Another approach is to use the largest possible hypothesis space. For example, why not let **H** be the class of all Turing machines? After all, every computable function can be represented by some Turing machine, and that is the best we can do. The problem with this idea is that it does not take into account the *computational complexity* of learning. *There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding simple, consistent hypotheses within that space.* For example, fitting straight lines to data is very easy; fitting high-degree polynomials is harder; and fitting Turing machines is very hard indeed because determining whether a given Turing machine is consistent with the data is not even decidable in general. A second reason to prefer simple hypothesis spaces is that the resulting hypotheses may be simpler to use—that is, it is faster to compute $h(x)$ when h is a linear function than when it is an arbitrary Turing machine program.

For these reasons, most work on learning has focused on relatively simple representations. In this chapter, we concentrate on propositional logic and related languages. Chapter 19 looks at learning theories in first-order logic. We will see that the expressiveness–complexity tradeoff is not a simple as it first seems: it is often the case, as we saw in Chapter 8, that an expressive language makes it possible for a *simple* theory to fit the data, whereas restricting the expressiveness of the language means that any consistent theory must be very complex. For example, the rules of chess can be written in a page or two of first-order logic, but require thousands of pages when written in propositional logic. In such cases, it should be possible to learn much faster by using the more expressive language.

18.3 Learning Decision Trees

Decision tree induction is one of the simplest, and yet most successful forms of learning algorithm. It serves as a good introduction to the area of inductive learning, and is easy to implement. We first describe the performance element, and then show how to learn it. Along the way, we will introduce ideas that appear in all areas of inductive learning.

Decision trees as performance elements

DECISION TREE
ATTRIBUTES
CLASSIFICATION
REGRESSION
POSITIVE
NEGATIVE

A **decision tree** takes as input an object or situation described by a set of **attributes** and returns a "decision"—the predicted output value for the input. The input attributes can be discrete or continuous. For now, we assume discrete inputs. The output value can also be discrete or continuous; learning a discrete-valued function is called **classification** learning; learning a continuous function is called **regression**. We will concentrate on *Boolean* classification, wherein each example is classified as true (**positive**) or false (**negative**).

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached. The decision tree representation seems to be very natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

GOAL PREDICATE

A somewhat simpler example is provided by the problem of whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** $WillWait$. In setting this up as a learning problem, we first have to state what attributes are available to describe examples in the domain. In Chapter 19, we will see how to automate this task; for now, let's suppose we decide on the following list of attributes:

1. *Alternate*: whether there is a suitable alternative restaurant nearby.
2. *Bar*: whether the restaurant has a comfortable bar area to wait in.
3. *Fri/Sat*: true on Fridays and Saturdays.
4. *Hungry*: whether we are hungry.
5. *Patrons*: how many people are in the restaurant (values are *None*, *Some*, and *Full*).
6. *Price*: the restaurant's price range ($, $$, $$$).
7. *Raining*: whether it is raining outside.
8. *Reservation*: whether we made a reservation.
9. *Type*: the kind of restaurant (French, Italian, Thai, or burger).
10. *WaitEstimate*: the wait estimated by the host (0–10 minutes, 10–30, 30–60, >60).

The decision tree usually used by one of us (SR) for this domain is shown in Figure 18.2. Notice that the tree does not use the *Price* and *Type* attributes, in effect considering them to be irrelevant. Examples are processed by the tree starting at the root and following the appropriate branch until a leaf is reached. For instance, an example with $Patrons = Full$ and $WaitEstimate = 0$–10 will be classified as positive (i.e., yes, we will wait for a table).

