Efficient C4.5

SALVATORE RUGGIERI
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa Italy
email: ruggieri@di.unipi.it
http://www-kdd.di.unipi.it

Abstract

We present an analytic evaluation of the run-time behavior of the C4.5 algorithm which highlights some efficiency improvements. Based on the analytic evaluation, we have implemented a more efficient version of the algorithm, called EC4.5. It improves on C4.5 by adopting the best among three strategies for computing information gain of continuous attributes. All the strategies adopt a binary search of the threshold in the whole training set starting from the local threshold computed at a node. The first strategy computes the local threshold using the algorithm of C4.5, which in particular sorts cases by means of the *quicksort* method. The second strategy also uses the algorithm of C4.5, but adopts a *counting sort* method. The third strategy calculates the local threshold using a main-memory version of the RainForest algorithm, which does not need sorting. Our implementation computes the same decision trees as C4.5 with a performance gain of up to 5 times.

Keywords. C4.5, Decision Trees, Inductive Learning, Supervised Learning, Data Mining.

1 Introduction

Classification algorithms have attracted considerable interest both in the machine learning and in the data mining research areas. Among classification algorithms, the C4.5 system of Quinlan [17, 18] deserves a special mention for several reasons. On the one hand, it represents the result of research in machine learning that traces back to the ID3 system [16]. As such, it has always been taken as the point of reference for the development and analysis of novel proposals. On the other hand, the results of [12] show that the C4.5 tree-induction algorithm provides good classification accuracy and is the fastest among the compared *main-memory* algorithms for machine learning and data mining. It should be mentioned that several external-memory algorithms [1, 8, 9, 14] and parallel implementations [11, 21] have been proposed with the aim of speeding up the execution time and reasoning on very large training sets.

We believe that it is worth developing efficient main-memory versions of C4.5 in addition to existing efficient external-memory and parallel algorithms. On the one hand, due to the increasing sizes of main memories, the applicability of main-memory algorithms is becoming wider and wider. On the other hand, smart external memory implementations [8, 9] switch to main-memory as soon as possible. In this paper, we present an analytic evaluation of the run-time behavior of the C4.5 tree construction algorithm (in the last release, which is Release 8), showing several efficiency limitations. We present a revised version of C4.5, called EC4.5, that improves on C4.5 by choosing the best among three strategies at each information gain computation of continuous attributes. All the strategies adopt a binary search of the threshold in the whole training set starting from the local threshold computed at a node. The first strategy computes the local threshold using the algorithm of C4.5, which in particular sorts cases by means of the *quicksort* method. The second strategy also uses the algorithm of C4.5, but adopts a *counting sort* method. The third strategy calculates the local threshold using a main-memory version of the RainForest algorithm, which does not need sorting. The selection of the strategy to adopt is performed accordingly to an analytic comparison of their efficiency. Our implementation computes the same decision trees as C4.5 with a performance gain of up to 5 times.

2 The C4.5 Tree-Construction Algorithm

2.1 Description

The algorithm constructs a decision tree starting from a *training set* $\mathcal{TS}$, which is a set of *cases*, or *tuples* in the database terminology. Each case specifies values for a collection of *attributes* and for a *class*. Each attribute may have either *discrete* or *continuous* values. Moreover, the special value *unknown* is allowed, to denote unspecified values. The class may have only *discrete* values. We denote with $C_1, \ldots, C_{NClass}$ the values of the class.

Decision trees. A *decision tree* is a tree data structure consisting of *decision nodes* and *leaves*. A leaf specifies a class value. A decision node specifies a *test* over one of the attributes, which is called the attribute *selected* at the node. For each possible outcome of the test, a child node is present. In particular, the test on a discrete attribute A has h possible outcomes $A = d_1$, $\ldots$, $A = d_h$, where $d_1, \ldots d_h$ are the known values for attribute A. The test on a continuous attribute has two possible outcomes, $A \leq t$ and $A > t$, where t is a value determined at the node, and called the *threshold*.

A decision tree is used to *classify* a case, i.e. to assign a class value to a case depending on the values of the attributes of the case. In fact, a path from the root to a leaf of the decision tree can be followed based on the attribute values of the case. The class specified at the leaf is the class *predicted* by the decision tree. A performance measure of a decision tree over a set of cases is called *classification error*. It is defined as the percentage of *mis-classified* cases, i.e. of cases whose predicted classes differ from the actual classes.

The Tree-Construction Algorithm. The C4.5 algorithm constructs the decision tree with a *divide and conquer* strategy. In C4.5, each node in a tree is *associated* with a set of cases. Also, cases are assigned *weights* to take into account unknown attribute values. At the beginning, only the root is present, with associated the whole training set $\mathcal{TS}$ and with all case weights equal to 1.0. At each node the following *divide and conquer* algorithm (see Program 1) is executed, trying to exploit the locally best choice, with no backtracking allowed.

Let T be the set of cases associated at the node. The weighted frequency $freq(C_i, T)$ is computed (step (1)) of cases in T whose class is C_i, for $i \in [1, NClass]$.

If all cases (step (2)) in T belong to a same class C_j (or the number of cases in T is less than a certain value) then the node is a leaf, with associated class C_j (resp., the most frequent class). The classification error of the leaf is the weighted sum of the cases in T whose class is not C_j (resp., the most frequent class).

