Lab 8: Bagging, Random Forests, and Boosting Trees

PSTAT 131/231, Fall 2023

Learning Objectives

- Bagged trees and random forest by randomForest()
- Variable importance by importance() and varImpPlot()
- Boosting by gbm()

In Lab 7 - Decision Trees, we used classification trees to analyze the Carseats data set. In this dataset, Sales is a continuous variable, and so we begin by recoding it as a binary variable. We use the `ifelse()` function to create a variable, called *High*, which takes on a value of *Yes* if the Sales variable exceeds the median of Sales, and takes a value *No* otherwise.

```
library(dplyr)
#install.packages("randomForest")
library(randomForest)
#install.packages("gbm")
library(gbm)
library(ISLR)
library(tree)
```

In this lab we will use the same dataset.

```
attach(Carseats)
Carseats = Carseats %>%
    mutate(High=as.factor(ifelse(Sales <= median(Sales), "No", "Yes"))) %>%
    select(-Sales)
```

Note that here we directly drop the `Sales` variable. In total, there are 10 predictor, which is used to predict the response variable `High`.

Identical to what we did in Lab 7, we split the data into training (75% data) and test set (25% data).

```
# Sample 75% observations as training data
set.seed(3)
train = sample(nrow(Carseats), 0.75*nrow(Carseats))
train.carseats = Carseats[train,]

# The rest as test data
test.carseats = Carseats[-train,]
```

2. Bagging

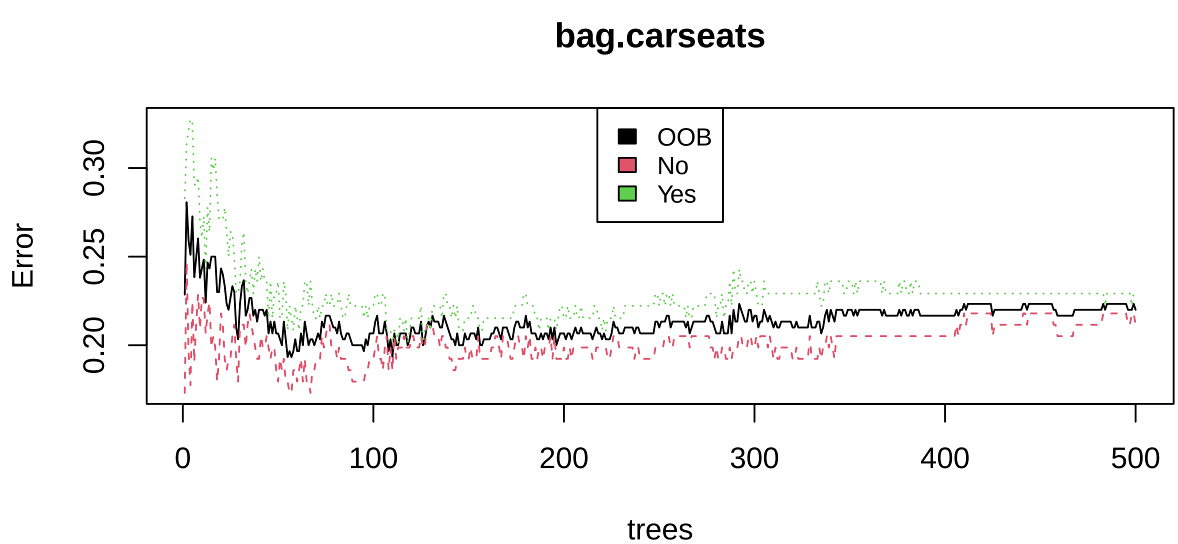
In the following, we apply bagging and random forests to the Carseats data, using the **`randomForest`** package in R and compare the same metric for bagged tree and random forest. Note that the exact results obtained in this section may depend on the version of R and the version of the **`randomForest`** package installed on your computer. Recall that bagging is simply a special case of a random forest with `m = p`. Therefore, the **`randomForest()`** function (again, same function and package names) can be used to perform both random forests and bagging. We build a random forest as follows:

```r
bag.carseats = randomForest(High ~ ., data=train.carseats,
                            mtry=10, importance=TRUE)
bag.carseats
```

```
##
## Call:
##  randomForest(formula = High ~ ., data = train.carseats, mtry = 10,      importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 10
##
##         OOB estimate of  error rate: 22%
## Confusion matrix:
##      No Yes class.error
## No  123  33   0.2115385
## Yes  33 111   0.2291667
```

