10
Boosting and Additive Trees

10.1 Boosting Methods

Boosting is one of the most powerful learning ideas introduced in the last twenty years. It was originally designed for classification problems, but as will be seen in this chapter, it can profitably be extended to regression as well. The motivation for boosting was a procedure that combines the outputs of many "weak" classifiers to produce a powerful "committee." From this perspective boosting bears a resemblance to bagging and other committee-based approaches (Section 8.8). However we shall see that the connection is at best superficial and that boosting is fundamentally different.

We begin by describing the most popular boosting algorithm due to Freund and Schapire (1997) called "AdaBoost.M1." Consider a two-class problem, with the output variable coded as $Y \in \{-1, 1\}$. Given a vector of predictor variables X, a classifier $G(X)$ produces a prediction taking one of the two values $\{-1, 1\}$. The error rate on the training sample is

$$\overline{\text{err}} = \frac{1}{N}\sum_{i=1}^{N} I(y_i \neq G(x_i)),$$

and the expected error rate on future predictions is $\text{E}_{XY} I(Y \neq G(X))$.

A weak classifier is one whose error rate is only slightly better than random guessing. The purpose of boosting is to sequentially apply the weak classification algorithm to repeatedly modified versions of the data, thereby producing a sequence of weak classifiers $G_m(x), m = 1, 2, \ldots, M$.

T. Hastie et al., *The Elements of Statistical Learning, Second Edition,*
DOI: 10.1007/b94608_10,

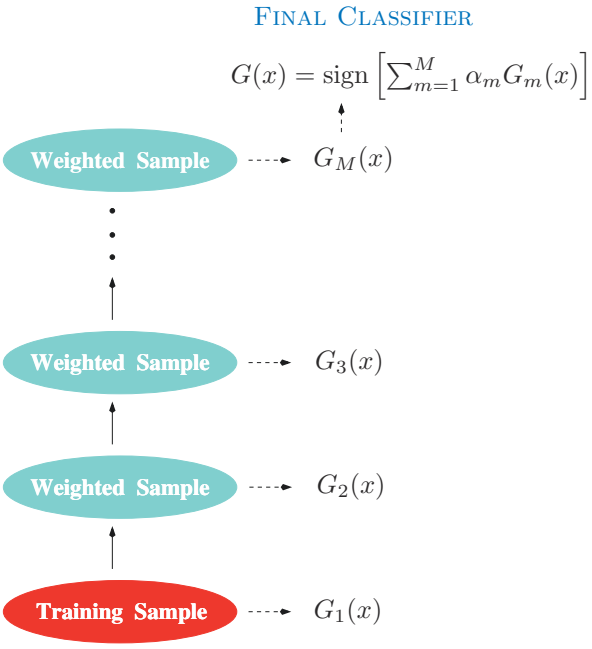

FIGURE 10.1. *Schematic of AdaBoost. Classifiers are trained on weighted versions of the dataset, and then combined to produce a final prediction.*

The predictions from all of them are then combined through a weighted majority vote to produce the final prediction:

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right). \tag{10.1}$$

Here $\alpha_1, \alpha_2, \ldots, \alpha_M$ are computed by the boosting algorithm, and weight the contribution of each respective $G_m(x)$. Their effect is to give higher influence to the more accurate classifiers in the sequence. Figure 10.1 shows a schematic of the AdaBoost procedure.

The data modifications at each boosting step consist of applying weights $w_1, w_2, \ldots, w_N$ to each of the training observations (x_i, y_i), $i = 1, 2, \ldots, N$. Initially all of the weights are set to $w_i = 1/N$, so that the first step simply trains the classifier on the data in the usual manner. For each successive iteration $m = 2, 3, \ldots, M$ the observation weights are individually modified and the classification algorithm is reapplied to the weighted observations. At step m, those observations that were misclassified by the classifier $G_{m-1}(x)$ induced at the previous step have their weights increased, whereas the weights are decreased for those that were classified correctly. Thus as iterations proceed, observations that are difficult to classify correctly receive ever-increasing influence. Each successive classifier is thereby forced

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

to concentrate on those training observations that are missed by previous ones in the sequence.

