Weka[24] Apriori 源代码分析

作者：Koala++/屈伟

曾经卖过一个 Apriori 的程序，那个程序大约有 50%的正确率(当然结果是正确的，是只实现上很不一样)，数据挖掘课上写了一个 Apriori，一部分懒地按书上的算法，大约对了 80%(当然结果仍然是正确的)，记得邱强有一次要用 Apriori 算法时说：weka 的太慢了，还好上次数据挖掘课实现了一下，还挺快的，注意的一点是关联规则不属于机器学习，这里我不想再分出来一个数据挖掘的组了。

从 buildAssociations 函数开始：

```java
double[] confidences, supports;
int[] indices;
FastVector[] sortedRuleSet;
int necSupport = 0;

instances = new Instances(instances);

if (m_removeMissingCols) {
    instances = removeMissingColumns(instances);
}
```

看一下 removeMissingColumns，虽然它是如此的不重要：

```java
protected Instances removeMissingColumns(Instances instances)
        throws Exception {

    int numInstances = instances.numInstances();
    StringBuffer deleteString = new StringBuffer();
    int removeCount = 0;
    boolean first = true;
    int maxCount = 0;

    for (int i = 0; i < instances.numAttributes(); i++) {
        AttributeStats as = instances.attributeStats(i);
        if (m_upperBoundMinSupport == 1.0 && maxCount != numInstances) {
            // see if we can decrease this by looking for the most frequent
            // value
            int[] counts = as.nominalCounts;
            if (counts[Utils.maxIndex(counts)] > maxCount) {
                maxCount = counts[Utils.maxIndex(counts)];
            }
        }
        if (as.missingCount == numInstances) {
            if (first) {
                deleteString.append((i + 1));
                first = false;
            } else {
                deleteString.append("," + (i + 1));
            }
            removeCount++;
        }
    }
    if (m_verbose) {
        System.err.println("Removed : " + removeCount
                + " columns with all missing " + "values.");
    }
    if (m_upperBoundMinSupport == 1.0 && maxCount != numInstances) {
        m_upperBoundMinSupport = (double) maxCount
                                        / (double) numInstances;
        if (m_verbose) {
            System.err.println("Setting upper bound min support to : "
                    + m_upperBoundMinSupport);
        }
    }

    if (deleteString.toString().length() > 0) {
        Remove af = new Remove();
        af.setAttributeIndices(deleteString.toString());
        af.setInvertSelection(false);
        af.setInputFormat(instances);
        Instances newInst = Filter.useFilter(instances, af);

        return newInst;
    }
    return instances;
}
```

For 循环中的第一个 if 不重要，不要理睬。它的作用是想在后面也就是 maxCount / numInstances 那看一下 support 的下界设多少。第二个 if 中是看是不是缺失值等于样本数，也就是在这个属性上所有的值都是缺失的，那么就用 deleteString 把这些都是缺失值的属性下标记下来，连成一个字符串，最后一个 if 就是标准的删除特征的代码，也就是把那么都是缺失值的特征给删除了。

```
if (m_car && m_metricType != CONFIDENCE)
    throw new Exception(
            "For CAR-Mining metric type has to be confidence!");
```

如果想得到与类别有关的规则，又没有设成 CONFIDENCE 是不可以的。

```
// only set class index if CAR is requested
if (m_car) {
    if (m_classIndex == -1) {
        instances.setClassIndex(instances.numAttributes() - 1);
    } else if (m_classIndex <= instances.numAttributes()
            && m_classIndex > 0) {
        instances.setClassIndex(m_classIndex - 1);
    } else {
        throw new Exception("Invalid class index.");
    }
}
```

如果用用户想得到与类别有关的规则又忘了设类别索引，就帮他设一下。

```
// can associator handle the data?
getCapabilities().testWithFail(instances);
```

看能不能外理这种数据。

```
m_cycles = 0;
if (m_car) {
    // m_instances does not contain the class attribute
    m_instances = LabeledItemSet.divide(instances, false);

    // m_onlyClass contains only the class attribute
    m_onlyClass = LabeledItemSet.divide(instances, true);
} else
    m_instances = instances;
```

```
if (m_car && m_numRules == Integer.MAX_VALUE) {
    // Set desired minimum support
    m_minSupport = m_lowerBoundMinSupport;
} else {
    // Decrease minimum support until desired number of rules found.
    m_minSupport = m_upperBoundMinSupport - m_delta;
    m_minSupport = (m_minSupport < m_lowerBoundMinSupport) ?
    m_lowerBoundMinSupport : m_minSupport;
}
```