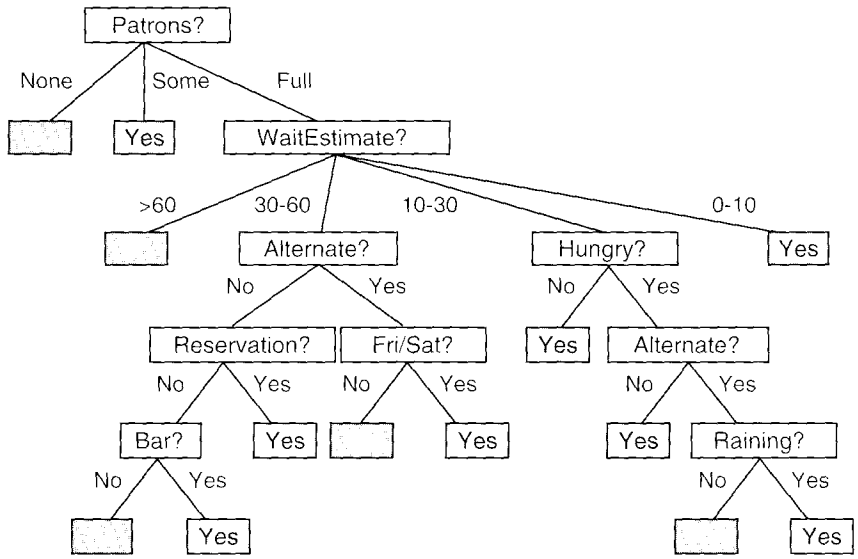

Figure 18.2 A decision tree for deciding whether to wait for a table.

Expressiveness of decision trees

Logically speaking, any particular decision tree hypothesis for the $WillWait$ goal predicate can be seen as an assertion of the form

$$\forall s \;\; WillWait(s) \;\Leftrightarrow\; (P_1(s) \vee P_2(s) \vee \cdots \vee P_n(s)) \;,$$

where each condition $P_i(s)$ is a conjunction of tests corresponding to a path from the root of the tree to a leaf with a positive outcome. Although this looks like a first-order sentence, it is, in a sense, propositional, because it contains just one variable and all the predicates are unary. The decision tree is really describing a relationship between $WillWait$ and some logical combination of attribute values. We cannot use decision trees to represent tests that refer to two or more different objects—for example,

$$\exists r_2 \;\; Nearby(r_2, r) \wedge Price(r, p) \wedge Price(r_2, p_2) \wedge Cheaper(p_2, p)$$

(is there a cheaper restaurant nearby?). Obviously, we could add another Boolean attribute with the name $CheaperRestaurantNearby$, but it is intractable to add *all* such attributes. Chapter 19 will delve further into the problem of learning in first-order logic proper.

Decision trees *are* fully expressive within the class of propositional languages; that is, any Boolean function can be written as a decision tree. This can be done trivially by having each row in the truth table for the function correspond to a path in the tree. This would yield an exponentially large decision tree representation because the truth table has exponentially many rows. Clearly, decision trees can represent many functions with much smaller trees.

PARITY FUNCTION

For some kinds of functions, however, this is a real problem. For example, if the function is the **parity function**, which returns 1 if and only if an even number of inputs are 1, then an exponentially large decision tree will be needed. It is also difficult to use a decision tree to represent a **majority function**, which returns 1 if more than half of its inputs are 1.

MAJORITY FUNCTION

In other words, decision trees are good for some kinds of functions and bad for others. Is there *any* kind of representation that is efficient for *all* kinds of functions? Unfortunately, the answer is no. We can show this in a very general way. Consider the set of all Boolean functions on n attributes. How many different functions are in this set? This is just the number of different truth tables that we can write down, because the function is defined by its truth table. The truth table has 2^n rows, because each input case is described by n attributes. We can consider the "answer" column of the table as a 2^n-bit number that defines the function. No matter what representation we use for functions, some of the functions (almost all of them, in fact) are going to require at least that many bits to represent.

If it takes 2^n bits to define the function, then there are 2^{2^n} different functions on n attributes. This is a scary number. For example, with just six Boolean attributes, there are $2^{2^6} = 18,446,744,073,709,551,616$ different functions to choose from. We will need some ingenious algorithms to find consistent hypotheses in such a large space.