Algorithm 10.1 shows the details of the AdaBoost.M1 algorithm. The current classifier $G_m(x)$ is induced on the weighted observations at line 2a. The resulting weighted error rate is computed at line 2b. Line 2c calculates the weight α_m given to $G_m(x)$ in producing the final classifier $G(x)$ (line 3). The individual weights of each of the observations are updated for the next iteration at line 2d. Observations misclassified by $G_m(x)$ have their weights scaled by a factor $\exp(\alpha_m)$, increasing their relative influence for inducing the next classifier $G_{m+1}(x)$ in the sequence.

The AdaBoost.M1 algorithm is known as "Discrete AdaBoost" in Friedman et al. (2000), because the base classifier $G_m(x)$ returns a discrete class label. If the base classifier instead returns a real-valued prediction (e.g., a probability mapped to the interval $[-1, 1]$), AdaBoost can be modified appropriately (see "Real AdaBoost" in Friedman et al. (2000)).

The power of AdaBoost to dramatically increase the performance of even a very weak classifier is illustrated in Figure 10.2. The features $X_1, \ldots, X_{10}$ are standard independent Gaussian, and the deterministic target Y is defined by

$$Y = \begin{cases} 1 & \text{if } \sum_{j=1}^{10} X_j^2 > \chi^2_{10}(0.5), \\ -1 & \text{otherwise.} \end{cases} \tag{10.2}$$

Here $\chi^2_{10}(0.5) = 9.34$ is the median of a chi-squared random variable with 10 degrees of freedom (sum of squares of 10 standard Gaussians). There are 2000 training cases, with approximately 1000 cases in each class, and 10,000 test observations. Here the weak classifier is just a "stump": a two terminal-node classification tree. Applying this classifier alone to the training data set yields a very poor test set error rate of 45.8%, compared to 50% for

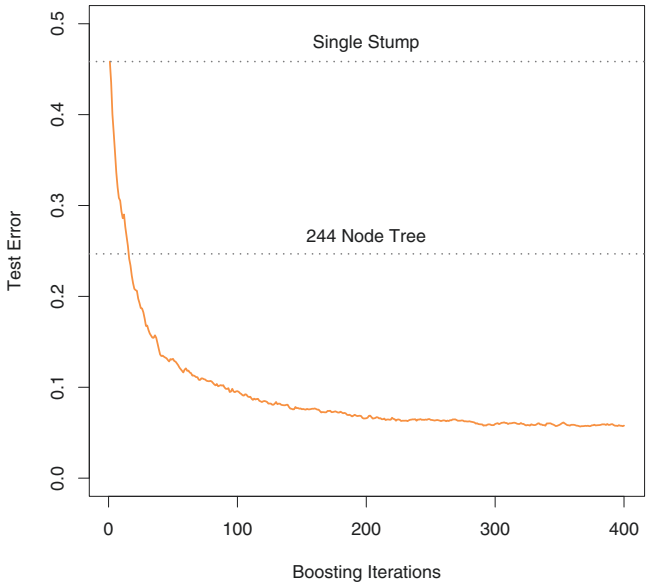

FIGURE 10.2. *Simulated data (10.2): test error rate for boosting with stumps, as a function of the number of iterations. Also shown are the test error rate for a single stump, and a* 244*-node classification tree.*

random guessing. However, as boosting iterations proceed the error rate steadily decreases, reaching 5.8% after 400 iterations. Thus, boosting this simple very weak classifier reduces its prediction error rate by almost a factor of four. It also outperforms a single large classification tree (error rate 24.7%). Since its introduction, much has been written to explain the success of AdaBoost in producing accurate classifiers. Most of this work has centered on using classification trees as the "base learner" $G(x)$, where improvements are often most dramatic. In fact, Breiman (NIPS Workshop, 1996) referred to AdaBoost with trees as the "best off-the-shelf classifier in the world" (see also Breiman (1998)). This is especially the case for data-mining applications, as discussed more fully in Section 10.7 later in this chapter.

10.1.1 Outline of This Chapter

Here is an outline of the developments in this chapter:

- We show that AdaBoost fits an additive model in a base learner, optimizing a novel exponential loss function. This loss function is

very similar to the (negative) binomial log-likelihood (Sections 10.2–10.4).

