Decision Tree Exercises

Simone Collier

Fitting Classification Trees

Start by loading the packages we need. If you need to install the packages first then run `install.packages("PACKAGENAME")` in your console before running the code chunk.

```
library(tree)
library(ISLR2)
library(randomForest)
library(gbm)
library(BART)
```

We will be making use of the `Carseats` data set in the `ISLR2` package. It contains informatin about the sales of carseats in 400 different stores.

```
attach(Carseats)
?Carseats
```

We want to create a tree to predict the sales of the carseats. Right now, the variable `Sales` is the number of carests in thouhsands that are sold at each location. Instead of predicting a number we want to predict whether the number of `Sales` is high (exceeds 8) or not. Let's make a new qualitative variable `High` that decribes whether the `Sales` are high and add it to the Carseats data frame.

```
High <- factor(ifelse(Sales <= 8, "No", "Yes"))
Carseats <- data.frame(Carseats, High)
```

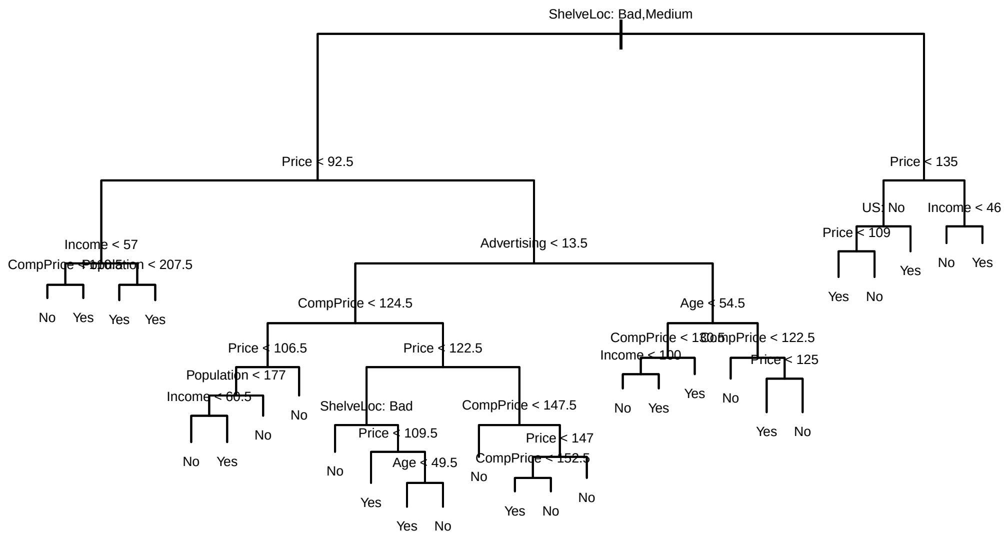
We can now fit a classification tree to the data in order to predict `High` using all the other variables in the dataset other than `Sales`. We can use the function `tree()` which is in the `tree` library.

```
tree.carseats <- tree(High ~. -Sales, Carseats)
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
##         8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##          16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
##          17) CompPrice > 110.5 5   6.730 Yes ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
##           18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
##           19) Population > 207.5 20   7.941 Yes ( 0.05000 0.95000 ) *
##        5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##         10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##           20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##             40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##               80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##                160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##                161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
##               81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##             41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##           21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##             42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
##               84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##               85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##                170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##                171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                  342) Age < 49.5 13  16.050 Yes ( 0.30769 0.69231 ) *
##                  343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
##             43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##               86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##               87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##                174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
##                  348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
##                  349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##                175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##         11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##           22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
##             44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
##               88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##               89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
##             45) CompPrice > 130.5 11   0.000 Yes ( 0.00000 1.00000 ) *
##           23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##             46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
##             47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##               94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
##               95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##      3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
##        6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
##         12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
##           24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
##           25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##         13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
##        7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##         14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##         15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *
```

