# Chapter 6
# Decision Trees

The classifiers discussed in the previous chapters expect all attribute values to be presented at the same time. Such a scenario, however, has its flaws. Thus a physician seeking to come to grips with the nature of her patient's condition often has nothing to begin with save a few subjective symptoms. And so, to narrow the field of diagnoses, she prescribes lab tests, and, based on the results, perhaps other tests still. At any given moment, then, the doctor considers only "attributes" that promise to add meaningfully to her current information or understanding. It would be absurd to ask for all possible lab tests (thousands and thousands of them) right from the start.

The lesson is, exhaustive information often is not immediately available; it may not even be needed. The classifier may do better choosing the attributes one at a time, according to the demands of the situation. The most popular tool targeting this scenario is a *decision tree*. The chapter explains its classification behavior, and then presents a simple technique that induces a decision tree from data. The reader will learn how to benefit from tree-pruning, and how to convert the induced tree to a set of rules.

## 6.1 Decision Trees as Classifiers

The training set shown in Table 6.1 consists of eight examples described by three attributes and labeled as positive or negative instances of a given class. To simplify the explanation of the basic concepts, we will assume that all attributes are discrete. Later, when the underlying principles become clear, we will slightly generalize the approach to make it usable also in domains with continuous attributes or with mixed sets of continuous and discrete attributes.

**Decision Tree** Figure 6.1 shows a few example decision trees that are capable of dealing with the data from Table 6.1. The internal nodes represent attribute-value



M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-63913-0_6

**Table 6.1** Eight training examples described by three symbolic attributes and classified as positive and negative examples of a given class

| Example | crust size | shape | filling size | Class |
|---|---|---|---|---|
| $e1$ | big | circle | small | **pos** |
| $e2$ | small | circle | small | **pos** |
| $e3$ | big | square | small | **neg** |
| $e4$ | big | triangle | small | **neg** |
| $e5$ | big | square | big | **pos** |
| $e6$ | small | square | small | **neg** |
| $e7$ | small | square | big | **pos** |
| $e8$ | big | circle | big | **pos** |

tests, the edges indicate how to proceed in the case of diverse test results, and the leafs[1] contain class labels. An example to be classified is first subjected to the test prescribed at the topmost node, the *root*. The result of this test then decides along which edge the example is to be sent down, and the process continues until a leaf node is reached. Once this happens, the example is labeled with the class associated with this leaf.

Let us illustrate the process using the tree from Fig. 6.1b. The root asks about the `shape`, each of whose three values is represented by one edge. In examples $e1$, $e2$, and $e8$, we find the value `shape=circle` which corresponds to the left edge. The example is sent down along this edge, ending in a leaf that labeled `pos`. This indeed is the class common to all these three examples. In $e4$, `shape=triangle`, and the corresponding edge ends in a leaf labeled `neg`—again, the correct class. Somewhat more complicated is the situation with examples $e3$, $e5$, $e6$, and $e7$ where `shape=square`. For this particular value, the edge ends not in a leaf, but only at a test-containing node, this one inquiring about the value of `filling-size`. In the case of $e5$ and $e7$, the value is `big`, which leads to a leaf labeled with `pos`. In the other two examples, $e3$ and $e6$, the value is `small`, and this sends them to a leaf labeled with `neg`.

We have shown that the decision tree from Fig. 6.1b identifies the correct class for all training examples. By way of a little exercise, the reader is encouraged to verify that the other trees shown in the picture are just as successful.[2]

**Interpretability** Comparing this classifier with those introduced earlier, we can see one striking advantage: *interpretability*. If anybody asks why example $e1$ is deemed positive, the answer is, "because its `shape` is `circle`." Other classifiers do not offer explanations of this kind. Especially the neural network is a real *black box*: when presented with an example, it simply returns the class and never offers any

[1] Both spellings are used: *leaves* and *leafs*. The latter is probably more appropriate because the "leaf" in question is supposed to be a data abstraction that has nothing to do with the original physical object.

[2] In as sense, the decision tree can be seen as a simple mechanism for data compression.

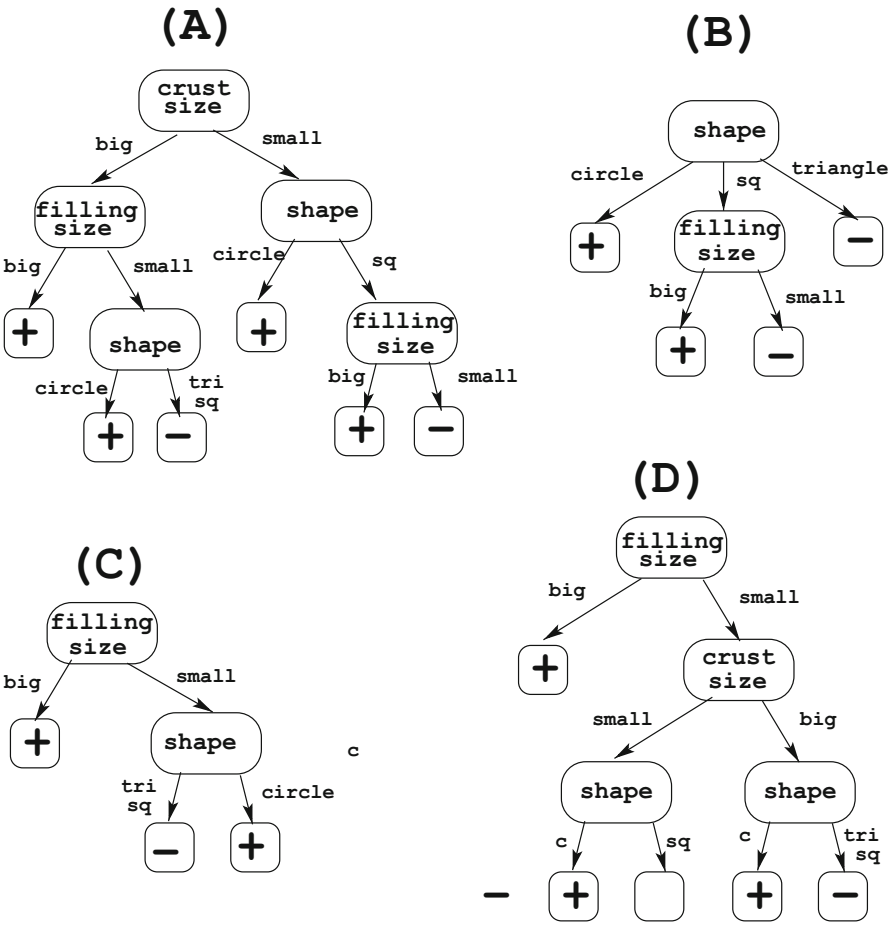



**Fig. 6.1** Example decision trees for the "pies" domain. Note how they differ in size and in the order of tests. Each of them classifies correctly all training examples listed in Table 6.1, `tri`, `sq`, and `c` stand for `triangle`, `square`, and `circle`, respectively

