Ensemble Methods in Machine Learning

Thomas G. Dietterich

Oregon State University, Corvallis, Oregon, USA,
tgd@cs.orst.edu,
WWW home page: http://www.cs.orst.edu/~tgd

Abstract. Ensemble methods are learning algorithms that construct a set of classifiers and then classify new data points by taking a (weighted) vote of their predictions. The original ensemble method is Bayesian averaging, but more recent algorithms include error-correcting output coding, Bagging, and boosting. This paper reviews these methods and explains why ensembles can often perform better than any single classifier. Some previous studies comparing ensemble methods are reviewed, and some new experiments are presented to uncover the reasons that Adaboost does not overfit rapidly.

1 Introduction

Consider the standard supervised learning problem. A learning program is given training examples of the form $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ for some unknown function $y = f(\mathbf{x})$. The $\mathbf{x}_i$ values are typically vectors of the form $\langle x_{i,1}, x_{i,2}, \ldots, x_{i,n} \rangle$ whose components are discrete- or real-valued such as height, weight, color, age, and so on. These are also called the *features* of $\mathbf{x}_i$. Let us use the notation x_{ij} to refer to the j-th feature of $\mathbf{x}_i$. In some situations, we will drop the i subscript when it is implied by the context.

The y values are typically drawn from a discrete set of classes $\{1, \ldots, K\}$ in the case of *classification* or from the real line in the case of *regression*. In this chapter, we will consider only classification. The training examples may be corrupted by some random noise.

Given a set S of training examples, a learning algorithm outputs a *classifier*. The classifier is an hypothesis about the true function f. Given new $\mathbf{x}$ values, it predicts the corresponding y values. I will denote classifiers by $h_1, \ldots, h_L$.

An ensemble of classifiers is a set of classifiers whose individual decisions are combined in some way (typically by weighted or unweighted voting) to classify new examples. One of the most active areas of research in supervised learning has been to study methods for constructing good ensembles of classifiers. The main discovery is that ensembles are often much more accurate than the individual classifiers that make them up.

A necessary and sufficient condition for an ensemble of classifiers to be more accurate than any of its individual members is if the classifiers are accurate and diverse (Hansen & Salamon, 1990). An accurate classifier is one that has an error rate of better than random guessing on new $\mathbf{x}$ values. Two classifiers are diverse if they make different errors on new data points. To see why accuracy and diversity are good, imagine that we have an ensemble of three classifiers: $\{h_1, h_2, h_3\}$ and consider a new case $\mathbf{x}$. If the three classifiers are identical (i.e., not diverse), then when $h_1(\mathbf{x})$ is wrong, $h_2(\mathbf{x})$ and $h_3(\mathbf{x})$ will also be wrong. However, if the errors made by the classifiers are uncorrelated, then when $h_1(\mathbf{x})$ is wrong, $h_2(\mathbf{x})$ and $h_3(\mathbf{x})$ may be correct, so that a majority vote will correctly classify $\mathbf{x}$. More precisely, if the error rates of L hypotheses h_ℓ are all equal to $p < 1/2$ and if the errors are independent, then the probability that the majority vote will be wrong will be the area under the binomial distribution where more than $L/2$ hypotheses are wrong. Figure 1 shows this for a simulated ensemble of 21 hypotheses, each having an error rate of 0.3. The area under the curve for 11 or more hypotheses being simultaneously wrong is 0.026, which is much less than the error rate of the individual hypotheses.

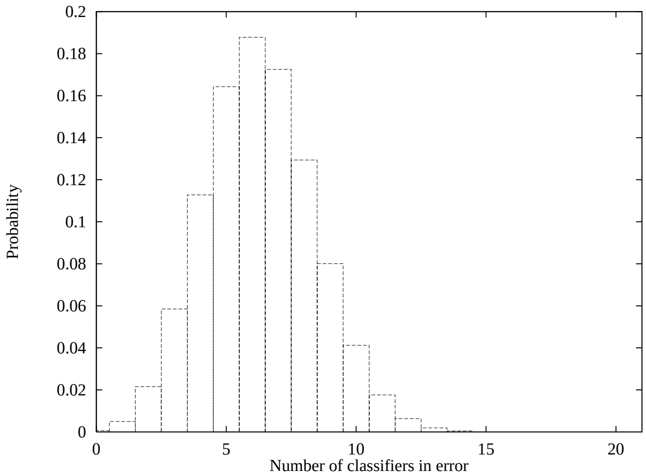

Fig. 1. The probability that exactly ℓ (of 21) hypotheses will make an error, assuming each hypothesis has an error rate of 0.3 and makes its errors independently of the other hypotheses.

