基于类关联规则的分类器的实现

刘砺志　　　陈大松　　　肖璐菁

22920142203873　22920142203785　22920142203933

{22920142203873, 22920142203785, 22920142203933}@stu.xmu.edu.cn

摘　要：分类问题就是确定对象属于哪一个预定义的目标类的过程。传统的关联分析通过挖掘隐藏在大型数据集背后的满足最小支持度和最小置信度约束的关联规则，来描述数据集中存在的有意义的联系，但是所发掘的联系都是事先无法预知的。通过将分类任务和关联分析相结合，我们利用一个关联规则的特殊子集，称为类关联规则，来训练一个分类器，对记录进行分类。实验结果表明，使用这种技术训练出的分类器，其准确率一般来说都是很高的。

关键字：分类；类关联规则；关联分析

An Implementation of Classifier Based on Class Association Rules

Lizhi Liu　　Dasong Chen　　Lujing Xiao

22920142203873　22920142203785　22920142203933

{22920142203873, 22920142203785, 22920142203933}@stu.xmu.edu.cn

Abstract: The classification problem is the process of determining which pre-defined target class the object belongs to. The traditional association analysis describes the meaningful links in the data set by mining the association rules that satisfy the minimum support and the minimum confidence constraint behind the large data set, but those links are unpredictable in advance. By combining classification task with association analysis, we use a special subset of association rules, called class association rules, to train a classifier to classify records. Experimental results show that the use of this technology training out of the classifier, the accuracy rate is very high in general.

Keywords: Classification; Class Association Rules; Association Analysis;

1 引言

传统的关联规则挖掘意在从海量数据中挖掘出有意义的规则，以描述数据之间存在的相互联系。而分类问题则是利用分类器，判断一个样本究竟应该归为哪一个预定义的目标类。对于前者，所能发现的规则是无法事先知晓的；对于后者，目标类已经事先划定并且最终的判定结果只能有一个。因而合理的整合两种技术，以构造一个更为强大的分类器，成为了我们这篇文章所要实现的东西。本文中通过修改 Apriori 算法 [3]，寻找一组特殊的频繁项集，以生成一组特殊的关联规则，称为类关联规则。我们可以使用这些类关联规则，去训练一个基于规则的分类器，实现对样本的分类。

由于现实中存在的数据集完整程度高低有别，各属性值类型也千差万别，在挖掘类关联规则之前，我们必须要对数据进行预处理。预处理过程主要包括两个部分。一个是对缺失值进行处理。对于缺失值较多的特征，我们直接舍弃；对于缺失值较少的特征，我们可以把缺失本事直接作为一个特征，或是使用均值、上下文数据、众数法进行填充，也可以使用插值、随机森林算法 [4] 拟合等方式填补。第二个问题是对连续型属性进行离散化的问题。文献 [5] 和 [6] 提出了基于信息熵的的连续型属性值离散化方法，文献 [7] 中对这项工作进行了很好的概述。我们将在第 2 部分第 1 节详细介绍我们所采用的方法。

本文的重点和难点在于如何生成类关联规则并利用这些规则训练出性能优异的分类器。文献 [1] 提出的 CBA 算法很好的解决了这个问题。这个算法包含两个部分。第一部分是规则生成，即 CBA-CG 算法。这个算法在 Apriori 算法的基础上进行修改，以生成一组类关联规则而非传统的关联规则。这一部分实现的关键在于如何筛选出候选项集，以及如何利用悲观误差进行规则剪枝。这一部分我们将在第 2 部分第 2 节进行讨论。算法的第二部分就是训练分类器。刘兵等人首先提出了一个简单的 CBA-CB M1 算法，利用贪心策略，按序逐步抽取规则，最后构建一个有序规则集来作为最终的分类器。这一部分我们将在第 2 部分第 3 节进行讨论。但是这个算法对于大样本情形并不适用，因此又提出了改进版的 CBA-CB M2 算法。这个算法包括 3 个阶段。第 1 阶段对类关联规则进行分类，缩小最终用于生成分类器的规则数。第 2 阶段对未决规则进行裁决，确定最后用于分类器生成的规则。第 3 阶段则在筛选出的那一部分规则中提取用于分类器的那些规则。这一部分的详细讨论将在第 2 部分第 4 节进行。

完成了分类器的生成，就需要对其性能进行检验。我们选取了 UCI 机器学习知识库 [2] 中的 30 个数据集，进行了 10 折交叉检验 [8]。通过检验结果分析，我们所实现的分类器的判定精度极高。这一部分的分析讨论详见本文第 3 部分。

2 方法

2.1 数据预处理

从实际中采集到的数据往往存在着格式不统一、属性值缺失、噪声点等一系列问题。尽管我们实现时所使用的数据都是从 UCI 机器学习资源库中下载的，这些数据已经尽管了相关人员的一些预处理，但是仍然存在两个主要问题：a) 存在缺失值和 b) 连续型属性值离散化。下面我们将就这两个方面进行介绍。

2.1.1 缺失值处理

一般来说，对于缺失值，我们要分两种情况去考虑。对于缺失值较多的属性，我们应当毫无保留的删去，因为这样属性的存在将会对最后分类器的性能带来极大影响。对于缺失值较少的属性，则应当考虑保留。我们可以采取使用均值、众数、上下文数据（即紧邻该样本的上一条或下一条不含缺失值的记录）进行填充这样简单的策略，也可以使用插值或是使用随机森林方法进行拟合，完成数据的填补。

在本文中，由于我们的工作重心是在 CBA 算法的实现而非数据的预处理，因此我们选取了简单的策略：对于缺失值较少的特征，利用缺失值所属特征的众数进行填补；对于缺失值较多的特征，则丢弃这一列。这里选取的缺失率阈值 $\varepsilon = 0.5$。即

$$m = \begin{cases} mode(\{x : x \in col[m] \wedge x \neq ?\}) & ratio < \varepsilon \\ \text{discard } col[m] & \text{otherwise} \end{cases}$$

上式中，m 表示缺失值；$mode(\cdot)$ 操作表示取 $\cdot$ 的众数（若存在多个众数，则随机选取一个）；$col[m]$ 表示 m 所属的列；? 指代缺失值；$ratio$ 表示缺失率，其计算方式为 $ratio = \#m/N$，其中 $\#m$ 表示缺失值的个数，N 表示这一列涵盖缺失值下的总样本数。

2.1.2 连续型属性值的离散化

所谓连续性属性值，就是该属性的取值范围并非有限集。与之相对则是离散型的特征，其取值只可能落在一个有限集合中。在实际的数据集中，连续型属性值非常常见，比如商品的价格、某地的气温等等。但是在经典的决策树模型中，我们只能处理离散型的特征。因此我们需要对连续型属性进行离散化。例如对于商品的价格，在我们初始数据中，可能是具体的数字，但是经过我们的处理后，它就变成了便宜、中等、昂贵三个层次，用形式化的语言描述就是完成这样的映射：

$$\mathbb{R}^+ \to \{\text{cheap}, \text{medium}, \text{expensive}\}$$

文献 [7] 总结了文献 [5][6] 提出的递归的最小熵划分算法。它通过对该属性的取值范围不断进行划分，分隔出几个区间，每个区间对应于一个离散值，由此完成离散化。这个算法主要思想有两点：a) 使用熵去衡量一次划分的优劣，即这次划分会否将数据集的信息量最大化，以及 b) 划分不能无限制进行下去，以避免过拟合的出现。

给定样本集 S，特征 A 以及划分边界 T，由边界 T 引起的划分下的类信息熵定义为

$$E(A, T; S) = \frac{|S_1|}{|S|} Ent(S_1) + \frac{|S_2|}{|S|} Ent(S_2) \tag{1}$$

对于给定的特征 A，在所有划分中能够使得上述熵值最小化的那个划分边界 T_{min}，将被挑选出来作为这一轮划分的二元划分边界。事实上，这个定义就是将划分后产生的两个子集的信息熵进行加权平均，即考虑划分产生的两个子集的综合效果。在本文中，我们采用的信息熵定义为

$$Ent(S) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

其中，n 表示样本集 S 中的类的种数，$P(x_i)$ 表示样本集中第 i 类出现的概率，在实际计算时使用频率以近似替代。

但是作为递归程序，必须还要设计递归出口，否则划分就将无休止的持续下去。为此，Fayyad 和 Irani 提出了最小描述长度准则，来决定何时停止递归划分。这个准则告诉我们，对于集合 S 的递归划分停止，当且仅当

$$Gain(A, T; S) < \frac{\log_2(N-1)}{N} + \frac{\Delta(A, T; S)}{N} \tag{2}$$

成立。其中 N 是集合 S 的元素个数；信息增益 $Gain(A, T; S)$ 定义为

$$Gain(A, T; S) = Ent(S) - E(A, T; S)$$

即划分前后的信息量的增量；余量

$$\Delta(A,T;S)=\log_2(3^k-2)-[k\cdot Ent(S)-k_1\cdot Ent(S_1)-k_2\cdot Ent(S_2)]$$

其中 k_i 表示集合 S_i 中的样本个数，$i=1,2$。不等式 (2) 事实上给出了信息增益的下界，若一次划分的信息增益不能超过这个下界，则不再继续往下进行划分。格式化的算法描述如算法 1所示。

下面我们给出一个简单的例子 [9]，手工执行一下整个算法。表 1给出了我们的样本集，其中包括 5 个样本，每个样本包含 2 个分量，一个分量是连续型特征，另一个是二元分类结果，仅包括 Yes 和 No 两类。

表 1　用于解释递归的最小熵划分算法的简单数据集

Continuous Feature	Output Class
1.0	Yes
1.0	Yes
2.0	No
3.0	Yes
3.0	No

初始时，信息熵 $Ent(S)=-\frac{3}{5}\times\log_2\frac{3}{5}-\frac{2}{5}\times\log_2\frac{2}{5}=0.97$。下面开始选取二元分隔点。若选择 1.0 作为分界点，则对于严格小于 1.0 的这一部分，事实上为空，而大于等于 1.0 的另一部分，实际上就是原数据集，因此并没有产生增益，显然不满足式 (2) 的条件。类似地，可以分析选择 3.0 作为分界点的情形。下面着重讨论以 2.0 作为分界点的情形。分隔完成后，数据集被划分成两部分，一部分为小于 2.0 的那一部分，如表 4所示，另一部分为大于等于 2.0 的那一部分，如表 3所示。

表 2　对原始数据集以 2.0 作为分界点小于 2.0 的分块数据

Continuous Feature	Output Class
1.0	Yes
1.0	Yes

表 3　对原始数据集以 2.0 作为分界点大于等于 2.0 的分块数据

Continuous Feature	Output Class
2.0	No
3.0	Yes
3.0	No

由表 4数据，不难算出这一部分 S_1 的信息熵 $Ent(S_1)=-1\times\log_2 1=0$。而另一部分 S_2 的信息熵 $Ent(S_2)=-\frac{1}{3}\log_2\frac{1}{3}-\frac{2}{3}\log_2\frac{2}{3}=0.92$。所以此处划分后的类信息熵 $E=\frac{2}{5}Ent(S_1)+\frac{3}{5}Ent(S_2)=0.55$。因此此次划分产生的信息增益 $Gain=Ent(S)-E=0.42$。根据余量公式，可以求出 $\Delta=\log_2(3^5-2)-[5\times Ent(S)-2\times Ent(S_1)-3\times Ent(S_2)=5.82$。由此，可以计算出信息增益下界 $\inf Gain=\frac{\log_2(5-1)}{5}+\frac{\Delta}{5}=1.56$。因为 $Gain=0.42<1.56$，所以不进行划分。于是 Split 返回空集，Partition 也不再继续递归，那么 T 为空，即整个数据集的连续属性值全部标定成一类。

从这个简单的例子中我们发现了这个方法的一个问题，如果这一列的数据信息量不足以将其划分若干列，或者说划分的粒度过粗，则会导致之后的规则生成和分类器训练产生偏差。因此，我们在实际应用中，

