Minimum Splits Based Discretization for Continuous Features

Ke Wang and **Han Chong Goh**
Dept of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore, 119260
wangk@iscs.nus.sg

Abstract

Discretization refers to splitting the range of continuous values into intervals so as to provide useful information about classes. This is usually done by minimizing a goodness measure, subject to constraints such as the maximal number of intervals, the minimal number of examples per interval, or some stopping criterion for splitting. We take a different approach by searching for *minimum splits* that minimize the number of intervals with respect to a threshold of impurity (i.e., badness). We propose a "total entropy" motivated selection of the "best" split from minimum splits, without requiring additional constraints. Experiments show that the proposed method produces better decision trees.

1 Introduction

Continuous values refer to linearly ordered values, mainly numeric values. While continuous values are common in real applications, many learning algorithms focus on unordered discrete values. A common practice is to *discretize* continuous values into intervals so as to provide useful information with respect to classes. Discretization can be performed either on the whole dataset prior to induction, i.e, global discretization, or on local regions during induction, i.e., local discretization. We focus on local discretization as it takes into account the context sensitivity of the nature. One example of local discretization is the entropy-based C4.5 [Quinlan, 1993], in which continuous values are split into two intervals, i.e., binary splitting, for consideration at a node. However, as pointed out in [Fayyad and Irani, 1993], an interesting range is usually an internal interval within the feature's range, and to get to such an interval a binary-split-at-a-time leads to unnecessary and excessive partitioning of the examples. We now provide further reasons for multi-way splits.

One frequent argument against multi-way splitting is that a multi-way split can be "simulated" by a series of binary splittings. Though theoretically true, this argument is false in the process of generating decision trees where there is no guarantee that all "simulating" binary splittings will be finished up before considering other features because the splitting at each level is performed independently. As a result, a structured multi-way split is hardly simulated by binary splits in practice. In addition, by restricting to only binary splits, an unstructured feature could be selected instead of a structured but never explored multi-way split of a continuous feature, making the simple structure disappear. Consider the following two decision trees built in one of the 10-fold cross validation on Iris dataset. The first tree is produced by the multi-way split proposed in this paper, and the second by C4.5. Though both trees have the same size and same error rate on test data, the first tree classifies most examples at the first level using simple rules and thus is preferred. The reason why C4.5 didn't select petal length at the first level is because the optimal binary split of petal length loses to that of petal width. As a result, the simple one-level test for most examples, as in the first tree, is not discovered. This example also reveals the bias of the tree size measure: it does not take the frequency, i.e., importance, of rules into consideration.

```
petal length <= 1.9 : Iris-setosa (45.0)
petal length <= 4.7 : Iris-versicolor (41.0/1.0)
petal length > 5 : Iris-virginica (38.0)
petal length <= 5 :
|   petal width <= 1.7 : Iris-versicolor (4.0)
|   petal width > 1.7 : Iris-virginica (7.0/1.0)

petal width <= 0.6 : Iris-setosa (45.0)
petal width > 0.6 :
|   petal width > 1.7 : Iris-virginica (43.0/1.0)
|   petal width <= 1.7 :
|   |   petal length <= 5 : Iris-versicolor (45.0/1.0)
|   |   petal length > 5 : Iris-virginica (2.0)
```

Searching for a good multi-way split is significantly more complicated than searching for a good binary split.

The brute-force solution is exponential in $k-1$, where k is the number of intervals considered. Since the number of intervals in a multi-way split is unknown and not fixed a prior, goodness measures such as the gain that are always improved by further splitting do not work directly, and have to be coupled with other constraints or criterions. In this paper, we address the search problem in two steps. First, we define the notion of *minimum splits* as a necessary condition for a good split. A minimum split wrt a threshold of impurity is a split that minimizes the number of intervals subject to the threshold. We propose a dynamic programming algorithm for finding a minimum split for any impurity measure that is *additive* in the sense that the impurity of several intervals is the sum of the impurities of each interval. This includes most standard impurity measures, such as entropy [Quinlan, 1993], twoing rule and Gini index [Breiman *et al.*, 1984], Sum Minority, inconsistency rate, and others. The dynamic programming algorithm runs in a quadratic time in the number of starting intervals. We describe two approximation algorithms that can compute nearly optimal solutions efficiently for large datasets.