Of course, if the individual hypotheses make uncorrelated errors at rates exceeding 0.5, then the error rate of the voted ensemble will *increase* as a result of the voting. Hence, one key to successful ensemble methods is to construct individual classifiers with error rates below 0.5 whose errors are at least somewhat uncorrelated.

This formal characterization of the problem is intriguing, but it does not address the question of whether it is possible in practice to construct good ensembles. Fortunately, it is often possible to construct very good ensembles. There are three fundamental reasons for this.

The first reason is statistical. A learning algorithm can be viewed as searching a space $\mathcal{H}$ of hypotheses to identify the best hypothesis in the space. The statistical problem arises when the amount of training data available is too small compared to the size of the hypothesis space. Without sufficient data, the learning algorithm can find many different hypotheses in $\mathcal{H}$ that all give the same accuracy on the training data. By constructing an ensemble out of all of these accurate classifiers, the algorithm can "average" their votes and reduce the risk of choosing the wrong classifier. Figure 2(top left) depicts this situation. The outer curve denotes the hypothesis space $\mathcal{H}$. The inner curve denotes the set of hypotheses that all give good accuracy on the training data. The point labeled f is the true hypothesis, and we can see that by averaging the accurate hypotheses, we can find a good approximation to f.

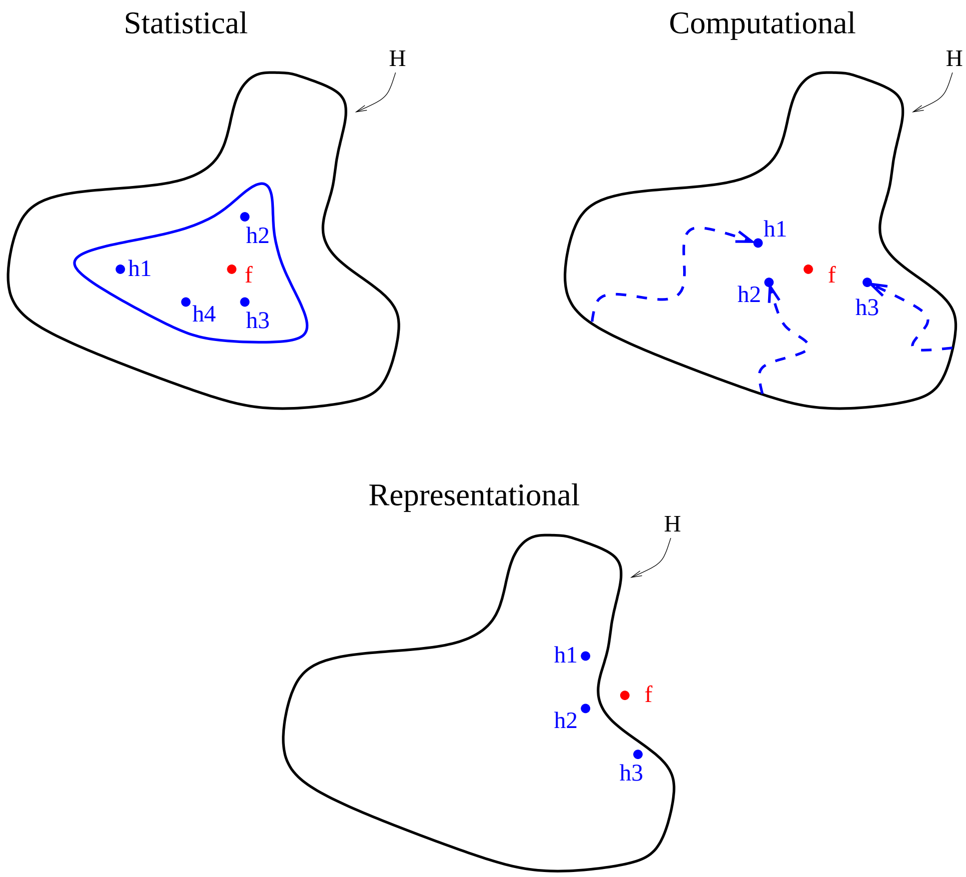

Fig. 2. Three fundamental reasons why an ensemble may work better than a single classifier

The second reason is computational. Many learning algorithms work by performing some form of local search that may get stuck in local optima. For example, neural network algorithms employ gradient descent to minimize an error function over the training data, and decision tree algorithms employ a greedy splitting rule to grow the decision tree. In cases where there is enough training data (so that the statistical problem is absent), it may still be very difficult computationally for the learning algorithm to find the best hypothesis. Indeed, optimal training of both neural networks and decisions trees is NP-hard (Hyafil & Rivest, 1976; Blum & Rivest, 1988). An ensemble constructed by running the local search from many different starting points may provide a better approximation to the true unknown function than any of the individual classifiers, as shown in Figure 2 (top right).

