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.

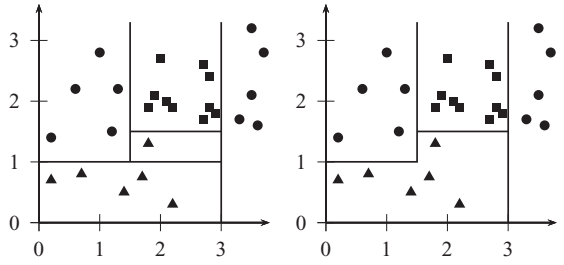
This paper is mainly related with the algorithms following separate and conquer strategy. According to this strategy, when a rule is learned for class C_i, the covered examples are removed from the training set. This procedure proceeds until no examples remain from class C_i in the training set. If we 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/



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, π)
2    RS = {}
3    for p = 1 to K
4      Pos = π_p, Neg = π_{p+1}, ..., π_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 ∪ 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).

```
1  Rule PruneRule(r, D)
2    improved = true
3    M_best = M(r, D)
4    while improved do
5      improved = false
6      for each condition c in r
7        r = r − c
8        M_current = M(r, D)
9        if (M_current ≥ M_best)
10         improved = true
11         c_best = c
12         M_best = M_current
13       r = r ∪ c
14     if improved
15       r = r − c_best
16   return r
```

Fig. 4. Pseudocode for pruning rule r using dataset D.

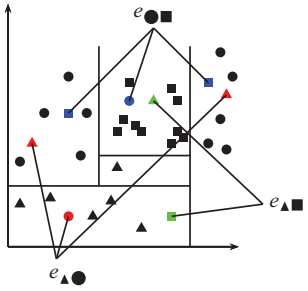


Fig. 5. The expected error of the ordering ▲●■ and its components (e_{ij}'s) for a dataset with $K = 3$.

3. Pairwise error approximation

In the training of Ripper, the ordering of classes is selected heuristically and may not be optimal in terms of error and/or complexity. A common approach that is used in *covering algorithms* is training the classes in the order of their increasing prior probabilities. At each iteration of the covering algorithms, the examples (even false positives) covered by the rule are removed from the training set. Removing examples during the training causes order dependencies between rules [3]. The last learned rule is dependent on the previous rules and their covered examples.

Our proposed algorithm, pairwise error approximation, assumes that the expected error of an ordering, that is, the expected error of the Ripper algorithm trained with that ordering, is the sum of $K(K-1)/2$ pairwise expected errors of classes. The algorithm then tries to estimate the expected error contribution of each pair of classes. More formally, the expected error of the Ripper algorithm with ordering π is defined as

$$E_\pi = \sum_{j=1}^{K-1} \sum_{k>j}^{K} e_{\pi_j \pi_k} \tag{4}$$

where π_j represents jth class in permutation π and $e_{\pi_j\pi_k}$ represents the error contribution of separation of class π_j from class π_k. For example, the expected error of the ordering ▲●■ (three class problem) is defined as

$$E_{▲●■} = e_{▲●} + e_{▲■} + e_{●■} \tag{5}$$

$e_{\pi_j\pi_k}$ contains two types of instances (see Fig. 5):

- False positives, instances of class π_k covered by the rules of class π_j, in other words, given class confusion matrix C, the element c_{kj}.
- False negatives, instances of class π_j covered by the rules of class π_k, in other words, given class confusion matrix C, the element c_{jk}.

```
1  PEA(D)
2    E_best = ∞
3    for i = 1 to T
4      π^(i) = RandomOrdering()
5      E_π^(i) = Ripper(D, π^(i))
6    QP = constructQuadraticProgram(π, E_π)
7    solveQuadraticProgram(QP)
8    for i = 1 to K!
9      E = 0
10     for j = 1 to N
11       for k = j + 1 to N
12         E += e_{π_j^(i) π_k^(i)}
13     if E < E_best
14       E_best = E
15       π^best = π^(i)
16   return π^best
```

Fig. 6. Pseudocode of PEA for dataset D with K classes.

