Committee Machines

Introduction

One of the most important research topics in machine learning is the problem of how to lower the generalization error of a learning algorithm, either by reducing the bias or the variance (or both).

A major complication of any attempt to reduce variance or bias (or both) is that the definitions of bias and variance of a classification rule are not as obvious as they are in regression.

In fact, there have been several conflicting suggestions for the bias-variance decomposition for classification problems.

Such a desire to control bias and variance, and, hence, generalization error, is related to the idea of instability of a prediction or classification method.

If a small perturbation of the learning set induces major changes in the resulting predictor or classifier, we say that the associated regression or classification method is unstable.

Unstable predictors or classifiers have high variance (due to overfitting) and low bias.

High bias occurs for predictors or classifiers that under-fit the data.

Decision trees and neural nets are, by this definition, unstable, whereas linear discriminant analysis is an example of a stable classifier with low variance and possibly high bias.

In this chapter, we show that the instability of a predictor or classifier (or, more generally, of any learning algorithm) is an important tool that can be used to improve the accuracy of that learning algorithm.

Novel approaches to the problem of predictor instability include bagging and boosting. Both of these approaches exploit the presence of instability in order to create a more accurate learning method (i.e., predictor or classifier).

By perturbing the learning set, these methods generate an ensemble of different base predictors or base classifiers, which are then combined into a single combined predictor or combined classifier, as appropriate.

The success of such combined learning methods - called ensemble learning or committee machines - often depends upon the degree of instability of the base predictors or classifiers.

Bagging and boosting can be distinguished from each other by the manner in which their respective perturbations are generated.

- The bagging process generates perturbations by random and independent drawings from the learning set, whereas

- the boosting process is deterministic and generates perturbations by successive reweightings of the learning set, where current weights depend upon the misclassification history of the process.

Bagging was designed specifically to reduce variance, whereas boosting appears to have more of a bias-reducing flavor.

Another example of a committee machine that will be described in this chapter is random forests.

Bagging

The word bagging is an acronym for the phrase bootstrap aggregating.

Bagging was the first procedure that successfully combined an ensemble of learning algorithms to improve performance over a single such algorithm.

Bagging is most successful if the predictor is unstable.

If the learning procedure is stable, the bagged predictor will not differ much from the single predictor and may even weaken its performance somewhat.

However, when the learning procedure is unstable, we tend to see a significant improvement for the bagged predictor over the original unstable procedure.

As before, we denote the learning set of n observations by

$$\mathcal{L} = \{(\mathbf{X}_i, Y_i),\ i = 1, 2, \ldots, n\}.$$

- $\{Y_i\}$ are continuous responses (a regression problem) or unordered class labels (a classification problem).

Bagging

- takes an ensemble of learning sets, $\{\mathcal{L}_k\}$, say, each containing n observations drawn from the same underlying distribution as those in $\mathcal{L}$, and

- combines the predictors from those learning sets in such a way that the resulting predictor improves upon that obtained from the single learning set $\mathcal{L}$.

The bagging procedure starts by drawing B bootstrap samples from $\mathcal{L}$.

Each bootstrap sample is obtained by repeated sampling with replacement from $\mathcal{L}$.

In other words, we place equal probabilities on the sample points (i.e., $p_i = 1/n$ on the ith observation $(\mathbf{X}_i, Y_i)$ in $\mathcal{L}$, $i = 1, 2, \ldots, n$) and then sample n times with replacement from this distribution.

We denote the bootstrap samples by

$$\mathcal{L}^{*b} = \{(\mathbf{X}_i^{*b}, Y_i^{*b}),\ i = 1, 2, \ldots, n\}, \quad b = 1, 2, \ldots, B.$$

Some of the original learning set will appear in $\mathcal{L}^{*b}$, some will appear several times, whereas others will not appear at all.

What we do next depends upon whether we are dealing with a classification or a regression problem.

Bagging Tree-Based Classifiers

In the classification case, $Y_i \in \{1, 2, \ldots, K\}$ is a class label attached to $\mathbf{X}_i$.

We grow a classification tree $\mathcal{T}^{*b}$ from the bth bootstrap sample $\mathcal{L}^{*b}$.

Note

To reduce bias, we grow this tree very large without pruning.

Suppose $(\mathbf{X}, Y)$ is independently drawn from the same joint distribution as the members in $\mathcal{L}$.

We drop $\mathbf{X}$ down each of the B bootstrap trees.

For each tree, when $\mathbf{X}$ falls into a terminal node associated with a particular class, we say that the tree votes for that class.

We then predict the class of $\mathbf{X}$ by the class that receives the most number of votes over all B trees.

We call this classification procedure the majority-vote rule.

In order to evaluate the bagging method, we need an independent test set of observations.

The fact that we are sampling (with replacement) from $\mathcal{L}$ means that about 37% of the observations in $\mathcal{L}$ will not be chosen for each bootstrap sample.

Question

Why 37%?

Let $\mathcal{L} - \mathcal{L}^{*b}$ denote those observations in $\mathcal{L}$ that are not selected for the bth bootstrap sample $\mathcal{L}^{*b}$.

If the observation $(\mathbf{X}, Y)$ is in $\mathcal{L} - \mathcal{L}^{*b}$ (which we write as $(\mathbf{X}, Y) \notin \mathcal{L}^{*b}$), then $(\mathbf{X}, Y)$ is called an out-of-bag (OOB) observation.

The collection of OOB observations (which we call an OOB sample) corresponding to the bootstrap sample $\mathcal{L}^{*b}$ will function as an independent test set.

The OOB approach to estimating generalization error is equivalent to using an independent test set of the same size.

The OOB approach is also able to use all the data, rather than partitioning the data into a separate (and smaller) learning set and a test set, and it does not require any additional computing as is needed for cross-validation.

Suppose $(\mathbf{X}_i, Y_i) \notin \mathcal{L}^{*b}$.

We drop $\mathbf{X}_i$ down the classification tree $\mathcal{T}^{*b}$ grown from $\mathcal{L}^{*b}$, and predict the class label for $\mathbf{X}_i$.

This acts as a classification vote on $\mathbf{X}_i$.

Suppose there are n_i ($\leq B$) trees for which $\mathbf{X}_i$ is a member of the corresponding OOB sample.

Drop $\mathbf{X}_i$ down each of those n_i trees and aggregate the votes for each of the K classes. Summarize the results by the K-vector,

$$\hat{\mathbf{p}}(\mathbf{x}_i) = \left(\hat{p}_1(\mathbf{x}_i), \hat{p}_2(\mathbf{x}_i), \ldots, \hat{p}_K(\mathbf{x}_i)\right)^T.$$

- $\hat{p}_k(\mathbf{x}_i)$ is the proportion of the n_i trees that votes for $\mathbf{X}_i = \mathbf{x}_i$ to be a member of the kth class Π_k.

