量级。
- 4) 从内存使用上来看, 当数据库很大时, 构造基于主存的 FP 树有时是不现实的。

4.3 结论

通过这次的实验, 我得出这样的结论:

- 1) 对于 Apriori 算法, 剪枝和先验知识的引进, 比用死算能节省很多的空间和时间开销; 但是它仍然可能产生大量的候选项集, 还可能需要重复的扫描整个数据库。
- 2) FP-Growth 算法对于频繁模式的挖掘是有效的和可伸缩的, 并且在时间复杂度方面比 Apriori 算法快了差不多一个数量级。
- 3) FP-Growth 是一种空间换时间的算法, 当数据库很大时, FP-growth 算法有时是不现实的。为了防止内存不足, 有时候还是需要用 Apriori 算法慢慢计算。

5 实验总结

通过这次实验, 彻底掌握了 Apriori 算法和 FP-Growth 算法, 并且懂了怎么进行关联规则的挖掘,

同时对这两个算法的理解，包括时间复杂度和空间复杂度的理解都有了很大的提升，让自己的大数据处理方面的能力有了进一步的提升。