数据挖掘导论作业二

作业及数据集简介

在作业二中，我们需要对给定的数据集进行关联分析，并提取出关联规则。我们需要比较使用基准算法，Apriori算法以及FP-growth算法产生原始数据集的频繁项集的效率差距，并观察从频繁项集中提取关联规则后的结果。

在本次实验中我们使用到了两个数据集，一个是杂货店商品销售记录的数据集，其中每条数据包括了一次销售记录中购买的商品；另一个是不同用户在Unix下的命令输入数据，一共包含9个用户的命令输入情况，其中每条指令输入都略去了输入参数而只保留了输入中调用程序。

方法介绍

我们的目标是在给定的数据集上，找出其频繁项集，并使用规则挖掘算法从频繁项集中找到置信度高的规则。实验任务要求我们使用一个基础的baseline方法，Apriori算法以及FP-Growth算法来分别进行频繁项集的搜索。在实验中我们使用Python代码完成了基础方法以及Apriori方法的实现，而FP-Growth算法由于在实现中需要构建FP树结构，且数据结构与搜索过程更为复杂，考虑到Python代码的效率和实现过程的复杂，我们使用了weka软件包的Java封装来完成FP-Growth算法的关联分析。我们在下面主要介绍基础方法以及Apriori方法的实现。

基础Baseline方法

最为朴素的基础算法应当是找到我们的数据集中的所有项，然后遍历这些项构成的集合的所有非空子集，找到其中支持度高于我们设定的最小支持度的项集，即为我们想要寻找的频繁项集。容易看出这样的方法具有极大的时间复杂度，以我们的杂货店购物清单数据集为例，该数据集中共有169个商品，如果我们想要遍历这169个商品构成的集合的所有子集，那么我们一共需要对2^{169}个项集计算支持度，这样的复杂度显然是我们的普通计算机难以接受的。因此我们对baseline算法做出了一些改进，使得我们能够在有限时间内运行。

我们在实现baseline方法时借鉴了Apriori算法的思想。我们知道Apriori方法是迭代进行频繁项集搜索的，即总是由L_{k-1}频繁项集得到L_k频繁项集，而得到的L_k为频繁项集的充要条件是构造它的L_{k-1}项集和新增的一项I均为频繁的。如果我们在迭代时不考虑L_{k-1}和I是否是频繁的，那么就是朴素的高时间复杂度的子集遍历方法，这样的方法必然难以在有限时间内得出结果。而我们设计的baseline方法是，每次只保留L_{k-1}项集中的频繁项，在迭代时将数据集中所有可能项I加入L_{k-1}项集构成L_k项集，再考虑得到的项集是否频繁。换而言之，我们对baseline方法的改进是，在每轮L_k项集迭代完成后，对其按最小支持度进行剪枝，然后再进行下一轮迭代。这样的方法使得我们可以在有限时间内完成频繁项集的搜索。

Apriori算法

我们在baseline方法中已经简要提到了Apriori方法的思想，即总是由已经得到的L_{k-1}频繁项集加入一个项I，得到L_k频繁项集，而要想使得L_k是一个频繁项集，显然加入的一项I本身也必须是频繁的，所以我们只需在每轮迭代中，从频繁L_1项集中选取L_{k-1}项集中没有的项I加入，构成L_k项集即可。

在这样的迭代中，我们不难发现，包含L_k项集的数据一定也包含L_{k-1}项集，因此我们可以在每轮迭代中，存储包含当前项集的数据行的索引，这样我们在进行下一轮迭代时，就可以直接从上一轮的索引中寻找包含当前项集的数据行，而无需访问全部数据集。实验中发现，这样的方法虽然会占用一定的存储空间，但是能够很大程度上的降低时间开销。

