the bank (mortgage, securities account, etc.). Table 9.2 shows a sample of the bank's customer database for 20 customers, to illustrate the structure of the data. Among these 5000 customers, only 480 (= 9.6%) accepted the personal loan that was offered to them in the earlier campaign.

Table 9.2 Sample of Data for 20 Customers of Universal Bank

ID	Age	Professional Experience	Income	Family Size	CC Avg	Education	Mortgage	Personal Loan	Securities Account	CD Account	Online Banking	Credit Card
1	25	1	49	4	1.60	UG	0	No	Yes	No	No	No
2	45	19	34	3	1.50	UG	0	No	Yes	No	No	No
3	39	15	11	1	1.00	UG	0	No	No	No	No	No
4	35	9	100	1	2.70	Grad	0	No	No	No	No	No
5	35	8	45	4	1.00	Grad	0	No	No	No	No	Yes
6	37	13	29	4	0.40	Grad	155	No	No	No	Yes	No
7	53	27	72	2	1.50	Grad	0	No	No	No	Yes	No
8	50	24	22	1	0.30	Prof	0	No	No	No	No	Yes
9	35	10	81	3	0.60	Grad	104	No	No	No	Yes	No
10	34	9	180	1	8.90	Prof	0	Yes	No	No	No	No
11	65	39	105	4	2.40	Prof	0	No	No	No	No	No
12	29	5	45	3	0.10	Grad	0	No	No	No	Yes	No
13	48	23	114	2	3.80	Prof	0	No	Yes	No	No	No
14	59	32	40	4	2.50	Grad	0	No	No	No	Yes	No
15	67	41	112	1	2.00	UG	0	No	Yes	No	No	No
16	60	30	22	1	1.50	Prof	0	No	No	No	Yes	Yes
17	38	14	130	4	4.70	Prof	134	Yes	No	No	No	No
18	42	18	81	4	2.40	UG	0	No	No	No	No	No
19	46	21	193	2	8.10	Prof	0	Yes	No	No	No	No
20	55	28	21	1	0.50	Grad	0	No	Yes	No	No	Yes

After randomly partitioning the data into training (3000 records) and validation (2000 records), we use the training data to construct a tree. A tree with 7 splits is shown in Figure 9.9 (this is the default tree produced by *rpart()* for this data). The top node refers to all the records in the training set, of which 2709 customers did not accept the loan and 291 customers accepted the loan. The "0" in the top node's rectangle represents the majority class ("did not accept" = 0). The first split, which is on the income variable, generates left and right child nodes. To the left is the child node with customers who have income less than 114. Customers with income greater than or equal to 114 go to the right. The splitting process continues; where it stops depends on the parameter settings of the algorithm. The eventual classification of customer appears in the terminal nodes. Of the eight terminal nodes, four lead to classification of "did not accept" and four lead to classification of "accept."

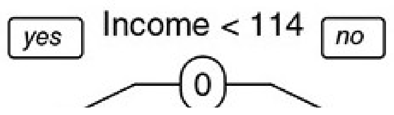
code for creating a default classification tree