insight as to why this particular label has been given preference over other labels. The situation is not much better in the case of Bayesian and linear classifiers. Only the *k-NN* classifier offers a semblance of a—rather rudimentary—argument. For instance, one can say that, "**x** should be labeled with `pos` because this is the class of the training example most similar to **x**." Such a statement, however, is a far cry from the explicit attribute-based explanation made possible by the decision tree.

One can go one step further and interpret a decision tree as a set of rules such as "if `shape=square` and `filling-size=big`, then the example belongs to class `pos`." A domain expert inspecting these rules may then decide whether they are intuitively appealing, and whether they agree with his or her "human

understanding" of the problem at hand. The expert may even be willing to suggest improvements to the tree; for instance, by pointing out spurious tests that have found their way into the data structure only on account of some random regularity in the data.

**Missing Edge** The reader will recall that, in linear classifiers, an example may find itself exactly on the class-separating hyperplane, in which case the class is selected more or less randomly. Something similar occasionally happens in decision trees, too. Suppose the tree from Fig. 6.1a is used to determine the class of the following example:

$$(\texttt{crust} - \texttt{size} = \texttt{small})\texttt{AND}(\texttt{shape} = \texttt{triangle})\texttt{AND}(\texttt{filling} - \texttt{size} = \texttt{small})$$

Let us follow the procedure step by step. The root inquires about `crust-size`. Realizing that the value is `small`, the classifier sends the example down the right edge, to the test on `shape`. Here, only two outcomes appear to be possible: `circle` or `square`, but not `triangle`. The reason is, whoever created the tree had no idea that an object with `crust-size=small` could be triangular: nothing of that kind could be found in the training set. Therefore, there did not seem to be any reason for creating the corresponding edge. And even if the edge were created, it would not be clear where it should lead to.

The engineer implementing this classifier in a computer program must make sure the program "knows" what to do in the case of "missing edges." Choosing the class randomly or preferring the most frequent class are the most obvious possibilities.

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Describe the mechanism that uses a decision tree to classify examples. Illustrate the procedure using the decision trees from Fig. 6.1 and the training examples from Table 6.1.
- What is meant by the statement that, "the decision tree's choice of a concrete class can be explained to the user"? Is something similar possible in the case of the classifiers discussed in the previous chapters?
- Under what circumstances can a decision tree find itself unable to determine an example's class? How would you handle the situation if you were the programmer?

## 6.2 Induction of Decision Trees

We will begin with a very crude induction algorithm. Applying it to a training set, we will realize that a great variety of alternative decision trees can be obtained. A brief discussion will convince us that, among these, the smaller ones are to be preferred. This observation will motivate an improved version of the technique, thus preparing the soil for the following sections.

**Divide and Conquer** Let us try our hand at creating a decision tree manually. Suppose we decide that the root node should test the value of `shape`. In the training set, three different outcomes are found: `circle`, `triangle`, and `square`. For each, the classifier will need a separate edge leading from the root. The first, defined by `shape=circle`, will be followed by examples $T_C = \{e1, e2, e8\}$; the second, defined by `shape=triangle`, will be followed by $T_T = \{e4\}$; and the last, defined by `shape=square`, will be followed by $T_S = \{e3, e5, e6, e7\}$. Each of the three edges will begin at the root and end in another node, either an attribute test or a leaf containing a class label.

Seeing that all examples in $T_C$ are positive, we will let this edge point at a leaf labeled with `pos`. Similarly, the edge followed by the examples from $T_T$ will point at a leaf labeled with `neg`. Certain difficulties will arise only in the case of the last edge because $T_S$ is a mixture of both classes. To separate them, we need another test, say, `filling-size`, to be placed at the end of the edge. This attribute can acquire two values, `small` and `big`, dividing $T_S$ into two subsets. Of these, $T_{S-S} = \{e3, e6\}$ is characterized by `filling-size=small`; the other, $T_{S-B} = \{e5, e7\}$, is characterized by `filling-size=big`. All examples in $T_{S-S}$ are positive, and all examples in $T_{S-B}$ are negative. This allows us to let both edges end in leafs, the former labeled with `pos`, the latter with `neg`. At this moment, the tree-building process can stop because each training example presented to the classifier thus created will reach a leaf.

The reader will have noticed that each node of the tree can be associated with a set of examples that pass through it or end in it. Starting with the root, each test divides the training set into disjoint subsets, and these into further subsets, and so on until each subset is "pure" in the sense that all its examples belong to the same class. This is why the approach is sometimes referred to as the *divide-and-conquer* technique.

**Alternative Trees** In the process thus described, the (rather arbitrary) choice of `shape` and `filling-size` resulted in the decision tree shown in Fig. 6.1b. To get used to the mechanism, the student is encouraged to experiment with alternatives such as placing at the root the test on `crust-size` or `filling-size`, and considering different options for the tests at the lower level(s). Quite a few other decision trees will thus be created—some of them depicted in Fig. 6.1.

That so many solutions can be found even in this very simple toy domain is a food for thought. Is there a way to decide which trees are better? So, an improved version of the divide-and-conquer technique should be able to arrive at a "good" tree *by design*, and not by mere chance.

**The Size of the Tree** The smallest of the data structures in Fig. 6.1 consists of two attribute tests; the largest, of five. Differences of this kind may have a strong impact on the classifier's behavior. Before proceeding to the various aspects of this phenomenon, however, let us emphasize that the number of nodes in the tree is not the only criterion of size. Just as important is the number of tests that have to be carried out when classifying an average example.

