7. Decision or classification trees

Next we are going to consider a rather different approach from those presented so far to machine learning that use one of the most common and important data structures, the binary tree. Generally, the computational cost of making the tree is fairly low, but the cost of using it is even lower, $O(\log m)$, where m is the number of nodes. This is important for machine learning, since querying the trained model should be as fast as possible since it happens more often and the result is often wanted immediately. Trees also have other benefits, such as the fact they are easy to understand.

In terms of optimisation and search, *decision trees use a greedy heuristic to make search*, evaluating the possible options at the current stage of learning and making the one that seems optimal at the point. This works well a large amount of the time.

Let us look at the example of student life in Fig. 7.1. First, one checks whether there is party and decides accordingly. Second, in the case of no party, it is good to check whether some deadline is close. If this is urgent, it is good to study, but otherwise selections are to be lazy or not, i.e. to study, or go to pub.

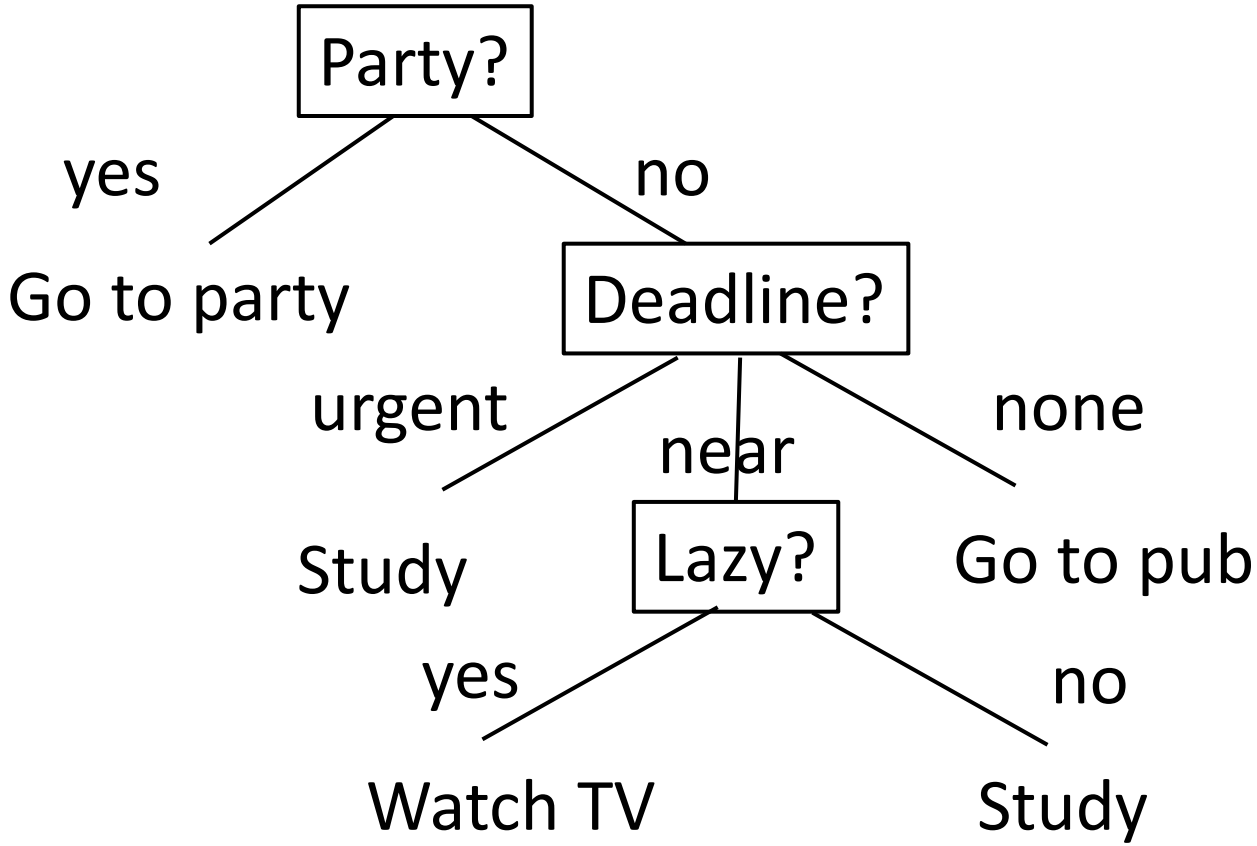


Fig. 7.1 A simple decision tree to decide how a student may be going to spend the evening.

