On the Feature Extraction in Discrete Space

Olcay Taner Yıldız

Department of Computer Engineering, Işık University, 34980, İstanbul Turkey
Tel: +90 (216) 5287157, Fax: +90 (216) 7102872

Abstract

In many pattern recognition applications, feature space expansion is a key step for improving the performance of the classifier. In this paper, we (i) expand the discrete feature space by generating all orderings of values of k discrete attributes exhaustively, (ii) modify the well-known decision tree and rule induction classifiers (ID3 [1] and Ripper [2]) using these orderings as the new attributes. Our simulation results on 15 datasets from UCI repository [3] show that the novel classifiers performs better than the proper ones in terms of error rate and complexity.

Keywords: Feature Extraction, Discrete Space, Decision Tree Induction, Rule Induction

1. Introduction

In pattern recognition the knowledge is extracted as patterns from a training sample for future prediction. Most pattern recognition algorithms such as neural networks [4] or support vector machines [5] make accurate predictions but are not interpretable, on the other hand decision trees or rule inducers are simple and easily comprehensible. They are robust to noisy data and can learn disjunctive expressions. Surveys of work on constructing and simplifying decision trees can be found in [6] and [7]. [8] is an old but an extended review of rule induction algorithms.

Email address: olcaytaner@isikun.edu.tr (Olcay Taner Yıldız)

Decision trees are tree-based structures where each internal node implements a decision function, $f_m(\boldsymbol{x})$, each branch of an internal node corresponds to one outcome of the decision, and each leaf corresponds to a class. In a univariate decision tree [9], the decision at internal node m uses only one attribute, i.e., one dimension of $\boldsymbol{x}$, x_j. If that attribute is discrete, there will be L children (branches) of each internal node corresponding to the L different outcomes of the decision. ID3 is one of the best known univariate decision tree algorithm with discrete features [1].

One of the drawbacks of the L-ary splits is that the training examples are separated into small subsets, which in turn gives us a poor predictor for the unseen test instances. One can convert discrete features having $L > 2$ different values to L binary features using 1-of-L encoding, this will result in a larger tree than the former. Although there are alternative approaches to handle the selection bias that favors the attributes having many values over those with few values [1], [10], those approaches can not help if the attributes have nearly similar number of values.

Rulesets are list-based structures where each rule in a ruleset is defined for a class and composed of a number of conditions, where each condition implements a decision function, $f_c(\boldsymbol{x})$. In a univariate ruleset [2], like the univariate decision tree, the decision at internal condition c uses only one attribute. If that attribute is discrete, the decision function is in the form $x_i = v_{ij}$, where i is the selected attribute and v_{ij} is the j'th possible value of the attribute x_i.

Another drawback of using discrete attributes in the form of $x_i = v_{ij}$ is that there is only a single possible split for each discrete attribute. On the other hand, there are $n - 1$ different possible splits for a continuous attribute, where n represents the number of distinct values of that continuous attribute. For example, the decision tree in Figure 1 gives an inefficient representation of a concept. While the instances x_1 = hot are described efficiently, there are two identical subtrees those separating class C_2 from class C_1. In general, conjunctions can be described efficiently by decision trees while disjunctions require a large tree to describe [11], [12]. Replication problem can also occur

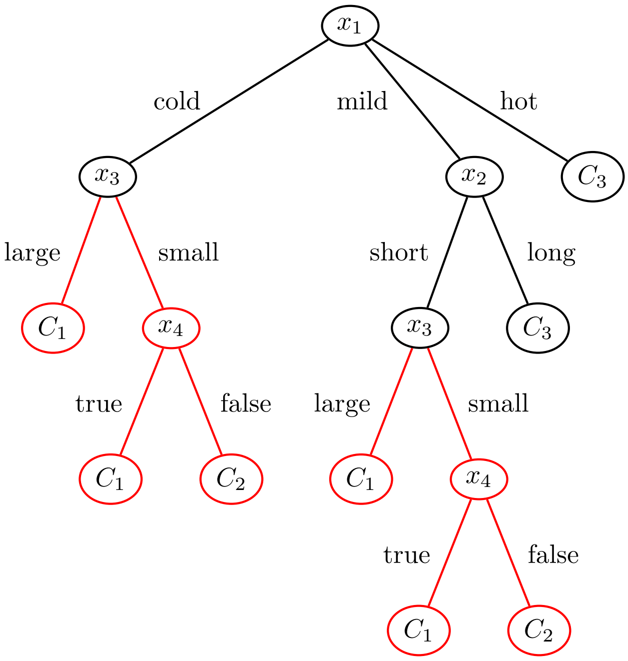


Figure 1: Univariate decision tree corresponding to the same concept represented by the K-tree in Figure 3.

when the data contains attributes with high arity values i.e., attributes with large number of possible values. If a tree has high arity attributes (say L $\geq$ 5) then it will quickly fragment the data in that node into small partitions.

