

# 18 LEARNING FROM OBSERVATIONS

*In which we describe agents that can improve their behavior through diligent study of their own experiences.*

The idea behind learning is that percepts should be used not only for acting, but also for improving the agent's ability to act in the future. Learning takes place as a result of the interaction between the agent and the world, and from observation by the agent of its own decision-making processes. Learning can range from trivial memorization of experience, as exhibited by the wumpus agent in Chapter 10, to the creation of entire scientific theories, as exhibited by Albert Einstein. Chapter 2 explained the basic structure of a learning agent; this chapter describes **inductive learning** from observations, which is the basic tool used by learning agents. In particular, we describe how to learn simple theories in propositional logic. We also give a theoretical analysis that explains why inductive learning works.

## 18.1 FORMS OF LEARNING

---

In Chapter 2, we saw that a learning agent can be thought of as containing a **performance element** that decides what actions to take and a **learning element** that modifies the performance element so that it makes better decisions (see Figure 2.15). Machine learning researchers have come up with a large variety of learning elements. To understand them, it will help to see how their design is affected by the context in which they will operate. The design of the learning element is affected by three major issues:

- Which *components* of the performance element are to be learned.
- What *feedback* is available to learn these components.
- What *representation* is used for the components.

We now analyze each of these issues in turn.

We have seen that there are many ways to build the performance element of an agent. Chapter 2 described several agent designs (Figures 2.9, 2.11, 2.13, and 2.14). The components of these agents include the following:

1. A direct mapping from conditions on the current state to actions.

2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible actions the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions in particular states.
6. Goals that describe classes of states whose achievement maximizes the agent's utility.

Each of the components can be learned, given the appropriate feedback. Consider, for example, an agent training to become a taxi driver. Every time the instructor shouts “Brake!!!”, the agent can learn a condition–action rule for when to brake (1). By seeing many camera images that it is told contain buses, it can learn to recognize them (2). By trying actions and observing the results—for example, braking hard on a wet road—it can learn the effects of its actions (3). Then, when it receives no tip from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function (4).

Let us now consider the type of feedback available for learning. The first three cases in the preceding paragraph can be seen as learning a *function* given examples of the inputs and outputs of the function. The condition–action rule for braking is a function from states to a Boolean output (whether or not to brake); the bus recognition process is a function from images to a Boolean output (whether or not the image contains a bus); and the theory of braking is a function from states and braking actions to, say, stopping distances. These are all instances of **supervised learning**; this is the simplest form of learning, and it will be the subject of this chapter. Notice that in the first two cases, a teacher provided the correct output value of the examples; in the third, the output value was available directly from the agent's percepts. For fully observable environments, it will always be the case that an agent can observe the effects of its actions and hence can use supervised learning methods to learn to predict them. For partially observable environments, the problem is more difficult because the immediate effects may be invisible.

Ultimately, an agent must learn *what to do*. If there is no teacher to tell the agent the correct action at every moment, then the agent must learn what to do from more subtle kinds of feedback. For example, the lack of a tip at the end of the journey (or the hefty bill for rearend the car in front) is called a **reinforcement**.<sup>1</sup> Learning what to do based on such information is called **reinforcement learning**. The subject is covered in Chapter 20.

Learning when there is no hint at all about the correct outputs is called **unsupervised learning**. An unsupervised learner can always learn relationships among its percepts using supervised learning methods—that is, it can learn to predict its future percepts given its previous percepts. It cannot learn what to *do* unless it already has a utility function.

The representation of the learned information plays a very important role in determining how the learning algorithm must work. Any of the components of an agent can be represented using any of the representation schemes in this book. We have seen several examples: linear weighted polynomials for utility functions in game-playing programs; propositional and first-order logical sentences for all of the components in a logical agent; and probabilistic descriptions such as Bayesian networks inferential components of a decision-theoretic agent.

<sup>1</sup> The term **reward** as used in Chapter 17 is a synonym for **reinforcement**.

SUPERVISED  
LEARNING

REINFORCEMENT  
REINFORCEMENT  
LEARNING

UNSUPERVISED  
LEARNING

Effective learning algorithms have been devised for all of these. This chapter will cover methods for propositional logic; Chapter 21 describes methods for first-order logic; and Chapter 19 covers methods for Bayesian networks and for neural networks (which include linear polynomials as a special case).

PRIOR KNOWLEDGE

One other issue that cuts across all forms of learning is the question of **prior knowledge**. The majority of learning research in AI, computer science, and psychology has studied the case in which the agent begins with no knowledge at all about what it is trying to learn. It has access only to the examples presented by its experience. Although this is an important special case, it is by no means the general case. Most human learning takes place in the context of a good deal of background knowledge. Some psychologists and linguists claim that even newborn babies exhibit knowledge of the world. Whatever the truth of this claim, there is no doubt that prior knowledge can help enormously in learning. A physicist examining a stack of bubble-chamber photographs may be able to induce a theory positing the existence of a new particle of a certain mass and charge; but an art critic examining the same stack might learn nothing more than that the “artist” must be some sort of abstract expressionist. Chapter 21 shows several ways in which learning is helped by the use of existing knowledge; it also shows how knowledge can be *compiled* in order to speed up decision making. Chapter 19 shows how prior knowledge helps in the learning of probabilistic theories.

## 18.2 INDUCTIVE LEARNING

EXAMPLE

An algorithm for deterministic supervised learning is given as input the correct value of the unknown function for particular inputs, and must try to recover the unknown function or something close to it. More formally, we say an **example** is a pair  $(x, f(x))$ , where  $x$  is the input and  $f(x)$  is the output of the function applied to  $x$ . The task of **pure inductive inference** (or **induction**) is this:

PURE INDUCTIVE INFERENCE

Given a collection of examples of  $f$ , return a function  $h$  that approximates  $f$ .

HYPOTHESIS

The function  $h$  is called a **hypothesis**. The reason that learning is difficult, from a conceptual point of view, is that the function  $f$  is not given so it is not easy to tell whether any particular  $h$  is a good approximation of  $f$ . A hypothesis that is a good approximation of  $f$  will **generalize** well—that is, will predict unseen examples correctly. This is the fundamental **problem of induction**. The problem has been studied for centuries; Section 18.5 provides a partial solution.

GENERALIZATION PROBLEM OF INDUCTION

HYPOTHESIS SPACE

Figure 18.1 shows a familiar example: fitting a function of a single variable to some data points. The examples are  $(x, f(x))$  pairs where both  $x$  and  $f(x)$  are real numbers. We choose the **hypothesis space  $H$** —the set of hypotheses we will consider—to be the set of polynomials of degree at most  $k$ , such as  $3x^2 + 2$ ,  $x^{17} - 4x^3$ , and so on. Figure 18.1(a) shows some data with an exact fit by a straight line (a polynomial of degree 1). The line is called a **consistent** hypothesis because it agrees with all the data. Figure 18.1(b) shows a high-degree polynomial that is also consistent with the same data. This illustrates the first fundamental issue in inductive learning: *how do we choose among multiple consistent hy-*

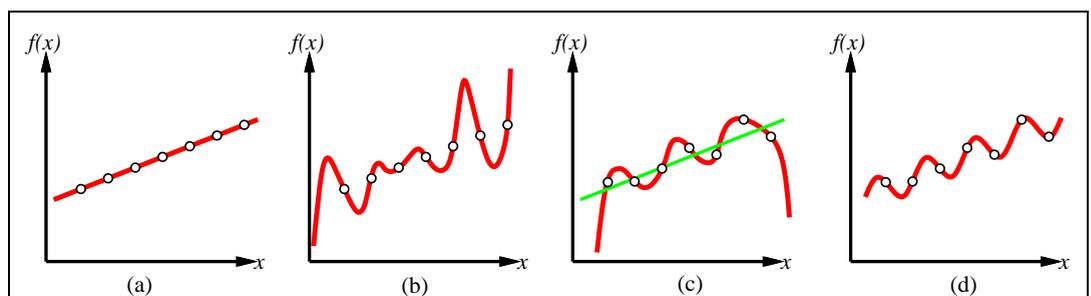
CONSISTENT



OCKHAM'S RAZOR

*potheses*? One answer is **Ockham's razor**<sup>2</sup>—prefer the *simplest* hypothesis consistent with the data. Intuitively, this makes sense, because hypotheses that are no simpler than the data themselves are failing to extract any *pattern* from the data. Of course, defining simplicity is not a simple problem, but it seems reasonable to say that a degree-1 polynomial is simpler than a degree-12 polynomial.