算法 1 递归的最小熵划分算法

```
T = ∅

function Split(S)
    sort(S)
    W = ∅
    for each candidate c in S do
        calculate g(c) = Gain(A, c; S)
        if (2) is not satisfied then
            W = W ∪ c
        end if
    end for
    if W = ∅ then
        return ∅
    else
        return arg max_{c∈W} g(c)
    end if
end function

function Partition(S)
    t = Split(S)
    if t = ∅ then
        return
    else
        T = T ∪ t
        S_1 = {x : x ∈ S ∧ x < t}
        S_2 = {x : x ∈ S ∧ x ≥ t}
        Partition(S_1)
        Partition(S_2)
    end if
end function

function main
    Partition(S)
    sort(T)
    return T
end function
```

若最小熵划分算法返回的划分边界集合为空，则从简计议，简单的将这一属性的取值区间等分成 3 份，对于样本 $x \in [\min(S), \min(S) + \frac{\max(S)-\min(S)}{3}]$，将其标为第一块；样本 $x \in (\min(S) + \frac{\max(S)-\min(S)}{3}, \min(S) + 2 \times \frac{\max(S)-\min(S)}{3}]$，将其标为第二块；最后把 $x \in (\min(S) + 2 \times \frac{\max(S)-\min(S)}{3}, \max(S)]$ 标为第三块。这样就避免了划分粒度过粗带来的影响。

2.2 规则生成的 CBA-CG 算法实现

2.2.1 规则项集和类关联规则

CBA-RG 算法的核心就是去寻找所有满足最小支持度约束的规则项集 (ruleitem)。因而在具体讨论算法之前，有必要对规则项集的相关定义进行一些讨论。

一个规则项集就是一个形如

$$< condset, y >$$

的偶对，其中，$condset$ 是项集的集合，$y \in Y$ 是所有类标号中的一个类标号，它表示类关联规则

$$condset \to t$$

举例来说，表 4给出了一个简单的数据集 [10]，其中包括 A、B 两个属性，以及 C 这一两类划分结果，每个样本都被划分入 y 或 n。比如，$< \{(A,e),(B,p)\},(C,y) >$ 就是一个规则项集，它表示了类关联规则 $\{(A,e),(B,p)\} \to y$。从这里不难看出，事实上，类关联规则就是一个简单的 IF$\cdots$THEN$\cdots$ 规则，前件 $condset$ 就是一个合取范式，它反映了属性集的一个子集的取值情况，而后件 y 则表示如果前件成立时该样本应归为哪一类。就前述的例子而言，就可以写为 $(A = e) \wedge (B = p) \to y$，即当 $A = e$ 并且 $B = p$ 时，样本应该归为 y 类。

表 4 用于说明 CBA 算法的示例数据集

A	B	C
e	p	y
e	p	y
e	q	y
g	q	y
g	q	y
g	q	n
g	w	n
g	w	n
e	p	n
f	q	n

定义 $condset$ 的支持度计数 $condsupCount$ 为数据集 D 上包含有 $condset$ 的样本个数，其实就是统计数据集中对应于 $condset$ 的那些属性值的取值与 $condset$ 的取值一致的那些样本的个数。就前述的规则项集而言，有 3 个样本的 A 和 B 属性取值满足 $condset$ 的取值（即 $A = e, B = p$），因此 $condsupCount = 3$。

更进一步，定义规则项集的支持度计数 $rulesupCount$ 为数据集 D 上包含有 $condset$ 并且最后的类标号也一致的样本个数，事实上就是统计数据集中不仅满足 $condset$ 的取值且最后分类一致的样本个数。还是看前面的例子，前面已经知道 $condsupCount = 3$，而其中最后归为 y 类的样本个数只有 2 个，因而 $rulesupCount = 2$。

根据前面的两个支持度定义，我们便可以定义类关联规则 $condset \to t$ 的支持度和置信度了，其中支持度 $support = \frac{rulesupCount}{|D|} \times 100\%$，置信度 $confidence = \frac{rulesupCount}{condsupCount} \times 100\%$。那么对于那个例子，我们不难计算出 $support = \frac{2}{10} \times 100\% = 20\%$，$confidence = \frac{2}{3} \times 100\% = 66.7\%$。

如果一个规则项集满足最小支持度约束 $minsup$，则其是一个频繁规则项集，与之相对应的则是非频繁的规则项集。依然看前面的例子，假如我们规定 $minsup = 10\%$，那么 $support = 20\% > minsup = 10\%$，因而这条规则是频繁的。如果一个规则的置信度满足最小置信度约束 $minconf$，那么它就是一个准确的规则。类关联规则集合中的规则应当都是频繁且准确的规则。

2.2.2 规则生成的 CBA-RG 算法

文献 [1] 所提出的 CBA-RG 算法事实上是在 Apriori 算法的基础进行修改得到的，因而其基本思想与 Apriori 算法十分接近。为了便于我们后面的讨论，这里将 [1] 中的算法抄录于此，见算法 2。在这个算法中，我们将一个规则项集表示为形如

$$< (condset, condsupCount), (y, rulesupCount) >$$

且称 $condset$ 内拥有 k 项元素的规则项集为 k-ruleitems。

算法 2 CBA-RG 算法

```
F_1 = {large 1-ruleitems}
CAR_1 = genRules(F_1)
prCAR_1 = pruneRules(CAR_1)
k = 2
while F_{k-1} ≠ ∅ do
    C_k = candidateGen(F_{k-1})
    for each data case d ∈ D do
        C_d = ruleSubset(C_k, d)
        for each candidate c ∈ C_d do
            c.condsupCount + +
            if d.class = c.class then
                c.rulesupCount + +
            end if
        end for
    end for
    F_k = {c ∈ C_k | c.rulesupCount ≤ minsup}
    CAR_k = genRules(F_k)
    prCAR_k = pruneRules(CAR_k)
    k + +
end while
CARs = ⋃_k CAR_k
prCARs = ⋃_k prCAR_k
```

刘兵等人已在 [1] 中对上述算法进行了详细的讨论，我们不再这里做重复的分析，我们下面只分析作者没有在论文中指明的步骤。

一个问题就是算法 2的第 2 行 genRules 的实现。我们在 2.2.1 节中给出了如何从规则项集生成类关联规则的过程。但是这个当面对一个规则项集时，简单的采用这个方法会存在一些问题。假如存在两个规则项集，它们拥有相同的 $condset$，那么我们应当选取那个具有最高置信度的规则项集。如果这两个规则项集的置信度也一致，那么就随机取一条作为该 $condset$ 的代表规则项集。依然拿前面表 4为例，并假设 $minconf = 60\%$，我们可以找到 2 个 $condset$ 一致的 2-ruleitems：

$$< (\{(A,g),(B,q)\},3),((C,y),2) >$$

$$< (\{(A,g),(B,q)\},3),((C,n),1) >$$

这两个规则项集的 $condset$ 一致，只是最后的所属类不同。对于第一个规则项集，其置信度为 $\frac{2}{3}$，第二个规则项集的置信度为 $\frac{1}{3}$。显然前者的置信度更高，所以我们选取第一个规则项集以生成类关联规则

$$\{(A,g),(B,q)\} \to (C,y)$$

因此，在实现 genRules 时，需要按照上述策略处理 $condset$ 一致的情形。详细的算法描述如算法 3所示。

算法 3 genRules 的实现

```
function genRules(F)
    CAR = ∅
    for each ruleitem r ∈ F do
        if r.condset in CAR then
            if r.confidence > max({c.confidence : c ∈ CAR ∧ c.condset = r.condset}) then
                replace c with r
            end if
        else
            CAR = CAR ∪ r
        end if
    end for
    return CAR
end function
```

第二个需要解决的问题就是规则剪枝的问题，即 pruneRules 的实现。文献 [1] 中作者介绍到其采用的是 C4.5 算法中的利用悲观错误率进行剪枝的方法。给定一个类关联规则 $r : A \to y$，考虑剪枝后的规则 $r' : A' \to y$，其中 A' 是 A 中去掉一个合取项后得到的。只要剪枝后的规则的悲观误差率不高于原规则的误差率，就保留其中悲观误差率最低的规则。重复规则剪枝步骤，直到规则的悲观误差率不能再进行改进为止。由于某些规则在剪枝后会变得相同，因而需要丢弃重复的规则。在上面的叙述中，我们需要强调剪枝的条件是剪枝后的规则的悲观误差率不高于原规则的误差率，这里的“不高于”而非严格的“低于”是基于众所周知的奥卡姆剃刀，即

奥卡姆剃刀：给定两个具有相同泛化误差的模型，较简单的模型比较复杂的模型更可取。

这个算法的形式化的描述见算法 4。

第三个需要解决的问题就是如何实现 candidateGen。candidateGen 函数和 Apriori 算法中的 Apriori-gen 函数十分类似。该操作由前一次迭代发现的 F_{k-1} 产生新的候选 k-ruleitems。我们这里采用 $F_{k-1} \times F_{k-1}$ 方法。由于 candidateGen 函数合并一对频繁 $(k-1)$-ruleitems，当且仅当它们的前 $k-2$ 个项都相同。但是这么做还不能满足先验原理，因为合并后的项的子集不一定是频繁的，因而还需要进行剪枝，即考虑候选

算法 4 pruneRules 的实现

```
function pruneRules(r)
    if |r.condset| = 0 then return
    else
        prs = ∅
        r' = r
        for each item t in r.condset do
            r'.condset = r.condset − t
            prs = prs ∪ r'
        end for
        pr = arg min {r'.errorRate : r' ∈ prs}
        if pr.errorRate ≤ r.errorRate then
            return pruneRules(pr)
        end if
    end if
end function
```

k-ruleitems $X = \{i_1, i_2, \cdots, i_k\}$，如果存在它的一个真子集 $X - \{i_j\}(\forall j = 1, 2, \cdots, k)$ 是非频繁的，那么 X 将被剪枝。用形式化的语言描述 [11]，有

$$\text{candidateGen}(F_{k-1}) = \{a \cup \{b\} | a \in F_{k-1} \wedge b \notin a\} - \{c | \{s | s \subseteq c \wedge |s| = k-1\} \nsubseteq F_{k-1}\}$$

至此，我们就不难实现 CBA-RG 算法了。

2.3 构建分类器的 CBA-CB M1 算法实现

2.3.1 相关概念

我们称规则 $r : condset \to y$ 覆盖样本 d，当且仅当 r 的前件 $condset$ 满足样本 d，即对应的属性值完全相同。如果这条规则的后件 y 与样本 d 的分类一致，则称这条规则正确分类了样本 d。

下面定义类关联规则的偏序。给定两个规则 r_i, r_j，定义 $r_i > r_j$ 当

a) r_i 的置信度高于 r_j 的置信度；

b) r_i 和 r_j 的置信度一致，但是 r_i 的支持度高于 r_j 的支持度；

c) r_i 和 r_j 的置信度和支持度都一致，但是 r_i 比 r_j 的生成时间要早。

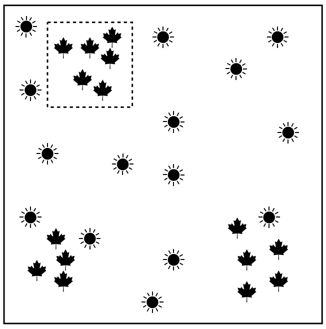
CBA-CB 算法的基本思想就是构建一个有序规则序列

$$< r_1, r_2, \cdots, r_n, default\_class >$$

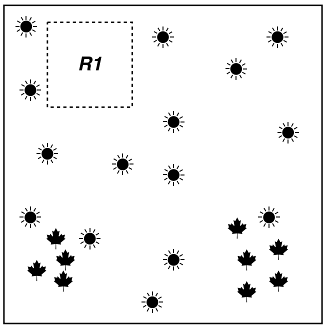
作为最后的分类器。其中 $r_i \in R$，R 为生成的类关联规则集合。序列中的两条规则 r_a, r_b，当 $b > a$ 时有 $r_a > r_b$。$default\_class$ 是默认类，当前面的那些规则都不能覆盖这条记录时，则将其划分入默认类中。