- The population minimizer of the exponential loss function is shown to be the log-odds of the class probabilities (Section 10.5).
- We describe loss functions for regression and classification that are more robust than squared error or exponential loss (Section 10.6).
- It is argued that decision trees are an ideal base learner for data mining applications of boosting (Sections 10.7 and 10.9).
- We develop a class of gradient boosted models (GBMs), for boosting trees with any loss function (Section 10.10).
- The importance of "slow learning" is emphasized, and implemented by shrinkage of each new term that enters the model (Section 10.12), as well as randomization (Section 10.12.2).
- Tools for interpretation of the fitted model are described (Section 10.13).

10.2 Boosting Fits an Additive Model

The success of boosting is really not very mysterious. The key lies in expression (10.1). Boosting is a way of fitting an additive expansion in a set of elementary "basis" functions. Here the basis functions are the individual classifiers $G_m(x) \in \{-1, 1\}$. More generally, basis function expansions take the form

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m), \tag{10.3}$$

where $\beta_m, m = 1, 2, \ldots, M$ are the expansion coefficients, and $b(x; \gamma) \in \mathbb{R}$ are usually simple functions of the multivariate argument x, characterized by a set of parameters γ. We discuss basis expansions in some detail in Chapter 5.

Additive expansions like this are at the heart of many of the learning techniques covered in this book:

- In single-hidden-layer neural networks (Chapter 11), $b(x; \gamma) = \sigma(\gamma_0 + \gamma_1^T x)$, where $\sigma(t) = 1/(1 + e^{-t})$ is the sigmoid function, and γ parameterizes a linear combination of the input variables.
- In signal processing, wavelets (Section 5.9.1) are a popular choice with γ parameterizing the location and scale shifts of a "mother" wavelet.
- Multivariate adaptive regression splines (Section 9.4) uses truncated-power spline basis functions where γ parameterizes the variables and values for the knots.

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M:
   (a) Compute
   $$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$
   (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.

- For trees, γ parameterizes the split variables and split points at the internal nodes, and the predictions at the terminal nodes.

Typically these models are fit by minimizing a loss function averaged over the training data, such as the squared-error or a likelihood-based loss function,

$$\min_{\{\beta_m,\gamma_m\}_1^M} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m)\right). \tag{10.4}$$

For many loss functions $L(y, f(x))$ and/or basis functions $b(x; \gamma)$, this requires computationally intensive numerical optimization techniques. However, a simple alternative often can be found when it is feasible to rapidly solve the subproblem of fitting just a single basis function,

$$\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, \beta b(x_i; \gamma)\right). \tag{10.5}$$

10.3 Forward Stagewise Additive Modeling

Forward stagewise modeling approximates the solution to (10.4) by sequentially adding new basis functions to the expansion without adjusting the parameters and coefficients of those that have already been added. This is outlined in Algorithm 10.2. At each iteration m, one solves for the optimal basis function $b(x; \gamma_m)$ and corresponding coefficient β_m to add to the current expansion $f_{m-1}(x)$. This produces $f_m(x)$, and the process is repeated. Previously added terms are not modified.

For squared-error loss

$$L(y, f(x)) = (y - f(x))^2, \tag{10.6}$$

one has

$$\begin{aligned} L(y_i, f_{m-1}(x_i) + \beta b(x_i;\gamma)) &= (y_i - f_{m-1}(x_i) - \beta b(x_i;\gamma))^2 \\ &= (r_{im} - \beta b(x_i;\gamma))^2, \end{aligned} \tag{10.7}$$

where $r_{im} = y_i - f_{m-1}(x_i)$ is simply the residual of the current model on the ith observation. Thus, for squared-error loss, the term $\beta_m b(x;\gamma_m)$ that best fits the current residuals is added to the expansion at each step. This idea is the basis for "least squares" regression boosting discussed in Section 10.10.2. However, as we show near the end of the next section, squared-error loss is generally not a good choice for classification; hence the need to consider other loss criteria.

10.4 Exponential Loss and AdaBoost

We now show that AdaBoost.M1 (Algorithm 10.1) is equivalent to forward stagewise additive modeling (Algorithm 10.2) using the loss function

$$L(y, f(x)) = \exp(-y\, f(x)). \tag{10.8}$$

The appropriateness of this criterion is addressed in the next section.

