[DATA-04] Decision trees

Miguel-Angel Canela, IESE Business School

September 30, 2016

What is a decision tree?

A **decision tree** is a collection of **decision nodes**, connected by branches, extending downward from the **root node**, until terminating in the **leaf nodes**. The usual graphical representation of a decision tree puts the root on top and the leaves at the bottom, as in Figure 1, which shows a tree with 11 decision nodes and 13 leaf nodes. Decision trees can be used for classification and regression purposes. This lecture has a classification perspective.

Binary trees contain two branches per decision node. Any tree can be redesigned as a binary tree. Note that the tree can be binary without the predicted attribute being binary (this is binary classification). The discussion of this lecture is restricted to binary trees and binary classification.

Note. The decision trees discussed here are not the same as those used in decision analysis, in which there are two types of nodes, event nodes and decision nodes.

Decision tree algorithms

The top popular methods for developing decision trees are those based on the CART and C4.5 algorithms. These algorithms were developed independently. **CART** (Classification And Regression Trees) was designed with both regression and classification in view, while the ancestor of C4.5, the ID3 method, was thought as a classification method.

- The CART algorithm produces binary trees. It proceeds by recursively splitting the data set. At each step, the best possible split is selected. The selection of the splits in CART is based on a least squares approach (the same as linear regression). Nevertheless, in many textbooks in which decision trees are discussed from a classification perspective, the CART algorithm is presented as maximizing the **heterogeneity** of the branches, based on a measure of heterogeneity called the **Gini diversity index**.
- The C4.5 algorithm is not restricted to binary trees. It uses the **entropy**, a formula from information theory, as a measure of heterogeneity.

In practice, the predictions givened by tree models developed by these two methods are similar, although the trees may look different, even with binary attributes. I describe briefly now the method used for selecting a split in the CART algorithm, as implemented in the package `rpart`.

The idea is simple: we take, as the predicted variable, a dummy variable for the positive class, and, as the predicted values of this variable, the scores . Then, the difference between the actual value of the predicted variable and the score can be taken as a residual. The **optimal split** is the one for which the sum of squared residuals is minimum.

This argument can be presented in a different way. Let me suppose first that the attributes used for the classification are dummy variables. You may remember, from your experience with the regression line, that a lower residual sum of squares is the same as a stronger correlation (in absolute value). So, the best split is that given by the attribute whose correlation with the predicted variable is stronger. It is easy to see that high correlation comes from having a low proportion of positive cases in one of the two groups produced by the split and a high proportion in the other group. So, this is what the CART algorithm creates.

If an attribute is numeric, the same idea is applied to decide how interesting a split based on that attribute may be, but the attribute is first binarized with a cutoff which is selected using the same principle.

Pruning

Decision trees are usually pruned. **Pruning** reduces the size of the tree by removing parts that provide little power to classify instances. The dual goal of pruning is to reduce the complexity of the final classifier as well as to prevent overfitting. A common strategy is to grow the tree until each node contains a small number of instances, and then remove the nodes that do not provide additional information.

There are many techniques for tree pruning, which differ in the measurement used to optimize the performance. For instance, CART pruned trees use a **cost-complexity model**. I do not discuss in depth here these technical issues here, because they are beyond the scope of this course. For more detail, you may take a look at references below.

Nevertheless, in any implementation of these algorithms, there is a default pruning rule, which is applied when nothing is said. If you are not satisfied, you can use an option for specifying how aggressively you want to prune. In the tree of Figures 1 and 2, the default of `rpart` gives a tree with 13 leaves, which is very low, considering that there are 51 variables fopr the prediction. By tampering with the **complexity parameter**, we can get more or less pruning, as we will see in the example.

The class imbalance problem

The example of this lecture uses a decision tree classifer, dealing also with two relevant issues in classification, class imbalance and validation. A two-class data set is said to be **imbalanced** when one of the classes is heavily under-represented in comparison to the other class. This is particularly relevant in the real world applications, such as diagnosis of rare diseases, fraud detection and filtering tasks. Most classification methods show a poor performance under class imbalance. A typical problem is that the proportion of instances assigned by the classifier to the minority class is too low for a classification model to be useful.

