R for Classification: Classification Trees
Dolores Romero Morales

The aim of this workshop is to work on building classification trees and random forests using a dataset you are familiar with (newhousing.txt). This file can be found on the course directory. This file has thirteen explanatory variables. The last column is the class membership ('pricelevel'). There are two classes: 'below' and 'above'.

Recall that we will need to install the corresponding R packages.
Note1: To build Classification Trees we will use the R package 'tree'. Follow the two steps below to complete the installation and uploading of the package:

```
install.packages('tree',dependencies=TRUE)
library('tree')
```

Note2: To build Random Forests we will use the R package 'randomForest'. Follow the two steps below to complete the installation and uploading of the package:

```
install.packages('randomForest',dependencies=TRUE)
library('randomForest')
```

Task 1

Step 1. Read the newhousing.txt file into a data frame and get the dimension of the dataset.
Answer:

```
mynewhousing <- read.table(file.choose(),header=T, stringsAsFactors=TRUE)
dim(mynewhousing)
```

Step 2. Reshuffle the dataset and take a subsample of 400 observations. Call this the *minihousing*. Call the remaining dataset the *testhousing*.
Answer:

```
set.seed(1)
reshuffle <- mynewhousing[sample(nrow(mynewhousing)),]
minihousing <- reshuffle[1:400,]
testhousing <- reshuffle[401: nrow(reshuffle),]
```

Note1:
Recall that once you have reshuffled the data using 'sample', we are able to split the new dataset 'reshuffle' into two subsamples.
Note2:
Prior to the function that reshuffles the dataset, you may want to set the seed for the random generator using the function set.seed(). If you do so, then you are ensuring that you can reproduce in the future the same reshuffling.
Note3:
It is a good habit to use the 'print' function or the 'summary' function to get acquainted with the data, see whether we have created the two subsamples properly.
Note4:
It is not asked in the question, but if you want the class split in the minihousing you can use these couple of lines

```
priceleveltable <- table(minihousing$pricelevel)
aboveinminihousing <- priceleveltable[1]/nrow(minihousing)
belowinminihousing <- priceleveltable[2]/nrow(minihousing)
```

Task 2

Step 1. Using the *minihousing* subsample build a tree using the crossvalidation function.
Answer:

```
mytree <- tree(pricelevel~.,minihousing)
plot(mytree)
text(mytree)
summary(mytree)
set.seed(1)
mycrossval <- cv.tree(mytree,FUN=prune.tree)
```

Note1:
We have decided to plot the initial tree, see Figure 1. Note that the tree represented by this package is slightly different from the one we saw during the lecture. First, we have the condition right underneath the parent node. The left child satisfies the condition while the right child does not. The leaf nodes have been assigned a class (as said in class, it is usually the majority class, i.e., the class with probability above 0.5).

Note2:
We have been asked to get a summary of the initial tree, see below. In this output, we see among others, the explanatory variables used to build the tree. We have used 10 of them, while we had 13 in total. We can also see the number of leaf (terminal) nodes. We have in total 19 of them. We can also see the training misclassification error. In the training sample (minihousing), 38 observations out of 400 are misclassified.

```
Classification tree:
tree(formula = pricelevel ~ ., data = minihousing)
Variables actually used in tree construction:
 [1] "LSTAT"   "RM"     "CRIM"   "INDUS"  "B"      "TAX"    "PTRATIO"
 [8] "NOX"    "AGE"    "DIS"
Number of terminal nodes:  19
Residual mean deviance:  0.3819 = 145.5 / 381
Misclassification error rate: 0.095 = 38 / 400
```

Note3:
We have decided to view the tree in a written form. For instance, we can see there that:

node 2) is a leaf node that is pure because it contains individuals from one class. The class split is (1.00000 0.00000), i.e., 100% individuals are from the above class, 0% from the below class. Note that this node contains 123 observations.

node 24) is also a leaf node that is pure because it contains individuals from one class. The class split is (0.00000 1.00000), i.e., 0% individuals are from the above class, 100% from the below class. Note that this node contains 10 observations.

node 458) is another leaf node but it is not pure because the class split is (0.85714 0.14286), i.e., 85.7% individuals are from the above class, 14.3% from the below class. Note that this node contains 123 observations. Note that this node contains 7 observations.

