Bagging & Random Forests

D2K Course Staff

4/1/2019

Introduction

In order to understand *bagging* (bootstrap + aggregating), we need to understand about the theory of *ensembling methods*. The main idea is that we are able to decrease the variance in our predictions by averaging the predictions of many different models. Bagging is an ensembling method that works by reducing the variance in the predictions. Therefore, bagging individual models that have high variance, but *low bias* (like decision trees) is particularly effective. Our results can be further improved by the de-correlation of the bagged models.

Bagging is just one form of ensembling, and has been shown to be outperformed by methods like boosting in most use cases. However, *random forests* incorporate other steps into the bagging algorithm to further reduce variabnce which we will cover later in this module. Random forests are one of the most widely used predictive models used in industry today and have been shown to perform very well on many different types of data sets.

Bagging

Bootstrap

The first part of the bagging algorithm is to generate *bootstrap* samples from our data. Let us suppose that we have a sample S from our population P. We compute estimators from our sample, but we aren't able to gauge our measurement error given the fact that we only have one sample from which to compute our data. Bootstrapping makes the assumption that the sample S is representative of the population P (i.e. $S \equiv P$), and by doing so allows us to generate independent samples $S_1, S_2, ..., S_M$ from the original sample S. Essentially, we are just re-sampling the data from the original sample **with** replacement in order to keep the samples independent.

Aggregating

After generating our independent samples $S_1, S_2, ..., S_M$, we fit our base learning models on each sample. After making all of our predictions, we can then **aggregate** our predictions into a final prediction $G(X)$ where

$$G(X) = \Sigma_m \frac{G_m(X)}{M}$$

is a method for "averaging" our predictions across all of the base models. For regression models, this can be an actual average, and for classification problems, it is often the majority vote. By training each base model on separate bootstrapped samples of our data, we can decorrelate our base learner predictions and thus reduce the final variance of our bagged model. Furthermore, during each iteration we automatically generate validation sets called *out-of-bag* samples consisting of all of the observations that were not selected in the specific bootstrap sample.

Even though we lose some interpretability by bagging simpler models like decision trees, we can still make inference through techniques like variable importance measure. This technique keeps track of all of the prediction errors throughout all of the bootstrapped models and calculates the decrease in predictive accuracy when a model does not split on a certain feature.

Random Forests

Hyper-parameters

Random Forests take the concept of bagged decision trees and extend bootstrapping to the feature space as well. Instead of searching over all p features for a base decision tree model, we only search through a random subset of k features where $k \in p$. This further decreases the correlation of the base learners and therefore reduces the variance of our final bagged model.

There are a handful of hyper-parameters that should be tuned before reporting the prediction accuracy of your final random forest model. The `randomForest` package gives us a nice framework to perform a grid search over the different combinations of these hyper-parameters, but we will cover what each of the hyper-parameter represents first.