To increase the number of distinct splits for datasets with discrete attributes, we define k-ordering, where for each subset (size k) of the attributes, we combine them by generating all possible orderings of the values of those attributes exhaustively. Then we apply the usual ID3 and Ripper algorithms using these orderings as the new attributes. In this way, the number of distinct possible splits for each extracted feature will be $n_1 \times n_2 \times \ldots \times n_k$, where n_i represents the number of distinct values of the selected feature i.

In the earlier version of this work [13], we proposed feature extraction from discrete space and its application to univariate tree induction; this present paper revisits the feature extraction, introduces its application to rule induction, proposes omnivariate versions, and makes a more through comparison with the original inducers ID3 and Ripper.

This paper is organized as follows: We give the definition of k-ordering in Section 2, its application to univariate decision tree and rule induction in Sections 3 and 4 respectively. We discuss omnivariate version of our algorithms in Section 5. We give experimental results in Section 6, and conclude in Section 7.

2. *k*-ordering

2.1. Definition

Let a_1, a_2, ..., a_d be d discrete attributes of a dataset D. Each attribute a_i can have n_i distinct values which can be represented as v_{i1}, v_{i2}, ..., v_{in_i}.

Definition 1. *For each k attributes a_{s_1}, a_{s_2}, ..., a_{s_k} from a d dimensional dataset D, k-ordering is defined as the permutation list (p_1, p_2, ..., p_k), where p_i is a permutation of values of the feature a_{s_i} (a permutation of v_{s_i1}, v_{s_i2}, ..., $v_{s_in_{s_i}}$).*

For example, ((red, blue, green), (yes, no), (large, extralarge, small, medium)) is a 3-ordering, where selected $k = 3$ features have three, two and four distinct values respectively. Note that we can use both categorical discrete attributes (such as red, green, blue) as well as nominal discrete attributes (such as cold, neutral, warm, very warm) to define the k-ordering. There are

$$\sum_{s_1, s_2, \ldots, s_k \in 1, 2, \ldots, d} n_{s_1}! n_{s_2}! \ldots n_{s_k}!$$

distinct k-orderings of a d dimensional dataset. As an example, given a dataset with two dimensions having values (red, green, blue) and (yes, no), there are 8 (3! + 2!) and 24 (3! 2! + 2! 3!) distinct 1 and 2-orderings respectively.

2.2. Ordering Relations based on k-ordering

Continuing the definition of k-ordering, we can produce relational operators that compare two instances. Formally, given a k-ordering of a dataset D and two instances (x_1, x_2, ..., x_d) and (y_1, y_2, ..., y_d) from this dataset;

Definition 2. *$(x_{s_1}, x_{s_2}, \ldots, x_{s_k}) \prec (y_{s_1}, y_{s_2}, \ldots, y_{s_k})$ if and only if $x_{s_i} = y_{s_i}$ for $i = 1, \ldots, t \geq 0$, and $x_{s_{t+1}}$ comes before $y_{s_{t+1}}$ in permutation p_{t+1}.*

Definition 3. *$(x_{s_1}, x_{s_2}, \ldots, x_{s_k}) \succ (y_{s_1}, y_{s_2}, \ldots, y_{s_k})$ if and only if $x_{s_i} = y_{s_i}$ for $i = 1, \ldots, t \geq 0$, and $x_{s_{t+1}}$ comes after $y_{s_{t+1}}$ in permutation p_{t+1}.*

Definition 4. *$(x_{s_1}, x_{s_2}, \ldots, x_{s_k}) = (y_{s_1}, y_{s_2}, \ldots, y_{s_k})$ if and only if $x_{s_i} = y_{s_i}$ for all $i = 1, \ldots, k$.*

For example, given the 2-ordering ((red, blue, green), (no, yes)), all possible values of the instances can be sorted as (red, no) $\prec$ (red, yes) $\prec$ (blue, no) $\prec$ (blue, yes) $\prec$ (green, no) $\prec$ (green, yes).

2.3. Splits based on k-ordering

If the instances of a dataset can be sorted based on a k-ordering, we can also list all possible splits based on that k-ordering. For example, given the 2-ordering ((red, blue, green), (no, yes)), all possible splits are: $\mathbf{x} \preceq$ (red, no), $\mathbf{x} \succ$ (red, no), $\mathbf{x} \preceq$ (red, yes), $\mathbf{x} \succ$ (red, yes), $\mathbf{x} \preceq$ (blue, no), $\mathbf{x} \succ$ (blue, no), $\mathbf{x} \preceq$ (blue, yes), $\mathbf{x} \succ$ (blue, yes), $\mathbf{x} \preceq$ (green, no), $\mathbf{x} \succ$ (green, no).

More formally, since each k-ordering defines a new attribute

$$\sum_{s_1, s_2, \ldots, s_k \in 1,2,\ldots,d} n_{s_1}! n_{s_2}! \ldots n_{s_k}!$$

new attributes are generated. For each new attribute, we can use one of the $2 \times n_{s_1} \times n_{s_2} \times \ldots \times n_{s_k} - 2$ distinct splits in a tree node or in a condition of a rule.

The pseudocode for finding the exhaustive set of k-ordered splits is shown in Figure 2. First we initialize the result set S (Line 1). For each k-permutation of the attributes (Line 2) we extract all possible k-orderings by traversing the attributes iteratively (Lines 3-5). Given an ordering r (Line 6), we also generate all possible split points for that ordering (Line 7). Each possible k-ordering and split point sp is added to S (Line 8).

