Association Rule Mining

王森 MG21370034

2021 年 11 月 7 日

1 实验说明

1.1 任务

本次实验的主要任务是对给定数据集进行关联规则挖掘。通过改变 support 和 confidence 的值，比较 Aprioir、FP-growth 和最基本的穷举法之间的差别，在进行比较时，主要着眼于产生的频繁项集的数目，算法运行过程中的内存消耗和时间消耗。

1.2 数据集

本次实验的数据集总共有两个，分别是 GroceryStore 和 UNIX_usage 。

GroceryStore 数据集包含某商店一个月内的交易记录，数据存储的格式为 csv，每一行是一条交易信息，购买的商品名称被“{”和“}”包围着。此数据集总共有 9835 条交易记录，涉及 169 种商品。

UNIX_usage 数据集总共包含 9 个文件，每个文件都是真实用户的命令行输入历史，其中第 0 个文件和第 1 个文件是同一用户在不同平台和项目中产生的命令行记录，其余的 7 个文件都是来自不同用户的命令行历史记录。出于保护隐私的考虑，移除了原始文件中的文件名、用户名、文件结构等所有可能暴漏个人信息的部分，而是用 <1> 来代替此部分。在一个 session 的开始和结束会包含 **SOF** 和 **EOF** ，而之前在命令行中用空格分隔的参数部分，在此处会单独占据一行。下面是两个 session 的原始记录。

```
# Start session 1
cd ~/private/docs
ls -laF | more
cat foo.txt bar.txt zorch.txt > somewhere
exit
# End session 1

# Start session 2
cd ~/games/
xquake &
fg
vi scores.txt
mailx john_doe@somewhere.com
exit
# End session 2
```

在我们拿到的数据中，这两个 session 用如下形式来表示。

```
**SOF**
cd
<1>       # one "file name" argument
ls
-laF
|
more
cat
<3>       # three "file" arguments
>
<1>
exit
**EOF**
**SOF**
cd
<1>
xquake
```

```
&
fg
vi
<1>
mailx
<1>
exit
**EOF**
```

2 代码实现

2.1 Apriori

Apriori 算法的核心是对先验知识的运用，具体来说是利用“频繁项集的所有非空子集也一定是频繁的”这条规则来减少搜索空间。我们利用 k 频繁项集的集合 L_k 生成 $k+1$ 频繁项集的集合 L_{k+1}，在生成的过程中，会有一个中间步骤，即 $k+1$ 频繁项集的候选集合 C_{k+1}。按照常规的做法，此时应该扫描一遍数据库，判断 C_{k+1} 中的哪个项集是频繁的，哪个是非频繁的。但是我们现在考虑一下刚才提到的先验知识，如果 C_{k+1} 集合中的某个 $k+1$ 项集 c 是频繁的，那么它的所有子集中长度为 k 的集合，即 k 项集，也一定是频繁的。所有的 k 频繁项集都在 L_k 中，因此可以通过判断 c 的子集中的 k 项集是否出现在 L_k 中来减少搜索空间。此算法的伪代码如 Algotithm1 所示。

整个算法在实现的时候，由四个函数构成。`apriori` 为主函数，`find_frequent_1_itemset` 用于从原始数据库中寻找 1 频繁项集，`apriori_gen` 用于从 k 频繁项集中生成 $k+1$ 候选频繁项集，上文中提到过，生成的过程中会进行剪枝，而 `has_infrequent_subset` 函数就是用于判断是否满足剪枝条件的。对候选集初步剪枝之后，需要去扫描一遍数据库，以找出那些真正的 $k+1$ 项集，此操作由 `subset` 函数辅助完成。

对于计算出来的每个 k 频繁项集，都是用 `list` 对象进行存储。比如

$$[item_1, item_2, ..., item_k, \text{frequency}]$$

此 `list` 对象共包含 $k+1$ 个元素，前 k 个元素对应于 k 频繁项集，最后一个元素是此 k 频繁项集在数据库中出现的次数。

Algorithm 1 Apriori

Input: DB: database of transactions; min_sup: minimum support threshold; $itemset$: all the items occured in DB