Decision trees can be turned into a set of logical conjunctions, if-then rules, that then go into program code simply:

- **If** *there is a party* **then** *go to it*
- **If** *there is not a party* **and** *one has an urgent deadline* **then** *study*
- etc.

7.1 Construction of decision trees

The basic question is how, based on the variables, we can construct the tree. There are a few different decision tree algorithms, but they are almost all variants of the same principle: the algorithms build the tree in a *greedy* manner starting at the root, choosing the most *informative* variable at each step. Greediness means the specific strategy or principle of the algorithm, in fact, here local optimization to express it precisely. We start with most common, traditional Quinlan's ID3.

To quantify information covered by a variable at each stage, we apply information theoretic concepts. Information theory was begun in 1948 when Claude Shannon published a paper called "A mathematical theory of communication", in which he proposed the information measure of *entropy.*

Entropy of a set of probabilities p_i is

$$\text{Entropy}(p) = -\sum_i p_i \log p_i$$

where the logarithm has base 2, because we are imaging that we encode everything using binary digits, bits, and we define 0 log 0 = 0. A graph of the entropy is presented in Fig. 7.2.

Let us suppose that there are cases of only 2 values. If all of the cases are positive, then we do not get any extra information from knowing the value of the variable for any particular case, since whatever the variable value, the case is positive. Thus, the entropy of that variable is 0.

