

《数据仓库与数据挖掘技术》项目报告  
Apriori 算法和 FP-growth 算法的实现和性能  
对比

姓名：沈程

学号：1400012908

`datouhou@pku.edu.cn`

院系：信息科学技术学院智能系

2017 年 6 月 15 日

# 目录

<b>第一章 算法特点与描述 (For project 3 &amp; 4)</b>	<b>1</b>
1.1 算法特点 . . . . .	1
1.2 算法描述 . . . . .	1
1.2.1 基本概念 . . . . .	1
1.2.2 Apriori 算法 . . . . .	2
1.2.3 FP-growth 算法 . . . . .	5
<b>第二章 实验</b>	<b>10</b>
2.1 程序运行环境和操作说明 . . . . .	10
2.1.1 运行环境 . . . . .	10
2.2 运行结果 . . . . .	10
2.2.1 运行结果 . . . . .	10
2.2.2 程序性能 . . . . .	12

## 摘要

本次项目选择了 Project4, 实现了两种频繁模式挖掘算法并做了性能上的对比分析, 实验数据使用的是 Groceries.csv。首先实现了 Apriori 算法, 在此基础上利用 FP-tree 实现了其改进版 **FP-growth** 算法。在实现 FP-growth 的过程中, 专门在一些内存调用的部分做了优化。同时在实验最后采用了大数据集进行了程序的抗压测试。整个项目可以在我个人的 **GitHub** 页面 <https://github.com/datouhou/FP-algorithm> 上看到, 说明文档使用 **LaTeX** 编写

# 第一章 算法特点与描述 (For project 3 & 4)

## 1.1 算法特点

**Apriori 算法：**该算法是很经典的数据挖掘算法，主要用于通过频繁项集挖掘关联规则。算法主要分为两个部分，通过逐层搜索的迭代方法找寻频繁项集和通过频繁项集规则挖掘关联规则。算法的优点正在于，关联规则的产生是在频繁项集的基础上产生的，所以可以保证规则左右项的最小支持度都满足设定要求，具有很高的普遍性和可信度。而 Apriori 的缺点在于：需要多次全扫描数据库，查找计数工作量大，耗费大量时间，要生成大量候选集，对空间也有很高要求。

**FP-growth 算法：**该算法主要源自与 Apriori 算法，主要是用于发现频繁项集，相较于 Apriori 有了极大的优化。主要是利用 FP-tree 结构对数据进行压缩，之后采用自底向上的顺序对树中节点递归进行条件 FP-tree 的构建，得到频繁项集。构建过程只需要对数据库进行两次扫描，比 Apriori 要快多个数量级，而且不需要生成大量候选项，在空间上也有所优化。但是，当生成的 FP-tree 较为茂盛甚至是完全树的情况下，对与子问题的合并需要占用大量时间，算法性能会出现明显降低。

## 1.2 算法描述

### 1.2.1 基本概念

在具体算法讲述之前，需要先定义一些关于频繁模式的基本概念：

(1)项:每条数据记录中的元素，本次项目里代指商品

(2)项集：若干项的集合

(3)k-项集：包含 k 个项的项集

(4)最小支持度，最小置信度：均由用户指定，分别决定了频繁项集的普遍程度和关联规则的可信度

(5)频繁项集：项集在各个事务中出现的次数大于最小支持数（最小支持数和事务总数的乘积）即为频繁项集

(6)强关联规则：既满足最小支持度又满足最小置信度的规则成为强关联规则，也正是频繁模式挖掘的最终目标

### 1.2.2 Apriori 算法

Apriori 算法总共包括两个部分，分别是发现频繁项集，找出关联规则，下面将依次介绍

#### 发现频繁项集

发现频繁项集的过程其核心在于逐层搜索进行迭代，也就是利用 K-1 项集迭代计算 K 项集。首先要进行的是扫描全数据，对每一件商品进行计数，将其中数量大于最小支持数的商品挑选出来，产生“1 阶频繁项集”，将其保存在 L1 中。

