17

Random Forests and Boosting

In the modern world we are often faced with enormous data sets, both in terms of the number of observations n and in terms of the number of variables p. This is of course good news—we have always said the more data we have, the better predictive models we can build. Well, we are there now—we have tons of data, and must figure out how to use it.

Although we can scale up our software to fit the collection of linear and generalized linear models to these behemoths, they are often too modest and can fall way short in terms of predictive power. A need arose for some general purpose tools that could scale well to these bigger problems, and exploit the large amount of data by fitting a much richer class of functions, almost automatically. Random forests and boosting are two relatively recent innovations that fit the bill, and have become very popular as "out-the-box" learning algorithms that enjoy good predictive performance. Random forests are somewhat more automatic than boosting, but can also suffer a small performance hit as a consequence.

These two methods have something in common: they both represent the fitted model by a sum of regression trees. We discuss trees in some detail in Chapter 8. A single regression tree is typically a rather *weak* prediction model; it is rather amazing that an ensemble of trees leads to the state of the art in black-box predictors!

We can broadly describe both these methods very simply.

Random forest Grow many deep regression trees to randomized versions of the training data, and average them. Here "randomized" is a wide-ranging term, and includes bootstrap sampling and/or subsampling of the observations, as well as subsampling of the variables.

Boosting Repeatedly grow shallow trees to the residuals, and hence build up an additive model consisting of a sum of trees.

The basic mechanism in random forests is variance reduction by averaging. Each deep tree has a high variance, and the averaging brings the vari-

ance down. In boosting the basic mechanism is bias reduction, although different flavors include some variance reduction as well. Both methods inherit all the good attributes of trees, most notable of which is variable selection.

17.1 Random Forests

Suppose we have the usual setup for a regression problem, with a training set consisting of an $n \times p$ data matrix $\boldsymbol{X}$ and an n-vector of responses $\boldsymbol{y}$. A tree (Section 8.4) fits a piecewise constant surface $\hat{r}(x)$ over the domain $\mathcal{X}$ by recursive partitioning. The model is built in a greedy fashion, each time creating two daughter nodes from a terminal node by defining a binary split using one of the available variables. The model can hence be represented by a binary tree. Part of the art in using regression trees is to know how deep to grow the tree, or alternatively how much to prune it back. Typically that is achieved using left-out data or cross-validation. Figure 17.1 shows a tree fit to the `spam` training data. The splitting variables and split points are indicated. Each node is labeled as `spam` or `ham` (not spam; see footnote 7 on page 115). The numbers beneath each node show misclassified/total. The overall misclassification error on the test data is 9.3%, which compares poorly with the performance of the lasso (Figure 16.9: 7.1% for linear lasso, 5.7% for lasso with interactions). The surface $\hat{r}(x)$ here is clearly complex, and by its nature represents a rather high-order interaction (the deepest branch is eight levels, and involves splits on eight different variables). Despite the promise to deliver interpretable models, this bushy tree is not easy to interpret. Nevertheless, trees have some desirable properties. The following lists some of the good and bad properties of trees.

- ▲ Trees automatically select variables; only variables used in defining splits are *in* the model.
- ▲ Tree-growing algorithms scale well to large n; growing a tree is a divide-and-conquer operation.
- ▲ Trees handle mixed features (quantitative/qualitative) seamlessly, and can deal with missing data.
- ▲ Small trees are easy to interpret.
- ▼ Large trees are not easy to interpret.
- ▼ Trees do not generally have good prediction performance.

Trees are inherently high-variance function estimators, and the bushier they are, the higher the variance. The early splits dictate the architecture of

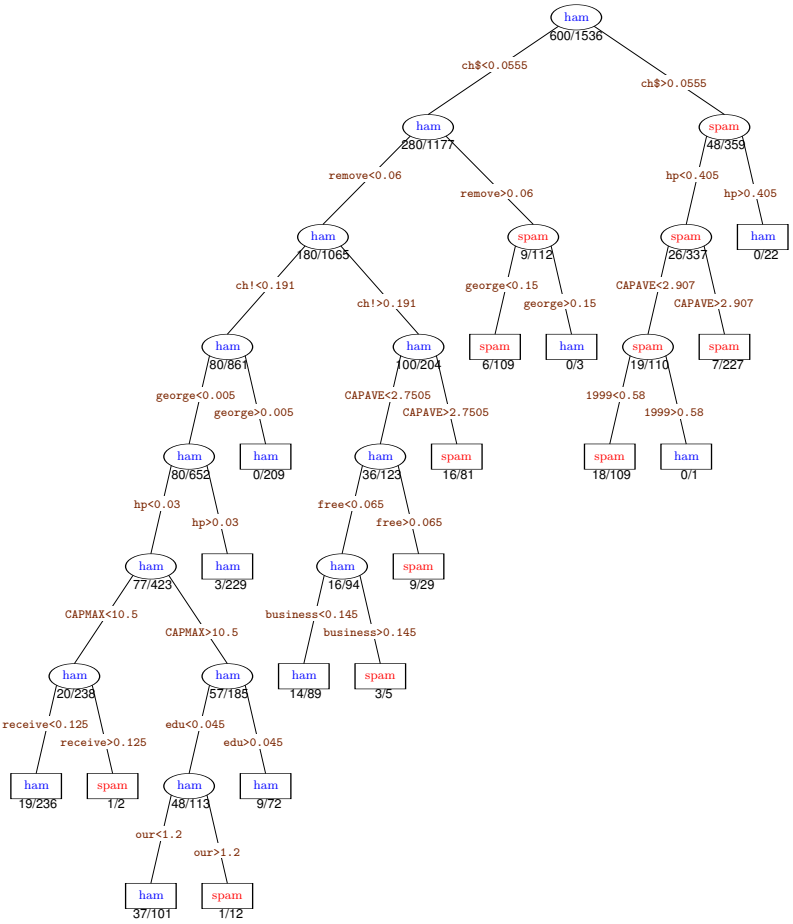

Figure 17.1 Regression tree fit to the binary `spam` data, a bigger version of Figure 8.7. The initial trained tree was far bushier than the one displayed; it was then *optimally* pruned using 10-fold cross-validation.

the tree. On the other hand, deep bushy trees localize the training data (using the variables that matter) to a relatively small region around the target point. This suggests low bias. The idea of random forests (and its predecessor bagging) is to grow many very bushy trees, and get rid of the variance by averaging. In order to benefit from averaging, the individual trees should not be too correlated. This is achieved by injecting some randomness into the tree-growing process. Random forests achieve this in two ways.

1 Bootstrap: each tree is grown to a bootstrap resampled training data set, which makes them different and somewhat decorrelates them.
2 Split-variable randomization: each time a split is to be performed, the search for the split variable is limited to a random subset of m of the p variables. Typical values of m are $\sqrt{p}$ or $p/3$.

When $m = p$, the randomization amounts to using only step 1, and was an earlier ancestor of random forests called *bagging*. In most examples the second level of randomization pays dividends.

Algorithm 17.1 RANDOM FOREST.

1 Given training data set $\boldsymbol{d} = (\boldsymbol{X}, \boldsymbol{y})$. Fix $m \leq p$ and the number of trees B.
2 For $b = 1, 2, \ldots, B$, do the following.
(a) Create a bootstrap version of the training data $\boldsymbol{d}_b^*$, by randomly sampling the n rows with replacement n times. The sample can be represented by the bootstrap frequency vector $\boldsymbol{w}_b^*$.
(b) Grow a maximal-depth tree $\hat{r}_b(x)$ using the data in $\boldsymbol{d}_b^*$, sampling m of the p features at random prior to making each split.
(c) Save the tree, as well as the bootstrap sampling frequencies for each of the training observations.
3 Compute the random-forest fit at any prediction point x_0 as the average

$$\hat{r}_{\text{rf}}(x_0) = \frac{1}{B}\sum_{b=1}^{B} \hat{r}_b(x_0).$$

4 Compute the OOB_i error for each response observation y_i in the training data, by using the fit $\hat{r}_{\text{rf}}^{(i)}$, obtained by averaging only those $\hat{r}_b(x_i)$ for which observation i was *not* in the bootstrap sample. The overall OOB error is the average of these OOB_i.