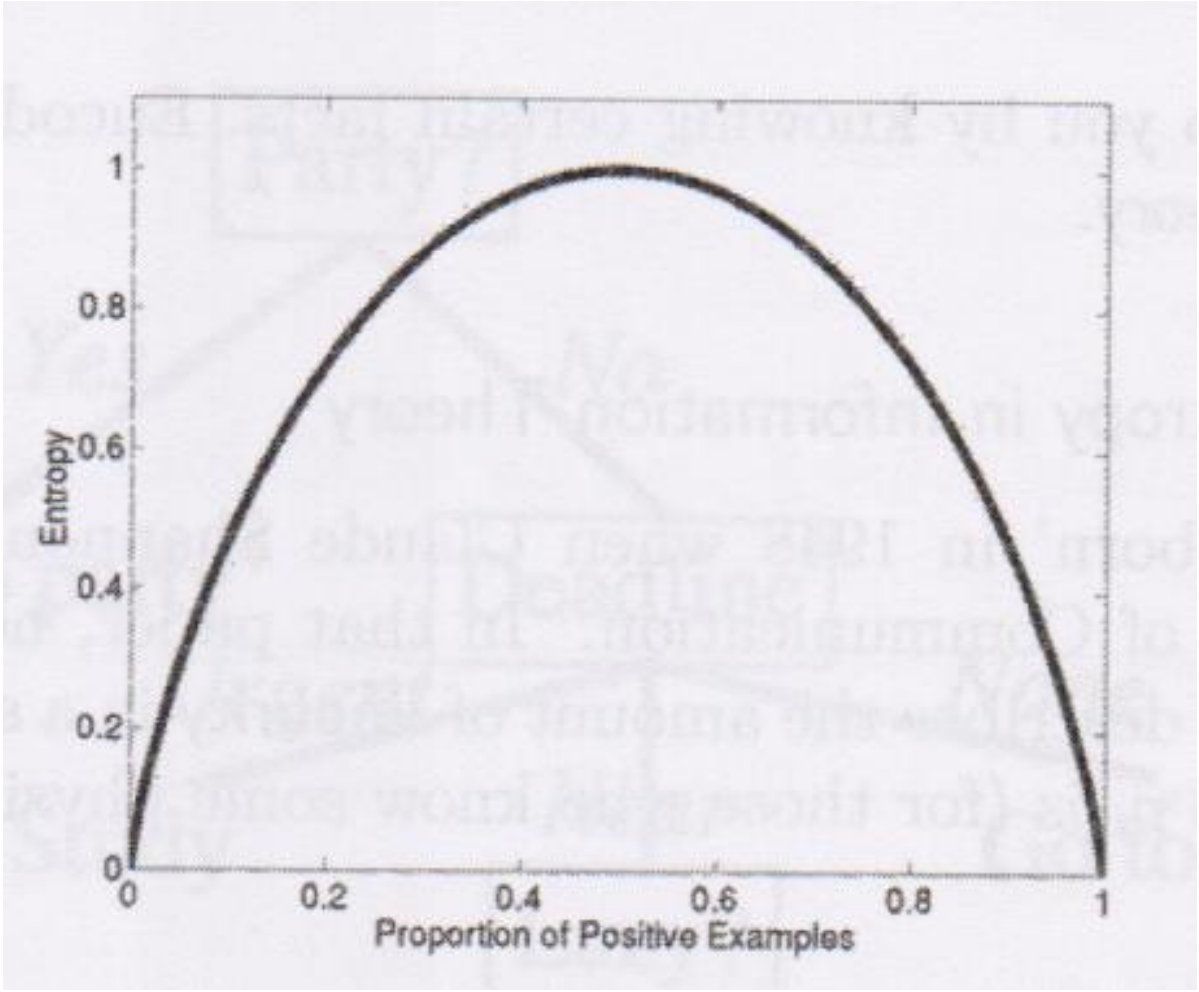


Fig. 7.2 A graph of entropy, detailing how much information is available from finding out another piece of information given what you already knows.

If the variable separates the cases into 50% positive and 50% negative, then the amount of entropy is at a maximum, and knowing about that variable is very useful. The basic concept is that the variable tells how much extra information we would get from knowing the value of that variable.

For a decision tree, the best variable to select the one to classify on now is the one that gives the most information, in the other words, the one with the highest entropy.

7.2 ID3

The important idea is to work out how much the entropy of the entire training set would decrease if we choose each particular variable for the next classification step in a node of the tree. This is known as the *information gain*, and it is defined as follows, where *S* is the set of cases, *V* is possible variable out of the set of all possible ones and $|S_v|$ is a number of members of *S* that have value *v* for variable *V*:

$$\text{Gain}(S,V) = \text{Entropy}(S) - \sum_{v \in \text{values}(V)} \frac{|S_v|}{|S|} \text{Entropy}(S_v) \qquad (7.1)$$

As an example, let us assume that there are data, with outcomes, S={s_1 = true, s_2 = false, s_3 = false, s_4 = false} and one variable V that can have values {v_1, v_2, v_3}. In the example, the variable value for s_1 could be v_2, for s_2 could be v_2, s_3, v_3 and s_4, v_1, then we can calculate the entropy of S as, where + means true, of which there is one case, and – means false, of which there are three cases (remember $\log_2 p = (\ln p)/\ln 2$):

$$\begin{aligned}\text{Entropy}(S) &= -p_+ \log_2 p_+ - p_- \log_2 p_- \\ &= -\frac{1}{4}\log_2 \frac{1}{4} - \frac{3}{4}\log_2 \frac{3}{4} \\ &= 0.5 + 0.311 \\ &= 0.811\end{aligned}$$

The function Entropy(S_v) is similar, but only computed with the subset of data where variable *V* has values *v*.

Now the information gain of variable *V* is calculated one value by one inside the sum (7.1).

In the example in Fig. 7.1 the variables are ’Deadline’, ’Party’ and ’Lazy’.

According to (7.1) and the values of outcomes *S* and variable *V* on p. 104 (Chapter 3) we obtain the following:

$$\frac{|S_{v_1}|}{|S|}\text{Entropy}(S_{v_1}) = \frac{1}{4}\left(-\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1}\right)$$

$$= 0$$

$$\frac{|S_{v_2}|}{|S|}\text{Entropy}(S_{v_2}) = \frac{2}{4}\left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right)$$

$$= \frac{1}{2}$$

$$\frac{|S_{v_3}|}{|S|}\text{Entropy}(S_{v_3}) = \frac{1}{4}\left(-\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1}\right)$$

$$= 0$$

The information gain from adding this variable is the entropy of *S* minus the sum of the three values above:

$$\text{Gain}(S,V) = 0.811 - (0 + 0.5 + 0) = 0.311$$

The ID3 algorithm computes this information gain for each variable and selects the one that produces the highest value. That is all there is to the algorithm. It searches for the space of possible trees in a greedy way by choosing the variable with the highest information gain at each stage. The output of the algorithm is the tree, that is, a list of inner nodes, edges and leaves. It is natural to be constructed recursively.

At each stage the best variable is selected and then removed from the dataset, and the algorithm is recursively called on the rest. The recursion stops when either there is only one class remaining in the data, in which case a leaf is added with that class as its label, or there are no variables left, the most common label in the remaining data is used.

ID3 algorithm

(1) If all cases (datapoints, samples, observations or examples) have the same label:

    (1.1) Return a leaf with that label.

(2) Else if there are no variables left to test:

    (2.1) Return a leaf with the most common label.

(3) Else:

    (3.1) Choose the variable $\tilde{V}$ that maximizes the information gain of S to be the next node using Equation (7.1).

    (3.2) Add a branch from the node for each possible value v in $\tilde{V}$.

(3.3) For each branch:

(3.3.1) Calculate S_v by removing $\tilde{V}$ from the set of variables.

(3.3.2) Recursively call the algorithm with S_v, to compute the gain relative to the current set of cases.

It is worth noting how ID3 generalizes from training cases to the set of all possible input. It uses a method known as the *inductive bias*. The choice of the next variable to add into the tree is the one with highest information gain, which biases the algorithm towards smaller trees, since it attempts to minimize the amount of information left. This is consistent with a well-known principle that short solutions are typically better than longer ones – not necessarily true, but simpler explanations are often easier to remember and understand. This follows the principle of *Occam's Razor*.

There is a sound information-theoretic way to write down this principle. This is known as *Minimum Description Length* (MDL) proposed by Jorma Rissanen in 1989. In essence it says that the shortest description of something, i.e., the most compressed one, is the best description.

Somewhat more formally the MDL principle says that *the complexity of a theory (model or hypothesis) is measured by the number of bits needed to encode the theory itself, plus the number of bits needed to encode the data using the theory*.

The MDL principle can be seen as a formalization of the Occam's razor heuristic.

Note that the algorithm is able to deal with noise in the dataset, since the (class) labels are assigned to the most common value of the target variable. Another benefit of the decision trees is that they can deal with missing data. If the variable value of a data case is missing, this case is skipped subject to the current node and the choice is made on the basis of the known values of this variable. This is virtually impossible for several other machine learning algorithms, such as neural networks which require a value, measured or at least imputed.

Characterizing ID3 to be biased towards short trees is only partly true. It uses all variables given, even if some of them are not necessary. This obviously runs the risk of overfitting (overlearning or overtraining), even makes it likely. To avoid overfitting, the simplest thing is to limit the size of the tree. One can also use a variant of *early stopping* by using a validation set (a third small set, separate from the training and test sets) and measuring the performance of the tree so far against it. Still, obviously the common way is to use *pruning*, e.g., applied in C4.5 invented by Quinlan to improve ID3.

There are a few versions of pruning. Their approach is to first compute a full tree and then to reduce it, evaluating the error of a validation set. The simplest version runs the decision tree algorithm until all of the variables are used, so that it is probably overfitted, and then produces smaller trees by running over the tree, picking each node in turn, and replacing the subtree below every node with a leaf labeled with the most common classification of the subtree. The error of the pruned tree is evaluated on the validation set, and the pruned tree is kept if the error is the same as or less than that of the original tree, and rejected otherwise.

Nevertheless, C4.5 uses a different technique called *post-pruning*. It takes the tree generated by ID3, converts this to a set of if-then rules, and then prunes each rule by removing preconditions if the accuracy on the rule increases without it. The rules are then sorted along with their accuracy on the training set and applied in order. The advantages of the rules are that they are easy to read and their order in the tree does not matter, just their accuracy in the classification.