2.3.2 CBA-CB M1 算法的实现

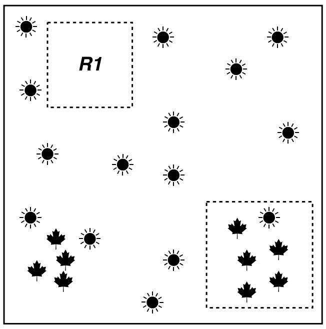
这个略显幼稚的算法的主要思想是基于贪心策略，尽可能早的选出高秩的那些规则。为了更好的说明这个算法的思想，我们用图示进行解释。图 1(a)展示了初始时数据集的情况，其中包括若干正例和反例，虚



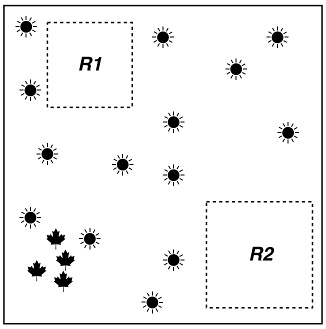
(a) 初始数据集



(b) 使用 R_1 覆盖后的数据集



(c) 选取下一块待覆盖的数据



(d) 使用 R_2 覆盖后的数据集

图 1　CBA-CB M1 算法示例

线框出区域内是秩最高的规则 R_1 所能覆盖的样本。去除了 R_1 所能覆盖的所有训练记录后，剩下的数据样本如图 1(b)所示。接着在剩余的样本中寻找秩次高的规则 R_2 所能覆盖的样本，如图 1(c)中的虚线框所示。然后删去 R_2 所能覆盖的这些样本，留下的记录如图 1(d)所示。算法描述见算法 5所示。

2.4　构建分类器的 CBA-CB M2 算法实现

尽管 CBA-CB M1 算法形式相对简单，易于编程实现，但是当面对大训练样本的情形时，其效率不高。所以文献 [1] 中又紧接着提出了 CBA-CB M2 算法，其中心思想同 M1 基本一致。该算法包括 3 个阶段，通过首先缩小候选规则的方法，减少不必要的规则判定，提高训练效率。算法的详细解释可见文献 [1]，这里只提供伪代码描述，见算法 6、7和 8。

但是需要注意的是，我们深切怀疑算法 8第 12 行与第 13 行之间漏写了一步操作，这里应当将规则 r 所覆盖的那些训练记录从样本集中删除，否则后续第 14 至 15 行更新当前数据集类分布的工作就无需进行了，因为样本集根本没有发生变化。不论是 M1 还是 M2 算法，其大体的框架应该是类似的，否则两个版本的算法得到的分类器完全不同，这显然不符合常理。

在实际编写程序时，由于某些数据集中所包含的规则太多，内存无法存放下，而且十分耗费时间。因而我们考虑当得到的类关联规则数达到 2000，或者收敛速度已呈现亚线性（即每一轮迭代所挖掘出的规则数少于 10 个）时，则不再继续进行挖掘，提前终止迭代。实验结果表明，采取这样的策略对于最后分类器的性能并没有太大影响。我们认为，这是由于高秩规则已经可以很好的描述数据集中存在的联系，过多的规则对于提高分类器性能的影响有限。

算法 5 CBA-CB M1 算法

$R = \mathrm{sort}(R)$
for each rule $r \in R$ in sequence **do**
  $temp = \varnothing$
  for each case $d \in D$ **do**
    if d satisfies the conditions of r **then**
      store d.id in $temp$ and mark r if it correctly classifies d
    end if
  end for
  if r is marked **then**
    insert r at the end of C
    delete all the cases with the ids in $temp$ from D
    selecting a default class for the current C
    compute the total number of errors of C
  end if
end for
Find the first rule p in C with the lowest total number of errors and drop all the rules after p in C
Add the default class associated with p to end of C, and return C (our classifier)

算法 6 CBA-CB M2 算法：Stage 1

$Q = \varnothing; U = \varnothing; A = \varnothing$
for each case $d \in D$ **do**
  $cRule = \mathrm{maxCoverRule}(C_c, d)$
  $wRule = \mathrm{maxCoverRule}(C_w, d)$
  $U = U \cup \{cRule\}$
  $cRule.\mathrm{classCasesCovered}[d.\mathrm{class}] ++$
  if $cRule > wRule$ **then**
    $Q = Q \cup \{cRule\}$
    mark $cRule$
  else $A = A \cup < d.\mathrm{id}, d.\mathrm{class}, cRule, wRule >$
  end if
end for

算法 7 CBA-CB M2 算法：Stage 2

for each entry $< dID, y, cRule, wRule > \in A$ **do**
  if $wRuile$ is marked **then**
    $cRule$.classCasesCovered$[y] - -$
    $wRule$.classCasesCovered$[y] + +$
  else
    $wSet =$ allCoverRules$(U, dID$.case$, cRule)$
    for each rule $w \in wSet$ **do**
      w.replace $= w$.replace $\cup \{< cRule, dID, y >\}$
      w.classCasesCovered$[y] + +$
    end for
    $Q = Q \cup wSet$
  end if
end for

算法 8 CBA-CB M2 算法：Stage 3

$classDistr =$ compClassDistri(D)
$ruleErrors = 0$
$Q =$ sort(Q)
for each rule r in Q in sequence **do**
  if r.classCasesCovered$[r$.class$] \neq 0$ **then**
    for each entry $< rul, dID, y >$ in r.replace **do**
      if the dID case has been covered by a previous r **then**
        r.classCasesCovered$[y] - -$
      else
        rul.classCasesCovered$[y] - -$
      end if
    end for
    $ruleErrors = ruleErrors +$errorsOfRule$(r)$
    $classDistr =$update$(r, classDistr)$
    $defaultClass =$ selectDefault$(classDistr)$
    $totalErrors = ruleErrors + defaultErrors$
    Insert $< r, default\text{-}class, totalErrors >$ at the end of C
  end if
end for
Find the first rule p in C with the lowest $totalErrors$, and then discard all the rules after p from C
Add the default class associated with p to end of C
Return C without $totalErrors$ and $default\text{-}class$

3 实验结果与分析

编写程序重现文献 [1] 的工作是本文的中心工作，因而如何检验我们的实现的程序至关重要。我们仿照文章中的检验手段，采用了 10 折交叉检验去计算分类器的错误率。因而在正式给出我们的测试结果之前，首先介绍一下 10 折交叉检验。

3.1 10 折交叉检验

所谓 10 折交叉检验 [8]，就是将数据随机分成大小相同的 10 份，在每次运行时，选择其中一份作为检验集，而其余的全部作为训练集，该过程重复进行 10 次，使得每份数据都用于检验恰好一次。最后取 10 次运行的误差的平均值（一说是总和 [12]）作为最后的误差估计。限于篇幅，图 2给出了 3 折交叉检验的简单示意，10 折交叉检验的思想和其基本一致。

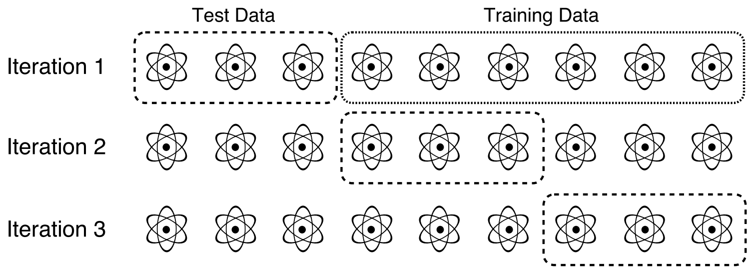


图 2　3 折交叉检验的简单示意图

3.2 样例测试

本文的实现程序全部使用 Python 3.6.0 编写，运行在 2.9 GHz Intel Core i5 处理器上。我们从 UCI 机器学习资源库 [2] 中选取了 30 个数据集，对我们的程序进行测试。我们的运行时间全部基于驻留在内存的数据集。所有指标都采用 10 折交叉检验，取 10 次检验的平均性能以评估其总体性能指标。具体的测试结果如表 5和表 6所示。表中每一行对应一个数据集，限于篇幅，数据集名称可能存在简写。表格分成分成两大部分，左侧 3 至 8 列对应于不进行剪枝时的各种测试结果，右侧 9 至 13 列对应剪枝下的测试结果。以左侧为例，第 3 列表示 10 折交叉检验下的平均错误率；第 4 列为 CBA-RG 算法生成的类关联规则的平均数量，若考虑剪枝，则这一列表示的是剪枝后的规则数；第 5 列记录 CBA-RG 算法的平均执行时间，若考虑剪枝，则这一部分时间还涵盖了剪枝所花费的时间；第 6 列表示 CBA-CB 算法的平均执行时间；第 7 列表示最后训练得到的分类器中所包含的类关联规则的平均数量。

从测试结果不难看出，我们训练出的分类器，其判定错误率较低，基本可以满足分类任务的要求。但是必须看到，我们的程序，尤其是 CBA-RG 算法的运行时间过长。我们分析，认为有以下几点原因：a) 我们所采用的 Python 语言是一种动态语言，其执行效率相比于论文中采用的 C++ 语言肯定要低许多；b) 为了调试程序的方便，我们在程序中额外增加了许多事实上与类关联规则和分类器无关的其他信息，这势必会增加程序运行的开销，降低程序的运行效率；c) 我们所采用的数据结构较为简陋，理论上应当采用树型结构实现规则和分类器。数据结构对算法执行效率的影响是主要的。

综上，我们可以总结，我们实现的基于类关联规则的分类器，其判定准确率较高，但是执行时间较慢，有待以后的优化提高。

表 5　使用 CBA-CB M1 方法下的测试结果

序号	数据集	不剪枝					剪枝				
		错误率/%	#CARs	RG 时间/s	CB 时间/s	# 分类器 CARs	错误率/%	#CARS	RG 时间/s	CB 时间/s	# 分类器 CARs
1	australian	1.9	1140	15.48	0.19	58	2.9	752	75.46	0.22	70
2	banknote	0.1	194	0.90	0.09	28	0.0	68	2.17	0.06	28
3	car	1.3	254	2.30	0.08	24	1.1	199	3.45	0.08	26
4	diagnosis	0.0	692	2.97	0.00	4	0.0	126	5.48	0.00	4
5	dresses	3.0	511	2.64	0.28	89	2.0	414	2.45	0.24	89
6	facebook	0.8	1995	8.19	0.15	59	1.2	1254	70.19	0.17	61
7	flare	1.0	567	6.17	0.04	44	1.1	423	20.05	0.04	46
8	forest	3.4	2116	2.88	0.18	40	3.5	1658	26.34	0.16	40
9	german	0.0	1294	10.43	0.48	85	0.1	906	15.74	0.30	80
10	heart-disease	1.6	659	16.75	0.06	35	2.4	475	39.65	0.05	34
11	horse-colic	0.0	2068	13.65	0.14	57	0.7	1594	44.12	0.17	38
12	ilpd	0.2	1002	18.48	0.22	64	0.4	666	21.42	0.21	67
13	ionosphere	1.1	2008	12.56	0.13	41	3.5	1378	34.76	0.11	46
14	iris	0.7	120	0.09	0.00	9	0.7	45	0.20	0.00	9
15	kr-vs-kp	0.0	2241	11.32	0.95	39	0.1	1149	69.80	0.78	29
16	led	5.0	2	0.01	0.00	2	17.5	1	0.01	0.00	1
17	mammo	0.0	306	3.04	0.24	64	0.5	220	4.02	0.19	65
18	messidor	9.6	2359	9.04	1.27	86	9.9	1879	30.59	1.12	73
19	monks	0.0	656	2.27	0.04	40	0.0	455	4.53	0.01	22
20	pima	2.3	751	14.58	0.24	68	2.4	630	33.56	0.19	52
21	quali-bank	0.0	731	2.89	0.01	9	0.0	141	10.27	0.00	10
22	seeds	1.9	749	3.87	0.01	23	1.9	168	17.79	0.01	23
23	tic-tac-toe	0.1	600	2.15	0.21	61	0.1	540	3.33	0.13	55
24	transfusion	4.3	9	0.00	0.00	4	2.1	9	0.01	0.00	4
25	user-know	4.3	7	0.00	0.00	5	4.3	7	0.01	0.00	5
26	vertebra	0.0	521	8.67	0.04	15	0.0	133	12.81	0.01	15
27	wiki4HE	0.1	1150	5.36	0.42	86	0.3	720	45.12	0.33	79
28	wine	0.6	1702	26.28	0.05	28	1.1	1004	83.82	0.04	27
29	yeast	3.1	3	0.11	0.01	3	3.1	3	0.12	0.02	3
30	zoo	1.0	2397	5.95	0.03	10	1.0	1780	38.76	0.02	11