Set k-OrderingSplitSet(k)

$S = \{\}$

for each attribute permutation $(s_1, \ldots, s_k)$

  for each permutation p_1 of values of a_{s_1}

    $\ldots$

    for each permutation p_k of values of a_{s_k}

      $r = (p_1, \ldots, p_k)$

      for each split point $sp = (t_1, \ldots, t_k)$ where $t_1 \in p_1, \ldots, t_k \in p_k$

        $S = S \cup \{(r, sp)\}$

return S

Figure 2: The pseudocode of the exhaustive k-ordered split search algorithm: k: parameter in the k-ordering

3. Application to decision tree induction

In this section, we apply k-ordering to find k-ordered splits in the univariate decision tree algorithm ID3 [1]. The idea is as follows: At each decision node, we generate all possible k-orderings and split points using the algorithm in Figure 2. For all attributes and for all split points of those attributes, we calculate impurity and choose the split point and k-ordering that has the minimum entropy.

Figure 3 shows an example k-tree with two decision and three leaf nodes. In the root node, the best ordering and the best split for that ordering are ((cold, mild, hot), (short, long)), $\mathbf{x} \preceq$ (mild, short) respectively. In this case, the decision tree shown in Figure 3 gives more efficient representation of the same concept that is inefficiently represented by Figure 1. This way, K-trees can effectively solve the replication problem of trees.

Figure 4 shows the pseudocode that finds the impurity of a k-ordered split for a given k-ordering r. First we initialize the counts of the left and right branches (Lines 2-3). For each instance $\mathbf{x}$ in the instance list $\mathcal{X}$, we compare its

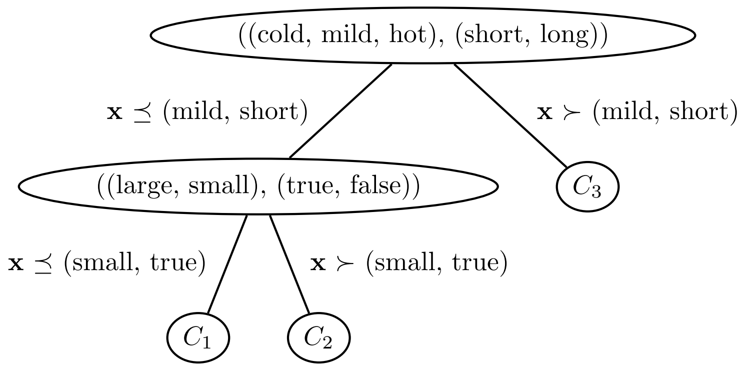


Figure 3: An example k-tree ($k = 2$).

Impurity((sp, r), $\mathcal{X}$)
1 **for** i = 1 **to** K
2 $\quad N_i^L = 0$
3 $\quad N_i^R = 0$
4 **for** i = 1 **to** $\mathcal{X}$.size
5 $\quad \mathbf{x} = \mathcal{X}$[i]
6 $\quad j$ = class of $\mathbf{x}$
7 $\quad$ **if** $\mathbf{x}$ satisfies sp according to ordering r
8 $\qquad N_j^L = N_j^L + 1$
9 $\quad$ **else**
10 $\qquad N_j^R = N_j^R + 1$
11 **return** entropy calculated from N_i^L and N_i^R's.

Figure 4: The pseudocode of the algorithm that finds the impurity of a k-ordered split sp of a k-ordering r for an instance list $\mathcal{X}$.

attribute values with the split point sp according to the current k-ordering (Line 7). If $\mathbf{x}$ satisfies the split sp according to the k-ordering r, we update counts of the left branch (Line 8); otherwise we update counts of the right branch (Line 10). Using the class counts of the left and right branches, we can calculate the impurity using entropy [4], Gini index [14], or any other metric (Line 11).

	C_1	C_2
(long, no)	5	2
(long, yes)	3	4
(short, no)	2	5
(short, yes)	10	4

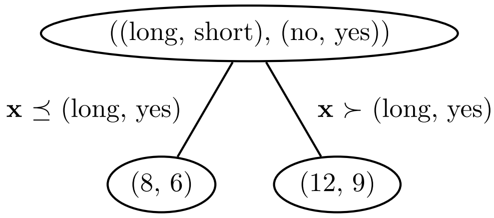


Figure 5: An example case for calculating the impurity of a 2-ordered split. Given the 2-ordering ((long, short), (no, yes)), the impurity of the split $\mathbf{x} \preceq$ (long, yes) will be calculated from the counts of the left branch ($N_1^L = 8$, $N_2^L = 6$) and right branch ($N_1^R = 12$, $N_2^R = 9$).