Since we cannot estimate e_{jk} from a single ordering, we run Ripper algorithm T times with T random orderings $\pi^{(i)}$ and get the test errors $E_{\pi^{(i)}}$. Average estimation error over T runs is then defined as

$$E_t = \frac{\sum_{i=1}^{T} (E_{\pi^{(i)}} - \hat{E}_{\pi^{(i)}})^2}{T} \tag{6}$$

An ordering with K classes has $K(K-1)/2$ different e_{jk} pairs. Since the number of all possible pairs is $K(K-1)$, each e_{jk} will appear $T/2$ times in T random orderings approximately.

In order to minimize the average estimation error, we form the following quadratic programming problem

$$\begin{aligned} \text{Min.} \quad & \sum_{i=1}^{T} (E_{\pi^{(i)}} - \hat{E}_{\pi^{(i)}})^2 \\ \text{s.t. } \hat{E}_{\pi^{(i)}} &= \sum_{j=1}^{K-1} \sum_{k>j}^{K} \hat{e}_{\pi_j^{(i)} \pi_k^{(i)}} \quad \forall i = 1, \ldots, T \\ \hat{e}_{jk} + \hat{e}_{kj} &\le \frac{N_j + N_k}{N} \quad \forall j, k = 1, \ldots, K \\ \hat{e}_{jk} &\ge 0 \qquad\quad \forall j, k = 1, \ldots, K \end{aligned} \tag{7}$$

where $\hat{e}_{jk}$'s are unknown variables and N_j represents the number of examples of class j.

After solving the quadratic program, we have the error contributions of all class pairs $\hat{e}_{jk}$ and can estimate the error of any ordering $\pi^{(i)}$ using Eq. (4) without actually running Ripper with that ordering $\pi^{(i)}$. Not only that, we can also search all possible class orderings to get the best ordering exhaustively

$$\pi^{\text{best}} = \arg\min_{\pi^{(i)}} \hat{E}_{\pi^{(i)}} \tag{8}$$

The pseudocode of PEA is given in Fig. 6. The algorithm tries to estimate the expected error contribution of each pair of classes. There are $K(K-1)$ e_{jk} pairs for a dataset with K classes. It initially generates T random orderings (line 4). Ripper is trained with these T orderings via 10-fold cross-validation (line 5). We construct the quadratic program QP using the test errors of these T orderings (line 6). After solving QP (line 7), the best ordering is obtained by searching over all possible orderings exhaustively (lines 8–15).

The time complexity of PEA is $\mathcal{O}(TCdN \log N) + \mathcal{O}(Q) + \mathcal{O}(K!N^2)$, where

- the time complexity of training T different Ripper's with T different orderings is $\mathcal{O}(TCdN \log N)$ (lines 3–5),
- the time complexity of solving quadratic program QP is $\mathcal{O}(Q)$ (line 7),
- the time complexity of searching the best ordering over all possible orderings is $\mathcal{O}(K!N^2)$ (lines 8–15).

Table 1
Details of the datasets. d: number of attributes, K: number of classes, N: sample size.

Dataset	d	C	N
balance	4	3	625
car	6	4	1728
cmc	9	3	1473
dermatology	34	6	366
ecoli	8	7	336
flags	26	8	194
glass	9	6	214
hayesroth	4	3	160
iris	4	3	150
leukemia1	5327	3	72
leukemia2	11 225	3	72
nursery	8	5	12 960
ocr	256	10	600
optdigits	64	10	3823
pendigits	16	10	7494
segment	19	7	2310
shuttle	9	7	58 000
splice	60	3	3175
srbct	2308	4	83
tae	5	3	151
vehicle	18	4	846
wave	21	3	5000
wine	13	3	178
winequality	11	7	6497
yeast	8	10	1484
zipcodes	256	10	7291
zoo	16	7	101

If the number of classes in a dataset is large ($K > 15$), searching the best ordering exhaustively (lines 8–15) becomes unfeasible. In those cases, one can resort to any heuristic algorithm to solve the exhaustive search problem, which is actually a traveling salesman problem with K cities. Note that, when we use any heuristic algorithm, the ordering found may not be the best ordering.