Figure 18.1(c) shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial (with 7 parameters) for an exact fit. There are just 7 data points, so the polynomial has as many parameters as there are data points; thus, it does not seem to be finding any pattern in the data and we do not expect it to generalize well. It might be better to fit a simple straight line that is not exactly consistent but may make reasonable predictions. This amounts to accepting that the true function may not be deterministic (or, roughly equivalently, that the true inputs are not fully observed). *For nondeterministic functions, there is an inevitable tradeoff between hypothesis complexity and degree of fit to the data.* Chapter 19 explains how to make this tradeoff using probability theory.



**Figure 18.1** (a) Example  $(x, f(x))$  pairs and a consistent, linear hypothesis. (b) A consistent, high-degree, polynomial hypothesis for the same data set. (c) A different data set that admits an exact high-degree polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

One should keep in mind that the possibility or impossibility of finding a simple, consistent hypothesis depends strongly on the hypothesis space chosen. Figure 18.1(d) shows that the data in (c) can be fit exactly by a simple function of the form  $ax + b + c \sin x$ . This example shows the importance of the choice of hypothesis space. A hypothesis space consisting of polynomials of finite degree cannot represent sinusoidal functions accurately, so a learner using this hypothesis space will not be able to learn from sinusoidal data. We say that a learning problem is **realizable** if the hypothesis space contains the true function; otherwise, it is **unrealizable**. Unfortunately, we cannot always tell if a given learning problem is realizable, because the true function is not known. One way to get around this is to use *prior knowledge* to derive a hypothesis space in which we know the true function must lie—this is covered in Chapter 21.

Another approach is to use the largest possible hypothesis space. For example, why not let  $\mathbf{H}$  be the class of all Turing machines? After all, every computable function can be

<sup>2</sup> Sometimes misspelled “Occam,” perhaps from the French rendering, “Guillaume d’Occam.”

REALIZABLE

UNREALIZABLE



represented by some Turing machine, and this 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; 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 languages is that the resulting hypotheses may be simpler to use—that is, inference may be much less expensive.

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 21 looks at learning theories in first-order logic. We will see that the expressiveness/complexity tradeoff is not as 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 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 may be discrete or continuous. For the most part we will assume discrete inputs. The output value can also be discrete or continuous; learning a discrete-valued function is called **classification learning**; the continuous case, which we defer to Chapter 19, is called **regression**. We will concentrate on *Boolean* classification, where 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 labelled 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*.<sup>3</sup> In

setting this up as a learning problem, we first have to decide what attributes are available to describe examples in the domain.<sup>4</sup> 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 the first author for this domain is shown in Figure 18.2. Notice that the tree does not use the *Price* and *Type* attributes, considering these to be irrelevant given the data it has seen. 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.

### 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

$$\forall s \text{ WillWait}(s) \Leftrightarrow (P_1(s) \vee P_2(s) \vee \dots \vee P_n(s))$$

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 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 predicates. 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,

$$\exists r_2 \text{ Nearby}(r_2, r) \wedge \text{Price}(r, p) \wedge \text{Price}(r_2, p_2) \wedge \text{Cheaper}(p_2, p)$$

(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 21 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

<sup>3</sup> Meanwhile, the automated taxi is learning whether to wait for the passengers in case they give up waiting for a table and want to go on to another restaurant.

<sup>4</sup> One might ask why this isn't the job of the learning program. In fact, it is, but we will not be able to explain how it is done until Chapter 21.

each row in the truth table for the function correspond to a path in the tree. This would not necessarily be a good way to represent the function, because the truth table is exponentially large in the number of attributes. Clearly, decision trees can represent many functions with much smaller trees.

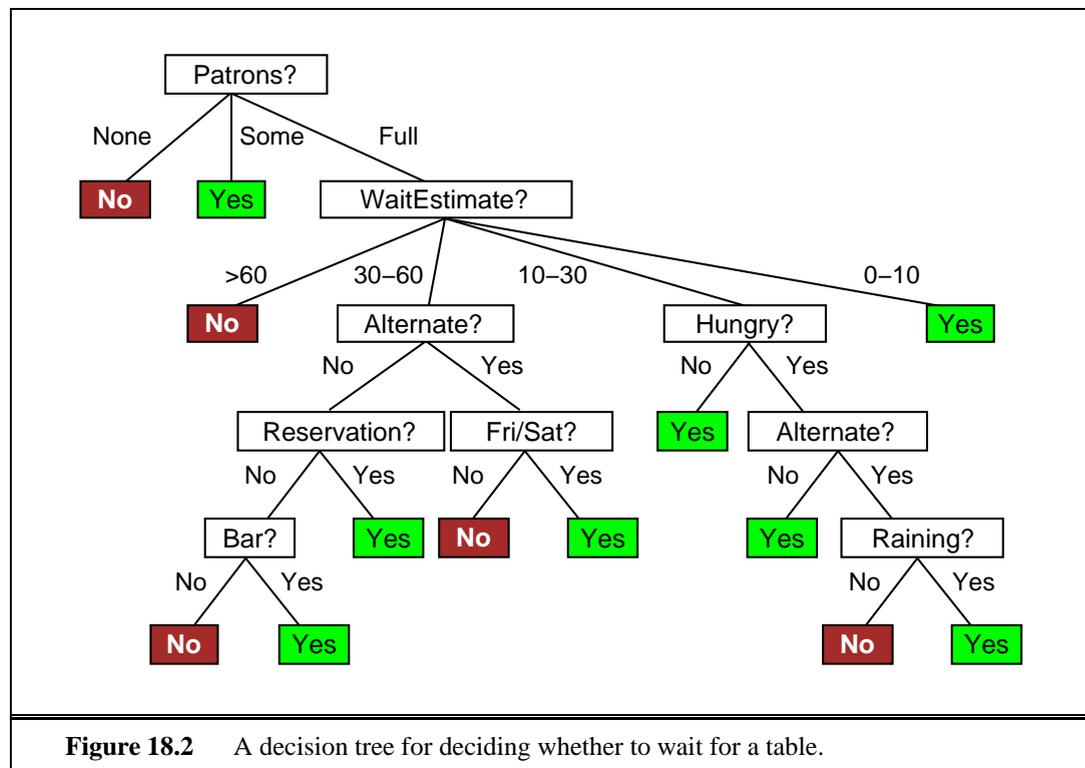
PARITY FUNCTION

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 this many bits to represent.

If it takes  $2^n$  bits to define the function, this means that 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



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

some ingenious algorithms to find consistent hypotheses in such a large space.

### Inducing decision trees from examples

An **example** is described by the values of the attributes and the value of the goal predicate. A set of examples  $X_1, \dots, X_{12}$  for the restaurant domain is shown in Figure 18.3. The positive examples are ones where the goal *WillWait* is true ( $X_1, X_3, \dots$ ) and negative examples are ones where it is false ( $X_2, X_5, \dots$ ). The complete set of examples is called the **training set**.

TRAINING SET

Example	Attributes										Goal
	<i>Alt</i>	<i>Bar</i>	<i>Fri</i>	<i>Hun</i>	<i>Pat</i>	<i>Price</i>	<i>Rain</i>	<i>Res</i>	<i>Type</i>	<i>Est</i>	<i>WillWait</i>
$X_1$	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 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,<sup>5</sup> 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 and 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. This 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.

<sup>5</sup> The same example or an example with the same description—this distinction is very important and we will return to it in Chapter 21.