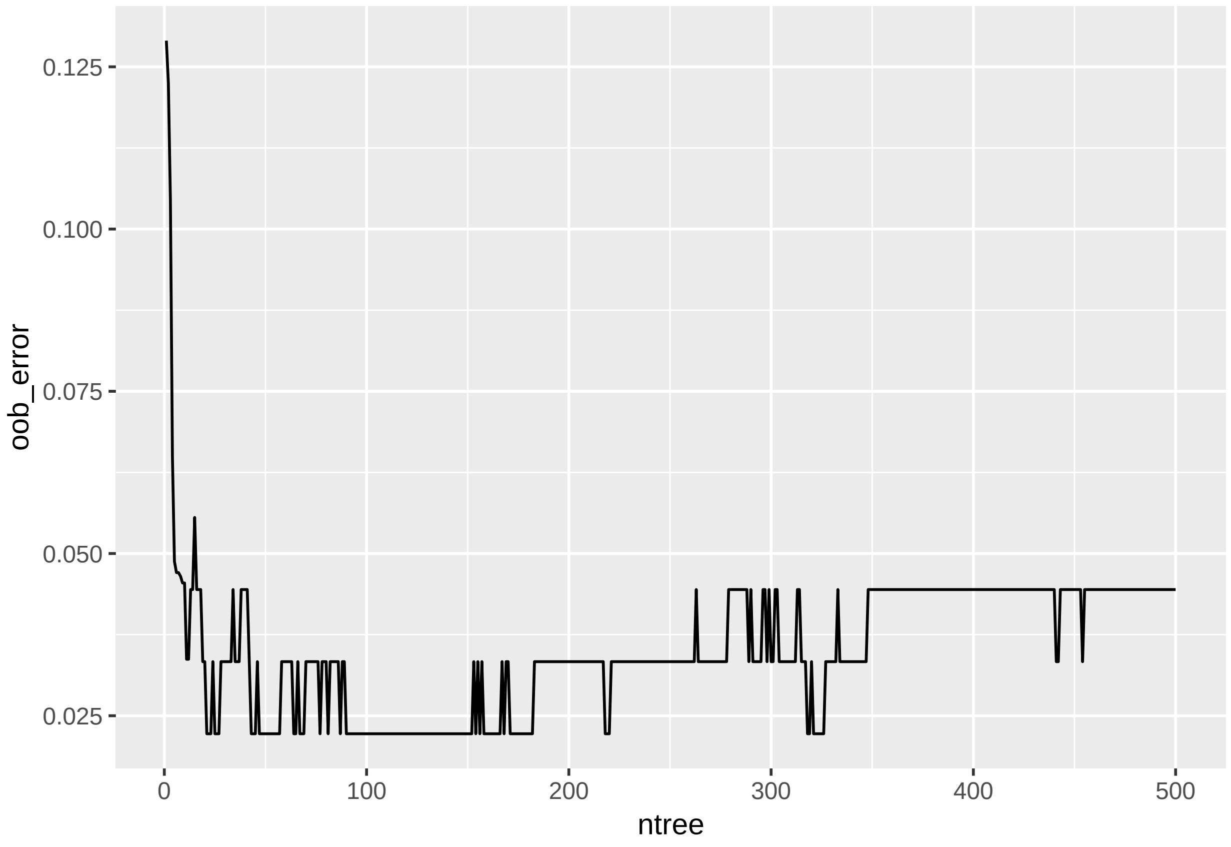
- `ntree`: number of trees to aggregate our results over. This is equivalent to the number of bootstrap samples we generate in our training process.
- `mtry`: the number of features to sample for each decision tree. The default value is $\sqrt{p}$, but a good range of values to search over is 2 to p.
- `sampsize`: the percentage of observations we sample for each bootstrap sample S_k to train on. The default value is .6325 as it is the expected number of unique observations in each sample. Lower values reduce the runtime, but can increase bias. In contrast, increasing the value increases the risk of overfitting by inducing too much variance.
- `nodesize`: the minimum number of samples within the terminal nodes. This determines how complex the tree is, smaller `nodesize` gives us deeper more complex trees while larger `nodesize` gives us shallower, less complex trees.

Example

Now, if we were to grid search over wide arrays of all possible combinations of the hyper-parameters listed above, we would be sitting in front of our computer waiting for a **long** time. However, we can first figure out the number of trees, or `ntree` where our error stops decreasing. We can then tune our other hyper-parameters after establishing a reasonable value. We will re-visit our old trusty `iris` data set to perform our classification task. First, we load the data and tune for the number of trees by plotting the prediction error vs. the number of trees.

```r
# Load data and necessary packages
library(randomForest)
data(iris)

# Set seed for reproducibility
set.seed(123)

# Create training/test index
tr_idx <- sample(1:nrow(iris), nrow(iris) * 0.6, replace = FALSE)

# Create training/validation set
iris_train <- iris[tr_idx, ]

iris_test <- iris[-tr_idx, ]
```

```r
# Run Random Forest model without tuning
tree_tune <- randomForest(formula = Species ~ ., data = iris_train)

# Build data frame for errors to plot
tree_tune_err <- data.frame(ntree = 1:length(tree_tune$err.rate[,
    1]), oob_error = tree_tune$err.rate[, 1])

# Plot error
ggplot(data = tree_tune_err, aes(x = ntree, y = oob_error)) +
    geom_line()
```



We can see that the out-of-bag error rate is at it's lowest in the range of ~50-150 trees. Since we get more stable results using more trees, we will use the upper bound of the range of `ntree` that still minimizes our error (let's say `ntree` = 150).

