Extremely randomized trees

Pierre Geurts

Technical report
June 2003

University of Liège
Department of Electrical Engineering and Computer Science
Institut Montefiore, Sart Tilman B28
B-4000 Liège Belgium
p.geurts@ulg.ac.be

Abstract. This paper presents a new learning algorithm based on decision tree ensembles. In opposition to the classical decision tree induction method, the trees of the ensemble are built by selecting the tests during their induction fully at random. This extreme randomization makes the construction of the ensemble very fast even on very large datasets with high dimensionality. The method is validated on several classification problems. On several problems, this method is as fast as the classical decision tree induction method but at the same time much more accurate. It is also much faster and more accurate than decision tree bagging. All in all, the proposed algorithm is a good generic method especially interesting to handle very large datasets.

1 Introduction

Recently, one the most active research field in machine learning is the study of methods based on an ensemble of models [11]. Ensemble methods consists in growing several models with a classical machine learning algorithm. Then, the predictions of these models are aggregated to provide a final prediction potentially better than individual ones. Ensemble methods essentially differ in the way the different models are produced from the same learning sample and in the way the individual predictions are aggregated.

One of the most popular family of ensemble methods is defined by Perturb and Combine methods. These methods consists in perturbing the learning algorithm and/or the learning sample so as to produce different models from the same learning sample. The predictions of these models are then aggregated by a simple average or a majority vote in the case of classification. The most popular representative of this family is certainly bagging [4] where the different models are produced by drawing a bootstrap replicate of the learning sample before each induction step. These methods are very successful mainly because of the improvement of accuracy they bring to traditional machine learning methods, like decision trees or neural networks. Nevertheless, the induction of a set of models requires much computational effort during the induction stage.

When applied to decision trees, most ensemble methods actually consists in perturbing the algorithm responsible for the search of the optimal test during tree growing. In the context of an ensemble method, it is thus necessary to somewhat deteriorate the classical deterministic induction algorithm to produce the different trees. The algorithm proposed in this paper exploits this characteristic of perturb and combine algorithms to improve the accuracy and the computational efficiency of decision trees. These motivations are especially important in the case of very large datasets where decision trees do not provide the expected accuracy and the computing time of ensemble methods like bagging remains quite costly.

This paper is structured as follows. In Section 2, we give a general description of what we call the Perturb and Combine (P&C) paradigm. Several instances of this paradigm proposed in the literature are then enumerated. After a brief summary of the classical TDIDT algorithm, Section 3.2 describes some P&C algorithms specifically designed for decision trees. A detailed description of our proposal is given in Section 4. In Section 5, this method is compared to single decision trees, bagging, and random forests (another recent P&C algorithms for decision trees) on several classification problems.

2 Perturb and Combine algorithms

Several ensemble methods which have been proposed in the literature satisfy to the perturb and combine (P&C) paradigm [5, 7] which proceeds as follows (from a learning sample ls and a given learning algorithm):

- **Learning stage:** For i going from 1 to T:
  - perturb the learning algorithm (e.g. by perturbing its data or settings of its parameters) with a set of random parameters $\underline{\epsilon}^i$ drawn independently from a distribution $P(\underline{\varepsilon}|ls)$
  - apply the perturbed algorithm to get a model $f_{ls,\underline{\epsilon}^i}$
- **Test stage:** To make a prediction at a point $\underline{a}$, build the aggregated prediction f_{PC} defined by:

$$f_{PC}(\underline{a}) = \mathrm{aggr}_{i=1}^{T}\{f_{ls,\underline{\epsilon}^i}(\underline{a})\}, \tag{1}$$

  where "aggr" consists either in taking the majority class among the predictions given by the T models from the ensemble or in averaging the class conditional probability estimates given by these models (of course, if models give these estimates) and outputting the most probable class according to the resulting conditional distribution.

Bagging belongs to this family of methods. In this case, the random parameter $\underline{\epsilon}^i$ describes a bootstrap sample from the initial learning sample. Breiman has proposed in [6] to perturb the learning algorithm by randomizing the output classes of the learning cases. Another P&C method has been proposed by Ho [17]. It consists in selecting a random subset of the input attributes before building each model of the ensemble. While all these methods are independent of any particular learning algorithm, there also exist several variants of P&C methods that directly randomize the learning algorithm and leave the data unchanged. The next section will be devoted to variants of decision trees.

