Online Ensemble Learning: An Empirical Study

Alan Fern AFERN@ECN.PURDUE.EDU
Robert Givan GIVAN@ECN.PURDUE.EDU
Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA

Abstract

We study resource-limited online learning, motivated by the problem of conditional-branch outcome prediction in computer architecture. In particular, we consider (parallel) time and space-efficient ensemble learners for online settings, empirically demonstrating benefits similar to those shown previously for offline ensembles. Our learning algorithms are inspired by the previously published "boosting by filtering" framework as well as the offline Arc-x4 boosting-style algorithm. We train ensembles of online decision trees using a novel variant of the ID4 online decision-tree algorithm as the base learner (our ID4 extensions significantly improve ID4 performance), and show empirical results for both boosting and bagging-style online ensemble methods. Our results evaluate these methods on both our branch prediction domain and online variants of three familiar machine-learning benchmarks. The results indicate poor performance for our bagging algorithm, but significant improvements in predictive accuracy with ensemble size for our boosting-style algorithm. In addition, we show that given tight space constraints, ensembles of depth-bounded trees are often a better use of space than single deeper trees.

1. Introduction

Ensemble methods such as boosting and bagging have provided significant advantages in offline learning settings—but little work has been done evaluating these methods in online settings. Here we consider an online setting motivated by the problem of predicting conditional branch outcomes in microprocessors. Like many online learning problems, branch prediction places tight time and space constraints on a learning algorithm due to limited chip real-estate and high processor speeds, making time and space efficiency critical. The application offers cheap parallelism, so our focus is on efficient parallel methods. Our main goal is to demonstrate that familiar ensemble performance gains can be seen in online settings using (parallel) time/space efficient online ensembles.

We consider the simplified problem of online binary concept learning with binary features. It is likely that our methods extend to non-binary problems in ways similar to offline ensemble extensions. We use as a base learner an online decision-tree algorithm extending ID4 (Schlimmer & Fisher, 1986) that we developed for the branch prediction problem. The full paper (Fern & Givan, 2000) gives results showing that our extensions improve ID4 classification accuracy both for single trees and for ensembles.

Due to our resource constraints, we consider online ensemble methods that do not store training instances (though methods that store a few instances may also be feasible)—this rules out directly applying offline methods by storing instances and reinvoking the algorithm.

Freund (1995) describes an online boosting algorithm called online *boost-by-majority (BBM)* for the "boosting by filtering" learning framework—there ensembles are also generated online without storing instances. The BBM algorithm implements a sequential ensemble generation approach—the ensemble members are generated one at a time. In practice, to use such an approach we must address at least two challenging issues: first, how to determine when to stop generating one ensemble member and begin the next (BBM provides a theoretical method using parameters that are generally not known in practice); and second, how to adapt to drifting target concepts, since ensemble members are not updated once they are created. Also, we expect methods that update only one ensemble member per training instance to warm up more slowly than "parallel update" methods. Therefore, we present and evaluate here a variation of the 'boosting by filtering' approach that generates ensemble members "in parallel". There is no "parallel time" cost to this approach in our application.

We describe two such "parallel-generation" online ensemble algorithms: one inspired by offline bagging, and one inspired by offline Arc-x4 (Brieman, 1996b). These methods have an implementation with parallel time complexity for both learning and making predictions that is logarithmic in the number T of ensemble members (critical for our application), and space complexity linear in T, dominated by the space occupied by the ensemble members.

We empirically evaluate our online ensemble methods against instances of the branch prediction problem drawn from widely-used computer-architecture benchmarks, as well as against online variants of several familiar machine-learning benchmarks. Our results show that online Arc-x4 consistently outperforms the online bagging method we

tried, for the problems we consider here. The ensembles of online trees produced by online Arc-x4 "boosting" generally significantly improve the error rate of single online decision-tree learners. We also find that ensembles of small trees often outperform large single trees or smaller ensembles of larger trees that use the same number of total tree nodes (again, important for our application).

This paper is organized as follows. In Section 2 we discuss the motivating problem of branch prediction. In Section 3 we introduce online ensemble learning and our two algorithms. In Section 4 we describe our online decision-tree base learner. In Sections 5 and 6, we give our empirical results for boosting and bagging ensembles.

2. Branch Prediction

This research is motivated by the problem of conditional-branch outcome prediction in computer architecture. It is not our primary goal here to beat current state-of-the-art branch predictors but rather to open a promising new avenue of branch-predictor research. Below we describe the branch prediction problem, aspects of the problem that are interesting from a machine learning perspective, and how this research contributes to branch prediction.

Problem description. Modern microprocessors prefetch instructions far ahead of the currently executing instruction(s), and must accurately predict the outcome of conditional branch instructions encountered during prefetch in order to perform well. Typical programs contain conditional branches about every third instruction, and individual branches are encountered hundreds of thousands of times. For each encounter, the processor predicts the outcome using the processor state during prefetch along with learned state obtained in prior encounters with the same branch. Branch prediction is thus a binary feature space two-class concept learning problem in an online setting.