The third reason is representational. In most applications of machine learning, the true function f cannot be represented by any of the hypotheses in $\mathcal{H}$. By forming weighted sums of hypotheses drawn from $\mathcal{H}$, it may be possible to expand the space of representable functions. Figure 2 (bottom) depicts this situation.

The representational issue is somewhat subtle, because there are many learning algorithms for which $\mathcal{H}$ is, in principle, the space of all possible classifiers. For example, neural networks and decision trees are both very flexible algorithms. Given enough training data, they will explore the space of all possible classifiers, and several people have proved asymptotic representation theorems for them (Hornik, Stinchcombe, & White, 1990). Nonetheless, with a finite training sample, these algorithms will explore only a finite set of hypotheses and they will stop searching when they find an hypothesis that fits the training data. Hence, in Figure 2, we must consider the space $\mathcal{H}$ to be the effective space of hypotheses searched by the learning algorithm for a given training data set.

These three fundamental issues are the three most important ways in which existing learning algorithms fail. Hence, ensemble methods have the promise of reducing (and perhaps even eliminating) these three key shortcomings of standard learning algorithms.

2 Methods for Constructing Ensembles

Many methods for constructing ensembles have been developed. Here we will review general purpose methods that can be applied to many different learning algorithms.

2.1 Bayesian Voting: Enumerating the Hypotheses

In a Bayesian probabilistic setting, each hypothesis h defines a conditional probability distribution: $h(\mathbf{x}) = P(f(\mathbf{x}) = y|\mathbf{x}, h)$. Given a new data point $\mathbf{x}$ and a training sample S, the problem of predicting the value of $f(\mathbf{x})$ can be viewed as the problem of computing $P(f(\mathbf{x}) = y|S, \mathbf{x})$. We can rewrite this as weighted sum over all hypotheses in $\mathcal{H}$:

$$P(f(\mathbf{x}) = y|S, \mathbf{x}) = \sum_{h \in \mathcal{H}} h(\mathbf{x}) P(h|S).$$

We can view this as an ensemble method in which the ensemble consists of all of the hypotheses in $\mathcal{H}$, each weighted by its posterior probability $P(h|S)$. By Bayes rule, the posterior probability is proportional to the likelihood of the training data times the prior probability of h:

$$P(h|S) \propto P(S|h)P(h).$$

In some learning problems, it is possible to completely enumerate each $h \in \mathcal{H}$, compute $P(S|h)$ and $P(h)$, and (after normalization), evaluate this Bayesian "committee." Furthermore, if the true function f is drawn from $\mathcal{H}$ according to $P(h)$, then the Bayesian voting scheme is optimal.

Bayesian voting primarily addresses the statistical component of ensembles. When the training sample is small, many hypotheses h will have significantly large posterior probabilities, and the voting process can average these to "marginalize away" the remaining uncertainty about f. When the training sample is large, typically only one hypothesis has substantial posterior probability, and the "ensemble" effectively shrinks to contain only a single hypothesis.

In complex problems where $\mathcal{H}$ cannot be enumerated, it is sometimes possible to approximate Bayesian voting by drawing a random sample of hypotheses distributed according to $P(h|S)$. Recent work on Markov chain Monte Carlo methods (Neal, 1993) seeks to develop a set of tools for this task.

The most idealized aspect of the Bayesian analysis is the prior belief $P(h)$. If this prior completely captures all of the knowledge that we have about f before we obtain S, then by definition we cannot do better. But in practice, it is often difficult to construct a space $\mathcal{H}$ and assign a prior $P(h)$ that captures our prior knowledge adequately. Indeed, often $\mathcal{H}$ and $P(h)$ are chosen for computational convenience, and they are known to be inadequate. In such cases, the Bayesian committee is not optimal, and other ensemble methods may produce better results. In particular, the Bayesian approach does not address the computational and representational problems in any significant way.

2.2 Manipulating the Training Examples

The second method for constructing ensembles manipulates the training examples to generate multiple hypotheses. The learning algorithm is run several times, each time with a different subset of the training examples. This technique works especially well for *unstable* learning algorithms—algorithms whose output classifier undergoes major changes in response to small changes in the training data. Decision-tree, neural network, and rule learning algorithms are all unstable. Linear regression, nearest neighbor, and linear threshold algorithms are generally very stable.

The most straightforward way of manipulating the training set is called *Bagging*. On each run, Bagging presents the learning algorithm with a training set that consists of a sample of m training examples drawn randomly with replacement from the original training set of m items. Such a training set is called a *bootstrap replicate* of the original training set, and the technique is called *bootstrap aggregation* (from which the term *Bagging* is derived; Breiman, 1996). Each bootstrap replicate contains, on the average, 63.2% of the original training set, with several training examples appearing multiple times.