3 Decision trees and P&C methods

The decision tree induction method [9] is a very popular machine learning algorithm. It is among the most efficient learning algorithms and it further provides interpretable results by means of logical classification rules. Nevertheless, decision trees suffer from a very high variance [15] which prevent them from competing with other algorithms from the point of view of accuracy. This high variance and their computational efficiency have made of this method the method of choice for the design of P&C algorithms. Indeed, because of the high variance, the improvement of accuracy with ensemble methods is often very impressive. On the other hand, the construction of several trees is not too limitative because of the computational efficiency. For this reason, we find in the literature a great number of P&C algorithms dedicated to decision trees. Before describing some of them, we briefly recall the principles behind top-down induction algorithms for decision trees (For more details, see for example [9] or [19]).

3.1 Top-down induction of decision trees

The top-down induction algorithm of decision trees consists in recursively partitioning the learning sample into subsamples where the output class is constant, with (usually binary) tests based on attribute values. To this end, a score measure is defined that evaluates the ability of a test to separate the learning sample into subsamples as pure as possible in terms of the output classification. The algorithm starts with the whole set of learning cases. For each input attribute, it searches for the test which maximizes the score. Then, globally, the test which has the highest score is selected to separate the learning sample and the same procedure is repeated to split the subsamples corresponding to both issues of the test. The algorithm stops when the subsamples are deemed pure enough in terms of the output classification.

Tests on numerical attributes are generally binary and of the form $[x_i < x_{th}]$ which compares the value of an attribute x_i to a threshold x_{th}, also called a discretization threshold. The algorithm which searches for the best test for an attribute x_i thus consists in determining the value of the discretization threshold which maximizes the score. Generally, score measures are sensible only to the proportion of objects in each class and thus, the search for the best threshold is carried out among at most N possible values for a subsample of size N and requires to sort this subsample according to the attribute.

3.2 Variants of P&C methods for decision trees

Several ensemble methods have been proposed which directly perturb this algorithm to generate different models from the same learning sample. For example, Ali and Pazzani [1] perturb the algorithm by replacing the choice of the best test by the choice of a test at random among the best ones. If S^* is the score of the optimal test, a test is randomly selected, with a probability proportional to its score, among the tests which have a score greater than $S^* - \beta.S^*$, where β is some constant between 0 and 1. Dietterich [12] proposes a very similar approach which consists in randomly selecting a test among the 20 best splits. Ho [17] builds decision trees by locally selecting the best test among only a random subset of all the input attributes. Breiman [7] combines this latter idea with bagging in his "random forests" algorithm. All these methods give at least competitive results with bagging but often are significantly better than this latter method.

4 Extremely randomized trees

A common characteristic of the ensemble methods described in the previous section is that they all consists in choosing a test during the induction which is not anymore the optimal test, i.e. the one that maximizes the score measure. One conclusion which can be drawn from the good behaviour of these algorithms is thus the fact that it is possible to deteriorate to some extent the classical algorithm of Section 3.1 which looks at each step for the best test and still improve accuracy by aggregating the predictions of several of these suboptimal trees.

From this analysis, the idea of the algorithm proposed here is to go one step further than existing algorithms in terms of perturbation in order (1) to better improve accuracy by reducing even more the variance and (2) to take advantage of this degree of freedom with respect to the classical induction algorithm to improve computational efficiency.

A detailed description of the proposed algorithm is given in the next subsection. Its computational complexity is discussed in Subsection 4.2.

4.1 Proposed algorithm

Although the P&C algorithms presented in Section 3.2 are quite different from the original TDIDT algorithm, they still require the determination of the best tests for every attributes (in Ali and Pazzani's and Dietterich's algorithms) or at best for a fixed number of them (in Ho's and Breiman's algorithms). Nevertheless, it is not clear that this optimization is really necessary in the context of ensemble methods. This question is especially justified since several authors (see for example [13] or [15]) have observed the very high variance of the parameters that define decision tree tests with (even small) variations of the learning sample. So, this high variance somewhat jeopardizes the need for an optimization.

