18 LEARNING FROM EXAMPLES

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

LEARNING

An agent is **learning** if it improves its performance on future tasks after making observations about the world. Learning can range from the trivial, as exhibited by jotting down a phone number, to the profound, as exhibited by Albert Einstein, who inferred a new theory of the universe. In this chapter we will concentrate on one class of learning problem, which seems restricted but actually has vast applicability: from a collection of input–output pairs, learn a function that predicts the output for new inputs.

Why would we want an agent to learn? If the design of the agent can be improved, why wouldn't the designers just program in that improvement to begin with? There are three main reasons. First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters. Second, the designers cannot anticipate all changes over time; a program designed to predict tomorrow's stock market prices must learn to adapt when conditions change from boom to bust. Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family members, but even the best programmers are unable to program a computer to accomplish that task, except by using learning algorithms. This chapter first gives an overview of the various forms of learning, then describes one popular approach, decision-tree learning, in Section 18.3, followed by a theoretical analysis of learning in Sections 18.4 and 18.5. We look at various learning systems used in practice: linear models, nonlinear models (in particular, neural networks), nonparametric models, and support vector machines. Finally we show how ensembles of models can outperform a single model.

18.1 FORMS OF LEARNING

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which *component* is to be improved.

- What *prior knowledge* the agent already has.
- What *representation* is used for the data and the component.
- What *feedback* is available to learn from.

Components to be learned

Chapter 2 described several agent designs. The components of these agents include:

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.
6. Goals that describe classes of states whose achievement maximizes the agent's utility.

Each of these components can be learned. Consider, for example, an agent training to become a taxi driver. Every time the instructor shouts "Brake!" the agent might learn a condition–action rule for when to brake (component 1); the agent also learns every time the instructor does not shout. 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).

Representation and prior knowledge

We have seen several examples of representations for agent components: propositional and first-order logical sentences for the components in a logical agent; Bayesian networks for the inferential components of a decision-theoretic agent, and so on. Effective learning algorithms have been devised for all of these representations. This chapter (and most of current machine learning research) covers inputs that form a **factored representation**—a vector of attribute values—and outputs that can be either a continuous numerical value or a discrete value. Chapter 19 covers functions and prior knowledge composed of first-order logic sentences, and Chapter 20 concentrates on Bayesian networks.

There is another way to look at the various types of learning. We say that learning a (possibly incorrect) general function or rule from specific input–output pairs is called **inductive learning**. We will see in Chapter 19 that we can also do **analytical** or **deductive learning**: going from a known general rule to a new rule that is logically entailed, but is useful because it allows more efficient processing.

INDUCTIVE LEARNING
DEDUCTIVE LEARNING

Feedback to learn from

There are three *types of feedback* that determine the three main types of learning:

In **unsupervised learning** the agent learns patterns in the input even though no explicit feedback is supplied. The most common unsupervised learning task is **clustering**: detecting

UNSUPERVISED LEARNING
CLUSTERING

potentially useful clusters of input examples. For example, a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days" without ever being given labeled examples of each by a teacher.

REINFORCEMENT LEARNING

In **reinforcement learning** the agent learns from a series of reinforcements—rewards or punishments. For example, the lack of a tip at the end of the journey gives the taxi agent an indication that it did something wrong. The two points for a win at the end of a chess game tells the agent it did something right. It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it.

SUPERVISED LEARNING

In **supervised learning** the agent observes some example input–output pairs and learns a function that maps from input to output. In component 1 above, the inputs are percepts and the output are provided by a teacher who says "Brake!" or "Turn left." In component 2, the inputs are camera images and the outputs again come from a teacher who says "that's a bus." In 3, the theory of braking is a function from states and braking actions to stopping distance in feet. In this case the output value is available directly from the agent's percepts (after the fact); the environment is the teacher.

SEMI-SUPERVISED LEARNING

In practice, these distinction are not always so crisp. In **semi-supervised learning** we are given a few labeled examples and must make what we can of a large collection of unlabeled examples. Even the labels themselves may not be the oracular truths that we hope for. Imagine that you are trying to build a system to guess a person's age from a photo. You gather some labeled examples by snapping pictures of people and asking their age. That's supervised learning. But in reality some of the people lied about their age. It's not just that there is random noise in the data; rather the inaccuracies are systematic, and to uncover them is an unsupervised learning problem involving images, self-reported ages, and true (unknown) ages. Thus, both noise and lack of labels create a continuum between supervised and unsupervised learning.

18.2 SUPERVISED LEARNING

The task of supervised learning is this:

RAINING SET

Given a **training set** of N example input–output pairs

$$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N) ,$$

where each y_j was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.

YPOTHESIS

Here x and y can be any value; they need not be numbers. The function h is a **hypothesis**.[1] Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis we give it a **test set** of examples that are distinct from the training set. We say a hypothesis

EST SET

[1] A note on notation: except where noted, we will use j to index the N examples; x_j will always be the input and y_j the output. In cases where the input is specifically a vector of attribute values (beginning with Section 18.3), we will use $\mathbf{x}_j$ for the jth example and we will use i to index the n attributes of each example. The elements of $\mathbf{x}_j$ are written $x_{j,1}, x_{j,2}, \ldots, x_{j,n}$.