Algorithm 17.1 gives some of the details; some more are given in the technical notes.† †1

Random forests are easy to use, since there is not much tuning needed. The package `randomForest` in `R` sets as a default $m = \sqrt{p}$ for classification trees, and $m = p/3$ for regression trees, but one can use other values. With $m = 1$ the split variable is completely random, so all variables get a chance. This will decorrelate the trees the most, but can create bias, somewhat similar to that in ridge regression. Figure 17.2 shows the

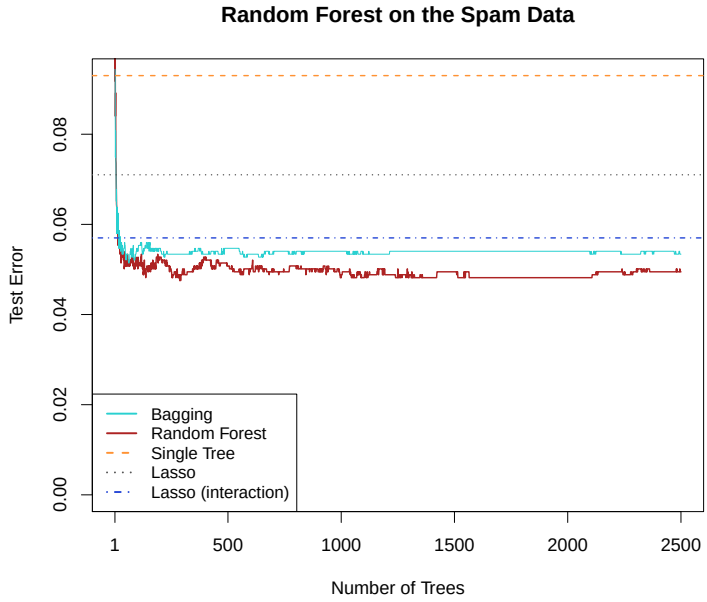

Figure 17.2 Test misclassification error of random forests on the **spam** data, as a function of the number of trees. The red curve selects $m = 7$ of the $p = 57$ features at random as candidates for the split variable, each time a split is made. The blue curve uses $m = 57$, and hence amounts to bagging. Both bagging and random forests outperform the lasso methods, and a single tree.

misclassification performance of a random forest on the **spam** test data, as a function of the number of trees averaged. We see that in this case, after a relatively small number of trees (500), the error levels off. The number B of trees averaged is not a real tuning parameter; as with the bootstrap (Chapters 10 and 11), we need a sufficient number for the estimate to stabilize, but cannot overfit by having too many.

Random forests have been described as adaptive nearest-neighbor estimators—adaptive in that they select predictors. A k-nearest-neighbor estimate finds the k training observations closest in feature space to the target point x_0, and averages their responses. Each tree in the random forest drills down by recursive partitioning to pure terminal nodes, often consisting of a single observation. Hence, when evaluating the prediction from each tree, $\hat{r}_b(x_0) = y_\ell$ for some ℓ, and for many of the trees this could be the same

ℓ. From the whole collection of B trees, the number of distinct ℓs can be fairly small. Since the partitioning that reaches the terminal nodes involves only a subset of the predictors, the neighborhoods so defined are adaptive.

Out-of-Bag Error Estimates

Random forests deliver cross-validated error estimates at virtually no extra cost. The idea is similar to the bootstrap error estimates discussed in Chapter 10. The computation is described in step 4 of Algorithm 17.1. In making the prediction for observation pair (x_i, y_i), we average all the random-forest trees $\hat{r}_b(x_i)$ for which that pair is not in the corresponding bootstrap sample:

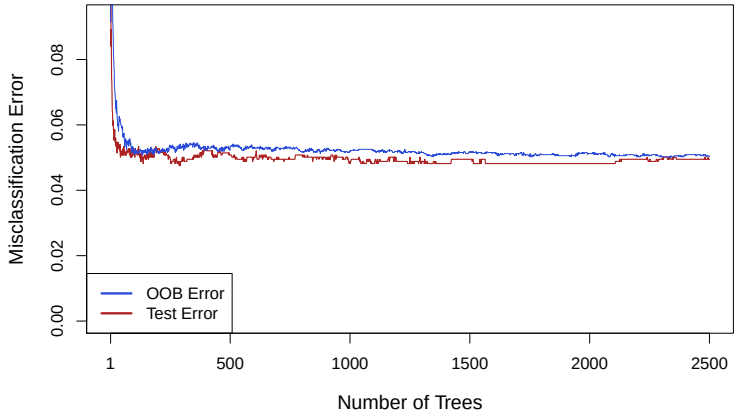

Figure 17.3 Out-of-bag misclassification error estimate for the `spam` data (blue) versus the test error (red), as a function of the number of trees.

$$\hat{r}_{\text{rf}}^{(i)}(x_i) = \frac{1}{B_i} \sum_{b\,:\,w_{bi}^*=0} \hat{r}_b(x_i), \tag{17.1}$$

where B_i is the number of times observation i was not in the bootstrap sample (with expected value $e^{-1}B \approx 0.37B$). We then compute the OOB error estimate

$$\text{err}_{\text{OOB}} = \frac{1}{n} \sum_{i=1}^{n} L[y_i, \hat{r}_{\text{rf}}^{(i)}(x_i)], \tag{17.2}$$

where L is the loss function of interest, such as misclassification or squared-error loss. If B is sufficiently large (about three times the number needed for the random forest to stabilize), we can see that the OOB error estimate is equivalent to leave-one-out cross-validation error.

Standard Errors

We can use very similar ideas to estimate the variance of a random-forest prediction, using the *jackknife* variance estimator (see (10.6) in Chapter 10). If $\hat{\theta}$ is a statistic estimated using all n training observations, then the jackknife estimate of the variance of $\hat{\theta}$ is given by

$$\widehat{\mathrm{V}}_{\mathrm{jack}}(\hat{\theta}) = \frac{n-1}{n}\sum_{i=1}^{n}\left(\hat{\theta}_{(i)} - \hat{\theta}_{(\cdot)}\right)^2, \tag{17.3}$$

where $\hat{\theta}_{(i)}$ is the estimate using all but observation i, and $\hat{\theta}_{(\cdot)} = \frac{1}{n}\sum_i \hat{\theta}_{(i)}$.