4. Experiments

4.1. Setup

We did our experiments on 24 datasets taken from UCI repository [5]. We also used 3 bioinformatics datasets (*leukemia1*, *leukemia2*, *srbct*) which contain cancer-related gene expression data [21]. The details of the datasets are given in Table 1. We use 10-fold cross-validation to generate training and test sets and we repeat experiments 10 times with different seeds to avoid the randomness factor. For simple datasets, where the number of classes is small ($K \leq 5$), we did an exhaustive search over all possible $K!$ orderings. For complex datasets, where the number of classes is large ($K > 5$), we take $T = 50$, that is, we run Ripper algorithm with 50 random orderings $\pi^{(i)}$.

4.2. Primary assumption

In the first part of our experiments, we tested our primary assumption, that is, the expected error of the Ripper algorithm trained with any ordering, is the sum of $K(K-1)/2$ pairwise expected errors of classes. Actually, this assumption can be rephrased in another way, the pairwise expected errors of classes (e_{jk}) are nearly equal in all possible class orderings.

To validate our assumption, we calculate the actual e_{jk}'s for each ordering in each experiment using the confusion matrices of the test results. Table 2 shows the average contribution of pairwise errors (e_{jk}) to the average test error for each ordering on 3-class datasets. Each cell contains the average of $10 \times 10 = 100$ e_{jk}'s. In order to test for equality, for each e_{jk} and for each experiment, we did 10-fold paired-t-test between each pairs of ordering. The null hypothesis for

Table 2
Average contribution of pairwise errors on selected 3-class datasets.

Set	π	e_{12}	e_{13}	e_{21}	e_{23}	e_{31}	e_{32}
balance	321			8.7		13.3	12.1
	312	3.4				4.5	20.3
	231			13.9	11.7	8.5	
	132	3.4	4.5				20.3
	213		3.4	4.5	20.9		
	123	4.4	3.5		21.5		
cmc	321			14.5		26.0	7.6
	312	19.1				11.0	22.7
	231			14.8	7.2	26.0	
	132	19.1	10.5				22.9
	213		23.8	5.2	16.9		
	123	4.2	25.3		18.0		
iris	321			0.7		0.0	5.7
	312	0.1				0.0	5.8
	231			2.5	4.9	1.0	
	132	0.1	0.0				5.5
	213		0.3	0.4	6.5		
	123	0.1	0.0		5.6		
splice	321			2.6		1.7	1.7
	312	2.4				1.8	1.8
	231			4.3	3.1	1.6	
	132	6.5	2.7				2.2
	213		2.6	3.9	3.3		
	123	3.3	5.6		4.8		
tae	321			28.3		28.4	5.2
	312	29.5				4.1	30.4
	231			28.4	5.0	28.4	
	132	29.6	4.3				30.9
	213		33.2	0.7	32.3		
	123	0.6	33.1		32.4		
wave	321			7.1		8.7	7.0
	312	6.9				6.4	9.4
	231			6.7	7.4	8.7	
	132	6.6	6.9				9.3
	213		8.5	5.0	9.3		
	123	5.0	8.4		9.2		
wine	321			4.3		1.0	2.9
	312	4.5				0.3	2.9
	231			7.6	2.4	1.1	
	132	5.1	1.0				5.2
	213		0.7	4.3	5.5		
	123	3.8	2.0		3.5		

the statistical test is:

$$H_0(e_{jk}, \pi^{(u)}, \pi^{(v)}) : e_{jk} \text{ in } \pi^{(u)} = e_{jk} \text{ in } \pi^{(v)} \tag{9}$$

For example, on 3-class datasets, there are 6 possible e_{jk}'s, and for each e_{jk}, there are 3 possible orderings to choose from and we repeat the experiments 10 times, so there are a total of $6 \times 3 \times 10 = 180$ pairwise tests.