If T contains cases belonging to two or more classes (step (3)), then the *information gain* of each attribute is calculated. For discrete attributes, the information gain is relative to the splitting of cases in T into sets with distinct attribute values. For continuous attributes, the information gain is relative to the splitting of T into two subsets, namely cases with attribute value *not greater than* and cases with attribute value *greater than* a certain *local threshold*, which is determined during information gain calculation.

The attribute with the highest information gain (step (4)) is selected for the test at the node. Moreover, in case a continuous attribute is selected, the *threshold* is computed (step (5)) as the greatest value of the *whole* training set that is below the local threshold.

A decision node has s children if $T_1, \ldots, T_s$ are the sets of the splitting produced by the test on the selected attribute (step (6)). Obviously, $s = 2$ when the selected attribute is continuous, and $s = h$ for discrete attributes with h known values.

For $i = [1, s]$, if T_i is empty, (step (7)) the child node is directly set to be a leaf, with associated class the most frequent class at the parent node and classification error 0.

If T_i is not empty, the *divide and conquer* approach consists of recursively applying the same operations (step (8)) on the set consisting of T_i plus those cases in T with unknown value of the selected attribute. Note that cases with unknown value of the selected attribute are replicated in each child with their weights proportional to the proportion of cases in T_i over cases in T with known value of the selected attribute.

Finally, the classification error (step (9)) of the node is calculated as the sum of the errors of the child nodes. If the result is greater than the error of classifying all cases in T as belonging to the most frequent class in T, then the node is set to be a leaf, and all sub-trees are removed.

Information Gain. The information gain of an attribute a for a set of cases T is calculated as follows. If a is discrete, and $T_1, \ldots, T_s$ are the subsets of T consisting of cases with distinct known value for attribute

```
FormTree(T)
(1)  ComputeClassFrequency(T);
(2)  if OneClass or  FewCases
        return a leaf;
     create a decision node N;
(3)  ForEach Attribute A
        ComputeGain(A);
(4)  N.test =  AttributeWithBestGain;
(5)  if N.test is continuous
        find Threshold;
(6)  ForEach T' in the splitting of T
(7)     if  T' is Empty
           Child of N is a leaf
        else
(8)        Child of N = FormTree( T' );
(9)  ComputeErrors of N;
     return N
```

Program 1: *Pseudo-code of the C4.5 Tree-Construction Algorithm*

a, then:

$$gain = info(T) - \sum_{i=1}^{s} \frac{|T_i|}{|T|} \times info(T_i).$$

where

$$info(T) = - \sum_{j=1}^{NClass} \frac{freq(C_j, T)}{|T|} \times log_2(\frac{freq(C_j, T)}{|T|})$$

is the entropy function. While having an option to select information gain, by default, however, C4.5 considers the *information gain ratio* of the splitting $T_1, \ldots, T_s$, which is the ratio of information gain to its split information:

$$Split(T) = - \sum_{i=1}^{s} \frac{|T_i|}{|T|} \times log_2(\frac{|T_i|}{|T|})$$

It is easy to see that if a discrete attribute has been selected at an ancestor node, then its gain and gain ratio are zero. Thus, C4.5 does not even computes the information gain of those attributes. If a is a continuous attribute, cases in T with known attribute value are first ordered, using a Quicksort ordering algorithm. Assume that the ordered values are $v_1, \ldots, v_m$. Consider for $i \in [1, m-1]$ the value $v = (v_i + v_{i+1})/2$ and the splitting:

$$T_1^v = \{v_j | v_j \leq v\} \quad T_2^v = \{v_j | v_j > v\}.$$

For each value v, the information gain $gain_v$ is computed by considering the splitting above. The value v' for which $gain_{v'}$ is maximum is set to be the *local threshold* and the information gain for the attribute a is defined as $gain_{v'}$. By default, again, C4.5 considers [18] the information gain ratio[1] of the splitting $T_1^{v'}, T_2^{v'}$. Finally, note that, in case the attribute is selected at the node, the threshold is calculated (step (5)) by means of a *linear* search in the whole training set $\mathcal{TS}$ of the attribute value that best approximates the local threshold v' from below (i.e., which is not greater than v'). Such a value is set to be the threshold at the node.

Simplification and Evaluation. Since constructed decision trees may be large and unreadable, and may suffer from the over fitting problem [17, Chapter 4], the C4.5 system offers a *simplified* tree, obtained by cutting paths according to a given *confidence level*. Both the decision tree and its simplified version are *evaluated* by computing the percentage of cases misclassified by the trees. Also, such an evaluation can be performed on a *test set*, a set of unseen cases during the tree construction.

2.2 Analytic Evaluation of Run-Time Behavior

The tree-construction algorithm is by far the most computational expensive phase of the C4.5 system. In fact, simplification and evaluation simply consist of traversing the decision tree, respectively checking a pruning condition or predicting a class for a case in the training and test sets. Except that for tiny training sets, the time spent for simplification and evaluation is a small fraction of the total execution time. Therefore, we concentrate on tree-construction.

In this section, we analyze the cost of constructing a single node. First, note that, while the use of information gain vs information gain ratio produces different trees with different classification errors, the computational cost of the construction of a single node is the same in both cases. In the rest of the paper, we will always refer to information gain ratio (the default in C4.5). The same reasonings apply to information gain as well.

We denote by T the set of cases associated with a node. Let us introduce some further notation.