不再看 LabeledItemSet.devide，只看一下注释，m_instances 不包含类别属性，而 m_onlyClass 只包含类别属性。下面的就不去管它了。

```
do {
    // Reserve space for variables
    m_Ls = new FastVector();
    m_hashtables = new FastVector();
    m_allTheRules = new FastVector[6];
    m_allTheRules[0] = new FastVector();
    m_allTheRules[1] = new FastVector();
    m_allTheRules[2] = new FastVector();
    if (m_metricType != CONFIDENCE || m_significanceLevel != -1) {
        m_allTheRules[3] = new FastVector();
        m_allTheRules[4] = new FastVector();
        m_allTheRules[5] = new FastVector();
    }
    sortedRuleSet = new FastVector[6];
    sortedRuleSet[0] = new FastVector();
    sortedRuleSet[1] = new FastVector();
    sortedRuleSet[2] = new FastVector();
    if (m_metricType != CONFIDENCE || m_significanceLevel != -1) {
        sortedRuleSet[3] = new FastVector();
        sortedRuleSet[4] = new FastVector();
        sortedRuleSet[5] = new FastVector();
    }
    if (!m_car) {
        // Find large itemsets and rules
        findLargeItemSets();
        if (m_significanceLevel != -1 || m_metricType != CONFIDENCE)
            findRulesBruteForce();
        else
            findRulesQuickly();
    } else {
        findLargeCarItemSets();
        findCarRulesQuickly();
    }
```

上面都是一些初始化的代码，不讲了，如果想知道，可以看一下我以前写的那一篇 Weka 开发 [17] ——关联规则之 Apriori。再下来，如果不是挖掘与类别相关的规则，那么先执行 findLargeItermSets:

```
private void findLargeItemSets() throws Exception {

    FastVector kMinusOneSets, kSets;
    Hashtable hashtable;
    int necSupport, necMaxSupport, i = 0;

    // Find large itemsets

    // minimum support
    necSupport = (int) (m_minSupport
            * (double) m_instances.numInstances() + 0.5);
    necMaxSupport = (int) (m_upperBoundMinSupport
            * (double) m_instances.numInstances() + 0.5);

    kSets = AprioriItemSet.singletons(m_instances);
    AprioriItemSet.upDateCounters(kSets, m_instances);
    kSets = AprioriItemSet.deleteItemSets(kSets, necSupport,
        necMaxSupport);
    if (kSets.size() == 0)
        return;
    do {
        m_Ls.addElement(kSets);
        kMinusOneSets = kSets;
        kSets = AprioriItemSet.mergeAllItemSets(kMinusOneSets, i,
                m_instances.numInstances());
        hashtable = AprioriItemSet.getHashtable(kMinusOneSets,
                kMinusOneSets.size());
        m_hashtables.addElement(hashtable);
        kSets = AprioriItemSet.pruneItemSets(kSets, hashtable);
        AprioriItemSet.upDateCounters(kSets, m_instances);
        kSets = AprioriItemSet.deleteItemSets(kSets, necSupport,
                necMaxSupport);
        i++;
    } while (kSets.size() > 0);
}
```

NecSupprot 是最小 support 的另一种衡量，就是计数，原来的 m_minSupport 是用百分比逢的，+0.5 当然就是四舍五入了。necMaxSupport 也是相同的意思。

下面看 AprioriIltermSet.singletons，AprioriItermSet 是继承自 ItemSet 的：

```
/**
 * Converts the header info of the given set of instances into a set of
 * item sets (singletons). The ordering of values in the header file
 * determines the lexicographic order.
 */
public static FastVector singletons(Instances instances)
throws Exception {

    FastVector setOfItemSets = new FastVector();
    ItemSet current;

    for (int i = 0; i < instances.numAttributes(); i++) {
        if (instances.attribute(i).isNumeric())
            throw new Exception("Can't handle numeric attributes!");
        for (int j = 0; j < instances.attribute(i).numValues(); j++) {
            current = new AprioriItemSet(instances.numInstances());
            current.m_items = new int[instances.numAttributes()];
            for (int k = 0; k < instances.numAttributes(); k++)
                current.m_items[k] = -1;
            current.m_items[i] = j;
            setOfItemSets.addElement(current);
        }
    }
    return setOfItemSets;
}
```

