Parallelizing Boosting and Bagging

C. Yu D.B. Skillicorn
Department of Computing and Information Science
Queen's University, Kingston, Canada
{yu,skill}@cs.queensu.ca

February 2001
External Technical Report
ISSN-0836-0227-
2001-442

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared February 1, 2001

Abstract

Bagging and boosting are two general techniques for building predictors based on small samples from a dataset. We show that boosting can be parallelized, and then present performance results for parallelized bagging and boosting using OC1 decision trees and two standard datasets. The main results are that sample sizes limit achievable accuracy, regardless of computational time spent; that parallel boosting is more accurate than parallel bagging; and (unexpectedly) that parallel boosting is also cheaper than parallel bagging (at least over OC1).

Parallelizing Boosting and Arcing

C. Yu and D.B. Skillicorn
{yu,skill}@cs.queensu.ca

Abstract: Bagging and boosting are two general techniques for building predictors based on small samples from a dataset. We show that boosting can be parallelized, and then present performance results for parallelized bagging and boosting using OC1 decision trees and two standard datasets. The main results are that sample sizes limit achievable accuracy, regardless of computational time spent; that parallel boosting is more accurate than parallel bagging; and (unexpectedly) that parallel boosting is also cheaper than parallel bagging (at least over OC1).

1 Introduction

One of the most common data mining scenarios is the construction of predictors. A dataset for mining typically consists of (a large number of) objects, each of which has several attributes and is labelled with a class label, indicating what kind of object it is. The goal is to build a predictor that, given a new object and its attributes, will predict a class label for it. For example, a credit card organization has many examples of transactions, and can discover, after the fact, which ones were fraudulent. It wishes to build a predictor that will predict, for new transactions, which ones are likely to be fraudulent.

The dataset with known class labels is called the *training set*. It is common to reserve part of the data with known class labels to be used as simulated new data for which, however, the correct class labels are known. This makes it possible to test predictors by comparing their predictions with the known correct answers on data that was not used to construct the predictor. Such data is called the *test set*.

Many techniques for building predictors are known [2, 11]. When deciding which to use, properties of interest are: accuracy, usually measured by the error rate on the test set; cost of construction, the time that it takes to build a predictor from a training set; and cost of deployment, the time it takes to generate predictions for new data objects. Accuracy typically depends on the problem domain, although some broad-brush qualitative comparisons between data mining techniques are possible.

For example, decision trees [7, 11], which are the workhorses of prediction, can achieve accuracies in the high ninety percents. For a training set with n examples, each with d attributes, producing a decision tree of size t, the cost of construction ranges from $\mathcal{O}(dn)$ to $\mathcal{O}(dn^2 \log n)$, depending on the precise kind of decision tree being built. The cost of deployment is $\mathcal{O}(\log t)$.

One approach to predictor generation is to use small subsets of the training data, build predictors based on each subset, and then deploy a predictor that combines the predictions of each of the individual predictors, using either voting or regression. The advantage of such ensemble approaches [4] is that the combined prediction tends to have much smaller variance than that of a single monolithic predictor trained on the same dataset. Techniques based on small subsets have been shown to have the greatest known accuracies on several problems [6].

When the subsets are chosen independently, for example uniformly randomly with replacement, then the approach is known as *bagging* [3]. It has been observed that bagging can be naturally parallelized [13].

A more complex ensemble technique, called variously *boosting* [8] or *arcing* [5], uses information about how difficult each object is to classify to select the objects in succeeding samples. Intuitively, the idea is that objects that are hard to classify should be overrepresented in samples so that new predictors can expend more effort in finding tight boundaries between classes. Such techniques do indeed improve the accuracy of the resulting prediction ensemble. However, they are widely believed to be inherently sequential because of the dependence of the selection of each new sample on the behavior of previous predictors.

Training datasets can be extremely large. In situations where predictors must be generated or updated on short time scales it is natural to consider parallelism in the predictor generation process. In this paper we present parallel implementations for both bagging, which is straightforward, and boosting, which is more complex because of the sequential dependencies involved. We also present performance results for several datasets on a shared-memory parallel computer. These results shed light on appropriate tradeoffs between parallelism and sample size to achieve the greatest prediction accuracy.

Section 2 introduces ensemble approaches to data mining using predictors. Section 3 describes sequential algorithms for bagging and boosting. Section 4 describes our experimental setting. Section 5 describes the parallelization strategies for bagging, which is straightforward, and boosting, which is novel. Section 6 presents our experimental results and interpretation. Section 7 describes the contribution of the paper.

2 Predictors

Suppose that we are given a training set of the form

$$T = \{(\bar{x}_i, y_i) \mid i = 1, \ldots, n\}$$

where $\bar{x}_i$ are the attributes of the ith example in the set, and y_i is the corresponding value or class label for that example. The goal of prediction is to compute a predictor

$$\psi_T(\bar{x}) \mapsto y$$

which, given a new example, generates a predicted value or class label for it. The accuracy of a predictor on a test set of the form

$$\{(\bar{x}_i, y_i) \mid i = 1, \ldots, m\}$$

is given by

$$\text{accuracy} = \frac{\text{cardinality}\{\psi_T(\bar{x}_i) = y_i\}}{m}$$

and is usually expressed as a percentage.

Ensemble techniques for data mining use the same underlying prediction approach, but learn predictors on subsets of the training set, and make predictions by combining the predictions of their component predictors, either by regression or plurality voting. Hence a

---

for i = 1 to K
    select a sample or "bag", T_i, from T randomly with replacement
    learn predictor ψ_i from T_i

Figure 1: Sequential bagging

general ensemble approach to predictor computation is first to select a set of subsets of the training set, T_i. A predictor, ψ_{T_i} is learned from each of these subsets. When the ensemble predictor is used as a predictor ψ its result is computed by

$$\psi(\bar{x}) \;=\; \text{argmax}_i \; N_j$$

where

$$N_j \;=\; \text{cardinality}\{k \mid \psi_{T_k}(\bar{x}_i) = j\}$$

The error of any predictor can be divided into three components:

$$\text{Error}(\psi) \;=\; \text{Error}(\psi^*) \;+\; \text{Bias}(\psi) \;+\; \text{Variance}(\psi)$$

where the first term is the minimum error of a Bayes classifier, the second term is the systematic error of the prediction technique, and the third term is the error resulting from the finite sample used to build the predictor. Bagging improves accuracy by decreasing the magnitude of the variance term, while boosting decreases both bias and variance [1].

3 Sequential bagging and boosting/arcing

3.1 Bagging

Bagging is based on selecting subsets randomly with replacement. Hence a given example may be overrepresented or underpresented in the sample used to generate each predictor. The basic sequential algorithm is given in Figure 1. When the resulting ensemble predictor is used,

$$\psi(\bar{x}) \;=\; \text{plurality vote among } \psi_i(\bar{x})$$

The number of predictors generated, K, is typically assumed to be an input parameter. Remarks in the literature suggest that 10–15 predictors are needed to achieve significant accuracy improvements. Our results suggest that much larger numbers of predictors continue to improve overall accuracy, although the rate of improvement slows.

