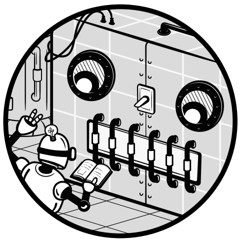
6 TWEAKING THE TREES: BAGGING, RANDOM FORESTS, AND BOOSTING

AdaBoost is the best off-the-shelf classifier in the world.
–CART co-inventor Leo Breiman, 1996
XGBoost is the algorithm of choice for many winning teams of machine learning competitions.
–Wikipedia entry, 2022



Here we talk about two general techniques in ML, *bagging* and *boosting*, and apply them to form extensions of decision tree analysis. The extensions, *random forests* and *tree-based gradient boosting*, are widely used–in fact, even more so than individual tree methods.

6.1 Bias vs. Variance, Bagging, and Boosting

For want of a nail the shoe was lost;
for want of a shoe the horse was lost;
and for want of a horse the man was lost.
—Old proverb

We must always bear in mind that we are dealing with sample data. Sometimes the "population" being sampled is largely conceptual; for example, in the taxi data in Section 5.3, we are considering the data a sample from the ridership in all days, past, present, and future. But in any case, there is sampling variation.

In the bike rental data, say, what if the data collection period had gone one more day? Even this slight change might affect the exact split at the top of the tree, node 1. And that effect could then change the splits (or possibly non-splits) at nodes 2 and 3 and so on, with the those changes cascading down to the lowest levels of the resulting tree. Note that not only might the split points in the nodes change, but the membership of the nodes could also change. A training set data point that had been in node 2 may now be in node 3. In other words:

> Decision trees can be very sensitive to slight changes in the inputs. That means they are very sensitive to sampling variation—that is, **decision trees have a high variance.**

Recall that splitting a node reduces bias, and that typically reducing bias also increases variance. But for the reason given above, variance may be especially problematic in DT settings.

In this chapter, we treat two major methods for handling this problem, *bagging/random forests* and *boosting*. Both take this point of view: "Variance too high? Well, that means the sample size is too small, so let's generate more trees!" But how?

6.2 Bagging: Generating New Trees by Resampling

The term *bagging* refers to an ML version of a handy tool from modern statistics known as the *bootstrap*. This consists of drawing many random subsamples from our data, applying our given estimator to each subsample, and then averaging (or otherwise combining) the results. Here we apply the bootstrap to decision trees.

Starting with our original data, once again considered a sample from some population, we'll generate s new samples from the original dataset. We generate a new sample by randomly sampling m of our n data points—*with* replacement. (We may get a few duplicates.) We'll fit a tree to each new sample, thus achieving the above goal of generating more trees, and combine the results in a manner to be presented shortly. The quantities s and m here are—you guessed it—hyperparameters.

6.2.1 Random Forests

Say we have a new case to be predicted. We will then *aggregate* the *s* trees by forming a prediction for each tree and then combining all those predicted values to form our final prediction as follows:

- In a numeric-*Y* setting, the combining would take the form of averaging all the predictions. In the taxi data, for example (Section 5.3), each tree would give us a predicted trip time, and our final predicted trip time would be the average of all those individual predictions.
- In a classification setting, such as the vertebrae example in Section 2.3, we could combine by using a *voting* process. For each tree, we would find the predicted class, NO, DH, or SP, and then see what class received the most "votes" among the various trees. That would be our predicted class. Or, we could find the estimated class probabilities for this new case, for each tree, and then average the probabilities. Our predicted class would be whichever one has the largest average.

So, we do a bootstrap and then aggregation, hence the short name *bagging*. It is also commonly known as *random forests*, a specific implementation by Leo Breiman. (The earliest proposal along these lines seems to be that of Tin Kam Ho. She called the method *random decision forests*.) That approach places a limit on the number of features under consideration for splitting at any given node, with a different candidate set at each step.

Why might this strategy, which is using a different candidate set of features each time, work? Ordinary bagging can result in substantially correlated trees because it tends to choose the same features every time. It can be shown that the average of positively correlated numbers has a higher variance than the average of independent numbers. Thus the approach in which we limit the candidate feature set at each step hopefully reduces variance.