Calling our tree gives the criterion for each branch. The asteriks indicate terminal nodes. We can use the `plot()` function to plot our tree and add the node labels using `text()`.

```r
plot(tree.carseats)
text(tree.carseats, pretty = 0, cex = 0.4)
```



Using the `summary()` function on our tree will gives us the borad overview of our tree as well as some errors associated with it.

```
summary(tree.carseats)
```

```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = Carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Income"      "CompPrice"   "Population"
## [6] "Advertising" "Age"         "US"
## Number of terminal nodes:  27
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

The training error rate of our tree is 9%. In order to properly assess the quality of our tree we need to estimate the test error. So, we start by splitting the data into a test set and a training set.

```
set.seed(2)
train <- sample(1:nrow(Carseats), nrow(Carseats)/2)
Carseats.test <- Carseats[-train, ]
High.test <- High[-train]
```

Now we can go ahead and fit the tree using the training obervations then use the `predict()` function to predict the responses for the test set.

```r
tree.carseats <- tree(High ~. - Sales, Carseats, subset = train)
tree.pred <- predict(tree.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred  No Yes
##       No  104  33
##       Yes  13  50
```

Recall that we comute the classification error rate as the sum of the missclassified observations divided by the total number of observations in the test set.

```r
(13 + 31)/200
```

```
## [1] 0.22
```

Now we will try out cost complexity pruning to see if we can get a tree with a better test error. The function `cv.tree()` performs corss-validation to find the best level of tree complexity. The argument `FUN = prune.miscall` indicates that we want to use the calssification error rate to guide the pruning process (the deault is deviance).

```r
set.seed(7)
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
cv.carseats
```

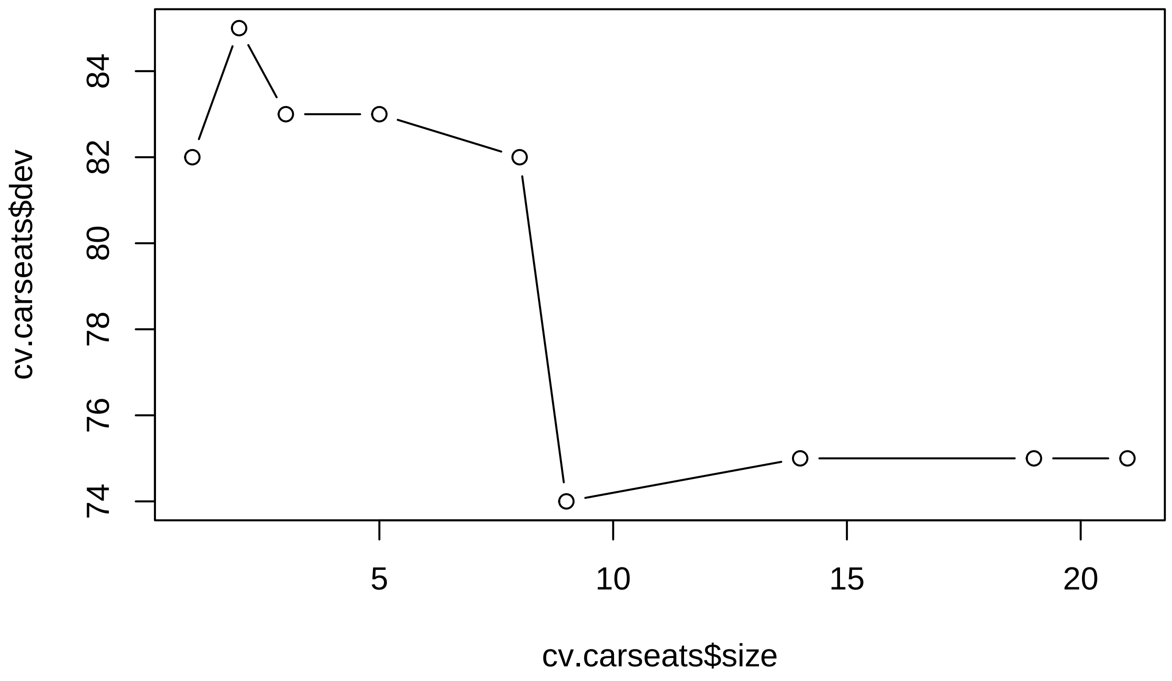
```
## $size
## [1] 21 19 14  9  8  5  3  2  1
##
## $dev
## [1] 75 75 75 74 82 83 83 85 82
##
## $k
## [1] -Inf  0.0  1.0  1.4  2.0  3.0  4.0  9.0 18.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