Here, we propose to go one step further than existing P&C algorithms by randomizing both discretization thresholds and attribute choices. Our random tree induction algorithm is described in Table 1. The trees that are grown by this algorithm are called extra-trees for

Table 1. Extra-tree growing algorithm

To build an extra-tree from a learning sample LS:

- Create a node $\mathcal{N}$ and attach to this node the whole learning sample $ls(\mathcal{N}) = LS$;
- Set $L = \{\mathcal{N}\}$, the list of open nodes;
- While L is not empty:
  1. Select and remove a node $\mathcal{N}$ from L; Set L_a to the list of all input attributes.
  2. If all the elements of the set $ls(\mathcal{N})$ are of the same class C or if all attributes are constant in $ls(\mathcal{N})$, then attach a prediction to the node $\mathcal{N}$ which becomes a terminal node of the tree;
  3. Otherwise, select an attribute A_i at random from L_a and remove it from L_a.
  4. Select a threshold a_{th} at random for this attribute.
  5. If the score of the test $[A_i < a_{th}]$ evaluated on $ls(\mathcal{N})$ is lower than the threshold s_{th} and if the list L_a is not empty, return to step 3;
  6. Otherwise, denote by $[A_* < a^*_{th}]$ the best test among the tests which have been evaluated so far and compute the subsamples $ls_l = \{(\underline{a}, c) \in ls(\mathcal{N}) | a_* < a^*_{th}\}$ and $ls_r = \{(\underline{a}, c) \in ls(\mathcal{N}) | a_* \geq a^*_{th}\}$;
  7. Create two nodes $\mathcal{N}_l$ and $\mathcal{N}_r$, insert them in L, and set $ls(\mathcal{N}_l) = ls_l$ and $ls(\mathcal{N}_r) = ls_r$;

Two variants to choose a threshold (step 4):

1. Compute the empirical mean, $\overline{A_i}(ls(\mathcal{N}))$, and standard deviation, $\sigma_{A_i}(ls(\mathcal{N}))$, of this attribute in the sample $ls(\mathcal{N})$ and randomly draw the threshold a_{th} according to the distribution $N(\overline{A_i}(ls(\mathcal{N})), \sigma_{A_i}(ls(\mathcal{N})))$.
2. Randomly draw one case $(\underline{a}, c)$ from $ls(\mathcal{N})$ and take $a_{th} = a_i$.

extremely randomized trees. When developing a node, an attribute and a discretization threshold are selected at random. Nevertheless, to avoid very bad test choice, the attribute choice is rejected if the score of the resulting test is lower than a given user-defined threshold, s_{th}.

To randomly choose a discretization threshold, two variants have been tested that are described in the lower part of Table 1. The first one consists in drawing the threshold at random from a Gaussian distribution whose mean and standard deviation are given by the corresponding empirical values estimated from the local learning subsample. One drawback of this variant is that it is necessary to compute the mean and the standard deviation of the attribute at each step of the induction. The second variant avoid this step by using as the discretization threshold the value of the attribute for one object randomly drawn from the local learning subsample.

With respect to existing random tree algorithms discussed in Section 3.2, the main originality of our extra-trees is the combination of the randomization of attribute and discretization threshold and the fact that the discretization threshold is really selected at random, i.e. not on the basis of any score (except for the filtering of very bad tests). So, the algorithm of Table 1 is not far from a totally random algorithm which would build a tree structure without even looking at the learning sample, and use this sample only to determine which nodes are terminal and what labels to attach to them.

Although this algorithm seems naive at first sight, it nevertheless builds trees that perfectly classifies instances from the learning sample (assuming that the attributes allow a perfect partitioning of the learning sample). Furthermore, the dependence of the tests to the learning sample should be very much reduced with respect to the classical induction algorithm which chooses the optimal test. The variance introduced by the random choices should be canceled out by the fact that several models are aggregated. So, globally, we expect an important improvement of accuracy at least with respect to one single classical tree.