For AdaBoost the basis functions are the individual classifiers $G_m(x) \in \{-1, 1\}$. Using the exponential loss function, one must solve

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{N} \exp[-y_i(f_{m-1}(x_i) + \beta\, G(x_i))]$$

for the classifier G_m and corresponding coefficient β_m to be added at each step. This can be expressed as

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{N} w_i^{(m)} \exp(-\beta\, y_i\, G(x_i)) \tag{10.9}$$

with $w_i^{(m)} = \exp(-y_i\, f_{m-1}(x_i))$. Since each $w_i^{(m)}$ depends neither on β nor $G(x)$, it can be regarded as a weight that is applied to each observation. This weight depends on $f_{m-1}(x_i)$, and so the individual weight values change with each iteration m.

The solution to (10.9) can be obtained in two steps. First, for any value of $\beta > 0$, the solution to (10.9) for $G_m(x)$ is

$$G_m = \arg\min_{G} \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)), \tag{10.10}$$

which is the classifier that minimizes the weighted error rate in predicting y. This can be easily seen by expressing the criterion in (10.9) as

$$e^{-\beta} \cdot \sum_{y_i = G(x_i)} w_i^{(m)} + e^{\beta} \cdot \sum_{y_i \neq G(x_i)} w_i^{(m)},$$

which in turn can be written as

$$\left(e^{\beta} - e^{-\beta}\right) \cdot \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)) + e^{-\beta} \cdot \sum_{i=1}^{N} w_i^{(m)}. \tag{10.11}$$

Plugging this G_m into (10.9) and solving for β one obtains

$$\beta_m = \frac{1}{2} \log \frac{1 - \text{err}_m}{\text{err}_m}, \tag{10.12}$$

where err_m is the minimized weighted error rate

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i^{(m)}}. \tag{10.13}$$

The approximation is then updated

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x),$$

which causes the weights for the next iteration to be

$$w_i^{(m+1)} = w_i^{(m)} \cdot e^{-\beta_m y_i G_m(x_i)}. \tag{10.14}$$

Using the fact that $-y_i G_m(x_i) = 2 \cdot I(y_i \neq G_m(x_i)) - 1$, (10.14) becomes

$$w_i^{(m+1)} = w_i^{(m)} \cdot e^{\alpha_m I(y_i \neq G_m(x_i))} \cdot e^{-\beta_m}, \tag{10.15}$$

where $\alpha_m = 2\beta_m$ is the quantity defined at line 2(c) of AdaBoost.M1 (Algorithm 10.1). The factor $e^{-\beta_m}$ in (10.15) multiplies all weights by the same value, so it has no effect. Thus (10.15) is equivalent to line 2(d) of Algorithm 10.1.

One can view line 2(a) of the Adaboost.M1 algorithm as a method for approximately solving the minimization in (10.11) and hence (10.10). Hence we conclude that AdaBoost.M1 minimizes the exponential loss criterion (10.8) via a forward-stagewise additive modeling approach.

Figure 10.3 shows the training-set misclassification error rate and average exponential loss for the simulated data problem (10.2) of Figure 10.2. The training-set misclassification error decreases to zero at around 250 iterations (and remains there), but the exponential loss keeps decreasing. Notice also in Figure 10.2 that the test-set misclassification error continues to improve after iteration 250. Clearly Adaboost is not optimizing training-set misclassification error; the exponential loss is more sensitive to changes in the estimated class probabilities.

