details, see the description of the `respect.unordered.factors` argument in `?ranger::ranger`. The **party** and **partykit** packages offer an implementation using conditional inference trees as the base learners (Section 3.4) for the base learners; our `crforest()` function from Section 7.2.3 follows the same approach. The CRAN task view on "Machine Learning & Statistical Learning" includes a section dedicated to RFs in R, so be sure to check that out as well: `https://cran.r-project.org/view=MachineLearning`.

While **randomForest** is a close port of Breiman's original Fortran code, the **ranger** package is far more scalable and implements a number of modern extensions and improvements discussed in this chapter (e.g., RF as a probability machine, Gini-corrected importance, quantile regression, case-specific RFs, extra-trees, etc.). Another scalable implementation is available from **h2o** [LeDell et al., 2021]. RFs are also part of Spark's MLlib library [Meng et al., 2016], which includes several R and Python interfaces (in particular, **SparkR**, **sparklyr**, and **pyspark**; an example using **SparkR** is provided in Section 7.9.5).

In Python, RFs, extra-trees, and isolation forests are available in the **sklearn.ensemble** module. Julia users can fit RFs via the **DecisionTree.jl** package. The official GUIDE software (Section 4.9) has the option to construct an RF from individual GUIDE trees; see Loh et al. [2019] and Loh [2020] for details.

Let's now work through several problems using random forest software.

7.9.1 Example: mushroom edibility

No! These data are easy and an ensemble would be overkill here. Remember, the original goal of the problem was to come up with an accurate but simple rule for determining the edibility of a mushroom. This was easily accomplished using a single decision tree (e.g., CART with some manual pruning) or a rule-based model like CORELS; see, for example, Figure 2.22.

7.9.2 Example: "deforesting" a random forest

In Section 5.5, I showed how the LASSO can be used to effectively post-process a tree-based ensemble by essentially zeroing out the predictions from some of the trees and reweighting the rest. The idea is that we can often reduce the number of trees quite substantially without sacrificing much in the way of performance. A smaller number of trees means we could, at least in theory, compute predictions faster, which has important implications for model deployment (e.g., when trying to score large data sets on a regular basis). However, unless we have a way to remove the zeroed out trees from

the fitted RF object, we can't really reap all the benefits. This is the purpose of the new `deforest()` function in the **ranger** package[m], which I'll demonstrate in this section using the Ames housing example.

Keep in mind that this method of post-processing is not specific to bagged tree ensembles and RFs, and can be fruitfully applied to other types of ensembles as well; see Section 8.9.3 for an example using a *gradient boosted tree ensemble*.

To start, I'll load a few packages, prep the data, and create a helper function for computing the RMSE as a function of the number of trees in a **ranger**-based RF:

```
library(ranger)
library(treemisc)  # for isle_post() function

# Load the Ames housing data and split into train/test sets
ames <- as.data.frame(AmesHousing::make_ames())
ames$Sale_Price <- ames$Sale_Price / 1000  # rescale response
set.seed(2101)  # for reproducibility
trn.id <- sample.int(nrow(ames), size = floor(0.7 * nrow(ames)))
ames.trn <- ames[trn.id, ]  # training data/learning sample
ames.tst <- ames[-trn.id, ]  # test data
xtst <- subset(ames.tst, select = -Sale_Price)  # test features only

# Function to compute RMSE as a function of number of trees
rmse <- function(object, X, y) {  # only works with "ranger" objects
  p <- predict(object, data = X, predict.all = TRUE)$predictions
  sapply(seq_len(ncol(p)), FUN = function(i) {
    pred <- rowMeans(p[, seq_len(i), drop = FALSE])
    sqrt(mean((pred - y) ^ 2))
  })
}
```

Next, I'll fit two different RFs:

RFO a default RF with $B = 1,000$ maximal depth trees;

RFO.4.5 an RF with $B = 1,000$ shallow (depth-4) trees, where each tree is built using only a 5% random sample (with replacement) from the training data.