For 3-class datasets, on *iris*, *leukemia1*, *leukemia2*, and *wine*, the assumption is almost correct, if we look the values in each column on those datasets, the pairwise errors do not change much from one ordering to another. Pairwise statistical tests also agree with these results, namely, in 97, 88, 95, and 94% of the cases, the null hypothesis is accepted. On four other datasets, the assumption is partially correct, on *balance*, *hayesroth*, and *splice*, in 53, 52, 50% of the cases, the null hypothesis is accepted. Lastly, three datasets, *cmc*, *tae*, and *wave* do loosely satisfy the assumption; only 29, 32, and 33% of the cases, the null hypothesis is accepted.

For 4-class datasets, the situation is similar. On *car* dataset, 66% , on *srbct* dataset, 94%, and on *vehicle* dataset, 33% of the statistical tests between pairwise errors are accepted. We can conclude that although there are exceptional cases, in general, our assumption about pairwise errors is correct.

Table 3
Average contribution of pairwise errors and their estimated values on 4-class datasets.

Dataset	e_{12} $\hat{e}_{12}$	e_{13} $\hat{e}_{13}$	e_{14} $\hat{e}_{14}$	e_{21} $\hat{e}_{21}$	e_{23} $\hat{e}_{23}$	e_{24} $\hat{e}_{24}$	e_{31} $\hat{e}_{31}$	e_{32} $\hat{e}_{32}$	e_{34} $\hat{e}_{34}$	e_{41} $\hat{e}_{41}$	e_{42} $\hat{e}_{42}$	e_{43} $\hat{e}_{43}$
car	1.30	0.16	0.22	9.12	2.46	3.53	0.93	1.83	0.92	1.14	2.12	1.05
	2.33	0.50	0.03	13.22	1.34	0.96	1.53	2.46	0.00	0.74	0.59	0.56
srbct	7.00	2.44	1.85	3.38	2.23	3.12	2.17	1.45	0.87	3.76	2.94	1.29
	6.71	3.32	1.59	2.00	2.65	2.86	3.44	0.45	1.77	2.26	4.03	1.90
vehicle	3.25	2.44	3.11	6.37	5.79	15.23	1.84	1.90	1.52	6.07	15.28	5.71
	10.39	17.20	0.58	12.35	3.65	0.05	17.38	1.45	0.48	1.61	0.58	3.44

Table 4
Average contribution of pairwise errors and their estimated values on 3-class datasets.

Dataset	e_{12} $\hat{e}_{12}$	e_{13} $\hat{e}_{13}$	e_{21} $\hat{e}_{21}$	e_{23} $\hat{e}_{23}$	e_{31} $\hat{e}_{31}$	e_{32} $\hat{e}_{32}$
balance	3.71	3.79	9.03	18.04	8.77	17.54
	10.82	13.49	14.86	1.97	16.08	2.72
cmc	14.14	19.88	11.48	14.06	21.00	17.74
	25.96	17.30	22.85	5.74	18.41	8.24
hayesroth	14.75	11.36	13.50	11.54	0.00	0.00
	2.76	20.04	1.78	13.93	9.53	2.87
iris	0.13	0.09	1.21	5.64	0.35	5.66
	0.87	1.95	1.80	3.32	2.72	2.29
leukemia1	3.56	6.15	2.53	2.38	8.40	5.34
	3.64	3.11	2.50	3.93	5.26	10.36
leukemia2	4.39	9.41	2.29	3.98	6.25	2.32
	3.22	8.00	2.92	7.33	2.69	4.82
splice	4.06	3.63	3.58	3.72	1.68	1.88
	3.53	6.30	1.64	3.36	3.07	0.51
tae	19.90	23.54	19.12	23.24	20.29	22.16
	22.85	34.85	21.91	8.83	32.04	8.04
wave	6.14	7.92	6.29	8.62	7.92	8.54
	6.66	6.93	6.98	8.95	6.82	9.10
wine	4.43	1.20	5.38	3.81	0.79	3.67
	1.68	5.53	2.65	2.70	3.94	2.37

4.3. Accuracy of pairwise-error estimation

In the second part of our experiments, we check if our estimations of pairwise errors ($\hat{e}_{jk}$) and their true values (e_{jk}) match. Tables 3 and 4 show the average contribution of pairwise errors (e_{jk}) to the average test error, and their averaged estimated values ($\hat{e}_{jk}$) found using quadratic programming on 4 and 3 class datasets respectively.