3.2 Boosting/Arcing

Boosting or arcing tries to improve the quality of the predictors learned from each sample by selecting examples that are "hard" to classify more often than their frequency in the training

given a training set $T = \{(\bar{x}_j, y_j) \mid j = 1, \ldots, m\}$
initialize $D_1(j) = 1/n$
for i = 1 to K
    select sample T_i from T using distribution D_i as selection probabilities
    learn ψ_i from T_i
    compute the error for ψ_i as
        $\varepsilon = \sum_{j\ s.t. \psi_i(\bar{x}_j) \neq y_j} D_i(j)$
        (if $\varepsilon > 1/2$ break)
    $\beta_i = \varepsilon_i/(1 - \varepsilon_i)$
    $\text{factor} = \begin{cases} \beta_i & \text{if} \psi_i(\bar{x}_j) = y_j \\ 1 & \text{if} \psi_i(\bar{x}_j) \neq y_j \end{cases}$
    $D_{i+1}(j) = D_i(j)/Z_i \times \text{ factor}$

Figure 2: Sequential boosting

set and selecting points that are "easy" to classify less frequently. The intuition is that more of the computational effort is being used to learn those aspects of the data that are more difficult to learn. The decision about which examples are hard is made incrementally, based on the ability of previous predictors to classify them correctly. The first predictor assumes that all examples are equally hard. If the first predictor is used to classify the examples in the *training set*, then those examples that it misclassifies may be tentatively classified as more difficult. Such examples are then disproportionately selected for inclusion in the second sample. The selection for the third sample is based on the classification achieved by both the first and second predictors. As the process continues, examples that are persistently misclassified become increasingly likely to be included in new samples.

Several variants of this idea have been described in the literature, varying in the way in which the probabilities of selection are varied in response to earlier classifications, and how the resulting predictors' outputs are weighted in use [5, 6]. The predictor construction phase of the Adaboost algorithm is shown in Figure 2 [8]. The variable D holds weights associated with each example in the training set, modelling a probability distribution used for sample selection. ε is a global error estimate; when it exceeds $1/2$, the algorithm cannot continue directly so, if this occurs, the distribution may be reinitialized and the algorithm continued or restarted. The variable β models the accuracy of the current predictor and determines how quickly correctly-classified examples have their weights reduced. The combined predictor is given by

$$\psi(\bar{x}) \ = \ \text{argmax} \sum_{j\, s.t. \psi_i(\bar{x}) = y} \log(1/\beta_i)$$

in other words, a vote weighted by the accuracies of the individual predictors on the training set.

Several variants have been suggested. *arc-fs* [5] removes "easy" examples from consideration more quickly than Adaboost by setting their selection probability to zero as soon as they have been classified correctly by any predictor. It uses weighted voting in the combined predictor. *arc-x4* [5] reduces selection probabilities by a function (involving fourth powers, hence the name) of the number of misclassifications so far by all predictors. However, it uses unweighted voting, a deliberate attempt to show that the power of boosting techniques comes from their sample selection rather than from the way they handle voting. There are many subtleties in the accuracy properties of both bagging and boosting. Results of a large empirical study can be found in [1].

All of these boosting/arcing ensemble techniques have an implied sequential dependency, since the selection process at phase $i+1$ depends on the classification behavior at phase i. However, as we shall show, it is possible to effectively parallelize ensemble techniques of this kind.

4 Experimental Setting

Ensemble techniques may be used with any weak learner as the underlying data mining technique. In our experiments we use the oblique decision tree predictor, OC1 [10]. Decision trees are usually constrained to test a single attribute at each internal node. OC1 allows a linear combination of attributes to be tested at internal nodes. Geometrically, a decision tree partitions the attribute space into regions delimited by axis-parallel hyperplanes, while OC1 may use oblique hyperplanes. For datasets in which the actual boundaries between classes are not well modelled by axis-parallel hyperplanes, OC1 can be expected to achieve good accuracy with smaller decision trees.

The cost of improved prediction is that building predictors is more expensive than for ordinary decision trees. The time complexity of tree construction is $\mathcal{O}(dn^2 \log n)$ for a training set with n examples and d attributes. However, real datasets do not seem to require this much time, as we shall see later.

We use a SUN Enterprise 3500 computer, with 6 Ultrasparc II 336Mhz processors, as the test platform. In order to reduce the effects of other processes on our test programs, no more than five processors were used. Elapsed times reported for our experiments are averages of 10 executions of the same code.

We use the Bulk Synchronous Parallel Library, *BSPlib* [9, 14], as the parallel programming environment. This is a high-performance library (i.e. typically faster than MPI) that is well-suited to problems whose large-scale structures consists of alternating phases of computation and communication. In particular, *BSPlib* is carefully optimized to implement total-exchange effectively, and we use this extensively in our implementations.

The datasets we use in our experiments were chosen to be large, so that partitioning the data across multiple processors would still allow fairly large subsets at each one. They were also chosen to be quite difficult, that is the accuracies reported for direct sequential mining left some room for improvement.

We use the two datasets described in Figure 3. These datasets are available from the University of California, Irvine, archive (`kdd.ics.uci.edu`). The *letters* dataset is unusual because the number of classes is large – this makes classification quite challenging.

Name	# examples	training set size	test set size	# attributes	# classes
letters	20000	20000	400	16 numeric	26
cover type	581012	11 340	3780	10 numeric 44 binary	8

Figure 3: Datasets used

5 Parallelization strategies

Parallelizing bagging is relatively straightforward. The training set is partitioned equally across the available processors. Each processor executes the sequential algorithm on its local data until sufficient predictors have been constructed. In general, it is a good idea to partition the training set randomly to ensure that the predictors generated by each processor do not contain unnecessary bias.

The cost of a single round of parallel bagging can be expressed in the form

$$\text{single round cost} \;=\; SAMPLE(n/P, m) \;+\; OC1(m) \tag{1}$$

where m is the size of samples being used, $SAMPLE$ is the cost of selecting a sample (which depends on both the size of the local partition and the size of the sample being selected), and $OC1$ is the cost of generating an OC1 decision tree from a sample of size m.

The parallelization of boosting is more complex. Again we partition the training set across the available processors. We then execute a number of rounds. In each round, each processor builds a predictor using a sample from its local data. These predictors are then exchanged among the processors, so that each processor now holds a copy of all of the predictors built during the current round. The reweighting of examples in preparation for the next round is done using all of these predictors: for each processor, an example is considered easy if it is correctly classified by some number of the predictors from the current round. This number is called the *threshold*. In other words, an example is reweighted as in the sequential algorithm, but a correct prediction is reinterpreted to mean "correct with respect to some number of predictors."

The cost of a single round of parallel boosting can be expressed in the form

$$\text{single round cost} \;=\; SAMPLE(n/P, m) \;+\; OC1(m) \;+\; tPg \;+\; VOTE(n/P \times P) \tag{2}$$

where m is the size of samples being used, $SAMPLE$ is the cost of selecting a sample, $OC1$ is the cost of generating an OC1 decision tree from a sample of size m, tPg is the cost of total exchange of trees of size t among P processors (where g is the BSP network permeability architecture parameter), and $VOTE$ is the cost of voting the n/P local examples in each partition against P trees from the current round.