In all these nodes we are also reporting the so-called deviance, which is proportional to the entropy that we saw in the lectures. Indeed,

deviance of node i = -2 * number of individuals in node i * entropy of node i

node), split, n, deviance, yval, (yprob)
* denotes terminal node

```
 1) root 400 549.700 above ( 0.55500 0.44500 )
   2) LSTAT < 7.765 123   0.000 above ( 1.00000 0.00000 ) *
   3) LSTAT > 7.765 277 361.200 below ( 0.35740 0.64260 )
     6) LSTAT < 11.67 84 111.600 above ( 0.61905 0.38095 )
      12) RM < 6.26 47  62.560 below ( 0.38298 0.61702 )
        24) CRIM < 0.05643 10   0.000 below ( 0.00000 1.00000 ) *
        25) CRIM > 0.05643 37  51.270 below ( 0.48649 0.51351 )
          50) INDUS < 10.3 21  25.130 below ( 0.28571 0.71429 )
           100) INDUS < 6.555 13  17.940 below ( 0.46154 0.53846 )
             200) RM < 5.9495 7   5.742 below ( 0.14286 0.85714 ) *
             201) RM > 5.9495 6   5.407 above ( 0.83333 0.16667 ) *
           101) INDUS > 6.555 8   0.000 below ( 0.00000 1.00000 ) *
          51) INDUS > 10.3 16  17.990 above ( 0.75000 0.25000 )
           102) B < 390.57 8  11.090 above ( 0.50000 0.50000 ) *
           103) B > 390.57 8   0.000 above ( 1.00000 0.00000 ) *
      13) RM > 6.26 37  20.820 above ( 0.91892 0.08108 )
        26) TAX < 357 30   0.000 above ( 1.00000 0.00000 ) *
        27) TAX > 357 7   9.561 above ( 0.57143 0.42857 ) *
     7) LSTAT > 11.67 193 214.300 below ( 0.24352 0.75648 )
      14) PTRATIO < 20.95 165 197.200 below ( 0.28485 0.71515 )
        28) CRIM < 38.0069 160 184.200 below ( 0.26250 0.73750 )
          56) NOX < 0.476 16   0.000 below ( 0.00000 1.00000 ) *
          57) NOX > 0.476 144 173.800 below ( 0.29167 0.70833 )
           114) TAX < 347.5 24  30.550 above ( 0.66667 0.33333 )
             228) PTRATIO < 18.1 8   0.000 above ( 1.00000 0.00000 ) *
             229) PTRATIO > 18.1 16  22.180 above ( 0.50000 0.50000 )
               458) AGE < 73.5 7   5.742 above ( 0.85714 0.14286 ) *
               459) AGE > 73.5 9   9.535 below ( 0.22222 0.77778 ) *
           115) TAX > 347.5 120 125.400 below ( 0.21667 0.78333 )
             230) LSTAT < 13.335 16  22.180 below ( 0.50000 0.50000 ) *
             231) LSTAT > 13.335 104  95.830 below ( 0.17308 0.82692 )
               462) DIS < 1.564 16   0.000 below ( 0.00000 1.00000 ) *
               463) DIS > 1.564 88  89.170 below ( 0.20455 0.79545 )
                 926) DIS < 1.8468 23  31.840 below ( 0.47826 0.52174 ) *
                 927) DIS > 1.8468 65  44.420 below ( 0.10769 0.89231 ) *
        29) CRIM > 38.0069 5   0.000 above ( 1.00000 0.00000 ) *
      15) PTRATIO > 20.95 28   0.000 below ( 0.00000 1.00000 ) *
```

Note4:
Recall that we stop with branching a node if it contains too few observations, otherwise we will overfit to the training data. This minimum number of observations to branch a node (minsize) is default equal to 10. If you want to control this parameter and change it to, for instance, 50, you can use the following lines

```
setup <- tree.control(nobs=nrow(minihousing), mincut = 25, minsize = 50,mindev=0.01)
mytree <- tree(pricelevel~.,minihousing, control=setup)
```

Note5:
Observe that before calling the crossvalidation function we have called the set.seed(1) command, which sets the seed used for random numbers to 1. This ensures that we can replicate the output of the crossvalidation function.