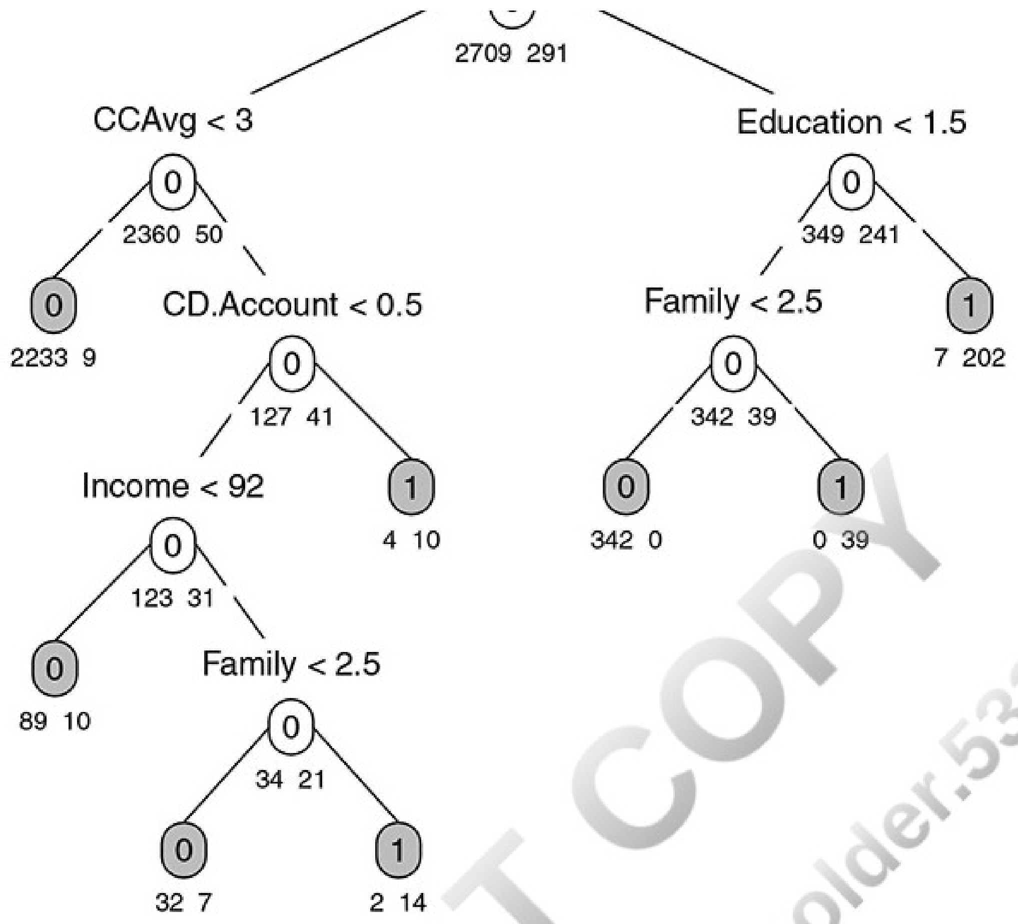
```
library(rpart)
library(rpart.plot)

bank.df <- read.csv("UniversalBank.csv")
bank.df <- bank.df[ , -c(1, 5)]  # Drop ID and zip code columns.

# partition
set.seed(1)
train.index <- sample(c(1:dim(bank.df)[1]), dim(bank.df)[1]*0.6)
train.df <- bank.df[train.index, ]
valid.df <- bank.df[-train.index, ]

# classification tree
default.ct <- rpart(Personal.Loan ~ ., data = train.df, method = "class")
# plot tree
prp(default.ct, type = 1, extra = 1, under = TRUE, split.font = 1, varlen = -10)
```



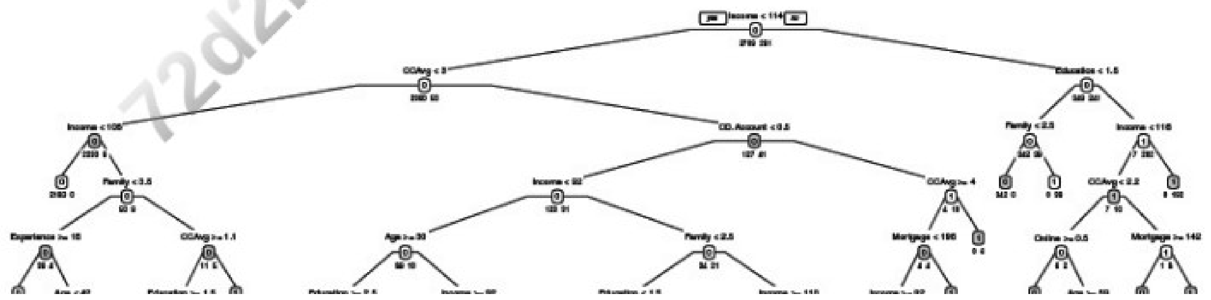


[Figure 9.9](#) Default classification tree for the loan acceptance data using the training set (3000 records)

A full-grown tree is shown in [Figure 9.10](#). Although it is difficult to see the exact splits, let us assess the performance of this tree with the validation data and compare it to the smaller default tree. Each record in the validation data is “dropped down” the tree and classified according to the terminal node it reaches. These predicted classes can then be compared to the actual memberships via a confusion matrix. When a particular class is of interest, a lift chart is useful for assessing the model’s ability to capture those members.

