Apriori and FP-Growth

环境:windos系统下的eclipse中.

编写语言:java

Apriori算法思想:Apriori算法是常用的用于挖掘出数据关联规则的算法，它用来找出数据值中频繁出现的数据集合，找出这些集合的模式有助于我们做一些决策。　Apriori算法采用了迭代的方法，先搜索出候选1项集及对应的支持度，剪枝去掉低于支持度的1项集，得到频繁1项集。然后对剩下的频繁1项集进行连接，得到候选的频繁2项集，筛选去掉低于支持度的候选频繁2项集，得到真正的频繁二项集，以此类推，迭代下去，直到无法找到频繁k+1项集为止，对应的频繁k项集的集合即为算法的输出结果。

伪代码:

// 找出频繁 1 项集

L1 =find\_frequent\_1-itemsets(D);

For(k=2;Lk-1 !=null;k++){

// 产生候选，并剪枝

Ck =apriori\_gen(Lk-1 );

// 扫描 D 进行候选计数

For each 事务t in D{

Ct =subset(Ck,t); // 得到 t 的子集

For each 候选 c 属于 Ct

c.count++;

}

//返回候选项集中不小于最小支持度的项集

Lk ={c 属于 Ck | c.count>=min\_sup}

}

Return L= 所有的频繁集；

第一步：连接（join）

Procedure apriori\_gen (Lk-1 :frequent(k-1)-itemsets)

For each 项集 l1 属于 Lk-1

For each 项集 l2 属于 Lk-1

If( (l1 [1]=l2 [1])&&( l1 [2]=l2 [2])&& ……&& (l1 [k-2]=l2 [k-2])&&(l1 [k-1]<l2 [k-1]) )

then{

c = l1 连接 l2 // 产生候选

//若k-1项集中已经存在子集c则进行剪枝

if has\_infrequent\_subset(c, Lk-1 ) then

delete c; // 删除非频繁候选

else add c to Ck;

}

Return Ck;

剪枝（prune）

Procedure has\_infrequent\_sub (c:candidate k-itemset; Lk-1 :frequent(k-1)-itemsets)

For each (k-1)-subset s of c

If s 不属于 Lk-1 then

Return true;

Return false;

预处理:

数据1:



C1 支持度

啤酒 4 4/5

尿布 4 4/5

婴儿爽身粉 2 2/5

面包 1 1/5

雨伞 1 1/5

牛奶 2 2/5

洗衣粉 1 1/5

可乐 1 1/5

支持度 大于或等于40%

L1 支持度

啤酒 4 4/5

尿布 4 4/5

婴儿爽身粉 2 2/5

牛奶 2 2/5

C2 支持度 L2 支持度

啤酒,尿布 3 3/5 啤酒,尿布 3/5

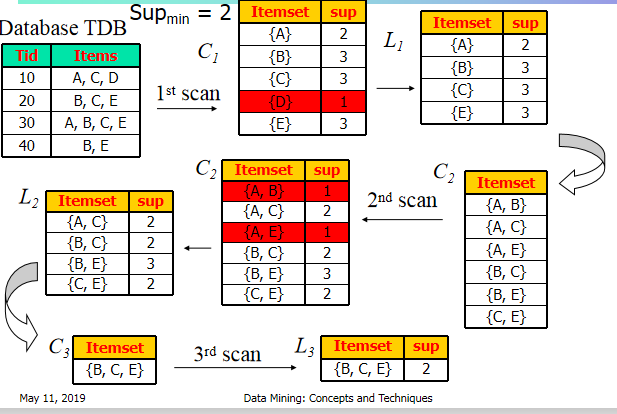
啤酒,婴儿爽身粉 1 1/5 啤酒,牛奶 2/5

啤酒,牛奶 2 2/5 尿布,婴儿爽身粉 2/5

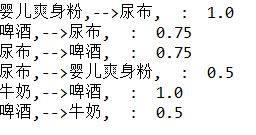
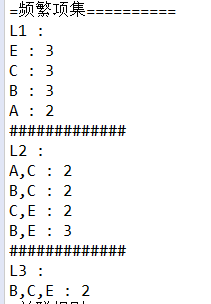
尿布,婴儿爽身粉 2 2/5