The argument `mtry=10` indicates that 10 predictors (which is the total number of predictors) should be considered for each split of the tree - recall that is exactly the bagging, i.e., a special case of random forests when $m = p$. The argument `importance=TRUE` tells whether independent variable importance in bagged trees should be assessed.

```r
plot(bag.carseats)
legend("top", colnames(bag.carseats$err.rate),col=1:4,cex=0.8,fill=1:4)
```



How well does this bagged model perform on the test set?

```r
yhat.bag = predict(bag.carseats, newdata = test.carseats, type = "response")
test.bag.err = mean(yhat.bag != test.carseats$High)
test.bag.err
```

```
## [1] 0.17
```

By default (i.e., with `type = "response"`), **`predict`** function defined for the `randomForest` object yields exact classes (in this case, `Yes` and `No`). To get the predicted probability, we need to specify `type = "prob"` in **predict**.

```
prob.bag = predict(bag.carseats, newdata = test.carseats, type = "prob")
head(prob.bag)
```

```
##       No   Yes
## 11 0.332 0.668
## 17 0.080 0.920
## 18 0.058 0.942
## 21 0.958 0.042
## 25 0.146 0.854
## 31 0.024 0.976
```

Note that the returned predicted probability has two columns: the probability for `No` and the probability for `Yes`.

The predict function for `randomForest` then classifies the response based on the probability. In this example, if the predicted probability of `Yes` is greater than the probability of `No`, then the predicted class is `Yes`.

```
all(yhat.bag == ifelse(prob.bag[, 2] > 0.5, "Yes", "No"))
```

```
## [1] TRUE
```

The test set error rate associated with the bagged classification tree is 0.17, lower than that obtained using an optimally-pruned single tree that we've seen in Lab 7. You may consider this a minor improvement, however there are many cases that the improvement could be as half. We could change the number of bagged trees grown by **randomForest()** using the **ntree** argument. For simplicity of output, we set the following code chunk option as `eval=FALSE`.

```
bag.carseats = randomForest(High ~ .,
                            data=train.carseats,
                            mtry=10, ntree=700, importance=TRUE)
yhat.bag = predict(bag.carseats, newdata = test.carseats)

test.bag.err = mean(yhat.bag != test.carseats$High)
test.bag.err
```

3. Random Forests

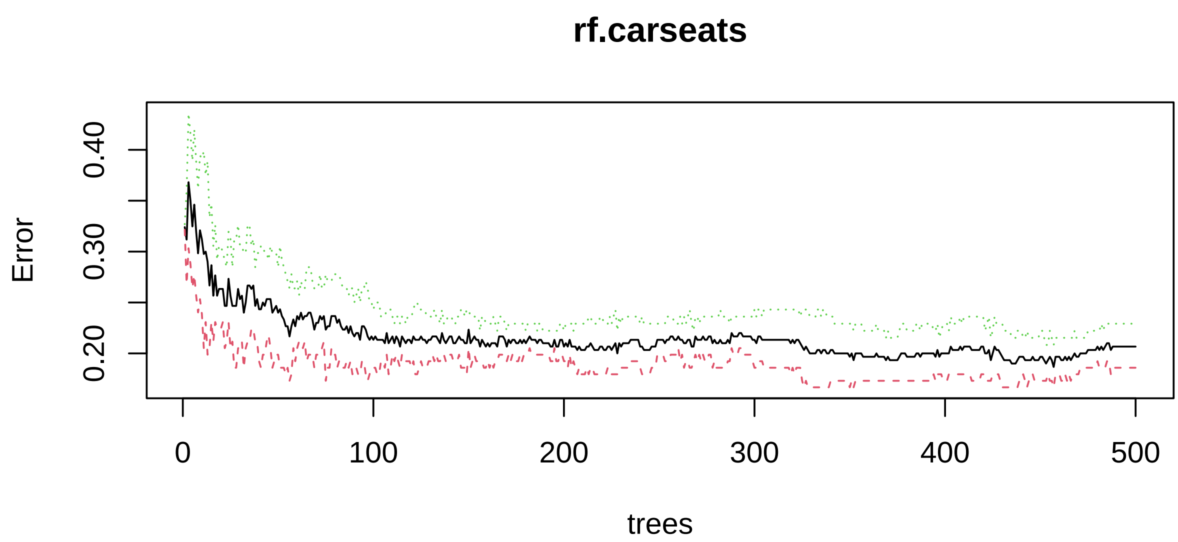
Growing a random forest proceeds in exactly the same way, except that we use a smaller value of the **mtry** argument. By default, **randomForest()** uses p/3 variables when building a random forest of regression trees, and $\sqrt{p}$ variables when building a random forest of classification trees. Here we use `mtry = 3`.

```
rf.carseats = randomForest(High ~ ., data=train.carseats,
                           mtry=3, importance=TRUE)
rf.carseats
```

```
##
## Call:
##  randomForest(formula = High ~ ., data = train.carseats, mtry = 3,      importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 20.67%
## Confusion matrix:
##      No Yes class.error
## No  127  29   0.1858974
```

```
## Yes  33 111   0.2291667
```

```r
plot(rf.carseats)
```




```r
yhat.rf = predict(rf.carseats, newdata = test.carseats)
test.rf.err = mean(yhat.rf != test.carseats$High)
test.rf.err
```

```
## [1] 0.21
```