Output: L: frequent itemsets in DB

```
// 找到所有的频繁 1 项集
L_k = find_frequent_1_itemsets(DB, itemset, min_sup)
while len(L_k) > 0 do
    L.extend(L_k)
    // 由 k 频繁集生成 k+1 候选集
    C_{k+1} = apriori_gen(L_k)
    Initialize c_count with 0
    for t in DB do
        c_count = subset(C_{k+1}, t, c_count)
    end for
    L_{k+1} = {c ∈ C_{k+1} | c_count[c] ≥ min_sup}
    L_k = L_{k+1}
end while
function apriori_gen(L_k)
    size = len(L_k)
    for i = 0; i < size; i++ do
        for j = i + 1; j < size; j++ do
            if L_k[i][0 : −2] == L_k[j][0 : −2] then
                c = L_k[i] ∪ L_k[j]
                if not has_infrequent_subset(c, L_k) then
                    C_{k+1}.append(c)
                end if
            end if
        end for
    end for
    return C_{k+1}
end function
function has_infrequent_subset(c, L_k)
    for each k − subset of c do
        if s ∉ L_k then
            return True
        end if
    end for
    return False
end function
```

2.2 Dummy

dummy 方法是一种穷尽搜索的方法，虽然使用起来效率低，但是实现较为简单。算法伪代码如 Algorithm2 所示。

Algorithm 2 Dummy

Input: DB: database of transactions; min_sup: minimum support threshold; $itemset$: all the items occured in DB

Output: L: frequent itemsets in DB

```
for k = 1; k ≤ len(itemsets); k++ do
    // 生成 itemset 的所有含有 k 个元素的子集
    C_k = generate_k_subset(itemset)
    // c_count 是一个和 C_k 等长的列表
    Initialize c_count with 0
    for t in DB do
        c_count = subset(C_k, t, c_count)
    end for
    L_k = {c ∈ C_k | c_count[c] ≥ min_sup}
    if L_k is empty then
        break
    end if
    L = ∪L_k
end for
return L
```

穷尽搜索，即尝试每一种可能的项集组合。先由 $itemset$ 生成所有只含一个元素的子集，即 1 频繁候选项集，然后扫描数据库，找出 1 频繁项集；再由 $itemset$ 生成所有只含两个元素的子集，即 2 频繁候选项集，然后扫描数据库，找出 2 频繁项集。依此类推，直到不能从 k 频繁候选项集中找出 k 频繁项集时结束。

2.3 FP-growth

FP-growth 方法比 Apriori 方法在空间消耗和时间消耗上都有所提升，尤其是当设定的支持度比较低时，这种提升尤为明显。

首先介绍一下 FP-Tree 的结构。FP-Tree 是一棵索引树，每个节点包含五个字段，`name` 字段表示项目的名称，`frequency` 字段表示此项目的频数，`parent` 字段表示此项

目的父节点索引， children 字段表示此项目的子节点索引，因为一个项目可能有多个子节点，所以 children 字段为 list 类型， nextit 字段表示下一个和此项目同名的节点索引号。

在 FPTree 这个类中，除了有 nodes 字段用来存储树的节点外，还有 headtables 字段用来存储每个项目的头节点索引。 headtables 是一个 dict 类型，每个元素的 key 为项目名称，value 是此项目第一次在 FPTree 中出现时的索引。 headtables 相当于一个头指针表，通过它，可以访问到某个项目在 FPTree 中的所有节点。

FP-growth 算法的伪代码如 Algorithm3 所示。

Algorithm 3 FP-growth

Input: DB: database of transactions; min_sup: minimum support threshold; $itemset$: all the items occured in DB