表 6　使用 CBA-CB M2 方法下的测试结果

序号	数据集	不剪枝					剪枝				
		错误率/%	#CARs	RG 时间/s	CB 时间/s	# 分类器 CARs	错误率/%	#CARS	RG 时间/s	CB 时间/s	# 分类器 CARs
1	australian	0.0	1225	11.69	0.63	15	0.0	905	99.24	0.21	76
2	banknote	0.0	194	0.86	0.24	13	0.1	67	2.18	0.06	27
3	car	0.6	276	2.64	0.30	8	1.6	192	4.14	0.07	23
4	diagnosis	0.0	695	2.89	0.08	4	0.0	125	5.73	0.00	4
5	dresses	0.0	536	2.29	0.20	29	2.2	410	2.31	0.20	85
6	facebook	1.0	1825	26.40	0.63	10	1.8	959	90.78	0.17	58
7	flare	0.0	475	57.25	0.17	19	0.3	379	99.89	0.17	20
8	forest	1.5	2285	5.22	0.82	12	2.1	1645	30.43	0.76	13
9	german	0.2	1281	12.44	0.86	13	0.2	884	31.78	0.73	9
10	heart-disease	1.0	492	9.52	0.14	6	1.9	374	37.69	0.12	10
11	horse-colic	0.4	1978	31.44	0.61	32	0.9	1238	83.42	0.52	29
12	ilpd	0.3	1015	20.00	0.43	1	0.4	780	53.17	0.37	4
13	ionosphere	0.0	1975	6.86	0.68	18	0.3	1499	30.12	0.56	20
14	iris	1.3	120	0.07	0.02	7	0.0	44	0.20	0.00	8
15	kr-vs-kp	0.0	2202	9.60	5.53	12	0.1	1308	70.34	4.67	21
16	led	10.0	2	0.01	0.00	1	15.0	1	0.01	0.00	1
17	mammo	0.0	299	2.71	0.25	8	0.3	220	3.93	0.20	64
18	messidor	0.1	1389	13.29	1.52	3	0.3	1123	40.12	1.31	5
19	monks	0.0	683	3.05	0.20	34	0.0	415	4.30	0.01	21
20	pima	0.0	740	11.98	0.43	6	0.3	681	42.70	0.39	8
21	quali-bank	0.0	748	3.24	0.16	7	0.0	147	10.44	0.00	10
22	seeds	1.0	790	3.93	0.15	20	2.4	170	18.99	0.00	23
23	tic-tac-toe	0.0	604	2.19	0.38	19	0.0	542	3.52	0.14	56
24	transfusion	0.0	9	0.00	0.01	2	0.0	9	0.01	0.00	4
25	user-know	5.2	7	0.00	0.00	3	4.4	7	0.01	0.00	5
26	vertebra	0.0	523	8.44	0.16	3	0.0	132	13.24	0.01	15
27	wiki4HE	0.9	1064	4.51	0.64	1	1.1	789	30.65	0.49	5
28	wine	0.1	1670	52.63	0.30	29	0.2	1232	97.79	0.25	27
29	yeast	3.1	3	0.13	0.02	2	3.1	3	0.12	0.01	3
30	zoo	1.0	2325	8.78	0.28	8	1.1	1892	53.20	0.24	10

4 致谢与后记

数据挖掘与知识发现的课程项目至此快要进入尾声了。从三月份完成选题、编制项目计划书，到现在完成了这份报告的编写，一路的酸甜苦辣已回荡在脑海之中。

陈大松同学完成了数据预处理的工作，编写了缺失值处理和连续型属性值离散化的相关程序，为我们的工作开了个好头。刘砺志同学主要负责 CBA 算法的实现工作，并且完成了这份报告的编写任务，并将其交由另外两名同学审阅并进行修改。肖璐菁同学负责数据集的格式归一化处理，并编写了测试程序，同时记录了各项数据指标。这项工作十分枯燥无味，但是肖同学还是坚持并优秀的完成了它们。感谢大家团结一致、戮力同心，坚持不懈的完成了论文的重现工作。

最后还要感谢林琛老师的博学多才，为我们开启了数据挖掘这个新世界的大门，带领我们领略数据科学的独特魅力。通过这项工作，让我们体会到数据科学的乐趣，为今后的从事相关工作奠定了坚实的基础。

参考文献

[1] Liu B, Hsu W, Ma Y. Integrating Classification and Association Rule Mining. Proc of Kdd, 1998:80–86.

[2] UC Irvine Machine Learning Repository, `http://archive.ics.uci.edu/ml/`

[3] Rakesh Agrawal, Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, pages 487-499, Santiago, Chile, September 1994.

[4] Ho, Tin Kam. Random Decision Forests. Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14−16 August 1995. pp. 278−282.

[5] Catlett J. On changing continuous attributes into ordered discrete attributes. European Working Session on Learning on Machine Learning. Springer-Verlag New York, Inc. 1991:164-178.

[6] Fayyad U M. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. International Joint Conference on Artificial Intelligence. 1993:1022-1027.

[7] Dougherty J, Kohavi R, Sahami M. Supervised and Unsupervised Discretization of Continuous Features. Machine Learning Proceedings, 1995(2):194-202.

[8] Cross-validation (statistics), `https://en.wikipedia.org/wiki/Cross-validation_(statistics)`

[9] Discretizing a continuous variable using Entropy, `http://www.clear-lines.com/blog/post/Discretizing-a-continuous-variable-using-Entropy.aspx`

[10] Integrating Classification and Association Rule Mining - the Secret Behind CBA, `https://www.cs.uic.edu/~hxiao/courses/cs594-slides.pdf`

[11] Apriori algorithm, `https://en.wikipedia.org/wiki/Apriori_algorithm`

[12] Tan P N, Steinbach M, Kumar V. Introduction to Data Mining, (First Edition). Addison-Wesley Longman Publishing Co. Inc. 2005.

附录 A　相关源程序

本文使用的程序均采用 Python 3.6.0 编写，运行在安装 Mac OS 10.12.2 操作系统、搭载有 2.9 GHz Intel Core i5 处理器、使用 8 GB 2133 MHz LPDDR3 内存的计算机上。

一、程序清单rmep.py

```python
"""
Description: Recursive minimal entropy partitioning, to discretize continuous-valued attributes. We use the supervised
    algorithm presented in Fayyad & Irani (1993) and introduced in Dougherty, Kohavi & Sahami (1995) section 3.3.
    We also refer to a F# code on GitHub (https://gist.github.com/mathias-brandewinder/5650553).
Input: a data table with several rows but only two column, the first column is continuous-valued (numerical) attributes,
    and the second column is the class label of each data case (categorical).
    e.g. data = [[1.0, 'Yes'], [0.5, 'No'], [2.0, 'Yes']]
Output: a list of partition boundaries of the range of continuous-valued attribute in ascending sort order.
    e.g. walls = [0.5, 0.8, 1.0], thus we can separate the range into 4 intervals: <=0.5, 0.5<*<=0.8, 0.8<*<=1.0 & >=1.0
Author: CBA Studio
Reference:
    1. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning, Fayyad & Irani, 1993
    2. Supervised and Unsupervised Discretization of Continuous Features, Dougherty, Kohavi & Sahami, 1995
    3. http://www.clear-lines.com/blog/post/Discretizing-a-continuous-variable-using-Entropy.aspx
"""
import math


# A block to be split
# It has 4 member:
#   data: the data table with a column of continuous-valued attribute and a column of class label
#   size: number of data case in this table
#   number_of_classes: obviously, the number of class in this table
#   entropy: entropy of dataset
class Block:
    def __init__(self, data):
        self.data = data
        self.size = len(data)
        classes = set([x[1] for x in data])     # get distinct class labels in this table
        self.number_of_classes = len(set(classes))
```

```
        self.entropy = calculate_entropy(data)


# Calculate the entropy of dataset
# parameter data: the data table to be used
def calculate_entropy(data):
    number_of_data = len(data)
    classes = set([x[1] for x in data])
    class_count = dict([(label, 0) for label in classes])
    for data_case in data:
        class_count[data_case[1]] += 1      # count the number of data case of each class
    entropy = 0
    for c in classes:
        p = class_count[c] / number_of_data
        entropy -= p * math.log2(p)         # calculate information entropy by its formula, where the base is 2
    return entropy


# Compute Gain(A, T: S) mentioned in Dougherty, Kohavi & Sahami (1995), i.e. entropy gained by splitting original_block
#   into left_block and right_block
# original_block: the block before partition
# left_block: the block split which its value below boundary
# right_block: the block above boundary
def entropy_gain(original_block, left_block, right_block):
    gain = original_block.entropy - \
            ((left_block.size / original_block.size) * left_block.entropy +
            (right_block.size / original_block.size) * right_block.entropy)
    return gain


# Get minimum entropy gain required for a split of original_block into 2 blocks "left" and "right", see Dougherty,
#   Kohavi & Sahami (1995)
# original_block: the block before partition
# left_block: the block split which its value below boundary
# right_block: the block above boundary
def min_gain(original_block, left_block, right_block):
    delta = math.log2(math.pow(3, original_block.number_of_classes) - 2) - \
            (original_block.number_of_classes * original_block.entropy -
             left_block.number_of_classes * left_block.entropy -
             right_block.number_of_classes * right_block.entropy)
    gain_sup = math.log2(original_block.size - 1) / original_block.size +
```

```
            delta / original_block.size
    return gain_sup


# Identify the best acceptable value to split block
# block: a block of dataset
# Return value: a list of (boundary, entropy gain, left block, right block) or
#   None when it's unnecessary to split
def split(block):
    candidates = [x[0] for x in block.data]     # candidates is a list of values can be picked up as boundary
    candidates = list(set(candidates))          # get different values in table
    candidates.sort()                           # sort ascending
    candidates = candidates[1:]                 # discard smallest, because by definition no value is smaller

    wall = []       # wall is a list storing final boundary
    for value in candidates:
        # split by value into 2 groups, below & above
        left_data = []
        right_data = []
        for data_case in block.data:
            if data_case[0] < value:
                left_data.append(data_case)
            else:
                right_data.append(data_case)

        left_block = Block(left_data)
        right_block = Block(right_data)

        gain = entropy_gain(block, left_block, right_block)
        threshold = min_gain(block, left_block, right_block)

        # minimum threshold is met, the value is an acceptable candidate
        if gain >= threshold:
            wall.append([value, gain, left_block, right_block])

    if wall:    # has candidate
        wall.sort(key=lambda wall: wall[1], reverse=True)   # sort descending by "gain"
        return wall[0]      # return best candidate with max entropy gain
    else:
        return None         # no need to split


# Top-down recursive partition of a data block, append boundary into "
```

```
walls"
# block: a data block
def partition(block):
    walls = []

    # inner recursive function, accumulate the partitioning values
    # sub_block: just a data block
    def recursive_split(sub_block):
        wall_returned = split(sub_block)         # binary partition, get bin boundary
        if wall_returned:                        # still can be spilt
            walls.append(wall_returned[0])       # record this partitioning value
            recursive_split(wall_returned[2])    # recursively process left block
            recursive_split(wall_returned[3])    # recursively split right block
        else:
            return                               # end of recursion

    recursive_split(block)       # call inner function
    walls.sort()                 # sort boundaries descending
    return walls


# just for test
if __name__ == '__main__':
    import random

    test_data = []
    for i in range(100):
        test_data.append([random.random(), random.choice(range(0, 2))])
        test_data.append([random.random() + 1, random.choice(range(2, 4))])
        test_data.append([random.random() + 2, random.choice(range(4, 6))])
        test_data.append([random.random() + 3, random.choice(range(6, 8))])

    test_block = Block(test_data)
    test_walls = partition(test_block)
    print(test_walls)         # should be [1+e, 2+e, 3+e], where e is a number very close to 0
```