For instance, in a domain where `shape` is almost always `circle` or `triangle` (and only very rarely `square`), the average number of tests prescribed by the tree from Fig. 6.1b will only slightly exceed 1 because both `shape=circle` and `shape=triangle` immediately point at leafs with class labels. But if the prevailing `shape` is `square`, the average number tests approaches 2. Quite often, then, a bigger tree may result in fewer tests than a smaller one.

**Small Trees Versus Big Trees** There are several reasons why small decision trees are preferred. One of them is *interpretability*. A human expert finds it easy to analyze, explain, and perhaps even correct, a decision tree that consists of no more than a few tests. The larger the tree, the more difficult this is.

Another advantage of small decision trees is their tendency to dispose of *irrelevant* and *redundant* information. Whereas the relatively large tree from Fig. 6.1a employs all three attributes, the smaller one from Fig. 6.1b is just as good at classifying the training set—without ever considering `crust-size`. Such economy will come handy in domains where certain attribute values are expensive or time-consuming to obtain.

Finally, larger trees are prone to *overfit* the training examples. This is because the divide-and-conquer method keeps splitting the training set into smaller and smaller subsets, the number of these splits being equal to the number of attribute tests in the tree. Ultimately, the resulting training subsets can become so small that the classes may get separated by an attribute that only by chance—or noise—has a different value in the remaining positive and negative examples.

**Induction of Small Decision Trees** When illustrating the behavior of the divide-and-conquer technique on the manual tree-building procedure, we picked the attributes at random. When doing so, we observed that some choices led to smaller trees than others. Apparently, the attributes differ in how much information they convey. For instance, `shape` is capable of immediately labeling some examples as positive (if the value is `circle`) or negative (if the value is `triangle`); but `crust-size` cannot do so unless assisted by some other attribute.

Assuming that there is a way to measure the amount of information provided by each attribute (and such a mechanism indeed exists, see Sect. 6.3), we are ready to formalize the technique for induction of decision trees by a pseudocode. The reader will find it in Table 6.2.

**Table 6.2** Induction of decision trees

Let $T$ be the training set.

*grow(T):*

(1) Find the attribute, *at*, that contributes the maximum information about the class labels.
(2) Divide $T$ into subsets, $T_i$, each characterized by a different value of *at*.
(3) For each $T_i$:
If all examples in $T_i$ belong to the same class, then create a leaf labeled with this class; otherwise, apply the same procedure recursively to each training subset: *grow*($T_i$).

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of the divide-and-conquer technique for induction of decision trees.
- What are the advantages of small decision trees in comparison to larger ones?
- What determines the size of the decision tree obtained by the divide-and-conquer technique?

## 6.3 How Much Information Does an Attribute Convey?

To create a relatively small decision tree, the divide-and-conquer technique relies on one critical component: the ability to decide how much information about the class labels is conveyed by the individual attributes. This section introduces a mechanism to calculate this quantity.

**Information Contents of a Message** Suppose we know that the training examples are labeled as `pos` or `neg`, the relative frequencies of these two classes being $p_{\texttt{pos}}$ and $p_{\texttt{neg}}$, respectively.[3] Let us select a random training example. How much information is conveyed by the message, "this example's class is `pos`"?

The answer depends on $p_{\texttt{pos}}$. In the extreme case where all examples are known to be positive, $p_{\texttt{pos}} = 1$, the message does not tell us anything new. The reader knows the example is positive even without being told so. The situation changes if

[3]Recall that the relative frequency of `pos` is the percentage (in the training set) of examples labeled with `pos`; this represents the probability that a randomly drawn example will be positive.

**Table 6.3** Some values of the information contents (measured in bits) of the message, "this randomly drawn example is positive."

| $p_{\mathtt{pos}}$ | $-\log_2 p_{\mathtt{pos}}$ |
|---|---|
| 1.00 | 0 bits |
| 0.50 | 1 bit |
| 0.25 | 2 bits |
| 0.125 | 3 bits |

Note that the message is impossible for $p_{\mathtt{pos}} = 0$

both classes are known to be equally represented so that $p_{\mathtt{pos}} = 0.5$. Here, the guess is no better than a flipped coin, so the message *does* offer some information. And if a great majority of examples are known to be negative so that, say, $p_{\mathtt{pos}} = 0.01$, then the reader is all but certain that the chosen example is going to be negative as well; the message telling him that this is not the case is unexpected. And the lower the value of $p_{\mathtt{pos}}$, the more information the message offers.

When quantifying the information contents of such a message, the following formula has been found convenient:

$$I_{\mathtt{pos}} = -\log_2 p_{\mathtt{pos}} \tag{6.1}$$

The negative sign compensates for the fact that the logarithm of $p_{\mathtt{pos}} \in (0, 1)$ is always negative. Table 6.3 shows the information contents for some typical values of $p_{\mathtt{pos}}$. Note that the unit for the amount of information is 1 *bit*. Another comment: the base of the logarithm being 2, it is fairly common to write $\log p_{\mathtt{pos}}$ instead of the more meticulous $\log_2 p_{\mathtt{pos}}$.

**Entropy (Average Information Contents)** So much for the information contents of a single message. Suppose, however, that the experiment is repeated many times. Both messages will occur, "the example is positive," and "the example is negative," the first with probability $p_{\mathtt{pos}}$, the second with probability $p_{\mathtt{neg}}$. The average information contents of all these messages is then obtained by the following formulas where the information contents of either message is weighted by its probability (the $T$ in the argument refers to the training set):

$$H(T) = -p_{\mathtt{pos}} \log_2 p_{\mathtt{pos}} - p_{\mathtt{neg}} \log_2 p_{\mathtt{neg}} \tag{6.2}$$

The attentive reader will protest that the logarithm of zero probability is not defined, and Eq. (6.2) may thus be useless if $p_{\mathtt{pos}} = 0$ or $p_{\mathtt{neg}} = 0$. Fortunately, a simple analysis (using limits and l'Hopital's rule) will convince us that, for $p$ converging to zero, the expression $p \log p$ converges to zero, too, which means that $0 \cdot \log 0 = 0$.

$H(T)$ is called *entropy of* $T$. Its value reaches its maximum, $H(T) = 1$, when $p_{\mathtt{pos}} = p_{\mathtt{neg}} = 0.5$ (because $0.5 \cdot \log 0.5 + 0.5 \cdot \log 0.5 = 1$); and it drops to its minimum, $H(T) = 0$, when either $p_{\mathtt{pos}} = 1$ or $p_{\mathtt{neg}} = 1$ (because $0 \cdot \log 0 + 1 \cdot$

$\log 1 = 0$). By the way, the case with $p_{\texttt{pos}} = 1$ or $p_{\texttt{neg}} = 1$ is regarded as perfect regularity because all examples belong to the same class; conversely, the case with $p_{\texttt{pos}} = p_{\texttt{neg}} = 0.5$ is seen as a total lack of regularity.