The proportion $\hat{p}_k(\mathbf{x}_i)$ is an estimate of the true probability, $p(\Pi_k|\mathbf{x}_i) = \text{Prob}(\mathbf{X} \in \Pi_k|\mathbf{X} = \mathbf{x}_i)$, that the observed $\mathbf{x}_i$ belongs to Π_k.

The OOB classifier, $C_{bag}(\mathbf{x}_i)$, of $\mathbf{x}_i$ is then obtained by the majority-vote rule:

$$C_{bag}(\mathbf{x}_i) = \arg\max_k \{\hat{p}_k(\mathbf{x}_i)\}.$$

That is, it assigns $\mathbf{x}_i$ to that class that enjoys the largest number of votes.

We repeat this for every observation in $\mathcal{L}$.

The OOB misclassification rate,

$$PE_{bag} = n^{-1} \sum_{i=1}^{n} \mathrm{I}\left[C_{bag}(\mathbf{x}_i) \neq y_i\right],$$

- is the proportion of times that the predicted class, $C_{bag}(\mathbf{x}_i)$, is different from the true class, $Y = y_i$, for all observations in $\mathcal{L}$, and
- is an unbiased estimate of generalization error.

Bagging Regression-Tree Predictors

In the regression case, $Y_i \in \mathcal{R}$.

Bagging regression-tree estimates is a very similar procedure to that applied to classification trees, but instead of using a voting mechanism to determine the predicted class of an observation, we average the predicted response values obtained from the individual regression trees.

Specifically, from the bth bootstrap sample $\mathcal{L}^{*b}$, we grow a regression tree $\mathcal{T}^{*b}$ and obtain the predictor $\hat{\mu}^{*b}(\mathbf{X})$.

We drop $\mathbf{X}$ down each of the B regression trees and then average the predictions,

$$\hat{\mu}_{bag}(\mathbf{X}) = B^{-1} \sum_{b=1}^{B} \hat{\mu}^{*b}(\mathbf{X}),$$

to arrive at a bagged estimate of Y.

To evaluate the predictive abilities of a bagged regression estimate, we again use the OOB approach.

Let $(\mathbf{X}_i, Y_i) \in \mathcal{L}$.

We drop $\mathbf{X}_i$ down each of the n_i bootstrap trees whose OOB samples contain $(\mathbf{X}_i, Y_i)$.

The OOB regression estimate, $\hat{\mu}_{bag}(\mathbf{X}_i)$, is found by averaging the n_i bootstrap predicted values; that is,

$$\hat{\mu}_{bag}(\mathbf{X}_i) = n_i^{-1} \sum_{b \in \mathcal{N}_i} \hat{\mu}^{*b}(\mathbf{X}_i).$$

- $\mathcal{N}_i$ is the set of n_i bootstrap samples that do not contain $(\mathbf{X}_i, Y_i)$.

We repeat this procedure for all observations in $\mathcal{L}$.

We then estimate the generalization error of the bagged estimate by the OOB error rate,

$$PE_{bag} = n^{-1} \sum_{i=1}^{n} \left(Y_i - \hat{\mu}_{bag}(\mathbf{X}_i) \right)^2,$$

which is computed as the mean-squared-error between the bagged estimates and their true response values.

Boosting

The underlying notion of boosting is to enhance the accuracy of a weak binary classification learning algorithm.

This idea originated in a field known in machine learning as probably approximately correct (PAC) learning.

The first successful boosting algorithms were provided by Schapire (1990) and Freund (1995).

The name derives from the idea of creating a strong classifier by substantially improving or boosting the performance of a single weak classifier, where improvement is obtained by combining the classification votes from an ensemble of similar classifiers.

We define a weak (or base) classifier to be one that correctly classifies slightly more than 50% of the time (i.e., a little better than random guessing).

Boosting algorithms combine M base classifiers $C_1, C_2, \ldots, C_M$ in the following way.

For an observation $\mathbf{X} = \mathbf{x}$, the boosted classifier is given by:

$$C_{\boldsymbol{\alpha}}(\mathbf{x}) = \text{sign}\{f_{\boldsymbol{\alpha}}(\mathbf{x})\}.$$

- $$f_{\boldsymbol{\alpha}}(\mathbf{x}) = \sum_{j=1}^{M} \left(\frac{\alpha_j}{\sum_{j'} \alpha'_j} \right) C_j(\mathbf{x}),$$

- $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_M)T$ is an M-vector of constant coefficients.

Different versions of boosting have been applied to a wide variety of data sets with enormous success.

Consequently, this class of improvement algorithms has become an important research topic in both the statistics and machine learning communities.

The most well-known of these boosting algorithms is AdaBoost.

AdaBoost: Boosting by Reweighting

AdaBoost (an acronym for adaptive boosting) is an algorithm that is designed to improve performance in binary classification problems.

It is generally regarded as the first step toward a truly practical boosting procedure.

AdaBoost for Binary Classification

1. Input: $\mathcal{L} = \{(\mathbf{X}_i, Y_i), i = 1, 2, \ldots, n\}$, $Y_i \in \{-1, +1\}$, $i = 1, 2, \ldots, n$, $\mathcal{C} = \{C_1, C_2, \ldots, C_M\}$, $T =$ number of iterations.

2. Initialize the weight vector: Set $\mathbf{w}_1 = (w_{11}, \cdots, w_{n1})^\tau$, where $w_{i1} = 1/n$, $i = 1, 2, \ldots, n$.

AdaBoost for Binary Classification

3. For $t = 1, 2, \ldots, T$:

   - Select a weak classifier $C_{j_t}(\mathbf{x}) \in \{-1, +1\}$ from $\mathcal{C}$, $j_t \in \{1, 2, \ldots, M\}$, and train it on the learning set $\mathcal{L}$, where the ith observation $(\mathbf{X}_i, Y_i)$ has (normalized) weight w_{it}, $i = 1, 2, \ldots, n$.
   - Compute the weighted prediction error:

$$PE_t = PE(\mathbf{w}_t) = \mathrm{E}_w\{I_{[Y_i \neq C_{j_t}(\mathbf{X}_i)]}\} = \left(\frac{\mathbf{w}_t^\tau}{\mathbf{1}_n^\tau \mathbf{w}_t}\right) \mathbf{e}_t,$$

     where E_w indicates taking expectation with respect to the probability distribution of $\mathbf{w}_t = (w_{1t}, \cdots, w_{nt})^\tau$, and $\mathbf{e}_t$ is an n-vector with ith entry $[\mathbf{e}_t]_i = I_{[Y_i \neq C_{j_t}(\mathbf{X}_i)]}$.

