Chapter 2
CART

Abstract CART stands for Classification And Regression Trees, and refers to a statistical method for constructing tree predictors (also called decision trees) for both regression and classification problems. This chapter focuses on CART trees, analyzing in detail the two steps involved in their construction: the maximal tree growing algorithm, which produces a large family of models, and the pruning algorithm, which is used to select an optimal or suitable final one. The construction is illustrated on the spam dataset using the rpart package. The chapter then addresses interpretability issues and how to use competing and surrogate splits. In the final section, trees are applied to two examples: predicting ozone concentration and analyzing genomic data.

2.1 The Principle

CART stands for Classification And Regression Trees, and refers to a statistical method, introduced by Breiman et al. (1984), for constructing tree predictors (also called decision trees) for both regression and classification problems.

Let us start by considering a very simple classification tree on our running example about spam detection (Fig. 2.1).

A CART tree is an upside-down tree: the root is at the top. The leaves of the tree are the nodes without descendants (for this example, 5 leaves) and the other nodes of the tree are nonterminal nodes (4 such nodes including the root) that have two child nodes. Hence, the tree is said to be binary. Nonterminal nodes are labeled by a condition (a question) and leaves by a class label or a value of the response variable. When a tree is given, it is easy to use it for prediction. Indeed, to determine the predicted value $\widehat{y}$ for a given x, it suffices to go through the only path from the root to a leaf, by answering the sequence of questions given by the successive splits and reading the value of y labeling the reached leaf. When you go through the tree, the rule is as follows: if the condition is verified then you go to the left node and if not, go to the right. In our example, an email with proportions of occurrences of characters "!" and "$", respectively, larger than 7.95 and 0.65% will thus be predicted as spam by this simple tree.

R. Genuer and J.-M. Poggi, *Random Forests with R*, Use R!,
https://doi.org/10.1007/978-3-030-56485-8_2

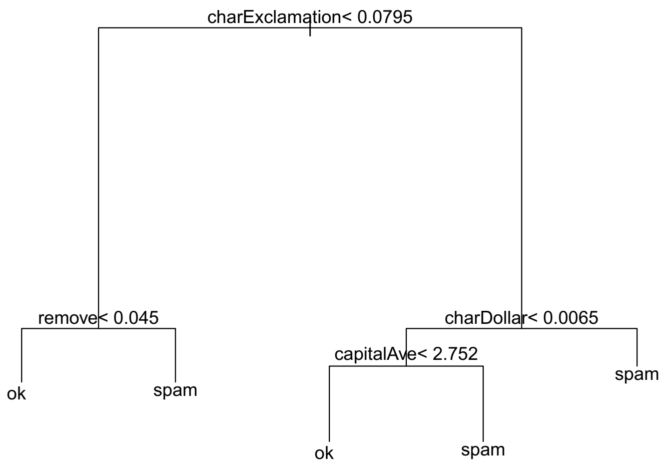

Fig. 2.1 A very simple first tree, `spam` data

Other methods for building decision trees, sometimes introduced before CART, are available, such as CHAID (Kass 1980) and C4.5 (Quinlan 1993). Tree methods are still of interest today, as can be seen in Patil and Bichkar (2012) in computer science and Loh (2014) in statistics. Several variants for building CART trees are possible, for example, by changing the family of admissible splits, the cost function, or the stopping rule. We limit ourselves in the sequel to the most commonly used variant, which is presented in Breiman et al. (1984). The latter contains many variants which have not been widely disseminated and implemented. Indeed, the success of the simplest version has been ensured by its ease of interpretation. A concise and clear presentation in French of the regression CART method can be found in Chap. 2 of Gey (2002) PhD thesis.

CART proceeds by recursive binary partitioning of the input space $\mathcal{X}$ and then determines an optimal sub-partition for prediction. Building a CART tree is therefore a two-step process. First, the construction of a maximal tree and the second step, called pruning, which builds a sequence of optimal subtrees pruned from the maximal tree sufficient from an optimization perspective.

2.2 Maximal Tree Construction

At each step of the partitioning process, a part of the space previously obtained is split into two pieces. We can therefore naturally associate a binary tree to the partition built step by step. The nodes of the tree correspond to the elements of the partition. For example, the root of the tree is associated with the entire input space, its two child nodes with the two subspaces obtained by the first split, and so on. Figure 2.2

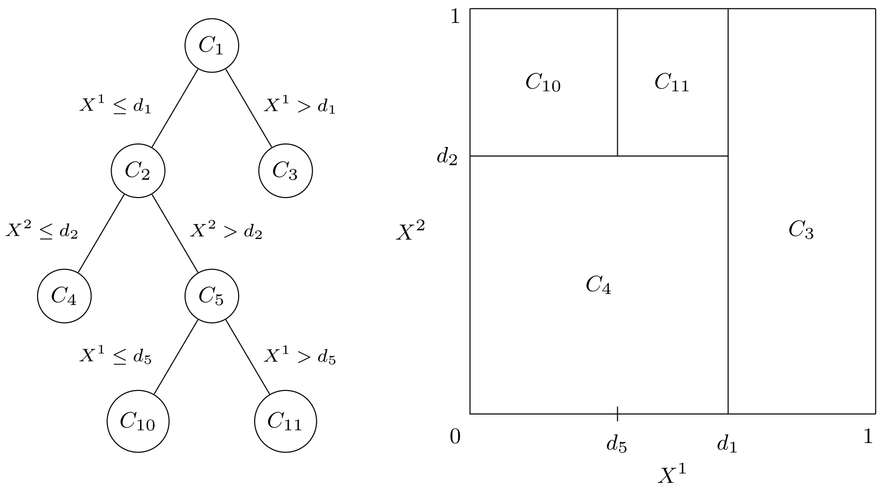

Fig. 2.2 Left: a classification tree to predict the class label corresponding to a given x. Right: the associated partition in the explanatory variables' space, here the unit square (C_1, C_2, and C_5 do not appear because they are not associated with leaves)

illustrates the correspondence between a binary tree and the associated partition of the explanatory variables' space (here the unit square).

Let us now detail the splitting rule. To make things simple, we limit to continuous explanatory variables while mentioning the qualitative case, whenever necessary. The input space is then $\mathbb{R}^p$, where p is the number of variables. Let us consider the root of the tree, associated with the entire space $\mathbb{R}^p$, which contains all the observations of the learning sample $\mathcal{L}_n$. The first step of CART is to optimally split this root into two child nodes, and then this splitting is repeated recursively in a similar way. We call a split an element of the form

$$\{X^j \le d\} \cup \{X^j > d\},$$

where $j \in \{1, \dots, p\}$ and $d \in \mathbb{R}$. Splitting according to $\{X^j \le d\} \cup \{X^j > d\}$ means that all observations whose value of the jth variable is smaller than d go into the left child node, and all those whose value is larger than d go into the right child node. The method then looks for the best split, i.e., the couple (j, d) that minimizes a certain cost function:

- In the regression case, one tries to minimize the within-group variance resulting from splitting a node t into two child nodes t_L and t_R, the variance of a node t being defined by $V(t) = \frac{1}{\#t} \sum_{i:x_i \in t} (y_i - \overline{y}_t)^2$ where $\overline{y}_t$ and $\#t$ are, respectively, the average and the number of the observations y_i belonging to the node t. We are therefore seeking to maximize:

$$V(t) - \left(\frac{\#t_L}{\#t} V(t_L) + \frac{\#t_R}{\#t} V(t_R) \right).$$

- In the classification case, the possible labels are $\{1, \ldots, C\}$, and the impurity of the child nodes is most often quantified through the Gini index. The Gini index of a node t is defined by $\Phi(t) = \sum_{c=1}^{C} \hat{p}_t^c (1 - \hat{p}_t^c)$, where $\hat{p}_t^c$ is the proportion of observations of class c in the node t. We are then led, for any node t and any admissible split, to maximize

$$\Phi(t) - \left(\frac{\#t_L}{\#t} \Phi(t_L) + \frac{\#t_R}{\#t} \Phi(t_R) \right).$$

It should be emphasized that at each node, the search for the best split is made among all the variables. Thus, a variable can be used in several splits (or only one time or never).