婴儿爽身粉,牛奶 1 1/5

数据2



实验结果

重要代码

**private** **boolean** hasInfrequentSubset(String candidateString, Map<String, Integer> setMap){

String[] strings = candidateString.split(***GAP***);

//找出候选集所有的子集，并判断每个子集是否属于频繁子集

**for** (**int** i=0; i<strings.length; ++i ) {

String subString = "";

**for** (**int** j = 0; j<strings.length;++j ) {

**if** (j!=i) {

subString += strings[j]+***GAP***;

}

}

**if** (setMap.get(subString)==**null**) {

**return** **true**;

}

}

**return** **false**;

}

**private** Map<String,Integer> aprioriGen(Map<String, Integer> setMap){

//此处传入的参数就是上面返回的频繁项集。

Map<String, Integer> candidateSetMap = **new** HashMap<>();

// 对每个商品取集合

Set<String> candidateSet = setMap.keySet();

//单独考虑每个商品的支持度，如果合格，就可以进行拼接。否则丢掉。

**for** (String s1 : candidateSet) {

String[] strings1 = s1.split(***GAP***);

String s1string = "";

**for** (String temp : strings1) {

s1string += temp+***GAP***;

}

**for** (String s2 : candidateSet) {

//此处也是默认商品序号是有序的。这样先判定前len-1项是否相等。

//如果前面相等，第len项不相等，那么就可以拼接成len+1长度的候选集了。

String[] strings2 = s2.split(***GAP***);

**boolean** flag = **true**;

**for** (**int** i=0; i< strings1.length-1;++i) {

**if** (strings1[i].compareTo(strings2[i]) != 0) {

flag = **false**;

**break**;

}

}

**if** (flag && strings1[strings1.length-1].compareTo(strings2[strings2.length-1])<0) {

//连接步：产生候选

String c=s1string+strings2[strings2.length-1]+***GAP***;

**if** (hasInfrequentSubset(c,setMap)) {

//剪枝步：删除非频繁的候选

}

**else** {

candidateSetMap.put(c,0);

}

}

}

}

**return** candidateSetMap;

}

**public** Map<String, Integer> apriori(ArrayList<String> dataList){

Map<String, Integer> setpFrequentSetMap = **new** HashMap<>();

setpFrequentSetMap.putAll(find1\_FrequentSet(dataList));

Map<String, Integer> frequentSetMap = **new** HashMap<String, Integer>();

frequentSetMap.putAll(setpFrequentSetMap);

// Into the loop choose

**while**(setpFrequentSetMap!=**null** && setpFrequentSetMap.size() > 0){

Map<String, Integer> candidateSetMap = aprioriGen(setpFrequentSetMap);

//得到的就是候选集 candidateSetMap ，当然我们只要key部分即可啦！

Set<String> candidateKeySet = candidateSetMap.keySet();

//扫描D，进行计数

**for** (String data : dataList) {

**for** (String candidate : candidateKeySet) {

**boolean** flag = **true**;

String[] strings = candidate.split(***GAP***);

**for** (String string : strings) {

//意味着在Data，也就是在初始的购物记录中查找当前的频繁项集中的某一条。寻找string如果不成功，则返回-1；

// indexOf(Object o)方法

// 功能：查找某个元素在ArrayList中第一次出现的位置。

**if** (data.indexOf(string+***GAP***)==-1) {

flag = **false**;

**break**;

}

}

//如果查找成功，那么就可以丢到正式的候选集中了。

**if** (flag) {

candidateSetMap.put(candidate,candidateSetMap.get(candidate)+1);

}

}

}

//从候选集中找到符合支持度的频繁项集，stepFrequentSetMap顾名思义就是每一次找到的新的频繁集。

//所以在置入新的频繁集之前，都要先把上次的清空掉。

setpFrequentSetMap.clear();

**for** (String candidate : candidateKeySet) {

Integer count = candidateSetMap.get(candidate);

**if** (count>=***SUPPORT***) {

setpFrequentSetMap.put(candidate,count);

}

}

// puaAll的作用是把一个Map的所有元素置入并且去重。

// 合并所有频繁集

frequentSetMap.putAll(setpFrequentSetMap);