We then propose a method of determining the "best" split from a collection of minimum splits called *candidate splits*, found in a single run of the algorithm for finding minimum splits. The "best" split of continuous values is the candidate split that has the smallest product of entropy and number of intervals. Intuitively, this product measures the total information, rather than the average information like the standard entropy, of all intervals. We compared the proposed minimum split method with Release 8 of C4.5, a substantial improvement of early releases on handling continuous values. The study shows that multi-way splits usually build decision trees that are shallow and classify more examples at upper levels of the tree, compared to binary splits. We propose the notion of testing depth to capture this aspect of simplicity of decision trees, which is not addressed by the tree size.

Several recent papers have examined discretization of continuous values. One approach, e.g., [Kerber, 1992; Richeldi and Rossotto, 1995], starts with one interval per value and repeatedly merges adjacent intervals based on some "similarity" measure. It could be difficult to specify a good threshold of similarity so that not too many intervals are constructed. Another approach aims at finding a split that optimizes some goodness criterion. Examples are [Quinlan, 1993; Catlett, 1991; Holte, 1993; Chiu *et al.*, 1990; Fulton *et al.*, 1995; Auer *et al.*, 1995]. See [Dougherty *et al.*, 1995] for more on these work. In these methods, additional constraints, such as the maximum number of intervals, the minimum number of examples in each interval, a penalty function on the number of intervals, are needed to control the number of intervals. [Catlett, 1991; Fayyad and Irani, 1993] use recursive application of binary splitting to obtain a multi-way split coupled with some criterion for stopping the splitting process. The problem with such splittings is that some optimal k-way splits do not come from bi-partitioning an interval in an optimal $(k-1)$-way split. For example, if Sum Minority is the impurity measure (i.e., the number of examples belonging to minority classes in an interval), the optimal 3-way split $(ABBBB)(AAA)(BBBBA)$ cannot be obtained by bi-partitioning one interval in any of the (only) two optimal binary splits

$$(A)(BBBBAAABBBBA),$$
$$(ABBBBAAABBBB)(A),$$

where A and B present examples of different classes. The approach of multivariate decision trees considers a combination of several continuous features rather than a single feature at a node test. Such decision trees are still univariate ones on the transformed features. Therefore, the search for univariate decision trees of high performance is a fundamental and important research area.

2 Minimum splits

We propose the notion of minimum splits as a necessary condition for a good split. The motivation is simple: for a given threshold of impurity, a good split should have no unnecessary splitting with respect to the threshold, that is, no intervals in the split can be merged without exceeding the threshold. We formalize this notion below. Assume that the training examples are initially partitioned into $X_1, \ldots, X_k$ such that all examples in X_i have a smaller value than those in X_j for $i < j$. $X_1, \ldots, X_k$ are called *bins*. A simple case is that each bin X_i corresponds to one observed value. Other cases which give a smaller k will be considered in Section 2.4. By *splitting* bins $X_1, \ldots, X_k$ into several intervals, we mean that adjacent bins may be put into the same interval, but no examples in the same bin can be placed in different intervals.