The crucial property of the parallelized algorithm is that the samples used, in every processor, for the ith round use the information from *all* of the predictors in the previous round. The quality of each processor's information about hard versus easy examples is

given a training set $T = \{(\bar{x}_j, y_j) \mid j = 1, \ldots, m\}$
partition T into P subsets, T_p
initialize $D_{i,p} := 1/nP$
for i = 1 to K
  forall p in 1,...,P
    select a sample $T_{i,p}$ from T_p using $D_{i,p}$
    build predictors $\psi_{i,p}$
    exchange the $\psi_{i,p}$'s among all processors
    compute errors $\varepsilon_{i,p} = 1/2 \sum_j D_{i,p}(j)$ if $\bar{x}_j$ is misclassified by more than
      threshold processors (i.e. $\mathrm{vote}_p \psi_{i,p}(\bar{x}_j) \neq y_j$)
      (if $\varepsilon > 1/2$ break)
    $\beta_{i,p} := \varepsilon_{i,p}/(1 - \varepsilon_{i,p})$
    $\text{factor} = \begin{cases} \beta_{i,p} & \text{if} \psi_i(\bar{x}_j) = y_j \\ 1 & \text{if} \psi_i(\bar{x}_j) \neq y_j \end{cases}$
    $D_{i+1,p} := D_{i,p}/Z_{i,p} \times \text{ factor}$

Figure 4: Parallelized boosting

informed by results learned by other processors as well as its own predictor. The algorithm is given in Figure 4.

The resulting global predictor is

$$\psi(\bar{x}) \;=\; \mathrm{argmax}_{PK} \sum_{y \in Y, (\psi_{i,p}(\bar{x}) = y)} \log(1/\beta_{i,p})$$

The parameters that must be chosen for a parallel ensemble technique are: the number of processors to use, the size of samples to use, and the appropriate threshold (for boosting). These choices are not obvious, even for bagging. In the next section we report the results of a set of experiments that explore the tradeoffs between these parameters.

6 Experimental results

Sample size limits accuracy, regardless of computational effort. The curves of accuracy versus number of iterations of the outer loop of the algorithm, for both bagging and boosting, have a characteristic shape, rising quickly at the beginning and then flattening out to an asymptotic accuracy value. The asymptotes increase with increasing sample size. Hence the best achievable accuracy depends on the size of sample chosen, but not on the total computational effort expended.

Figure 5 shows, for bagging, the effect of different sample sizes on accuracy for a small number of iterations, and Figure 6 shows the effect as the number of iterations increases. Figures 7 and 8 show the same effect for boosting, on the *letters* and *covtype* datasets

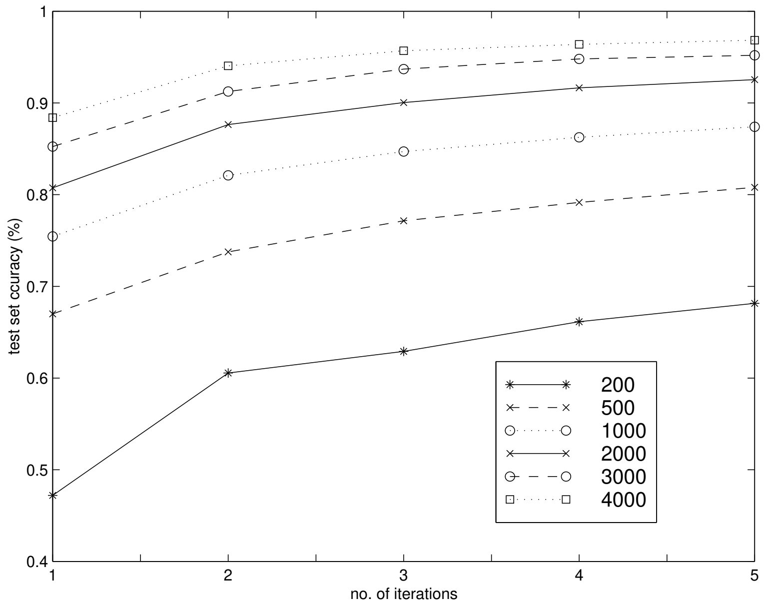

Figure 5: Increasing sample size increases accuracy (parallel bagging, 4 processors, *letters* dataset).

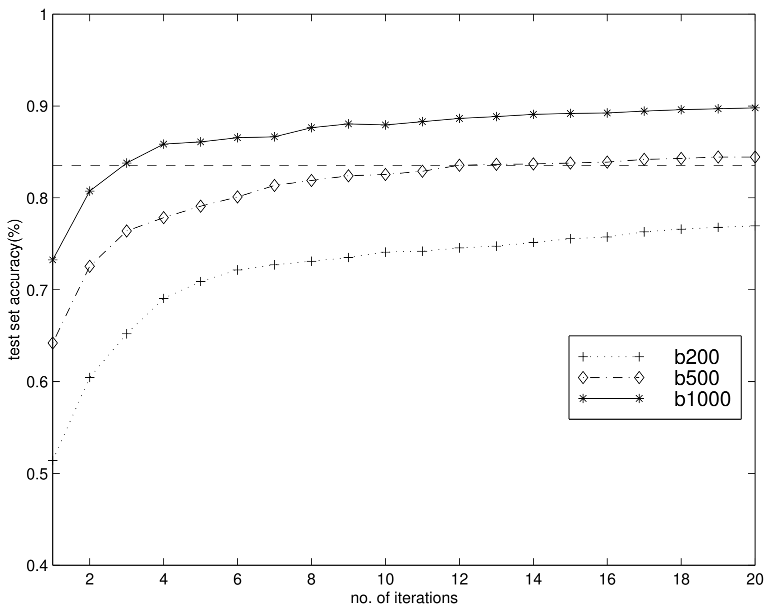

Figure 6: Accuracy is limited by sample size, regardless of how much computation is used. Moderate sample sizes must be used to outperform the sequential OC1 algorithm, shown by the dashed line (parallel bagging, 4 processors, *letters* dataset).

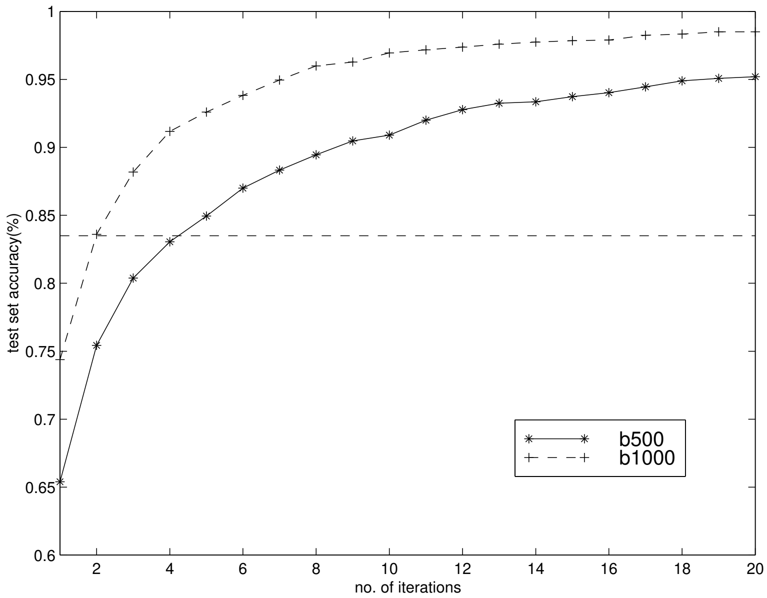

Figure 7: Increasing sample size increases accuracy (parallel boosting, 4 processors, *letters* dataset). The dashed line is the accuracy of the sequential OC1 algorithm.