}

//While循环结束的条件是新的频繁项集的大小为0.也就是必须要完全空了才出来。

//这时候已经确保了frequentSetMap包含有所有的频繁项集了。

**return** frequentSetMap;

}

FP growth 算法思想

FP树时一种输入数据的压缩表示，它通过逐个读入事务，并把事务映射到FP树中的一条路径来构造，由于不同的事务可能会有若干个相同的项，因此它们的路径可能部分重叠。路径相互重叠越多，使用FP树结构获得的压缩效果越好，如果FP树足够小，能够存放在内存中，就可以直接从这个内存中的结构提取频繁项集，而不必重复地扫描存放在硬盘上的数据。FP-growth是一种以自底向上方式探索树，由FP树产生频繁项集的算法，给定上面构建的FP树，算法首先查找以e结尾的频繁项集，接下来是b,c,d，最后是a，由于每一个事务都映射到FP树中的一条路径，因为通过仅考察包含特定节点（例如e）的路径，就可以发现以e结尾的频繁项集，使用与节点e相关联的指针，可以快速访问这些路径，下图显示了所提取的路径，后面详细解释如何处理这些路径，以得到频繁项集。

伪代码:

输入:事务集合 List<List<String>> transactions  
输出:频繁模式集合及相应的频数 Map<List<String>,Integer> FrequentPattens  
   
初始化 PostModel=[],CPB=transactions  
void FPGrowth(List<List<String>> CPB,List<String> PostModel){  
    if CPB为空：  
        return  
           
    统计CPB中每一个项目的计数，把计数小于最小支持数minSuport的删除掉，对于CPB中的每一条事务按项目计数降序排列。  
    由CPB构建FP-Tree，FP-Tree中包含了表头项headers，每一个header都指向了一个链表HeaderLinkList，链表中的每个元素都是FP-Tree上的一个节点，且节点名称与IMG_257header.name 相同。

    for header in headers:  
        newPostModel=IMG_257header.name+PostModel  
        把<newPostModel, header.count>加到FrequentPattens中。  
        newCPB=[]  
        for TreeNode in HeaderLinkList:  
            得到从FP-Tree的根节点到TreeNode的全路径path，把path作为一个事务添加到newCPB中，要重复添加TreeNode.count次。  
        FPGrowth(newCPB,newPostModel)

procedure FP\_growth(Tree, a)  
if Tree 含单个路径P then{  
         for 路径P中结点的每个组合（记作b）  
         产生模式b U a，其支持度support = b 中结点的最小支持度；  
} else {  
         for each a i 在Tree的头部(按照支持度由低到高顺序进行描){  
产生一个模式b = a i U a，其支持度support = a i .support；  
   构造b的条件模式基，然后构造b的条件FP-树Treeb；

  if Treeb 不为空 then

调用 FP\_growth (Treeb, b)；  
}  
}

预处理:

数据1

|  |  |
| --- | --- |
| **TID** | **Items Bought** |
| 100 | a, b, c, d, e, f, g, h |
| 200 | a, f, g |
| 300 | b, d, e, f, j |
| 400 | a, b, d, i, k |
| 500 | a, b, e, g |