code for creating a deeper classification tree

```
deeper.ct <- rpart(Personal.Loan ~ ., data = train.df, method = "class", cp = 0, minsplit = 1)
# count number of leaves
length(deeper.ct$frame$var[deeper.ct$frame$var == ""])
# plot tree
prp(deeper.ct, type = 1, extra = 1, under = TRUE, split.font = 1, varlen = -10,
    box.col=ifelse(deeper.ct$frame$var == "", 'gray', 'white'))
```



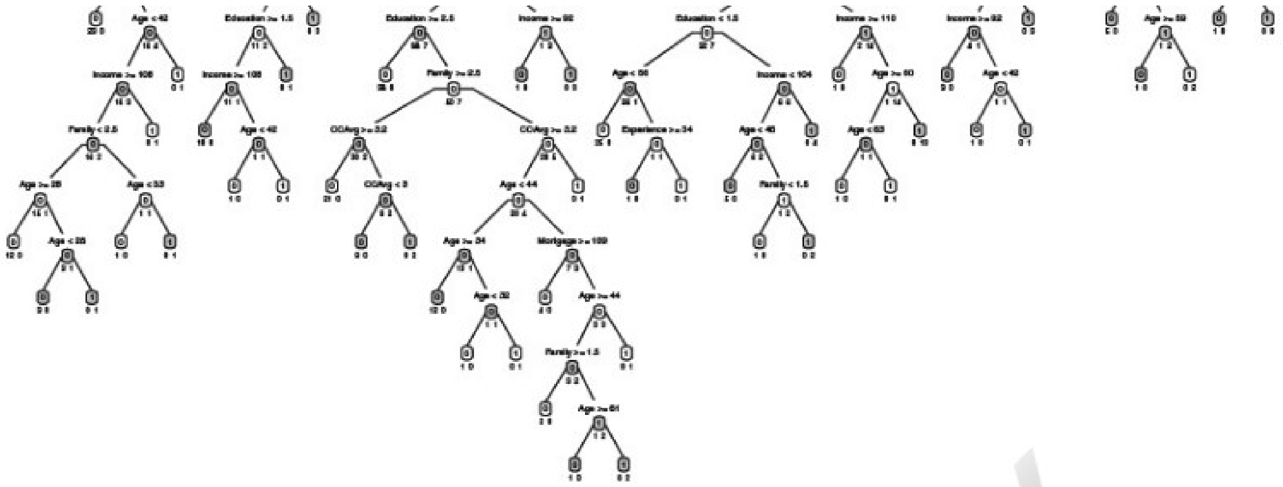


Figure 9.10 A full tree for the loan acceptance data using the training set (3000 records)

Table 9.3 displays the confusion matrices for the training and validation sets of the small default tree (top) and for the full tree (bottom). Comparing the two training matrices, we see that the full tree has higher accuracy: it is 100% accurate in classifying the training data, which means it has completely pure terminal nodes. In contrast, the confusion matrices for the validation data show that the smaller tree is more accurate. The main reason is that the full-grown tree overfits the training data (to perfect accuracy!). This motivates the next section, where we describe ways to avoid overfitting by either stopping the growth of the tree before it is fully grown or by pruning the full-grown tree.

Table 9.3 Confusion matrices and accuracy for the default (small) and deeper (full) classification trees, on the training and validation sets of the personal loan data

code for classifying the validation data using a tree and computing the confusion matrices and accuracy for the training and validation data

```
# classify records in the validation data.
# set argument type = "class" in predict() to generate predicted class membership.
default.ct.point.pred.train <- predict(default.ct,train.df,type = "class")
# generate confusion matrix for training data
confusionMatrix(default.ct.point.pred.train, train.df$PersonalLoan)
### repeat the code for the validation set, and the deeper tree
Output
> # default tree: training
> confusionMatrix(default.ct.point.pred.train, train.df$Personal.Loan)
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 2696   26
         1   13  265

               Accuracy : 0.987

> # default tree: validation
> confusionMatrix(default.ct.point.pred.valid, valid.df$Personal.Loan)
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 1792   18
         1   19  171

               Accuracy : 0.9815

> # deeper tree: training
> confusionMatrix(deeper.ct.point.pred.train, train.df$Personal.Loan)
> confusionMatrix(deeper.ct.point.pred.train, train.df$Personal.Loan)
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 2709    0
         1    0  291

               Accuracy : 1

> # deeper tree: validation
> confusionMatrix(deeper.ct.point.pred.valid, valid.df$Personal.Loan)
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 1788   19
         1   23  170

               Accuracy : 0.979
```

