Classification Trees

Load and prepare data set

For this lab, we will use the *Carseats* data set from the *ISLR* package.

(install and) load the package with the data set

```
# load ISLR package
# install.packages('ISLR')
library(ISLR)
```

Carseats is a simulated data set containing data about sales of child car seats at 400 different stores. To inform about this data set, type *?Carseats*.

```
# get the Carseats dataset docs
?Carseats
```

We'll start by examining the structure of the data set.

```
# examine dataset structure
str(Carseats)

## 'data.frame':    400 obs. of  11 variables:
##  $ Sales      : num  9.5 11.22 10.06 7.4 4.15 ...
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Income     : num  73 48 35 100 64 113 105 81 110 113 ...
##  $ Advertising: num  11 16 10 4 3 13 0 15 0 0 ...
##  $ Population : num  276 260 269 466 340 501 45 425 108 131 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2
3 3 ...
##  $ Age        : num  42 65 59 55 38 78 71 67 76 76 ...
##  $ Education  : num  17 10 12 14 13 16 15 10 10 17 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
```

Based on the *Sales* variable, we'll add a new categorical (factor) variable to be used for classification.

```
# examine Sales variable distribution
summary(Carseats$Sales)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   0.000   5.390   7.490   7.496   9.320  16.270
```

Name the new variable *HighSales* and define it as a factor with two values: 'Yes' if *Sales* value is greater than the 3rd quartile, and 'No' otherwise.

```
# get the 3rd quartile of the Sales variable
sales.3Q <- quantile(Carseats$Sales, 0.75)

# create a new variable HighSales based on the value of the Sales variable
Carseats$HighSales <- ifelse(test = Carseats$Sales > sales.3Q, yes = 'Yes',
no = 'No')
head(Carseats[,c('Sales', 'HighSales')])

##   Sales HighSales
## 1  9.50       Yes
## 2 11.22       Yes
## 3 10.06       Yes
## 4  7.40        No
## 5  4.15        No
## 6 10.81       Yes

# check the type of the HighSales variable
class(Carseats$HighSales)

## [1] "character"
```

We have created a character vector. Now, we need to transform it into a factor variable.

```
# convert HighSales into a factor variable
Carseats$HighSales <- as.factor(Carseats$HighSales)
head(Carseats$HighSales)

## [1] Yes Yes Yes No  No  Yes
## Levels: No Yes
```

Let’s check the distribution of the two values.

```
# get the distribution of the HighSales variable
table(Carseats$HighSales)

##
##  No Yes
## 301  99

# examine the distribution through proportions
prop.table(table(Carseats$HighSales))

##
##     No    Yes
## 0.7525 0.2475
```

So, in 75.25% of shops, the company did not achieve high sales.

The objective is to develop a model that would be able to predict if the company will have a large sale in a certain shop. More precisely, the company is interested in spotting shops where high sales are not expected, so that it can take some interventions to improve sales. This means that the class we are particularly interested in - the so-called *positive class* is No.

Create train and test datasets

Remove the *Sales* variable as we do not need it anymore - since it was used for creating the outcome variable, it cannot be used as a predictor.

```
# remove Sales variable
Carseats$Sales <- NULL
```

We should randomly select observations for training and testing. We should also assure that the distribution of the output variable (*HighSales*) is the same in both datasets (train and test); this is referred to as *stratified partitioning*. To do that easily, we'll use appropriate functions from the *caret* package.

```
# load caret package
library(caret)
```

We'll use 80% of all the observations for training and the rest for testing.

```
# create train and test datasets
set.seed(10)
train.indices <- createDataPartition(Carseats$HighSales, # the class variable
                                     p = .80,            # the proportion of
observations in the training set
                                     list = FALSE)       # do not return the
result as a list
train.data <- Carseats[train.indices,]
test.data <- Carseats[-train.indices,]
```

We can check that the distribution of *HighSales* is really (roughly) the same in the two datasets.

```
# print distributions of the outcome variable on the train and test datasets
prop.table(table(train.data$HighSales))

##
##        No       Yes
## 0.7507788 0.2492212

prop.table(table(test.data$HighSales))

##
##        No       Yes
## 0.7594937 0.2405063
```

Create a prediction model using Classification Trees

We will use the **rpart** R package to build a classification tree.

Note: this is just one of the available R packages for working with classification trees.

```r
# load rpart library
library(rpart)
```

Build a tree using the *rpart* function and all the variables.

```r
?rpart

# build the model
tree1 <- rpart(HighSales ~ ., data = train.data, method = "class")
```

Note the parameter *method*; it is set to the “class” value as we are building a classification tree; if we want to build a regression tree (to perform a regression task), we would set this parameter to ‘anova’.