Typical remedies for populations with class imbalance are:

- In methods that produce scores, such as logistic regression, to use a low cutoff. Setting the cutoff as the proportion of the minority class in the target population usually works, except when this proportion is very low (say less than 5%).
- To use a training set in which the proportion of the minority class is increased. In this case, a statistic like the accuracy does not make sense. Nevertheless, we can evaluate the model through the TP rate and the FP rate, which still make sense for an artificially balanced data set. The example of this lecture illustrates this approach.

What is the validation?

A typical problem of predictive models is **overfitting**. A complex classifier can be right when predicting the training instances, but fail with new instances. Broadly speaking, the **validation** of a model consists in checking that the model works as expected on data which have not been used to develop it.

In the simplest approach to validation, we develop the classifier in a **training set**, trying it on a **test set**. The training and test sets can be predefined (e.g. January and February data) or can be obtained from a **random split** of a unique data set.

More sophisticated methods for validation are usually available in specialized software. In k-**fold cross-validation**, a data set is partitioned into k subsets and each of the subsets is used as a test set for a model developed on a training set resulting from merging the other $k-1$ subsets. $k=10$ is a typical choice. As a general rule, we can say that, although the idea of the split is straightforward, we do not split data sets when they are not big enough (how big depends on the complexity of the model), so cross-validation is a better approach for data sets of moderate size.

Example: The spam filter

In this example, I develop a **spam filter**, that is, an algorithm which classifies e-mail messages as either spam or non-spam, based on a collection of attributes such as the frequency of certain words or characters. I use data collected at Hewlett-Packard, merging a collection of spam e-mail from the company postmaster and the individuals who had filed spam with a collection of non-spam e-mail, extracted from filed work and personal e-mail.

I have to take into account that the proportion of false positives, that is, of non-spam messages wrongly classified as spam, is expected to be very low in a good spam filter.

The data set contains data on 4,601 e-mail messages. Among these messages, 1,813 have been classified as spam. The variables are:

- A dummy for the e-mail being considered spam (`spam`).
- 48 numeric variables whose names start with 'word_', followed by a word. They indicate the **frequency**, in percentage scale, with which that word appears in the message. Example: `word_make=0.21` , for a particular message, means that 0.21% of the words in the message match the word 'make'.
- 3 numeric variables indicating, respectively, the average length of uninterrupted sequences of capital letters, the length of the longest uninterrupted sequence of capital letters and the total number of capital letters in the message.