Output: L: frequent itemsets in DB

1: // 找出所有的频繁 1 项集
2: $tree\_items$ = find_frequent_1_itemsets($database$, $itemsets$, min_sup)
3: // 将这些频繁 1 项集按照出现的次数逆序排序
4: $tree\_items$.sort()
5: // 再扫描一遍数据库，将每条交易中的 item 按照 $tree\_items$ 中的顺序排序，并移除没有出现在 $tree\_items$ 中的 item
6: $database$ = sort_frequent_item($database$, $tree\_items$)
7: // 创建 FP 树
8: $fptree$.create_tree($database$, $False$)
9: // 由 fp 树生成频繁项集，结果存放在 $freq\_list$ 中
10: generate_fre_itemset($fptree$, min_sup, [100000], $freq\_list$, $tree\_items$)
11: return fre_list

按照 FP-growth 算法，首先需要扫描一遍数据库，找出所有的频繁 1 项集，然后再按照每个项集出现的次数逆序排序。之后再扫描一遍数据库，此时需要对数据库中的数据做一些修改，首先将那些没有出现在频繁 1 项集中的 item 从数据库中移除，再按照频繁 1 项集中每个 item 的顺序，对数据库中的每条 transaction 进行排序。上述操作完成之后，就可以用数据库中的数据来构建 FP-Tree 了。构建树的过程，其实就是把一个个结点插入到树中的过程。为了表述方便，假设现在我们要把一条包含 n 个 item 的交易记录 t 插入到 FP 树中，因为 t 有 n 个 item，所以相当于是要插入 n 个结点。为了找到插入的位置，我们首先需要对现在的 FP 树进行递归搜索。搜索过程从根节点开始，

即最初把根节点当作当前结点。如果根节点存在与 $t[0]$(交易记录中的第一个 item) 相同的子节点 n_0，则递归的去搜索 n_0 是否存在与 $t[1]$ 相同的子节点，依此类推，直到找不到相同的子节点或 t 中的所有 item 在 FP 树中都有相同的结点。找到插入的位置之后，插入操作相对比较简单，只是需要注意一下，在插入结点的同时要构建 $headtables$。

到目前为止，关于数据库的 FP 树已经构建出来了，接下来要做的是根据 FP 树生成所有的频繁项集。将 FP 树中出现的所有的 item 按照频数升序排序，然后遍历这些 item，遍历过程由 for 循环完成，每当遍历到一个 item 时，将当前的 item 加入到前缀序列中，然后将此前缀序列作为频繁项集加入到总的频繁项集集合 L 中。之后计算当前 item 所对应的条件模式基，若此条件模式基不为空，则由其继续构造 FP 树，然后递归的由此 FP 树生成所有的频繁项集。

利用 FP-growth 方法生成的频繁项集和用 Apriori 方法生成的相比，在表示形式上是相同，都是用 list 存储，前 n-1 个元素表示频繁项集中的 item，最后一个元素表示此频繁项集的频数。

2.4 Association Rule

按照上述的方法，已经把所有的频繁项集都找出来了，接下来要做的就是从这些频繁项集中挖掘关联规则。对于一个频繁项集 fi 来说，我们需要计算出它所有的非空真子集，然后遍历这些集合，对于 fi 的某个真子集 s 来说，如果满足：

$$\frac{support(l)}{support(s)} \geq confidence$$

则输出规则 $s \Rightarrow (l - s)$

3 三、实验方案

在进行实验设计时，主要从四个方面切入。

1. 比较不同的数据集产生的频繁项集的数量有什么特点。

2. 对于相同的数据集，使用不同的频繁项集生成算法，在时间消耗上各自的表现如何。

3. 对于相同的数据集，使用不同的频繁项集生成算法，在空间消耗上各自的表现如何。

4. 寻找一些有趣的关联规则，并对这些规则进行讨论。

对于第一个方面，我们可以变换不同的 support，然后用本次实验提供的 Groceries 和 UNIX_usage 这两个数据集进行实验。

对于第二个方面，我们可以使用同一数据集（我选用的是 Groceries），变换不同的 support，分别使用 dummy、Aprioir、FP-growth 算法进行实验，来查看不同算法上的时间消耗。