4.2 Complexity

In the first variant, the determination of a random test to split a subsample of size N requires the computation of the mean and standard deviation for one randomly drawn attribute and thus demands order N operations. In the second variant, the determination of the random test requires order 1 operations since it is sufficient to choose one object randomly from the subsample. If the score threshold is s_{th} is zero, the first random split is always accepted and the algorithm complexity is independent of the number of attributes. However, when the score threshold becomes positive, several attributes may be considered to find a good enough split and so the complexity will somehow depend on the number of attributes. When a test is found, order N operations are necessary to split the sample in two parts (whatever the variant) and this step must be repeated at each node of the tree. On average, the depth of a tree built from a learning sample of size N is of order $\log(N)$ and, to increase the depth by one level, we must consider subsamples which total size is always N. So, if $s_{th} = 0$, growing one extra-tree requires order $N.\log(N)$ operations and, when $s_{th} > 0$, the complexity will be $M.N.\log(N)$, where M is the number of attributes. Assuming that the two variants build trees of the same complexity, the number of operations will be smaller in the case of the second variant.

On the other hand, when developing one node in the classical TDIDT algorithm, the search of the best test requires to sort the subsample for every attributes and to compute the score for at most N different thresholds per attribute. So, the whole optimization requires order $M.N.\log(N)$ operations. However, this sort can be done only once before growing the ensemble and then subsamples can be sorted in linear time by projection against the sorted learning sample. Nevertheless, this variant requires to store M vectors of indexes of the learning sample size and hence may be detrimental in the case of very large datasets. When $s_{th} = 0$ and the learning sample is pre-sorted, we thus gain with extra-trees a factor M in complexity for the development of one node. However, when $s_{th} > 0$, the complexity of splitting one node of an extra-tree only differs by a constant from the complexity of splitting one node of a classical tree. This constant of course depends on the score threshold. Nevertheless, since at most M splits must be evaluated at each node of an extra-tree, we can expect that splitting one node of an extra-tree will be faster than splitting one node of a classical tree even if the score threshold is high. Of course, the total gain in computing time will also depend on the final complexity of the trees (which depends on the size of the learning sample, on the algorithm, and also on the problem into consideration).

To make a prediction with an ensemble of decision trees, it is sufficient to propagate the test instance into all the trees of the ensemble. In average, tree depth will be of order $\log(N)$. The computation of the prediction of a set of T trees will thus require in average order $T.\log(N)$ comparisons. So, even if extra-trees are very complex in term of the total number of nodes, their use will remain very efficient even for very large learning sample sizes.

5 Experimentation

Our algorithm has been validated on 8 classification problems taken from the UCI machine learning repository [3]. They are described in Table 2. These problems have been chosen for their quite large size and because all attributes are numerical. The experimentations have two goals: first, validate our algorithm in terms of accuracy, then evaluate its computational efficiency. In both cases, the two variants of extra-trees are compared to classical decision trees, bagging, and random forests [7] (as it is the most recent representative of P&C methods for decision trees). Before describing the experiments, the next subsection copes with some implementation issues.

5.1 Implementation details

The score measure used throughout our experiments is the score measure proposed by Wehenkel in [18], which is written:

$$S_C^T = \frac{2I_C^T}{H_C + H_T}, \tag{2}$$

Table 2. Datasets

Dataset	Nb. attributs	Nb. classes	$\#LS$	$\#TS$
Waveform	21	3	4000	1000
Two-norm	20	2	8000	2000
Satellite	36	6	4435	2000
Pendigits	16	10	7494	3498
Dig44	16	10	14000	4000
Letter	16	26	16000	4000
Isolet	617	26	6238	1559
Mnist	784	10	60000	10000

where H_C denotes class entropy, H_T test entropy (also called split information by Quinlan), and I_C^T their mutual information. It is easy to show that this score measure takes its value in $[0, 1]$.

Whatever the algorithm, we have not used any stop splitting criterion or pruning algorithm. So, trees are always fully developed so as to obtain leaves as pure as possible in terms of the output classification.

For ensemble algorithms, the more trees are aggregated, the better is the improvement of error rates. For all ensemble variants, we built $T = 50$ trees. On most problems, larger values of T would give negligible improvements. For extra-trees, we have tried two values of the score threshold: $s_{th} = 0.0$ and $s_{th} = 0.1$. The random forests method also depends on only one parameter, k, which is the number of randomly selected attributes considered when developing a test node. In our experiments, we have fixed the value of k to the square root of the total number of attributes. This heuristic value has been suggested by the author of the algorithm in [8] and he claims that it gives error rates near to the optimum on most problems.