---

#### Algorithm 1 发现 1 阶频繁项集

---

```

1: for each line ∈ database do
2:   for each goods ∈ line do
3:     number[goods] ++
4:   end for
5: end for
6: for each goods do
7:   if number[goods] > minsupport then
8:     L1.insert(goods)
9:   end if
10: end for

```

---

在产生了“1 阶频繁项集”之后，要开始进行迭代，发现高阶的频繁项集，迭代过程包括 (1) 连接组合产生候选集 (2) 剪枝 (3) 计数产生频繁项集，下面分别介绍

**连接组合产生候选集：**连接的元素从  $k-1$  阶频繁项集中选取，而且进行连接的两个项集的前缀要相同，要将最后一位的不同元素整合到一起即可，产生候选集，这是因为频繁项集的子集一定也是频繁项集，这种选取连接的方式满足了这一条件的必要性，可以节省时间。

---

**Algorithm 2** 发现  $k$  阶候选集

---

```

1: for each  $x \in L_{k-1}$  do
2:   for each  $y \in L_{k-1}$  do
3:     if  $x \neq y$  then
4:       if ( then( $x[1] == y[1]$ )&&( $x[2] == y[2]$ )&&...&&( $x[k-2] == y[k-2]$ )&&( $x[k-1] \neq y[k-1]$ ) )
5:          $C_k.insert(x + y[k-1])$ 
6:       end if
7:     end if
8:   end for
9: end for

```

---

**剪枝：**候选集剪枝的规则主要也是依据上文提到的，频繁项集的子集一定是频繁项集，所以剪枝的主要过程实际上就是检查当前候选项集的子集中是否存在非频繁项集，如果存在，直接将当前的候选项集删去

---

**Algorithm 3** 剪枝

---

```

1: for each  $x \in C_k$  do
2:   for each  $subset \in x$  do
3:      $n = subset.size()$ 
4:     if  $subset \notin L_n$  then
5:        $C_k.delete(x)$ 
6:       Break
7:     end if
8:   end for
9: end for

```

---

**计数产生频繁项集：**对数据库进行扫描，对经过剪枝的候选项集进行计数统计，在所有事务中总共出现了多少次，将其中数量大于最小支持度的候选项集保存作为  $k$  阶频繁项集

**Algorithm 4** 计数产生频繁项集

---

```

1: for each  $x \in Ck$  do
2:   for each  $item \in database$  do
3:     if  $x \subset item$  then
4:        $num[x]++$ 
5:     end if
6:   end for
7:   if  $num[x] \geq minsupport$  then
8:      $Lk.insert(x)$ 
9:   end if
10: end for

```

---

上面详细介绍了一次迭代的过程，而整个频繁项集的发现，都是一直循环这个过程，直到已经找不到一个更高阶的频繁项集为止

**Algorithm 5** 迭代发现频繁项集

---

```

1:  $L1 = find_1item(database)$ 
2:  $k = 1$ 
3: while ( $doLk \neq empty$ )
4:    $k = k + 1$ 
5:    $Lk = find_kitem(database, Lk - 1)$ 
6: end while

```

---

**找出关联规则**

找出关键规则是在之前发现的频繁项集的基础上进行的。在此之前，我们先看一下置信度如何计算的，现在假设有两个项集 A 和 B，那么  $A \Rightarrow B$  的置信度的计算公式如下

$$P(B|A) = \frac{P(AB)}{P(A)} \quad (1.1)$$

在清楚了计算公式之后，可以开始进行强关联模式的挖掘。拿来之前找到的最高阶的频繁项集，循环找出频繁项集中的每一个子集作为关联关系

中左侧的项集，子集的补集作为关联关系中右侧的项集，分别找出每组关联关系的置信度，与最小置信度进行比较，得到强关联关系

---

**Algorithm 6** 找出强关联关系
 

---

```

1: for each  $x \in L_{max}$  do
2:   for each  $subset \in x$  do
3:      $comset = x - subset$ 
4:      $conf = (\frac{P(comset, subset)}{P(subset)})$ 
5:     if  $conf \geq minconf$  then
6:        $res.insert(sub \Rightarrow com)$ 
7:     end if
8:   end for
9: end for