The natural jackknife variance estimate for a random-forest prediction at x_0 is obtained by simply plugging into this formula:

$$\widehat{\mathrm{V}}_{\mathrm{jack}}(\hat{r}_{\mathrm{rf}}(x_0)) = \frac{n-1}{n}\sum_{i=1}^{n}\left(\hat{r}_{\mathrm{rf}}^{(i)}(x_0) - \hat{r}_{\mathrm{rf}}(x_0)\right)^2. \tag{17.4}$$

This formula is derived under the $B = \infty$ setting, in which case $\hat{r}_{\mathrm{rf}}(x_0)$ is an expectation under bootstrap sampling, and hence is free of Monte Carlo variability. This also makes the distinction clear: we are estimating the sampling variability of a random-forest prediction $\hat{r}_{\mathrm{rf}}(x_0)$, as distinct from any Monte Carlo variation. In practice B is finite, and expression (17.4) will have Monte Carlo bias and variance. All of the $\hat{r}_{\mathrm{rf}}^{(i)}(x_0)$ are based on B bootstrap samples, and they are hence noisy versions of their expectations. Since the n quantities summed in (17.4) are squared, by Jensen's inequality we will have positive bias (and it turns out that this bias dominates the Monte Carlo variance). Hence one would want to use a much larger value of B when estimating variances, than was used in the original random-forest fit. Alternatively, one can use the same B bootstrap samples as were used to fit the random forest, along with a *bias-corrected* version of the jackknife variance estimate:[†] †2

$$\widehat{\mathrm{V}}_{\mathrm{jack}}^{\mathrm{u}}(\hat{r}_{\mathrm{rf}}(x_0)) = \widehat{\mathrm{V}}_{\mathrm{jack}}(\hat{r}_{\mathrm{rf}}(x_0)) - (e-1)\frac{n}{B}\hat{v}(x_0), \tag{17.5}$$

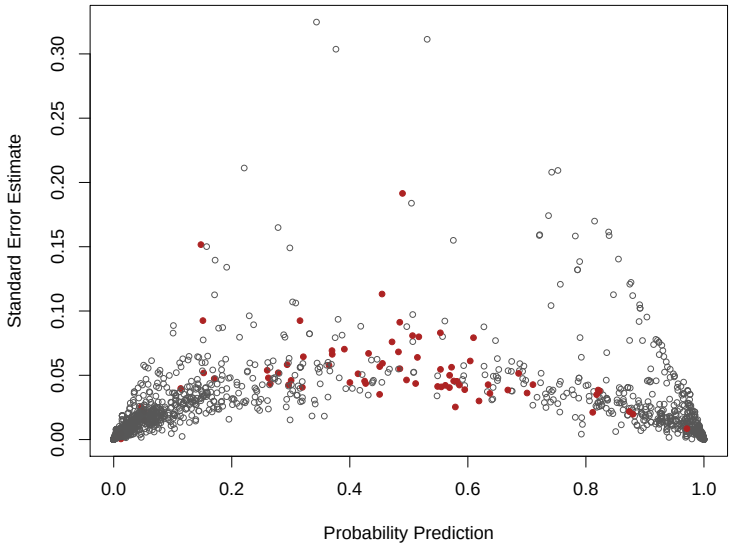

Figure 17.4 Jackknife Standard Error estimates (with bias correction) for the probability estimates in the `spam` test data. The points labeled red were misclassifications, and tend to concentrate near the decision boundary (0.5).

where $e = 2.718\ldots$, and

$$\hat{v}(x_0) = \frac{1}{B}\sum_{b=1}^{B}(\hat{r}_b(x_0) - \hat{r}_{\text{rf}}(x_0))^2\,, \tag{17.6}$$

the bootstrap estimate of the variance of a single random-forest tree. All these quantities are easily computed from the output of a random forest, so they are immediately available. Figure 17.4 shows the predicted probabilities and their jackknife estimated standard errors for the `spam` test data. The estimates near the decision boundary tend to have higher standard errors.

Variable-Importance Plots

A random forest is something of a black box, giving good predictions but usually not much insight into the underlying surface it has fit. Each random-forest tree $\hat{r}_b$ will have used a subset of the predictors as splitting variables, and each tree is likely to use overlapping but not necessarily

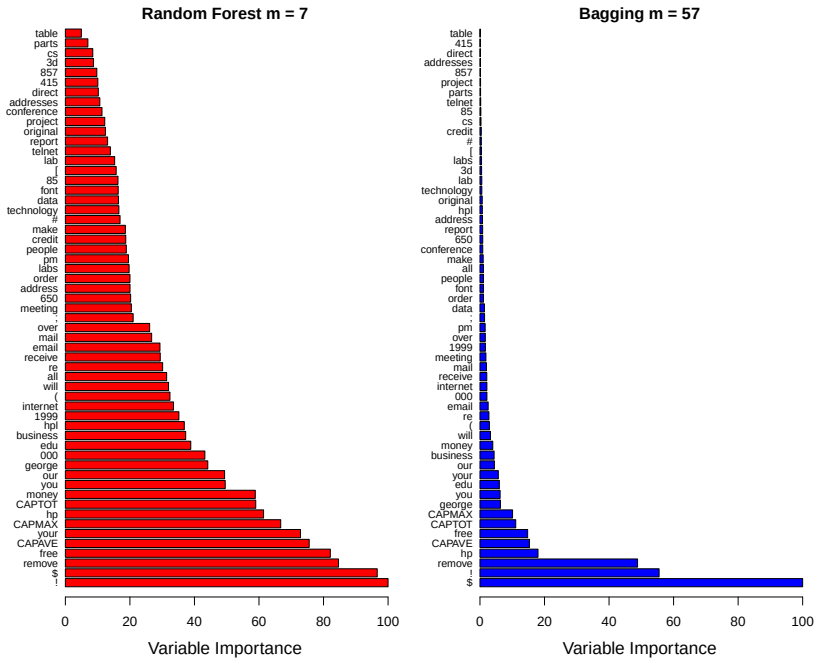

Figure 17.5 Variable-importance plots for random forests fit to the spam data. On the left we have the $m = 7$ random forest; due to the split-variable randomization, it spreads the importance among the variables. On the right is the $m = 57$ random forest or bagging, which focuses on a smaller subset of the variables.

identical subsets. One might conclude that any variable never used in any of the trees is unlikely to be important, but we would like a method of assessing the relative importance of variables that are included in the ensemble. Variable-importance plots fit this bill. Whenever a variable is used in a tree, the algorithm logs the decrease in the split-criterion due to this split. These are accumulated over all the trees, for each variable, and summarized as relative importance measures. Figure 17.5 demonstrates this on the **spam** data. We see that the $m = 7$ random forest, by virtue of the split-variable randomization, spreads the importance out much more than bagging, which always gets to pick the best variable for splitting. In this sense small m has some similarity to ridge regression, which also tends to share the coefficients evenly among correlated variables.

17.2 Boosting with Squared-Error Loss

Boosting was originally proposed as a means for improving the performance of "weak learners" in binary classification problems. This was achieved through resampling training points—giving more weight to those which had been misclassified—to produce a new classifier that would boost the performance in previously problematic areas of feature space. This process is repeated, generating a stream of classifiers, which are ultimately combined through voting[1] to produce the final classifier. The prototypical weak learner was a decision tree.

Boosting has evolved since this earliest invention, and different flavors are popular in statistics, computer science, and other areas of pattern recognition and prediction. We focus on the version popular in statistics—gradient boosting—and return to this early version later in the chapter. Algorithm 17.2

Algorithm 17.2 GRADIENT BOOSTING WITH SQUARED-ERROR LOSS.

1 Given a training sample $\boldsymbol{d} = (\boldsymbol{X}, \boldsymbol{y})$. Fix the number of steps B, the shrinkage factor ϵ and the tree depth d. Set the initial fit $\widehat{G}_0 \equiv 0$, and the residual vector $\boldsymbol{r} = \boldsymbol{y}$.

2 For $b = 1, 2, \ldots, B$ repeat:

(a) Fit a regression tree $\tilde{g}_b$ to the data $(\boldsymbol{X}, \boldsymbol{r})$, grown best-first to depth d: this means the total number of splits are d, and each successive split is made to that terminal node that yields the biggest reduction in residual sum of squares.

(b) Update the fitted model with a shrunken version of $\tilde{g}_b$: $\widehat{G}_b = \widehat{G}_{b-1} + \hat{g}_b$, with $\hat{g}_b = \epsilon \cdot \tilde{g}_b$.

(c) Update the residuals accordingly: $r_i = r_i - \hat{g}_b(x_i),\ i = 1, \ldots, n$.