Inducing decision trees from examples

An example for a Boolean decision tree consists of a vector of input attributes, X, and a single Boolean output value y. A set of examples $(X_1, y_1) \ldots (X_{12}, y_{12})$ is shown in Figure 18.3.

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	$$$	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	$	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	$	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	$	*Yes*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	$$$	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	$$	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	$	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	$$	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	$	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	$$$	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	$	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	$	*No*	*No*	*Burger*	*30–60*	*Yes*

Figure 18.3 Examples for the restaurant domain.

The positive examples are the ones in which the goal *WillWait* is true ($X_1, X_3, \ldots$); the negative examples are the ones in which it is false ($X_2, X_5, \ldots$). The complete set of examples is called the **training set**.

TRAINING SET

The problem of finding a decision tree that agrees with the training set might seem difficult, but in fact there is a trivial solution. We could simply construct a decision tree that has one path to a leaf for each example, where the path tests each attribute in turn and follows the value for the example and the leaf has the classification of the example. When given the same example again,[3] the decision tree will come up with the right classification. Unfortunately, it will not have much to say about any other cases!

The problem with this trivial tree is that it just memorizes the observations. It does not extract any pattern from the examples, so we cannot expect it to be able to extrapolate to examples it has not seen. Applying Ockham's razor, we should find instead the *smallest* decision tree that is consistent with the examples. Unfortunately, for any reasonable definition of "smallest," finding the smallest tree is an intractable problem. With some simple heuristics, however, we can do a good job of finding a "smallish" one. The basic idea behind the DECISION-TREE-LEARNING algorithm is to test the most important attribute first. By "most important," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.

Figure 18.4 shows how the algorithm gets started. We are given 12 training examples, which we classify into positive and negative sets. We then decide which attribute to use as the first test in the tree. Figure 18.4(a) shows that *Type* is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive and negative examples. On the other hand, in Figure 18.4(b) we see that *Patrons* is a fairly important

[3] The same example *or an example with the same description*—this distinction is very important, and we will return to it in Chapter 19.

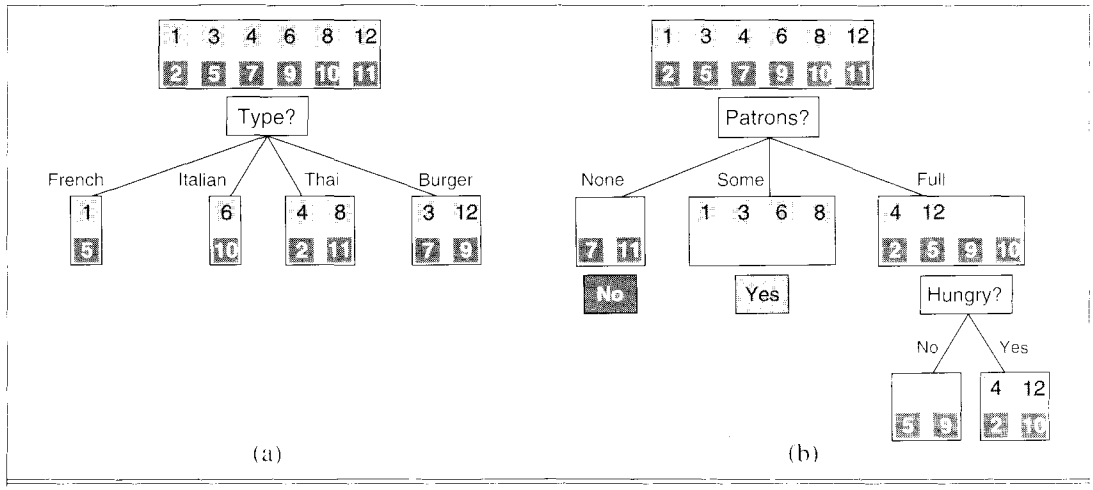

Figure 18.4 Splitting the examples by testing on attributes. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

attribute, because if the value is *None* or *Some*, then we are left with example sets for which we can answer definitively (*No* and *Yes*, respectively). If the value is *Full*, we are left with a mixed set of examples. In general, after the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one fewer attribute. There are four cases to consider for these recursive problems:

1. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 18.4(b) shows *Hungry* being used to split the remaining examples.
2. If all the remaining examples are positive (or all negative), then we are done: we can answer *Yes* or *No*. Figure 18.4(c) shows examples of this in the *None* and *Some* cases.
3. If there are no examples left, it means that no such example has been observed, and we return a default value calculated from the majority classification at the node's parent.
4. If there are no attributes left, but both positive and negative examples, we have a problem. It means that these examples have exactly the same description, but different classifications. This happens when some of the data are incorrect; we say there is **noise** in the data. It also happens either when the attributes do not give enough information to describe the situation fully, or when the domain is truly nondeterministic. One simple way out of the problem is to use a majority vote.

NOISE

The DECISION-TREE-LEARNING algorithm is shown in Figure 18.5. The details of the method for CHOOSE-ATTRIBUTE are given in the next subsection.

The final tree produced by the algorithm applied to the 12-example data set is shown in Figure 18.6. The tree is clearly different from the original tree shown in Figure 18.2, despite the fact that the data were actually generated from an agent using the original tree. One might conclude that the learning algorithm is not doing a very good job of learning the correct

```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
  inputs: examples, set of examples
          attribs, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attribs is empty then return MAJORITY-VALUE(examples)
  else
      best ← CHOOSE-ATTRIBUTE(attribs, examples)
      tree ← a new decision tree with root test best
      m ← MAJORITY-VALUE(examples_i)
      for each value v_i of best do
          examples_i ← {elements of examples with best = v_i}
          subtree ← DECISION-TREE-LEARNING(examples_i, attribs − best, m)
          add a branch to tree with label v_i and subtree subtree
      return tree
```

Figure 18.5 The decision tree learning algorithm.

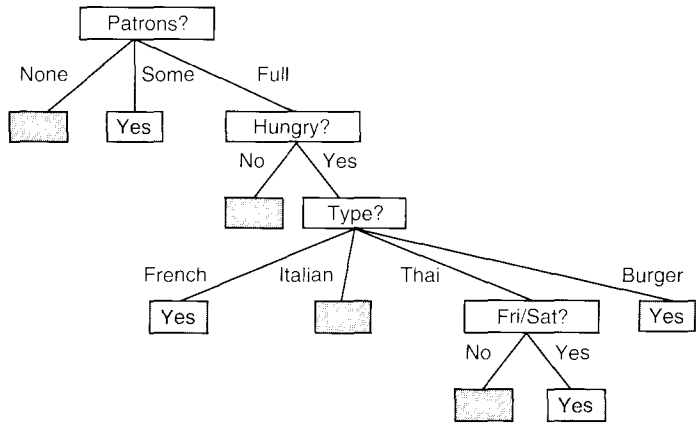

Figure 18.6 The decision tree induced from the 12-example training set.

function. This would be the wrong conclusion to draw, however. The learning algorithm looks at the *examples*, not at the correct function, and in fact, its hypothesis (see Figure 18.6) not only agrees with all the examples, but is considerably simpler than the original tree. The learning algorithm has no reason to include tests for *Raining* and *Reservation*, because it can classify all the examples without them. It has also detected an interesting and previously unsuspected pattern: the first author will wait for Thai food on weekends.

Of course, if we were to gather more examples, we might induce a tree more similar to the original. The tree in Figure 18.6 is bound to make a mistake; for example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full. For a case where

Hungry is false, the tree says not to wait, but I (SR) would certainly wait. This raises an obvious question: if the algorithm induces a consistent, but incorrect, tree from the examples, how incorrect will the tree be? We will show how to analyze this question experimentally, after we explain the details of the attribute selection step.

Choosing attribute tests

The scheme used in decision tree learning for selecting attributes is designed to minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect attribute divides the examples into sets that are all positive or all negative. The *Patrons* attribute is not perfect, but it is fairly good. A really useless attribute, such as *Type*, leaves the example sets with roughly the same proportion of positive and negative examples as the original set.