I'll record the computation time of each fit using `system.time()` (this function will also be used later to measure scoring time), which will provide some insight into the potential computational savings offered by this post-processing method[n]:

[m] The `deforest()` function is not available in versions of **ranger** $\leq$ 0.13.0.

[n] Note that there are better ways to benchmark and time expressions in R; see, for example, the **microbenchmark** package [Mersmann, 2021].

```
# Fit a default RF with 1,000 maximal depth trees
set.seed(942)  # for reproducibility
system.time({
  rfo <- ranger(Sale_Price ~ ., data = ames.trn, num.trees = 1000)
})
```

```
#>    user  system elapsed
#>   5.845   0.112   1.899
```

```
# Fit an RF with 1,000 shallow (depth-4) trees on 5% bootstrap samples
set.seed(1021)  # for reproducibility
system.time({
  rfo.4.5 <- ranger(Sale_Price ~ ., data = ames.trn, num.trees = 1000,
                    max.depth = 4, sample.fraction = 0.05)
})
```

```
#>    user  system elapsed
#>   0.275   0.009   0.113
```

```
# Test set MSE as a function of the number of trees
rmse.rfo <- rmse(rfo, X = xtst, y = ames.tst$Sale_Price)
rmse.rfo.4.5 <- rmse(rfo.4.5, X = xtst, y = ames.tst$Sale_Price)
c("Test RMSE (RFO)" = rmse.rfo[1000],
  "Test RMSE (RFO.4.5)" = rmse.rfo.4.5[1000])
```

```
#>     Test RMSE (RFO) Test RMSE (RFO.4.5)
#>                24.8                36.7
```

The test RMSE for the RFO model is comparable to the test RMSE from the conditional RF fit in Section 7.2.3. In comparison, the RFO.4.5 model has a much larger test RMSE, which we might have expected given the shallowness of each tree and the tiny fraction of the learning sample each was built from. Consequently, the RFO.4.5 model finished training in only a fraction of the time it took the RFO model. As we'll see shortly, post-processing will help improve the performance of RFO.4.5 so that it is comparable to RFO in terms of performance, while substantially reducing the number of trees (i.e., comparable performance, faster training time, and fewer trees in the end).

Next, I'll obtain the individual tree predictions from each forest and post-process them using the LASSO via **treemisc**'s `isle_post()` function. Note that k-fold cross-validation can be used here instead of (or in conjunction with) a test set; see `?treemisc::isle_post` for details. For brevity, I'll use a simple prediction wrapper, called `treepreds()`, to compute and extract the individual tree predictions from each RF model:

```
treepreds <- function(object, newdata) {
  p <- predict(object, data = newdata, predict.all = TRUE)
  p$predictions  # return predictions component
}

# Post-process RFO ensemble using an independent test set
```

```
preds.trn <- treepreds(rfo, newdata = ames.trn)
preds.tst <- treepreds(rfo, newdata = ames.tst)
rfo.post <- treemisc::isle_post(
  X = preds.trn,
  y = ames.trn$Sale_Price,
  newX = preds.tst,
  newy = ames.tst$Sale_Price,
  family = "gaussian"
)

# Post-process RFO.4.5 ensemble using an independent test set
preds.trn.4.5 <- treepreds(rfo.4.5, newdata = ames.trn)
preds.tst.4.5 <- treepreds(rfo.4.5, newdata = ames.tst)
rfo.4.5.post <- treemisc::isle_post(
  X = preds.trn.4.5,
  y = ames.trn$Sale_Price,
  newX = preds.tst.4.5,
  newy = ames.tst$Sale_Price,
  family = "gaussian"
)
```

