数据挖掘大作业报告

1200012863 刘敏行 1200012864 刘子渊

（一）背景

自20世纪80年代起，全球信息每隔十几个月甚至几个月就要增长一倍，呈现出爆炸式增长。至2000年，全球数据存储容量已达到300万个TB——这是信息爆炸的时代！

如何从大量数据中提取或挖掘知识，成为当前信息科学与技术领域科学工作者面临的一大挑战！

关联规则挖掘指的是发现大量数据中项集之间的关联或相关联系，这对于现实十分有帮助！

挖掘关联规则，我们主要采用两种算法Apriori算法和FP-Tree算法，下面将一一介绍。

（二）人员分工

组长：刘子渊 负责编写FP-Tree算法及其优化、撰写报告

组员：刘敏行 负责编写Apriori算法及其优化、画图、撰写总报告

（三）各自算法报告及感想

第一部分 Apriori算法——

**一．算法简介**

Apriori算法使用频繁项集的先验知识，使用一种称作逐层搜索的迭代方法，通过k项集用于探索(k+1)项集。首先，通过扫描事务（交易）记录，找出所有的频繁1项集，该集合记做L1，然后利用L1找频繁2项集的集合L2，L2找L3，如此下去，直到不能再找到任何频繁k项集。算法具有这样一条核心性质：任一频繁项集的所有非空子集也必须是频繁的。算法借此对候选集剪枝。

下面，对于输入①数据库D；②最小支持度阀值min\_sup。

给出算法伪代码：

STEP1: L1 = {large 1-itemsets}//得到频繁一项目集

STEP2: for( k = 2 ; Lk-1≠ Φ; k ++)

{

Ck = apriori-gen(Lk-1);//连接由频繁项目集k-1得到候选项目集k

For all transaction t∈D

{

Ct = subset(Ck,t)//遍历事务数据库集查看Ck中是否在事务t

For all candidates c ∈ Ct

{

c.count++;//对遍历后且包含在某一事务t中的Ck计数

//计算支持度

}

}

Lk = { c ∈ Ct | c.count ≥ min\_sup}

}

由原始算法思想可以看出，频繁集、候选集中有多个元素（即一个频繁/候选项），而每个元素又有多个元素（即每一个项包含的各个事物），因此可以想到频繁/候选集都是一个二维数组，所以本算法也是采用二位数组作为主要数据结构（具体实现应用了stl里的vector）

**二．算法理论时空复杂度分析**

（1）时间复杂度分析：基于上一部分对基础Apriori算法给出的伪代码，可以看到算法的时间花费主要体现在3个方面：

①从k-频繁集自连接得到k+1-频繁集。因为需要用两个循环访问k-频繁集，而为了比较是否可以自连接，往往要逐位比较，因此时间复杂度为O（|Lk|^2\*k）

②生成初步候选项之后对于其每一个子集在频繁集中查找以剪枝。对于每一个候选项，都要生成k个子集，去频繁集中一一匹配、筛选，时间复杂度O（|Ck|\*|L(k-1)|\*k）

③对于候选集统计支持度。对于每一个候选项，去事物集中匹配从而计算支持度，时间复杂度O（|D|\*k\*|Ck|）

在上述算法分析中，除了总事物集长度|D|是定的以外，其他均不是定值，但又是互相关联的，因为频繁集最开始由事务集得来，候选集由频繁集得来，而新的频繁集又由候选集得来。其中，由频繁集得出候选集类似于一个两两组合的过程，所以复杂度是个组合数，而上面分析总复杂度设计两个组合数的相乘的和，我们知道这是指数级别的复杂度（具体函数表达式因实现的不同而不同），Apriori是NP算法。

（2）空间复杂度分析：Apriori算法所需空间主要花在存储候选集和频繁集上了，对于一个结果为k项的情形集，如果认为最小阈值无限接近0，则候选/频繁集可能存储所有k元子集，即2^k-1个，仍然是指数级别的，在数量大的情况下很大而且很难估计。