All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the CHOOSE-ATTRIBUTE function of Figure 18.5. The measure should have its maximum value when the attribute is perfect and its minimum value when the attribute is of no use at all. One suitable measure is the expected amount of **information** provided by the attribute, where we use the term in the mathematical sense first defined in Shannon and Weaver (1949). To understand the notion of information, think about it as providing the answer to a question—for example, whether a coin will come up heads. The amount of information contained in the answer depends on one's prior knowledge. The less you know, the more information is provided. Information theory measures information content in **bits**. One bit of information is enough to answer a yes/no question about which one has no idea, such as the flip of a fair coin. In general, if the possible answers v_i have probabilities $P(v_i)$, then the information content I of the actual answer is given by

INFORMATION

$$I(P(v_1), \ldots, P(v_n)) = \sum_{i=1}^{n} -P(v_i) \log_2 P(v_i) .$$

To check this equation, for the tossing of a fair coin, we get

$$I\left(\tfrac{1}{2}, \tfrac{1}{2}\right) = -\tfrac{1}{2}\log_2 \tfrac{1}{2} - \tfrac{1}{2}\log_2 \tfrac{1}{2} = 1 \text{ bit.}$$

If the coin is loaded to give 99% heads, we get $I(1/100, 99/100) = 0.08$ bits, and as the probability of heads goes to 1, the information of the actual answer goes to 0.

For decision tree learning, the question that needs answering is: for a given example, what is the correct classification? A correct decision tree will answer this question. An estimate of the probabilities of the possible answers before any of the attributes have been tested is given by the proportions of positive and negative examples in the training set. Suppose the training set contains p positive examples and n negative examples. Then an estimate of the information contained in a correct answer is

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2 \frac{p}{p+n} - \frac{n}{p+n}\log_2 \frac{n}{p+n} .$$

The restaurant training set in Figure 18.3 has $p = n = 6$, so we need 1 bit of information.

Now a test on a single attribute A will not usually tell us this much information, but it will give us some of it. We can measure exactly how much by looking at how much

information we still need *after* the attribute test. Any attribute A divides the training set E into subsets $E_1, \ldots, E_v$ according to their values for A, where A can have v distinct values. Each subset E_i has p_i positive examples and n_i negative examples, so if we go along that branch, we will need an additional $I(p_i/(p_i+n_i), n_i/(p_i+n_i))$ bits of information to answer the question. A randomly chosen example from the training set has the ith value for the attribute with probability $(p_i+n_i)/(p+n)$, so on average, after testing attribute A, we will need

$$Remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

INFORMATION GAIN

bits of information to classify the example. The **information gain** from the attribute test is the difference between the original information requirement and the new requirement:

$$Gain(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - Remainder(A) .$$

The heuristic used in the CHOOSE-ATTRIBUTE function is just to choose the attribute with the largest gain. Returning to the attributes considered in Figure 18.4, we have

$$Gain(Patrons) = 1 - \left[\frac{2}{12}I(0,1) + \frac{4}{12}I(1,0) + \frac{6}{12}I\left(\frac{2}{6}, \frac{4}{6}\right)\right] \approx 0.541 \text{ bits},$$

$$Gain(Type) = 1 - \left[\frac{2}{12}I\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{2}{12}I\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{4}{12}I\left(\frac{2}{4}, \frac{2}{4}\right) + \frac{4}{12}I\left(\frac{2}{4}, \frac{2}{4}\right)\right] = 0,$$

confirming our intuition that *Patrons* is a better attribute to split on. In fact, *Patrons* has the highest gain of any of the attributes and would be chosen by the decision-tree learning algorithm as the root.

Assessing the performance of the learning algorithm

A learning algorithm is good if it produces hypotheses that do a good job of predicting the classifications of unseen examples. In Section 18.5, we will see how prediction quality can be estimated in advance. For now, we will look at a methodology for assessing prediction quality after the fact.