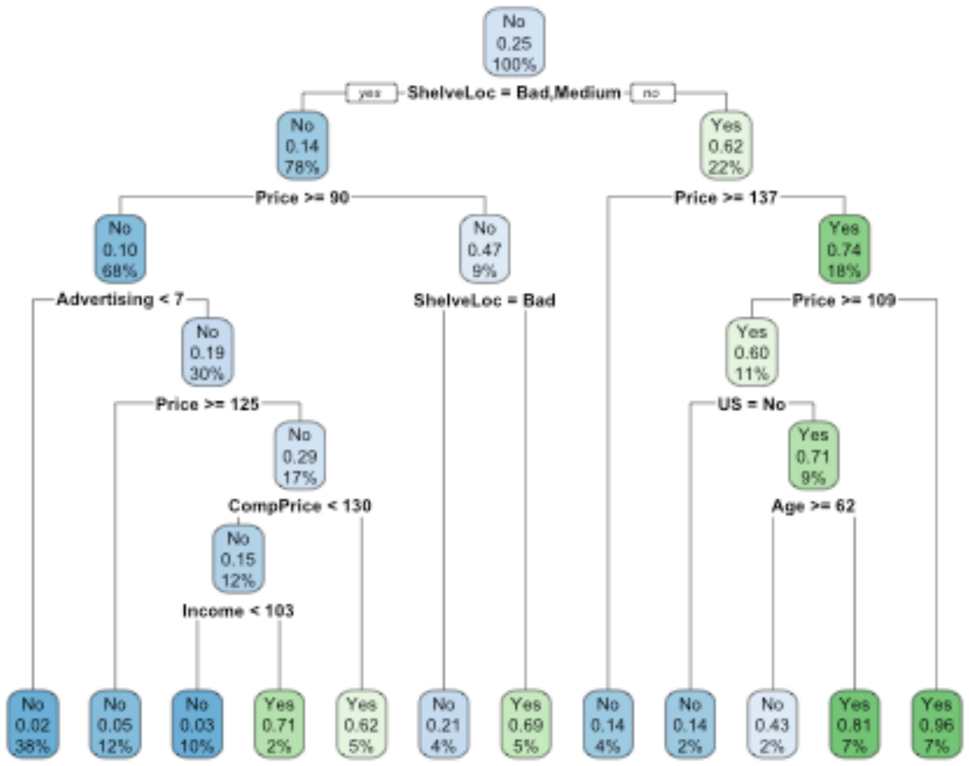
```r
# print the model
print(tree1)

## n= 321
## 
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
## 
##  1) root 321 80 No (0.75077882 0.24922118)  
##    2) ShelveLoc=Bad,Medium 249 35 No (0.85943775 0.14056225)  
##      4) Price>=89.5 219 21 No (0.90410959 0.09589041)  
##        8) Advertising< 6.5 123  3 No (0.97560976 0.02439024) *
##        9) Advertising>=6.5 96 18 No (0.81250000 0.18750000)  
##         18) Price>=124.5 40  2 No (0.95000000 0.05000000) *
##         19) Price< 124.5 56 16 No (0.71428571 0.28571429)  
##           38) CompPrice< 129.5 40  6 No (0.85000000 0.15000000)  
##             76) Income< 102.5 33  1 No (0.96969697 0.03030303) *
##             77) Income>=102.5 7  2 Yes (0.28571429 0.71428571) *
##           39) CompPrice>=129.5 16  6 Yes (0.37500000 0.62500000) *
##      5) Price< 89.5 30 14 No (0.53333333 0.46666667)  
##       10) ShelveLoc=Bad 14  3 No (0.78571429 0.21428571) *
##       11) ShelveLoc=Medium 16  5 Yes (0.31250000 0.68750000) *
##    3) ShelveLoc=Good 72 27 Yes (0.37500000 0.62500000)  
##      6) Price>=136.5 14  2 No (0.85714286 0.14285714) *
##      7) Price< 136.5 58 15 Yes (0.25862069 0.74137931)  
##       14) Price>=108.5 35 14 Yes (0.40000000 0.60000000)  
##         28) US=No 7  1 No (0.85714286 0.14285714) *
##         29) US=Yes 28  8 Yes (0.28571429 0.71428571)  
##           58) Age>=61.5 7  3 No (0.57142857 0.42857143) *
##           59) Age< 61.5 21  4 Yes (0.19047619 0.80952381) *
##       15) Price< 108.5 23  1 Yes (0.04347826 0.95652174) *
```

Let’s plot the tree, to understand it better. To that end, we will use the **rpart.plot** package.

```r
# load rpart.plot library
# install.packages("rpart.plot")
library(rpart.plot)
```

Plot the tree.

```r
# plot the tree
rpart.plot(tree1)
```



TASK: Check the type and extra parameters of the rpart.plot() function to see how they affect the visualization of the tree model.

