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 ...  
## $ 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(7)  
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.

# load rpart library  
library(rpart)

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

?rpart

# rpart uses random sampling, so, we have to set the seed value before calling the function  
set.seed(7)  
# 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’.

# print the model  
print(tree1)

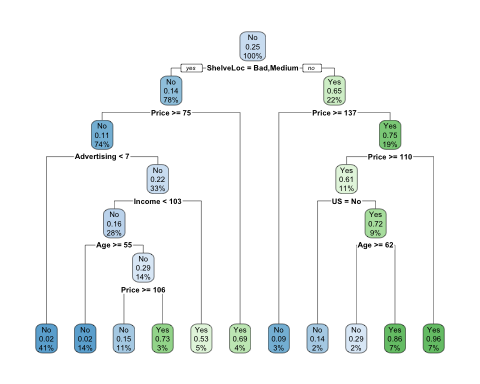
## n= 321   
##   
## node), split, n, loss, yval, (yprob)  
## \* denotes terminal node  
##   
## 1) root 321 80 No (0.75077882 0.24922118)   
## 2) ShelveLoc=Bad,Medium 250 34 No (0.86400000 0.13600000)   
## 4) Price>=75 237 25 No (0.89451477 0.10548523)   
## 8) Advertising< 6.5 131 2 No (0.98473282 0.01526718) \*  
## 9) Advertising>=6.5 106 23 No (0.78301887 0.21698113)   
## 18) Income< 102.5 89 14 No (0.84269663 0.15730337)   
## 36) Age>=55 44 1 No (0.97727273 0.02272727) \*  
## 37) Age< 55 45 13 No (0.71111111 0.28888889)   
## 74) Price>=106 34 5 No (0.85294118 0.14705882) \*  
## 75) Price< 106 11 3 Yes (0.27272727 0.72727273) \*  
## 19) Income>=102.5 17 8 Yes (0.47058824 0.52941176) \*  
## 5) Price< 75 13 4 Yes (0.30769231 0.69230769) \*  
## 3) ShelveLoc=Good 71 25 Yes (0.35211268 0.64788732)   
## 6) Price>=136.5 11 1 No (0.90909091 0.09090909) \*  
## 7) Price< 136.5 60 15 Yes (0.25000000 0.75000000)   
## 14) Price>=109.5 36 14 Yes (0.38888889 0.61111111)   
## 28) US=No 7 1 No (0.85714286 0.14285714) \*  
## 29) US=Yes 29 8 Yes (0.27586207 0.72413793)   
## 58) Age>=61.5 7 2 No (0.71428571 0.28571429) \*  
## 59) Age< 61.5 22 3 Yes (0.13636364 0.86363636) \*  
## 15) Price< 109.5 24 1 Yes (0.04166667 0.95833333) \*

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

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

Plot the tree.

# 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.

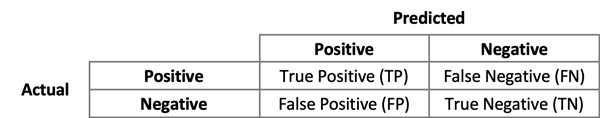
# make the predictions with tree1 over the test dataset  
tree1.pred <- predict(object = tree1, newdata = test.data, type = "class")

Examine what the predictions look like.

# print several predictions  
head(tree1.pred)

## 3 6 13 23 24 26   
## No Yes No No No Yes   
## 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).



Confusion Matrix

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

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

## predicted  
## true No Yes  
## No 53 7  
## Yes 9 10

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 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 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.

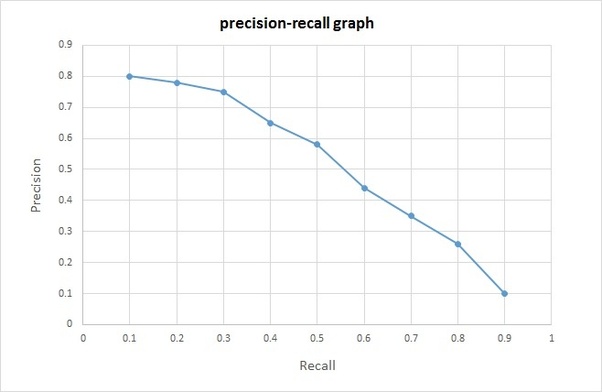
Two other important evaluation measures are *Accuracy* and *F1-measure*.

