《数据挖掘导论》

Assignment 2

Introduction to the problem and the data sets

刘潇远

161220083

liuxy@smail.nju.edu.cn

一、 问题及数据集介绍

本次实验着眼于关联规则挖掘，希望在数据集中找出满足支持度和置信度的关联(共现)规则。实验在两个数据集上进行挖掘，第一个数据集 GroceryStore 记录了一段时间内，某杂货店顾客消费情况，每条数据都是某位顾客在一次结账时购买的全部物品的集合；第二个数据集 UNIX_usage 记录了 9 位学生在使用 UNIX 时输入的命令行指令，每条记录代表在一个命令行窗口从打开到关闭输入的所有命令，每条记录以***SOF***开头，每一行记录一个命令，之后以***EOF***结尾

实验一共实现了三个方法，分别是 naïve 方法，Apriori 方法和 FPGrowth 方法。其中，naïve 方法是没有进行“剪枝”的 Apriori 方法，其余两种方法完全按照课本伪代码进行实现，原理简单讲解清楚，在此不做赘述。

2、方法实现

软件入口放置在 Main.java 中，运行时根据控制台引导，分别进行：输入支持度，输入置信度(以小数形式)，数据集选择，挖掘方法选择。输入完毕后程序执行挖掘，将频繁项集和关联规则分别输出到 frequentSet.txt 文件和 rules.txt 文件，并在控制台输出找出频繁项集所用时间，程序结束。

程序主要实现了三个主类：naiveMethod、Apriori 和 FPGrowth 类，其中 naiveMethod 与 Apriori 完全一致，只是没有包含对于非频繁项集的“剪枝”。除三个主类外，还有数据集格式处理类 DataBase，用于构建 FPTree 的类 FPTree，表示 FPTree 的结点的 FPNode 类，用于按照 1 频繁项集支持度对每条记录进行重排序的 QuickSort 类，用于计算置信度并返回关联规则的 confidence 类。除此之外用于，还有记录一组数据的 Itemset 类，这个类在程序中非常重要，许多其他类的数据结构都是由 Itemset 组织起来的，在 DataBase 中，数据集一条记录被储存在一个 Itemset 中，在三个主类中，一个频繁项集的所有项都被储存在一个 Itemset 中，在 confidence 类中，最终产生的关联规则输出的形式为 <ArrayList<ArrayList<Itemset>>，其中外层 ArrayList 是为了储存多条关联规则，内层给 ArrayList 中只有两个 Itemset，记为 A 和 B，每一对 A 和 B 表示的都是 A 储存的频繁项集出现时，B 也出现，既满足支持度要求(频繁项集挖掘过程中已实现)，也满足置信度要求(confidence 类计算)，置信度储存在 A 的私有成员 confidence 中。

程序入口在 Main.java 中，根据用户输入，选择调用 naiveMethod、Apriori 或 FPGrowth 其中一个。这三个类的构造函数都有四个参数，分别是支持度 min_sup，置信度 confidence，数据集(Grocery 数据集为 1，UNIX_usage 为 2)和数据集存放路径，其中数据集合数据集存放路径在构造函数中传给 DataBase 类，根据数据集本身结构进行规范化处理，每条数据存放在一个 Itemset 中。

Aprioir 在构造函数中自动调用频繁项集挖掘方法 findFrequentItemsets 挖掘所有频发项集，之后程序在主方法中调用 ApprioriMining 方法，在该方法中调用 confidence 中的静态方法 calculateConfidcence，该方法有三个参数，分别是 DataBase DB，ArrayList<Itemset> frequentSet, float con，顾名思义，第一个参数就是数据集内容，第二个参数是已经挖掘好的频繁项集，第三个参数为置信度。calculateConfidence 方法中，通过 for 循环将频繁项集中不同的两项 A 和 B 进行结合，计算 A 出现时，B 出现的置信度和 B 出现时，A 出现的置信度，之后将满足置信度要求的规则储存在返回值中。

FPGrowth 方法大致相同，难点在于模式基 FPTree 的递归建立在实现时有非常多要注意的小点。FPGrowth 构造函数首先构造初始 FPTree，并找出 1 频繁项集，在主方法中调用 FPGrowth 的 FPGrowthMining 进行关联规则挖掘。调用私有方法 FPMing 进行挖掘，该方法只有一个参数 ArrayList<Itemset> recoder，这里面储存有当前模式基的记录，这个方法是一个递归方法，如果 recoder 中只有一种项，那么返回空集。否则对于 recoder 的每种项执行：在 recoder 种去除所有该项后递归调用 FPMing 方法，之后将返回值得到的每一个频繁项集，都将该项加入，之后将该项与其余项的组合，也加入到频繁项集中，然后返回。在得到了频繁项后，与 Apriori 一样调用 confidence 静态方法 calculateConfidence，得到关联规则。