3 Return the sequence of fitted functions $\widehat{G}_b,\ b = 1, \ldots, B$.

gives the most basic version of gradient boosting, for squared-error loss. This amounts to building a model by repeatedly fitting a regression tree to the residuals. Importantly, the tree is typically quite small, involving a small number d of splits—it is indeed a *weak learner*. After each tree has been grown to the residuals, it is shrunk down by a factor ϵ before it is added to the current model; this is a means of slowing the learning process. Despite the obvious similarities with a random forest, boosting is different in a fundamental way. The trees in a random forest are identically

[1] Each classifier $\hat{c}_b(x_0)$ predicts a class label, and the class with the most "votes" wins.

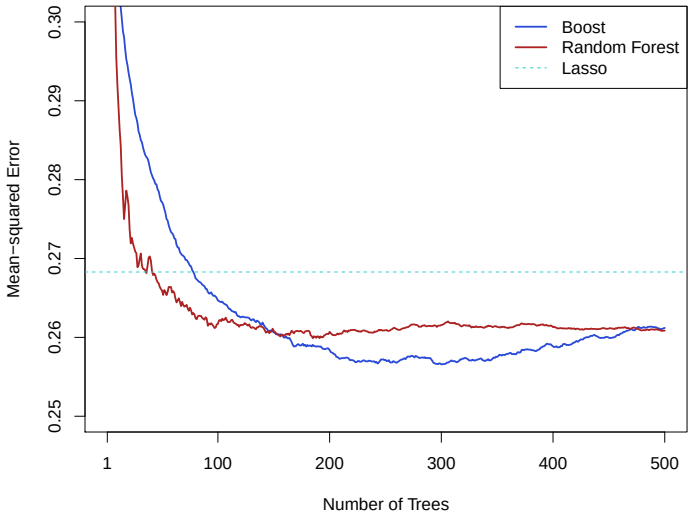

Figure 17.6 Test performance of a boosted regression-tree model fit to the **ALS** training data, with $n = 1197$ and $p = 369$. Shown is the mean-squared error on the 625 designated test observations, as a function of the number of trees. Here the depth $d = 4$ and $\epsilon = 0.02$. Boosting achieves a lower test MSE than a random forest. We see that as the number of trees B gets large, the test error for boosting starts to increase—a consequence of overfitting. The random forest does not overfit. The dotted blue horizontal line shows the best performance of a linear model, fit by the lasso. The differences are less dramatic than they appear, since the vertical scale does not extend to zero.

distributed—the same (random) treatment is repeatedly applied to the same data. With boosting, on the other hand, each tree is trying to amend errors made by the ensemble of previously grown trees. The number of terms B is important as well, because unlike random forests, a boosted regression model can overfit if B is too large. Hence there are three tuning parameters, B, d and ϵ, and each can change the performance of a boosted model, sometimes considerably.

†3 Figure 17.6 shows the test performance of boosting on the **ALS** data.† These data represent measurements on patients with *amyotrophic lateral sclerosis* (Lou Gehrig's disease). The goal is to predict the rate of progression of an ALS functional rating score (**FRS**). There are 1197 training

measurements on 369 predictors and the response, with a corresponding test set of size 625 observations.

As is often the case, boosting slightly outperforms a random forest here, but at a price. Careful tuning of boosting requires considerable extra work, with time-costly rounds of cross-validation, whereas random forests are almost automatic. In the following sections we explore in more detail some of the tuning parameters. The `R` package `gbm` implements gradient boosting, with some added bells and whistles. By default it grows each new tree on a 50% random sub-sample of the training data. Apart from speeding up the computations, this has a similar effect to bagging, and results in some variance reduction in the ensemble.

We can also compute a variable-importance plot, as we did for random forests; this is displayed in Figure 17.7 for the `ALS` data. Only 267 of the 369 variables were ever used, with one variable `Onset.Delta` standing out ahead of the others. This measures the amount of time that has elapsed since the patient was first diagnosed with ALS, and hence a larger value will indicate a slower progression rate.

Tree Depth and Interaction Order

Tree depth d is an important parameter for gradient boosted models, and the right choice will depend on the data at hand. Here depth $d = 4$ appears to be a good choice on the test data. Without test data, we could use cross-validation to make the selection. Apart from a general complexity measure, tree depth also controls the interaction order of the model.[2] The easiest case is with $d = 1$, where each tree consists of a single split (a stump). Suppose we have a fitted boosted model $\widehat{G}_B(x)$, using B trees. Denote by $\mathcal{B}_j \subseteq \mathcal{B} = \{1, 2, \ldots, B\}$ the indices of the trees that made the single split using variable j, for $j = 1, \ldots, p$. These $\mathcal{B}_j$ are disjoint (some $\mathcal{B}_\ell$ can be

[2] A $(k-1)$th-order interaction is also known as a k-way interaction. Hence an order-one interaction model has two-way interactions, and an order-zero model is additive.

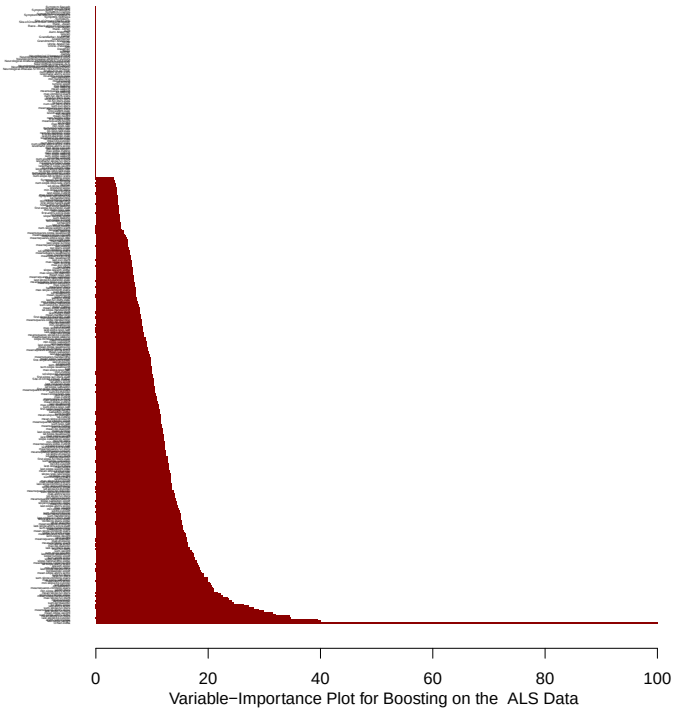

Figure 17.7 Variable importance plot for the `ALS` data. Here 267 of the 369 variables were used in the ensemble. There are too many variables for the labels to be visible, so this plot serves as a visual guide. Variable `Onset.Delta` has relative importance 100 (the lowest red bar), more than double the next two at around 40 (`last.slope.weight` and `alsfrs.score.slope`). However, the importances drop off slowly, suggesting that the model requires a significant fraction of the variables.

empty), and $\bigcup_{j=1}^{p} \mathcal{B}_j = \mathcal{B}$. Then we can write

$$\begin{aligned} \widehat{G}_B(x) &= \sum_{b=1}^{B} \hat{g}_b(x) \\ &= \sum_{j=1}^{p} \sum_{b \in \mathcal{B}_j} \hat{g}_b(x) \\ &= \sum_{j=1}^{p} \hat{f}_j(x_j). \end{aligned} \tag{17.7}$$

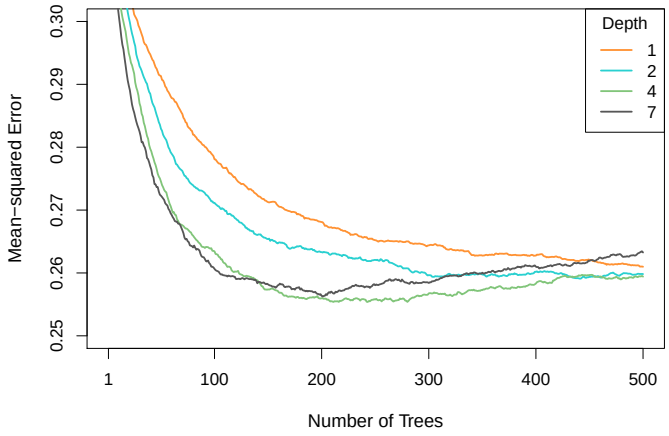

Figure 17.8 `ALS` test error for boosted models with different depth parameters d, and all using the same shrinkage parameter $\epsilon = 0.02$. It appears that $d = 1$ is inferior to the rest, with $d = 4$ about the best. With $d = 7$, overfitting begins around 200 trees, with $d = 4$ around 300, while neither of the other two show evidence of overfitting by 500 trees.