6.2.2 The qeRF() Function

The `qe*`-series of functions actually includes several for random forests. For a given application, one may be more accurate or faster than others, but they all use the general random forest paradigm described previously. We'll use `qeRF()` here.

Recall that the `qe*` functions all have the following arguments:

`data` A data frame containing our training data.

`yName` The name of the column in `data` containing *Y*, the outcome variable to be predicted. The user distinguishes between numeric-*Y* and classification settings by having this column be numeric or an R factor, respectively.

holdout Size of the optional holdout set.

Application-specific arguments For example, as the number `k` of near neighbors in the case of `qeKNN()`.

Each `qe*` function is a wrapper interface to a function in a standard R ML package. In the case of random forests, `qeRF()` is a wrapper for `randomForest` in the package of the same name. The call form is:

```
qeRF(data, yName, nTree = 500, minNodeSize = 10,
   holdout = floor(min(1000,0.1 * nrow(data))))
```

The application-specific arguments are `ntree`, which is the number of bootstrapped trees to generate, and `minNodeSize`, which is similar to `minsplit` in `ctree()`.

6.2.3 Example: Vertebrae Data

Let's look again at the vertebrae dataset in Section 2.3, now applying random forests instead of k-NN. We'll predict the same hypothetical new case as in that earlier example:

```
# fit RF model
> rfout <- qeRF(vert,'V7',holdout=NULL)
# new case to predict
> z <- vert[1,-7]
> z$V2 <- 18
# predict
> predict(rfout,z)
$predClasses
[1] "DH"

$probs
     DH    NO   SL
2 0.532 0.378 0.09
attr(,"class")
[1] "matrix" "array"  "votes"
```

With k-NN, we had predicted the same class, DH, but with slightly different class probabilities:

```
> predict(kout,z)
$predClasses
[1] "DH"

$probs
      DH  NO  SL
[1,] 0.6 0.2 0.2
```

The difference between the two sets of probabilities is due both to the fact that we used two different ML algorithms and to the small n in this dataset (310), which caused large sample variability.

We used the default values here for `nTree` and `minNodeSize`. We could explore a few other pairs of these hyperparameters and then compare the performance of random forests and k-NN on this dataset.

6.2.4 Example: Remote-Sensing Soil Analysis

Here we will analyze the African Soil Property dataset from Kaggle.[1] From the data site:

> Advances in rapid, low cost analysis of soil samples using infrared spectroscopy, georeferencing of soil samples, and greater availability of earth remote sensing data provide new opportunities for predicting soil functional properties at unsampled locations. . . . Digital mapping of soil functional properties, especially in data sparse regions such as Africa, is important for planning sustainable agricultural intensification and natural resources management.

We wish to predict various soil properties without directly testing the soil.

One important property of this dataset that we have not encountered before is that it has $p > n$ (that is, more columns than rows). The original first column, an ID variable, has been removed.

```
> dim(afrsoil)
[1] 1157 3599
```

Traditionally, the statistics field has been wary of this kind of setting, as linear models (Chapter 8) do not work there. One must first do dimension reduction. Tree-based methods do this as an integral aspect of their operation, so let's give it a try using `qeRF()`.

Here are the names of the columns:

```
> names(afrsoil)
...
[3547] "m659.543" "m657.615" "m655.686" "m653.758" "m651.829" "m649.901"
[3553] "m647.972" "m646.044" "m644.115" "m642.187" "m640.258" "m638.33"
[3559] "m636.401" "m634.473" "m632.544" "m630.616" "m628.687" "m626.759"
[3565] "m624.83"  "m622.902" "m620.973" "m619.045" "m617.116" "m615.188"
[3571] "m613.259" "m611.331" "m609.402" "m607.474" "m605.545" "m603.617"
[3577] "m601.688" "m599.76"  "BSAN"     "BSAS"     "BSAV"     "CTI"
[3583] "ELEV"     "EVI"      "LSTD"     "LSTN"     "REF1"     "REF2"
[3589] "REF3"     "REF7"     "RELI"     "TMAP"     "TMFI"     "Depth"
[3595] "Ca"       "P"        "pH"       "SOC"      "Sand"
```

1. *https://www.kaggle.com/c/afsis-soil-properties/data*

Columns 1 through 3594 are the X variables, with cryptic code names. The remaining columns are Y, some with more easily guessable names. We'll predict pH, the soil acidity.