I import the data with `read.csv` . The data set has 4,601 rows and 52 columns.

```
spam <- read.csv(file="spam.csv")
str(spam)
```

```
## 'data.frame':    4601 obs. of  52 variables:
##  $ word_make      : num  0 0.21 0.06 0 0 0 0 0 0.15 0.06 ...
##  $ word_address   : num  0.64 0.28 0 0 0 0 0 0 0 0.12 ...
##  $ word_all       : num  0.64 0.5 0.71 0 0 0 0 0 0.46 0.77 ...
##  $ word_3d        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_our       : num  0.32 0.14 1.23 0.63 0.63 1.85 1.92 1.88 0.61 0.19 ...
##  $ word_over      : num  0 0.28 0.19 0 0 0 0 0 0 0.32 ...
##  $ word_remove    : num  0 0.21 0.19 0.31 0.31 0 0 0 0.3 0.38 ...
##  $ word_internet  : num  0 0.07 0.12 0.63 0.63 1.85 0 1.88 0 0 ...
##  $ word_order     : num  0 0 0.64 0.31 0.31 0 0 0 0.92 0.06 ...
##  $ word_mail      : num  0 0.94 0.25 0.63 0.63 0 0.64 0 0.76 0 ...
##  $ word_receive   : num  0 0.21 0.38 0.31 0.31 0 0.96 0 0.76 0 ...
##  $ word_will      : num  0.64 0.79 0.45 0.31 0.31 0 1.28 0 0.92 0.64 ...
##  $ word_people    : num  0 0.65 0.12 0.31 0.31 0 0 0 0 0.25 ...
##  $ word_report    : num  0 0.21 0 0 0 0 0 0 0 0 ...
##  $ word_addresses : num  0 0.14 1.75 0 0 0 0 0 0 0.12 ...
##  $ word_free      : num  0.32 0.14 0.06 0.31 0.31 0 0.96 0 0 0 ...
##  $ word_business  : num  0 0.07 0.06 0 0 0 0 0 0 0 ...
##  $ word_email     : num  1.29 0.28 1.03 0 0 0 0.32 0 0.15 0.12 ...
##  $ word_you       : num  1.93 3.47 1.36 3.18 3.18 0 3.85 0 1.23 1.67 ...
##  $ word_credit    : num  0 0 0.32 0 0 0 0 0 3.53 0.06 ...
##  $ word_your      : num  0.96 1.59 0.51 0.31 0.31 0 0.64 0 2 0.71 ...
##  $ word_font      : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_000       : num  0 0.43 1.16 0 0 0 0 0 0 0.19 ...
##  $ word_money     : num  0 0.43 0.06 0 0 0 0 0 0.15 0 ...
##  $ word_hp        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_hpl       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_george    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_650       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_lab       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_labs      : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_telnet    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_857       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_data      : num  0 0 0 0 0 0 0 0 0.15 0 ...
##  $ word_415       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_85        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_technology: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_1999      : num  0 0.07 0 0 0 0 0 0 0 0 ...
##  $ word_parts     : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_pm        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_direct    : num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ word_cs        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_meeting   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_original  : num  0 0 0.12 0 0 0 0 0 0.3 0 ...
##  $ word_project   : num  0 0 0 0 0 0 0 0 0 0.06 ...
##  $ word_re        : num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ word_edu       : num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ word_table     : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ word_conference: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ cap_ave        : num  3.76 5.11 9.82 3.54 3.54 ...
##  $ cap_long       : int  61 101 485 40 40 15 4 11 445 43 ...
##  $ cap_total      : int  278 1028 2259 191 191 54 112 49 1257 749 ...
##  $ spam           : int  1 1 1 1 1 1 1 1 1 1 ...
```

For validation purposes, I use in this case a 50-50 split. More exactly, the training set contains 2,301 instances and the test set 2,300 instances. The instances allocated to each subset are selected randomly.

```
train <- sample(1:4601, size=2301, replace=F)
```

The argument `replace=F` is needed here, because the default of `sample` uses **sampling with replacement**, meaning that an instance can be sampled more than once, which does not make sense here. `train` is a numeric vector containing the row numbers for the training instances. Now, `spam[train,]` is the training set. So, `spam[-train,]` is the test set.

```
train[1:50]
```

```
##  [1] 3014  883 2543 2762  713 2798 3125  155 2758 3981 1270 1180 4442 2271
## [15] 3769 4068 3110 3409 1138 2040 3100 2938  486   15  743 2955  524 2025
## [29] 4332 1706 4347  369  131 1961 3906 2981  477 3920 1701 4128 1139 2237
## [43] 3899 1749 1972 4467 1315 4098 3619 1779
```

Next, I specify the formula, with a trick for getting it short. The dot on the right side of the formula stands for all the attributes in the data set except the one at the left side. I set the cutoff at 0.5.

```
fm1 <- spam ~ .
cut <- 0.5
```

I use the function `rpart`, from the package with the same name, to develop a classifier based on a decision tree. First of all, I need to load this package, which is not loaded by default.

```
library(rpart)
```

Please, mind that, for `library` to work, the package requested must be installed, which, in practical terms, means that it must found in a folder called `library` which is located with the rest of your R installation. In my Windows computer, the path is `C:\Program files\R\R-3.1.1\library`. In my Macintosh, it is `/Library/Frameworks/R.framework/Versions/3.2/Resources/library`. In the case of `rpart`, there is no problem for calling it, because, although it is not loaded by default, it comes with the default installation.