Hence boosted stumps fits an additive model, but in a fully adaptive way. It selects variables, and also selects how much action to devote to each variable. We return to additive models in Section 17.5. Figure 17.9 shows the three functions with highest relative importance. The first function confirms that a longer time since diagnosis (more negative `Onset.Delta`) predicts a slower decline. `last.slope.weight` is the difference in body weight at the last two visits—again positive is good. Likewise for `alsfrs.score.slope`, which measures the local slope of the `FRS` score after the first two visits.

In a similar way, boosting with $d = 2$ fits a two-way interaction model; each tree involves at most two variables. In general, boosting with $d = k$ leads to a $(k - 1)$th-order interaction model. Interaction order is perhaps a more natural way to think of model complexity.

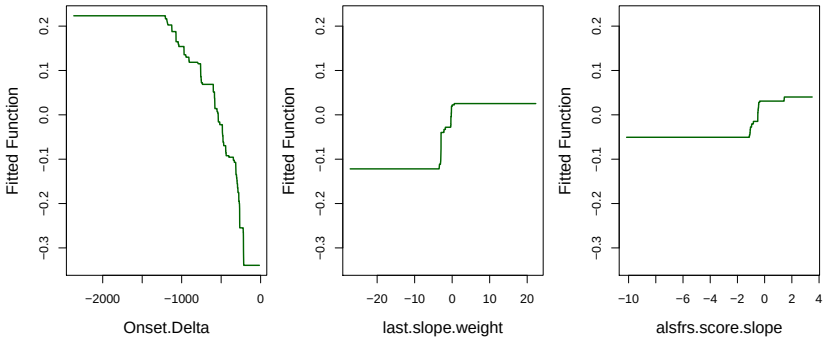

Figure 17.9 Three of the fitted functions (17.7) for the ALS data, in a boosted stumps model ($d = 1$), each centered to average zero over the training data. In terms of the outcome, bigger is better (slower decline in FRS). The first function confirms that a longer time since diagnosis (more negative value of `Onset.Delta`) predicts a slower decline. The variable `last.slope.weight` is the difference in body weight at the last two visits—again positive is good. Likewise for `alsfrs.score.slope`, which measures the local slope of the FRS score after the first two visits.

Shrinkage

The shrinkage parameter ϵ controls the rate at which boosting fits—and hence overfits—the data. Figure 17.10 demonstrates the effect of shrinkage on the ALS data. The under-shrunk ensemble (red) quickly overfits the data, leading to poor validation error. The blue ensemble uses a shrinkage parameter 20 times smaller, and reaches a lower validation error. The downside of a very small shrinkage parameter is that it can take many trees to adequately fit the data. On the other hand, the shrunken fits are smoother, take much longer to overfit, and hence are less sensitive to the stopping point B.

17.3 Gradient Boosting

We now turn our attention to boosting models using other than square-error loss. We focus on the family of generalized models generated by the exponential family of response distributions (see Chapter 8). The most popular and relevant in this class is logistic regression, where we are interested in modeling $\mu(x) = \Pr(Y = 1|X = x)$ for a Bernoulli response variable.

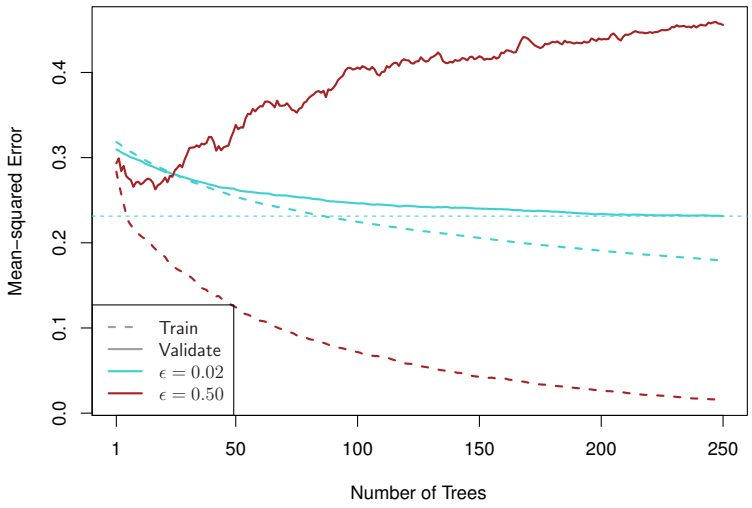

Figure 17.10 Boosted $d = 3$ models with different shrinkage parameters, fit to a subset of the ALS data. The solid curves are validation errors, the dashed curves training errors, with red for $\epsilon = 0.5$ and blue for $\epsilon = 0.02$. With $\epsilon = 0.5$, the training error drops rapidly with the number of trees, but the validation error starts to increase rapidly after an initial decrease. With $\epsilon = 0.02$ (25 times smaller), the training error drops more slowly. The validation error also drops more slowly, but reaches a lower minimum (the horizontal dotted line) than the $\epsilon = 0.5$ case. In this case, the slower learning has paid off.

The idea is to fit a model of the form

$$\lambda(x) = G_B(x) = \sum_{b=1}^{B} g_b(x; \gamma_b), \tag{17.8}$$

where $\lambda(x)$ is the natural parameter in the conditional distribution of $Y|X = x$, and the $g_b(x; \gamma_b)$ are simple functions such as shallow trees. Here we have indexed each function by a parameter vector γ_b; for trees these would capture the identity of the split variables, their split values, and the constants in the terminal nodes. In the case of the Bernoulli response, we have

$$\lambda(x) = \log\left(\frac{\Pr(Y = 1|X = x)}{\Pr(Y = 0|X = x)}\right), \tag{17.9}$$

the logit link function that relates the mean to the natural parameter. In general, if $\mu(x) = \mathrm{E}(Y|X = x)$ is the conditional mean, we have $\eta[\mu(x)] = \lambda(x)$, where η is the monotone *link function*.

Algorithm 17.3 outlines a general strategy for building a model by forward stagewise fitting. L is the loss function, such as the negative log-likelihood for Bernoulli responses, or squared-error for Gaussian responses. Although we are thinking of trees for the simple functions $g(x;\gamma)$, the ideas generalize. This algorithm is easier to state than to implement. For

Algorithm 17.3 GENERALIZED BOOSTING BY FORWARD-STAGEWISE FITTING

1 Define the class of functions $g(x;\gamma)$. Start with $\hat{G}_0(x) = 0$, and set B and the shrinkage parameter $\epsilon > 0$.

2 For $b = 1, \ldots, B$ repeat the following steps.

(a) Solve

$$\hat{\gamma}_b = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, \widehat{G}_{b-1}(x_i) + g(x_i;\gamma)\right)$$

(b) Update $\widehat{G}_b(x) = \widehat{G}_{b-1}(x) + \hat{g}_b(x)$, with $\hat{g}_b(x) = \epsilon \cdot g(x;\hat{\gamma}_b)$.