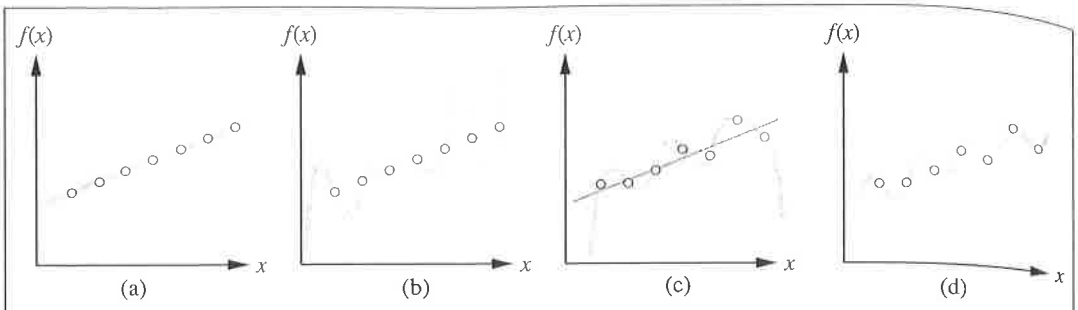

Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

GENERALIZATION

generalizes well if it correctly predicts the value of y for novel examples. Sometimes the function f is stochastic—it is not strictly a function of x, and what we have to learn is a conditional probability distribution, $\mathbf{P}(Y \mid x)$.

CLASSIFICATION

REGRESSION

When the output y is one of a finite set of values (such as *sunny, cloudy* or *rainy*), the learning problem is called **classification**, and is called Boolean or binary classification if there are only two values. When y is a number (such as tomorrow's temperature), the learning problem is called **regression**. (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found *exactly* the right real-valued number for y is 0.)

HYPOTHESIS SPACE

CONSISTENT

Figure 18.1 shows a familiar example: fitting a function of a single variable to some data points. The examples are points in the (x, y) plane, where $y = f(x)$. We don't know what f is, but we will approximate it with a function h selected from a **hypothesis space**, $\mathcal{H}$, which for this example we will take to be the set of polynomials, such as $x^5 + 3x^2 + 2$. Figure 18.1(a) shows some data with an exact fit by a straight line (the polynomial $0.4x + 3$). 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 a fundamental problem in inductive learning: *how do we choose from among multiple consistent hypotheses?*

OCKHAM'S RAZOR

One answer is to prefer the *simplest* hypothesis consistent with the data. This principle is called **Ockham's razor**, after the 14th-century English philosopher William of Ockham, who used it to argue sharply against all sorts of complications. Defining simplicity is not easy, but it seems clear that a degree-1 polynomial is simpler than a degree-7 polynomial, and thus (a) should be preferred to (b). We will make this intuition more precise in Section 18.4.3.

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 for an exact fit. There are just 7 data points, so a polynomial with 7 parameters does not seem to be finding any pattern in the data and we do not expect it to generalize well. A straight line that is not consistent with any of the data points, but might generalize fairly well for unseen values of x, is also shown in (c). *In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better.* In Figure 18.1(d) we expand the

hypothesis space $\mathcal{H}$ to allow polynomials over both x and $\sin(x)$, and find that the data in (c) can be fitted exactly by a simple function of the form $ax + b + c\sin(x)$. This shows the importance of the choice of hypothesis space. We say that a learning problem is **realizable** if the hypothesis space contains the true function. Unfortunately, we cannot always tell whether a given learning problem is realizable, because the true function is not known.

REALIZABLE

In some cases, an analyst looking at a problem is willing to make more fine-grained distinctions about the hypothesis space, to say—even before seeing any data—not just that a hypothesis is possible or impossible, but rather how probable it is. Supervised learning can be done by choosing the hypothesis h^* that is most probable given the data:

$$h^* = \operatorname*{argmax}_{h \in \mathcal{H}} P(h|data) \ .$$

By Bayes' rule this is equivalent to

$$h^* = \operatorname*{argmax}_{h \in \mathcal{H}} P(data|h)\, P(h) \ .$$

Then we can say that the prior probability $P(h)$ is high for a degree-1 or -2 polynomial, lower for a degree-7 polynomial, and especially low for degree-7 polynomials with large, sharp spikes as in Figure 18.1(b). We allow unusual-looking functions when the data say we really need them, but we discourage them by giving them a low prior probability.

Why not let $\mathcal{H}$ be the class of all Java programs, or Turing machines? After all, every computable function can be represented by some Turing machine, and that is the best we can do. One 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 a good hypothesis within that space.* For example, fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting Turing machines is in general undecidable. A second reason to prefer simple hypothesis spaces is that presumably we will want to use h after we have learned it, and computing $h(x)$ when h is a linear function is guaranteed to be fast, while computing an arbitrary Turing machine program is not even guaranteed to terminate. For these reasons, most work on learning has focused on simple representations.