对于第三个方面，实验方案与探究时间消耗的方案大致相同，只不过我们用 mprof run 命令代替 python 命令来记录内存使用情况，当程序运行结束之后，会在当前目录下生成 mprofile_xxxxxxxxxxx.data 文件，里面存储了相同时间跨度的时间点上内存的使用情况，使用 mprof list 可以查看文件对应的索引 (0,1,2 ...)，然后通过执行 mprof plot file_index 加载对应文件数据来绘制图形

```
mprof plot 0
```

对于第四个方面，我们可以变换不同的 support 和 confidence，并且根据 lift 来查看生成的关联规则。

4 四、实验结果

4.1 频繁项集的数量

在 support 设置为 0.01 的情况下，在两个数据集上进行频繁项集数量的比较。

对于 Groceries 数据集，不同的频繁项集的数量如表 1 所示。

表 1: Groceries 数据集

k	1	2	3	4
数量	88	213	32	0

对于 UNIX_usage 数据集，不同的频繁项集的数量如表 2 所示

表 2: UNIX_usage 数据集

k	1	2	3	4	5	6	7	8	9	10	11
数量	61	329	781	1086	1004	635	289	91	19	2	0

1 61 2 329 3 781 4 1086 5 1004 6 635 7 289 8 91 9 19 10 2

Groceries 数据集中共有 9835 条 transaction，UNIX_usage 数据集中共有 5058 条 session。从表 1 和表 2 的对比中可以看出，UNIX_usage 数据集最多可以产生 10 频繁项集，而 Groceries 数据集最多只能产生 3 频繁项集。从数量关系上来看，虽然 UNIX_usage 的数据要比 Groceries 少，但是前者的 k 频繁项集的数目要远多于后者。出现这种现象的原因，与两个数据集本身的特性有关。UNIX_usage 数据集是由真实用户的命令行记录构成的，我们在输入 cd <1> 命令后，可能会习惯性的再输入 ls 命令查看当前目录的所有文件。

4.2 频繁项集生成算法的时间性能

用 Groceries 数据进行实验，来评判 dummy、Apriori、FP-growth 这三个频繁项集生成算法的时间性能。进行实验的设备为个人笔记本电脑，重要参数为：2 GHz 4 核 Intel Core i5 处理器，16 GB 3733 MHz LPDDR4X 内存。分别设置 support 为 0.1，0.01 和 0.001，生成所有频繁项集所用的时间如表 3 所示。

dummy 算法进行的是穷尽搜索，假设数据集中总共有 n 种不同的 item，那么总共有 C_n^k 个候选 k 频繁项集，因此要想找到所有的 k 频繁项集，需要扫描数据库 C_n^k 遍。在进行实验时，发现对于三种不同的 support，dummy 算法都无法在 20 分钟内结束，因

表 3: 不同算法的时间性能

算法 \ support	0.1	0.01	0.001
FP-growth	0.03549 s	0.71302 s	1.70202 s
Aprioir	0.05842 s	7.29286 s	172.11253 s
dummy	-	-	-

此表 3 中 dummy 算法的执行时间用 - 代替，表示未测得其执行时间。

Apriori 算法相较于 dummy 算法在时间性能上有了很大的提升。相较于 dummy 的穷尽搜索，Apriori 用了一些剪枝的方法，因此减少了搜索空间，提高了执行效率。根据先验知识，“k 频繁项集的所有非空子集也是频繁的”，我们可以由 k 频繁项集生成候选 (k+1) 频繁项集，相较于 dummy 算法会产生 C_n^{k+1} 个候选 (k+1) 频繁项集，此方法会减少候选频繁项集的数量。除此之外，我们还可以再做一次剪枝操作。如果一个 (k+1) 项集是频繁的，那么它的所有 k 子集也一定是频繁的，也就是说它的所有 k 子集一定在 k 频繁项集的集合里面，假若存在某个 k 子集不满足此条件，那么这个候选 (k+1) 频繁项集也一定不是频繁的。通过这两步剪枝操作，可以尽可能的减少候选频繁项集的数目，从而减少扫描数据库的次数（扫描数据库非常耗时）。