Minsuport = 3 重新排序

a 4 a,b,d,e,f,g

b 4 a,f,g

d 3 b,d,e,f

e 3 a,b,d

f 3 a,b,e,g

g 3

数据2

牛奶,鸡蛋,面包,薯片

鸡蛋,爆米花,薯片,啤酒

鸡蛋,面包,薯片

牛奶,鸡蛋,面包,爆米花,薯片,啤酒

牛奶,面包,啤酒

鸡蛋,面包,啤酒

牛奶,面包,薯片

牛奶,鸡蛋,面包,黄油,薯片

牛奶,鸡蛋,黄油,薯片

minsuport = 3

薯片:7 鸡蛋:7 面包:7 牛奶:6 啤酒:4

重新排序

薯片,鸡蛋,面包,牛奶

薯片,鸡蛋,啤酒

薯片,鸡蛋,面包

薯片,鸡蛋,面包,牛奶,啤酒

面包,牛奶,啤酒

鸡蛋,面包,啤酒

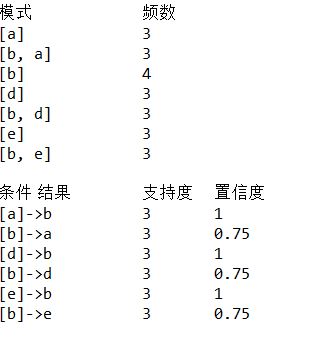
薯片,面包,牛奶

薯片,鸡蛋,面包,牛奶

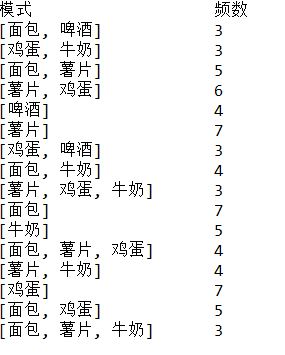
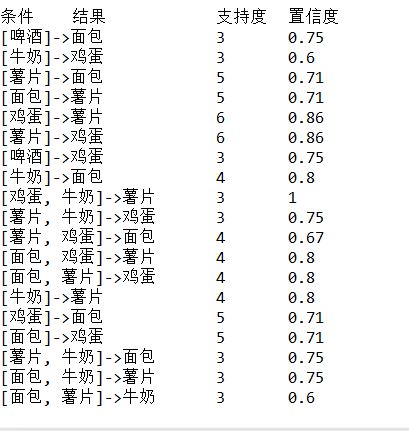
薯片,鸡蛋,牛奶

实验结果:

1



2

主要代码:

**private** **void** FPGrowth(List<List<String>> cpb, LinkedList<String> postModel) {

Map<String, Integer> freqMap = getFrequency(cpb);

Map<String, TreeNode> headers = **new** HashMap<String, TreeNode>();

**for** (Entry<String, Integer> entry : freqMap.entrySet()) {

String name = entry.getKey();

**int** cnt = entry.getValue();

//每一次递归时都有可能出现一部分模式的频数低于阈值

**if** (cnt >= minSuport) {

TreeNode node = **new** TreeNode(name);

node.setCount(cnt);

headers.put(name, node);

}

}

TreeNode treeRoot = buildSubTree(cpb, freqMap, headers);

//如果只剩下虚根节点，则递归结束

**if** ((treeRoot.getChildren() == **null**) || (treeRoot.getChildren().size() == 0)) {

**return**;

}

//如果树是单枝的，则直接把“路径的各种组合+后缀模式”添加到频繁模式集中。这个技巧是可选的，即跳过此步进入下一轮递归也可以得到正确的结果

**if** (isSingleBranch(treeRoot)) {

LinkedList<TreeNode> path = **new** LinkedList<TreeNode>();

TreeNode currNode = treeRoot;

**while** (currNode.getChildren() != **null**) {

currNode = currNode.getChildren().get(0);

path.add(currNode);

}

//调用combine时path不宜过长，否则会OutOfMemory

**if** (path.size() <= 20) {

List<List<TreeNode>> results = **new** ArrayList<List<TreeNode>>();

combine(path, results);

**for** (List<TreeNode> list : results) {

**int** cnt = 0;

List<String> rule = **new** ArrayList<String>();

**for** (TreeNode node : list) {

rule.add(node.getName());

cnt = node.getCount();//cnt最FPTree叶节点的计数

}

**if** (postModel != **null**) {

rule.addAll(postModel);

}

frequentMap.put(rule, cnt);

}

**return**;

} **else** {

System.***err***.println("length of path is too long: " + path.size());

}

}

**for** (TreeNode header : headers.values()) {

List<String> rule = **new** ArrayList<String>();

rule.add(header.getName());

**if** (postModel != **null**) {

rule.addAll(postModel);

}

//表头项+后缀模式 构成一条频繁模式（频繁模式内部也是按照F1排序的），频繁度为表头项的计数

frequentMap.put(rule, header.getCount());

//新的后缀模式：表头项+上一次的后缀模式（注意保持顺序，始终按F1的顺序排列）

LinkedList<String> newPostPattern = **new** LinkedList<String>();

newPostPattern.add(header.getName());

**if** (postModel != **null**) {

newPostPattern.addAll(postModel);

}

//新的条件模式基

List<List<String>> newCPB = **new** LinkedList<List<String>>();

TreeNode nextNode = header;