$|T|$ cardinality of T, and analogously for any set,

[1] As described in [18], the information gain is always used to find out the local threshold, while the information gain ratio is calculated instead of the information gain only with reference to the splitting at the local threshold.

n_c the number of continuous attributes,

n_d the number of discrete attributes not yet selected at ancestor nodes,

s the number of sets of the splitting of T at the node,

p the probability that the selected attribute is continuous[2].

The construction of a *single* node requires in average the following steps:

$|T|$ to compute class frequencies (step (1)),

$n_d \cdot |T|$ to compute the information gain ratio of discrete attributes (step (3)),

$n_c \cdot |T| \cdot (log|T| + 1)$ to sort cases in $|T|$ and to compute (step (3)) the $gain_v$ values and the information gain ratio (recall that C4.5 adopts a *Quicksort* ordering algorithm),

$p \cdot |\mathcal{TS}|/2$ to find the threshold in the whole training set starting from the local threshold (step (5)),

$(s+1) \cdot |T|$ to arrange cases in memory to compute the subsets associated with child nodes (step (6)).

These give rise to the following number of steps *per case:*

$$n_c \cdot (log|T| + 1) + p \cdot |\mathcal{TS}|/(2 \cdot |T|) + (s + 2 + n_d) \quad (1)$$

Moreover, we provide an estimation for the memory needed by C4.5. For each case, the following information is maintained: attribute values, class value, case weight, info and gain values of continuous attribute splittings at the case, and a pointer to the case. Assuming that `int`, `float` and pointer data types require one memory word, the algorithm requires $|\mathcal{TS}| \cdot (n_a + 5)$ words of memory, where n_a is the number of attributes. In addition, the memory needed to store the decision tree must be considered. Other minor data structures are also maintained, but their size is negligible.

3 From C4.5 to EC4.5

3.1 The EC4.5 System

A serious bottleneck of C4.5 is highlighted by formula (1), where the second term, namely $p \cdot |\mathcal{TS}|/(2 \cdot |T|)$, can be very high, especially for nodes with few cases. The bottleneck is due to the use of a linear search algorithm for the threshold in the whole training set, i.e. step (5) in Program 1. This choice is forced since the cases in the training set may not be ordered with respect to the attribute selected for test. We present an implementation that eliminates the bottleneck using binary search instead of linear search. In addition, the first term in formula (1) can be reduced by using a different sorting algorithm or a different algorithm to calculate gains. Our implementation, called EC4.5, uses the best among three strategies for computing gains and thresholds of continuous attributes (steps (3) and (5) in Program 1).

All the strategies share the following preliminary phase. At the root node, for each continuous attribute a, the whole training set must be sorted[3] to calculate gains. After sorting, the *distinct* and *ordered* attribute values for continuous attribute a are stored in an array $DV[a]$ (DV is an abbreviation for $DistinctValues$). Moreover, the attribute value v of every case is replaced by the index i of DV such that $DV[a][i] = v$. From now on, we call i the index of v for attribute a, and omit the attribute when clear from the context.

All the strategies use DV to perform a *binary* search of thresholds at step (5) in Program 1, so eliminating the C4.5 bottleneck. Even better, at each node, we maintain for each attribute the range $[low, high]$ of indexes that case values belong to. Thus, the binary search is performed only in that range and not in the whole training set.

The three strategies differ in the way they compute information gain (ratio) of continuous attributes. In particular, strategy one and two use different sorting algorithms, while using the same method of C4.5 for calculating information gain. Strategy three, instead, use a method for calculating information gain that does not require sorting at all.

3.2 EC4.5 Strategy One

The first strategy of EC4.5 is the same as the C4.5 algorithm for the tasks of sorting and of finding out the local threshold. The difference between this strategy and C4.5 consists then of using binary search instead of linear search for the task of finding out the threshold starting from the local threshold[4].

[2] Note that, at the root p is $n_c/(n_c + n_d)$, but, since discrete attributes are never selected twice, p increases up to 1 as tree grows (unless there is no continuous attribute).

[3] Actually, the EC4.5 system does not sort attributes that have only integer values. In this case, the third strategy is applicable, as it does not require sorting.

[4] A slight variant of this strategy does not require the array DV. In fact, case partitioning can be done using the local

3.3 EC4.5 Strategy Two

The second strategy of EC4.5 adopts a sorting algorithm that exploits the *divide and conquer* nature of the decision-tree algorithm, and then computes gain as in C4.5. The basic consideration of this strategy relies on the fact that, when a continuous attribute is selected, the range $[low, high]$ of indexes of case values is partitioned among child nodes into $[low, h]$ and $[h+1, high]$, for some h. Therefore, if the range $high - low$ is small, it could be worth to use an algorithm which is in $O(high - low + |T|)$ rather than Quicksort, which is in $O(|T| log|T|)$ in average. Typical instances of this case are attributes such as *age*, *temperature*, *humidity*, *time (years, months, weeks, hours)*. However, even in the general case, as long as the tree grows, case splitting allows us to restrict the range of indexes more and more sharply.

```
CountingSort
        (1) For i = low to high
                Position[i] = 0;
        (2) For i = 1 to n
                Position[A[i]] = Position[A[i]] + 1;
        (3) For i = low+1 to high
                Position[i] = Position[i] + Position[i-1];
        (4) For i = 1 to n
                B[ Position[ A[i] ] ] = A[i];
                Position[ A[i] ] = Position[ A[i] ] - 1;
        (5) For i = 1 to n
                A[i] = B[i];
```