FP-growth 算法和 Apriori 算法的思想不同，它通过“树”这种数据结构，使得在生成频繁项集的时候，只需要扫描两次数据库即可。第一次扫描数据库，找出所有的 1 频繁项集，第二次扫描数据库，构建 FP 树，之后的所有操作，都是针对 FP 树进行的。

FP-growth 算法克服了 Apriori 算法需要多次扫描数据库的缺点，因此当生成的频繁项集较多时，FP-growth 的执行时间要远少于 Apriori。从表 3 中可以看出，当 support 为 0.1 时，FP-growth 算法和 Apriori 算法的执行时间都是毫秒级的，且两者的差异不明显。当 support 减小为 0.01 时，生成的频繁项集的数量会增加，此时就能看出 FFP-growth 算法的优势了，它的时间消耗仅为 Apriori 算法的 $\frac{1}{10}$。当 support 继续减小到 0.001 时，FP-growth 算法的优势表现的更为明显，生成同样的频繁项集，FP-growth 算法的时间消耗仅为 Apriori 算法的 $\frac{1}{100}$，时间性能上提升了两个数量级。

当 support 减小时，生成的频繁项集的数量会增多。对于 Apriori 算法来说，它的时间主要是消耗在频繁扫描数据库上，而对于 FP-growth 算法来说，它的时间主要是消耗在递归构建 FP 树上。因为扫描数据库是一件非常耗时的工作，所以 FP-grow 算法的执行时间要远小于 Apriori。

4.3 频繁项集生成算法的空间性能

与时间性能的实验类似，仍然使用 Groceries 数据集评估不同算法的空间性能。由之前的实验已经知道，dummy 算法即使在 support 较小的情况下也很难在短时间内终止，因此本实验只考虑 Apriori 和 FP-growth 算法的空间性能。对于不同的 support，两种算法占用内存的峰值如表 4 所示。

表 4: 不同算法的空间性能

算法 \ support	0.1	0.01	0.001
FP-growth	15.707 MiB	27.031 MiB	30.688 MiB
Aprioir	15.551 MiB	21.527 MiB	43.516 MiB

从表 4 中可以看出，随着 support 的减小，两种算法所占用的内存都在增加。对于 Apriori 算法，当 support 减小时，需要更多的内存空间来存储候选频繁项集。对于 FP-growth 算法，当 support 减小时，需要更多的内存空间来递归生成 FP 树。从实验结果中可以发现一个有趣的现象，当 support 较大时，FP-growth 算法的空间效率要低于 Apriori，但是当 support 较小时，前者的空间效率却又优于后者。

4.4 挖掘关联规则

4.4.1 挖掘 Groceries

在进行关联规则挖掘时，首先需要选择合适的 support 和 confidence。若参数的值设的太小，则容易产生大量的关联规则，从而很难从中发现有价值的规则。同理，若参数的值设的太大，则可能会忽略掉一些有意义的关联规则。经过多次尝试，发现对于 Groceries 数据，support=0.01，confidence=0.5 可以产生 15 条关联规则。所有的这些规则如表 5 所示。

在表 5 中，除了我们熟悉的 support 和 confidence 之外，还引入了一个新的指标：lift。lift 指标的计算公式为：

$$lift(A \to B) = \frac{p(B|A)}{p(B)} = \frac{confidence}{support}$$

它的含义是：在已知 A 的前提下 B 发生的概率，和没有先验条件时 B 单独发生的概率之间的相对关系。若 lift<1，则说明 A 的发生对 B 的发生起着抑制作用；若 lift=1，则说明 A 的发生对 B 是否发生没有任何影响，即 A 和 B 相互独立；若 lift>1，则说明 A 的发生对 B 的发生起着促进作用。