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

, where N is the total number of predictions.

**F1-measure** conveys the model performance when Precision and Recall are balanced. The formula is:

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 our model when Precision and Recall are balanced.



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.

# 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.

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

## accuracy precision recall F1   
## 0.7974684 0.8548387 0.8833333 0.8688525

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:

# 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.

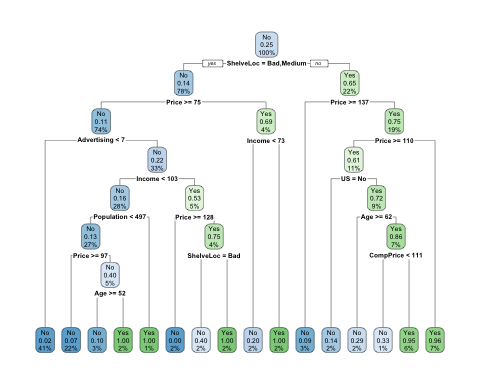
We will decrease the values of both parameters to grow a larger tree.

# build the second model with minsplit = 10 and cp = 0.001  
set.seed(7)  
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 (0.75077882 0.24922118)   
## 2) ShelveLoc=Bad,Medium 250 34 No (0.86400000 0.13600000)   
## 4) Price>=75 237 25 No (0.89451477 0.10548523)   
## 8) Advertising< 6.5 131 2 No (0.98473282 0.01526718) \*  
## 9) Advertising>=6.5 106 23 No (0.78301887 0.21698113)   
## 18) Income< 102.5 89 14 No (0.84269663 0.15730337)   
## 36) Population< 496.5 86 11 No (0.87209302 0.12790698)   
## 72) Price>=96.5 71 5 No (0.92957746 0.07042254) \*  
## 73) Price< 96.5 15 6 No (0.60000000 0.40000000)   
## 146) Age>=52 10 1 No (0.90000000 0.10000000) \*  
## 147) Age< 52 5 0 Yes (0.00000000 1.00000000) \*  
## 37) Population>=496.5 3 0 Yes (0.00000000 1.00000000) \*  
## 19) Income>=102.5 17 8 Yes (0.47058824 0.52941176)   
## 38) Price>=127.5 5 0 No (1.00000000 0.00000000) \*  
## 39) Price< 127.5 12 3 Yes (0.25000000 0.75000000)   
## 78) ShelveLoc=Bad 5 2 No (0.60000000 0.40000000) \*  
## 79) ShelveLoc=Medium 7 0 Yes (0.00000000 1.00000000) \*  
## 5) Price< 75 13 4 Yes (0.30769231 0.69230769)   
## 10) Income< 72.5 5 1 No (0.80000000 0.20000000) \*  
## 11) Income>=72.5 8 0 Yes (0.00000000 1.00000000) \*  
## 3) ShelveLoc=Good 71 25 Yes (0.35211268 0.64788732)   
## 6) Price>=136.5 11 1 No (0.90909091 0.09090909) \*  
## 7) Price< 136.5 60 15 Yes (0.25000000 0.75000000)   
## 14) Price>=109.5 36 14 Yes (0.38888889 0.61111111)   
## 28) US=No 7 1 No (0.85714286 0.14285714) \*  
## 29) US=Yes 29 8 Yes (0.27586207 0.72413793)   
## 58) Age>=61.5 7 2 No (0.71428571 0.28571429) \*  
## 59) Age< 61.5 22 3 Yes (0.13636364 0.86363636)   
## 118) CompPrice< 111 3 1 No (0.66666667 0.33333333) \*  
## 119) CompPrice>=111 19 1 Yes (0.05263158 0.94736842) \*  
## 15) Price< 109.5 24 1 Yes (0.04166667 0.95833333) \*

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 54 6  
## Yes 11 8

Next, we’ll compute the evaluation metrics.