```

---

### 1.2.3 FP-growth 算法

FP-growth 算法主要是用于对频繁项集查找的优化，通过前面的介绍可以发现，在生成频繁项集的时候，Apriori 每次在从候选集中进行筛选的时候，都需要进行数据库扫描。同时大量的候选集也需要很大消耗很大的空间。针对这两个方面，FP-growth 都做了优化，仅需要在构建 headtable 和初始 FP-tree 的时候进行两次数据库的扫描，而且不需要生成大量的候选集。在介绍算法之前，需要首先介绍一下 FP-tree，这一数据结构被用于 FP-growth 中对数据的压缩，起到至关重要的作用。

#### FP-tree

FP-tree 的树节点结构：如代码

---

**Algorithm 7** FP-tree 节点结构
 

---

```

FP - treenode{
    node * parent
    node * child[]
    name
    num
}

```

---



parent 是指向树节点父节点的指针, child 指向树节点的子节点空间, name 和 num 分别是当前树节点所代表的商品的名称和数量。

**FP-tree 构建:** 构建 FP-tree 首先设立一个空节点作为树的根节点, 之后逐行扫描当前数据库, 每次扫描到新的一行, 都要从根节点开始构建 FP-tree, 设立观察节点为根节点, 遍历当前交易中的商品, 对于遍历到的商品, 判断其是否在观察节点的子节点集中, 如果存在, 则将对应的子节点的数量加 1, 如果不存在, 则给观察节点创建新的子节点, 数量赋值为 1。之后将该子节点设立为当前的观察节点, 遍历下一个商品。

---

**Algorithm 8** FP-tree 构建

---

```

1: for each item  $\in$  database do
2:   root = null
3:   for each goods  $\in$  item do
4:     if goods  $\in$  root.child[] then
5:       root.child[goods].num ++
6:     else
7:       root.child[].insert(goods, 1)
8:     end if
9:     root = root.child[goods]
10:  end for
11: end for

```

---

**FP-growth 自增长**

介绍过了 FP-tree, 开始正式自增长部分的算法。FP-growth 主要通过递归实现子问题的增长。

**构建条件 FP-tree:** 首先需要扫描一遍当前数据库, 要将所有数据库中满足最小支持数的商品的名称以及对应的数量按大小顺序保存在一个 headtable 中。之后根据 headtable, 对当前数据库的每一次交易的项进行排序, 这是为了方便之后构建树的时候, 数量小的节点能够在下层, 方便统一从底向上搜索。之后开始按上面的方法进行 FP-tree 的构建

---

**Algorithm 9** 构建条件 FP-tree

---

```

1: function BUILD TREE(database)
2:   for each item  $\in$  database do
3:     for each goods  $\in$  item do
4:       num[goods] ++
5:     end for
6:   end for
7:   for ( do each goods)
8:     if num[goods]  $\geq$  minsupport then
9:       headtable.insert(goods, num[goods])
10:    end if
11:  end for
12:  sort(database) according to headtable
13:  FP-tree(database)
14: end function

```

---

**输出频繁项：**将当前 headtable 中的所有商品依次输出，每次输出都要加上当前的前缀。

---

**Algorithm 10** 输出频繁项集

---

```

1: function OUTPUT(database, post)
2:   for each goods  $\in$  headtable do
3:     out << goods
4:     for each goods  $\in$  headtable do
5:       out << goods
6:     end for
7:     out << num[goods]
8:   end for
9: end function