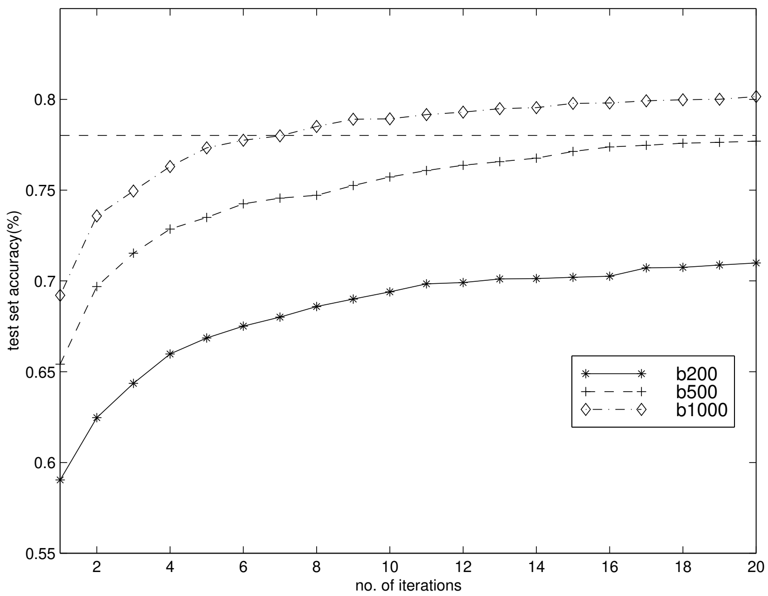

Figure 8: Increasing sample size increases accuracy (parallel boosting, 4 processors, *covtype* dataset). The dashed line is the accuracy of the sequential algorithm.

1 proc.		2 procs.		4 procs.		5 procs.	
work	acc. time	work	acc. time	work	acc. time	work	acc. time
20×1	0.7075 154.98	10×2	0.7105 78.67	5×4	0.7090 40.12	4×5	0.6800 32.04
40×1	0.7350 314.77	20×2	0.7460 154.53	10×4	0.7410 80.03	8×5	0.7460 63.84
60×1	0.7525 470.37	30×2	0.7685 233.82	15×4	0.7575 118.40	12×5	0.7670 97.91
80×1	0.7625 625.75	40×2	0.7760 308.30	20×4	0.7695 157.78	16×5	0.7660 128.43

Figure 9: When the same number of examples are seen, accuracy and total execution time are approximately the same, regardless of how the computation is arranged (parallel bagging, *letters* dataset, sample size = 200). Work is expressed as the product of number of rounds $\times$ number of processors.

respectively. For the *letters* dataset, boosting implementations fairly easily outperform the accuracy of a pure sequential OC1 implementation. For the *covtype* dataset, this is much more difficult, and large sample sizes are required. It is important to point out, however, that the sequential implementations use *much* longer execution times than any of the ensemble implementations – 4382s for the *letters* dataset, and 1035s for the *covtype* dataset.

One corollary to the dependence of accuracy on sample size is to suggest an upper bound for the degree of parallelism that is worth using in ensemble-based approaches. If the smallest sample size that achieves better accuracy than the sequential algorithm is s, then there is no reason to choose P, the number of processors, larger than the value that satisfies $n/P = s$. Fortunately, n is extremely large in most data mining applications, while the number of processors available in commonly available parallel computers is relatively small. The algorithms described here are therefore still of practical interest.

One question that our results do not answer is whether a reasonable sample size (that is, one that allows accuracies better than the straightforward sequential algorithm) is best regarded as an absolute size or a fixed fraction of the training dataset. Two arguments suggest that it is the former. First, if there is redundancy in the training data, as there usually is in data mining applications, it seems counter-intuitive that doubling the size of the training dataset should require selecting bigger samples. Second, it is clear from the figures that the relative accuracy improvement obtained by using a larger sample becomes quite small (perhaps a percentage point) by the time a few thousand examples are included in the sample. There seems little to be gained by increasing sample sizes once such large samples are being used.

6.1 Bagging

The effect of parallelism. Parallelizing bagging amounts to rearranging the computation of the sequential bagging algorithm, and therefore we might expect that speedups would be

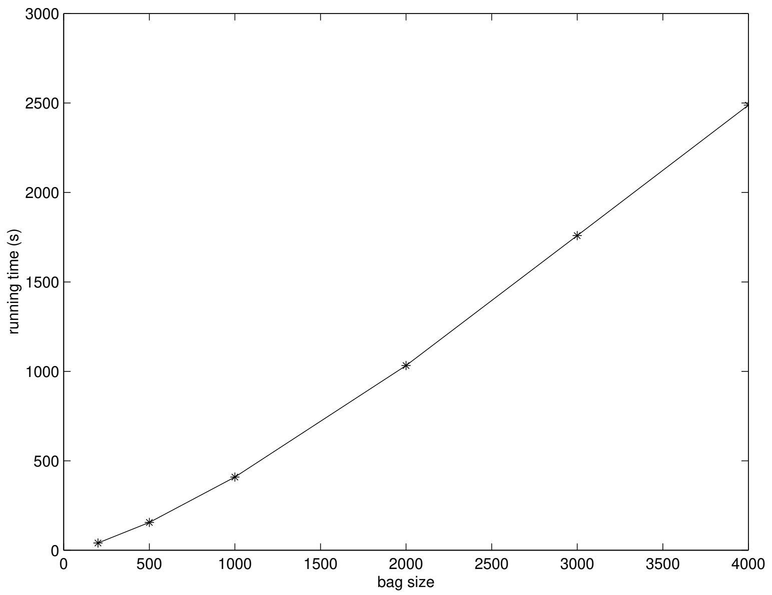

Figure 10: The time required for 5 rounds increases faster than linearly as a function of sample size (parallel bagging, 4 processors, *letters* dataset).

linear with the number of processors used, for the same sample size. The only potential source of performance improvement is the fact that each processor is selecting its samples from a partition of the entire dataset, and therefore has to make fewer memory accesses to select them. The data in Figure 9 shows that this effect is negligible – for the same number of examples processed, the total time spent remains the same, regardless of how many processors are used. Hence, of course, perfect linear speedup is achieved. Puzzlingly, using two processors produces better accuracy than using one, if only by a slim margin. The limits on scalability of this algorithm are that the partition allocated to each processor should not become too small to select samples of appropriate sizes.