这段代码的作用现在还看不出来，可以看一下注释，意思是：将给定数据集的头信息转换成一个项集的集合，头信息中的值的顺序是按字典序。如果你也是用 weather.nominal 数

据集，那么它产生的结果应该是这个：

```
0,-1,-1,-1,-1
1,-1,-1,-1,-1
2,-1,-1,-1,-1
-1,0,-1,-1,-1
-1,1,-1,-1,-1
-1,2,-1,-1,-1
-1,-1,0,-1,-1
-1,-1,1,-1,-1
-1,-1,-1,0,-1
-1,-1,-1,1,-1
-1,-1,-1,-1,0
-1,-1,-1,-1,1
```

一共有 12 组，因为有 2 个属性有 3 种属性值，3 个属性有 2 种属性值，2*3+3*2=12。而且 m_iterm 的第 i 个元素值设为 j。

接下来看 AprioriItemSet.upDateCounters：

```
public static void upDateCounters(FastVector itemSets, Instances
instances) {

    for (int i = 0; i < instances.numInstances(); i++) {
        Enumeration enu = itemSets.elements();
        while (enu.hasMoreElements())
            ((ItemSet) enu.nextElement()).upDateCounter(instances
                    .instance(i));
    }
}
```

外层循环是样本的个数，内层循环是 itermSets 的数量。里面的 upDateCounter：

```
public void upDateCounter(Instance instance) {
    if (containedBy(instance))
        m counter++;
}
```

也就是如果满足 containBy 那么就计数加 1，看一下 containBy：

```
public boolean containedBy(Instance instance) {
    for (int i = 0; i < instance.numAttributes(); i++)
        if (m_items[i] > -1) {
            if (instance.isMissing(i))
                return false;
            if (m_items[i] != (int) instance.value(i))
                return false;
        }
    return true;
}
```

这段代码就相对容易一点了，如果在相应的特征值上为缺失值，或是与我们要找的特征词不相同，那么就返回 false。

那么在 upDateCounters 执行完之后，结果大概是这样的，最后的是记数：

```
0,-1,-1,-1,-1 : 5
1,-1,-1,-1,-1 : 4
2,-1,-1,-1,-1 : 5
-1,0,-1,-1,-1 : 4
-1,1,-1,-1,-1 : 6
-1,2,-1,-1,-1 : 4
-1,-1,0,-1,-1 : 7
-1,-1,1,-1,-1 : 7
-1,-1,-1,0,-1 : 6
-1,-1,-1,1,-1 : 8
```

```
-1,-1,-1,-1,0 : 9
-1,-1,-1,-1,1 : 5
```

这样也就理解了，m_item 的意义，也就是现在是 1 项集。

接下来看 deleteItemSets：

```
public static FastVector deleteItemSets(FastVector itemSets,
        int minSupport, int maxSupport) {

    FastVector newVector = new FastVector(itemSets.size());

    for (int i = 0; i < itemSets.size(); i++) {
        ItemSet current = (ItemSet) itemSets.elementAt(i);
        if ((current.m_counter >= minSupport)
                && (current.m_counter <= maxSupport))
            newVector.addElement(current);
    }
    return newVector;
}
```

这里可以看到，只有 m_counter 在 minSupport 和 maxSupport 之间的项我们才要。

接下来就要看 findLargeItemSets 这个函数中的 do/while 循环了，m_Ls 是保存所有项的一个集合，那 kMinusOneSets 就是上一个项集，接下来是 AprioriItemSet.mergeAllItemSets，但是这个函数在执行第一次的时候，它并看不出来什么作用：

```
public static FastVector mergeAllItemSets(FastVector itemSets, int size,
        int totalTrans) {

    FastVector newVector = new FastVector();
    ItemSet result;
    int numFound, k;

    for (int i = 0; i < itemSets.size(); i++) {
        ItemSet first = (ItemSet) itemSets.elementAt(i);
        out: for (int j = i + 1; j < itemSets.size(); j++) {
            ItemSet second = (ItemSet) itemSets.elementAt(j);
            result = new AprioriItemSet(totalTrans);
            result.m_items = new int[first.m_items.length];

            // Find and copy common prefix of size 'size'
            numFound = 0;
            k = 0;
            while (numFound < size) {
                if (first.m_items[k] == second.m_items[k]) {
                    if (first.m_items[k] != -1)
                        numFound++;
                    result.m_items[k] = first.m_items[k];
                } else
                    break out;
                k++;
            }

            // Check difference
            while (k < first.m_items.length) {
                if ((first.m_items[k] != -1) && (second.m_items[k] != -1))
                    break;
                else {
                    if (first.m_items[k] != -1)
                        result.m_items[k] = first.m_items[k];
                    else
```

```
                        result.m_items[k] = second.m_items[k];
                }
                k++;
            }
            if (k == first.m_items.length) {
                result.m_counter = 0;
                newVector.addElement(result);
            }
        }
    }
    return newVector;
}
```