Qualitative domain characteristics. Several characteristics make branch prediction an interesting and challenging problem from a machine learning viewpoint: first, it is a bounded time/space problem—predictions must typically be made in a few nanoseconds; second, a highly-parallel space-sensitive hardware implementation is required; third, branch prediction requires an online setting where warm-up effects are important (due to process context switching, aliasing[1], and unknown number of instances); fourth, branch prediction provides a fertile source for large automatically-labelled machine-learning problems; fifth, significant progress in branch prediction could have a large impact—reducing branch predictor error rates by even a few percent is thought to result in a significant processor speedup (Chang et al., 1995).

Contribution to branch prediction. The full version of this paper (Fern & Givan, 2000) contains an overview of past and present branch prediction research. Virtually all proposed branch predictors are table based (i.e., they maintain predictive information for each possible combination of feature values) causing their sizes to grow exponentially with the number of features considered. Thus, state-of-the-art predictors can only use a small subset of the available processor state as features for prediction.

The methods we describe avoid exponential growth—our predictors (ensembles of depth-bounded decision trees) grow linearly with the number of features considered. This approach is able to flexibly incorporate large amounts of processor state within architecturally-realistic space constraints, possibly resulting in substantial improvements in the prediction accuracy available for a fixed space usage.

The empirical work in this paper uses the same feature space used by current state-of-the-art predictors (rather than exploit our linear growth in feature space dimension to consider additional processor state)—this is because our immediate goal is to explore the utility of online ensemble methods. This goal also motivates our inclusion of results on familiar machine learning data sets. Future work will explore the use of additional processor state to exceed the state of the art in branch prediction. Similarly, this work does not yet attempt a full empirical comparison to current techniques because architecturally-convincing comparison is enormously computationally demanding.[2]

Additionally, we note that on a chip we cannot dynamically allocate tree nodes, so we must provide space for full-depth decision trees—as a result, our predictors grow exponentially with the tree-depth bound. We show below that using ensembles allows us to more effectively use the limited space by trading off depth for more trees.

3. Online Learning using Ensembles

This research addresses the problem of online concept learning in two class problems with binary features. In online settings, training instances are made available one at a time, and the algorithm must update some hypothesis concept after each example is presented. Given a sequence of training instances, an online algorithm will produce a sequence of hypotheses. It is straightforward to construct an online algorithm from any offline algorithm by arranging to store the training instances seen "so far" and constructing each updated hypothesis from scratch. However, online settings typically have resource constraints that make this direct application of an offline algorithm infeasible. Online learning algorithms are designed to reuse the previous hypothesis to reduce update times.[3] In addition, online algorithms may face space constraints preventing the storage of the entire stream of training instances, or in a distributed setting network bandwidth may limit the ability to consider all the training data at once.

1. Aliasing occurs when one predictor is responsible for predicting the outcomes of instances from two different branches (without knowing which instances come from which branches). Aliasing is a result of space limits forcing fewer predictors than actual unique branches. Context switching also forces a learner to handle multiple branches.

2. Since simulations are carried out on serial machines the cost of running large simulations is proportional to the ensemble size and will take months on current high-performance multiprocessors.

3. Hypothesis reuse is even more important in ensemble algorithms.

Ensemble algorithms provide methods for invoking a "base" learning algorithm multiple times and for combining the resulting hypotheses into an ensemble hypothesis. We explore online variants of the two most popular methods, bagging (Breiman, 1996a) and boosting (Schapire, 1990; Freund, 1995; Brieman, 1996b). To our knowledge, all previous empirical evaluations of ensemble methods have taken place in offline learning settings (Freund & Schapire, 1996; Quinlan, 1996; Bauer & Kohavi, 1999; Dietterich, in press)—our evaluation demonstrates similar online performance gains and also shows that ensemble methods are useful in meeting tight resource constraints.

3.1 Online Approaches to Ensemble Learning

Directly adapting an offline approach to produce the same ensembles online appears to require both storing the instances seen and reinvoking the base learning algorithm (particularly for boosting). Due to resource constraints, we consider methods that do not store previous instances.

We say that an online ensemble algorithm takes a sequential-generation approach if it generates the members one at a time, ceasing to update each member once the next one is started (otherwise, the approach is parallel-generation). We say the algorithm takes a single-update approach if it updates only one ensemble member for each training instance (otherwise multiple-update). Note that a sequential-generation approach is by definition single-update.

We wish to avoid the single-update/sequential approach. *Offline* methods of boosting and bagging allow a single training instance to contribute to many ensemble members—we seek this property in the online setting. This is particularly important in the presence of concept drift/change. Sequential-generation algorithms also suffer additionally in the presence of concept drift because most ensemble members are never going to be updated again—this patently requires adapting such algorithms with some kind of restart mechanism. Sequential methods also require a difficult-to-design method for determining when to start on another member, "freezing" the previous one.

To address these problems, we considered in this work only algorithms taking the parallel-generation multiple-update approach. This approach interacts well with our motivating application in that multiple updates can easily be carried out simultaneously on a highly parallel implementation platform such as VLSI. Freund (1995) described the *boost-by-majority (BBM)* algorithm for an online setting, taking a sequential generation approach. The online boosting algorithm we evaluate below can be viewed as a parallel-generation multiple-update variant of this algorithm that uses Arc-x4-style instance weighting.