This kind of setting is considered tough. There is a major potential for overfitting, since with so many features, one or more of them may accidentally look to be a strong predictor due to p-hacking (Section 1.13). Let's see how well `qeRF()` does here.

```
> set.seed(9999)
> rfo <- qeRF(afrsoil[,c(1:3578,3597)],'pH',holdout=500)
> rfo$testAcc
[1] 0.3894484
> rfo$baseAcc
[1] 0.6858574
```

Use of the features has cut MAPE almost in half. Note the range under the pH scale used here:

```
> range(afrsoil$pH)
[1] -1.886946  3.416117
```

We are now ready to predict, say, on a hypothetical new case like that of row 88 in the training data:

```
> predict(rfo,afrsoil[88,1:3594])
       88
0.6068828
```

We would predict a pH level of about 0.61.

6.3 Boosting: Repeatedly Tweaking a Tree

Imagine a classification problem with just two classes, so $Y = 1$ or 0, and just one feature, X, say, age. We fit a tree with just one level. Suppose our rule is to guess $Y = 1$ if $X > 12.5$ and guess $Y = 0$ if $X \leq 12.5$. *Boosting* would involve exploring the effect of small changes to the 12.5 threshold on our overall rate of correct classification.

Consider a data point for which $X = 5.2$. In the original analysis, we'd guess Y to be 0. And, here is the point, if we were to move the threshold to, say, 11.9, we would *still* guess $Y = 0$. But the move may turn some misclassified data points near 12.5 to correctly classified ones. If more formerly misclassified points become correctly classified than vice versa, it's a win.

So the idea of boosting is to tweak the original tree, thus forming a new tree, then in turn tweak that new tree, forming a second new tree, and so on. After generating s trees (s is a hyperparameter), we predict a new case by plugging it into all those trees and somehow combining the resulting predicted values.

6.3.1 Implementation: AdaBoost

The first proposal made for boosting was *AdaBoost*. The tweaking involves assigning weights to the points in our training set, which change with each tree. Each time we form a new tree, we fit a tree according to the latest set of weights, updating them with each new tree.

In a numeric-*Y* situation, to predict a new case with a certain *X* value, we plug that value into all the trees, yielding *s* predicted values. Our final predicted value in a numeric-*Y* setting is a weighted average of the individual predictions. In a classification setting, we would take a weighted average of the estimated probabilities of $Y = 1$ to get the final probability estimate, or use weighted voting.

To make this idea concrete, below is an outline of how the process could be implemented with `ctree()`. It relies on the fact that one of the arguments in `ctree()`, named `weights`, is a vector of nonnegative numbers, one for each data point. Say our response is named `y`, with features *x*. Denote the portion of the data frame `d` for *x* by `dx`.

In the pseudocode code below, we will maintain two vectors of weights:

1. `wts` will store the current weightings of the various rows in the training data. Recall that as the boosting process evolves, we will weight some rows more heavily than others according to their current impact om misclassification.
2. `alpha` will store the current weights of our various trees. Recall that in the end, when we do prediction, we will place more weight on some trees than others.