Observing the tree, we can see that seven (out of eleven) variables were used to build the model including ShelveLoc, Price, Advertising, US, etc.

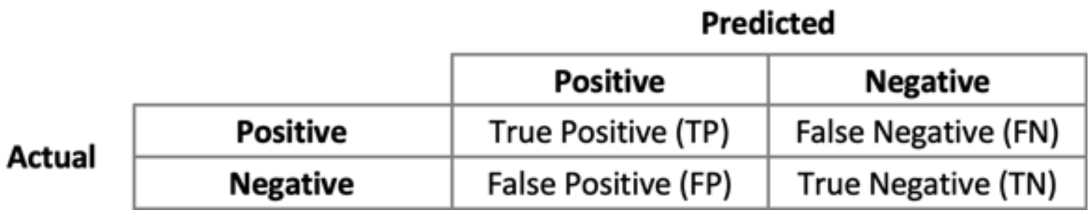
Let's evaluate our model on the test set.

```r
# make the predictions with tree1 over the test dataset
tree1.pred <- predict(object = tree1, newdata = test.data, type = "class")
```

Examine what the predictions look like.

```r
# print several predictions
head(tree1.pred)

##   2   5   7  10  13  20
## Yes  No  No  No  No  No
## Levels: No Yes
```

To start evaluating the predictive quality of our model, we will first create the confusion matrix. The **confusion matrix** is used for visualizing and calculating the performance of a classification model. Each row of the matrix represents the instances in an actual class, while each column represents the instances in a predicted class (or vice versa).



		Predicted	
		Positive	**Negative**
Actual	**Positive**	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Confusion Matrix

In our example, this is the confusion matrix (recall that we set 'No' as the positive class):

```r
# create the confusion matrix
tree1.cm <- table(true=test.data$HighSales, predicted=tree1.pred)
tree1.cm
```

```
##       predicted
## true   No Yes
##   No   57   3
##   Yes   3  16
```

There are several measures used for used for evaluating the performance of a classification model.

Precision tells us how precise our model is, that is, what proportion of observations that are predicted as positive (= belonging to the positive class) are actually positive. The formula is:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Precision is a good measure to use in a model where the 'cost' of False Positive is high. For instance, in email spam detection, a False Positive means that an email that is non-spam (actual negative) has been identified as spam (predicted positive). The email user might miss important emails if the precision is not high for the spam detection model.

Recall (also known as *sensitivity*) tells us what proportion of observations that are actually positive (= belong to the positive class) were predicted as positive. The formula is:

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Recall is important in a model when there is a high 'cost' associated with False Negatives. For instance, in a sick patient detection problem, if a sick patient (actual positive) is classified as not sick (predicted negative). The cost associated with False Negative will be extremely high if the sickness is contagious.

To illustrate these measures, suppose we have a classifier for recognizing dogs in photographs. And in a picture containing 12 dogs and some cats, the classifier identifies 8 dogs. Of the 8 identified as dogs, 5 actually are dogs (True Positives), while the rest are cats (False Positives). The program's precision is 5/8 while its recall is 5/12.

Two other important evaluation measures are *Accuracy* and *F1-measure*.

Accuracy is defined as the percentage of correct predictions. Informally, accuracy is the fraction of predictions our model got right. The formula is:

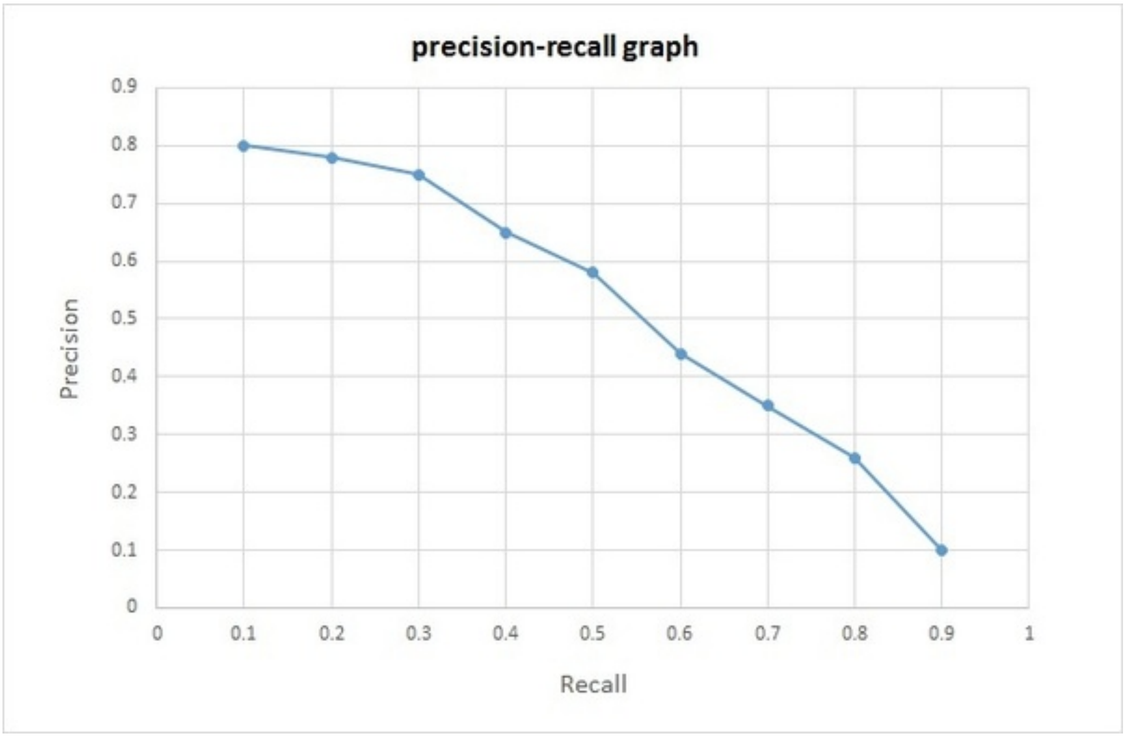
$$Accuracy = \frac{True\ Positive + True\ Negative}{N}$$

, where N is the total number of predictions.

F1-measure conveys the balance between the Precision and the Recall. The formula is:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

The need for balancing Precision and Recall stems from the fact that Precision and Recall are in a kind of 'antagonistic' relation: whenever we try to improve precision, we negatively affect recall and vice versa (see figure below). Hence, there was a need for a measure (F1-measure) that would give us a way to evaluate how well balanced Precision and Recall are.



Precision-Recall curve

Since we'll need to compute evaluation metrics couple of times, it's handy to have a function for that. The f. receives a confusion matrix, and returns a named vector with the values for Accuracy, Precision, Recall, and F1-measure.

```r
# function for computing evaluation measures
compute.eval.metrics <- function(cmatrix) {
  TP <- cmatrix[1,1] # true positive
  TN <- cmatrix[2,2] # true negative
  FP <- cmatrix[2,1] # false positive
  FN <- cmatrix[1,2] # false negative
  acc <- sum(diag(cmatrix)) / sum(cmatrix)
  precision <- TP / (TP + FP)
  recall <- TP / (TP + FN)
  F1 <- 2*precision*recall / (precision + recall)
  c(accuracy = acc, precision = precision, recall = recall, F1 = F1)
}
```

Now, we'll use the function to compute evaluation metrics for our tree model.

```r
# compute the evaluation metrics
tree1.eval <- compute.eval.metrics(tree1.cm)
tree1.eval

##  accuracy precision    recall        F1
## 0.9240506 0.9500000 0.9500000 0.9500000
```

Not bad…

The *rpart* function uses a number of parameters to control the growth of a tree. In the above call of the *rpart* function, we relied on the default values of those parameters. To inspect the parameters and their defaults, type:

```r
# get the docs for the rpart.control function
?rpart.control
```

Let's now change some of these parameters to try to create a better model. Two parameters that are often considered important are:

- **cp** - the so-called *complexity parameter*. It regulates the splitting of nodes and growing of a tree by preventing splits that are deemed not important enough. In particular, those would be the splits that would not improve the fitness of the model by at least the *cp* value,
- **minsplit** - minimum number of instances in a node for a split to be attempted at that node.

We will decrease the values of both parameters to grow a larger tree.