二、程序清单pre_processing.py

```
"""
Description: Pre-process original data. Firstly, we process the missing values (donated as '?'), discarding this column
```

```python
    when missing ratio above 50%, or filling blanks when below. We "guess" missing values by simply filling the mode of
    existing values in the same column. And then, for the numerical attribute, we discretizate it by recursive minimal
    entropy partitioning (see rmep.py). For the categorical attribute, we just replace the label with a
    positive integer. For more information, see [1].
Input: a data table with several data case, many attributes and class label in the last column, a list of the name of
    each attribute, and a list of the type of each column.
Output: a data list without numerical values and "str" categorical values.
Author: CBA Studio
Reference:
    1. http://cgi.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DN/lucs-kdd_DN.html
"""
import rmep


# Identify the mode of a list, both effective for numerical and categorical list. When there exists too many modes
#   having the same frequency, return the first one.
# arr: a list need to find mode
def get_mode(arr):
    mode = []
    arr_appear = dict((a, arr.count(a)) for a in arr)   # count appearance times of each key
    if max(arr_appear.values()) == 1:       # if max time is 1
        return      # no mode here
    else:
        for k, v in arr_appear.items():     # else, mode is the number which has max time
            if v == max(arr_appear.values()):
                mode.append(k)
    return mode[0]  # return first number if has many modes


# Fill missing values in column column_no, when missing values ration below 50%.
# data: original data list
# column_no: identify the column No. of that to be filled
def fill_missing_values(data, column_no):
    size = len(data)
    column_data = [x[column_no] for x in data]      # get that column
    while '?' in column_data:
        column_data.remove('?')
    mode = get_mode(column_data)
```

```
    for i in range(size):
        if data[i][column_no] == '?':
            data[i][column_no] = mode              # fill in mode
    return data


# Get the list needed by rmep.py, just glue the data column with class column.
# data_column: the data column
# class_column: the class label column
def get_discretization_data(data_column, class_column):
    size = len(data_column)
    result_list = []
    for i in range(size):
        result_list.append([data_column[i], class_column[i]])
    return result_list


# Replace numerical data with the No. of interval, i.e. consecutive positive integers.
# data: original data table
# column_no: the column No. of that column
# walls: the split point of the whole range
def replace_numerical(data, column_no, walls):
    size = len(data)
    num_spilt_point = len(walls)
    for i in range(size):
        if data[i][column_no] > walls[num_spilt_point - 1]:
            data[i][column_no] = num_spilt_point + 1
            continue
        for j in range(0, num_spilt_point):
            if data[i][column_no] <= walls[j]:
                data[i][column_no] = j + 1
                break
    return data


# Replace categorical values with a positive integer.
# data: original data table
# column_no: identify which column to be processed
def replace_categorical(data, column_no):
    size = len(data)
    classes = set([x[column_no] for x in data])
    classes_no = dict([(label, 0) for label in classes])
    j = 1
    for i in classes:
        classes_no[i] = j
        j += 1
```

```
    for i in range(size):
        data[i][column_no] = classes_no[data[i][column_no]]
    return data, classes_no


# Discard all the column with its column_no in discard_list
# data: original data set
# discard_list: a list of column No. of the columns to be discarded
def discard(data, discard_list):
    size = len(data)
    length = len(data[0])
    data_result = []
    for i in range(size):
        data_result.append([])
        for j in range(length):
            if j not in discard_list:
                data_result[i].append(data[i][j])
    return data_result


# Main method here, see Description in detail
# data: original data table
# attribute: a list of the name of attribute
# value_type: a list identifying the type of each column
# Returned value: a data table after process
def pre_process(data, attribute, value_type):
    column_num = len(data[0])
    size = len(data)
    class_column = [x[-1] for x in data]
    discard_list = []
    for i in range(0, column_num - 1):
        data_column = [x[i] for x in data]

        # process missing values
        missing_values_ratio = data_column.count('?') / size
        if missing_values_ratio > 0.5:
            discard_list.append(i)
            continue
        elif missing_values_ratio > 0:
            data = fill_missing_values(data, i)
            data_column = [x[i] for x in data]

        # discretization
        if value_type[i] == 'numerical':
            discretization_data = get_discretization_data(data_column,
               class_column)
            block = rmep.Block(discretization_data)
            walls = rmep.partition(block)
```

```python
            if len(walls) == 0:
                max_value = max(data_column)
                min_value = min(data_column)
                step = (max_value - min_value) / 3
                walls.append(min_value + step)
                walls.append(min_value + 2 * step)
            print(attribute[i] + ":", walls)        # print out split points
            data = replace_numerical(data, i, walls)
        elif value_type[i] == 'categorical':
            data, classes_no = replace_categorical(data, i)
            print(attribute[i] + ":", classes_no)   # print out replacement list

    # discard
    if len(discard_list) > 0:
        data = discard(data, discard_list)
        print("discard:", discard_list)              # print out discard list
    return data


# just for test
if __name__ == '__main__':
    test_data = [
        ['red', 25.6, 56, 1],
        ['green', 33.3, 1, 1],
        ['green', 2.5, 23, 0],
        ['blue', 67.2, 111, 1],
        ['red', 29.0, 34, 0],
        ['yellow', 99.5, 78, 1],
        ['yellow', 10.2, 23, 1],
        ['yellow', 9.9, 30, 0],
        ['blue', 67.0, 47, 0],
        ['red', 41.8, 99, 1]
    ]
    test_attribute = ['color', 'average', 'age', 'class']
    test_value_type = ['categorical', 'numerical', 'numerical', 'label']
    test_data_after = pre_process(test_data, test_attribute, test_value_type)
    print(test_data_after)
```

三、程序清单read.py

```python
"""
Description: Read initial dataset and decode it into a list. Here we replace all missing value and discretizate
    the numerical values.
Input: initial dataset stored in *.data file, and scheme description
```

```
    stored in *.names file.
Output: a data list after pre-processing.
Author: CBA Studio
"""
import csv


# Read dataset and convert into a list.
# path: directory of *.data file.
def read_data(path):
    data = []
    with open(path, 'r') as csv_file:
        reader = csv.reader(csv_file, delimiter=',')
        for line in reader:
            data.append(line)
        while [] in data:
            data.remove([])
    return data


# Read scheme file *.names and write down attributes and value types.
# path: directory of *.names file.
def read_scheme(path):
    with open(path, 'r') as csv_file:
        reader = csv.reader(csv_file, delimiter=',')
        attributes = next(reader)
        value_type = next(reader)
    return attributes, value_type


# convert string-type value into float-type.
# data: data list returned by read_data.
# value_type: list returned by read_scheme.
def str2numerical(data, value_type):
    size = len(data)
    columns = len(data[0])
    for i in range(size):
        for j in range(columns-1):
            if value_type[j] == 'numerical' and data[i][j] != '?':
                data[i][j] = float(data[i][j])
    return data


# Main method in this file, to get data list after processing and scheme
    list.
# data_path: tell where *.data file stores.
# scheme_path: tell where *.names file stores.
def read(data_path, scheme_path):
```

```python
    data = read_data(data_path)
    attributes, value_type = read_scheme(scheme_path)
    data = str2numerical(data, value_type)
    return data, attributes, value_type


# just for test
if __name__ == '__main__':
    import pre_processing

    test_data_path = '/Users/liulizhi/Desktop/iris.data'
    test_scheme_path = '/Users/liulizhi/Desktop/iris.names'
    test_data, test_attributes, test_value_type = read(test_data_path, test_scheme_path)
    result_data = pre_processing.pre_process(test_data, test_attributes, test_value_type)
    print(result_data)
```

四、程序清单ruleitem.py

```python
"""
Description: Definition of class RuleItem, including condset, class label (y in paper), condsupCount, rulesupCount,
    support and confidence.
Input: condset which is a set of items, class label and the dataset.
Output: a ruleitem with its condsupCount, rulesupCount, support and confidence.
Author: CBA Studio
Reference: https://www.cs.uic.edu/~hxiao/courses/cs594-slides.pdf
"""


class RuleItem:
    """
    cond_set: a dict with following fashion:
            {item name: value, item name: value, ...}
        e.g.
            {A: 1, B: 1} (A, B are name of columns, here called "item", and in our code should be numerical index
                                but not string)
    class_label: just to identify the class it belongs to.
    dataset: a list returned by read method. (see read.py)
    cond_sup_count, rule_sup_count, support and confidence are number.
    """
    def __init__(self, cond_set, class_label, dataset):
        self.cond_set = cond_set
        self.class_label = class_label
        self.cond_sup_count, self.rule_sup_count = self._get_sup_count(dataset)
```

```
        self.support = self._get_support(len(dataset))
        self.confidence = self._get_confidence()

    # calculate condsupCount and rulesupCount
    def _get_sup_count(self, dataset):
        cond_sup_count = 0
        rule_sup_count = 0
        for case in dataset:
            is_contained = True
            for index in self.cond_set:
                if self.cond_set[index] != case[index]:
                    is_contained = False
                    break
            if is_contained:
                cond_sup_count += 1
                if self.class_label == case[-1]:
                    rule_sup_count += 1
        return cond_sup_count, rule_sup_count

    # calculate support count
    def _get_support(self, dataset_size):
        return self.rule_sup_count / dataset_size

    # calculate confidence
    def _get_confidence(self):
        if self.cond_sup_count != 0:
            return self.rule_sup_count / self.cond_sup_count
        else:
            return 0

    # print out the ruleitem
    def print(self):
        cond_set_output = ''
        for item in self.cond_set:
            cond_set_output += '(' + str(item) + ', ' + str(self.cond_set[item]) + '), '
        cond_set_output = cond_set_output[:-2]
        print('<({' + cond_set_output + '}, ' + str(self.cond_sup_count) + '), (' +
              '(class, ' + str(self.class_label) + '), ' + str(self.rule_sup_count) + ')>')

    # print out rule
    def print_rule(self):
        cond_set_output = ''
        for item in self.cond_set:
            cond_set_output += '(' + str(item) + ', ' + str(self.cond_set[item]) + '), '
```

```python
        cond_set_output = '{' + cond_set_output[:-2] + '}'
        print(cond_set_output + ' -> (class, ' + str(self.class_label) + ')')


# just for test
if __name__ == '__main__':
    cond_set = {0: 1, 1: 1}
    class_label = 1
    dataset = [[1, 1, 1], [1, 1, 1], [1, 2, 1], [2, 2, 1], [2, 2, 1],
               [2, 2, 0], [2, 3, 0], [2, 3, 0], [1, 1, 0], [3, 2, 0]]
    rule_item = RuleItem(cond_set, class_label, dataset)
    rule_item.print()
    rule_item.print_rule()
    print('condsupCount =', rule_item.cond_sup_count)   # should be 3
    print('rulesupCount =', rule_item.rule_sup_count)   # should be 2
    print('support =', rule_item.support)               # should be 0.2
    print('confidence =', rule_item.confidence)         # should be 0.667
```