**三．算法实现**

（1）朴素Apriori算法实现

针对基本Apriori算法的实现，我用了4个主要函数，Get\_one\_item（获得1-频繁集），apriori\_gen（生成k-候选集），check\_frequent（判断该候选集各子集是不是频繁的）。主要的数据存储结构是oridata（原事务集），candidate(候选集)，frequent（频繁集），都是vector<vector<int>>。四个函数与之前给出的伪代码中的功能一一对应，实现思路也比较直观、简单，因此对这四个函数的各自具体功能不再赘述。具体参考源代码Pure\_Apriori.cpp。

不过，这里需要做说明的是，尽管这个cpp程序用的是最朴素的Apriori算法思想，但我还是做了几处的优化（否则实在有点慢。。。），因为这些优化在之后真正的优化版本中也有涉及，因此在这里解释一下，方便后面的阅读。这里按照代码运行顺序列出——

1. while(frequent.size()>=level+1)

{

...

}

这里出现在mian函数中的每一次由k-频繁集计算k+1频繁集的时候，意思是说当k-频繁集的个数小于k+1时，就不要再考虑k+1-频繁集了，原因很简单，因为一个k+1-频繁集的k+1个子集都在k-频繁集里，所以k-频繁集至少有k+1项才对。这个优化可以省去最后一次所做的无用的频繁集自连接即剪枝生成候选集的时间

1. for(int i=0;i<fresiz;++i)

for(int j=i+1;j<fresiz;++j)

{

mer=true;

…

if(mer)

{

…

}

else

break;

}

这里出现在apriori\_gen函数中的判断可否连接的部分，这里优化意思如果i和j不能连接，则i和j之后的肯定也不能连接，所以直接break找下一个i，原因也很很简单，因为我在实现的时候频繁集是内部有序的，第i个频繁项无法和第j个频繁项连接，本质上是他们的前k-1项不完全相同，而即i<j（字典序），而j后面的项更加比i大，所以不必判断。这一优化减少了判断是否可以相互连接的次数。

③while(lef<=righ)

{

tag=true;

mid=(lef+righ)/2;

for(int j=0;j<level;++j)

{

if(temp[j]>frequent[mid][j])

{

lef=mid+1;

tag=false;

break;

}

else if(temp[j]<frequent[mid][j])

{

righ=mid-1;

tag=false;

break;

}

}

if(tag)

break;

}

这里出现在check\_frequent函数中，在一一在频繁集中查找候选项的子集时，因为只是判断存在性，因此改为用二分查找，降低判断时间复杂度。

④for(int k=0;k<level;++k)

{

if(vis[j][candidate[i][k]]==false)

{

check=false;

break;

}

}

…

if(temp\_cnt+T-j<min\_num)

break;

}

这里出现在get\_fre函数中，有两处优化。第一处意思是对于每个频繁项的每个元素在事项集查找时，用一个vis数组记录，vis[i][j]为true代表j在第i个事项集出现，这样省去了一些遍历事项集查找时间；第二个优化是如果即使假设后面某一候选项在之后的事务集中全出现也无法达到阈值，它肯定不是频繁集，就不用再去事务集访问了，减少了访问事务集的次数。

⑤整个shorten\_fre函数是一个优化，主要思想是如果一个事物在k-频繁集中出现的频次小于k，则把所有含有该事物的频繁集删掉。这样做是对的因为对于任意一个k+1-频繁项，它的k个子集必在频繁集中，而任一个事项至少出现k次，因此出现少于k次的某事项绝不会出现在k+1-频繁集中，故删掉。这步优化将频繁集的长度裁剪，生成的候选集会减少。

以上五点有自己的原创，有同学的提示，也有从网上看到的，它们所做的都无非是一些常数级别的优化，但叠加起来仍然对程序性能影响不小（mushroom25%数据从102s降到24s），在数据量比较大的时候虽然还是会迅速陷入指数增长，但是比起原始的版本又要好不少。