2.1 Definition

Definition 2.1 (Minimum splits) *Given threshold γ of impurity, a n-way split of $X_1, \ldots, X_k$ is* minimum *wrt γ if n is the lowest value so that no more than γ impurity is produced. A minimum split of $X_1, \ldots, X_k$ wrt γ is* strong *if it has the smallest impurity among all minimum splits of $X_1, \ldots, X_k$ wrt γ. The* minimum split problem *is to find a minimum split wrt a given threshold of impurity.*

All minimum splits wrt γ (if any) have the same number of intervals but may have different impurities. All strong minimum splits wrt γ (if any) have both the same number of intervals and the same impurity. The split selection algorithm in the next section will always choose a

strong minimum split. The minimum split problem has immediate application in feature selection. For example, if the impurity measure is the inconsistency rate, detecting irrelevant continuous features is equivalent to finding features whose minimum splits with respect to zero impurity have only one interval. As another example, if a discrete feature A has entropy a, solving the minimum split problem for a continuous feature C with respect to a gives the minimum number of intervals for C to beat A in entropy. With this information, one can choose the feature that has fewer branches.

We now examine algorithms for finding a minimum split for a general impurity measure that is additive as explained in Introduction; algorithms for a strong minimum split can be obtained easily from such algorithms. Let $imp(i,j)$ denote the impurity of the single interval containing all continuous values in $X_i, \ldots, X_j$. Let $I(x,j)$ and $M(x,j)$, respectively, denote the number of intervals and the impurity of a minimum split of $X_1, \ldots, X_j$ with respect to x. $I(x,j)$ and $M(x,j)$ are undefined if there is no such minimum split. From the minimality of the number of intervals and the additivity of the impurity measure,

$$I(x,i) = min_{0 \le j < i} I(x - imp(j+1,i), j) + 1,$$

where only defined $I(x - imp(j+1,i), j)$ are considered. That is, a minimum split of $X_1, \ldots, X_i$ wrt x comes from a minimum split of $X_1, \ldots, X_j$ wrt threshold $x - imp(j+1,i)$, $0 \le j < i$, such that the right-most cutpoint j minimizes $I(x - imp(j+1,i), j)$. Assume that j' minimizes $I(x - imp(j+1,i), j)$. Then

$$M(x,i) = M(x - imp(j'+1,i), j') + imp(j'+1,i).$$

However, such a recursive computation of $I(x,i)$ and $M(x,i)$ is very expensive because of repeated calls of the same computation. We present a dynamic programming algorithm which runs in quadratic time in the number of starting bins.

2.2 The dynamic programming algorithm

We compute $I(x,i)$ and $M(x,i)$ in the increasing order of x and i. For a general impurity such as the real-valued entropy, it does not work to enumerate all possible values for x. Our approach is to represent the domain $[0,\gamma]$ of x by a number of values determined from a desired precision of impurity denoted ϵ below. For example, a difference in entropy less than 0.01 is usually considered insignificant, so we can choose $\epsilon = 0.01$. Now if threshold $\gamma = 0.3562$, there are 37 values for x to be examined: $0.00, 0.01, 0.02, \ldots, 0.36$. In implementation, we replace the domain $[0,\gamma]$ of x with the integers $\{0, 1, \ldots, \gamma/\epsilon\}$, where γ is rounded to precision ϵ. Accordingly, $imp(j+1,i)$ is replaced with $Round(imp(j+1,i)/\epsilon)$, i.e., the rounding of $imp(j+1,i)/\epsilon$ to the closest integer.

However, one critical issue must be addressed with care: the error caused by rounding $imp(j+1,i)/\epsilon$ could accumulate through intervals and cause defined $I(x,i)$ to be undefined. To see this, let $\epsilon = 0.01, imp(1,1) = 0.545$, and $imp(2,2) = 0.255$. Thus, X_1, X_2 can be split into two intervals at cutpoint $j = 1$ without exceeding threshold $x = 0.80$. On the other hand, $Round(imp(2,2)/\epsilon) = 26$, which leaves only $0.80/\epsilon - 26 = 54$ for X_1, making $I(54,1)$ undefined because $Round(imp(1,1)/\epsilon) = 55$. As a result, the split at $j = 1$ is mistakenly not considered. In general, since $x - Round(imp(j+1,i)/\epsilon)$ is the threshold for splitting $X_1, \ldots, X_j$, the error caused by the rounding may accumulate through intervals. To avoid this, we perform the rounding at each interval in such a way that the accumulated error is always within $\pm\epsilon$. This is done by replacing $Round(imp(j+1,i)/\epsilon)$ with $Round(imp(j+1,i)/\epsilon, E(j))$, where $E(j)$ is the accumulated error up to X_j. As before, $Round(imp(j+1,i)/\epsilon, E(j))$ returns the rounding of $imp(j+1,i)/\epsilon$, but the rounding direction is to reduce the error in $E(j)$. See below. We have omitted the part of retrieving cutpoints as it can be easily done by storing the right-most cutpoint j for each pair of x and i. Initially, $I(x,0) = M(x,0) = 0$ for $x = 0, 1, \ldots, \gamma/\epsilon$; all other elements of I and M are undefined. y abbreviates $Round(imp(j+1,i)/\epsilon, E(j))$.

