CHAPTER 12

Boosting

The following idea may have occurred to you after reading the chapter on regression. Imagine you have a regression that makes errors. You could try to produce a second regression that fixes those errors. You may have dismissed this idea, though, because if one uses only linear regressions trained using least squares, it's hard to see how to build a second regression that fixes the first regression's errors.

Many people have a similar intuition about classification. Imagine you have trained a classifier. You could try to train a second classifier to fix errors made by the first. There doesn't seem to be any reason to stop there, and you might try and train a third classifier to fix errors made by the first and the second, and so on. The details take some work, as you would expect. It isn't enough to just fix errors. You need some procedure to decide what the overall prediction of the system of classifiers is, and you need some way to be confident that the overall prediction will be better than the prediction produced by the initial classifier.

It is fruitful to think about correcting earlier predictions with a new set. I will start with a simple version that can be used to avoid linear algebra problems for least squares linear regression. Each regression will see different features from the previous regressions, so there is a prospect of improving the model. This approach extends easily to cover regressions that use something other than a linear function to predict values (I use a tree as an example).

Getting the best out of the idea requires some generalization. Regression builds a function that accepts features and produces predictions. So does classification. A regressor accepts features and produces numbers (or, sometimes, more complicated objects like vectors or trees, though we haven't talked about that much). A classifier accepts features and produces labels. We generalize, and call any function that accepts a feature and produces predictions a **predictor**. Predictors are trained using losses, and the main difference between a classifier and a regressor is the loss used to train the predictor.

We will build an optimal predictor as a sum of less ambitious predictors, often known as **weak learners**. We will build the optimal predictor incrementally using a greedy method, where we construct a new weak learner to improve over the sum of all previous weak learners, without adjusting the old ones. This process is called **boosting**. Setting this up will take some work, but it is worth doing, as we then have a framework makes it possible to boost a very wide range of classifiers and regressors. Boosting is particularly attractive when one has a weak learner that is simple and easy to train; one can often produce a predictor that is very accurate and can be evaluated very fast.

12.1 GREEDY AND STAGEWISE METHODS FOR REGRESSION

The place to start is linear regression. We will assume that we have so many features that we cannot solve the linear algebra problem resulting from least-squares regression. Recall to build a linear regression of y against some high dimensional vector $\mathbf{x}$ (using the notation of chapter 10), we will need to solve

$$\mathcal{X}^T\mathcal{X}\beta = \mathcal{X}^T\mathbf{y}$$

but this might be hard to do if $\mathcal{X}$ was really big. You're unlikely to see many problems where this really occurs, because modern software and hardware are very efficient at dealing with even enormous linear algebra problems. However, thinking about this case is very helpful. What we could do is choose some subset of the features to work with, to obtain a smaller problem, solve that, and go again.

12.1.1 Example: Greedy Stagewise Linear Regression

Write $\mathbf{x}^{(i)}$ for the i'th subset of features. For the moment, we will assume this is a small set of features and worry about how to choose the set later. Write $\mathcal{X}^{(i)}$ for the matrix constructed out of these features, etc. Now we regress $\mathbf{y}$ against $\mathcal{X}^{(1)}$. This chooses the $\hat{\beta}^{(1)}$ that minimizes the squared length of the residual vector

$$\mathbf{e}^{(1)} = \mathbf{y} - \mathcal{X}^{(1)}\hat{\beta}^{(1)}.$$

We obtain this $\hat{\beta}^{(1)}$ by solving

$$\left(\mathcal{X}^{(1)}\right)^T\mathcal{X}^{(1)}\hat{\beta}^{(1)} = \left(\mathcal{X}^{(1)}\right)^T\mathbf{y}.$$

We would now like to obtain an improved predictor by using more features in some way. We will build an improved predictor by adding some linear function of these new features to the original predictor. There are some important constraints. The improved predictor should correct errors made by the original predictor, but we do not want to change the original predictor. One reason not to is that we are building a second linear function to avoid solving a large linear algebra problem. Adjusting the original predictor at the same time will land us back where we started (with a large linear algebra problem).

To build the improved predictor, form $\mathcal{X}^{(2)}$ out of these features. The improved predictor will be

$$\mathcal{X}^{(1)}\hat{\beta^{(1)}} + \mathcal{X}^{(2)}\beta^{(2)}.$$

We do *not* want to change $\hat{\beta^{(1)}}$ and so we want to minimize

$$\left(\mathbf{y} - \left[\mathcal{X}^{(1)}\hat{\beta^{(1)}} + \mathcal{X}^{(2)}\beta^{(2)}\right]\right)^T\left(\mathbf{y} - \left[\mathcal{X}^{(1)}\hat{\beta^{(1)}} + \mathcal{X}^{(2)}\beta^{(2)}\right]\right)$$

as a function of $\beta^{(2)}$ alone. To simplify, write

$$\begin{aligned}\mathbf{e}^{(2)} &= \left(\mathbf{y} - \left[\mathcal{X}^{(1)}\beta^{(1)} + \mathcal{X}^{(2)}\beta^{(2)}\right]\right)\\ &= \mathbf{e}^1 - \mathcal{X}^{(2)}\beta^{(2)}\end{aligned}$$

and we must choose $\beta^{(2)}$ to minimize

$$\left(\mathbf{e}^1 - \mathcal{X}^{(2)}\beta^{(2)}\right)^T \left(\mathbf{e}^1 - \mathcal{X}^{(2)}\beta^{(2)}\right).$$

This follows a familiar recipe. We obtain this $\hat{\beta}^{(1)}$ by solving

$$\left(\mathcal{X}^{(2)}\right)^T \mathcal{X}^{(2)}\hat{\beta}^{(2)} = \left(\mathcal{X}^{(2)}\right)^T \mathbf{e}^{(1)}.$$

Notice this is a linear regression of $\mathbf{e}^{(1)}$ against the features in $\mathcal{X}^{(2)}$. This is extremely convenient. The linear function that improves the original predictor is obtained using the same procedure (linear regression) as the original predictor. We just regress the residual (rather than $\mathbf{y}$) against the new features.

The new linear function is not guaranteed to make the regression better, but it will not make the regression worse. Because our choice of $\hat{\beta}^{(2)}$ minimizes the squared length of $\mathbf{e}^{(2)}$, we have that

$$\mathbf{e}^{(2)^T}\mathbf{e}^{(2)} \leq \mathbf{e}^{(1)^T}\mathbf{e}^{(1)}$$

with equality only if $\mathcal{X}^{(2)}\hat{\beta}^{(2)} = \mathbf{0}$. In turn, the second round did not make the residual worse. If the features in $\mathcal{X}^{(2)}$ aren't all the same as those in $\mathcal{X}^{(1)}$, it is very likely to have made the residual better.

Extending all this to an R'th round is just a matter of notation; you can write an iteration with $\mathbf{e}^{(0)} = \mathbf{y}$. Then you regress $\mathbf{e}^{(j-1)}$ against the features in $\mathcal{X}^{(j)}$ to get $\hat{\beta}^{(j)}$, and

$$\mathbf{e}^{(j)} = \mathbf{e}^{(j-1)} - \mathcal{X}^{(j)}\hat{\beta}^{(j)} = \mathbf{e}^{(0)} - \sum_{u=1}^{j} \mathcal{X}^{(u)}\hat{\beta}^{(u)}.$$

The residual never gets bigger (at least if your arithmetic is exact). This procedure is referred to as **greedy stagewise linear regression**. It's stagewise, because we build up the model in steps. It's greedy, because we do not adjust our estimate of $\hat{\beta}^{(1)}, \ldots, \hat{\beta}^{(j-1)}$ when we compute $\hat{\beta}^{(j)}$, etc.

This process won't work for a linear regression when we use all the features in $\mathcal{X}^{(1)}$. It's worth understanding why. Consider the first step. We will choose β to minimize $(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta)$. But there's a closed form solution for this β (which is $\hat{\beta} = (\mathcal{X}^T\mathcal{X})^{-1}\mathcal{X}^T\mathbf{y}$; remind yourself if you've forgotten by referring to chapter 10), and this is a global minimizer. So to minimize

$$\left(\left[\mathbf{y} - \mathcal{X}\hat{\beta}\right] - \mathcal{X}\gamma\right)^T \left(\left[\mathbf{y} - \mathcal{X}\hat{\beta}\right] - \mathcal{X}\gamma\right)$$

by choice of γ, we'd have to have $\mathcal{X}\gamma = 0$, meaning that the residual wouldn't improve.

At this point, greedy stagewise linear regression may look like nothing more than a method of getting otherwise unruly linear algebra under control. But it's actually a model recipe, exposed in the box below. This recipe admits very substantial generalization.

