**Apriori算法**

**1． 挖掘关联规则**

**1.1**   **什么是关联规则**

一言蔽之，关联规则是形如X→Y的蕴涵式，表示通过X可以推导“得到”Y，其中X和Y分别称为关联规则的先导(antecedent或left-hand-side, LHS)和后继(consequent或right-hand-side, RHS)

**1.2**   **如何量化关联规则**

关联规则挖掘的一个典型例子便是购物车分析。通过关联规则挖掘能够发现顾客放入购物车中的不同商品之间的关联，分析顾客的消费习惯。这种关联规则的方向能够帮助卖家了解哪些商品被顾客频繁购买，从而帮助他们开发更好的营销策略。比如：将经常同时购买的商品摆近一些，以便进一步刺激这些商品一起销售；或者，将两件经常同时购买的商品摆远一点，这样可能诱发买这两件商品的用户一路挑选其他商品。

在数据挖掘当中，通常用“支持度”（support）和“置性度”（confidence）两个概念来量化事物之间的关联规则。它们分别反映所发现规则的有用性和确定性。比如：

Computer => antivirus\_software , 其中 support=2%, confidence=60%

表示的意思是所有的商品交易中有2%的顾客同时买了电脑和杀毒软件，并且购买电脑的顾客中有60%也购买了杀毒软件。在关联规则的挖掘过程中，通常会设定最小支持度阈值和最小置性度阈值，如果某条关联规则满足最小支持度阈值和最小置性度阈值，则认为该规则可以给用户带来感兴趣的信息。

**1.3**   **关联规则挖掘过程**

1）几个基本概念：

关联规则A->B的**支持度**support=P(AB)，指的是事件A和事件B同时发生的概率。

**置信度**confidence=P(B|A)=P(AB)/P(A),指的是发生事件A的基础上发生事件B的概率。

同时满足最小支持度阈值和最小置信度阈值的规则称为**强规则**。

如果事件A中包含k个元素，那么称这个事件A为**k项集，**并且事件A满足最小支持度阈值的事件称为**频繁k项集**。

2）挖掘过程：

第一，找出所有的频繁项集；

第二，由频繁项集产生强规则。

**2． 什么是Apriori**

**2.1**   **Apriori介绍**

**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，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。最后再在所有的频繁集中找出强规则，即产生用户感兴趣的关联规则。

其中，**Apriori**算法具有这样一条**性质**：任一频繁项集的所有非空子集也必须是频繁的。因为假如P(I)< 最小支持度阈值，当有元素A添加到I中时，结果项集（A∩I）不可能比I出现次数更多。因此A∩I也不是频繁的。

**2.2**   **连接步和剪枝步**

在上述的关联规则挖掘过程的两个步骤中，第一步往往是总体性能的瓶颈。**Apriori**算法采用连接步和剪枝步两种方式来找出所有的频繁项集。

1）  连接步

为找出Lk（所有的频繁k项集的集合），通过将Lk-1（所有的频繁k-1项集的集合）与自身连接产生候选k项集的集合。候选集合记作Ck。设l1和l2是Lk-1中的成员。记li[j]表示li中的第j项。假设Apriori算法对事务或项集中的项按字典次序排序，即对于（k-1）项集li，li[1]<li[2]<……….<li[k-1]。将Lk-1与自身连接，如果(l1[1]=l2[1])&&( l1[2]=l2[2])&&……..&& (l1[k-2]=l2[k-2])&&(l1[k-1]<l2[k-1])，那认为l1和l2是可连接。连接l1和l2 产生的结果是{l1[1],l1[2],……,l1[k-1],l2[k-1]}。

2）  剪枝步

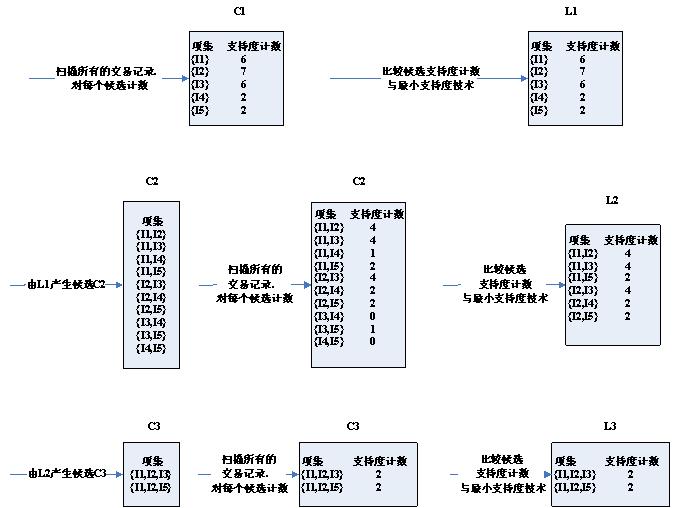
CK是LK的超集，也就是说，CK的成员可能是也可能不是频繁的。通过扫描所有的事务（交易），确定CK中每个候选的计数，判断是否小于最小支持度计数，如果不是，则认为该候选是频繁的。为了压缩Ck,可以利用Apriori性质：任一频繁项集的所有非空子集也必须是频繁的，反之，如果某个候选的非空子集不是频繁的，那么该候选肯定不是频繁的，从而可以将其从CK中删除。