Generic multiple-update algorithm. Here we present formally a generic online ensemble algorithm allowing multiple updates, and two instances of this algorithm. An ensemble is a 2-tuple consisting of a sequence of T hypotheses $(h_1,...,h_T)$ and a corresponding sequence of T scalar voting weights $(v_1,...,v_T)$. A hypothesis h_i is a mapping from the target concept domain to zero or one (i.e., $h_i(x) \in \{0,1\}$ for each domain element x). Given a domain element x the prediction returned by an ensemble $H = \langle (h_1,...,h_T), (v_1,...,v_T) \rangle$ is simply a weighted vote of the hypotheses, i.e., one if $(v_1[2h_1(x)-1] + \ldots + v_T[2h_T(x)-1]) > 0$ and zero otherwise. A training instance is a tuple $\langle x,c \rangle$ where x is a domain element and c is the classification in $\{0,1\}$ assigned to x by the target concept (assuming no noise). We assume Learn() is our base online learning algorithm: taking as input a hypothesis, a training instance, and a weight; the output of Learn is an updated hypothesis.

Table 1. Generic multiple-update online ensemble learner.

Input:
ensemble $H = \langle (h_{1,\ldots}, h_T), (v_1, \ldots, v_T) \rangle$
new training instance $I = \langle x, c \rangle$
base online learner Learn (*instance, weight, hypothesis*)
voting wt. update fun Update-Vote (*ensemble, instance, t*)
instance weight function Weight (*ensemble, instance, t*)

1. for each $t \in \{1, 2, \ldots, T\}$, ;; *possibly in parallel*
2. do $\hat{v}_t$ = Update-Vote (H, I, t) ;; *the new wt of* h_t
3. w_t = Weight (H, I, t) ;; *instance* w_t *for* h_t
4. $\hat{h}_t$ = Learn (I, w_t, h_t)

Output: new ensemble $\hat{H} = \langle (\hat{h}_{1\ldots}, \hat{h}_T), (\hat{v}_1, \ldots, \hat{v}_T) \rangle$

Table 1 shows the generic multiple-update algorithm we will use. The algorithm outputs an updated ensemble, taking as input an ensemble, a training instance, an online learning algorithm, and two functions Update-Vote() and Weight(). The function Update-Vote() is used to update the $(v_1,...,v_T)$ vector of ensemble member voting weights—e.g., if Update-Vote() always returns the number one, the ensemble prediction will simply be the majority vote. The function Weight() is used for each ensemble member h_t to assign a weight w_t to the new instance for updating h_t—to resemble boosting, this weight is related to the number of mistakes made by previous hypotheses on the current instance; for bagging the weight might be random.

For each hypothesis h_t the algorithm performs the following steps. First, in line 2 a new scalar voting weight v_t is computed by the function Update-Vote(). In line 3 a scalar instance weight w_t is computed by Weight(). In line 4, h_t is updated by Learn() using the training instance with the computed weight w_t. Each hypothesis and voting weight is updated in this manner (possibly in parallel). Our immediate research goal is to find (parallel) time and space efficient functions Update-Vote() and Weight() that produce ensembles that outperform single hypotheses.

3.2 Online Bagging

The bagging ensemble method has a natural parallel implementation since it does not require any interaction among the T hypotheses—our online variant simply ensures that each of the T hypotheses are the result of applying the base learner Learn() to a different sequence of

training instances. We use the generic algorithm from Figure 1 with the instance weight function given by

$$\underline{\text{Weight}}\,(H, I, t) = \underline{\text{coin}}\,(P_u), \quad 0 < P_u < 1 \qquad (1)$$

where $\underline{\text{coin}}(P)$ returns one with probability P and zero otherwise, and the probability P_u is user specified.

For our online bagging variant the function Update-Vote() simply counts the number of correct predictions made by a hypothesis on the training instances,

$$\underline{\text{Update-Vote}}\,(H, I, t) = v_t + 1 - \left|h_t(x) - c\right| \qquad (2)$$

So accurate hypotheses tend to get larger voting weights.[4]

3.3 Online Arc-x4

Online Arc-x4 uses the same instance weight function Weight() used by the offline algorithm Arc-x4 (Brieman, 1996b). The weight function is computed in two steps:

$$\underline{\text{Weight}}\,(H, I, t) = 1 + m_t^4, \qquad m_t = \sum_{i=1}^{t-1} \left|h_i(x) - c\right| \qquad (3)$$

The weight for the t'th hypothesis w_t is calculated by first counting the number m_t of previous hypotheses that incorrectly classify the new instance. The weight used is then one more than m_t to the fourth power, resulting in a boosting-style weighting that emphasizes instances that many previous hypotheses get wrong. This function was arrived at (partly) empirically in the design of offline Arc-x4 (Brieman, 1996b). Nevertheless, it has performed well in practice and its simplicity (e.g., compared to AdaBoost) made it an attractive choice for this application. Online Arc-x4 uses the same accuracy based voting weight function Update-Vote() as online bagging (Equation 2 above). Note that the offline version of Arc-x4 uses a majority rather than a weighted vote. We found, however, an empirical advantage to weighted voting for small ensembles.

Alternative weight functions. We note that other offline weighting functions can fairly easily be adapted to the online setting, including those used in AdaBoost and Boost-by-Majority (BBM). One issue of concern is that the bell-shaped weighting function used in BBM that gives small weights for both very easy *and* very hard instances may be inappropriate for multiple-update online learners, especially in the presence of target concept drift or change.