Procedure: 12.1 *Greedy stagewise linear regression*

We choose to minimize the squared length of the residual vector. Write

$$\mathcal{L}^{(j)}(\beta) = \left\| (\mathbf{e}^{(j-1)} - \mathcal{X}^{(j)}\beta) \right\|^2.$$

Start with $\mathbf{e}^{(0)} = \mathbf{y}$ and $j = 1$. Now iterate:

- choose a set of features to form $\mathcal{X}^{(j)}$;
- construct $\hat{\beta}^{(j)}$ by minimizing $\mathcal{L}^{(j)}(\beta)$; do so by solving the linear system
$$\left(\mathcal{X}^{(j)}\right)^T \mathcal{X}^{(j)}\hat{\beta}^{(j)} = \left(\mathcal{X}^{(j)}\right)^T \mathbf{e}^{(j-1)}.$$
- form $\mathbf{e}^{(j)} = \mathbf{e}^{(j-1)} - \mathcal{X}^{(j)}\hat{\beta}^{(j)}$;
- increment j to be $j + 1$.

The prediction for the training data is

$$\sum_j \mathcal{X}^{(j)}\hat{\beta}^{(j)}.$$

Write $\mathbf{x}$ for a test point for which you want a prediction, and $\mathbf{x}^{(j)}$ for the j'th set of features from that test point. The prediction for $\mathbf{x}$ is

$$\sum_j \mathbf{x}^{(j)}\hat{\beta}^{(j)}.$$

It is natural to choose the features at random, as more complicated strategies might be hard to execute. There isn't an obvious criterion for stopping, but looking at a plot of the test error with iterations will be informative.

Remember this: *A linear regression that has a very large number of features could result in a linear algebra problem too large to be conveniently solved. In this case, there is an important strategy. Choose a small subset of features and fit a model. Now choose a small random subset of features and use them to fit a regression that predicts the residual of the current model. Add the regression to the current model, and go again. This recipe can be aggressively generalized, and is extremely powerful.*

12.1.2 Regression Trees

I haven't seen the recipe in box 12.1 used much for linear regressions, but as a model recipe it's extremely informative. It becomes much more interesting when applied to regressions that don't use linear functions. It is straightforward to coopt machinery we saw in the context of classification to solve regression problems, too. A **regression tree** is defined by analogy with a decision tree (section 2.2). One builds a tree by splitting on coordinates, so each leaf represents a cell in space where the coordinates satisfy some inequalities. For the simplest regression tree, each leaf contains a single value representing the value the predictor takes in that cell (one can place other prediction methods in the leaves; we won't bother). The splitting process parallels the one we used for classification, but now we can use the error in the regression to choose the split instead of the information gain.

Worked example 12.1 *Regressing prawn scores against location*

At http://www.statsci.org/data/oz/reef.html you will find a dataset describing prawns caught between the coast of northern Queensland and the Great Barrier Reef. There is a description of the dataset at that URL; the data was collected and analyzed by Poiner et al, cited at that URL. Build a regression tree predicting prawn score 1 (whatever that is!) against latitude and longitude using this dataset.

Solution: This data set is nice, because it is easy to visualize interesting predictors. Figure 12.1 shows a 3D scatter plot of score 1 against latitude and longitude. There are good packages for building such trees (I used R's `rpart`). Figure 12.1 shows a regression tree fitted with that package, as an image. This makes it easy to visualize the function. The darkest points are the smallest values, and the lightest points are the largest. You can see what the tree does: carve space into boxes, then predict a constant inside each.

Remember this: *A regression tree is like a classification tree, but a regression tree stores values rather than labels in the leaves. The tree is fitted using a procedure quite like fitting a classification tree.*

12.1.3 Greedy Stagewise Regression with Trees

We wish to regress y against $\mathbf{x}$ using many regression trees. Write $f(\mathbf{x}; \theta^{(j)})$ for a regression tree that accepts $\mathbf{x}$ and produces a prediction (here $\theta^{(j)}$ are parameters internal to the tree: where to split; what is in the leaves; and so on). Write the

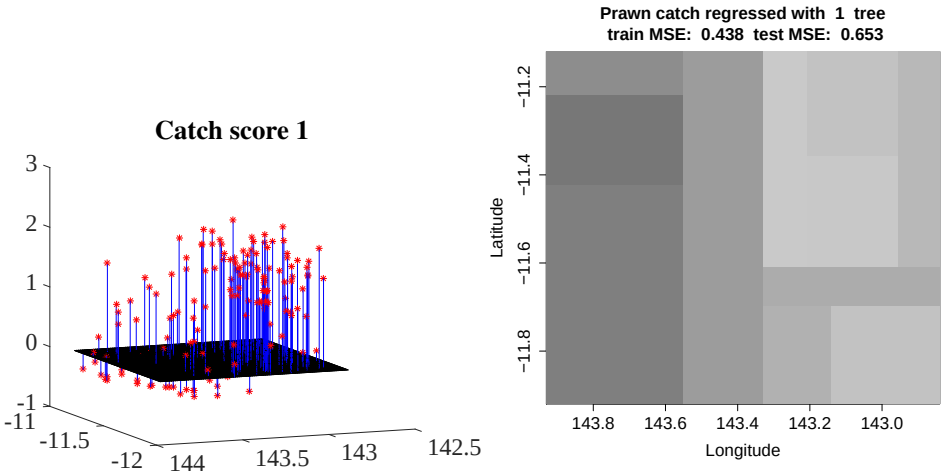

FIGURE 12.1: *On the* **left**, *a 3D scatter plot of score 1 of the prawn trawls data from http://www.statsci.org/data/oz/reef.html, plotted as a function of latitude and longitude. On the* **right**, *a regression using a single regression tree, to help visualize the kind of predictor these trees produce. You can see what the tree does: carve space into boxes, then predict a constant inside each. The intensity scale is chosen so that the range is symmetric; because there are no small negative numbers, there are no very dark boxes. The odd axes (horizontal runs from bigger to smaller) are chosen so that you can register the left to the right by eye.*

regression as

$$F(\mathbf{x}) = \sum_j f(\mathbf{x}; \theta^{(j)})$$

where there might be quite a lot of trees indexed by j. Now we must fit this regression model to the data by choosing values for each $\theta^{(j)}$. We could fit the model by minimizing

$$\sum_i (y_i - F(\mathbf{x}_i))^2$$

as a function of all the $\theta^{(j)}$'s. This is unattractive, because it isn't clear how to solve this optimization problem.

The recipe for greedy stagewise linear regression applies here, with very little change. The big difference is there is no need to choose a new subset of independent variables each time (the regression tree fitting procedure will do this). The recipe looks (roughly) like this: regress y against $\mathbf{x}$ using a regression tree; construct the residuals; and regress the residuals against $\mathbf{x}$; and repeat until some termination criterion.

In notation, start with a $F^{(0)} = 0$ (the initial model) and $j = 0$. Write $e_i^{(j)}$ for the residual at the j'th round and the i'th example. Set $e_i^{(0)} = y_i$. Now iterate the following steps:

- Choose $\hat{\theta}^{(j)}$ to minimize

$$\sum_i \left(e_i^{(j-1)} - f(\mathbf{x}_i; \theta)\right)^2$$

  as a function of θ using some procedure to fit regression trees.
- Set

$$e_i^{(j)} = e_i^{(j-1)} - f(\mathbf{x}_i; \hat{\theta}^{(j)}).$$

- Increment j to $j + 1$.

This is sometimes referred to as **greedy stagewise regression**. Notice that there is no particular reason to stop, unless (a) the residual is zero at all data points or (b) for some reason, it is clear that no future progress would be possible. For reference, I have put this procedure in a box below.

Worked example 12.2 *Greedy stagewise regression for prawns*

Construct a stagewise regression of score 1 against latitude and longitude, using the prawn trawls dataset from http://www.statsci.org/data/oz/reef.html. Use a regression tree.

Solution: There are good packages for building regression trees (I used R's `rpart`). Stagewise regression is straightforward. I started with a current prediction of zero. Then I iterated: form the current residual (score 1 - current prediction); regress that against latitude and longitude; then update the current residual. Figures 12.2 and 12.3 show the result. For this example, I used a function of two dimensions so I could plot the regression function in a straightforward way. It's easy to visualize a regression tree in 2D. The root node of the tree splits the plane in half, usually with an axis aligned line. Then each node splits its parent into two pieces, so each leaf is a rectangular cell on the plane (which might stretch to infinity). The value is constant in each leaf. You can't make a smooth predictor out of such trees, but the regressions are quite good (Figure 12.3).

