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 L_1，然后利用 L_1 找频繁 2 项集的集合 L_2，L_2 找 L_3，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 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 L_k（所有的频繁 k 项集的集合），通过将 L_{k-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 C_k。设 l_1 和 l_2 是 L_{k-1} 中的成员。记 $l_i[j]$表示 l_i 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 l_i，$l_i[1]<l_i[2]<\ldots\ldots\ldots<l_i[k-1]$。将 L_{k-1} 与自身连接，如果$(l_1[1]=l_2[1])\&\&(l_1[2]=l_2[2])\&\&\ldots\ldots..\&\&(l_1[k-2]=l_2[k-2])\&\&(l_1[k-1]<l_2[k-1])$，那认为 l_1 和 l_2 是可连接。连接 l_1 和 l_2 产生的结果是 $\{l_1[1],l_1[2],\ldots\ldots,l_1[k-1],l_2[k-1]\}$。

2） 剪枝步

C_K 是 L_K 的超集，也就是说，C_K 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 C_K 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 C_k,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 C_K 中删除。

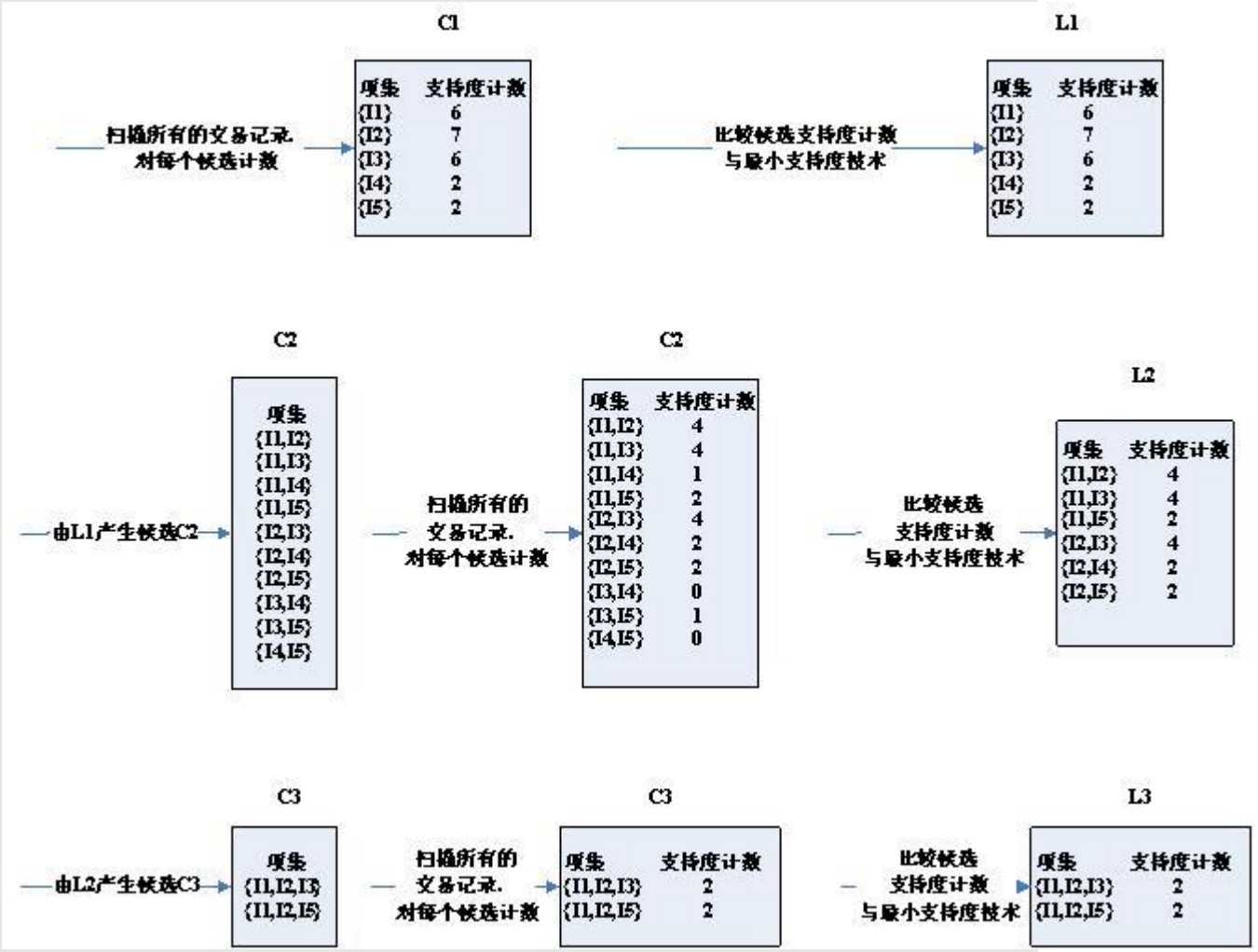
（Tip：为什么要压缩 C_K 呢？因为实际情况下事务记录往往是保存在外存储上，比如数据库或者其他格式的文件上，在每次计算候选计数时都需要将候选与所有事务进行比对，众所周知，访问外存的效率往往都比较低，因此 Apriori 加入了所谓的剪枝步，事先对候选集进行过滤，以减少访问外存的次数。）

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:



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：

L_1=find_frequent_1-itemsets(D); // 找出所有频繁 1 项集

For(k=2;L_{k-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));
            }
    }
}
```