FP-Growth算法

尽管我们在自己实现的Apriori算法中采取了一些以空间换时间的方法，使得算法扫描数据集的时间开销有所降低，但是我们仍然需要以较高的频率访问原始数据。FP-Growth方法通过建立起FP-Tree数据结构，使得我们只需要扫描两次数据集，即可基于FP-Tree数据结构完成频繁项集的搜索。FP-Growth算法可以分为建立项头表，建立FP-Tree以及挖掘FP-Tree三个步骤，借助FP-Tree的高效数据结构更快地获取数据集的频繁项集。我们在实践中使用了weka提供的封装来调用FP-Growth算法。

规则挖掘算法

通过上述算法获取了频繁项集后，我们还需要从频繁项集中挖掘出规则，并按置信度排序输出规则。我们在代码实现中用了一个启发式的方法来减少挖掘规则的迭代数，即如果规则$A \rightarrow B$不成立，那么对任何项集C满足$B \subseteq C$，规则$A \rightarrow C$也一定不成立。我们对每个大于1的频繁项集L进行划分，将其划分为两个非空的子集A, B，计算C的出现次数与A的出现次数比，如果大于设定的最小置信度，就说明规则$A \rightarrow B$是成立的，否则我们维护一个数据结构存储失败的规则，以避免后续迭代时的重复计算。

实验设计

对于杂货店数据集，其数据较为干净，我们在预处理中将每行数据的商品进行分离，同时，代码中也提供了将数据转换为可供weka读取的arff格式的方法。对于Unix命令数据集，我们在读取原始数据时以一次登录会话为数据集中的一行，删除了数据中的一些无关参数，并将所有属性名都转换成了小写，同样，我们也提供了将其转换为可供weka读取的arff格式的代码。

实验要求我们比较使用三种方法下的复杂度大小，包括占用的内存容量以及时间消耗。由于对多个方法内存占用的比较无法在一次程序中同时完成（前面执行的方法可能存在数据暂未释放，被计入到后面的方法内存占用中），因此我们只比较了杂货店数据集上三种算法在相同的参数设置下的内存占用。而对于时间消耗情况，我们则比较了杂货店数据，9个Unix用户分别的命令数据，以及所有用户的总体命令数据上三种算法在相同参数设置下的时间消耗。最后我们尝试在不同参数进行关联分析，以找到数据集上的一些关联规则信息。

实验结果

内存占用情况

我们测试了三种算法在杂货店数据集上的内存占用情况，设置的参数为最小支持度0.01，最小置信度0.5，得到的结果如下

算法	Baseline	Apriori	FP-Growth
内存占用（字节）	81391616	79249408	266338304

可以看出baseline方法和Apriori方法的内存占用大致相同，而FP-Growth算法的内存占用比起前两者有着显著的提升，这应该是因为FP-Growth算法需要维护FP-Tree数据结构，利用高效的数据结构实现以空间换取时间。

时间消耗情况

我们测试了三种算法在杂货店数据集，9个Unix用户分别的命令数据以及9个用户的所有命令数据上的时间消耗，得到下面的表格，表中括号内的数值分别表示设置的最小支持度和最小置信度，即`(minSup, minConf)`。

耗时	Baseline	Apriori	FP-Growth
Grocery (0.01, 0.5)	322.985801s	2.735973s	0.355962s
Unix-User0 (0.02, 0.8)	42.439821s	0.992606s	0.014854s
Unix-User1 (0.02, 0.8)	170.300485s	3.899034s	0.010452s
Unix-User2 (0.02, 0.8)	68.440717s	1.252259s	0.018946s
Unix-User3 (0.02, 0.8)	644.737970s	14.778876s	0.016146s
Unix-User4 (0.02, 0.8)	1065.319859s	19.120121s	0.015297s
Unix-User5 (0.02, 0.8)	5642.428383s	143.216976s	0.007668s
Unix-User6 (0.02, 0.8)	612.320783s	11.133965s	0.054474s
Unix-User7 (0.02, 0.8)	51.043992s	1.058168s	0.030031s
Unix-User8 (0.02, 0.8)	1815.199563s	33.050908s	0.036498s
Unix-User-All (0.02, 0.8)	2859.677559s	51.441275s	0.106931s