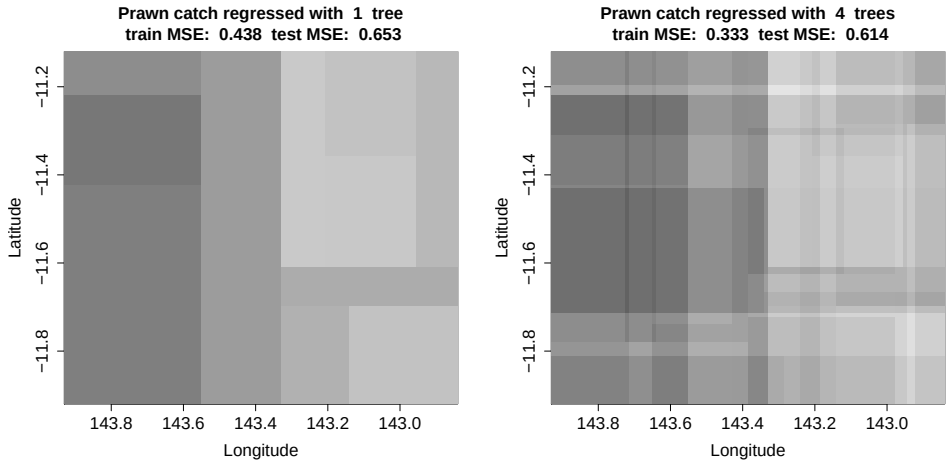

FIGURE 12.2: *Score 1 of the prawn trawls data from* http://www.statsci.org/data/oz/reef.html*, regressed against latitude and longitude (I did not use depth, also in that dataset; this means I could plot the regression easily). The axes (horizontal runs from bigger to smaller) are chosen so that you can register with the plot in figure 12.1 by eye. The intensity scale is chosen so that the range is symmetric; because there are no small negative numbers, there are no very dark boxes. The figure shows results using 1 and 4 trees. Notice the model gets more complex as we add trees. Further stages appear in figure 12.3, which uses the same intensity scale.*

Procedure: 12.2 *Greedy stagewise regression with trees*

Write $f(\mathbf{x};\theta)$ for a regression tree, where θ encodes internal parameters (where to split, thresholds, and so on). We will build a regression that is a sum of trees, so the regression is

$$F(\mathbf{x};\theta) = \sum_j f(\mathbf{x};\theta^{(j)}).$$

We choose to minimize the squared length of the residual vector, so write

$$\mathcal{L}^{(j)}(\theta) = \sum_i \left(e_i^{(j-1)} - f(\mathbf{x}_i;\theta)\right)^2.$$

Start with $e_i^{(0)} = y_i$ and $j = 0$. Now iterate:

- construct $\hat{\theta}^{(j)}$ by minimizing $\mathcal{L}^{(j)}(\theta)$ using regression tree software (which should give you an approximate minimizer);
- form $e_i^{(j)} = e_i^{(j-1)} - f(\mathbf{x}_i;\hat{\theta}^{(j)})$;
- increment j to $j+1$.

The prediction for a data item $\mathbf{x}$ is

$$\sum_j f(\mathbf{x};\hat{\theta}^{(j)})$$

There isn't an obvious criterion for stopping, but looking at a plot of the test error with iterations will be informative.

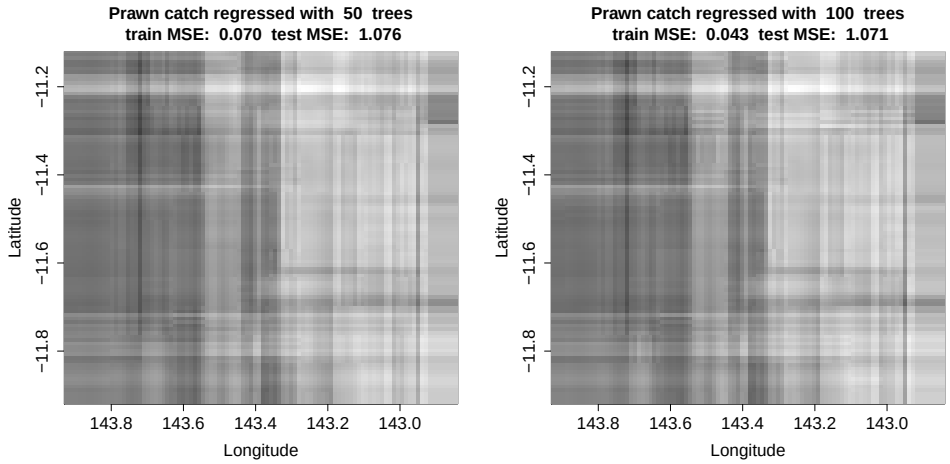

FIGURE 12.3: *Score 1 of the prawn trawls data from* http://www.statsci.org/data/oz/reef.html*, regressed against latitude and longitude (I did not use depth, also in that dataset; this means I could plot the regression easily). The axes (horizontal runs from bigger to smaller) are chosen so that you can register with the plot in figure 12.1 by eye. The intensity scale is chosen so that the range is symmetric; because there are no small negative numbers, there are no very dark boxes. The figure shows results of a greedy stagewise regression using regression trees using 50 and 100 trees. Notice that both train and test error go down, and the model gets more complex as we add trees.*

None of this would be helpful if the regression using trees $1 \dots j$ is worse than the regression using trees $1 \dots j-1$. Here is an argument that establishes that greedy stagewise regression should make progress in the training error. Assume that, if there is any tree that reduces the residual, the software will find one such tree; if not, it will return a tree that is a single leaf containing 0. Then $\left\|\mathbf{e}^{(j)}\right\|^2 \leq \left\|\mathbf{e}^{(j-1)}\right\|^2$, because the tree was chosen to minimize $\left\|\mathbf{e}^{(j)}\right\|^2 = \mathcal{L}^{(j)}(\theta) = \left\|(\mathbf{e}^{(j-1)} - f(\mathbf{x};\theta))\right\|$. If the tree is successful at minimizing this expression, the error will not go up.

In practice, greedy stagewise regression is well-behaved. One could reasonably fear overfitting. Perhaps only the training error goes down as you add trees, but the test error might go up. This can happen, but it tends not to happen (see the examples). We can't go into the reasons here (and they have some component of mystery, anyhow).

Remember this: *It is difficult to fit a weighted sum of regression trees to data directly. Greedy stagewise methods offer a straightforward procedure. Fit a tree to the data to obtain an initial model. Now repeat: Fit a tree to predict the residual of the current model; add that tree to the current model. Stop by looking at validation error.*

12.2 BOOSTING A CLASSIFIER

The recipes I have given above are manifestations of a general approach. This approach applies to both regression and classification. The recipes seem more natural in the context of regression (which is why I did those versions first). But in both regression and classification we are trying to build a **predictor** – a function that accepts features and reports either a number (regression) or a label (classification). Notice we can encode the label as a number, meaning we could classify with regression machinery. In particular, we have some function $F(\mathbf{x})$, where $\mathbf{x}$. For both regression and classification, we apply F to example $\mathbf{x}$ to obtain a prediction. The regressor or classifier is learned by choosing a function that gets good behavior on a training set. This notation is at a fairly high level of abstraction (so, for example, the procedure we've used in classification where we take the sign of some function is represented by F).

12.2.1 The Loss

In early chapters, it seemed as though we used different kinds of predictor for classification and regression. But you might have noticed that the predictor used for linear support vector machines bore a strong similarity to the predictor used for linear regression, though we trained these two in quite different ways. There are many kinds of predictor – linear functions; trees; and so on. We now take the view that the kind of predictor you use is just a matter of convenience (what package you have available; what math you feel like doing; etc.). Once you know what kind of predictor you will use, you must choose the parameters of that predictor. In this new view, the really important difference between classification and regression is the **loss** that you use to choose these parameters. The loss is the cost function used to evaluate errors, and so to train the predictor. Training a classifier involves using a loss that penalizes errors in class prediction in some way, and training a regressor means using a loss that penalizes prediction errors.