   - Set $\beta_t = \frac{1}{2} \log\left(\frac{1-PE_t}{PE_t}\right)$.
   - Update weights:

$$w_{i,t+1} = \frac{w_{it}}{W_t} \exp\{2\beta_t I_{[Y_i \neq C_{j_t}(\mathbf{X}_i)]}\}, \quad i = 1, 2, \ldots, n,$$

     where W_t is a normalizing constant needed to ensure that the vector $\mathbf{w}_{t+1} = (w_{1,t+1}, \cdots, w_{n,t+1})^\tau$ represents a true weight distribution over $\mathcal{L}$; that is, $\mathbf{1}_n^\tau \mathbf{w}_{t+1} = 1$.

AdaBoost for Binary Classification

4. Output: $\text{sign}\{f(\mathbf{x})\}$, where $f(\mathbf{x}) = \sum_{t=1}^{T} \beta_t C_{j_t}(\mathbf{x}) = \sum_{j=1}^{M} \alpha_j C_j(\mathbf{x})$, and $\alpha_j = \sum_{t=1}^{T} \beta_t I_{[j_t=j]}$.

It is also known as Discrete AdaBoost (because the goal is to predict class labels).

A simple generalization of AdaBoost to more than two classes is called AdaBoost.M1.

AdaBoost was originally devised with the specific intention of driving the prediction error from the learning set (i.e., the learning set error) quickly to zero.

In the AdaBoost algorithm for binary classification, we start with a learning set $\mathcal{L} = \{(\mathbf{x}_i, y_i)\}$, where $\mathbf{x}_i$ is an r-vector of inputs and $y_i \in \{-1, +1\}$ is a class label.

AdaBoost weights the observations in $\mathcal{L}$ by a weight vector,

$$\mathbf{w} = (w_1, w_2, \ldots, w_n)^T,$$

and these weights are recalculated at each iteration.

Initially, we use equal weights for each observation in $\mathcal{L}$.

At each iteration, the algorithm selects a weak classifier from a very large, but finite, set $\mathcal{C}$ of all possible weak classifiers.

The finiteness assumption always holds for classification problems where each classifier in $\mathcal{C}$ has a finite set of possible outputs.

For example, in binary classification, at most 2^n distinct labelings can be applied to the learning set.

Because $\mathcal{C}$ is finite, it is entirely possible that, in constructing the ensemble, certain of the weak classifiers in $\mathcal{C}$ will be selected more than once (i.e., the smaller the set, the more likely that repetitions will occur).

At the t-th iteration of AdaBoost, we modify the weighting system so that observations misclassified in the previous iteration will be more heavily weighted in the current iteration.

In this way, AdaBoost tries hard to classify correctly any previously misclassified observations.

After T iterations, we have a sequence, $C_{j_1}(\mathbf{x}), C_{j_2}(\mathbf{x}), \ldots, C_{j_T}(\mathbf{x})$, of weak classifiers, where $j_t \in \{1, 2, \ldots, M\}$, $t = 1, 2, \ldots, T$.

If the weak classifier C_j is selected multiple times in the process of the algorithm, then the coefficient for that component in the combined classifier is the sum of those coefficients obtained at all iterations when C_j was chosen.

If the classifiers are small decision trees (as they often are when boosting is applied), then the jth weak classifier can be parameterized as $C_j(\mathbf{x}; \mathbf{a}_j)$.

- The parameter vector $\mathbf{a}_j$ contains information on the splitting variables, split points, and the mean at each terminal node of the jth tree.

The value of the boosted classifier $C(\mathbf{x})$ depends upon the sign of the linear combination, $f(\mathbf{x}) = \sum_{j=1}^{M} \alpha_j C_j(\mathbf{x})$, of the weak classifiers, where α_j is the coefficient for C_j.

In other words,

$$C(\mathbf{x}) = \begin{cases} +1, & \text{if } f(\mathbf{x}) > 0, \\ \\ -1, & \text{otherwise.} \end{cases}$$

AdaBoost does not restrict the sum of the coefficients $\{\alpha_j\}$, which may grow to be very large; all AdaBoost assumes is that f is in the linear span of the class $\mathcal{C}$ of weak classifiers.

If we restrict the coefficients to be nonnegative with a fixed sum λ, say, this produces a regularized version of AdaBoost, where λ acts as a smoothing parameter; in this case, $f \in \text{conv}(\mathcal{C})$, the convex hull of $\mathcal{C}$.

Convergence Issues and Overfitting

Empirical experiments have demonstrated that AdaBoost tends to be quite resistant to overfitting: the test set error (an estimate of generalization error) almost always continues to decline (and then levels off) as we increase the number of classifiers involved even after the learning-set error has been driven to zero!

Recall that in a typical classification scenario, test-set error decreases for a little while and then begins to increase as the classifier becomes more and more complex.

The discovery that AdaBoost is resistant to overfitting led to it being called the most accurate, general-purpose, classification algorithm available.

Since AdaBoost was introduced, hundreds of articles have been published attempting to penetrate the mysterious secret of why it appears to be resistant to overfitting.

Many explanations have been attempted, but the question still remains open.

This mystery has been described as the most important unsolved problem in machine learning.

This does not mean, however, that AdaBoost never overfits.

Indeed, examples of AdaBoost have been constructed in which the test-set error increases (i.e., AdaBoost does overfit) as the number of iterations increases.

Breiman (2004) suggests that the AdaBoost process may actually consist of two stages.

In the first stage (which may consist of several thousand iterations), the test-set error approaches close to the optimal Bayes error, mimicking its population (i.e., infinite sample size n) behavior.

In its population version, Breiman showed that AdaBoost is Bayes consistent; that is, its risk converges in probability to the Bayes risk.

If, for whatever reason, convergence fails, its test-set error then starts increasing.

This second-stage behavior is not yet understood.

Further study of the convergence problem has shown that, for finite sample sizes, AdaBoost can be Bayes consistent only if it is regularized.

One possible type of regularization for AdaBoost is that of stopping the algorithm very early (e.g., after 10 or 100 iterations), rather than letting it run forever; essentially, the argument is that overfitting will occur as soon as the classifier becomes too complicated and that continuing to run the algorithm will only produce larger misclassification rates.