The results are plotted in Figure 7.20. Here, we can see that both models benefited from post-processing, but the RFO model only experienced a marginal increase in performance compared to RFO.4.5. Is the slightly better performance in the default RFO model enough to justify its larger training time? Maybe in this particular example, but for larger data sets, the difference in training time can be huge, making it extremely worthwhile. For the post-processed RFO.4.5 model, the test RMSE is minimized using only 93 (reweighted) trees.

```
palette("Okabe-Ito")
plot(rmse.rfo, type = "l", ylim = c(20, 50),
     las = 1, xlab = "Number of trees", ylab = "Test RMSE")
lines(rmse.rfo.4.5, col = 2)
lines(sqrt(rfo.post$results$mse), col = 1, lty = 2)
lines(sqrt(rfo.4.5.post$results$mse), col = 2, lty = 2)
legend("topright", col = c(1, 2, 1, 2), lty = c(1, 1, 2, 2),
       legend = c("RFO", "RFO.4.5","RFO (post)", "RFO.4.5 (post)"),
       inset = 0.01, bty = "n")
palette("default")
```

To make this useful in practice, we need a way to remove trees from a fitted RF (i.e., to "deforest" the forest of trees). This could vastly speed up prediction time and reduce the memory footprint of the final model. Fortunately, the **ranger** package includes such a function; see `?ranger::deforest` for details.

In the code snippet below, I "deforest" the RFO.4.5 ensemble by removing trees corresponding to the zeroed-out LASSO coefficients, which requires es-

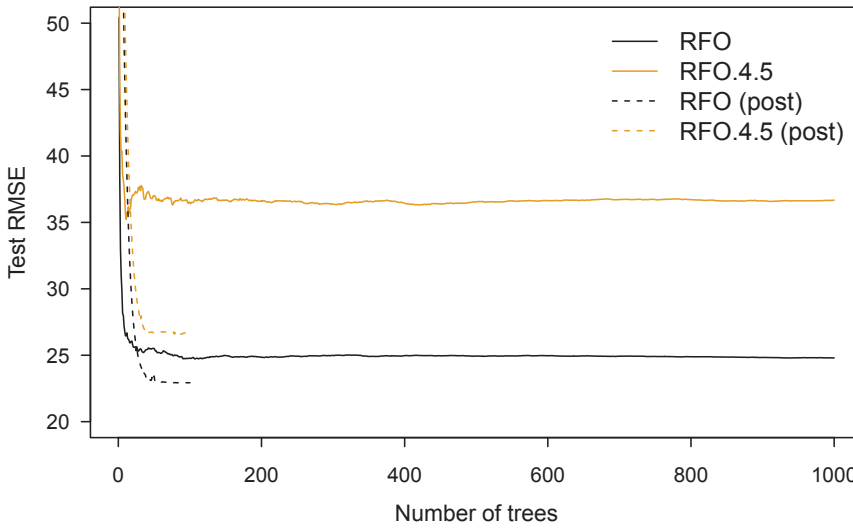

FIGURE 7.20: Test RMSE for the RFO and RFO.4.5 fits. The dashed lines correspond to the post-processed versions of each model. Note how the RFO model only experienced a marginal increase in performance compared to the RFO.4.5 model.

timating the optimal value for the penalty parameter λ (it might be helpful to read the help page for `?glmnet::coef.glmnet`):

```
res <- rfo.4.5.post$results  # post-processing results on test set
lambda <- res[which.min(res$mse), "lambda"]   # optimal penalty parameter
coefs <- coef(rfo.4.5.post$lasso.fit, s = lambda)[, 1L]
int <- coefs[1L]  # intercept
tree.coefs <- coefs[-1L]  # no intercept
trees <- which(tree.coefs == 0)  # trees to remove

# Remove trees corresponding to zeroed-out coefficients
rfo.4.5.def <- deforest(rfo.4.5, which.trees = trees)

# Check size of each object
c(
  "RFO.4.5" = format(object.size(rfo.4.5), units = "MB"),
  "RFO.4.5 (deforested)" = format(object.size(rfo.4.5.def), units = "MB")
)
```

```
#>              RFO.4.5 RFO.4.5 (deforested)
#>               "1 Mb"              "0.1 Mb"
```

Notice the impact this had on reducing the overall size of the fitted model. This can often lead to a much more compact model that's easier to save and load when memory requirements are a concern.