The **empirical loss** is the average loss on the training set. Different predictors F produce different losses at different examples, so the loss depends on the predictor F. Notice the kind of predictor isn't what's important; instead, the loss scores the difference between what a predictor produced and what it should have produced. Now write $\mathcal{L}(F)$ for this empirical loss. There are many plausible losses that apply to different prediction problems. Here are some examples:

- For least squares regression, we minimized the least squares error:

$$\mathcal{L}_{ls}(F) = \frac{1}{N}\sum_i \left(y_i - F(\mathbf{x}_i)\right)^2$$

  (though the $1/N$ term sometimes was dropped as irrelevant; section 10.2.2).
- For a linear SVM, we minimized the hinge loss:

$$\mathcal{L}_h(F) = \frac{1}{N}\sum_i \max(0, 1 - y_i F(\mathbf{x}_i))$$

  (assuming that labels are 1 or -1; section 2.1.1).
- For logistic regression, we minimized the logistic loss:

$$\mathcal{L}_{lr}(F) = \frac{1}{N}\sum_i \left[\log\left(e^{\frac{-(y_i+1)}{2}F(\mathbf{x}_i)} + e^{\frac{1-y_i}{2}F(\mathbf{x}_i)}\right)\right]$$

  (again, assuming that labels are 1 or -1; section 11.3.1).

We construct a loss by taking the average over the training data of a **pointwise loss** – a function ℓ that accepts three arguments: a y-value, a vector $\mathbf{x}$, and a prediction $F(\mathbf{x})$. This average is an estimate of the expected value of that pointwise loss over all data.

- For least squares regression,

$$\ell_s(y, \mathbf{x}, F) = \left(y - F(\mathbf{x})\right)^2.$$

- For a linear SVM,

$$\ell_h(y, \mathbf{x}, F) = \max(0, 1 - yF(\mathbf{x})).$$

- For logistic regression,

$$\ell_r(y, \mathbf{x}, F) = \left[\log\left(e^{\frac{-(y+1)}{2}F(\mathbf{x})} + e^{\frac{1-y}{2}F(\mathbf{x})}\right)\right].$$

We often used a regularizer with these losses. It is quite common in boosting to ignore this regularization term, for reasons I will explain below.

Remember this: *Models are predictors that accept a vector and predict some value. All our models are scored using an average of a pointwise loss function that compares the prediction at each data point with the training value at that point. The important difference between classification and regression is the pointwise loss function that is used.*

12.2.2 Recipe: Stagewise Reduction of Loss

We have used a predictor that was a sum of weak learners (equivalently, individual linear regressions; regression trees). Now generalize by noticing that scaling each weak learner could possibly produce a better result. So our predictor is

$$F(\mathbf{x};\theta,\mathbf{a}) = \sum_j a_j f(\mathbf{x};\theta^{(j)})$$

where a_j is the scaling for each weak learner.

Assume we have some F, and want to compute a weak learner that improves it. Whatever the particular choice of loss $\mathcal{L}$, we need to minimize

$$\frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F(\mathbf{x}_i) + a_j f(\mathbf{x}_i;\theta^{(j)})).$$

For most reasonable choices of loss, we can differentiate ℓ and we write

$$\left.\frac{\partial \ell}{\partial F}\right|_i$$

to mean the partial derivative of that function with respect to the F argument, evaluated at the point $(y_i, \mathbf{x}_i, F(\mathbf{x}_i))$. Then a Taylor series gives us

$$\begin{aligned}\frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F(\mathbf{x}_i) + a_j f(\mathbf{x}_i;\theta^{(j)})) \quad &\approx \quad \frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F(\mathbf{x}_i)) + \\ &\quad a_j \frac{1}{N}\sum_i \left[\left(\left.\frac{\partial \ell}{\partial F}\right|_i\right) f(\mathbf{x}_i;\theta^{(j)})\right].\end{aligned}$$

In turn, this means that we can minimize by finding parameters $\hat{\theta}^{(j)}$ such that

$$\frac{1}{N}\sum_i \left(\left.\frac{\partial \ell}{\partial F}\right|_i\right) f(\mathbf{x}_i;\hat{\theta}^{(j)})$$

is negative. This predictor should cause the loss to go down, at least for small values of a_j. Now assume we have chosen an appropriate predictor, represented by $\hat{\theta}^{(j)}$ (the estimate of the predictor's parameters). Then we can obtain a_j by minimizing

$$\Phi(a_j) = \frac{1}{N}\sum_i \ell(y_i, \mathbf{x}_i, F(\mathbf{x}_i) + a_j f(\mathbf{x}_i;\hat{\theta}^{(j)}))$$

which is a one-dimensional problem (remember, F and $\hat{\theta}^{(j)}$ are known; only a_j is unknown).

This is quite a special optimization problem. It is one-dimensional (which simplifies many important aspects of optimization). Furthermore, from the Taylor series argument, we expect that the best choice of a_j is greater than zero. A problem with these properties is known as a **line search** problem, and there are strong and effective procedures in any reasonable optimization package for line search problems. You could use a line search method from an optimization package, or just minimize

this function with an optimization package, which should recognize it as line search. The overall recipe, which is extremely general, is known as **gradient boost**; I have put it in a box, below.

Procedure: 12.3 *Gradient boost*

We wish to choose a predictor F that minimizes a loss

$$\mathcal{L}(F) = \frac{1}{N}\sum_j \ell(y_j, \mathbf{x}_j, F).$$

We will do so iteratively by searching for a predictor of the form

$$F(\mathbf{x};\theta) = \sum_j \alpha_j f(\mathbf{x};\theta^{(u)}).$$

Start with $F = 0$ and $j = 0$. Now iterate:

- form a set of weights, one per example, where

$$w_i^{(j)} = \left.\frac{\partial \ell}{\partial F}\right|_i$$

  (this means the partial derivative of $\ell(y, \mathbf{x}, F)$ with respect to the F argument, evaluated at the point $(y_i, \mathbf{x}_i, F(\mathbf{x}_i))$);
- choose $\hat{\theta}^{(j)}$ (and so the predictor f) so that

$$\sum_i w_i^{(j)} f(\mathbf{x}_i;\hat{\theta}^{(j)})$$

  is negative;
- now form $\Phi(a_j) = \mathcal{L}(F + a_j f(\cdot;\hat{\theta}^{(j)}))$ and search for the best value of a_j using a line search method.

The prediction for any data item $\mathbf{x}$ is

$$\sum_j \alpha_j f(\mathbf{x};\hat{\theta}^{(j)}).$$

There isn't an obvious criterion for stopping, but looking at a plot of the test error with iterations will be informative.

The important problem here is finding parameters $\hat{\theta}^{(j)}$ such that

$$\sum_i w_i^{(j)} f(\mathbf{x}_i;\hat{\theta}^{(j)})$$

is negative. For some predictors, this can be done in a straightforward way. For others, this problem can be rearranged into a regression problem. We will do an example of each case.

Remember this: *Gradient boosting builds a sum of predictors using a greedy stagewise method. Fit a predictor to the data to obtain an initial model. Now repeat: Compute the appropriate weight at each data point; fit a predictor using these weights; search for the best weight with which to add this predictor to the current model; and add the weighted predictor to the current model. Stop by looking at validation error. The weight is a partial derivative of the loss with respect to the predictor, evaluated at the current value of the predictor.*

12.2.3 Example: Boosting Decision Stumps

The name "weak learner" comes from the considerable body of theory covering when and how boosting should work. An important fact from that theory is that the predictor $f(\cdot; \hat{\theta}^{(j)})$ needs only to be a descent direction for the loss — i.e. we need to ensure that adding some positive amount of $f(\cdot; \hat{\theta}^{(j)})$ to the prediction will result in an improvement in the loss. This is a very weak constraint in the two-class classification case (it boils down to requiring that the learner can do slightly better than a 50% error rate on a weighted version of the dataset), so that it is reasonable to use quite a simple classifier for the predictor.

One very natural classifier is a **decision stump**, which tests one linear projection of the features against a threshold. The name follows, rather grossly, because this is a highly reduced decision tree. There are two common strategies. In one, the stump tests a single feature against a threshold. In the other, the stump projects the features onto some vector chosen during learning, and tests that against a threshold.

Decision stumps are useful because they're easy to learn, though not in themselves a particularly strong classifier. We have examples $(\mathbf{x}_i, y_i)$. We will assume that y_i are 1 or -1. Write $f(\mathbf{x}; \theta)$ for the stump, which will predict -1 or 1. For gradient boost, we will receive a set of weights h_i (one per example), and try to learn a decision stump that *minimizes* the sum $\sum_i h_i f(\mathbf{x}_i; \theta)$ by choice of θ. We use a straightforward search, looking at each feature and for each, checking a set of thresholds to find the one that maximises the sum. If we seek a stump that projects features, we project the features onto a set of random directions first. The box below gives the details.