The syntax of the function `rpart`, if we use the default for all the arguments (brief discussion later), is the same as in `lm`, with the arguments `formula` and `data`. As the data, I specify here the training data.

```
tree1 <- rpart(formula=fm1, data=spam[train,])
```

This produces an `rpart` object, whose contents can be explored with the function `str`, although I do not recommend this, since the output is quite long and difficult, due to the complexity of `rpart` objects. Also, the `help` function explains the various elements of the object `tree1`.

```
help(rpart.object)
```

`summary` can also be used in this case, but the output is quite long, probably too long for a beginner. For a decision tree, if it is not big, the best option is probably to print the `rpart` object itself.

```
tree1
```

```
## n= 2301
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##   1) root 2301 550.1130000 0.395480200
##     2) word_free< 0.03 1685 309.7234000 0.242730000
##       4) word_remove< 0.02 1522 219.5112000 0.174770000
##         8) word_000< 0.135 1449 173.8399000 0.139406500
##          16) word_money< 0.02 1390 140.0403000 0.113669100
##            32) cap_ave< 2.883 1074  68.0381800 0.067970200 *
##            33) cap_ave>=2.883 316  62.1360800 0.268987300
##              66) word_hp>=0.385 122   0.9918033 0.008196721 *
##              67) word_hp< 0.385 194  47.6288700 0.432989700
##               134) word_george>=0.04 30   0.0000000 0.000000000 *
##               135) word_george< 0.04 164  40.9756100 0.512195100
##                 270) word_edu>=0.185 33   0.9696970 0.030303030 *
##                 271) word_edu< 0.185 131  30.4122100 0.633587800 *
##          17) word_money>=0.02 59  11.1864400 0.745762700 *
##         9) word_000>=0.135 73   7.8904110 0.876712300 *
##       5) word_remove>=0.02 163  17.5460100 0.877300600
##        10) word_george>=0.14 10   0.0000000 0.000000000 *
##        11) word_george< 0.14 153   9.3464050 0.934640500 *
##     3) word_free>=0.03 616  93.5308400 0.813311700
##       6) word_hp>=0.12 57   8.2456140 0.175438600 *
##       7) word_hp< 0.12 559  59.7280900 0.878354200
##        14) word_edu>=0.17 29   4.7586210 0.206896600 *
##        15) word_edu< 0.17 530  41.1792500 0.915094300
##          30) word_george>=0.11 13   0.0000000 0.000000000 *
##          31) word_george< 0.11 517  30.0193400 0.938104400 *
```

Paying a bit of attention, this output is not difficult to read. First, `n=2301` reports the number of training instances. Then, we have one line for each node. First, in the root, we have `1) root 2301 550.1130000 0.395480200`. This tells us that we have 2,301 instances in the root node, that the average of the predicted variable (`spam`), that is, the spam rate, is 39.5%, and that the sum of squared residuals (actual value minus mean) is 550.11. The residual sum of squares is called **deviance** here.

In the next line, we have the second node, which contains the 1,685 instances with `word_free<0.03` , with a 24.3% spam rate. The other branch resulting from the root node split ends in node 3, which is reported when all the branching stemming from node 2 is finished. We have there the 616 instances satisfying `word_free>=0.03` , with a 81.3% spam rate. This is the optimal split. What is optimal? That the spam rate is low in one branch and high in the other branch, making classification easier.