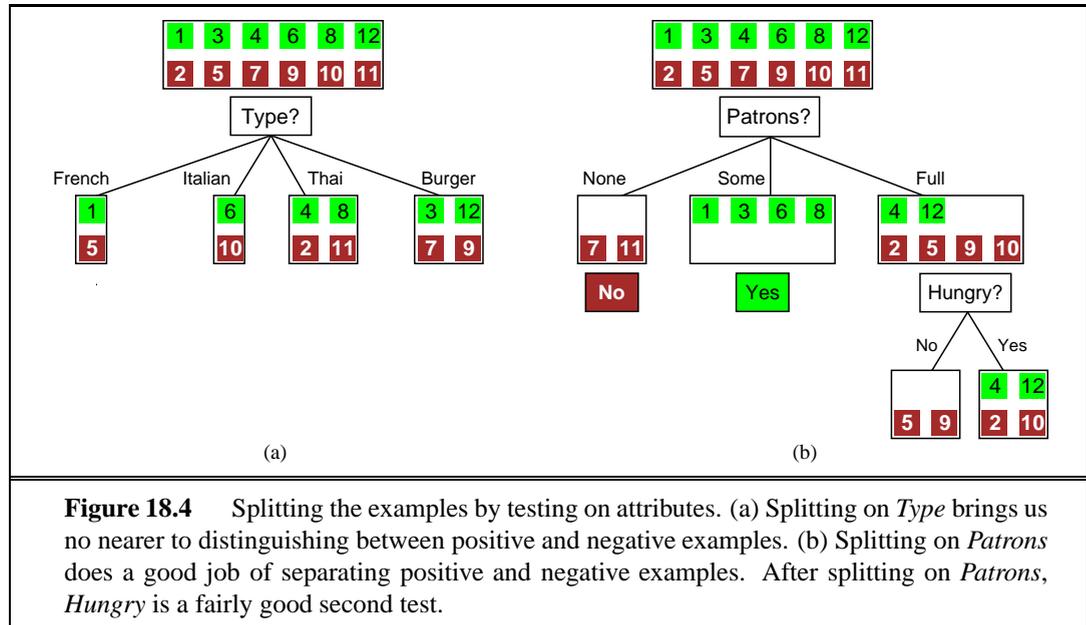


Figure 18.4 shows how the algorithm gets started. We are given 12 training examples, which we classify into positive and negative sets. We then decide which attribute to use as the first test in the tree. Figure 18.4(a) shows that *Type* is a poor attribute, because it leaves us with four possible outcomes, 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 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 when the attributes do not give enough information to fully describe the situation, or when the domain is truly nondeterministic. One simple way out of the problem is to use a majority vote.

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

NOISE

```

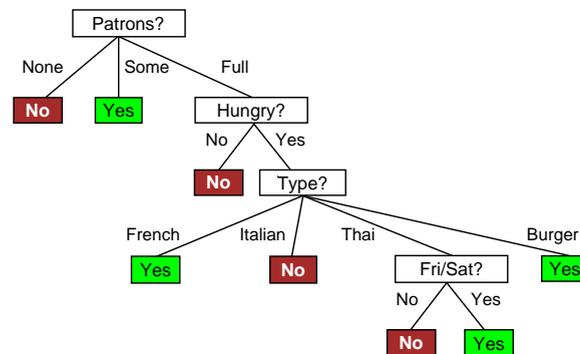
function DECISION-TREE-LEARNING(examples, attributes, default) returns a decision tree
  inputs: examples, set of examples
           attributes, set of attributes
           default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attributes is empty then return MAJORITY-VALUE(examples)
  else
    best ← CHOOSE-ATTRIBUTE(attributes, examples)
    tree ← a new decision tree with root test best
    for each value  $v_i$  of best do
      examplesi ← {elements of examples with best =  $v_i$ }
      subtree ← DECISION-TREE-LEARNING(examplesi, attributes – best,
                                         MAJORITY-VALUE(examples))

      add a branch to tree with label  $v_i$  and subtree subtree
    end
  return tree

```

**Figure 18.5** The decision tree learning algorithm.



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

The final tree produced by the algorithm applied to the 12-example data set is shown in Figure 18.6. The tree is distinctly 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. This would be the wrong conclusion to draw. 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 regularity in the data (namely, that the first author will wait for Thai food on weekends) that was not even suspected.

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 the author 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? The next section shows how to analyze this experimentally.

### 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

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

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

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

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  $E_1, \dots, E_v$  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 random example has the  $i$ th 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:

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

and the heuristic used in the CHOOSE-ATTRIBUTE function is just to choose the attribute with the largest gain.

Looking at the attributes *Patrons* and *Type* and their classifying power, as shown in Figure 18.4, we have

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

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

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 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:

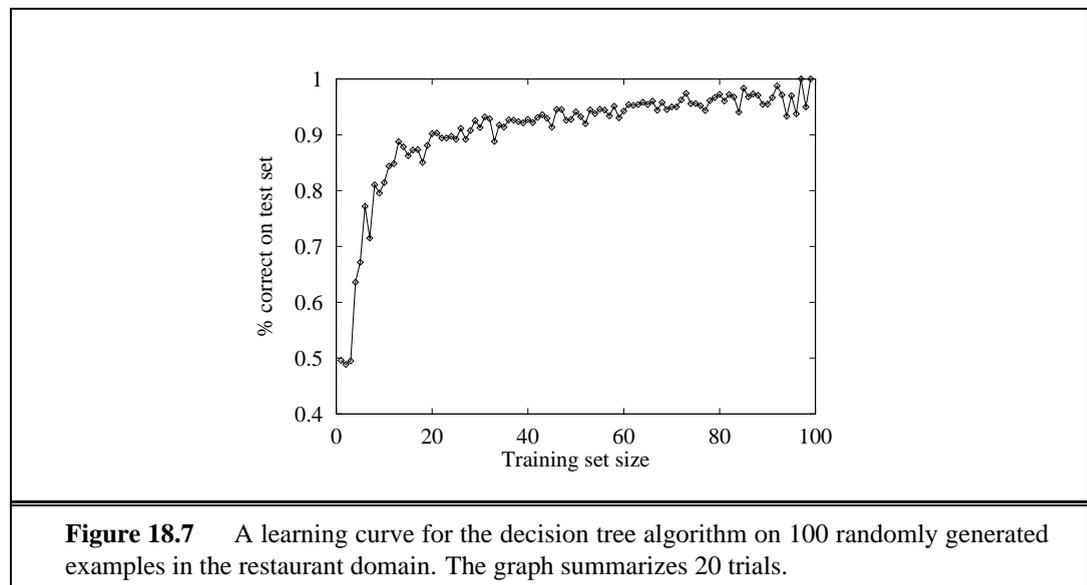
TEST SET

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.

## LEARNING CURVE

The result of this 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 can be plotted on a graph, giving what is called the **learning curve** for the algorithm on the particular domain. The 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.



## PEEKING

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 may 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.

### Noise and overfitting

We saw earlier that if there are two or more examples with the same descriptions (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 or report the estimated probabilities of each classification using the relative frequencies. The former is appropriate for an agent that requires the decision tree to represent a strict logical function, whereas the latter can be used by a decision-theoretic agent.

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 there are no two examples with 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**. It is a very general phenomenon, and occurs even when the target function is not at all random. It afflicts every kind of learning algorithm, not just decision trees.

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

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.<sup>6</sup> 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 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 it deviates from a perfect absence of pattern. If the degree of deviation is statistically unlikely (usually taken to mean a 5% probability or less),

<sup>6</sup> In fact, the gain will be greater than zero unless the proportions are all exactly the same (see Exercise 18.9).

OVERFITTING

DECISION TREE  
PRUNING

SIGNIFICANCE TEST

NULL HYPOTHESIS

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 due to random sampling.

In this case, the null hypothesis is that the attribute is irrelevant, and hence 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  $v$  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  $n_i$ , to the expected numbers  $\hat{p}_i$  and  $\hat{n}_i$  assuming true irrelevance:

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

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  $\chi^2$  (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  $\chi^2$  tables, or with statistical software. Exercise 18.10 asks you to make the appropriate changes to DECISION-TREE-LEARNING to implement this form of pruning, which is known as  $\chi^2$  **pruning**.

$\chi^2$  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 trees. 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, 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 may not have been recorded, or they may 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 18.11.