五、程序清单cba_rg.py

```python
"""
Description: The implementation of CBA-RG algorithm, generating the complete set of CARs (Class Association Rules).
    We just follow up algorithm raised up in the paper without improvement.
Input: a dataset got from pre_process (see pre_processing.py), minsup and minconf
Output: CARs
Author: CBA Studio
Reference: https://www.cs.uic.edu/~hxiao/courses/cs594-slides.pdf
"""
import ruleitem


class FrequentRuleitems:
    """
    A set of frequent k-ruleitems, just using set.
    """
    def __init__(self):
        self.frequent_ruleitems_set = set()

    # get size of set
    def get_size(self):
        return len(self.frequent_ruleitems_set)

    # add a new ruleitem into set
    def add(self, rule_item):
        is_existed = False
```

```
        for item in self.frequent_ruleitems_set:
            if item.class_label == rule_item.class_label:
                if item.cond_set == rule_item.cond_set:
                    is_existed = True
                    break
        if not is_existed:
            self.frequent_ruleitems_set.add(rule_item)

    # append set of ruleitems
    def append(self, sets):
        for item in sets.frequent_ruleitems:
            self.add(item)

    # print out all frequent ruleitems
    def print(self):
        for item in self.frequent_ruleitems_set:
            item.print()


class Car:
    """
    Class Association Rules (Car). If some ruleitems has the same condset, the ruleitem with the highest confidence is
    chosen as the Possible Rule (PR). If there're more than one ruleitem with the same highest confidence, we randomly
    select one ruleitem.
    """
    def __init__(self):
        self.rules = set()
        self.pruned_rules = set()

    # print out all rules
    def print_rule(self):
        for item in self.rules:
            item.print_rule()

    # print out all pruned rules
    def print_pruned_rule(self):
        for item in self.pruned_rules:
            item.print_rule()

    # add a new rule (frequent & accurate), save the ruleitem with the highest confidence when having the same condset
    def _add(self, rule_item, minsup, minconf):
        if rule_item.support >= minsup and rule_item.confidence >= minconf:
            if rule_item in self.rules:
                return
```

```
            for item in self.rules:
                if item.cond_set == rule_item.cond_set and item.confidence < rule_item.confidence:
                    self.rules.remove(item)
                    self.rules.add(rule_item)
                    return
                elif item.cond_set == rule_item.cond_set and item.confidence >= rule_item.confidence:
                    return
            self.rules.add(rule_item)

    # convert frequent ruleitems into car
    def gen_rules(self, frequent_ruleitems, minsup, minconf):
        for item in frequent_ruleitems.frequent_ruleitems_set:
            self._add(item, minsup, minconf)

    # prune rules
    def prune_rules(self, dataset):
        for rule in self.rules:
            pruned_rule = prune(rule, dataset)

            is_existed = False
            for rule in self.pruned_rules:
                if rule.class_label == pruned_rule.class_label:
                    if rule.cond_set == pruned_rule.cond_set:
                        is_existed = True
                        break

            if not is_existed:
                self.pruned_rules.add(pruned_rule)

    # union new car into rules list
    def append(self, car, minsup, minconf):
        for item in car.rules:
            self._add(item, minsup, minconf)


# try to prune rule
def prune(rule, dataset):
    import sys
    min_rule_error = sys.maxsize
    pruned_rule = rule

    # prune rule recursively
    def find_prune_rule(this_rule):
        nonlocal min_rule_error
        nonlocal pruned_rule

```

```
        # calculate how many errors the rule r make in the dataset
        def errors_of_rule(r):
            import cba_cb_m1

            errors_number = 0
            for case in dataset:
                if cba_cb_m1.is_satisfy(case, r) == False:
                    errors_number += 1
            return errors_number

        rule_error = errors_of_rule(this_rule)
        if rule_error < min_rule_error:
            min_rule_error = rule_error
            pruned_rule = this_rule
        this_rule_cond_set = list(this_rule.cond_set)
        if len(this_rule_cond_set) >= 2:
            for attribute in this_rule_cond_set:
                temp_cond_set = dict(this_rule.cond_set)
                temp_cond_set.pop(attribute)
                temp_rule = ruleitem.RuleItem(temp_cond_set, this_rule.
                   class_label, dataset)
                temp_rule_error = errors_of_rule(temp_rule)
                if temp_rule_error <= min_rule_error:
                    min_rule_error = temp_rule_error
                    pruned_rule = temp_rule
                    if len(temp_cond_set) >= 2:
                        find_prune_rule(temp_rule)

    find_prune_rule(rule)
    return pruned_rule


# invoked by candidate_gen, join two items to generate candidate
def join(item1, item2, dataset):
    if item1.class_label != item2.class_label:
        return None
    category1 = set(item1.cond_set)
    category2 = set(item2.cond_set)
    if category1 == category2:
        return None
    intersect = category1 & category2
    for item in intersect:
        if item1.cond_set[item] != item2.cond_set[item]:
            return None
    category = category1 | category2
    new_cond_set = dict()
    for item in category:
        if item in category1:
```

```
            new_cond_set[item] = item1.cond_set[item]
        else:
            new_cond_set[item] = item2.cond_set[item]
    new_ruleitem = ruleitem.RuleItem(new_cond_set, item1.class_label, dataset)
    return new_ruleitem


# similar to Apriori-gen in algorithm Apriori
def candidate_gen(frequent_ruleitems, dataset):
    returned_frequent_ruleitems = FrequentRuleitems()
    for item1 in frequent_ruleitems.frequent_ruleitems_set:
        for item2 in frequent_ruleitems.frequent_ruleitems_set:
            new_ruleitem = join(item1, item2, dataset)
            if new_ruleitem:
                returned_frequent_ruleitems.add(new_ruleitem)
                if returned_frequent_ruleitems.get_size() >= 1000:      # not allow to store more than 1000 ruleitems
                    return returned_frequent_ruleitems
    return returned_frequent_ruleitems


# main method, implementation of CBA-RG algorithm
def rule_generator(dataset, minsup, minconf):
    frequent_ruleitems = FrequentRuleitems()
    car = Car()

    # get large 1-ruleitems and generate rules
    class_label = set([x[-1] for x in dataset])
    for column in range(0, len(dataset[0])-1):
        distinct_value = set([x[column] for x in dataset])
        for value in distinct_value:
            cond_set = {column: value}
            for classes in class_label:
                rule_item = ruleitem.RuleItem(cond_set, classes, dataset)
                if rule_item.support >= minsup:
                    frequent_ruleitems.add(rule_item)
    car.gen_rules(frequent_ruleitems, minsup, minconf)
    cars = car

    last_cars_number = 0
    current_cars_number = len(cars.rules)
    while frequent_ruleitems.get_size() > 0 and current_cars_number <= 2000 and \
                    (current_cars_number - last_cars_number) >= 10:
        candidate = candidate_gen(frequent_ruleitems, dataset)
        frequent_ruleitems = FrequentRuleitems()
        car = Car()
```

```python
        for item in candidate.frequent_ruleitems_set:
            if item.support >= minsup:
                frequent_ruleitems.add(item)
        car.gen_rules(frequent_ruleitems, minsup, minconf)
        cars.append(car, minsup, minconf)
        last_cars_number = current_cars_number
        current_cars_number = len(cars.rules)

    return cars


# just for test
if __name__ == "__main__":
    dataset = [[1, 1, 1], [1, 1, 1], [1, 2, 1], [2, 2, 1], [2, 2, 1],
               [2, 2, 0], [2, 3, 0], [2, 3, 0], [1, 1, 0], [3, 2, 0]]
    minsup = 0.15
    minconf = 0.6
    cars = rule_generator(dataset, minsup, minconf)

    print("CARs:")
    cars.print_rule()

    print("prCARs:")
    cars.prune_rules(dataset)
    cars.print_pruned_rule()
```

六、程序清单cba_cb_m1.py

```python
"""
Description: The implementation of a naive algorithm for CBA-CB: M1. The class Classifier includes the set of selected
    rules and default_class, which can be expressed as <r1, r2, ..., rn, default_class>. Method classifier_builder_m1
    is the main method to implement CBA-CB: M1.
Input: a set of CARs generated from rule_generator (see cab_rg.py) and a dataset got from pre_process
    (see pre_processing.py)
Output: a classifier
Author: CBA Studio
Reference: http://www.docin.com/p-586554186.html
"""
import cba_rg
from functools import cmp_to_key
import sys


# check the rule whether covers the data case (a line in table with the class label at the end of list) or not
# if covers (LHS of rule is the same as the data case, and they belongs
```

```
to the same class), return True;
# else if LHSs are the same while the class labels are different, return False;
# else (LHSs are different), return None
def is_satisfy(datacase, rule):
    for item in rule.cond_set:
        if datacase[item] != rule.cond_set[item]:
            return None
    if datacase[-1] == rule.class_label:
        return True
    else:
        return False


class Classifier:
    """
    This class is our classifier. The rule_list and default_class are useful for outer code.
    """
    def __init__(self):
        self.rule_list = list()
        self.default_class = None
        self._error_list = list()
        self._default_class_list = list()

    # insert a rule into rule_list, then choose a default class, and calculate the errors (see line 8, 10 & 11)
    def insert(self, rule, dataset):
        self.rule_list.append(rule)             # insert r at the end of C
        self._select_default_class(dataset)     # select a default class for the current C
        self._compute_error(dataset)            # compute the total number of errors of C

    # select the majority class in the remaining data
    def _select_default_class(self, dataset):
        class_column = [x[-1] for x in dataset]
        class_label = set(class_column)
        max = 0
        current_default_class = None
        for label in class_label:
            if class_column.count(label) >= max:
                max = class_column.count(label)
                current_default_class = label
        self._default_class_list.append(current_default_class)

    # compute the sum of errors
```

```
    def _compute_error(self, dataset):
        if len(dataset) <= 0:
            self._error_list.append(sys.maxsize)
            return

        error_number = 0

        # the number of errors that have been made by all the selected rules in C
        for case in dataset:
            is_cover = False
            for rule in self.rule_list:
                if is_satisfy(case, rule):
                    is_cover = True
                    break
            if not is_cover:
                error_number += 1

        # the number of errors to be made by the default class in the training set
        class_column = [x[-1] for x in dataset]
        error_number += len(class_column) - class_column.count(self._default_class_list[-1])
        self._error_list.append(error_number)

    # see line 14 and 15, to get the final classifier
    def discard(self):
        # find the first rule p in C with the lowest total number of errors and drop all the rules after p in C
        index = self._error_list.index(min(self._error_list))
        self.rule_list = self.rule_list[:(index+1)]
        self._error_list = None

        # assign the default class associated with p to default_class
        self.default_class = self._default_class_list[index]
        self._default_class_list = None

    # just print out all selected rules and default class in our classifier
    def print(self):
        for rule in self.rule_list:
            rule.print_rule()
        print("default_class:", self.default_class)


# sort the set of generated rules car according to the relation ">", return the sorted rule list
def sort(car):
```

```
    def cmp_method(a, b):
        if a.confidence < b.confidence:     # 1. the confidence of ri > rj
            return 1
        elif a.confidence == b.confidence:
            if a.support < b.support:       # 2. their confidences are the same, but support of ri > rj
                return 1
            elif a.support == b.support:
                if len(a.cond_set) < len(b.cond_set):   # 3. both confidence & support are the same, ri earlier than rj
                    return -1
                elif len(a.cond_set) == len(b.cond_set):
                    return 0
                else:
                    return 1
            else:
                return -1
        else:
            return -1

    rule_list = list(car.rules)
    rule_list.sort(key=cmp_to_key(cmp_method))
    return rule_list


# main method of CBA-CB: M1
def classifier_builder_m1(cars, dataset):
    classifier = Classifier()
    cars_list = sort(cars)
    for rule in cars_list:
        temp = []
        mark = False
        for i in range(len(dataset)):
            is_satisfy_value = is_satisfy(dataset[i], rule)
            if is_satisfy_value is not None:
                temp.append(i)
                if is_satisfy_value:
                    mark = True
        if mark:
            temp_dataset = list(dataset)
            for index in temp:
                temp_dataset[index] = []
            while [] in temp_dataset:
                temp_dataset.remove([])
            dataset = temp_dataset
            classifier.insert(rule, dataset)
    classifier.discard()
```

```python
    return classifier


# just for test
if __name__ == '__main__':
    dataset = [[1, 1, 1], [1, 1, 1], [1, 2, 1], [2, 2, 1], [2, 2, 1],
               [2, 2, 0], [2, 3, 0], [2, 3, 0], [1, 1, 0], [3, 2, 0]]
    minsup = 0.15
    minconf = 0.6
    cars = cba_rg.rule_generator(dataset, minsup, minconf)
    classifier = classifier_builder_m1(cars, dataset)
    classifier.print()

    print()
    dataset = [[1, 1, 1], [1, 1, 1], [1, 2, 1], [2, 2, 1], [2, 2, 1],
               [2, 2, 0], [2, 3, 0], [2, 3, 0], [1, 1, 0], [3, 2, 0]]
    cars.prune_rules(dataset)
    cars.rules = cars.pruned_rules
    classifier = classifier_builder_m1(cars, dataset)
    classifier.print()
```

