using N-fold (or leave-one-out) cross-validation and computed at virtually no extra cost to the fitting algorithm. However, as stated in Section 7.3, the OOB approach can provide overly pessimistic estimates of the true error, but can still be used for hyperparameter tuning [Janitza and Hornung, 2018]. Since GBMs have lots of tuning parameters, the OOB approach provides a computationally feasible solution to selecting a reasonable learning rate, number of trees, etc.

It's important to note that Janitza and Hornung [2018] refer specifically to OOB-based error estimates for RFs, not GBMs. To this day, I have yet to see an extensive study on the usefulness of OOB-based error estimates in GBMs compared to more traditional cross-validation approaches.

8.4.1 Column subsampling

Column subsampling is another technique that can be used to improve model performance and speed up fitting. Similar to column subsampling in an RF, a subsample of columns can be used for building each individual tree[j]. Apparently subsampling the columns prior to building each tree, can reduce the chances of overfitting even more than traditional row subsampling [Chen and Guestrin, 2016].

To illustrate, consider the test MSE curves for the ALS data displayed in Figure 8.3. In this example, subsampling the columns appears to outperform subsampling the rows (here, I arbitrarily chose a subsampling rate of 0.3). In practice these parameters need to be tuned, but it's probably safe and more computationally efficient in practice to just deal with one of these two hyperparameters. If you're dealing with a really wide data set, it may be more efficient to consider column subsampling, or both column subsampling and row subsampling if you have many rows as well.

8.5 Gradient tree boosting from scratch

Let's implement a quick-and-dirty gradient tree boosting function based on LS loss. The function, called `lsboost()`, is available in package **treemisc** (see `?treemisc::lsboost` for details and a description of the arguments), but the code is relatively straightforward and reproduced in the code chunk below. It's

[j]While similar, an RF chooses a random subsample of features prior to each split of every tree.

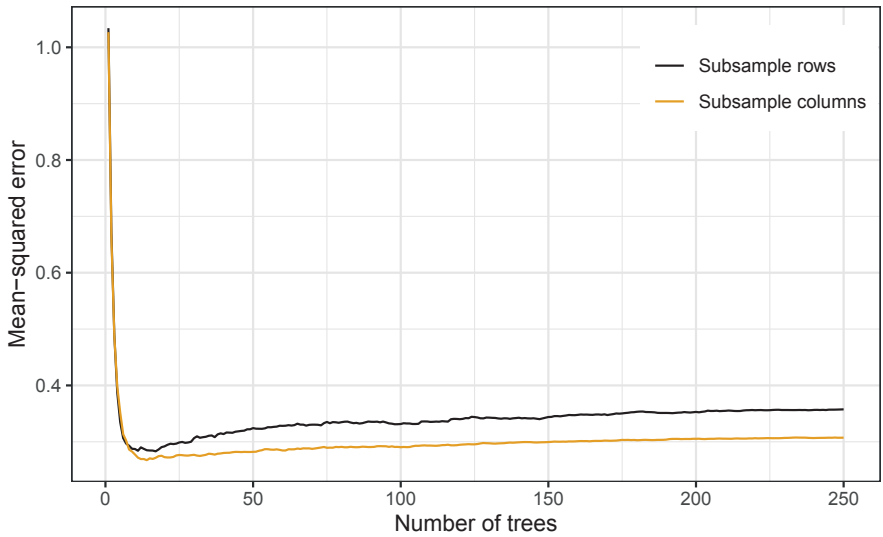

FIGURE 8.3: Effect of subsampling in GBMs on the ALS data. In this case, randomly subsampling the columns (yellow curve) slightly outperforms randomly subsampling the rows (black curve).

worth noting a few things about `lsboost()` and `predict.lsboost()`, before we continue:

- the code uses R's built-in S3 *object-oriented (OO) programming* system [Wickham, 2019, Chap. 13], which allows us to extend R's built-in `predict()` generic via the `predict.lsboost()` function (e.g., so we can compute predictions with, say, `predict(my.lsboost.model, newdata = some.data)`);
- `lsboost()` uses **rpart** to fit the individual regression trees, but other implementations could be used instead (e.g., CTrees via **partykit**);
- `lsboost()` returns an object of class `"lsboost"`, which is essentially a list of **rpart** trees that the `predict()` function knows how to combine;
- these functions are for illustration and not meant for serious use—they are not optimized in any sense.