Complexity and Implementation of Online Arc-x4. An efficient parallel implementation is particularly significant to our target domain of conditional-branch outcome prediction. In the full paper (Fern & Givan, 2000) we show that the time complexity of a parallel implementation of online Arc-x4 is $O(\log^2 T)$ plus the prediction time used by an individual base learner; and that the space complexity of the same implementation is $O(T \cdot \log T)$ plus the space used by the T individual base learners.

4. We have also implemented online bagging using straight majority vote and the empirical results are not substantially different.

4. Online Decision-Tree Induction

Here we briefly describe an online decision tree learning algorithm that will be used as the base learner in our ensemble experiments. Most decision-tree methods are designed for offline settings, as in the well-known ID3 algorithm (Quinlan, 1986); but there has also been research on online algorithms, with two key methods being ID5R (Utgoff, 1989) and ID4 (Schlimmer & Fisher, 1986).

The ID5R method incrementally build trees by storing previous training instances and restructuring the current tree if necessary when a new instance arrives. The tree restructuring operations required are expensive and somewhat complex for use in resource-bounded online settings. In addition, although the recursive restructuring operation is straightforward to implement in software, our motivating domain requires a hardware implementation that appears quite difficult for these methods. For these reasons, and also to avoid the space costs of storing instances, we use a variant of the simpler online decision tree algorithm ID4. Below we describe our extensions to ID4. The full paper also contains a complete description of ID4 and empirical results showing that our extensions significantly improve the accuracy of both single trees and tree ensembles.

ID4 incrementally updates a decision-tree by maintaining an estimate of the split criterion of each feature at each node, and using these estimates to dynamically select the split feature as well as to prune the tree via pre-pruning.

Advanced warm-up extension. In the ID4 algorithm when a leaf node is "split" to become an internal node, its new children must begin learning from scratch. We extend ID4 to allow for *advanced warm-up*—leaf nodes (for prediction) have descendents that are learning from examples (even though they are not used for predictions).

Post-pruning by subtree monitoring extension. The decision to make a node a leaf in ID4 is determined by a χ^2-test on potential split features, using *pre-pruning*. When using advanced warm-up we can monitor the performance of subtrees of leaves, and use the result to *post-prune* by comparing the monitored accuracy to the leaf accuracy.

Feature-switch suppression by subtree monitoring. In the original ID4, when a new split feature is selected at a node, the subtrees of the node are discarded (regardless of how well they are performing). To avoid frequent discarding, our ID4 variant refuses to change the split feature of a node (and hence prune the subtrees) unless the accuracy of making predictions with the candidate new split feature (discarding the subtrees) is better than that of making predictions with the current split feature and subtrees.

5. Empirical Results for Arc-x4

From here on we refer to our online variants of bagging and Arc-x4 as simply "bagging" and "Arc-x4", respectively. In this section we present empirical results using Arc-x4 to generate decision tree ensembles for several problems, starting with machine learning benchmarks then

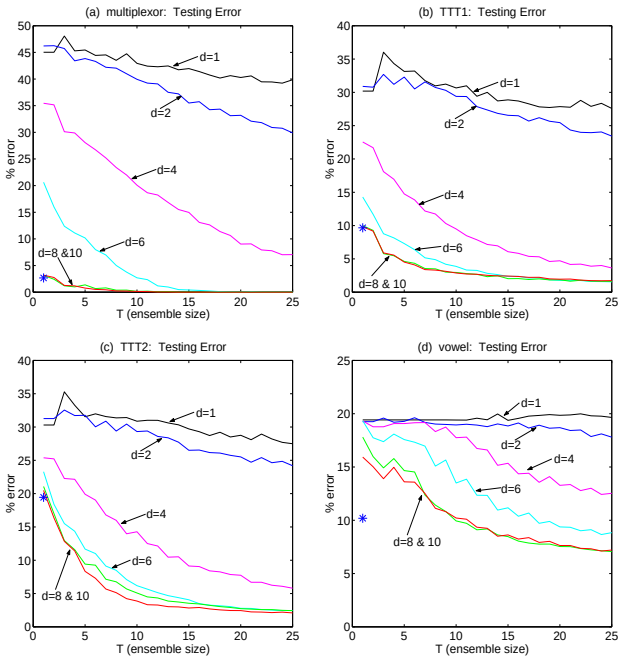

Figure 1. Final Arc-x4 test error vs. ensemble size for the machine learning data sets. (*Note: the x-axis is not ~time)* Results after ensembles encounter 50,000 training instances (100,000 for *vowel*). Each curve varies ensemble size using trees of a fixed depth limit *d*. Stars show unbounded-depth T=1 performance.

moving to our domain of branch prediction. Arc-x4 is shown to generally significantly improve prediction accuracy over single trees. In addition, we show that boosting produces ensembles of small trees that often outperform large single trees with the same number of nodes.

5.1 Results for Machine Learning Data Sets

Full details of the data sets and experimental protocol used are available in the full paper (Fern & Givan, 2000); we provide a brief summary here. We considered four familiar machine learning benchmarks[5], dividing each into equal test and training sets twenty different ways (randomly) and averaging the results—each problem is treated as an online problem by sampling training instances with replacement from the training set. Error is measured periodically by "freezing" the learned concept and checking its accuracy on the testing or training set. Most of our plots use only the average testing error of the final ensemble.