The effect of sample size. Since building OC1 decision trees potentially requires time superlinear in the number of examples used, we might expect that using small samples would provide a significant performance enhancement. For the *letters* dataset, the time required to build trees is worse than linear, but not quadratic. Figure 10 shows the required execution time as a function of sample size.

The overall effect of sample size is shown in Figure 11. Each row of this table gives the accuracy and cost of computing a predictor after having seen the same number of examples, but in samples of different sizes. Unsurprisingly, given the comments at the beginning of this section, accuracies are greater for larger sample sizes. However, the total time required increases quite substantially with increasing sample size. This time increase is due to the increased complexity of building decision trees from larger samples. In other words, in Equation 1 the *SAMPLE* term is linear in both n/P and m, but the *OC1* term is worse than linear in m, and this accounts for the increasing time.

Parallel bagging holds few surprises. The parallel algorithm is simply a rearrangement of

200(seq)		200(par)		500(par)		1000(par)		2000(par)	
work	acc. time	work	acc. time	work	acc. time	work	acc. time	work	acc. time
20×1	0.7075 154.98	5×4	0.7090 40.12	2×4	0.7210 63.62	1×4	0.7270 83.22		
40×1	0.7350 314.77	10×4	0.7410 80.03	4×4	0.7800 124.99	2×4	0.8230 166.35	1×4	0.8210 206.16
60×1	0.7525 470.37	15×4	0.7575 118.40	6×4	0.8075 190.10	3×4	0.8335 248.01		
80×1	0.7625 625.75	20×4	0.7695 157.78	8×4	0.8090 253.39	4×4	0.8590 326.39	2×4	0.8680 415.92

Figure 11: When the same number of examples are seen, accuracy improves with increasing sample size, but total execution time also increases (parallel bagging, 4 processors, *letters* dataset).

sequential bagging across a set of processors. The overall accuracy and computational cost remains the same regardless of this rearrangement.

6.2 Boosting

It is far less obvious what performance to expect from parallel boosting. On the one hand, there are considerable overheads in exchanging OC1 trees and voting after each round, which suggests that speedups may be poor. On the other hand, each processor is acquiring information learned by all of the other processors in a compact way, and this has often led to superlinear speedups in data mining applications [12, 15].

The effect of threshold. The threshold determines how many decision trees must agree for an example to be classified as easy. Figures 12 and 13 show that choosing a threshold value of 2 is best for the *letters* dataset, while Figures 14 and 15 show that choosing a threshold value of 3 is best for the *covtype* dataset. In both cases, the best choice of threshold is independent of sample size. Also in both cases, the accuracy as a function of threshold is strongly concave downwards, so that both small and large values of threshold perform much worse than moderate values. It appears that the best threshold is a constant, for each dataset, rather than a fraction of the number of available processors but we were unable to explore this in more detail.

The effect of parallelism. For implementations that have seen the same number of examples, it is clear from Figure 16 (for the *letters* dataset) and Figure 17 (for the *covtype* dataset) that increasing parallelism results in greater accuracies. This suggests that the sharing of information gleaned by processors amongst themselves by voting does indeed improve the ability of the algorithm to learn. It is, however, also clear from these Tables that this improved accuracy has a performance cost. The total amount of work required increases

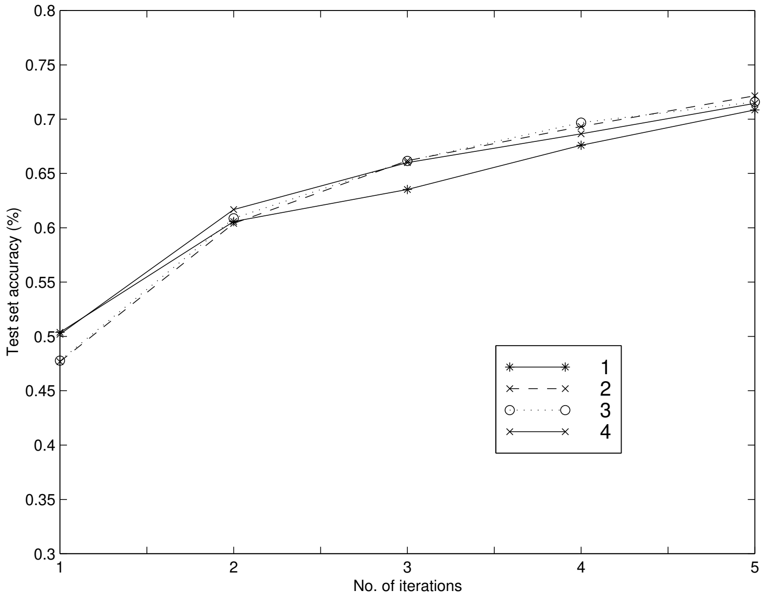

Figure 12: Choosing threshold=2 is (marginally) best for the *letters* dataset (parallel boosting, 4 processors, sample size 200).

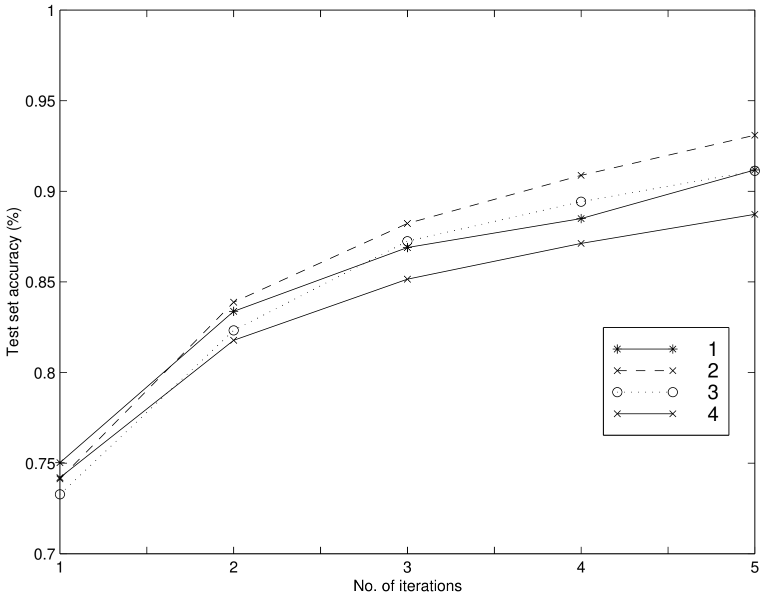

Figure 13: Choosing threshold=2 is best for the *letters* dataset (parallel boosting, 4 processors, sample size 1000).