As an example, given the 2-ordering ((long, short), (no, yes)), all possible split points will be $\mathbf{x} \preceq$ (long, no), $\mathbf{x} \preceq$ (long, yes), $\mathbf{x} \preceq$ (short, no). Let say there are 5, 3, 2, 10 instances of class C_1 and 2, 4, 5, 4 instances of class C_2 having those values respectively. Then the split $\mathbf{x} \preceq$ (long, yes) will divide the instances into two where the left branch will have $5 + 3 = 8$, $2 + 4 = 6$ instances from classes C_1 and C_2 respectively and the right branch will have $2 + 10 = 12$, $5 + 4 = 9$ instances from classes C_1 and C_2 respectively (See Figure 5).

4. Application to rule induction

Similar to decision tree case, we can also apply k-ordering to find k-ordered splits in the univariate rule induction algorithm Ripper [2]. The idea is as follows: At each condition of a rule, we generate all possible k-orderings and split points using the algorithm in Figure 2. For all attributes and for all split points of those attributes, we calculate the information gain and choose the split point and k-ordering that has the maximum information gain.

IF $\mathbf{x} \succ$ (mild, short) THEN Class $= C_3$
ELSE IF $\mathbf{x} \succ$ (small, true) THEN Class $= C_2$
ELSE Class $= C_1$

Figure 6: An example k-ruleset ($k = 2$).

Figure 6 shows an example k-rule with two decision rules and two decision conditions. In the first condition, the best ordering and the best split for that ordering are ((cold, mild, hot), (short, long)), $\mathbf{x} \succ$ (mild, short) respectively.

```
InformationGain((sp, r), X)
N_c^Pos = N_c^Neg = 0
N_u^Pos = N_u^Neg = 0
for i = 1 to X.size
  x = X[i]
  j = class of x
  if x satisfies sp according to ordering r
    if j is positive class
      N_c^Pos = N_c^Pos + 1
    else
      N_c^Neg = N_c^Neg + 1
  else
    if j is positive class
      N_u^Pos = N_u^Pos + 1
    else
      N_u^Neg = N_u^Neg + 1
return information gain calculated from N_c and N_u's.
```

Figure 7: The pseudocode of the algorithm that finds the information gain of a k-ordered split sp of a k-ordering r for an instance list $\mathcal{X}$.

Figure 7 shows the pseudocode that finds the information gain of a k-ordered split for a given k-ordering r. In the separate and conquer approach, which is the main approach in rule induction, one learns rules for a single class by separating its instances from the instances of other classes. For that reason, there are always two classes in rule induction: positive class, whose instances are tried to be covered, and negative class which is composed of other classes except the positive class.

First we initialize the counts of the covered and uncovered positive and negative classes (Lines 1-2). For each instance $\mathbf{x}$ in the instance list $\mathcal{X}$, we compare its attribute values with the split point sp according to the current k-ordering (Line 6). If $\mathbf{x}$ satisfies the split sp according to the k-ordering r, it is covered and we update positive (negative) counts of the covered if $\mathbf{x}$ is from positive (negative) class (Lines 7-10). If $\mathbf{x}$ is not covered, we update positive (negative) counts of the uncovered if $\mathbf{x}$ is from positive (negative) class (Lines 12-15). Using positive and negative class counts of the covered and uncovered groups, we can calculate the information gain (Line 16).

5. Omnivariate induction

The omnivariate idea was first used in [15], which can be summarized as follows: Decision trees are model augmenting structures and each node m tries to discriminate two class groups using a decision model $f_m(\boldsymbol{x})$. Using a decision tree with the same type of model at each node, one assumes the same internal data complexity at all the decision nodes. Omnivariate tree chooses the optimal model (instead of the same model everywhere) for each decision node depending on the internal complexity of the data arriving at that node. In the omnivariate decision tree [15], at each node, three models; univariate, linear multivariate, and nonlinear multivariate are trained and the optimal model is chosen according to a statistical test.

In our case, by using the k-ordering with the same k everywhere, we just assume the same bias at each decision node in the decision trees or decision condition in the rulesets. So, borrowing the omnivariate idea, we can find the best ordering and the best split for each k, and then choose the model that has the minimum entropy in decision trees or maximum information gain in rulesets.

Figure 8 shows the pseudocode of the omnivariate split search algorithm. We combine the k-orderings and split points produced by the algorithm k-OrderingSplitSet for $i = 1, \ldots, k$ (Line 3).

Note that, the set of splits produced by a k-ordering is always a subset of

```
Set OmnivariateOrderingSplitSet(k)
S = {}
for i = 1 to k
    S = S ∪ k-OrderingSplitSet(k)
return S
```

Figure 8: The pseudocode of the omnivariate split search algorithm

the set of splits produced by a $k+1$-ordering. Therefore, the minimum entropy (maximum information gain) that can be obtained using a k-ordering can never be better than a $k+1$-ordering. But there are cases where $k+1$-ordering produce the same splits as k-ordering, and in those cases choosing k-ordering corresponds to choosing the best model both in terms of error and complexity.

6. Experiments

We use a total of 15 data sets where 13 of them are from UCI repository [3] and 2 are (*acceptors* and *donors*) bioinformatics datasets (see Table 1). Since the time complexity of our proposed algorithms change exponentially with the number of distinct values of an attribute (n_i), we run the algorithms for the datasets with $n_i \leq 5$.