The four benchmarks used are: an eleven-bit multiplexor problem (as investigated in (Quinlan, 1988)); "easy" and "hard" encodings of the Tic-Tac-Toe endgame database at UCI (see (Merz & Murphy, 1996))—one encoding uses two bits per game cell, the other uses ASCII eight-bit encodings for 'x'/'o'/'b' ('b' for blank); and a two-class version of the letter recognition problem in the UCI repository where the two-class task is to recognize vowels.

5. These four benchmarks were selected to offer the relatively large quantities of labelled data needed for online learning as well as a natural encoding as binary feature space two-class learning problems. We have not run these algorithms on any other machine learning data sets.

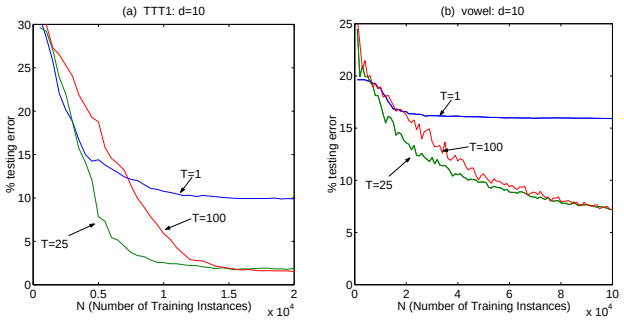

Figure 2. Arc-x4 warm-up performance (varying ~time). Training error vs. number of instances (~time) for two benchmarks. Each curve reflects a single ensemble with depth bound ten.

We vary the ensemble size (T) and the tree-depth bound (d). Figure 1 shows the test set errors versus T for the four benchmarks. Each figure has one curve for each value of d. Stars on the graphs indicate the unbounded-depth base-learner error.

Advantages of larger ensemble size. In all four problems, increased ensemble size generally reduces the error effectively. Increasing ensemble size leads to significant improvements for the benchmarks, even though the stars show an apparent performance limit for single trees.

We also note that weaker learners (low d values) are generally less able to exploit ensemble size, as reflected in the slopes of the plots for varying depth bounds—we find a steeper error reduction with ensemble size for deeper trees. Apparently ensemble learning is exploiting different leverage on the problem than increased depth—i.e., increasing ensemble size arbitrarily can never get all the benefits available by increasing depth, and vice versa.

Training error comparisons. Space precludes showing the very similar training error graphs. Similar trends prevail in those graphs, indicating that Arc-x4 is generalizing the four concepts well. We also note that larger ensembles can improve testing error (over smaller ones) even when the training error has gone to zero (e.g., the training error for TTT1 depth ten goes to zero at T=5).

Warm-up behavior. Figures 2a and 2b show percent error versus number of instances N encountered for two problems. Comparing the curves for ensemble sizes one and 100, we see that (as expected) the large ensemble achieves a lower percent error after many training instances, but for a smaller number of training instances the single tree is superior. This observation indicates that the ideal ensemble size may depend on the number of training instances we expect to encounter (or may even be ideally selected dynamically as more instances are encountered). However, for these two benchmarks it appears that ensemble size T=25 achieves both the early performance of T=1 and the late performance of T=100 ensembles. Comparing the 25 member ensemble and the 100 member ensemble reveals that large ensembles suffer from poor early performance.

Table 2. Branches used in our experiments.

Branch Name	# of Instances	% Taken	State-of-the-art % Error	Benchmark Program
go-A	413,908	35%	5.3%	**go:** an AI program that plays the game of go
go-B	370,719	47%	19.8%	
go-C	407,380	33%	13.8%	
go-D	451,042	57%	14.0%	
li-A	2,653,159	20%	5.3%	**li:** a LISP interpreter
li-B	1,238,803	71%	1.6%	
com-A	253,031	56%	4.84%	**com:** UNIX compress
com-B	17,104	75%	2.59%	

5.2 Branch Prediction Domain and Results

Experimental Procedure. We used the trace-driven microprocessor simulator described in (Burger & Austin, 1997), and focused on eight branches from three different benchmark programs in the SPECint95 benchmark suite (Reilly, 1995)—we selected "hard" branches where single trees are outperformed by current table-based branch predictors. Table 2 provides information about the benchmark programs used and the branches selected from these benchmarks. The "State-of-the-art % Error" shown is from the highly-specialized "hybrid" table-based predictor from computer architecture (McFarling, 1993)—it is not our current goal to improve on these results with our general-purpose method, particularly on these "hard" branches.

The online nature of this application makes separate testing and training data sets unnatural—instead, we present each branch to the learner during simulation as a test instance, and then provide the correct answer for training when the branch is resolved. The final percent error plotted is the percent of test instances predicted incorrectly, varying both ensemble size (T) and tree depth (d).

Basic Arc-x4 Performance. Figures 3a-3f give the percent error versus ensemble size for six branches, with curves plotted for six different depth bounds (*d*), as well as stars showing the percent error achieved by a single (online) tree of unbounded depth. These graphs exhibit the same trends as the results above for the machine learning benchmarks, with error decreasing with increasing *T*, even well beyond the unbounded-depth single online tree error.