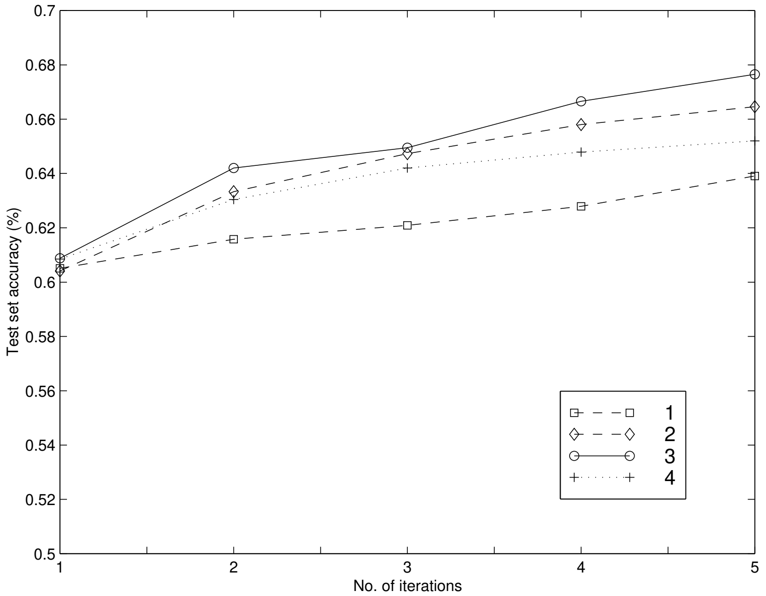

Figure 14: Choosing threshold=3 is best for the *covtype* dataset (parallel boosting, 4 processors, sample size 200).

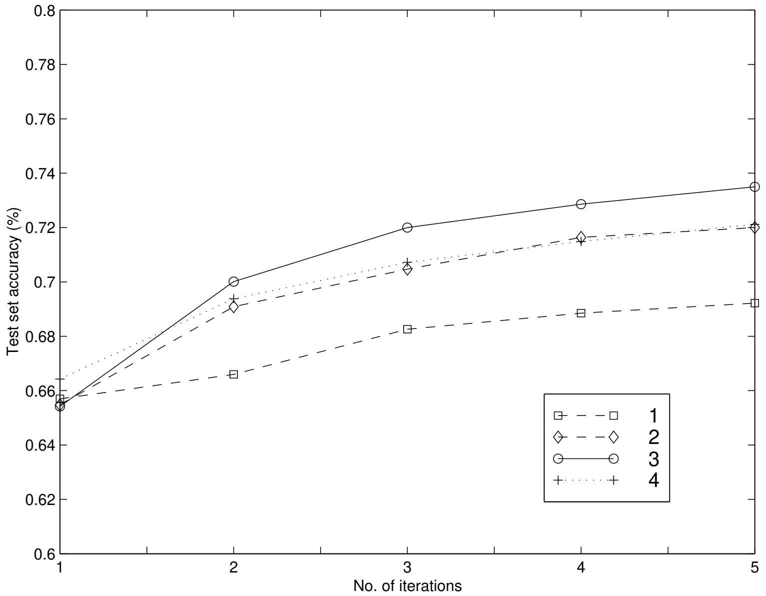

Figure 15: Choosing threshold=3 is best for the *covtype* dataset (parallel boosting, 4 processors, sample size 500).

1 proc.		2 procs.		4 procs.		5 procs.	
work	acc. time	work	acc. time	work	acc. time	work	acc. time
20×1	0.6950 177.55	10×2	0.7150 90.03	5×4	0.7183 49.75	4×5	0.7215 46.38
40×1	0.7580 358.60	20×2	0.7645 181.58	10×4	0.7765 104.82	8×5	0.7895 89.72
60×1	0.7650 534.22	30×2	0.7920 277.37	15×4	0.8208 154.37	12×5	0.8145 130.73
80×1	0.7695 711.65	40×2	0.8055 368.19	20×4	0.8360 199.97	16×5	0.8438 171.24

Figure 16: When the same number of examples are seen, accuracy increases with parallelism, but so does total execution time (parallel boosting, *letters* dataset, sample size = 200).

1 proc.		2 procs.		4 procs.		5 procs.	
work	acc. time	work	acc. time	work	acc. time	work	acc. time
20×1	0.6490 88.56	10×2	0.6345 50.46	5×4	0.6765 24.86	4×5	0.6755 24.10
40×1	0.6507 182.83	20×2	0.6451 88.56	10×4	0.6948 49.62	8×5	0.6950 47.80
60×1	0.6519 273.40	30×2	0.6571 148.60	15×4	0.7037 77.88	12×5	0.7058 73.73
80×1	0.6538 371.57	40×2	0.6626 201.81	20×4	0.7089 102.59	16×5	0.7102 89.71

Figure 17: When the same number of examples are seen, accuracy increases very slightly with parallelism, but so does total execution time (parallel boosting, *covtype* dataset, sample size = 200).

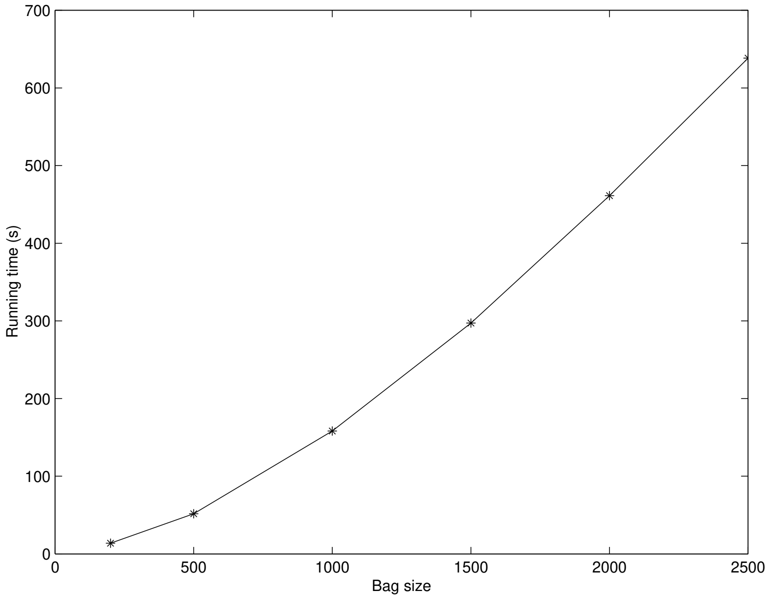

Figure 18: The time required for 5 rounds increases faster than linearly as a function of sample size (parallel boosting, 4 processors, *covtype* dataset).

(slowly) as the number of processors increases. In other words, the speedup is sublinear. This reflects the increased communication cost as more processors are used. Notice also that the cost of voting at the end of each round, the $VOTE$ term in the cost of Equation 2, is a function of n and so does not parallelize. In practice, the cost of tree construction dominates communication and voting to update the distribution for these modest numbers of processors. For example, in an early round of boosting, using 4 processors and the *letters* dataset, sampling took 0.14% of the time, tree generation 97.76%, and voting and updating 2.11% of the time. The approach can be expected to scale well, perhaps up to 50 processors, provided that the dataset is large enough that partitions allow reasonable samples to be selected.

The effect of sample size. We have already observed that the time required to build an OC1 tree increases faster than linearly with the number of examples used for the *letters* dataset. Figure 18 shows that the relationship is quite similar for the *covtype* dataset. Hence we expect to see performance penalties using larger sample sizes.