这里有一个 out 标签，也就是 break out 执行后会跳到 out 那里接着执行，很象 goto。这里的 size 是我们所需要的多少项的，这里 first.m_items[k]要等于 second.m_items[k]的原因是在相应的特征上值一定要相同才可以合并成一个新的规则，并且要注意，在注释的那一行中也提到了，就是前面部分一定要一样，大小为 size-1。也就是 first.m_item[1] = second.m_items[1]，first.m_item[2] = second.m_items[2]，……，first.m_item[size-1] = second.m_items[size-1]。而在下面就是给 result.items 赋值余下的部分赋值，这里 first.m_items[k]与 second.m_items[k]不能同时为-1，因为同时为-1 又多出来一项，所以就要去除这种情况。

下面是 IntermSet.getHashtable：

```
public static Hashtable getHashtable(FastVector itemSets, int
initialSize) {

    Hashtable hashtable = new Hashtable(initialSize);

    for (int i = 0; i < itemSets.size(); i++) {
        ItemSet current = (ItemSet) itemSets.elementAt(i);
        hashtable.put(current, new Integer(current.m_counter));
    }
    return hashtable;
}
```

Hashtable 的 key 是一个 ItemSet，而值是它的计数。下面是 ItemSet.pruneItemSet：

```
public static FastVector pruneItemSets(FastVector toPrune,
        Hashtable kMinusOne) {

    FastVector newVector = new FastVector(toPrune.size());
    int help, j;

    for (int i = 0; i < toPrune.size(); i++) {
        ItemSet current = (ItemSet) toPrune.elementAt(i);
        for (j = 0; j < current.m_items.length; j++)
            if (current.m_items[j] != -1) {
                help = current.m_items[j];
                current.m_items[j] = -1;
                if (kMinusOne.get(current) == null) {
                    current.m_items[j] = help;
                    break;
                } else {
                    current.m_items[j] = help;
                }
            }
        if (j == current.m_items.length)
            newVector.addElement(current);
    }
    return newVector;
}
```

pruneItemSets 里面的前面这一段代码，主要是为了让 m_item[j]这项为-1 来，再用 kMinusOne.get，如果为空，那么就 break，如果这项中每一个(n-1)个特征值都是存在的，那么才将它放到 newVector 中去。似乎完全不明白这在做什么，现在看 findRulesQuickly：

```
private void findRulesQuickly() throws Exception {

    FastVector[] rules;

    // Build rules
    for (int j = 1; j < m_Ls.size(); j++) {
        FastVector currentItemSets = (FastVector) m_Ls.elementAt(j);
        Enumeration enumItemSets = currentItemSets.elements();
        while (enumItemSets.hasMoreElements()) {
            AprioriItemSet currentItemSet = (AprioriItemSet) enumItemSets
                    .nextElement();
            // AprioriItemSet currentItemSet = new
            // AprioriItemSet((ItemSet)enumItemSets.nextElement());
            rules = currentItemSet.generateRules(m_minMetric,
                m_hashtables,j + 1);
            for (int k = 0; k < rules[0].size(); k++) {
                m_allTheRules[0].addElement(rules[0].elementAt(k));
                m_allTheRules[1].addElement(rules[1].elementAt(k));
                m_allTheRules[2].addElement(rules[2].elementAt(k));
            }
        }
    }
}
```