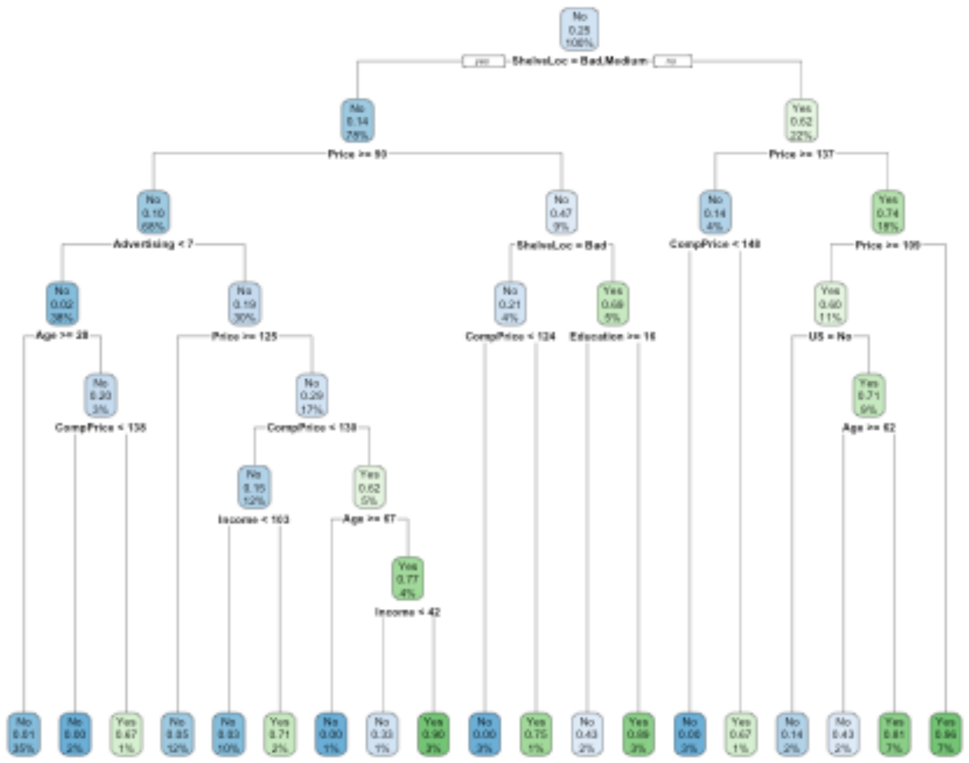
```r
# build the second model with minsplit = 10 and cp = 0.001
tree2 <- rpart(HighSales ~ ., data = train.data, method = "class",
                     control = rpart.control(minsplit = 10, cp = 0.001))
```

```
# print the model
print(tree2)
```

```
## n= 321
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##   1) root 321 80 No (0.750778816 0.249221184)
##     2) ShelveLoc=Bad,Medium 249 35 No (0.859437751 0.140562249)
##       4) Price>=89.5 219 21 No (0.904109589 0.095890411)
##         8) Advertising< 6.5 123  3 No (0.975609756 0.024390244)
##          16) Age>=27.5 113  1 No (0.991150442 0.008849558) *
##          17) Age< 27.5 10  2 No (0.800000000 0.200000000)
##            34) CompPrice< 137.5 7  0 No (1.000000000 0.000000000) *
##            35) CompPrice>=137.5 3  1 Yes (0.333333333 0.666666667) *
##         9) Advertising>=6.5 96 18 No (0.812500000 0.187500000)
##          18) Price>=124.5 40  2 No (0.950000000 0.050000000) *
##          19) Price< 124.5 56 16 No (0.714285714 0.285714286)
##            38) CompPrice< 129.5 40  6 No (0.850000000 0.150000000)
##              76) Income< 102.5 33  1 No (0.969696970 0.030303030) *
##              77) Income>=102.5 7  2 Yes (0.285714286 0.714285714) *
##            39) CompPrice>=129.5 16  6 Yes (0.375000000 0.625000000)
##              78) Age>=66.5 3  0 No (1.000000000 0.000000000) *
##              79) Age< 66.5 13  3 Yes (0.230769231 0.769230769)
##               158) Income< 42 3  1 No (0.666666667 0.333333333) *
##               159) Income>=42 10  1 Yes (0.100000000 0.900000000) *
##       5) Price< 89.5 30 14 No (0.533333333 0.466666667)
##        10) ShelveLoc=Bad 14  3 No (0.785714286 0.214285714)
##          20) CompPrice< 123.5 10  0 No (1.000000000 0.000000000) *
##          21) CompPrice>=123.5 4  1 Yes (0.250000000 0.750000000) *
##        11) ShelveLoc=Medium 16  5 Yes (0.312500000 0.687500000)
##          22) Education>=15.5 7  3 No (0.571428571 0.428571429) *
##          23) Education< 15.5 9  1 Yes (0.111111111 0.888888889) *
##     3) ShelveLoc=Good 72 27 Yes (0.375000000 0.625000000)
##       6) Price>=136.5 14  2 No (0.857142857 0.142857143)
##        12) CompPrice< 148 11  0 No (1.000000000 0.000000000) *
##        13) CompPrice>=148 3  1 Yes (0.333333333 0.666666667) *
##       7) Price< 136.5 58 15 Yes (0.258620690 0.741379310)
##        14) Price>=108.5 35 14 Yes (0.400000000 0.600000000)
##          28) US=No 7  1 No (0.857142857 0.142857143) *
##          29) US=Yes 28  8 Yes (0.285714286 0.714285714)
##            58) Age>=61.5 7  3 No (0.571428571 0.428571429) *
##            59) Age< 61.5 21  4 Yes (0.190476190 0.809523810) *
##        15) Price< 108.5 23  1 Yes (0.043478261 0.956521739) *
```