The effect of sample size is shown in Figures 19 and 20. As before, each row gives the accuracy and cost of computing a predictor after having seen the same number of examples, but in samples of different sizes. Once again, accuracies are greater for larger sample sizes (although less so for the *covtype* dataset), but the total time required also increases substantially. These costs reflect the cost of building decision trees from larger samples. In other words, the *SAMPLE* cost is linear in both its arguments, the *OC1* cost grows faster than linearly in m, the communication term is independent of m (since tree sizes remain more or less constant as sample sizes vary), and the $VOTE$ cost is constant.

Parallel boosting does achieve improved accuracies compared to sequential boosting be-

200(seq)		200(par)		500(par)		1000(par)		2000(par)	
work	acc. time	work	acc. time	work	acc. time	work	acc. time	work	acc. time
20×1	0.6950 177.55	5×4	0.7215 49.75	2×4	0.7620 63.17	1×4	0.7450 78.56		
40×1	0.7580 358.60	10×4	0.7895 104.82	4×4	0.8410 122.60	2×4	0.8320 148.38	1×4	0.8115 182.01
60×1	0.7650 534.22	15×4	0.8208 154.37	6×4	0.8725 185.93	3×4	0.8830 218.70		
80×1	0.7695 711.65	20×4	0.8360 199.97	8×4	0.8990 235.04	4×4	0.8980 282.28	2×4	0.8985 333.25

Figure 19: When the same number of examples are seen, accuracy improves with increasing sample size, but total execution time also increases (parallel boosting, 4 processors, *letters* dataset).

200(seq)		200(par)		500(par)		1000(par)		2000(par)	
work	acc. time	work	acc. time	work	acc. time	work	acc. time	work	acc. time
20×1	0.6490 88.56	5×4	0.6765 24.86	2×4	0.7026 25.61	1×4	0.6889 30.14		
40×1	0.6507 182.83	10×4	0.6948 49.62	4×4	0.7286 48.72	2×4	0.7349 60.60	1×4	0.7160 80.21
60×1	0.6519 273.40	15×4	0.7037 77.88	6×4	0.7397 73.82	3×4	0.7499 91.87		
80×1	0.6538 371.57	20×4	0.7089 102.59	8×4	0.7465 98.51	4×4	0.7669 117.24	2×4	0.7610 151.54

Figure 20: When the same number of examples are seen, accuracy improves with increasing sample size but then flattens. Total execution time also increases (parallel boosting, 4 processors, *covtype* dataset).

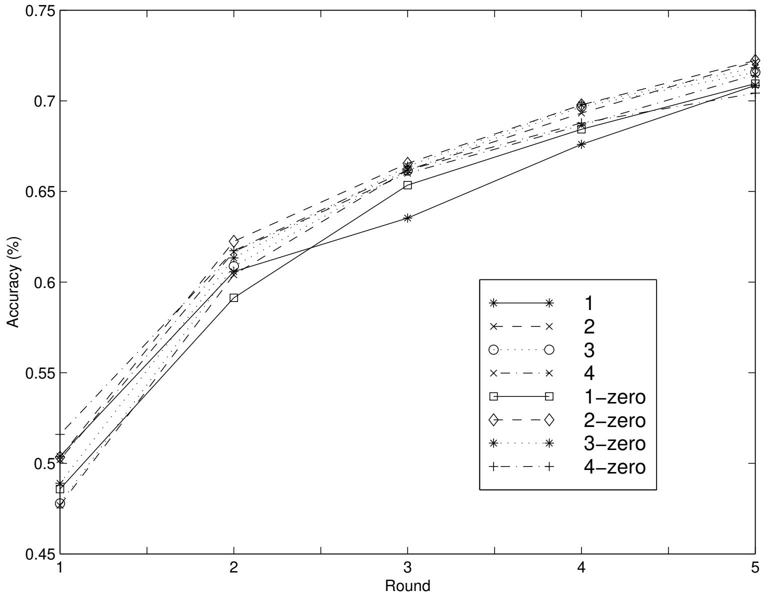

Figure 21: Zeroing selection probabilities quickly has little effect on accuracy or best choice of threshold (boosting, *letters* dataset, sample size 200).

cause of the information sharing that is implicit in the use of all of the predictors from a round to select the samples for the subsequent round. However, there is a computational cost to this improved accuracy – exchanging the predictors adds a communication cost that is linear in P, and the resulting voting step cannot be parallelized (recall its magnitude is a function only of n), so that there is some upper limit to the effectiveness of parallelism.

6.3 Adding arcing features

The effect of zeroing selection probabilities quickly. Several of the arcing ensemble algorithms reduce the selection probability of "easy" examples to zero rapidly in comparison to boosting. Figures 21 and 22 show that using such a strategy for updating selection probabilities makes almost no difference in either threshold selection or achieved accuracy.

6.4 Comparing bagging and boosting

Accuracy of boosting and bagging. Figure 23 shows that, for the same scenario, boosting always achieves greater accuracy than bagging. This vindicates the intuition that it is better to devote resources to the "hard" examples at the expense of the "easy" examples.

Cost of bagging and boosting. More surprisingly, Figure 24 shows that, for the same scenario, boosting is also computationally cheaper than bagging. When the costs of each round are compared in detail, it becomes clear that this is because the time to compute OC1 decision trees decreases in the later rounds of boosting, but does not decrease in the later rounds of bagging. The samples used in later rounds of boosting are more repetitive than

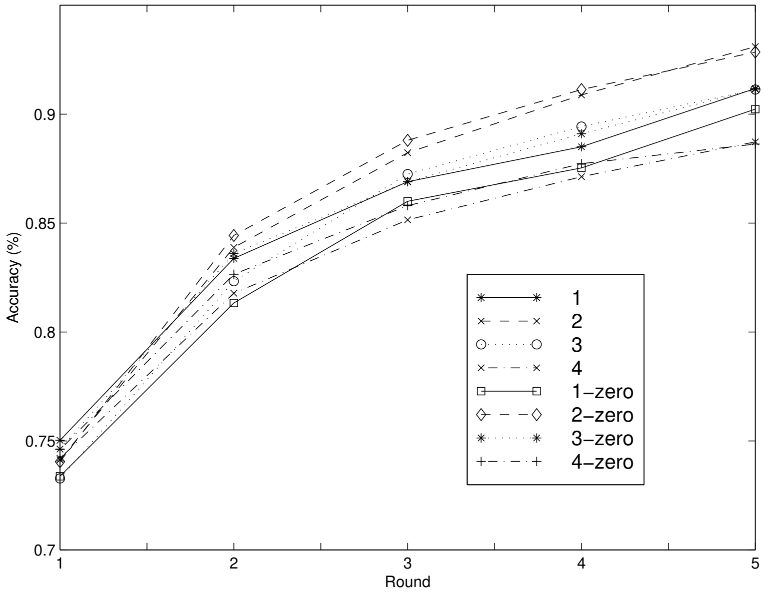

Figure 22: Zeroing selection probabilities quickly has little effect on accuracy or best choice of threshold (boosting, *letters* dataset, sample size 1000).