Our methodology in generating train, validation and test sets is as follows: A data set is first divided into two parts, with 1/3 as the test set, *test*, and 2/3 as the training set. The training set is resampled using 2×5 cross-validation to generate ten training and validation folds, tra_i, val_i, $i = 1, \ldots, 10$. tra_i are used to train the decision trees and val_i are used to prune the decision trees using cross-validation based postpruning. *test* is used to estimate the expected error of the decision trees. We use paired t test for pairwise comparison ($\alpha = 0.05$). Since we are doing 3 pairwise comparisons on each dataset, to alleviate type I errors, we use Bonferroni correction and adjust p-value to α / 3 = 0.017.

We also use *Nemenyi's test* as the post-hoc test to compare neighboring

Table 1: Details of the datasets. d: Number of attributes, C: Number of classes, N: Sample size, n/v: n attributes of the dataset has v distinct values

Dataset	d	C	N	n/v
acceptors	88	2	3889	88/4
artificial	10	2	320	10/2
balance	4	3	625	4/5
car	6	4	1728	3/3, 2/4, 1/5
donors	13	2	6246	13/4
hayesroth	4	3	160	1/3, 3/4
krvskp	36	2	3196	35/2, 1/3
monks	6	2	432	2/2, 3/3, 1/4
nursery	8	5	12960	1/2, 4/3, 2/4, 1/5
promoters	57	2	106	57/4
spect	22	2	267	22/2
splice	60	3	3175	60/4
tictactoe	9	2	958	9/3
titanic	3	2	2201	2/2, 1/4
vote	16	2	435	16/2

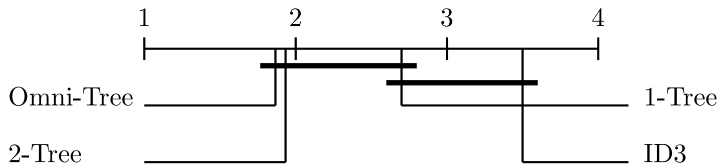
algorithms for significant difference in rank [16]. Two algorithms lead to classifiers with significantly different performance ranks at significance level α if the difference of their average ranks is greater than or equal to the critical difference

$$CD = q_\alpha \sqrt{\frac{L(L+1)}{6M}} \quad (1)$$

where L represents the number of algorithms to be compared, M represents the number datasets on which the comparison is done, and q_α is the Studentized range statistic divided by $\sqrt{2}$. This allows us to find cliques of equally good subsets which we can represent by underlining them.

Table 2: The averages and standard deviations of the error rates of decision trees generated using ID3, Omni-tree, and K-tree algorithms with $k = 1, 2$. Statistically significant differences are shown in boldface. The figure below shows the result of post-hoc Nemenyi's test.

Dataset	ID3	1-tree	2-tree	Omni-tree
acceptors	15.7 ± 1.7	**12.7 ± 0.8**	**12.0 ± 0.9**	**12.1 ± 1.1**
artificial	0.7 ± 1.6	0.7 ± 1.6	0.7 ± 1.6	0.7 ± 1.6
balance	40.5 ± 2.7	**27.1 ± 3.4**	**24.9 ± 2.3**	**25.1 ± 2.5**
car	12.5 ± 1.9	**4.0 ± 0.6**	**2.4 ± 0.5**	**2.3 ± 0.6**
donors	7.8 ± 0.7	**6.2 ± 0.3**	**6.7 ± 0.4**	**6.7 ± 0.4**
hayesroth	26.7 ± 1.5	**21.5 ± 4.1**	27.8 ± 4.8	27.3 ± 4.7
krvskp	1.2 ± 0.4	**1.0 ± 0.4**	0.9 ± 0.4	0.9 ± 0.4
monks	14.7 ± 6.1	12.7 ± 5.8	**0.0 ± 0.0**	**0.0 ± 0.0**
nursery	5.5 ± 0.5	**1.7 ± 0.3**	**0.4 ± 0.2**	**0.4 ± 0.2**
promoters	24.4 ± 10.3	27.2 ± 12.2	20.6 ± 4.0	20.0 ± 4.9
spect	20.3 ± 2.5	20.3 ± 2.5	20.9 ± 0.7	20.9 ± 0.7
splice	9.8 ± 0.9	**7.1 ± 0.8**	**6.1 ± 0.6**	**6.3 ± 1.1**
tictactoe	22.8 ± 1.6	**10.5 ± 3.7**	**9.0 ± 2.7**	**9.1 ± 2.2**
titanic	21.8 ± 0.5	21.7 ± 0.5	21.3 ± 0.4	21.3 ± 0.4
vote	5.1 ± 0.4	5.1 ± 0.4	5.0 ± 0.3	5.0 ± 0.3



6.1. Decision Trees

In this section, we compare the performance of our proposed decision tree algorithm (K-tree) with ID3 in terms of generalization error and model complexity as measured by the number of nodes in the decision tree. We run our proposed algorithm for $k = 1$ and 2.