In regression, we are therefore looking for splits that tend to reduce the variance of the resulting nodes. In classification, we try to decrease the Gini purity function, and thus to increase the homogeneity of the obtained nodes, a node being perfectly homogeneous if it contains only observations of the same class label. It should be noted that the homogeneity of the nodes could be measured by another function, such as the misclassification rate, but this natural choice does not lead to a strictly concave purity function guaranteeing the uniqueness of the optimum at each split. This property, while not absolutely essential from a statistical point of view, is useful from a computational point of view by avoiding ties for the best split selection.

In the case of a nominal explanatory variable X^j, the above remains valid except that in this case, a split is simply an element of the form

$$\{X^j \in d\} \cup \{X^j \in \bar{d}\},$$

where d and $\bar{d}$ are not empty and define a partition of the finite set of possible values of the variable X^j.

Remark 2.1 In CART, we can take into account underrepresented classes by using prior probabilities. The relative probability assigned to each class can be used to adjust the magnitude of classification errors for each class. Another way of doing this is to oversample observations from rare classes, which is more or less equivalent to an overweighting of these observations.

Once the root of the tree has been split, we consider each of the child nodes and then, using the same procedure, we look for the best way to split them into two new nodes, and so on. The tree is thus developed until a stopping condition is reached. The most natural condition is not to split a pure node, i.e., a node containing only observations with the same outputs (typically in classification). But this criterion can lead to unnecessarily deep trees. It is often associated with the classical criterion of not splitting nodes that contain less than a given number of observations. The terminal nodes, which are no longer split, are called the leaves of the tree. We will

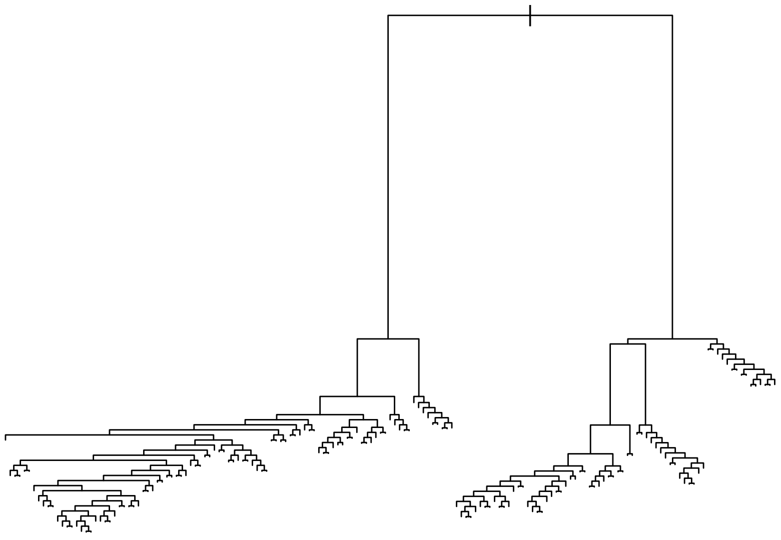

Fig. 2.3 Skeleton of the maximal tree, `spam` data

call the fully developed tree, the maximal tree, and denote it by $T_{\max}$. At the same time, each node t of the tree is associated with a value: $\overline{y}_t$ for regression or the label of the majority class of observations present in the node t in the classification case. Thus, a tree is associated not only with a partition defined by its leaves but also by the values that are attached to each piece of this partition. The tree predictor is then the piecewise constant function associated with the tree (Fig. 2.2).

The skeleton of the maximal tree on `spam` data is plotted in Fig. 2.3. Note that the edges are all not of the same length. In fact, the height of an edge connecting a node to its two children is proportional to the reduction in heterogeneity resulting from the splitting. Thus, for splits close to the root, the homogeneity gains are significant while for those located toward the leaves of the maximal tree, the gains are very small.

2.3 Pruning

Pruning is the second step of the CART algorithm. It consists in searching for the best pruned subtree of the maximal tree, the best in the sense of the generalization error. The idea is to look for a good intermediate tree between the two extremes: the maximal tree which has a high variance and a low bias, and the tree consisting only of the root (which corresponds to a constant predictor) which has a very low variance but a high bias. Pruning is a model selection procedure, where the competing models are the pruned subtrees of the maximal tree, i.e., all binary subtrees of $T_{\max}$ having the same root as $T_{\max}$.

Since the number of these subtrees is finite, it would therefore be possible, at least in principle, to build the sequence of all the best trees with k leaves for $1 \leq k \leq |T_{\max}|$, where $|T|$ denotes the number of leaves of the tree T, and compare them, for example, on a test sample. However, the number of admissible models is exponential in the characteristic sizes of the learning data leading to an explosive algorithmic complexity. Fortunately, an effective alternative allows a sufficient implicit enumeration to achieve an optimal result. The process simply consists in the pruning algorithm, which ensures the extraction of a sequence of nested subtrees (i.e., pruned from each other) $T_1, \dots, T_K$ all pruned from $T_{\max}$, where T_k minimizes a penalized criterion where the penalty term is proportional to the number of leaves of the tree. This sequence is obtained iteratively by cutting branches at each step, which reduces complexity to a reasonable level. In the following few lines, we will limit ourselves to the regression case, the situation being identical in the classification case.

The key idea is to penalize the training error of a subtree T pruned from $T_{\max}$

$$\overline{\text{err}}(T) = \frac{1}{n} \sum_{\{t \text{ leaf of } T\}} \sum_{(x_i, y_i) \in t} (y_i - \overline{y}_t)^2 \tag{2.1}$$

by a linear function of the number of leaves $|T|$ leading to the following penalized least squares criterion:

$$\text{crit}_\alpha(T) = \overline{\text{err}}(T) + \alpha |T| \ .$$

Thus, $\overline{\text{err}}(T)$ measures the fit of the model T to the data and decreases when the number of leaves increases while $|T|$ quantifies the complexity of the model T. The parameter α, positive, tunes the intensity of the penalty: the larger the coefficient α, the more penalized are the complex models, i.e., with many leaves.

The pruning algorithm is summarized in Table 2.1, with the following conventions. For any internal node (i.e., a node that is not a leaf) t of a tree T, we note T_t the branch of T resulting from the node t, i.e., all descendants of the node t. The error of the node t is given by $\overline{\text{err}}(t) = n^{-1} \sum_{\{x_i \in t\}} (y_i - \bar{y}_t)^2$ and the error of the tree T_t $\overline{\text{err}}(T_t)$ is defined by Eq. 2.1.

The central result of the book of Breiman et al. (1984) states that the strictly increasing sequence of parameters $(0 = \alpha_1, \dots, \alpha_K)$ and the associated with the sequence $T_1 \succ \cdots \succ T_K$ made up of nested models (in the sense of pruning) are such for all $1 \leq k \leq K$:

$$\begin{aligned} \forall \alpha \in [\alpha_k, \alpha_{k+1}[\quad T_k &= \underset{\{T \text{ subtree of } T_{\max}\}}{\text{argmin}} \ \text{crit}_\alpha(T) \\ &= \underset{\{T \text{ subtree of } T_{\max}\}}{\text{argmin}} \ \text{crit}_{\alpha_k}(T) \end{aligned}$$

by setting $\alpha_{K+1} = \infty$.