◇ **Multivalued attributes:** When an attribute has a large number of possible values, the information gain measure gives an inappropriate indication of the attribute's usefulness. Consider the extreme case where every example has a different value for the attribute—for instance, if we were to use an attribute *RestaurantName* in the restaurant domain. In such a case, each subset of examples is a singleton and therefore has a unique classification, so the information gain measure would have its highest value for this attribute. However, the attribute may be irrelevant or useless. One possible solution is to use the **gain ratio**, as described in Exercise 18.12.

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 may 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 an artwork, 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 apply linear regression using the remaining attributes (or some subset thereof).

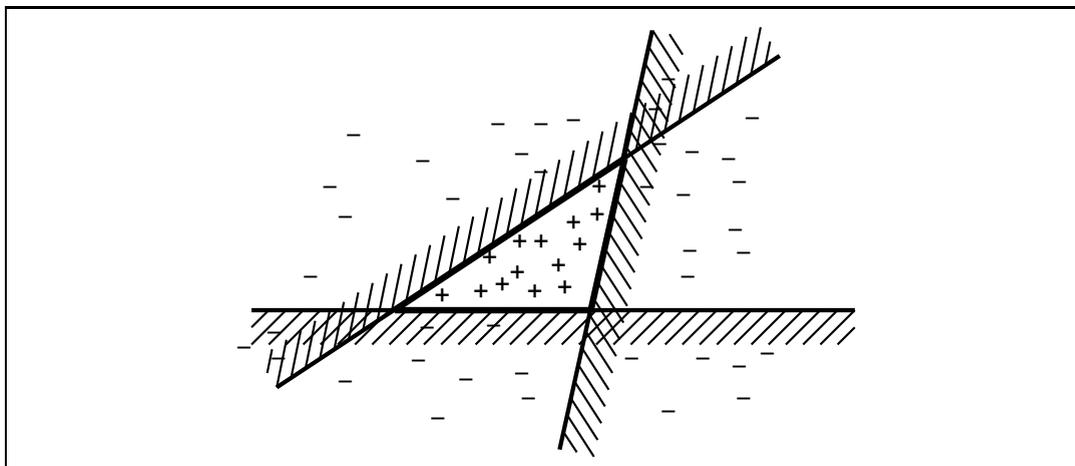
REGRESSION TREE

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, which are also widely used (see Chapter 19).

## 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



**Figure 18.8** Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the non-shaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

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 18.13) shows that using an ensemble of five hypotheses reduces an error rate of one in ten down to an error rate of less than one in a hundred. Now, obviously the assumption of independence is unreasonable, because hypotheses are likely to be misled in the same way by any misleading aspects of the training data. In the extreme case, where all the hypotheses in the ensemble are identical, nothing is gained. If we can find a way to make the hypotheses at least a little bit different, thereby reducing the correlation between their errors, then ensemble learning should still be very useful.

Another way to think about the ensemble idea is as a generic way of enlarging the hypothesis space. That is, think of the ensemble itself as a hypothesis, and the new hypothesis space as the set of all possible ensembles constructible from hypotheses in the original space. Figure 18.8 shows how this can result in a more expressive hypothesis space. If the original hypothesis space allows for a simple and efficient learning algorithm, then the ensemble method provides a way to learn a much more expressive class of hypotheses without incurring

much additional computational or algorithmic complexity.

BOOSTING  
WEIGHTED TRAINING  
SET

The most widely used ensemble method is called **boosting**. To understand how it works, we need first to explain the idea of a **weighted training set**. In such a training set, each example has an associated weight  $w_j \geq 0$ , such that the sum of the weights is 1. The higher the weight of an example, the higher the importance attached to it when learning a hypothesis. It is straightforward to modify the learning algorithms we have seen so far to operate with weighted training sets.<sup>7</sup>

Boosting starts with  $w_j = 1$  for all the examples, i.e., a normal training set. From this set, it generates the first hypothesis,  $h_1$ . This hypothesis will classify some of the training examples correctly and some incorrectly. We would like the next hypothesis to do better on the misclassified examples, so we increase their weights while decreasing the weights of the correctly classified examples. From this new, weighted training set, we generate hypothesis  $h_2$ . The process continues in this way until we have generated  $M$  hypotheses, where  $M$  is an input to the boosting algorithm. The final ensemble hypothesis is a weighted-majority combination of all the  $M$  hypotheses, where each is weighted according to how well it performed on the training set. Figure 18.9 shows how the algorithm works conceptually. There are many variants of the basic boosting idea with different ways of adjusting the weights and combining the hypotheses. One specific algorithm, called ADABOOST, is shown in Figure 18.10. While the details of the weight adjustments are not so important, ADABOOST does have a very important property: provided the input learning algorithm  $L$  is a **weak learning** algorithm—which means that  $L$  always returns a hypothesis with weighted error greater than  $1/2$ —ADABOOST will return a hypothesis that *classifies the training data perfectly* for large enough  $M$ . Thus, the algorithm *boosts* the accuracy of the original learning algorithm on the training data. This result holds no matter how inexpressive the original hypothesis space and no matter how complex the function being learned.

WEAK LEARNING

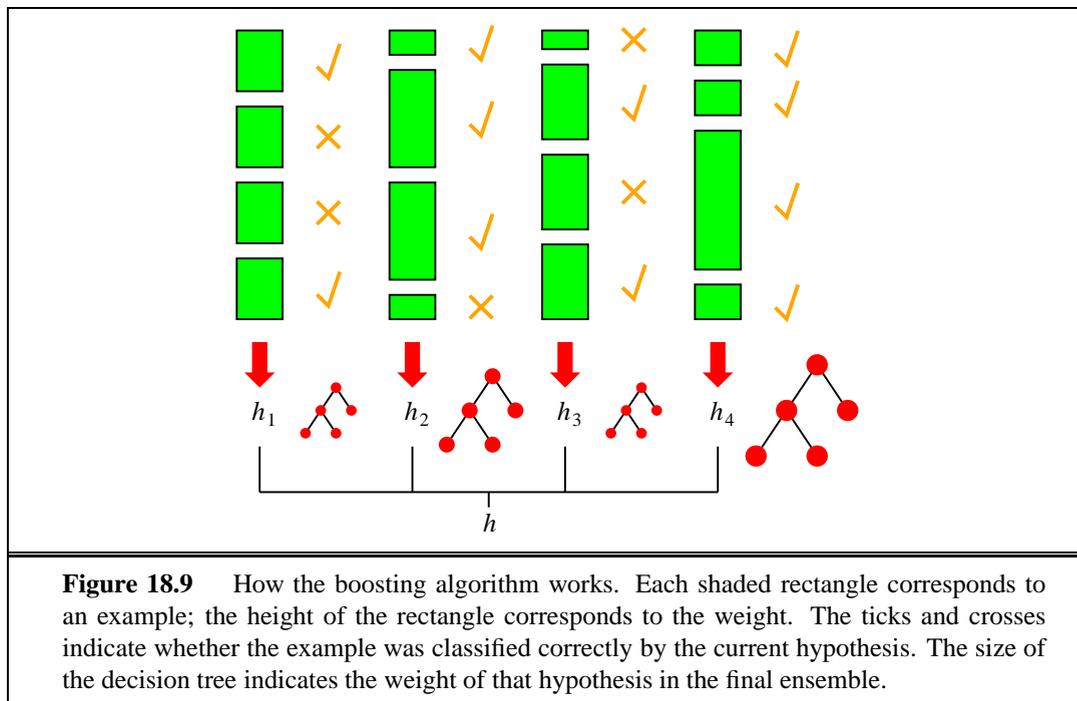
Let us see how well boosting does on the restaurant data. We will choose as our original hypothesis space the class of **decision stumps**, which are decision trees with just one test at the root. The lower curve in Figure 18.11(a) shows that decision stumps are not very effective for this data set, reaching a peak prediction performance of only 72%. When boosting is applied with  $M = 100$ , the performance is much better, reaching 90% after only 58 examples.