```

---

**构建新数据集：**从 headtable 的末尾开始，从小到大，依次选定为前缀，每个前缀都向上查找到所有的家长节点，每一条路径都作为一个交易添加到新的数据集中，数量为前缀节点的数量。最终将新的数据集作为输入，递归构造条件 FP-tree。要注意的是，如果当前的增长函数中，前缀参

数不为空, 要将两次的前缀进行合并, 传入下一次递归。如果在增长过程中, 出现条件 FP-tree 仅剩一个节点, 说明只有空的根节点存在, 直接退出当前递归即可。

---

**Algorithm 11** FP-growth
 

---

```

1: function FP-GROWTH(database, post)
2:   Buildtree(database)
3:   if  $FP-tree.nodenum == 1$  then
4:     return
5:   end if
6:   ouput(database, post)
7:   for each  $goods \in rev-headtable$  do
8:      $new-post = merge(goods, post)$ 
9:     for ( do each  $path$  such that  $path.leaf == goods$  )
10:       $new-database.insert(goods.path())$ 
11:    end for
12:     $FP-growth(new-database, new-post)$ 
13:  end for
14: end function

```

---

**单路径优化:** 这里考虑到空间代价的问题, 单路径上只做了部分优化, 在构建新的数据集的时候, 如果所有的数据节点都在一条路径上, 那么久放弃递归, 只要叶节点的数量大于最小支持数, 直接将该条路径上所有项的排列组合输出, 作为频繁项集。

---

**Algorithm 12** FP-growth

---

```

1: function ONE-PATH(database,post)
2:   .....
3:   for each goods  $\in$  rev - headtable do
4:     new - post = merge(goods, post)
5:     for ( do each path such that path.leaf == goods)
6:       new - database.insert(goods.path())
7:     end for
8:     if pathnum == 1 then
9:       out << all subset of path
10:    else
11:      FP - growth(new - database, new - post)
12:    end if
13:  end for
14: end function

```

---

## 第二章 实验

### 2.1 程序运行环境和操作说明

#### 2.1.1 运行环境

操作系统

win10 64bit

语言编译环境

使用 c++ 语言编写，所提交文件包括两个 cpp 文件:Apriori.cpp 和 FP-growth.cpp，分别对应两种算法的实现，数据库文件也一起打包，相对目录已经设置好，可以直接编译运行。

特殊情况解释：在完成项目期间，试着在本机上安装 VS2013，2015，2017 三个版本，运行 c++ 程序均不能够成功响应。所以，项目代码均是在 dev c++ 中实现的，编译也都是在这个 IDE 下进行的，这一点在助教师兄检查的时候，还需要麻烦您了。

### 2.2 运行结果

#### 2.2.1 运行结果

两个算法的运行结果都直接输出到相应目录的 **out.txt**文档当中。

Apriori 算法输出找到的强关联规则，以  $A \Rightarrow B$  的形式表示出来，每一行的末尾还会输出一个浮点数，表示对应的置信度。图 2.1

FP-growth 算法输出发现的频繁项集，每一行输出一组频繁项集，在末尾还会输出一个整数，表示该项集的支持数。图 2.2

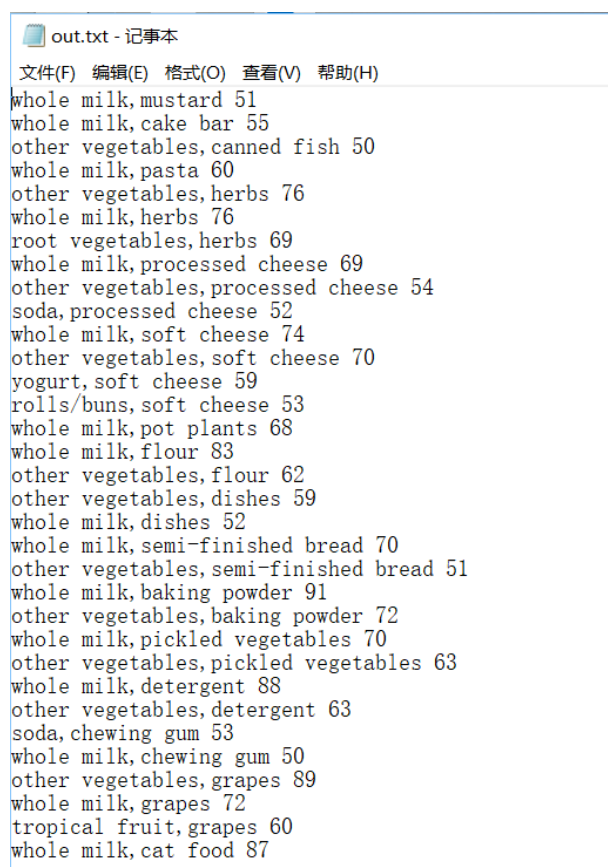
out.txt - 记事本  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```