The test set error rate is 0.21; this indicates that random forests did not provide an improvement over bagging in this case.

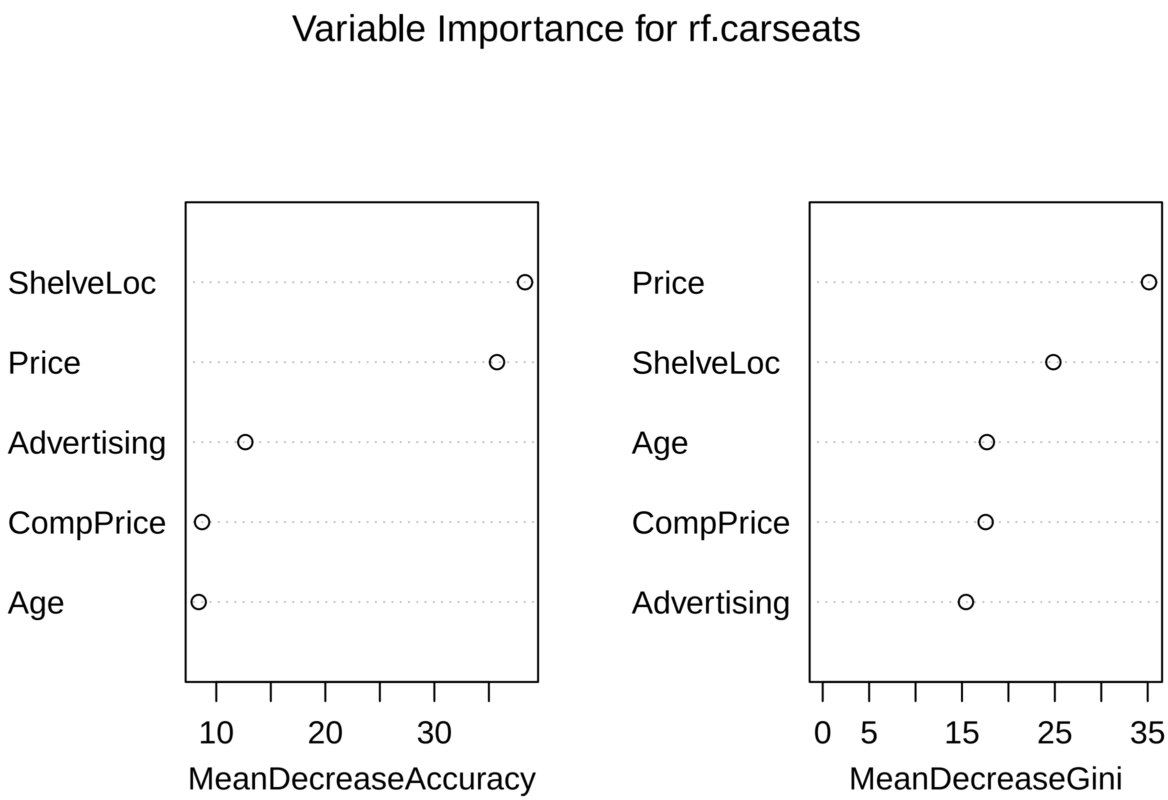
Using the **importance()** function, we can view the importance of each **importance()** variable.

```r
importance(rf.carseats)
```

```
##                      No        Yes MeanDecreaseAccuracy MeanDecreaseGini
## CompPrice    8.35248271  4.7102793            8.6960977        17.549021
## Income      -0.06315732  1.4469209            0.8646496        13.678564
## Advertising  8.72373991 10.1855829           12.6630292        15.430023
## Population  -1.48767796 -2.2076200           -2.6537248        12.978464
## Price       28.86578016 27.1497159           35.7440043        35.146263
## ShelveLoc   32.40291215 29.6944689           38.3160824        24.841752
## Age          8.18109496  4.0036279            8.3864050        17.684315
## Education   -1.76822948  0.7569492           -0.7307049         7.913950
## Urban        1.25311690 -1.0094501            0.2163290         1.764151
## US           2.15480228  1.5747006            2.9423775         2.169262
```