Program 2: *Counting Sort of* $A[1], \ldots, A[n]$

A well-studied algorithm in $O(high - low + |T|)$ is *Counting Sort* (see e.g., [4]), attributed to H. Seward (1954). The algorithm (Program 2) sorts an array A of n naturals belonging to the range $[low, high]$. In the EC4.5 system, such an algorithm is used to sort the indexes of attribute values, and, *a fortiori*, the attribute values of cases. The algorithm uses two auxiliary arrays, *Position* and B. At step (1), *Position* is initialised. At step (2), the number of occurrences of each value are counted. Actually, for each value k in $[low, high]$, we need (step (3)) the number of occurrences of values less or equal than k. At this point, we have that $Position[k-1]+1, \ldots, Position[k]$ are the array positions where to move $A[i]$ if it is equal to k. Thus, in step (4), given the value $A[i]$, we move it in the array B at position $Position[A[i]]$, decrementing $Position[A[i]]$ to hold the new position where to store the next value equal to $A[i]$. At the end, the array B contains the elements of A ordered. So, we simply copy B to A (step (5)). We have implemented an *on-the-fly* version of the Counting Sort algorithm that does not need array B, but only another array of the same type as *Position*. Such an implementation takes $2 \cdot (high - low + 1) + 2 \cdot |T|$ steps, and requires $2 \cdot max\{d_a\}$ words, where d_a denotes the number of distinct values in the training set $\mathcal{TS}$ for the continuous attribute a.

threshold only. In fact, the threshold is needed only in the simplification phase. Thus, after the tree has been constructed, cases can be sorted for each continuous attribute a in turn and then the thresholds can be computed for nodes with selected attribute a. However, since the other strategies adopted by EC4.5 require DV, we prefer to follow a homogeneous presentation.

3.4 EC4.5 Strategy Three

The third strategy that EC4.5 may adopt is a main-memory implementation of the RainForest algorithm [9] to calculate information gain. This strategy does not require sorting at all in the calculation of information gain. Consider a continuous attribute a. Let us recall that $[low,\ high]$ is the range of indexes of attribute a values for cases at a node. First, an array $AVC[c][i]$ is filled in with the weighted sum of cases with class c and attribute value index i, with $i \in [low,\ high]$. Second, the information gain $gain_v$ is computed by considering the splitting at values $v = (DV[a][i] + DV[a][i+1])/2$ for $i \in [low,\ high-1]$. Note, that the AVC array contains all the information needed for calculating information gains. Third, the value v' for which $gain_{v'}$ is maximum is set to be the *local threshold*.

This approach requires the availability of the distinct attribute values of cases, which in EC4.5 is provided by the $DV[a]$ array. The algorithm requires $(high - low + 1) \cdot NClass$ steps to initialize AVC, $|T|$ steps to fill in the AVC array and $(high - low + 1) \cdot NClass$ steps to calculate information gain. Moreover, at most $NClass \cdot max\{d_a\}$ additional words of memory are required to maintain array AVC. However, we will see in Section 3.6 that much less memory is actually required.

3.5 Selecting a Strategy

In the following we denote by d the integer $high - low + 1$, i.e. the range of indexes of attribute values for which information gain has to be calculated.

3.5.1 Strategy One vs Strategy Two

The difference between strategy one and strategy two consists in the sorting algorithm. Theoretically, the Counting Sort algorithm is more efficient than Quicksort when:

$$2 \cdot d + 2 \cdot |T| \leq c \cdot |T| \cdot log(c' \cdot |T|),$$

where c and c' model *multiplicative constants* due to the actual implementations of the algorithms. Called $\alpha = d/|T|$, the fraction of the range of indexes over the number of cases, the inequality above can be rewritten as follows:

$$4^{(\alpha+1)/c}/c' \leq |T|. \tag{2}$$

Unfortunately, we do not know c and c'. We followed an experimental approach. From experimental results, it has been observed improvement of Counting Sort over Quicksort when $\alpha \leq 16$ independently of the number $|T|$. This implies that for $\alpha = 16$, $4^{(\alpha+1)/c}/c'$ is approximatively 1. Consider now $\alpha > 16$. We can divide the inequality (2) by 1 at the right side and by its approximation $4^{(16+1)/c}/c'$ at the left side, thus obtaining:

$$4^{(\alpha-16)/c} \leq |T|.$$

This means that when $\alpha > 16$ we can still fruitfully use Counting Sort, if there are sufficiently many cases. The constant c can now be approximated experimentally, and in the actual implementation it is set to $3/4$.