The output of `cv.tree()` contains the following information

- `size`: the number of terminal nodes for each tree that was considered.
- `dev`: the cross-validation errors.
- `k`: the cost-complexity tuning parameter.

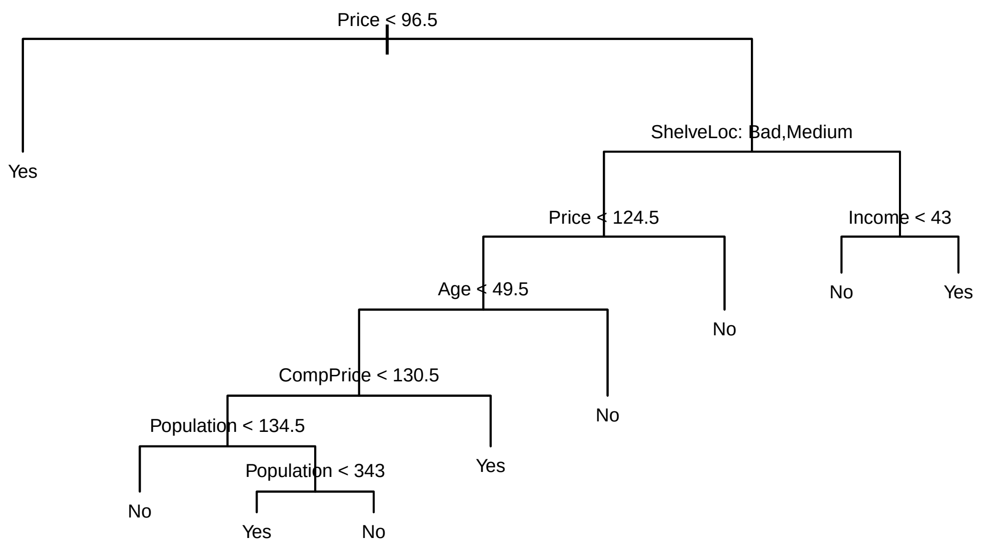
We can plot the error rate as a function of the `size`.

```
plot(cv.carseats$size, cv.carseats$dev, type = 'b')
```



We see that the error rate is at a minimum when `size` = 9 thus we use the function `prune.misclass()` to obtain this tree.

```
prune.carseats <- prune.misclass(tree.carseats, best = 9)
plot(prune.carseats)
text(prune.carseats, pretty = 0, cex = 0.6)
```



Compute the test error rate of this pruned tree. How does the test error rate and the interpretability of this tree compare to the inital tree?

Fitting Regression Trees

Recall the `Boston` dataset from the Linear Regression section. We will be fitting a regression tree to predict the median value of houses `medv` in Boston suburbs based on the information in the data set. First, we split the data into a training and test set.

```
attach(Boston)
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston)/2)
Boston.test <- Boston[-train, ]
medv.test <- Boston[-train, 'medv']
```