Obviously, we got a significantly larger tree. Let’s plot this tree.

```
# plot the tree2
rpart.plot(tree2)
```



Is this larger tree better than the initial one (tree1)? To check that, we need to evaluate the 2nd tree on the test set.

```
# make the predictions with tree2 over the test dataset
tree2.pred <- predict(tree2, newdata = test.data, type = "class")
```

Again, we'll create a confusion matrix.

```
# create the confusion matrix for tree2 predictions
tree2.cm <- table(true=test.data$HighSales, predicted=tree2.pred)
tree2.cm

##      predicted
## true  No Yes
##   No  57   3
##   Yes  5  14
```

Next, we'll compute the evaluation metrics.

```
# compute the evaluation metrics
tree2.eval <- compute.eval.metrics(tree2.cm)
tree2.eval

##  accuracy precision    recall        F1
## 0.8987342 0.9193548 0.9500000 0.9344262
```

Let's compare this model to the first one:

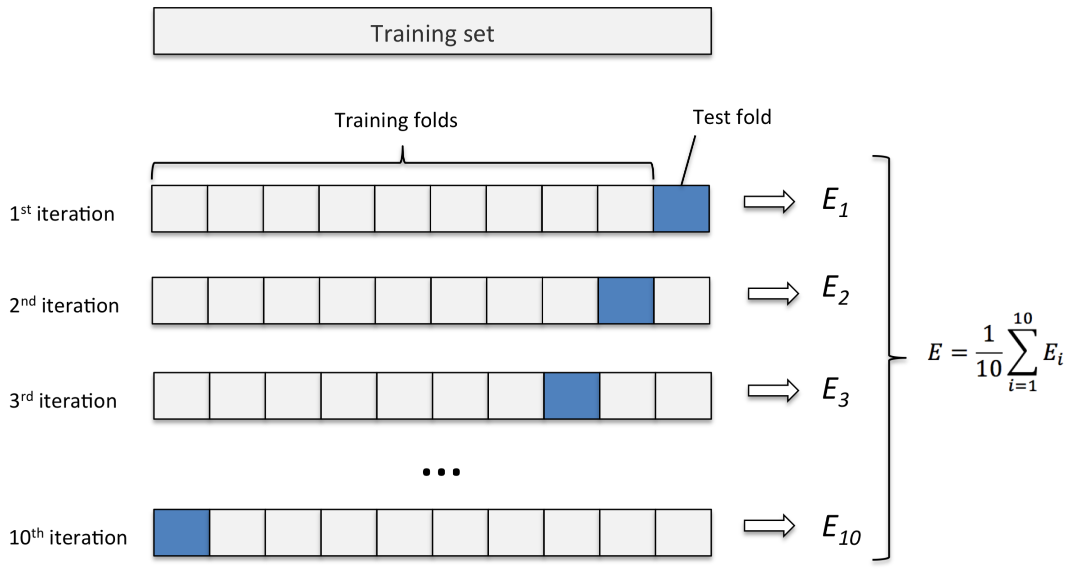
```r
# compare the evaluation metrics for tree1 and tree2
data.frame(rbind(tree1.eval, tree2.eval),
           row.names = c("tree_1", "tree_2"))
```

```
##         accuracy precision recall        F1
## tree_1 0.9240506 0.9500000   0.95 0.9500000
## tree_2 0.8987342 0.9193548   0.95 0.9344262
```

The metrics show worse performance on the new model compared to the initial one - the new model is obviously overly complex and overfitted to the training data. So, our guess for the parameter values was not good and we have to choose wiser.

Instead of relying on guessing, we should adopt a systematic way of examining the parameters values, looking for the optimal ones. An often applied approach is to perform cross-validation with a range of different values of parameters of interest.

Cross-validation is a model validation technique for assessing how well the model will generalize on an independent data set. It involves partitioning a dataset into *k* complementary equal-size subsets, using *k-1* subsets for model building and one subset for validation, and repeating this procedure *k* times, so that each time different subset is used for validation (and the rest of the subsets for training). Typically, *k* is set to 10, in which case we talk about *10-fold cross-validation* (see figure below).



10-fold cross-validation

We will apply that approach here, tuning the value of the *cp* parameter since it is considered the most important parameter when growing trees with the *rpart* function.