表 5: Groceries 的规则

Rule	support	confidence	lift
yogurt, curd ⇒ whole milk	0.0101	0.5824	57.8529
root vegetables, citrus fruit ⇒ other vegetables	0.0104	0.5862	56.5230
other vegetables, butter ⇒ whole milk	0.0115	0.5736	49.9239
yogurt, whipped/sour cream ⇒ whole milk	0.0109	0.5245	48.2108
root vegetables, tropical fruit ⇒ other vegetables	0.0123	0.5845	47.5121
root vegetables, tropical fruit ⇒ whole milk	0.0120	0.5700	47.5121
other vegetables, domestic eggs ⇒ whole milk	0.0123	0.5525	44.9087
rolls/buns, root vegetables ⇒ other vegetables	0.0122	0.5021	41.1506
rolls/buns, root vegetables ⇒ whole milk	0.0127	0.5230	41.1506
yogurt, root vegetables ⇒ other vegetables	0.0129	0.5000	38.7205
yogurt, root vegetables ⇒ whole milk	0.0145	0.5630	38.7205
other vegetables, pip fruit ⇒ whole milk	0.0135	0.5175	38.2685
other vegetables, whipped/sour cream ⇒ whole milk	0.0146	0.5070	34.6303
yogurt, tropical fruit ⇒ whole milk	0.0151	0.5174	34.1493
other vegetables, yogurt ⇒ whole milk	0.0223	0.5129	23.0328

从表 5 中可以看出，每一条强关联规则的 lift 都大于 1，且都大于 20，这说明这些规则的关联性较强。在后续进行商品布局时，可以参考上述规则设计每种商品的摆放位置。比如说将 yogurt、curd 和 whole milk 摆放在一起，根据过往的统计规律，当顾客购买了 yogurt 和 curd 之后，有 58.24% 的概率会购买 whole milk。

观察挖掘出来的所有关联规则，可以发现规则的左部出现最频繁的是 root/other vegetables，规则的右部出现最频繁的是 whole milk，但是强关联规则中却没有 vegetables—>whole milk 这条规则。另一个有趣的现象是，挖掘出来的这些规则都是来自 3 频繁项集，由表 1 可知，当 support 为 0.01 时，有 221 个 2 频繁项集，有 32 个 3 频繁项集，也就是说 2 频繁项集的数目要远多于 3 频繁项集，但是却不能从 2 频繁项集中挖掘出强关联规则。

4.4.2 **挖掘** UNIX_usage

在 1.2 节中，我们对 UNIX_usage 数据集进行了介绍。每个 session 相当于 Groceries 数据集的一个 transaction，但是每行数据却不能等同于 Groceries 中的一个 item。

UNIX_usage 数据集文件中的每行数据是一个 unix 命令或者命令的参数，如果我们直接对原始的数据集进行挖掘，那么可能会挖掘到一些含有参数的规则，而这种规则是没有实际意义的，因为我们要找的是命令于命令之间的联系。为了避免这种现象的发生，在数据挖掘之前，首先对原始数据集进行清洗，即把那些命令参数过滤掉，只保留真正的命令。

同 Groceries 数据集一样，我们首先需要找到合适的 support 和 confidence，经过多次尝试，最终选定 support=0.1，confidence=0.8。UNIX_usage 数据集的关联性比较高，即便选择了限制性比较强的参数，得到的关联规则的 confidence 仍较高。在此设定下，总共有 11 条关联规则，如表 6 所示。

表 6: UNIX_usage 的规则

ls, fg ⇒ elm	0.1121	0.9844	8.7812
exit, fg ⇒ elm	0.1174	1.0000	8.5152
cd, exit, vi ⇒ ls	0.1121	0.9403	8.3881
finger, fg ⇒ elm	0.1174	0.9851	8.3881
elm, vi ⇒ ls	0.1068	0.8824	8.2647
ls, exit, vi ⇒ cd	0.1121	0.8514	7.5946
cd, elm ⇒ ls	0.1281	0.9600	7.4933
exit, vi ⇒ ls	0.1317	0.8222	6.2444
cd, vi ⇒ ls	0.1601	0.9574	5.9787
cd, exit ⇒ ls	0.1779	0.9434	5.3019
ls, vi ⇒ cd	0.1601	0.8333	5.2037

