数算实习大作业报告

倪泽堃 (1200012747)　罗翔宇 (1200012779)　顾澄 (1200012882)

November 25, 2013

1　概况

这次大作业主要是编写课上介绍的数据挖掘算法：Apriori 和 FP 算法，并对它们进行改进，比起前一个大作业来说，模块之间比较独立。

项目结构比较简单，主要文件为：

- apriori.cpp: 暴力的 Apriori 算法
- fp-tree.cpp: （实现得比较好的）FP 算法
- optimized.cpp: 在 Apriori（Eclat）算法的基础上所得的优化算法
- Makefile: 生成可执行文件的 Makefile

各个可执行文件都是从标准输入读入数据，并将答案输出到标准输出，阈值（绝对数，非百分数）以命令行参数形式传入。

2　分模块介绍

2.1　Apriori 算法

本模块由罗翔宇同学编写。我使用了 STL vector 来储存候选项集，每次遍历候选项集进行判断合法性并进行合并。但是由于算法的局限性，改算法实际运行时表现得不尽人意，比如对于 mushroom 数据，将阈值设为 5% 的时候，就无法在可以容忍的时间范围内跑出结果。具体的效率分析在本报告末的图表中有详细的说明。

除了算法的局限性之外，我觉得程序效率瓶颈在 vector 的插入和排序上，具体的优化思路在下文都有详细的描述，这里我就不再赘述了。

2.2　FP 算法

本部分由顾澄同学编写。经典的 FP 树算法分为创建 FP 树和对 FP 树挖掘两部分。所以本程序分为初始化、创建 FP 树以及挖掘 FP 树三部分。

初始化阶段首先读入所有数据，用 vector 存储每条事务以节省空间，同时统计每个项目出现的次数。随后对所有项目数进行排序，删除每条事务中支持度达不到最小支持度的项目，并用 number 数组记录每个项目所在的排名，方便以后的查询。

在创建 FP 树时，使用指针结构存储 FP 树，但是考虑到每个节点的儿子数可能较多，因此使用了 STL map 来维护指向儿子节点的指针，依此将每个事务数中达到要求的项目数插入树中。同时，将树中 key 值（即项目编号）相同的节点用指针相连。

在挖掘阶段。从支持度最小的项目开始，依此找到支持度达到要求的项目组，然后每次将条件模式基构建成一棵新的 fp 树，并递归求解。

由此可见 FP 树算法在挖掘方面所用时间最多。阈值越大，FP 树上节点越小，所用时间越少，而在阈值较小的时候，需要很长时间进行挖掘。

2.3 改进算法

本模块由倪泽堃同学负责编写。FP 树和暴力的 Apriori 算法并没有非常显然的改进思路，因而选取了 Apriori 算法变种 Eclat 算法为切入点进行改进。Eclat 算法对于每一个频繁项集记录包含它的交易编号，每次作扩展的时候，就求出两个项集的交易编号集合之交，判断其元素个数即可。但是首先的问题是对于 mushroom 这项数据与支持度 K=5%，最终结果就有几百万项，如果计算过程中直接用 vector 保存每项的交易编号集合，那么内存消耗极大，一般的 4G 内存电脑不能承受。因而我们需要寻求其他的方式，比如用一个 bitset 来记录交易编号集合，每位 1/0 表示该交易在/不在集合里。这样 32 个 bit 压成一个 4 字节 int 储存，内存消耗大大减小。每次直接对两个 bitset 做与运算并调用库函数 count 获取集合大小。这样最初的改进版本写成，但是对 mushroom 5% 这组数据运行时间达到了 27.9 秒（g++ 4.8.1 -O3，不算最后排序输出，运行环境为 Intel Core i5-2430M CPU @ 2.40GHz×4、Arch Linux 64-bit，下同），实在是有点长，需要大力优化。由于其他两个数据风格不一样，在前 4 个优化中，就仅仅使用 mushroom 5% 这个数据。

优化 1：vector 改成静态数组 运用 gprof 对最初版本进行分析显示，运行时间的大部分消耗在 vector 的插入上。显然，vector 实在是太慢了。因此经过一番努力，我将求解部分的大部分用到 vector 的地方改用自己实现的静态 vector 存储，用一个巨大的内存池和一个标明可用空间起点的指针，每次直接计算出需要存储的序列，并将该序列写入内存池中，将指针后移。这样经过改进，所得的优化版本 1 的性能有很大提升。具体可见下表。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
最初版本	-	27.9	-	-	-	
优化版本 1	使用静态数组	14.5	5082240280	5002205	1016	24.9%