（2）优化Apriori算法实现

·第一次优化

我第一次进行的优化是基于矩阵压缩的算法。主要原理是用一个bool矩阵维护，行代表事务项，列代表实际物品，物品在该行出现赋为1，否则为0 。然后，每次在从k-1-频繁集产生k-候选集的时候，都进行bool矩阵的剪裁，应用三条规则：1、统计各个事物在频繁集中的频次对事物（列）裁剪（就是上面的shorten\_fre）。2、查看行和，如果<k+1就删除行（也就是说某一事务不足k+1个物品）3、查看列和，如果<支持度阈值就删除列。不断重复2、3以不断压缩矩阵。

这个算法的核心是不断压缩事务集，这样即使每次访问事务集也会不断减少复杂度。但是，这种算法严重依赖数据，要求每一个事务不能包含太多项目，而且长度要有长有短才行（没有经过事实验证，理论推断而已），而我们所用的数据显然不符合这些规则，此算法效率极低，也给了我一个教训，就是在尝试编写一个算法之前一定要好好分析，否则是耽误时间。

·第二次优化

一路不通，只好选择另一条路，最终经过优化的Apriori算法主要是基于Eclat算法的倒排思想，即记录一个事项在事务集中出现的行号，将出现赋为“1”，否则是“0”。然后对于两个可以相接的频繁项，只需要将它们的各自的行号向量做与操作，得到的新向量不但记录了新的事项在事务集出现行号的集合，还可以直接读该集合中“1”的个数以得到其支持度，直接得到k+1-频繁集。这么做的优点是，只要扫描一遍事务集得到一项集对应的行号集，之后就再也用不着扫描事务集了。同时，用不着通过频繁集生成候选集这一步，而是直接通过快捷的与操作得到结果，从而生成频繁集。如此，大大提高程序效率。

我实现这个算法主要用到的函数Get\_one\_item（获得1-频繁集），apriori\_gen（生成k-频繁集）。主要的数据存储结构是，frequent（k-频繁集），fre（k+1-频繁集），都是vector<vector<int>>，Bitand（k-行号集），\_Bitand（k+1-行号集），都是vector<unsigned int\*>

对几处核心代码作一下解释

1. unsigned int \*Btemp=new unsigned int[T/32+1];

for(int j=0;j<tidsiz;++j)

Btemp[tid[i][j]/32]+=1<<(tid[i][j]%32);

前面提到在产生1-频繁集时要扫描一遍事务集，记录每一个事物的行号集，这可以用一个bitmap实现，也就是对于一个长为事项集大小的bit链，在哪一行出现了该事物就给bit链的第几位赋成1，否则是0，将这条bit链压缩到一个unsigned int数组中，于是，之后对两个频繁项的行号集做“与”，实际上是将对应的unsigned int数组一一做与，然后统计bit位“1”的个数，即：（vv是新频繁项的行号集）

for(int cal=0;cal<=T/32;++cal)

{

vv[cal]=Bitand[i][cal] & Bitand[j][cal];

unsigned int xx=vv[cal];

while(xx)

{

if(xx&1)

++tid\_cnt;

xx>>=1;

}

}

②注意到新的算法没有check\_frequent函数了，是因为前面也分析了这个剪枝过程很费时间（即使用了二分），而Eclat算法对于潜在的候选项只是简单地“与”操作，用时很短，我经过实验发现，让Eclat多计算几个不是潜在候选集的支持度所用的时间远少于每一次费劲判断的时间，故删去这一函数。

经过Eclat算法改进，程序性能大为提高！具体比较见下。

**四．算法比较**

测试环境：Windows 7系统，2.30GHz，4G内存，DEV 4.9.9.2编译（除mushroom.dat的5%数据）

1. 对于第一组数据mushroom.dat,可以看到对于优化后的Apriori算法明显占优，变化也不大，优化效果明显。要说明的是缺少朴素算法运行5%的时间，因为太慢了……另外，优化算法在本地运行5%数据会爆内存，因此上传到了ics服务器上运行的。同时，如果在DEV中打开-O3优化选项，优化算法的时间会进一步减少。