Another training set sampling method is to construct the training sets by leaving out disjoint subsets of the training data. For example, the training set can be randomly divided into 10 disjoint subsets. Then 10 overlapping training sets can be constructed by dropping out a different one of these 10 subsets. This same procedure is employed to construct training sets for 10-fold cross-validation, so ensembles constructed in this way are sometimes called *cross-validated committees* (Parmanto, Munro, & Doyle, 1996).

The third method for manipulating the training set is illustrated by the AdaBoost algorithm, developed by Freund and Schapire (1995, 1996, 1997, 1998). Like Bagging, AdaBoost manipulates the training examples to generate multiple hypotheses. AdaBoost maintains a set of weights over the training examples. In each iteration ℓ, the learning algorithm is invoked to minimize the weighted error on the training set, and it returns an hypothesis h_ℓ. The weighted error of h_ℓ is computed and applied to update the weights on the training examples. The effect of the change in weights is to place more weight on training examples that were misclassified by h_ℓ and less weight on examples that were correctly classified. In subsequent iterations, therefore, AdaBoost constructs progressively more difficult learning problems.

The final classifier, $h_f(x) = \sum_\ell w_\ell h_\ell(x)$, is constructed by a weighted vote of the individual classifiers. Each classifier is weighted (by w_ℓ) according to its accuracy on the weighted training set that it was trained on.

Recent research (Schapire & Singer, 1998) has shown that AdaBoost can be viewed as a stage-wise algorithm for minimizing a particular error function. To define this error function, suppose that each training example is labeled as $+1$ or -1, corresponding to the positive and negative examples. Then the quantity $m_i = y_i h(x_i)$ is positive if h correctly classifies x_i and negative otherwise. This quantity m_i is called the *margin* of classifier h on the training data. AdaBoost can be seen as trying to minimize

$$\sum_i exp\left(-y_i \sum_\ell w_\ell h_\ell(x_i)\right), \tag{1}$$

which is the negative exponential of the margin of the weighted voted classifier. This can also be viewed as attempting to maximize the margin on the training data.

2.3 Manipulating the Input Features

A third general technique for generating multiple classifiers is to manipulate the set of *input features* available to the learning algorithm. For example, in a project to identify volcanoes on Venus, Cherkauer (1996) trained an ensemble of 32 neural networks. The 32 networks were based on 8 different subsets of the 119 available input features and 4 different network sizes. The input feature subsets were selected (by hand) to group together features that were based on different image processing operations (such as principal component analysis and the fast fourier transform). The resulting ensemble classifier was able to match the performance of human experts in identifying volcanoes. Tumer and Ghosh (1996) applied a similar technique to a sonar dataset with 25 input features. However, they found that deleting even a few of the input features hurt the performance of the individual classifiers so much that the voted ensemble did not perform very well. Obviously, this technique only works when the input features are highly redundant.

2.4 Manipulating the Output Targets

A fourth general technique for constructing a good ensemble of classifiers is to manipulate the y values that are given to the learning algorithm. Dietterich & Bakiri (1995) describe a technique called error-correcting output coding. Suppose that the number of classes, K, is large. Then new learning problems can be constructed by randomly partioning the K classes into two subsets A_ℓ and B_ℓ. The input data can then be re-labeled so that any of the original classes in set A_ℓ are given the derived label 0 and the original classes in set B_ℓ are given the derived label 1. This relabeled data is then given to the learning algorithm, which constructs a classifier h_ℓ. By repeating this process L times (generating different subsets A_ℓ and B_ℓ), we obtain an ensemble of L classifiers $h_1, \ldots, h_L$.

Now given a new data point $\mathbf{x}$, how should we classify it? The answer is to have each h_ℓ classify $\mathbf{x}$. If $h_\ell(\mathbf{x}) = 0$, then each class in A_ℓ receives a vote. If $h_\ell(\mathbf{x}) = 1$, then each class in B_ℓ receives a vote. After each of the L classifiers has voted, the class with the highest number of votes is selected as the prediction of the ensemble.

An equivalent way of thinking about this method is that each class j is encoded as an L-bit codeword C_j, where bit ℓ is 1 if and only if $j \in B_\ell$. The ℓ-th learned classifier attempts to predict bit ℓ of these codewords. When the L classifiers are applied to classify a new point $\mathbf{x}$, their predictions are combined into an L-bit string. We then choose the class j whose codeword C_j is closest (in Hamming distance) to the L-bit output string. Methods for designing good error-correcting codes can be applied to choose the codewords C_j (or equivalently, subsets A_ℓ and B_ℓ).