**while** ((nextNode = nextNode.getNextHomonym()) != **null**) {

**int** counter = nextNode.getCount();

//获得从虚根节点（不包括虚根节点）到当前节点（不包括当前节点）的路径，即一条条件模式基。注意保持顺序：你节点在前，子节点在后，即始终保持频率高的在前

LinkedList<String> path = **new** LinkedList<String>();

TreeNode parent = nextNode;

**while** ((parent = parent.getParent()).getName() != **null**) {//虚根节点的name为null

path.push(parent.getName());//往表头插入

}

//事务要重复添加counter次

**while** (counter-- > 0) {

newCPB.add(path);

}

}

FPGrowth(newCPB, newPostPattern);

}

}

**private** TreeNode buildSubTree(List<List<String>> transRecords,

**final** Map<String, Integer> freqMap,

**final** Map<String, TreeNode> headers) {

TreeNode root = **new** TreeNode();//虚根节点

**for** (List<String> transRecord : transRecords) {

LinkedList<String> record = **new** LinkedList<String>(transRecord);

TreeNode subTreeRoot = root;

TreeNode tmpRoot = **null**;

**if** (root.getChildren() != **null**) {

//延已有的分支，令各节点计数加1

**while** (!record.isEmpty()

&& (tmpRoot = subTreeRoot.findChild(record.peek())) != **null**) {

tmpRoot.countIncrement(1);

subTreeRoot = tmpRoot;

record.poll();

}

}

//长出新的节点

addNodes(subTreeRoot, record, headers);

}

**return** root;

}

**private** **void** addNodes(TreeNode ancestor, LinkedList<String> record,

**final** Map<String, TreeNode> headers) {

**while** (!record.isEmpty()) {

String item = (String) record.poll();

//单个项的出现频数必须大于最小支持数，否则不允许插入FP树。达到最小支持度的项都在headers中。每一次递归根据条件模式基本建立新的FPTree时，把要把频数低于minSuport的排除在外，这也正是FPTree比穷举法快的真正原因

**if** (headers.containsKey(item)) {

TreeNode leafnode = **new** TreeNode(item);

leafnode.setCount(1);

leafnode.setParent(ancestor);

ancestor.addChild(leafnode);

TreeNode header = headers.get(item);

TreeNode tail=header.getTail();

**if**(tail!=**null**){

tail.setNextHomonym(leafnode);

}**else**{

header.setNextHomonym(leafnode);

}

header.setTail(leafnode);

addNodes(leafnode, record, headers);

}

}

}

**public** List<StrongAssociationRule> getAssociateRule() {

**assert** totalSize > 0;

List<StrongAssociationRule> rect = **new** ArrayList<StrongAssociationRule>();

//遍历所有频繁模式

**for** (Entry<List<String>, Integer> entry : frequentMap.entrySet()) {

List<String> items = entry.getKey();

**int** count1 = entry.getValue();

//一条频繁模式可以生成很多关联规则

List<StrongAssociationRule> rules = getRules(items);

//计算每一条关联规则的支持度和置信度

**for** (StrongAssociationRule rule : rules) {

**if** (frequentMap.containsKey(rule.condition)) {

**int** count2 = frequentMap.get(rule.condition);

**double** confidence = 1.0 \* count1 / count2;

**if** (confidence >= **this**.confident) {

rule.support = count1;

rule.confidence = confidence;

rect.add(rule);

}

} **else** {

System.***err***.println(rule.condition + " is not a frequent pattern, however "

+ items + " is a frequent pattern");

}

}

}

**return** rect;

}

总结:apriori算法与FP-growth 算法分析事物间的频繁项集来分析事物间的关联关系.就编码难度来说apriori算法会简单很多,但apriori算法多次扫描交易数据库，每次利用候选频繁集产生频繁集；而FP-growth则利用[树形结构](https://www.baidu.com/s?wd=%E6%A0%91%E5%BD%A2%E7%BB%93%E6%9E%84&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)，无需产生候选频繁集而是直接得到频繁集，大大减少扫描交易数据库的次数，从而提高了算法的效率。实验中面对较大问题应该是fp树结构的构建,包括节点的组成以及相互间的联系.通过查询资料和参考网友的分享才基本掌握树的生成过程.挖掘fptree的时候一开始也没有思路,然后通过手动演算挖掘各项频繁项集的过程才有了编码思路.