To compare our algorithm with the fastest possible implementation of classical trees, we have used pre-sorting for each tree variant that requires optimization of the discretization thresholds (i.e. all methods but extra-trees).

5.2 Experimentation protocol

On each problem, the dataset has been divided into a learning and a test sample which respective sizes are given in Table 5.2 (respectively $\#LS$ and $\#TS$). The experiment is repeated 10 times by randomizing the LS/TS split. Table 5.2 gives the average and standard deviation of some statistics estimated from these 10 runs. The first column is the error rate, the second one the complexity of the tree (the total number of nodes of the tree or of the ensemble of trees) and the last columns give the computing times (in msec) respectively for growing and testing the models[1]. These results are discussed in the next subsections.

5.3 Accuracy

The first variant of extra-trees is always better than the second one and it also produces less complex trees. Except on Two-norm, a score threshold of 0.1 always produces simpler and more accurate trees. The effect of filtering is especially important on Isolet where otherwise extra-trees are worst than classical trees.

Whatever the variant and the value of the score threshold, extra-trees improve decision tree accuracy very significantly on all problems (except on Isolet when $s_{th} = 0$). They also improves the accuracy with respect to bagging on most problems. The improvement with $s_{th} = 0.1$ is not significant on Waveform, Two-norm, Satellite, and Isolet but it is more important on the other four problems (on Mnist, with the first variant only). The comparison with random forests is more tight. Extra-trees with $s_{th} = 0.1$ slightly win on 5 problems and loose on 3 problems. The difference is significant on only one problem, Isolet, where random forests are better than extra-trees. However, on this latter problem, the high number of attributes suggests that extra-trees (which are more randomized than bagging and random forests) would require a higher number

[1] The system is implemented in C and runs on a pentium 4 2.4GHz with 1Go of main memory. In our experiments, data and models are stored in main memory.