Note6:
The function cv.tree() reports results on a collection of pruned trees. These trees are found by cutting some branches from the tree in Figure 1, constructed by the function tree(), and thus reducing the number of leaf nodes. The purpose of cutting these nodes is to construct a tree that will generalize better, i.e., show better accuracy on unseen data. As we will see in the next step of this workshop, after we have run the cv.tree() function we still need to do a bit of more work. We need to train the tree again with the best parameters returned by the cv.tree() function.

Note7:
The function cv.tree() performs k-fold cross-validation (default is 10) to. This means that following two lines are doing exactly the same
mycrossval <- cv.tree(mytree,FUN=prune.tree)
mycrossval <- cv.tree(mytree,FUN=prune.tree,K=10)

This crossvalidation is performed to tune a parameter that weighs two objectives, namely, the prediction quality of the tree (this can be measured by the misclassification error) and its complexity (this can be measured by the number of leaf nodes).

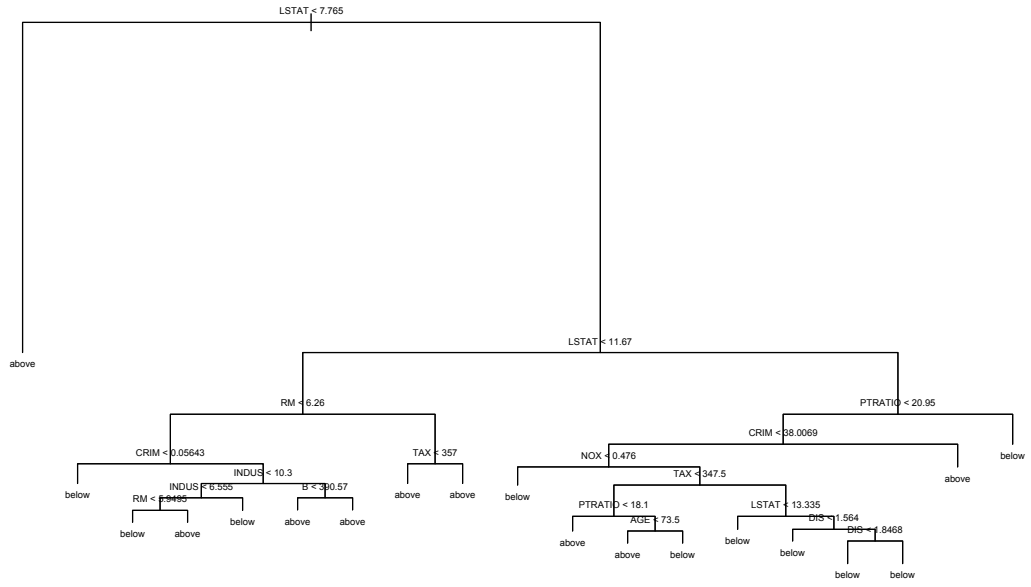

Figure 1 Tree plot obtained with the package tree

Step 2. Use this tree to test the model in *testhousing* and report the misclassification matrix as well as the classification accuracy.