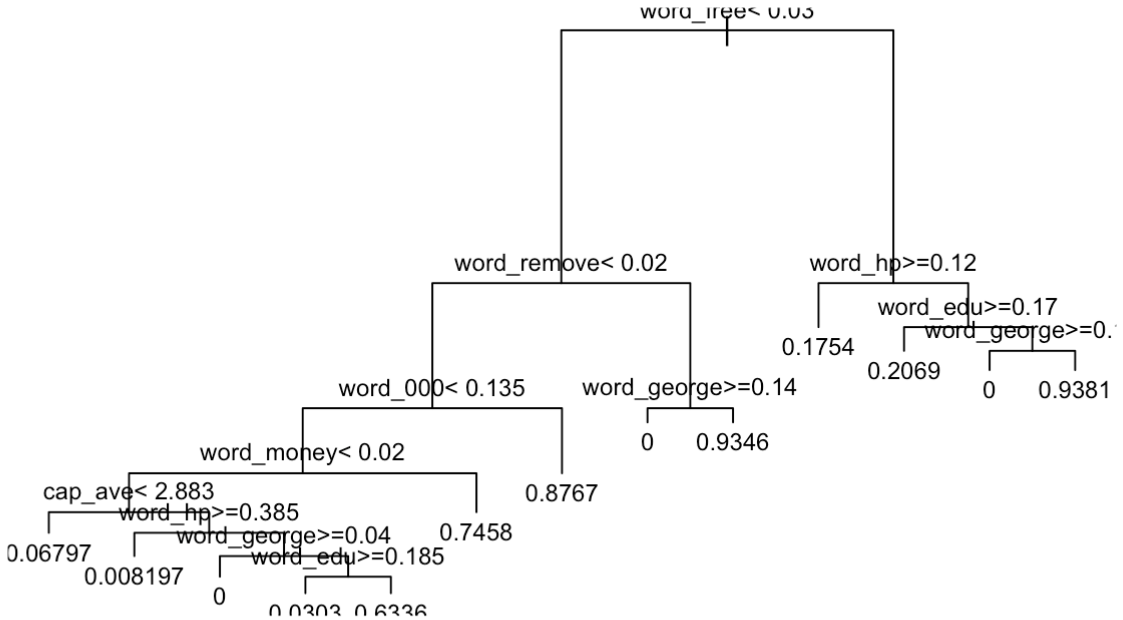
The leaf nodes are marked with an asterisk. For instance, node 32. We have there the 52 instances that satisfy `word_free<0.03 & word_remove<0.02 & word_000<0.135 & word_money<0.02 & cap_ave<2.883` . The spam rate in this subset is 6.8%. So the predictive score for all these instances is 0.068. This is what the `predict` function will give us.

Since there are 13 leaf nodes, this tree gives us a partition of the training set in 13 subsets. In each subset, all the instances have the same score. To use the model on a new instance, the classifier puts this instance in the appropriate leaf, so that it gets the corresponding score. Note that, to build the partition, we are using only 8 variables, out of the 51 variables available. This is due to the pruning performed by `rpart` .

The tree can be plotted with the `plot` function, which prints the tree on the graphics device. Then `text` prints the labels. See this in Figure 1.

```
plot(tree1, main="Figure 1. Decision tree for spam filtering")
text(tree1, font=1, cex=0.75)
```