9.4 Avoiding Overfitting

One danger in growing deeper trees on the training data is overfitting. As discussed in Chapter 5, overfitting will lead to poor performance on new data. If we look at the overall error at the various sizes of the tree, it is expected to decrease as the number of terminal nodes grows until the point of overfitting. Of course, for the training data the overall error decreases more and more until it is zero at the maximum level of the tree. However, for new data, the overall error is expected to decrease until the point where the tree fully models the relationship between class and the predictors. After that, the tree starts to model the noise in the training set, and we expect the overall error for the validation set to start increasing. This is depicted in Figure 9.11. One intuitive reason a large tree may overfit is that its final splits are based on very small numbers of records. In such cases, class difference is likely to be attributed to noise rather than predictor information.

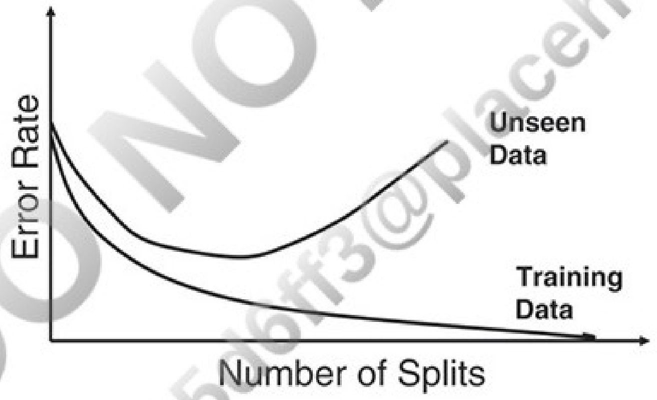


Figure 9.11 Error rate as a function of the number of splits for training vs. validation data: overfitting

Stopping Tree Growth: Conditional Inference Trees

One can think of different criteria for stopping the tree growth before it starts overfitting the data. Examples are tree depth (i.e., number of splits), minimum number of records in a terminal node, and minimum reduction in impurity. In R's *rpart()*, for example, we can control the depth of the tree with the *complexity parameter* (CP). The problem is that it is not simple to determine what is a good stopping point using such rules.

Previous methods developed were based on the idea of recursive partitioning, using rules to prevent the tree from growing excessively and overfitting the training data. One popular method called *CHAID* (chi-squared automatic interaction detection) is a recursive partitioning method that predates classification and regression tree (CART) procedures by several years and is widely used in database marketing applications to this day. It uses a well-known statistical test (the chi-square test for independence) to assess whether splitting a node improves the purity by a statistically significant amount. In particular, at each node, we split on the predictor with the strongest association with the outcome variable. The strength of association is measured by the *p*-value of a chi-squared test of with the outcome variable. The strength of association is measured by the *p*-value of a chi-squared test of independence. If for the best predictor the test does not show a significant improvement, the split is not carried out, and the tree is terminated. This method is more suitable for categorical predictors, but it can be adapted to continuous predictors by binning the continuous values into categorical bins.

A more general class of trees based on this idea is called *conditional inference trees* (see Hothorn et al., 2006). The R implementation is given in the party and partykit packages, and is suitable for both numerical and categorical outcome and predictor variables.

Pruning the Tree

An alternative popular solution that has proven to be more successful than stopping tree growth is pruning the full-grown tree. This is the basis of methods such as CART (developed by Breiman et al., implemented in multiple data mining software packages such as R's rpart package, SAS Enterprise Miner, CART, and MARS) and C4.5 (developed by Quinlan and implemented in packages such as IBM SPSS Modeler). In C4.5, the training data are used both for growing and pruning the tree. In CART, the innovation is to use the validation data to prune back the tree that is grown from training data. CART and CART-like procedures use validation data to prune back the tree that has deliberately been overgrown using the training data.