Table 2 shows the averages and standard deviations of the error rates of decision trees generated using ID3, Omni-tree, and K-tree algorithms with $k = 1, 2$. If the difference between ID3 and K-tree is statistically significant, we show the winner in boldface. We see from the results that, K-tree is significantly better than ID3 in terms of error rate. 1-tree is significantly better than ID3 in 9 datasets, 2-tree is significantly better than ID3 in 8 datasets, and Omni-tree is significantly better than ID3 in 8 datasets out of 15. Especially, on *balance*, *tictactoe*, *car*, and *nursery* datasets, K-tree has 2, 3, 6, 10 times better error rate compared to ID3. On *monks* dataset, 2-tree extracts the hidden rule that returns us zero error rate. Post-hoc Nemenyi's test's results show that the best algorithm is Omni-tree and there are two cliques of algorithms (Omni-tree, 2-tree, 1-tree) and (1-tree, ID3).

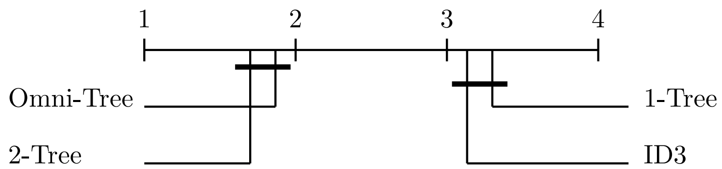
On 11 datasets, increasing k also improves accuracy. On the other hand, on other datasets such as *hayesroth*, *donors*, and *spect* increasing k may cause over-fitting, although there is pruning. On *artificial*, *spect*, and *vote* datasets, the number of distinct values of the attributes is 2, therefore the number of distinct possible splits for the extracted features are significantly smaller than other datasets. For this reason, K-tree performs worse on *artificial*, *spect*, and *vote* than on other datasets.

Table 3 shows the averages and standard deviations of the number of nodes of decision trees generated using ID3, Omni-tree, and K-tree algorithms with $k = 1, 2$. Similar to the results above, on ten datasets K-tree generates smaller trees compared to ID3. 1-tree is significantly better than ID3 in 2 datasets out of 15, 2-tree is significantly better than ID3 in 9 datasets out of 15, and Omni-tree is significantly better than ID3 in 9 datasets out of 15. Especially on *donors*, *monks*, *nursery*, and *splice* the K-trees are at least 3 times smaller than the ID3's trees. Post-hoc Nemenyi's test's results show that the best algorithm is 2-tree. There are two cliques of algorithms and clique (Omni-tree, 2-tree) is significantly better than clique (1-tree, ID3).

In most of the cases, as we increase k, the tree complexity decreases. Note that, if you only store the index of the new feature and the split point as integers, the decision nodes of K-tree's are equally complex as the decision nodes of ID3.

Table 3: The averages and standard deviations of the number of nodes of decision trees generated using ID3, and K-tree algorithms with $k = 1$, 2.

Dataset	ID3	1-tree	2-tree	Omni-tree
acceptors	10.2 ± 6.9	8.8 ± 4.8	7.2 ± 4.0	6.9 ± 4.5
artificial	4.6 ± 0.8	4.6 ± 0.8	$\mathbf{2.8 \pm 0.4}$	$\mathbf{2.8 \pm 0.4}$
balance	2.2 ± 1.8	12.8 ± 4.6	10.6 ± 2.6	10.4 ± 2.8
car	24.1 ± 3.4	31.6 ± 3.3	$\mathbf{18.5 \pm 2.1}$	$\mathbf{20.1 \pm 3.2}$
donors	24.2 ± 6.4	$\mathbf{19.3 \pm 6.4}$	$\mathbf{8.5 \pm 4.6}$	$\mathbf{8.3 \pm 4.1}$
hayesroth	5.2 ± 0.8	7.9 ± 1.4	4.5 ± 0.7	4.7 ± 0.7
krvskp	26.9 ± 4.2	25.9 ± 3.1	$\mathbf{16.2 \pm 1.7}$	$\mathbf{16.4 \pm 1.5}$
monks	15.0 ± 3.8	21.8 ± 7.9	$\mathbf{4.0 \pm 0.0}$	$\mathbf{4.0 \pm 0.0}$
nursery	103.7 ± 9.4	121.9 ± 6.1	$\mathbf{34.3 \pm 2.5}$	$\mathbf{34.4 \pm 2.6}$
promoters	1.6 ± 1.1	1.2 ± 0.9	1.5 ± 0.7	1.6 ± 0.8
spect	0.8 ± 2.5	0.8 ± 2.5	0.5 ± 1.6	0.5 ± 1.6
splice	21.3 ± 2.5	$\mathbf{15.9 \pm 3.1}$	$\mathbf{8.2 \pm 1.8}$	$\mathbf{8.0 \pm 2.7}$
tictactoe	23.6 ± 4.7	23.7 ± 3.9	$\mathbf{14.5 \pm 4.2}$	$\mathbf{14.7 \pm 4.9}$
titanic	3.9 ± 0.7	4.8 ± 1.0	$\mathbf{2.5 \pm 1.4}$	$\mathbf{2.5 \pm 1.4}$
vote	1.9 ± 1.2	1.9 ± 1.2	2.0 ± 1.8	2.0 ± 1.8



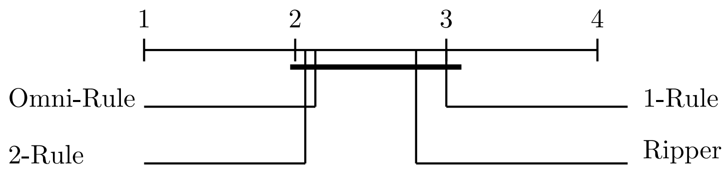
6.2. Rule Inducers

In this section, we compare the performance of our proposed rule induction algorithm (K-rule) with Ripper proper in terms of generalization error and model complexity as measured by the number of rules in the rulesets. We run our proposed algorithm for $k = 1$ and 2.