other vegetables root vegetables whole milk ==>citrus fruit 0.25
citrus fruit root vegetables whole milk ==>other vegetables 0.633333
root vegetables whole milk ==>citrus fruit other vegetables 0.118503
citrus fruit other vegetables whole milk ==>root vegetables 0.445312
other vegetables whole milk ==>citrus fruit root vegetables 0.0774457
citrus fruit whole milk ==>other vegetables root vegetables 0.19
citrus fruit other vegetables root vegetables ==>whole milk 0.558824
other vegetables root vegetables ==>citrus fruit whole milk 0.122318
citrus fruit root vegetables ==>other vegetables whole milk 0.327586
root vegetables ==>citrus fruit other vegetables whole milk 0.0531716
citrus fruit other vegetables ==>root vegetables whole milk 0.200704
citrus fruit ==>other vegetables root vegetables whole milk 0.0700246
other vegetables whole milk yogurt ==>fruit/vegetable juice 0.228311
fruit/vegetable juice whole milk yogurt ==>other vegetables 0.537634
whole milk yogurt ==>fruit/vegetable juice other vegetables 0.0907441
fruit/vegetable juice other vegetables yogurt ==>whole milk 0.617284
other vegetables yogurt ==>fruit/vegetable juice whole milk 0.117096
fruit/vegetable juice yogurt ==>other vegetables whole milk 0.271739
fruit/vegetable juice other vegetables whole milk ==>yogurt 0.485437
other vegetables whole milk ==>fruit/vegetable juice yogurt 0.0679348
fruit/vegetable juice whole milk ==>other vegetables yogurt 0.19084
fruit/vegetable juice other vegetables ==>whole milk yogurt 0.241546
fruit/vegetable juice ==>other vegetables whole milk yogurt 0.0703235
pip fruit root vegetables whole milk ==>other vegetables 0.613636
other vegetables root vegetables whole milk ==>pip fruit 0.236842
root vegetables whole milk ==>other vegetables pip fruit 0.112266
other vegetables pip fruit whole milk ==>root vegetables 0.406015
pip fruit whole milk ==>other vegetables root vegetables 0.182432
other vegetables whole milk ==>pip fruit root vegetables 0.0733696
other vegetables pip fruit root vegetables ==>whole milk 0.675
pip fruit root vegetables ==>other vegetables whole milk 0.352941
other vegetables root vegetables ==>pip fruit whole milk 0.11588
root vegetables ==>other vegetables pip fruit whole milk 0.0503731
other vegetables pip fruit ==>root vegetables whole milk 0.210117
pip fruit ==>other vegetables root vegetables whole milk 0.0725806
pip fruit whole milk yogurt ==>other vegetables 0.531915
other vegetables whole milk yogurt ==>pip fruit 0.228311
whole milk yogurt ==>other vegetables pip fruit 0.0907441
other vegetables pip fruit yogurt ==>whole milk 0.625

```

图 2.1: Apriori 的运行输出文档截图



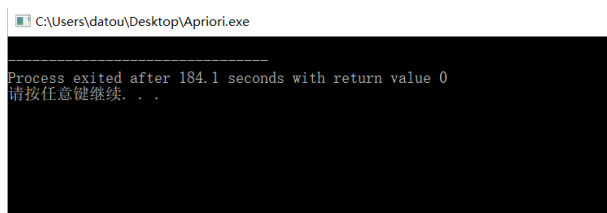
```
out.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
whole milk, mustard 51
whole milk, cake bar 55
other vegetables, canned fish 50
whole milk, pasta 60
other vegetables, herbs 76
whole milk, herbs 76
root vegetables, herbs 69
whole milk, processed cheese 69
other vegetables, processed cheese 54
soda, processed cheese 52
whole milk, soft cheese 74
other vegetables, soft cheese 70
yogurt, soft cheese 59
rolls/buns, soft cheese 53
whole milk, pot plants 68
whole milk, flour 83
other vegetables, flour 62
other vegetables, dishes 59
whole milk, dishes 52
whole milk, semi-finished bread 70
other vegetables, semi-finished bread 51
whole milk, baking powder 91
other vegetables, baking powder 72
whole milk, pickled vegetables 70
other vegetables, pickled vegetables 63
whole milk, detergent 88
other vegetables, detergent 63
soda, chewing gum 53
whole milk, chewing gum 50
other vegetables, grapes 89
whole milk, grapes 72
tropical fruit, grapes 60
whole milk, cat food 87
```

图 2.2: FP-growth 算法的输出稳定截图

### 2.2.2 程序性能

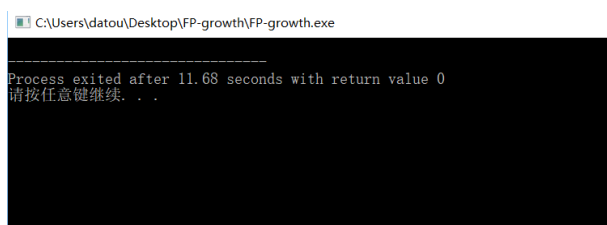
程序运行配置为: Intel(R)Core(TM)i7-6700HQ CPU@2.60GHz 8.00GB RAM

两个算法使用相同的数据集, 最小支持数为 50, 最小置信度为 0.05, 运行时间如图 2.3 图 2.4。可以看到 FP-growth 相较于 Apriori 算法在时间上有了很高程度的优化。这里要解释一下的是, 因为使用的 IDE 为 Dev c++, 对栈空间限制很大, 所以在算法实现的过程中, 很多地方为了减少内存调用, 选择了耗费时间的做法, 所以这两个算法的时间值都比较长。不过通过相对值还是能看出性能差异很大。