Variable importance plot is also a useful tool and can be plotted using **varImpPlot()** function. By default, top 10 variables are selected and plotted based on Model Accuracy and Gini value. We can also get a plot with decreasing order of importance based on Model Accuracy and Gini value.

```r
varImpPlot(rf.carseats, sort=T,
           main="Variable Importance for rf.carseats", n.var=5)
```



The results indicate that across all of the trees considered in the random forest, the `price` and `ShelfLoc` are by far the two most important variables in terms of Model Accuracy and Gini index.

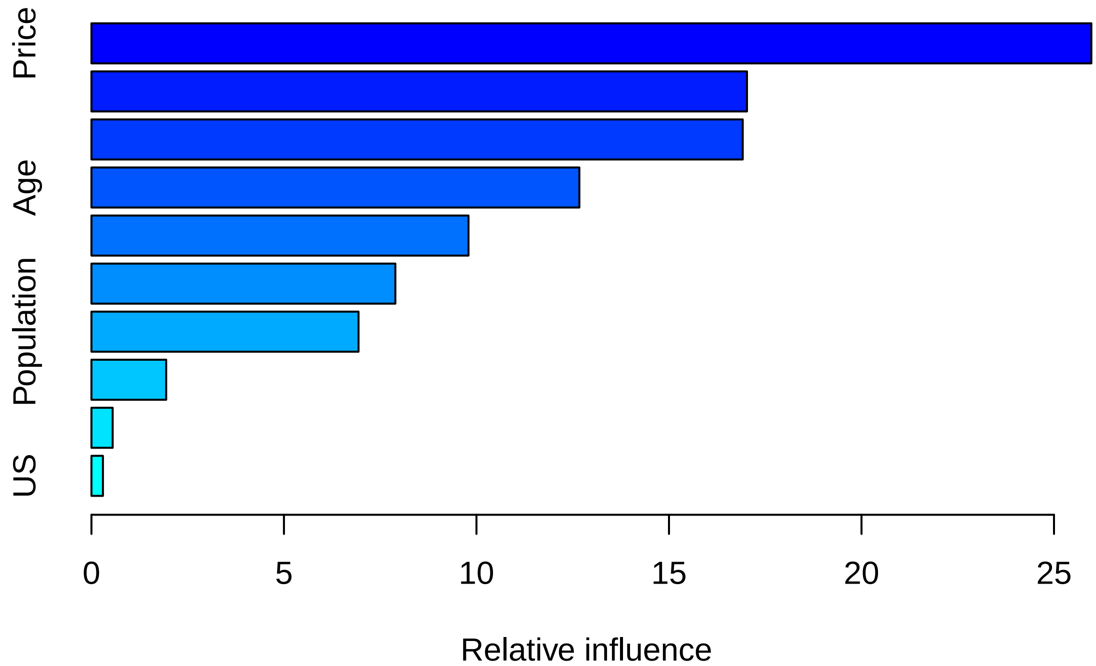
4. Boosting

Here we use the **`gbm`** package, and within it the **`gbm()`** function, to fit boosted classification trees to the Carseats data set. To use **`gbm()`**, we have to guarantee that the response variable is coded as $\{0, 1\}$ instead of two levels. We run **`gbm()`** with the option `distribution="bernoulli"` since this is a binary classification problem; if it were a regression problem, we would use `distribution="gaussian"`. The argument `n.trees=500` indicates that we want 500 bagged trees, and the option `interaction.depth=2` limits the depth of each tree (which is an equivalent parameter to `d` in lecture note and the textbook). The argument `shrinkage` is the learning rate (λ) or step-size reduction in every step of the boosting. Its default value is 0.001.

```
set.seed(1)
boost.carseats = gbm(ifelse(High=="Yes",1,0)~., data=train.carseats,
                     distribution="bernoulli", n.trees=500, interaction.depth=2)
```

The **`summary()`** function produces a relative influence plot and also outputs the relative influence statistics.