7.3 Dealing with continuous variables

Thus far, we have only considered discrete values for variables of decision trees. For continuous variable the simplest approach is to discretize them.

It is also possible to leave the variable continuous and modify the tree algorithm itself. For a continuous variable there is not just one place to split it, but the variable can be splitted between any pair of datapoints, as shown in Fig. 7.3. There are, of course, infinite number of locations to split along the axis, but they are not different to this smaller set of locations. Even this smaller set makes the algorithm computationally more expensive for continuous variables than it is for discrete ones, since as well as calculating the information gain of each variable to pick the best one, the information gain of many points within each variable has to be computed. Usually, only one split is made to a continuous variable, although several are possible.

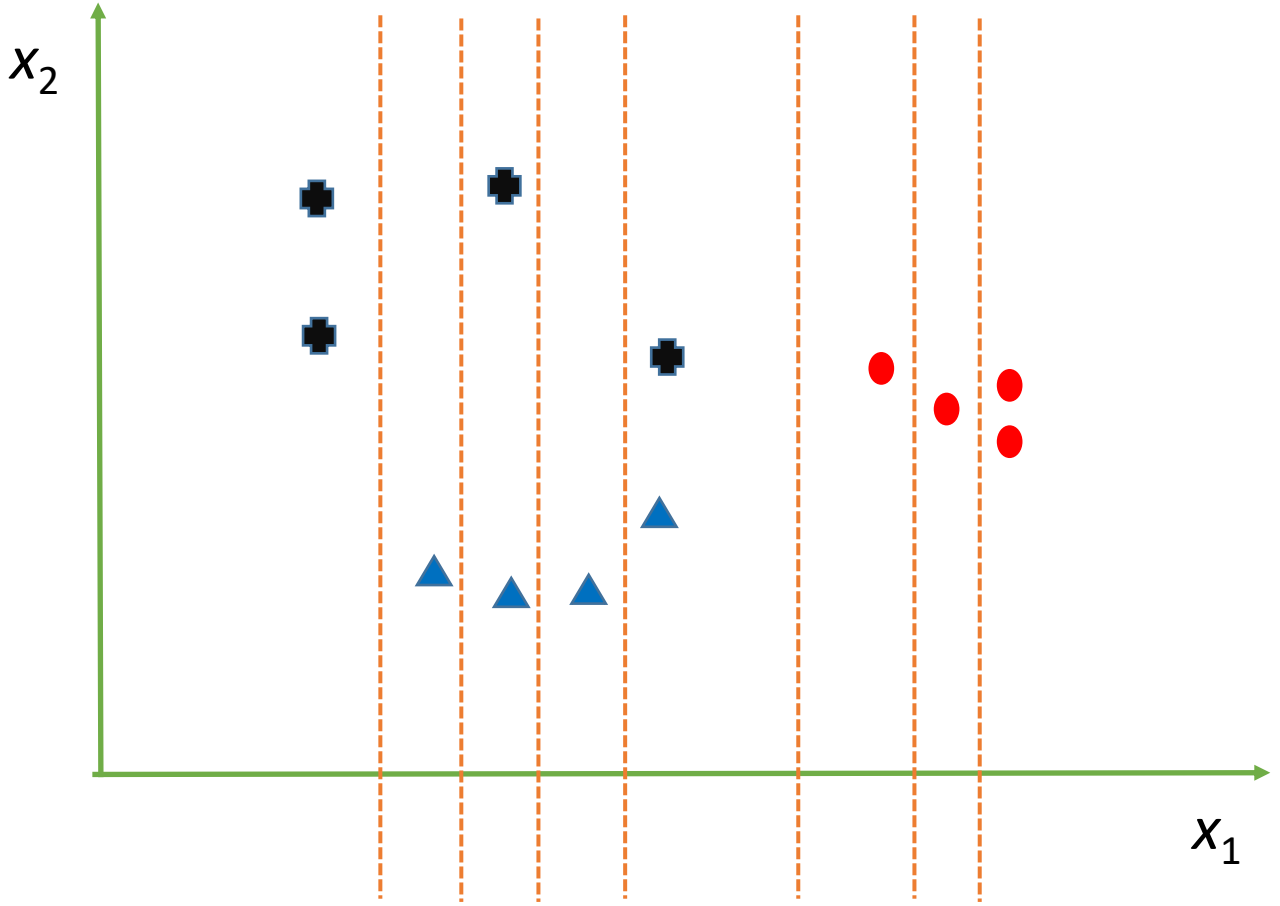
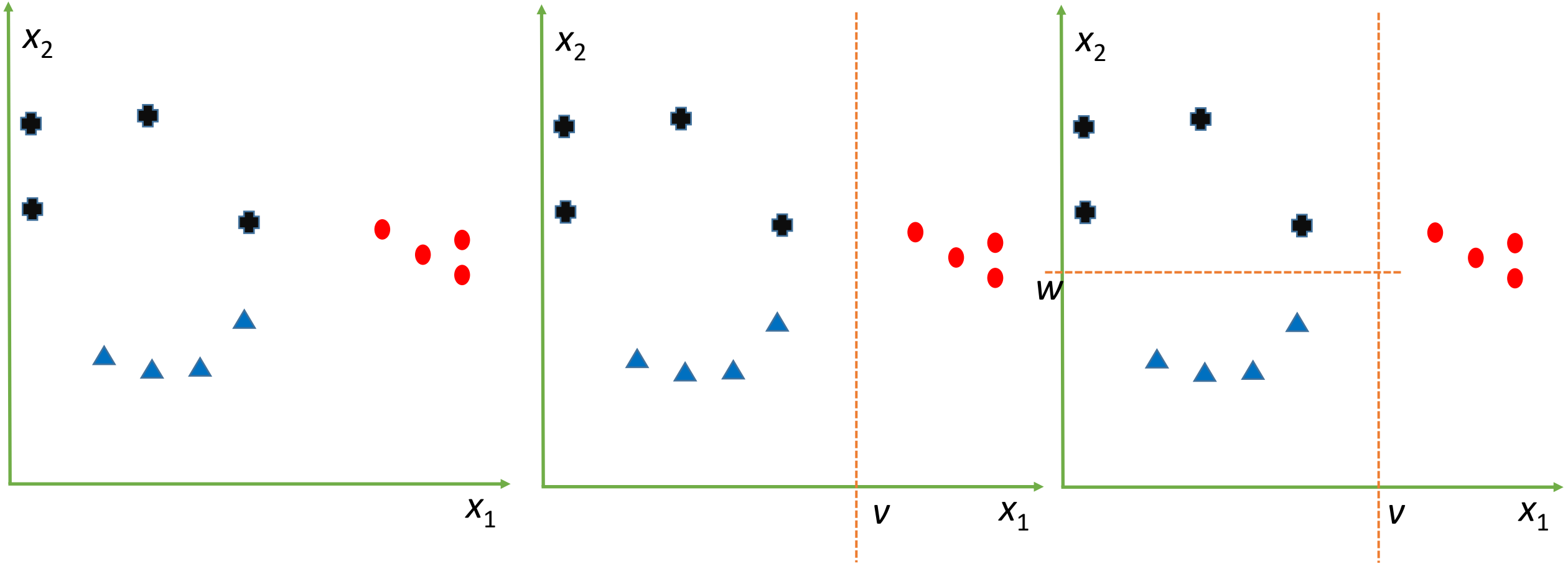