Answer:
mybestsize <- mycrossval$size[which(mycrossval$dev==min(mycrossval$dev))]
myprunedtree <- prune.tree(mytree,best =mybestsize[1])

```
plot(myprunedtree)
summary(myprunedtree)
myprediction <- predict(myprunedtree, testhousing[,-14], type='class')
classificationtable <- table(myprediction,testhousing[,14])
acctesttree <- sum(diag(classificationtable))/sum(classificationtable)
```

Note1:
It is important to note that the cv.tree() function does not return a tree, but a collection of trees. As mentioned above, the cv.tree() function performs an analysis of different levels of pruning, a very aggressive pruning will remove a lot of branches, and will construct a very small tree. A very mild pruning will construct a very similar tree to the one in Figure 1. Therefore, we have to retrieve the best option for the pruning. In the first line of command, we retrieve the best value for the pruning of the tree. Then, in the second line, we prune the tree using this best value. The output of this function is then OUR FINAL tree that can be used for prediction.

Note2:
It is important to add the option " type='class' " to the function predict(). By doing this the predict() returns a class label. Please try out this function without the option and see that we obtain a very different output: the class split of the leaf node in which the observation will fall into.

Note3:
You probably got a warning

```
Warning message:
In best <= size :
  longer object length is not a multiple of shorter object length
```

This means that we have several prunings of the tree with the same performance and therefore, mybestsize does not contain one single value, but several. Therefore, it is better if you use mybestsize[1], instead of mybestsize.

Note4:
We have decided to plot the pruned tree, see Figure 2.

Note5:
We have decided to get the summary of the pruned tree. Because we have pruned the tree, the summary function tells you where the tree was pruned. See snip.tree().

```
Classification tree:
snip.tree(tree = mytree, nodes = c(13L, 12L, 7L))
Variables actually used in tree construction:
[1] "LSTAT" "RM"
Number of terminal nodes:  4
Residual mean deviance:  0.7516 = 297.7 / 396
Misclassification error rate: 0.17 = 68 / 400
```

Note6:
We have decided to view the tree in a written form.

```
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 1) root 400 549.70 above ( 0.55500 0.44500 )
```

```
2) LSTAT < 7.765 123   0.00 above ( 1.00000 0.00000 ) *
3) LSTAT > 7.765 277 361.20 below ( 0.35740 0.64260 )
  6) LSTAT < 11.67 84 111.60 above ( 0.61905 0.38095 )
   12) RM < 6.26 47  62.56 below ( 0.38298 0.61702 ) *
   13) RM > 6.26 37  20.82 above ( 0.91892 0.08108 ) *
  7) LSTAT > 11.67 193 214.30 below ( 0.24352 0.75648 ) *
```