Dietterich and Bakiri report that this technique improves the performance of both the C4.5 decision tree algorithm and the backpropagation neural network algorithm on a variety of difficult classification problems. Recently, Schapire

(1997) has shown how AdaBoost can be combined with error-correcting output coding to yield an excellent ensemble classification method that he calls AdaBoost.OC. The performance of the method is superior to the ECOC method (and to Bagging), but essentially the same as another (quite complex) algorithm, called AdaBoost.M2. Hence, the main advantage of AdaBoost.OC is implementation simplicity: It can work with any learning algorithm for solving 2-class problems.

Ricci and Aha (1997) applied a method that combines error-correcting output coding with feature selection. When learning each classifier, h_ℓ, they apply feature selection techniques to choose the best features for learning that classifier. They obtained improvements in 7 out of 10 tasks with this approach.

2.5 Injecting Randomness

The last general purpose method for generating ensembles of classifiers is to inject randomness into the learning algorithm. In the backpropagation algorithm for training neural networks, the initial weights of the network are set randomly. If the algorithm is applied to the same training examples but with different initial weights, the resulting classifier can be quite different (Kolen & Pollack, 1991).

While this is perhaps the most common way of generating ensembles of neural networks, manipulating the training set may be more effective. A study by Parmanto, Munro, and Doyle (1996) compared this technique to Bagging and to 10-fold cross-validated committees. They found that cross-validated committees worked best, Bagging second best, and multiple random initial weights third best on one synthetic data set and two medical diagnosis data sets.

For the C4.5 decision tree algorithm, it is also easy to inject randomness (Kwok & Carter, 1990; Dietterich, 2000). The key decision of C4.5 is to choose a feature to test at each internal node in the decision tree. At each internal node, C4.5 applies a criterion known as the information gain ratio to rank-order the various possible feature tests. It then chooses the top-ranked feature-value test. For discrete-valued features with V values, the decision tree splits the data into V subsets, depending on the value of the chosen feature. For real-valued features, the decision tree splits the data into 2 subsets, depending on whether the value of the chosen feature is above or below a chosen threshold. Dietterich (2000) implemented a variant of C4.5 that chooses randomly (with equal probability) among the top 20 best tests. Figure 3 compares the performance of a single run of C4.5 to ensembles of 200 classifiers over 33 different data sets. For each data set, a point is plotted. If that point lies below the diagonal line, then the ensemble has lower error rate than C4.5. We can see that nearly all of the points lie below the line. A statistical analysis shows that the randomized trees do statistically significantly better than a single decision tree on 14 of the data sets and statistically the same in the remaining 19 data sets.

Ali & Pazzani (1996) injected randomness into the FOIL algorithm for learning Prolog-style rules. FOIL works somewhat like C4.5 in that it ranks possible conditions to add to a rule using an information-gain criterion. Ali and Pazzani

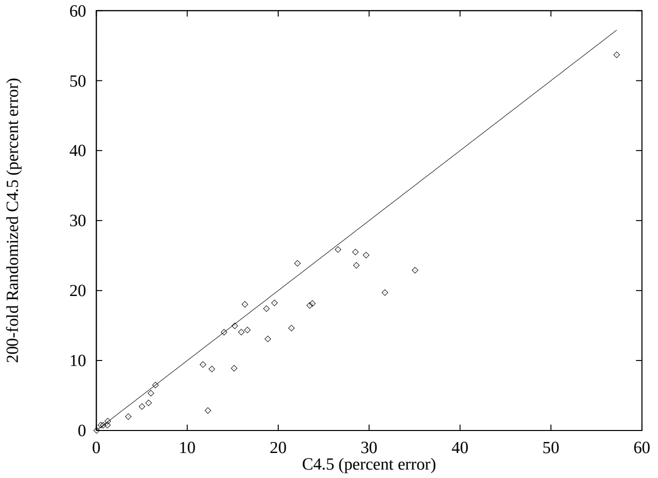

Fig. 3. Comparison of the error rate of C4.5 to an ensemble of 200 decision trees constructed by injecting randomness into C4.5 and then taking a uniform vote.

computed all candidate conditions that scored within 80% of the top-ranked candidate, and then applied a weighted random choice algorithm to choose among them. They compared ensembles of 11 classifiers to a single run of FOIL and found statistically significant improvements in 15 out of 29 tasks and statistically significant loss of performance in only one task. They obtained similar results using 11-fold cross-validation to construct the training sets.