Jiang (2004) and Bickel and Ritov (2004) show that for any finite n, there is a stopping time t_n such that if the algorithm is stopped at t_n iterations, then AdaBoost will be Bayes consistent.

The question then becomes, if the strategy is to stop AdaBoost early, how does one determine the best time to stop (i.e., an optimal t_n)?

One suggested method is to use a data-based procedure, such as cross-validation.

There is empirical evidence (Mease and Wyner, 2007) that shows that early stopping may not be the panacea needed to prevent overfitting.

Indeed, the evidence suggests that overfitting tends to occur very early in the life of the algorithm and that running the algorithm for a much larger number of iterations actually reduces the amount of overfitting (to a level close to that of the Bayes risk) rather than increases it.

Classification Margins

One interesting argument put forward to explain why boosting works so well in classification problems involves the concept of a margin.

Let $\mathcal{C}$ be the set of all potential weak classifiers. For example, weak classifiers could be chosen from all those decision trees that have a specified number of terminal nodes.

Consider a boosted classifier f, where the weights, $\{\alpha_t\}$, are each nonnegative and sum to one. Then, $f \in \text{conv}(\mathcal{C})$ is a weighted average of weak classifiers from $\mathcal{C}$.

If the weak classifiers are defined by a voting scheme, then the prediction is that label y that receives the highest vote from the weak classifiers.

Let $g(\mathbf{x}, y)$ denote a classifier that predicts the label y for an observation $\mathbf{x}$. Then, g predicts y iff $g(\mathbf{x}, y) > \max_{y' \neq y} g(\mathbf{x}, y')$.

The classification margin of the labeled observation $(\mathbf{x}, y)$ is defined as

$$m(\mathbf{x}, y) = g(\mathbf{x}, y) - \max_{y' \neq y} g(\mathbf{x}, y').$$

Thus, if y is the correct label for $\mathbf{x}$, then g misclassifies $\mathbf{x}$ iff $m(\mathbf{x}, y) < 0$.

Let

$$g(\mathbf{x}, y) = \sum_t \mathsf{I}[C_{j_t}(\mathbf{x}) = y]$$

be the total number of votes for y obtained from all the weak classifiers, then the classification margin is the amount by which the total vote for the correct class y exceeds the highest total vote for any incorrect class.

That is,

$$m(\mathbf{x}, y) = \sum_t \mathsf{I}[C_{j_t}(\mathbf{x}) = y] - \max_{y' \neq y} \left\{ \sum_t \mathsf{I}[C_{j_t}(\mathbf{x}) = y'] \right\}.$$

Thus, an observation $(\mathbf{x}, y)$ is misclassified by the voting scheme iff its margin is not positive.

Because the observation $(\mathbf{x}, y)$ is misclassified by the boosted classifier f only if $yf(\mathbf{x}) \leq 0$, we can think of the margin of $(\mathbf{x}, y)$ with respect to f as $m(\mathbf{x}, y) = yf(\mathbf{x})$.

The margin of the boosted classifier f is the minimum margin over all n observations in $\mathcal{L}$.

In binary classification problems (with labels -1 and $+1$), the margin can be viewed in the following terms: the bigger the margin, the more confidence we have that the observation has been correctly classified.

If the margin is large but negative, this tells us we are very confident that the observation has been misclassified.

Small margins indicate doubtful reliance on classifications.

To assess the performance of a boosted classifier, Schapire et al. (1998) derive a probabilistic upper-bound on its generalization error.

The upper bound turns out to depend upon the sum of the empirical margin distribution,

$$\frac{1}{n}\sum_{i=1}^{n} \mathsf{I}[y_i f(\mathbf{x}_i) < \delta]$$

and the **VC-dimension** of the class of boosted classifiers (Vapnik and Chervonenkis, 1971), but is independent of the number of weak classifiers being combined.

From the upper-bound, they argue that the bigger the margins (over a learning set), the lower the generalization error of the classifier.

They then conjecture that AdaBoost is successful because it produces large margins for the learning set.

Unfortunately, the probabilistic upper-bound only tells part of the story.

Schapire et al. (1998) realized that their bound is much too loose to be useful for a majority-vote classifier.

Although not asymptotic by construction (the bound is not dependent upon the size of the learning set), empirical results show that for the bound to be of any practical use, the size of the learning set would have to be huge (of the order tens of thousands).

Constructing tighter upper bounds on the generalization error remains an open problem (see, e.g., Koltchinskii and Panchenko, 2002).

Breiman (1999) demonstrated also that high margins alone cannot explain the success of AdaBoost.

Using a game-theoretic argument, he constructed a boosted classifier that not only had large margins (higher indeed than obtained by AdaBoost on each of a number of data sets) but also had higher generalization error in each case.

AdaBoost and Maximal Margins

So far, we have adopted a nonoptimal point of view (or strategy) for AdaBoost, where it is only necessary to provide a sufficiently good classifier at each iteration (not necessarily the best one) from the set $\mathcal{L}$ of weak classifiers.

Examples of nonoptimal AdaBoost include decision trees and neural networks.

We can also identify an optimal AdaBoost strategy, where the best weak classifier is selected at each iteration from $\mathcal{C}$.

This strategy has the effect of introducing an optimality step into the AdaBoost algorithm, so that, in principle, specific weak classifiers can be chosen again and again from $\mathcal{C}$.

From the above discussion, we know that AdaBoost induces large margins.

In fact, if ρ is the maximum achievable margin, then AdaBoost produces a margin $m(\mathbf{x}, y)$ that is bounded below and above by

$$\frac{\rho}{2} \leq -\frac{\log(1-\rho^2)}{\log\left(\frac{1+\rho}{1-\rho}\right)} \leq m(\mathbf{x}, y) \leq \rho.$$

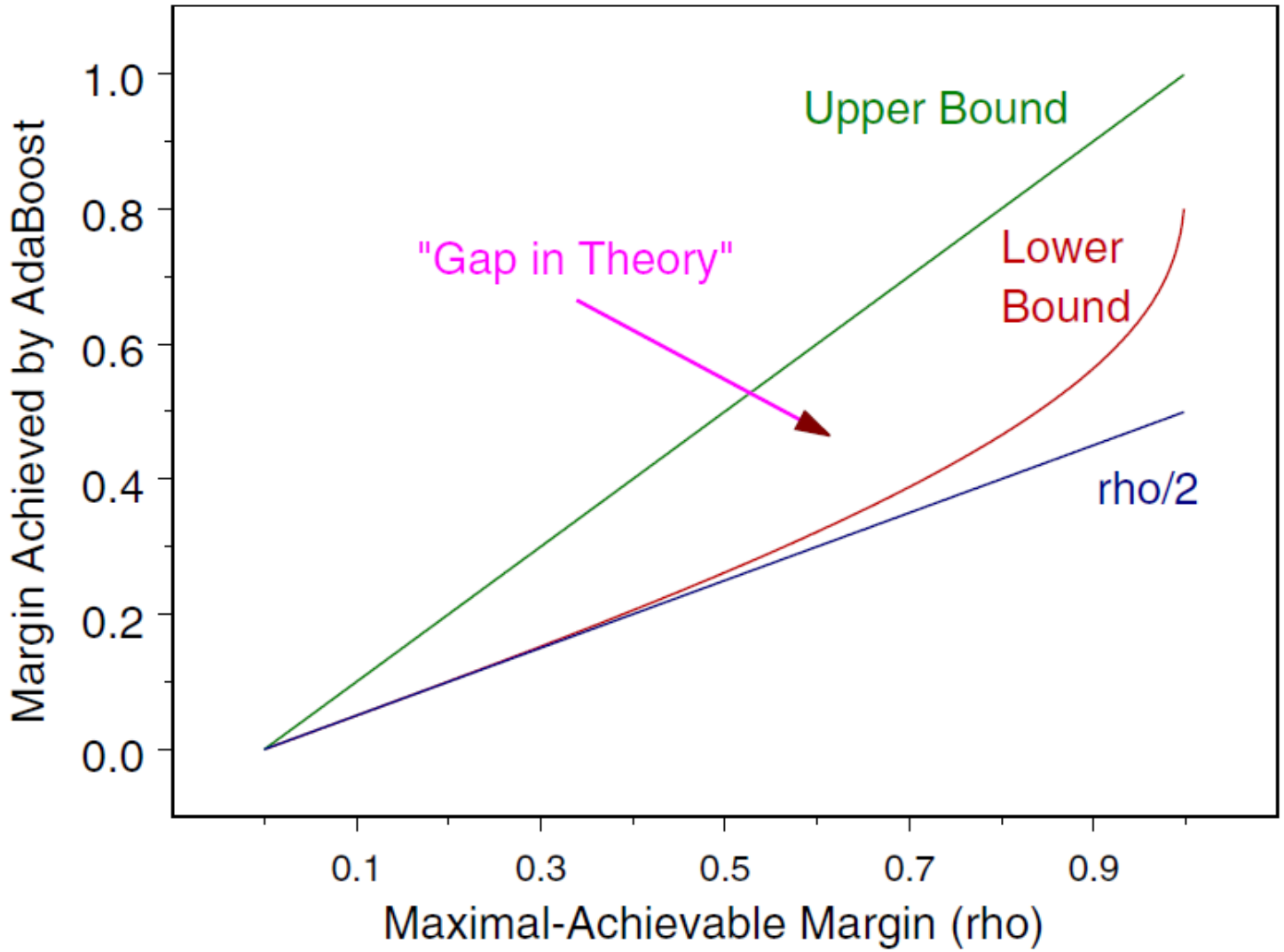

FIGURE: The margin achieved by AdaBoost as a function of the maximal-achievable margin ρ.

The vertical distance between the upper and lower bounds has been referred to as the gap in theory (Rätsch and Warmuth, 2005).

The lower bound (i.e., the red curve) has been shown to be exactly tight, however, for the nonoptimal AdaBoost strategy (Rudin, Schapire, and Daubechies 2007).

Recall that the closer a classifier gets to the maximum margin, the more confidence we have in that classifier.

Even though AdaBoost was not specially designed to attain the maximum margin (margin theory came just after the introduction of AdaBoost), there is widespread belief that (as a by-product of its remarkable practical properties) AdaBoost also maximizes the margin.

Rätsch and Warmuth (2005) noted, however, that empirical evidence showed that might not always be the case.

The conjecture that AdaBoost does not always attain the maximum possible margin turns out to be true (Rudin, Daubechies, and Schapire, 2004).

Because the margin does not increase monotonically as the iterations proceed, standard methods for examining convergence properties of AdaBoost margins are not applicable.

Instead, following the remarkable work by Rudin et al., we look at the limiting performance of the sequence of weight vectors $\{\mathbf{w}_t\}$ that defines AdaBoost.

The Details of Derivation

See the textbook.

Using specific low-dimensional examples that are simple enough for the details to be worked out completely, Rudin et al. showed that AdaBoost does not always converge to a maximum-margin solution.

Instead, AdaBoost may converge to a solution whose margin is significantly below the maximum value.

It may do this for nonoptimal AdaBoost even if optimal AdaBoost converges to a maximum-margin solution.

AdaBoost can also operate in chaotic mode, where the algorithm moves into and out of cyclic behavior, possibly due to a sensitivity to initial conditions.

A Statistical Interpretation of AdaBoost

Can we give a statistical interpretation of the AdaBoost algorithm?

Friedman, Hastie, and Tibshirani (2000) showed that AdaBoost is equivalent to running a coordinate-descent algorithm to fit an additive, logistic-discrimination model to the learning set.

That article (and the discussants) had much to say about the philosophical, statistical, and computational issues of boosting.

The Outline of Some Development Work

See the textbook.

Basic Idea

The objective is to minimize the risk function

$$R(f) = \mathrm{E}_{\mathbf{X}}\left[E_Y\{L(Y, f(\mathbf{x}))|\mathbf{x}\}\right].$$

- The loss function $L\big(y, f(\mathbf{x})\big)$ is defined as

$$L\big(y, f(\mathbf{x})\big) = e^{-yf(\mathbf{x})}, \quad y \in \{-1, +1\}.$$

Reference

 Friedman, J., Hastie, T., and Tibshirani, R. (2000).

Additive logistic regression: a statistical view of boosting (with discussion).

Annals of Statistics, **28**, 337–407.

Some Questions About AdaBoost

Since the Friedman, Hastie, and Tibshirani (2000) paper on the statistical view of boosting appeared, much has been written on the subject, with extensions in many directions.

Many studies using real and simulated data have appeared that try to examine the statistical issues discussed in the Friedman et al. paper.

Simulated data have been particularly important in understanding the behavior of AdaBoost because then the joint distribution of $(\mathbf{X}, Y)$ is completely known.

However, several major questions about AdaBoost have been left unanswered by Friedman et al. and other researchers.

Why Does AdaBoost Work?

This is probably the most important question of interest to users of AdaBoost.

As we have seen, AdaBoost can be viewed as algorithmically similar to an approach consisting of an amalgam of three separate components:

(1) an additive logistic regression model (e.g., a linear combination of classification trees),

(2) an exponential loss criterion,

(3) a coordinate-wise fitting procedure.

This interpretation of AdaBoost has since encouraged researchers to develop other boosting algorithms by changing either the type of smooth, convex loss function used in the basic algorithm, or the numerical fitting procedure, or both.

But this still begs the question of why AdaBoost works so well.

AdaBoost yields very small misclassification rates (compared with other competing classifiers) over a wide variety of data sets and is (in most cases) highly resistant to overfitting.

As we have already noted, not all data sets are immune to overfitting; there are a number of specially constructed examples that show that AdaBoost can indeed overfit.

What Friedman et al. gave us is a useful description of a way of thinking statistically about AdaBoost.

But they did not address the main issue of why AdaBoost is so resistant to overfitting, whether for simulated or real data.

Since the appearance of that article, many suggestions have been made as to why AdaBoost is successful in classification situations.

Some researchers have pointed to the stagewise fitting machine, or to the $0-1$ loss function, or to the notion of margin, but none of these explanations are really convincing.

It is still an open question as to why AdaBoost works as well as it does.

Specifically, we would like to know under what conditions we can expect AdaBoost not to overfit, and under what conditions we should expect AdaBoost to overfit.

How Well Can AdaBoost Estimate Conditional Class Probabilities?

On a related problem to classification, we may wish to estimate the conditional class probability function,

$$p(\mathbf{x}) = P\{Y = 1|\mathbf{x}\}.$$

If we can estimate $p(\mathbf{x})$ well across the entire range of $\mathbf{x}$, we would then be able to obtain a solution to the classification problem by choosing an appropriate quantile q of this function to be the class boundary.

That is, find q such that all cases in the region $p(\mathbf{x}) > q$ are classified as positive $(+1)$.

Building upon the connection between AdaBoost and logistic regression, Friedman, Hastie, and Tibshirani (2000, Algorithm 3) introduced the LogitBoost algorithm to estimate $p(\mathbf{x})$ directly using the link function; that is,

$$\hat{p}_j(\mathbf{x}) = \frac{1}{1 + e^{-2f_j(\mathbf{x})}}.$$

- f_j is the classifier evaluated at the jth iteration.

LogitBoost is a modified version of the AdaBoost algorithm that uses stage-wise minimization of the binomial log-likelihood loss function (in place of exponential loss).

Thus, the current estimate of the boosted classifier $f_j(\mathbf{x})$ is transformed via the above link to produce a current estimate of $p(\mathbf{x})$.

In simulations, Mease, Wyner, and Buja (2007) show that boosting classification trees (and LogitBoost, in particular) is not well-suited to estimating $p(\mathbf{x})$, except for estimating the median of that probability function (and other special cases).

Indeed, there is empirical evidence that boosting can severely overfit the estimate of $p(\mathbf{x})$ - even when the AdaBoost classification rule performs well with no appearance of overfitting.

These results throw doubt on the popularly made claim that the success of boosting is due to its close relationship to logistic regression.

Do Stumps Make the Best Base Classifiers for AdaBoost?

It has been argued (Friedman, Hastie, and Tibshirani, 2000, pp. 360–361; Hastie, Tibshirani, and Friedman, 2001, Section 10.11) that larger trees introduce higher-level interaction effects among the input variables $\mathbf{X}$.

Thus, stumps represent main effects (X_j), the second level of 4 nodes represents first-order interactions ($X_j X_k$), the third level of 8 nodes represents second-order interactions ($X_j X_k X_l$), and so on.

Such higher-order interactions, it is argued, then lead to overfitting.

A corollary to this argument is that if we believe that the optimal Bayes risk can be closely approximated by an additive function of elements of $\mathbf{X}$, then only stumps provide an additive model.

Although larger trees are not ruled out as base classifiers, stumps, in this context, are said to provide an ideal match and, according to this argument, are to be preferred to larger trees.

Yet, simulations have shown (Mease and Wyner, 2007) that stumps do not necessarily provide the best base classifiers for AdaBoost even if the optimal Bayes risk is additive, and that larger trees can actually be more effective.

The solubility example shows that using stumps as base classifiers gives a relatively poor performance when compared with the results from using larger trees with 4, 8, or 16 terminal nodes.

Noisy Class Labels

In classification problems, label noise exists when the learning set contains observations with incorrect class labels.

Dietterich (2000) showed that noisy labels degrade the accuracy of AdaBoost when applied to classification trees, whereas bagging appears to be quite robust against label noise.

We can create label noise by randomly selecting (without replacement) a fraction (e.g., 5%) of the observations from a data set and then changing the class label of each chosen observation using a random assignment from the set of incorrect class labels.

We knew that, on average, about 37% of the observations from the learning set are omitted from each bootstrap sample.

Thus, it is likely that a large proportion of the mislabeled observations will not appear in a bootstrap sample.

The omission of misclassified observations (which should behave like regression outliers) from the bootstrap sample will increase stability and, hence, improve the performance of bagging.

On the other hand, after a few iterations, AdaBoost will keep assigning large weights to the fraction of mislabeled observations because it will have difficulty classifying the corrupted observations, and this may, in turn, degrade performance and lead to overfitting.

When noisy class labels are present, there is empirical evidence that we can improve the classifier's performance if we apply bagging following boosting (a BB algorithm).

Specifically, we generate $B = \rho n$ bootstrap samples from the learning set ($0 < \rho < 1$), compute a boosted classifier from each bootstrap sample using M iterations, combine the B different boosted classifiers into an ensemble, and then average over the ensemble.

Studies show, using real data, that the BB classifier averages out (or smoothes) the overfitting in AdaBoost and, hence, decreases test error.

Random Forests

We have seen how perturbing the learning set $\mathcal{L}$ in various ways can be used to generate an ensemble (or forest) of tree-structured classifiers.

- A classification tree T_k is grown for each perturbation $\mathcal{L}_k$ of the learning set, $k = 1, 2, \ldots, K$;
- a test set observation $\mathbf{x}$ is dropped down each tree; and
- the classifier predicts the class of that observation by that class that enjoys the largest number of total votes over all of the trees.

In bagging, randomization is used only in selecting the data set on which to grow each tree.

An extension of this idea is **random forests**, where randomization adds another layer onto bagging and is a crucial part of constructing each tree.

Suggestions on how to introduce randomization into tree construction include random split selection in which each node is split by randomly choosing one of the t best splits at that node and random input selection in which the split at each node is decided by a random choice of subset of the r input features.

Randomizing Tree Construction

In random forests, we start in the same way that bagging starts, with B bootstrap samples drawn from the learning set $\mathcal{L}$, but the difference is how the trees are grown from those samples.