Procedure: 12.4 *Learning a decision stump*

We have examples $(\mathbf{x}_i, y_i)$. We will assume that y_i are 1 or -1, and $\mathbf{x}_i$ have dimension d. Write $f(\mathbf{x}; \theta)$ for the stump, which will predict -1 or 1. We receive a set of weights h_i (one per example), and wish to learn a decision stump that *minimizes* the sum $\sum_i h_i f(\mathbf{x}_i; \theta)$. If the dataset is too large for your computational resources, obtain a subset by sampling uniformly at random without replacement. The parameters will be a projection, a threshold and a sign. Now for $j = 1 : d$

- Set $\mathbf{v}_j$ to be either a random d-dimensional vector *or* the j'th basis vector (i.e. all zeros, except a one in the j'th component).
- Compute $r_i = \mathbf{v}_j^T \mathbf{x}_i$.
- Sort these r's; now construct a collection of thresholds t from the sorted r's where each threshold is halfway between the sorted values.
- For each t, construct two predictors. One reports 1 if $r > t$, and -1 otherwise; the other reports -1 if $r > t$ and 1 otherwise. For each of these predictors, compute the value $\sum_i h_i f(\mathbf{x}_i; \theta)$. If this value is smaller than any seen before, keep $\mathbf{v}_j$, t, and the sign of the predictor.

Now report the $\mathbf{v}_j$, t, and sign that obtained the best value.

Remember this: *Decision stumps are very small decision trees. They are easy to fit, and have a particularly good record with gradient boost.*

12.2.4 Gradient Boost with Decision Stumps

We will work with two-class classification, as boosting multiclass classifiers can be tricky. One can apply gradient boost to any loss that appears convenient. However, there is a strong tradition of using the **exponential loss**. Write y_i for the true label for the i'th example. We will label examples with 1 or -1 (it is easy to derive updates for the case when the labels are 1 or 0 from what follows). Then the exponential loss is

$$\ell_e(y, \mathbf{x}, F(\mathbf{x})) = e^{[-yF(\mathbf{x})]}.$$

Notice if $F(\mathbf{x})$ has the right sign, the loss is small; if it has the wrong sign, the loss is large.

We will use a decision stump. Decision stumps report a label (i.e. 1 or -1). Notice this doesn't mean that F reports only 1 or -1, because F is a weighted sum

of predictors. Assume we know F_{r-1}, and seek a_r and f_r. We then form

$$w_i^{(j)} = \left.\frac{\partial \ell}{\partial F}\right|_i = -y_i e^{[-y_i F(\mathbf{x}_i)]}.$$

Notice there is one weight per example. The weight is negative if the label is positive, and positive if the label is negative. If F gets the example right, the weight will have small magnitude, and if F gets the example wrong, the weight will have large magnitude. We want to choose parameters $\hat{\theta}^{(j)}$ so that

$$C(\theta^{(j)}) = \sum_i w_i^{(j)} f(\mathbf{x}_i; \hat{\theta}^{(j)}).$$

is negative. Assume that the search described in box 12.4 is successful at producing such an $f(\mathbf{x}_i; \hat{\theta}^{(j)})$. To get a negative value, $f(\cdot; \hat{\theta}^{(j)})$ should try to report the same sign as the example's label (recall the weight is negative if the label is positive, and positive if the label is negative). This means that (mostly) if F gets a positive example right, $f(\cdot; \hat{\theta}^{(j)})$ will try to increase the value that F takes, etc.

The search produces an $f(\cdot; \hat{\theta}^{(j)})$ that has a large absolute value of $C(\theta^{(j)})$. Such a $f(\cdot; \hat{\theta}^{(j)})$ should have large absolute values for examples where $w_i^{(j)}$ has large magnitude. But these are examples that F got very badly wrong (i.e. produced a prediction of large magnitude, but the wrong sign).

It is easy to choose a decision stump that minimizes this expression$C(\theta)$. The weights are fixed, and the stump reports either 1 or -1, so all we need to do is search for a split that achieves a minimum. You should notice that the minimum is always negative (unless all weights are zero, which can't happen). This is because you can multiply the stump's prediction by -1 and so flip the sign of the score.

12.2.5 Gradient Boost with other Predictors

A decision stump makes it easy to construct a predictor such that

$$\sum_i w_{r-1,i} f_r(\mathbf{x}_i; \theta_r)$$

is negative. For other predictors, it may not be so easy. It turns out that this criterion can be modified, making it straightforward to use other predictors. There are two ways to think about these modifications, which end up in the same place: choosing $\hat{\theta}^{(j)}$ to minimize

$$\sum_i (\left[-w_i^{(j)}\right] - f(\mathbf{x}_i; \theta^{(j)}))^2$$

is as good (or good enough) for gradient boost to succeed. This is an extremely convenient result, because many different regression procedures can minimize this loss. I will give both derivations, as different people find different lines of reasoning easier to accept.

Reasoning about minimization: Notice that

$$\sum_i (\left[-w_i^{(j)}\right] - f(\mathbf{x}_i; \theta^{(j)}))^2 = \sum_i \begin{bmatrix} \left(w_i^{(j)}\right)^2 \\ +(f(\mathbf{x}_i; \theta^{(j)}))^2 \\ +2(w_i^{(j)} f(\mathbf{x}_i; \theta^{(j)})) \end{bmatrix}.$$

Now assume that $\sum_i (f(\mathbf{x}_i; \theta^{(j)}))^2$ is not affected by $\theta^{(j)}$. For example, f could be a decision tree that reports either 1 or -1. In fact, it is usually sufficient that $\sum_i (f(\mathbf{x}_i; \theta^{(j)}))^2$ is not much affected by $\theta^{(j)}$. Then if you have a small value of

$$\sum_i (\left[-w_i^{(j)}\right] - f(\mathbf{x}_i; \theta^{(j)})^2,$$

that must be because $\sum_i w_{r-1,i} f(\mathbf{x}_i; \theta_r)$ is negative. So we seek $\theta^{(j)}$ that minimizes

$$\sum_i (\left[-w_i^{(j)}\right] - f(\mathbf{x}_i; \theta^{(j)}))^2.$$

Reasoning about descent directions: You can think of $\mathcal{L}$ as a function that accepts a vector of prediction values, one at each data point. Write $\mathbf{v}$ for this vector. The values are produced by the current predictor. In this model, we have that

$$\nabla_{\mathbf{v}} \mathcal{L} \propto w_i^{(j)}.$$

In turn, this suggests we should minimize $\mathcal{L}$ by obtaining a new predictor f which takes values as close as possible to $-\nabla_{\mathbf{v}} \mathcal{L}$ – that is, choose f_r that minimizes

$$\sum_i (\left[-w_i^{(j)}\right] - f(\mathbf{x}_i; \theta^{(j)}))^2.$$

Remember this: *The original fitting criterion for predictors in gradient boost is awkward. An easy argument turns this into a familiar regression problem*

12.2.6 Example: Is a Prescriber an Opiate Prescriber?

You can find a dataset of prescriber behavior focusing on opiate prescriptions at https://www.kaggle.com/apryor6/us-opiate-prescriptions. One column of this data is a 0-1 answer, giving whether the individual prescribed opiate drugs more than 10 times in the year. The question here is: does a doctor's pattern of prescribing predict whether that doctor will predict opiates?

You can argue this question either way. It is possible that doctors who see many patients who need opiates also see many patients who need other kinds of drug for similar underlying conditions. This would mean the pattern of drugs prescribed would suggest whether the doctor prescribed opiates. It is possible that there are prescribers engaging in deliberate fraud (e.g. prescribing drugs that aren't necessary, for extra money). Such prescribers would tend to prescribe drugs that have informal uses that people are willing to pay for, and opiates are one such drug, so the pattern of prescriptions would be predictive. The alternative possibility is that patients who need opiates attend doctors randomly, so that the pattern of drugs prescribed isn't predictive.