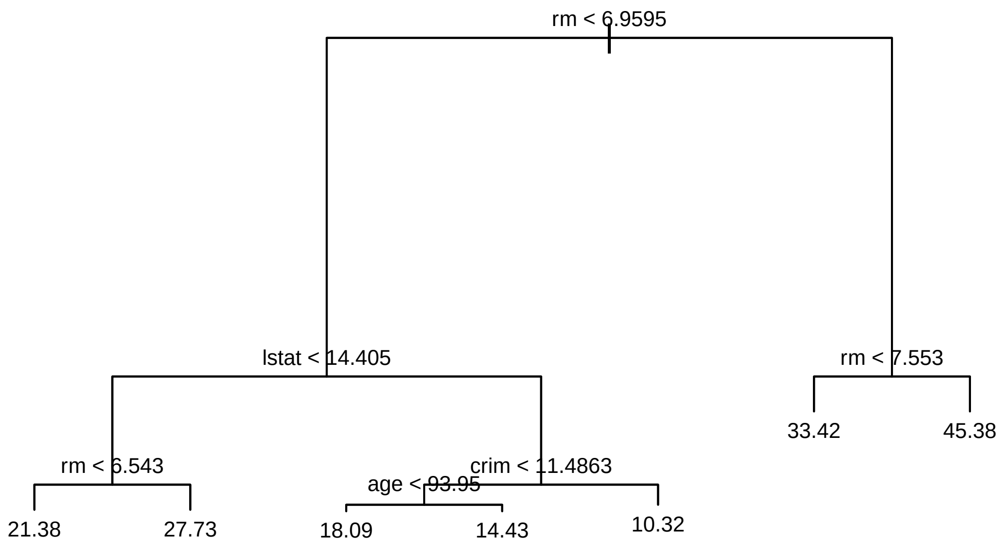
Now we can fit our tree with the training set.

```
tree.boston <- tree(medv ~., Boston, subset = train)
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
## Variables actually used in tree construction:
```

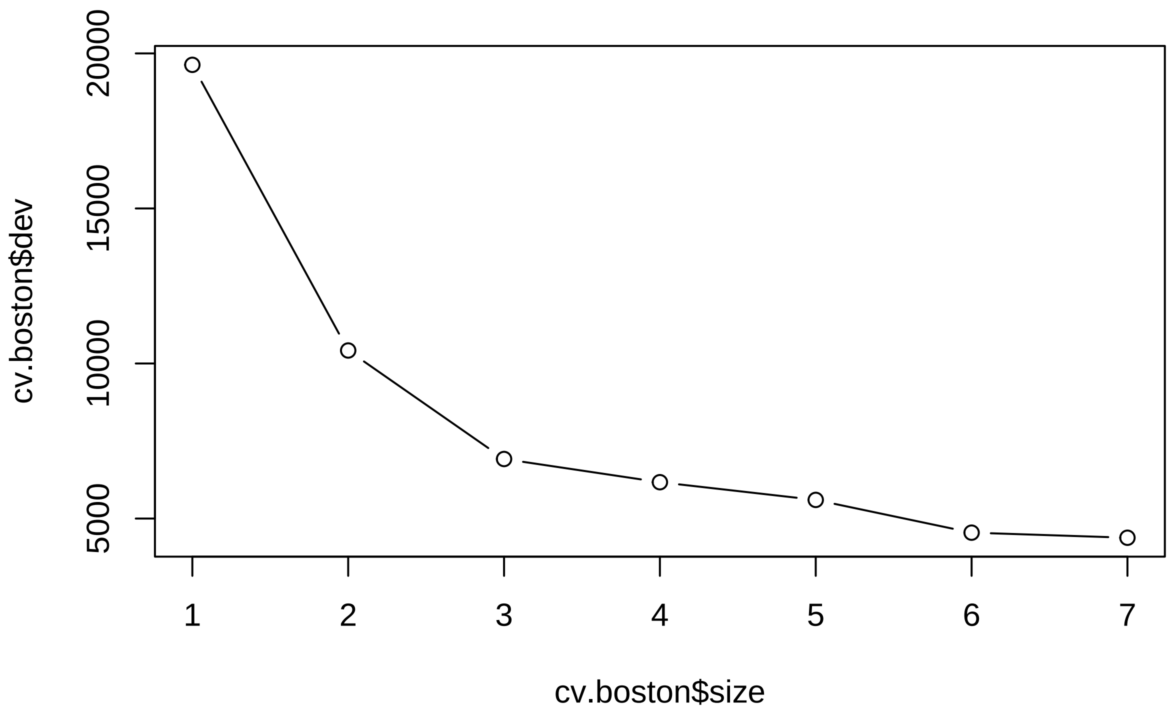
```
## [1] "rm"    "lstat" "crim"  "age"
## Number of terminal nodes:  7
## Residual mean deviance:  10.38 = 2555 / 246
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.1800  -1.7770  -0.1775   0.0000   1.9230  16.5800
```

```
plot(tree.boston)
text(tree.boston, pretty = 0, cex = 0.7)
```