优化 2：去除 bitset 中连续 0 显然，使用 bitset 存储之后，还有可以改进之处：k 越大，频繁 k-项集的支持度计数就越小，此时 bitset 中将有许多的连续 0。我们对此作出改进：在 32 位压成一个 int 存储的 bitset 中，将等于 0 的 int 从中去除，用二元组记录不为 0 的元素的下标与值。尽管由一元变成了二元组，作出改进后的优化版本 2 中内存使用仍然减少了，而且时间效率有极大的上升。具体可见下表。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
优化版本 1	使用静态数组	14.5	5082240280	5002205	1016	24.9%
优化版本 2	去除连续 0	7.1	3460183168	5002205	692	24.9%

优化 3：重标号、排序 压缩连续 0 的效果仍然不够理想。可以想像，其中原因是编号过于分散，存储的数据不够紧凑。因此需要对交易重新排序，使得内容相近的交易处在尽量近的位置。这时，可以发现，将含有最频繁的项的交易排在一起，再将含有第二频繁的项的交易排在一起，以此类推，所得的交易序列相邻项相似度非常高，是比较优的。因此先将项按频繁程度重新标号，将交易排序，然后进行求解。加上该优化所得的优化版本 3 效率见下表。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
优化版本 2	去除连续 0	7.1	3460183168	5002205	692	24.9%
优化版本 3	重标号、排序	6.1	2791211776	12782139	218	70.6%

优化 4：频繁序列降序存储 令人稍有惊讶的是，优化版本 3 的优化幅度不甚理想。从表中可以很显然地看到，虽然数据存储更加紧凑了，但是该算法多增长出了大量支持度小于给定阈值的候选项集。经过分析可以发现，该算法由序列 $\{a_0, a_1, ..., a_{n-1}\}$ 与 $\{a_0, a_1, ..., a_{n-2}, a_n\}$ 增长出 $\{a_0, a_1, ..., a_n\}$, 但是由于 a_{n-1}, a_n 比较罕见，增长出的序列支持度可能会剧烈下降。相反，如果从 $\{a_1, ..., a_n\}$ 与 $\{a_0, a_2, ..., a_n\}$ 增长出目标序列，支持度下降幅度则一般不会很大。因此改用降序存储频繁项集，经测试，所得的优化版本 4 效率有极大的提升，如下表所示。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
优化版本 3	重标号、排序	6.1	2791211776	12782139	218	70.6%
优化版本 4	降序增长	2.0	978700888	3780460	259	0.7%

在优化版本 4 中，废弃率已经降低到惊人的程度，数据的存储也很紧凑，运行时间降低到了 2 秒，而之后的排序就要 1.6 秒，可以说效率已经相当高了。将其他两组数据也纳入统计，可以对刚才的所有优化做个总结。

版本	优化方法	mushroom 5%	T10I4D100K 0.1%	retail 0.1%
最初版本	-	27.9	-	-
优化版本 1	使用静态数组	14.5	16.6	123.5
优化版本 2	去除连续 0	7.1	3.6	7.5
优化版本 3	重标号、排序	6.1	2.2	6.8
优化版本 4	降序增长	2.0	2.0	4.4

优化 5：预处理频繁 2 项集 我们发现，T10I4D100K 0.1% 这组数据答案才 27532，但是却花了 2.0 秒才计算出答案。这是因为计算频繁 2 项集时，多扩展了许多支持度很低的项集。因而可以使用一个很简单的方法，就是预处理出频繁 2 项集。优化后的版本 5 效率如下表所示。

版本	优化方法	mushroom 5%	T10I4D100K 0.1%	retail 0.1%
优化版本 4	降序增长	2.0	2.0	4.4
优化版本 5	预处理 2 项集	2.0	0.8	1.0

优化 6：部分基数排序 此时我们的程序已经非常优了。不过如果最后进行排序输出，效率还是比较堪忧的，其中排序就要花费比较长的时间。观察到我们是对序列排序，而每一项并不大。因而在序列长度较长的时候，可以先用基数排序，当每个桶的元素个数小于一定数的时候再用快排排序。经过改进后排序效率有所增加，如下表所示，其中运行时间加上了排序的时间，但不包括输出时间。

版本	优化方法	mushroom 5%	T10I4D100K 0.1%	retail 0.1%
优化版本 5	预处理 2 项集	3.6	0.8	1.0
优化版本 6	部分基数排序	3.0	0.8	1.0