Here is an outline of the algorithm:

```
ctboost <- function(d,s) {
   # uniform weights to begin
   wts <- rep(1/n,n)
   trees <- list()
   alpha <- vector(length=s)  # alpha[i] = coefficient for tree i
   for(treeNum in 1:s) {
      trees[[i]] <- ctree(y ~ x,data=d,weights=wts)
      preds <- predict(trees[[i]],dx)
      # update wts, placing larger weight on data points on which
      # we had the largest errors (regression case) or which we
      # misclassified (classification case)
      wts <- (computation not shown)
      # find latest tree weight
      alpha[i] <- (computation not shown)
   }
   l <- list(trees=trees,treeWts=alpha)
   class(l) <- 'ctboost'
   return(l)
}
```

And to predict a new case, newx:

```
predict.ctboost <- function(ctbObject,newx)
{
   trees <- ctbObject$trees
   alpha <- ctbObject$alpha
   pred <- 0.0
   for (i in 1:s) {
      pred <- pred + alpha[i] * predict(trees[[i]],newx)
   }
   return(pred)
}
```

Since this book is aimed to be nonmathematical, we omit the formulas for wts and alpha. It should be noted, though, that alpha is an increasing sequence, so when we predict new cases, the later trees play a larger role.

The qeML package has a function for AdaBoost, qeAdaBoost(). But it is applicable to classification settings only, so let's go right to the next form of boosting.

6.3.2 Gradient Boosting

In statistics/ML there is the notion of a *residual*—that is, the difference between a predicted value and actual value. *Gradient boosting* works by fitting trees to residuals. Given our dataset, a rough description of the process is as follows:

1. Start with some initial tree. Set *CurrentTree* to it.
2. For each of our data points, calculate the residuals for *CurrentTree*.
3. Fit a tree *to the residuals*—that is, take our residuals as the "data" and fit a tree T on it. Set *CurrentTree* = *T*.
4. Go to Step 2.

These steps are iterated for the number of trees specified by the user. Then, to predict a new case, we plug it into all the trees. The predicted value is simply the sum of the predicted values from the individual trees.

At any given step, we are saying, "Good, we've got a certain predictive ability so far, so let's work on what is left over—that is, our current errors." Hence our predicted value for any new case is the sum of what each tree predicts for that case.

6.3.2.1 The qeGBoost() Function

The qe* function for gradient boosting is qeGBoost(), a wrapper for gbm() in the package of the same name. Its call form is:

```
qeGBoost(data, yName, nTree = 100, minNodeSize = 10, learnRate = 0.1,
    holdout = floor(min(1000, 0.1 * nrow(data))))
```

This is similar to `qeRF()`, but with a new argument, the *learning rate*. The latter is a common notion in ML and will be explained shortly.

NOTE *A number of gradient boosting packages are available for R. We chose this one for its simplicity. Just as was the case above for random forests, other packages may be faster or more accurate on some datasets, notably `qeXGBoost`. Here, `qeGBoost()` sticks to the "quick and easy" philosophy of the qe* series, but the reader is encouraged to explore other packages as an advanced topic.*

6.3.3 Example: Call Network Monitoring

Let's first apply boosting to a dataset titled Call Test Measurements for Mobile Network Monitoring and Optimization,[2] which rates quality of service on mobile calls. The aim is to predict the quality rating.

6.3.3.1 The Data

Here is an introduction to the data:

```
> ds <- read.csv('dataset.csv',stringsAsFactors=TRUE)
> names(ds)
[1] "Date.Of.Test"            "Signal..dBm."
[3] "Speed..m.s."             "Distance.from.site..m."
[5] "Call.Test.Duration..s."  "Call.Test.Result"
[7] "Call.Test.Technology"    "Call.Test.Setup.Time..s."
[9] "MOS"
> ds <- ds[,-1]
> head(ds)
  Signal..dBm. Speed..m.s. Distance.from.site..m. Call.Test.Duration..s.
1          -61       68.80                1048.60                     90
2          -61       68.77                1855.54                     90
3          -71       69.17                1685.62                     90
4          -65       69.28                1770.92                     90
5         -103        0.82                 256.07                     60
6          -61       68.86                 452.50                     90
  Call.Test.Result Call.Test.Technology Call.Test.Setup.Time..s. MOS
1          SUCCESS                 UMTS                     0.56 2.1
2          SUCCESS                 UMTS                     0.45 3.2
3          SUCCESS                 UMTS                     0.51 2.1
4          SUCCESS                 UMTS                     0.00 1.0
5          SUCCESS                 UMTS                     3.35 3.6
6          SUCCESS                 UMTS                     0.00 1.0
...
```

Here Y is `MOS`, the quality of service.