**Amount of Information Contributed by an Attribute** The concept of entropy (lack of regularity) will help us deal with the main question: how much does the knowledge of the value of a discrete attribute, *at*, tell us about an example's class?

Let us remind ourselves that *at* divides the training set, $T$, into subsets, $T_i$, each characterized by a different value of *at*. Quite naturally, each subset will be marked by its own probabilities (estimated by relative frequencies) of the two classes, $p_{ipos}$ and $p_{ineg}$. Based on the knowledge of these, Eq. (6.2) will give us the corresponding entropies, $H(T_i)$.

Now let $|T_i|$ be the number of examples in $T_i$, and let $|T|$ be the number of examples in the whole training set, $T$. The probability that a randomly drawn training example will be in $T_i$ is estimated as follows:

$$P_i = \frac{|T_i|}{|T|} \tag{6.3}$$

We are ready to calculate the weighted average of the entropies of the subsets.

$$H(T, at) = \Sigma_i P_i \cdot H(T_i) \tag{6.4}$$

The obtained result, $H(T, at)$, is the entropy of a system where not only the class labels, but also the values of *at* are known for each training example. The amount of information contributed by *at* is then the difference between the entropy *before* *at* has been considered and the entropy *after* this attribute has been considered:

$$I(T, at) = H(T) - H(T, at) \tag{6.5}$$

It would be easy to prove that this difference cannot be negative; information can only be gained, never lost, by considering *at*. In certain rare cases, however, $I(T, at) = 0$, which means that no information has been gained, either.

Applying Eq. (6.5) separately to each attribute, we can find out which of them provides the maximum amount of information, and as such is the best choice for the "root" test in the first step of the algorithm from Table 6.2.

The procedure just described is summarized by the pseudocode in Table 6.4. The process starts by the calculation of the entropy of the system where only class percentages are known. Next, the algorithm calculates the information gain conveyed by each attribute. The attribute that offers the highest information gain is deemed best.

**Illustration** Table 6.5 shows how to use the mechanism for the selection of the most informative attribute in the domain from Table 6.1. At the beginning, the entropy, $H(T)$, of the system without attributes is established. Then, we observe that, for instance, the attribute `shape` divides the training set into three subsets. The

**Table 6.4** The algorithm to find the most informational attribute

1. Calculate the entropy of the training set, $T$, using the percentages, $p_{\texttt{pos}}$ and $p_{\texttt{neg}}$, of the positive and negative examples:

$$H(T) = -p_{\texttt{pos}} \log_2 p_{\texttt{pos}} - p_{\texttt{neg}} \log_2 p_{\texttt{neg}}$$

2. For each attribute, *at*, that divides $T$ into subsets, $T_i$, with relative sizes $P_i$, do the following:
   (i) calculate the entropy of each subset, $T_i$;
   (ii) calculate the average entropy: $H(T, at) = \Sigma_i P_i \cdot H(T_i)$;
   (iii) calculate information gain: $I(T, at) = H(T) - H(T, at)$
3. Choose the attribute with the highest value of information gain.

average of their entropies, $H(T, \texttt{shape})$, is calculated, and the difference between $H(T)$ and $H(T, \texttt{shape})$ gives the amount of information conveyed by this attribute. Repeating the procedure for `crust-size` and `filling-size`, and comparing the results, we realize the `shape` contributes more information than the other two attributes, and this is why we choose `shape` for the root test.

This, by the way, is how the decision tree from Fig. 6.1b was obtained.

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What do we mean when we talk about the "amount of information conveyed by a message"? How is this amount determined, and what units are used?
- What is *entropy* and how does it relate to the frequency of the positive and negative examples in the training set?
- How do we use entropy when assessing the amount of information contributed by an attribute?

## 6.4 Binary Split of a Numeric Attribute

The entropy-based mechanism from the previous section requested that all attributes should be discrete. With a little modification, however, the same approach can be applied to continuous attributes as well. All we need is to convert them to boolean attributes.

**Table 6.5** Illustration of the search for the most informative attribute

| Example | crust size | shape | filling size | Class |
|---|---|---|---|---|
| $e1$ | big | circle | small | **pos** |
| $e2$ | small | circle | small | **pos** |
| $e3$ | big | square | small | **neg** |
| $e4$ | big | triangle | small | **neg** |
| $e5$ | big | square | big | **pos** |
| $e6$ | small | square | small | **neg** |
| $e7$ | small | square | big | **pos** |
| $e8$ | big | circle | big | **pos** |

Here is the entropy of the training set where only class labels are known:

$$\begin{aligned} H(T) &= -p_{\texttt{pos}} \log_2 p_{\texttt{pos}} - p_{\texttt{neg}} \log_2 p_{\texttt{neg}} \\ &= -(5/8)\log(5/8) - (3/8)\log(3/8) = 0.954 \end{aligned}$$

Next, we calculate the entropies of the subsets defined by the values of `shape`:

$$\begin{aligned} H(\texttt{shape=square}) &= -(2/4)\cdot\log(2/4) - (2/4)\cdot\log(2/4) = 1 \\ H(\texttt{shape=circle}) &= -(3/3)\cdot\log(3/3) - (0/3)\cdot\log(0/3) = 0 \\ H(\texttt{shape=triangle}) &= -(0/1)\cdot\log(0/1) - (1/1)\cdot\log(1/1) = 0 \end{aligned}$$

From these, we obtain the average entropy of the system where the class labels *and* the value of `shape` is known:

$$H(T, \texttt{shape}) = (4/8)\cdot 1 + (3/8)\cdot 0 + (1/8)\cdot 0 = 0.5$$

Repeating the same procedure for the other two attributes, we obtain the following:

$$\begin{aligned} H(T, \texttt{crust}-\texttt{size}) &= 0.951 \\ H(T, \texttt{filling}-\texttt{size}) &= 0.607 \end{aligned}$$

These values give the following information gains:

$$\begin{aligned} I(T, \texttt{shape}) &= H(T) - H(T, \texttt{shape}) &&= 0.954 - 0.5 &&= 0.454 \\ I(T, \texttt{crust}-\texttt{size}) &= H(T) - H(T, \texttt{crust}-\texttt{size}) &&= 0.954 - 0.951 &&= 0.003 \\ I(T, \texttt{filling}-\texttt{size}) &= H(T) - H(T, \texttt{filling}-\texttt{size}) &&= 0.954 - 0.607 &&= 0.347 \end{aligned}$$

We conclude that maximum information is contributed by `shape`.