Obviously, a prediction is good if it turns out to be true, so we can assess the quality of a hypothesis by checking its predictions against the correct classification once we know it. We

TEST SET

do this on a set of examples known as the **test set**. If we train on all our available examples, then we will have to go out and get some more to test on, so often it is more convenient to adopt the following methodology:

1. Collect a large set of examples.
2. Divide it into two disjoint sets: the **training set** and the **test set**.
3. Apply the learning algorithm to the training set, generating a hypothesis h.
4. Measure the percentage of examples in the test set that are correctly classified by h.
5. Repeat steps 1 to 4 for different sizes of training sets and different randomly selected training sets of each size.

The result of this procedure is a set of data that can be processed to give the average prediction quality as a function of the size of the training set. This function can be plotted on a graph,

LEARNING CURVE

giving what is called the **learning curve** for the algorithm on the particular domain. The

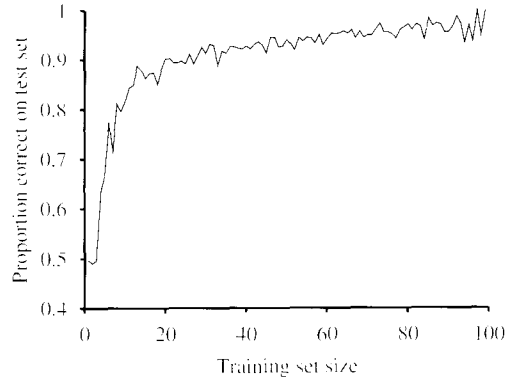

Figure 18.7 A learning curve for the decision tree algorithm on 100 randomly generated examples in the restaurant domain. The graph summarizes 20 trials.

learning curve for DECISION-TREE-LEARNING with the restaurant examples is shown in Figure 18.7. Notice that, as the training set grows, the prediction quality increases. (For this reason, such curves are also called **happy graphs**.) This is a good sign that there is indeed some pattern in the data and the learning algorithm is picking it up.

Obviously, the learning algorithm must not be allowed to "see" the test data before the learned hypothesis is tested on them. Unfortunately, it is all too easy to fall into the trap of **peeking** at the test data. Peeking typically happens as follows: A learning algorithm can have various "knobs" that can be twiddled to tune its behavior—for example, various different criteria for choosing the next attribute in decision tree learning. We generate hypotheses for various different settings of the knobs, measure their performance on the test set, and report the prediction performance of the best hypothesis. Alas, peeking has occurred! The reason is that the hypothesis was selected *on the basis of its test set performance*, so information about the test set has leaked into the learning algorithm. The moral of this tale is that any process that involves comparing the performance of hypotheses on a test set must use a *new* test set to measure the performance of the hypothesis that is finally selected. In practice, this is too difficult, so people continue to run experiments on tainted sets of examples.

PEEKING

Noise and overfitting

We saw earlier that if there are two or more examples with the same description (in terms of the attributes) but different classifications, then the DECISION-TREE-LEARNING algorithm must fail to find a decision tree consistent with all the examples. The solution we mentioned before is to have each leaf node report either the majority classification for its set of examples, if a deterministic hypothesis is required, or report the estimated probabilities of each classification using the relative frequencies. Unfortunately, this is far from the whole story. It is quite possible, and in fact likely, that even when vital information is missing, the decision tree learning algorithm will find a decision tree that is consistent with all the examples. This is because the algorithm can use the *irrelevant* attributes, if any, to make spurious distinctions among the examples.

Consider the problem of trying to predict the roll of a die. Suppose that experiments are carried out during an extended period of time with various dice and that the attributes describing each training example are as follows:

1. *Day*: the day on which the die was rolled (Mon, Tue, Wed, Thu).
2. *Month*: the month in which the die was rolled (Jan or Feb).
3. *Color*: the color of the die (Red or Blue).