How big is it?

2. *https://www.kaggle.com/valeriol93/predict-qoe*

```
> dim(ds)
[1] 105828      8
```

Now, let's fit the model.

6.3.3.2 Fitting the Model

With over 100,000 data points and just 8 features, overfitting should not be an issue in this dataset. It easily satisfies our rough rule of thumb, $p < \sqrt{n}$ (Section 3.1.3). So, let's not bother with a holdout set. There is still some randomness in the algorithm, though, so for consistency, let's set the random seed.

```
> set.seed(9999)
> gbout <- qeGBoost(ds,'MOS',nTree=750,holdout=NULL)
```

The default value for `nTree` is only 100, but we tried a much larger number, 750, for reasons that will become clear below.

Let's do a prediction. Say we have a case like `ds[3,]`, but with distance being 1,500 and duration 62:

```
> ds3 <- ds[3,-8]
> ds3[,3] <- 1500
> ds3[,4] <- 62
> predict(gbout,ds3)
[1] 2.462538
```

6.3.3.3 Hyperparameter: Number of Trees

But should we have used so many trees? After all, 750 may be overfitting. Maybe the later trees were doing "noise fitting." The package has a couple of ways of addressing that issue, one of which is to use the auxiliary function `gbm.perf()`. Applied to the output of `gbm()`, it estimates the optimal number of trees.

As noted, `qeGBoost()` calls `gbm()` and places the output of the latter in the `gbmOuts` component of its own output. So, we are able to call `gbm.perf()`:

```
> gbm.perf(gbout$gbmOuts)
```

See the output graph in Figure 6-1. The dashed vertical line shows the estimated "sweet spot"—that is, the best number of trees, 382 in this case. (This value is also printed to the R console.)

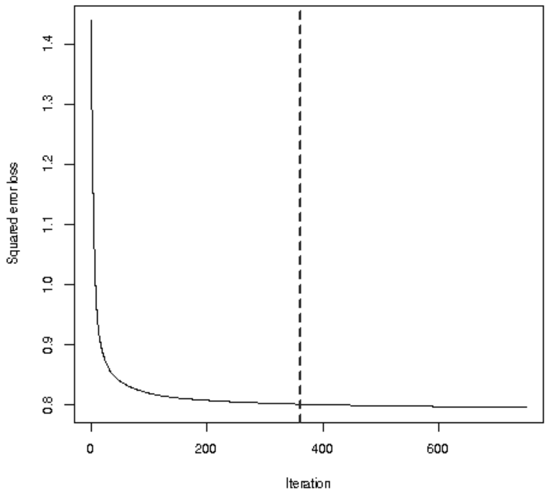


Figure 6-1: Output from `gbm.perf`

But we need not refit the model. We can change the number of trees in the prediction:

```
> predict(gbout,ds3,newNTree=382)
[1] 2.45214
```

Since we did not form a holdout set, we'll need to calculate MAPE manually:

```
> mean(abs(preds - ds[,8]))
[1] 0.6142699
```

Details on other features of the `gbm` package are available in its documentation.

6.3.4 Example: Vertebrae Data

Boosting can be used in classification settings as well as numeric-Y cases. (And its usage is probably much more common on the classification side.) Here is `qeGBoost()` applied to the the vertebrae data (Section 2.3).

```
> set.seed(9999)
> gbout <- qeGBoost(vert,'V7')
```

And, say we were to predict a new case like that of row 12 in the training set:

```
> predict(gbout,vert[12,-7])
$predClasses
[1] "DH"
```

```
$probs
            DH        NO          SL
[1,] 0.6283904 0.3694108 0.002198735

attr(,"class")
[1] "qeGBoost"
```

We predict DH, with an estimated probability of about 0.63. (Unfortunately, `gbm.perf()` is not available for the multiclass case.)