原始数据：（单位：second）

|  |  |  |  |
| --- | --- | --- | --- |
| 数据/版本 | Pure\_Apriori | Opti\_Apriori | Opti\_Apriori(-O3) |
| mushroom.dat25% | 25.052 | 1.334 | 0.325 |
| mushroom.dat20% | 254.58 | 6.91 | 1.419 |
| mushroom.dat15% | 468.505 | 13.391 | 2.497 |
| mushroom.dat10% | 4830.395 | 73.265 | 12.266 |
| mushroom.dat5% |  | 74.02 | 22.01 |

朴素与优化比较

优化与打开-O3编译优化比较

1. retail.dat顶多只有1-项集，所以主要时间都花在运行求1-项集已经连接为2-项集上，差距并不大，但还是可以看出优化的算法比朴素的更优（虽然很可能是误差），由于本身时间很少，没有统计开-O3优化的结果。

原始数据：（单位：second）

|  |  |  |
| --- | --- | --- |
| 数据/版本 | Pure\_Apriori | Opti\_Apriori |
| retail40%.dat | 2.201 | 1.441 |
| retail45%.dat | 2.286 | 1.642 |
| retail50%.dat | 2.255 | 1.736 |
| retail55%.dat | 2.344 | 1.656 |
| retail60%.dat | 2.331 | 1.468 |

1. T10I4D100K.dat的数据量很大，同样很明显优化后的Apriori算法效率远高于朴素算法，但是其实本身优化算法也要花多达几分钟，而开了-O3优化之后均只要十几秒，这么大的差距究竟是为何我暂时还没有想到，留给以后分析吧。这里要说明的一点是，运行这五组数据时我对根据数据特点对代码稍做了改变，否则会爆内存。另外，跟上面一样，0.1%的时候朴素的太慢了，没有测出具体数据。

原始数据：（单位：second）

|  |  |  |  |
| --- | --- | --- | --- |
| 数据/版本 | Pure\_Apriori | Opti\_Apriori | Opti\_Apriori(-O3) |
| T10I4D100K0.5%.dat | 6364.581 | 90.165 | 4.008 |
| T10I4D100K0.4%.dat | 8213.671 | 113.832 | 4.592 |
| T10I4D100K0.3%.dat | 14895.474 | 142.259 | 5.579 |
| T10I4D100K0.2%.dat | 27776.409 | 178.403 | 7.954 |
| T10I4D100K0.1%.dat |  | 282.065 | 14.506 |

朴素与优化比较

优化与-O3优化比较

关于使用空间的分析，由于没找到监控空间使用量的函数，同时本身对空间使用量理解的也不太清楚（是指new出来的，还是静态的也算？）所有最终没有统计。这里只做简要说明，我的朴素算法只要同时保存k-频繁集/候选集，所以占用的空间与最大频繁集/候选集的个数直接相关。跑所有数据，系统都正常运行，没有提示内存不足。而对于Eclat算法，其一大特点便是空间消耗较大，因为它要求同时存k-1-频繁集和k-频繁集，而且对候选集不做裁剪（前面提到了），所以空间消耗较朴素而言要多，而5%的mushroom提示内存不足只好去服务器上跑便是例证。

**五．总结与感想**

我觉得我通过这次大作业的编写，我主要感受到了算法的重要性，尤其是优化算法的重要性。其实，我很早就完成了Apriori基础算法的编写，我以为就算可以优化，也应该变化不会太大。结果，到了最后一周和同学的一比，发现同样是基于Apriori算法，他们的比我快近10倍，而且数据越大差距越明显，我才终于明白这个大作业的真正难点与重点在于怎么对算法进行优化上，我也用了整整三天的时间进行各种优化，才把我的程序优化到一个不错的级别，算是深切感受到了优化的重要性。