```
C:\Users\datou\Desktop\Apriori.exe
-----
Process exited after 184.1 seconds with return value 0
请按任意键继续. . .
```

图 2.3: Apriori 运行时间



```
C:\Users\datou\Desktop\FP-growth\FP-growth.exe
-----
Process exited after 11.68 seconds with return value 0
请按任意键继续. . .
```

图 2.4: FP-growth 算法运行时间

另外经过反复测试, 当最小支持数较大的时候, 数据剪枝较快, 同时 FP-tree 较为稀疏, 时间能够大幅缩进。图 2.5 图 2.6



```
C:\Users\datou\Desktop\新建文件夹\Apriori\Apriori.exe
请指定最小支持数 (建议50左右)
300
请指定最小置信度 (建议0.1左右)
0.2
-----
Process exited after 19.2 seconds with return value 0
请按任意键继续. . .
```

图 2.5: Apriori 算法运行时间





图 2.6: FP-growth 算法运行时间

## 网络资料

- Apriori 算法详解之【一、相关概念和核心步骤】<http://blog.csdn.net/lizhengnanhua/article/details/9061755>
- Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach [http://hanj.cs.illinois.edu/pdf/dami04\\_fptree.pdf](http://hanj.cs.illinois.edu/pdf/dami04_fptree.pdf)