Table 3. Experiment results

Dataset	Method	Error (%)	Complexity	Growing (msec)	Test (msec)
Waveform	One DT	24.78 ± 1.03	833 ± 22	1284 ± 38	1 ± 3
	Bagging	16.75 ± 0.99	29717 ± 256	43498 ± 613	35 ± 5
	Random forests	15.95 ± 1.32	38229 ± 296	11722 ± 100	35 ± 5
	Extra-trees 1 (0.0)	16.54 ± 1.20	244023 ± 1083	1026 ± 10	65 ± 5
	Extra-trees 1 (0.1)	15.44 ± 0.89	127217 ± 761	2455 ± 48	42 ± 4
	Extra-trees 2 (0.0)	16.72 ± 1.22	246716 ± 1129	932 ± 6	65 ± 5
	Extra-trees 2 (0.1)	15.40 ± 0.83	128007 ± 675	1746 ± 37	43 ± 4
Two-norm	One DT	14.89 ± 0.77	1009 ± 22	4476 ± 214	4 ± 4
	Bagging	3.31 ± 0.31	36430 ± 259	139986 ± 963	127 ± 4
	Random forests	3.15 ± 0.36	48310 ± 170	27087 ± 173	118 ± 4
	Extra trees 1 (0.0)	3.07 ± 0.46	306284 ± 934	1654 ± 5	153 ± 5
	Extra trees 1 (0.1)	3.32 ± 0.35	157216 ± 692	6533 ± 63	124 ± 4
	Extra trees 2 (0.0)	3.65 ± 0.34	310667 ± 1354	1483 ± 5	157 ± 5
	Extra trees 2 (0.1)	3.67 ± 0.43	159169 ± 793	4621 ± 44	123 ± 4
Satellite	One DT	14.68 ± 0.27	732 ± 21	723 ± 12	2.0 ± 4
	Bagging	9.73 ± 0.36	26411 ± 330	25465 ± 348	98 ± 4
	Random forests	8.95 ± 0.56	34145 ± 299	6717 ± 59	104 ± 4
	Extra trees 1 (0.0)	9.88 ± 0.48	172545 ± 1269	925 ± 10	145 ± 5
	Extra trees 1 (0.1)	8.93 ± 0.40	113559 ± 952	2208 ± 55	118 ± 4
	Extra trees 2 (0.0)	10.01 ± 0.48	185031 ± 1238	890 ± 6	150 ± 4
	Extra trees 2 (0.1)	9.08 ± 0.44	121323 ± 995	1597 ± 23	118 ± 4
Pendigits	One DT	3.82 ± 0.28	487 ± 22	529 ± 17	5.0 ± 5
	Bagging	1.74 ± 0.23	19775 ± 370	19164 ± 248	154.0 ± 4
	Random forests	0.98 ± 0.14	29222 ± 489	6930 ± 54	156.0 ± 4
	Extra trees 1 (0.0)	1.15 ± 0.08	220085 ± 2942	1429 ± 12	242 ± 6
	Extra trees 1 (0.1)	0.82 ± 0.12	132621 ± 2004	1586 ± 19	198.0 ± 4
	Extra trees 2 (0.0)	1.52 ± 0.20	275092 ± 2365	1524 ± 11	258 ± 4
	Extra trees 2 (0.1)	1.05 ± 0.17	172633 ± 2041	1507 ± 16	210.0 ± 4
Dig44	One DT	13.10 ± 0.54	2301 ± 26	6288 ± 114	2 ± 4
	Bagging	7.44 ± 0.47	84640 ± 788	203891 ± 2483	260 ± 6
	Random forests	4.98 ± 0.33	115328 ± 1060	55340 ± 231	282 ± 4
	Extra trees 1 (0.0)	5.48 ± 0.38	761721 ± 4998	4218 ± 68	447 ± 8
	Extra trees 1 (0.1)	4.35 ± 0.40	453526 ± 2580	4699 ± 40	350 ± 0
	Extra trees 2 (0.0)	6.87 ± 0.42	877359 ± 2719	4457 ± 56	467 ± 10
	Extra trees 2 (0.1)	4.83 ± 0.40	526620 ± 3018	4517 ± 26	370 ± 4
Letter	One DT	12.51 ± 0.52	3707 ± 34	943 ± 7	9 ± 3
	Bagging	6.76 ± 0.46	141261 ± 646	33840 ± 155	293 ± 4
	Random forests	3.85 ± 0.38	199388 ± 1315	14564 ± 76	344 ± 4
	Extra trees 1 (0.0)	4.81 ± 0.39	812938 ± 3987	5971 ± 25	538 ± 6
	Extra trees 1 (0.1)	3.84 ± 0.43	557215 ± 3175	6647 ± 31	427 ± 4
	Extra trees 2 (0.0)	5.25 ± 0.31	836460 ± 4177	6144 ± 44	547 ± 5
	Extra trees 2 (0.1)	4.25 ± 0.37	582029 ± 1897	6177 ± 42	442 ± 4
Isolet	One DT	17.96 ± 0.85	938 ± 10	175359 ± 1822	2 ± 4
	Bagging	8.63 ± 0.64	35119 ± 410	5325362 ± 29173	122 ± 4
	Random forests	5.99 ± 0.71	62102 ± 546	318709 ± 1265	160 ± 0
	Extra trees 1 (0.0)	19.26 ± 0.68	534433 ± 1468	3286 ± 17	276 ± 7
	Extra trees 1 (0.1)	8.52 ± 0.66	388136 ± 2245	4447 ± 37	241 ± 3
	Extra trees 2 (0.0)	24.22 ± 1.22	550567 ± 912	3336 ± 12	279 ± 5
	Extra trees 2 (0.1)	9.62 ± 0.77	422002 ± 1752	4252 ± 35	242 ± 4
Mnist	One DT	11.77 ± 0.20	7138 ± 65	357815 ± 50270	45 ± 14
	Bagging	4.64 ± 0.26	263114 ± 556	12539385 ± 373057	1960 ± 112
	Random forests	3.20 ± 0.14	457989 ± 963	1628626 ± 9226	1796 ± 18
	Extra trees 1 (0.0)	4.71 ± 0.20	3093799 ± 5874	41500 ± 798	3607 ± 170
	Extra trees 1 (0.1)	3.36 ± 0.19	1754887 ± 6809	589975 ± 26747	2294 ± 21
	Extra trees 2 (0.0)	7.35 ± 0.23	3839282 ± 10688	60606 ± 1145	2976 ± 115
	Extra trees 2 (0.1)	4.48 ± 0.20	2151508 ± 5211	487030 ± 6288	2167 ± 22