程序除了在控制台实时输出当前计算的频繁项集大小 k 和置信度计算过程外，还产生两个输出文件，分别是 frequentSet.txt 和 rules.txt，frequentSet.txt 存有产

生的频繁项集及其支持度，rules.txt 存有挖掘出的关联规则。

3、结果

注：naïve 方法运行时间过长，因此并没有等到他运行结束，因此只记录 Apriori 和 FPGrowth 方法的运行时间。另外数据集规模，行数对于 UNIX_usage 数据集来说并不能反映数据集真实大小

方法	数据集	规模(行)	频繁项集挖掘时间 (ns:纳秒)
Apriori	Grocery(支持度 100, 置信度 0.4)	9835	4588237228ns
	UNIX_usage(USER0，支持度 20，置信度 0.4)	8974	225080744ns
	UNIX_usage(USER1，支持度 20，置信度 0.4)	19881	542538501ns
	UNIX_usage(USER2，支持度 20，置信度 0.4)	18738	704240495ns
	UNIX_usage(USER3，支持度 20，置信度 0.4)	16867	1288426163ns
	UNIX_usage(USER4，支持度 20，置信度 0.4)	37817	20841934456ns
	UNIX_usage(USER5，支	34821	41224111854ns

	持度 20，置信度 0.4)		
	UNIX_usage(USER6，支持度 20，置信度 0.4)	64152	69656402208ns
	UNIX_usage(USER7，支持度 20，置信度 0.4)	17329	1358742257ns
	UNIX_usage(USER8，支持度 20，置信度 0.4)	54042	1153241968322ns
FPGrowth	Grocery(支持度 100，置信度 0.4)	9835	288452385ns
	UNIX_usage(USER0，支持度 20，置信度 0.4)	8974	49924511ns
	UNIX_usage(USER1，支持度 20，置信度 0.4)	19881	138641519ns
	UNIX_usage(USER2，支持度 20，置信度 0.4)	18738	88557451ns
	UNIX_usage(USER3，支持度 20，置信度 0.4)	16867	130319284ns
	UNIX_usage(USER4，支持度 20，置信度 0.4)	37817	465533954ns
	UNIX_usage(USER5，支持度 20，置信度 0.4)	34821	405103207ns
	UNIX_usage(USER6，支	64152	406757875ns

	持度 20，置信度 0.4)		
	UNIX_usage(USER7，支持度 20，置信度 0.4)	17329	152029536ns
	UNIX_usage(USER8，支持度 20，置信度 0.4)	54042	1024804033ns

4、结论

由此可以得出，在运行速度上 FPGrowth 远优于 Apriori 远优于 naïve 方法。这是因为 Apriori 在每次 k 频繁项集挖掘时进行了"剪枝"，去掉了子集为非频繁项集的 k 候选频繁项集，但是每一步仍然产生大量候选频繁项集，需要对其进行剪枝。而 FPGrowth 更进一步，直接利用 FP 树的结构，消除了 Apriori 方法的缺点，因此速度更快。

注意看 UNIX_usage 数据集中 USER4、USER5、USER6、USER8 三个文件的测试结果，FPGrowth 频繁项集挖掘所需时间远远小于 Apriori 方法……，真的太快了，尤其是 USER8 本来 Apriori 方法让我等到怀疑人生，FPGrowth 一下子就出了结果，二者速度在数量级上有着差别。

至于 naïve 的方法。。。尝试运行了一下，实在太久了，就没有继续进行运行

5、讨论

关于算法，没什么好说的了，反正 Apriori 和 FPGrowth 算法随便翻翻书找找网上的例子就明白是什么事，两者在效率上地差别从上述实验结果中也能清楚地得出结论。关键在于实现。。。FPGrowth 算法主体部分，FPTree 和频繁项集挖掘两部分，涉及细节多，而且这次我没有用自己擅长的 C++而是使用了半生不熟的 Java，在软件规模控制，软件设计上产生了问题，也很少使用继承、模板等方式，有很多函数在不同类中反复出现，没能实现很好地复用。另外在软件测试上，并没有进行测试而是一口气写完了所有代码，导致程序 debug 十分困难，对于两者中较为简单的 Apriori 算法还好说，FPGrowth 算法 debug 时间(不摸鱼地)超过 12 个小时，每次都是连续 de 一个晚上到凌晨，一遍一遍重新运行，一步一步跟着看(我哭了，你呢)。中途也有考虑直接放弃去使用 WEKA 包，但是想到 de 了这么久，而且自己实现能够对算法有更深入的理解，硬着头皮继续下来了，然后结果还不错。不过再有类似的作业我一定会选择直接使用开源开源工具包 ORZ