The idea is to introduce a randomization component into tree construction so that, for the tree T^{*b}, each node is split in a random manner.

Possible options for developing a randomized splitting strategy at each node include using some form of random input selection and linear combinations of inputs.

Recall that bagging applied to a tree-structured classifier reduces variance (due to aggregation) and bias (if the trees are fully grown).

A random forest reduces the correlation between the tree-structured classifiers that enter into the averaging step.

Algorithm

1. Input: $\mathcal{L} = \{(\mathbf{x}_i, y_i), i = 1, 2, \ldots, n\}$, $y_i \in \{1, 2, \ldots, K\}$, m = number of variables to be chosen at each node ($m << r$), B = number of bootstrap samples.
2. For $b = 1, 2, \ldots, B$:
   - Draw a bootstrap sample $\mathcal{L}^{*b}$ from the learning set $\mathcal{L}$.
   - From $\mathcal{L}^{*b}$, grow a tree classifier T^{*b} using random input selection: at each node, randomly select a subset m of the r input variables, and, using only the m selected variables, determine the best split at that node (using entropy or the Gini index). To reduce bias, grow the tree to a maximum depth with no pruning.
   - The tree T^{*b} generates an associated random vector $\boldsymbol{\theta}_b$, which is independent of the previous $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{b-1}$, and whose form and dimensionality are determined by context.
   - Using $\boldsymbol{\theta}_b$ and an input vector $\mathbf{x}$, define a classifier $h(\mathbf{x}, \boldsymbol{\theta}_b)$ having a single vote for the class of $\mathbf{x}$.

Algorithm

3. The B randomized tree-structured classifiers $\{h(\mathbf{x}, \boldsymbol{\theta}_b)\}$ are collectively called a *random forest*.

4. The observation $\mathbf{x}$ is assigned to the majority vote-getting class as determined by the random forest.

Note

In other words, in building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available predictors.

This may sound crazy, but it has a clever rationale.

Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors.

Then in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split.

Consequently, all of the bagged trees will look quite similar to each other.

Hence the predictions from the bagged trees will be highly correlated.

Unfortunately, averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities.

In particular, this means that bagging will not lead to a substantial reduction in variance over a single tree in this setting.

Random forests overcome this problem by forcing each split to consider only a subset of the predictors.

Therefore, on average $(r - m)/r$ of the splits will not even consider the strong predictor, and so other predictors will have more of a chance.

We can think of this process as decorrelating the trees, thereby making the average of the resulting trees less variable and hence more reliable.

Tuning Parameters

There are only two tuning parameters for a random forest:

- the number m of variables randomly chosen as a subset at each node
- the number B of bootstrap samples.

The procedure is relatively insensitive to a wide range of values of m and B.

The choice of m

A good starting point is to take m as $\sqrt{r}$; if that is not sufficient, it is recommended to rerun the program with $m = 2\sqrt{r}$ and $m = 0.5\sqrt{r}$ as a way of monitoring the procedure.

We have often found that values smaller than $\sqrt{r}$ yield smaller misclassification rates.

For regression, the default value for m is $r/3$ and the minimum node size is five.

The choice of B

The number B of bootstrap samples can be taken to be at least 1,000.

If r is very large, then B can be around 5,000.

Generalization Error

Consider an ensemble (or committee) of B randomized tree-structured classifiers,

$$h(\mathbf{x}_1, \boldsymbol{\theta}_1), h(\mathbf{x}_2, \boldsymbol{\theta}_2), \ldots, h(\mathbf{x}_B, \boldsymbol{\theta}_B).$$

Define the generalization error for a random forest having B trees as

$$PE_B = P_{\mathbf{X},Y}\{m_B(\mathbf{X}, Y) < 0\}.$$

- $$m_B(\mathbf{X}, Y) = \frac{1}{B}\sum_{b=1}^{B} \mathrm{I}[h(\mathbf{X}, \boldsymbol{\theta}_b) = Y] - \max_{k \neq y}\left\{\frac{1}{B}\sum_{b=1}^{B} \mathrm{I}[h(\mathbf{X}, \boldsymbol{\theta}_b) = k]\right\}$$

  is the classification margin for the ensemble, and the probability is computed over the $(\mathbf{X}, Y)$-space.

Note

If $m_B(\mathbf{X}, Y) > 0$, then the committee votes for the correct classification, whereas otherwise it does not.

Breiman (2001b) showed, using the strong law of large numbers, that, as the number of trees increases ($B \to \infty$), PE_B converges almost surely ($\{\boldsymbol{\theta}_b\}$) to the generalization error,

$$PE = P_{\mathbf{X},Y}(m(\mathbf{X}, Y) < 0\}.$$

- $$m(\mathbf{X}, Y) = P_{\boldsymbol{\Theta}}\{h(\mathbf{X}, \boldsymbol{\Theta}) = Y\} - \max_{k \neq Y} P_{\boldsymbol{\Theta}}\{h(\mathbf{X}, \boldsymbol{\Theta}) = k\}$$

  is defined as the margin function for a random forest.

The margin, $m(\mathbf{X}, Y)$, is the amount by which the average number of votes at $(\mathbf{X}, Y)$ for the correct class exceeds the average vote for any other class.

This limiting result is important: it shows that as we increase the number of trees in the forest, generalization error for a random forest converges to a limit.

In other words, random forests cannot overfit, even if we have an infinite number of trees in the forest.

An Upper Bound on Generalization Error

See the textbook.

Assessing Variable Importance

If the objective is to classify new observations, it is useful to know which variables really control the classification process.

In a regression situation, we need to know which subset of variables best explains the response values.

We recognize, of course, that identifying which variables are important can be complicated by the existence of interactions between variables.

Random forests can be used to evaluate the variables in a data set and provide a graphical display to assess the importance of each variable.

Computations are carried out one tree at a time. As before, let T^{*b} be the tree classifier constructed from the bootstrap sample $\mathcal{L}^{*b}$.

First, drop the OOB observations corresponding to $\mathcal{L}^{*b}$ down the tree T^{*b}, record the resulting classifications, and compute the OOB error rate, $PE_b(OOB)$.

Next, randomly permute the OOB values on the jth variable X_j while leaving the data on all other variables unchanged.

If X_j is important, permuting its observed values will reduce our ability to classify successfully each of the OOB observations.

Then, we drop the altered OOB observations down the tree T^{*b}, record the resulting classifications, and compute the OOB error rate, $PE_b(OOB_j)$, which should be larger than the error rate of the unaltered data.