**Converting a Continuous Attribute to a Boolean One** Let us denote the continuous attribute by $x$. The trick is to choose a threshold, $\theta$, and then decide that if $x < \theta$, then the value of the newly created boolean attribute is *true*, and otherwise it is *false* (or vice versa).

Simple enough. But what concrete $\theta$ to choose? Surely there are many of them? Here is one possibility.

Suppose that $x$ has a different value in each of the $N$ training examples. Let us sort these values in ascending order, denoting by $x_1$ the smallest, and by $x_N$ the highest. Any pair of neighboring values, $x_i$ and $x_{i+1}$, then defines a threshold, $\theta_i = (x_i + x_{i+1})/2$. For instance, a four-example training set where $x$ has values 3, 4, 6, and 9 leads us to consider $\theta_1 = 3.5, \theta_2 = 5.0$, and $\theta_3 = 7.5$. For each of these $N-1$ thresholds, we calculate the amount of information contributed by the boolean attribute thus defined, and then choose the threshold where the information gain is maximized.

**Candidate Thresholds** The approach just described deserves to be criticized for its high computational costs. Indeed, in a domain with a hundred thousand examples described by a hundred attributes (nothing extraordinary), the information contents of $10^5 \times 10^2 = 10^7$ different thresholds would have to be calculated. Fortunately, mathematicians have been able to prove that a great majority of these thresholds can just as well be ignored. This reduces the costs to a mere fraction.

The principle is illustrated in Table 6.6. In the upper part, 13 values of $x$ are ordered from left to right, each labeled with the class (positive or negative) of the training example in which the value was found. And here is the rule: the best threshold never finds itself between values that are labeled with the same class. This means that it is enough to investigate the contributed information only for locations between values with opposite class labels. In the specific case shown in Table 6.6, only three *candidate thresholds*, $\theta_1, \theta_2$, and $\theta_3$, need to be investigated (among the three, $\theta_1$ is shown to be best).

**The Root of a Numeric Decision Tree** The algorithm summarized by the pseudocode in Table 6.7 determines the best attribute test for the root of a decision tree in a domain where all attributes are continuous. Note that the test consists of a pair, $[at_i, \theta_{ij}]$, where $at_i$ is the selected attribute and $\theta_{ij}$ is the best threshold found for this attribute. If an example's value of the $i$-th attribute is below the threshold, $at_i < \theta_{ij}$, the left branch of the decision tree is followed; otherwise, the right branch is chosen.

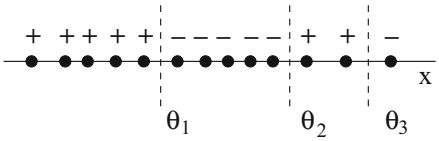
## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain why this section suggested to divide the domain of a continuous attribute into two subdomains.
- What mathematical finding has reduced the number of thresholds that need to be investigated?

**Table 6.6** Illustration of the search for the best threshold

The values of attribute $x$ are sorted in ascending order. The candidate thresholds are located between values labeled with opposite class labels.




Here is the entropy of the training set, ignoring attribute values:

$$H(T) = -p_+ \log p_+ - p_- \log p_-$$
$$= -(7/13)\log(7/13) - (6/13)\log(6/13) = 0.9957$$

Here are the entropies of the training subsets defined by the three candidate thresholds:

$$H(x < \theta_1) = -(5/5)\log(5/5) - (0/5)\log(0/5) = 0$$
$$H(x > \theta_1) = -(2/8)\log(2/8) - (6/8)\log(6/8) = 0.8113$$

$$H(x < \theta_2) = -(5/10)\log(5/10) - (5/10)\log(5/10) = 1$$
$$H(x > \theta_2) = -(2/3)\log(2/3) - (1/3)\log(1/3) = 0.9183$$

$$H(x < \theta_3) = -(7/12)\log(7/12) - (5/12)\log(5/12) = 0.9799$$
$$H(x > \theta_3) = -(0/1)\log(0/1) - (1/1)\log(1/1) = 0$$

Average entropies associated with the individual thresholds:

$$H(T, \theta_1) = (5/13) \cdot 0 + (8/13) \cdot 0.8113 = 0.4993$$
$$H(T, \theta_2) = (10/13) \cdot 1 + (3/13) \cdot 0.9183 = 0.9811$$
$$H(T, \theta_3) = (12/13) \cdot 0.9799 + (1/13) \cdot 0 = 0.9045$$

Information gains entailed by the individual candidate thresholds:

$$I(T, \theta_1) = H(T) - H(T, \theta_1) = 0.9957 - 0.4993 = 0.4964$$
$$I(T, \theta_2) = H(T) - H(T, \theta_2) = 0.9957 - 0.9811 = 0.0146$$
$$I(T, \theta_3) = H(T) - H(T, \theta_3) = 0.9957 - 0.9045 = 0.0912$$

Threshold $\theta_1$ gives the highest information gain.

**Table 6.7** Algorithm to find the best numeric-attribute test

1. For each attribute $at_i$:
   (i) Sort the training examples by the values of $at_i$;
   (ii) Determine the candidate thresholds, $\theta_{ij}$, as those lying between examples with opposite labels;
   (iii) For each $\theta_{ij}$, determine the amount of information contributed by the boolean attribute thus created.
2. Choose the pair $[at_i, \theta_{ij}]$ that offers the highest information gain.

## 6.5 Pruning

Section 6.2 extolled the virtues of small decision trees: interpretability, removal of irrelevant and redundant attributes, reduced danger of overfitting. These were the arguments that motivated the use of information theory in the course of decision-tree induction; they also motivate the step that follows: *pruning*.