6.3.5 Bias vs. Variance in Boosting

Boosting is "tweaking" a tree, potentially making it more stable, especially since we are averaging many trees, thus smoothing out "For want of a nail . . ." problems. So, it may reduce variance. By making small adjustments to a tree, we are potentially developing a more detailed analysis, thus reducing bias.

But all of that is true only "potentially." Though the tweaking process has some theoretical basis, it still can lead us astray, actually *increasing* bias and possibly increasing variance too. If the hyperparameter s is set too large, producing too many trees, we may overfit.

6.3.6 Computational Speed

Boosting can take up tons of CPU cycles, so we may need something to speed things up. The `n.cores` argument in `gbm()` tries to offload computation to different cores in your machine. If you have a quad core system, you may try setting this argument to 4, or even 8 (and then call `gbm()` directly rather than through `qeGBoost()`).

6.3.7 Further Hyperparameters

Boosting algorithms typically have a number of hyperparameters. We have already mentioned `nTree` (`n.trees` in `gbm()`), which is the number of trees to be generated.

Another hyperparameter is `minNodeSize` (`n.minobsinnode` in `gbm()`), which is the minimum number of data points we are willing to have in one tree node. As we saw in Chapter 5, reducing this value will reduce bias but increase variance.

The `shrinkage` hyperparameter is so important in the general ML context that we'll cover it in a separate subsection, next.

6.3.8 The Learning Rate

The notion of a *learning rate* comes up often in ML. We'll describe it here in general and then explain how it works for gradient boosting. We'll see it again in our material on support vector machines (Chapter 10) and neural networks (Chapter 11).

This section has a bit of math in it, in the form of curves and lines tangent to them, which is an exception to the avowedly nonmathematical nature of this book. But there are still no equations, and even math-averse readers should be able to follow the discussion.

6.3.8.1 General Concepts

Recall that in ML methods we are usually trying to minimize some loss function, such as MAPE, or the overall misclassification error OME. Computationally, this minimization can be a challenge.

Consider the function graphed in Figure 6-2. It is a function of a one-dimensional variable x, whereas typically our x is high-dimensional, but it will make our point.

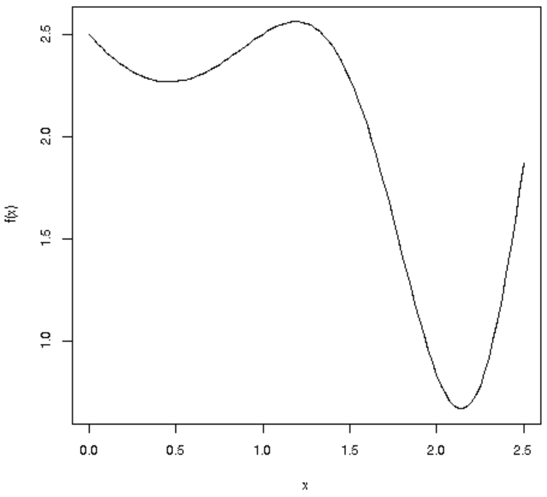


Figure 6-2: A function to be minimized

There is an overall minimum at approximately $x = 2.2$. This is termed the *global minimum*. But there is also a *local minimum*, at about $x = 0.4$; that term means that this is the minimum value of the function only for points near—"local to"— 0.4. Let's give the name x_0 to the value of x at the global minimum.

To us humans looking at the graph, it's clear where x_0 is, but we need our software to be able to find it. That may be problematic. Here's why:

Most ML algorithms use an *iterative* approach to finding the desired minimum point x_0. This involves a series of guesses for x_0. The code starts with an initial guess, g_0, say, randomly chosen, then evaluates $f(g_0)$. Based on the result, the algorithm then somehow (see below) updates the guess to g_1. It then evaluates $f(g_1)$, producing the next guess, g_2, and so on.

The algorithm keeps generating guesses until they don't change much, say, until $|g_{i+1} - g_i| < 0.00000001$ for step i. We say that the algorithm has *converged* to this point. Let's give the name c to that value of i. It then reports x_0, the global minimum point, to be the latest guess, g_c.