3.5.2 Strategy Two vs Strategy Three

Strategy three requires $|T| + 2 \cdot d \cdot NClass$ steps to calculate information gain ratio of an attribute. Strategy two requires sorting, which takes $2 \cdot d + 2 \cdot |T|$ steps, and the calculation of information gain ratio as in the C4.5 algorithm, which takes $|T|$ steps. Theoretically, strategy three is better than strategy two when:

$$|T| + 2 \cdot d \cdot NClass \leq c'' \cdot (2 \cdot d + 3 \cdot |T|)$$

where c'' models *multiplicative constants* due to the actual implementations of the algorithms. Again, called $\alpha = d/|T|$, the inequality can be rewritten as follows: $\alpha \leq (3 \cdot c'' - 1)/(2 \cdot NClass - 2 \cdot c'')$. The constant c'' has been approximated experimentally, and in the actual implementation it is set to 1, which gives rise to:

$$\alpha \leq 1/(NClass - 1). \tag{3}$$

3.5.3 Selecting a Strategy

Summarizing, the criterion adopted by EC4.5 in order to select one of the three strategies for computing the information gain ratio of a continuous attribute is the following. According to (2) and (3), when $\alpha \leq 1/(NClass - 1)$, strategy three is selected. Otherwise, if $\alpha \leq 16$ or $4^{(\alpha-16)\cdot 4/3} \leq |T|$, strategy two is selected. In the remaining cases, strategy one is selected.

3.6 Memory Occupation

We recall that d_a denotes the number of distinct attribute values in the training set $\mathcal{TS}$ for the continuous attribute a. Compared to C4.5 we need at most the following additional memory:

$\sum_a d_a$ words to store distinct attribute values, i.e. array DV,

$2 \cdot max\{d_a\}$ words for the arrays used by the Counting Sort algorithm,

$NClass \cdot |\mathcal{TS}|/(NClass - 1)$ words for the AVC array of strategy three. In fact, since strategy three is applied when $\alpha \leq 1/(NClass - 1)$, we have $d \leq |\mathcal{TS}|/(NClass - 1)$, which implies that AVC has at most $|\mathcal{TS}|/(NClass - 1)$ columns.

To mitigate the requirement for additional memory, on the one hand we limit the number of columns of AVC (and hence the applicability of strategy three) to $|\mathcal{TS}|/10$, which experimentally seems to be a good trade-off. On the other hand, we looked for memory savings in the C4.5 data structures, noting that the routine that calculates gain of continuous attributes maintains in memory the info and gain values of splitting at the attribute value of each case. This is not strictly necessary. Summing up, the additional required memory is at most:

$$2 \cdot max_a\{d_a\} + \Sigma_a d_a - 2 \cdot |\mathcal{TS}|$$
$$+ NClass \cdot |\mathcal{TS}|/max\{NClass - 1, 10\}$$

words. At the best (cases with only one attribute, which is continuous and with very few distinct values), this means saving 30% of memory required by C4.5. At worst (all attributes are continuous and with only distinct values), this means 100% additional memory. In the next section, we present some experimental results showing that the additional amount of memory required in practice is a modest fraction.

4 Experimental Results

The relevant characteristics of the training sets used in experiments are reported in Table 1. Each row contains the name of the training set ($\mathcal{TS}$ name), the number of cases ($|\mathcal{TS}|$), the number of class values ($NClass$), the number of discrete attributes, the number of continuous attributes that have a number of distinct attribute values belonging to the intervals $\leq 2^5$, $2^5..2^{10}$, $2^{10}..2^{15}$, and $> 2^{15}$, and finally the total number of attributes. Training sets $1-9$ were taken from the UCI Machine-Learning Repository [3]. Training sets $10-11$ are subsets respectively of the *KDD Cup Competition 1999* and of the *Forest Cover Type* from the UCI KDD Repository [2]. Finally, training sets $12-13$ are synthetic training sets generated by the Quest Generator [15], using function 5.

Table 2 reports the elapsed execution times (Computer: Pentium II 266 Mhz 128 Mb RAM, Operating System: Linux) of the tree-construction phase for C4.5 and EC4.5, the additional fraction of memory required by EC4.5 and the ratio of the time of EC4.5 over C4.5. As one would expect, there is no advantage of EC4.5 over C4.5 for training sets with only discrete attributes ($\mathcal{TS}$ no 2). We note a modest advantage for training sets with few cases ($\mathcal{TS}$ no 3-4). This is due to the fact that EC4.5 has some overhead to do indexing, which is not negligible for small training sets. The advantage of EC4.5 is considerable for training sets with more and more cases ($\mathcal{TS}$ no 5-13). Among those, we observe that EC4.5 performs better when the number of distinct attribute values is small ($\mathcal{TS}$ no 7-8). Finally, note that for non-synthetic training sets the additional memory required by EC4.5 is a small fraction.

Let us now compare in more detail the run-time behavior of EC4.5 and C4.5 on the training set *Adult*. Fig. 1 shows on the Y-axis the average time per case needed to construct a node of X cases. The two plots show ranges for X belonging to [0-50000] and [0-500]. The up-and-down behavior of C4.5 in the [0-500] plot is due to the highly variable time needed by linear search of thresholds in the whole training set. To the end of validating our theoretical analysis, consider now each of the three strategies running separately on *Adult*. Fig. 2 shows the average time per case needed to compute information gain of continuous attributes for cases with α in the range [0-40]. It is worth noting that no strategy is the best for all α. Finally, the column concerning EC4.5 in Table 3 shows, in detail, the efficiency improvements over C4.5 for the various tasks of tree construction. The improvement due to binary search contributes the most, then we have the one due to the use of counting sort, and finally improvement due to the use of the RainForest algorithm.