可以看出，我们实现的Apriori算法的时间效率往往比基础baseline算法有着数十倍的提升，而FP-Growth算法的效率更为高效，往往能在1秒之内完成关联分析，基础baseline算法在一些大数据集上甚至需要几个小时的时间完成频繁项集的搜索，这是在实际应用中不能允许的。也可以看出，FP-Growth在时间消耗上的优越性使得我们完全可以容忍其在空间上更多的占用。

规则分析

我们主要观察杂货店数据集。如果我们将算法的最小支持度调至0.001，就可以发现一些置信度极高的规则：

```
1. [sugar=T, rice=T]: 12 ==> [wholemilk=T]: 12   <conf:(1)>
2. [hygienearticles=T, cannedfish=T]: 11 ==> [wholemilk=T]: 11   <conf:(1)>
3. [rootvegetables=T, pipfruit=T, hygienearticles=T]: 10 ==> [wholemilk=T]: 10
<conf:(1)>
4. [rootvegetables=T, whipped/sourcream=T, hygienearticles=T]: 10 ==>
[wholemilk=T]: 10   <conf:(1)>
5. [rootvegetables=T, whipped/sourcream=T, flour=T]: 17 ==> [wholemilk=T]: 17
<conf:(1)>
6. [rootvegetables=T, butter=T, rice=T]: 10 ==> [wholemilk=T]: 10   <conf:(1)>
7. [pipfruit=T, butter=T, hygienearticles=T]: 10 ==> [wholemilk=T]: 10   <conf:
(1)>
8. [domesticeggs=T, butter=T, softcheese=T]: 10 ==> [wholemilk=T]: 10   <conf:
(1)>
9. [domesticeggs=T, curd=T, sugar=T]: 10 ==> [wholemilk=T]: 10   <conf:(1)>
10. [domesticeggs=T, napkins=T, creamcheese=T]: 11 ==> [wholemilk=T]: 11
<conf:(1)>
```

这些规则的置信度均为1，表明了它们之间具有完全的依赖关系。可以看出，虽然支撑这些规则的数据数目不多（最多的也只有不到20个），但是这些规则对商家而言仍然是具有参考价值的。倘若商家能够提高其中某些商品的销量，那么就极有可能带动规则中其他对应商品的销售。

但是上述这样的规则并没有足够大的支持度来支撑，我们将算法的最小支持度调至0.005，就可以发现一些支撑证据更充分的规则：

```
 1. [yogurt=T, rootvegetables=T, tropicalfruit=T]: 80 ==> [wholemilk=T]: 56
<conf:(0.7)>
 2. [othervegetables=T, rootvegetables=T, pipfruit=T]: 80 ==> [wholemilk=T]: 54
    <conf:(0.68)>
 3. [whipped/sourcream=T, butter=T]: 100 ==> [wholemilk=T]: 66   <conf:(0.66)>
 4. [pipfruit=T, whipped/sourcream=T]: 91 ==> [wholemilk=T]: 59   <conf:(0.65)>
 5. [yogurt=T, butter=T]: 144 ==> [wholemilk=T]: 92   <conf:(0.64)>
 6. [rootvegetables=T, butter=T]: 127 ==> [wholemilk=T]: 81   <conf:(0.64)>
 7. [tropicalfruit=T, curd=T]: 101 ==> [wholemilk=T]: 64   <conf:(0.63)>
 8. [wholemilk=T, rootvegetables=T, citrusfruit=T]: 90 ==> [othervegetables=T]:
57   <conf:(0.63)>
 9. [othervegetables=T, yogurt=T, pipfruit=T]: 80 ==> [wholemilk=T]: 50   <conf:
(0.63)>
10. [pipfruit=T, domesticeggs=T]: 85 ==> [wholemilk=T]: 53   <conf:(0.62)>
```