Now we can use the `cv.tree()` function to see whether the tree would benefit from pruning.

```
cv.boston <- cv.tree(tree.boston)
plot(cv.boston$size, cv.boston$dev, type = 'b')
```



This indicates that this tree does not require pruning since the tree with 7 terminal nodes has the lowest error rate. If we did wich to prune it however, we would use the function `prune.tree()` the same as in the classification setting. We can compute the test error rate by estimating the test MSE.

```
pred <- predict(tree.boston, newdata = Boston.test)
mean((pred - medv.test)^2)
```

```
## [1] 35.28688
```

The test set MSE is 35.29.

Bagging and Random Forests

We will use bagging and random forests on the `Boston` data set. Since bagging is a special case of random forests with $m = p$, we can use the same function `randomForest()` from the `randomForest` library to perform both. We start with bagging. The argument `mtry = 12` indicates that all 12 of the predictors should be considered for each split of the tree.

```
set.seed(1)
bag.boston <- randomForest(medv ~., data = Boston, subset = train, mtry = 12, importance = TRUE)
bag.boston
```

```
##
```

```
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRUE,      subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 12
##
##           Mean of squared residuals: 11.40162
##                     % Var explained: 85.17
```

We can change the number of trees grown using the `ntree` argument.

Use this tree to predict the responses for the test set and estimate the test MSE. How does this compare to the MSE from the tree fitted without bagging?

Now let's try building a random forest of regression trees with `mtry = 6`.

```
set.seed(1)
rf.boston <- randomForest(medv ~., data = Boston, subset = train, mtry = 6, importance = TRUE)
medv.rf <- predict(rf.boston, newdata = Boston.test)
mean((medv.rf - medv.test)^2)
```

```
## [1] 20.06644
```

The test set MSE is 20.07 so the random forests provided a better tree than bagging in this case. The `importance()` function shows how important each of the variables are in the tree.