$Round(u,v)$:
if $v \ge 0$ then return $floor(u)$;
  else return $ceiling(u)$;

Dynamic Programming Algorithm:
for $x = 0$ to γ/ϵ do
  $E(0) = 0.0$;
  for $i = 1$ to k do
(*) find a j, $0 \le j < i$, such that
      $I(x-y,j)$ is defined and minimized;
    if found then
      $I(x,i) = I(x-y,j) + 1$;
      $M(x,i) = M(x-y,j) + y * \epsilon$;
      $E(i) = E(j) + y * \epsilon - imp(j+1,i)$;

Section 3 will discuss how to determine threshold γ while applying the minimum split problem to select a good split of continuous values. The precision ϵ is usually chosen so that the number γ/ϵ of impurity values considered is between 10 and 100, usually between 10 and 20. A too large γ/ϵ does not necessarily improve the test quality since a very small difference is likely due to randomness.

Let us consider the complexity of the above algorithm. Matrix $imp(i,j)$ can be computed outside the main loop. For impurity measures that use only count information of examples and classes, such as entropy, Sum Minority, and many others, matrix $imp(i,j)$ can be computed incrementally from small i,j to large ones in

time $O(W * k^2)$, where W is the number of classes.

Theorem 2.1 *Let ϵ be the precision of the impurity measure. The dynamic programming finds a minimum split for k bins (if any) wrt γ impurity in $O((\gamma/\epsilon + W) * k^2)$ time, where W is the number of classes.*

The largest k corresponds to the case where each bin contains exactly examples that have the same continuous value. A smaller k is possible by merging adjacent bins, which will be considered in subsection 2.4. Note that the dynamic programming not only returns a minimum split of $X_1, \ldots, X_k$ wrt γ, but also a minimum split of $X_1, \ldots, X_k$ wrt x, where $x = 0, \epsilon, \ldots, \gamma - \epsilon$ for "free". In Section 3, these minimum splits form the search space of the optimal split. To find a strong minimum split, the only change required is to add the minimization of $M(x - y, j) + y * \epsilon$ as the second order requirement in step (*) of the above dynamic programming algorithm.

2.3 The greedy algorithm

If the number k of bins is large, the dynamic programming algorithm could be expensive. We present a linear time algorithm by greedily merging adjacent intervals until the threshold γ is exceeded. Initially, each bin X_i itself forms an interval. Let imp_0 be the impurity for this initial state. At each step the two adjacent intervals whose merging adds the least impurity are merged, thus greedily maximizing the number of mergings or minimizing the number of intervals. We assume $imp_0 \leq \gamma$, otherwise, there is no minimum split wrt γ. For an interval I_i, let $imp(I_i)$ denote the impurity of I_i. $I_i \cup I_{i+1}$ denotes the merged interval of two adjacent intervals I_i and I_{i+1}.