A raw T^{*b}-score for X_j can be computed by the difference between those two OOB error rates,

$$raw_b(j) = PE_b(OOB_j) - PE_b(OOB), \quad b = 1, 2, \ldots, B.$$

Finally, average the raw scores over all the B trees in the forest,

$$imp(j) = \frac{1}{B}\sum_{b=1}^{B} raw_b(j),$$

to obtain an overall measure of the importance of X_j.

Call this measure the raw permutation accuracy importance score for the jth variable.

Assuming the B raw scores are independent from tree to tree, we can compute a straightforward estimate of the standard error.

Empirical studies using many different types of data sets show that a good case can be made for independence: indeed, scores between the trees appear to have low correlations.

If this estimate of standard error is acceptable, we compute a z-score by dividing the raw score by the estimated standard error and then compute an appropriate Gaussian-based significance level for that z-score.

Call this z-score the mean decrease in accuracy for the jth variable.

A second measure of variable importance derives from the fact that the Gini impurity index for a given parent node is larger than the value of that measure for its two daughter nodes.

By averaging the (Gini) decreases in node impurities over all trees in the forest, we obtain a measure we call the Gini importance index.

Proximities for Classical Scaling

One of the most useful notions incorporated into random forests is that of computing proximities between pairs of observations.

Proximities can be used for imputing missing values and identifying multivariate outliers, if they are present in the data.

Suppose we construct a random forest of trees $\{T^{*b}\}$ from a learning set $\mathcal{L}$.

Recall that each tree $\{T^{*b}\}$ is unpruned and, hence, each terminal node in $\{T^{*b}\}$ will contain only a few observations.

If we drop all cases in $\mathcal{L}$ (including the OOB observations) down all the trees in the forest, how often do pairs of observations occupy the same terminal node?

The answer to this question gives us a measure of closeness (or proximity) of those pairs of observations to each other.

We, therefore, wish to define a similarity measure, prox$(\mathbf{x}_i, \mathbf{x}_j)$, between pairs of observations, $\mathbf{x}_i$ and $\mathbf{x}_j$, say, so that the closer $\mathbf{x}_i$ and $\mathbf{x}_j$ are to each other, the larger the value of prox$(\mathbf{x}_i, \mathbf{x}_j)$.

If the two observations$\mathbf{x}_i$ and $\mathbf{x}_j$ end up at the same terminal node in $\{T^{*b}\}$, we increase prox$(\mathbf{x}_i, \mathbf{x}_j)$ by one.

We repeat this procedure over all B trees in the forest, and then divide the frequency totals of pairwise proximities by the number, B, of trees in the forest.

This gives us the proportion of all trees for which each pair of observations end up at the same terminal nodes.

The results are subtracted from one to yield dissimilarities:

$$\delta_{ij} = 1 - \text{prox}(\mathbf{x}_i, \mathbf{x}_j), \quad \mathbf{x}_i, \mathbf{x}_j \in \mathcal{L}, \quad i, j = 1, \ldots, n.$$

We collect these pairwise dissimilarities into an $(n \times n)$ proximity matrix $\mathbf{\Delta} = (\delta_{ij})$, which is symmetric, positive-definite, with diagonal entries equal to zero.

Identifying Multivariate Outliers

Detecting and identifying outliers in multivariate data can be very difficult, especially when the dimensionality is high.

So, any procedure that is successful in outlier-detection is worth its weight in gold.

The proximities computed for random forests can be used to detect outliers.

The basic idea is that we identify an outlier by how far away it is from all other observations belonging to its class in the learning set.

Suppose $\mathbf{x}_i \in \Pi_k$.

If the proximity of, say, $\mathbf{x}_i$ to another kth-class observation, say, $\mathbf{x}_j$ is small, then it is rare for those two observations to end up at the same terminal nodes when they are simultaneously dropped down all the trees in the forest.

In other words, $\mathbf{x}_i$ and $\mathbf{x}_j$ are far apart from each other iff their proximity is small.

If $\mathbf{x}_i$ is far away from all the other kth-class observations in the learning set, then all the proximities, $\text{prox}(\mathbf{x}_i, \mathbf{x}_l)$, of $\mathbf{x}_i$ with $\mathbf{x}_l$, $l \neq i$, will be small.

Breiman and Cutler (2004) suggest that a raw outlier measure for the ith observation, $\mathbf{x}_i$, in the kth class be given by

$$u_{ik} = \frac{n}{\sum_{\mathbf{x}_l \in \Pi_k,\ l \neq i} \left[\text{prox}(\mathbf{x}_i, \mathbf{x}_l)\right]^2}, \quad i = 1, 2, \ldots, n, \quad k = 1, 2, \ldots, K.$$

Thus, if $\mathbf{x}_i$ is really an outlier for the kth class, the denominator will be small, so that u_{ik} will be large.

Let $m_k = \text{median}_{\mathbf{x}_l \in \Pi_k}\{u_{lk}\}$ be the median of the raw outlier measures over all kth-class observations.

Then, for $k = 1, 2, \ldots, K$, a standardized version of u_{ik} is given by

$$\tilde{u}_{ik} = \frac{u_{ik} - m_k}{\sum_{\mathbf{x}_l \in \Pi_k} |u_{lk} - m_k|}, \quad i = 1, 2, \ldots, n.$$

The values of the above formula are plotted against sequence number, with each class's values plotted using either a different symbol or color.

Values of the above formula in excess of 10 should generate concern.

Treating Unbalanced Classes

A major impediment to good classification in practical problems occurs when at least one of the classes (often the class of primary interest) contains only a very small proportion of the observations.

Examples of such unbalanced situations include detection of fraudulent telephone calls, information retrieval and filtering, diagnosis of rare thyroid diseases, and detection of oil spills from satellite images.

In each of these examples, the result is wildly varying prediction errors for the different classes.

Classification algorithms, which focus on minimizing the overall misclassification rate, classify most observations according to the class of the majority of observations (the majority class).

As a result, the misclassification rate will be very low, but the observations belonging to the class of primary interest (the minority class) will be totally misclassified.

In the case of random forests, for example, the bootstrap samples will contain very few (and maybe none) of the minority class observations, and so we will see poor class prediction (i.e., high prediction error) for the minority class.

To alleviate such difficulties, various modifications to the random forest classifier were considered by Chen, Liaw, and Breiman (2004), including balanced random forest (BRF), where the majority class is undersampled, and weighted random forest (WRF), where a heavier weight is placed upon selecting the minority class in bootstrap samples in order to prevent misclassifying that class.

Based upon experiments with various data sets, no real difference in prediction error has been found between BRF and WRF, although BRF turns out to be computationally more efficient.