Raviv and Intrator (1996) combine bootstrap sampling of the training data with injecting noise into the input features for the learning algorithm. To train each member of an ensemble of neural networks, they draw training examples with replacement from the original training data. The $\mathbf{x}$ values of each training example are perturbed by adding Gaussian noise to the input features. They report large improvements in a synthetic benchmark task and a medical diagnosis task.

Finally, note that Markov chain Monte Carlo methods for constructing Bayesian ensembles also work by injecting randomness into the learning process. However, instead of taking a uniform vote, as we did with the randomized decision trees, each hypothesis receives a vote proportional to its posterior probability.

3 Comparing Different Ensemble Methods

Several experimental studies have been performed to compare ensemble methods. The largest of these are the studies by Bauer and Kohavi (1999) and by Dietterich (2000). Table 1 summarizes the results of Dietterich's study. The table shows that AdaBoost often gives the best results. Bagging and randomized trees give similar performance, although randomization is able to do better in some cases than Bagging on very large data sets.

Table 1. All pairwise combinations of the four ensemble methods. Each cell contains the number of wins, losses, and ties between the algorithm in that row and the algorithm in that column.

	C4.5	AdaBoost C4.5	Bagged C4.5
Random C4.5	14 – 0 – 19	1 – 7 – 25	6 – 3 – 24
Bagged C4.5	11 – 0 – 22	1 – 8 – 24	
AdaBoost C4.5	17 – 0 – 16		

Most of the data sets in this study had little or no noise. When 20% artificial classification noise was added to the 9 domains where Bagging and AdaBoost gave different performance, the results shifted radically as shown in Table 2. Under these conditions, AdaBoost overfits the data badly while Bagging is shown to work very well in the presence of noise. Randomized trees did not do very well.

Table 2. All pairwise combinations of C4.5, AdaBoosted C4.5, Bagged C4.5, and Randomized C4.5 on 9 domains with 20% synthetic class label noise. Each cell contains the number of wins, losses, and ties between the algorithm in that row and the algorithm in that column.

	C4.5	AdaBoost C4.5	Bagged C4.5
Random C4.5	5 – 2 – 2	5 – 0 – 4	0 – 2 – 7
Bagged C4.5	7 – 0 – 2	6 – 0 – 3	
AdaBoost C4.5	3 – 6 – 0		

The key to understanding these results is to return again to the three shortcomings of existing learning algorithms: statistical support, computation, and representation. For the decision-tree algorithm C4.5, all three of these problems can arise. Decision trees essentially partition the input feature space into rectangular regions whose sides are perpendicular to the coordinate axes. Each rectangular region corresponds to one leaf node of the tree.

If the true function f can be represented by a small decision tree, then C4.5 will work well without any ensemble. If the true function can be correctly represented by a large decision tree, then C4.5 will need a very large training data set in order to find a good fit, and the statistical problem will arise.

The computational problem arises because finding the best (i.e., smallest) decision tree consistent with the training data is computationally intractable, so C4.5 makes a series of decisions greedily. If one of these decisions is made incorrectly, then the training data will be incorrectly partitioned, and all subsequent decisions are likely to be affected. Hence, C4.5 is highly unstable, and small changes in the training set can produce large changes in the resulting decision tree.

The representational problem arises because of the use of rectangular partitions of the input space. If the true decision boundaries are not orthogonal to the coordinate axes, then C4.5 requires a tree of infinite size to represent those boundaries correctly. Interestingly, a voted combination of small decision trees is equivalent to a much larger single tree, and hence, an ensemble method can construct a good approximation to a diagonal decision boundary using several small trees. Figure 4 shows an example of this. On the left side of the figure are plotted three decision boundaries constructed by three decision trees, each of which uses 5 internal nodes. On the right is the boundary that results from a simple majority vote of these trees. It is equivalent to a single tree with 13 internal nodes, and it is much more accurate than any one of the three individual trees.

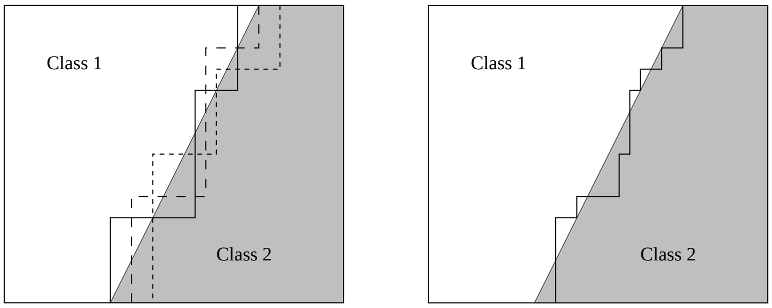

Fig. 4. The left figure shows the true diagonal decision boundary and three staircase approximations to it (of the kind that are created by decision tree algorithms). The right figure shows the voted decision boundary, which is a much better approximation to the diagonal boundary.