For finding the optimal *cp* value through cross-validation, we will use some handy functions from the **caret** package (it has already been loaded). Since these functions internally call cross-validation functions from the **e1071** package, we need to (install and) load that package.

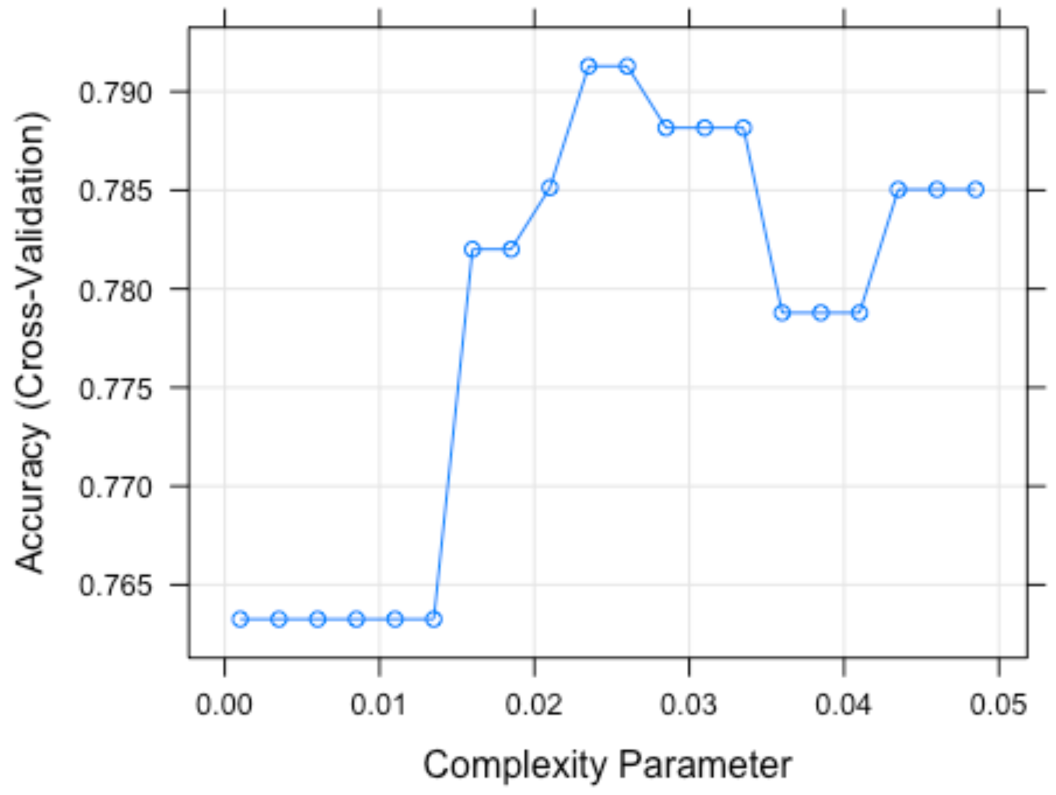
```
# load e1071 library
# install.packages('e1071')
library(e1071)

# define cross-validation (cv) parameters; we'll perform 10-fold cross-
validation
numFolds = trainControl( method = "cv", number = 10 )

# define the range for the cp values to examine in the cross-validation
cpGrid = expand.grid( .cp = seq(0.001, to = 0.05, by = 0.0025))
```

Perform parameter search through cross-validation.

```
# since cross-validation is a probabilistic process, we need to set the seed
so that the results can be replicated
set.seed(10)

# run the cross-validation
dt.cv <- train(x = train.data[,-11],
               y = train.data$HighSales,
               method = "rpart",
               control = rpart.control(minsplit = 10),
               trControl = numFolds,
               tuneGrid = cpGrid)
dt.cv

## CART
##
## 321 samples
##  10 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 289, 289, 288, 289, 289, 289, ...
## Resampling results across tuning parameters:
##
##   cp      Accuracy   Kappa
##   0.0010  0.7632576  0.3766010
##   0.0035  0.7632576  0.3766010
##   0.0060  0.7632576  0.3766010
##   0.0085  0.7632576  0.3732677
##   0.0110  0.7632576  0.3732677
##   0.0135  0.7632576  0.3732677
##   0.0160  0.7820076  0.4075508
##   0.0185  0.7820076  0.4075508
##   0.0210  0.7851326  0.4076702
##   0.0235  0.7912879  0.4122651
##   0.0260  0.7912879  0.4122651
##   0.0285  0.7881629  0.3914888
##   0.0310  0.7881629  0.3914888
##   0.0335  0.7881629  0.3914888
##   0.0360  0.7787879  0.3797924
##   0.0385  0.7787879  0.3797924
##   0.0410  0.7787879  0.3797924
##   0.0435  0.7850379  0.3766889
##   0.0460  0.7850379  0.3766889
##   0.0485  0.7850379  0.3766889
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.026.
```

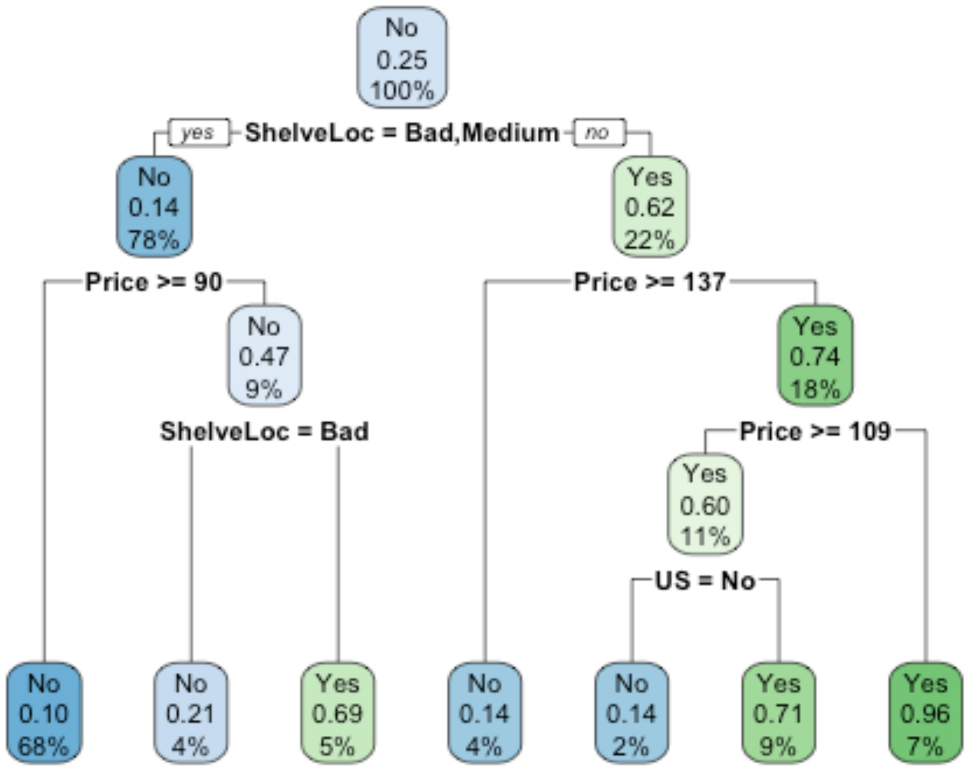
Plot the results of parameter tuning.

```
# plot the cross-validation results
plot(dt.cv)
```



So, we got the best value for the *cp* parameter: 0.026. Since it suggests a simpler model (smaller tree) than the previous one (tree2), we can **prune** the second tree using this *cp* value.

```r
# prune the tree2 using the new cp value
optimal_cp <- dt.cv$bestTune$cp
tree3 <- prune(tree2, cp = optimal_cp)

# plot the new tree
rpart.plot(tree3)
```



Create predictions for the *tree3*.

```r
# make the predictions with tree3 over the test dataset
tree3.pred <- predict(tree3, newdata = test.data, type = "class")

# create the confusion matrix for tree3 predictions
tree3.cm <- table(true = test.data$HighSales, predicted = tree3.pred)
tree3.cm

##      predicted
## true  No Yes
##   No  59   1
##   Yes  7  12
```

Compute evaluation metrics for the *tree3*.

```
# compute the evaluation metrics
tree3.eval <- compute.eval.metrics(tree3.cm)
tree3.eval
```

```
##  accuracy precision    recall        F1
## 0.8987342 0.8939394 0.9833333 0.9365079
```

Let's compare all 3 models we have built so far.

```
# compare the evaluation metrics for tree1, tree2 and tree3
data.frame(rbind(tree1.eval, tree2.eval, tree3.eval),
           row.names = c(paste("tree", 1:3, sep = "_")))
```

```
##         accuracy precision    recall        F1
## tree_1 0.9240506 0.9500000 0.9500000 0.9500000
## tree_2 0.8987342 0.9193548 0.9500000 0.9344262
## tree_3 0.8987342 0.8939394 0.9833333 0.9365079
```

The 2nd and the 3rd model have the same accuracy but differ in terms of precision and recall. To look for a better model, we might consider altering some other parameters. Another option is to reduce the number of variables that are used for model building.

TASK: Create a new tree (tree4) by using only variables that proved relevant in the previous models (tree1, tree2, tree3). Evaluate the model on the test set and compare the evaluation metrics with those obtained for the previous three models.