Pattern Recognition Letters 54 (2015) 63–68

Contents lists available at ScienceDirect

Pattern Recognition Letters

journal homepage: www.elsevier.com/locate/patrec

Quadratic programming for class ordering in rule induction[☆]

Olcay Taner Yıldız[*]

Işık University, Meşrutiyet Koyu Universite Sokak, Dış Kapı No. 2 Şile/Istanbul, Turkey

ARTICLE INFO

Article history:
Received 7 July 2014
Available online 12 December 2014

Keywords:
Rule induction
Quadratic programming
Class ordering

ABSTRACT

Separate-and-conquer type rule induction algorithms such as Ripper, solve a $K > 2$ class problem by converting it into a sequence of $K - 1$ two-class problems. As a usual heuristic, the classes are fed into the algorithm in the order of increasing prior probabilities. Although the heuristic works well in practice, there is much room for improvement. In this paper, we propose a novel approach to improve this heuristic. The approach transforms the ordering search problem into a quadratic optimization problem and uses the solution of the optimization problem to extract the optimal ordering. We compared new Ripper (guided by the ordering found with our approach) with original Ripper (guided by the heuristic ordering) on 27 datasets. Simulation results show that our approach produces rulesets that are significantly better than those produced by the original Ripper.

1. Introduction

Rule induction algorithms learn a ruleset from a training set. A ruleset is typically an ordered list of rules, where a rule contains a conjunction of terms and a class code which is the label assigned to an instance that is covered by the rule [9]. The terms are of the form $x_i = v$, $x_i < \theta$ or $x_i \geq \theta$, depending on respectively whether the input feature x_i is discrete or continuous. There is also a default class assigned to instances not covered by any rule. An example ruleset containing two rules for famous iris problem is:

If $(x_3 < 1.9)$ **and** $(x_4 \geq 5.1)$ **Then** class = iris-setosa
Else
  If $(x_3 < 4.7)$ **Then** class = iris-versicolor
  Else class = iris-virginica

There are two main groups of rule learning algorithms. Separate-and-conquer algorithms and divide-and-conquer algorithms. Separate-and-conquer algorithms first find the best rule that explains part of the training data. After *separating* the examples those are covered by this rule, the algorithms *conquer* remaining data by finding next best rules recursively. Consequently, previously learned rules directly influence the data of the other rules. Separate-and-conquer algorithms use hill-climbing [8,13], beam search [7,22], best first search [16], genetic algorithms [24], ant colony optimization [15,18], fuzzy rough set [4,20,25], neural networks [12] to extract rules from data.

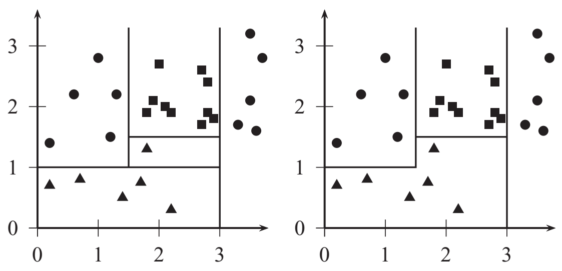
Divide-and-conquer algorithms greedily find the split that best separates data in terms of some predefined impurity measure such as information gain, entropy, Gini index, etc. After *dividing* examples according to the best split, the algorithms *conquer* each part of the data by finding next best splits recursively. In this case, previously learned splits in the parent nodes directly influence the data of the descendant nodes. Divide-and-conquer algorithms use stepwise-improvement [6,17], neural networks [11], linear discriminant analysis [10,14,26], support vector machines [2,23] to learn trees from data.

This paper is mainly related with the algorithms following separate and conquer strategy. According to this strategy, when a rule is learned for class C_i, the covered examples are removed from the training set. This procedure proceeds until no examples remain from class C_i in the training set. If we have two classes, we separate positive class from negative class. But if we have $K > 2$ classes, as a heuristic, every class is classified in the order of their increasing prior probabilities, i.e., in the order of their sample size. The aim of this paper is (i) to determine the effect of this ordering on the performance of the algorithms and (ii) to propose a better algorithm for selecting the ordering.

Ripper, arguably one of the best algorithms following separate-and-conquer strategy, learns rules to separate a positive class from a negative class. In the example above, Ripper first learns rules to separate class *iris-setosa* from both classes *iris-versicolor* and *iris-virginica*, then learns rules to separate class *iris-versicolor* from class *iris-virginica*. The ordering of classes is selected heuristically and may not be optimal in terms of error and/or complexity. In Fig. 1 we see an

[☆] This paper has been recommended for acceptance by Eckart Michaelsen.
[*] Tel.: +90 216 528 7157, 90 216 3341508; fax: +90 216 710 2872.
E-mail address: olcaytaner@isikun.edu.tr, olcaytaner@gmail.com

http://dx.doi.org/10.1016/j.patrec.2014.12.002
0167-8655/© 2015 Elsevier B.V. All rights reserved.



```
If ($x_1$ < 3) and ($x_2$ < 1)
  Then class = ▲
Else
  If ($x_1$ > 1.5) and ($x_2$ < 1.5)
    Then class = ▲
  Else
    If ($x_1$ < 1.5)
      Then class = ●
    Else
      If ($x_1$ > 3)
        Then class = ●
      Else class = ■
```

```
If ($x_1$ < 1.5) and ($x_2$ > 1)
  Then class = ●
Else
  If ($x_1$ > 3)
    Then class = ●
  Else
    If ($x_1$ < 1.5)
      Then class = ▲
    Else class = ■
```