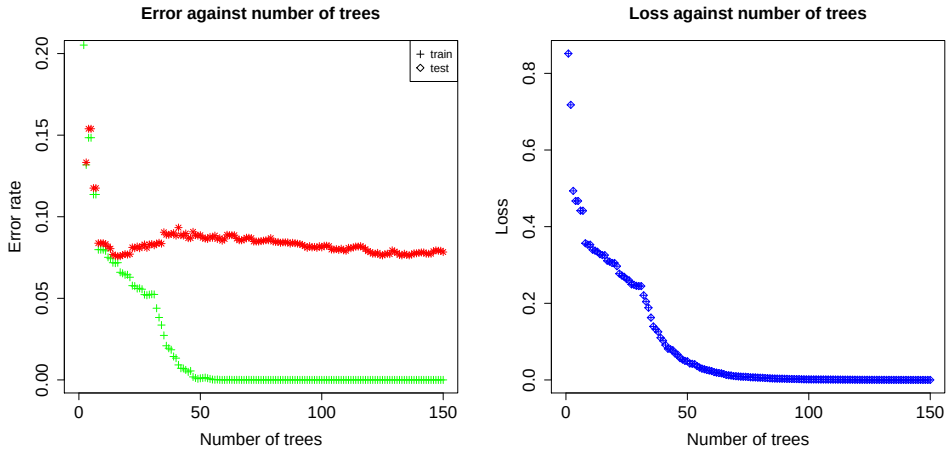

FIGURE 12.4: *Models for a boosted decision tree classifier predicting whether a given prescriber will write more than 10 opioid prescriptions in a year, using the data of https://www.kaggle.com/apryor6/us-opiate-prescriptions.* **Left:** *train and test error against number of trees;* **right:** *exponential loss against number of trees. Notice that both test and train error go down, but there is a test-train gap. Notice also a characteristic property of boosting; continuing to boost after the training error is zero (about 50 trees in this case) still results in improvements in the test error. Note also that lower exponential loss doesn't guarantee lower training error.*

We will predict the `Opioid.Prescriber` column from the other entries, using a boosted decision tree and the exponential loss function. Confusingly, the column is named `Opioid.Prescriber` but all the pages, etc. use the term "opiate"; the internet suggests that "opiates" come from opium, and "opioids" are semi-synthetic or synthetic materials that bind to the same receptors. Quite a lot of money rides on soothing the anxieties of internet readers about these substances, so I'm inclined to assume that easily available information is unreliable; for us, they will mean the same thing.

This is a fairly complicated classification problem. It is natural to try gradient boost using a regression tree. To fit the regression tree, I used R's `rpart`; for linesearch, I used Newton's method. Doing so produces quite good classification (figure 12.4). This figure illustrates two very useful and quite typical feature of a boosted classifier.

- **The test error usually declines even after the training error is zero.** Look at figure 12.4, and notice the training error hits zero shortly after 50 trees. The loss is not zero there – exponential loss can never be zero – and continues to decline as more trees are added, even when the training error hits zero. Better, the test error continues to decline, though slowly. The exponential loss has the property that, even if the training error is zero, the predictor tries to have larger magnitude at each trainiong point (i.e. boosting tries to make $F(\mathbf{x}_i)$ larger if y_i is positive, and smaller if y_i is negative). In

turn, this means that, even after the training error is zero, changes to F might cause some test examples to change sign.

- **The test error doesn't increase sharply however far boosting proceeds.** You could reasonably be concerned that adding new weak learners to a boosted predictor would eventually cause overfitting problems. This can happen, but doesn't happen very often. It's also quite usual that the overfitting is mild. For this reason, it was believed until relatively recently that overfitting could never happen. Mostly, adding weak learners results in slow improvements to test error. This effect is most reliable when the weak learners are relatively simple, like decision stumps. The predictor we learn is regularized by the fact that a collection of decision stumps is less inclined to overfit than one might reasonably expect. In turn, this justifies the usual practice of not incorporating explicity regularizers in a boosting loss.

12.2.7 Pruning the Boosted Predictor with the Lasso

You should notice there is a relatively large number of predictors here, and it's reasonable to wonder if one could get good results with fewer. This is a good question. When you construct a set of boosted predictors, there is no guarantee they are all necessary to achieve a particular error rate. Each new predictor is constructed to cause the loss to go down. But the loss could go down without causing the error rate to go down. There is a reasonable prospect that some of the predictors are redundant.

Whether this matters depends somewhat on the application. It may be important to evaluate the minimum number of predictors. Furthermore, having many predictors might (but doesn't usually) create generalization problems. One strategy to remove redundant predictors is to use the Lasso. For a two-class classifier, one uses a generalized linear model (logistic regression) applied to the values of the predictors at each example. Figure 12.5 shows the result of using a Lasso (from `glmnet`) to the predictors used to make Figure 12.4. Notice that reducing the size of the model seems not to result in significant loss of classification accuracy here.

There is one point to be careful about. You should not compute a cross-validated estimate of error on all data. That estimate of error will be biased low, because you are using some data on which the predictors were trained (you must have a training set to fit the boosted model and obtain the predictors in the first place). There are two options: you could fit a Lasso on the training data, then evaluate on test; or you could use cross-validation to evaluate a fitted Lasso on the test set alone. Neither strategy is perfect. If you fit a Lasso to the training data, you may not make the best estimate of coefficients, because you are not taking into account variations caused by test-train splits (Figure 12.5). But if you use cross-validation on the test set alone, you will be omitting quite a lot of data. This is a large dataset (25,000 prescribers) so I tried both approaches (compare Figure 12.5 with Figure 12.6). A better option might be to apply the Lasso *during* the boosting process, but this is beyond our scope.

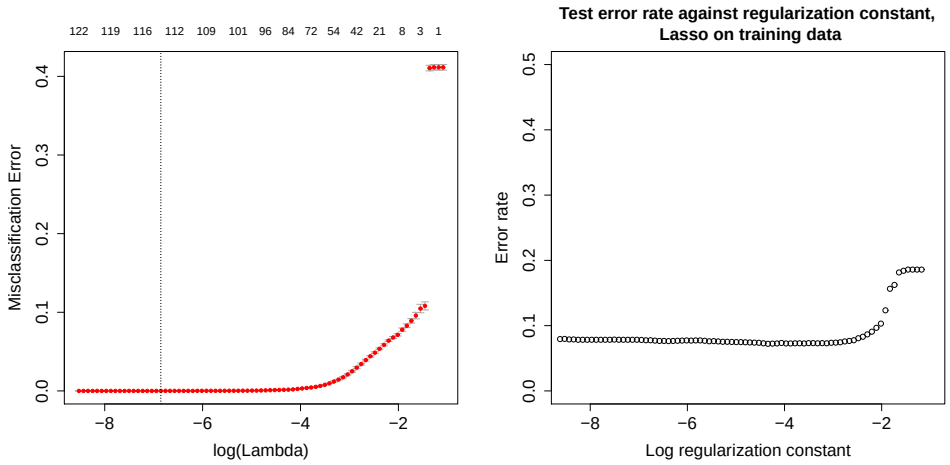

FIGURE 12.5: **Left:** *A cross-validation plot from* `cv.glmnet` *for the lasso applied to predictions on training data obtained from all 150 trees from the boosted model of Figure 12.4. You should not believe this cross-validated error, because the predictors were trained on this data. This means that data on both sides of the cross-validation split have been seen by the model (though not by the lasso). This plot suggests zero error is attainable by quite small models. But the cross-validated error here is biased low as the plot on the right confirms.* **Right:** *The error of the best model for each value of the regularization constant in the plot on the left, now evaluated on a held-out set. If you have enough data, you could break it into train (for training predictors and the lasso), validation (to select a model, using a plot like this) and test (to evaluate the resulting model.*

> **Remember this:** *You can prune boosted models with the lasso. It's often very effective, but you need to be careful about how you choose a model – it's easy to accidentally evaluate on training data.*

12.2.8 Gradient Boosting Software

Up to this point, the examples shown have used simple loops I wrote with R. This is fine for small datasets, but gradient boost can be applied very successfully to extremely large datasets. For example, many recent Kaggle competitions have been won with gradient boosting methods. Quite a lot of the work in boosting methods admits parallelization across multiple threads or across multiple machines. Various clever speedups are also available. When the dataset is large, you need to have software that can exploit these tricks. As of writing, the standard is **XGBoost**, which can be obtained from https://xgboost.ai. This is the work of a large open-source developer community, based around code by Tianqi Chen and a paper by

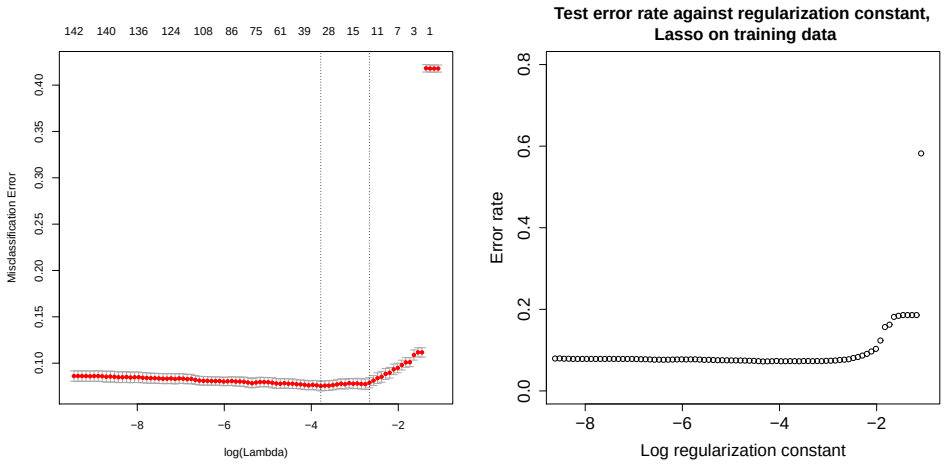

FIGURE 12.6: *Here I have split the data into two. I used a training set to produce predictors using gradient boosting applied to decision stumps. I then applied* `cv.glmnet` *to a separate test set.* **Left:** *A cross-validation plot from* `cv.glmnet` *for the lasso applied to predictions on* test *data passed through each of the 150 trees made by the boosted model of Figure 12.4. This gives an accurate estimate of the error, as you can see by comparing to the test error of the best model for each value of the regularization constant (***right***). This approach gives a better estimate of what the model will do, but may present problems if you have little data.*

Tianqi Chen and Carlos Guestrin. The paper is *XGBoost: A Scalable Tree Boosting System*, which you can find in Proc. SIGKDD 2016, or at https://arxiv.org/abs/1603.02754.