DECISION STUMP

An interesting thing happens as the ensemble size  $M$  increases. Figure 18.11(b) shows the training set error (on 60 examples) as a function of  $M$ . Notice that the error reaches zero (as the boosting theorem tells us) when  $M$  is 27; that is, a weighted-majority combination of 27 decision stumps suffices to fit the 60 examples exactly. As more stumps are added to the ensemble, the error remains at zero. The graph also shows that *the test set error continues to decrease long after the training set error has reached zero*. This finding, which is quite robust across data sets and hypothesis spaces, came as quite a surprise when it was first noticed. Ockham's razor tells us not to make hypotheses more complex than necessary, but the graph tells us that the predictions *improve* as the ensemble hypothesis gets more complex! Various explanations have been proposed for this; one view is that boosting approximates **Bayesian**



<sup>7</sup> For learning algorithms where this is not possible, one can instead create a **replicated training set** where the  $i$ th example appears  $w_j$  times, using randomization to handle fractional weights.



**function** ADABOOST(*examples*, *L*, *M*) **returns** a weighted-majority hypothesis

**inputs:** *examples*, set of  $N$  labelled examples  $(x_1, y_1), \dots, (x_N, y_N)$

*L*, a learning algorithm

*M*, the number of hypotheses in the ensemble

**local variables:** **w**, a vector of  $N$  example weights, initially  $1/N$

**h**, a vector of  $M$  hypotheses

**z**, a vector of  $M$  hypothesis weights

**for**  $m = 1$  **to**  $M$  **do**

**h**[ $m$ ]  $\leftarrow L(\text{examples}, \mathbf{w})$

$error \leftarrow 0$

**for**  $j = 1$  **to**  $N$  **do**

**if**  $\mathbf{h}[m](x_j) \neq y_j$  **then**  $error \leftarrow error + \mathbf{w}[j]$

**for**  $j = 1$  **to**  $N$  **do**

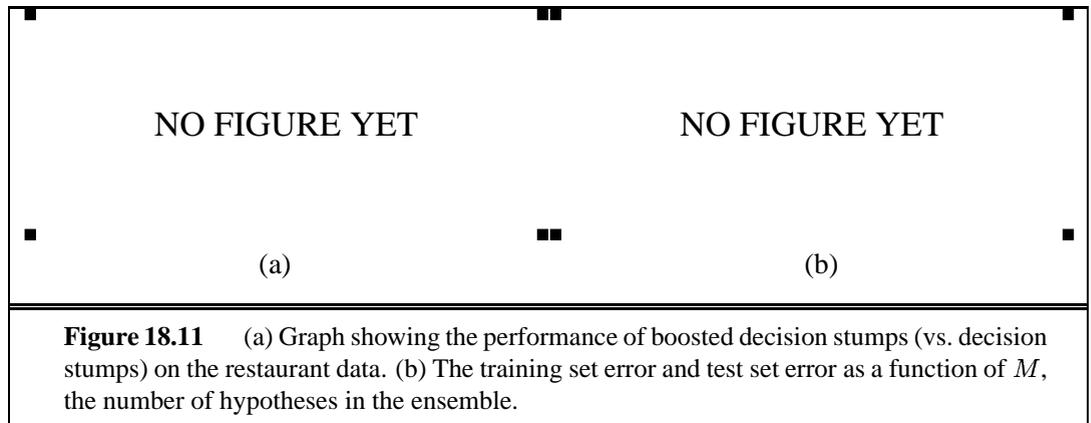
**if**  $\mathbf{h}[m](x_j) = y_j$  **then**  $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error / (1 - error)$

$\mathbf{w} \leftarrow \text{NORMALIZE}(\mathbf{w})$

$\mathbf{z}[m] \leftarrow \log(1 - error) / error$

**return** WEIGHTED-MAJORITY(**h**, **z**)

**Figure 18.10** The ADABOOST variant of the boosting method for ensemble learning. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in **h**, with votes weighted by **z**.



**learning** (see Chapter 19), which can be shown to be an optimal learning algorithm, and the approximation improves as more hypotheses are added.

## 18.5 WHY LEARNING WORKS: COMPUTATIONAL LEARNING THEORY

The main unanswered question posed in Section 18.2 was this: how can one possibly know that one's learning algorithm has produced a theory that will correctly predict the future? In terms of the definition of inductive learning, how do we know that the hypothesis  $h$  is close to the target function  $f$  if we don't know what  $f$  is? These questions have been pondered for several centuries, but unless we find some answers, machine learning will, at best, be puzzled by its own success.

COMPUTATIONAL  
LEARNING THEORY



PROBABLY  
APPROXIMATELY  
CORRECT  
PAC-LEARNING

The approach taken in this section is based on **computational learning theory**, a field at the intersection of AI, statistics, and theoretical computer science. The underlying principle is the following: *any hypothesis that is seriously wrong will almost certainly be “found out” with high probability after a small number of examples, because it will make an incorrect prediction. Thus, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong—that is, it must be **probably approximately correct**.* An algorithm that returns probably approximately correct hypotheses is called a **PAC-learning** algorithm.

STATIONARITY

There are some subtleties in the preceding argument. The main question is the connection between the training and the test examples—after all, we want the hypothesis to be approximately correct on the test set, not just on the training set. The key assumption is that the training and test sets are drawn randomly from the same population of examples using the *same probability distribution*. This is called the **stationarity** assumption. Without the stationarity assumption, the theory can make no claims at all about the future because there would be no necessary connection between future and past. The stationarity assumption amounts to supposing that the process that selects examples is not malevolent. Obviously, if the training set consisted only of weird examples—two-headed dogs, for instance—then the learning algorithm cannot help but make unsuccessful generalizations about how to recognize dogs.

### How many examples are needed?

In order to put these insights into practice, we will need some notation:

- Let  $\mathbf{X}$  be the set of all possible examples.
- Let  $D$  be the distribution from which examples are drawn.
- Let  $\mathbf{H}$  be the set of possible hypotheses.
- Let  $N$  be the number of examples in the training set.

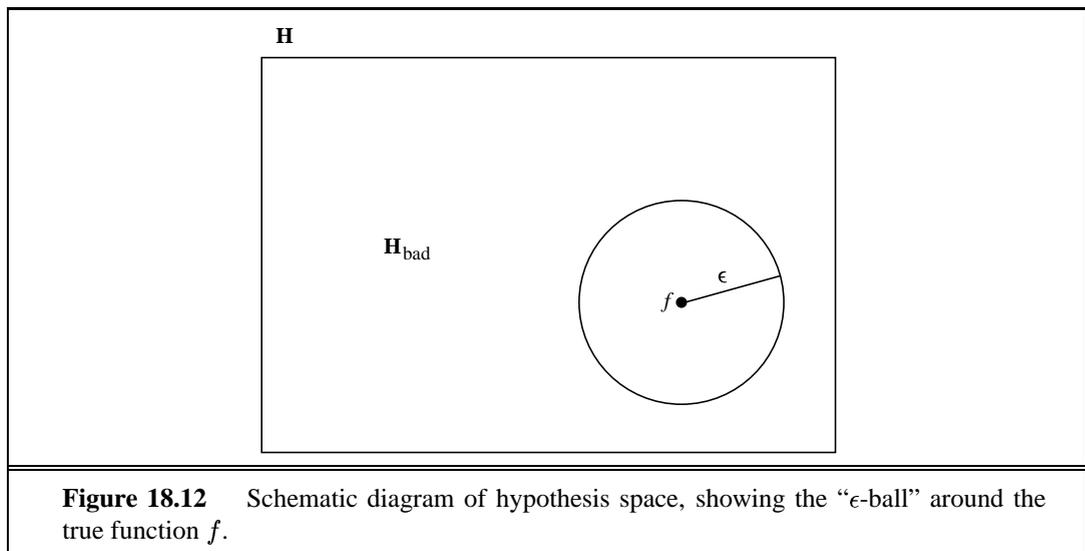
ERROR

Initially, we will assume that the true function  $f$  is a member of  $\mathbf{H}$ . Now we can define the **error** of a hypothesis  $h$  with respect to the true function  $f$  given a distribution  $D$  over the examples as the probability that  $h$  is different from  $f$  on an example:

$$\text{error}(h) = P(h(x) \neq f(x) | x \text{ drawn from } D)$$

This is the same quantity being measured experimentally by the learning curves shown earlier.