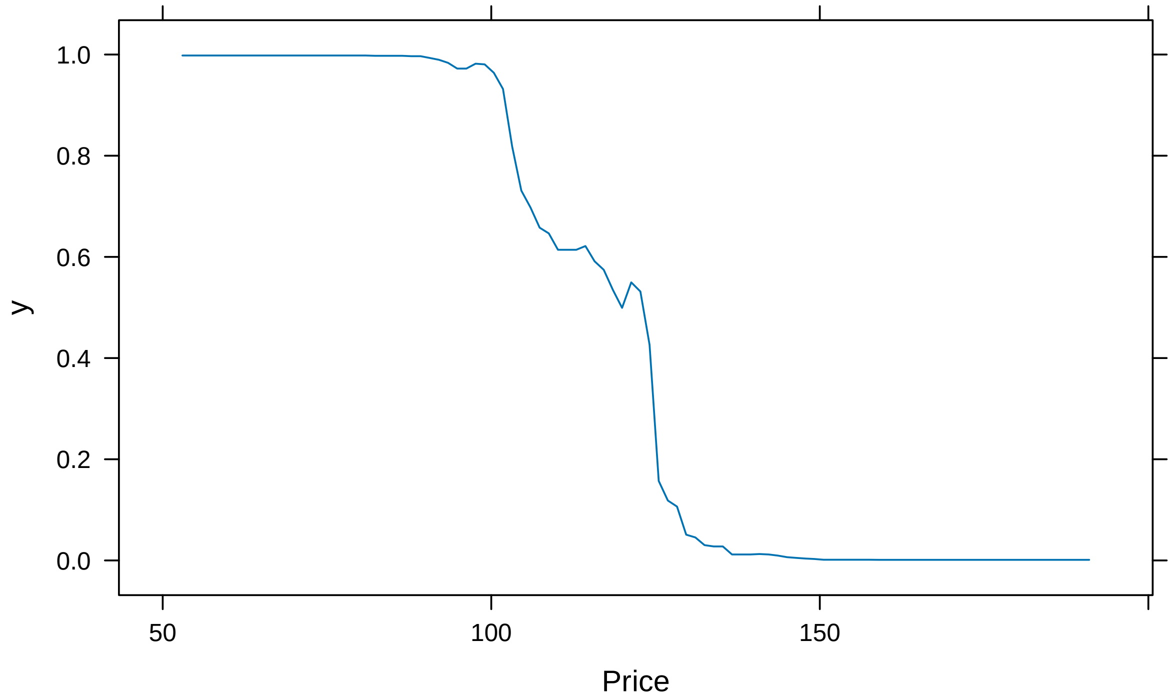
```
summary(boost.carseats)
```



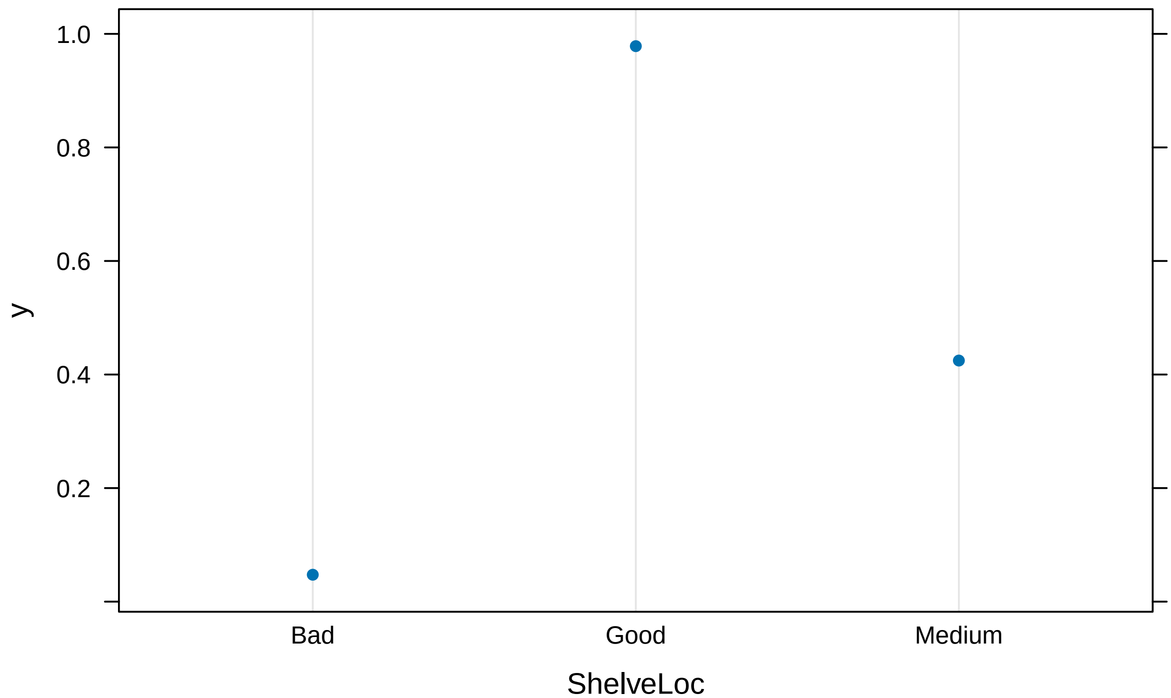

```
##                     var    rel.inf
## Price             Price 25.9701887
## CompPrice     CompPrice 17.0261700
## ShelveLoc     ShelveLoc 16.9160904
## Age                 Age 12.6725963
## Advertising Advertising  9.7932952
## Income           Income  7.8933670
## Population   Population  6.9370688
## Education     Education  1.9413887
## Urban             Urban  0.5512830
## US                   US  0.2985519
```

We see that `Price` is by far the most important variable. We can also produce partial dependence plots for these variables. These plots illustrate the marginal effect of the selected variables on the response after integrating out the other variables.

```
par(mfrow =c(1,2))
plot(boost.carseats ,i="Price", type = "response")
```




```r
plot(boost.carseats ,i="ShelveLoc", type = "response")
```



We now use the boosted model to predict `High` on the test set:

```
# setting type = "response" gives the predicted probability
yhat.boost = predict(boost.carseats, newdata = test.carseats,
                     n.trees=500, type = "response")
# then convert the probability to labels
yhat.boost = ifelse(yhat.boost > 0.5, 1, 0)
# compare with the 0,1 coding used in training gbm
test.boost.err = mean(yhat.boost != ifelse(test.carseats$High=="Yes",1,0))
test.boost.err
```

```
## [1] 0.19
```

Note that different from the **`predict`** function for the `randomForest` object, the **`predict`** function defined for the `gbm` object yields the predicted probability instead of the exact classes when `type = "reponse"`. Furthermore, the returned predicted probability is a vector that contains the probability for `Yes`. In order to get exact predicted classes, we convert the predicted probability to classes by comparing the probability with a threshold (0.5 in the code above).

The test error rate obtained is 0.19; slightly better than the test error rate for random forests and slightly worse to that for bagging.