XGBoost has a variety of features to notice (see the tutorials at https://xgboost.readthedocs.io/en/latest/tutorials/index.html). XGBoost doesn't do line-search. Instead, one sets a parameter `eta` – the value of α in procedure 12.3 – which is fixed. Generally, larger `eta` values result in larger changes of the model with each new tree, but a greater chance of overfitting. There is an interaction between this parameter and the maximum depth of the tree you use. Generally, the larger the maximum depth of a tree (which can be selected), the more likely you will see overfitting, unless you set `eta` small.

XGBoost offers early stopping. If properly invoked, it can monitor error on an appropriate set of data (training or validation, your choice) and, if there is insufficient progress, it will stop training. Using this requires care. If you stop early using test data, the estimate of model performance that XGBoost returns must be biased. This is because it chooses a model (when to stop) using test data. You should follow the recipe of splitting data into three (train, validation, test), then train on training data, use validation for early stopping, and evaluate on test data.

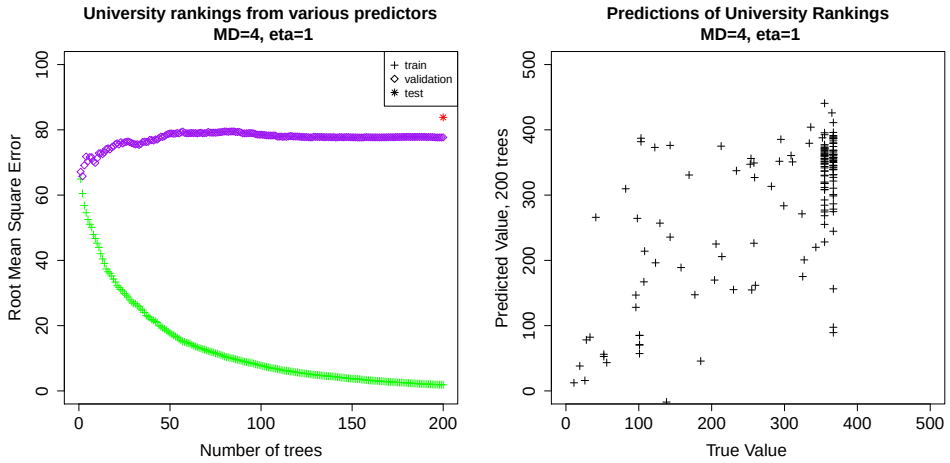

FIGURE 12.7: *On the* **left**, *training and test error for reported by XGBoost for the university ranking data as in example 12.3. The error is RMS error, because I modelled the rank as a continuous variable. The test error is slightly larger than validation error, likely a result of having a small test set. On the* **right**, *plots of predicted value against true value for the final regression. A regression is not a particularly good way of predicting rankings, because it does not know there needs to be a distinct value for each university and it does not know that there can be no ranking greater than the number of universities. Despite these disadvantages, the method can generally predict when a university will be highly ranked but tends to mix up the rankings of lower ranked (= larger number) universities.*

Remember this: *Very good, very fast, very scalable gradient boosting software is available.*

Worked example 12.3 *Predicting the quality of education of a university*

You can find a dataset of measures of universities at https://www.kaggle.com/mylesoneill/world-university-rankings/data. These measures are used to predict rankings. From these measures, but not using the rank or the name of the university, predict the quality of education using a stagewise regression. Use XGBoost.

Solution: Ranking universities is a fertile source of light entertainment for assorted politicians, bureaucrats, and journalists. I have no idea what any of the numbers in this dataset mean (and I suspect I may not be the only one). Anyhow, one could get some sense of how reasonable they are by trying to predict the quality of education score from the others. This is a nice model problem for getting used to XGBoost. I modelled the rank as a continuous variable (which isn't really the best way to produce learned rankers – but we're just trying to see what a new tool does with a regression problem). This means that root-mean-square error is a natural loss. Figure 12.7 shows plots of a simple model. This is trained with trees whose maximum depth is 4. For Figure 12.7, I used $\eta = 1$, which is quite aggressive. The scatter plot of predictions against true values for held-out data (in Figure 12.7) suggests the model has a fairly good idea whether a university is strong or weak, but isn't that good at predicting the rank of universities where the rank is quite large (i.e. there are many stronger universities). For Figure 12.8, I used a maximum depth of 8 and $\eta = 0.1$, which is much more conservative. I allowed the training procedure to stop early if it saw 100 trees without an improvement on a validation set. This model is distinctly better than the model of Figure 12.7. The scatter plot of predictions against true values for held-out data (in Figure 12.7) suggests the model has a fairly good idea whether a university is strong or weak, but isn't that good at predicting the rank of universities where the rank is quite large (i.e. there are many stronger universities).

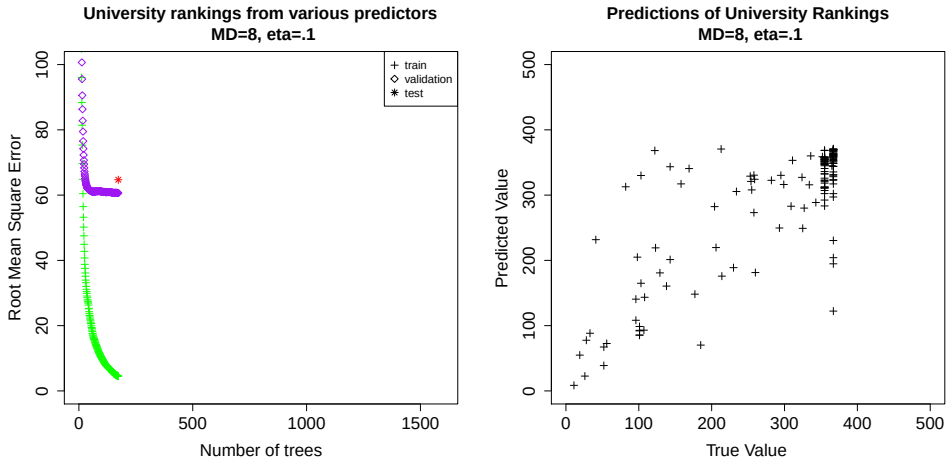

FIGURE 12.8: *On the* **left**, *training and test error for reported by* `xgboost` *for the university ranking data as in example 12.3. This model uses a deeper tree (maximum depth of 8) and a smaller* `eta` *(0.1). It stops once adding 100 trees hasn't changed the validation error much. On the* **right**, *plots of predicted value against true value for the final regression. This model is notably more accurate than that of Figure 12.7.*

Worked example 12.4 *Opioid prescribers with XGBoost*

Use XGBoost to obtain the best test accuracy you can on the dataset of section 12.2.6. Investigate how accurate a model you can build by varying the parameters.