If you cut out the fluff, gradient tree boosting, at least with LS loss, can be implemented in as little as 10 lines of code (probably less):

```
lsboost <- function(X, y, ntree = 100, shrinkage = 0.1, depth = 6,
                    subsample = 0.5, init = mean(y)) {
  yhat <- rep(init, times = nrow(X))  # initialize fit; f_0(x)
  trees <- vector("list", length = ntree)  # to store each tree
  ctrl <-  # control tree-specific parameters
```

```
    rpart::rpart.control(cp = 0, maxdepth = depth, minbucket = 10)
  for (tree in seq_len(ntree)) {  # Step 2) of Algorithm 8.1
    id <- sample.int(nrow(X), size = floor(subsample * nrow(X)))
    samp <- X[id, ]  # random subsample
    samp$pr <- y[id] - yhat[id]  # pseudo residual
    trees[[tree]] <-  # fit tree to current pseudo residual
      rpart::rpart(pr ~ ., data = samp, control = ctrl)
    yhat <- yhat + shrinkage * predict(trees[[tree]], newdata = X)
  }
  res <- list("trees" = trees, "shrinkage" = shrinkage,
              "depth" = depth, "subsample" = subsample, "init" = init)
  class(res) <- "lsboost"
  res
}

# Extend R's generic predict() function to work with "lsboost" objects
predict.lsboost <- function(object, newdata, ntree = NULL,
                            individual = FALSE, ...) {
  if (is.null(ntree)) {
    ntree <- length(object[["trees"]])  # use all trees
  }
  shrinkage <- object[["shrinkage"]]  # extract learning rate
  trees <- object[["trees"]][seq_len(ntree)]
  pmat <- sapply(trees, FUN = function(tree) {  # all predictions
    shrinkage * predict(tree, newdata = newdata)
  })  # compute matrix of (shrunken) predictions; one for each tree
  if (isTRUE(individual)) {
    pmat  # return matrix of (shrunken) predictions
  } else {
    rowSums(pmat) + object$init  # return boosted predictions
  }
}
```

Gradient tree boosting with LS loss is simpler to implement because there's no need to perform the line search step in Algorithm 8.1 (i.e., the terminal node estimates are already optimal). A slightly more complicated function that also implements gradient tree boosting with **rpart**, but using LAD loss, is shown below; this function is also part of **treemisc** (see `?treemisc::ladboost` for details). Here, care needs to be taken to update the terminal node summaries accordingly (see the commented section starting with `# Line search`). For LAD loss, we simply use the terminal node sample medians, as discussed in Section 8.2; here, I update the `frame` component of the **rpart** tree, but **partykit** could also be used, as illustrated in the commented out section. Also, note that the initial fit (`init`) defaults to the median response as well.

```
ladboost <- function(X, y, ntree = 100, shrinkage = 0.1, depth = 6,
                     subsample = 0.5, init = median(y)) {
  yhat <- rep(init, times = nrow(X))  # initialize fit
  trees <- vector("list", length = ntree)  # to store each tree
```

```
  ctrl <-  # control tree-specific parameters
    rpart::rpart.control(cp = 0, maxdepth = depth, minbucket = 10)
  for (tree in seq_len(ntree)) {
    id <- sample.int(nrow(X), size = floor(subsample * nrow(X)))
    samp <- X[id, ]
    samp$pr <- sign(y[id] - yhat[id])  # use signed residual
    trees[[tree]] <-
      rpart::rpart(pr ~ ., data = samp, control = ctrl)
    #---------------------------------------------------------------
    # Line search; update terminal node estimates using median
    #---------------------------------------------------------------
    where <- trees[[tree]]$where  # terminal node assignments
    map <- tapply(samp$pr, INDEX = where, FUN = median)
    trees[[tree]]$frame$yval[where] <- map[as.character(where)]
    #
    # Could use partykit instead:
    #
    # trees[[tree]] <- partykit::as.party(trees[[tree]])
    # med <- function(y, w) median(y)  # see ?partykit::predict.party
    # yhat <- yhat +
    #   shrinkage * partykit::predict.party(trees[[tree]],
    #                                       newdata = X, FUN = med)
    #---------------------------------------------------------------
    yhat <- yhat + shrinkage * predict(trees[[tree]], newdata = X)
  }
  res <- list("trees" = trees, "shrinkage" = shrinkage,
              "depth" = depth, "subsample" = subsample, "init" = init)
  class(res) <- "ladboost"
  res
}
```

8.5.1 Example: predicting home prices

Let's apply the `lsboost()` function to the Ames housing data. Below, I use the same train/test split for the Ames housing data we've been using throughout this book, then call `lsboost()` to fit a GBM to the training set; here, I'll use a shrinkage factor of $\nu = 0.1$:

```
library(treemisc)

# Split Ames data into train/test sets using a 70/30 split
ames <- as.data.frame(AmesHousing::make_ames())
ames$Sale_Price <- ames$Sale_Price / 1000  # rescale response
set.seed(4919)  # for reproducibility
id <- sample.int(nrow(ames), size = floor(0.7 * nrow(ames)))
ames.trn <- ames[id, ]
ames.tst <- ames[-id, ]
```

```
# Fit a gradient tree boosted ensemble with 500 trees
set.seed(1110)  # for reproducibility
ames.bst <-
  lsboost(subset(ames.trn, select = -Sale_Price),  # features only
          y = ames.trn$Sale_Price, ntree = 500, depth = 4,
          shrinkage = 0.1)
```