A hypothesis  $h$  is called **approximately correct** if  $\text{error}(h) \leq \epsilon$ , where  $\epsilon$  is a small constant. The plan of attack is to show that after seeing  $N$  examples, with high probability, all consistent hypotheses will be approximately correct. One can think of an approximately correct hypothesis as being “close” to the true function in hypothesis space—it lies inside what is called the  **$\epsilon$ -ball** around the true function  $f$ . Figure 18.12 shows the set of all hypotheses  $\mathbf{H}$ , divided into the  $\epsilon$ -ball around  $f$  and the remainder, which we call  $\mathbf{H}_{\text{bad}}$ .

 $\epsilon$ -BALL

**Figure 18.12** Schematic diagram of hypothesis space, showing the “ $\epsilon$ -ball” around the true function  $f$ .

We can calculate the probability that a “seriously wrong” hypothesis  $h_b \in \mathbf{H}_{\text{bad}}$  is consistent with the first  $N$  examples as follows. We know that  $\text{error}(h_b) > \epsilon$ . Thus, the probability that it agrees with any given example is  $\leq (1 - \epsilon)$ . The bound for  $N$  examples is

$$P(h_b \text{ agrees with } N \text{ examples}) \leq (1 - \epsilon)^N$$

The probability that  $\mathbf{H}_{\text{bad}}$  contains at least one consistent hypothesis is bounded by the sum of the individual probabilities:

$$P(\mathbf{H}_{\text{bad}} \text{ contains a consistent hypothesis}) \leq |\mathbf{H}_{\text{bad}}|(1 - \epsilon)^N \leq |\mathbf{H}|(1 - \epsilon)^N$$

where we have used the fact that  $|\mathbf{H}_{\text{bad}}| \leq |\mathbf{H}|$ . We would like to reduce the probability of this event below some small number  $\delta$ :

$$|\mathbf{H}|(1 - \epsilon)^{\delta} \leq \delta$$

Given that  $1 - \epsilon \leq e^{-\epsilon}$ , we can achieve this if we allow the algorithm to see

$$N \geq \frac{1}{\epsilon} \left( \ln \frac{1}{\delta} + \ln |\mathbf{H}| \right) \quad (18.1)$$

examples. Thus, if a learning algorithm returns a hypothesis that is consistent with this many examples, then with probability at least  $1 - \delta$ , it has error at most  $\epsilon$ . In other words, it is probably approximately correct. The number of required examples, as a function of  $\epsilon$  and  $\delta$ , is called the **sample complexity** of the hypothesis space.

SAMPLE  
COMPLEXITY

It appears, then, that the key question is the size of the hypothesis space. As we saw earlier, if  $\mathbf{H}$  is the set of all Boolean functions on  $n$  attributes, then  $|\mathbf{H}| = 2^{2^n}$ . Thus, the sample complexity of the space grows as  $2^n$ . Because the number of possible examples is also  $2^n$ , this says that any learning algorithm for the space of all Boolean functions will do no better than a lookup table, if it merely returns a hypothesis that is consistent with all known examples. Another way to see this is to observe that for any unseen example, the hypothesis space will contain as many consistent hypotheses predicting a positive outcome as predict a negative outcome.

The dilemma we face, then, is that unless we restrict the space of functions the algorithm can consider, it will not be able to learn; but if we do restrict the space, we may eliminate the true function altogether. There are two ways to “escape” this dilemma. The first way is to insist that the algorithm returns not just any consistent hypothesis, but preferably a simple one (as is done in decision tree learning). The theoretical analysis of such algorithms is beyond the scope of this book, but in cases where finding simple consistent hypotheses is tractable, the sample complexity results are generally better than for analyses based only on consistency. The second escape, which we pursue here, is to focus on learnable subsets of the entire set of Boolean functions. The idea is that in most cases we do not need the full expressive power of Boolean functions, and can get by with more restricted languages. We now examine one such restricted language in more detail.

### Learning decision lists

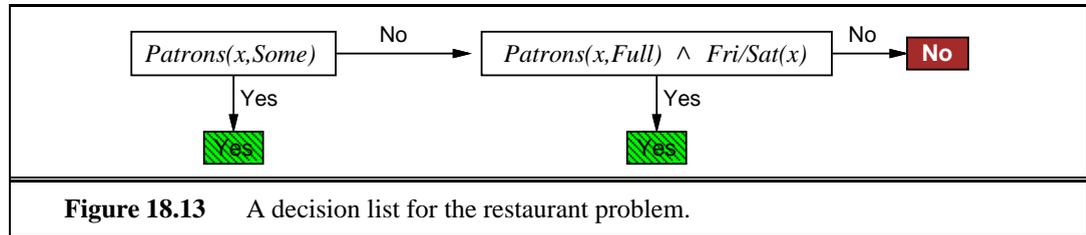
DECISION LIST

A **decision list** is a logical expression of a restricted form. It consists of a series of tests, each of which is a conjunction of literals. If a test succeeds when applied to an example description, the decision list specifies the value to be returned. If the test fails, processing continues with the next test in the list.<sup>8</sup> Decision lists resemble decision trees, but their overall structure is simpler, whereas the individual tests are more complex. Figure 18.13 shows a decision list that represents the following hypothesis:

$$\forall x \text{ WillWait}(x) \Leftrightarrow \text{Patrons}(x, \text{Some}) \vee (\text{Patrons}(x, \text{Full}) \wedge \text{Fri/Sat}(x))$$

If we allow tests of arbitrary size, then decision lists can represent any Boolean function (Exercise 18.14). On the other hand, if we restrict the size of each test to at most  $k$  literals,

<sup>8</sup> A decision list is therefore identical in structure to a COND statement in Lisp.



**Figure 18.13** A decision list for the restaurant problem.

$k$ -DL  
 $k$ -DT

then it is possible for the learning algorithm to generalize successfully from a small number of examples. We call this language  $k$ -DL. The example in Figure 18.13 is in 2-DL. It is easy to show (Exercise 18.14) that  $k$ -DL includes as a subset the language  $k$ -DT, the set of all decision trees of depth at most  $k$ . It is important to remember that the particular language referred to by  $k$ -DL depends on the attributes used to describe the examples. We will use the notation  $k$ -DL( $n$ ) to denote a  $k$ -DL language using  $n$  Boolean attributes.

The first task is to show that  $k$ -DL is learnable—that is, any function in  $k$ -DL can be accurately approximated after seeing a reasonable number of examples. To do this, we need to calculate the number of hypotheses in the language. Let the language of tests—conjunctions of at most  $k$  literals using  $n$  attributes—be  $Conj(n, k)$ . Because a decision list is constructed of tests, and each test can be attached to either a *Yes* or a *No* outcome or can be absent from the decision list, there are at most  $3^{|Conj(n, k)|}$  distinct sets of component tests. Each of these sets of tests can be in any order, so

$$|k\text{-DL}(n)| \leq 3^{|Conj(n, k)|} |Conj(n, k)|!$$

The number of conjunctions of  $k$  literals from  $n$  attributes is given by

$$|Conj(n, k)| = \sum_{i=0}^k \binom{2n}{i} = O(n^k)$$

Hence, after some work, we obtain

$$|k\text{-DL}(n)| = 2^{O(n^k \log_2(n^k))}$$

We can plug this into Equation (18.1) to show that the number of examples needed for PAC-learning a  $k$ -DL function is polynomial in  $n$ :

$$m \geq \frac{1}{\epsilon} \left( \ln \frac{1}{\delta} + O(n^k \log_2(n^k)) \right)$$

Therefore, any algorithm that returns a consistent decision list will PAC-learn a  $k$ -DL function in a reasonable number of examples, for small  $k$ .

The next task is to find an efficient algorithm that returns a consistent decision list. We will use a greedy algorithm called DECISION-LIST-LEARNING that repeatedly finds a test that agrees exactly with some subset of the training set. Once it finds such a test, it adds it to the decision list under construction and removes the corresponding examples. It then constructs the remainder of the decision list using just the remaining examples. This is repeated until there are no examples left. The algorithm is shown in Figure 18.14.