（Tip：为什么要压缩CK呢？因为实际情况下事务记录往往是保存在外存储上，比如数据库或者其他格式的文件上，在每次计算候选计数时都需要将候选与所有事务进行比对，众所周知，访问外存的效率往往都比较低，因此Apriori加入了所谓的剪枝步，事先对候选集进行过滤，以减少访问外存的次数。）

**2.3**   **Apriori算法实例**

|  |  |
| --- | --- |
| **交易ID** | **商品ID列表** |
| T100 | I1，I2，I5 |
| T200 | I2，I4 |
| T300 | I2，I3 |
| T400 | I1，I2，I4 |
| T500 | I1，I3 |
| T600 | I2，I3 |
| T700 | I1，I3 |
| T800 | I1，I2，I3，I5 |
| T900 | I1，I2，I3 |

上图为某商场的交易记录，共有9个事务，利用Apriori算法寻找所有的频繁项集的过程如下:

[](http://photo.blog.sina.com.cn/showpic.html#blogid=4882f26d0100spl1&url=http://s1.sinaimg.cn/orignal/4882f26dgaa472c8c0b00)

详细介绍下候选3项集的集合C3的产生过程：从连接步，首先C3={{I1,I2,I3}，{I1,I2,I5}，{I1,I3,I5}，{I2,I3,I4}，{I2,I3,I5}，{I2,I4,I5}}（C3是由L2与自身连接产生）。根据Apriori性质，频繁项集的所有子集也必须频繁的，可以确定有4个候选集{I1,I3,I5}，{I2,I3,I4}，{I2,I3,I5}，{I2,I4,I5}}不可能时频繁的，因为它们存在子集不属于频繁集，因此将它们从C3中删除。注意，由于Apriori算法使用逐层搜索技术，给定候选k项集后，只需检查它们的（k-1）个子集是否频繁。

**3． Apriori伪代码**

|  |
| --- |
| **算法：Apriori**  **输入：D -** **事务数据库；min\_sup -** **最小支持度计数阈值**  **输出：L - D中的频繁项集**  **方法：**   L1=find\_frequent\_1-itemsets(D); // 找出所有频繁1项集       For(k=2;Lk-1!=null;k++){          Ck=apriori\_gen(Lk-1); // 产生候选，并剪枝          For each 事务t in D{ // 扫描D进行候选计数              Ct =subset(Ck,t); // 得到t的子集              For each 候选c 属于 Ct   c.count++;          }          Lk={c属于Ck | c.count>=min\_sup}  }  Return L=所有的频繁集；    **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 //连接步：产生候选                     if has\_infrequent\_subset(c,Lk-1) then                         delete c; //剪枝步：删除非频繁候选                     else add c to Ck;                    }            Return Ck;     **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; |

**4． 由频繁项集产生关联规则**

Confidence(A->B)=P(B|A)=support\_count(AB)/support\_count(A)

关联规则产生步骤如下：

1）  对于每个频繁项集l，产生其所有非空真子集；

2）  对于每个非空真子集s,如果support\_count(l)/support\_count(s)>=min\_conf，则输出 s->(l-s)，其中，min\_conf是最小置信度阈值。

例如，在上述例子中，针对频繁集{I1，I2，I5}。可以产生哪些关联规则？该频繁集的非空真子集有{I1，I2}，{I1，I5}，{I2，I5}，{I1 }，{I2}和{I5}，对应置信度如下：