For 3-class datasets, on *iris*, *leukemia1*, *leukemia2*, *splice*, *wave*, and *wine*, the estimated and the true values of pairwise errors are close to each other. On the other hand, on *balance*, *cmc*, *hayesroth*, and *tae*, the estimated values are way off the true values. For 4-class datasets, on *car* and *srbct*, the estimation works well, whereas on *vehicle* at least 5 of the estimated values do not match the true values.

When we combine these results with the previous results, we can say that, if the primary assumption is correct, our algorithm estimates e_{jk}'s well. If the primary assumption is loosely correct, there is a high chance that, the algorithm will fail to estimate e_{jk}'s. But note that, even if we fail to estimate e_{jk}'s, our algorithm may correctly estimate the expected error of orderings $\pi^{(i)}$.

4.4. Accuracy of test error estimation

In the third part of our experiments, we compare the performance of Ripper trained with our estimated best ordering (π^{best}), found by using algorithm PEA in Fig. 6, with the performance of Ripper trained with heuristic ordering ($\pi^{\text{heuristic}}$).

For simple datasets, we have the test errors of all possible orderings, therefore we can position π^{best} according to $\pi^{\text{heuristic}}$. For each dataset, we assign ranks to all $K!$ orderings so that the best gets the rank of 1, the second gets the rank of 2, and so on. We then calculate average ranks of both π^{best} and $\pi^{\text{heuristic}}$ over 10 experiments. Table 5 shows those average ranks with respect to all $K!$ orderings. We also compare the rank performances using Wilcoxon signed-rank test and show statistically significant differences in boldface.

Table 5
Average and standard deviation of the ranks of the Ripper algorithm trained with the heuristic ordering and the best ordering found by quadratic programming (QP) with respect to all $K!$ orderings. Statistically significant differences are show in boldface.

Dataset	Heuristic	QP
balance	3.3 ± 1.3	$\mathbf{2.1 \pm 1.4}$
cmc	3.1 ± 0.7	$\mathbf{1.2 \pm 0.4}$
hayesroth	1.3 ± 0.7	1.5 ± 0.7
iris	2.0 ± 1.0	2.1 ± 1.1
leukemia1	1.5 ± 0.7	1.6 ± 0.8
leukemia2	1.9 ± 1.1	$\mathbf{1.2 \pm 0.4}$
splice	1.5 ± 0.5	1.5 ± 0.5
tae	4.9 ± 0.6	$\mathbf{1.2 \pm 0.4}$
wave	3.7 ± 1.9	$\mathbf{1.0 \pm 0.0}$
wine	2.1 ± 1.3	2.2 ± 1.4
car	18.2 ± 3.8	$\mathbf{2.8 \pm 2.7}$
srbct	6.8 ± 5.2	5.0 ± 4.3
vehicle	21.6 ± 2.6	$\mathbf{4.9 \pm 3.0}$
nursery	27.3 ± 10.4	25.1 ± 11.7

First of all, although the heuristic work well for some datasets such as *hayesroth*, *leukemia1*, and *splice*, as the number of classes increases, the performance of $\pi^{\text{heuristic}}$ decreases, which supports our motivation. For example, on *balance*, *cmc*, *tae*, and *wave*, the number of possible orderings is 6 and the rank of the heuristic ordering is larger than 3, worse than the rank of a random ordering. When the number of classes increases, on *car* and *vehicle* datasets, the number of possible orderings is 24 and the rank of the heuristic ordering is near 20, which is significantly worse than a random ordering, which will have a rank of 12.

Second, since our algorithm is informed about all performances of all orderings, as expected, the performance of π^{best} is better than $\pi^{\text{heuristic}}$. In 7 out of 14 datasets, Ripper algorithm trained with π^{best} is assigned statistically significantly better ranks than the Ripper algorithm trained with $\pi^{\text{heuristic}}$.

For complex datasets, we take $T = 50$, that is, we run Ripper algorithm with 50 random orderings $\pi^{(i)}$. In this case, since we cannot know the rank performance of Ripper trained with orderings π^{best} or $\pi^{\text{heuristic}}$ ($K!$ is very large), we compare the test errors of the algorithms trained with those orderings. Table 6 shows those average test errors. We also compare the expected errors using paired-t test and show statistically significant differences in boldface.

In this case, although PEA is only informed about a small sample of performances of all orderings, it finds significantly better orderings than the heuristic ordering. In 7 datasets out of 13 datasets, Ripper algorithm trained with π^{best} is statistically significantly better than

Table 6
Average and standard deviation of test errors of the Ripper algorithm trained with the heuristic ordering and the best ordering found by quadratic programming. Statistically significant differences are show in boldface.

Dataset	Heuristic	QP
dermatology	7.75 ± 0.88	$\mathbf{3.71 \pm 0.66}$
glass	36.18 ± 2.02	36.20 ± 5.25
segment	6.54 ± 0.34	$\mathbf{4.89 \pm 0.41}$
shuttle	0.04 ± 0.01	$\mathbf{0.02 \pm 0.01}$
winequality	46.32 ± 0.38	50.45 ± 3.25
zoo	12.37 ± 1.47	$\mathbf{9.80 \pm 1.25}$
ecoli	19.41 ± 0.59	20.08 ± 2.01
flags	39.13 ± 0.90	38.30 ± 2.94
ocr	26.61 ± 1.38	26.35 ± 1.38
optdigits	11.08 ± 0.47	$\mathbf{8.98 \pm 0.45}$
pendigits	5.30 ± 0.20	$\mathbf{4.90 \pm 0.11}$
yeast	$\mathbf{43.10 \pm 0.74}$	47.50 ± 3.89
zipcodes	15.27 ± 0.32	$\mathbf{13.95 \pm 0.36}$

the Ripper algorithm trained with $\pi^{\text{heuristic}}$, where the reverse occur only on one dataset.

Another important point is, on equally distributed datasets, where the number of instances in each class is the same, heuristic ordering is the same as the random ordering, and in those cases (*segment*, *optdigits*, *pendigits*), PEA is significantly superior compared to the heuristic ordering.

5. Conclusion

Current heuristic approach used in Ripper that orders the classes in a dataset according to their sample sizes, usually does not give the most accurate classification. In this paper, we propose a novel algorithm to improve this heuristic. Our proposed algorithm PEA, although not guarantees to find a better ordering, is usually better than Ripper proper and can be improved by including more random orderings in the optimization.

This study is an important step for understanding the impact of the training ordering of classes on the performance and thus can be extended via taking each ordering as a classifier and get better classifiers by producing intelligent ensembles of these orderings. The ensemble idea is based on weak classifiers, which are slightly better than random classifiers. Using PEA, we not only can estimate best rule classifiers based on best orderings they are trained upon, but also estimate "worse" classifiers based on worse orderings they are trained upon. Combining those "worse" classifiers may lead us to better ensembles.