This algorithm does not specify the method for selecting the next test to add to the decision list. Although the formal results given earlier do not depend on the selection method,

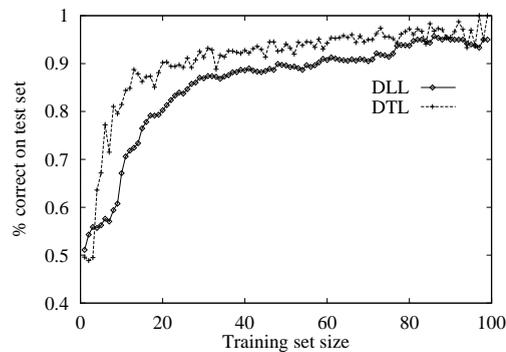
```

function DECISION-LIST-LEARNING(examples) returns a decision list, or failure

if examples is empty then return the trivial decision list No
t ← a test that matches a nonempty subset examplest of examples
      such that the members of examplest are all positive or all negative
if there is no such t then return failure
if the examples in examplest are positive then o ← Yes else o ← No
return a decision list with initial test t and outcome o and remaining tests given by
      DECISION-LIST-LEARNING(examples − examplest)

```

**Figure 18.14** An algorithm for learning decision lists.



**Figure 18.15** Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

it would seem reasonable to prefer small tests that match large sets of uniformly classified examples, so that the overall decision list will be as compact as possible. The simplest strategy is to find the smallest test  $t$  that matches any uniformly classified subset, regardless of the size of the subset. Even this approach works quite well, as Figure 18.15 suggests.

## Discussion

Computational learning theory has generated a new way of looking at the problem of learning. In the early 1960s, the theory of learning focussed on the problem of **identification in the limit**. An identification algorithm must return a hypothesis that exactly matches the true function. One way to do this is as follows. First, order all the hypotheses in  $\mathbf{H}$  according to some measure of simplicity. Then, choose the simplest hypothesis consistent with all the examples so far. As new examples arrive, this method will abandon a simpler hypothesis that is invalidated and adopt a more complex one instead. Once it reaches the true function, it will never abandon it. Unfortunately, in many hypothesis spaces, the number of examples and the computation time required to reach the true function are enormous. Computational

learning theory does not insist that the learning agent find the “one true law” governing its environment, but instead that it find a hypothesis with a certain degree of predictive accuracy. It also brings sharply into focus the trade-off between the expressiveness of the hypothesis language and the complexity of learning.

The PAC-learning results we have shown are worst-case complexity results, and do not necessarily reflect the average-case sample complexity as measured by the learning curves we have shown. An average-case analysis must also make assumptions as to the distribution of examples and the distribution of true functions that the algorithm will have to learn. As these issues become better understood, computational learning theory is providing valuable guidance to machine learning researchers who are interested in predicting or modifying the learning ability of their algorithms. Besides decision lists, results have been obtained for almost all known subclasses of Boolean functions, for neural networks (see Chapter 19) and for sets of first-order logical sentences (see Chapter 21). The results show that the pure inductive learning problem, where the agent begins with no prior knowledge about the target function, is generally very hard. As we show in Chapter 21, the use of prior knowledge to guide inductive learning makes it possible to learn quite large sets of sentences from reasonable numbers of examples, even in a language as expressive as first-order logic.

---

## 18.6 SUMMARY

---

This chapter has concentrated on inductive learning of deterministic functions from examples. The main points were as follows:

- Learning takes many forms, depending on the nature of the performance element, the component to be improved, and the available feedback.
- If the available feedback, either from a teacher or from the environment, provides the correct value for the examples, the learning problem is called **supervised learning**. The task, also called **inductive learning**, is then to learn a function from examples of its inputs and outputs. Learning a discrete-valued function is called **classification**; learning a continuous function is called **regression**.
- Inductive learning involves finding a **consistent** hypothesis that agrees with the examples. **Ockham’s razor** suggests choosing the simplest consistent hypothesis. The difficulty of this task depends on the chosen representation.
- **Decision trees** can represent all Boolean functions. The **information gain** heuristic provides an efficient method for finding a simple, consistent decision tree.
- The performance of inductive learning algorithms is measured by the **learning curve**, which shows the prediction accuracy on the **test set** as a function of the size of the **training set**.
- **Computational learning theory** analyzes the sample complexity and computational complexity of inductive learning. There is a trade-off between the expressiveness of the hypothesis language and the ease of learning.

---

**BIBLIOGRAPHICAL AND HISTORICAL NOTES**

Chapter 1 outlined the history of philosophical investigations into inductive learning. William of Ockham (c. 1285–1349), the most influential philosopher of his century and a major contributor to medieval epistemology, logic, and metaphysics, is credited with a statement called “Ockham’s Razor”—in Latin, *Entia non sunt multiplicanda praeter necessitatem*, and in English, “Entities are not to be multiplied without necessity.” Unfortunately, this laudable piece of advice is nowhere to be found in his writings in precisely these words.

EPAM, the “Elementary Perceiver And Memorizer” (Feigenbaum, 1961), was one of the earliest systems to use decision trees (or **discrimination nets**). EPAM was intended as a cognitive-simulation model of human concept learning. CLS (Hunt *et al.*, 1966) used a heuristic lookahead method to construct decision trees. ID3 (Quinlan, 1979) added the crucial idea of using information content to provide the heuristic function. Information theory itself was developed by Claude Shannon to aid in the study of communication (Shannon and Weaver, 1949). (Shannon also contributed one of the earliest examples of machine learning, a mechanical mouse named Theseus that learned to navigate through a maze by trial and error.) The  $\chi^2$  method of tree pruning was described by Quinlan (1986). C4.5, an industrial-strength decision tree package, can be found in Quinlan (1993). An independent tradition of decision tree learning also exists in the statistical literature. *Classification and Regression Trees* (Breiman *et al.*, 1984), known as the “CART book,” is the principal reference.

Many other algorithmic approaches to learning have been tried. The **current-best-hypothesis** approach maintains a single hypothesis, specializing it when it proves too broad and generalizing it when it proves too narrow. This is an old idea in philosophy (Mill, 1843). Early work in cognitive psychology also suggested that this is a natural form of concept learning in humans (Bruner *et al.*, 1957). In AI, the approach is most closely associated with the work of Patrick Winston, whose Ph.D. thesis (Winston, 1970) addressed the problem of learning descriptions of complex objects. The **version space** method (Mitchell, 1977, 1982) takes a different approach, maintaining the set of *all* consistent hypotheses and eliminating those found to be inconsistent with new examples. The approach was used in the Meta-DENDRAL expert system for chemistry (Buchanan and Mitchell, 1978), and later in Mitchell’s (1983) LEX system, which learns to solve calculus problems. A third influential thread was formed by the work of Michalski and colleagues on the AQ series of algorithms, which learned sets of logical rules (Michalski, 1969; Michalski *et al.*, 1986).

BAGGING

Ensemble learning is an increasingly popular technique for improving the performance of learning algorithms. **Bagging** (Breiman, 1996), the first effective method, worked by generating multiple **bootstrap** data sets by subsampling the original data set; the final hypothesis is the majority vote of the hypotheses obtained from the bootstrap data sets. The **boosting** method described in the chapter originated with theoretical work by Schapire (1999) showing that the prediction accuracy of a weak learning algorithm could be boosted by generating several hypotheses from distinct data sets. The ADABOOST algorithm was developed by Freund and Schapire (1996) and analyzed theoretically by Schapire (1999). Friedman *et al.* (2000) explain boosting from a statistician’s viewpoint.

Theoretical analysis of learning algorithms began with the work of Gold (1967) on **identification in the limit**. This approach was motivated in part by models of scientific discovery from the philosophy of science (Popper, 1962), but has been applied mainly to the problem of learning grammars from example sentences. Osherson, Stob, and Weinstein (1986) provide a modern and rigorous treatment of the field.