现在的问题是，如果这是第一次执行，那么我们的 m_Ls 大小为 1，也就是什么都不执行，直接跳过去了。其中的 generateRules 如下：

```
public FastVector[] generateRules(double minConfidence,
        FastVector hashtables, int numItemsInSet) {

    FastVector premises = new FastVector(), consequences = new
        FastVector(), conf = new FastVector();
    FastVector[] rules = new FastVector[3], moreResults;
    AprioriItemSet premise, consequence;
    Hashtable hashtable = (Hashtable) hashtables
            .elementAt(numItemsInSet - 2);

    // Generate all rules with one item in the consequence.
    for (int i = 0; i < m_items.length; i++)
        if (m_items[i] != -1) {
            premise = new AprioriItemSet(m_totalTransactions);
            consequence = new AprioriItemSet(m_totalTransactions);
            premise.m_items = new int[m_items.length];
            consequence.m_items = new int[m_items.length];
            consequence.m_counter = m_counter;

            for (int j = 0; j < m_items.length; j++)
                consequence.m_items[j] = -1;
            System.arraycopy(m_items, 0, premise.m_items, 0,
                    m_items.length);
            premise.m_items[i] = -1;

            consequence.m_items[i] = m_items[i];
            premise.m_counter = ((Integer) hashtable.get(premise))
                    .intValue();
            premises.addElement(premise);
            consequences.addElement(consequence);
            conf.addElement(new Double(confidenceForRule(premise,
                    consequence)));
        }
    rules[0] = premises;
    rules[1] = consequences;
    rules[2] = conf;
    pruneRules(rules, minConfidence);

    // Generate all the other rules
    moreResults = moreComplexRules(rules, numItemsInSet, 1,
        minConfidence,
            hashtables);
    if (moreResults != null)
        for (int i = 0; i < moreResults[0].size(); i++) {
            rules[0].addElement(moreResults[0].elementAt(i));
            rules[1].addElement(moreResults[1].elementAt(i));
            rules[2].addElement(moreResults[2].elementAt(i));
        }
    return rules;
}
```

其中比较重要的有两句就是 premise.m_items[i]=-1 表示我们把 premise 去掉一项，还将这一项做为 consequence，其中的 confidenceForRule 没什么好讲的，就是公式：

```
public static double confidenceForRule(AprioriItemSet premise,
        AprioriItemSet consequence) {

    return (double) consequence.m_counter / (double) premise.m_counter;
}
```

而 rules 这里可以看出来，三个元素就是关联规则的左部，右部，和置信度。

```
public static void pruneRules(FastVector[] rules, double minConfidence)
{

    FastVector newPremises = new FastVector(rules[0].size()),
         newConsequences = new FastVector(rules[1].size()),
         newConf = new FastVector(rules[2].size());

    for (int i = 0; i < rules[0].size(); i++)
        if (!(((Double) rules[2].elementAt(i)).doubleValue() <
             minConfidence)) {
            newPremises.addElement(rules[0].elementAt(i));
            newConsequences.addElement(rules[1].elementAt(i));
            newConf.addElement(rules[2].elementAt(i));
        }
    rules[0] = newPremises;
    rules[1] = newConsequences;
    rules[2] = newConf;
}
```

这里对规则进行裁减，如果小于 minConfidence 就将这个规则删除。

```
private final FastVector[] moreComplexRules(FastVector[] rules,
        int numItemsInSet, int numItemsInConsequence,
        double minConfidence,FastVector hashtables) {

    AprioriItemSet newPremise;
    FastVector[] result, moreResults;
    FastVector newConsequences, newPremises = new FastVector(),
        newConf = new FastVector();
    Hashtable hashtable;

    if (numItemsInSet > numItemsInConsequence + 1) {
        hashtable = (Hashtable) hashtables.elementAt(numItemsInSet
                - numItemsInConsequence - 2);
        newConsequences = mergeAllItemSets(rules[1],
                numItemsInConsequence - 1, m_totalTransactions);
        Enumeration enu = newConsequences.elements();
        while (enu.hasMoreElements()) {
            AprioriItemSet current = (AprioriItemSet) enu.nextElement();
            current.m_counter = m_counter;
            newPremise = subtract(current);
            newPremise.m_counter = ((Integer) hashtable.get(newPremise))
                    .intValue();
            newPremises.addElement(newPremise);
            newConf.addElement(new Double(confidenceForRule(newPremise,
                    current)));
        }
        result = new FastVector[3];
        result[0] = newPremises;
        result[1] = newConsequences;
        result[2] = newConf;
        pruneRules(result, minConfidence);
        moreResults = moreComplexRules(result, numItemsInSet,
                numItemsInConsequence + 1, minConfidence, hashtables);
        if (moreResults != null)
            for (int i = 0; i < moreResults[0].size(); i++) {
                result[0].addElement(moreResults[0].elementAt(i));
                result[1].addElement(moreResults[1].elementAt(i));
                result[2].addElement(moreResults[2].elementAt(i));
            }
        return result;
    } else
        return null;
}
```