其次，这次大作业很好地锻炼了我将算法描述转换为代码的能力，因为我看到的文献资料全部是讲述算法思路，没有任何源码，我必须在理解了算法思想后自己用高效的方法进行实现，这个能力很重要，对今后的学习甚至工作很有帮助。感谢这次大作业！希望今后还有这样的机会！

·第二部分 FP-Tree算法——

一．FP-Tree算法简介

FP-tree是用于实现FP-Growth算法的一种高度压缩的数据结构，其构造方式与Trie即字符树有些相似，在节省了空间的同时为实现FP-Growth算法提供了良好的基础。首先FP树统计所有item的出现频率并按照出现频率从大到小的顺序重新排列每一条交易中的item顺序。这样使得出现频率大的项更加接近FPTree的根节点从而会被更多的记录共享，以达到压缩数据量的目的。在进行这一环节的同时，实际上也统计出了所有长度为1的频繁项，可以合并两个功能。

二．算法理论时空复杂度分析

时空复杂度：

时间复杂度：建树部分是O(logn)，挖据部分，基于代码的分析每一次挖掘递归需要O(n2logn)的操作，总共平均应是O(n2log2n)；

空间复杂度：由于我们的算法将每次递归所需要的空间都事先开好在全局变量当中，所以空间复杂度并不高，每次需要o(n)的table项和O(n)的建树，所以空间复杂度为O(n);

三．算法实现

（1）朴素算法

上述第一环节代码：

while(fgets(data,5010,fin)){

if(data==NULL) break;

dataOper(k++);

}

Void dataOper(int k){

int len = strlen(data),num = 0,ptr = 0,i;

bool check = false;

for(i = 0;i<len;i++){

if(data[i]<='9'&&data[i]>='0'){

num = num\*10+data[i]-'0';

check = true;

} else if(check){

trans[k].data[ptr++] = num;

table[0][num].tot++;

num = 0;

check = false;

}

}

trans[k].len = ptr;

}

这段代码中处理了每一条记录将其转化为程序中所储存的格式。

在对频率进行排序后，建树函数中枚举每一条记录枚举进行插入操作。

BuildFPT(k);

bool DataCmp(const int &a,const int &b){

return table[0][a].tot<table[0][b].tot;

}

void BuildFPT(int k){

int i,temp = 0;

for(i = 0;i<k;i++){

temp = 0;

sort(trans[i].data,trans[i].data+trans[i].len,DataCmp);

if(temp==2) println(trans[i]);\*/

insert(trans[i],0);

}

}

最后是进行数据挖掘：

数据挖掘以递归形式进行，每次枚举一个符合最小支持度的项，上溯其所有祖先结点建成一棵新的FP-Tree（即当前item的条件模式基），在新的FP树上递归进行数据挖掘知道新建成的树为空。

void search(int level){

int i,j;

if(x[level].len>1)

ans[AnsNum++] = x[level];

if(root[level]->child==NULL){

return;

}

int now = level+1;

if(now>maxlevel){

for(i = 0;i<=ObNum[t];i++)

table[now][i] = Table(i);

maxlevel = now;

}

FPT \*test;

int k;

x[now] = x[level];

for(i = ObNum[t];i>=0;i--){

if(table[level][i].tot<minsup) continue;

root[now] = new FPT();

for(j = 0;j<=ObNum[t];j++){

table[now][j].tot = 0;

table[now][j].id = j;

table[now][j].last = new FPT(j);

}

test = table[level][i].last;

k = 0;

while(test->pre!=NULL){

RecursiveUp(test,now);

insert(var,now);

test = test->pre;

k++;

}

sort(table[now],table[now]+ObNum[t]+1,TableCmp);

x[now].data[x[now].len++] = table[level][i].id;

if(x[now].count) x[now].count = min(x[now].count,table[level][i].tot);

else x[now].count = table[level][i].tot;

if(k==1)