As long as no two examples have identical descriptions, DECISION-TREE-LEARNING will find an exact hypothesis. The more attributes there are, the more likely it is that an exact hypothesis will be found. Any such hypothesis will be totally spurious. What we would like is that DECISION-TREE-LEARNING return a single leaf node with probabilities close to 1/6 for each roll, once it has seen enough examples.

Whenever there is a large set of possible hypotheses, one has to be careful not to use the resulting freedom to find meaningless "regularity" in the data. This problem is called **overfitting**. A very general phenomenon, overfitting occurs even when the target function is not at all random. It afflicts every kind of learning algorithm, not just decision trees.

OVERFITTING

A complete mathematical treatment of overfitting is beyond the scope of this book. Here we present a simple technique called **decision tree pruning** to deal with the problem. Pruning works by preventing recursive splitting on attributes that are not clearly relevant, even when the data at that node in the tree are not uniformly classified. The question is, how do we detect an irrelevant attribute?

DECISION TREE PRUNING

Suppose we split a set of examples using an irrelevant attribute. Generally speaking, we would expect the resulting subsets to have roughly the same proportions of each class as the original set. In this case, the information gain will be close to zero.[4] Thus, the information gain is a good clue to irrelevance. Now the question is, how large a gain should we require in order to split on a particular attribute?

We can answer this question by using a statistical **significance test**. Such a test begins by assuming that there is no underlying pattern (the so-called **null hypothesis**). Then the actual data are analyzed to calculate the extent to which they deviate from a perfect absence of pattern. If the degree of deviation is statistically unlikely (usually taken to mean a 5% probability or less), then that is considered to be good evidence for the presence of a significant pattern in the data. The probabilities are calculated from standard distributions of the amount of deviation one would expect to see in random sampling.

SIGNIFICANCE TEST

NULL HYPOTHESIS

In this case, the null hypothesis is that the attribute is irrelevant and, hence, that the information gain for an infinitely large sample would be zero. We need to calculate the probability that, under the null hypothesis, a sample of size n would exhibit the observed deviation from the expected distribution of positive and negative examples. We can measure the deviation by comparing the actual numbers of positive and negative examples in each subset, p_i and n_i, with the expected numbers, $\hat{p}_i$ and $\hat{n}_i$, assuming true irrelevance:

$$\hat{p}_i = p \times \frac{p_i + n_i}{p + n} \qquad \hat{n}_i = n \times \frac{p_i + n_i}{p + n} .$$

[4] In fact, the gain will be positive unless the proportions are all exactly the same. (See Exercise 18.10.)

A convenient measure of the total deviation is given by

$$D = \sum_{i=1}^{v} \frac{(p_i - \hat{p}_i)^2}{\hat{p}_i} + \frac{(n_i - \hat{n}_i)^2}{\hat{n}_i} .$$

Under the null hypothesis, the value of D is distributed according to the χ^2 (chi-squared) distribution with $v - 1$ degrees of freedom. The probability that the attribute is really irrelevant can be calculated with the help of standard χ^2 tables or with statistical software. Exercise 18.11 asks you to make the appropriate changes to DECISION-TREE-LEARNING to implement this form of pruning, which is known as χ^2 **pruning**.

χ^2 PRUNING

With pruning, noise can be tolerated: classification errors give a linear increase in prediction error, whereas errors in the descriptions of examples have an asymptotic effect that gets worse as the tree shrinks down to smaller sets. Trees constructed with pruning perform significantly better than trees constructed without pruning when the data contain a large amount of noise. The pruned trees are often much smaller and hence easier to understand.

CROSS-VALIDATION

Cross-validation is another technique that reduces overfitting. It can be applied to any learning algorithm, not just decision tree learning. The basic idea is to estimate how well each hypothesis will predict unseen data. This is done by setting aside some fraction of the known data and using it to test the prediction performance of a hypothesis induced from the remaining data. K-fold cross-validation means that you run k experiments, each time setting aside a different $1/k$ of the data to test on, and average the results. Popular values for k are 5 and 10. The extreme is $k = n$, also known as leave-one-out cross-validation. Cross-validation can be used in conjunction with any tree-construction method (including pruning) in order to select a tree with good prediction performance. To avoid peeking, we must then measure this performance with a new test set.