The test RMSE as a function of the number of trees in the ensemble is computed below using the previously defined `predict()` method; the results are shown in Figure 8.4 (black curve). For brevity, the code uses `sapply()` to essentially iterate cumulatively through the $B = 500$ trees and computes the test RMSE for the first tree, first two trees, etc. For comparison, the test RMSEs from a default RF are also computed and displayed in Figure 8.4 (yellow curve). In this example, the GBM slightly outperforms the RF.

```
set.seed(1128)  # for reproducibility
ames.rfo <-  # fit a default RF for comparison
  randomForest(subset(ames.trn, select = -Sale_Price),
               y = ames.trn$Sale_Price, ntree = 500,
               # Monitor test set performance (MSE, in this case)
               xtest = subset(ames.tst, select = -Sale_Price),
               ytest = ames.tst$Sale_Price)

# Helper function for computing RMSE
rmse <- function(pred, obs, na.rm = FALSE) {
  sqrt(mean((pred - obs)^2, na.rm = na.rm))
}

# Compute RMSEs from both models on the test set as a function of the
# number of trees in each ensemble (i.e., B = 1, 2, ..., 500)
rmses <- matrix(nrow = 500, ncol = 2)  # to store results
colnames(rmses) <- c("GBM", "RF")
rmses[, "GBM"] <- sapply(seq_along(ames.bst$trees), FUN = function(B) {
  pred <- predict(ames.bst, newdata = ames.tst, ntree = B)
  rmse(pred, obs = ames.tst$Sale_Price)
})  # add GBM results
rmses[, "RF"] <- sqrt(ames.rfo$test$mse)  # add RF results
```

8.6 Interpretability

Interpreting GBMs is no different from any other nonparametric model. For example, Section 5.4 discussed how the individual tree-based importance scores (Section 2.8) can be aggregated across all the trees in an ensemble

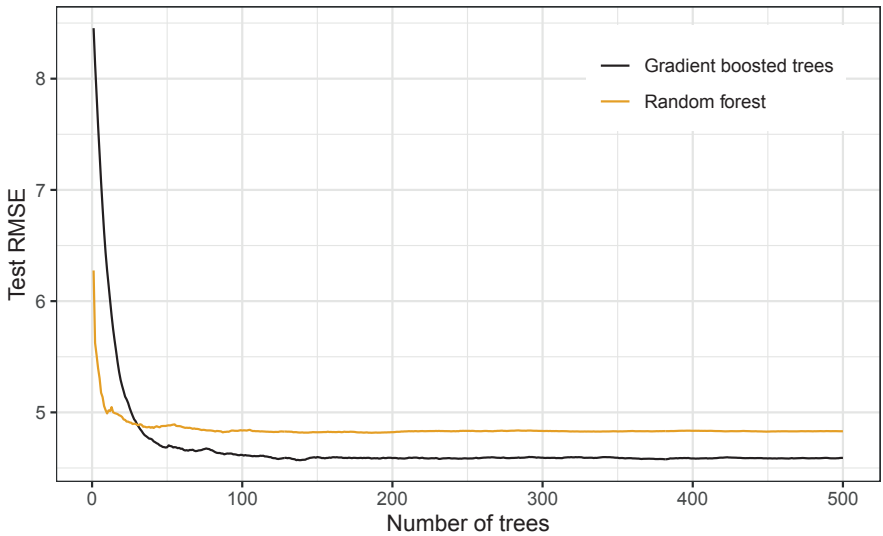

FIGURE 8.4: Root mean-squared error for the Ames housing test set as a function of B, the number of trees in the ensemble. Here, I show both a GBM (black curve) and a default RF (yellow curve). In this case, gradient tree boosting with LS loss, a shrinkage of $\lambda = 0.1$, and a maximum tree depth of $d = 4$ (black curve) slightly outperforms a default RF (yellow curve).

to form a more stable measure of predictor importance; however, as with CART and RFs, this measure is also biased for GBMs, although, the permutation importance method (Section 6.1.1) applies equally well to GBMs, or any supervised learning model, for that matter. PDPs and ICE plots can be used to visualize the global and local effect that subsets of features have on the model's predictions, respectively. Shapley values, among other techniques, can be used to infer the contribution each feature value has on the difference between its associated prediction and the model's baseline (or average training) prediction, which can also be used to generate global measures of both feature importance and feature effects. The next two sections discuss specialized interpretability techniques often associated with GBMs.

8.6.1 Faster partial dependence with the recursion method

For regression trees based on single-variable splits, Friedman [2001] described a fast procedure for computing the partial dependence of $\widehat{f}(\boldsymbol{x})$ on a subset of features using a weight traversal of each tree (henceforth referred to as the