FixIt(0,var.len,x[now]);

else

search(now);

x[now].len--;

x[now].count = x[level].count;

TreeDestruction(now);

delete root[now];

}

return;

}

（2）优化：

单路径优化：

由于事先进行了排序，所以当前生成的子树仅有一条路径的时候，可以确定这条路径上所有节点所组成的全部组合均是频繁项，所以直接进行组合的枚举，这样能大幅减少操作时间。

代码：

void FixIt(int left,int len,Pattern xx){

if(xx.len!=1) ans[AnsNum++] = xx;

if(left>=len) return;

int i;

Pattern temp = xx;

for(int i = left;i<len;i++){

temp.data[temp.len++] = var.data[i];

//fprintf(fout,"%d %d %d\n",AnsNum,i,level);

FixIt(i+1,len,temp);

temp.len--;

}

return;

}

四．算法比较

测试环境：Windows 7系统，2.10GHz，8G内存，DEV 5.2.0.3编译

朴素算法数据未经-O3优化

优化版本数据经过-O3优化

1. mushroom.dat

原始数据：（单位：second）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据/版本 | Pure\_FP-Tree | | Opti\_FP-Tree | |
| mushroom.dat25% | 0.145 | | 0.13 | |
| mushroom.dat20% | 1.137 | | 0.242 | |
| mushroom.dat15% | 2.027 | | 0.386 | |
| mushroom.dat10% | 12.128 | | 1.752 | |
| mushroom.dat5% | 75.255 | | 8.481 | |
|  |  | |  | |
| 1. retail.dat   原始数据：（单位second）  数据/版本 | Pure\_FP-Tree | Opti\_FP-Tree | |
| retail40%.dat | 0.899 | 0.684 | |
| retail45%.dat | 0.93 | 0.606 | |
| retail50%.dat | 0.89 | 0.823 | |
| retail55%.dat | 0.862 | 0.666 | |
| retail60%.dat | 0.909 | 0.638 | |

③ T10I4D100K.dat

原始数据：（单位second）

|  |  |  |
| --- | --- | --- |
| 数据/版本 | Pure\_FP-Tree | Opti\_FP-Tree |
| T10I4D100K0.5%.dat | 2.427 | 1.664 |
| T10I4D100K0.4%.dat | 2.9 | 2.139 |
| T10I4D100K0.3%.dat | 4.007 | 2.422 |
| T10I4D100K0.2%.dat | 6.983 | 4.536 |
| T10I4D100K0.1%.dat | 11.243 | 7.033 |

优化后的版本效率明显高于朴素版本

五．总结与感想

这次的大作业中对于这一项目之前完全没有了解，所以都是从零开始。看文献到百度都得自己先看懂了再写。中途出了很多bug，调试优化跑数据都是很新鲜的体验，之前写程序都没有过。这次作业当中，第一次跑出正确答案、优化之后时间大幅减少的时候都兴奋异常。希望今后还能有机会参与这种项目。

第三部分 两种算法之间比较——

下面比较两个算法：

显然，FP-Tree算法效率明显高于Apriori算法

分析原因：AP算法在拓展可能的频繁项上花费了大量时间，每次都要比较当前长度下每两个频繁项之间是否相等（除最后一个元素之外），虽然中途可以进行剪枝但是始终会产生爆炸式的时间消费。相比之下，FP算法在使用较少数据库遍历次数的前提下对数据进行了高度压缩，将大量的数据压缩成少量的结点，而且随着挖掘的进行，需要进行挖掘的树会由于剪枝而变得越来越小今儿提升挖掘速度并且减少开销。综上，FP在时间开销上比AP优秀很多。

（四）总的感想

很感谢老师给我们提供了这样一个完成、分析、比较两个一种功能的不同算法的大作业实习机会，通过这个大作业，我们不但进一步提高了编程能力和素养，还进一步锻炼了分工合作能力，这样的机会很难得！