Fig. 1. For two different class orderings, separation of data and learned rulesets.

```
1  Ruleset Ripper($D$, $\pi$)
2    $RS$ = {}
3    for $p$ = 1 to $K$
4      Pos = $\pi_p$, Neg = $\pi_{p+1}, \ldots, \pi_K$
5      $RS_p$ = {}
6      DL = DescLen($RS$,Pos,Neg)
7      while $D$ contains positive samples do
8        Divide $D$ into Grow set $G$ and Prune set $P$
9        $r$ = GrowRule($G$)
10       PruneRule($r$, $P$)
11       DL' = DescLen($RS_p + r$, Pos, Neg)
12       if DL' > $DL$ + 64
13         $RS$ = PruneRuleSet($RS_p + r$, Pos, Neg)
14         return $RS$
15       else
16         $RS_p = RS_p + r$
17         Remove examples covered by $r$ from $D$
18     for $i$ = 1 to 2
19       OptimizeRuleset($RS_p$, $D$)
20     $RS = RS + RS_p$
21   return $RS$
```

Fig. 2. Pseudocode for learning a ruleset using Ripper on dataset D according to class ordering π.

```
1  Rule GrowRule($D$)
2    $r$ = {}
3    while $r$ covers negative examples
4      Use exhaustive search to find best condition $c$
5      $r = r \cup c$
6  return $r$
```

Fig. 3. Pseudocode for growing a rule using dataset D.

example case, where two different orderings produce two different rulesets with the same error but different complexity, one composed of four rules with six terms, other composed of three rules with four terms. Although we prefer the second ordering, the heuristic may lead us to the first ordering.

In this paper, we propose an algorithm to find the optimal class ordering. Pairwise error approximation (PEA) assumes that the error of an ordering is the sum of $K(K-1)/2$ pairwise errors of classes. We train a random set of orderings and use the test error of them as training data to estimate the pairwise errors. Given the estimated pairwise errors, the algorithm searches for the optimal ordering exhaustively.

In the earlier version of this work [1], we proposed unconstrained quadratic optimization for extracting the optimal ordering; this present paper extends (i) the quadratic optimization by both formulation and explanation, (ii) the experiments significantly to include newer results on significantly more datasets. In the former publication, the quadratic optimization is not constrained and therefore can be easily (but sometimes wrongly in terms of pairwise error estimations) solved by just taking derivatives. In this paper, we constrain the quadratic optimization problem, and now the pairwise error estimations must obey the constraints.

This paper is organized as follows: In Section 2, we explain the rule induction algorithm Ripper. In Section 3 we explain our novel PEA algorithm. We give our experimental results in Section 4 and conclude in Section 5.

2. Ripper

Ripper learns rules from scratch starting from an empty ruleset. It has two phases: in the first phase, it builds an initial set of rules, one at a time, and in the second phase, it optimizes the ruleset m times [8].

The pseudocode for learning ruleset from examples using Ripper is given in Fig. 2. When there are $K > 2$ classes, the classes of the dataset are increasingly sorted according to their prior probabilities resulting in permutation, π (line 1). For each class π_p, its examples are considered as positive and the examples of the remaining classes $\pi_{p+1}, \ldots, \pi_K$ are considered as negative (line 4). Rules are grown (line 9), pruned (line 10) and added (line 16) one by one to the ruleset. If the recent ruleset's description length is 64 bits more than the previous ruleset's description length rule adding stops and the ruleset is pruned (lines 12–14). The description length of a ruleset is the number of bits to represent all the rules in the ruleset, plus the description length of examples not covered by the ruleset. Ripper uses

$$\text{DescLen} = ||k|| + k\log_2\frac{n}{k} + (n-k)\log_2\frac{n}{n-k} \tag{1}$$

bits to send rule r with k conditions, where n is the number of possible conditions that could appear in a rule and $||k||$ is the number of bits needed to send the integer k [8]. If there are no remaining positive examples (line 7) rule adding stops. After learning a ruleset, it is optimized twice (line 18).

Fig. 3 shows the pseudocode of growing a rule. Learning starts with an empty rule (line 2), and conditions are added one by one. At each iteration, the algorithm finds the condition with maximum information gain on the dataset D (line 4) by using the information gain defined as follows

$$\text{Gain}(R', R) = s\left(\log_2\frac{N'_+}{N'} - \log_2\frac{N_+}{N}\right) \tag{2}$$

where N is the number of examples, N_+ is the number of true positives covered by rule R and N', N'_+ represent the same descriptions for the candidate rule R'. s is the number of true positives after adding the condition in R [19]. When the best condition is found, we add that condition to the rule (line 5). We stop adding conditions to a rule when there are no negative examples left in the grow set (line 3).

The pseudocode for pruning a rule is given in Fig. 4. We search for a condition whose removal causes the most increase in rule value metric (lines 9–12) and if such a condition is found, we remove it (lines 14 and 15). Rule value metric is calculated by

$$M(R, D) = \frac{N_+ - N_-}{N_+ + N_-} \tag{3}$$

where N_+ and N_- are the number of positive and negative examples covered by R in the pruning set D. We stop pruning when there is no more improvement in rule value metric (line 4).