of trees to cancel out the effect of the randomization. And, indeed, increasing T to 200 trees on this problem reduces the error of the first variant of extra-trees with $s_{th} = 0.1$ to 6.42% while the error of random forests becomes 5.58%. So, even on this problem, the difference between the two algorithms is not very significant.

5.4 Computational efficiency

As expected, decision tree bagging multiplies the computing times of single trees by a factor close to 50. Random forests are much more efficient but they still increases very significantly the computing times of single trees (by a factor 9 in average). Extra-trees allow to reduce very strongly the computing times with respect to bagging and random forests. When $s_{th} = 0$, growing 50 extra-trees is even faster than growing one regular tree on five problems (Waveform, Two-norm, Dig44, Isolet, and Mnist). Of course, the variant with $s_{th} = 0.1$ is often slower but the difference is significant only on Waveform, Two-norm, Satellite, and Mnist. On several problems, the construction of 50 extra-trees with $s_{th} = 0.1$ still requires comparable (on Waveform, Two-norm, and Mnist) or less time (on Dig44 and Isolet[2]) than the construction of a single classical tree. From the point of view of computing times, the second variant of extra-trees is better than the first one, even though it builds more complex trees. Nevertheless, the improvement is not very important.

Even if the models induced by our algorithm are very complex, their use also remains extremely efficient. The test of one instance requires less than one millisecond on all problems and whatever the algorithm. Note also that even if the increase of complexity with respect to bagging and random forests is important on some problems, computing times for testing extra-trees are only slightly higher.

5.5 Discussion

With the same computational effort, the extra-tree method is thus much more accurate than decision trees. Its main advantage with respect to other ensemble methods like bagging and random forests is again its computational efficiency. Between the two variants, the first one should be preferred since it is only slightly slower and it produces less complex and more accurate models. Although the use of a score threshold makes the method less elegant and adds some complexity to the algorithm, the filtering of random split nevertheless seems necessary to obtain the best possible results in terms of accuracy. However, it is interesting to note that, on our problems, completely random trees (obtained with $s_{th} = 0$) are often competitive with classical decision trees and bagging in terms of accuracy.

At this point, it is nevertheless important to make some comments about the way our experiments have been carried out:

- We have not use any pruning. Pruning single decision trees could decrease error rates with respect to full trees. However, from the analysis presented in [2], we do not expect that it will improve very much the accuracy of ensemble methods.
- The score measure we use in our experiments is based on the computation of logarithms. Computing times could be reduced by using a score measure, like the gini entropy, that does not use logarithms and hence is faster to compute. Of course, the classical TDIDT algorithm and bagging, which compute many scores, would take more advantage of this improvement.
- Our experiments with only two values of the score threshold show that totally random trees, although more elegant, are unfortunately not the best in terms of accuracy. The combination of our score measure and a threshold of 0.1 seems to work well in most problems (at least in comparison with random forests) but it is clear that further experiments are necessary to study the exact impact of the score threshold on accuracy and computing times.

[2] On this problem, there is a high number of attributes and these attributes take their values on the real axis. Consequently, the number of potential values of discretization thresholds is very important and this explains the very impressive improvement of computing times with the extra-trees.

– Along the same idea, we have not tried different values of the parameter k in random forests. Just like the score threshold, this parameter influences both the accuracy and the computing times of the method and it would be interesting to study the impact of its value.
– One potential drawback of extra-trees is their very high complexity. While, fortunately, this high complexity does not come with high computing times for testing, it can be very detrimental in terms of the memory required to store the model. This question certainly deserves further experiments.

