CHAPTER 9

Classification and Regression Trees

This chapter describes a flexible data-driven method that can be used for both classification (called *classification tree*) and prediction (called *regression tree*). Among the data-driven methods, trees are the most transparent and easy to interpret. Trees are based on separating records into subgroups by creating splits on predictors. These splits create logical rules that are transparent and easily understandable, for example, "IF Age $<$ 55 AND Education $>$ 12 THEN class = 1." The resulting subgroups should be more homogeneous in terms of the outcome variable, thereby creating useful prediction or classification rules. We discuss the two key ideas underlying trees: *recursive partitioning* (for constructing the tree) and *pruning* (for cutting the tree back). In the context of tree construction, we also describe a few metrics of homogeneity that are popular in tree algorithms, for determining the homogeneity of the resulting subgroups of records. We explain that limiting tree size is a useful strategy for avoiding overfitting and show how it is done. We also describe alternative strategies for avoiding overfitting. As with other data-driven methods, trees require large amounts of data. However, once constructed, they are computationally cheap to deploy even on large samples. They also have other advantages such as being highly automated, robust to outliers, and able to handle missing values. In addition to prediction and classification, we describe how trees can be used for dimension reduction. Finally, we introduce *random forests* and *boosted trees*, which combine results from multiple trees to improve predictive power.

Python

In this chapter, we will use `pandas` for data handling, `scikit-learn` for the models, and `matplotlib` and `pydotplus` for visualization. We will also make

Data Mining for Business Analytics: Concepts, Techniques, and Applications in Python, First Edition.
Galit Shmueli, Peter C. Bruce, Peter Gedeck, and Nitin R. Patel
© 2020 John Wiley & Sons, Inc. Published 2020 by John Wiley & Sons, Inc.

use of the utility functions from the Python Utilities Functions Appendix. Use the following import statements for the Python code in this chapter.

import required functionality for this chapter

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
import matplotlib.pylab as plt
from dmba import plotDecisionTree, classificationSummary, regressionSummary
```

9.1 INTRODUCTION

If one had to choose a classification technique that performs well across a wide range of situations without requiring much effort from the analyst while being readily understandable by the consumer of the analysis, a strong contender would be the tree methodology developed by Breiman et al. (1984). We discuss this classification procedure first, then in later sections we show how the procedure can be extended to prediction of a numerical outcome. The program that Breiman et al. created to implement these procedures was called CART (Classification And Regression Trees). A related procedure is called C4.5.

What is a classification tree? Figure 9.1 shows a tree for classifying bank customers who receive a loan offer as either acceptors or nonacceptors, based on information such as their income, education level, and average credit card expenditure.

Tree Structure

We have two types of nodes in a tree: decision (=splitting) nodes and terminal nodes. Nodes that have successors are called *decision nodes* because if we were to use a tree to classify a new record for which we knew only the values of the predictor variables, we would "drop" the record down the tree so that at each decision node, the appropriate branch is taken until we get to a node that has no successors. Such nodes are called the *terminal nodes* (or *leaves* of the tree), and represent the partitioning of the data by predictors.

It is useful to note that the type of trees grown by Python's *DecisionTreeClassifier()* method, also known as CART or *binary trees*, have the property that the number of terminal nodes is exactly one more than the number of decision nodes.

When using the *export_graphviz()* function from scikit-learn, nodes are represented as boxes. The function has a large number of arguments that allow

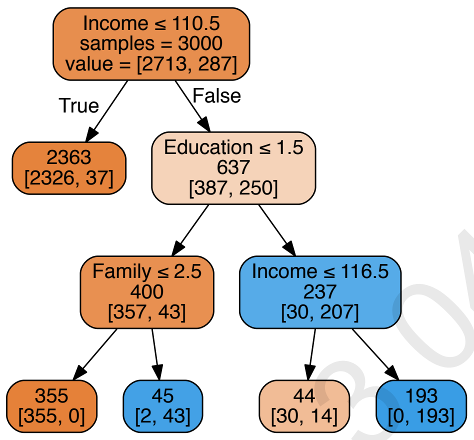

FIGURE 9.1 EXAMPLE OF A TREE FOR CLASSIFYING BANK CUSTOMERS AS LOAN ACCEPTORS OR NONACCEPTORS

controlling the final graph. We use the utility function *plotDecisionTree* from the appendix for plotting the graphs in this chapter. With the chosen settings, all nodes contain information about the number of records in that node (samples), the distribution of the classes, and the majority class of that node. In addition, we color the nodes by the average value of the node for regression or purity of node for classification.

For decision nodes, the name of the predictor variable chosen for splitting and its splitting value appear at the top. Of the two child nodes connected below a decision node, the left box is for records that meet the splitting condition ("True"), while the right box is for records that do not meet it ("False").

Decision Rules

One of the reasons that tree classifiers are very popular is that they provide easily understandable decision rules (at least if the trees are not too large). Consider the tree in the example. The *terminal nodes* are colored orange or blue corresponding to a nonacceptor (0) or acceptor (1) classification. The condition at the top of each splitting node gives the predictor and its splitting value for the split (e.g. *Income* $\leq$110.5 in the top node). *samples*= shows the number of records in that node, and *values*= shows the counts of the two classes (0 and 1) in that node; the labels are only shown in the top node. This tree can easily be translated into a set of rules for classifying a bank customer. For example, the bottom-left node under the "Family" decision node in this tree gives us the following rule:

IF(*Income* > 110.5) AND (*Education* ≤ 1.5) AND (*Family* ≤ 2.5)
THEN *Class* = 0 (nonacceptor).

Classifying a New Record

To classify a new record, it is "dropped" down the tree. When it has dropped all the way down to a terminal node, we can assign its class simply by taking a "vote" (or average, if the outcome is numerical) of all the training data that belonged to the terminal node when the tree was grown. The class with the highest vote is assigned to the new record. For instance, a new record reaching the leftmost terminal node in Figure 9.1, which has a majority of records that belong to the 0 class, would be classified as "nonacceptor." Alternatively, we can convert the number of class 0 records in the node to a proportion (propensity) and then compare the proportion to a user-specified cutoff value. In a binary classification situation (typically, with a success class that is relatively rare and of particular interest), we can also establish a lower cutoff to better capture those rare successes (at the cost of lumping in more failures as successes). With a lower cutoff, the votes for the *success* class only need attain that lower cutoff level for the entire terminal node to be classified as a *success*. The cutoff therefore determines the proportion of votes needed for determining the terminal node class. See Chapter 5 for further discussion of the use of a cutoff value in classification, for cases where a single class is of interest.

In the following sections, we show how trees are constructed and evaluated.

9.2 Classification Trees

The key idea underlying tree construction is *recursive partitioning* of the space of the predictor variables. A second important issue is avoiding over-fitting. We start by explaining recursive partitioning (Section 9.2) and then describe approaches for evaluating and fine-tuning trees while avoiding overfitting (Section9.3).

Recursive Partitioning

Let us denote the outcome variable by Y and the input (predictor) variables by $X_1, X_2, X_3, \ldots, X_p$. In classification, the outcome variable will be a categorical variable. Recursive partitioning divides up the p-dimensional space of the X predictor variables into nonoverlapping multidimensional rectangles. The predictor variables here are considered to be continuous, binary, or ordinal. This division is accomplished recursively (i.e., operating on the results of prior divisions). First, one of the predictor variables is selected, say X_i, and a value of X_i, say s_i, is chosen to split the p-dimensional space into two parts: one part

TABLE 9.1 LOT SIZE, INCOME, AND OWNERSHIP OF A RIDING MOWER FOR 24 HOUSEHOLDS

Household Number	Income ($000s)	Lot Size (000s ft^2)	Ownership of Riding Mower
1	60.0	18.4	Owner
2	85.5	16.8	Owner
3	64.8	21.6	Owner
4	61.5	20.8	Owner
5	87.0	23.6	Owner
6	110.1	19.2	Owner
7	108.0	17.6	Owner
8	82.8	22.4	Owner
9	69.0	20.0	Owner
10	93.0	20.8	Owner
11	51.0	22.0	Owner
12	81.0	20.0	Owner
13	75.0	19.6	Nonowner
14	52.8	20.8	Nonowner
15	64.8	17.2	Nonowner
16	43.2	20.4	Nonowner
17	84.0	17.6	Nonowner
18	49.2	17.6	Nonowner
19	59.4	16.0	Nonowner
20	66.0	18.4	Nonowner
21	47.4	16.4	Nonowner
22	33.0	18.8	Nonowner
23	51.0	14.0	Nonowner
24	63.0	14.8	Nonowner

that contains all the points with $X_i < s_i$ and the other with all the points with $X_i \geq s_i$. Then, one of these two parts is divided in a similar manner by again choosing a predictor variable (it could be X_i or another variable) and a split value for that variable. This results in three (multidimensional) rectangular regions. This process is continued so that we get smaller and smaller rectangular regions. The idea is to divide the entire X-space up into rectangles such that each rectangle is as homogeneous or "pure" as possible. By *pure*, we mean containing records that belong to just one class. (Of course, this is not always possible, as there may be records that belong to different classes but have exactly the same values for every one of the predictor variables.)

Let us illustrate recursive partitioning with an example.

Example 1: Riding Mowers

We again use the riding-mower example presented in Chapter 3. A riding-mower manufacturer would like to find a way of classifying families in a city into those likely to purchase a riding mower and those not likely to buy one. A pilot random sample of 12 owners and 12 nonowners in the city is undertaken. The data are shown and plotted in Table 9.1 and Figure 9.2.

If we apply the classification tree procedure to these data, the procedure will choose *Income* for the first split with a splitting value of 60. The (X_1, X_2) space

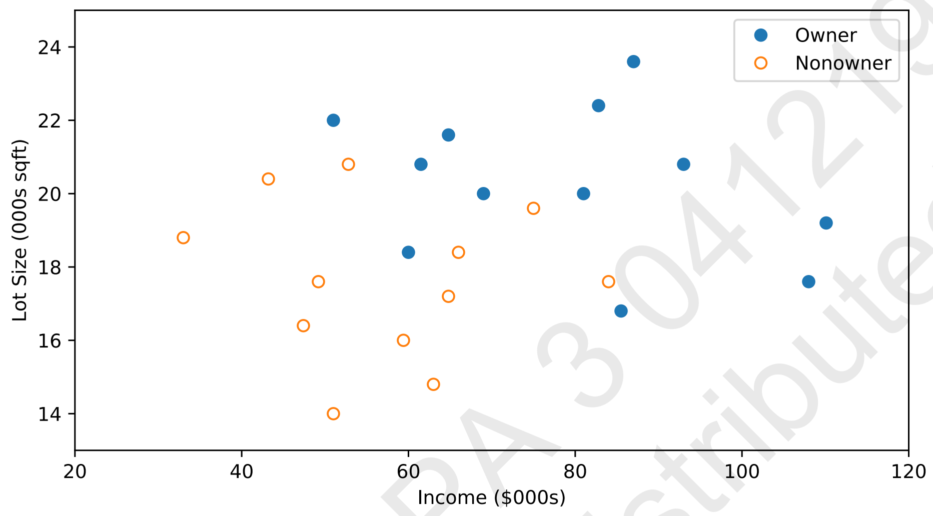

FIGURE 9.2 SCATTER PLOT OF LOT SIZE VS. INCOME FOR 24 OWNERS AND NONOWNERS OF RIDING MOWERS

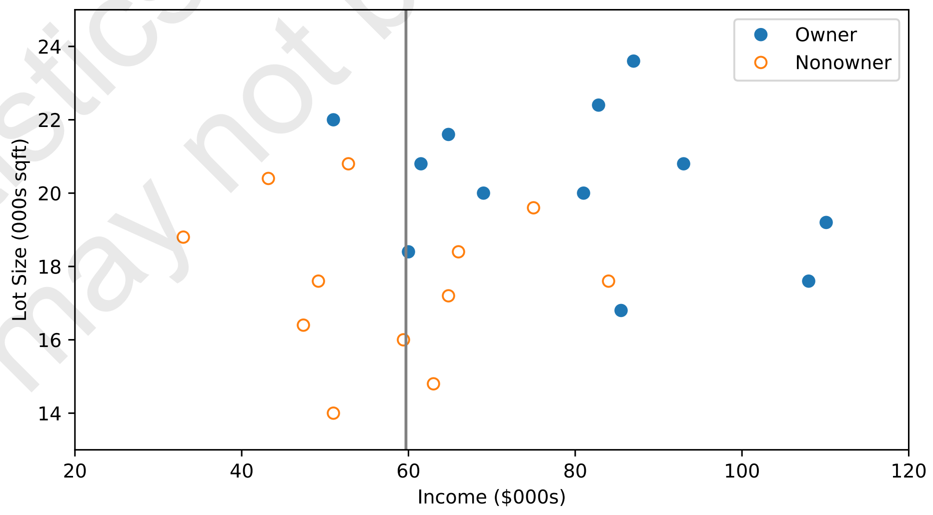

FIGURE 9.3 SPLITTING THE 24 RECORDS BY INCOME VALUE OF 59.7

is now divided into two rectangles, one with Income ≤ 59.7 and the other with Income > 59.7. This is illustrated in Figure 9.3.

Notice how the split has created two rectangles, each of which is much more homogeneous than the rectangle before the split. The left rectangle contains points that are mostly nonowners (seven nonowners and one owner) and the right rectangle contains mostly owners (11 owners and five nonowners).

How was this particular split selected? The algorithm examined each predictor variable (in this case, Income and Lot Size) and all possible split values for each variable to find the best split. What are the possible split values for a variable? They are simply the midpoints between pairs of consecutive values for the predictor. The possible split points for Income are $\{38.1, 45.3, 50.1, \ldots, 109.5\}$ and those for Lot Size are $\{14.4, 15.4, 16.2, \ldots, 23\}$. These split points are ranked according to how much they reduce impurity (heterogeneity) in the resulting rectangle. A pure rectangle is one that is composed of a single class (e.g., owners). The reduction in impurity is defined as overall impurity before the split minus the sum of the impurities for the two rectangles that result from a split.

Categorical Predictors The previous description used numerical predictors; however, categorical predictors can also be used in the recursive partitioning context. To handle categorical predictors, the split choices for a categorical predictor are all ways in which the set of categories can be divided into two subsets. For example, a categorical variable with four categories, say {a, b, c, d}, can be split in seven ways into two subsets: {a} and {b, c, d}; {b} and {a, c, d}; {c} and {a, b, d}; {d} and {a, b, c}; {a, b} and {c, d}; {a, c} and {b, d}; and finally {a, d} and {b, c}. When the number of categories is large, the number of splits becomes very large. As with k-nearest-neighbors, a predictor with m categories ($m > 2$) should be factored into m dummies (not $m - 1$).

Normalization Whether predictors are numerical or categorical, it does not make any difference if they are standardized (normalized) or not.

Measures of Impurity

There are a number of ways to measure impurity. The two most popular measures are the *Gini index* and an *entropy measure*. We describe both next. Denote the m classes of the response variable by $k = 1, 2, \ldots, m$.

The Gini impurity index for a rectangle A is defined by

$$I(A) = 1 - \sum_{k=1}^{m} p_k^2,$$

where p_k is the proportion of records in rectangle A that belong to class k. This measure takes values between 0 (when all the records belong to the same class) and $(m-1)/m$ (when all m classes are equally represented). Figure 9.4 shows the values of the Gini index for a two-class case as a function of p_k. It can be seen that the impurity measure is at its peak when $p_k = 0.5$ (i.e., when the rectangle contains 50% of each of the two classes).

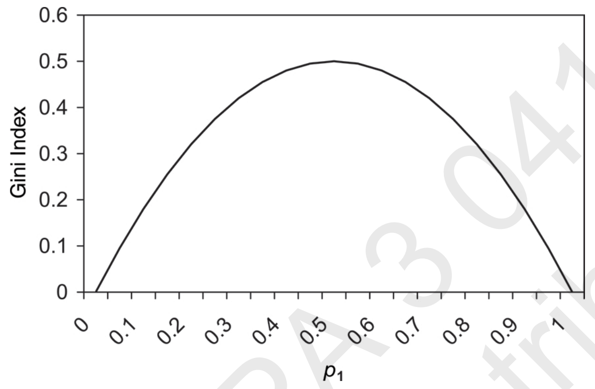

FIGURE 9.4 VALUES OF THE GINI INDEX FOR A TWO-CLASS CASE AS A FUNCTION OF THE PROPORTION OF RECORDS IN CLASS 1 (p_1)

A second impurity measure is the entropy measure. The entropy for a rectangle A is defined by

$$\text{entropy}(A) = -\sum_{k=1}^{m} p_k \log_2(p_k)$$

(to compute $\log_2(x)$ in Python, use function *math.log2(x)*). This measure ranges between 0 (most pure, all records belong to the same class) and $\log_2(m)$ (when all m classes are represented equally). In the two-class case, the entropy measure is maximized (like the Gini index) at $p_k = 0.5$.

Let us compute the impurity in the riding mower example before and after the first split (using Income with the value of 59.7). The unsplit dataset contains 12 owners and 12 nonowners. This is a two-class case with an equal number of records from each class. Both impurity measures are therefore at their maximum value: Gini = 0.5 and entropy = $\log_2(2) = 1$. After the split, the left rectangle contains seven nonowners and one owner. The impurity measures for this rectangle are:

gini_left = $1 - (7/8)^2 - (1/8)^2 = 0.219$
entropy_left = $-(7/8)\log_2(7/8) - (1/8)\log_2(1/8) = 0.544$

The right node contains 11 owners and five nonowners. The impurity measures of the right node are therefore

gini_right = $1 - (11/16)^2 - (5/16)^2 = 0.430$
entropy_right = $-(11/16)\log_2(11/16) - (5/16)\log_2(5/16) = 0.896$

The combined impurity of the two nodes created by the split is a weighted average of the two impurity measures, weighted by the number of records in each:

gini = (8/24)(0.219) + (16/24)(0.430) = 0.359
entropy = (8/24)(0.544) + (16/24)(0.896) = 0.779

Thus, the Gini impurity index decreased from 0.5 before the split to 0.359 after the split. Similarly, the entropy impurity measure decreased from 1 before the split to 0.779 after the split.

By comparing the reduction in impurity across all possible splits in all possible predictors, the next split is chosen. If we continue splitting the mower data, the next split is on the Lot Size variable at the value 21.4. Figure 9.5 shows that once again the tree procedure has astutely chosen to split a rectangle to increase the purity of the resulting rectangles. The lower-left rectangle, which contains records with Income $\leq$ 59.7 and Lot Size $\leq$ 21.4, has all records that are nonowners; whereas the upper-left rectangle, which contains records with Income $\leq$ 59.7 and Lot Size $>$ 21.4, consists exclusively of a single owner. In other words, the two left rectangles are now "pure."

We can see how the recursive partitioning is refining the set of constituent rectangles to become purer as the algorithm proceeds. The final stage of the recursive partitioning is shown in Figure 9.6.

Notice that each rectangle is now pure: it contains data points from just one of the two classes.

The reason the method is called a *classification tree algorithm* is that each split can be depicted as a split of a node into two successor nodes. The first split is shown as a branching of the root node of a tree in Figure 9.7. The full-grown tree is shown in Figure 9.9.

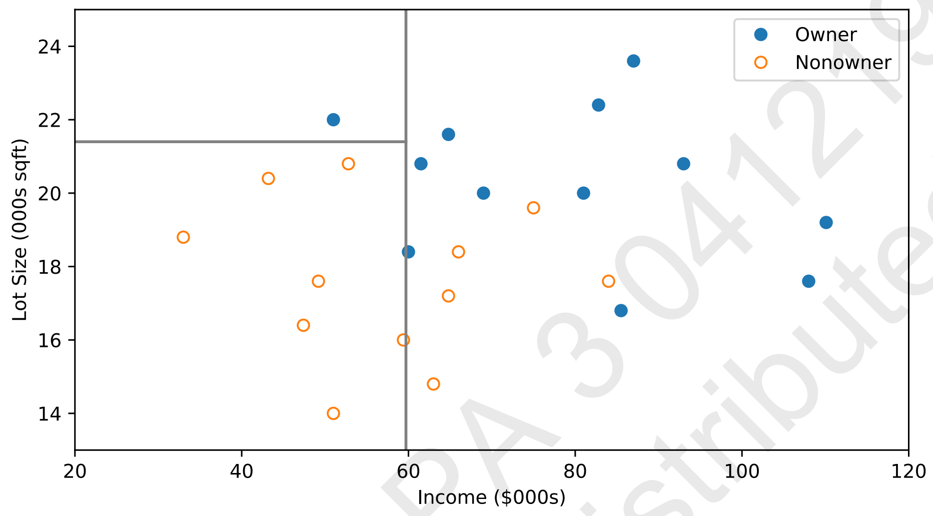

FIGURE 9.5 SPLITTING THE 24 RECORDS FIRST BY INCOME VALUE OF 59.7 AND THEN LOT SIZE VALUE OF 21.4

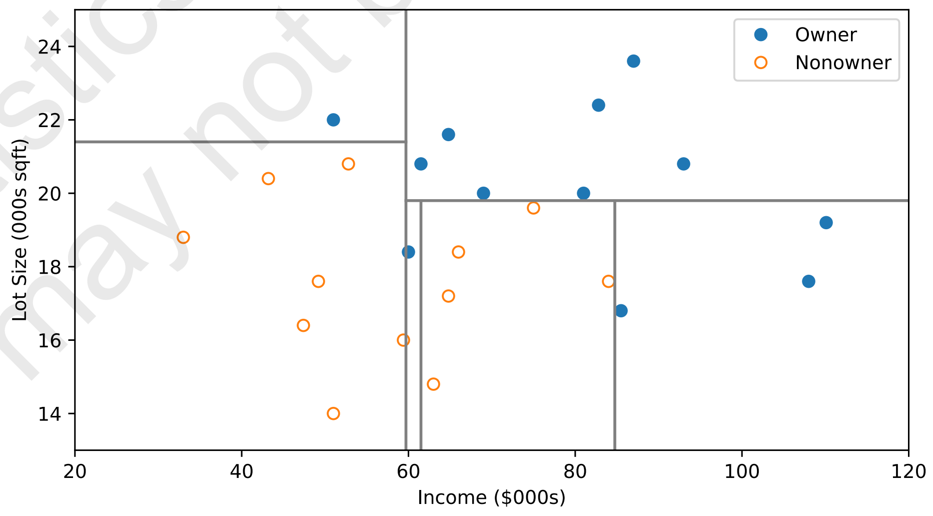

FIGURE 9.6 FINAL STAGE OF RECURSIVE PARTITIONING; EACH RECTANGLE CONSISTING OF A SINGLE CLASS (OWNERS OR NONOWNERS)

code for running and plotting classification tree

```
mower_df = pd.read_csv('RidingMowers.csv')
# use max_depth to control tree size (None = full tree)
classTree = DecisionTreeClassifier(random_state=0, max_depth=1)
classTree.fit(mower_df.drop(columns=['Ownership']), mower_df['Ownership'])
print("Classes: {}".format(', '.join(classTree.classes_)))
plotDecisionTree(classTree, feature_names=mower_df.columns[:2], class_names=classTree.classes_))
```

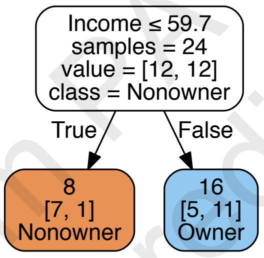

FIGURE 9.7 TREE REPRESENTATION OF FIRST SPLIT (CORRESPONDS TO FIGURE 9.3)

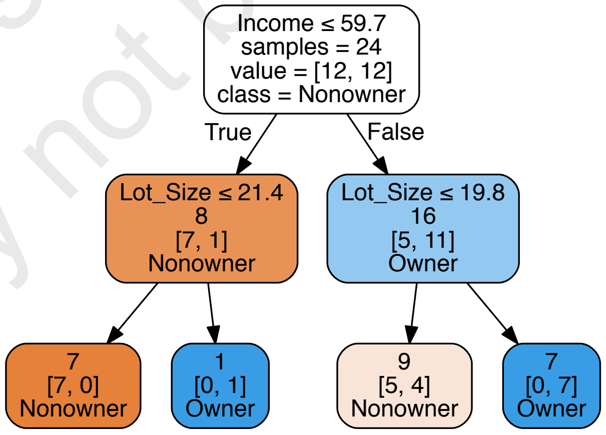

FIGURE 9.8 TREE REPRESENTATION OF FIRST THREE SPLITS.

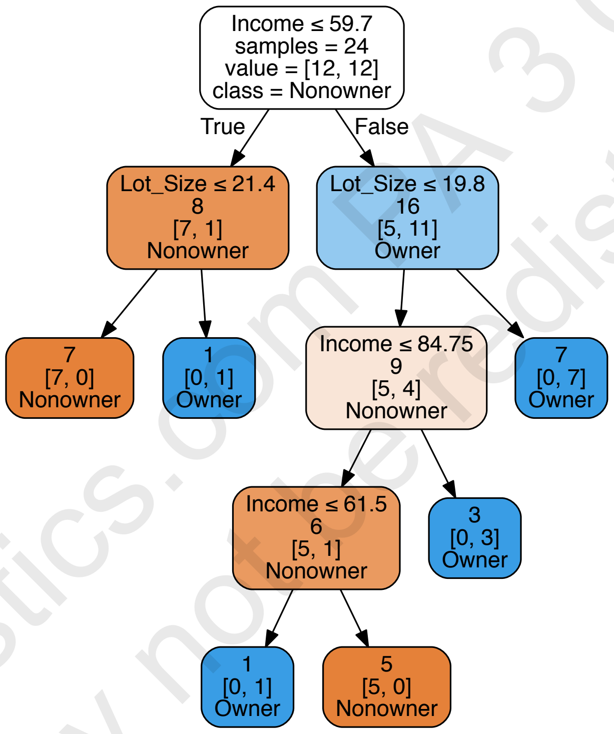

FIGURE 9.9 TREE REPRESENTATION AFTER ALL SPLITS (CORRESPONDS TO FIGURE 9.6). THIS IS THE FULL GROWN TREE

9.3 Evaluating the Performance of a Classification Tree

We have seen with previous methods that the modeling job is not completed by fitting a model to training data; we need out-of-sample data to assess and tune the model. This is particularly true with classification and regression trees, for two reasons:

- Tree structure can be quite unstable, shifting substantially depending on the sample chosen.
- A fully-fit tree will invariably lead to overfitting.

To visualize the first challenge, potential instability, imagine that we partition the data randomly into two samples, A and B, and we build a tree with each. If there are several predictors of roughly equal predictive power, you can see that it would be easy for samples A and B to select different predictors for the top level split, just based on which records ended up in which sample. And a different split at the top level would likely cascade down and yield completely different sets of rules. So we should view the results of a single tree with some caution.

To illustrate the second challenge, overfitting, let's examine another example.

Example 2: Acceptance of Personal Loan

Universal Bank is a relatively young bank that is growing rapidly in terms of overall customer acquisition. The majority of these customers are liability customers with varying sizes of relationship with the bank. The customer base of asset customers is quite small, and the bank is interested in growing this base rapidly to bring in more loan business. In particular, it wants to explore ways of converting its liability (deposit) customers to personal loan customers.

A campaign the bank ran for liability customers showed a healthy conversion rate of over 9% successes. This has encouraged the retail marketing department to devise smarter campaigns with better target marketing. The goal of our analysis is to model the previous campaign's customer behavior to analyze what combination of factors make a customer more likely to accept a personal loan. This will serve as the basis for the design of a new campaign.

Our predictive model will be a classification tree. To assess the accuracy of the tree in classifying new records, we start with the tools and criteria discussed in Chapter 5—partitioning the data into training and validation sets, and later introduce the idea of *cross-validation*.

The bank's dataset includes data on 5000 customers. The data include customer demographic information (age, income, etc.), customer response to the last personal loan campaign (*Personal Loan*), and the customer's relationship with the bank (mortgage, securities account, etc.). Table 9.2 shows a sample of the

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

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.

After randomly partitioning the data into training (3000 records) and validation (2000 records), we use the training data to construct a tree. The result is shown in Figure 9.12 – this is the default tree produced by *DecisionTreeClassifier()* for these data. Although it is difficult to see the exact splits, we note that the top decision node refers to all the records in the training set, of which 2704 customers did not accept the loan and 296 customers accepted the loan. The "class=0" in the top node represents the majority class (nonacceptors). 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 of 110.5 or lower. Customers with income greater than 110.5 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 43 terminal nodes, 24 lead to classification of "nonacceptor" and 19 lead to classification of "acceptor."

The default tree has no restrictions on the maximum depth of the tree (or number of leaf nodes), nor on the magnitude of decrease in impurity measure. These default values for the parameters controlling the size of the tree lead to a fully grown tree, with pure leaf nodes.

Let us assess the performance of this full tree with the validation data. 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 outcome 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 records. Table 9.3 shows the confusion matrices for the full grown tree. We see that the training data are perfectly classified (accuracy=1), whereas the validation data have a lower accuracy (0.98).

Sensitivity Analysis Using Cross Validation

Due to the issue of tree instability, it is possible that the performance results will differ markedly when using a different partitioning into training and validation data. It is therefore useful to use cross-validation to evaluate the variability in performance on different data partitioning (see Section 2.5 in Chapter 2). Table 9.4 shows code and the result of applying Python's *cross_val_score* method to perform 5-fold cross-validation on the full grown (default) tree. We see that the validation accuracy changes significantly across different folds from 0.972 to 0.992.

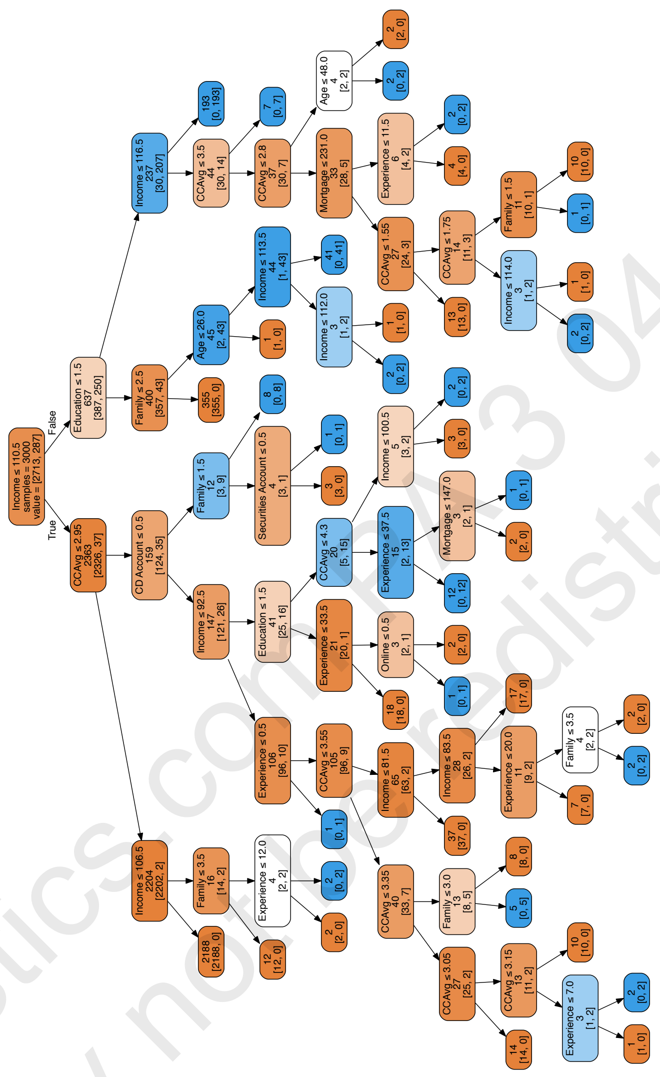

FIGURE 9.10 A FULL TREE FOR THE LOAN ACCEPTANCE DATA USING THE TRAINING SET (3000 RECORDS)

code for creating a full-grown classification tree

```
bank_df = pd.read_csv('UniversalBank.csv')
bank_df.drop(columns=['ID', 'ZIP Code'], inplace=True)

X = bank_df.drop(columns=['Personal Loan'])
y = bank_df['Personal Loan']
train_X, valid_X, train_y, valid_y = train_test_split(X, y, test_size=0.4, random_state=1)

fullClassTree = DecisionTreeClassifier(random_state=1)
fullClassTree.fit(train_X, train_y)

plotDecisionTree(fullClassTree, feature_names=train_X.columns)
```

TABLE 9.3 CONFUSION MATRICES AND ACCURACY FOR THE DEFAULT (FULL) CLASSIFICATION TREE, 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

```
classificationSummary(train_y, fullClassTree.predict(train_X))
classificationSummary(valid_y, fullClassTree.predict(valid_X))
```

Output

```
# full tree: training
Confusion Matrix (Accuracy 1.0000)

       Prediction
Actual    0    1
     0 2727    0
     1    0  273

# full tree: validation
Confusion Matrix (Accuracy 0.9790)

       Prediction
Actual    0    1
     0 1790   17
     1   25  168
```

TABLE 9.4 ACCURACY OF THE DEFAULT (FULL) CLASSIFICATION TREE, ON FIVE VALIDATION FOLDS USING 5-FOLD CROSS-VALIDATION

code for computing validation accuracy using 5-fold cross-validation on the full tree

```
treeClassifier = DecisionTreeClassifier(random_state=1)

scores = cross_val_score(treeClassifier, train_X, train_y, cv=5)
print('Accuracy scores of each fold: ', [f'{acc:.3f}' for acc in scores])
```

Output

```
Accuracy scores of each fold:  ['0.985', '0.972', '0.992', '0.987', '0.992']
```

9.4 Avoiding Overfitting

One danger in growing deep 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.

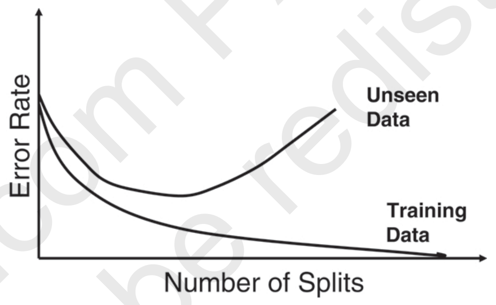

FIGURE 9.11 ERROR RATE AS A FUNCTION OF THE NUMBER OF SPLITS FOR TRAINING VS. VALIDATION DATA: OVERFITTING

Stopping Tree Growth

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 Python's *DecisionTreeClassifier()*, we can control the depth of the tree, the minimum number of records in a node needed in order to split, the mimimum number of records in a terminal node, etc. The problem is that it is not simple to determine what is a good stopping point using such rules.

Let us return to the personal loan example, this time restricting tree depth, the minimum number of records in a node required for splitting, and the minimum impurity decrease required.[1] The code and resulting tree are shown in

[1]The parameter values chosen are the default values used by the R function *rpart()*

code for creating a smaller classification tree

```
smallClassTree = DecisionTreeClassifier(max_depth=30, min_samples_split=20,
                        min_impurity_decrease=0.01, random_state=1)
smallClassTree.fit(train_X, train_y)

plotDecisionTree(smallClassTree, feature_names=train_X.columns)
```

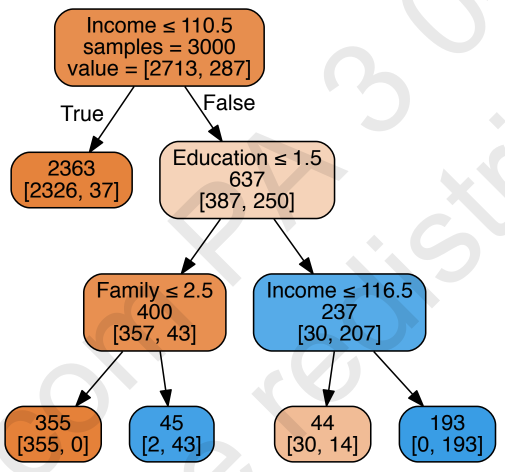

FIGURE 9.12 SMALLER CLASSIFICATION TREE FOR THE LOAN ACCEPTANCE DATA USING THE TRAINING SET (3000 RECORDS)

Figure 9.12. The resulting tree has 4 terminal nodes, where two are pure but two are not. The smallest terminal node has 44 records.

Table 9.5 displays the confusion matrices for the training and validation sets of the smaller tree. Compared to the full grown tree, we see that the full tree has higher training accuracy: it is 100% accurate in classifying the training data, which means it has completely pure terminal nodes. The smaller tree achieves the same validation performance, but importantly, it has a smaller gap in performance between training and validation, whereas the full tree's training-vs-validation performance gap is bigger. The main reason is that the full-grown tree overfits the training data (to perfect accuracy!).

TABLE 9.5 CONFUSION MATRICES AND ACCURACY FOR THE SMALL CLASSIFICATION TREE, ON THE TRAINING AND VALIDATION SETS OF THE PERSONAL LOAN DATA

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

```
classificationSummary(train_y, smallClassTree.predict(train_X))
classificationSummary(valid_y, smallClassTree.predict(valid_X))
```

Output

```
# small tree: training
Confusion Matrix (Accuracy 0.9823)

       Prediction
Actual    0    1
     0 2711    2
     1   51  236

# small tree: validation
Confusion Matrix (Accuracy 0.9770)

       Prediction
Actual    0    1
     0 1804    3
     1   43  150
```

Fine-tuning Tree Parameters

As mentioned earlier, the challenge with using tree growth stopping rules such as maximum tree depth is that it is not simple to determine what is a good stopping point using such rules. A solution is to use grid search over combinations of different parameter values. For example, we might want to search for trees with {maximum depths in the range of [5, 30], minimum number of records in terminal nodes between [20, 100], impurity criterion decrease in the range [0.001, 0.01]}. We can use an exhaustive grid search to find the combination that leads to the tree with smallest error (highest accuracy).

If we use the training set to find the tree with the lowest error among all the trees in the searched range, measuring accuracy on that same training data, then we will be overfitting the training data. If we use the validation set to measure accuracy, then, with the numerous grid search trials, we will be overfitting the validation data! A solution is therefore to use cross-validation on the training set, and, after settling on the best tree, use that tree with the validation data to evaluate likely actual performance with new data. This will help detect and avoid possible overfitting.

In Python, an exhaustive grid search of this type can be achieved using *GridSearchCV()*. Table 9.6 shows the code and result of using this method with

TABLE 9.6 **EXHAUSTIVE GRID SEARCH TO FINE TUNE METHOD PARAMETERS**

code for using GridSearchCV to fine tune method parameters

```
# Start with an initial guess for parameters
param_grid = {
    'max_depth': [10, 20, 30, 40],
    'min_samples_split': [20, 40, 60, 80, 100],
    'min_impurity_decrease': [0, 0.0005, 0.001, 0.005, 0.01],
}
gridSearch = GridSearchCV(DecisionTreeClassifier(random_state=1), param_grid, cv=5,
                          n_jobs=-1)  # n_jobs=-1 will utilize all available CPUs
gridSearch.fit(train_X, train_y)
print('Initial score: ', gridSearch.best_score_)
print('Initial parameters: ', gridSearch.best_params_)

# Adapt grid based on result from initial grid search
param_grid = {
    'max_depth': list(range(2, 16)),  # 14 values
    'min_samples_split': list(range(10, 22)), # 11 values
    'min_impurity_decrease': [0.0009, 0.001, 0.0011], # 3 values
}
gridSearch = GridSearchCV(DecisionTreeClassifier(random_state=1), param_grid, cv=5,
                          n_jobs=-1)
gridSearch.fit(train_X, train_y)
print('Improved score: ', gridSearch.best_score_)
print('Improved parameters: ', gridSearch.best_params_)

bestClassTree = gridSearch.best_estimator_
```

Output

```
Initial score:  0.988
Initial parameters:  {'max_depth': 10, 'min_impurity_decrease': 0.001,
                      'min_samples_split': 20}

Improved score:  0.9883333333333333
Improved parameters:  {'max_depth': 5, 'min_impurity_decrease': 0.0011,
                       'min_samples_split': 13}
```

code for plotting and evaluating performance of fine-tuned classification tree

Evaluating performance

```
# fine-tuned tree: training
> classificationSummary(train_y, bestClassTree.predict(train_X))
Confusion Matrix (Accuracy 0.9900)

       Prediction
Actual    0    1
     0 2703   10
     1   20  267

# fine-tuned tree: validation
> classificationSummary(valid_y, bestClassTree.predict(valid_X))
Confusion Matrix (Accuracy 0.9825)

       Prediction
Actual    0    1
     0 1793   14
     1   21  172
```

Plotting tree

```
plotDecisionTree(bestClassTree, feature_names=train_X.columns)
```

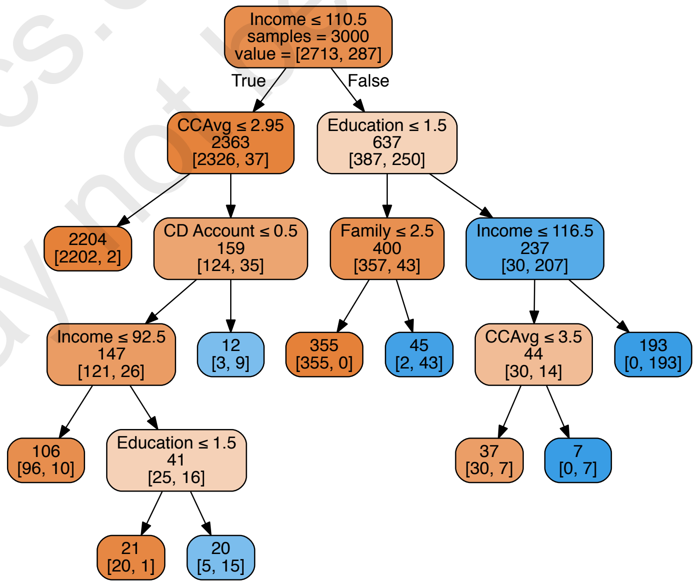

FIGURE 9.13 FINE-TUNED CLASSIFICATION TREE FOR THE LOAN ACCEPTANCE DATA USING THE TRAINING SET (3000 RECORDS)

5-fold cross-validation. The resulting parameter combination is *max_depth* 5, *min_impurity_decrease* 0.0011, and *min_samples_split* 13. The final tree and the model performance on the training and validation sets is shown in Figure 9.13.

The exhaustive grid search can quickly become very time consuming. In our example, the first grid search assessed $4 \times 5 \times 5 = 100$ combinations and the second $14 \times 11 \times 3 = 462$ combinations. With even more tune-able parameters the number of possible combinations can become very large. In this case it is better to use *RandomizedSearchCV()* which will randomly sample all possible combinations while limiting the total number of tries (parameter *n_iter*). With *RandomizedSearchCV()*, it is also possible to sample parameter from distributions (e.g., any number between 0 and 1) without listing them individually.

Other Methods for Limiting Tree Size

Several other methods for limiting tree size have been used widely. They are not implemented in Python at this writing, but we note them here for completeness.

CHAID CHAID stands for chi-squared automatic interaction detection, 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 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. A more general class of trees based on this idea is called *conditional inference trees* (see Hothorn et al., 2006).

Pruning The idea behind pruning is to recognize that a very large tree is likely to overfit the training data, and that the smallest branches, which are the last to be grown and are furthest from the trunk, are likely fitting noise in the training data. They may actually contribute to error in fitting new data, so they are lopped off. Pruning the full-grown tree is the basis of the popular CART method (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.

More specifically, the CART algorithm uses a cost-complexity function that balances tree size (complexity) and misclassification error (cost) in order to choose tree size. The cost complexity 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 ("complexity parameter") for tree size. When $\alpha = 0$, there is no penalty for having too many nodes in a tree, and this yields 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. From this sequence of trees it seems natural to choose the one that gave the lowest misclassification error on the validation dataset. Alternatively, rather than relying on a single validation partition, we can use k-fold cross-validation for choosing α.

9.5 Classification Rules from Trees

As described in Section 9.1, 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 left-most terminal node in the fine-tuned tree (Figure 9.13) gives us the rule

IF (*Income* $\leq$ 110.5) AND (*CCAvg* $\leq$ 2.95)
THEN *Class* = 0.

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* $\leq$ 110.5) AND (*CCAvg* $>$ 2.95) AND (*CD.Account* $\leq$ *0.5*)
AND (*Income* $\leq$ 92.5)
THEN *Class* = 0

This rule can be simplified to

IF (*Income* $\leq$ 92.5) AND (*CCAvg* $>$ 2.95) AND (*CD.Account* $\leq$ *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 and Linoff (2000) give the example of health insurance underwriting, where the insurer is required to show that coverage denial is not based on discrimination. By showing rules that led to denial (e.g., income $<$ \$20K AND low credit history), the company can avoid law suits. Compared to the output of other classifiers, such as discriminant functions, tree-based classification rules are easily explained to managers and operating staff. Their logic is certainly far more transparent than that of weights in neural networks!

9.6 Classification Trees for More Than Two Classes

Classification trees can be used with an outcome variable that has more than two classes. In terms of measuring impurity, the two measures presented earlier (the Gini impurity index and the entropy measure) were defined for m classes and hence can be used for any number of classes. The tree itself would have the same structure, except that its terminal nodes would take one of the m–class labels.

9.7 Regression Trees

The tree method can also be used for a numerical outcome variable. Regression trees for prediction operate in much the same fashion as classification trees. The outcome variable (Y) is a numerical variable in this case, but both the principle and the procedure are the same: Many splits are attempted, and for each, we measure "impurity" in each branch of the resulting tree. The tree procedure then selects the split that minimizes the sum of such measures. To illustrate a regression tree, consider the example of predicting prices of Toyota Corolla automobiles (from Chapter 6). The dataset includes information on 1000 sold Toyota Corolla cars. (We use the first 1000 cars from the dataset *ToyotoCorolla.csv*). The goal is to find a predictive model of price as a function of 10 predictors (including mileage, horsepower, number of doors, etc.). A regression tree for these data was built using a training set of 600 records. The fine-tuned tree is shown in Figure 9.14. Code for training and evaluating the regression tree is given in Table 9.7.

We see that the top three levels of the regression tree are dominated by age of the car, its weight, and mileage. In the lower levels we can see that individual details of a car, like powered windows, lead to smaller adjustments of the price.

Three details differ between regression trees and classification trees: prediction, impurity measures, and evaluating performance. We describe these next.

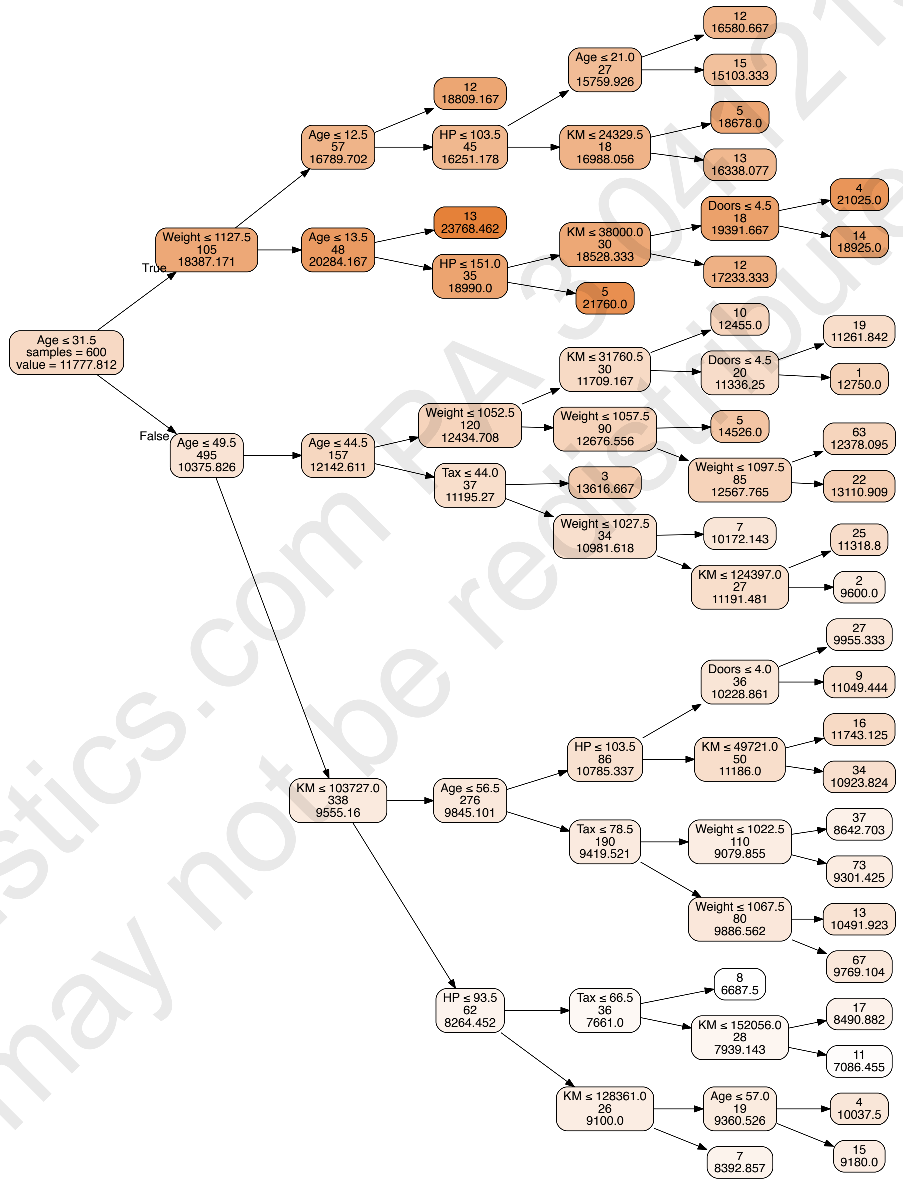

FIGURE 9.14 FINE-TUNED REGRESSION TREE FOR TOYOTA COROLLA PRICES

TABLE 9.7 **TRAIN AND EVALUATE A FINE-TUNED REGRESSION TREE**

code for building a regression tree

```
from sklearn.tree import DecisionTreeRegressor
toyotaCorolla_df = pd.read_csv('ToyotaCorolla.csv').iloc[:1000,:]
toyotaCorolla_df = toyotaCorolla_df.rename(columns={'Age_08_04': 'Age', 'Quarterly_Tax': 'Tax'})

predictors = ['Age', 'KM', 'Fuel_Type', 'HP', 'Met_Color', 'Automatic', 'CC',
              'Doors', 'Tax', 'Weight']
outcome = 'Price'

X = pd.get_dummies(toyotaCorolla_df[predictors], drop_first=True)
y = toyotaCorolla_df[outcome]

train_X, valid_X, train_y, valid_y = train_test_split(X, y, test_size=0.4, random_state=1)

# user grid search to find optimized tree
param_grid = {
    'max_depth': [5, 10, 15, 20, 25],
    'min_impurity_decrease': [0, 0.001, 0.005, 0.01],
    'min_samples_split': [10, 20, 30, 40, 50],
}
gridSearch = GridSearchCV(DecisionTreeRegressor(), param_grid, cv=5, n_jobs=-1)
gridSearch.fit(train_X, train_y)
print('Initial parameters: ', gridSearch.best_params_)

param_grid = {
    'max_depth': [3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
    'min_impurity_decrease': [0, 0.001, 0.002, 0.003, 0.005, 0.006, 0.007, 0.008],
    'min_samples_split': [14, 15, 16, 18, 20, ],
}
gridSearch = GridSearchCV(DecisionTreeRegressor(), param_grid, cv=5, n_jobs=-1)
gridSearch.fit(train_X, train_y)
print('Improved parameters: ', gridSearch.best_params_)

regTree = gridSearch.best_estimator_

regressionSummary(train_y, regTree.predict(train_X))
regressionSummary(valid_y, regTree.predict(valid_X))
```

Output

```
Initial parameters:  {'max_depth': 10,'min_impurity_decrease': 0.001,'min_samples_split': 20}
Improved parameters: {'max_depth': 6,'min_impurity_decrease': 0.01,'min_samples_split': 12}

Regression statistics

                      Mean Error (ME) : 0.0000
       Root Mean Squared Error (RMSE) : 1058.8202
            Mean Absolute Error (MAE) : 767.7203
          Mean Percentage Error (MPE) : -0.8074
Mean Absolute Percentage Error (MAPE) : 6.8325

Regression statistics

                      Mean Error (ME) : 60.5241
       Root Mean Squared Error (RMSE) : 1554.9146
            Mean Absolute Error (MAE) : 1026.3487
          Mean Percentage Error (MPE) : -1.3082
Mean Absolute Percentage Error (MAPE) : 9.2311
```

Prediction

Predicting the outcome value for a record is performed in a fashion similar to the classification case: We will use the truncated tree from Figure 9.14 to demonstrate how it works. The predictor information is used for "dropping" the record down the tree until reaching a terminal node. For instance, to predict the price of a Toyota Corolla with Age = 60, Mileage (KM) = 160,000, and Horse_Power (HP) = 100, we drop it down the tree and reach the node that has the value $8392.857 (the lowest node). This is the price prediction for this car according to the tree. In classification trees, the value of the terminal node (which is one of the categories) is determined by the "voting" of the training records that were in that terminal node. In regression trees, the value of the terminal node is determined by the average outcome value of the training records that were in that terminal node. In the example above, the value $8392.857 is the average price of the 7 cars in the training set that fall in the category of Age $>$ 49.5, KM $>$ 128361, and HP $>$ 93.5.

Measuring Impurity

We described two types of impurity measures for nodes in classification trees: the Gini index and the entropy-based measure. In both cases, the index is a function of the *ratio* between the categories of the records in that node. In regression trees, a typical impurity measure is the sum of the squared deviations from the mean of the terminal node. This is equivalent to the sum of the squared errors, because the mean of the terminal node is exactly the prediction. In the example above, the impurity of the node with the value $8392.857 is computed by subtracting 8392.857 from the price of each of the 7 cars in the training set that fell in that terminal node, then squaring these deviations and summing them up. The lowest impurity possible is zero, when all values in the node are equal.

Evaluating Performance

As stated above, predictions are obtained by averaging the outcome values in the nodes. We therefore have the usual definition of predictions and errors. The predictive performance of regression trees can be measured in the same way that other predictive methods are evaluated (e.g., linear regression), using summary measures such as RMSE.

9.8 Improving Prediction: Random Forests and Boosted Trees

Notwithstanding the transparency advantages of a single tree as described above, in a pure prediction application, where visualizing a set of rules does not matter, better performance is provided by several extensions to trees that combine results from multiple trees. These are examples of *ensembles* (see Chapter 13). One popular multitree approach is *random forests*, introduced by Breiman and Cutler.[2] Random forests are a special case of *bagging*, a method for improving predictive power by combining multiple classifiers or prediction algorithms. See Chapter 13 for further details on bagging.

Random Forests

The basic idea in random forests is to:

1. Draw multiple random samples, with replacement, from the data (this sampling approach is called the *bootstrap*).
2. Using a random subset of predictors at each stage, fit a classification (or regression) tree to each sample (and thus obtain a "forest").
3. Combine the predictions/classifications from the individual trees to obtain improved predictions. Use voting for classification and averaging for prediction.

The code and output in Table 9.8 illustrates applying a random forest in Python to the personal loan example. The validation accuracy of the random forest in this example (0.982) is similar to the single fine-tuned tree, and slightly higher than the single small tree that we fit earlier (0.977 - see Table 9.5).

Unlike a single tree, results from a random forest cannot be displayed in a tree-like diagram, thereby losing the interpretability that a single tree provides. However, random forests can produce "variable importance" scores, which measure the relative contribution of the different predictors. The importance score for a particular predictor is computed by summing up the decrease in the Gini index for that predictor over all the trees in the forest. Figure 9.15 shows the variable importance plot generated from the random forest model for the personal loan example. We see that Income and Education have the highest scores, with CCAvg being third. Importance scores for the other predictors are considerably lower.

[2]For further details on random forests, see https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm.

TABLE 9.8 **RANDOM FOREST (PERSONAL LOAN EXAMPLE)**

code for running a random forest, plotting variable importance plot, and computing accuracy

```
bank_df = pd.read_csv('UniversalBank.csv')
bank_df.drop(columns=['ID', 'ZIP Code'], inplace=True)

X = bank_df.drop(columns=['Personal Loan'])
y = bank_df['Personal Loan']
train_X, valid_X, train_y, valid_y = train_test_split(X, y, test_size=0.4, random_state=1)

rf = RandomForestClassifier(n_estimators=500, random_state=1)
rf.fit(train_X, train_y)

# variable (feature) importance plot
importances = rf.feature_importances_
std = np.std([tree.feature_importances_ for tree in rf.estimators_], axis=0)

df = pd.DataFrame({'feature': train_X.columns, 'importance': importances, 'std': std})
df = df.sort_values('importance')
print(df)

ax = df.plot(kind='barh', xerr='std', x='feature', legend=False)
ax.set_ylabel('')
plt.show()

# confusion matrix for validation set
classificationSummary(valid_y, rf.predict(valid_X))
```

Output

```
              feature  importance       std
7   Securities Account    0.003964  0.004998
9               Online    0.006394  0.005350
10          CreditCard    0.007678  0.007053
6             Mortgage    0.034243  0.023469
1           Experience    0.035539  0.016061
0                  Age    0.036258  0.015858
8           CD Account    0.057917  0.043185
3               Family    0.111375  0.053146
4                CCAvg    0.172105  0.103011
5            Education    0.200772  0.101002
2               Income    0.333756  0.129227

Confusion Matrix (Accuracy 0.9820)

       Prediction
Actual    0    1
     0 1803    4
     1   32  161
```

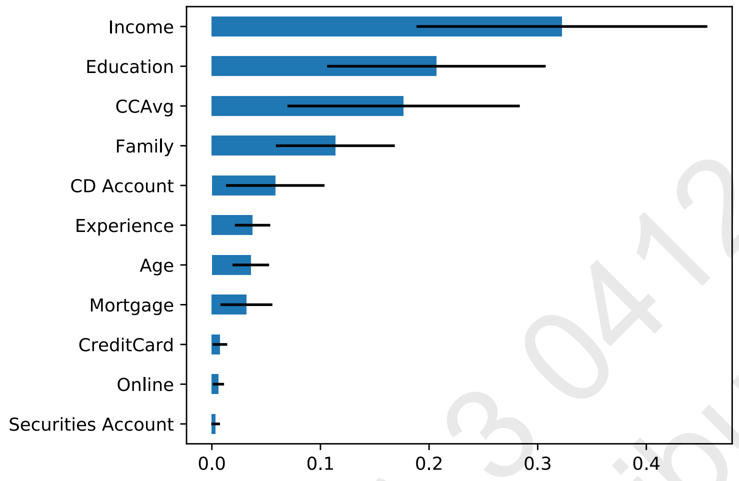

FIGURE 9.15 VARIABLE IMPORTANCE PLOT FROM RANDOM FOREST (PERSONAL LOAN EXAMPLE, FOR CODE SEE TABLE 9.8)

Boosted Trees

The second type of multitree improvement is *boosted trees*. Here a sequence of trees is fitted, so that each tree concentrates on misclassified records from the previous tree.

1. Fit a single tree.
2. Draw a sample that gives higher selection probabilities to misclassified records.
3. Fit a tree to the new sample.
4. Repeat Steps 2 and 3 multiple times.
5. Use weighted voting to classify records, with heavier weight for later trees.

Table 9.9 shows the result of running a boosted tree on the loan acceptance example that we saw earlier. We can see that compared to the performance of the single small tree (Table 9.5), the boosted tree has better performance on the validation data in terms of overall accuracy (0.9835) and especially in terms of correct classification of 1’s—the rare class of special interest. Where does boosting’s special talent for finding 1’s come from? When one class is dominant (0’s constitute over 90% of the data here), basic classifiers are tempted to classify cases as belonging to the dominant class, and the 1’s in this case constitute most of the misclassifications with the single tree. The boosting algorithm concentrates on the misclassifications (which are mostly 1’s), so it is naturally going to do well in reducing the misclassification of 1’s (from 43 in the single small tree to 25 in the boosted tree, in the validation set).

TABLE 9.9 BOOSTED TREE: CONFUSION MATRIX FOR THE VALIDATION SET (LOAN DATA)

code for running boosted trees

```
boost = GradientBoostingClassifier()
boost.fit(train_X, train_y)
classificationSummary(valid_y, boost.predict(valid_X))
```

Output

```
Confusion Matrix (Accuracy 0.9835)

       Prediction
Actual    0    1
     0 1799    8
     1   25  168
```

9.9 Advantages and Weaknesses of a Tree

Tree methods are good off-the-shelf classifiers and predictors. They are also useful for variable selection, with the most important predictors usually showing up at the top of the tree. Trees require relatively little effort from users in the following senses: First, there is no need for transformation of variables (any monotone transformation of the variables will give the same trees). Second, variable subset selection is automatic since it is part of the split selection. In the loan example, note that out of the set of 14 predictors available, the small tree automatically selected only three predictors (Income, Education, and Family) and the fine-tuned tree selected five (Income, Education, Family, CCAvg, and CD Account).

Trees are also intrinsically robust to outliers, since the choice of a split depends on the *ordering* of values and not on the absolute *magnitudes* of these values. However, they are sensitive to changes in the data, and even a slight change can cause very different splits!

Unlike models that assume a particular relationship between the outcome and predictors (e.g., a linear relationship such as in linear regression and linear discriminant analysis), classification and regression trees are nonlinear and nonparametric. This allows for a wide range of relationships between the predictors and the outcome variable. However, this can also be a weakness: Since the splits are done on one predictor at a time, rather than on combinations of predictors, the tree is likely to miss relationships between predictors, in particular linear structures like those in linear or logistic regression models. Classification trees are useful classifiers in cases where horizontal and vertical splitting of the predictor space adequately divides the classes. But consider, for instance, a

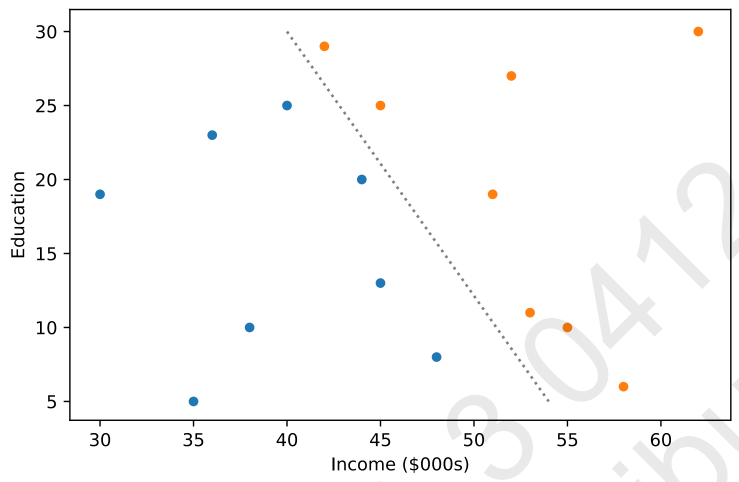

FIGURE 9.16 A TWO-PREDICTOR CASE WITH TWO CLASSES. THE BEST SEPARATION IS ACHIEVED WITH A DIAGONAL LINE, WHICH CLASSIFICATION TREES CANNOT DO

dataset with two predictors and two classes, where separation between the two classes is most obviously achieved by using a diagonal line (as shown in Figure 9.16). In such cases, a classification tree is expected to have lower performance than methods such as discriminant analysis. One way to improve performance is to create new predictors that are derived from existing predictors, which can capture hypothesized relationships between predictors (similar to interactions in regression models). Random forests are another solution in such situations.

Another performance issue with classification trees is that they require a large dataset in order to construct a good classifier. From a computational aspect, trees can be relatively expensive to grow, because of the multiple sorting involved in computing all possible splits on every variable. Methods for avoiding overfitting, such as cross-validation or pruning the data using the validation set, add further computation time.

Although trees are useful for variable selection, one challenge is that they "favor" predictors with many potential split points. This includes categorical predictors with many categories and numerical predictors with many different values. Such predictors have a higher chance of appearing in a tree. One simplistic solution is to combine multiple categories into a smaller set and bin numerical predictors with many values. Alternatively, some special algorithms avoid this problem by using a different splitting criterion [e.g., conditional inference trees in the R package `party`—see Hothorn et al. (2006)—and QUEST classification trees—see Loh and Shih (1997)].

An appealing feature of trees is that they handle missing data without having to impute values or delete records with missing values. Finally, a very important practical advantage of trees is the transparent rules that they generate. Such transparency is often useful in managerial applications, though this advantage is lost in the ensemble versions of trees (random forests, boosted trees).

PROBLEMS

9.1 **Competitive Auctions on eBay.com.** The file *eBayAuctions.csv* contains information on 1972 auctions that transacted on eBay.com during May–June 2004. The goal is to use these data to build a model that will classify auctions as competitive or noncompetitive. A *competitive auction* is defined as an auction with at least two bids placed on the item auctioned. The data include variables that describe the item (auction category), the seller (his/her eBay rating), and the auction terms that the seller selected (auction duration, opening price, currency, day-of-week of auction close). In addition, we have the price at which the auction closed. The task is to predict whether or not the auction will be competitive.

Data Preprocessing. Convert variable *Duration* into a categorical variable. Split the data into training (60%) and validation (40%) datasets.

a. Fit a classification tree using all predictors. To avoid overfitting, set the minimum number of records in a terminal node to 50 and the maximum tree depth to 7. Write down the results in terms of rules. (*Note*: If you had to slightly reduce the number of predictors due to software limitations, or for clarity of presentation, which would be a good variable to choose?)

b. Is this model practical for predicting the outcome of a new auction?

c. Describe the interesting and uninteresting information that these rules provide.

d. Fit another classification tree (using a tree with a minimum number of records per terminal node = 50 and maximum depth = 7), this time only with predictors that can be used for predicting the outcome of a new auction. Describe the resulting tree in terms of rules. Make sure to report the smallest set of rules required for classification.

e. Plot the resulting tree on a scatter plot: Use the two axes for the two best (quantitative) predictors. Each auction will appear as a point, with coordinates corresponding to its values on those two predictors. Use different colors or symbols to separate competitive and noncompetitive auctions. Draw lines (you can sketch these by hand or use Python) at the values that create splits. Does this splitting seem reasonable with respect to the meaning of the two predictors? Does it seem to do a good job of separating the two classes?

f. Examine the lift chart and the confusion matrix for the tree. What can you say about the predictive performance of this model?

g. Based on this last tree, what can you conclude from these data about the chances of an auction obtaining at least two bids and its relationship to the auction settings set by the seller (duration, opening price, ending day, currency)? What would you recommend for a seller as the strategy that will most likely lead to a competitive auction?

9.2 **Predicting Delayed Flights.** The file *FlightDelays.csv* contains information on all commercial flights departing the Washington, DC area and arriving at New York during January 2004. For each flight, there is information on the departure and arrival airports, the distance of the route, the scheduled time and date of the flight, and so on. The variable that we are trying to predict is whether or not a flight is delayed. A delay is defined as an arrival that is at least 15 minutes later than scheduled.

Data Preprocessing. Transform variable day of week (DAY_WEEK) info a categorical variable. Bin the scheduled departure time into eight bins. Use these and all other columns as predictors (excluding DAY_OF_MONTH). Partition the data into training (60%) and validation (40%) sets.

a. Fit a classification tree to the flight delay variable using all the relevant predictors. Do not include DEP_TIME (actual departure time) in the model because it is unknown at the time of prediction (unless we are generating our predictions of delays after the plane takes off, which is unlikely). Use a tree with maximum depth 8 and minimum impurity decrease = 0.01. Express the resulting tree as a set of rules.

b. If you needed to fly between DCA and EWR on a Monday at 7:00 AM, would you be able to use this tree? What other information would you need? Is it available in practice? What information is redundant?

c. Fit the same tree as in (a), this time excluding the Weather predictor. Display both the resulting (small) tree and the full-grown tree. You will find that the small tree contains a single terminal node.

i. How is the small tree used for classification? (What is the rule for classifying?)

ii. To what is this rule equivalent?

iii. Examine the full-grown tree. What are the top three predictors according to this tree?

iv. Why, technically, does the small tree result in a single node?

v. What is the disadvantage of using the top levels of the full-grown tree as opposed to the small tree?

vi. Compare this general result to that from logistic regression in the example in Chapter 10. What are possible reasons for the classification tree's failure to find a good predictive model?

9.3 **Predicting Prices of Used Cars (Regression Trees).** The file *ToyotaCorolla.csv* contains the data on used cars (Toyota Corolla) on sale during late summer of 2004 in the Netherlands. It has 1436 records containing details on 38 attributes, including Price, Age, Kilometers, HP, and other specifications. The goal is to predict the price of a used Toyota Corolla based on its specifications. (The example in Section 9.7 is a subset of this dataset).

Data Preprocessing. Split the data into training (60%), and validation (40%) datasets.

a. Run a full-grown regression tree (RT) with outcome variable Price and predictors Age_08_04, KM, Fuel_Type (first convert to dummies), HP, Automatic, Doors, Quarterly_Tax, Mfr_Guarantee, Guarantee_Period, Airco, Automatic_airco, CD_Player, Powered_Windows, Sport_Model, and Tow_Bar. Set random_state=1.

i. Which appear to be the three or four most important car specifications for predicting the car's price?

ii. Compare the prediction errors of the training and validation sets by examining their RMS error and by plotting the two boxplots. How does the predictive performance of the validation set compare to the training set? Why does this occur?

iii. How might we achieve better validation predictive performance at the expense of training performance?

iv. Create a smaller tree by using GridSearchCV() with cv=5 to find a fine-tuned tree. Compared to the full-grown tree, what is the predictive performance on the validation set?

b. Let us see the effect of turning the price variable into a categorical variable. First, create a new variable that categorizes price into 20 bins. Now repartition the data keeping Binned_Price instead of Price. Run a classification tree with the same set of input variables as in the RT, and with Binned_Price as the output variable. As in the less deep regression tree, create a smaller tree by using GridSearchCV() with cv=5 to find a fine-tuned tree.

i. Compare the smaller tree generated by the CT with the smaller tree generated by RT. Are they different? (Look at structure, the top predictors, size of tree, etc.) Why?

ii. Predict the price, using the smaller RT and CT, of a used Toyota Corolla with the specifications listed in Table 9.10.

TABLE 9.10 SPECIFICATIONS FOR A PARTICULAR TOYOTA COROLLA

Variable	Value
Age_-08_-04	77
KM	117,000
Fuel_Type	Petrol
HP	110
Automatic	No
Doors	5
Quarterly_Tax	100
Mfg_Guarantee	No
Guarantee_Period	3
Airco	Yes
Automatic_airco	No
CD_Player	No
Powered_Windows	No
Sport_Model	No
Tow_Bar	Yes

iii. Compare the predictions in terms of the predictors that were used, the magnitude of the difference between the two predictions, and the advantages and disadvantages of the two methods.