Now that we have determined an appropriate number of trees to grow for our random forest model, we can go about tuning the other hyper-parameters using a grid search. We can again use the naturally occuring out-of-bag samples to evaluate our prediction error as we train our model.

```r
# Create hyper-parameter grid
tune_grid <- expand.grid(mtry = seq(1, 4), sampsize = round(c(0.55,
    0.632, 0.7, 0.75, 0.8) * nrow(iris_train)), nodesize = seq(3,
    9, by = 3))

# Print number of grids to search over
nrow(tune_grid)
```

```
[1] 60
```

We can see that using our current configurations we will have to run the model 80 times. With a greater number of trees, this can get computationally very expensive.

```r
# Set seed for reproducibility
set.seed(123)

# Tune model using OOB error
for (i in 1:nrow(tune_grid)) {

    # train model
    grid_fit <- randomForest(formula = Species ~ ., data = iris_train,
        num.trees = 150, mtry = tune_grid$mtry[i], sampsize = tune_grid$sampsize[i],
        nodesize = tune_grid$nodesize[i])

    # Record OOB error
    tune_grid$OOB_error[i] <- grid_fit$err.rate[, 1]
}

# Print best combination of results
head(tune_grid[order(tune_grid$OOB_error), ])
```

```
   mtry sampsize nodesize OOB_error
8     4       57        3         0
20    4       72        3         0
38    2       72        6         0
48    4       57        9         0
51    3       63        9         0
54    2       68        9         0
```

We can see from the results from the cross-validation process that there are many different combinations of hyper-parameters that yield an out-of-bag-error of 0. We will simply choose the first combination of values, but this somewhat arbitrary as there is reasonable justification for any of the combinations that resulted in OOB error of 0.

Now, we retrain our model at the optimal values of our hyper-parameters and predict on our unseen test set. We will then report the final classification error on our test set.

```r
# Set seed for reproducibility
set.seed(123)

# Retrain random forest model on optimal hyper-parameter
# values
best_fit <- randomForest(formula = Species ~ ., data = iris_train,
    num.trees = 150, mtry = 4, sampsize = 57, nodesize = 3)

# Predict on test set
y_hat <- predict(best_fit, iris_test)

# Test misclassification error
test_error <- 1 - mean(y_hat == iris_test$Species)
test_error
```

```
[1] 0.06666667
```

We can see that our final model predicts pretty darn well on the test data. Along with prediction, we are able to make some inference using random forests. As previously mentioned, we can report the amount of predictive accuracy lost when we exclude a feature during the random forest training process. In the `randomForest` package, this is called **Mean Decrease in Gini** which is the amount that the Gini loss function decreases when that variable is excluded from a decision tree model. To be less convoluted, it is a measure of how important that variable is for prediction. We rank the most important features below.

```
# Show MeanDecreaseGini
best_fit$importance
```

```
             MeanDecreaseGini
Sepal.Length        0.1699578
Sepal.Width         0.1214376
Petal.Length       22.2596883
Petal.Width        13.9294886
```

From this output, we can see that **Petal.Length** is the most important feature for predicting **Species** in the `iris` dataset.

References

Boehmke, Bradley. "Random Forests." Random Forests · UC Business Analytics R Programming Guide, uc-r.github.io/random_forests.

Hastie, Trevor, et al. The Elements of Statistical Learning Data Mining, Inference, and Prediction. Springer, 2009.

Townshend, Raphael John Lamarre. "Ensembling Methods." CS229 Lecture Notes, Stanford University, cs229.stanford.edu/notes/cs229-notes-ensemble.pdf.