Greedy Algorithm:
$exceed = false$;
$n = k$;
$z = imp_0$;
while ($exceed = false$ and $n > 1$) do
  find the pair of adjacent intervals (I_j, I_{j+1}) minimizing
    $\Delta = imp(I_j \cup I_{j+1}) - imp(I_j) - imp(I_{j+1})$;
  if $z + \Delta \leq \gamma$ then
    merge I_j and I_{j+1};
    $z = z + \Delta$;
    $n = n - 1$;
  else $exceed = true$;

Finding the next pair (I_j, I_{j+1}) for merging in a fixed time without scanning all pairs of adjacent intervals is the key of efficiency. For this purpose, we use the B-tree [Elmasri and Navathe, 1994] from the database research to index pairs of adjacent intervals (I_j, I_{j+1}) in the increasing order of increment $\Delta = imp(I_j \cup I_{j+1}) - imp(I_j) - imp(I_{j+1})$. With the B-tree of branching factor b (i.e., the average number of branches at a node) on k indexed values, search, insertion and deletion of a value Δ can be done by accessing a number of tree nodes equal to the height of the B-tree, that is, $log_b k$. Usually a large b, e.g., more than 20, is chosen to reduce the height of the B-tree. For example, if $b = 20$ and if there are three million distinct values, inserting, deleting, and searching a Δ value needs to access at most 5 nodes of the tree.

We build the initial B-tree by inserting increment Δ for all pairs of adjacent bins (X_j, X_{j+1}). Within each iteration, we find the smallest Δ at the front of the leaf level (remember increments Δ are sorted at leaves) and merge the two intervals that define this Δ. Suppose that intervals I_a, I_b, I_c, I_d are adjacent and that pair (I_b, I_c) is merged. After the merging, Δ_{ab} for (I_a, I_b) and Δ_{bc} for (I_b, I_c) are affected, and Δ_{cd} for (I_c, I_d) should be deleted. In the B-tree, this is done by deleting old $\Delta_{ab}, \Delta_{bc}, \Delta_{cd}$ and inserting new Δ_{ab}, Δ_{bc}. To find the old Δ_{ab}, Δ_{bc}, we store two pointers, p_l and p_r, together with increment Δ of every adjacent pair (I_1, I_2). p_l points to the increment Δ for adjacent pair (I_0, I_1), and p_r points to the increment Δ for pair (I_2, I_3). Therefore, after Δ_{bc} is deleted, p_l and p_r stored at Δ_{bc} give the addresses of the old Δ_{ab} and Δ_{cd} for deletion. The new Δ_{ab} and Δ_{cd} are then inserted into the B-tree. The p_l and p_r links are updated to reflect that new Δ_{ab} and Δ_{cd} are adjacent on these links. It can be seen that these operations cost only a small factor of the height of the B-tree, which still can be considered a constant for large data size.

Theorem 2.2 *The greedy algorithm runs in time $O(h * k)$ for k bins, where h is the height of the B-tree on $k-1$ indexed values.*

For a large branching factor of the B-tree, h is practically a constant as explained early.

2.4 The hybrid algorithm

The third strategy is a compromise between optimality and speed through the following two-phase hybrid approach: the initial bins $X_1, \ldots, X_k$ are formed by placing all examples having the same continuous value in a bin. In the first phase, we apply the greedy algorithm to $X_1, \ldots, X_k$ to merge adjacent bins. Due to the greedy nature, bins that are merged into the same interval have similar class distribution and thus little or no impurity is introduced. Let k' be the number of intervals produced. Note that $k' < k$. In the second phase, we apply the dynamic programming algorithm to the k' intervals produced by the greedy algorithm, by treating such intervals as starting bins. The idea is to run a fast but less accurate algorithm on the initial large data and switch to an accurate but slower algorithm after the number of intervals is reduced. By applying the greedy algorithm as interval reduction, the quadratic term k^2 in Theorem

2.1 is reduced to k'^2. In general, the switching point k' represents a trade-off between optimality and speed.

3 The optimal split

We now address the central question of how to determine the optimal split for a continuous feature at a node of decision trees. Once the optimal split is determined for every continuous feature, any existing selection criterion for discrete features is applied to select the best feature at the node. We use the entropy [Quinlan, 1993] as the impurity measure.