七、程序清单cba_cb_m2.py

```python
"""
Description: The following code implements an improved version of the algorithm, called CBA-CB: M2. It contains three
    stages. For stage 1, we scan the whole database, to find the cRule and wRule, get the set Q, U and A at the same
    time. In stage 2, for each case d that we could not decide which rule should cover it in stage 1, we go through d
    again to find all rules that classify it wrongly and have a higher precedence than the corresponding cRule of d.
    Finally, in stage 3, we choose the final set of rules to form our final classifer.
Input: a set of CARs generated from rule_generator (see cab_rg.py) and a dataset got from pre_process
    (see pre_processing.py)
Output: a classifier
Author: CBA Studio
"""
import ruleitem
import cba_cb_m1
from functools import cmp_to_key


class Classifier_m2:
    """
    The definition of classifier formed in CBA-CB: M2. It contains a list of rules order by their precedence, a default
```

```python
    class label. The other member are private and useless for outer code.
    """
    def __init__(self):
        self.rule_list = list()
        self.default_class = None
        self._default_class_list = list()
        self._total_errors_list = list()

    # insert a new rule into classifier
    def add(self, rule, default_class, total_errors):
        self.rule_list.append(rule)
        self._default_class_list.append(default_class)
        self._total_errors_list.append(total_errors)

    # discard those rules that introduce more errors. See line 18-20, CBA-CB: M2 (Stage 3).
    def discard(self):
        index = self._total_errors_list.index(min(self._total_errors_list))
        self.rule_list = self.rule_list[:(index + 1)]
        self._total_errors_list = None

        self.default_class = self._default_class_list[index]
        self._default_class_list = None

    # just print out rules and default class label
    def print(self):
        for rule in self.rule_list:
            rule.print_rule()
        print("default_class:", self.default_class)


class Rule(ruleitem.RuleItem):
    """
    A class inherited from RuleItem, adding classCasesCovered and replace field.
    """
    def __init__(self, cond_set, class_label, dataset):
        ruleitem.RuleItem.__init__(self, cond_set, class_label, dataset)
        self._init_classCasesCovered(dataset)
        self.replace = set()

    # initialize the classCasesCovered field
    def _init_classCasesCovered(self, dataset):
        class_column = [x[-1] for x in dataset]
        class_label = set(class_column)
        self.classCasesCovered = dict((x, 0) for x in class_label)

```

```

# convert ruleitem of class RuleItem to rule of class Rule
def ruleitem2rule(rule_item, dataset):
    rule = Rule(rule_item.cond_set, rule_item.class_label, dataset)
    return rule


# finds the highest precedence rule that covers the data case d from the set of rules having the same class as d.
def maxCoverRule_correct(cars_list, data_case):
    for i in range(len(cars_list)):
        if cars_list[i].class_label == data_case[-1]:
            if cba_cb_m1.is_satisfy(data_case, cars_list[i]):
                return i
    return None


# finds the highest precedence rule that covers the data case d from the set of rules having the different class as d.
def maxCoverRule_wrong(cars_list, data_case):
    for i in range(len(cars_list)):
        if cars_list[i].class_label != data_case[-1]:
            temp_data_case = data_case[:-1]
            temp_data_case.append(cars_list[i].class_label)
            if cba_cb_m1.is_satisfy(temp_data_case, cars_list[i]):
                return i
    return None


# compare two rule, return the precedence.
#   -1: rule1 < rule2, 0: rule1 < rule2 (randomly here), 1: rule1 > rule2
def compare(rule1, rule2):
    if rule1 is None and rule2 is not None:
        return -1
    elif rule1 is None and rule2 is None:
        return 0
    elif rule1 is not None and rule2 is None:
        return 1

    if rule1.confidence < rule2.confidence:     # 1. the confidence of ri > rj
        return -1
    elif rule1.confidence == rule2.confidence:
        if rule1.support < rule2.support:       # 2. their confidences are the same, but support of ri > rj
            return -1
        elif rule1.support == rule2.support:
            if len(rule1.cond_set) < len(rule2.cond_set):   # 3.
```

```
                confidence & support are the same, ri earlier than rj
                return 1
            elif len(rule1.cond_set) == len(rule2.cond_set):
                return 0
            else:
                return -1
        else:
            return 1
    else:
        return 1


# finds all the rules in u that wrongly classify the data case and have higher precedences than that of its cRule.
def allCoverRules(u, data_case, c_rule, cars_list):
    w_set = set()
    for rule_index in u:
        # have higher precedences than cRule
        if compare(cars_list[rule_index], c_rule) > 0:
            # wrongly classify the data case
            if cba_cb_m1.is_satisfy(data_case, cars_list[rule_index]) == False:
                w_set.add(rule_index)
    return w_set


# counts the number of training cases in each class
def compClassDistr(dataset):
    class_distr = dict()

    if len(dataset) <= 0:
        class_distr = None

    dataset_without_null = dataset
    while [] in dataset_without_null:
        dataset_without_null.remove([])

    class_column = [x[-1] for x in dataset_without_null]
    class_label = set(class_column)
    for c in class_label:
        class_distr[c] = class_column.count(c)
    return class_distr


# sort the rule list order by precedence
def sort_with_index(q, cars_list):
    def cmp_method(a, b):
        # 1. the confidence of ri > rj
```

```
        if cars_list[a].confidence < cars_list[b].confidence:
            return 1
        elif cars_list[a].confidence == cars_list[b].confidence:
            # 2. their confidences are the same, but support of ri > rj
            if cars_list[a].support < cars_list[b].support:
                return 1
            elif cars_list[a].support == cars_list[b].support:
                # 3. both confidence & support are the same, ri earlier than rj
                if len(cars_list[a].cond_set) < len(cars_list[b].cond_set):
                    return -1
                elif len(cars_list[a].cond_set) == len(cars_list[b].cond_set):
                    return 0
                else:
                    return 1
            else:
                return -1
        else:
            return -1

    rule_list = list(q)
    rule_list.sort(key=cmp_to_key(cmp_method))
    return set(rule_list)


# get how many errors the rule wrongly classify the data case
def errorsOfRule(rule, dataset):
    error_number = 0
    for case in dataset:
        if case:
            if cba_cb_m1.is_satisfy(case, rule) == False:
                error_number += 1
    return error_number


# choose the default class (majority class in remaining dataset)
def selectDefault(class_distribution):
    if class_distribution is None:
        return None

    max = 0
    default_class = None
    for index in class_distribution:
        if class_distribution[index] > max:
            max = class_distribution[index]
            default_class = index
```

```
    return default_class


# count the number of errors that the default class will make in the remaining training data
def defErr(default_class, class_distribution):
    if class_distribution is None:
        import sys
        return sys.maxsize

    error = 0
    for index in class_distribution:
        if index != default_class:
            error += class_distribution[index]
    return error


# main method, implement the whole classifier builder
def classifier_builder_m2(cars, dataset):
    classifier = Classifier_m2()

    cars_list = cba_cb_m1.sort(cars)
    for i in range(len(cars_list)):
        cars_list[i] = ruleitem2rule(cars_list[i], dataset)

    # stage 1
    q = set()
    u = set()
    a = set()
    mark_set = set()
    for i in range(len(dataset)):
        c_rule_index = maxCoverRule_correct(cars_list, dataset[i])
        w_rule_index = maxCoverRule_wrong(cars_list, dataset[i])
        if c_rule_index is not None:
            u.add(c_rule_index)
        if c_rule_index:
            cars_list[c_rule_index].classCasesCovered[dataset[i][-1]] += 1
        if c_rule_index and w_rule_index:
            if compare(cars_list[c_rule_index], cars_list[w_rule_index]) > 0:
                q.add(c_rule_index)
                mark_set.add(c_rule_index)
            else:
                a.add((i, dataset[i][-1], c_rule_index, w_rule_index))
        elif c_rule_index is None and w_rule_index is not None:
            a.add((i, dataset[i][-1], c_rule_index, w_rule_index))

```

```
    # stage 2
    for entry in a:
        if cars_list[entry[3]] in mark_set:
            if entry[2] is not None:
                cars_list[entry[2]].classCasesCovered[entry[1]] -= 1
            cars_list[entry[3]].classCasesCovered[entry[1]] += 1
        else:
            if entry[2] is not None:
                w_set = allCoverRules(u, dataset[entry[0]], cars_list[entry[2]], cars_list)
            else:
                w_set = allCoverRules(u, dataset[entry[0]], None, cars_list)
            for w in w_set:
                cars_list[w].replace.add((entry[2], entry[0], entry[1]))
                cars_list[w].classCasesCovered[entry[1]] += 1
            q |= w_set

    # stage 3
    rule_errors = 0
    q = sort_with_index(q, cars_list)
    data_cases_covered = list([False] * len(dataset))
    for r_index in q:
        if cars_list[r_index].classCasesCovered[cars_list[r_index].class_label] != 0:
            for entry in cars_list[r_index].replace:
                if data_cases_covered[entry[1]]:
                    cars_list[r_index].classCasesCovered[entry[2]] -= 1
                else:
                    if entry[0] is not None:
                        cars_list[entry[0]].classCasesCovered[entry[2]] -= 1
            for i in range(len(dataset)):
                datacase = dataset[i]
                if datacase:
                    is_satisfy_value = cba_cb_m1.is_satisfy(datacase, cars_list[r_index])
                    if is_satisfy_value:
                        dataset[i] = []
                        data_cases_covered[i] = True
            rule_errors += errorsOfRule(cars_list[r_index], dataset)
            class_distribution = compClassDistr(dataset)
            default_class = selectDefault(class_distribution)
            default_errors = defErr(default_class, class_distribution)
            total_errors = rule_errors + default_errors
            classifier.add(cars_list[r_index], default_class, total_errors)
    classifier.discard()
```

```python

    return classifier


# just for test
if __name__ == "__main__":
    import cba_rg

    dataset = [[1, 1, 1], [1, 1, 1], [1, 2, 1], [2, 2, 1], [2, 2, 1],
               [2, 2, 0], [2, 3, 0], [2, 3, 0], [1, 1, 0], [3, 2, 0]]
    minsup = 0.15
    minconf = 0.6
    cars = cba_rg.rule_generator(dataset, minsup, minconf)
    classifier = classifier_builder_m2(cars, dataset)
    classifier.print()

    print()
    dataset = [[1, 1, 1], [1, 1, 1], [1, 2, 1], [2, 2, 1], [2, 2, 1],
               [2, 2, 0], [2, 3, 0], [2, 3, 0], [1, 1, 0], [3, 2, 0]]
    cars.prune_rules(dataset)
    cars.rules = cars.pruned_rules
    classifier = classifier_builder_m2(cars, dataset)
    classifier.print()
```