5 Windowing and Trials

C4.5 offers the technique of *windowing*, i.e. the construction of decision trees for subsets of large training sets. If the resulting tree is not enough accurate to classify the cases out of the *window*, then an enlarged window is considered. The process is iterated until convergence to a final tree. In principle, windowing does not have to consider the whole training set and may produce smaller decision trees. Since the final tree may be different for different initial windows, the windowing approach can be repeated many times, choosing the best tree constructed. This further technique is called the *trials* technique. Since windowing and trials consist of iterating tree construction, the advantages of EC4.5 over C4.5 are preserved, as confirmed by the experiments shown in Table 4.

6 Related Work

A comparison of prediction accuracy, complexity, and training time of thirty-three (main-memory) classification algorithms, including decision trees, rules and neural networks, is presented in [12]. The results show that C4.5 provides good classification accuracy and is the fastest among the compared algorithms. The state of the art of out-of-core algorithms is described in [1, 8, 9], while we refer to [11, 21] for parallel implementations.

Either for main-memory, out-of-core, or parallel implementations, the techniques proposed to overcome the efficiency limitations of C4.5 in handling continuous attributes include pre-sorting of attributes (SLIQ [14], SPRINT [20]), correlation analysis among attributes (SONAR [7]), discretization of continuous attributes in a pre-processing phase ([10]), use of clustering to reduce the set of candidate local thresholds (SPEC [22]), use of more efficient algorithms to compute information gain or information gain ratio (Fayyad and Irani [6], RainForest [9]), estimation of thresholds from a sample of cases (CLOUDS [1]), integration of tree construction and simplification (PUBLIC [19]), and optimistic approaches exploiting statistical technique to validate an initial tree constructed from a subset of the training set (BOAT [8]). The system presented in this paper falls into the techniques

	$\mathcal{TS}$ name	$\|\mathcal{TS}\|$	$NClass$	No. of attributes					
				Disc	Continuous				Tot
					$\leq 2^5$	$2^5..2^{10}$	$2^{10}..2^{15}$	$> 2^{15}$	
1	*Contraceptive*	1473	3	7	1	1			9
2	*Statlog Dna*	2000	3	60					60
3	*St. Image Seg.*	2310	7		3	11	5		19
4	*Thyroid*	3772	3	15	4	2			21
5	*Statlog Satel.*	4435	6			36			36
6	*Musk Clean2*	6598	2	2	5	161			168
7	*Letter*	20000	26		16				16
8	*Adult*	48842	2	8	1	4	1		14
9	*St. Shuttle*	58000	7			9			9
10	*KDD Cup 99*	80000	22	8	13	18	2		41
11	*Forest Cover*	100000	7	44		7	3		54
12	*SyD200*	200000	10	3	1	1	1	3	9
13	*SyD400*	400000	10	3	1	1	1	3	9

Table 1: *Training sets used in experiments*

	$\mathcal{TS}$ name	Tree size	C4.5 Time	EC4.5		EC4.5 /C4.5
				Time	Memory	
1	*Contraceptive*	665	0.149	0.066	-8.9%	0.45
2	*Statlog Dna*	309	0.150	0.150	-2.7%	1.00
3	*St. Image Seg.*	101	1.50	1.21	+28%	0.81
4	*Thyroid*	23	0.151	0.130	-4.7%	0.86
5	*Statlog Satel.*	505	4.66	2.29	-2.6%	0.49
6	*Musk Clean2*	235	15.3	7.92	+3.7%	0.52
7	*Letter*	2675	24.7	4.7	-8.3%	0.20
8	*Adult*	10285	54.2	12.1	-0.2%	0.23
9	*St. Shuttle*	61	17.8	6.1	-13.8%	0.35
10	*KDD Cup 99*	312	134.3	56.5	-1.3%	0.42
11	*Forest Cover*	5905	374.0	115.4	-2.2%	0.31
12	*SyD200*	6084	176.9	73.2	+16.3%	0.42
13	*SyD400*	8438	430.5	151.4	+13.3%	0.35

Table 2: *Tree-construction elapsed times (in secs)*

Task	C4.5	C4.5+FI		EC4.5		EC4.5+FI	
	Time	Time	Ratio	Time	Ratio	Time	Ratio
Sorting	5.14	5.14	1.00	1.74	0.34	1.74	0.34
Gain Computation	8.96	6.25	0.70	7.54	0.84	4.90	0.55
Threshold Search	36.6	36.6	1.00	0.01	0.01	0.01	0.01
Other Tasks	3.57	3.57	1.00	2.81	0.80	2.81	0.80
Total	54.2	51.6	0.95	12.1	0.23	9.46	0.18

Table 3: *Elapsed time (in secs) and ratio (over C4.5) for the tasks of tree construction, for the training set* Adult. *"FI" is an abbreviation for the optimization of Fayyad and Irani described in Section 6.*

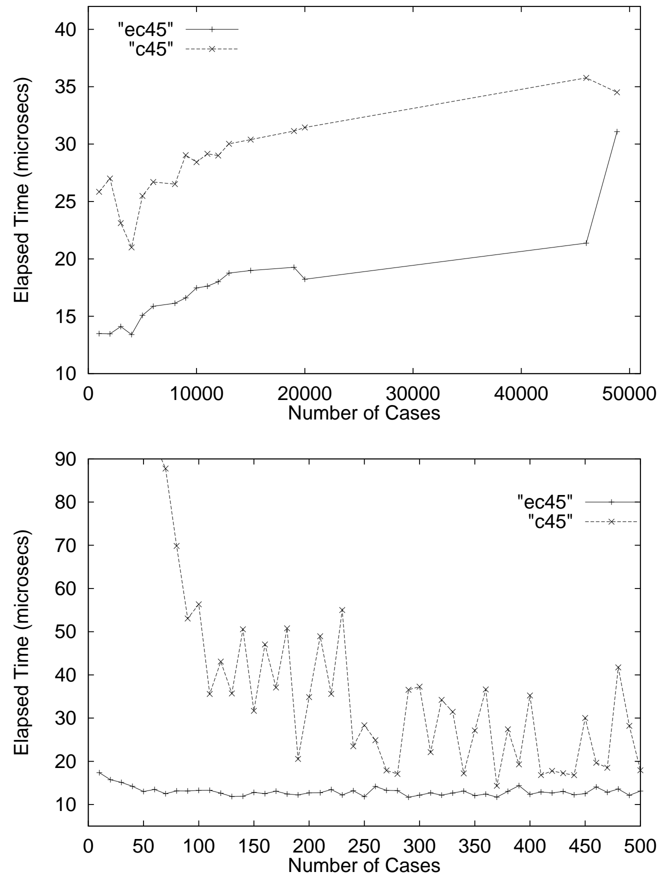

Figure 1: *Average time per case (Y-axis) needed to construct a node of X cases, for the training set* Adult.

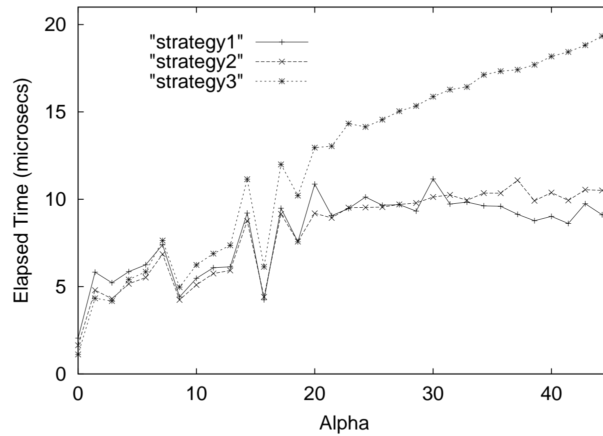

Figure 2: *For the training set* Adult, *the Y-axis reports the average time per case needed to compute information gain of continuous attributes for cases with* α *as in the X-axis.*

	$\mathcal{TS}$ name	C4.5	EC4.5	EC4.5 /C4.5		$\mathcal{TS}$ name	C4.5	EC4.5	EC4.5 /C4.5
1	*Contraceptive*	4.23	1.65	0.39	8	*Adult*	1377	247.3	0.18
2	*Statlog Dna*	3.26	3.26	1.00	9	*St. Shuttle*	16.7	6.5	0.39
3	*St. Image Seg.*	18.5	15.8	0.86	10	*KDD Cup 99*	213	105.7	0.50
4	*Thyroid*	0.66	0.49	0.75	11	*Forest Cover*	7396	1930	0.26
5	*Statlog Satel.*	146.6	70.5	0.48	12	*SyD200*	2769	638	0.23
6	*Musk Clean2*	67.6	32.8	0.49	13	*SyD400*	10213	1542	0.16
7	*Letter*	810	119.9	0.15					

Table 4: *Tree-construction elapsed times (in secs) for 3 trials (training sets 1-10) and 1 trial (training sets 11-13). The experiments were conducted forcing the selection of the initial window to be deterministic (rather than random) in order the compared algorithms to construct the same trees.*

that use new algorithms to compute information gain. We observe that some other techniques could be integrated within the EC4.5 system as alternative strategies, including use of clustering to reduce the set of candidate local thresholds, estimation of thresholds, and integration of tree construction and simplification. Conversely, the EC4.5 strategies could help the optimistic approach of BOAT in the construction of the initial tree. The pre-sorting techniques maintain data structures (constructed after the first sorting of attributes at the root node), that are used to avoid sorting. The integration of pre-sorting with the strategies of EC4.5 must be balanced with memory occupation of those data structures that, when considering main-memory versions of the techniques, require 100% additional memory over C4.5. Moreover, the experiments reported in [14] show an efficiency improvement over IND-C4 (a predecessor of C4.5) that is lower than the improvement of EC4.5 over C4.5.

Finally, the commercial successor of C4.5 is the C5.0 system [13]. In particular, the binary search of the threshold is implemented in C5.0 (Quinlan, personal communication). Although EC4.5 seems more efficient than C5.0, in general the two systems cannot be properly compared. On the one hand, C5.0 includes additional (time consuming) features that are not present in C4.5. On the other hand, C5.0 produces smaller (hence, time saving) decision trees.

While it is out of the scope of this paper to present a complete review of all the mentioned approaches, we would like to present a direct comparison with the approach of Fayyad and Irani [6] (further extended by Elomaa and Rousu [5]). Basically, their approach speeds up finding the local threshold for continuous attributes by considering splittings T_1^v, T_2^v for $v = (v_i + v_{i+1})/2$ only if v_i is a *boundary* value. v_i is a boundary value if there exist two cases at the node with attribute value v_i and with distinct class value, or if all cases with attribute value v_i at the node have the same class which is not the class of all cases with attribute value v_{i+1} at the node. For the *Adult* training set, for instance, only the 50% of values must be considered. However, this approach speeds up only the task of calculating the information gain, not the tasks of sorting or searching thresholds. We have implemented the Fayyad and Irani strategy and combined it with EC4.5. Table 3 shows for the training set *Adult* the detail of the execution times and ratio (over C4.5). The algorithms considered are C4.5, C4.5 together with the approach of Fayyad and Irani, EC4.5, and EC4.5 together with the approach of Fayyad and Irani. Looking at the ratio values, we note that EC4.5 improves over C4.5 in all tasks. Moreover, for the task of computing information gain, the integration of EC4.5 and the approach of Fayyad and Irani results into a faster implementation.

7 Conclusions

We believe that it is worth developing efficient main-memory versions of C4.5. On the one hand, the increasing size of main memories makes its applicability wider. On the other hand, smart external memory implementations switch to main-memory as soon as possible. We have presented an analytical evaluation of the run-time behavior of the C4.5 algorithm which gave rise to a more efficient implementation, with a performance gain of up to 5 times. The main advantage of EC4.5 over C4.5 is the possibility to select the best among three strategies accordingly to an analytic comparison of their efficiency.

Acknowledgments. I am grateful to the anonymous referees for many helpful comments, to P. Becuzzi and

M. Coppola for discussions on the algorithm of C4.5, and to F. Bonchi for several tests on EC4.5.

References

[1] K. Alsabti, S. Ranka, and V. Singh, "CLOUDS: Classification for large out-of-core datasets," *Proc. Int'l Conf. on Knowledge Discovery and Data Mining*, AAAI Press, 1998, pp. 2-8.

[2] S. D. Bay, "UCI KDD Archive," http://kdd.ics.uci.edu, 1999.

[3] E. Keogh C. Blake and C.J. Merz, "UCI repository of machine learning databases," http://www.ics.uci.edu/~mlearn/-MLRepository.html, 1998.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms,* 6th edition, MIT Press and McGraw-Hill Book Company, 1992.

[5] T. Elomaa and J. Rousu, "General and efficient multisplitting of numerical attributes," *Machine Learning*, Vol. 36, No. 3, Sept. 1999, pp 201-244.

[6] U. M. Fayyad and K. B. Irani, "On the handling of continuous-valued attributes in decision tree generation," *Machine Learning*, Vol. 8, 1992, pp. 87-102.

[7] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, "Constructing efficient decision trees by using optimized numeric association rules," *Proc. Int'l Conf. Very Large Data Bases*, Morgan Kaufmann, 1996, pp. 146-155.

[8] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh, "BOAT-optimistic decision tree construction," *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, ACM Press, 1999, pp. 169-180.

[9] J. E. Gehrke, R. Ramakrishnan, and V. Ganti, "RainForest - A framework for fast decision tree construction of large datasets," *Data Mining and Knowledge Discovery*, Vol. 4, No. 2/3, Jul. 2000, pp. 127-162.

[10] S. J. Hong, "Use of contextual information for feature ranking and discretization," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 9, No. 5, Sept./Oct. 1997, pp. 718-730.

[11] M. Joshi, G. Karypis, and V. Kumar, "ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets," *Proc. of 1998 Int'l Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, IEEE Computer Society, 1998, pp. 573-579.

[12] T.S. Lim, W.Y. Loh, and Y.S. Shih, "A comparison of prediction accuracy, complexity, and training time of thirty-tree old and new classification algorithms," *Machine Learning*, Vol. 40, No. 3, Sept. 2000, pp. 203-228.

[13] Rulequest Research Ltd, "C5.0," On-line documentation, http://www.rulequest.com, 1999.

[14] M. Mehta, R. Agrawal, and J. Rissanen, "SLIQ: A fast scalable classifier for data mining," *Proc. of the 1996 Int'l Conference on Extending Database Technology*, Lecture Notes in Computer Science, Vol. 1057, 1996, pp. 18-32.

[15] Quest Group, "Quest synthetic data generation code", On-line documentation, http://www.almaden.ibm.com/cs/quest/syndata.-html, 1999

[16] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, Vol. 1, No. 1, 1996, 81-106.

[17] J. R. Quinlan, *C4.5: Programs for machine learning,* Morgan Kaufmann, San Mateo, CA, 1993.

[18] J. R. Quinlan, "Improved use of continuous attributes in C4.5," *Journal of Artificial Intelligence Research*, Vol. 4, 1996, pp. 77-90.

[19] R. Rastogi and K. Shim, "PUBLIC: A decision tree classifier that integrates building and pruning," *Data Mining and Knowledge Discovery*, Vol. 4 No. 4, Oct. 2000, 315-344.

[20] J. C. Shafer, R. Agrawal, and M. Mehta, "SPRINT: A scalable parallel classifier for data mining," *Proc. Int'l Conf. Very Large Databases*, Morgan Kaufmann, 1996, pp. 544-555.

[21] A. Srivastava, E.-H. (Sam) Han, V. Kumar, and V. Singh, "Parallel formulations of decision-tree classification algorithms," *Data Mining and Knowledge Discovery*, Vol. 3, No. 3, Sept. 1999, pp. 237-261.

[22] A. Srivastava, V. Singh, E. H. Han, and V. Kumar, *An efficient scalable parallel classifier for data mining,* Tech. Report TR-97-010, Department of Computer Science, Univ. of Minnesota, Minneapolis, 1997.