Whereas the identification-in-the-limit approach concentrates on eventual convergence, the study of **Kolmogorov complexity** or **algorithmic complexity**, developed independently by Solomonoff (1964) and Kolmogorov (1965), attempts to provide a formal definition for the notion of simplicity used in Ockham's razor. To escape the problem that simplicity depends on the way in which information is represented, it is proposed that simplicity be measured by the length of the shortest program for a universal Turing machine that correctly reproduces the observed data. Although there are many possible universal Turing machines, and hence many possible "shortest" programs, these programs differ in length by at most a constant that is independent of the amount of data. This beautiful insight, which essentially shows that *any* initial representation bias will eventually be overcome by the data itself, is marred only by the undecidability of computing the length of the shortest program. Approximate measures such as the **minimum description length** or MDL (Rissanen, 1984) can be used instead, and have produced excellent results in practice. The text by Li and Vitanyi (1993) is the best source for Kolmogorov complexity.

Computational learning theory in the modern sense, that is, the theory of PAC-learning, was inaugurated by Leslie Valiant (1984). Valiant's work stressed the importance of computational as well as sample complexity. With Michael Kearns (1990), Valiant showed that several concept classes cannot be PAC-learned tractably even though sufficient information is available in the examples. Some positive results have been obtained for classes such as decision lists (Rivest, 1987).

An independent tradition of sample complexity analysis has existed in statistics, beginning with the work on **uniform convergence theory** (Vapnik and Chervonenkis, 1971). The so-called **VC dimension** provides a measure roughly analogous to, but more general than, the  $\ln |\mathbf{H}|$  measure obtained from PAC analysis. The VC dimension can be applied to continuous function classes, to which standard PAC analysis does not apply. PAC-learning theory and VC theory were first connected by the "four Germans" (none of whom are actually German): Blumer, Ehrenfeucht, Haussler, and Warmuth (1989). Subsequent developments in VC theory led to the invention of **support vector machines** or SVMs (Cortes and Vapnik, 1995), later subsumed under the heading of **kernel methods**. This is an extremely active research area and has provided effective methods for several large-scale applications. Cristianini and Shawe-Taylor (2000) and Scholkopf and Smola (2002) provide thorough introductions.

A large number of important papers on machine learning have been collected in *Readings in Machine Learning* (Shavlik and Dietterich, 1990). The two volumes *Machine Learning 1* (Michalski *et al.*, 1983) and *Machine Learning 2* (Michalski *et al.*, 1986) also contain many important papers as well as huge bibliographies. Weiss and Kulikowski (1991) provide a broad introduction to function-learning methods from machine learning, statistics, and neural networks. The STATLOG project (Michie *et al.*, 1994) is by far the most exhaustive investigation into the comparative performance of learning algorithms. Good current research

KOLMOGOROV  
COMPLEXITYMINIMUM  
DESCRIPTION  
LENGTHUNIFORM  
CONVERGENCE  
THEORY  
VC DIMENSIONSUPPORT VECTOR  
MACHINES  
KERNEL METHODS

in machine learning is published in the annual proceedings of the International Conference on Machine Learning and the Neural Information Processing Systems conference, in *Machine Learning* and the *Journal of Machine Learning Research*, and in mainstream AI journals. Work in computational learning theory also appears in the annual ACM Workshop on Computational Learning Theory (COLT), and is described in the texts by Kearns and Vazirani (1994) and Anthony and Bartlett (1999).

---

## EXERCISES

**18.1** Consider the problem faced by an infant learning to speak and understand a language. Explain how this process fits into the general learning model, identifying each of the components of the model as appropriate.

**18.2** Repeat Exercise 18.1 for the case of learning to play tennis (or some other competitive sport with which you are familiar). Is this supervised learning or reinforcement learning?

**18.3** Draw a decision tree for the problem of deciding whether or not to move forward at a road intersection given that the light has just turned green.

**18.4** We never test the same attribute twice along one path in a decision tree. Why not?

**18.5** Suppose we generate a training set from a decision tree and then apply decision-tree learning to that training set. Is it the case that the learning algorithm will eventually return the correct tree as the training set size goes to infinity? Why, or why not?

**18.6** A good “straw man” learning algorithm is as follows: create a table out of all the training examples. Determine which output occurs most often among the training examples; call it  $d$ . Then when given an input that is not in the table, just return  $d$ . For inputs that *are* in the table, return the output associated with it (or the most frequent output, if there is more than one). Implement this algorithm and see how well it does on the restaurant domain. This should give you an idea of the baseline for the domain—the minimal performance that any algorithm should be able to obtain.

**18.7** In the recursive construction of decision trees, it sometimes occurs that a mixed set of positive and negative examples remains at a leaf node, even after all the attributes have been used. Suppose that we have  $p$  positive examples and  $n$  negative examples.

**a.** Show that the solution used by DECISION-TREE-LEARNING, which picks the majority classification, minimizes the absolute error over the set of examples at the leaf.

CLASS PROBABILITY

**b.** Show that the **class probability**  $p/(p + n)$  minimizes the sum of squared errors.

**18.8** Suppose that a learning algorithm is trying to find a consistent hypothesis when the classifications of examples are actually random. There are  $n$  Boolean attributes, and examples are drawn uniformly from the set of  $2^n$  possible examples. Calculate the number of examples required before the probability of finding a contradiction in the data reaches 0.5.

**18.9** Suppose that an attribute splits the set of examples  $E$  into subsets  $E_i$ , and that each subset has  $p_i$  positive examples and  $n_i$  negative examples. Show that unless the ratio  $p_i/(p_i + n_i)$  is the same for all  $i$ , the attribute has strictly positive information gain.

**18.10** Modify DECISION-TREE-LEARNING to include  $\chi^2$ -pruning. You may wish to consult Quinlan (1986) for details.



**18.11** The standard DECISION-TREE-LEARNING algorithm described in the chapter does not handle cases in which some examples have missing attribute values.

- a. First, we need to find a way to classify such examples, given a decision tree that includes tests on the attributes for which values may be missing. Suppose an example  $X$  has a missing value for attribute  $A$ , and that the decision tree tests for  $A$  at a node that  $X$  reaches. One way to handle this case is to pretend that the example has *all* possible values for the attribute, but to weight each value according to its frequency among all of the examples that reach that node in the decision tree. The classification algorithm should follow all branches at any node for which a value is missing, and should multiply the weights along each path. Write a modified classification algorithm for decision trees that has this behavior.
- b. Now modify the information gain calculation so that in any given collection of examples  $C$  at a given node in the tree during the construction process, the examples with missing values for any of the remaining attributes are given “as-if” values according to the frequencies of those values in the set  $C$ .

**18.12** In the chapter, we noted that attributes with many different possible values can cause problems with the gain measure. Such attributes tend to split the examples into many small classes or even singleton classes, thereby appearing to be highly relevant according to the gain measure. The **gain ratio** criterion selects attributes according to the ratio between their gain and their intrinsic information content, that is, the amount of information contained in the answer to the question, “What is the value of this attribute?” The gain ratio criterion therefore tries to measure how efficiently an attribute provides information on the correct classification of an example. Write a mathematical expression for the information content of an attribute, and implement the gain ratio criterion in DECISION-TREE-LEARNING.

**18.13** Consider an ensemble learning algorithm that uses simple majority voting among  $M$  learned hypotheses. Suppose that each hypothesis has error  $\epsilon$  and that the errors made by each hypothesis are independent of the others’. Calculate a formula for the error of the ensemble algorithm in terms of  $M$  and  $\epsilon$  and evaluate it for the cases where  $M = 5, 10,$  and  $20$  and  $\epsilon = 0.1, 0.2,$  and  $0.4$ . If the independence assumption is removed, is it possible for the ensemble error to be *worse* than  $\epsilon$ ?

**18.14** This exercise concerns the expressiveness of decision lists (Section 18.5).

- a. Show that with tests of unlimited size, decision lists can represent any Boolean function.
- b. Show that if the tests can contain at most  $k$  literals each, then decision lists can represent any function that can be represented by a decision tree of depth  $k$ .

**18.15** Suppose you are running a learning experiment on a new algorithm. You have a data set consisting of 25 examples of each of 2 classes. You plan to use leave-one-out cross-validation. As a baseline, you run your experimental setup on a simple majority classifier. (A majority classifier is given a set of training data, and then always outputs the class that is in the majority in the training set, regardless of the input.) You expect the majority classifier to score about 50% on leave-one-out cross-validation, but to your surprise, it scores zero. Can you explain why?