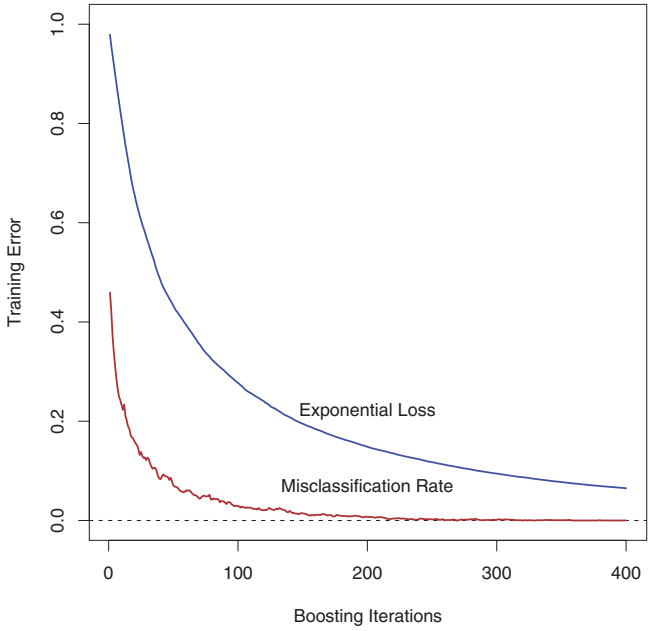

FIGURE 10.3. *Simulated data, boosting with stumps: misclassification error rate on the training set, and average exponential loss:* $(1/N)\sum_{i=1}^{N}\exp(-y_i f(x_i))$. *After about* 250 *iterations, the misclassification error is zero, while the exponential loss continues to decrease.*

10.5 Why Exponential Loss?

The AdaBoost.M1 algorithm was originally motivated from a very different perspective than presented in the previous section. Its equivalence to forward stagewise additive modeling based on exponential loss was only discovered five years after its inception. By studying the properties of the exponential loss criterion, one can gain insight into the procedure and discover ways it might be improved.

The principal attraction of exponential loss in the context of additive modeling is computational; it leads to the simple modular reweighting AdaBoost algorithm. However, it is of interest to inquire about its statistical properties. What does it estimate and how well is it being estimated? The first question is answered by seeking its population minimizer.

It is easy to show (Friedman et al., 2000) that

$$f^*(x) = \arg\min_{f(x)} \mathrm{E}_{Y|x}(e^{-Yf(x)}) = \frac{1}{2}\log\frac{\Pr(Y=1|x)}{\Pr(Y=-1|x)}, \tag{10.16}$$

or equivalently

$$\Pr(Y = 1|x) = \frac{1}{1 + e^{-2f^*(x)}}.$$

Thus, the additive expansion produced by AdaBoost is estimating one-half the log-odds of $P(Y = 1|x)$. This justifies using its sign as the classification rule in (10.1).

Another loss criterion with the same population minimizer is the binomial negative log-likelihood or *deviance* (also known as cross-entropy), interpreting f as the logit transform. Let

$$p(x) = \Pr(Y = 1 \,|\, x) = \frac{e^{f(x)}}{e^{-f(x)} + e^{f(x)}} = \frac{1}{1 + e^{-2f(x)}} \tag{10.17}$$

and define $Y' = (Y + 1)/2 \in \{0, 1\}$. Then the binomial log-likelihood loss function is

$$l(Y, p(x)) = Y' \log p(x) + (1 - Y') \log(1 - p(x)),$$

or equivalently the deviance is

$$-l(Y, f(x)) = \log\left(1 + e^{-2Yf(x)}\right). \tag{10.18}$$

Since the population maximizer of log-likelihood is at the true probabilities $p(x) = \Pr(Y = 1 \,|\, x)$, we see from (10.17) that the population minimizers of the deviance $\mathrm{E}_{Y|x}[-l(Y, f(x))]$ and $\mathrm{E}_{Y|x}[e^{-Yf(x)}]$ are the same. Thus, using either criterion leads to the same solution at the population level. Note that e^{-Yf} itself is not a proper log-likelihood, since it is not the logarithm of any probability mass function for a binary random variable $Y \in \{-1, 1\}$.

10.6 Loss Functions and Robustness

In this section we examine the different loss functions for classification and regression more closely, and characterize them in terms of their robustness to extreme data.

Robust Loss Functions for Classification

Although both the exponential (10.8) and binomial deviance (10.18) yield the same solution when applied to the population joint distribution, the same is not true for finite data sets. Both criteria are monotone decreasing functions of the "margin" $yf(x)$. In classification (with a $-1/1$ response) the margin plays a role analogous to the residuals $y-f(x)$ in regression. The classification rule $G(x) = \text{sign}[f(x)]$ implies that observations with positive margin $y_i f(x_i) > 0$ are classified correctly whereas those with negative margin $y_i f(x_i) < 0$ are misclassified. The decision boundary is defined by