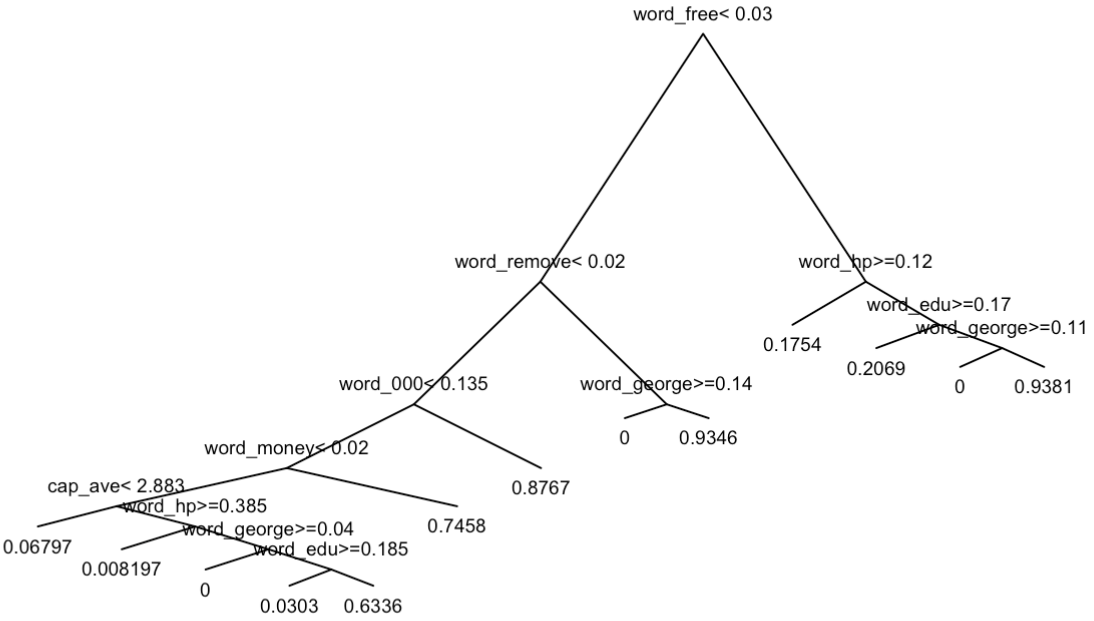
Figure 1. Decision tree for spam filtering



Playing with the graphical arguments, you can obtain other formats. See an improved example in Figure 2. First, with the argument `margin`, we put extra of white space around the borders of the tree, to prevent labels to get cut off. Second, the argument `branch` controls the shape of the branches from parent to child node. Finally, the argument `cex` controls the font size in the labels.

```
plot(tree1, branch=0, main="Figure 2. Alternative layout", margin=0.01)
text(tree1, font=1, cex=0.6)
```

Figure 2. Alternative layout



The function `predict` works for `rpart` the same as for `lm`. I first get the scores for the training instances, deriving the confusion matrix from them as for the logistic regression models.

```
pred_train1 <- predict(tree1, newdata=spam[train,])
conf_train1 <- table(pred_train1 > cut, spam[train,]$spam==1); conf_train1
```

```
##
##         FALSE TRUE
##   FALSE  1277   91
##   TRUE    114  819
```

```
tp_train1 <- conf_train1["TRUE", "TRUE"]/sum(conf_train1[,"TRUE"]); tp_train1
```

```
## [1] 0.9
```

```
fp_train1 <- conf_train1["TRUE", "FALSE"]/sum(conf_train1[,"FALSE"]); fp_train1
```

```
## [1] 0.08195543
```

Note that, in this case, the data set has been artificially created, by joining two collections which come from different sources, avoiding a class imbalance situation. So, the proportion of spam is probably much higher than the real one. This means that a statistic like the accuracy does not make sense. Nevertheless, we can evaluate the classifier examining the two columns of the confusion matrix separately, as we do when we look at the TP and FP rates. These two figures make sense separately, since the instances of the test set are classified independently from one another.

The TP rate is great, but the FP rate, from the perspective of the implementation in a spam filter, is a bit high. Can we improve this with a better cutoff? Instead of exploring this with histograms, as I did for my logistic fregression example, it is better to use a table here, since there are only 13 different scores, one for each leaf of the tree. With the function `table`, I get a very clear view.

```
table(pred_train1, spam$spam[train])
```

```
##
## pred_train1              0    1
##   0                     53    0
##   0.00819672131147541  121    1
##   0.0303030303030303    32    1
##   0.0679702048417132  1001   73
##   0.175438596491228     47   10
##   0.206896551724138     23    6
##   0.633587786259542     48   83
##   0.745762711864407     15   44
##   0.876712328767123      9   64
##   0.934640522875817     10  143
##   0.938104448742747     32  485
```

We see in the table a wide gap between 0.207 and 0.634. This is a by-product of the tree algorithm, which selects the splits so that they produce extreme scores. Here, since the data set has been built artificially, using the spam rate as a cutoff would not make sense. Moreover, lowering the cutoff would increase the proportion of positives (true and false) and, in particular, the FP rate would increase, an undesired effect. But, raising the cutoff, the FP rate would decrease, which could be interesting. It also makes sense from a common sense perspective, because it means that we do not label e-mail as spam unless we have stronger evidence.