Fig. 7.3 Possible positions to split the variable x_1, between each of the datapoints (cases).

Trees made by these algorithms are all *univariate*, since they pick one variable (dimension in the variable space) at a time and split along to that one. *Multivariate* trees utilizing combinations of variables can be used for other algorithms, but it is, of course, more complicated. Producing much smaller trees is possible provided that it is possible to find straight lines that separate the data well, but are not parallel to any axis.

When one variable is chosen at a time, the decision tree building process can be easily visualized as in Figs. 7.4 and 7.5.



(a) (b) (c)

Fig. 7.4 The process of decision tree choices where the dataset in (a) is split first (b) by selecting variable x_1, (c) then x_2, which separates out the tree classes. The final tree is in Fig. 7.5.

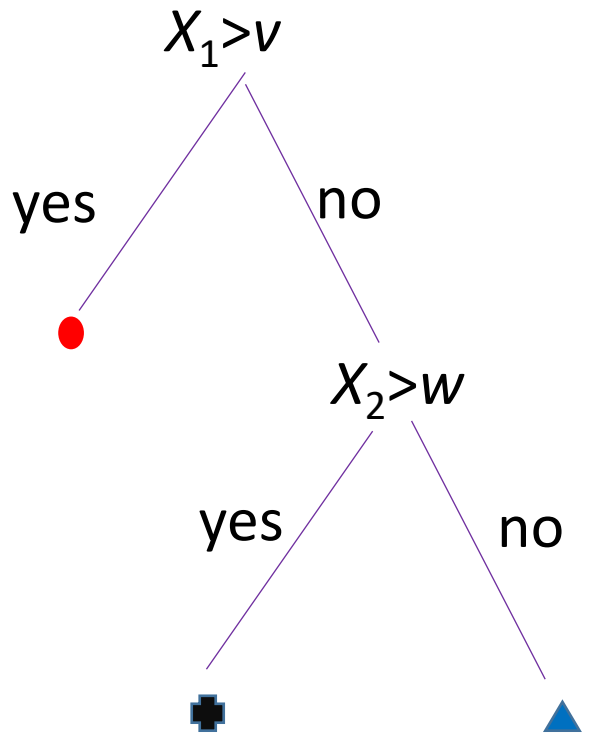


Fig. 7.5 The final tree generated by the splits in Fig. 7.4.

7.4 Computational complexity

The computational time complexity is $O(m \log m)$ as known for building binary trees and $O(\log m)$ for returning a particular leaf, in which m is the number of nodes. Nonetheless, these results are not *balanced* or *complete* binary trees, and decision trees are often not such. Although the information measures attempt to keep the tree balanced by finding splits separating the data into two even parts (this gives the greatest entropy), there is no guarantee of this. Nor are they necessarily binary.

7.5 Classification and regression trees (CART)

There is another well-known tree-based algorithm, CART, which can be used for both classification and regression. Notwithstanding the two ”roles” of CART, classification is not much different compared to the previous approaches although it is usually constrained to construct binary trees. This is actually no limitation, since we can always turn questions in binary questions. For example returning back to situation in Fig. 7.1 on p. 210, where a question has three answers ’urgent’, ’near’ or ’none’. This can be split into two questions: first, ’is the deadline urgent?’, and then ’if the answer to that is ’no’, second ’is the deadline near?’. In general, more complicated than binary trees can always be transformed into binary trees.

Gini impurity

The only real difference with classification in CART is that a different information measure is commonly used.

The *Gini impurity* is applied in CART. The impurity suggests that the aim of the decision tree is to have every leaf to represent a set of datapoints (data cases) from the same class, so that there are no mismatches. This is *purity*. If a leaf is pure, then all of the training data within it have just one class. Then, if we count the number $N(i)$ of datapoints at the node or better the **fraction** of the number of datapoints, that belong to class i, it should be 0 for all except one value of i.

To decide which variable to select for a split, the algorithm loops over the variables and checks how many points belong to each class. If a node is pure, then $N(i)=0$ for all values of i except one particular one. Thus, for any particular variable k, we compute

$$G_k = \sum_{i=1}^{C} \sum_{j \neq i} N(i)N(j) \qquad (7.2)$$

where C is the number of classes. The algorithmic effort required can be reduced by noticing that $\Sigma_i N(i)=1$ (there has to be some output class) and so $\Sigma_{j \neq i} N(j)=1 - N(i)$. The equation (7.2) is equivalent to:

$$G_k = 1 - \sum_{i=1}^{C} N(i)^2 \qquad (7.3)$$

The Gini impurity means computing the expected error rate if the classification was picked according to the class distribution. *The information gain can then be measured in the same way, subtracting each value G_i from the total Gini impurity.*

The information measure can be changed by adding a weight to the misclassifications. We then consider the cost of misclassifying a case of class *i* as class *j* also called risk and add a weight that tells how important each datapoint is. This is labeled as λ_{ij} and presented as a matrix, with element λ_{ij} representing the cost of of misclassifying *i* as *j* as follows.