Small ensemble effects. The graphs show erratic behavior for small ensemble size—we conjecture that at small sizes, an "unlucky" sequence (or weighting) of instances affecting a few consecutive ensemble members can easily dominate the vote. We note that this erratic behavior is even worse if we use unweighted voting (not shown), supporting this conjecture—as *T* increases unweighted and weighted voting perform nearly identically.

Comparing to ensemble size one. The graphs in Figure 3 all exhibit a similar trend with respect to increasing *d*. For small *d*, large ensembles are needed to see much benefit, but the eventual benefits are larger (than when *d* is large). However, every curve shows peak performance at an ensemble size larger than one. The stars showing the best single tree performance indicate that bounded-depth ensembles can outperform unbounded-depth single trees.

Space usage. Figures 4a–4f show error versus log space usage, giving a basis for selecting *d* and *T* to optimize ac-

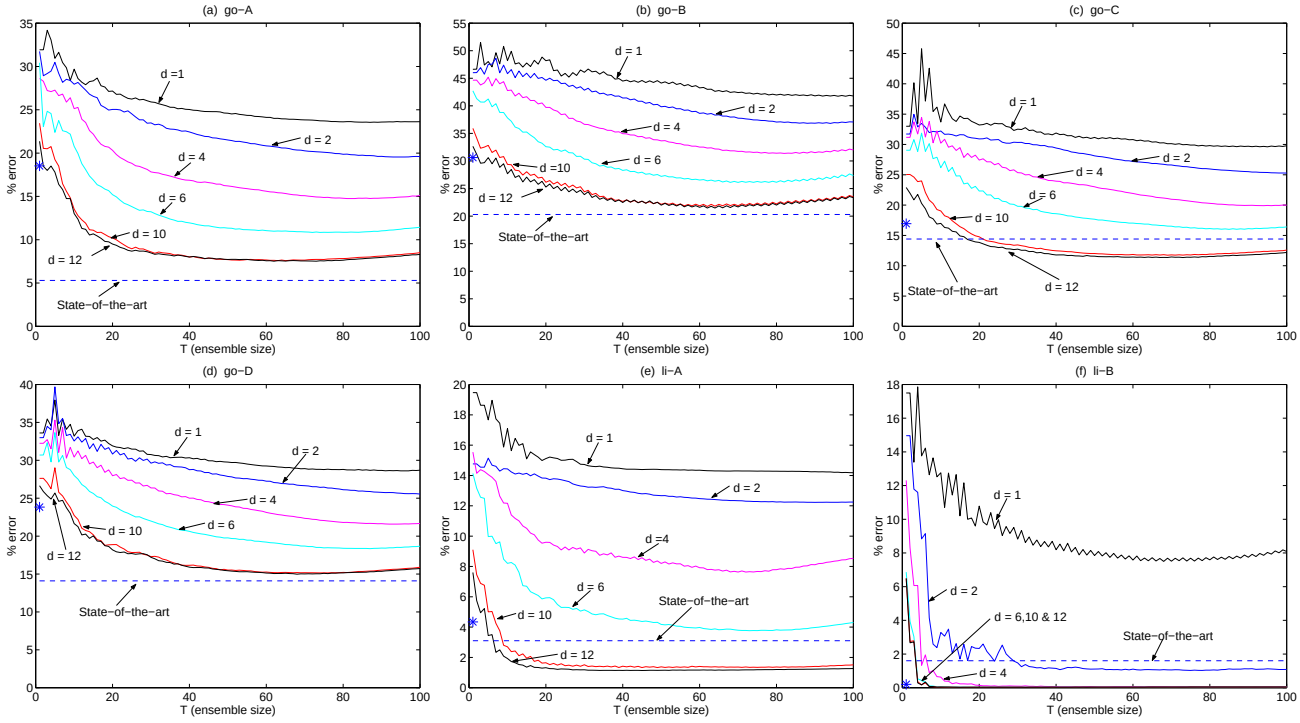

Figure 3. Arc-x4 Percent Error vs. Ensemble Size for six hard branches. (*x-axis is not ~time)* Each curve varies ensemble size fixing a depth limit. The stars show the error achieved by a single tree with unbounded depth. The dotted line show state-of-the-art error.