We will see that the expressiveness–complexity tradeoff is not as simple as it first seems: it is often the case, as we saw with first-order logic in Chapter 8, that an expressive language makes it possible for a *simple* hypothesis to fit the data, whereas restricting the expressiveness of the language means that any consistent hypothesis 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.

18.3 LEARNING DECISION TREES

Decision tree induction is one of the simplest and yet most successful forms of machine learning. We first describe the representation—the hypothesis space—and then show how to learn a good hypothesis.

18.3.1 The decision tree representation

DECISION TREE

A **decision tree** represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value. The input and output values can be discrete or continuous. For now we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each example input will be classified as true (a **positive** example) or false (a **negative** example).

POSITIVE

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 input attributes, A_i, and the branches from the node are labeled with the possible values of the attribute, $A_i = v_{ik}$. Each leaf node in the tree specifies a value to be returned by the function. The decision tree representation is 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

As an example, we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** $WillWait$. First we list the attributes that we will consider as part of the input:

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, or >60).

Note that every variable has a small set of possible values; the value of $WaitEstimate$, for example, is not an integer, rather it is one of the four discrete values 0–10, 10–30, 30–60, or >60. The decision tree usually used by one of us (SR) for this domain is shown in Figure 18.2. Notice that the tree ignores the $Price$ and $Type$ attributes. 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).

18.3.2 Expressiveness of decision trees

A Boolean decision tree is logically equivalent to the assertion that the goal attribute is true if and only if the input attributes satisfy one of the paths leading to a leaf with value $true$. Writing this out in propositional logic, we have

$$Goal \Leftrightarrow (Path_1 \vee Path_2 \vee \cdots),$$

where each $Path$ is a conjunction of attribute-value tests required to follow that path. Thus, the whole expression is equivalent to disjunctive normal form (see page 283), which means

that any function in propositional logic can be expressed as a decision tree. As an example, the rightmost path in Figure 18.2 is

$$Path = (Patrons = Full \wedge WaitEstimate = 0\text{–}10) .$$

For a wide variety of problems, the decision tree format yields a nice, concise result. But some functions cannot be represented concisely. For example, the majority function, which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree. 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 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. A truth table over n attributes has 2^n rows, one for each combination of values of the attributes. We can consider the "answer" column of the table as a 2^n-bit number that defines the function. That means there are 2^{2^n} different functions (and there will be more than that number of trees, since more than one tree can compute the same function). This is a scary number. For example, with just the ten Boolean attributes of our restaurant problem there are 2^{1024} or about 10^{308} different functions to choose from, and for 20 attributes there are over $10^{300,000}$. We will need some ingenious algorithms to find good hypotheses in such a large space.

18.3.3 Inducing decision trees from examples

An example for a Boolean decision tree consists of an $(\mathbf{x}, y)$ pair, where $\mathbf{x}$ is a vector of values for the input attributes, and y is a single Boolean output value. A training set of 12 examples

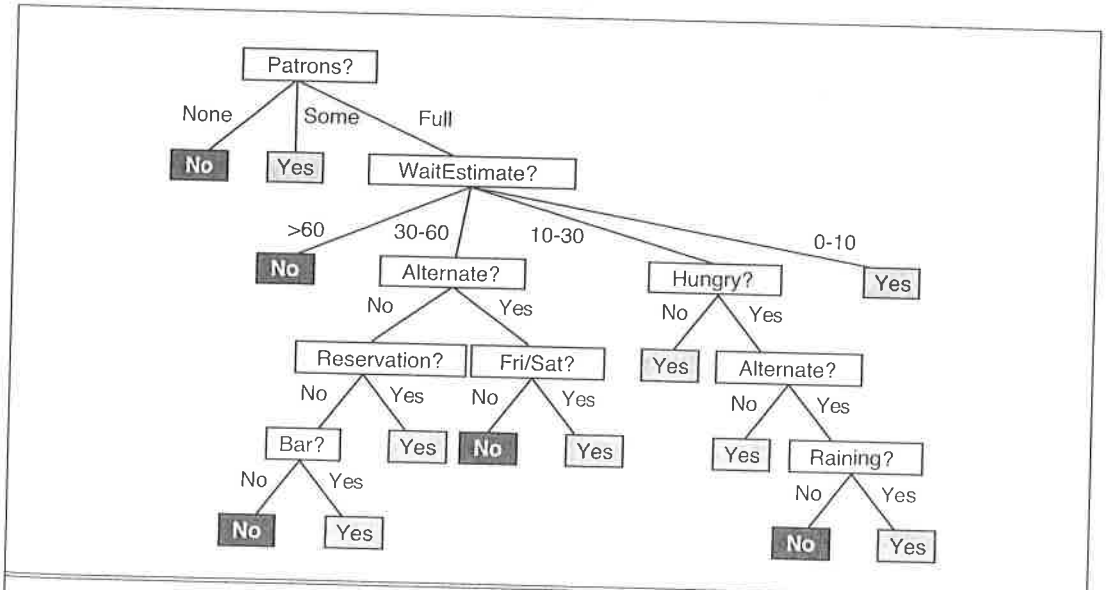

Figure 18.2 A decision tree for deciding whether to wait for a table.

Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Figure 18.3 Examples for the restaurant domain.

is shown in Figure 18.3. The positive examples are the ones in which the goal *WillWait* is true $(\mathbf{x}_1, \mathbf{x}_3, \ldots)$; the negative examples are the ones in which it is false $(\mathbf{x}_2, \mathbf{x}_5, \ldots)$.

We want a tree that is consistent with the examples and is as small as possible. Unfortunately, no matter how we measure size, it is an intractable problem to find the smallest consistent tree; there is no way to efficiently search through the 2^{2^n} trees. With some simple heuristics, however, we can find a good approximate solution: a small (but not smallest) consistent tree. The DECISION-TREE-LEARNING algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. By "most important attribute," 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 shallow.

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 as negative examples. On the other hand, in (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 less attribute. There are four cases to consider for these recursive problems:

1. If the remaining examples are all positive (or all negative), then we are done: we can answer *Yes* or *No*. Figure 18.4(b) shows examples of this happening in the *None* and *Some* branches.
2. 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.
3. If there are no examples left, it means that no example has been observed for this com-

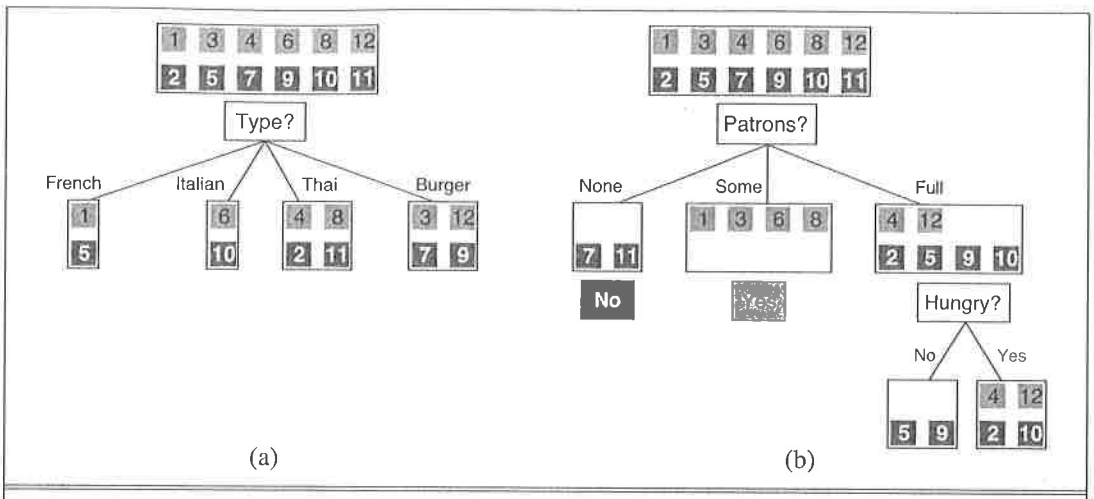

Figure 18.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of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

bination of attribute values, and we return a default value calculated from the plurality classification of all the examples that were used in constructing the node's parent. These are passed along in the variable *parent_examples*.

4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an error or **noise** in the data; because the domain is nondeterministic; or because we can't observe an attribute that would distinguish the examples. The best we can do is return the plurality classification of the remaining examples.

NOISE

The DECISION-TREE-LEARNING algorithm is shown in Figure 18.5. Note that the set of examples is crucial for *constructing* the tree, but nowhere do the examples appear in the tree itself. A tree consists of just tests on attributes in the interior nodes, values of attributes on the branches, and output values on the leaf nodes. The details of the IMPORTANCE function are given in Section 18.3.4. The output of the learning algorithm on our sample training set is shown in Figure 18.6. The tree is clearly different from the original tree shown in Figure 18.2. 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, however. The learning algorithm looks at the *examples*, not at the correct function, and in fact, its hypothesis (see Figure 18.6) not only is consistent 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. It is also bound to make some mistakes for cases where it has seen no examples. For example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full.

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns a tree

  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← {e : e ∈ examples and e.A = v_k}
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

Figure 18.5 The decision-tree learning algorithm. The function IMPORTANCE is described in Section 18.3.4. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

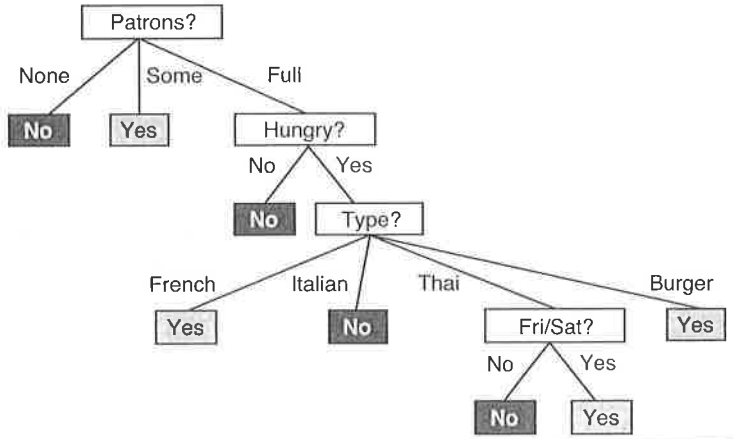

Figure 18.6 The decision tree induced from the 12-example training set.