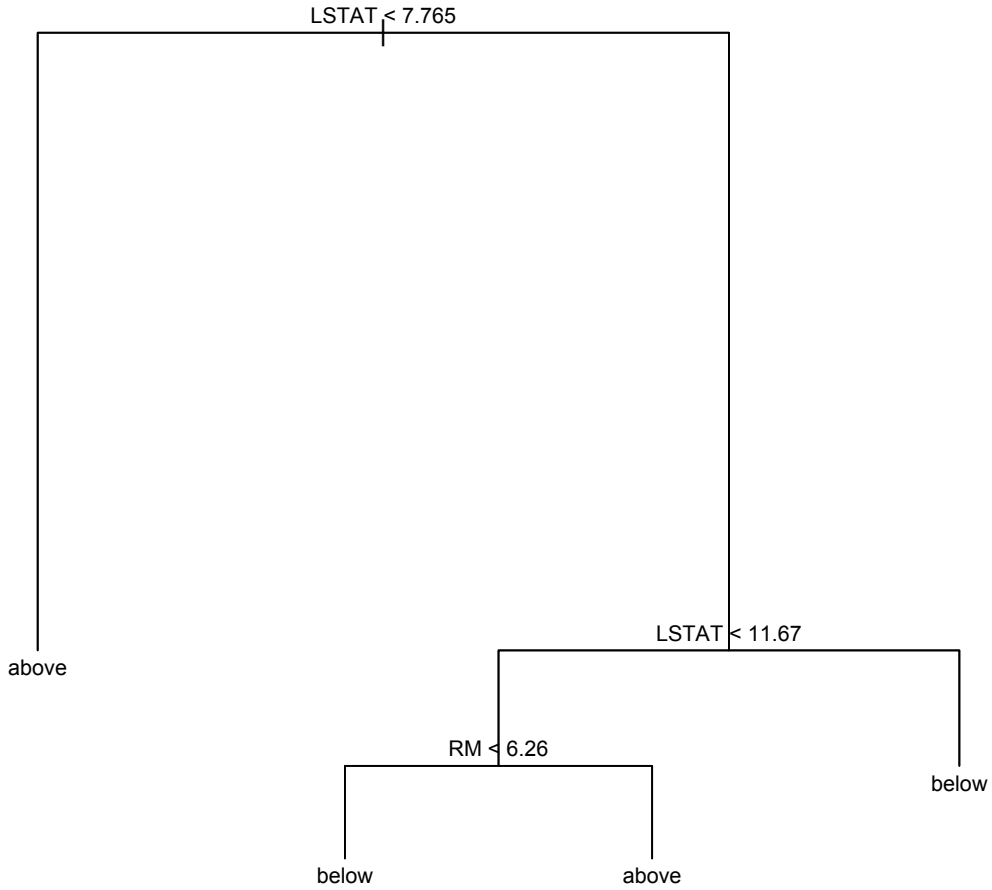

Figure 2 Tree plot obtained with the package tree after pruning

Task 3

Step 1. Using the *minihousing* subsample build a random forest and report the variable importance of each explanatory variable.

Answer:

```
myrf <- randomForest(pricelevel~.,minihousing,ntree=500,mtry=4,importance=TRUE)
importance(myrf)
varImpPlot(myrf)
```

Note1:
The default value of ntree is 500. This means that the random forest consists of 500 trees.

Note2:
The default value of mtry is squared root of the number of explanatory variables and rounded down. This is the parameter that controls how many explanatory variables we can use in each branch node. Once mtry is fixed, the set of explanatory variables available at a branch node is chosen randomly.

Note3:
Please have a look at what is behind myrf. You can see there that we have built 500 trees, and that we can use 4 variables at each branch node. You can also see the Out Of Bag (OOB) estimate of error rate. Recall that each tree is built on a bootstrapped sample. This sample is used as the training sample to build the tree, and the remaining sample can be seen as a testing sample. On that testing sample you can report the misclassification error. Then, you can average this error for all the 500 testing samples you have. This is what we call the OOB estimate of error rate.

```
Call:
 randomForest(formula = pricelevel ~ ., data = minihousing, ntree = 500,          mtry = 4,
importance = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 14%
Confusion matrix:
      above below class.error
above   186    36   0.1621622
below    20   158   0.1123596
```

Note4:
Look into the numbers provided for the variable importance. Please note that not many methods in Machine Learning are able to provide such a proxy. This together with their strong performance is one of the reasons why Random Forests are so popular. We have plotted this information in Figure 3.

```
            above     below MeanDecreaseAccuracy MeanDecreaseGini
CRIM     8.421940  5.584385           11.549191        14.620887
ZN       2.460336  5.382400            6.958940         1.400185
INDUS    2.347615 12.554050           13.188369         9.935382
CHAS     3.257596  7.117229            7.583251         1.950501
NOX     11.326259  7.691043           14.999696        16.573833
RM      20.394914 25.936464           32.603137        34.292945
AGE      4.795845 19.200040           19.381102        15.020252
DIS      6.212926  7.072173           11.216172        12.374913
RAD      2.577932  4.767621            6.170998         2.820529
TAX     12.832901  6.714173           15.398679         9.073997
PTRATIO 13.941956 16.230466           21.184452        17.531486
B        7.408615  2.727041            8.098385         9.333546
LSTAT   17.617011 43.584310           42.687504        51.027881
```