Broadening the applicability of decision trees

In order to extend decision tree induction to a wider variety of problems, a number of issues must be addressed. We will briefly mention each, suggesting that a full understanding is best obtained by doing the associated exercises:

- **Missing data**: In many domains, not all the attribute values will be known for every example. The values might have gone unrecorded, or they might be too expensive to obtain. This gives rise to two problems: First, given a complete decision tree, how should one classify an object that is missing one of the test attributes? Second, how should one modify the information gain formula when some examples have unknown values for the attribute? These questions are addressed in Exercise 18.12.
- **Multivalued attributes**:When an attribute has many possible values, the information gain measure gives an inappropriate indication of the attribute's usefulness. In the extreme case, we could use an attribute, such as $RestaurantName$, that has a different value for every example. Then each subset of examples would be a singleton with a unique classification, so the information gain measure would have its highest value for this attribute. Nonetheless, the attribute could be irrelevant or useless. One solution is to use the **gain ratio** (Exercise 18.13).

GAIN RATIO

♢ **Continuous and integer-valued input attributes**: Continuous or integer-valued attributes such as *Height* and *Weight*, have an infinite set of possible values. Rather than generate infinitely many branches, decision-tree learning algorithms typically find the **split point** that gives the highest information gain. For example, at a given node in the tree, it might be the case that testing on $Weight > 160$ gives the most information. Efficient dynamic programming methods exist for finding good split points, but it is still by far the most expensive part of real-world decision tree learning applications.

SPLIT POINT

♢ **Continuous-valued output attributes**: If we are trying to predict a numerical value, such as the price of a work of art, rather than a discrete classification, then we need a **regression tree**. Such a tree has at each leaf a linear function of some subset of numerical attributes, rather than a single value. For example, the branch for hand-colored engravings might end with a linear function of area, age, and number of colors. The learning algorithm must decide when to stop splitting and begin applying linear regression using the remaining attributes (or some subset thereof).

REGRESSION TREE

A decision-tree learning system for real-world applications must be able to handle all of these problems. Handling continuous-valued variables is especially important, because both physical and financial processes provide numerical data. Several commercial packages have been built that meet these criteria, and they have been used to develop several hundred fielded systems. In many areas of industry and commerce, decision trees are usually the first method tried when a classification method is to be extracted from a data set. One important property of decision trees is that it is possible for a human to understand the output of the learning algorithm. (Indeed, this is a *legal requirement* for financial decisions that are subject to anti-discrimination laws.) This is a property not shared by neural networks (see Chapter 20).

18.4 Ensemble Learning

ENSEMBLE LEARNING

So far we have looked at learning methods in which a single hypothesis, chosen from a hypothesis space, is used to make predictions. The idea of **ensemble learning** methods is to select a whole collection, or **ensemble**, of hypotheses from the hypothesis space and combine their predictions. For example, we might generate a hundred different decision trees from the same training set and have them vote on the best classification for a new example.

The motivation for ensemble learning is simple. Consider an ensemble of $M = 5$ hypotheses and suppose that we combine their predictions using simple majority voting. For the ensemble to misclassify a new example, *at least three of the five hypotheses have to misclassify it*. The hope is that this is much less likely than a misclassification by a single hypothesis. Suppose we assume that each hypothesis h_i in the ensemble has an error of p—that is, the probability that a randomly chosen example is misclassified by h_i is p. Furthermore, suppose we assume that the errors made by each hypothesis are *independent*. In that case, if p is small, then the probability of a large number of misclassifications occurring is minuscule. For example, a simple calculation (Exercise 18.14) shows that using an ensemble of five hypotheses reduces an error rate of 1 in 10 down to an error rate of less than 1 in 100. Now, obviously