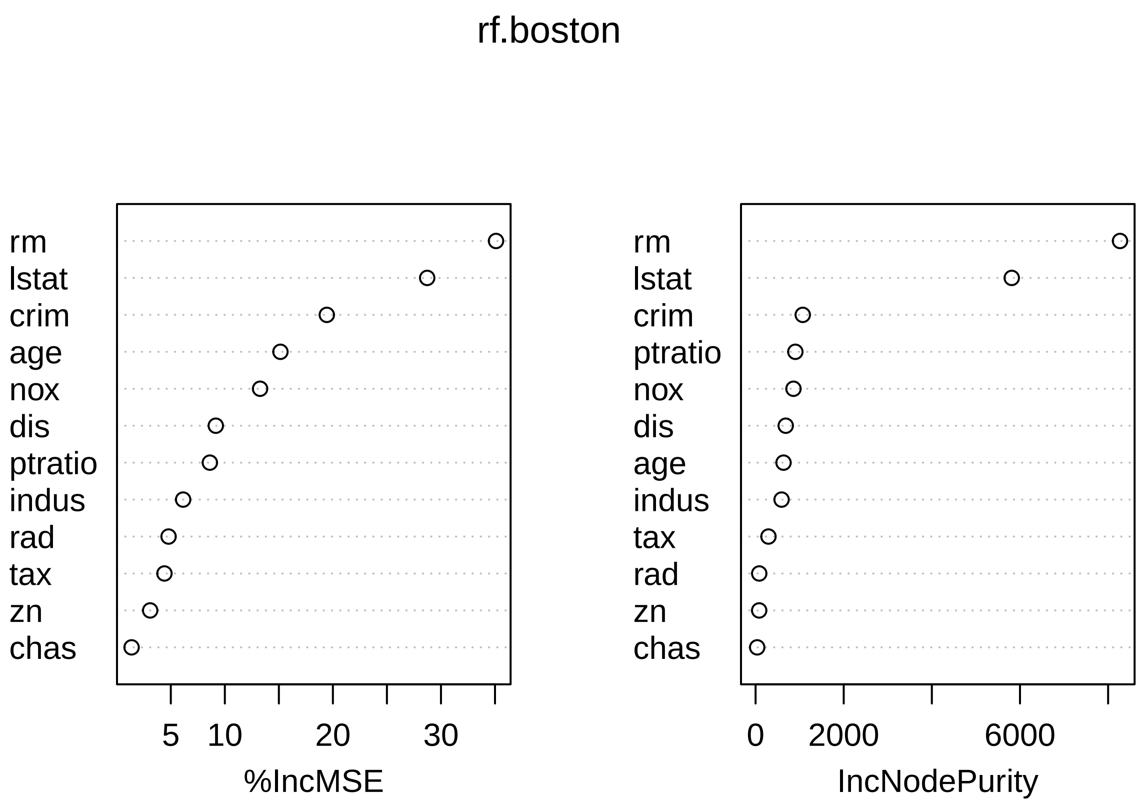
```
importance(rf.boston)
```

```
##           %IncMSE IncNodePurity
## crim    19.435587    1070.42307
## zn       3.091630      82.19257
## indus    6.140529     590.09536
## chas     1.370310      36.70356
## nox     13.263466     859.97091
## rm      35.094741    8270.33906
## age     15.144821     634.31220
## dis      9.163776     684.87953
## rad      4.793720      83.18719
## tax      4.410714     292.20949
## ptratio  8.612780     902.20190
## lstat   28.725343    5813.04833
```

- `%IncMSE` summarises the mean decrease of accuracy in predictions on the out of bag samples when the given variable is permuted.
- `IncNodePurity` measures the total decrease in node impurity that results from splits over the given variable (averaged over all trees).

We can plot these measures using the `varImpPlot()`

```
varImpPlot(rf.boston)
```