八、程序清单validation.py

```python
"""
Description: This is our experimental code. We provide 4 test modes for experiments:
    10-fold cross-validations on CBA (M1) without pruning
    10-fold cross-validations on CBA (M1) with pruning
    10-fold cross-validations on CBA (M2) without pruning
    10-fold cross-validations on CBA (M2) with pruning
Input: the relative directory path of data file and scheme file
Output: the experimental results (similar to Table 1: Experiment Results in this paper)
Author: CBA Studio
"""
from read import read
from pre_processing import pre_process
from cba_rg import rule_generator
from cba_cb_m1 import classifier_builder_m1
from cba_cb_m1 import is_satisfy
from cba_cb_m2 import classifier_builder_m2
import time
import random


# calculate the error rate of the classifier on the dataset
```

```
def get_error_rate(classifier, dataset):
    size = len(dataset)
    error_number = 0
    for case in dataset:
        is_satisfy_value = False
        for rule in classifier.rule_list:
            is_satisfy_value = is_satisfy(case, rule)
            if is_satisfy_value == True:
                break
        if is_satisfy_value == False:
            if classifier.default_class != case[-1]:
                error_number += 1
    return error_number / size


# 10-fold cross-validations on CBA (M1) without pruning
def cross_validate_m1_without_prune(data_path, scheme_path, minsup=0.01,
    minconf=0.5):
    data, attributes, value_type = read(data_path, scheme_path)
    random.shuffle(data)
    dataset = pre_process(data, attributes, value_type)

    block_size = int(len(dataset) / 10)
    split_point = [k * block_size for k in range(0, 10)]
    split_point.append(len(dataset))

    cba_rg_total_runtime = 0
    cba_cb_total_runtime = 0
    total_car_number = 0
    total_classifier_rule_num = 0
    error_total_rate = 0

    for k in range(len(split_point)-1):
        print("\nRound %d:" % k)

        training_dataset = dataset[:split_point[k]] + dataset[split_point
            [k+1]:]
        test_dataset = dataset[split_point[k]:split_point[k+1]]

        start_time = time.time()
        cars = rule_generator(training_dataset, minsup, minconf)
        end_time = time.time()
        cba_rg_runtime = end_time - start_time
        cba_rg_total_runtime += cba_rg_runtime

        start_time = time.time()
        classifier_m1 = classifier_builder_m1(cars, training_dataset)
        end_time = time.time()
```

```python
        cba_cb_runtime = end_time - start_time
        cba_cb_total_runtime += cba_cb_runtime

        error_rate = get_error_rate(classifier_m1, test_dataset)
        error_total_rate += error_rate

        total_car_number += len(cars.rules)
        total_classifier_rule_num += len(classifier_m1.rule_list)

        print("CBA's error rate without pruning: %.1lf%%" % (error_rate * 100))
        print("No. of CARs without pruning: %d" % len(cars.rules))
        print("CBA-RG's run time without pruning: %.2lf s" % cba_rg_runtime)
        print("CBA-CB M1's run time without pruning: %.2lf s" % cba_cb_runtime)
        print("No. of rules in classifier of CBA-CB M1 without pruning: %d" % len(classifier_m1.rule_list))

    print("\nAverage CBA's error rate without pruning: %.1lf%%" % (error_total_rate / 10 * 100))
    print("Average No. of CARs without pruning: %d" % int(total_car_number / 10))
    print("Average CBA-RG's run time without pruning: %.2lf s" % (cba_rg_total_runtime / 10))
    print("Average CBA-CB M1's run time without pruning: %.2lf s" % (cba_cb_total_runtime / 10))
    print("Average No. of rules in classifier of CBA-CB M1 without pruning: %d" % int(total_classifier_rule_num / 10))


# 10-fold cross-validations on CBA (M1) with pruning
def cross_validate_m1_with_prune(data_path, scheme_path, minsup=0.01, minconf=0.5):
    data, attributes, value_type = read(data_path, scheme_path)
    random.shuffle(data)
    dataset = pre_process(data, attributes, value_type)

    block_size = int(len(dataset) / 10)
    split_point = [k * block_size for k in range(0, 10)]
    split_point.append(len(dataset))

    cba_rg_total_runtime = 0
    cba_cb_total_runtime = 0
    total_car_number = 0
    total_classifier_rule_num = 0
    error_total_rate = 0

```

```
    for k in range(len(split_point)-1):
        print("\nRound %d:" % k)

        training_dataset = dataset[:split_point[k]] + dataset[split_point[k+1]:]
        test_dataset = dataset[split_point[k]:split_point[k+1]]

        start_time = time.time()
        cars = rule_generator(training_dataset, minsup, minconf)
        cars.prune_rules(training_dataset)
        cars.rules = cars.pruned_rules
        end_time = time.time()
        cba_rg_runtime = end_time - start_time
        cba_rg_total_runtime += cba_rg_runtime

        start_time = time.time()
        classifier_m1 = classifier_builder_m1(cars, training_dataset)
        end_time = time.time()
        cba_cb_runtime = end_time - start_time
        cba_cb_total_runtime += cba_cb_runtime

        error_rate = get_error_rate(classifier_m1, test_dataset)
        error_total_rate += error_rate

        total_car_number += len(cars.rules)
        total_classifier_rule_num += len(classifier_m1.rule_list)

        print("CBA's error rate with pruning: %.1lf%%" % (error_rate * 100))
        print("No. of CARs with pruning: %d" % len(cars.rules))
        print("CBA-RG's run time with pruning: %.2lf s" % cba_rg_runtime)
        print("CBA-CB M1's run time with pruning: %.2lf s" % cba_cb_runtime)
        print("No. of rules in classifier of CBA-CB M1 with pruning: %d" % len(classifier_m1.rule_list))

    print("\nAverage CBA's error rate with pruning: %.1lf%%" % (error_total_rate / 10 * 100))
    print("Average No. of CARs with pruning: %d" % int(total_car_number / 10))
    print("Average CBA-RG's run time with pruning: %.2lf s" % (cba_rg_total_runtime / 10))
    print("Average CBA-CB M1's run time with pruning: %.2lf s" % (cba_cb_total_runtime / 10))
    print("Average No. of rules in classifier of CBA-CB M1 with pruning: %d" % int(total_classifier_rule_num / 10))
```

```
# 10-fold cross-validations on CBA (M2) without pruning
def cross_validate_m2_without_prune(data_path, scheme_path, minsup=0.01,
    minconf=0.5):
    data, attributes, value_type = read(data_path, scheme_path)
    random.shuffle(data)
    dataset = pre_process(data, attributes, value_type)

    block_size = int(len(dataset) / 10)
    split_point = [k * block_size for k in range(0, 10)]
    split_point.append(len(dataset))

    cba_rg_total_runtime = 0
    cba_cb_total_runtime = 0
    total_car_number = 0
    total_classifier_rule_num = 0
    error_total_rate = 0

    for k in range(len(split_point)-1):
        print("\nRound %d:" % k)

        training_dataset = dataset[:split_point[k]] + dataset[split_point
            [k+1]:]
        test_dataset = dataset[split_point[k]:split_point[k+1]]

        start_time = time.time()
        cars = rule_generator(training_dataset, minsup, minconf)
        end_time = time.time()
        cba_rg_runtime = end_time - start_time
        cba_rg_total_runtime += cba_rg_runtime

        start_time = time.time()
        classifier_m2 = classifier_builder_m2(cars, training_dataset)
        end_time = time.time()
        cba_cb_runtime = end_time - start_time
        cba_cb_total_runtime += cba_cb_runtime

        error_rate = get_error_rate(classifier_m2, test_dataset)
        error_total_rate += error_rate

        total_car_number += len(cars.rules)
        total_classifier_rule_num += len(classifier_m2.rule_list)

        print("CBA's error rate without pruning: %.1lf%%" % (error_rate *
             100))
        print("No. of CARs without pruning: %d" % len(cars.rules))
        print("CBA-RG's run time without pruning: %.2lf s" %
            cba_rg_runtime)
        print("CBA-CB M2's run time without pruning: %.2lf s" %
```

```
            cba_cb_runtime)
        print("No. of rules in classifier of CBA-CB M2 without pruning: %d" % len(classifier_m2.rule_list))

    print("\nAverage CBA's error rate without pruning: %.1lf%%" % (error_total_rate / 10 * 100))
    print("Average No. of CARs without pruning: %d" % int(total_car_number / 10))
    print("Average CBA-RG's run time without pruning: %.2lf s" % (cba_rg_total_runtime / 10))
    print("Average CBA-CB M2's run time without pruning: %.2lf s" % (cba_cb_total_runtime / 10))
    print("Average No. of rules in classifier of CBA-CB M2 without pruning: %d" % int(total_classifier_rule_num / 10))


# 10-fold cross-validations on CBA (M2) with pruning
def cross_validate_m2_with_prune(data_path, scheme_path, minsup=0.01, minconf=0.5):
    data, attributes, value_type = read(data_path, scheme_path)
    random.shuffle(data)
    dataset = pre_process(data, attributes, value_type)

    block_size = int(len(dataset) / 10)
    split_point = [k * block_size for k in range(0, 10)]
    split_point.append(len(dataset))

    cba_rg_total_runtime = 0
    cba_cb_total_runtime = 0
    total_car_number = 0
    total_classifier_rule_num = 0
    error_total_rate = 0

    for k in range(len(split_point)-1):
        print("\nRound %d:" % k)

        training_dataset = dataset[:split_point[k]] + dataset[split_point[k+1]:]
        test_dataset = dataset[split_point[k]:split_point[k+1]]

        start_time = time.time()
        cars = rule_generator(training_dataset, minsup, minconf)
        cars.prune_rules(training_dataset)
        cars.rules = cars.pruned_rules
        end_time = time.time()
        cba_rg_runtime = end_time - start_time
        cba_rg_total_runtime += cba_rg_runtime

```

```
        start_time = time.time()
        classifier_m2 = classifier_builder_m2(cars, training_dataset)
        end_time = time.time()
        cba_cb_runtime = end_time - start_time
        cba_cb_total_runtime += cba_cb_runtime

        error_rate = get_error_rate(classifier_m2, test_dataset)
        error_total_rate += error_rate

        total_car_number += len(cars.rules)
        total_classifier_rule_num += len(classifier_m2.rule_list)

        print("CBA's error rate with pruning: %.1lf%%" % (error_rate * 100))
        print("No. of CARs without pruning: %d" % len(cars.rules))
        print("CBA-RG's run time with pruning: %.2lf s" % cba_rg_runtime)
        print("CBA-CB M2's run time with pruning: %.2lf s" % cba_cb_runtime)
        print("No. of rules in classifier of CBA-CB M2 with pruning: %d" % len(classifier_m2.rule_list))

    print("\nAverage CBA's error rate with pruning: %.1lf%%" % (error_total_rate / 10 * 100))
    print("Average No. of CARs with pruning: %d" % int(total_car_number / 10))
    print("Average CBA-RG's run time with pruning: %.2lf s" % (cba_rg_total_runtime / 10))
    print("Average CBA-CB M2's run time with pruning: %.2lf s" % (cba_cb_total_runtime / 10))
    print("Average No. of rules in classifier of CBA-CB M2 with pruning: %d" % int(total_classifier_rule_num / 10))


# test entry goes here
if __name__ == "__main__":
    # using the relative path, all data sets are stored in datasets directory
    test_data_path = 'datasets/australian.data'
    test_scheme_path = 'datasets/australian.names'

    # just choose one mode to experiment by removing one line comment and running
    cross_validate_m1_without_prune(test_data_path, test_scheme_path)
    # cross_validate_m1_with_prune(test_data_path, test_scheme_path)
    # cross_validate_m2_without_prune(test_data_path, test_scheme_path)
    # cross_validate_m1_with_prune(test_data_path, test_scheme_path)
```

附录 B　本文使用的数据集格式与示例

本文中的数据集全部选取自 UCI 机器学习资料库 [2]，但是由于网站中数据集格式千差万别，不便于我们的程序读取和处理数据，所以我们在不改变数据内容的前提下，对所采用的 30 个数据集进行格式归一化处理。

对于一个数据集，我们需要两份文件。一份以`.data`为后缀，里面存储的是以逗号分隔的 csv 格式的样本观测数据，每一行为一个样本，每一列对应一个属性，最后一列为该样本所属的类的标号。若某一样本的某一属性值缺失，则将其用英文问号“?”标注。另一份以`.names`为后缀，描述了该数据集的格式。第一行为属性列的名称，以逗号分隔，最后一列命名为“class”，以表示所属类。第二行为各属性值的性质，若属性值为连续型（又称“数值型”），则标记为“numerical”；若属性值为离散型（又称“标签型”），则标记为“categorical”；最后一列为类标号，记为“label”。必须保证这两行的列数相等，并且和`.data`文件的列数一致。

以著名的 iris 数据集为例。下面给出的是`iris.data`文件中前 10 行数据，它们对应 10 个样本，前 4 列为观测到的属性值，最后一列对应这个样本所属的类。

文件清单 1　`iris.data` 文件的前 10 行

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

接着给出的是`iris.names`文件的内容。第 1 行前 4 列均为各属性的名称；最后 1 列为“class”，表明是类标号。第 2 行前四列全为“numerical”，表示这些属性值都是连续型；最后 1 列的“label”表示其为类的标号。

文件清单 2　`iris.names` 文件

```
sepal length,sepal width,petal length,petal width,class
numerical,numerical,numerical,numerical,label
```

其他的数据集均按照上述规则进行归一化处理。运用这些数据集，可以对我们的程序进行实例验证，以测试我们实现的性能。