sample size	200		500		1000		2000	
	work	acc.	work	acc.	work	acc.	work	acc.
bagging	5×4	0.7090	2×4	0.7210	1×4	0.7270		
boosting		0.7215		0.7620		0.7450		
bagging	10×4	0.7410	4×4	0.7800	2×4	0.8230	1×4	0.8210
boosting		0.7895		0.8410		0.8320		0.8115
bagging	15×4	0.7575	6×4	0.8075	3×4	0.8335		
boosting		0.8208		0.8725		0.8830		
bagging	20×4	0.7695	8×4	0.8090	4×4	0.8590	2×4	0.8680
boosting		0.8360		0.8990		0.8980		0.8985

Figure 23: Parallel boosting is always more accurate than parallel bagging for the same scenario.

sample size	200		500		1000		2000	
	work	time	work	time	work	time	work	time
bagging boosting	5×4	40.12 49.75	2×4	63.62 63.17	1×4	83.22 78.56		
bagging boosting	10×4	80.03 104.82	4×4	124.99 122.60	2×4	166.35 148.38	1×4	206.16 182.01
bagging boosting	15×4	118.40 154.37	6×4	190.10 185.93	3×4	248.01 218.70		
bagging boosting	20×4	157.78 199.97	8×4	253.39 235.04	4×4	326.39 282.28	2×4	415.92 333.25

Figure 24: Parallel boosting is always cheaper to execute than parallel bagging for the same scenario.

those of earlier rounds since the pool of examples to select from is effectively shrinking each time. One possible explanation for the observed results is that the OC1 algorithm does less work to achieve the same quality of decision tree for such homogeneous samples. Figure 25 shows that this is indeed the case. The solid line shows the computation time for OC1 on random samples of increasing size from the *letters* dataset. The dashed line shows the computation time for samples of matching size obtained by replicating a random sample of 500 examples an appropriate number of times. It is clear that an OC1 decision tree can be built much more quickly from the more homogeneous sample. The actual samples in later rounds of parallel boosting are not nearly as homogeneous as the data used in this figure, so the effect is much smaller – but it suffices to outperform parallel bagging. Bauer and Kohavi [1] postulate a similar effect for their more-typical decision tree builder, MC4.

7 Conclusions

There are four main contributions of this work:

1. We have shown that the achievable accuracy of ensemble-based techniques is bounded by the choice of sample size. Choosing a small sample size reduces the time to compute each predictor (disproportionately when the underlying prediction technique has complexity worse than linear in the sample size). However, we have shown that this must be balanced by the need to achieve reasonable accuracy, and hence that there are limits to fine-grained ensemble approaches.

2. We have shown that boosting can be usefully parallelized despite its apparent sequential structure.

3. We have shown that parallel boosting achieves greater accuracy than parallel bagging for comparable scenarios. This vindicates the intuition behind boosting, that it is useful to spend resources disproportionately on the "hard" examples in a dataset.

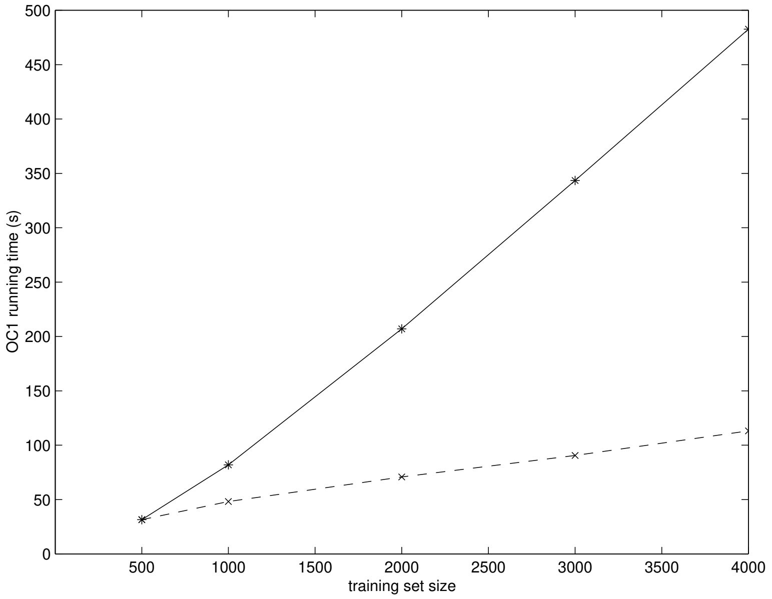

Figure 25: The time required to build an OC1 decision tree is much smaller for multiple copies of the same 500 examples than for a randomly chosen dataset of the same size.

4. We have shown that, at least for OC1, parallel boosting also outperforms parallel bagging for comparable scenarios. This result was unexpected and is due to the fact that OC1 tree construction is faster on more homogeneous datasets.

We have also shown that parallel bagging behaves as expected – the overall accuracy and total execution time are unaffected by the way in which computing each predictor is arranged in space and time. Hence the only reason for parallelizing bagging is to reduce the elapsed time for generating a global predictor.

The performance tradeoffs for parallel boosting are more subtle. Parallelized boosting learns better predictors after seeing the same number of examples, because partial information learned by one processor is effectively shared with others early in the learning process. On the other hand, parallelized boosting does require more total computation to learn from the same number of examples. This extra cost arises from the communication overheads of the parallel algorithm, and from the extra voting required – but it grows fairly slowly with the number of processors.

We also discovered that the amount of agreement between predictors learned by different processors (the threshold) that is required to classify an example as "easy" is remarkably small. We had expected that the greatest accuracy would be achieved by calling examples "easy" only when most or all predictors agreed about them – in fact, it suffices for only 2 or 3 processors to agree. There are indications that this threshold is independent of the total number of processors used.

Parallel boosting is the strategy of choice for parallel ensemble generation of predictors. Using large sample sizes produces accuracies greater than those of the corresponding sequential algorithm, in less elapsed time, although more overall computational time is required.

References

[1] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36:105–142, August 1999.

[2] C. Bishop. *Neural networks for pattern recognition.* Oxford University Press, 1995.

[3] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[4] L. Breiman. Bias, variance and arcing classifiers. Technical Report 460, Department of Statistics, University of California, Berkeley, 1996.

[5] L. Breiman. Arcing classifiers. *Annals of Statistics*, 26(3):801–849, 1998.

[6] L. Breiman. Pasting bites together for prediction in large data sets and on-line. *Machine Learning*, 36(1&2), 1999.

[7] Leo Brieman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth International Group, 1984.

[8] Y. Freund and R. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*, pages 148–156, 1996.

[9] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.

[10] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.

[11] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan-Kaufmann, 1993.

[12] R.O. Rogers and D.B. Skillicorn. Using the BSP cost model to optimize parallel neural network training. *Future Generation Computer Systems*, 14:409–424, 1998.

[13] D.B. Skillicorn. Parallel predictor generation. In *Proceedings of a Workshop on Large-Scale Parallel KDD Systems, KDD'99*, number 1759 in Lecture Notes in Artificial Intelligence, pages 190–196. Springer-Verlag, 2000.

[14] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[15] Y. Wang and D.B. Skillicorn. Parallel inductive logic for data mining. In *Workshop on Distributed and Parallel Knowledge Discovery, KDD2000*, Boston, to appear. ACM Press.