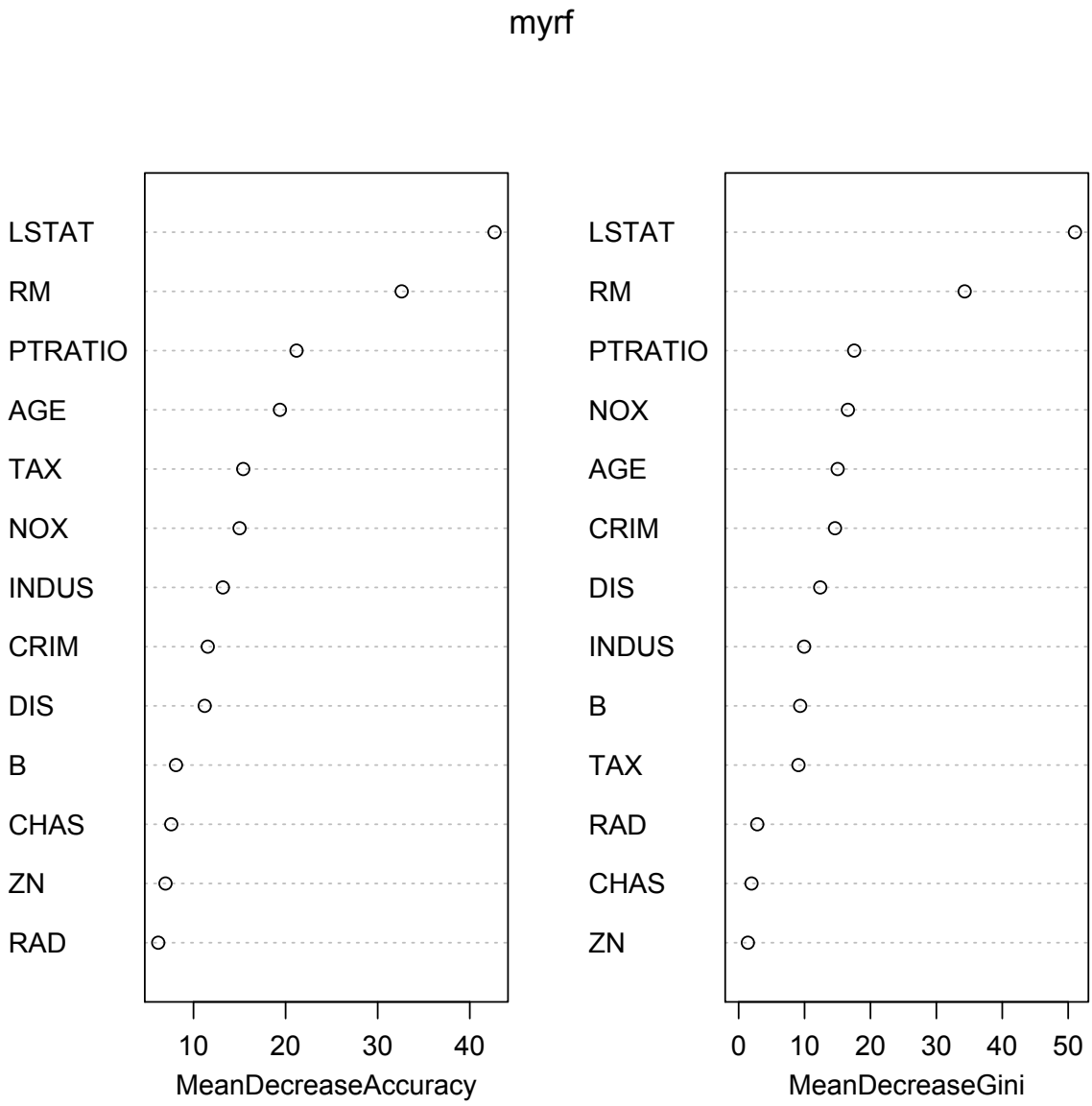

Figure 3 Variable Importance for NewHousing data

Step 2. Use this random forest to test the model in *testhousing* and report the classification matrix as well as the classification accuracy.

Answer:

```
myprediction <- predict(myrf, testhousing[,-14], type='class')
classificationtable <- table(myprediction,testhousing[,14])
acctestrandomforest <- sum(diag(classificationtable))/sum(classificationtable)
```