The idea behind pruning is to recognize that a very large tree is likely to overfit the training data, and that the weakest branches, which hardly reduce the error rate, should be removed. In the mower example, the last few splits resulted in rectangles with very few points (four rectangles in the full tree had a single record). We can see intuitively that these last splits are likely just capturing noise in the training set rather than reflecting patterns that would occur in future data, such as the validation data. Pruning consists of successively selecting a decision node and redesignating it as a terminal node [lopping off the branches extending beyond that decision node (its *subtree*) and thereby reducing the size of the tree]. The pruning process trades off misclassification error in the validation dataset against the number of decision nodes in the pruned tree to arrive at a tree that captures the patterns—but not the noise—in the training data.

Cross-Validation

Pruning the tree with the validation data solves the problem of overfitting, but it does not address the problem of instability. Recall that the CART algorithm may be unstable in choosing one or another variable for the top-level splits, and this effect then cascades down and produces highly variable rule sets. The solution is to avoid relying on just one partition of the data into training and validation. Rather, we do so repeatedly using cross-validation (see below), then pool the results. Of course, just accumulating a set of different trees with their different rules will not do much by itself. However, we can use the results from all those trees to learn how deep to grow the original tree. In this process, we introduce a parameter that can measure, and control, how deep we grow the tree. We will note this parameter value for each minimum-error tree in the cross-validation process, take an average, then apply that average to limit tree growth to this optimal depth when working with new data.

The cost complexity (CC) of a tree is equal to its misclassification error (based on the training data) plus a penalty factor for the size of the tree. For a tree T that has $L(T)$ terminal nodes, the cost complexity can be written as

$$CC(T) = \text{err}(T) + \alpha L(T),$$

where err(T) is the fraction of training records that are misclassified by tree T and α is a penalty factor for tree size. When $\alpha = 0$, there is no penalty for having too many nodes in a tree, and this yields a tree using the cost complexity criterion that is the full-grown unpruned tree. When we increase α to a very large value the penalty cost component swamps the misclassification error component of the cost complexity criterion, and the result is simply the tree with the fewest terminal nodes: namely, the tree with one node. So there is a range of trees, from tiny to large, corresponding to a range of α, from large to small.

Returning to the cross-validation process, we can now associate a value of α with the minimum error tree developed in each iteration of that process.

Here is a simple version of the algorithm:

1. Partition the data into training and validation sets.
2. Grow the tree with the training data.
3. Prune it successively, step by step, recording CP (using the *training* data) at each step.
4. Note the CP that corresponds to the minimum error on the *validation* data.
5. Repartition the data into training and validation, and repeat the growing, pruning and CP recording process.

5. Repartition the data into training and validation, and repeat the growing, pruning and CP recording process.
6. Do this again and again, and average the CP's that reflect minimum error for each tree.
7. Go back to the original data, or future data, and grow a tree, stopping at this optimum CP value.

Typically, cross-validation is done such that the partitions (also called "folds") used for validation are non-overlapping. R automatically does this cross-validation process to select CP and build a default pruned tree, but the user can, instead, specify an alternate value of CP (e.g., if you wanted to see what a deeper tree looked like).

In Table 9.4, we see the complexity-parameter table of cross-validation errors for eight trees of increasing depth grown on the Universal Bank data. We can simply choose the tree with the lowest cross-validation error (*xerror*). In this case, the tree in row 6 has the lowest cross-validation error. Figure 9.12 displays the pruned tree with 15 terminal nodes. This tree was pruned back from the largest tree using the complexity parameter value CP = 0.003436426, which yielded the lowest cross-validation error in Table 9.4.

code for pruning the tree

```
# prune by lower cp
pruned.ct <- prune(cv.ct,
    cp = cv.ct$cptable[which.min(cv.ct$cptable[,"xerror"]),"CP"])
length(pruned.ct$frame$var[pruned.ct$frame$var == ""])
prp(pruned.ct, type = 1, extra = 1, split.font = 1, varlen = -10)
```