$$G_i = \sum_{j \neq i} \lambda_{ij} N(i) N(j)$$

Gini index of formula (7.3) can be defined as

$$G_k = 1 - \sum_{i=1}^{C} (p(i|n))^2$$

where $p(i|n)$ denotes the fraction of cases from class *i* at the node *n* of a classification tree. Entropy can be also used for finding the most discriminatory variable or feature.

$$Entropy(n) = -\sum_{i=1}^{c} p(i|n) \log_2 p(i|n)$$

Regression in trees

To use regression in trees needs only a straightforward modification to the algorithm. If outputs are continuous, a regression model is appropriate. None of the node impurity measures considered so far will work. Instead, we use, of course, the error of the sum of squares (**the least sum of squares**). To evaluate which variable to choose next, one has to to find the value at which to split the dataset according to that variable. The output is a value at each leaf. In general, this is just a constant for the output, computed as the mean of all the datapoints that are located in the leaf.

This is the optimal choice in order to minimize the error of the sum of squares, but it also means that one can choose the split point quickly for a given variable, by selecting it to minimize the error of the sum of squares. The variable is taken that has the split point providing the best (least) error of the sum of squares, and continue to use the algorithm as for classification.

Example 1: Demographic vs. crime variables

Let us look at the previous example datasets. On p. 120-123, the dataset of demographic vs. crime variables was considered. The dataset was small including two classes with only three countries. That is why, the minimum size of a leaf was decreased down to 6, 3 or 2 countries only (e.g. default 10 in Matlab). (Naturally 6 was too large because of three countries in two small classes, but was used just to show what happens with it.) This is a stopping criterion not to generate branches to very small nodes causing possible overfitting.

The results in Table 7.1 show that they are clearly lower than those given by several other methods.

Table 7.1 Accuracy rates in per cent for the dataset of demographic vs. crime variables.

	Not scaled	Scaled into [0,1]	Standardized
Minimum size of node (number of cases)	Accuracy %	Accuracy %	Accuracy %
6	57.1	57.1	58.9
3	66.1	66.1	66.1
2	67.9	67.9	67.9

Example 2: Vertigo data

For Vertigo data the results in Table 7.2 were obtained. They are presented class by class because of the fourth class, the smallest one, which was especially difficult for trees. The sensitivities (true positive rates) are presented.

The tests were run as 10 times 10-fold crossvalidation, 100 altogether for which means (and standard deviations) were computed.

The mean accuracy of 77.0% was gained for all decision trees here.

Table 7.2 Vertigo data: six classes of 815 cases with quite unbalanced class distribution (130, 146, 313, 41, 65, and 120 cases, respectively).

	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6
	Vestibular schwannoma	Benign positional vertigo	Meniere's disease	Sudden deafness	Traumatic vertigo	Vestibular neuritis
Sensitivity	72.7	63.8	87.4	43.6	81.1	80.1

The sensitivities are quite high except for Classes 2 and particularly 4. However, some other classifiers presented earlier produced better results.

Class 4 was difficult for the sake of an exceptional reason. The missing values of the dataset were hit particularly to one variable that was very important to Class 4 and, thus, the variable could not be left out. There were even 53% of its values missing. The imputation by using mode for it caused like fairly close to a constant variable (other used methods needed imputation). The missing portions were 88%, 54%, 59%, 15%, 34% and 62% in the six classes. The problematic variable was not so important for other classes than Class 4 for which it was obviously the most important of all. Nevertheless, the exceptional situation did deteriorate tree classification, because the tree processing was not able to overcome the problem of ”close to a constant variable”. This variable separated classes only weakly in trees. Therefore, Class 4 achieved poor results that also dropped the mean accuracy.

The ability of decision trees in classification may vary depending on data as is the cases for most classifier types. As to the cardiomyocytes example from p. 201, the decision trees were in the middle compared to the results of several other classifiers.

(In Matlab decision tree models are built with ’fitctree’. Tree models are then tested with ’predict’. A tree built can be presented with ’view’ as a figure or if-then rule presentation.)