Table 4 shows the averages and standard deviations of the error rates of rulesets generated using Ripper, and K-rule algorithms with $k = 1$, 2. We see

Table 4: The averages and standard deviations of the error rates of rulesets generated using Ripper, and K-rule algorithms with k = 1, 2. Statistically significant differences are shown in boldface (if K-rule is better) and italic (if Ripper is better).

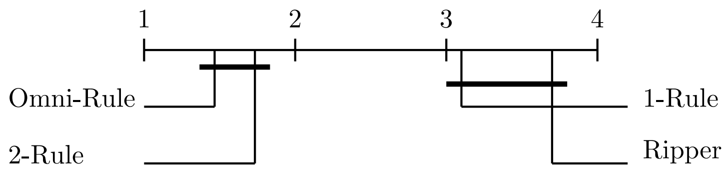
Dataset	Ripper	1-rule	2-rule	Omni-rule
acceptors	14.9 ± 0.9	**11.8 ± 1.1**	**11.8 ± 1.0**	**11.4 ± 0.9**
artificial	0.4 ± 1.2	0.4 ± 1.2	0.0 ± 0.0	0.0 ± 0.0
balance	34.6 ± 1.1	**28.1 ± 2.5**	**26.0 ± 1.6**	**27.8 ± 2.3**
car	20.4 ± 2.1	**10.7 ± 1.9**	**5.9 ± 2.6**	**7.1 ± 3.4**
donors	6.1 ± 0.3	*6.7 ± 0.6*	*6.7 ± 0.3*	*6.4 ± 0.3*
hayesroth	23.5 ± 4.2	*32.4 ± 0.8*	*32.9 ± 5.2*	*33.8 ± 7.9*
krvskp	1.4 ± 0.6	1.1 ± 0.5	**0.9 ± 0.2**	**0.9 ± 0.3**
monks	0.0 ± 0.0	3.0 ± 7.5	0.0 ± 0.0	0.0 ± 0.0
nursery	5.9 ± 0.6	**3.8 ± 0.4**	**1.2 ± 0.5**	**1.1 ± 0.4**
promoters	20.6 ± 3.5	21.4 ± 2.6	21.1 ± 1.9	21.1 ± 4.0
spect	21.3 ± 4.0	21.3 ± 4.0	21.6 ± 1.1	21.2 ± 0.4
splice	6.7 ± 0.8	6.8 ± 1.0	6.3 ± 0.5	7.0 ± 0.9
tictactoe	1.6 ± 0.2	1.6 ± 0.0	*9.2 ± 5.1*	*13.1 ± 8.6*
titanic	22.4 ± 0.6	22.2 ± 0.8	22.0 ± 1.0	22.1 ± 0.9
vote	6.0 ± 2.0	6.0 ± 2.0	6.0 ± 2.0	5.6 ± 1.5



from the results that, although K-rule is better than Ripper, the difference is not as meaningful as the decision tree case. 1-rule wins against Ripper in 4 to 2 out of 15 datasets, 2-rule wins against Ripper in 5 to 3 out of 15 datasets, and Omni-Rule wins against Ripper in 5 to 3 out of 10 datasets. Nemenyi's test also supports this claim, it does not find any significant difference between algorithms and returns a single clique of algorithms (1-Rule, 2-Rule, Omni-Rule,

Table 5: The averages and standard deviations of the number of rules of rulesets generated using Ripper, and K-rule algorithms with $k = 1, 2$. Statistically significant differences are shown in boldface (if K-rule is better) and italic (if Ripper is better).