Table 2.1 CART pruning algorithm

Input	Maximal tree $T_{\max}$
Initialization	$\alpha_1 = 0$, $T_1 = T_{\alpha_1} = \operatorname{argmin}_{T \text{ pruned from } T_{\max}} \overline{\text{err}}(T)$.
	`initialize` $T = T_1$ and $k = 1$
Iteration	`While` $\|T\| > 1$,
	`Calculate`
	$\alpha_{k+1} = \min_{\{t \text{ internal node of } T\}} \frac{\overline{\text{err}}(t) - \overline{\text{err}}(T_t)}{\|T_t\| - 1}$.
	`Prune` all T_t branches of T such that
	$\overline{\text{err}}(T_t) + \alpha_{k+1}\|T_t\| = \overline{\text{err}}(t) + \alpha_{k+1}$
	`Take` T_{k+1} the pruned subtree thus obtained
	`Loop on` $T = T_{k+1}$ et $k = k+1$
Output	Trees $T_1 \succ \cdots \succ T_K = \{t_1\}$,
	Parameters $(0 = \alpha_1; \ldots; \alpha_K)$

In other words, the sequence T_1 (which is nothing else than $T_{\max}$), $T_2, \ldots, T_K$ (which is nothing else than the tree reduced to the root) contains all the useful information since for any $\alpha \geqslant 0$, the subtree minimizing crit_α is a subtree of the sequence produced by the pruning algorithm.

This sequence can be visualized by means of the sequence of values $(\alpha_k)_{1 \le k \le K}$ and the generalization errors of the corresponding trees $T_1, \ldots, T_K$. In the graph of Fig. 2.5 obtained on the `spam` data (see p. 20), each point represents a tree: the abscissa is placed according to the value of the corresponding α_k, the ordinate according to the error estimated by cross-validation with the estimation of the standard deviation of the error materialized by a vertical segment.

The choice of the optimal tree can be made directly, by minimizing the error obtained by cross-validation or by applying the "1 standard error rule" ("1-SE rule" in brief). This rule aims at selecting in the sequence a more compact tree reaching statistically the same error. It consists in choosing the most compact tree reaching an error lower than the value of the previous minimum augmented by the estimated standard error of this error. This quantity is represented by the horizontal dotted line on the example of Fig. 2.5.

Remark 2.2 The cross-validation procedure (V-fold cross-validation), executed by default in the **rpart** package is as follows. First, starting from $\mathcal{L}_n$ and applying the pruning algorithm, we obtain the sequences $(T_k)_{1 \le k \le K}$ and $(\alpha_k)_{1 \le k \le K}$. Then, the learning sample is randomly divided into subsamples (often $V = 10$) so that $\mathcal{L}_n = E_1 \cup E_2 \cup \cdots \cup E_V$. For each $v = 1, \ldots, V$, we build the sequence of subtrees $(T_k^v)_{1 \le k \le K_v}$ with $\mathcal{L}_n \setminus E_v$ as learning sample. Then we calculate the validation errors of the sequence of trees built on $\mathcal{L}_n$: $R^{cv}(T_k) = \frac{1}{V} \sum_{v=1}^{V} \sum_{(x_i, y_i) \in E_v} \left(y_i - T_k^v(x_i)\right)^2$, where T_k^v minimizes the penalized criterion $\text{crit}_{\alpha'_k}$, with $\alpha'_k = (\alpha_k \alpha_{k+1})^{1/2}$. We finally choose the model $T_{\hat{k}}$ where $\hat{k} = \text{argmin}_{1 \leq k \leq K} R^{cv}(T_k)$.

Let us mention that the choice of α by a validation sample is not available in the **rpart** package.

Finally, it should be noted that, of course, if a tree in the sequence has k leaves, it is the best tree with k leaves. On the other hand, this sequence does not necessarily contain all the best trees with k leaves for $1 \leq k \leq |T_{\max}|$ but only a part of them. However, the "missing" trees are simply not competitive because they correspond to larger values of the penalized criterion, so it is useless to calculate them. In addition their calculation could be more expensive since they are not, in general, pruned from the trees of the sequence.

As we will see below, random forests are, in most cases, forests of unpruned trees. However, it should be stressed that a CART tree, if it used alone, must be pruned. Otherwise, it would suffer from overfitting by being too adapted to the data in $\mathcal{L}_n$ and exhibit a too large generalization error.

2.4 The rpart Package

The **rpart** package (Therneau and Atkinson 2018) implements the CART method as a whole and is installed by default in R. The `rpart()` function allows to build a tree whose development is controlled by the parameters of the `rpart.control()` function and pruning is achieved through the `prune()` function. Finally, the methods `print()`, `summary()`, `plot()`, and `predict()` allow retrieving and illustrate the results. It should also be noted that **rpart** fully handles missing data, both for prediction (see Sect. 2.5.2) and for learning (see details on the `Ozone` example in Sect. 2.6.1).

Other packages that implement decision trees are used in R, such as

- **tree** (Ripley 2018), quite close to **rpart** but which allows, for example, to trace the partition associated with a tree in small dimension and use a validation sample for pruning.
- **rpart.plot** (Milborrow 2018) which offers advanced graphics functions.
- **party** (Hothorn et al. 2017) which proposes other criteria for optimizing the splitting of a node.

Now let us detail the use of the functions `rpart()` and `prune()` on the spam detection example.

The tree built with the default values of `rpart()` is obtained as follows. Note that only the syntax `formula =, data =` is allowed for this function.

```
> library(rpart)
> treeDef <- rpart(type ~ ., data = spamApp)
> print(treeDef, digits = 2)
```

```
n= 2300

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

 1) root 2300 910 ok (0.606 0.394)
   2) charExclamation< 0.08 1369 230 ok (0.834 0.166)
     4) remove< 0.045 1263 140 ok (0.892 0.108)
       8) money< 0.15 1217 100 ok (0.917 0.083) *
       9) money>=0.15 46  11 spam (0.239 0.761) *
     5) remove>=0.045 106  15 spam (0.142 0.858) *
   3) charExclamation>=0.08 931 250 spam (0.271 0.729)
     6) charDollar< 0.0065 489 240 spam (0.489 0.511)
      12) capitalAve< 2.8 307 100 ok (0.674 0.326)
        24) remove< 0.09 265  60 ok (0.774 0.226)
          48) free< 0.2 223  31 ok (0.861 0.139) *
          49) free>=0.2 42  13 spam (0.310 0.690) *
        25) remove>=0.09 42   2 spam (0.048 0.952) *
      13) capitalAve>=2.8 182  32 spam (0.176 0.824)
        26) hp>=0.1 14   2 ok (0.857 0.143) *
        27) hp< 0.1 168  20 spam (0.119 0.881) *
     7) charDollar>=0.0065 442  13 spam (0.029 0.971) *
```

```
> plot(treeDef)
> text(treeDef, xpd = TRUE)
```

The `print()` method allows to obtain a text representation of the obtained decision tree, and the sequence of methods `plot()` then `text()` give a graphical representation (Fig. 2.4).

Remark 2.3 Caution, contrary to what one might think, the tree obtained with the default values of the package is not an optimal tree in the pruning sense. In fact, it is a tree whose development has been stopped, thanks to the parameters `minsplit` (the minimum number of data in a node necessary for the node to be possibly split) and `cp` (the normalized complexity-penalty parameter), documented in the `rpart.control()` function help page. Thus, as `cp = 0.01` by default, the tree provided corresponds to the one obtained by selecting the one corresponding to $\alpha = 0.01 * \overline{err}(T_n)$ (where T_1 is the root), provided that `minsplit` is not the parameter that stops the tree development. It is therefore not the optimal tree but generally a more compact one.

The maximal tree is then obtained using the following command (using the `set.seed()` function to ensure reproducibility of cross-validation results):

```
> set.seed(601334)
> treeMax <- rpart(type ~ ., data = spamApp, minsplit = 2, cp = 0)
> plot(treeMax)
```

The application of the `plot()` method allows to obtain the skeleton of the maximal tree (Fig. 2.3).

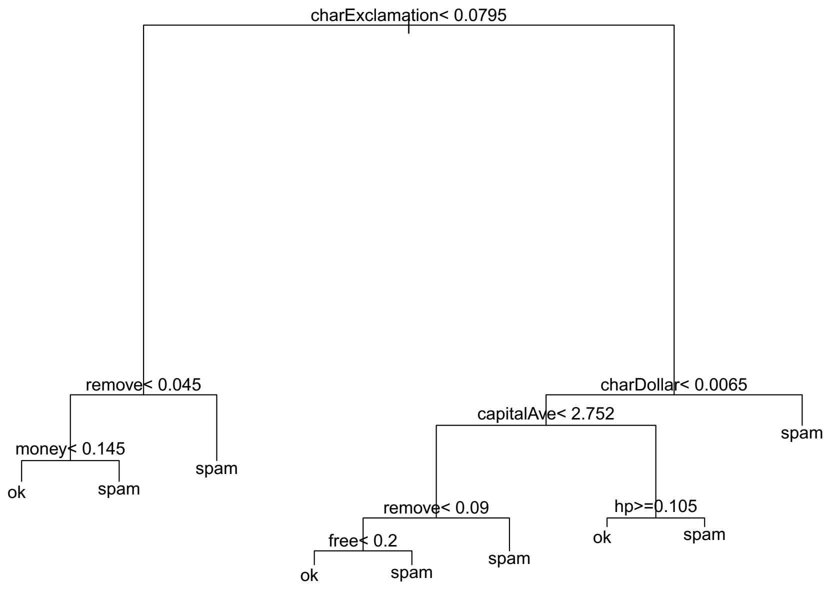

Fig. 2.4 Classification tree obtained with the default values of `rpart()`, `spam` data

Information on the optimal sequence of the pruned subtrees of $T_{\max}$ obtained by applying the pruning algorithm is given by the command:

```
> treeMax$cptable
```

In the columns of Table 2.2, we find the value of the penalty parameter, the number of splits of the corresponding optimal tree, the relative empirical error with respect to the one made by the tree restricted to the root, then the relative cross-validation error, and an estimate of the standard deviation of the associated estimator.

More graphically, we can visualize the sequence of the pruned subtrees of $T_{\max}$ (Fig. 2.5), thanks to the `plotcp()` function:

```
> plotcp(treeMax)
```

Each point thus represents a tree, with the estimation of the standard deviation of the cross-validation error as a vertical segment, quite hard to distinguish on this example (see Fig. 2.11 for a more meaningful graph). The position of the point indicates on the y-axis the (relative) cross-validation error, on the bottom x-axis the value of the penalty parameter, and on the top x-axis the number of leaves of the tree.

The shape of this graph is typical. Let us read it from left to right. When the model is too simple, the bias dominates and the error is significant. Then, it decreases fairly quickly until it reaches a minimum reflecting a good balance between bias and variance and finally rises slightly as the complexity of the model increases.

Table 2.2 Component `cptable` of the object `treeMax`, `spam` data

CP	nsplit	rel error	xerror	xstd
0.4713	0	1.0000	1.0000	0.0259
0.0839	1	0.5287	0.5519	0.0218
0.0591	2	0.4448	0.4570	0.0203
0.0419	4	0.3267	0.3565	0.0184
0.0265	5	0.2848	0.3146	0.0174
0.0177	6	0.2583	0.2947	0.0170
0.0110	7	0.2406	0.2815	0.0166
0.0088	8	0.2296	0.2638	0.0162
0.0055	9	0.2208	0.2517	0.0158
0.0044	14	0.1932	0.2483	0.0157
0.0039	16	0.1843	0.2528	0.0158
0.0033	18	0.1766	0.2373	0.0154
0.0028	24	0.1567	0.2296	0.0152
0.0022	26	0.1512	0.2296	0.0152
0.0015	48	0.1015	0.2307	0.0152
0.0015	53	0.0938	0.2329	0.0153
0.0011	56	0.0894	0.2351	0.0153
0.0009	102	0.0386	0.2439	0.0156
0.0008	110	0.0309	0.2506	0.0158
0.0007	118	0.0243	0.2494	0.0158
0.0006	124	0.0199	0.2517	0.0158
0.0006	131	0.0155	0.2704	0.0163
0.0004	153	0.0033	0.2726	0.0164
0.0000	162	0.0000	0.2759	0.0165

In addition, we find in Fig. 2.6 the same cross-validation error together with the empirical error. This last one decreases until reaching 0 for the maximal tree.

The tree minimizing the cross-validation error is sometimes still a little too complex (23 leaves here).

The optimal pruned tree is plotted in Fig. 2.7 and is obtained by

```
> cpOpt <- treeMax$cptable[which.min(treeMax$cptable[, 4]), 1]
> treeOpt <- prune(treeMax, cp = cpOpt)
> plot(treeOpt)
> text(treeOpt, xpd = TRUE, cex = 0.8)
```

By relaxing a little the condition of minimizing the generalization error by applying the “1-SE rule” of Breiman (which takes into account the uncertainty of the error estimation of the trees in the sequence), we obtain the tree of Fig. 2.8, the “1-SE” pruned tree:

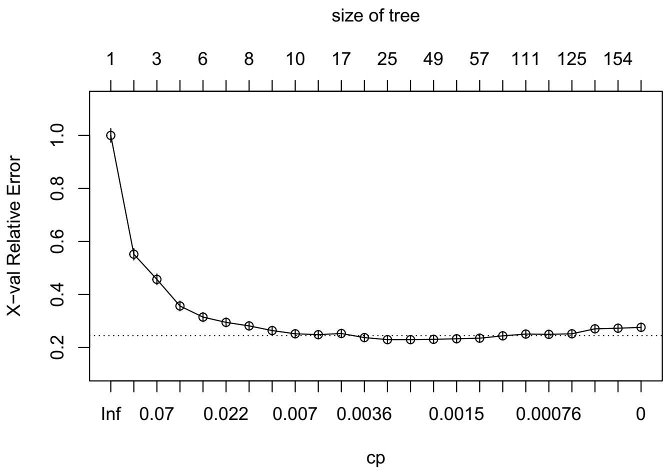

Fig. 2.5 Errors estimated by cross-validation of the sequence of subtrees pruned from the maximal tree, `spam` data

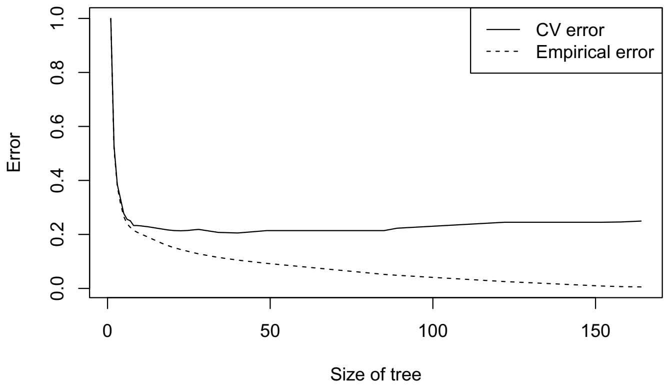

Fig. 2.6 Evolution of the error of pruned trees, learning and test, on the `spam` data

```
> thres1SE <- sum(treeMax$cptable[
    which.min(treeMax$cptable[, 4]), 4:5])
> cp1SE <- treeMax$cptable[
    min(which(treeMax$cptable[, 4] <= thres1SE)), 1]
> tree1SE <- prune(treeMax, cp = cp1SE)
> plot(tree1SE)
> text(tree1SE, xpd = TRUE, cex = 0.8)
```

The best pruned subtree of the maximal tree (up to one standard deviation according to “1-SE” rule) has 19 leaves and only 15 of the 57 initial variables are involved in the splits associated with the 18 internal nodes: `charExclamation`,

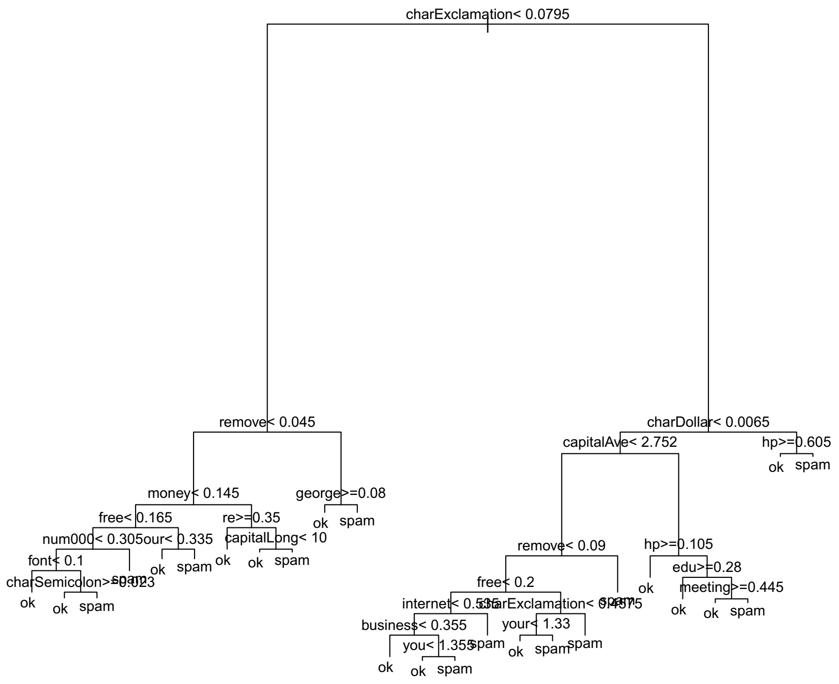

Fig. 2.7 Optimal pruned tree, `spam` data

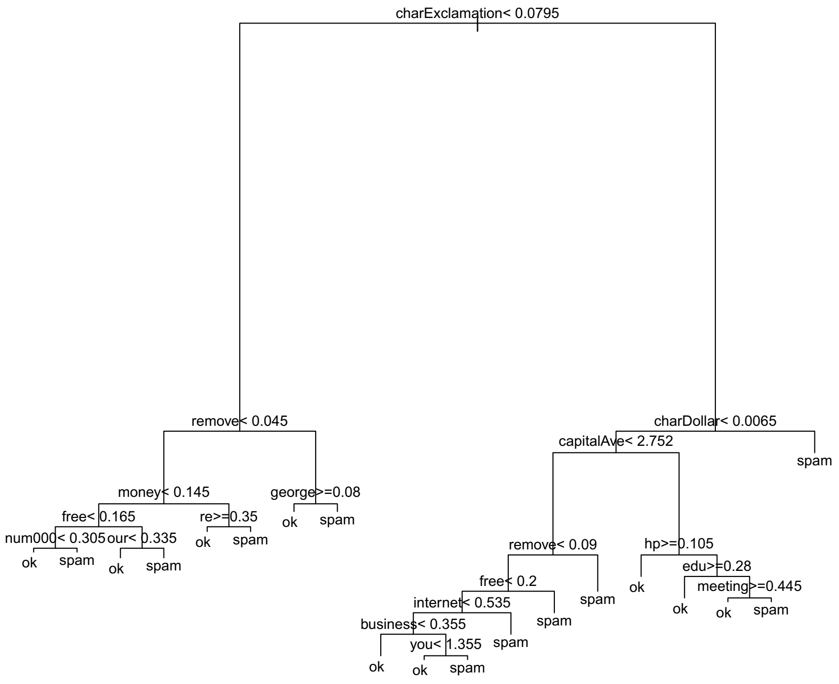

Fig. 2.8 Tree "1-SE" pruned, `spam` data

Table 2.3 Test and empirical errors for the maximal tree, optimal tree, 1-SE tree, and 2-leaf tree (Stump), `spam` data

	Max tree	Optimal tree	1-SE tree	Stump
Test error	0.096	0.086	0.094	0.209
Empirical error	0.000	0.062	0.070	0.208

`charDollar`, `remove`, `capitalAve`, `money`, `george`, `hp`, `free`, `re`, `num000`, `our`, `edu`, `internet`, `business`, and `meeting`.

We can illustrate the ease of interpretation by considering, for example, the path from the root to the rightmost leaf which says that an email containing a lot of "$" and "!" is almost always spam. Conversely, the path from the root to the third most right leaf expresses that an email containing a lot of "!", capital letters, and occurrences of the word `hp` but a few "$" is almost never spam. For this last interpretation, it is worth recalling that the emails examined are the professional emails of a single individual working for HP.

Finally, the empirical and test errors obtained by the different trees are summarized in Table 2.3 and calculated, for example, for the maximal tree, using the `predict()` function (which calculates the predictions for a given tree of a set of observations) by the following commands:

```
> errTestTreeMax <- mean(
    predict(treeMax, spamTest, type = "class") != spamTest$type)
> errEmpTreeMax <- mean(
    predict(treeMax, spamApp, type = "class") != spamApp$type)
```

It should be noted that, as announced, the maximal tree (too complex) has an empirical error (i.e., on the learning sample) of 0 and that the two-leaf tree (too simple) has similar test and empirical errors. The optimal tree has the best test error of 8.6%.

2.5 Competing and Surrogate Splits

2.5.1 Competing Splits

We have at each node of the tree, the sequence of all the splits (one per explanatory variable) ordered by decreasing reduction of the heterogeneity. These are called the competing splits. They are, in all nodes, necessarily calculated during the construction of the maximal tree but only a small number of them are, in general, kept (we refer to the summary of the object `treeStump` on the next page for an illustration on the `spam` data). The possibility of manual development of the maximal tree can be valuable and can be achieved by choosing in each of the nodes in the ordered list of splits, either the optimal one, or a slightly worse split. The actual split variable could be less uncertain, easier, cheaper to measure, or even more interpretable (see, for example, Ghattas 2000).

2.5.2 Surrogate Splits

One of the practical difficulties in calculating a prediction is the presence of missing values. CART offers an effective and very elegant way to circumvent it. First of all, it should be noted that when some input variables are missing for a given x, there is a problem only if the path to calculate the predicted value goes through a node whose split is based on one of these variables. Then, in a node where the split variable is missing, one of the other variables can be used, for example, the second competing split. But this idea is not optimal, since the routing rule in the right and left nodes, respectively, can be very different from the routing rule induced by the optimal split. Hence, the idea is to calculate at each node the list of surrogate splits, defined by the splits minimizing the number of routing errors with respect to the routing rule induced by the optimal split. This provides a method for handling missing values for prediction that is both local and efficient, avoiding to use global and often too coarse imputation methods.

These two aspects are illustrated by the following instructions.

```
> treeStump <- rpart(type ~ ., data = spamApp, maxdepth = 1)
> summary(treeStump)
```

```
Call:
rpart(formula = type ~ ., data = spamApp, maxdepth = 1)
  n= 2300


          CP nsplit rel error    xerror       xstd
1 0.4713024      0 1.0000000 1.0000000 0.02586446
2 0.0100000      1 0.5286976 0.5474614 0.02177043

Variable importance
charExclamation            free            your      charDollar
             44              12              12              12
    capitalLong             all
             11              10

Node number 1: 2300 observations,    complexity param=0.4713024
  predicted class=ok    expected loss=0.393913  P(node) =1
    class counts:  1394   906
   probabilities: 0.606 0.394
  left son=2 (1369 obs) right son=3 (931 obs)
  Primary splits:
      charExclamation < 0.0795 to the left,  improve=351.9304
      charDollar      < 0.0555 to the left,  improve=337.1138
      free            < 0.095  to the left,  improve=296.6714
      remove          < 0.01   to the left,  improve=290.1446
      your            < 0.605  to the left,  improve=272.6889
  Surrogate splits:
      free        < 0.135  to the left,  agree=0.710, adj=0.285
      your        < 0.755  to the left,  agree=0.703, adj=0.267
      charDollar  < 0.0555 to the left,  agree=0.702, adj=0.264
      capitalLong < 53.5   to the left,  agree=0.694, adj=0.245
      all         < 0.325  to the left,  agree=0.685, adj=0.221

Node number 2: 1369 observations
  predicted class=ok    expected loss=0.1658145  P(node) =0.5952174
    class counts:  1142   227
   probabilities: 0.834 0.166

Node number 3: 931 observations
  predicted class=spam  expected loss=0.2706767  P(node) =0.4047826
    class counts:   252   679
   probabilities: 0.271 0.729
```

This tree is the default tree of depth 1 (called *stump*), a typical weak classifier. The result of the method `summary()` provides information not only about the visible parts of the tree (such as structure and splits) but also about the hidden parts, involving variables that do not necessarily appear in the selected tree. Thus, we first find competing splits and then surrogate splits. It should be noted that to mimic the optimal routing rule, the best alternative split is `free < 0.135` which differs from the competing split based on the same variable that is `free < 0.095`.

2.5.3 Interpretability

The interpretability of CART trees is one of the ingredients of their success. It is indeed very easy to answer the question of why, for a given x, a particular value of y is expected. To do this, it suffices to provide the sequence of the answers to the questions constituted by the successive splits encountered to go through the only path from the root to the associated leaf.

But more generally, beyond the interpretation of a particular prediction, once the CART tree has been built, we can consider the variables that intervene in the splits of the nodes of the tree. It is natural to think that the variables involved in splits close to the root are the most important, since they correspond to those whose contribution to the heterogeneity reduction is important. In a complementary way, we would tend to think that the variables that do not appear in any split are not important. Actually, this first intuition gives partial and biased results. Indeed, variables that do not appear in the tree can be important and even useful in this same model to deal with the problem of missing data in prediction, for example. A more sophisticated variable importance index is provided by CART trees. It is based on the concept of surrogate splits. According to Breiman et al. (1984), the importance of a variable can be defined by evaluating, in each node, the heterogeneity reduction generated by the use of the surrogate split for that variable and then summing them over all nodes.

In **rpart**, the importance of the variable X^j is defined by the sum of two terms. The first is the sum of the heterogeneity reductions generated by the splits involving X^j, and the second is weighted the sum of the heterogeneity reductions generated by the surrogate splits when X^j does not define the split. In the second case, the weighting is equal to the relative agreement, in excess of the majority routing rule, given for a node t of size n_t, by

$$\begin{cases} (n_{X^j} - n_{\text{maj}})/(n_t - n_{\text{maj}}) & \text{if } n_{X^j} > n_{\text{maj}} \\ 0 & \text{otherwise} \end{cases}$$

where n_{X^j} and n_{maj} are the numbers of observations well routed with respect to the optimal split of the node t, respectively, by the surrogate split involving X^j and by the split according to the majority rule (which routes all observations to the child node of largest size). This weighting reflects the adjusted relative agreement between the optimal routing rule and the one associated with the surrogate split involving X^j, in excess of the majority rule. The raw relative agreement would be simply given by n_{X^j}/n_t.

However, as interesting as this variable importance index may be, it is no longer so widely used today. It is indeed not very intuitive, it is unstable because it is strongly dependent on a given tree, and it is less relevant than the importance of variables by permutation in the sense of random forests. In addition, its analog, which does not use surrogate splits, exists for random forests but tends to favor nominal variables that have a large number of possible values (we will come back to this in Sect. 4.1).

```
> par(mar = c(7, 3, 1, 1) + 0.1)
> barplot(treeMax$variable.importance, las = 2, cex.names = 0.8)
```

Nevertheless, it can be noted that the variable importance indices, in the sense of CART, for the maximal tree (given by Fig. 2.9) provide for the spam detection example very reasonable and easily interpretable results. It makes it possible to identify a group of variables: `charExclamation` clearly at the top then `capitalLong`, `charDollar`, `free` and `remove` then, less clearly, a group of 8 variables among which `capitalAve`, `capitalTotal`, `money` but also `your` and `num000` intuitively less interesting.

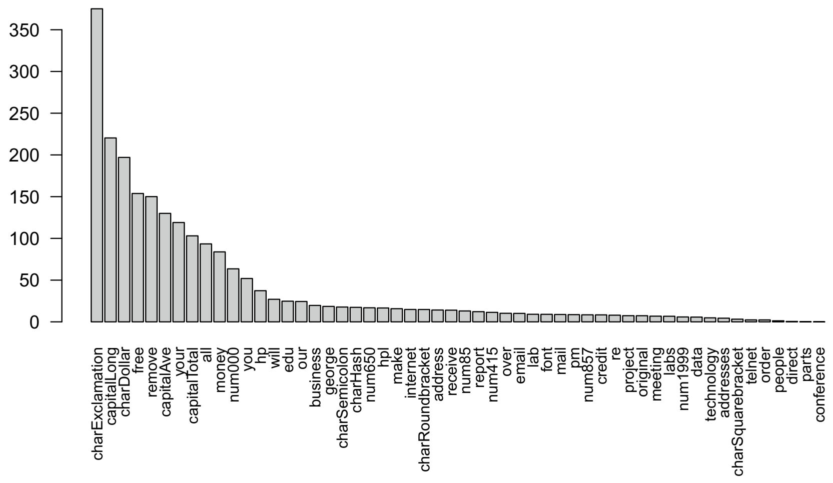

Fig. 2.9 Importance of variables in the sense of CART for the maximal tree, `spam` data

2.6 Examples

2.6.1 Predicting Ozone Concentration

For a presentation of this dataset, see Sect. 1.5.2.

Let us load the **rpart** package and the `Ozone` data:

```
> library("rpart")
> data("Ozone", package = "mlbench")
```

Let us start by building a tree using the default values of `rpart()`:

```
> OzTreeDef <- rpart(V4 ~ ., data = Ozone)
> plot(OzTreeDef)
> text(OzTreeDef, xpd = TRUE, cex = 0.9)
```

Note that the response variable is in column 4 of the data Table and is denoted by `V4`.

Let us analyze this first tree (Fig. 2.10). This dataset has already been considered in many studies and, although these are real data, the results are relatively easy to interpret.

Looking at the first splits in the tree, we notice that the variables `V8`, `V10` then `V1` and `V2` define them. Let us explain why.

Ozone is a secondary pollutant, since it is produced by the chemical transformation of primary precursor gases generated directly from the exhaust pipes (hydrocarbons and nitrogen oxides) in the presence of a catalyst for the chemical reaction: ultraviolet

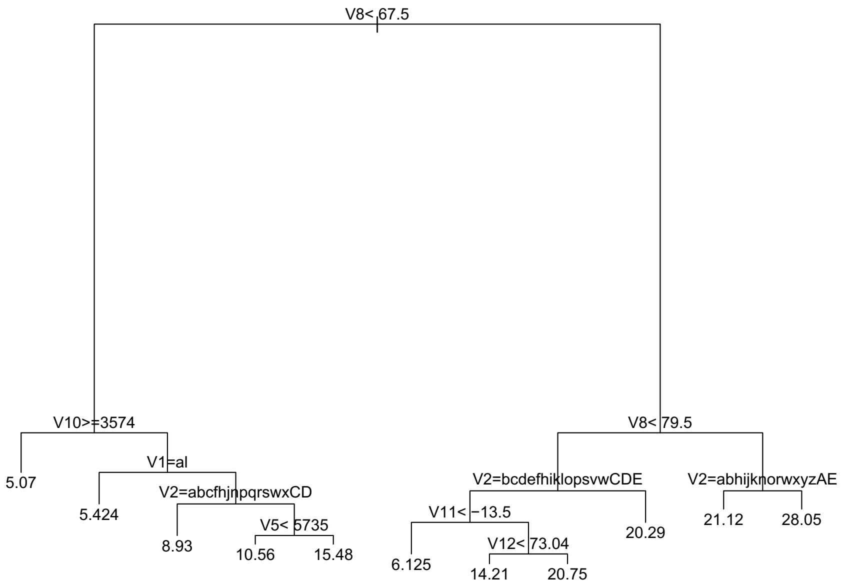

Fig. 2.10 Default tree, `Ozone` data

radiation. The latter is highly correlated with temperature (`V8` or `V9`), which is one of the most important predictors of ozone. As a result, ozone concentrations peak during the summer months and this explains why the month number (`V1`) is among the influential variables. Finally, above an agglomeration, the pollutants are dispersed in a box whose base is the agglomeration and whose height is given by the inversion base height (`V10`).

Continuing to explore the tree, we notice that `V2` defines splits quite close to the root of the tree, despite the fact that `V2` is the number of the day within the month whose relationship with the response variable can only be caused by sampling effects. The explanation is classical in tree-based methods: it comes from the fact that `V2` is a nominal variable with many possible values (here 31) involving a favorable selection bias when choosing the best split.

Predictions can also be easily interpreted: for example, the leftmost leaf gives very low predictions because it corresponds to cold days with a high inversion base height. On the other hand, the rightmost leaf provides much larger predictions because it corresponds to hot days.

Of course, it is now necessary to optimize the choice of the final tree. Let us study for that the sequence of the pruned subtrees (Fig. 2.11) which is the result of the pruning step starting from the the maximal tree.

```
> set.seed(727325)
> OzTreeMax <- rpart(V4 ~ ., data = Ozone, minsplit = 2, cp = 0)
> plotcp(OzTreeMax)
```

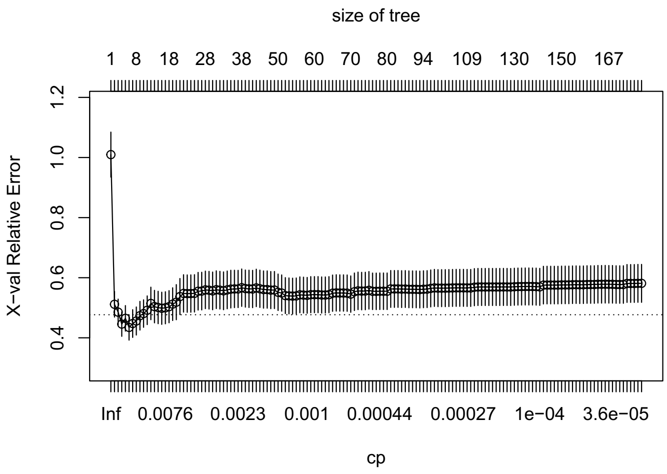

Fig. 2.11 Errors estimated by cross-validation of the sequence of subtrees pruned from the maximal tree, Ozone data

```
> OzIndcpOpt <- which.min(OzTreeMax$cptable[, 4])
> OzcpOpt <- OzTreeMax$cptable[OzIndcpOpt, 1]
> OzTreeOpt <- prune(OzTreeMax, cp = OzcpOpt)
> plot(OzTreeOpt)
> text(OzTreeOpt, xpd = TRUE)
```

The best tree (Fig. 2.12) is particularly compact since it has only six leaves, those involving splits on two of the three major variables highlighted above. We will see that a more complete, but above all, more automatic exploration of this aspect is provided by the permutation variable importance index in the context of random forests. This notion will allow eliminating the variable V2 whose interest is doubtful.

There is a lot of missing data in this dataset and rpart() offers a clever way to handle it during the learning step, without any imputation:

- The data with y missing are eliminated as well as those with all the components of x missing.
- Otherwise, to split a node:
  - Calculate the impurity reductions for each variable using only the associated available data and choose the best split as usual.
  - For this split, observations that have a missing value for the associated split variable are routed, either using a surrogate split or to the most popular node in case the variables determining the best available surrogate splits are all missing for this observation (the maximum number is a parameter, set to 5 by default).
  - Finally, weights in the impurity reduction are updated to take into account the new data routed.

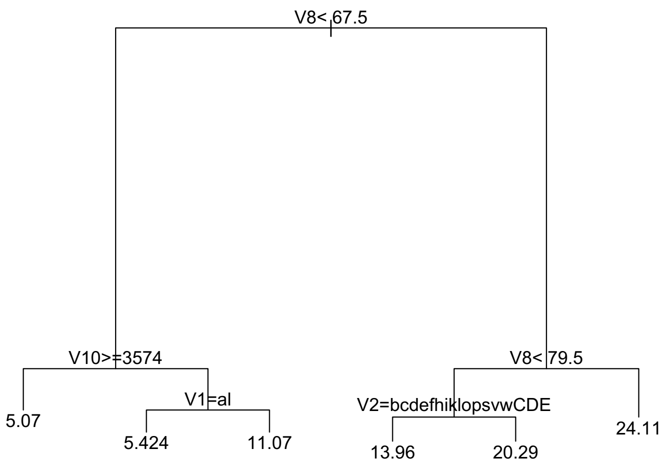

Fig. 2.12 Optimal pruned tree, `Ozone` data

2.6.2 *Analyzing Genomic Data*

For a presentation of this dataset, see Sect. 1.5.3.