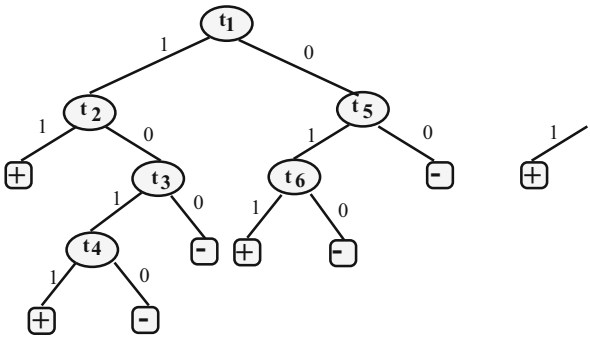



**Fig. 6.2** A simple approach to pruning will replace a subtree with a leaf

**The Essence** Figure 6.2 will help us explain the principle. On the left is the original decision tree whose six attribute tests are named $t_1, \ldots t_6$. On the right is a pruned version. Note that the subtree rooted in test $t_3$ in the original tree is in the pruned tree replaced with a leaf labeled with the negative class; and the subtree rooted in test $t_6$ is replaced with a leaf labeled with the positive class. The reader can see the point: pruning consists of replacing one or more subtrees with leafs, each labeled with the class most common among the training examples that reach—in the original classifier—the removed subtree.

This last idea sounds counterintuitive: the induction mechanism seeks to create a decision tree that scores zero errors on the training examples, but this perfection may be lost in the pruned tree! But the practically minded engineer is not alarmed. The ultimate goal is *not* to classify the training examples (their classes are known anyway). Rather, we want a tool capable of labeling *future* examples. And experience shows that this kind of performance is often improved by reasonable pruning.

**Error Estimate** Pruning is typically carried out in a sequence of steps: first replace with a leaf one subtree, then another, and so on, as long as the replacements appear to be beneficial according to some reasonable criterion. The term "beneficial" is meant to warn us that small-tree advantages should not be outbalanced by reduced classification performance.

Which brings us to the issue of *error estimate*. The principle is illustrated in Fig. 6.2. Let $m$ be the number of training examples that reach test $t_3$ in the decision tree on the left. If we replace the subtree rooted in $t_3$ by a leaf (as happened in the tree on the right), some of these $m$ examples may become misclassified. Denoting the number of these misclassified examples by $e$, we may be tempted to estimate the probability that an example will be misclassified at this leaf by the relative frequency: $e/m$. But admitting that small values of $m$ may render this estimate problematic, we prefer the following formula where $N$ is the total number of training examples:

$$E_{estimate} = \frac{e+1}{N+m} \tag{6.6}$$

The attentive reader may want to recall (or re-read) what Sect. 2.3 had to say about the difficulties of probability estimates of rare events.

**Error Estimates for the Whole Tree** Once again, let us return to Fig. 6.2. The tree on the left has two subtrees, one rooted at $t_2$, the other at $t_5$. Let $m_2$ and $m_5$ be the numbers of the training examples reaching $t_2$ and $t_5$, respectively; and let $E_2$ and $E_5$ be the error estimates of the two subtrees, obtained by Eq. (6.6). For the total of $N = m_2 + m_5$ training examples, the error rate of the whole subtree is estimated as the weighted average of the two subtrees:

$$E_R = \frac{m_2}{N} E_2 + \frac{m_5}{N} E_5 \tag{6.7}$$

Of course, in a situation with more than just two subtrees, the weighted average has to be taken over all of them. This should present no major difficulties.

As for the values of $E_2$ and $E_5$, these are obtained from the error rates of the specific subtrees, and these again from the error rates of their sub-subtrees, and so on, all the way down to the lowest level tests. The error-estimating procedure is a recursive undertaking.

Suppose that the tree to be pruned is the one rooted at $t_3$, which happens to be one of the two children of test $t_2$. The error estimate for $t_2$ is calculated as the weighted average of $E_3$ and the error estimate for the other child of $t_2$ (the leaf labeled with the positive class). The resulting estimate would then be combined with $E_5$ as shown above.

**Post-pruning** The term refers to the circumstance that the decision tree is pruned *after* it has been fully induced from data (an alternative is the subject of the next subsection). We already know that the essence is to replace a subtree with a leaf

labeled with the class most frequent among the training examples reaching that leaf. Since there are usually several (or many) subtrees that can thus be replaced, a choice has to be made; and the existence of a choice means we need a criterion to guide our decision.

Here is one possibility. We know that pruning is likely to change the classifier's performance. One way to assess this change is to compare the error estimate of the decision tree after the pruning with that of the tree before the pruning:

$$D = E_{after} - E_{before} \tag{6.8}$$

From the available pruning alternatives, we choose the one where this difference is the smallest, $D_{min}$; but we carry out the pruning only if $D_{min} < c$, where $c$ is a user-set threshold for how much performance degradation can be tolerated in exchange for the tree's compactness. The mechanism is then repeated, with the decision tree becoming smaller and smaller, the stopping criterion being imposed by the constant $c$. Thus in Fig. 6.2, the first pruning step might have removed the subtree rooted at $t_3$; and the second step, the subtree rooted at $t_6$. Here the procedure was stopped because any further attempt at pruning resulted in a tree whose error estimate increased too much: the difference between the estimated error of the final (pruned) tree and that of the original tree on the left of Fig. 6.2 exceeded the user's threshold: $D > c$.

The principle is summarized by the pseudocode in Table 6.8.

**On-line Pruning** In the divide-and-conquer procedure, each subsequent attribute divides the set of training examples into smaller and smaller subsets. Inevitably, the evidence supporting the choice of the tests at lower tree-levels will be weak. When a tree node is reached by only, say, two training examples, one positive and one negative, a totally irrelevant attribute may by mere coincidence succeed in separating the positive example from the negative. The only "benefit" to be gained from adding this test to the decision tree is training-set overfitting.

The motivation behind *on-line* pruning is to make sure this situation is prevented. Here is the rule: if the training subset is smaller than a user-set minimum, $m$, stop further expansion of the tree.

**Impact of Pruning** How far the pruning goes is controlled by two parameters: $c$ in post-pruning, and $m$ in on-line pruning. In both cases, higher values result in smaller trees.

**Table 6.8** The algorithm for decision-tree pruning

$c$ . . . a user-set constant

(1) Estimate the error rate of the original decision tree. Let its value be denoted by $E_{before}$.
(2) Estimate the error rates of the trees obtained by alternative ways of pruning the original tree.
(3) Choose the pruning after which the estimated error rate experiences minimum increase, $D_{min} = E_{before} - E_{after}$, but only if $D_{min} < c$.
(4) Repeat steps (2) and (3) as long as $E_{before} - E_{after} < c$.