Dataset	Ripper	1-rule	2-rule	Omni-rule
acceptors	7.5 ± 1.6	**2.6 ± 1.1**	**2.2 ± 0.8**	**2.0 ± 0.8**
artificial	3.0 ± 0.0	3.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0
balance	4.0 ± 0.8	**2.5 ± 0.7**	**2.0 ± 0.0**	**2.0 ± 0.0**
car	9.0 ± 2.2	**6.9 ± 1.4**	**5.5 ± 0.7**	**5.4 ± 1.3**
donors	10.5 ± 1.6	**7.5 ± 1.5**	**5.0 ± 1.1**	**5.8 ± 0.8**
hayesroth	5.6 ± 0.5	**4.0 ± 0.0**	**3.0 ± 0.5**	**3.0 ± 0.7**
krvskp	12.0 ± 2.4	13.2 ± 1.8	**7.7 ± 0.8**	**7.8 ± 0.8**
monks	4.0 ± 0.0	3.8 ± 1.0	**3.4 ± 0.5**	**3.4 ± 0.5**
nursery	71.3 ± 9.4	**45.1 ± 5.0**	**18.8 ± 2.3**	**19.6 ± 3.8**
promoters	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
spect	0.8 ± 0.4	0.8 ± 0.4	**0.2 ± 0.4**	**0.1 ± 0.3**
splice	8.1 ± 1.8	**5.1 ± 1.8**	**4.7 ± 0.9**	**4.5 ± 1.0**
tictactoe	8.2 ± 0.6	8.1 ± 0.3	7.3 ± 1.3	**6.1 ± 2.0**
titanic	2.0 ± 0.9	1.4 ± 0.5	**1.1 ± 0.3**	**1.1 ± 0.3**
vote	1.3 ± 0.5	1.3 ± 0.5	1.3 ± 0.5	1.1 ± 0.3



Ripper).

On 12 datasets, increasing k also improves accuracy. On the other hand, for other datasets such as *hayesroth*, *tictactoe*, and *spect*, although there is pruning, increasing k causes over-fitting.

Table 5 shows the averages and standard deviations of the number of rules of rulesets generated using Ripper, and K-rule algorithms with $k = 1, 2$. For this case, the results are the same as the decision tree case. K-rule is again significantly better than Ripper. 1-rule wins against Ripper in 7 to 0 out of 15 datasets, 2-rule wins against Ripper in 11 to 0 out of 15, and Omni-rule wins against Ripper in 12 to 0 datasets out of 15 datasets. Post-hoc Nemenyi's test's results show that the best algorithm is Omni-rule. There are two cliques of algorithms and again clique (Omni-rule, 2-rule) is significantly better than clique (1-rule, Ripper).

The same logic also applies here. In all datasets except *promoters* and *vote*, as we increase k, the ruleset complexity decreases. Note again that, if you only store the index of the new feature and the split point as integers, the decision conditions of K-rule's are equally complex as the decision conditions of Ripper.

7. Conclusions

In this paper, we propose a new framework to order a subset of k discrete attributes. Using all orderings of values of those k attributes as new extracted features, we propose two novel classifiers based on ID3 and Ripper. Our simulation results on 15 discrete datasets show that our proposed algorithms performs better than their counterparts in terms of error rate and tree complexity.

Although k-ordering help both tree and rule algorithms to produce significantly better classifiers than their counterparts, the time complexity in the training phase (due to the exhaustive search nature in the induced space) may prevent them from being good competitors. Since bootstrapping is based on different combinations and k-ordering is based on different permutations, one can establish a possible relationship between them. Similar to K-trees, K-forests based on random forests can be proposed and instead of searching for the best split on the whole induced space, one can search for the best split on a significantly smaller and therefore tractable subset of the induced space.

Acknowledgments

This work has been supported by the Turkish Scientific Technical Research Council (TÜBİTAK) EEEAG 107E127.

References

[1] J. R. Quinlan, Induction of Decision Trees, Machine Learning 1 (1986) 81–106.

[2] W. W. Cohen, Fast Effective Rule Induction, in: The Twelfth International Conference on Machine Learning, 115–123, 1995.

[3] C. Blake, C. Merz, UCI Repository of Machine Learning Databases, URL `http://www.ics.uci.edu/~mlearn/MLRepository.html`, 2000.

[4] E. Alpaydın, Introduction to Machine Learning, The MIT Press, 2010.

[5] V. Vapnik, The Nature of Statistical Learning Theory, Springer Verlag, New York, 1995.

[6] S. K. Murthy, Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey, Data Mining and Knowledge Discovery 2 (4) (1998) 345–389.

[7] O. T. Yıldız, E. Alpaydın, Linear Discriminant Trees, International Journal of Pattern Recognition and Artificial Intelligence 19 (3) (2005) 323–353.

[8] J. Fürnkranz, Separate-and-Conquer Learning, Artificial Intelligence Review 13 (1999) 3–54.

[9] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann, San Meteo, CA, 1993.

[10] R. L. D. Mantaras, A distance based attribute selection measure for decision tree induction, Machine Learning 6 (1986) 81–92.

[11] G. Pagallo, D. Haussler, Boolean feature discovery in empirical learning, Machine Learning (1990) 71–99.

[12] C. J. Mathues, L. A. Rendell, Constructive induction on decision trees, in: Eleventh International Joint Conference on Artificial Intelligence, 645–650, 1989.

[13] O. T. Yıldız, Feature Extraction from Discrete Attributes, in: Proceedings of the 20th International Conference on Pattern Recognition, 3915–3918, 2010.

[14] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification and Regression Trees, John Wiley and Sons, 1984.

[15] O. T. Yıldız, E. Alpaydın, Omnivariate Decision Trees, IEEE Transactions on Neural Networks 12 (6) (2001) 1539–1546.

[16] J. Demsar, Statistical Comparisons of Classifiers over Multiple Data Sets, Journal of Machine Learning Research 7 (2006) 1–30.