As motivated in Section 2, a good split must be a minimum split, therefore we consider only minimum splits when searching for the optimal split. Since any threshold larger than e_b gives a minimum split having at most two intervals, where e_b is the entropy of optimal binary splits, we need only to consider thresholds not more than e_b. In other words, the search space of the optimal split is the set of minimum splits of $X_1, \ldots, X_k$ wrt x, where $x = 0, \epsilon, 2\epsilon, \ldots, e_b$, which have at least two intervals. From Section 2, these minimum splits can be found in a single run of the dynamic programming algorithm wrt e_b. We call these splits *candidate splits* for feature A. The set of candidate splits is empty only if all examples belong to the same class, in which case there is no need of splitting at the node. If the greedy algorithm is run instead, splits after each iteration form an approximation of candidate splits.

Let $I(x,k)$ and $M(x,k)$ denote the number of intervals and entropy of a minimum split of $X_1, \ldots, X_k$ wrt x. In the search for the goodness measure of a split, we observed that the product $I(x,k) * M(x,k)$ usually gives a reasonable quality measure of the minimum split. On one hand, a "good" split usually has both a small number of intervals and a small entropy, thus yielding a small product. On the other hand, a small product but a "not-so-small" interval number entails a very small entropy, thus a nearly pure classification with no unnecessary splitting (due to minimum splits). It is possible, however, that some of these intervals are very small. We avoid such splits by requiring a minimal number of examples in an interval of a minimum split. With these said, we have:

The optimal split for a continuous feature: choose the candidate split that has the smallest product of entropy and number of intervals.

There is a natural interpretation for the above selection. Suppose that feature A is split into d intervals, with n_i examples and e_i entropy for the ith interval. Let N be the total number of examples. The entropy of the split is then given by

$$\Sigma_{i=1}^{d} \frac{n_i}{N} e_i.$$

This is exactly the weighed average of entropies for all intervals. Therefore, the above minimum product selection aims at minimizing the total entropy of a split. The following corollary says that the optimal split is well behaved by being actually a strong minimum split.

Corollary 3.1 *The optimal split selected above is a strong minimum split of $X_1, \ldots, X_k$ wrt some x.*

In fact, if the optimal split is not strong, there must be another minimum split (also in the set of candidate splits) having the same number of intervals but a smaller entropy, which yields a smaller product and thus it is preferred to the other one.

4 Empirical evaluation

We compared three algorithms: Release 8 of C4.5 (the latest release) with the default setting, the multi-way splitting based on dynamic programming, the multi-way splitting based on the greedy algorithm. These algorithms are denoted by C4.5(R8), Dynamic, and Greedy, respectively. The hybrid algorithm is not included because its performance is expected to lie between Dynamic and Greedy. In Dynamic, the bins to start with correspond to examples having the continuous value. Unlike early releases, C4.5(R8) improves the performance on continuous values by employing an MDL-inspired penalty to adjust the gain of a binary split of continuous values. As shown in [Quinlan, 1996], C4.5(R8) compares favorably with the multi-way split method T2 [Auer *et al.*, 1995] and the discretization method [Fayyad and Irani, 1993]. Therefore, we choose C4.5(R8) as a benchmark. Dynamic and Greedy determine the optimal split of continuous values as in Section 3 and select a feature for branching as in C4.5 using the optimal splits for continuous features. All three algorithms are applied to 15 datasets from the UCI repository [Murphy and Aha, 1994], all involving some continuous features and some involving many. All experiments are performed using 10-fold cross validation. The size and error rate of pruned decision trees are collected on test data. A summary is given in Table 1. The numbers following $\pm$ are standard errors.