The main reason why pruning tends to improve classification performance on future examples is that the removal of low-level tests, which have poor statistical support, usually reduces the danger of overfitting. This, however, works only up to a certain point. If overdone, a very high extent of pruning can (in the extreme) result in the decision being replaced with a single leaf labeled with the majority class. Such classifier is unlikely to be useful.

Figure 6.3 shows the effect that gradually increased pruning typically has on classification performance. Along the horizontal axis is plotted the extent of pruning as controlled by *c* or *m* or both. The vertical axis represents the error rate measured on the training set as well as on some hypothetical testing set (the latter consisting of examples that have *not* been used for learning, but whose class labels are known).

On the training-set curve, error rate is minimized when there is no pruning at all. More interesting, however, is the testing-set curve. Its shape is telling us that the unpruned tree usually scores poorly on testing data, which is explained by the unpruned tree's tendency to overfit the training set, a phenomenon that can be reduced by increasing the extent of pruning. Excessive pruning, however, will remove attribute tests that *do* carry useful information, and this will have a detrimental effect on classification performance.

By the way, the two curves can tell us a lot about the underlying data. In some applications, even very modest pruning will impair error rate on testing data; for instance, in a noise-free domain with a relatively small training set.

Another thing to notice is that the error rate on the testing set is almost always greater than the error rate on the training set.

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What are the potential benefits of decision-tree pruning?
- How can we estimate the tree's error rate on future data? Write down the formula and explain how it is used.

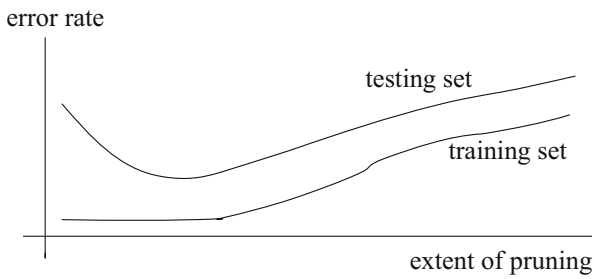



**Fig. 6.3** With the growing extent of pruning, error rate on the testing set usually drops, then starts growing again. Error rate on the training set usually increases monotonically

- Describe the principle of post-pruning and the principle of on-line pruning.
- What parameters control the extent of pruning? How do they affect the error rate on the training set, and how do they affect the error rate on the testing set?

## 6.6 Converting the Decision Tree into Rules

One of the advantages of decision trees in comparison with the other classifiers is their interpretability. Any sequence of tests along the path from the root to a leaf represents an *if-then* rule, and this rule explains why the classifier has labeled a given example with this or that class.

**Rules Generated by a Decision Tree** The reader will find it easy to convert a decision tree to a set of rules. It is enough to notice that a leaf is reached through a series of edges whose specific choice is determined by the results of the attribute tests encountered along the way. Each leaf is thus associated with a concrete conjunction of test results.

For the sake of illustration, let us write down the complete set of rules for the `pos` class as obtained from the decision tree in Fig. 6.1a.

*if* `crust-size=big AND filling-size=big` *then* `pos`
*if* `crust-size=big AND filling-size=small AND shape=circle` *then* `pos`
*if* `crust-size=small AND shape=circle` *then* `pos`
*if* `crust-size=small AND (shape=square OR triangle) AND filling-size=big` *then* `pos`
*else* `neg`

Note the *default class*, `neg`, in the last line. An example is labeled with the default class if all rules fail, which means that the value of the *if*-part of each rule is *false*. We notice that, in this two-class domain, we need to consider only the rules resulting in the `pos` class, the other class being the default option. We could have done it the other way round, considering only the rules for the `neg` class, and making `pos` the default class. This would actually be more economical because there are only two leafs labeled with the `neg`, and therefore only two corresponding rules. The reader is encouraged to write down these two rules by way of a simple exercise.

At any rate, the lesson is clear: in a domain with $K$ classes, only the rules for $K-1$ classes are needed, the last class serving as the default.

**Pruning the Rules** The tree post-pruning mechanism described earlier replaced a subtree with a leaf. This means that lower-level tests were the first to go, the technique being unable to remove a higher-level node before the removal of the nodes below it. The situation is similar in on-line pruning.

Once the tree has been converted to rules, however, pruning gains in flexibility: *any* test in the *if*-part of *any* rule is a potential candidate for removal; and entire

**Table 6.9** The algorithm for rule pruning

Re-write the decision tree as a set of rules.

Let $c$ be a user-set constant controlling the extent of pruning

(1) In each rule, calculate the increase in error estimate brought about by the removal of individual tests.
(2) Choose those removals where this increase, $D_{min}$ is smallest. Remove the tests, but only if $D_{min} < c$.
(3) In the set of rules, search for the weakest rules to be removed.
(4) Choose the default class.
(5) Order the rules according to their strengths (how many training examples they cover).

**Table 6.10** Illustration of the algorithm for rule pruning

Suppose that the decision from the left part of Fig. 6.2 has been converted into the following set of rules (**neg** is the default label to be used when the if-parts of all rules are *false*).

$t_1 \wedge t_2 \rightarrow$ pos
$t_1 \wedge \neg t_2 \wedge t_3 \wedge t_4 \rightarrow$ pos
$\neg t_1 \wedge t_5 \wedge t_6 \rightarrow$ pos
*else* neg

Suppose that the evaluation of the tests in the rules has resulted in the conclusion that $t_3$ in the second rule and $t_5$ in the third rule can be removed without a major increase in the error estimate. We obtain the following set of modified rules.

$t_1 \wedge t_2 \rightarrow$ pos
$t_1 \wedge \neg t_2 \wedge t_4 \rightarrow$ pos
$\neg t_1 \wedge t_6 \rightarrow$ pos
*else* neg

The next step can reveal that the second (already modified) rule can be removed without a major increase in the error estimate. After the removal, the set of rules will look as follows.

