Machine Learning: Proceedings of the Thirteenth International Conference, 1996.

Experiments with a New Boosting Algorithm

Yoav Freund **Robert E. Schapire**
AT&T Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974-0636
{yoav, schapire}@research.att.com

Abstract. In an earlier paper, we introduced a new "boosting" algorithm called **AdaBoost** which, theoretically, can be used to significantly reduce the error of any learning algorithm that consistently generates classifiers whose performance is a little better than random guessing. We also introduced the related notion of a "pseudo-loss" which is a method for forcing a learning algorithm of multi-label concepts to concentrate on the labels that are hardest to discriminate. In this paper, we describe experiments we carried out to assess how well **AdaBoost** with and without pseudo-loss, performs on real learning problems.

We performed two sets of experiments. The first set compared boosting to Breiman's "bagging" method when used to aggregate various classifiers (including decision trees and single attribute-value tests). We compared the performance of the two methods on a collection of machine-learning benchmarks. In the second set of experiments, we studied in more detail the performance of boosting using a nearest-neighbor classifier on an OCR problem.

1 INTRODUCTION

"Boosting" is a general method for improving the performance of any learning algorithm. In theory, boosting can be used to significantly reduce the error of any "weak" learning algorithm that consistently generates classifiers which need only be a little bit better than random guessing. Despite the potential benefits of boosting promised by the theoretical results, the true practical value of boosting can only be assessed by testing the method on real machine learning problems. In this paper, we present such an experimental assessment of a new boosting algorithm called **AdaBoost**.

Boosting works by repeatedly running a given weak[1] learning algorithm on various distributions over the training data, and then combining the classifiers produced by the weak learner into a single composite classifier. The first provably effective boosting algorithms were presented by Schapire [20] and Freund [9]. More recently, we described and analyzed **AdaBoost**, and we argued that this new boosting algorithm has certain properties which make it more practical and easier to implement than its predecessors [10]. This algorithm, which we used in all our experiments, is described in detail in Section 2.

This paper describes two distinct sets of experiments. In the first set of experiments, described in Section 3, we compared boosting to "bagging," a method described by Breiman [1] which works in the same general fashion (i.e., by repeatedly rerunning a given weak learning algorithm, and combining the computed classifiers), but which constructs each distribution in a simpler manner. (Details given below.) We compared boosting with bagging because both methods work by combining many classifiers. This comparison allows us to separate out the effect of modifying the distribution on each round (which is done differently by each algorithm) from the effect of voting multiple classifiers (which is done the same by each).

In our experiments, we compared boosting to bagging using a number of different weak learning algorithms of varying levels of sophistication. These include: (1) an algorithm that searches for very simple prediction rules which test on a single attribute (similar to Holte's very simple classification rules [14]); (2) an algorithm that searches for a single good decision rule that tests on a conjunction of attribute tests (similar in flavor to the rule-formation part of Cohen's RIPPER algorithm [3] and Fürnkranz and Widmer's IREP algorithm [11]); and (3) Quinlan's **C4.5** decision-tree algorithm [18]. We tested these algorithms on a collection of 27 benchmark learning problems taken from the UCI repository.

The main conclusion of our experiments is that boosting performs significantly and uniformly better than bagging when the weak learning algorithm generates fairly simple classifiers (algorithms (1) and (2) above). When combined with **C4.5**, boosting still seems to outperform bagging slightly, but the results are less compelling.

We also found that boosting can be used with very simple rules (algorithm (1)) to construct classifiers that are quite good relative, say, to **C4.5**. Kearns and Mansour [16] argue that **C4.5** can itself be viewed as a kind of boosting algorithm, so a comparison of **AdaBoost** and **C4.5** can be seen as a comparison of two competing boosting algorithms. See Dietterich, Kearns and Mansour's paper [4] for more detail on this point.

In the second set of experiments, we test the performance of boosting on a nearest neighbor classifier for handwritten digit recognition. In this case the weak learning algorithm is very simple, and this lets us gain some insight into the interaction between the boosting algorithm and the

Home page: "http://www.research.att.com/orgs/ssr/people/uid". Expected to change to "http://www.research.att.com/~uid" sometime in the near future (for uid $\in$ {yoav, schapire}).

[1]We use the term "weak" learning algorithm, even though, in practice, boosting might be combined with a quite strong learning algorithm such as **C4.5**.

nearest neighbor classifier. We show that the boosting algorithm is an effective way for finding a small subset of prototypes that performs almost as well as the complete set. We also show that it compares favorably to the standard method of Condensed Nearest Neighbor [13] in terms of its test error.

There seem to be two separate reasons for the improvement in performance that is achieved by boosting. The first and better understood effect of boosting is that it generates a hypothesis whose error on the training set is small by combining many hypotheses whose error may be large (but still better than random guessing). It seems that boosting may be helpful on learning problems having either of the following two properties. The first property, which holds for many real-world problems, is that the observed examples tend to have varying degrees of hardness. For such problems, the boosting algorithm tends to generate distributions that concentrate on the harder examples, thus challenging the weak learning algorithm to perform well on these harder parts of the sample space. The second property is that the learning algorithm be sensitive to changes in the training examples so that significantly different hypotheses are generated for different training sets. In this sense, boosting is similar to Breiman's bagging [1] which performs best when the weak learner exhibits such "unstable" behavior. However, unlike bagging, boosting tries actively to force the weak learning algorithm to change its hypotheses by changing the distribution over the training examples as a function of the errors made by previously generated hypotheses.

The second effect of boosting has to do with variance reduction. Intuitively, taking a weighted majority over many hypotheses, all of which were trained on different samples taken out of the same training set, has the effect of reducing the random variability of the combined hypothesis. Thus, like bagging, boosting may have the effect of producing a combined hypothesis whose variance is significantly lower than those produced by the weak learner. However, unlike bagging, boosting may also reduce the bias of the learning algorithm, as discussed above. (See Kong and Dietterich [17] for further discussion of the bias and variance reducing effects of voting multiple hypotheses, as well as Breiman's [2] very recent work comparing boosting and bagging in terms of their effects on bias and variance.) In our first set of experiments, we compare boosting and bagging, and try to use that comparison to separate between the bias and variance reducing effects of boosting.

Previous work. Drucker, Schapire and Simard [8, 7] performed the first experiments using a boosting algorithm. They used Schapire's [20] original boosting algorithm combined with a neural net for an OCR problem. Follow-up comparisons to other ensemble methods were done by Drucker et al. [6]. More recently, Drucker and Cortes [5] used **AdaBoost** with a decision-tree algorithm for an OCR task. Jackson and Craven [15] used **AdaBoost** to learn classifiers represented by sparse perceptrons, and tested the algorithm on a set of benchmarks. Finally, Quinlan [19] recently conducted an independent comparison of boosting and bagging combined with **C4.5** on a collection of UCI benchmarks.

Algorithm AdaBoost.M1
Input: sequence of m examples $\langle(x_1, y_1), \ldots, (x_m, y_m)\rangle$
with labels $y_i \in Y = \{1, \ldots, k\}$
weak learning algorithm **WeakLearn**
integer T specifying number of iterations
Initialize $D_1(i) = 1/m$ for all i.
Do for $t = 1, 2, \ldots, T$:

1. Call **WeakLearn**, providing it with the distribution D_t.
2. Get back a hypothesis $h_t : X \to Y$.
3. Calculate the error of h_t: $\epsilon_t = \sum_{i:h_t(x_i)\neq y_i} D_t(i)$.
   If $\epsilon_t > 1/2$, then set $T = t - 1$ and abort loop.
4. Set $\beta_t = \epsilon_t/(1 - \epsilon_t)$.
5. Update distribution D_t:
   $$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \beta_t & \text{if } h_t(x_i) = y_i \\ 1 & \text{otherwise} \end{cases}$$
   where Z_t is a normalization constant (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:
$$h_{fin}(x) = \arg\max_{y\in Y} \sum_{t:h_t(x)=y} \log \frac{1}{\beta_t}.$$

Figure 1: The algorithm **AdaBoost.M1**.

2 THE BOOSTING ALGORITHM

In this section, we describe our boosting algorithm, called **AdaBoost**. See our earlier paper [10] for more details about the algorithm and its theoretical properties.

We describe two versions of the algorithm which we denote **AdaBoost.M1** and **AdaBoost.M2**. The two versions are equivalent for binary classification problems and differ only in their handling of problems with more than two classes.

2.1 ADABOOST.M1

We begin with the simpler version, **AdaBoost.M1**. The boosting algorithm takes as input a training set of m examples $S = \langle(x_1, y_1), \ldots, (x_m, y_m)\rangle$ where x_i is an instance drawn from some space X and represented in some manner (typically, a vector of attribute values), and $y_i \in Y$ is the class label associated with x_i. In this paper, we always assume that the set of possible labels Y is of finite cardinality k.

In addition, the boosting algorithm has access to another unspecified learning algorithm, called the weak learning algorithm, which is denoted generically as **WeakLearn**. The boosting algorithm calls **WeakLearn** repeatedly in a series of rounds. On round t, the booster provides **WeakLearn** with a distribution D_t over the training set S. In response, **WeakLearn** computes a classifier or *hypothesis* $h_t : X \to Y$ which should correctly classify a fraction of the training set that has large probability with respect to D_t. That is, the weak learner's goal is to find a hypothesis h_t which minimizes the (training) error $\epsilon_t = \Pr_{i\sim D_t}\left[h_t(x_i) \neq y_i\right]$. Note that this error is measured with respect to the distribution D_t that was provided to the weak learner. This process continues for T rounds, and, at last, the booster combines the weak hypotheses $h_1, \ldots, h_T$ into a single final hypothesis h_{fin}.

Algorithm AdaBoost.M2
Input: sequence of m examples $\langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$
with labels $y_i \in Y = \{1, \ldots, k\}$
weak learning algorithm **WeakLearn**
integer T specifying number of iterations

Let $B = \{(i, y) : i \in \{1, \ldots, m\}, y \neq y_i\}$
Initialize $D_1(i, y) = 1/|B|$ for $(i, y) \in B$.
Do for $t = 1, 2, \ldots, T$

1. Call **WeakLearn**, providing it with mislabel distribution D_t.
2. Get back a hypothesis $h_t : X \times Y \to [0, 1]$.
3. Calculate the pseudo-loss of h_t:
$\epsilon_t = \frac{1}{2} \sum_{(i,y) \in B} D_t(i, y) \big(1 - h_t(x_i, y_i) + h_t(x_i, y)\big).$
4. Set $\beta_t = \epsilon_t / (1 - \epsilon_t)$.
5. Update D_t:
$D_{t+1}(i, y) = \frac{D_t(i, y)}{Z_t} \cdot \beta_t^{(1/2)(1 + h_t(x_i, y_i) - h_t(x_i, y))}$
where Z_t is a normalization constant (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:
$$h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} \left(\log \frac{1}{\beta_t} \right) h_t(x, y).$$

Figure 2: The algorithm **AdaBoost.M2**.

Still unspecified are: (1) the manner in which D_t is computed on each round, and (2) how h_{fin} is computed. Different boosting schemes answer these two questions in different ways. **AdaBoost.M1** uses the simple rule shown in Figure 1. The initial distribution D_1 is uniform over S so $D_1(i) = 1/m$ for all i. To compute distribution D_{t+1} from D_t and the last weak hypothesis h_t, we multiply the weight of example i by some number $\beta_t \in [0, 1)$ if h_t classifies x_i correctly, and otherwise the weight is left unchanged. The weights are then renormalized by dividing by the normalization constant Z_t. Effectively, "easy" examples that are correctly classified by many of the previous weak hypotheses get lower weight, and "hard" examples which tend often to be misclassified get higher weight. Thus, **AdaBoost** focuses the most weight on the examples which seem to be hardest for **WeakLearn**.

The number β_t is computed as shown in the figure as a function of ϵ_t. The final hypothesis h_{fin} is a weighted vote (i.e., a weighted linear threshold) of the weak hypotheses. That is, for a given instance x, h_{fin} outputs the label y that maximizes the sum of the weights of the weak hypotheses predicting that label. The weight of hypothesis h_t is defined to be $\log(1/\beta_t)$ so that greater weight is given to hypotheses with lower error.

The important theoretical property about **AdaBoost.M1** is stated in the following theorem. This theorem shows that if the weak hypotheses consistently have error only slightly better than $1/2$, then the training error of the final hypothesis h_{fin} drops to zero exponentially fast. For binary classification problems, this means that the weak hypotheses need be only slightly better than random.

Theorem 1 ([10]) *Suppose the weak learning algorithm* **WeakLearn**, *when called by* **AdaBoost.M1**, *generates hypotheses with errors $\epsilon_1, \ldots, \epsilon_T$, where ϵ_t is as defined in Figure 1. Assume each $\epsilon_t \leq 1/2$, and let $\gamma_t = 1/2 - \epsilon_t$. Then the following upper bound holds on the error of the final hypothesis h_{fin}:*

$$\frac{|\{i : h_{fin}(x_i) \neq y_i\}|}{m} \leq \prod_{t=1}^{T} \sqrt{1 - 4\gamma_t^2} \leq \exp\left(-2 \sum_{t=1}^{T} \gamma_t^2 \right).$$

Theorem 1 implies that the *training error* of the final hypothesis generated by **AdaBoost.M1** is small. This does not necessarily imply that the *test* error is small. However, if the weak hypotheses are "simple" and T "not too large," then the difference between the training and test errors can also be theoretically bounded (see our earlier paper [10] for more on this subject).

The experiments in this paper indicate that the theoretical bound on the training error is often weak, but generally correct qualitatively. However, the test error tends to be much better than the theory would suggest, indicating a clear defect in our theoretical understanding.

The main disadvantage of **AdaBoost.M1** is that it is unable to handle weak hypotheses with error greater than $1/2$. The expected error of a hypothesis which randomly guesses the label is $1 - 1/k$, where k is the number of possible labels. Thus, for $k = 2$, the weak hypotheses need to be just slightly better than random guessing, but when $k > 2$, the requirement that the error be less than $1/2$ is quite strong and may often be hard to meet.

2.2 ADABOOST.M2

The second version of **AdaBoost** attempts to overcome this difficulty by extending the communication between the boosting algorithm and the weak learner. First, we allow the weak learner to generate more expressive hypotheses, which, rather than identifying a single label in Y, instead choose a set of "plausible" labels. This may often be easier than choosing just one label. For instance, in an OCR setting, it may be hard to tell if a particular image is "7" or a "9", but easy to eliminate all of the other possibilities. In this case, rather than choosing between 7 and 9, the hypothesis may output the set $\{7, 9\}$ indicating that both labels are plausible.

We also allow the weak learner to indicate a "degree of plausibility." Thus, each weak hypothesis outputs a vector $[0, 1]^k$, where the components with values close to 1 or 0 correspond to those labels considered to be plausible or implausible, respectively. Note that this vector of values is *not* a probability vector, i.e., the components need not sum to one.[2]

While we give the weak learning algorithm more expressive power, we also place a more complex requirement on the performance of the weak hypotheses. Rather than using the usual prediction error, we ask that the weak hypotheses do well with respect to a more sophisticated error measure that we call the *pseudo-loss*. Unlike ordinary error which is computed with respect to a distribution over examples, pseudo-loss is computed with respect to a distribution

[2]We deliberately use the term "plausible" rather than "probable" to emphasize the fact that these numbers should not be interpreted as the probability of a given label.

over the set of all pairs of examples and incorrect labels. By manipulating this distribution, the boosting algorithm can focus the weak learner not only on hard-to-classify examples, but more specifically, on the incorrect labels that are hardest to discriminate. We will see that the boosting algorithm **AdaBoost.M2**, which is based on these ideas, achieves boosting if each weak hypothesis has pseudo-loss slightly better than random guessing.

More formally, a *mislabel* is a pair (i, y) where i is the index of a training example and y is an *incorrect* label associated with example i. Let B be the set of all mislabels: $B = \{(i, y) : i \in \{1, \ldots, m\}, y \neq y_i\}$. A *mislabel distribution* is a distribution defined over the set B of all mislabels.

On each round t of boosting, **AdaBoost.M2** (Figure 2) supplies the weak learner with a mislabel distribution D_t. In response, the weak learner computes a hypothesis h_t of the form $h_t : X \times Y \to [0, 1]$. There is no restriction on $\sum_y h_t(x, y)$. In particular, the prediction vector does not have to define a probability distribution.

Intuitively, we interpret each mislabel (i, y) as representing a binary question of the form: "Do you predict that the label associated with example x_i is y_i (the correct label) or y (one of the incorrect labels)?" With this interpretation, the weight $D_t(i, y)$ assigned to this mislabel represents the importance of distinguishing incorrect label y on example x_i.

A weak hypothesis h_t is then interpreted in the following manner. If $h_t(x_i, y_i) = 1$ and $h_t(x_i, y) = 0$, then h_t has (correctly) predicted that x_i's label is y_i, not y (since h_t deems y_i to be "plausible" and y "implausible"). Similarly, if $h_t(x_i, y_i) = 0$ and $h_t(x_i, y) = 1$, then h_t has (incorrectly) made the opposite prediction. If $h_t(x_i, y_i) = h_t(x_i, y)$, then h_t's prediction is taken to be a random guess. (Values for h_t in $(0, 1)$ are interpreted probabilistically.)

This interpretation leads us to define the *pseudo-loss* of hypothesis h_t with respect to mislabel distribution D_t by the formula

$$\epsilon_t = \tfrac{1}{2} \sum_{(i,y) \in B} D_t(i, y) \big(1 - h_t(x_i, y_i) + h_t(x_i, y)\big).$$

Space limitations prevent us from giving a complete derivation of this formula which is explained in detail in our earlier paper [10]. It can be verified though that the pseudo-loss is minimized when correct labels y_i are assigned the value 1 and incorrect labels $y \neq y_i$ assigned the value 0. Further, note that pseudo-loss $1/2$ is trivially achieved by any constant-valued hypothesis h_t.

The weak learner's goal is to find a weak hypothesis h_t with small pseudo-loss. Thus, standard "off-the-shelf" learning algorithms may need some modification to be used in this manner, although this modification is often straightforward. After receiving h_t, the mislabel distribution is updated using a rule similar to the one used in **AdaBoost.M1**. The final hypothesis h_{fin} outputs, for a given instance x, the label y that maximizes a weighted average of the weak hypothesis values $h_t(x, y)$.

The following theorem gives a bound on the training error of the final hypothesis. Note that this theorem requires only that the weak hypotheses have pseudo-loss less than $1/2$, i.e., only slightly better than a trivial (constant-valued) hypothesis, regardless of the number of classes. Also, although the weak hypotheses h_t are evaluated with respect to the pseudo-loss, we of course evaluate the final hypothesis h_{fin} using the ordinary error measure.

Theorem 2 ([10]) *Suppose the weak learning algorithm* **WeakLearn**, *when called by* **AdaBoost.M2** *generates hypotheses with pseudo-losses* $\epsilon_1, \ldots, \epsilon_T$, *where* ϵ_t *is as defined in Figure 2. Let* $\gamma_t = 1/2 - \epsilon_t$. *Then the following upper bound holds on the error of the final hypothesis* h_{fin}:

$$\begin{aligned} \frac{|\{i : h_{fin}(x_i) \neq y_i\}|}{m} &\leq (k-1) \prod_{t=1}^{T} \sqrt{1 - 4\gamma_t^2} \\ &\leq (k-1) \exp\left(-2 \sum_{t=1}^{T} \gamma_t^2\right) \end{aligned}$$

where k *is the number of classes.*

3 BOOSTING AND BAGGING

In this section, we describe our experiments comparing boosting and bagging on the UCI benchmarks.

We first mention briefly a small implementation issue: Many learning algorithms can be modified to handle examples that are weighted by a distribution such as the one created by the boosting algorithm. When this is possible, the booster's distribution D_t is supplied directly to the weak learning algorithm, a method we call boosting by *reweighting*. However, some learning algorithms require an unweighted set of examples. For such a weak learning algorithm, we instead choose a set of examples from S independently at random according to the distribution D_t with replacement. The number of examples to be chosen on each round is a matter of discretion; in our experiments, we chose m examples on each round, where m is the size of the original training set S. We refer to this method as boosting by *resampling*.

Boosting by resampling is also possible when using the pseudo-loss. In this case, a set of mislabels are chosen from the set B of all mislabels with replacement according to the given distribution D_t. Such a procedure is consistent with the interpretation of mislabels discussed in Section 2.2. In our experiments, we chose a sample of size $|B| = m(k-1)$ on each round when using the resampling method.

3.1 THE WEAK LEARNING ALGORITHMS

As mentioned in the introduction, we used three weak learning algorithms in these experiments. In all cases, the examples are described by a vector of values which corresponds to a fixed set of features or attributes. These values may be discrete or continuous. Some of the examples may have missing values. All three of the weak learners build hypotheses which classify examples by repeatedly testing the values of chosen attributes.

The first and simplest weak learner, which we call **FindAttrTest**, searches for the single attribute-value test

name	# examples train	test	# classes	# attributes disc.	cont.	missing values
soybean-small	47	-	4	35	-	-
labor	57	-	2	8	8	×
promoters	106	-	2	57	-	-
iris	150	-	3	-	4	-
hepatitis	155	-	2	13	6	×
sonar	208	-	2	-	60	-
glass	214	-	7	-	9	-
audiology.stand	226	-	24	69	-	×
cleve	303	-	2	7	6	×
soybean-large	307	376	19	35	-	×
ionosphere	351	-	2	-	34	-
house-votes-84	435	-	2	16	-	×
votes1	435	-	2	15	-	×
crx	690	-	2	9	6	×
breast-cancer-w	699	-	2	-	9	×
pima-indians-di	768	-	2	-	8	-
vehicle	846	-	4	-	18	-
vowel	528	462	11	-	10	-
german	1000	-	2	13	7	-
segmentation	2310	-	7	-	19	-
hypothyroid	3163	-	2	18	7	×
sick-euthyroid	3163	-	2	18	7	×
splice	3190	-	3	60	-	-
kr-vs-kp	3196	-	2	36	-	-
satimage	4435	2000	6	-	36	-
agaricus-lepiot	8124	-	2	22	-	-
letter-recognit	16000	4000	26	-	16	-

Table 1: The benchmark machine learning problems used in the experiments.

with minimum error (or pseudo-loss) on the training set. More precisely, **FindAttrTest** computes a classifier which is defined by an attribute a, a value v and three predictions p_0, p_1 and $p_?$. This classifier classifies a new example x as follows: if the value of attribute a is missing on x, then predict $p_?$; if attribute a is discrete and its value on example x is equal to v, or if attribute a is continuous and its value on x is at most v, then predict p_0; otherwise predict p_1. If using ordinary error (**AdaBoost.M1**), these "predictions" p_0, p_1, $p_?$ would be simple classifications; for pseudo-loss, the "predictions" would be vectors in $[0,1]^k$ (where k is the number of classes).

The algorithm **FindAttrTest** searches exhaustively for the classifier of the form given above with minimum error or pseudo-loss with respect to the distribution provided by the booster. In other words, all possible values of a, v, p_0, p_1 and $p_?$ are considered. With some preprocessing, this search can be carried out for the error-based implementation in $O(nm)$ time, where n is the number of attributes and m the number of examples. As is typical, the pseudo-loss implementation adds a factor of $O(k)$ where k is the number of class labels.

For this algorithm, we used boosting with reweighting.

The second weak learner does a somewhat more sophisticated search for a decision rule that tests on a conjunction of attribute-value tests. We sketch the main ideas of this algorithm, which we call **FindDecRule**, but omit some of the finer details for lack of space. These details will be provided in the full paper.

First, the algorithm requires an unweighted training set, so we use the resampling version of boosting. The given training set is randomly divided into a growing set using 70% of the data, and a pruning set with the remaining 30%.

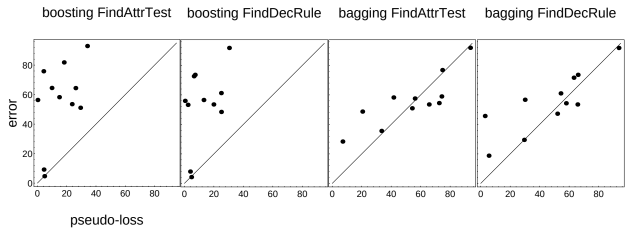

Figure 3: Comparison of using pseudo-loss versus ordinary error on multi-class problems for boosting and bagging.

In the first phase, the growing set is used to grow a list of attribute-value tests. Each test compares an attribute a to a value v, similar to the tests used by **FindAttrTest**. We use an entropy-based potential function to guide the growth of the list of tests. The list is initially empty, and one test is added at a time, each time choosing the test that will cause the greatest drop in potential. After the test is chosen, only one branch is expanded, namely, the branch with the highest remaining potential. The list continues to be grown in this fashion until no test remains which will further reduce the potential.

In the second phase, the list is pruned by selecting the prefix of the list with minimum error (or pseudo-loss) on the pruning set.

The third weak learner is Quinlan's **C4.5** decision-tree algorithm [18]. We used all the default options with pruning turned on. Since **C4.5** expects an unweighted training sample, we used resampling. Also, we did not attempt to use **AdaBoost.M2** since **C4.5** is designed to minimize error, not pseudo-loss. Furthermore, we did not expect pseudo-loss to be helpful when using a weak learning algorithm as strong as **C4.5**, since such an algorithm will usually be able to find a hypothesis with error less than $1/2$.

3.2 BAGGING

We compared boosting to Breiman's [1] "bootstrap aggregating" or "bagging" method for training and combining multiple copies of a learning algorithm. Briefly, the method works by training each copy of the algorithm on a bootstrap sample, i.e., a sample of size m chosen uniformly at random with replacement from the original training set S (of size m). The multiple hypotheses that are computed are then combined using simple voting; that is, the final composite hypothesis classifies an example x to the class most often assigned by the underlying "weak" hypotheses. See his paper for more details. The method can be quite effective, especially, according to Breiman, for "unstable" learning algorithms for which a small change in the data effects a large change in the computed hypothesis.

In order to compare **AdaBoost.M2**, which uses pseudo-loss, to bagging, we also extended bagging in a natural way for use with a weak learning algorithm that minimizes pseudo-loss rather than ordinary error. As described in Section 2.2, such a weak learning algorithm expects to be provided with a distribution over the set B of all mislabels. On each round of bagging, we construct this distribution using the bootstrap method; that is, we select $|B|$ mislabels from B (chosen uniformly at random with replacement),

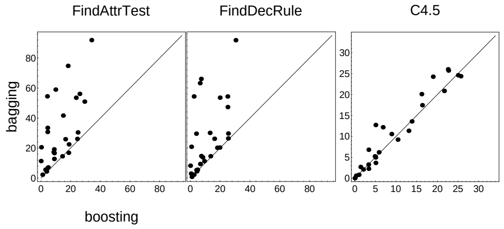

Figure 4: Comparison of boosting and bagging for each of the weak learners.

and assign each mislabel weight $1/|B|$ times the number of times it was chosen. The hypotheses h_t computed in this manner are then combined using voting in a natural manner; namely, given x, the combined hypothesis outputs the label y which maximizes $\sum_t h_t(x, y)$.

For either error or pseudo-loss, the differences between bagging and boosting can be summarized as follows: (1) bagging always uses resampling rather than reweighting; (2) bagging does not modify the distribution over examples or mislabels, but instead always uses the uniform distribution; and (3) in forming the final hypothesis, bagging gives equal weight to each of the weak hypotheses.

3.3 THE EXPERIMENTS

We conducted our experiments on a collection of machine learning datasets available from the repository at University of California at Irvine.[3] A summary of some of the properties of these datasets is given in Table 1. Some datasets are provided with a test set. For these, we reran each algorithm 20 times (since some of the algorithms are randomized), and averaged the results. For datasets with no provided test set, we used 10-fold cross validation, and averaged the results over 10 runs (for a total of 100 runs of each algorithm on each dataset).

In all our experiments, we set the number of rounds of boosting or bagging to be $T = 100$.

3.4 RESULTS AND DISCUSSION

The results of our experiments are shown in Table 2. The figures indicate test error rate averaged over multiple runs of each algorithm. Columns indicate which weak learning algorithm was used, and whether pseudo-loss (**AdaBoost.M2**) or error (**AdaBoost.M1**) was used. Note that pseudo-loss was not used on any two-class problems since the resulting algorithm would be identical to the corresponding error-based algorithm. Columns labeled "–" indicate that the weak learning algorithm was used by itself (with no boosting or bagging). Columns using boosting or bagging are marked "boost" and "bag," respectively.

One of our goals in carrying out these experiments was to determine if boosting using pseudo-loss (rather than error) is worthwhile. Figure 3 shows how the different algorithms performed on each of the many-class ($k > 2$) problems using pseudo-loss versus error. Each point in the scatter plot represents the error achieved by the two competing algorithms on a given benchmark, so there is one point

[3] URL "http://www.ics.uci.edu/~mlearn/MLRepository.html"

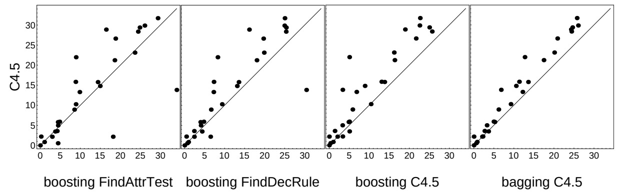

Figure 5: Comparison of **C4.5** versus various other boosting and bagging methods.

for each benchmark. These experiments indicate that boosting using pseudo-loss clearly outperforms boosting using error. Using pseudo-loss did dramatically better than error on every non-binary problem (except it did slightly worse on "iris" with three classes). Because **AdaBoost.M2** did so much better than **AdaBoost.M1**, we will only discuss **AdaBoost.M2** henceforth.

As the figure shows, using pseudo-loss with bagging gave mixed results in comparison to ordinary error. Overall, pseudo-loss gave better results, but occasionally, using pseudo-loss hurt considerably.

Figure 4 shows similar scatterplots comparing the performance of boosting and bagging for all the benchmarks and all three weak learner. For boosting, we plotted the error rate achieved using pseudo-loss. To present bagging in the best possible light, we used the error rate achieved using either error or pseudo-loss, whichever gave the better result on that particular benchmark. (For the binary problems, and experiments with **C4.5**, only error was used.)

For the simpler weak learning algorithms (**FindAttrTest** and **FindDecRule**), boosting did significantly and uniformly better than bagging. The boosting error rate was worse than the bagging error rate (using either pseudo-loss or error) on a very small number of benchmark problems, and on these, the difference in performance was quite small. On average, for **FindAttrTest**, boosting improved the error rate over using **FindAttrTest** alone by 55.2%, compared to bagging which gave an improvement of only 11.0% using pseudo-loss or 8.4% using error. For **FindDecRule**, boosting improved the error rate by 53.0%, bagging by only 18.8% using pseudo-loss, 13.1% using error.

When using **C4.5** as the weak learning algorithm, boosting and bagging seem more evenly matched, although boosting still seems to have a slight advantage. On average, boosting improved the error rate by 24.8%, bagging by 20.0%. Boosting beat bagging by more than 2% on 6 of the benchmarks, while bagging did not beat boosting by this amount on any benchmark. For the remaining 20 benchmarks, the difference in performance was less than 2%.

Figure 5 shows in a similar manner how **C4.5** performed compared to bagging with **C4.5**, and compared to boosting with each of the weak learners (using pseudo-loss for the non-binary problems). As the figure shows, using boosting with **FindAttrTest** does quite well as a learning algorithm in its own right, in comparison to **C4.5**. This algorithm beat **C4.5** on 10 of the benchmarks (by at least 2%), tied on 14, and lost on 3. As mentioned above, its average performance relative to using **FindAttrTest** by itself was 55.2%. In comparison, **C4.5**'s improvement in performance

	FindAttrTest					**FindDecRule**					**C4.5**		
		error		pseudo-loss			error		pseudo-loss			error	
name	–	boost	bag	boost	bag	–	boost	bag	boost	bag	–	boost	bag
soybean-small	57.6	56.4	48.7	0.2	20.5	51.8	56.0	45.7	0.4	2.9	2.2	3.4	2.2
labor	25.1	8.8	19.1			24.0	7.3	14.6			15.8	13.1	11.3
promoters	29.7	8.9	16.6			25.9	8.3	13.7			22.0	5.0	12.7
iris	35.2	4.7	28.4	4.8	7.1	38.3	4.3	18.8	4.8	5.5	5.9	5.0	5.0
hepatitis	19.7	18.6	16.8			21.6	18.0	20.1			21.2	16.3	17.5
sonar	25.9	16.5	25.9			31.4	16.2	26.1			28.9	19.0	24.3
glass	51.5	51.1	50.9	29.4	54.2	49.7	48.5	47.2	25.0	52.0	31.7	22.7	25.7
audiology.stand	53.5	53.5	53.5	23.6	65.7	53.5	53.5	53.5	19.9	65.7	23.1	16.2	20.1
cleve	27.8	18.8	22.4			27.4	19.7	20.3			26.6	21.7	20.9
soybean-large	64.8	64.5	59.0	9.8	74.2	73.6	73.6	73.6	7.2	66.0	13.3	6.8	12.2
ionosphere	17.8	8.5	17.3			10.3	6.6	9.3			8.9	5.8	6.2
house-votes-84	4.4	3.7	4.4			5.0	4.4	4.4			3.5	5.1	3.6
votes1	12.7	8.9	12.7			13.2	9.4	11.2			10.3	10.4	9.2
crx	14.5	14.4	14.5			14.5	13.5	14.5			15.8	13.8	13.6
breast-cancer-w	8.4	4.4	6.7			8.1	4.1	5.3			5.0	3.3	3.2
pima-indians-di	26.1	24.4	26.1			27.8	25.3	26.4			28.4	25.7	24.4
vehicle	64.3	64.4	57.6	26.1	56.1	61.3	61.2	61.0	25.0	54.3	29.9	22.6	26.1
vowel	81.8	81.8	76.8	18.2	74.7	82.0	72.7	71.6	6.5	63.2	2.2	0.0	0.0
german	30.0	24.9	30.4			30.0	25.4	29.6			29.4	25.0	24.6
segmentation	75.8	75.8	54.5	4.2	72.5	73.7	53.3	54.3	2.4	58.0	3.6	1.4	2.7
hypothyroid	2.2	1.0	2.2			0.8	1.0	0.7			0.8	1.0	0.8
sick-euthyroid	5.6	3.0	5.6			2.4	2.4	2.2			2.2	2.1	2.1
splice	37.0	9.2	35.6	4.4	33.4	29.5	8.0	29.5	4.0	29.5	5.8	4.9	5.2
kr-vs-kp	32.8	4.4	30.7			24.6	0.7	20.8			0.5	0.3	0.6
satimage	58.3	58.3	58.3	14.9	41.6	57.6	56.5	56.7	13.1	30.0	14.8	8.9	10.6
agaricus-lepiot	11.3	0.0	11.3			8.2	0.0	8.2			0.0	0.0	0.0
letter-recognit	92.9	92.9	91.9	34.1	93.7	92.3	91.8	91.8	30.4	93.7	13.8	3.3	6.8

Table 2: Test error rates of various algorithms on benchmark problems.

over **FindAttrTest** was 49.3%.

Using boosting with **FindDecRule** did somewhat better. The win-tie-lose numbers for this algorithm (compared to **C4.5**) were 13-12-2, and its average improvement over **FindAttrTest** was 58.1%.

4 BOOSTING A NEAREST-NEIGHBOR CLASSIFIER

In this section we study the performance of a learning algorithm which combines **AdaBoost** and a variant of the nearest-neighbor classifier. We test the combined algorithm on the problem of recognizing handwritten digits. Our goal is not to improve on the accuracy of the nearest neighbor classifier, but rather to speed it up. Speed-up is achieved by reducing the number of prototypes in the hypothesis (and thus the required number of distance calculations) without increasing the error rate. It is a similar approach to that of nearest-neighbor editing [12, 13] in which one tries to find the minimal set of prototypes that is sufficient to label all the training set correctly.

The dataset comes from the US Postal Service (USPS) and consists of 9709 training examples and 2007 test examples. The training and test examples are evidently drawn from rather different distributions as there is a very significant improvement in the performance if the partition of the data into training and testing is done at random (rather than using the given partition). We report results both on the original partitioning and on a training set and a test set of the same sizes that were generated by randomly partitioning the union of the original training and test sets.

Each image is represented by a 16×16-matrix of 8-bit pixels. The metric that we use for identifying the nearest neighbor, and hence for classifying an instance, is the standard Euclidean distance between the images (viewed as vectors in $\mathbb{R}^{256}$). This is a very naive metric, but it gives reasonably good performance. A nearest-neighbor classifier which uses all the training examples as prototypes achieves a test error of 5.7% (2.3% on randomly partitioned data). Using the more sophisticated tangent distance [21] is in our future plans.

Each weak hypothesis is defined by a subset P of the training examples, and a mapping $\pi : P \rightarrow [0,1]^k$. Given a new test point x, such a weak hypothesis predicts the vector $\pi(x_0)$ where $x_0 \in P$ is the closest point to x.

On each round of boosting, a weak hypothesis is generated by adding one prototype at a time to the set P until the set reaches a prespecified size. Given any set P, we always choose the mapping π which minimizes the pseudo-loss of the resulting weak hypothesis (with respect to the given mislabel distribution).

Initially, the set of prototypes P is empty. Next, ten candidate prototypes are selected at random according to the current (marginal) distribution over the training examples. Of these candidates, the one that causes the largest decrease in the pseudo-loss is added to the set P, and the process is repeated. The boosting process thus influences the weak learning algorithm in two ways: first, by changing the way the ten random examples are selected, and second by changing the calculation of the pseudo-loss.

It often happens that, on the following round of boosting, the same set P will have pseudo-loss significantly less than $1/2$ with respect to the new mislabel distribution (but possibly using a different mapping π). In this case, rather than choosing a new set of prototypes, we reuse the same set P in additional boosting steps until the advantage that can be gained from the given partition is exhausted (details

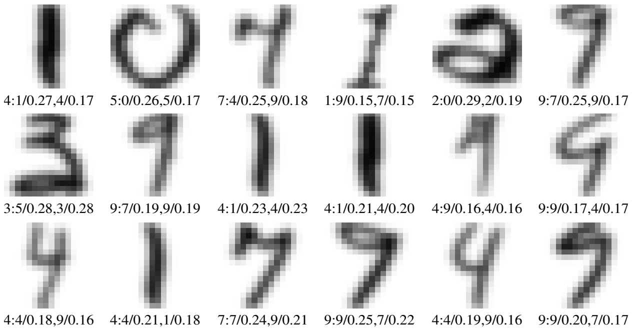

Figure 6: A sample of the examples that have the largest weight after 3 of the 30 boosting iterations. The first line is after iteration 4, the second after iteration 12 and the third after iteration 25. Underneath each image we have a line of the form d:ℓ_1/w_1,ℓ_2/w_2, where d is the label of the example, ℓ_1 and ℓ_2 are the labels that get the highest and second highest vote from the combined hypothesis at that point in the run of the algorithm, and w_1, w_2 are the corresponding normalized votes.

omitted).

We ran 30 iterations of the boosting algorithm, and the number of prototypes we used were 10 for the first weak hypothesis, 20 for the second, 40 for the third, 80 for the next five, and 100 for the remaining twenty-two weak hypotheses. These sizes were chosen so that the errors of all of the weak hypotheses are approximately equal.

We compared the performance of our algorithm to a strawman algorithm which uses a single set of prototypes. Similar to our algorithm, the prototype set is generated incrementally, comparing ten prototype candidates at each step, and always choosing the one that minimizes the empirical error. We compared the performance of the boosting algorithm to that of the strawman hypothesis that uses the same number of prototypes. We also compared our performance to that of the condensed nearest neighbor rule (CNN) [13], a greedy method for finding a small set of prototypes which correctly classify the entire training set.

4.1 RESULTS AND DISCUSSION

The results of our experiments are summarized in Table 3 and Figure 7. Table 3 describes the results from experiments with **AdaBoost** (each experiment was repeated 10 times using different random seeds), the strawman algorithm (each repeated 7 times) , and CNN (7 times). We compare the results using a random partition of the data into training and testing and using the partition that was defined by USPS.

We see that in both cases, after more than 970 examples, the training error of **AdaBoost** is much better than that of the strawman algorithm. The performance on the test set is similar, with a slight advantage to **AdaBoost** when the hypotheses include more than 1670 examples, but a slight advantage to strawman if fewer rounds of boosting are used. After 2670 examples, the error of **AdaBoost** on the random partition is (on average) 2.7%, while the error achieved by using the whole training set is 2.3%. On the random partition, the final error is 6.4%, while the error using the whole training set is 5.7%.

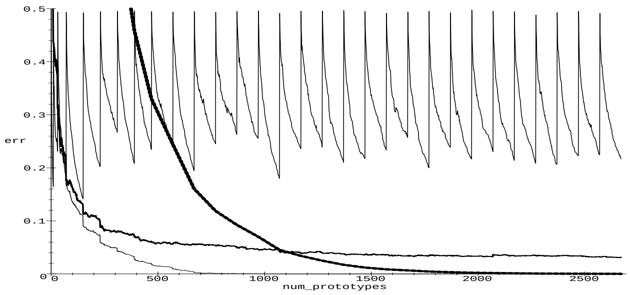

Figure 7: Graphs of the performance of the boosting algorithm on a randomly partitioned USPS dataset. The horizontal axis indicates the total number of prototypes that were added to the combined hypothesis, and the vertical axis indicates error. The topmost jagged line indicates the error of the weak hypothesis that is trained at this point on the weighted training set. The bold curve is the bound on the training error calculated using Theorem 2. The lowest thin curve and the medium-bold curve show the performance of the combined hypothesis on the training set and test set, respectively.

Comparing to CNN, we see that both the strawman algorithm and **AdaBoost** perform better than CNN even when they use about 900 examples in their hypotheses. Larger hypotheses generated by **AdaBoost** or strawman are much better than that generated by CNN. The main problem with CNN seems to be its tendency to overfit the training data. **AdaBoost** and the strawman algorithm seem to suffer less from overfitting.

Figure 7 shows a typical run of **AdaBoost**. The uppermost jagged line is a concatenation of the errors of the weak hypotheses with respect to the mislabel distribution. Each peak followed by a valley corresponds to the beginning and end errors of a weak hypothesis as it is being constructed, one prototype at a time. The weighted error always starts around 50% at the beginning of a boosting iteration and drops to around 20-30%. The heaviest line describes the upper bound on the training error that is guaranteed by Theorem 2, and the two bottom lines describe the training and test error of the final combined hypothesis.

It is interesting that the performance of the boosting algorithm on the test set improved significantly after the error on the training set has already become zero. This is surprising because an "Occam's razor" argument would predict that increasing the complexity of the hypothesis after the error has been reduced to zero is likely to degrade the performance on the test set.

Figure 6 shows a sample of the examples that are given large weights by the boosting algorithm on a typical run. There seem to be two types of "hard" examples. First are examples which are very atypical or wrongly labeled (such as example 2 on the first line and examples 3 and 4 on the second line). The second type, which tends to dominate on later iterations, consists of examples that are very similar to each other but have different labels (such as examples 3 versus 4 on the third line). Although the algorithm at this point was correct on all training examples, it is clear from the votes it assigned to different labels for these example pairs that it was still trying to improve the discrimination

		random partition						USPS partition					
		AdaBoost			**Strawman**		**CNN**	**AdaBoost**			**Strawman**		**CNN**
rnd	size	theory	train	test	train	test	test (size)	theory	train	test	train	test	test (size)
1	10	524.6	45.9	46.1	37.9	38.3		536.3	42.5	43.1	36.1	37.6	
5	230	86.4	6.3	8.5	4.9	6.2		83.0	5.1	12.3	4.2	10.6	
10	670	16.0	0.4	4.6	2.0	4.3		10.9	0.1	8.6	1.4	8.3	
13	970	4.5	0.0	3.9	1.5	3.8	4.4 (990)	3.3	0.0	8.1	1.0	7.7	8.6 (865)
15	1170	2.4	0.0	3.6	1.3	3.6		1.5	0.0	7.7	0.8	7.5	
20	1670	0.4	0.0	3.1	0.9	3.3		0.2	0.0	7.0	0.6	7.1	
25	2170	0.1	0.0	2.9	0.7	3.0		0.0	0.0	6.7	0.4	6.9	
30	2670	0.0	0.0	2.7	0.5	2.8		0.0	0.0	6.4	0.3	6.8	

Table 3: Average error rates on training and test sets, in percent. For columns labeled "random partition," a random partition of the union of the training and test sets was used; "USPS partition" means the USPS-provided partition into training and test sets was used. Columns labeled "theory" give theoretical upper bounds on training error calculated using Theorem 2. "Size" indicates number of prototypes defining the final hypothesis.

between similar examples. This agrees with our intuition that the pseudo-loss is a mechanism that causes the boosting algorithm to concentrate on the hard to discriminate labels of hard examples.

5 CONCLUSIONS

We have demonstrated that **AdaBoost** can be used in many settings to improve the performance of a learning algorithm. When starting with relatively simple classifiers, the improvement can be especially dramatic, and can often lead to a composite classifier that outperforms more complex "one-shot" learning algorithms like **C4.5**. This improvement is far greater than can be achieved with bagging. Note, however, that for non-binary classification problems, boosting simple classifiers can only be done effectively if the more sophisticated pseudo-loss is used.

When starting with a complex algorithm like **C4.5**, boosting can also be used to improve performance, but does not have such a compelling advantage over bagging. Boosting combined with a complex algorithm may give the greatest improvement in performance when there is a reasonably large amount of data available (note, for instance, boosting's performance on the "letter-recognition" problem with 16,000 training examples). Naturally, one needs to consider whether the improvement in error is worth the additional computation time. Although we used 100 rounds of boosting, Quinlan [19] got good results using only 10 rounds.

Boosting may have other applications, besides reducing the error of a classifier. For instance, we saw in Section 4 that boosting can be used to find a small set of prototypes for a nearest neighbor classifier.

As described in the introduction, boosting combines two effects. It reduces the bias of the weak learner by forcing the weak learner to concentrate on different parts of the instance space, and it also reduces the variance of the weak learner by averaging several hypotheses that were generated from different subsamples of the training set. While there is good theory to explain the bias reducing effects, there is need for a better theory of the variance reduction.

Acknowledgements. Thanks to Jason Catlett and William Cohen for extensive advice on the design of our experiments. Thanks to Ross Quinlan for first suggesting a comparison of boosting and bagging. Thanks also to Leo Breiman, Corinna Cortes, Harris Drucker, Jeff Jackson, Michael Kearns, Ofer Matan, Partha Niyogi, Warren Smith, David Wolpert and the anonymous ICML reviewers for helpful comments, suggestions and criticisms. Finally, thanks to all who contributed to the datasets used in this paper.

References

[1] Leo Breiman. Bagging predictors. Technical Report 421, Department of Statistics, University of California at Berkeley, 1994.

[2] Leo Breiman. Bias, variance, and arcing classifiers. Unpublished manuscript, 1996.

[3] William Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123, 1995.

[4] Tom Dietterich, Michael Kearns, and Yishay Mansour. Applying the weak learning framework to understand and improve C4.5. In *Machine Learning: Proceedings of the Thirteenth International Conference*, 1996.

[5] Harris Drucker and Corinna Cortes. Boosting decision trees. In *Advances in Neural Information Processing Systems 8*, 1996.

[6] Harris Drucker, Corinna Cortes, L. D. Jackel, Yann LeCun, and Vladimir Vapnik. Boosting and other ensemble methods. *Neural Computation*, 6(6):1289–1301, 1994.

[7] Harris Drucker, Robert Schapire, and Patrice Simard. Boosting performance in neural networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(4):705–719, 1993.

[8] Harris Drucker, Robert Schapire, and Patrice Simard. Improving performance in neural networks using a boosting algorithm. In *Advances in Neural Information Processing Systems 5*, pages 42–49, 1993.

[9] Yoav Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, 1995.

[10] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. Unpublished manuscript available electronically (on our web pages, or by email request). An extended abstract appeared in *Computational Learning Theory: Second European Conference, EuroCOLT '95*, pages 23–37, Springer-Verlag, 1995.

[11] Johannes Fürnkranz and Gerhard Widmer. Incremental reduced error pruning. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 70–77, 1994.

[12] Geoffrey W. Gates. The reduced nearest neighbor rule. *IEEE Transactions on Information Theory*, pages 431–433, 1972.

[13] Peter E. Hart. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, IT-14:515–516, May 1968.

[14] Robert C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–91, 1993.

[15] Jeffrey C. Jackson and Mark W. Craven. Learning sparse perceptrons. In *Advances in Neural Information Processing Systems 8*, 1996.

[16] Michael Kearns and Yishay Mansour. On the boosting ability of top-down decision tree learning algorithms. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, 1996.

[17] Eun Bae Kong and Thomas G. Dietterich. Error-correcting output coding corrects bias and variance. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 313–321, 1995.

[18] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[19] J. Ross Quinlan. Bagging, boosting, and C4.5. In *Proceedings, Fourteenth National Conference on Artificial Intelligence*, 1996.

[20] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[21] Patrice Simard, Yann Le Cun, and John Denker. Efficient pattern recognition using a new transformation distance. In *Advances in Neural Information Processing Systems*, volume 5, pages 50–58, 1993.