3 Return the sequence $\widehat{G}_b(x),\ b = 1, \ldots, B$.

squared-error loss, at each step we need to solve

$$\underset{\gamma}{\text{minimize}} \sum_{i=1}^{n} (r_i - g(x_i;\gamma))^2, \tag{17.10}$$

with $r_i = y_i - \widehat{G}_{b-1}(x_i),\ i = 1, \ldots, n$. If $g(\cdot;\gamma)$ represents a depth-d tree, (17.10) is still difficult to solve. But here we can resort to the usual greedy heuristic, and grow a depth-d tree to the residuals by the usual top-down splitting, as in step 2(a) of Algorithm 17.2. Hence in this case, we have exactly the squared-error boosting Algorithm 17.2. For more general loss functions, we rely on one more heuristic for solving step 2(a), inspired by gradient descent. Algorithm 17.4 gives the details. The idea is to perform functional gradient descent on the loss function, in the n-dimensional space of the fitted vector. However, we want to be able to evaluate our new function everywhere, not just at the n original values x_i. Hence once the (negative) gradient vector has been computed, it is approximated by a depth-d tree (which *can* be evaluated everywhere). Taking a step of length ϵ down

the gradient amounts to adding ϵ times the tree to the current function.[†] [†4] Gradient boosting is quite general, and can be used with any differentiable

Algorithm 17.4 GRADIENT BOOSTING

1 Start with $\hat{G}_0(x) = 0$, and set B and the shrinkage parameter $\epsilon > 0$.
2 For $b = 1, \ldots, B$ repeat the following steps.

(a) Compute the pointwise negative gradient of the loss function at the current fit:

$$r_i = -\frac{\partial L(y_i, \lambda_i)}{\partial \lambda_i}\bigg|_{\lambda_i = \hat{G}_{b-1}(x_i)}, \ 1 = 1, \ldots, n.$$

(b) Approximate the negative gradient by a depth-d tree by solving

$$\underset{\gamma}{\text{minimize}} \sum_{i=1}^{n} (r_i - g(x_i; \gamma))^2 .$$

(c) Update $\hat{G}_b(x) = \hat{G}_{b-1}(x) + \hat{g}_b(x)$, with $\hat{g}_b(x) = \epsilon \cdot g(x; \hat{\gamma}_b)$.

3 Return the sequence $\hat{G}_b(x), \ b = 1, \ldots, B$.

loss function. The R package **gbm** implements Algorithm 17.4 for a variety of loss functions, including squared-error, binomial (Bernoulli), Laplace (ℓ_1 loss), multinomial, and others. Included as well is the partial likelihood for the Cox proportional hazards model (Chapter 9). Figure 17.11 compares the misclassification error of boosting on the **spam** data, with that of random forests and bagging. Since boosting has more tuning parameters, a careful comparison must take these into account. Using the McNemar test we would conclude that boosting and random forest are not significantly different from each other, but both outperform bagging.

17.4 Adaboost: the Original Boosting Algorithm

The original proposal for boosting looked quite different from what we have presented so far. Adaboost was developed for the two-class classification problem, where the response is coded as -1/1. The idea was to fit a sequence of classifiers to modified versions of the training data, where the modifications give more weight to misclassified points. The final classification is by weighted majority vote. The details are rather specific, and are given in Algorithm 17.5. Here we distinguish a classifier $C(x) \in \{-1, 1\}$, which returns a class label, rather than a probability. Algorithm 17.5 gives

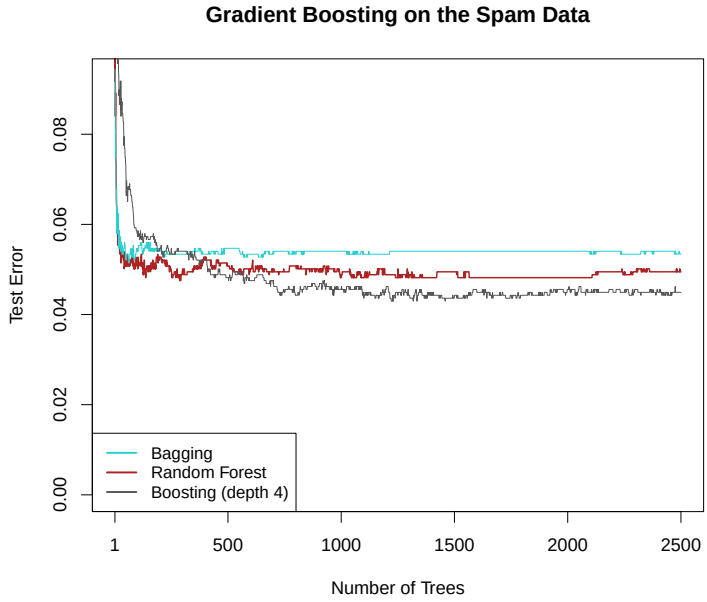

Figure 17.11 Test misclassification for gradient boosting on the `spam` data, compared with a random forest and bagging. Although boosting appears to be better, it requires crossvaldiation or some other means to estimate its tuning parameters, while the random forest is essentially automatic.

the *Adaboost.M1* algorithm. Although the classifier in step 2(a) can be arbitrary, it was intended for *weak learners* such as shallow trees. Steps 2(c)–(d) look mysterious. Its easy to check that, with the reweighted points, the classifier $\hat{c}_b$ just learned would have weighted error 0.5, that of a coin flip. We also notice that, although the individual classifiers $\hat{c}_b(x)$ produce values ± 1, the ensemble $\widehat{G}_b(x)$ takes values in $\mathbb{R}$.

It turns out that the Adaboost Algorithm 17.5 fits a logistic regression model via a version of the general boosting Algorithm 17.3, using an exponential loss function. The functions $\widehat{G}_b(x)$ output in step 3 of Algorithm 17.5 are estimates of (half) the logit function $\lambda(x)$.

To show this, we first motivate the exponential loss, a somewhat unusual choice, and show how it is linked to logistic regression. For a -1/1 response y and function $f(x)$, the exponential loss is defined as $L_E(y, f(x)) = \exp[-yf(x)]$. A simple calculation shows that the solution to the (condi-

Algorithm 17.5 Adaboost

1 Initialize the observation weights $w_i = 1/n,\ i = 1, \ldots, n$.
2 For $b = 1, \ldots, B$ repeat the following steps.
(a) Fit a classifier $\hat{c}_b(x)$ to the training data, using observation weights w_i.
(b) Compute the weighted misclassification error for $\hat{c}_b$:

$$\text{err}_b = \frac{\sum_{i=1}^{n} w_i I[y_i \neq \hat{c}_b(x_i)]}{\sum_{i=1}^{n} w_i}.$$

(c) Compute $\alpha_b = \log[(1 - \text{err}_b)/\text{err}_b]$.
(d) Update the weights $w_i \leftarrow w_i \cdot \exp(\alpha_b \cdot I[y_i \neq c_b(x_i)])$, $i = 1, \ldots, n$.
3 Output the sequence of functions $\widehat{G}_b(x) = \sum_{\ell=1}^{b} \alpha_m \hat{c}_\ell(x)$ and corresponding classifiers $\widehat{C}_b(x) = \text{sign}\left[\widehat{G}_b(x)\right], b = 1, \ldots, B$.

tional) population minimization problem

$$\underset{f(x)}{\text{minimize}}\, \text{E}[e^{-yf(x)} \mid x] \tag{17.11}$$

is given by

$$f(x) = \frac{1}{2} \log\left(\frac{\Pr(y = +1|x)}{\Pr(y = -1|x)}\right). \tag{17.12}$$

Inverting, we get

$$\Pr(y = +1|x) = \frac{e^{f(x)}}{e^{-f(x)} + e^{f(x)}} \text{ and } \Pr(y = -1|x) = \frac{e^{-f(x)}}{e^{-f(x)} + e^{f(x)}}, \tag{17.13}$$

a perfectly reasonable (and symmetric) model for a probability. The quantity $yf(x)$ is known as the margin (see also Chapter 19); if the margin is positive, the classification using $C_f(x) = \text{sign}(f(x))$ is correct for y, else it is incorrect if the margin is negative. The magnitude of $yf(x)$ is proportional to the (signed) distance of x from the classification boundary (exactly for linear models, approximately otherwise). For -1/1 data, we can also write the (negative) binomial log-likelihood in terms of the margin.