Now let us consider the three algorithms: AdaBoost, Bagging, and Randomized trees. Bagging and Randomization both construct each decision tree independently of the others. Bagging accomplishes this by manipulating the input data, and Randomization directly alters the choices of C4.5. These methods are acting somewhat like Bayesian voting; they are sampling from the space of all possible hypotheses with a bias toward hypotheses that give good accuracy on the training data. Consequently, their main effect will be to address the statistical problem and, to a lesser extent, the computational problem. But they do not directly attempt to overcome the representational problem.

In contrast, AdaBoost constructs each new decision tree to eliminate "residual" errors that have not been properly handled by the weighted vote of the previously-constructed trees. AdaBoost is directly trying to optimize the weighted vote. Hence, it is making a direct assault on the representational problem. Directly optimizing an ensemble can increase the risk of overfitting, because the space of ensembles is usually much larger than the hypothesis space of the original algorithm.

This explanation is consistent with the experimental results given above. In low-noise cases, AdaBoost gives good performance, because it is able to optimize the ensemble without overfitting. However, in high-noise cases, AdaBoost puts a large amount of weight on the mislabeled examples, and this leads it to overfit very badly. Bagging and Randomization do well in both the noisy and noise-free cases, because they are focusing on the statistical problem, and noise increases this statistical problem.

Finally, we can understand that in very large datasets, Randomization can be expected to do better than Bagging because bootstrap replicates of a large training set are very similar to the training set itself, and hence, the learned decision tree will not be very diverse. Randomization creates diversity under all conditions, but at the risk of generating low-quality decision trees.

Despite the plausibility of this explanation, there is still one important open question concerning AdaBoost. Given that AdaBoost aggressively attempts to maximize the margins on the training set, why doesn't it overfit more often? Part of the explanation may lie in the "stage-wise" nature of AdaBoost. In each iteration, it reweights the training examples, constructs a new hypothesis, and chooses a weight w_ℓ for that hypothesis. It never "backs up" and modifies the previous choices of hypotheses or weights that it has made to compensate for this new hypothesis.

To test this explanation, I conducted a series of simple experiments on synthetic data. Let the true classifier f be a simple decision rule that tests just one feature (feature 0) and assigns the example to class $+1$ if the feature is 1, and to class -1 if the feature is 0. Now construct training (and testing) examples by generating feature vectors of length 100 at random as follows. Generate feature 0 (the important feature) at random. Then generate each of the other features randomly to agree with feature 0 with probability 0.8 and to disagree otherwise. Assign labels to each training example according to the true function f, but with 10% random classification noise. This creates a difficult learning problem for simple decision rules of this kind (decision stumps), because all 100 features are correlated with the class. Still, a large ensemble should be able to do well on this problem by voting separate decision stumps for each feature.

I constructed a version of AdaBoost that works more aggressively than standard AdaBoost. After every new hypothesis h_ℓ is constructed and its weight assigned, my version performs a gradient descent search to minimize the negative exponential margin (equation 1). Hence, this algorithm reconsiders the weights of all of the learned hypotheses after each new hypothesis is added. Then it reweights the training examples to reflect the revised hypothesis weights.

Figure 5 shows the results when training on a training set of size 20. The plot confirms our explanation. The Aggressive AdaBoost initially has much higher error rates on the test set than Standard AdaBoost. It then gradually improves. Meanwhile, Standard AdaBoost initially obtains excellent performance on the test set, but then it overfits as more and more classifiers are added to the ensemble. In the limit, both ensembles should have the same representational properties, because they are both minimizing the same function (equation 1). But we can see that the exceptionally good performance of Standard AdaBoost on this problem is due to the stage-wise optimization process, which is slow to fit the data.

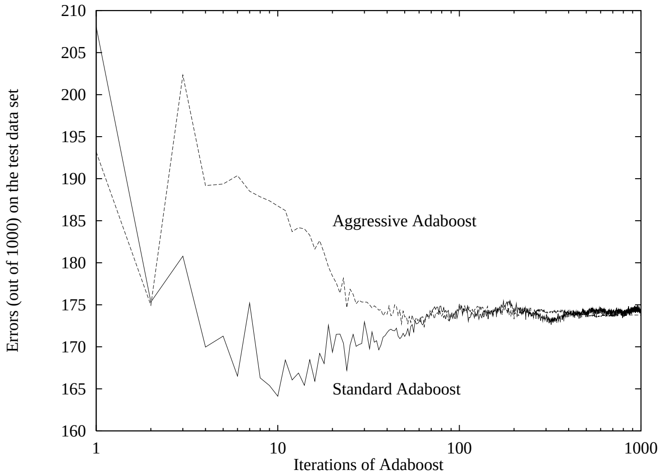

Fig. 5. Aggressive AdaBoost exhibits much worse performance than Standard AdaBoost on a challenging synthetic problem

4 Conclusions

Ensembles are well-established as a method for obtaining highly accurate classifiers by combining less accurate ones. This paper has provided a brief survey of methods for constructing ensembles and reviewed the three fundamental reasons why ensemble methods are able to out-perform any single classifier within the ensemble. The paper has also provided some experimental results to elucidate one of the reasons why AdaBoost performs so well.