I1&&I2->I5            confidence=2/4=50%

I1&&I5->I2            confidence=2/2=100%

I2&&I5->I1            confidence=2/2=100%

I1 ->I2&&I5            confidence=2/6=33%

I2 ->I1&&I5            confidence=2/7=29%

I5 ->I1&&I2            confidence=2/2=100%

如果min\_conf=70%,则强规则有I1&&I5->I2，I2&&I5->I1，I5 ->I1&&I2。

**5． Apriori Java代码**

package com.apriori;

import java.util.ArrayList;

import java.util.Collections;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Set;

public class Apriori {

         private final static int SUPPORT = 2; // 支持度阈值

         private final static double CONFIDENCE = 0.7; // 置信度阈值

         private final static String ITEM\_SPLIT=";"; // 项之间的分隔符

         private final static String CON="->"; // 项之间的分隔符

         private final static List<String> transList=new ArrayList<String>(); //所有交易

         static{//初始化交易记录

                   transList.add("1;2;5;");

                   transList.add("2;4;");

                   transList.add("2;3;");

                   transList.add("1;2;4;");

                   transList.add("1;3;");

                   transList.add("2;3;");

                   transList.add("1;3;");

                   transList.add("1;2;3;5;");

                   transList.add("1;2;3;");

         }

         public Map<String,Integer> getFC(){

        Map<String,Integer> frequentCollectionMap=new HashMap<String,Integer>();//所有的频繁集

        frequentCollectionMap.putAll(getItem1FC());

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

        itemkFcMap.putAll(getItem1FC());

        while(itemkFcMap!=null&&itemkFcMap.size()!=0){

          Map<String,Integer> candidateCollection=getCandidateCollection(itemkFcMap);

          Set<String> ccKeySet=candidateCollection.keySet();

          //对候选集项进行累加计数

          for(String trans:transList){

             for(String candidate:ccKeySet){

                      boolean flag=true;// 用来判断交易中是否出现该候选项，如果出现，计数加1

                      String[] candidateItems=candidate.split(ITEM\_SPLIT);

                      for(String candidateItem:candidateItems){

                               if(trans.indexOf(candidateItem+ITEM\_SPLIT)==-1){

                                         flag=false;

                                         break;

                               }

                      }

                      if(flag){

                               Integer count=candidateCollection.get(candidate);

                               candidateCollection.put(candidate, count+1);

                      }

             }

          }

          //从候选集中找到符合支持度的频繁集项

          itemkFcMap.clear();

          for(String candidate:ccKeySet){

             Integer count=candidateCollection.get(candidate);

             if(count>=SUPPORT){

                 itemkFcMap.put(candidate, count);

             }

          }

          //合并所有频繁集

          frequentCollectionMap.putAll(itemkFcMap);

        }

        return frequentCollectionMap;

         }

         private Map<String,Integer> getCandidateCollection(Map<String,Integer> itemkFcMap){

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

                   Set<String> itemkSet1=itemkFcMap.keySet();

                   Set<String> itemkSet2=itemkFcMap.keySet();

                   for(String itemk1:itemkSet1){

                            for(String itemk2:itemkSet2){

                                     //进行连接

                                     String[] tmp1=itemk1.split(ITEM\_SPLIT);

                                     String[] tmp2=itemk2.split(ITEM\_SPLIT);

                                     String c="";

                                     if(tmp1.length==1){

                                               if(tmp1[0].compareTo(tmp2[0])<0){

                                                        c=tmp1[0]+ITEM\_SPLIT+tmp2[0]+ITEM\_SPLIT;

                                               }

                                     }else{

                                               boolean flag=true;

                    for(int i=0;i<tmp1.length-1;i++){

                           if(!tmp1[i].equals(tmp2[i])){

                                    flag=false;

                                    break;

                           }

                    }

                    if(flag&&(tmp1[tmp1.length-1].compareTo(tmp2[tmp2.length-1])<0)){

                           c=itemk1+tmp2[tmp2.length-1]+ITEM\_SPLIT;

                    }

                                     }

                                     //进行剪枝

                                     boolean hasInfrequentSubSet = false;

                                     if (!c.equals("")) {

                                               String[] tmpC = c.split(ITEM\_SPLIT);

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

                                                        String subC = "";

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

                                                                 if (i != j) {

                                                                           subC = subC+tmpC[j]+ITEM\_SPLIT;

                                                                 }

                                                        }

                                                        if (itemkFcMap.get(subC) == null) {

                                                                 hasInfrequentSubSet = true;

                                                                 break;

                                                        }

                                               }

                                     }else{

                                               hasInfrequentSubSet=true;

                                     }

                                     if(!hasInfrequentSubSet){

                                               candidateCollection.put(c, 0);

                                     }

                            }

                   }

                   return candidateCollection;

         }

         private Map<String,Integer> getItem1FC(){

                   Map<String,Integer> sItem1FcMap=new HashMap<String,Integer>();

