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 ε = 0.5。即

m =〈 {[ col[m] ∧ x ?}) 

上式中， m 表⽰缺失值； mode( ·) 操作表⽰取 · 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：

R+ → {cheap, medium, 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) = Ent(S1 )+ Ent(S2 ) (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 Tmin ，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) = −  P(xi )log2 P(xi )

其中， n 表⽰样本集 S 中的类的种数， P(xi ) 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又， 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) <  + 

(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

∆(A, T;S) = log2 (3k − 2) − [k · Ent(S) − k1 · Ent(S1 ) − k2 · Ent(S2 )]

其中 ki 表⽰集合 Si 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) = 2  2  = 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  1.0 | Yes  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 S1 的信息熵 Ent(S1 ) = −1 × log2 1 = 0。而另一部分 S2 的信息熵

Ent(S2 ) = 2  

log2

2 3

= 0.92。所以此处划分后的类信息熵 E =

2 5

Ent(S1 )+

3 5

Ent(S2 ) = 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 ∆ = log2 (35 − 2)− [5× Ent(S)−2×Ent(S1 )−3×Ent(S2 ) = 5.82。由此，可以计算出信息增益下界 infGain =  = 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 |
| 1: T = ∅ |
| 2: |
| 3: **function** Split(S) |
| 4: sort(S) |
| 5: W = ∅ |
| 6: **for** each candidate c in S **do** |
| 7: calculate g (c) = Gain(A,c;S) |
| 8: **if** (2) is not satisfied **then** |
| 9: W = W ∪ c |
| 10: **end if** |
| 11: **end for** |
| 12: **if** W = ∅ **then** |
| 13: **return** ∅ |
| 14: **else** |
| 15: **return** arg max g(c)  c=W |
| 16: **end if** |
| 17: **end function** |
| 18: |
| 19: **function** Partition(S) |
| 20: t = Split(S) |
| 21: **if** t = ∅ **then** |
| 22: **return** |
| 23: **else** |
| 24: T = T ∪ t |
| 25: S1 = {x : x ∈ S ∧ x < t} |
| 26: S2 = {x : x ∈ S ∧ x ≥ t} |
| 27: Partition(S1) |
| 28: Partition(S2) |
| 29: **end if** |
| 30: **end function** |
| 31: |
| 32: **function** main |
| 33: Partition(S) |
| 34: sort(T) |
| 35: **return** T |
| 36: **end function** |