Solution: XGBoost is very fast (somewhere between 10 and 100 times faster than my homebrew gradient booster using rpart), so one can fiddle with hyperparameters to see what happens. Figure 12.9 shows models trained with depth 1 trees (so decision stumps, and comparable with the models of Figure 12.4). The `eta` values were 1 and 0.5. You should notice distinct signs of overfitting – the validation error is drifting upwards quite early in training and continues to do so. There is a very large number of stumps (800) so that all effects are visible. Figure 12.10 shows a model trained with max depth 1 and `eta` of 0.2; again there are notable signs of overfitting. Training conservatively with a deeper tree (max depth 4, `eta` of 0.1, and early stopping) leads to a somewhat better behaved model. All these models are more accurate than those of Figure 12.4

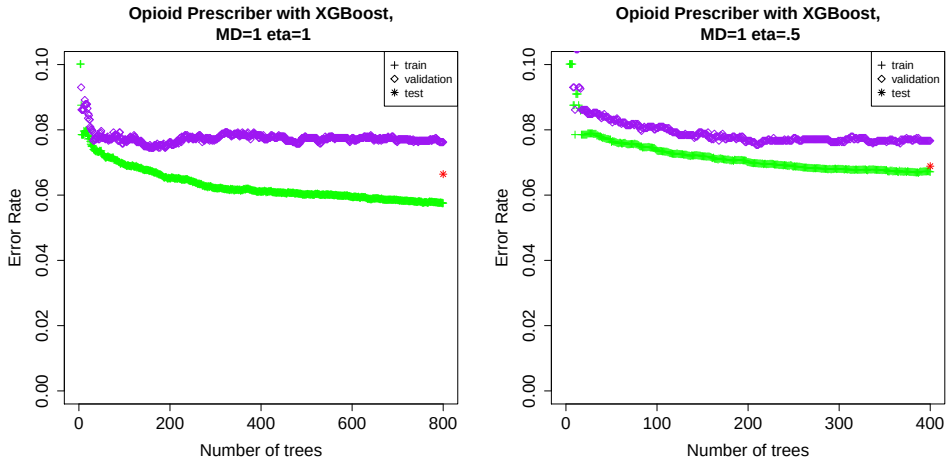

FIGURE 12.9: *On the* **left**, *training, validation and test error reported by* `xgboost` *for the opioid data as in example 12.2.6. Validation error is not strictly needed, as I did not apply early stopping. But the plot is informative. Notice how the validation error drifts up as the number of trees increases. Although this effect is slow and small, it's a sign of overfitting. Decreasing the* `eta` *(***right***) does not cure this trend (see also Figure 12.10).*

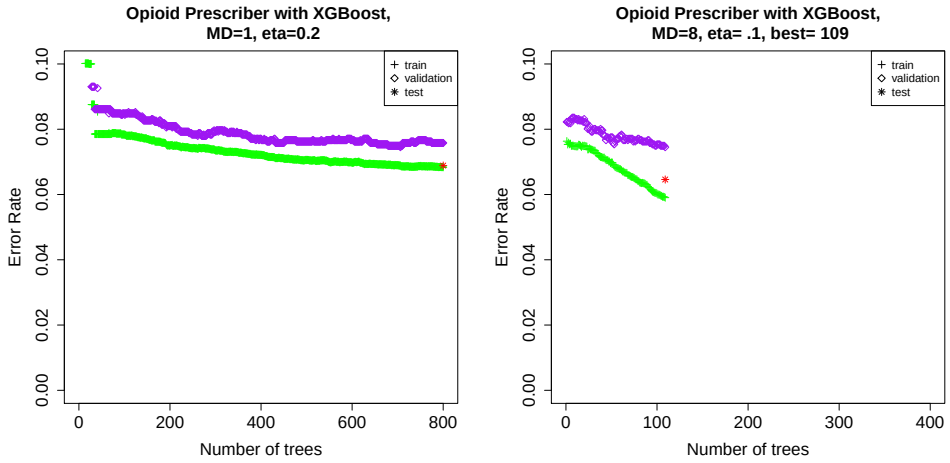

FIGURE 12.10: *On the* **left**, *training, validation and test error reported by* `xgboost` *for the opioid data as in example 12.2.6. This figure should be compared to Figure 12.9. As in that figure, I used a fixed number of trees but now a rather small* `eta` *– this still doesn't cure the problem. On the* **right**, *I used deeper trees, and an even smaller* `eta`, *with early stopping. This produces the strongest model so far.*

12.3 YOU SHOULD

12.3.1 remember these definitions:

12.3.2 remember these terms:

12.3.3 remember these facts:

12.3.4 remember these procedures:

12.3.5 be able to:

- Set up and solve a regression problem using

PROBLEMS

12.1. Show that you cannot improve the training error of a linear regression using all features by a stagewise step.
(a) First, write $\hat{\beta}$ for the value that minimizes

$$(\mathbf{y} - \mathcal{X}\beta)^T(\mathbf{y} - \mathcal{X}\beta).$$

Now show that for $\overline{\beta} \neq \hat{\beta}$, we have

$$(\mathbf{y} - \mathcal{X}\overline{\beta})^T(\mathbf{y} - \mathcal{X}\overline{\beta}) \geq (\mathbf{y} - \mathcal{X}\hat{\beta})^T(\mathbf{y} - \mathcal{X}\hat{\beta}).$$

(b) Now explain why this means the residual can't be improved by regressing it against the features.

12.2. This exercise compares regression trees to linear regression. Mostly, one can improve the training error of a regression tree model by a stagewise step. Write $f(\mathbf{x}; \theta)$ for a regression tree, where θ encodes internal parameters (where to split, thresholds, and so on).

(a) Write $\hat{\theta}$ for the parameters of the regression tree that minimizes

$$\mathcal{L}(\theta) = \sum_i (y_i - f(\mathbf{x}_i; \theta))^2 .$$

over all possible depths, splitting variables, and splitting thresholds. Why is $\mathcal{L}(\hat{\theta}) = 0$?

(b) How many trees achieve this value? Why would you not use that tree (those trees) in practice?

(c) A regression tree is usually regularized by limiting the maximum depth. Why (roughly) should this work?

12.3. We will fit a regression model to N one dimensional points x_i. The value at the i'th point is y_i. We will use regression stumps. These are regression trees that have two leaves. A regression stump can be written as

$$f(x; t, v_1, v_2) = \begin{cases} v_1 & \text{for } x > t \\ v_2 & \text{otherwise} \end{cases}$$

(a) Assume that each data point is distinct (so you don't have $x_i = x_j$ for $i \neq j$). Show that you can build a regression with zero error with N stumps.

(b) Is it possible to build a regression with zero error with fewer than N stumps?

(c) Is there a procedure for fitting stumps that guarantees that gradient boosting results in a model that has zero error when you use exactly N stumps? **Warning:** *This might be quite hard.*

12.4. We will fit a classifier to N data points $\mathbf{x}_i$ with labels y_i (which are 1 or -1). The data points are distinct. We use the exponential loss, and use decision stumps identified using procedure 12.4. Write

$$F_r(\mathbf{x}; \theta, \mathbf{a}) = \sum_{j=1}^{r} a_j f(\mathbf{x}; \theta^{(j)})$$

for the predictor that uses r decision stumps, $F_0 = 0$, and $L(F_r)$ for the exponential loss evaluated for that predictor on the dataset.

(a) Show that there is some α_1 so that $L(F_1) < L(F_0)$ when you use this procedure for fitting stumps.

(b) Show that there is some α_i so that $L(F_i) < L(F_{i-1})$ when you use this procedure for fitting stumps.

(c) All this means that the loss must continue to decline through an arbitrary number of rounds of boosting. Why does it not stop declining?

(d) If the loss declines at every round of boosting, does the training error do so as well? Why?

PROGRAMMING EXERCISES

General remark: *These exercises are suggested activities, and are rather open ended. Installing multi-threaded XGBoost — which you'll need — on a Mac can get quite exciting, but nothing that can't be solved with a bit of searching.*

12.5. Reproduce the example of section 12.2.6, using a decision stump. You should write your own code for this stump and gradient boost. Prune the boosted predictor with the lasso. What test accuracy do you get?

12.6. Reproduce the example of section 12.2.6, using XGBoost and adjusting hyperparameters (`eta`; the maximum depth of the tree; and so on) to get the best result. What test accuracy do you get?

12.7. Use XGBoost to classify MNIST digits, working directly with the pixels. This means you will have a 784 dimensional feature set. What test accuracy do you get? (mine was suprisingly high compared to the example of section 17.2.1).

12.8. Investigate feature constructions for using XGBoost to classify MNIST digits. The subexercises suggest feature constructions. What test accuracy do you get?

- **(a)** One natural construction is to project the images onto a set of principal components (50 is a good place to start, yielding a 50 dimensional feature vector).
- **(b)** Another natural construction is to project the images each of the per-class principal components (50 is a good place to start, yielding a 500 dimensional feature vector).
- **(c)** Yet another natural construction is to use vector quantization for windows on a grid in each image.

12.9. Use XGBoost to classify CIFAR-10 images, working directly with the pixels. This means you will have a 3072 dimensional feature set. What test accuracy do you get?

12.10. Investigate feature constructions for using XGBoost to classify CIFAR-10 images. The subexercises suggest feature constructions. What test accuracy do you get?

- **(a)** One natural construction is to project the images onto a set of principal components (50 is a good place to start, yielding a 50 dimensional feature vector).
- **(b)** Another natural construction is to project the images each of the per-class principal components (50 is a good place to start, yielding a 500 dimensional feature vector).
- **(c)** Yet another natural construction is to use vector quantization for windows on a grid in each image.