从这些规则中可以看出，在酸奶，牛奶，蔬菜，水果这几件商品间组合所产生的规则都有着较高的置信度，尤其是购买酸奶，蔬菜，水果，黄油，凝乳等商品的人更是有很高的置信度同时购买牛奶，且这些规则都有着超过0.005的支持度的支撑。

下面我们考虑Unix用户数据集的结果，尽管我们使用了0.02的最小支持度以作区分，并采用了0.8的最小置信度，这要求挖掘出的规则有着较高的关联性。但在我们实际得出的结果中，可以发现大多数的规则都有着极高的置信度（很多规则甚至达到了1的置信度），这也反映了我们在Unix系统的终端操作中某几条命令的使用重复度是很高的。我们先来观察所有用户的命令合集中的关联规则情况：

```
 1. [cp=T]: 682 ==> [cd=T]: 663   <conf:(0.97)>
 2. [mv=T]: 761 ==> [cd=T]: 724   <conf:(0.95)>
 3. [ll=T]: 1115 ==> [cd=T]: 1051   <conf:(0.94)>
 4. [cd=T, more=T]: 655 ==> [ls=T]: 608   <conf:(0.93)>
 5. [vi=T, rm=T]: 747 ==> [cd=T]: 689   <conf:(0.92)>
 6. [ls=T, vi=T]: 1334 ==> [cd=T]: 1225   <conf:(0.92)>
 7. [ls=T, more=T]: 665 ==> [cd=T]: 608   <conf:(0.91)>
 8. [cd=T, exit=T]: 867 ==> [ls=T]: 774   <conf:(0.89)>
 9. [ls=T, rm=T]: 1056 ==> [cd=T]: 932   <conf:(0.88)>
10. [vi=T]: 1946 ==> [cd=T]: 1710   <conf:(0.88)>
```

在其中可以看到最常用的指令往往是`cp`，`cd`，`mv`，`ll`，`more`，`ls`，`vi`等等，这些指令的出现频繁度都有着很高的支持度作为支撑。从规则上看，`cp`和`mv`等等命令都往往伴随着`cd`命令出现，可以看出在使用到很多命令时都会同时使用`cd`命令，以切换工作目录。

对应到具体用户，我们发现User0经常使用`elm`命令，几乎每次登陆都会使用`elm`命令；而User1使用的大多是常规命令，切换工作目录的`cd`命令和列出文件的`ls`命令尤其频繁；User2经常使用`latex`和`xcc`工具，并在使用时伴随着`cd`命令出现，同时也经常使用`a.out`执行编译的结果；User3常使用的命令是`lo`和`fg`，且这些命令常常伴随`elm`出现；User4常使用的命令是`vi`，`ll`和`dir`，这些命令常常同`cd`一起出现；User5常使用的命令是`rm`，`make`，`ls`等等，它们也常常同`cd`一起出现；User6常使用`cd`，`rm`，`elm`等命令，这些命令往往跟随`ls`出现，说明该用户经常在修改文件系统后使用`ls`；User7有一些出现次数不多但置信度较高的指令，如参数`-fu`往往与`ps`指令一起出现；User8同时使用`ll`和`cd`等指令的支持度很高。

数据集的说明中提到了User0和User1是同一个人在不同项目工作时的记录，可以看出来，即便是同一个人的操作序列，当其任务不一样时使用的指令也会有着很大区别。

结论

我们通过在两份数据集上运行三种不同的关联分析算法，找出了数据集中的一些规律，利用这些关联规则可以帮助我们了解数据集中隐藏的规律，并帮助决策者（对于杂货店数据而言）制订更佳的方案。从我们使用的三种不同算法（基础baseline，Apriori算法，FP-Growth算法）也可以看出，采用Apriori算法进行适度的优化就能显著提高关联分析的执行速度，而采用高效的FP-Growth算法虽然会占用更多的内存空间，但是对于关联分析的时间效率有着极其明显的提升，而快速的关联分析算法对我们在更大的数据集上挖掘数据内部的关键信息有着很大的帮助。