We can't just use the "deforested" tree ensemble directly; remember, the estimated LASSO coefficients imply a reweighting of the remaining trees! To obtain the reweighted predictions from the "deforested" model, we need to do a bit more work. Here, I'll create a new prediction function, called `predict.def()`, that will compute the reweighted predictions from the remaining trees using the estimated LASSO coefficients—similar to how predictions in a linear model are computed.

To test it out, I'll stack the learning sample (`ames.trn`) on top of itself 100 times, resulting in $N = 205,100$ observations for scoring. Below, I compare the prediction times for both the original (i.e., non-processed) and "deforested" RFO.4.5 fits:

```
ames.big <-  # stack data on top of itself 100 times
  do.call("rbind", args = replicate(100, ames.trn, simplify = FALSE))

# Compute reweighted predictions from a ``deforested'' ranger object
predict.def <- function(rf.def, weights, newdata, intercept = TRUE) {
  preds <- predict(rf.def, data = newdata,
                   predict.all = TRUE)$predictions
  res <- if (isTRUE(intercept)) {  # returns a one-column matrix
    cbind(1, preds) %*% weights
  } else {
    preds %*% weights
  }
  res[, 1, drop = TRUE]  # coerce to atomic vector
}

# Scoring time for original RFO.4.5 fit
system.time({  # full random forest
  preds <- predict(rfo.4.5, data = ames.big)
})
```

```
#>    user  system elapsed
#>   37.15    2.64   13.35
```

```
# Scoring time for post-processed RFO.4.5 fit using updated weights
weights <- coefs[coefs != 0]  # LASSO-based weights for remaining trees
system.time({
  preds.post <- predict.def(rfo.4.5.def, weights = weights,
                            newdata = ames.big)
})
```

```
#>    user  system elapsed
#>    4.17    0.73    4.47
```

The final model contains only 93 trees and achieved a test RMSE of 26.59, while also being orders of magnitude faster to initially train. The computational advantages are easier to appreciate on even larger data sets.

In summary, I used the LASSO to post-process and "deforest" a large ensemble of shallow trees (which trained relatively fast), producing a much smaller ensemble with fewer trees that scores faster compared to the default RFO. While the default RFO model had a slightly smaller test RMSE of 24.72 compared to the "deforested" RFO.4.5 test RMSE of 111.29, the difference is arguably negligible (especially when you take the differences in both training and scoring time into account).

7.9.3 Example: survival on the Titanic

In this example, I'll walk through a simple RF analysis of the well-known Titanic data set, where the goal is to understand survival probability aboard the ill-fated Titanic. A more thoughtful analysis using logistic regression and spline-based techniques is provided in Harrell [2015, Chap. 12].

Several versions of this data set are publicly available; for example, in the R package **titanic** [Hendricks, 2015]. Here, I'll use a more complete version of the data[o] which can be loaded using the `getHdata()` from package **Hmisc** [Harrell, 2021]; the raw data can also be downloaded from `https://hbiostat.org/data/`. In this example, I'll only consider a handful of the original variables:

```
t3 <- read.csv("https://hbiostat.org/data/repo/titanic3.csv",
               stringsAsFactors = TRUE)
keep <- c("survived", "pclass", "age", "sex", "sibsp", "parch")
t3 <- t3[, keep]  # only retain key variables
```

Note that roughly 20.09% of the values for `age`, the age in years of the passenger, are missing:

```
sapply(t3, FUN = function(x) mean(is.na(x)))
```

```
#> survived   pclass      age      sex    sibsp    parch
#>    0.000    0.000    0.201    0.000    0.000    0.000
```

Following Harrell [2015, Sec. 12.4], I use a decision tree to investigate which kinds of passengers tend to have a missing value for `age`. In the example below, I use the **partykit** package to apply the CTree algorithm (Chapter 3) using a missing value indicator for `age` as the response. From the tree output we can see that third-class passengers had the highest rate of missing `age` values (29.3%), followed by first-class male passengers with no siblings or

[o] A description of the original source of these data is provided in Harrell [2015, p. 291].