In Table 1, the *testing depth* of a decision tree is defined as the average length of root-to-leaf paths weighed by the numbers of examples covered by leaves. Thus, the testing depth measures the average number of tests needed to classify an example, thus, the average complexity of rules used. A decision tree with a small testing depth is likely to classify examples by simple rules. This aspect of complexity is not reflected by the simple tree size. We highlight a few results shown in Table 1: (a) Except for a few datasets, Dynamic wins over Greedy in all three measurements. (b) On tree size Dynamic wins over C4.5(R8) in 12 out of 15 datasets, with 1 tie, and on error rate Dynamic wins over C4.5(R8) in 8 out of 15,

Dataset	Tree Size			Error Rate			Testing Depth		
	Dynamic	Greedy	C4.5(R8)	Dynamic	Greedy	C4.5(R8)	Dynamic	Greedy	C4.5(R8)
anneal	77.0 ± 1.4	75.6 ± 1.6	72.8 ± 2.3	5.6 ± 0.7	6.2 ± 0.8	7.5 ± 0.5	3.6 ± 0.0	3.8 ± 0.1	4.5 ± 0.1
australian	52.7 ± 2.6	57.4 ± 2.0	35.7 ± 3.5	14.0 ± 1.6	14.6 ± 1.5	14.9 ± 1.2	2.0 ± 0.1	2.1 ± 0.1	4.6 ± 2.1
breast-w	25.2 ± 1.4	21.7 ± 1.7	28.2 ± 1.7	5.0 ± 0.7	4.6 ± 0.8	5.6 ± 0.8	2.3 ± 0.1	2.3 ± 0.2	2.8 ± 0.1
bupa	42.2 ± 4.0	65.4 ± 4.5	43.8 ± 4.0	32.7 ± 2.0	37.3 ± 2.4	34.8 ± 1.5	3.8 ± 0.3	4.0 ± 0.2	4.3 ± 0.2
cleve	64.5 ± 3.9	67.8 ± 2.8	76.5 ± 3.6	44.3 ± 2.0	46.2 ± 1.0	47.2 ± 2.0	2.7 ± 0.1	2.7 ± 0.1	3.6 ± 0.1
diabetes	46.6 ± 3.9	50.1 ± 6.3	46.6 ± 4.4	25.6 ± 1.8	26.9 ± 1.7	26.1 ± 1.5	3.8 ± 0.1	3.3 ± 0.2	7.3 ± 0.4
german	118.9 ± 4.5	156.1 ± 5.0	142.8 ± 6.3	27.3 ± 0.8	29.1 ± 1.2	26.4 ± 0.7	5.0 ± 0.1	5.4 ± 0.2	6.1 ± 0.2
heart	26.7 ± 2.8	32.1 ± 3.2	36.4 ± 1.8	20.7 ± 1.4	20.3 ± 2.4	21.8 ± 1.6	3.0 ± 0.1	3.1 ± 0.2	3.5 ± 0.1
hepatitis	13.4 ± 1.2	13.7 ± 0.9	18.2 ± 1.7	18.9 ± 2.4	20.2 ± 3.3	18.2 ± 2.0	1.9 ± 0.1	1.9 ± 0.1	2.6 ± 0.2
hypothy.	11.0 ± 1.0	17.5 ± 2.1	12.2 ± 0.8	0.7 ± 0.2	0.8 ± 0.1	0.7 ± 0.1	1.2 ± 0.0	1.2 ± 0.0	1.2 ± 0.0
ionosphere	21.7 ± 2.2	28.5 ± 1.8	27.2 ± 1.1	11.1 ± 2.0	10.6 ± 1.8	10.5 ± 1.4	2.9 ± 0.1	2.8 ± 0.1	4.8 ± 0.2
iris	8.6 ± 0.7	11.0 ± 0.6	8.8 ± 0.4	7.3 ± 2.1	4.0 ± 1.5	4.7 ± 2.0	1.2 ± 0.0	1.5 ± 0.0	2.1 ± 0.0
labor	6.1 ± 1.3	6.7 ± 1.2	6.9 ± 0.9	22.7 ± 4.9	22.7 ± 4.9	22.3 ± 5.5	1.4 ± 0.2	1.5 ± 0.2	1.8 ± 0.2
sick-euthy.	24.0 ± 1.2	30.4 ± 2.7	24.1 ± 1.7	2.2 ± 0.3	2.3 ± 0.2	2.4 ± 0.3	1.5 ± 0.0	1.5 ± 0.0	1.6 ± 0.0
vehicle	129.0 ± 6.8	156.9 ± 5.5	135.2 ± 7.4	28.5 ± 1.1	29.7 ± 1.5	28.5 ± 1.4	6.3 ± 0.2	5.7 ± 0.1	6.6 ± 0.2