                   Map<String,Integer> rItem1FcMap=new HashMap<String,Integer>();//频繁1项集

                   for(String trans:transList){

                            String[] items=trans.split(ITEM\_SPLIT);

                            for(String item:items){

                                     Integer count=sItem1FcMap.get(item+ITEM\_SPLIT);

                                     if(count==null){

                                               sItem1FcMap.put(item+ITEM\_SPLIT, 1);

                                     }else{

                                               sItem1FcMap.put(item+ITEM\_SPLIT, count+1);

                                     }

                            }

                   }

                   Set<String> keySet=sItem1FcMap.keySet();

                   for(String key:keySet){

                            Integer count=sItem1FcMap.get(key);

                            if(count>=SUPPORT){

                                     rItem1FcMap.put(key, count);

                            }

                   }

                   return rItem1FcMap;

         }

         public Map<String,Double> getRelationRules(Map<String,Integer> frequentCollectionMap){

                   Map<String,Double> relationRules=new HashMap<String,Double>();

                   Set<String> keySet=frequentCollectionMap.keySet();

                   for (String key : keySet) {

                            double countAll=frequentCollectionMap.get(key);

                            String[] keyItems = key.split(ITEM\_SPLIT);

                            if(keyItems.length>1){

                                     List<String> source=new ArrayList<String>();

                                     Collections.addAll(source, keyItems);

                                     List<List<String>> result=new ArrayList<List<String>>();

                                     buildSubSet(source,result);//获得source的所有非空子集

                                     for(List<String> itemList:result){

                    if(itemList.size()<source.size()){//只处理真子集

                           List<String> otherList=new ArrayList<String>();

                           for(String sourceItem:source){

                                    if(!itemList.contains(sourceItem)){

                                             otherList.add(sourceItem);

                                    }

                           }

                        String reasonStr="";//前置

                        String resultStr="";//结果

                        for(String item:itemList){

                                reasonStr=reasonStr+item+ITEM\_SPLIT;

                        }

                        for(String item:otherList){

                                resultStr=resultStr+item+ITEM\_SPLIT;

                        }

                        double countReason=frequentCollectionMap.get(reasonStr);

                        double itemConfidence=countAll/countReason;//计算置信度

                        if(itemConfidence>=CONFIDENCE){

                                String rule=reasonStr+CON+resultStr;

                                relationRules.put(rule, itemConfidence);

                        }

                    }

                                     }

                            }

                   }

                   return relationRules;

         }

         private  void buildSubSet(List<String> sourceSet, List<List<String>> result) {

                   // 仅有一个元素时，递归终止。此时非空子集仅为其自身，所以直接添加到result中

                   if (sourceSet.size() == 1) {

                            List<String> set = new ArrayList<String>();

                            set.add(sourceSet.get(0));

                            result.add(set);

                   } else if (sourceSet.size() > 1) {

                            // 当有n个元素时，递归求出前n-1个子集，在于result中

                            buildSubSet(sourceSet.subList(0, sourceSet.size() - 1), result);

                            int size = result.size();// 求出此时result的长度，用于后面的追加第n个元素时计数

                            // 把第n个元素加入到集合中

                            List<String> single = new ArrayList<String>();

                            single.add(sourceSet.get(sourceSet.size() - 1));

                            result.add(single);

                            // 在保留前面的n-1子集的情况下，把第n个元素分别加到前n个子集中，并把新的集加入到result中;

                            // 为保留原有n-1的子集，所以需要先对其进行复制

                            List<String> clone;

                            for (int i = 0; i < size; i++) {

                                     clone = new ArrayList<String>();

                                     for (String str : result.get(i)) {

                                               clone.add(str);

                                     }

                                     clone.add(sourceSet.get(sourceSet.size() - 1));

                                     result.add(clone);

                            }

                   }

         }

         public static void main(String[] args){

                   Apriori apriori=new Apriori();

                   Map<String,Integer> frequentCollectionMap=apriori.getFC();

                   System.out.println("----------------频繁集"+"----------------");

                   Set<String> fcKeySet=frequentCollectionMap.keySet();

                   for(String fcKey:fcKeySet){

                            System.out.println(fcKey+"  :  "+frequentCollectionMap.get(fcKey));

                   }

        Map<String,Double> relationRulesMap=apriori.getRelationRules(frequentCollectionMap);

        System.out.println("----------------关联规则"+"----------------");

        Set<String> rrKeySet=relationRulesMap.keySet();

        for(String rrKey:rrKeySet){

                            System.out.println(rrKey+"  :  "+relationRulesMap.get(rrKey));

                   }

         }

}