To illustrate this, I raise the cutoff to 0.7, which means moving one leaf from the spam basket to the non-spam basket. More specifically, we are going to have 131 less spam messages. 83 of these messages were spam, but this improves the FP rate, as seen below.

```
cut <- 0.7
pred_train1 <- predict(tree1, newdata=spam[train,])
conf_train1 <- table(pred_train1 > cut, spam[train,]$spam==1); conf_train1
```

```
##
##         FALSE TRUE
##   FALSE  1325  174
##   TRUE     66  736
```

```
tp_train1 <- conf_train1["TRUE", "TRUE"]/sum(conf_train1[,"TRUE"]); tp_train1
```

```
## [1] 0.8087912
```

```
fp_train1 <- conf_train1["TRUE", "FALSE"]/sum(conf_train1[,"FALSE"]); fp_train1
```

```
## [1] 0.04744788
```

So, now I get an FP rate below 5%, more reasonable, although losing some ability to filter out spam. I do not continue this argument, which you can obviously do. I address the validation issue now, checking the confusion matrix for the test set.

```
cut <- 0.5
pred_test1 <- predict(tree1, newdata=spam[-train,])
conf_test1 <- table(pred_test1 > cut, spam[-train,]$spam==1); conf_test1
```

```
##
##         FALSE TRUE
##   FALSE  1260   83
##   TRUE    137  820
```

```
tp_test1 <- conf_test1["TRUE", "TRUE"]/sum(conf_test1[,"TRUE"]); tp_test1
```

```
## [1] 0.9080842
```

```
fp_test1 <- conf_test1["TRUE", "FALSE"]/sum(conf_test1[,"FALSE"]); fp_test1
```

```
## [1] 0.09806729
```

The results for the test set are similar to those obtained for the training set. What does this mean? That the tree model that I have developed is not complex enough to overfit the data. This is not a surprise, since the tree has only 13 leaves, which is low for 2,301 instances. So, pruning prevents overfitting.

In `rpart`, pruning is controlled by complexity parameter, specified in the argument `cp`. Any split that does not decrease the overall lack of fit by a factor equal to the specified value `cp` is not attempted. The main role of this parameter is to save computing time by pruning off splits that are obviously not worthwhile. The default is `cp=0.01`.