Using (17.13) we have

$$\begin{aligned} L_B(y, f(x)) &= -\{I(y=-1)\log \Pr(y=-1|x) \\ &\qquad + I(y=+1)\log \Pr(y=+1|x)\} \\ &= \log\left(1+e^{-2yf(x)}\right). \end{aligned} \tag{17.14}$$

$\mathrm{E}\left[\log\left(1+e^{-2yf(x)}\right) \mid x\right]$ also has population minimizer $f(x)$ equal to half the logit (17.12).[3] Figure 17.12 compares the exponential loss function with this binomial loss. They both asymptote to zero in the right tail—the area of correct classification. In the left tail, the binomial loss asymptotes to a linear function, much less severe than the exponential loss.

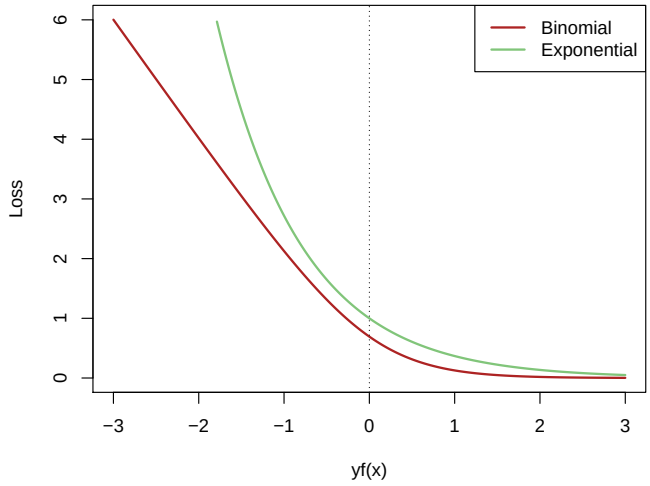

Figure 17.12 Exponential loss used in Adaboost, versus the binomial loss used in the usual logistic regression. Both estimate the logit function. The exponential left tail, which punishes misclassifications, is much more severe than the asymptotically linear tail of the binomial.

The exponential loss simplifies step 2(a) in the gradient boosting Algo-

[3] The half comes from the symmetric representation we use.

rithm 17.3.

$$\begin{aligned}\sum_{i=1}^{n} L_E\left(y_i, \widehat{G}_{b-1}(x_i) + g(x_i;\gamma)\right) &= \sum_{i=1}^{n} \exp[-y_i(\widehat{G}_{b-1}(x_i) + g(x_i;\gamma)] \\ &= \sum_{i=1}^{n} w_i \exp[-y_i g(x_i;\gamma)] \qquad (17.15) \\ &= \sum_{i=1}^{n} w_i L_E\left(y_i, g(x_i;\gamma)\right),\end{aligned}$$

with $w_i = \exp[-y_i \widehat{G}_{b-1}(x_i)]$. This is just a weighted exponential loss with the past history encapsulated in the observation weight w_i (see step 2(a) in Algorithm 17.5). We give some more details in the chapter endnotes on how this reduces to the Adaboost algorithm.[†] †5

The Adaboost algorithm achieves an error rate on the **spam** data comparable to binomial gradient boosting.

17.5 Connections and Extensions

Boosting is a general nonparametric function-fitting algorithm, and shares attributes with a variety of existing methods. Here we relate boosting to two different approaches: generalized additive models and the lasso of Chapter 16.

Generalized Additive Models

Boosting fits additive, low-order interaction models by a forward stagewise strategy. Generalized additive models (GAMs) are a predecessor, a semi-parametric approach toward nonlinear function fitting. A GAM has the form

$$\lambda(x) = \sum_{j=1}^{p} f_j(x_j), \qquad (17.16)$$

where again $\lambda(x) = \eta[\mu(x)]$ is the natural parameter in an exponential family. The attraction of a GAM is that the components are interpretable and can be visualized, and they can move us a big step up from a linear model.

There are many ways to specify and fit additive models. For the f_j, we could use parametric functions (e.g. polynomials), fixed-knot regression splines, or even linear functions for some terms. Less parametric options

are smoothing splines and local regression (see Section 19.8). In the case of squared-error loss (the Gaussian case), there is a natural set of *backfitting* equations for fitting a GAM:

$$\hat{f}_j \leftarrow \mathcal{S}_j(\boldsymbol{y} - \sum_{\ell \neq j} \hat{\boldsymbol{f}}_\ell), \ j = 1, \ldots, p. \tag{17.17}$$

Here $\hat{\boldsymbol{f}}_\ell = [\hat{f}_\ell(x_{1\ell}), \ldots, (\hat{f}_\ell(x_{n\ell})]'$ is the n-vector of fitted values for the current estimate of function f_ℓ. Hence the term in parentheses is a *partial residual*, removing all the current function fits from $\boldsymbol{y}$ except the one about to be updated. $\mathcal{S}_j$ is a smoothing operator derived from variable $\boldsymbol{x}_j$ that gets applied to this residual and delivers the next estimate for function f_ℓ. Backfitting starts with all the functions zero, and then cycles through these equations for $j = 1, 2, \ldots, p, 1, 2, \ldots$ in a block-coordinate fashion, until all the functions stabilize.

The first pass through all the variables is similar to the regression boosting Algorithm 17.2, where each new function takes the residuals from the past fits, and models them using a tree (for $\mathcal{S}_j$). The difference is that boosting never goes back and fixes up past functions, but fits in a forward-stagewise fashion, leaving all past functions alone. Of course, with its adaptive fitting mechanism, boosting can select the same variables as used before, and thereby update that component of the fit. Boosting with stumps (single-split trees, see the discussion on tree depth on 335 in Section 17.2) can hence be seen as an adaptive way for fitting an additive model, that simultaneously performs variable selection and allows for different amounts of smoothing for different variables.

Boosting and the Lasso

In Section 16.7 we drew attention to the close connection between the forward-stagewise fitting of boosting (with shrinkage) and the lasso, via infinitesimal forward-stagewise regression. Here we take this a step further, by using the lasso as a post-processor for boosting (or random forests).

Boosting with shrinkage does a good job in building a prediction model, but at the end of the day can involve a lot of trees. Because of the shrinkage, many of these trees could be similar to each other. The idea here is to use the lasso to select a subset of these trees, reweight them, and hence produce a prediction model with far fewer trees and, one hopes, comparable accuracy. Suppose boosting has produced a sequence of fitted trees $\hat{g}_b(x), \ b = 1, \ldots, B$. We then solve the lasso problem

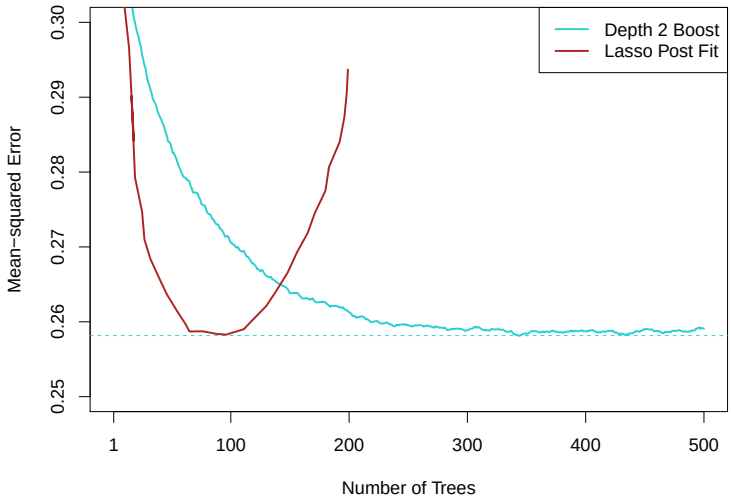

Figure 17.13 Post-processing of the trees produced by boosting on the **ALS** data. Shown is the test prediction error as a function of the number of trees selected by the (nonnegative) lasso. We see that the lasso can do as good a job with one-third the number of trees, although selecting the correct number is critical.

$$\underset{\{\beta_b\}_1^B}{\text{minimize}} \sum_{i=1}^{n} L\left[y_i, \sum_{b=1}^{B} \hat{g}_b(x_i)\beta_b\right] + \lambda \sum_{b=1}^{B} |\beta_b| \tag{17.18}$$

for different values of λ. This model selects some of the trees, and assigns differential weights to them. A reasonable variant is to insist that the weights are nonnegative. Figure 17.13 illustrates this approach on the **ALS** data. Here we could use one-third of the trees. Often the savings are much more dramatic.