这里的 mergeAllItemSets 把所有 numItemInConsequnce 个的项得到了，然后对得到的 newConsequences 进行循环，下面有一个 subtract：

```
public final AprioriItemSet subtract(AprioriItemSet toSubtract) {

    AprioriItemSet result = new AprioriItemSet(m_totalTransactions);

    result.m_items = new int[m_items.length];

    for (int i = 0; i < m_items.length; i++)
        if (toSubtract.m_items[i] == -1)
            result.m_items[i] = m_items[i];
        else
            result.m_items[i] = -1;
    result.m_counter = 0;
    return result;
}
```

也就是如果这里是当 toSubtract 中的不为-1 的值，将值换为-1，如果 toSubtract 中相应的值为-1，那么就保留原值。下面有一个递归的过程，就是将 numItemsInConsequence+1，

也就是 Consequence 中的项数可以是 numItemsInConsequence+1 个了。而它的值不能大于 numTermsInSet。最后在 generateRules 把两个结果合并了。

接下来则应该是对规则结果进行排序，先是对 support 进行排序，再对 confidence 进行排序，这里之所以要 supports[j-i]这样反着写是因为 stableSort 结果是升序的，stable 的意思是稳定，懒地解释了，不懂看一下数据结构吧：

```
// Sort rules according to their support
int j = m_allTheRules[2].size() - 1;
supports = new double[m_allTheRules[2].size()];
for (int i = 0; i < (j + 1); i++)
    supports[j - i] = ((double) ((ItemSet) m_allTheRules[1]
            .elementAt(j - i)).support()) * (-1);
indices = Utils.stableSort(supports);
for (int i = 0; i < (j + 1); i++) {
    sortedRuleSet[0].addElement(m_allTheRules[0]
            .elementAt(indices[j - i]));
    sortedRuleSet[1].addElement(m_allTheRules[1]
            .elementAt(indices[j - i]));
    sortedRuleSet[2].addElement(m_allTheRules[2]
            .elementAt(indices[j - i]));
    if (m_metricType != CONFIDENCE || m_significanceLevel != -1) {
        sortedRuleSet[3].addElement(m_allTheRules[3]
                .elementAt(indices[j - i]));
        sortedRuleSet[4].addElement(m_allTheRules[4]
                .elementAt(indices[j - i]));
        sortedRuleSet[5].addElement(m_allTheRules[5]
                .elementAt(indices[j - i]));
    }
}

// Sort rules according to their confidence
m_allTheRules[0].removeAllElements();
m_allTheRules[1].removeAllElements();
m_allTheRules[2].removeAllElements();
if (m_metricType != CONFIDENCE || m_significanceLevel != -1) {
    m_allTheRules[3].removeAllElements();
    m_allTheRules[4].removeAllElements();
    m_allTheRules[5].removeAllElements();
}
confidences = new double[sortedRuleSet[2].size()];
int sortType = 2 + m_metricType;

for (int i = 0; i < sortedRuleSet[2].size(); i++)
    confidences[i] = ((Double) sortedRuleSet[sortType].elementAt(i))
            .doubleValue();
indices = Utils.stableSort(confidences);
for (int i = sortedRuleSet[0].size() - 1; (i >= (sortedRuleSet[0]
        .size() - m_numRules))
        && (i >= 0); i--) {
    m_allTheRules[0].addElement(sortedRuleSet[0]
            .elementAt(indices[i]));
    m_allTheRules[1].addElement(sortedRuleSet[1]
            .elementAt(indices[i]));
    m_allTheRules[2].addElement(sortedRuleSet[2]
            .elementAt(indices[i]));
    if (m_metricType != CONFIDENCE || m_significanceLevel != -1) {
        m_allTheRules[3].addElement(sortedRuleSet[3]
                .elementAt(indices[i]));
        m_allTheRules[4].addElement(sortedRuleSet[4]
                .elementAt(indices[i]));
        m_allTheRules[5].addElement(sortedRuleSet[5]
                .elementAt(indices[i]));
    }
}
```

关于对类别规则的分析，我现在用不着，也就没分析。