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

## accuracy precision recall F1   
## 0.7848101 0.8307692 0.9000000 0.8640000

Let’s compare this model to the first one:

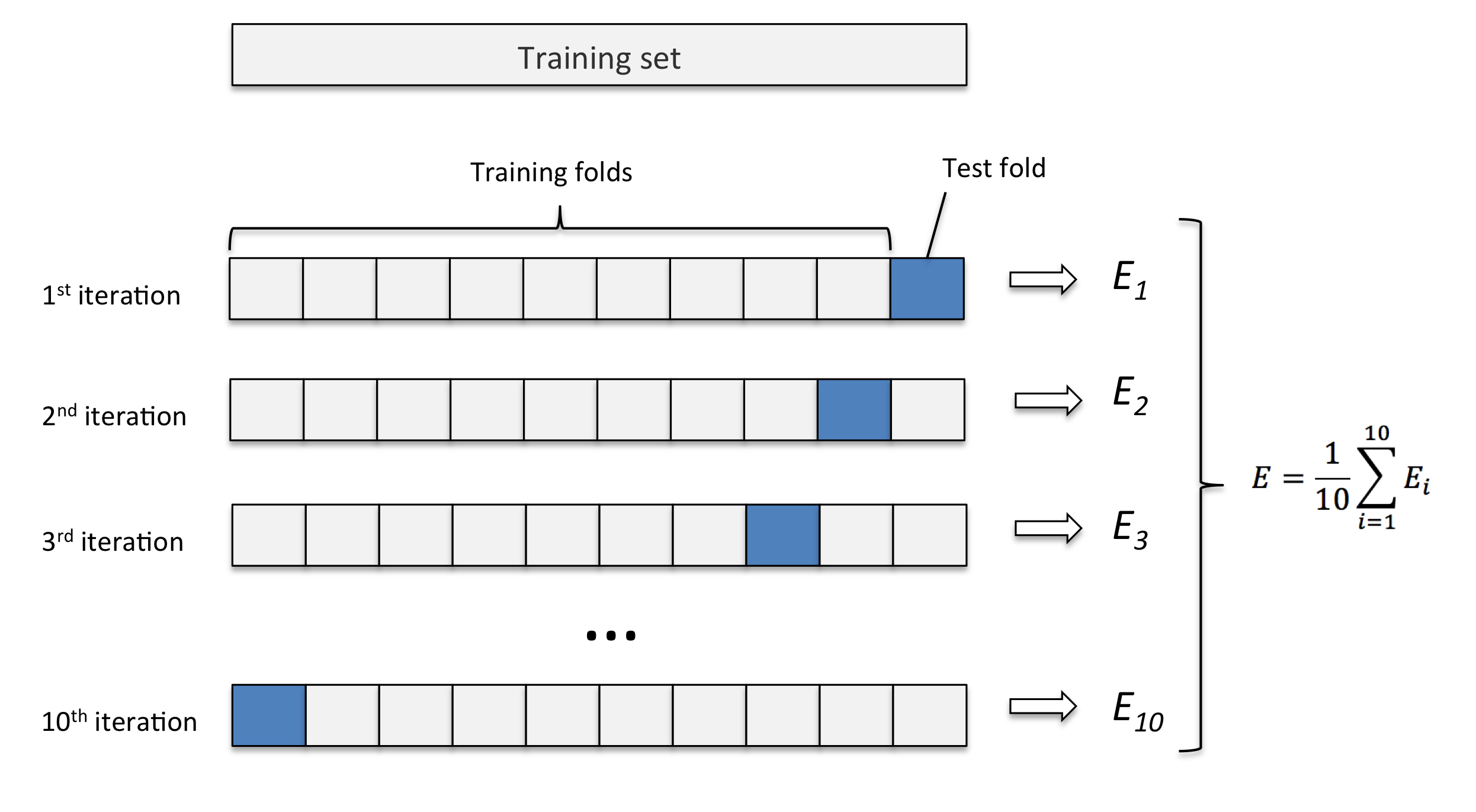
# compare the evaluation metrics for tree1 and tree2  
data.frame(rbind(tree1.eval, tree2.eval),   
 row.names = c("tree\_1", "tree\_2"))