$t_1 \wedge t_2 \rightarrow$ pos
$\neg t_1 \wedge t_6 \rightarrow$ pos
*else* neg

This completes the pruning.

rules can be deleted. This is done by the rule-pruning algorithm summarized by the pseudocode in Table 6.9 and illustrated by the example in Table 6.10. Here, the initial set of rules was obtained from the (now familiar) tree in the left part of Fig. 6.2. The first pruning step removes those tests that do not appear to contribute much to the overall classification performance; the next step deletes the weakest rules.

We haste to admit, however, that the price for this added flexibility is a significant increase in computational costs.

## What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the mechanism that converts a decision tree to a set of rules. How many rules are thus obtained? What is the motivation behind such conversion?
- What is meant by the term, *default class*? How would you choose it?
- Discuss the possibilities of rule-pruning. In what sense can we claim that rule-pruning offers more flexibility than decision-tree pruning? What is the price for this increased flexibility?

## 6.7 Summary and Historical Remarks

- In decision trees, the attribute values are tested one at a time, the result of each test indicating what should happen next: either another attribute test, or a decision about the class label if a leaf has been reached. One can say that a decision tree consists of a set of partially ordered set of tests, each sequence of tests defining one branch in the tree terminated by a leaf.
- From a typical training set, many alternative decision trees can be created. As a rule, smaller trees are to be preferred, their main advantages being interpretability, removal of irrelevant and redundant attributes, and lower danger of overfitting noisy training data.
- The most typical procedure for induction of decision trees from data proceeds in a recursive manner, always seeking to identify the attribute that conveys maximum information about the class label. This approach tends to make the induced decision trees smaller. The "best" attribute is identified by simple formulas borrowed from information theory.
- An important aspect of decision-tree induction is pruning. The main motivation is to make sure that all tree branches are supported by sufficient evidence. Further on, pruning reduces the tree size which has certain advantages (see above). Two generic types of pruning exist. (1) In post-pruning, the tree is first fully developed, and then pruned. (2) In on-line pruning (which is perhaps a bit of a misnomer), the development of the tree is stopped once the training subsets used to determine the next attribute test become too small. In both cases, the extent of pruning is controlled by user-set parameters (denoted $c$ and $m$, respectively).
- A decision tree can be converted to a set of rules that can further be pruned. In a domain with $K$ classes, it is enough to specify the rules for $K - 1$ classes,

the remaining class becoming the *default class*. The rules are usually easier to interpret. Rule-pruning algorithms sometimes lead to more compact classifiers, though at significantly increased computational costs.

**Historical Remarks** The idea behind decision trees was first put forward by Hoveland and Hund in the late 1950s. The work was later summarized in the book Hunt et al. [39] that reports experience with several implementations of their Concept Learning System (CLS). Friedman et al. [30] developed a similar approach independently. An early high point of the research was reached by Breiman et al. [11] where the system CART is described. The idea was then imported to the machine-learning world by Quinlan [75, 76]. Perhaps the most famous implementation is C4.5 from Quinlan [78]. This chapter was based on a simplified version of C4.5.

## 6.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

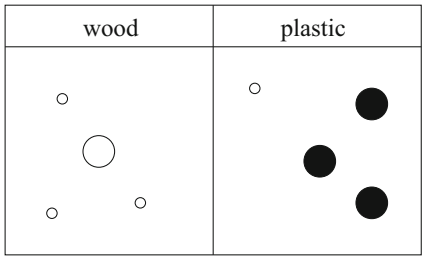



**Fig. 6.4** Another example with wooden and plastic circles

## Exercises

1. In Fig. 6.4, eight training examples are described by two attributes, `size` and `color`, the class label being the material: either `wood` or `plastic`.
   - What is the entropy of the training set when only the class labels are considered (ignoring the attribute values)?
   - Using the information-based mechanism from Sect. 6.3, decide which of the two attributes better predicts the class

2. Take the decision tree from Fig. 6.1a and remove from it the bottom-right test on `filling-size`. Based on the training set from Table 6.1, what will be the error estimate before and after this "pruning"?
3. Choose one of the decision trees in Fig. 6.1 and convert it to a set of rules. Pick one of these rules and decide which of its tests can be removed with the minimum increase in the estimated error.
4. Consider a set of ten training examples. Suppose there is a continuous attribute that has the following values: $3.6, 3.2, 1.2, 4.0, 0.8, 1.2, 2.8, 2.4, 2, 2, 1.0$. Suppose that the first five of these examples, and also the last one, are positive, all other examples being negative. What will be the best binary split of the range of this attribute's values?

## Give It Some Thought

1. The baseline performance criteria used for the evaluation of decision trees are error rate and the size of the tree (the number of nodes). These, however, may not be appropriate in certain domains. Suggest applications where either the size of the decision tree or its error rate may be less important. Hint: Consider the costs of erroneous decisions and the costs of obtaining attribute values.
2. What are likely to be the characteristics of a domain where a decision tree clearly outperforms the baseline 1-NN classifier? Hint: Consider such characteristics as noise, irrelevant attributes, or the size of the training set; and then make your own judgement as to what influence each of them is likely to have on the classifier's behavior.
3. On what kind of data may a linear classifier do better than a decision tree? Give at least two features characterizing such data. Rely on the same hint as the previous question.
4. Having found the answers to the previous two questions, you should be able to draw the logical conclusion: applying to the given data both decision-tree induction and linear-classifier induction, what will their performance betray about the characteristics of the data?
5. The decision tree as described in this chapter gives only "crisp" yes-or-no decisions about the given example's class (in this sense, one can argue that Bayesian classifiers or multilayer perceptrons are more flexible). By way of mitigating this weakness, suggest a mechanism that would modify the decision-trees framework so as to give, for each example, not only the class label, but also the classifier's confidence in this class label.

## Computer Assignments

1. Implement the baseline algorithm for the induction of decision trees and test its behavior on a few selected domains from the UCI repository.[4] Compare the results with those achieved by the $k$-NN classifier.
2. Implement the simple pruning mechanism described in this chapter. Choose a data file from the UCI repository. Run several experiments and observe how different extent of pruning affects the error rate on the training and testing sets.
3. Choose a sufficiently large domain from the UCI repository. Put aside 30% of the examples for testing. For training, use 10%, 20%, ... 70% of the remaining examples, respectively. Plot a graph where the horizontal axis gives the number of examples, and the vertical axis gives the computational time spent on the induction. Plot another graph where the vertical axis will give the error rate on the testing set. Discuss the obtained results.

[4] www.ics.uci.edu/~mlearn/MLRepository.html.