Table 1: 10-fold cross validation results

with 2 ties. (c) On tree size, C4.5(R8) performs better than Greedy in general, and on error rate, about half-half. (d) On testing depth, both Dynamic and Greedy win over C4.5(R8) for all 15 datasets. Decision trees produced by Dynamic and Greedy usually are not as deep as those produced by C4.5(R8); they tend to have more "parallel" branches at a node, instead of "nested" ones into the tree. After comparing actual trees, we feel that "parallel" branches are easier to understand than "nested" ones. In running time, Dynamic is slowest and the other two are comparable. In this regard, the hybrid algorithm could be more promising to offer both quality trees and fast speed. In conclusion, the proposed multi-way splitting of continuous values shows some advantages over C4.5(R8). We believe that for many application domains a multi-way splitting coupled with a careful control on over-splitting is a powerful technique for handling continuous values.

References

[Auer *et al.*, 1995] P. Auer, R.C. Holte, W. Maass. Theory and Application of Agnostic Pac-Learning with Small Decision Trees. In *12th International Conference on Machine Learning.*

[Breiman *et al.*, 1984] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Belmont, CA: Wadsworth International Group, 1984.

[Catlett, 1991] J. Catlett. On Changing Continuous Attributes into Ordered Discrete Attributes. In *the European Working Session on Learning*, Springer Verlag, 164-178

[Chiu *et al.*, 1990] D.K.Y. Chiu, B. Cheung, and A.K.C. Wong. Information Synthesis Based on Hierarchical Maximum Entropy Discretization. *Journal of Experimental and Theoretical Artificial Intelligence*, 2(1990), 117-129

[Dougherty *et al.*, 1995] J. Dougherty, R. Kohavi, M. Sahami. Supervised and Unsupervised Discretization of Continuous Features. In *the 12th International Conference on Machine Learning.*

[Elmasri and Navathe, 1994] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Second Edition, The Benjamin/Cummings Publishing Company, Inc.

[Fayyad and Irani, 1993] U.M. Fayyad and K.B. Irani. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. In *13th International Joint Conference on Artificial Intelligence*, 1022-1027.

[Fulton *et al.*, 1995] T. Fulton, S. Kasif, and S. Salzberg. Efficient Algorithms for Finding Multi-Way Splits for Decision Trees. *Machine Learning.*

[Holte, 1993] R.C. Holte. Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. *Machine Learning*, 11, 63-91.

[Kerber, 1992] R. Kerber. ChiMerge: Discretization of Numeric Attributes. In *Ninth National Conference on Artificial Intelligence*, 123-128.

[Murphy and Aha, 1994] P. Murphy and D. Aha. Uci Repository of Machine Learning Databases. http://www.ics.uci.edu/ mlearn/MLRepository.html

[Quinlan, 1993] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Los Altos, CA: Morgan Kaufmann.

[Quinlan, 1996] J.R. Quinlan. Improved Use of Continuous Attributes in C4.5. In *Journal of Artificial Intelligence Research 4*, 77-90

[Richeldi and Rossotto, 1995]
M. Richeldi and M. Rossotto. Class-driven Statistical Discretization of Continuous Attributes. In *Proc. of European Conference on Machine Learning.* Lecture Notes in Artificial Intelligence 914, Springer Verlag, 335-338