## accuracy precision recall F1  
## tree\_1 0.7974684 0.8548387 0.8833333 0.8688525  
## tree\_2 0.7848101 0.8307692 0.9000000 0.8640000

Except for the recall, the other 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))

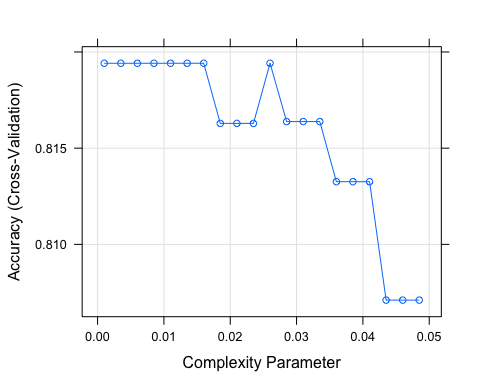
Perform parameter search through cross-validation.

# since cross-validation is a probabilistic process, we need to set the seed so that the results can be replicated   
set.seed(7)  
  
# run the cross-validation  
dt.cv <- train(x = train.data[,-11],   
 y = train.data$HighSales,   
 method = "rpart",   
 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, 289, 289, 288, 289, ...   
## Resampling results across tuning parameters:  
##   
## cp Accuracy Kappa   
## 0.0010 0.8194129 0.5060456  
## 0.0035 0.8194129 0.5060456  
## 0.0060 0.8194129 0.5060456  
## 0.0085 0.8194129 0.5060456  
## 0.0110 0.8194129 0.5060456  
## 0.0135 0.8194129 0.5060456  
## 0.0160 0.8194129 0.5076398  
## 0.0185 0.8162879 0.4965726  
## 0.0210 0.8162879 0.4879479  
## 0.0235 0.8162879 0.4879479  
## 0.0260 0.8194129 0.4909109  
## 0.0285 0.8163826 0.4777412  
## 0.0310 0.8163826 0.4777412  
## 0.0335 0.8163826 0.4777412  
## 0.0360 0.8132576 0.4750657  
## 0.0385 0.8132576 0.4750657  
## 0.0410 0.8132576 0.4750657  
## 0.0435 0.8071023 0.4370566  
## 0.0460 0.8071023 0.4370566  
## 0.0485 0.8071023 0.4370566  
##   
## Accuracy was used to select the optimal model using the largest value.  
## The final value used for the model was cp = 0.026.

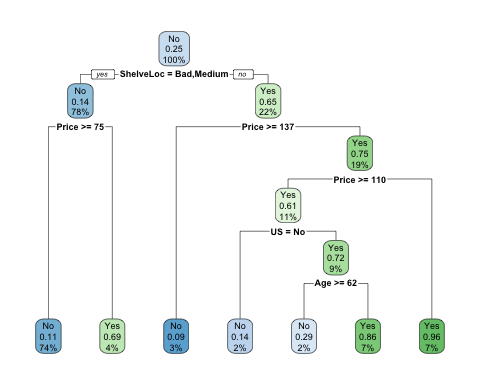
Plot the results of parameter tuning.

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



So, we got the best value for the *cp* parameter: 0.026.

# create a new tree using the new cp value  
optimal\_cp <- dt.cv$bestTune$cp  
set.seed(7)  
tree3 <- rpart(HighSales ~ ., data = train.data, method = "class",  
 control = rpart.control(cp = optimal\_cp))  
  
# plot the new tree  
rpart.plot(tree3)



Create predictions for the *tree3*.

# 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 13 6

Compute evaluation metrics for the *tree3*.

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

## accuracy precision recall F1   
## 0.8227848 0.8194444 0.9833333 0.8939394

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 0.7974684 0.8548387 0.8833333 0.8688525  
## tree\_2 0.7848101 0.8307692 0.9000000 0.8640000  
## tree\_3 0.8227848 0.8194444 0.9833333 0.8939394

The 3rd model outperformed the other two on all the metrics except precision. 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.