值得注意的是，将输出时间加上，对于 mushroom 5% 这组数据，程序总共运行时间达到了 7s，这说明效率主要瓶颈在输出了。也就是说现在的优化版本 6 已经是非常优了。

一些思考 可以发现，最终的这个版本其实和 FP 树有异曲同工之处。对项按频繁程度排序并对交易重标号、排序的过程和 FP 树的构建其实是相同的，而降序增长其实跟 FP 算法从底层向上递归的思想也是一致的，去除了连续 0，也就是选取了 FP 树中一些包含当前频繁项集的节点。但是这个算法的优势在于，FP 树中不频繁项出现得很零散，不利于处理，但是该算法将节点进行了压位存储，并且调用内置函数快速统计 1 出现次数，在这种情况下这个算法比 FP 算法优很多。但是当 FP 算法递归到高层频繁节点时，FP 算法已经能统计出每个高层节点下有多少个包含当前频繁项集的交易，从而往上递归时直接利用当前计算结果，而本算法增长方式还是 Apriori 的，必须通过求集合交来获得增长后的支持度，因而在包含高层频繁节点下可能会劣于 FP 算法，但是另一方面来说，FP 算法只进行前缀扩展而不通过两个频繁项集的交，可能会多扩展出废状态来。总体来说，这个改进算法属于 FP 算法和 Apriori（Eclat）算法的结合，但是核心操作仍然在于 Eclat 算法的集合交。从总体效率上来讲，这个算法比起实现较好的 FP 算法来说仍然是略优的。

2.4 效率对比

将暴力 Apriori 版本、FP 树版本和优化版本作对比，不加上排序输出，在上一小节所述测试条件下，各算法在各个实验数据和阈值下的运行时间如下：

算法名称	mushroom					retail					T10I4D100K				
	25%	20%	15%	10%	5%	0.5%	0.4%	0.3%	0.2%	0.1%	0.5%	0.4%	0.3%	0.2%	0.1%
Apriori 版本	1.2	7.8	16.3	101	>300	110	242	>300	>300	>300	>300	>300	>300	>300	>300
FP 树版本	0.0	0.1	0.1	0.3	1.7	0.2	0.3	0.4	0.7	1.2	1.8	2.1	2.5	3.2	4.1
优化版本	0.2	0.2	0.3	0.5	2.0	0.4	0.4	0.5	0.6	1.0	0.6	0.6	0.6	0.7	0.8

由此可见，优化版本在除了 mushroom 以外的其他测试数据上效率都略高于 FP 树，不过令人遗憾的是，优化版本的优化幅度仍然非常有限，特别是在 FP 树本身比较占优势的 mushroom 这一个点上。好在我们找到了 Apriori 和 FP 树的一个结合点，或许在此之上运用相互借鉴的思想，我们能够找到一个更加快速高效的算法。

3 心得体会

- 这次大作业是我第一次进行数据挖掘领域相关的算法的分析与实现。完成作业的过程中，我对数据挖掘这项工作有了更加清晰的认识。它不仅是一个抽象的数学模型问题，而且在诸多场合都有

着非常重要的应用，比如音乐网站的后台推荐系统，可以通过分析每位用户的浏览记录，来对歌曲进行分类并对用户进行有针对性的推荐。这次大作业不仅使我在编程能力上有了很大的提升，也开阔了视野，对" 大数据" 时代这个名词有了更深刻的认识。

(罗翔宇)

- 通过这次大作业，我感受到了数据挖掘的魅力。FP 树算法非常有趣，在编程的过程中，提高了我的实际编程能力，感受到编程的乐趣。在不同阈值和事务数的时候每个算法的性能相差很大，这也是在实际动手编程的时候才能体会到的。大作业让我学会了新的算法并对数据挖掘这一领域的有了初步的认识。

  (顾澄)

- 这次大作业使我对程序优化思路有了一些基本的认识，其主要在于找到瓶颈，并尽量减小这个瓶颈，使得程序在运行时尽量少做无用功，将效率发挥到极致。最终也正是如此，通过优化，所有数据点的运行时间除 mushroom 5% 以外都降低到了 1 秒以内，而 mushroom 5% 这组数据本身答案巨大，因此这已经是一个非常令人满意的结果了。当然优化的结果仍远远不算到了“极致”，然而这次 NP Hard 问题的优化过程仍然给以后大数据时代的程序优化带来了有益的启迪。

  (倪泽堃)