So, what about that "somehow" alluded to above? How does the algorithm generate the next guess from the present one? The answer lies in the *gradient*. In our simple example here with x being one-dimensional, the gradient is the slope of the function at the given point—that is, the slope of the tangent line to the curve.

Say our initial guess $g_0 = 1.1$. The tangent line is shown in Figure 6-3. The line is pointing upward to the right—that is, it has positive slope—so it tells us that by going to the left we will go to smaller values of the function. We want to find the point at which *f*() is smallest, and the tangent line is saying, "Oh, you want a smaller value than *f*(1.1)? Move to the left!" But actually we should be moving to the right, toward 2.2, where the global minimum is.

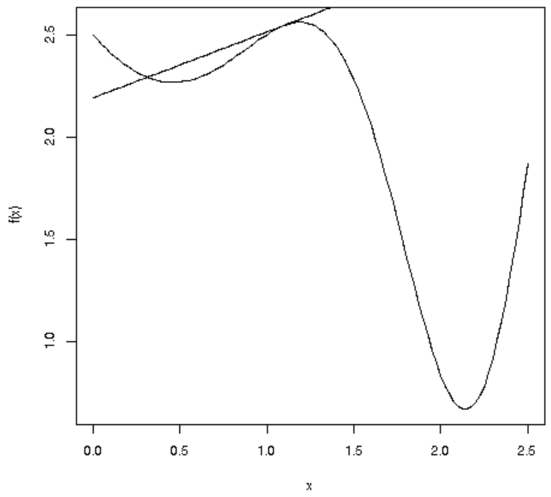


Figure 6-3: Function to be minimized, plus tangent

So, the reader can already see that iterative algorithms are fraught with danger. Worse, it also adds yet another hyperparameter: we must decide not only in *which direction* to move for our next guess but also *how far* to move in that direction. The *learning rate* addresses the latter point.

As noted, we should be moving to the right from 1.1, not to the left. The function $f(x)$ is fooling the algorithm here. Actually, in this scenario, our algorithm may converge to the wrong point. Or it may not even converge at all and just wander aimlessly.

This is why typical ML packages allow the user to set the learning rate. Small values may be preferable, as large ones may result in our guesses lurching back and forth, always missing the target. On the other hand, if it is too small, we will just inch along, taking a long time to get there. Or worse, we converge to a local minimum.

Once again, we have a hyperparameter that we need to be at a "Goldilocks" level—not too large and not too small—and may have to experiment with various values.

6.3.8.2 The Learning Rate in gbm

This is the `shrinkage` argument in `gbm()`, called `learnRate` in `qeGBoost()`. Say we set it to 0.2. Recall the pseudocode describing gradient boosting in Section 6.3.2. The revised version is this:

1. Start with an initial tree. Set *CurrentTree* to it.
2. For each of our data points, calculate the residuals for *CurrentTree*.
3. Fit a tree *to the residuals*—that is, take our residuals as the "data" and fit a tree T on it. Set *CurrentTree* to the old *CurrentTree*, plus `shrinkage * T`.
4. Go to Step 2.

Here, `shrinkage * T` means multiplying all the values in the terminal nodes of the tree by the factor `shrinkage`. In the end, we still add up all our trees to produce the "supertree" used in prediction of new cases.

Again, a small value of `shrinkage` is more cautious and slower, and it may cause us to need more trees in order to get good predictive power. But it may help prevent overfitting.

6.4 Pitfall: No Free Lunch

There is no such thing as a free lunch.
—Old economics saying

Though Leo Breiman had a point on the considerable value of AdaBoost (especially in saying "off the shelf," meaning usable with just default values of hyperparameters), that old saying about no free lunch applies as well. As always, applying cross-validation and so on is indispensable to developing good models.

Similar advice concerns another famous Breiman statement: that it is impossible to overfit using random forests. The reader who has come this far in this book will immediately realize that Breiman did not mean his statement in the way some have interpreted it. Any ML method may overfit. What Breiman meant was that it is impossible to set the value of s, the number of trees, too high. But the trees themselves still can overfit, for example, by having too small a minimum value for the number of data points in a node or, for that matter, by including too many features.