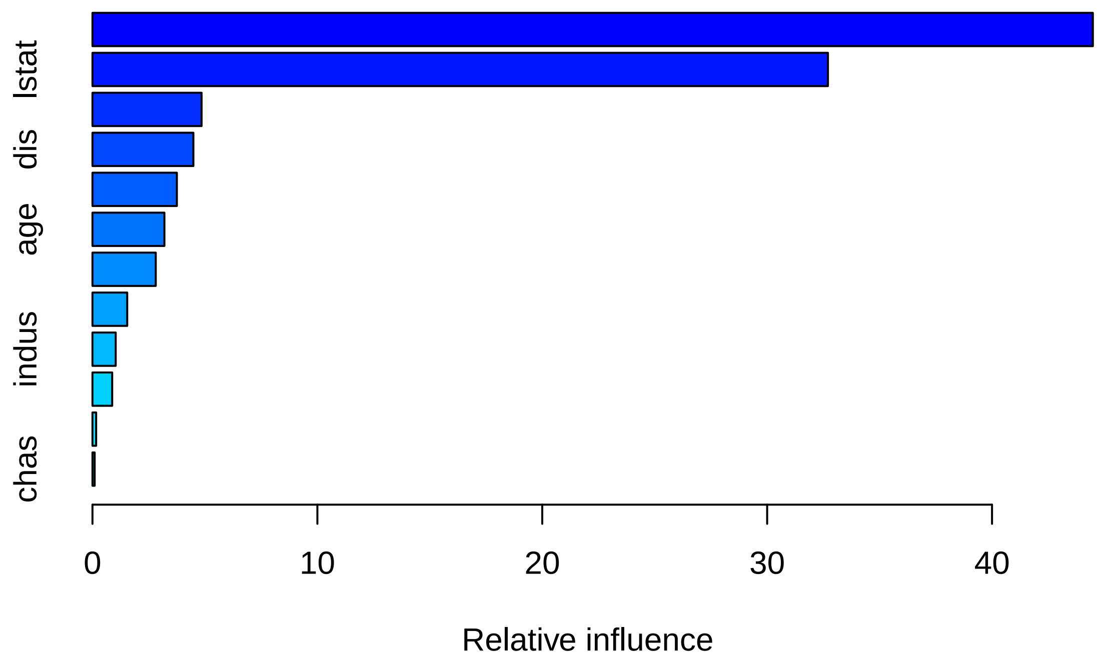


Which two variables are the most important when determining median house values in Boston suburbs?

Boosting

The `bm()` function from the `gbm` package will allow us to it boosted regression trees to the `Boston` data set. We set the argument `distribution` to `"gaussian"` since this is a regression problem (`"bernoulli"` for binary classification). The argument `n.trees` indicates how many trees and `interaction.depth` limits the depth of each tree.

```
set.seed(1)
boost.boston <- gbm(medv ~., data = Boston[train, ], distribution = "gaussian",
                    n.trees = 5000, interaction.depth = 4)
summary(boost.boston)
```



```
##              var     rel.inf
## rm            rm 44.48249588
## lstat      lstat 32.70281223
## crim        crim  4.85109954
## dis          dis  4.48693083
## nox          nox  3.75222394
## age          age  3.19769210
## ptratio ptratio  2.81354826
## tax          tax  1.54417603
## indus      indus  1.03384666
## rad          rad  0.87625748
## zn            zn  0.16220479
## chas        chas  0.09671228
```

In this case, the `summary()` function outputs the relative influence along with a plot of it.

We can now use the boosted model to predict `medv` on the test set.

```
medv.boost <- predict(boost.boston, newdata = Boston.test, n.trees = 5000)
mean((medv.boost - medv.test)^2)
```

```
## [1] 18.39057
```

The test MSE is 18.39 which is the best test MSE from all the methods so far.

Note that we can change the shrinkage parameter λ which is `shrinkage` in the `gbm()` function. The default value is 0.001.

Try fitting a new boosted model to the training set using a higher value for `shrinkage` and compute the test MSE. Which shrinkage parameter (between the two) yields the model with the best test error?

Bayesian Additive Regression Trees

We will use the `gbart()` function in the `BART` package to fit a Bayesian additive regression tree model to the `Boston` data. For this function, we need our data in the form of matrices.

```
x <- Boston[, 1:12]
y <- Boston[, 'medv']
xtrain <- x[train, ]
ytrain <- y[train]
xtest <- x[-train, ]
ytest <- y[-train]
```

We supply the test observations to the `gbart()` function directly so the fitting and predictions are made in one step. We can extract them and compute the test set MSE.

```
bart.tree <- gbart(xtrain, ytrain, x.test = xtest)
```

```
medv.bart <- bart.tree$yhat.test.mean
mean((medv.bart - medv.test)^2)
```

```
## [1] 15.51253
```

The test error rate for BART is the lowest of all the methods we tried.

These exercises were adapted from : James, Gareth, et al. An Introduction to Statistical Learning: with Applications in R, 2nd ed., Springer, 2021.