从表 6 中，我们可以看到一些符合事实的关联规则，比如 cd, vi ⇒ls，它表示在一个 session 中，如果执行了 cd 和 vi 命令，那么有 95.74% 的概率也执行了 ls 命令。这是非常符合人的直觉的，首先用 cd 命令进入某个指定的文件夹，然后用 vi 命令编辑文件，最后用 ls 命令查看一下这个文件。

如果将 UNIX_usage 的关联规则和 Groceries 的关联规则做类比，那么会发现一些违反事实的规则。比如 cd, exit ⇒ ls，按照之前的理解，这条关联规则应该解释成执行了 cd 和 exit 这两条命令之后，在后续的过程中执行 ls 的概率为 94.34%，但是这一解释显然不符合常识。在执行 exit 之后，当前 session 就已经结束了，后续也不会再执行其他命令，当然也不可能执行 ls 命令了。

出现上述现象的原因是 UNIX_usage 数据集和 Groceries 数据集本质上并不相同。

Groceries 数据集是 transaction 的集合，每个 transaction 由若干商品名构成，且若干商品名之间是无序的；UNIX_usage 数据集是 session 的集合，每个 session 由若干命令构成，且若干命令之间是有序的，可以先执行 ls 再执行 exit，但是反过来却不行。

5 总结

关联规则的挖掘过程分为两步：第一步找到所有的频繁项集，第二步从这些频繁项集中生成强关联规则。第二步相对来说比较简单，只需要遍历每个频繁项集的所有非空子集，然后判断 confidence 是否满足要求即可。第一步要困难许多，这种困难也在本次实验中有所体现。当用 dummy 方法找所有的频繁项集时，即使是 support 设置成 0.1，也无法在规定的时间内完成。为了应对这种挑战，人们发明了各种各样的方法来寻找频繁项集，其中最具代表性的便是 Apriori 算法和 FP-growth 算法。Apriori 算法的精髓是运用先验知识来减少搜索空间，从而提高执行效率，相比于 dummy 算法，它有了质的提升。但是人们对于这种算法仍然不是特别满意，这主要归因于它的两个最主要的缺点：当 support 较小时会生成大量的候选频繁项集和多次重复扫描数据库。为了克服这两个缺点，人们提出了 FP-growth 算法。这种方法无论 support 多小，都只需要扫描两遍数据库，且不会产生候选频繁项集。FP-growth 算法的核心思想是用到了“树”这种数据结构，将原始数据库构建成一棵树，这样后续的所有操作都只需在这棵树上进行。

穷尽搜索方法 dummy 在实际场景中的应用性不强，因为它会暴力穷举所有可能的候选频繁项集。对于含有 n 种 item 的数据集，至多可以产生 $2^n - 1$ 个候选项集，所以 dummy 方法的时间复杂度是 $O(2^n)$ 级别的。同时由于穷举所有的候选频繁项集，所以空间复杂度也很高。Apriori 方法相较于 dummy 方法，减少了候选频繁项集的数目，因此提高了时间和空间的效率。FP-growth 算法没有从候选频繁项集的角度考虑，而是直接生成频繁项集，因此又进一步地提高了时空效率。

Groceries 数据集来自于真实顾客的购物记录，它的关联规则的置信度（confidence）并不算特别高，但是由于 support 也较小，所以 lift 较大，也就是说如果购买了关联规则左边的商品，那么对关联规则右边的商品的销售也有极大的促进作用。UNIX_usage 数据集来自于真实用户的命令行历史记录，处于隐私保护的需求，替换了一些敏感内容。这个数据集的关联性比较高，置信度（confidence）最高可达 1.0，但是因为 support 也较高，所以 lift 整体上不如 Groceries。在用程序挖掘出关联规则之后，并不能直接应用这些规则来解决现实问题，因为有部分规则并不符合实际情况，所以还需要再人工筛选一次。