6 Conclusion and future work

In this paper, we have proposed a new ensemble methods based on decision trees. This method consists in growing trees by choosing their tests fully at random. On our test problems, this method is almost as fast as classical trees and improves very significantly their accuracy. It is also much faster and often more accurate than bagging and it gives comparable error rates with random forests but again more efficiently. Another contribution of this paper is to show, by the extreme nature of the proposed algorithm, that it is possible to take much freedom with respect to the classical TDIDT algorithm to design new ensemble methods. In particular, the good behaviour of extra-trees shows that the optimization of the discretization threshold is not relevant anymore in the context of ensemble methods. Of course, other variants could be imagined besides the two variants proposed here that could be better from the point of view of accuracy and/or computational efficiency.

Besides mining very large datasets, our method could also be very interesting in the context of incremental and adaptive learning. In such applications, the learning algorithm should be able to update the model when new instances are available and possibly to adapt itself to an output classification that evolves with time when new data arrive (i.e. the problem of concept drift). As the dependence of the tests on the learning sample is very low in extra-trees, it should be possible to design an efficient incremental version of this algorithm.

From a more theoretical point of view, the question of the good behaviour of ensemble methods is still an open question in machine learning. Although we have not addressed this question in this paper, we carry out in [15] an empirical study of ensemble methods from the point of view of bias and variance. This study shows that P&C methods works mainly by reducing the variance of decision trees while on the other hand they often increase their bias. Other analysis are given for example in [11] or [7]. In both papers, the improvement of accuracy with ensemble methods is related to a measure of the diversity among the predictions given by the models of the ensemble.

Finally, there also exist other recent learning algorithms which share several nice characteristics with P&C methods with trees (computational efficiency, possibility to handle a very large number of variables, robustness, generality...). It would be interesting to compare all these algorithms. For example, boosting (see [16] for the most recent variants) is another family of ensemble methods. Unlike P&C algorithms, models in a boosted ensemble are built sequentially by updating the learning sample according to the models that have already been grown. In several empirical comparisons, boosting has been shown to be significantly better than bagging (e.g. [2]). Support vector machines [10] often give very impressive results in terms of accuracy and there also exist very efficient implementation of this algorithm [14].

References

1. K. Ali and M. Pazzani. Error reduction through learning multiple descriptions. *Machine Learning*, 24(3):173–206, 1996.
2. E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms : Bagging, boosting, and variants. *Machine Learning*, 36:105–139, 1999.
3. C. Blake and C. Merz. UCI repository of machine learning databases, 1998. http://www.ics.uci.edu/~mlearn/MLRepository.html.
4. L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

5. L. Breiman. Arcing classifiers. *Annals of statistics*, 26:801–849, 1998.
6. L. Breiman. Randomizing outputs to increase prediction accuracy. *Machine Learning*, 40(3):229–242, September 2000.
7. L. Breiman. Random forests. *Machine learning*, 45:5–32, 2001.
8. L. Breiman. *Setting up, using, and understanding random forests V4.0*. University of California, Department of Statistics, 2003.
9. L. Breiman, J. Friedman, R. Olsen, and C. Stone. *Classification and Regression Trees*. Wadsworth International (California), 1984.
10. C. Cristianini and J. Shawe-Taylor. *An introduction to support vector machines*. MIT Press, Cambridge, MA, 2000.
11. T. G. Dietterich. Ensemble methods in machine learning. In *Proc. of the first International Workshop on Multiple Classifier Systems*, 2000.
12. T. G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157, August 2000.
13. T. G. Dietterich and E. B. Kong. Machine learning bias, statistical bias, and statistical variance of decision tree algorithms. Technical report, Department of Computer Science, Oregon State University, 1995.
14. J.-X. Dong, A. Krzyzak, and C. Suen. A fast svm training algorithm. In Springer, editor, *International workshop on pattern recognition with support vector machines*, pages 53–67, August 2002.
15. P. Geurts. *Contributions to decision tree induction: bias/variance tradeoff and time series classification*. PhD thesis, University of Liège, may 2002.
16. T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2001.
17. T. K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.
18. L. Wehenkel. *Automatic learning techniques in power systems*. Kluwer Academic, Boston, 1998.
19. D. Zighed and R. Rakotomalala. *Graphes d'Induction : Apprentissage et Data Mining*. Editions Hermes, 2000.