Let us load the **rpart** package, the `vac18` data, and group in the same dataframe in which the gene expressions and the stimulation are to be predicted:

```
> library(rpart)
> data("vac18", package = "mixOmics")
> VAC18 <- data.frame(vac18$genes, stimu = vac18$stimulation)
```

The tree obtained with the default values of `rpart()` is obtained as follows (note the use of the argument `use.n = TRUE` in the `text()` function which displays the class distribution for each leaf):

```
> VacTreeDef <- rpart(stimu ~ ., data = VAC18)
> VacTreeDef
```

```
n= 42

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

1) root 42 31 LIPO5 (0.262 0.238 0.238 0.262)
  2) ILMN_2136089>=9.05 11  0 LIPO5 (1.000 0.000 0.000 0.000) *
  3) ILMN_2136089< 9.05 31 20 NS (0.000 0.323 0.323 0.355)
    6) ILMN_2102693< 8.59 18  8 GAG+ (0.000 0.556 0.444 0.000)*
    7) ILMN_2102693>=8.59 13  2 NS (0.000 0.000 0.154 0.846) *
```

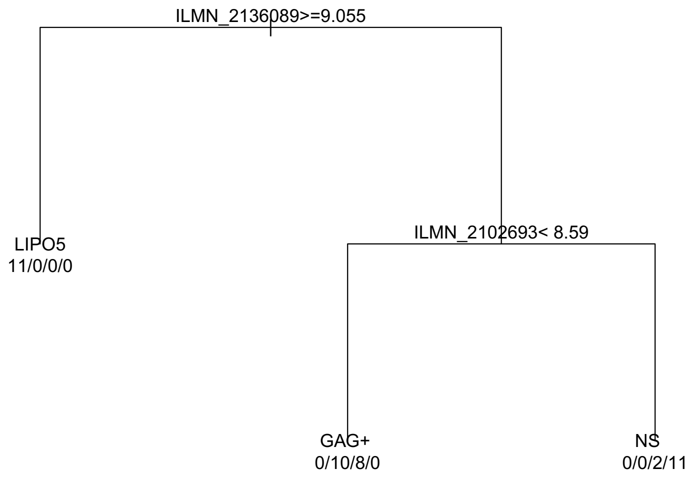

Fig. 2.13 Default tree obtained with `rpart()` on the `Vac18` data

```
> plot(VacTreeDef)
> text(VacTreeDef, use.n = TRUE, xpd = TRUE)
```

The default tree, represented in Fig. 2.13, consists of only 3 leaves. This is due, on the one hand, to the fact that there are only 42 observations in the dataset, and, on the other hand, that the classes `LIPO5` and `NS` are separated from the others very quickly. Indeed, that the first split sends all the observations of class `LIPO5`, and only them, to the left child node.

The maximal tree has 6 leaves (Fig. 2.14). Thus in 5 splits, the classes are perfectly separated. We see on this example that considering only the variables appearing in the splits of a tree (here the deepest tree which can be built using these data) can be very restrictive: indeed, only 5 variables (corresponding to probe identifiers of biochips) among the 1,000 variables appear in the tree.

```
> set.seed(788182)
> VacTreeMax <- rpart(stimu ~ ., data = VAC18, minsplit = 2, cp = 0)
> plot(VacTreeMax)
> text(VacTreeMax, use.n = TRUE, xpd = TRUE)
```

The error estimated by validation for the sequence of pruned subtrees is plotted in the left graph of Fig. 2.15.

Given the small number of individuals, a leave-one-out estimate of the cross-validation error may be preferred. This is obtained by setting the argument `xval`, the number of folds of the cross-validation (argument of the `rpart.control()` function), as follows:

```
> set.seed(413745)
> VacTreeMaxLoo <- rpart(stimu ~ ., data = VAC18, minsplit = 2,
```

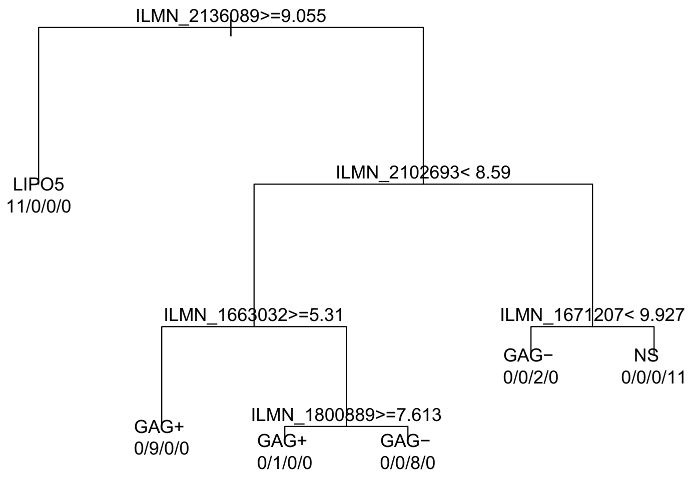

Fig. 2.14 Maximal tree on `Vac18` data

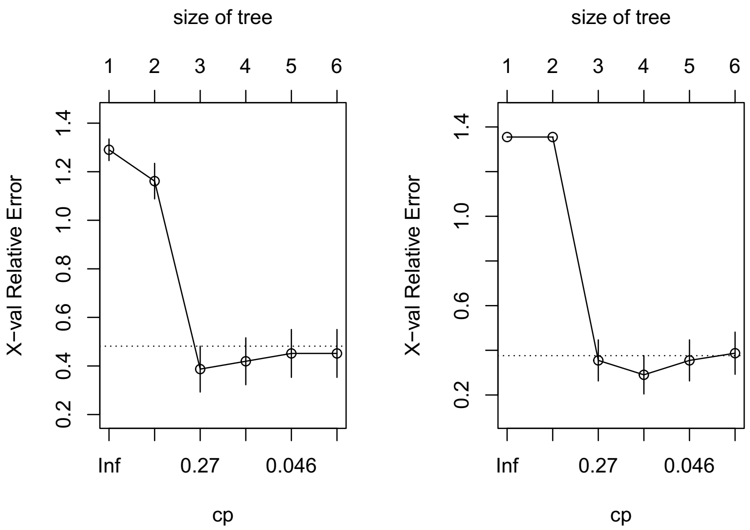

Fig. 2.15 Errors estimated by 10-fold cross-validation (left) and leave-one-out (right) of the sequence of subtrees pruned from the maximal tree, `Vac18` data

```
    cp = 0, xval = nrow(VAC18))
> par(mfrow = c(1, 2))
> plotcp(VacTreeMax)
> plotcp(VacTreeMaxLoo)
```

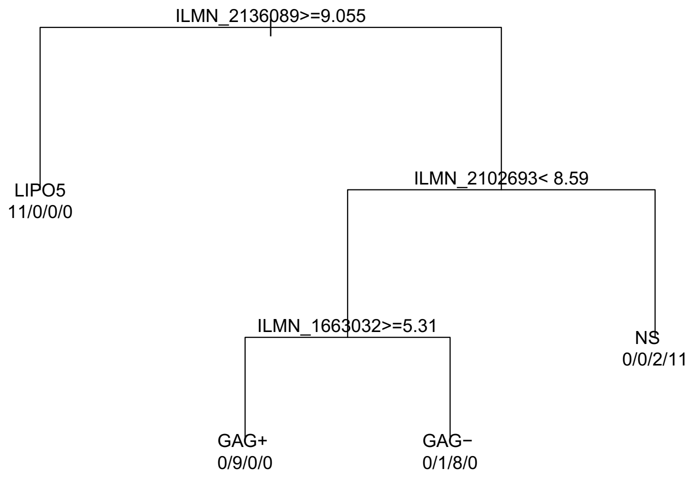

Fig. 2.16 Optimal pruned tree, `Vac18` data

We can easily see on the right side of Fig. 2.15 that for leave-one-out cross-validation, the optimal tree consists of 4 leaves while the 1-SE tree is the same as the default tree (3 leaves). The optimal tree (Fig. 2.16) is obtained using the following commands:

```
> VacIndcpOpt <- which.min(VacTreeMaxLoo$cptable[, 4])
> VaccpOpt <- VacTreeMaxLoo$cptable[VacIndcpOpt, 1]
> VacTreeOpt <- prune(VacTreeMaxLoo, cp = VaccpOpt)
> plot(VacTreeOpt)
> text(VacTreeOpt, use.n = TRUE, xpd = TRUE)
```