Università di Pisa

Dipartimento di Informatica

Technical Report : TR-00-01

Efficient C4.5

Salvatore Ruggieri

February 02, 2000

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

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. We have implemented a more efficient version of the algorithm, called EC4.5, that improves on C4.5 by adopting the best among three strategies at each node construction. The first strategy uses a *binary search* of thresholds instead of the linear search of C4.5. The second strategy adopts a *counting sort* method instead of the quicksort of C4.5. The third strategy uses a main-memory version of the RainForest algorithm for constructing decision trees. Our implementation computes the same decision trees as C4.5 with a performance gain of up to 5 times.

Keywords. C4.5, Decision Trees Induction Algorithms, Supervised Machine 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 [15, 16] 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 [14]. 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 [10] 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, 6, 7, 12] and parallel implementations [9, 19] 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 dimensions of main memories, the applicability of main-memory algorithms is becoming wider and wider. On the other hand, smart external memory implementations [6, 7] 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 adopting the best among three strategies at each node construction. The first strategy uses a *binary search* of thresholds instead of the linear search of C4.5. The second strategy adopts a *counting sort* method instead of the quicksort used by C4.5. The third strategy uses a main-memory version of the RainForest algorithm [7] for constructing decision trees. Our implementation computes the same decision trees as C4.5 with a performance gain of up to 5 times over C4.5.

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 and does not admit the *unknown* value. We denote with $C_1, \ldots, C_{NClass}$ the values of the class.

Decision trees. The objective of tree-induction algorithms is to construct tree data structures (called *decision trees*) that predict the class of a case in term of the attribute values of the case.

In C4.5, each node in the tree is *associated* with a set of cases. Also, cases are assigned *weights* to take into account unknown attribute values. In particular, the root is associated with the whole training set $\mathcal{TS}$ and all weights are 1.0. Each non-leaf node (or *decision* node) of the tree specifies a *test* over an attribute, which splits the cases into subsets associated with the children of the node. The splitting is repeated down the tree trying to produce leaves that contain only nodes belonging to the same class.

The path from the root to a leaf provides a description, in terms of a conjunct of conditions on attributes, of the class of the cases associated with the leaf. A measure of the accuracy of the description is called *classification error*, defined as the percentage of *misclassified* cases (i.e., of cases whose predicted class differs from the actual class).

The Tree-Construction Algorithm. The C4.5 algorithm constructs the decision tree with a *divide and conquer* strategy. At the beginning, only the root is present. 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 belongs to a same class C_j (resp., 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 number of cases in T whose class is not C_j.

If T contains cases belonging to two or more classes (step (3)), then the *information gain* of each attribute is calculated, as will be described later. The attribute with the highest information gain (step (4)) is selected for test at the node. Tests of discrete attributes consider partitions of T into n subsets, one for each possible value of the attribute. Tests of continuous attributes partition T into two subsets, namely the set

```
FormTree(T)
    (1) ComputeFrequency(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*

of cases with attribute value *not greater than* and the set of cases with attribute value *greater than* a certain *threshold* which has to be determined (step (5)). 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 possible values. In addition cases with undefined value of the selected attribute are replicated in all sets T_i, $i \in [1, s]$, and weights for the cases are computed by the previous weights and the percentage of undefined cases.

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 node and classification error 0.

The *divide and conquer* approach consists of recursively applying the same operations on each non-empty T_i (step (8)). 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 value for attribute a, then:

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

where

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

is the entropy function. It is easy to see that if a discrete attribute has been selected at an ancestor node, then its gain is zero. Thus, C4.5 does not even computes the information gain of those attributes. If a is a continuous attribute, cases in T are ordered with respect to the value of a. 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 information gain for the attribute a is defined as the maximum $gain_v$. However, in case the attribute is selected at the node, the threshold is not just the value v for which $gain_v$ is maximum. Instead, at (step (5)), a search in the whole training set $\mathcal{TS}$ is performed in order to find the attribute value t that best approximates v from below (i.e., which is not greater than v). Such a value t is set to be the threshold.

Simplification and Evaluation. Since constructed decision trees may be large and unreadable, 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 tree(s). Also, such an evaluation can be performed on a *test set*, a set of cases distinct from the training set.

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

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

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[1].

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 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 (the C4.5 system adopts a *Quicksort* ordering algorithm),

[1]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 (unless there is no continuous attribute) as tree grows.

$p \cdot |\mathcal{TS}|/2$ to find the threshold in the whole training set when the selected attribute is continuous (step (5)),

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

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

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

Also, we provide an estimation for the memory needed by C4.5. For each case, the following information is maintained: the attribute and class values, the weight, the info and gain of splitting a continuous attribute at the attribute value of the case. Assuming that `int`, `float` and pointers data types require one word memory, the algorithm requires $|\mathcal{TS}| \cdot (n_a + 3)$ words of memory, where n_a is the number of attributes. In addition, the memory needed to store the tree must be considered. Some 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 the formula (1), where the second addend, 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 the respect to the attribute selected for test. We present an implementation that eliminates the bottleneck using a binary search instead of the linear search. In addition, the first addend in formula (1) can be reduced by using a different sorting algorithm. Our implementation, called EC4.5, uses the best among three strategies for computing gains of continuous attributes (step (3) 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[2] 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.

3.2 EC4.5 Strategy One

The first strategy of EC4.5 consists of using the C4.5 algorithm for sorting cases and computing gain. The difference between this strategy and C4.5 consists then of using a binary search instead of a linear search. The array DV can be used for binary search of thresholds, so eliminating the C4.5 bottleneck. Even better, 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[3].

[2] Actually, the EC4.5 system do not sort attributes that have only integer values. In this case, one of the strategies (the third) that will be presented is applicable, and it does not require sorting.

[3] A slight variant of this strategy does not require the array DV. In fact, case partitioning can be done without searching the actual threshold, that is needed only in the simplification phase. Thus,

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 compute 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 is partitioned into $[low, h]$ and $[h+1, high]$ among child nodes. Therefore, 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 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: *The Counting Sort Algorithm*

A well-studied algorithm in $O(high - low + |T|)$ is the *Counting Sort* algorithm (see e.g., [4]), attributed to H. Seward (1954). The algorithm (Program 2) sorts an array A of 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 $Position[k-1]+1, \dots, Position[k]$ 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 $2 \cdot max\{\tilde{d}_a\}$ words, where $\tilde{d}_a$ denotes the number of distinct values in the training set $\mathcal{TS}$ for the continuous attribute a.

after the tree has been constructed, cases can be sorted for each continuous attribute a in turn and then the actual 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 [7], which does not require sorting. The basic idea is to calculate information gain by first filling in an array $AVC[c][i]$ the sum of weights of cases with attribute value index i and class c, and then computing the maximum information gain of splitting at $v = DV[a][low], DV[a][low+1], \ldots, DV[a][high]$ ($[low, high]$ is the range of indexes for attribute values of cases at the node). The AVC array contains all the information needed for calculating information gains. 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 $NClass \cdot max\{\tilde{d}_a\}$ 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 disequation 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 disequation (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 actual implementation 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 of an attribute. Strategy two requires sorting, which takes $2 \cdot d + 2 \cdot |T|$ steps, and the calculation of information gain 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 disequation 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 actual implementation is set to 1, which gives rise to:

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

3.5.3 Selecting a Strategy

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 $\tilde{d}_a$ 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:

$\sum_a \tilde{d}_a$ words to store distinct attribute values, i.e. array DV

$2 \cdot max\{\tilde{d}_a\}$ 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 a good trade-off. On the other hand, we looked for memory saving 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:

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

words. At the best (cases with only one attribute, which is continuous and with very few distinct values), this means saving 50% 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.

				No. of attributes					
				Disc	Continuous				Tot
	$\mathcal{TS}$ name	$\lvert\mathcal{TS}\rvert$	$NClass$		$\leq 2^5$	$2^5..2^{10}$	$2^{10}..2^{15}$	$> 2^{15}$	
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*

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 subset 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 [13], 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 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 a training set. Fig. 1 shows on the Y-axis the average time per case needed to construct a node of X cases, for the training set *Adult*. The two plots show ranges for X belonging to [0-50000] and [0-500]. The up-and-down behavior of C4.5 in the right hand side plot is due to the highly variable time needed by linear search. Finally, Fig. 2 shows the average time per case needed by each of the three strategies of EC4.5 depending on α.

	$\mathcal{TS}$ name	Tree size	C4.5 Time	EC4.5 Time	EC4.5 Memory	EC4.5 /C4.5
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 execution times (in secs)*

	$\mathcal{TS}$ 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 3: *Tree-construction execution 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.*

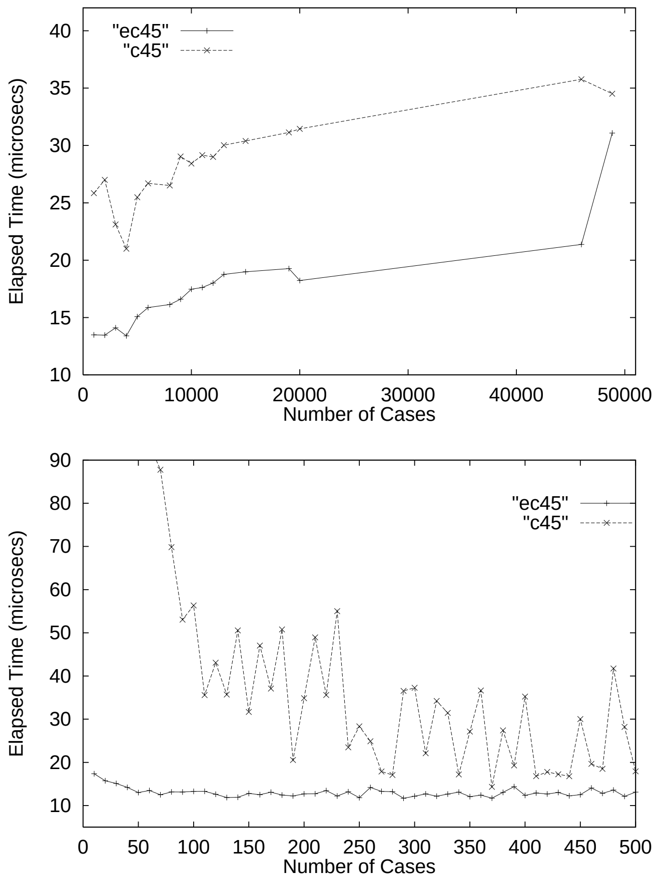

Figure 1: *Average time per case (Y-axis) needed to construct a node of X cases, for the training set* Adult.

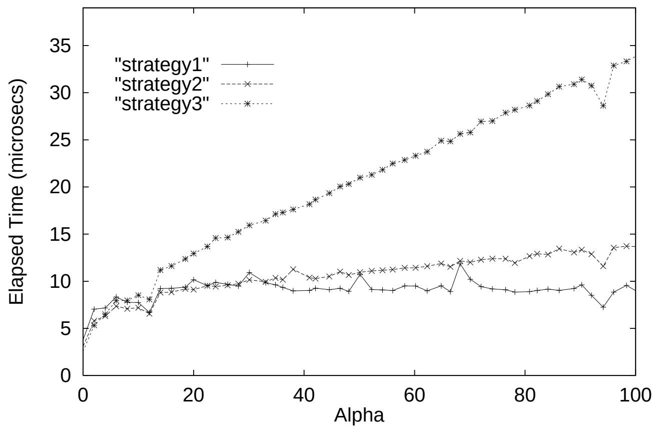

Figure 2: *Average time per case (Y-axis) needed to compute information gain depending on* α *(X-axis), for the training set* Adult.

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 has not 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 3.

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 [10]. 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, 6, 7], while we refer to [9, 19] 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 [12], SPRINT [18]), correlation analysis among attributes (SONAR [5]), discretization of continuous attributes in a pre-processing phase ([8]), discretization at run-time (SPEC [20]), use of new algorithms to compute information gain (RainForest [7]), estimation of thresholds from a sample of cases (CLOUDS [1]), integration of tree construction and simplification (PUBLIC [17]), and optimistic approaches exploiting statistical technique to validate an initial tree constructed from a subset of the training set (BOAT [6]). The system presented in this paper falls into the techniques 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 discretization at run-time, 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 [12] 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 [11]. 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.

7 Conclusions

We believe that it is worth developing efficient main-memory versions of C4.5. On the one hand, the increasing dimensions of main memories make 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 P. Becuzzi and M. Coppola for many 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". In *Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining (KDD98)*, pages 2–8, 1998.

[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.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[5] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. "Constructing efficient decision trees by using optimized numeric association rules". In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 146–155, 1996.

[6] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh. "BOAT-optimistic decision tree construction". In *Proc. of the ACM SIGMOD International Conference on Management of Data*, volume 28,2 of *SIGMOD Record*, pages 169–180. ACM Press, 1999.

[7] J. E. Gehrke, R. Ramakrishnan, and V. Ganti. "RainForest — A framework for fast decision tree construction of large datasets". In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 416–427, 24–27 August 1998.

[8] S. J. Hong. "Use of contextual information for feature ranking and discretization". *IEEE Transactions on Knowledge and Data Engineering*, 9(5):718–730, 1997.

[9] M. Joshi, G. Karypis, and V. Kumar. "ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets". In *Proc. of 1998 International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 573–579. IEEE Computer Society, 1998.

[10] 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 Journal*, 1999. To appear, http://www.Recursive-Partitioning.com/datasets.html.

[11] Rulequest Research Ltd. C5.0, 1998. On-line documentation, http://www.rulequest.com.

[12] M. Mehta, R. Agrawal, and J. Rissanen. "SLIQ: A fast scalable classifier for data mining". In *Proceedings of the 1996 International Conference on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, 1996.

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

[14] J. R. Quinlan. "Induction of decision trees". *Machine Learning*, 1(1):81–106, 1986.

[15] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[16] J. R. Quinlan. "Improved Use of Continuous Attributes in C4.5". *Journal of Artificial Intelligence*, 4:77–90, 1996.

[17] R. Rastogi and K. Shim. "PUBLIC: A decision tree classifier that integrates building and pruning". In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 404–415, 1998.

[18] J. C. Shafer, R. Agrawal, and M. Mehta. "SPRINT: A scalable parallel classifier for data mining". In *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 544–555, 1996.

[19] A. Srivastava, E.-H. (Sam) Han, V. Kumar, and V. Singh. "Parallel formulations of decision-tree classification algorithms". *Data Mining and Knowledge Discovery*, 3(3), 1999.

[20] A. Srivastava, V. Singh, E. H. Han, and V. Kumar. "An Efficient Scalable Parallel Classifier for Data Mining". Technical Report TR-97-010, Department of Computer Science, University of Minnesota, Minneapolis, 1997. http://www.cs.umn.edu/~kumar.