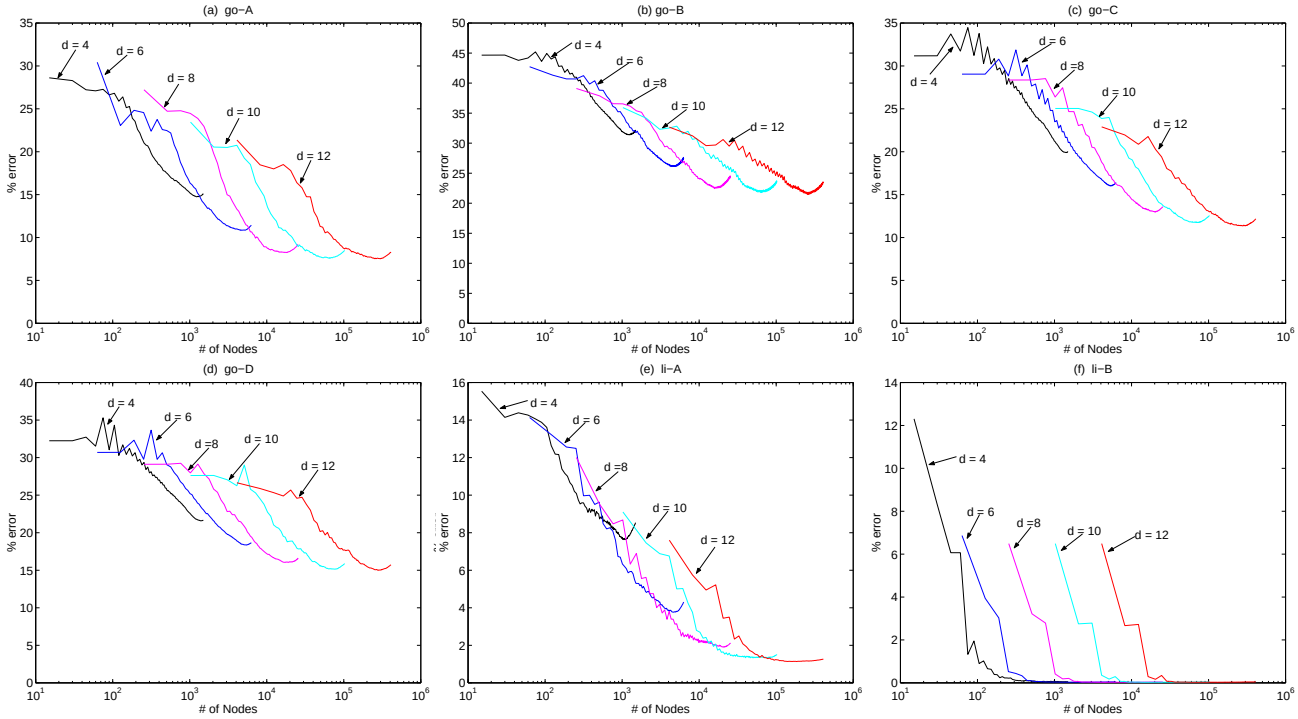

Figure 4. Arc-x4 Percent Error vs. Ensemble Node Count for six hard branches. Again, each curve fixes the tree-depth limit and varies ensemble size—but here we plot total number of tree nodes (space usage).

curacy when facing space constraints. A size T ensemble of depth d trees has $T \cdot (2^{d+1} - 1)$ non-leaf nodes (hardware implementations cannot virtually allocate nodes).

Note that as d increases the ensemble curves shift to the right—ensemble size is generally a better use of space than tree depth. For a fixed node usage the best error is usually achieved by an ensemble with T greater than one. This observation is strongest for the go branches and weakest for li-A. (Consider a vertical cross-section and determine whether the lowest error corresponds to a small d and thus a large T—e.g., at 1000 nodes, d equal to three shows the best performance on five of the six graphs).

Now suppose that instead of a size constraint we are given a maximum percent error constraint. Figure 4 shows that using ensembles often allows us to achieve a particular percent error using much less space with a large ensemble of small trees rather than smaller ensembles or single trees. These observations suggest that online boosting may be particularly useful in domains with space constraints.

Poor performance on com branches. Two of the eight branches showed poor performance for Arc-x4 ensembles. Figures 5a and 5b show the percent error versus T plots for these branches, as well as the state-of-the-art error and unbounded single-tree performance. The com-A and com-B branches show little benefit for ensembles as well as generally poor performance for single trees, suggesting that these concepts are not well captured by the ID4-style base learners here. In addition, we note that com-B has only 17,104 instances, suggesting that the ensembles have also not had enough time to warm up.

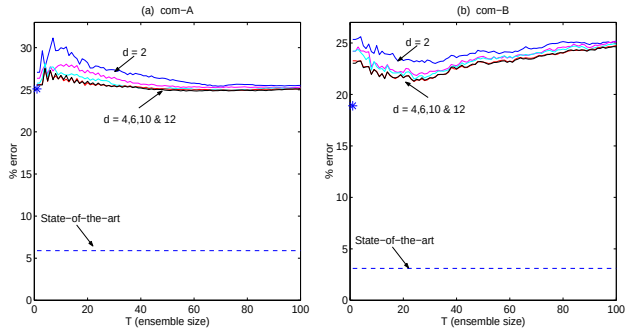

Figure 5. Arc-x4 Percent Error vs. Ensemble Size for two hard branches. Again, each curve corresponds to ensembles using the same depth limit. We do not show space curves (like Figure 4) for these branches due to page limitations.

6. Empirical Results for Bagging

Figure 6 compares the performance of bagging and Arc-x4 on three online problems—these three plots typify performance on our other problems (particularly Figure 6a). In each figure we plot percent error versus ensemble size T for four different ensemble methods (Arc-x4 and three P_u choices for bagging) using trees with a depth bound of twelve. The bagging plots for branch prediction are averaged over ten runs due to the random choices made in the algorithm. These results indicate poor performance for P_u equal to 0.1, most likely because trees in the ensembles are updated too infrequently. We show in Figure 6b the most significant improvement over Arc-x4 achieved by bagging in any of our bagging experiments (many not shown). Arc-x4 outperforms bagging for small and large ensembles.

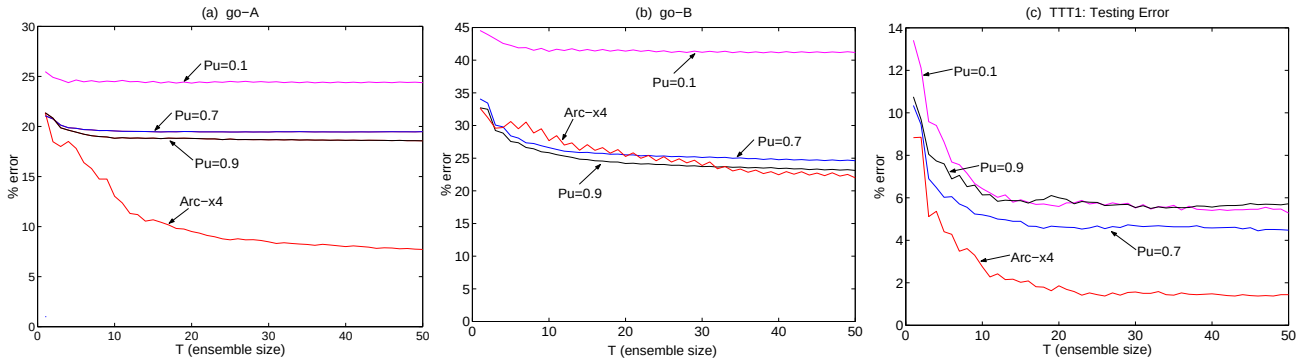

Figure 6. Bagging versus Arc-x4. Percent error versus the ensemble size T for three problems (as indicated above each graph). Three curves per graph show the performance of bagging for P_u = 0.1, 0.7, 0.9 and one curve shows the performance of Arc-x4. All trees used a depth bound of twelve. Our other domains give graphs strongly resembling Graph (a).

A possible explanation for the poor performance of bagging may be the use of zero/one instance weighting: offline bagging can weight an instance higher than one. This is explored in the full paper (Fern & Givan, 2000).

7. Conclusions and Future Work

In this work we empirically studied two online ensemble learning algorithms—bagging and boosting-style ensemble approaches—that do not store training instances and have efficient parallel hardware implementations. We gave empirical results for the algorithms, using ID4-like decision tree learners using conditional branch prediction as well as online variants of familiar machine learning data sets. These results indicate that online Arc-x4 significantly outperforms our online bagging method. The online Arc-x4 ensembles are shown to achieve significantly higher accuracies than single trees—with ensembles of small trees often outperforming single large trees or smaller ensembles of larger trees using the same total nodes, suggesting the use of ensembles when facing space constraints.

Future research is needed on issues raised by this work, including: parallel vs. sequential ensemble generation; bagging using weights other than zero/one; characterizing when boosting performs poorly (and/or bagging well); dynamically varying ensemble size during warm-up/concept drift; varying tree depth within an ensemble; and developing other resource-constrained online application domains.

Acknowledgements

This material is based upon work supported under a National Science Foundation Graduate Fellowship and Award No. 9977981-IIS to the second author.

References

Bauer, E., & Kohavi, R. (1999). An empirical comparison of voting classification algorithms: bagging, boosting and variants. *Machine Learning, 36*, 105-142.

Breiman, L. (1996a). Bagging predictors. *Machine Learning, 24*, 123-140.

Breiman, L. (1996b). *Arcing classifiers* (Technical Report). Dept. of Statistics, Univ. of California, Berkeley, CA.

Burger, D., & Austin, T. (1997). *The SimpleScalar tool set, version 2.0* (Technical Report 1342). Computer Science Department, University of Wisconsin-Madison.

Chang, P.-Y., Hao, E., & Patt, Y. (1995). Alternative implementations of hybrid branch predictors. *The 28th ACM/IEEE International Symposium on Microarchitecture.*

Dietterich, T. G. (in press). An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting, and randomization. *Machine Learning*.

Fern, A., & Givan, R. (2000). *Online ensemble learning: an empirical study.* (unpublished manuscript). Dept. of Elect. and Comp. Eng., Purdue University, W. Lafayette, IN. Available now at http://www.ece.purdue.edu/~givan/

Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Information and Computation, 121*, 256-285.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Proceedings of the Thirteenth International Conference on Machine Learning* (pp. 148-156). San Francisco: Morgan Kaufmann.

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences, 55*, 119-139.

McFarling, S. (1993). *Combining branch predictors* (Technical Note TN-36). Western Research Laboratory, DEC.

Merz, C. J., & Murphy, P. M. (1996). UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81-106.

Quinlan, J. R. (1988). An empirical comparison of genetic and decision-tree classifiers. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 135-141). San Francisco: Morgan Kaufmann.

Quinlan, J. R. (1996). Bagging, boosting and C4.5. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 725-730). Cambridge, MA: MIT Press.

Reilly, J. (1995). SPEC describes SPEC95 products and benchmarks. *Standard Performance Evaluation Corporation* August 1995 newsletter, http://www.spec.org.

Schlimmer, J. C., & Fisher, D. (1986). A case study of incremental concept induction. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 496-501). San Francisco: Morgan Kaufmann.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning, 5*, 197-227.

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