References

[1] S. Ata, O.T. Yıldız, Searching for the optimal ordering of classes in rule induction, in: Proceedings of the International Conference on Pattern Recognition, Tsukuba, Japan, 2012, pp. 1277–1280.
[2] K. Bennett, J. Blue, A support vector machine approach to decision trees, in: Proceedings of the International Joint Conference on Neural Networks, IEEE, Anchorage, Alaska, 1998, pp. 2396–2401.
[3] M. Berthold, D.J. Hand, Intelligent Data Analysis: An Introduction, Springer-Verlag, 2003.
[4] R. Bhatt, M. Gopal, Frct: fuzzy-rough classification trees, Pattern Anal. Appl. 11 (2008) 73–88.
[5] C. Blake, C. Merz, UCI repository of machine learning databases, 2000, http://www.ics.uci.edu/~mlearn/MLRepository.html.
[6] L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone, Classification and Regression Trees, John Wiley and Sons, 1984.
[7] M. Chisholm, P. Tadepalli, Learning decision rules by randomized iterative local search, in: Proceedings of the 19th International Conference on Machine Learning, 2002, pp. 75–82.
[8] W.W. Cohen, Fast effective rule induction, in: The Twelfth International Conference on Machine Learning, 1995, pp. 115–123.
[9] J. Fürnkranz, Separate-and-conquer learning, Artif. Intell. Rev. 13 (1999) 3–54.
[10] J. Gama, Discriminant trees, in: 16th International Conference on Machine Learning, Morgan Kaufmann, New Brunswick, New Jersey, 1999, pp. 134–142.
[11] H. Guo, S.B. Gelfand, Classification trees with neural network feature extraction, IEEE Trans. Neural Networks 3 (1992) 923–933.
[12] B. Hammer, A. Rechtien, M. Strickert, T. Villmann, Rule extraction from self-organizing networks, in: ICANN, 2002, pp. 877–883.
[13] L.A. Kurgan, K.J. Cios, S. Dick, Highly scalable and robust rule learner: performance evaluation and comparison, IEEE Trans. Syst., Man, Cybern. Part B: Cybern. 36 (2006) 32–53.
[14] W.Y. Loh, Y.S. Shih, Split selection methods for classification trees, Stat. Sinica 7 (1997) 815–840.
[15] D. Martens, M.D. Backer, J. Vanthienen, M. Snoeck, B. Baesens, Classification with ant colony optimization, IEEE Trans. Evol. Comput. 11 (2007) 651–665.
[16] S.H. Muggleton, Inverse entailment and PROGOL, New Gener. Comput. 13 (1995) 245–286.
[17] S.K. Murthy, S. Kasif, S. Salzberg, A system for induction of oblique decision trees, J. Artif. Intell. Res. 2 (1994) 1–32.
[18] J.L. Olmo, J.R. Romero, S. Ventura, Using ant programming guided by grammar for building rule-based classifiers, IEEE Trans. Syst., Man, Cybern. Part B: Cybern. 41 (2011) 1585–1599.
[19] J.R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publisher, San Meteo, CA, 1993.
[20] Q. Shen, A. Chouchoulas, A rough-fuzzy approach for generating classification rules pattern recognition, Pattern Recognit. 35 (2002) 2425–2438.
[21] A. Statnikov, C. Aliferis, I. Tsamardinos, D. Hardin, S. Levy, A comprehensive evaluation of multicategory classification methods for microarray gene expression cancer diagnosis, Bioinformatics 21 (2005) 631–643.
[22] H. Theron, I. Cloete, BEXA: a covering algorithm for learning propositional concept descriptions, Mach. Learn. 24 (1996) 5–40.
[23] R. Tibshirani, T. Hastie, Margin trees for high-dimensional classification, J. Mach. Learn. Res. 8 (2007) 637–652.
[24] G. Venturini, SIA: a supervised inductive algorithm with genetic search for learning attributes based concepts, in: Proceedings of the 6th European Conference on Machine Learning, Vienna, Austria, 1993, pp. 280–296.
[25] Y.-C. Hu, R.-S. Chen, G.-H. Tzeng, Finding fuzzy classification rules using data mining techniques, Pattern Recognit. Lett. 24 (2003) 509–519.
[26] O.T. Yıldız, E. Alpaydın, Linear discriminant trees, in: 17th International Conference on Machine Learning, Morgan Kaufmann, 2000, pp. 1175–1182.