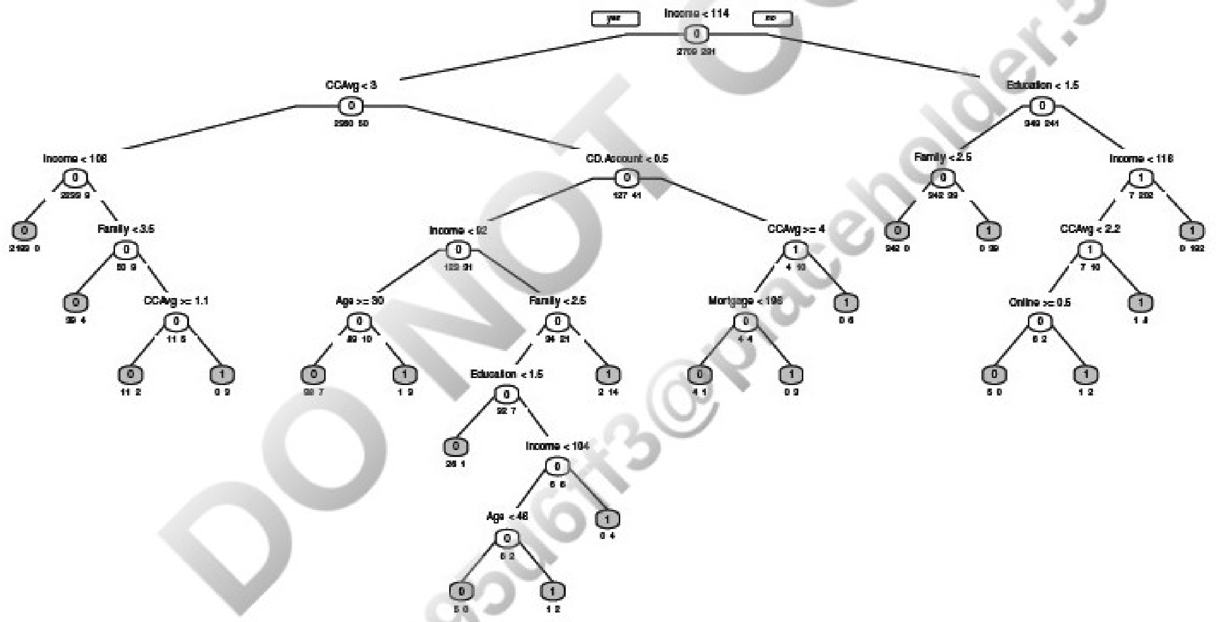


Figure 9.12 Pruned classification tree for the loan acceptance data using CP that yielded lowest xerror in Table 9.4

Table 9.4 Table of complexity parameter (CP) values and associated tree errors

code for tabulating tree error as a function of the complexity parameter (CP)

```
# argument xval refers to the number of folds to use in rpart's built-in
# cross-validation procedure
# argument cp sets the smallest value for the complexity parameter.
cv.ct <- rpart(Personal.Loan ~ ., data = train.df, method = "class",
    cp = 0.00001, minsplit = 5, xval = 5)
# use printcp() to print the table.
printcp(cv.ct)
printcp(cv.ct)

Output
         CP nsplit rel error  xerror     xstd
1 0.3350515      0  1.000000 1.00000 0.055705
2 0.1340206      2  0.329897 0.37457 0.035220
3 0.0154639      3  0.195876 0.19931 0.025917
4 0.0068729      7  0.134021 0.17182 0.024096
5 0.0051546     12  0.099656 0.17182 0.024096
6 0.0034364     14  0.089347 0.16838 0.023858
7 0.0022910     19  0.072165 0.17182 0.024096
8 0.0000100     25  0.058419 0.17182 0.024096
```

Best-Pruned Tree

A further enhancement, in the interest of model parsimony, is to incorporate the sampling error which might cause this minimum to vary if we had a different sample. The enhancement uses the estimated standard error of the cross-validation error (*xstd*) to prune the tree even further; we can add one standard error to the minimum *xerror*. This is sometimes called the *Best-Pruned Tree*. For example, *xstd* for the tree in row 6 is 0.023858. We can choose a smaller tree by going up to the row with a cross-validation error that is larger, but still within one standard error—that is, *xerror* plus *xstd* (0.16838 + 0.023858 = 0.192238). Here, the tree in row 4 is a smaller tree with the lowest *xerror* in this range. The best-pruned tree for the loan acceptance example is shown in Figure 9.13. In this case it coincides with the default tree (Figure 9.9).

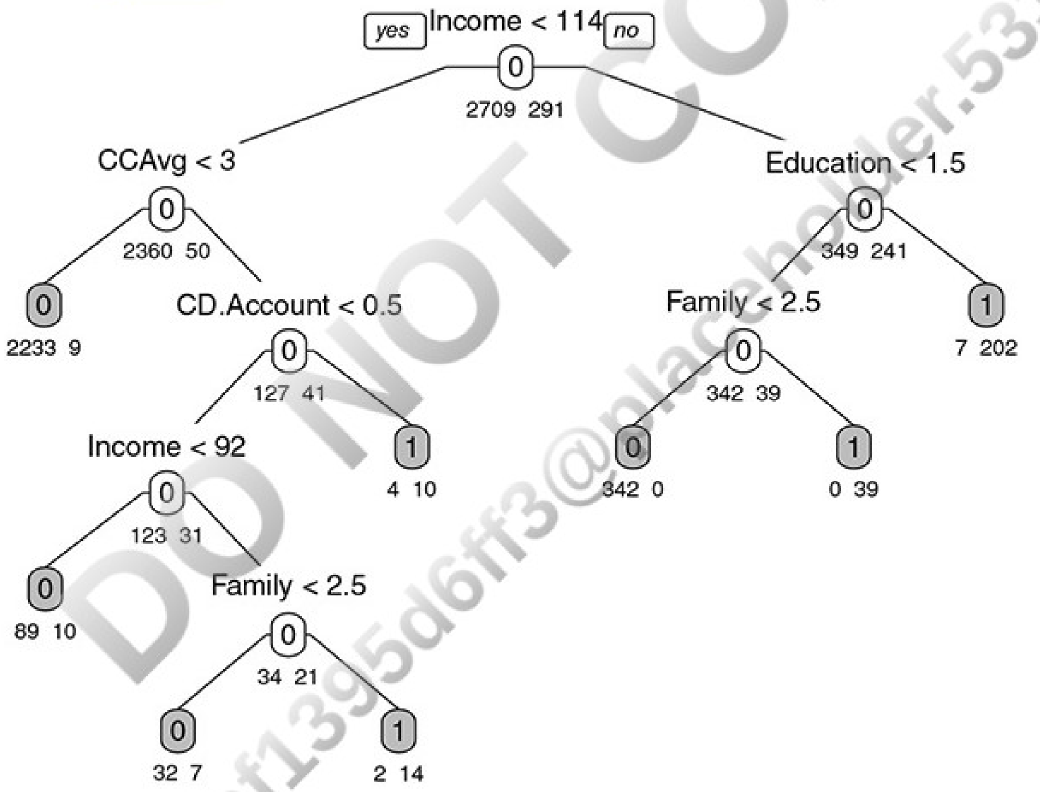


Figure 9.13 Best-pruned tree obtained by fitting a full tree to the training data, pruning it using the cross-validation data, and choosing the smallest tree within one standard error of the minimum xerror tree

9.5 Classification Rules from Trees

As described in Section 9.2, classification trees provide easily understandable *classification rules* (if the trees are not too large). Each terminal node is equivalent to a classification rule. Returning to the example, the right-most terminal node in the best-pruned tree (Figure 9.13) gives us the rule

IF (*Income* ≥ 114) AND (*Education* ≥ 1.5)

THEN *Class* = 1

THEN *Class* = 1.

However, in many cases, the number of rules can be reduced by removing redundancies. For example, consider the rule from the second-from-bottom-left-most terminal node in Figure 9.13:

IF (*Income* < 114) AND (*CCAvg* ≥ 3) AND (*CD.Account* < *0.5*)

AND (*Income* < 92)

THEN *Class* = 0

This rule can be simplified to

IF (*Income* < 92) AND (*CCAvg* ≥ 3) AND (*CD.Account* < *0.5*)

THEN *Class* = 0

This transparency in the process and understandability of the algorithm that leads to classifying a record as belonging to a certain class is very advantageous in settings where the final classification is not the only thing of interest. Berry