One open question not discussed in this paper concerns the interaction between AdaBoost and the properties of the underlying learning algorithm. Most of the learning algorithms that have been combined with AdaBoost have been algorithms of a global character (i.e., algorithms that learn a relatively low-dimensional decision boundary). It would be interesting to see whether local algorithms (such as radial basis functions and nearest neighbor methods) can be profitably combined via AdaBoost to yield interesting new learning algorithms.

Bibliography

Ali, K. M., & Pazzani, M. J. (1996). Error reduction through learning multiple descriptions. *Machine Learning*, *24*(3), 173–202.

Bauer, E., & Kohavi, R. (1999). An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, *36*(1/2), 105–139.

Blum, A., & Rivest, R. L. (1988). Training a 3-node neural network is NP-Complete (Extended abstract). In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pp. 9–18 San Francisco, CA. Morgan Kaufmann.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*(2), 123–140.

Cherkauer, K. J. (1996). Human expert-level performance on a scientific image analysis task by a system using combined artificial neural networks. In Chan, P. (Ed.), *Working Notes of the AAAI Workshop on Integrating Multiple Learned Models*, pp. 15–21. Available from `http://www.cs.fit.edu/~imlm/`.

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*.

Dietterich, T. G., & Bakiri, G. (1995). Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, *2*, 263–286.

Freund, Y., & Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. Tech. rep., AT&T Bell Laboratories, Murray Hill, NJ.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Proc. 13th International Conference on Machine Learning*, pp. 148–146. Morgan Kaufmann.

Hansen, L., & Salamon, P. (1990). Neural network ensembles. *IEEE Trans. Pattern Analysis and Machine Intell.*, *12*, 993–1001.

Hornik, K., Stinchcombe, M., & White, H. (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, *3*, 551–560.

Hyafil, L., & Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-Complete. *Information Processing Letters*, *5*(1), 15–17.

Kolen, J. F., & Pollack, J. B. (1991). Back propagation is sensitive to initial conditions. In *Advances in Neural Information Processing Systems*, Vol. 3, pp. 860–867 San Francisco, CA. Morgan Kaufmann.

Kwok, S. W., & Carter, C. (1990). Multiple decision trees. In Schachter, R. D., Levitt, T. S., Kannal, L. N., & Lemmer, J. F. (Eds.), *Uncertainty in Artificial Intelligence 4*, pp. 327–335. Elsevier Science, Amsterdam.

Neal, R. (1993). Probabilistic inference using Markov chain Monte Carlo methods. Tech. rep. CRG-TR-93-1, Department of Computer Science, University of Toronto, Toronto, CA.

Parmanto, B., Munro, P. W., & Doyle, H. R. (1996). Improving committee diagnosis with resampling techniques. In Touretzky, D. S., Mozer, M. C., & Hesselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 8, pp. 882–888 Cambridge, MA. MIT Press.

Raviv, Y., & Intrator, N. (1996). Bootstrapping with noise: An effective regularization technique. *Connection Science*, *8*(3–4), 355–372.

Ricci, F., & Aha, D. W. (1997). Extending local learners with error-correcting output codes. Tech. rep., Naval Center for Applied Research in Artificial Intelligence, Washington, D.C.

Schapire, R. E. (1997). Using output codes to boost multiclass learning problems. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pp. 313–321 San Francisco, CA. Morgan Kaufmann.

Schapire, R. E., Freund, Y., Bartlett, P., & Lee, W. S. (1997). Boosting the margin: A new explanation for the effectiveness of voting methods. In Fisher, D. (Ed.), *Machine Learning: Proceedings of the Fourteenth International Conference*. Morgan Kaufmann.

Schapire, R. E., & Singer, Y. (1998). Improved boosting algorithms using confidence-rated predictions. In *Proc. 11th Annu. Conf. on Comput. Learning Theory*, pp. 80–91. ACM Press, New York, NY.

Tumer, K., & Ghosh, J. (1996). Error correlation and error reduction in ensemble classifiers. *Connection Science*, *8*(3–4), 385–404.