To show you how this works, I change the specification to `cp=0.005`.

```
tree2 <- rpart(formula=fm1, data=spam[train,], cp=0.005)
tree2
```

```
## n= 2301
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##    1) root 2301 550.1130000 0.395480200
##      2) word_free< 0.03 1685 309.7234000 0.242730000
##        4) word_remove< 0.02 1522 219.5112000 0.174770000
##          8) word_000< 0.135 1449 173.8399000 0.139406500
##           16) word_money< 0.02 1390 140.0403000 0.113669100
##             32) cap_ave< 2.883 1074  68.0381800 0.067970200
##               64) word_you< 3.045 892  36.3811700 0.042600900 *
##               65) word_you>=3.045 182  28.2692300 0.192307700
##                130) word_business< 0.07 173  22.0924900 0.150289000
##                  260) word_internet< 0.295 166  16.8253000 0.114457800 *
##                  261) word_internet>=0.295 7   0.0000000 1.000000000 *
##                131) word_business>=0.07 9   0.0000000 1.000000000 *
##             33) cap_ave>=2.883 316  62.1360800 0.268987300
##               66) word_hp>=0.385 122   0.9918033 0.008196721 *
##               67) word_hp< 0.385 194  47.6288700 0.432989700
##                134) word_george>=0.04 30   0.0000000 0.000000000 *
##                135) word_george< 0.04 164  40.9756100 0.512195100
##                  270) word_edu>=0.185 33   0.9696970 0.030303030 *
##                  271) word_edu< 0.185 131  30.4122100 0.633587800
##                    542) word_will>=1.89 11   0.0000000 0.000000000 *
##                    543) word_will< 1.89 120  25.5916700 0.691666700
##                     1086) word_your< 0.525 79  19.3670900 0.569620300 *
##                     1087) word_your>=0.525 41   2.7804880 0.926829300 *
##           17) word_money>=0.02 59  11.1864400 0.745762700
##             34) word_hp>=0.635 7   0.0000000 0.000000000 *
##             35) word_hp< 0.635 52   6.7692310 0.846153800 *
##          9) word_000>=0.135 73   7.8904110 0.876712300 *
##        5) word_remove>=0.02 163  17.5460100 0.877300600
##         10) word_george>=0.14 10   0.0000000 0.000000000 *
##         11) word_george< 0.14 153   9.3464050 0.934640500 *
##      3) word_free>=0.03 616  93.5308400 0.813311700
##        6) word_hp>=0.12 57   8.2456140 0.175438600
##         12) word_free< 0.4 40   0.0000000 0.000000000 *
##         13) word_free>=0.4 17   4.1176470 0.588235300 *
##        7) word_hp< 0.12 559  59.7280900 0.878354200
##         14) word_edu>=0.17 29   4.7586210 0.206896600
##           28) word_you>=0.235 22   0.0000000 0.000000000 *
##           29) word_you< 0.235 7   0.8571429 0.857142900 *
##         15) word_edu< 0.17 530  41.1792500 0.915094300
##           30) word_george>=0.11 13   0.0000000 0.000000000 *
##           31) word_george< 0.11 517  30.0193400 0.938104400
##             62) cap_total< 64 48  10.3125000 0.687500000 *
##             63) cap_total>=64 469  16.3838000 0.963752700 *
```

I get now a tree with 22 leaves, involving 14 variables.

```
pred_train2 <- predict(tree2, newdata=spam[train,])
conf_train2 <- table(pred_train2 > cut, spam[train,]$spam==1); conf_train2
```

```
##
##         FALSE TRUE
##   FALSE  1287   59
##   TRUE    104  851
```

```
tp_train2 <- conf_train2["TRUE", "TRUE"]/sum(conf_train2[,"TRUE"]); tp_train2
```

```
## [1] 0.9351648
```

```
fp_train2 <- conf_train2["TRUE", "FALSE"]/sum(conf_train2[,"FALSE"]); fp_train2
```

```
## [1] 0.07476636
```

The confusion matrix looks a bit better, but I should check that this does not come at the price of overfitting, which I do using the test set.

```
pred_test2 <- predict(tree2, newdata=spam[-train,])
conf_test2 <- table(pred_test2 > cut, spam[-train,]$spam==1); conf_test2
```

```
##
##         FALSE TRUE
##   FALSE  1243   69
##   TRUE    154  834
```

```
tp_test2 <- conf_test2["TRUE", "TRUE"]/sum(conf_test2[,"TRUE"]); tp_test2
```

```
## [1] 0.923588
```

```
fp_test2 <- conf_test2["TRUE", "FALSE"]/sum(conf_test2[,"FALSE"]); fp_test2
```

```
## [1] 0.1102362
```

The TP rate is still very high, but the FP rate jumps to 11%. So, the improvement is unclear.

Questions

1. What if you change the cutoff for the second tree?
2. Develop a spam filter based on a logistic regression equation and compare your model with the model presented in the example.
3. Drop the three `cap_` variables and binarize all the `word_` variables, transforming them into dummies for the occurrence of the corresponding word. Develop a spam filter using this binarized data set and compare your results with those presented in the example.

References

[1] DT Larose (2005), *Discovering Knowledge in Data*, Wiley.

[2] IH Witten, E Frank & MA Hall (2011), *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann.