Machine Learning 36, 85–103 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

Pasting Small Votes for Classification in Large Databases and On-Line

LEO BREIMAN leo@stat.berkeley.edu
Statistics Department, University of California, Berkeley, CA 94708

Editors: David Wolpert, Philip Chan and Salvatore Stolfo

Abstract. Many databases have grown to the point where they cannot fit into the fast memory of even large memory machines, to say nothing of current workstations. If what we want to do is to use these data bases to construct predictions of various characteristics, then since the usual methods require that all data be held in fast memory, various work-arounds have to be used. This paper studies one such class of methods which give accuracy comparable to that which could have been obtained if all data could have been held in core and which are computationally fast.The procedure takes small pieces of the data, grows a predictor on each small piece and then pastes these predictors together. A version is given that scales up to terabyte data sets. The methods are also applicable to on-line learning.

Keywords: combining, database, votes, pasting

1. Introduction

Suppose that the data base D is a large collection of examples $(y_n, \mathbf{x}_n)$ where the $\mathbf{x}_n$ are input vectors and the y_n are class labels. What we want to do is use this data to construct a classifier which can accurately assign a class label to future inputs $\mathbf{x}$. But D is too large to hold in the memory of any currently available computer.

What we show in this paper is that the aggregation of many classifiers, each grown on a training set of modest size N selected from the data base D, can achieve almost optimum classification accuracy. The procedure, which we refer to as pasting votes together, has two key ideas in its implementation:

(i) Suppose that up to the present, k predictors have been constructed. A new training set of size N is selected from D either by random or importance sampling. The $(k + 1)$st predictor is grown on this new training set and aggregated with the previous k. The aggregation is by unweighted plurality voting. If random sampling is used to select the training set, the training set is called an *Rprecinct* and the procedure is called *pasting Rvotes* (R = random). If it is done using importance sampling, the training set is called an *Iprecinct* and the procedure as *pasting Ivotes* (I = importance). The importance sampling used (see Section 2) samples more heavily from the instances more likely to be misclassified.

(ii) An estimate $e(k)$ of the generalization error for the kth aggregation $e(k)$ is updated. The pasting stops when $e(k)$ stops decreasing. The estimate of $e(k)$ can be gotten using a test set, but our implementation uses out-of-bag estimation (Breiman, 1996).

CART (Breiman et al., 1984) is used as a test bed predictor, but it is clear that pasting votes will work with other prediction methods. In Section 2, we explore two versions in classification-pasting Ivotes and pasting Rvotes. Using Rvotes is less complicated, but pasting Ivotes gives considerably more accuracy. We experiment on five moderate sized data sets. The accuracy of pasting Ivotes is compared to trees, single and multiple, grown using the entire data set.

Section 3 applies a version of pasting Ivotes designed to minimize disk accesses to a synthetic data base of a million records, each containing data on 61 variables. The total disk access and read times were tallied together with the tree construction times and the time needed to construct the Iprecincts. They had similar magnitudes. An analysis shows that this version of pasting Ivotes scales up to terabyte data bases. In Section 4 pasting Ivotes is applied to on-line learning using a synthetic data set as a test bed. Comments and conclusions are given in Section 5 and a look at related work in Section 6.

2. Pasting votes

2.1. Method description

The simplest version of pasting is to select each training set of size N by random sampling from the data base D, grow the classifier, repeat a preassigned number K of times, stop and aggregate the classifiers by voting. This is certainly workable and cheap. If, after aggregation, accuracy is checked on a test set, then further runs can be used to optimize the values of K and N. A more sophisticated version estimates $e(k)$ after the kth aggregation and stops when $e(k)$ stops decreasing.

There are three methods that can be used to estimate $e(k)$:

First: Set aside a fixed test set of examples in D. Run the kth aggregated classifier on the test set. Estimate $e(k)$ by the error on this test set.

Second: If T is the $(k+1)$st training set, let $r(k)$ be the error rate of the kth aggregated classifier on T. Since N is small, $r(k)$ with be a noisy estimate of $e(k)$. Smooth it by defining $e(k) = p * e(k-1) + (1-p)^* r(k)$. The value $p = 0.75$ was used in all of our experiments, but results are not sensitive to the value of p.

If the total number of examples used in the repeated sampling of training sets gets above an appreciable fraction of the number in D, the second estimate will be biased downward because some of the examples in the $(k+1)$st training set will have been used to construct the previous classifiers.

Third: To eliminate the bias in the second method, if T_h is the hth training set, and $\mathrm{C}(\mathbf{x}, h)$ the classifier for input vector $\mathbf{x}$ constructed using T_h, then classify an example $(y, \mathbf{x})$ that is a candidate for the $(k+1)$st training set by aggregating all of the classifiers $\mathrm{C}(\mathbf{x}, h)$, $h < k+1$, such that $(y, \mathbf{x})$ is not in T_h. This is the out-of-bag classifier $\mathrm{C}^{\mathrm{OB}}(\mathbf{x}, k)$. Estimate the error $r(k)$ as the proportion of misclassifications made by C^{OB}. Smooth the $r(k)$ as in the second method to get the estimate $e^{\mathrm{OB}}(k)$.

Out-of-bag (OB) estimation is shown in Breiman (1996) to give effective estimates of the generalization error. (Regarding OB estimation, see also Wolpert (1996) and

Tibshirani (1996)) In the examples we run, both the test set $e^{\text{TS}}(k)$ and the estimate $e^{\text{OB}}(k)$ are computed and compared. Also compared are two methods for selecting the examples for the $(k+1)$st training sets.

Random: This is simple random selection from D with all examples having the same probability of being selected. Continue until N examples are selected. The classifier grown on this training set is called a Rvote.

Importance: In this procedure, an example $(y, \mathbf{x})$ is selected at random from D with all examples having the same probability of being selected. Let $\text{C}^{\text{OB}}(\mathbf{x}, k)$ be the out-of-bag classifier at stage k. If $y \neq \text{C}^{\text{OB}}(\mathbf{x}, k)$ then put this example in the training set. Otherwise, put it in the training set with probability $e(k)/(1-e(k))$. Repeat until N examples have been collected. Refer to the classifier grown on this training sets as an Ivote.

The rationale for the importance sampling procedure is that the new training set will contain about equal numbers of incorrectly and correctly classified examples. The probability that the next instance sampled is incorrectly classified and put into the training set is $e(k)$. The probability that the next instance sampled is correctly classified and put into the training set is also $e(k) = (1-e(k)) * [e(k)/(1-e(k))]$. The sampling probabilities change as more classifiers are pasted together. In this respect it is an arcing algorithm, where arcing is an acronym standing for **a**daptive **r**esampling and **c**ombin**ing** (Breiman, 1998).

An apparent question is: if one wants to get equal numbers of misclassified and correctly classified instances into a training set of size N, why not sample until we get $N/2$ of each? One answer is that for $e(k) < 1/2$, the expected number of instances we have to sample to accomplish this is about the same as the expected number needed using the rejection sampling. The rejection sampling adds a randomness to the training set selection that may be important in sequential sampling from large data bases (see Section 3).

Although the algorithm for importance sampling used here differs essentially from those used in the Breiman (1998) paper where the entire data base was used to grow each classifier, it retains the basic idea—put increased weight on those examples more likely to be misclassified. The first effective arcing algorithm is due to Freund and Schapire (1995, 1996) and named Adaboost (see also (Drucker & Cortes, 1996; Quinlan, 1996; Breiman, 1997). An idea similar to pasting Ivotes was suggested by Schapire (1990) in the context of boosting in PAC learning, but used a sequence of training sets increasing in size.

In our experiments, N is taken to be a few hundred examples out of data sets having up to 43,500 examples; In four of the data sets we use in our experiments Adaboost-CART was shown to have a lower overall error rate than any of the other 22 well-known classification methods reported on in the Statlog Project (Michie, Spiegelhalter, & Taylor, 1994). The fifth data base is the famous Post Office handwritten digit data on which Adaboosted CART is competitive with hand tailored neural nets.

Two surprising and gratifying things emerge in the experiments:

(1) *Pasting together Ivotes, each one grown using only a few hundred examples, gives accuracy comparable to running Adaboost using the whole data set at each iteration.*

(2) *The computing times to construct the pasted classifiers are very nominal, making the procedure computationally feasible even for data bases much larger than fast memory.*

Pasting never requires storage of the entire data base D in core. Examples are selected, tested, and pulled out to form the small training sets. The K trees constructed to date need to be stored. This takes about 14KN bytes—very workstation feasible. The trees produced are not pruned—the aggregation seems to eliminate the overfitting. Each tree construction takes order $MN \log N$ flops where M is the number of input variables.

The selection of the small training sets adds a computational burden. Assuming that the estimated error $e = e(k)$ is about equal to the true error rate over the whole database and $e < 1/2$, the expected number of instances sampled from the database to form the kth training set is $N/2e$ (see Appendix). To decide whether to accept an instance, it must be run through all trees constructed to date that do not use the instance in the training set. This is accomplished as follows: For each instance, we store an integer N_L equal to the tree number when the instance was last run through the previous trees together with J integers $nc(1), \ldots, nc(J)$ where $nc(j)$ is the number of times the instance was classified as class j in the run through the trees constructed prior to N_L.

Then, to update the $nc(1), \ldots, nc(J)$ the instance is passed through the trees constructed from tree N_L up to the last tree constructed. If N_B is the number of instances in the entire database, then the expected number of flops needed to form the training set for each tree construction is proportional to $\log(N)\ N_B$ (see Appendix). For large N_B, this becomes appreciable. But keeping track of which trees to run the instances through requires only the updating of N_L and the J-vector **nc** for each instance selected in forming the current Iprecinct.

2.2. Pasting Ivotes

The data sets used these experiments are summarized in Table 1.

The first four of these data sets were used in the Statlog project and are described in Michie, Spiegelhalter, and Taylor (1994). The last data set is the well-known handwritten digit recognition data set. The data exists as separated into test and training set. The results of 10cv-CART and Adaboost-CART are given in Table 2. On the digit data 100 iterations were used in Adaboost. The other Adaboost results are based on 50 iterations.

To track the effect of N, the size of the training sets, we ran pasting on all data sets with $N = 100, 200, 400, 800$. The number of total iterations was chosen, in each data set, to be

Table 1. Data set summary.

Data set	Classes	Inputs	Training	Test
letters	26	16	15000	5000
satellite	6	36	4435	2000
shuttle	7	9	43500	14500
dna	3	60	2000	1186
digit	10	256	7291	2007

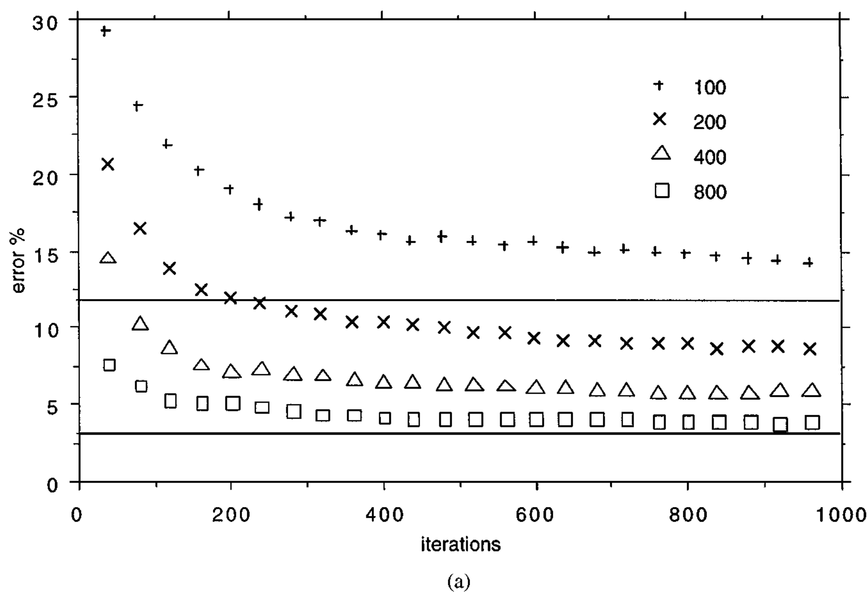
Table 2. Test set error (% misclassification).

Data set	10cv-CART	Adaboost-CART
letters	12.4	3.4
satellite	14.8	8.8
shuttle	0.062	0.007
dna	6.2	4.2
digit	27.1	6.2

past the point of decreasing test set error. In letters, this was 1000, 500 for the digits data, 250 for the satellite data, 100 for the dna data, and 50 for the shuttle data. In these runs, we kept track of the test set error, the OB estimates and compute times.

2.2.1. Test set error. The test set error $e^{\mathrm{TS}}(k)$ is computed by running the set aside test set through the classifiers pasted together after the kth Ibite. These are shown in figure 1(a–e) which give graphs of the test set error $e^{\mathrm{TS}}(k)$ at the kth iteration as a function of k. Each graph contains plots of $e^{\mathrm{TS}}(k)$ for $N = 100, 200, 400, 800$. These can usually be distinguished by the fact that for a given value of k, test set error is highest for $N = 100$, and decreases to $N = 800$. The higher horizontal line is the error rate for 10cv-CART; the lower for Adaboost-CART.

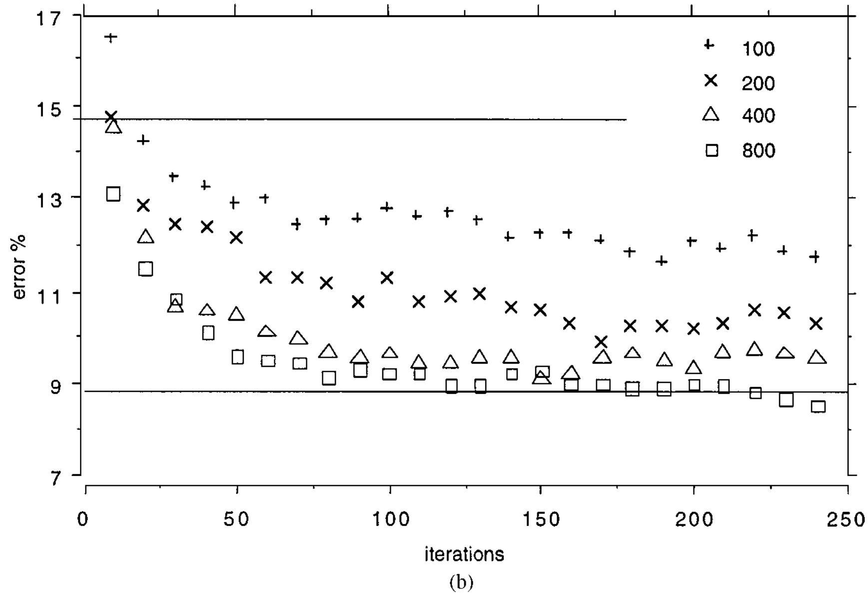
These conclusions can be read from the graphs:



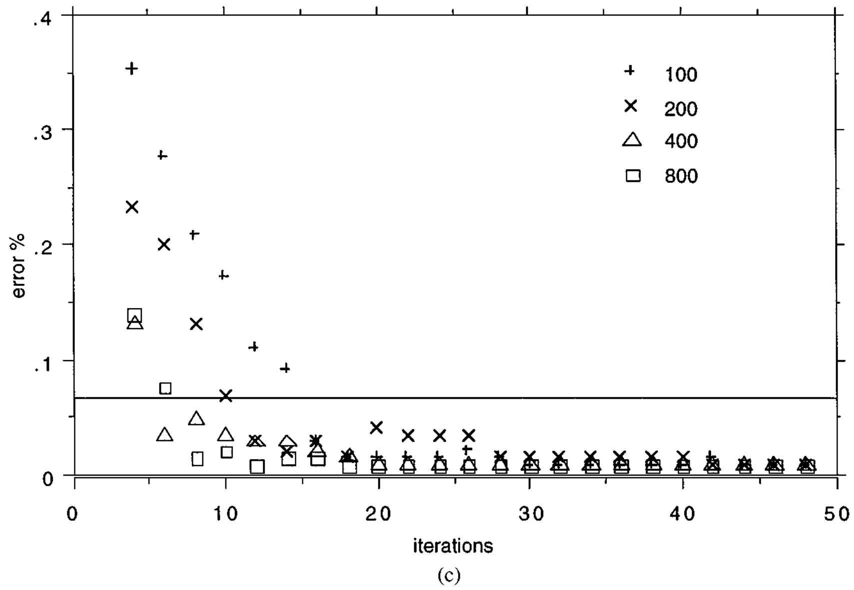
(a)

Figure 1. Test set error (%): (a) letter data; (b) satellite data; (c) shuttle data; (d) dna data; (e) digit data.

(*Continued on next page.*)



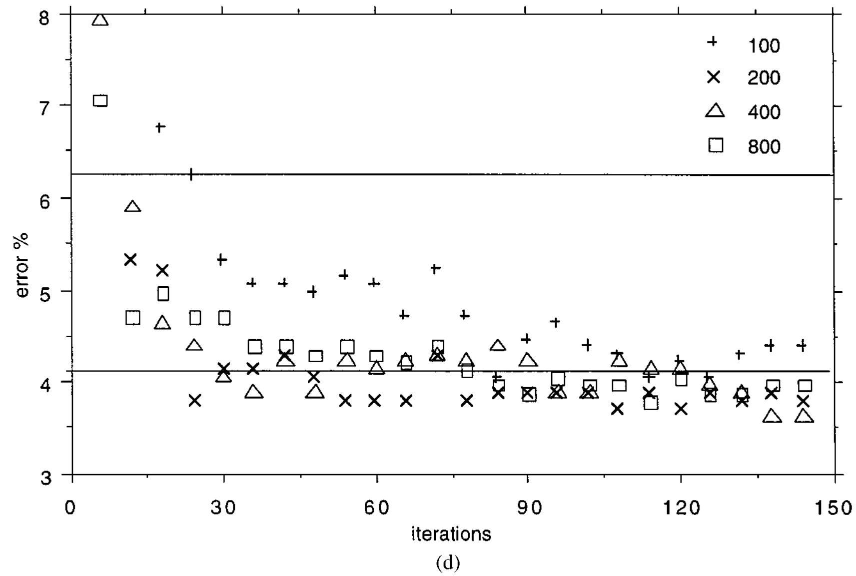
(b)



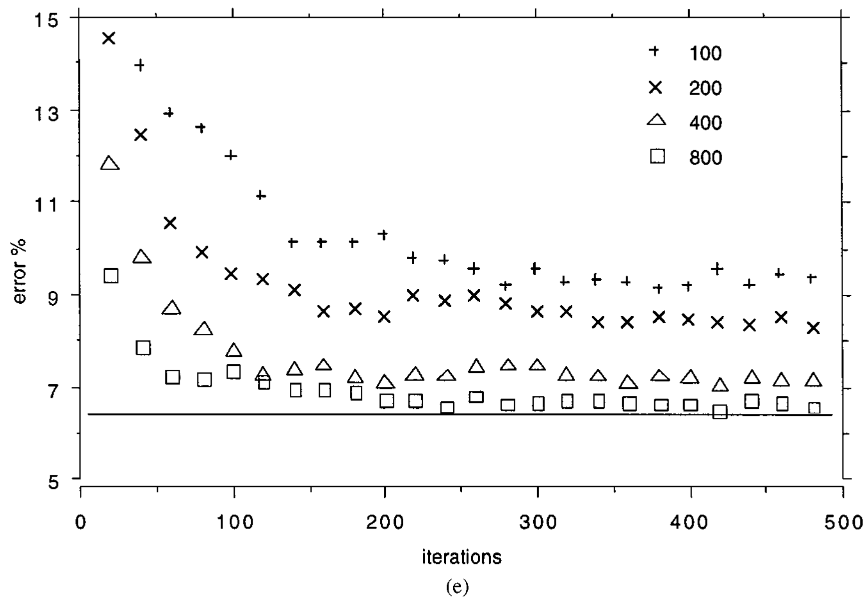
(c)

Figure 1. (*Continued*).

(*Continued on next page.*)



(d)



(e)

Figure 1. (*Continued*).

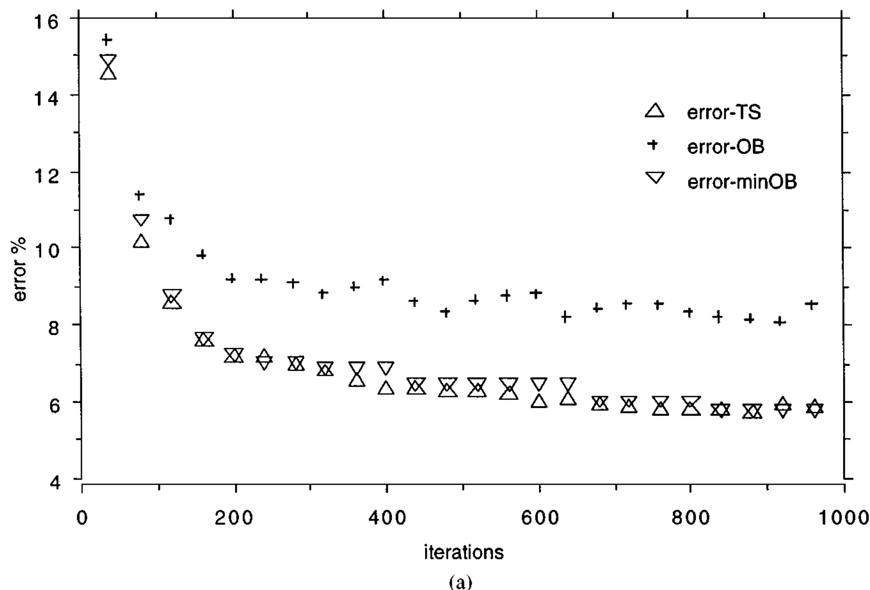
(i) The test set error falls rapidly as the number k of iterations increases, and then reaches an asymptotic value for large k.

(ii) The accuracy in pasting Ivotes together is similar to that of Adaboost, especially for the larger values of N. For instance, the test set error percentages at the end of the $N = 800$ runs are: letters 3.8%, satellite 8.7%, shuttle .007%, dna 3.8%, digit 6.5%.

(iii) In all of the data sets, using even the smallest training sets ($N = 100, 200$) gave test set error comparable to or lower than 10cv-CART after a small number of iterations.
(iv) The effect of increasing N is data dependent. In the dna and shuttle data sets, $N = 100$ gives the same accuracy as $N = 800$. Generally, the asymptotic value is approached faster in the large N runs.
(v) In all data sets, the major decrease in test set error has occurred by 100 iterations. In the letters data set, it appears as though the error is decreasing out to $K = 1000$, although the decrease is small after $K = 200$. In the other data sets, the decrease has stopped by $K = 200$.

2.2.2. Monitoring and selecting using the OB estimates. Suppose that the out-of-bag test set estimates $e^{\text{OB}}(k)$ are used to monitor the pasting procedures in the sense that we stop when the values of $e^{\text{OB}}(k)$ become flat and select the pasted classifier corresponding to the lowest value of $e^{\text{OB}}(k)$ seen to date. Then if we decide to stop after k iterations, the true test set error will be $e^{\text{TS}}(h(k))$ where $h(k) = \text{argmin}\{e^{\text{OB}}(h), h = 1, \ldots, k\}$. The loss in using this method depends on how close or far apart the values of $e^{\text{TS}}(k)$ and $e^{\text{TS}}(h(k))$ are.

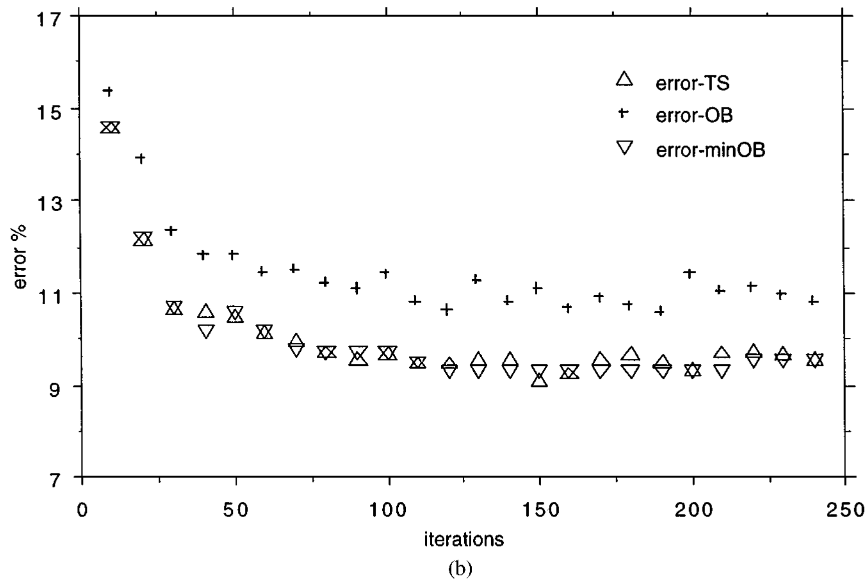
Figure 2(a–e) give plots of $e^{\text{OB}}(k)$, $e^{\text{TS}}(k)$, and $e^{\text{TS}}(h(k))$ vs. k for each of the five data sets for $N = 200$. The plot of $e^{\text{OB}}(k)$ can be recognized as the noisiest. The plots of $e^{\text{TS}}(k)$ and $e^{\text{TS}}(h(k))$ lie of top of each other. In all five data sets $e^{\text{TS}}(k)$ and $e^{\text{TS}}(h(k))$ differ very little. Using the out-of-bag estimates to select a pasted classifier works well. However, one thing that emerges from these graphs is that while e^{OB} tracks e^{TS} quite well as k increases, it may be systematically low or high.



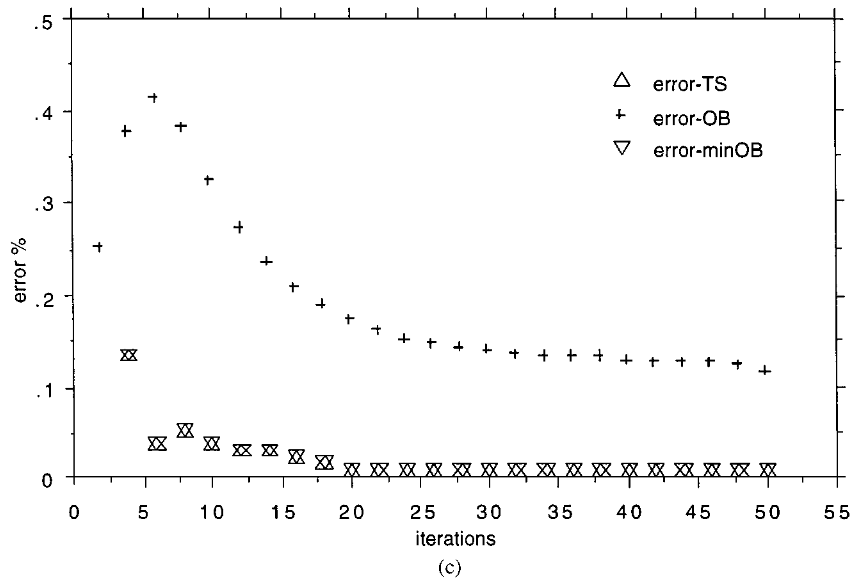
(a)

Figure 2. Test set, OB, and minimum OB error estimates: (a) letter data; (b) satellite data; (c) shuttle data; (d) dna data; (e) digit data.

(*Continued on next page.*)



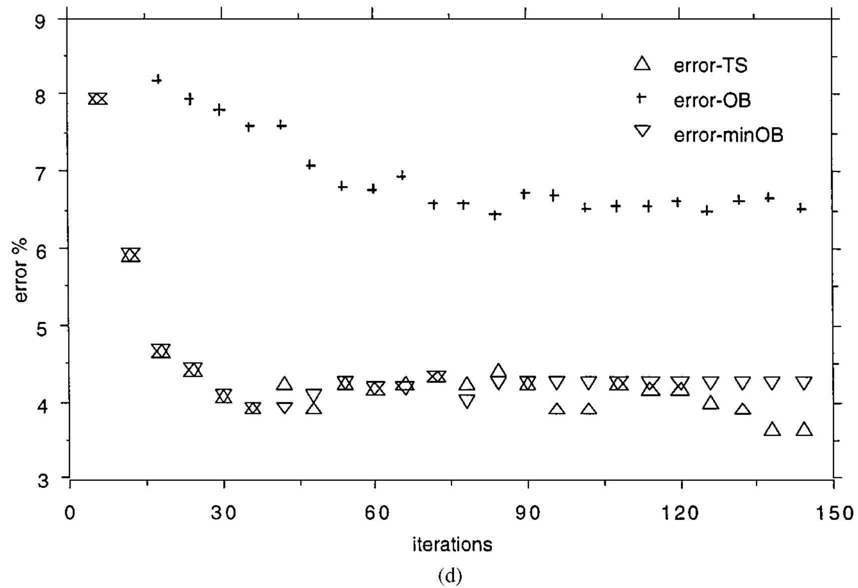
(b)



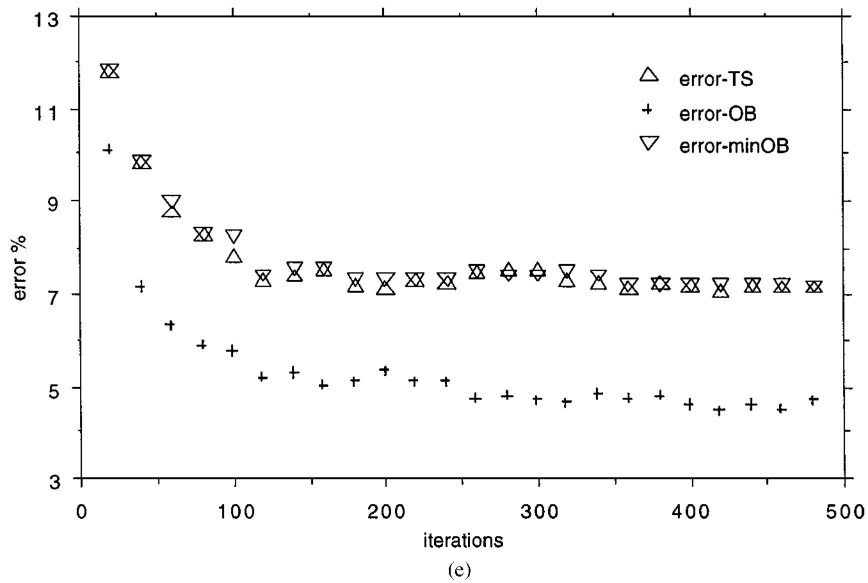
(c)

Figure 2. (*Continued*).

Initially, we thought that this bias might be due to the fact that for the same classifier, the "true" test set classification error may differ from the "true" training set classification error. That is, the data sets used above come already separated into training and test sets. It may be that the test set is intrinsically either more or less difficult to classify than the training



(d)



(e)

Figure 2. (*Continued*).

set. We ran a test of this hypothesis by interchanging test and training sets. The result is that if the OB estimate is biased high on the training set, it is also biased high on the test set. Our conclusion, after additional research, is that the bias comes from some complex interaction of the data set and the importance sampling.

In the initial stages of pasting, the out-of-bag estimates are usually too high. One obvious cause is that since $e^{\mathrm{OB}}(k)$ is a smoothed version of past out-of-bag estimates, it reflects the earlier and higher values of the error. The less obvious is this: in data sets of moderate size with low misclassification error, examples that are prone to be misclassified are used over and over again in the Iprecinct training sets. These examples will tend to be out-of-bag in a relatively small number of the Ivotes. Therefore, their misclassification rate will be elevated. Another question in using Ivotes is how big to take N. In general, it seems that the bigger, the better. But taking N larger also slows down the compute time. The out-of-bag estimates can be used to resolve this issue, since modulo a possible offset due to systematic bias, they will track the true test set error consistently. That is, looking at the out-of-bag estimates for different N will give a fairly accurate idea of how much one buys by using larger N.

2.2.3. Compute times. Table 3 gives the compute times per 100 iterations for the five data sets by training set size. The times are scaled to a SUN Ultrasparc 2 and do not include the time to load the data.

These times were computed from the total elapsed time of the run. Thus, if the run was 500 iterations, the time was divided by 5.

2.3. Pasting Rvotes

Pasting Rvotes does not work as well as pasting Ivotes. To illustrate, random sampling was used on the five data sets to form training sets of size $N = 200$. The number of iterations was set at the upper limits specified in Section 2.2. The final test set error was divided by the corresponding test set error for pasting Ivotes. These ratios are given in Table 4.

These ratios are disappointingly large and show that pasting Rvotes is not competitive with pasting Ivotes in terms of accuracy. Still, except for the shuttle data, pasting Rvotes has lower test set error than 10cv-CART.

Table 3. Compute time per 100 iterations (min).

Data set	$N = 100$	200	400	800
letters	15	0.24	0.59	0.84
satellite	0.06	0.12	0.26	0.55
shuttle	0.40	0.48	0.62	0.70
dna	0.10	0.18	0.43	0.47
digit	0.34	0.92	1.94	3.61

Table 4. Ratio of test set errors (Rvotes/Ivotes).

letters	satellite	shuttle	dna	digit
2.41	1.32	73.95	1.26	1.68

2.4. Of-bag can't be ignored

Keeping track of which examples are in and out-of-bag is a simple but extra complication. A natural question is what happens if this complication is ignored and all incoming examples treated as if they had never been seen—that is, everything is out-of-bag. If the database is semi-infinite, so that duplicate examples in the training sets used in the pasting is infrequent, then this works. But if the database is finite in the sense that duplication is not infrequent, then performance degenerates.

One consequence is that the error, being a resubstitution estimate, is increasingly biased low and generally goes to zero as the iterations continue. Then the Iprecincts have to search more instances to find enough misclassified ones and the procedure bogs down. For instance, using the letters data and $N = 400$, it bogged down at around 90 iterations with a test set error of 10.3, almost twice as large as the error gotten using out-of-bag.

3. Minimizing disk access—An alternative version

A point strongly made by the referees is that the timings given in Section 2 excluded the many random disk accesses needed and that the time taken by these would swamp the cpu times. This point was valid and to deal with it, the following algorithm was constructed. Let a record consist of all data for a single instance.

(a) read a record
 if at eof, rewind
 check to see if instance is acceptable
 if the number of instances accepted is $< N$, goto (a)
 construct tree
 goto (a)

The algorithm stops after a specified number of trees have been constructed or after a specified number of epochs, where an epoch consists of running through the entire database from beginning to a rewind.

The records are read sequentially, so no random accesses are required. To get an idea of the times needed in a larger database, one million instances of synthetic data with 61 input variables and 10 classes was generated, comprising about 250 Mbyte. For a description of this data, see the Appendix. We set $N = 1000$ and ran for one epoch with 310 trees constructed. On a 10,000 instance test set the error rate dropped to 17.1%. (Constructing a single tree using 100,000 instances resulted in 21.1% test set error). We kept track of the disk read and access times, the time to select instances for the training sets and the time for tree construction. These are tabled below for one epoch.

The machine used is a 250 MHz Macintosh. The total time for the epoch was about 50 min, and the dominant factor is not the disk reads, but the selection cpu usage. The select time is sublinear at the first, and then becomes linear in the number of trees. The disk read and tree construction times are linear. Let $R = N_B/N$. Per epoch the select time T_S is proportional to $2eN\log(N)R^2$. The tree time T_T is proportional to $2eM\log(N)N_B$. Dividing gives $T_T/T_S = cM/R$ (see Appendix). From Table 5, it appears that c is about 3.

Table 5. Accumulated times—one epoch (sec).

Disk time	Selection time	Tree time	Total
797	1829	333	2949

3.1. Scaling up

To get some idea of how the above algorithm scales up, assume a database with $N_B = 10^8$ records such that each record contains data for $M = 10^3$ variables. Assuming numerical variables, this is about half a terabyte. Also assume that $2e = .1$. From Table 5, it took 333 seconds to construct trees for one epoch with $e = 0.2$, $M = 61$, $N = 1000$, and $N_B = 10^6$. Thus, the tree construction time for an epoch of the larger data set using $N = 10^5$ is about $T_T = 62.5$ h on my Macintosh. Making the assumption that we are using a server five times as fast as my machine gives an epoch tree building time of about 12 h.

Using cM/R for the ratio T_T/T_S with $c = 3$ leads to $T_S = .33T_T$. So add 4 h to the running time. Allow about the same magnitude for disk read time, say 16 h. Then, the time needed is about 32 h per epoch—reasonable for a half terabyte database. This is only about double the time needed to sequentially access the entire database. No other algorithm that accesses the entire database can improve on this time by more than 50%.

These results depend on the $MN\log(N)$ time for tree construction. The Appendix gives justification. One consequence of the $MN\log(N)$ time is that even if there was a machine with a terabyte of RAM, constructing a tree using the entire database would take over 11 days on a server five times as fast as my machine.

Here is how memory scales up. Keeping track of N_L and $nc(1), \ldots, nc(J)$ requires about $2JN_B$ bytes. For $J = 2$, this comes to about 0.4 of a Gbyte. The training set consists of 10^5 records, each containing 1000 4-byte variables—another 0.4 Gbyte. Storing all trees in an epoch costs about N_B bytes—100 Mbytes. The only memory requirements that expand as we go to multiple epochs is the number of trees stored. But this is the least memory extensive requirement. Since gigabyte servers are getting common nowadays, the scaling on memory is reasonable.

3.2. Accuracy of the alternative

An important question is how much accuracy is lost using this less randomized alternative (version 2). Our experimental results indicate that the loss is small. We ran version 2 on the databases used in Section 2. Training set sizes of 800 were used and the algorithm run for the same number of trees as used by version 1 in reaching the optimum result. A comparison of the test set errors at the end of the runs is given in Table 6.

3.3. A caveat

Version 2 will not work if the database has a highly non-uniform distribution of classes. For instance, if all class 1 instances come first in the database. If this structure is known, it

Table 6. Comparison of test set error (%).

Data set	Version 1	Version 2
letters	3.8	4.3
satellite	8.7	9.4
shuttle	0.007	0.000
dna	3.8	5.4
digit	6.5	6.4

can be dealt with using additional disk accesses. For instance, a sequential disk read could be used to accept $N/2$ class 1 instances, and then jump to the last accessed class 2 instance and do a sequential read to accept the other $N/2$ instances. This costs two disk accesses per tree constructed.

4. On-line learning

In situations where there is a steady flow of new examples being formed, incremental or on-line learning research has focused on the continual updating of a prespecified architecture. For instance, given a flow of examples how is a neural net with specified architecture updated as each new example occurs? Or given a binary tree structure, find efficient ways to update the tree as new examples are added. There has been research on incremental decision tree building starting with Utgoff (1989).

What we propose instead is to do on-line learning by the steady pasting on of new Ivotes. Thus the architecture grows as the information flows in. The drawback is that the storage requirements cannot be set in advance. Ivotes are added until an asymptote is reached. But this is offset by two important advantages—accuracy and speed. As seen in the previous section, pasting Ivotes gives generally low test set error. Further, the compute times required per example are small.

In on-line learning the possibility of a sampled example being sampled again is zero. This simplifies the algorithms. In forming test set error estimates, one can assume that none of the incoming examples have been previously used in a training set. Similarly, in deciding whether an incoming example is correctly classified or not, we can assume that it has not previously been used as a training example.

The on-line procedure generates training sets of size N in the way similar to the Section 2 method. But instead of dealing with a fixed database D, there is a flow of incoming examples. As the new examples come in, they are checked to see how they are classified by the current pasted together classifier. If misclassified, they are accepted into the training set. If not, they are accepted with probability $e^{\mathrm{TR}}/(1 - e^{\mathrm{TR}})$. Continue this procedure until there are N examples in the training set. The error estimates $e^{\mathrm{TR}}(k)$ are computed as the smoothed version of $r(k)$, where $r(k)$ is the proportion of misclassified examples found in forming the kth training set.

To study on-line performance, 10 class, 61 input synthetic data was manufactured (see Appendix). Before going on-line, a test set of 5000 examples was generated. At the kth

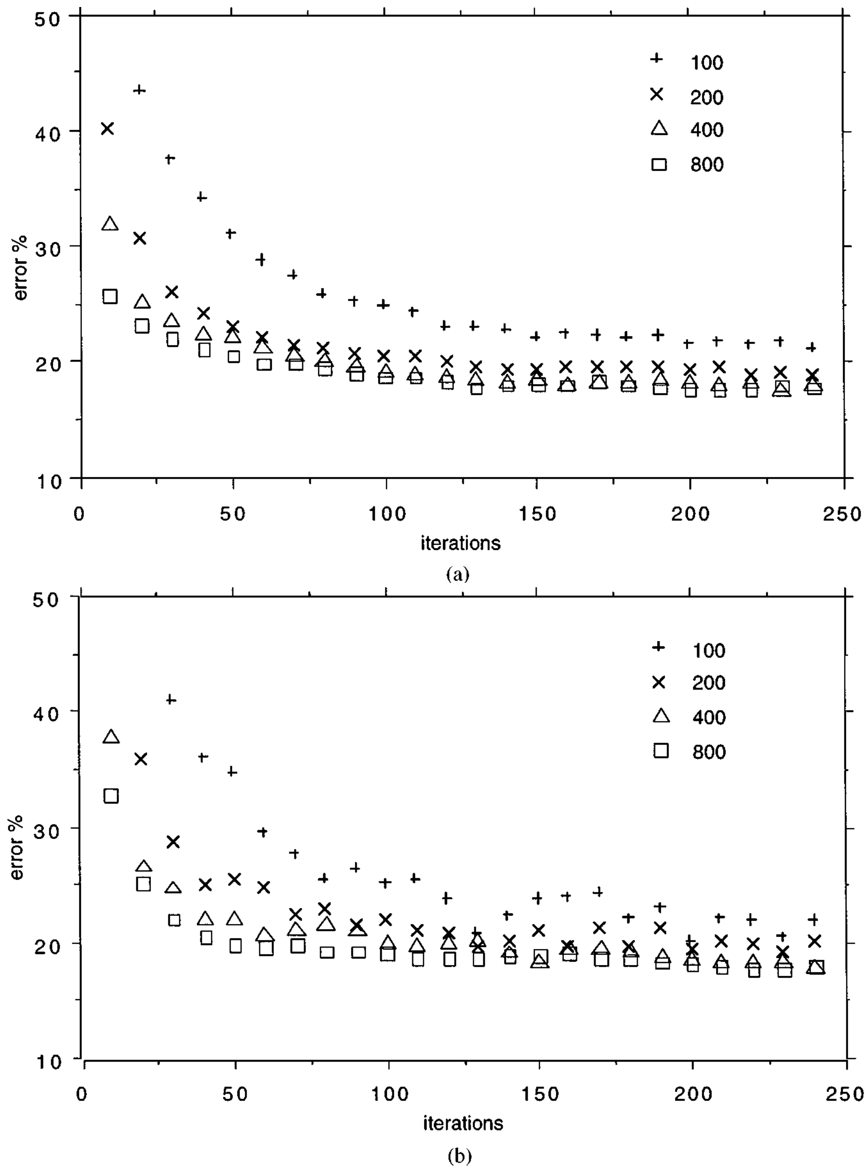


Figure 3. On-line synthetic data: (a) test set error (%); (b) training set error estimates (%).

iteration, both $e^{\mathrm{TS}}(k)$ and $e^{\mathrm{TR}}(k)$ are computed. Again, we use $N = 100, 200, 400, 800$. Figure 3(a) shows the test set errors vs. the number of iterations out to 250 iterations. All the test set plots show a rapid decrease to an error of about 20% followed by a slow decrease. Although we stopped at 250 iterations, the test set error was still slowly decreasing. To see what happens if we kept going, we did the $N = 400$ pasting out to 750 iterations. The

test set error dropped from 17.7 at 250 to 16.6 at 750. In application, there is usually no test set available, and the stopping decision must be made on the basis of the e^{TR} values. Figure 3(b) plots the values of e^{TR} for the 250 iterations. The values are noisy and need more smoothing, but generally agree with the test set values.

4.1. A modification to bound compute times and memory requirements

At the $(k + 1)$st iteration each candidate example for the training set has to be passed down k trees. The compute time to do this for the kth iteration increases linearly with k, and the total compute time increases quadratically with k. Furthermore, the memory used grows unboundedly.

This is not as problematic with finite data sets, since the out-of-bag condition bounds the number of trees used to classify each candidate example and the size of the run is limited. To explore a possible remedy for the problem in on-line learning, we revised the procedure so that the Iprecinct training set is selected based only on the misclassifications in the pasting together of the last K_B trees where K_B is set by the user. The compute times now increase linearly. At 500 iterations the total compute time using $K_B = 100$, and $N = 400$ is 8.3 min compared to 22.0 min for the unrestricted procedure.

The final test set error is similar—17.2% for the unrestricted vs. 17.4% for the restricted version. One drawback is that the e^{TR} estimates in the restricted arcing now reflect the accuracy of the last K_B Ivotes, instead of all Ivotes. Therefore they have a pessimistic bias. In the original procedure, the final value of e^{TR} is 18.0%. In the restricted procedure it is 19.8%. Further experiments are needed to see if the K_B restriction gives generally satisfactory results and whether the pessimistic bias in e^{TR} can be corrected. If it works, then the memory requirements for the on-line learning can be bounded.

5. Comments and conclusions

Our prime conclusion is that pasting small Ivotes together is a promising approach to constructing classifiers for large data sets and for on-line learning. On the data sets we experimented with, it is fast, accurate, and has reasonable memory requirements. It will scale up to terabyte databases. Its performance raises interesting issues and possibilities.

The increase in accuracy using Ivotes as compared to Rvotes is newsworthy. It emphasizes that in classification, concentrating on the examples near the classification boundaries pays off in terms of reduced generalization error. The importance sampling approach used here is differs from the Freud-Schapire algorithm. In Adaboost sampling weights for the entire training set are updated in terms of the classifications done by the last classifier constructed. In pasting Ivotes, the sampling for the new training set is done on the basis of the classifications of the entire past sequence of classifiers and the current estimate of test set error.

The process of creating Iprecincts does not depend on the classifier used—it is simply a method of forming the successive training sets. This leads to a question that will be explored in the near future: suppose that there are a number of classifications methods available in our repertoire, i.e., trees, nearest neighbor, neural nets, etc. Suppose that each of these is

run on the Kth Iprecinct and one is selected to paste onto the previous classifiers. Can this significantly upgrade performance over the use of a single type of classifier?

6. Related work for large databases

There has been much interest lately on prediction in large databases. A good survey of is given by Provost and Kolluri (1997a, 1997b). Interesting related work appears in Chan and Stolfo (1997a, 1997b). Their approach, called meta-learning, is to split the data set into pieces, run a classifier on each piece and then to grow a tree that arbitrates or combines the different classifiers. Meta-learning is shown to give good accuracy on two moderate sized databases, but no timings are given.

Provost and Hennessey (1996) distribute the data over multiple workstations, run a rule learning program on each and then use an algorithm to select a subset of the rules generated. Timings were done on what they characterized as a massive database—over 1,000,000 records with 31 variables per record. Using 5 Sparc10 workstations, the run took about 1200 sec. Since their database is about half of the size of the synthetic database used in Section 3, the times are comparable. Modifications of the basic algorithm are suggested that cut the computing time significantly. No data was given concerning the accuracy of the basic or modified algorithm. This paper also contains a nice summary of earlier work.

Breiman and Spector (1994) distributed construction of CART out to a workstation network where each workstation owned a subset of the variables. The vector of the values of each variable was sorted once and converted to ranks which eliminated the need for further sorting at the nodes. At each node, each workstation searched its variable rankings for the best split and reported the decrease in Gini due to that split to the master machine. The master machine reported the best split to all slave machines, and they rearranged the their variable rank values to correspond to the two new nodes. Unfortunately, with a 10 Mbyte network, the communication time proved to be a bottle neck and performance degenerated if more than five machines were used. We plan to rerun on our recent 100 Mbyte network.

Shafer, Agrawal, and Mehta (1996) design SPRINT—a clever parallel version of decision tree construction based also on sort once at the top. To get timings they generated a synthetic 1,600,000 instance database with 9 variables per instance. Using a 16 node IBM SP 2 Model 9076 with the cpu's running at 62.5 MHz, growing a tree took 375 sec. On the same database pasting Ivotes, using training sets of size 1600, took 242 sec for an epoch (22 trees) on the 250 MHz Macintosh. Of this 242 sec, 71% was disk read time, 27% the select time and 2% the tree construction time. Their paper does not specify accuracies so comparison is not possible.

Appendix

(a) *Synthetic data*: Let $w(k, m)$ be a triangular function in m, with a maximum of 10 at $m = 10k$, becoming zero at $10k - 10$, $10k + 10$, and zero for m outside this range.

There are 10 distinct subsets of size 3 of the integers {1, 2, 3, 4, 5}. Assigning each of these subsets to a class and denote by $T(j)$ the subset assigned to class j. Then the 61 dimensional inputs corresponding to class j are given by:

$$x(m) = \sum_{k \in T(j)} z_k w(k, m) + u_m$$

where the u_m are independent mean zero normals with $\text{sd} = 10$, and the z_k are independent mean zero normals with $\text{sd} = 1$.

The data for each of the classes has a multivariate 61-dimensional mean-zero normal distribution. The optimal classification scheme for this data is quadratic discrimination, and the curved classification boundaries are hard for trees to approximate.

(b) *Tree construction is MNlog(N) cpu time*: My present version of CART, which dates from Breiman and Spector (1994) sorts all variables only at the root node using Quicksort and replaces them by their ranks. This takes about $MN \log(N)$ flops. Then at each node, the best split is found by a sequential search through the ranks of each variable. This takes time about M^*(node population). At each tree depth, the flops needed to split each node at that depth is MN. Assuming a balanced tree, the tree grows to depth $\log(N)$ adding another $MN \log(N)$ flops.

Some empirical support comes from this fact—evaluate the constant c in $c\,MN \log(N)$ using a training set of size 1000 of my 61 variable synthetic data. Consider running on 10,000 instances of the 9-variable synthetic data used in the Shafer, Agrawal, and Mehta (1996) paper cited in Section 6. Then the predicted time is 2.1 sec. The actual time is 1.5 sec. The faster than predicted run time on the SPRINT data was probably due to the many tied values in each variable. CART does not compute the floating point Gini update on values tied with the value ahead. So the effective number of operations per tree level is smaller than MN.

(c) *Select time*: Assume $e < 1/2$. The expected number of picks until K incorrectly classified examples are selected is K/e. So the number of picks until $N/2$ examples are selected is $N/(2e)$. But in selecting these $N/2$ incorrects, $N/2$ corrects are also gathered. Therefore, the total number of selections needed for a training set of size N is about $L = N/(2e)$. So in constructing each tree, L instances are drawn at random out of N_B.

For any given instance, the probability that it appeared in the last draw is $P = L/N_B$. The probability that it occurred last I draws ago is $P(1 - P)I - 1$. The expected number of draws ago that it last occurred is $1/P$. On average, each instance used in constructing the current tree has to be run through $1/P$ trees. It takes about $\log(N)$ flops to run an instance down a tree built using N instances. To construct a tree, L instances need to be checked. Therefore, it takes time proportional to $\log(N)L/P = \log(N)NR$ to check the instances needed to grow a tree, where $R = N_B/N$. Since construction time for each tree is proportional to $MN \log(N)$, this results in $T_T/T_S = cM/R$.

Acknowledgments

I'm grateful to Yoav Freund and Harold Drucker for numbers of suggestions that led to improvements on the first draft. The comments of the referees and an editor were especially

helpful in making this a better paper. This research was partially supported by NSF Grant 1-444063-21445.

References

Breiman, L. (1996). Out-of-bag estimation, available at ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation.ps.

Breiman, L. (1998). Arcing classifiers. *Annals of Statistics*, *26*, 801–824.

Breiman, L., Freidman, J., Olshen, R., & Stone, C. (1984). *Classification and regression trees*. Wadsworth.

Breiman, L., & Spector, P. (1994). Parallelizing cart using a workstation network. *Proceedings Annual American Statistical Association Meeting*, San Francisco, available at ftp.stat.berkeley.edu/usrs/breiman/pcart.ps.Z.

Chan, P., & Stolfo, S. (1997a). Scalability of hierarchical meta-learning on partitioned data, submitted to the Journal of Data Mining and Knowledge Discovery, available at www.cs.fit.edu/∼pkc/papers/dmkd-scale.ps.

Chan, P., & Stolfo, S. (1997b). On the accuracy of meta-learning for scalable data mining. *Journal of Intelligent Information Systems*, *9*, 5–28.

Drucker, H., & Cortes, C. (1996). Boosting decision trees. *Neural information processing 8* (pp. 479–485). Morgan Kaufmann.

Freund, Y., & Schapire, R. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. http://www.research.att.com/orgs/ssr/people/yoav or http://www.research.att.com/orgs/ssr/people/schapire.

Freund, Y., & Schapire, R. (1996). Experiments with a new boosting algorithm. *Machine Learning: Proceedings of the Thirteenth International Conference* (pp. 148–156).

Michie, D., Spiegelhalter, D., & Taylor, C. (1994). *Machine learning, neural and statistical classification*. London: Ellis Horwood.

Provost, F.J., & Hennessey, D. (1996). Scaling up: Distributed machine learning with cooperation. *Proceedings AAAI-96*.

Provost, F.J., & Kolluri, V. (1997a). A survey of methods for scaling up inductive learning algorithms. *Accepted by Data Mining and Knowledge Discovery Journal: Special Issue on Scalable High-Performance Computing for KDD* available at www.pitt.edu/∼uxkst/survey-paper.ps.

Provost, F.J., & Kolluri, V. (1997b). Scaling up inductive algorithms: An overview. *Proc. of Knowledge Discovery in Databases* (Vol. KDD'97, pp. 239–242).

Quinlan, J.R. (1996). Bagging, boosting, and C4.5. *Proceedings of AAAI'96 National Conference* (Vol. 1, pp. 725–730).

Schapire, R.E. (1990). The strength of weak learnability. *Machine Learning*, *5*(2), 197–226.

Shafer, J., Agrawal, R., & Mehta, M. (1996). SPRINT: A scalable parallel classifier for data mining. *Proceedings of the 22nd VLDB Conference* (pp. 544–555).

Tibshirani, R. (1996). *Bias, variance, and prediction error for classification rules* (Technical Report). Statistics Department, University of Toronto.

Utgoff, P. (1989). Incremental induction of decision trees. *Machine Learning*, *4*, 161–186 .

Wolpert, D.H., & Macready, W.G. (1996). An efficient method to estimate bagging's generalization error. *Machine Learning*, to appear.

Received September 26, 1997
Accepted September 3, 1998