17.6 Notes and Details

Random forests and boosting live at the cutting edge of modern prediction methodology. They fit models of breathtaking complexity compared with classical linear regression, or even with standard GLM modeling as practiced in the late twentieth century (Chapter 8). They are routinely used as prediction engines in a wide variety of industrial and scientific applications. For the more cautious, they provide a terrific benchmark for how well a traditional parametrized model is performing: if the random forests

does much better, you probably have some work to do, by including some important interactions and the like.

The regression and classification trees discussed in Chapter 8 (Breiman *et al.*, 1984) took traditional models to a new level, with their ability to adapt to the data, select variables, and so on. But their prediction performance is somewhat lacking, and so they stood the risk of falling by the wayside. With their new use as building blocks in random forests and boosting, they have reasserted themselves as critical elements in the modern toolbox.

Random forests and bagging were introduced by Breiman (2001), and boosting by Schapire (1990) and Freund and Schapire (1996). There has been much discussion on why boosting works (Breiman, 1998; Friedman *et al.*, 2000; Schapire and Freund, 2012); the statistical interpretation given here can also be found in Hastie *et al.* (2009), and led to the gradient boosting algorithm (Friedman, 2001). Adaboost was first described in Freund and Schapire (1997). Hastie *et al.* (2009, Chapter 15) is devoted to random forests. For the examples in this chapter we used the `randomForest` package in `R` (Liaw and Wiener, 2002), and for boosting the `gbm` (Ridgeway, 2005) package. The lasso post-processing idea is due to Friedman and Popescu (2005), which we implemented using `glmnet` (Friedman *et al.*, 2009). Generalized additive models are described in Hastie and Tibshirani (1990).

We now give some particular technical details on topics covered in the chapter.

†1 [p. 327] *Averaging trees.* A maximal-depth tree splits every node until it is *pure*, meaning all the responses are the same. For very large n this might be unreasonable; in practice, one can put a lower bound on the minimum count in a terminal node. We are deliberately vague about the response type in Algorithm 17.1. If it is quantitative, we would fit a regression tree. If it is binary or multilevel qualitative, we would fit a classification tree. In this case at the averaging stage, there are at least two strategies. The original random-forest paper (Breiman, 2001) proposed that each tree should make a classification, and then the ensemble uses a plurality vote. An alternative reasonable strategy is to average the class probabilities produced by the trees; these procedures are identical if the trees are grown to maximal depth.

†2 [p. 330] *Jackknife variance estimate.* The jackknife estimate of variance for a random forest, and the bias-corrected version, is described in Wager *et al.* (2014). The jackknife formula (17.3) is applied to the $B = \infty$ ver-

sion of the random forest, but of course is estimated by plugging in finite B versions of the quantities involved. Replacing $\hat{r}_{\text{rf}}^{(\cdot)}(x_0)$ by its expectation $\hat{r}_{\text{rf}}(x_0)$ is not the problem; its that each of the $\hat{r}_{\text{rf}}^{(i)}(x_0)$ vary about their bootstrap expectations, compounded by the square in expression (17.4). Calculating the bias requires some technical derivations, which can be found in that reference.

They also describe the infinitesimal jackknife estimate of variance, given by

$$\widehat{\text{V}}_{\text{IJ}}(\hat{r}_{\text{rf}}(x_0)) = \sum_{i=1}^{n} \widehat{\text{cov}}_i^2, \tag{17.19}$$

with

$$\widehat{\text{cov}}_i = \widehat{\text{cov}}(w^*, \hat{r}_*(x_0)) = \frac{1}{B}\sum_{b=1}^{B}(w^*_{b\,i} - 1)(\hat{r}_b(x_0) - \hat{r}_{\text{rf}}(x_0)), \tag{17.20}$$

as discussed in Chapter 20. It too has a bias-corrected version, given by

$$\widehat{\text{V}}^u_{\text{IJ}}(\hat{r}_{\text{rf}}(x_0)) = \widehat{\text{V}}_{\text{IJ}}(\hat{r}_{\text{rf}}(x_0)) - \frac{n}{B}\hat{v}(x_0), \tag{17.21}$$

similar to (17.5).

†3 [p. 334] *The* **ALS** *data.* These data were kindly provided by Lester Mackey and Lilly Fang, who won the DREAM challenge prediction prize in 2012 (Kuffner *et al.*, 2015). It includes some additional variables created by them. Their winning entry used Bayesian trees, not too different from random forests.

†4 [p. 341] *Gradient-boosting details.* In Friedman's gradient-boosting algorithm (Hastie *et al.*, 2009, Chapter 10, for example), a further refinement is implemented. The tree in step 2(b) of Algorithm 17.4 is used to define the structure (split variables and splits), but the values in the terminal nodes are left to be updated. We can think of partitioning the parameters $\gamma = (\gamma_s, \gamma_t)$, and then represent the tree as $g(x;\gamma) = T(x;\gamma_s)'\gamma_t$. Here $T(x;\gamma_s)$ is a vector of $d+1$ binary basis functions that indicate the terminal node reached by input x, and γ_t are the $d+1$ values of the terminal nodes of the tree. We learn $\hat{\gamma}_s$ by approximating the gradient in step 2(b) by a tree, and then (re-)learn the terminal-node parameters $\hat{\gamma}_t$ by solving the optimization problem

$$\underset{\gamma_t}{\text{minimize}} \sum_{i=1}^{n} L\left(y_i, \hat{G}_{b-1}(x_i) + T(x_i;\hat{\gamma}_s)'\gamma_t\right). \tag{17.22}$$

Solving (17.22) amounts to fitting a simple GLM with an *offset*.

†5 [p. 345] *Adaboost and gradient boosting.* Hastie *et al.* (2009, Chapter 10) derive Adaboost as an instance of Algorithm 17.3. One detail is that the trees $g(x;\gamma)$ are replaced by a simplified scaled classifier $\alpha \cdot c(x;\gamma')$. Hence, from (17.15), in step 2(a) of Algorithm 17.3 we need to solve

$$\underset{\alpha,\gamma'}{\text{minimize}} \sum_{i=1}^{n} w_i \exp[-y_i \alpha c(x_i;\gamma')]. \tag{17.23}$$

The derivation goes on to show that

- minimizing (17.23) for any value of $\alpha > 0$ can be achieved by fitting a classification tree $c(x;\hat{\gamma}')$ to minimize the weighted misclassification error
$$\sum_{i=1}^{n} w_i I[y_i \neq c(x_i,\gamma')];$$
- given $c(x;\hat{\gamma}')$, α is estimated as in step 2(c) of Algorithm 17.5 (and is non-negative);
- the weight-update scheme in step 2(d) of Algorithm 17.5 corresponds exactly to the weights as computed in (17.15).