In that case it says not to wait when *Hungry* is false, but I (SR) would certainly wait. With more training examples the learning program could correct this mistake.

We note there is a danger of over-interpreting the tree that the algorithm selects. When there are several variables of similar importance, the choice between them is somewhat arbitrary: with slightly different input examples, a different variable would be chosen to split on first, and the whole tree would look completely different. The function computed by the tree would still be similar, but the structure of the tree can vary widely.

LEARNING CURVE

We can evaluate the accuracy of a learning algorithm with a **learning curve**, as shown in Figure 18.7. We have 100 examples at our disposal, which we split into a training set and

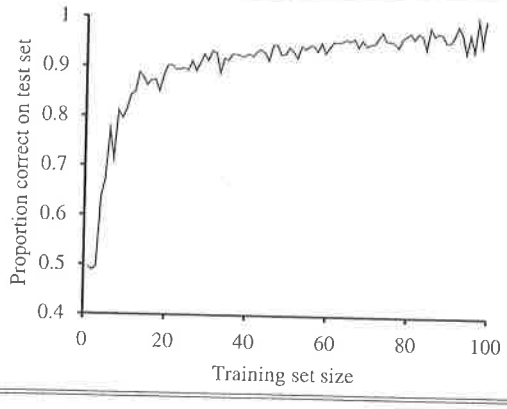

Figure 18.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

a test set. We learn a hypothesis h with the training set and measure its accuracy with the test set. We do this starting with a training set of size 1 and increasing one at a time up to size 99. For each size we actually repeat the process of randomly splitting 20 times, and average the results of the 20 trials. The curve shows that as the training set size grows, the accuracy increases. (For this reason, learning curves are also called **happy graphs**.) In this graph we reach 95% accuracy, and it looks like the curve might continue to increase with more data.

18.3.4 Choosing attribute tests

The greedy search used in decision tree learning is designed to approximately 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, each of which are all positive or all negative and thus will be leaves of the tree. 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.

ENTROPY

All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the IMPORTANCE function of Figure 18.5. We will use the notion of information gain, which is defined in terms of **entropy**, the fundamental quantity in information theory (Shannon and Weaver, 1949).

Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy. A random variable with only one value—a coin that always comes up heads—has no uncertainty and thus its entropy is defined as zero; thus, we gain no information by observing its value. A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and we will soon show that this counts as "1 bit" of entropy. The roll of a fair *four*-sided die has 2 bits of entropy, because it takes two bits to describe one of four equally probable choices. Now consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coin—if we guess heads we'll be wrong only 1% of the time—so we would like it to have an entropy measure that is close to zero, but

positive. In general, the entropy of a random variable V with values v_k, each with probability $P(v_k)$, is defined as

$$\text{Entropy:} \quad H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k) \; .$$

We can check that the entropy of a fair coin flip is indeed 1 bit:

$$H(Fair) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1 \; .$$

If the coin is loaded to give 99% heads, we get

$$H(Loaded) = -(0.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits.}$$

It will help to define $B(q)$ as the entropy of a Boolean random variable that is true with probability q:

$$B(q) = -(q \log_2 q + (1-q) \log_2 (1-q)) \; .$$

Thus, $H(Loaded) = B(0.99) \approx 0.08$. Now let's get back to decision tree learning. If a training set contains p positive examples and n negative examples, then the entropy of the goal attribute on the whole set is

$$H(Goal) = B\left(\frac{p}{p+n}\right) \; .$$

The restaurant training set in Figure 18.3 has $p = n = 6$, so the corresponding entropy is $B(0.5)$ or exactly 1 bit. A test on a single attribute A might give us only part of this 1 bit. We can measure exactly how much by looking at the entropy remaining *after* the attribute test.

An attribute A with d distinct values divides the training set E into subsets $E_1, \ldots, E_d$. Each subset E_k has p_k positive examples and n_k negative examples, so if we go along that branch, we will need an additional $B(p_k/(p_k+n_k))$ bits of information to answer the question. A randomly chosen example from the training set has the kth value for the attribute with probability $(p_k + n_k)/(p+n)$, so the expected entropy remaining after testing attribute A is

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k+n_k}{p+n} B\left(\frac{p_k}{p_k+n_k}\right) \; .$$

INFORMATION GAIN

The **information gain** from the attribute test on A is the expected reduction in entropy:

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

In fact $Gain(A)$ is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 18.4, we have

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

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

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

18.3.5 Generalization and overfitting

On some problems, the DECISION-TREE-LEARNING algorithm will generate a large tree when there is actually no pattern to be found. Consider the problem of trying to predict whether the roll of a die will come up as 6 or not. Suppose that experiments are carried out with various dice and that the attributes describing each training example include the color of the die, its weight, the time when the roll was done, and whether the experimenters had their fingers crossed. If the dice are fair, the right thing to learn is a tree with a single node that says "no," But the DECISION-TREE-LEARNING algorithm will seize on any pattern it can find in the input. If it turns out that there are 2 rolls of a 7-gram blue die with fingers crossed and they both come out 6, then the algorithm may construct a path that predicts 6 in that case. This problem is called **overfitting**. A general phenomenon, overfitting occurs with all types of learners, even when the target function is not at all random. In Figure 18.1(b) and (c), we saw polynomial functions overfitting the data. Overfitting becomes more likely as the hypothesis space and the number of input attributes grows, and less likely as we increase the number of training examples.

OVERFITTING

DECISION TREE PRUNING

For decision trees, a technique called **decision tree pruning** combats overfitting. Pruning works by eliminating nodes that are not clearly relevant. We start with a full tree, as generated by DECISION-TREE-LEARNING. We then look at a test node that has only leaf nodes as descendants. If the test appears to be irrelevant—detecting only noise in the data—then we eliminate the test, replacing it with a leaf node. We repeat this process, considering each test with only leaf descendants, until each one has either been pruned or accepted as is.

The question is, how do we detect that a node is testing an irrelevant attribute? Suppose we are at a node consisting of p positive and n negative examples. If the attribute is irrelevant, we would expect that it would split the examples into subsets that each have roughly the same proportion of positive examples as the whole set, $p/(p+n)$, and so the information gain will be close to zero.[2] 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?

SIGNIFICANCE TEST

NULL HYPOTHESIS

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.

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 $v = n + p$ 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

[2] The gain will be strictly positive except for the unlikely case where all the proportions are *exactly* the same. (See Exercise 18.5.)

each subset, p_k and n_k, with the expected numbers, $\hat{p}_k$ and $\hat{n}_k$, assuming true irrelevance:

$$\hat{p}_k = p \times \frac{p_k + n_k}{p + n} \qquad \hat{n}_k = n \times \frac{p_k + n_k}{p + n} .$$

A convenient measure of the total deviation is given by

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k} .$$

Under the null hypothesis, the value of Δ is distributed according to the χ^2 (chi-squared) distribution with $v - 1$ degrees of freedom. We can use a χ^2 table or a standard statistical library routine to see if a particular Δ value confirms or rejects the null hypothesis. For example, consider the restaurant type attribute, with four values and thus three degrees of freedom. A value of $\Delta = 7.82$ or more would reject the null hypothesis at the 5% level (and a value of $\Delta = 11.35$ or more would reject at the 1% level). Exercise 18.8 asks you to extend the DECISION-TREE-LEARNING algorithm to implement this form of pruning, which is known as χ^2 **pruning**.

χ^2 PRUNING

With pruning, noise in the examples can be tolerated. Errors in the example's label (e.g., an example ($\mathbf{x}$, *Yes*) that should be ($\mathbf{x}$, *No*)) give a linear increase in prediction error, whereas errors in the descriptions of examples (e.g., $Price = \$$ when it was actually $Price = \$\$$) have an asymptotic effect that gets worse as the tree shrinks down to smaller sets. Pruned trees perform significantly better than unpruned trees when the data contain a large amount of noise. Also, the pruned trees are often much smaller and hence easier to understand.

One final warning: You might think that χ^2 pruning and information gain look similar, so why not combine them using an approach called **early stopping**—have the decision tree algorithm stop generating nodes when there is no good attribute to split on, rather than going to all the trouble of generating nodes and then pruning them away. The problem with early stopping is that it stops us from recognizing situations where there is no one good attribute, but there are combinations of attributes that are informative. For example, consider the XOR function of two binary attributes. If there are roughly equal number of examples for all four combinations of input values, then neither attribute will be informative, yet the correct thing to do is to split on one of the attributes (it doesn't matter which one), and then at the second level we will get splits that are informative. Early stopping would miss this, but generate-and-then-prune handles it correctly.

EARLY STOPPING

18.3.6 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 several, 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 example 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 18.9.

- **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, an attribute such as *ExactTime* has a different value for every example, which means each subset of examples is a singleton with a unique classification, and the information gain measure would have its highest value for this attribute. But choosing this split first is unlikely to yield the best tree. One solution is to use the **gain ratio** (Exercise 18.10). Another possibility is to allow a Boolean test of the form $A = v_k$, that is, picking out just one of the possible values for an attribute, leaving the remaining values to possibly be tested later in the tree.

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 methods exist for finding good split points: start by sorting the values of the attribute, and then consider only split points that are between two examples in sorted order that have different classifications, while keeping track of the running totals of positive and negative examples on each side of the split point. Splitting is 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 output value, such as the price of an apartment, then we need a **regression tree** rather than a classification tree. A regression tree has at each leaf a linear function of some subset of numerical attributes, rather than a single value. For example, the branch for two-bedroom apartments might end with a linear function of square footage, number of bathrooms, and average income for the neighborhood. The learning algorithm must decide when to stop splitting and begin applying linear regression (see Section 18.6) over the attributes.

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 thousands of 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 reason for 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 some other representations, such as neural networks.

18.4 Evaluating and Choosing the Best Hypothesis

STATIONARITY ASSUMPTION

We want to learn a hypothesis that fits the future data best. To make that precise we need to define "future data" and "best." We make the **stationarity assumption**: that there is a probability distribution over examples that remains stationary over time. Each example data point (before we see it) is a random variable E_j whose observed value $e_j = (x_j, y_j)$ is sampled from that distribution, and is independent of the previous examples:

$$\mathbf{P}(E_j|E_{j-1}, E_{j-2}, \ldots) = \mathbf{P}(E_j) \ ,$$

and each example has an identical prior probability distribution:

$$\mathbf{P}(E_j) = \mathbf{P}(E_{j-1}) = \mathbf{P}(E_{j-2}) = \cdots \ .$$

I.I.D.

Examples that satisfy these assumptions are called *independent and identically distributed* or **i.i.d.**. An i.i.d. assumption connects the past to the future; without some such connection, all bets are off—the future could be anything. (We will see later that learning can still occur if there are *slow* changes in the distribution.)

ERROR RATE

The next step is to define "best fit." We define the **error rate** of a hypothesis as the proportion of mistakes it makes—the proportion of times that $h(x) \neq y$ for an (x, y) example. Now, just because a hypothesis h has a low error rate on the training set does not mean that it will generalize well. A professor knows that an exam will not accurately evaluate students if they have already seen the exam questions. Similarly, to get an accurate evaluation of a hypothesis, we need to test it on a set of examples it has not seen yet. The simplest approach is the one we have seen already: randomly split the available data into a training set from which the learning algorithm produces h and a test set on which the accuracy of h is evaluated. This

HOLDOUT CROSS-VALIDATION

method, sometimes called **holdout cross-validation**, has the disadvantage that it fails to use all the available data; if we use half the data for the test set, then we are only training on half the data, and we may get a poor hypothesis. On the other hand, if we reserve only 10% of the data for the test set, then we may, by statistical chance, get a poor estimate of the actual accuracy.

K-FOLD CROSS-VALIDATION

We can squeeze more out of the data and still get an accurate estimate using a technique called **k-fold cross-validation**. The idea is that each example serves double duty—as training data and test data. First we split the data into k equal subsets. We then perform k rounds of learning; on each round $1/k$ of the data is held out as a test set and the remaining examples are used as training data. The average test set score of the k rounds should then be a better estimate than a single score. Popular values for k are 5 and 10—enough to give an estimate that is statistically likely to be accurate, at a cost of 5 to 10 times longer computation time. The extreme is $k = n$, also known as **leave-one-out cross-validation** or **LOOCV**.

LEAVE-ONE-OUT CROSS-VALIDATION

LOOCV

PEEKING

Despite the best efforts of statistical methodologists, users frequently invalidate their results by inadvertently **peeking** at the test data. Peeking can happen like this: A learning algorithm has various "knobs" that can be twiddled to tune its behavior—for example, various different criteria for choosing the next attribute in decision tree learning. The researcher generates hypotheses for various different settings of the knobs, measures their error rates on the test set, and reports the error rate of the best hypothesis. Alas, peeking has occurred! The

reason is that the hypothesis was selected *on the basis of its test set error rate*, so information about the test set has leaked into the learning algorithm.

Peeking is a consequence of using test-set performance to both *choose* a hypothesis and *evaluate* it. The way to avoid this is to *really* hold the test set out—lock it away until you are completely done with learning and simply wish to obtain an independent evaluation of the final hypothesis. (And then, if you don't like the results ... you have to obtain, and lock away, a completely new test set if you want to go back and find a better hypothesis.) If the test set is locked away, but you still want to measure performance on unseen data as a way of selecting a good hypothesis, then divide the available data (without the test set) into a training set and a **validation set**. The next section shows how to use validation sets to find a good tradeoff between hypothesis complexity and goodness of fit.

DATION SET

18.4.1 Model selection: Complexity versus goodness of fit

In Figure 18.1 (page 696) we showed that higher-degree polynomials can fit the training data better, but when the degree is too high they will overfit, and perform poorly on validation data. Choosing the degree of the polynomial is an instance of the problem of **model selection**. You can think of the task of finding the best hypothesis as two tasks: model selection defines the hypothesis space and then **optimization** finds the best hypothesis within that space.

:L SELECTION

IIZATION

In this section we explain how to select among models that are parameterized by *size*. For example, with polynomials we have $size = 1$ for linear functions, $size = 2$ for quadratics, and so on. For decision trees, the size could be the number of nodes in the tree. In all cases we want to find the value of the *size* parameter that best balances underfitting and overfitting to give the best test set accuracy.

An algorithm to perform model selection and optimization is shown in Figure 18.8. It is a **wrapper** that takes a learning algorithm as an argument (DECISION-TREE-LEARNING, for example). The wrapper enumerates models according to a parameter, *size*. For each size, it uses cross validation on *Learner* to compute the average error rate on the training and test sets. We start with the smallest, simplest models (which probably underfit the data), and iterate, considering more complex models at each step, until the models start to overfit. In Figure 18.9 we see typical curves: the training set error decreases monotonically (although there may in general be slight random variation), while the validation set error decreases at first, and then increases when the model begins to overfit. The cross-validation procedure picks the value of *size* with the lowest validation set error; the bottom of the U-shaped curve. We then generate a hypothesis of that *size*, using all the data (without holding out any of it). Finally, of course, we should evaluate the returned hypothesis on a separate test set.

'ER

This approach requires that the learning algorithm accept a parameter, *size*, and deliver a hypothesis of that size. As we said, for decision tree learning, the size can be the number of nodes. We can modify DECISION-TREE-LEARNER so that it takes the number of nodes as an input, builds the tree breadth-first rather than depth-first (but at each level it still chooses the highest gain attribute first), and stops when it reaches the desired number of nodes.

```
function CROSS-VALIDATION-WRAPPER(Learner, k, examples) returns a hypothesis
  local variables: errT, an array, indexed by size, storing training-set error rates
                   errV, an array, indexed by size, storing validation-set error rates
  for size = 1 to ∞ do
    errT[size], errV[size] ← CROSS-VALIDATION(Learner, size, k, examples)
    if errT has converged then do
      best_size ← the value of size with minimum errV[size]
      return Learner(best_size, examples)

function CROSS-VALIDATION(Learner, size, k, examples) returns two values:
        average training set error rate, average validation set error rate
  fold_errT ← 0; fold_errV ← 0
  for fold = 1 to k do
    training_set, validation_set ← PARTITION(examples, fold, k)
    h ← Learner(size, training_set)
    fold_errT ← fold_errT + ERROR-RATE(h, training_set)
    fold_errV ← fold_errV +ERROR-RATE(h, validation_set)
  return fold_errT/k, fold_errV/k
```

Figure 18.8 An algorithm to select the model that has the lowest error rate on validation data by building models of increasing complexity, and choosing the one with best empirical error rate on validation data. Here $errT$ means error rate on the training data, and $errV$ means error rate on the validation data. $Learner(size, examples)$ returns a hypothesis whose complexity is set by the parameter $size$, and which is trained on the $examples$. PARTITION($examples, fold, k$) splits $examples$ into two subsets: a validation set of size N/k and a training set with all the other examples. The split is different for each value of $fold$.

18.4.2 From error rates to loss

So far, we have been trying to minimize error rate. This is clearly better than maximizing error rate, but it is not the full story. Consider the problem of classifying email messages as spam or non-spam. It is worse to classify non-spam as spam (and thus potentially miss an important message) then to classify spam as non-spam (and thus suffer a few seconds of annoyance). So a classifier with a 1% error rate, where almost all the errors were classifying spam as non-spam, would be better than a classifier with only a 0.5% error rate, if most of those errors were classifying non-spam as spam. We saw in Chapter 16 that decision-makers should maximize expected utility, and utility is what learners should maximize as well. In machine learning it is traditional to express utilities by means of a **loss function**. The loss function $L(x, y, \hat{y})$ is defined as the amount of utility lost by predicting $h(x) = \hat{y}$ when the correct answer is $f(x) = y$:

LOSS FUNCTION

$$\begin{aligned} L(x, y, \hat{y}) &= \mathit{Utility}(\text{result of using } y \text{ given an input } x) \\ &\quad - \mathit{Utility}(\text{result of using } \hat{y} \text{ given an input } x) \end{aligned}$$

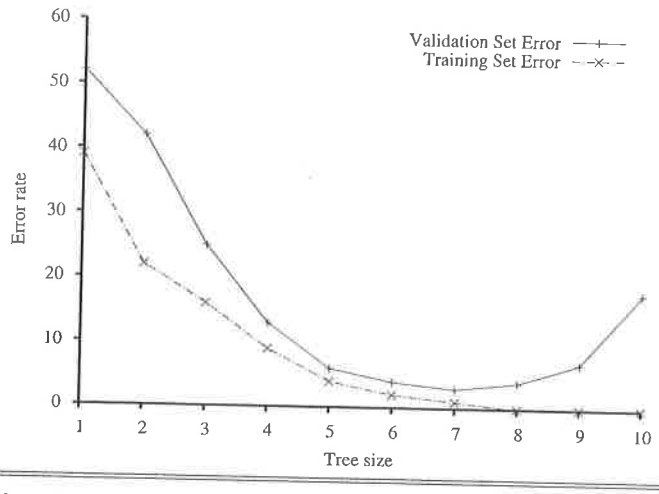

Figure 18.9 Error rates on training data (lower, dashed line) and validation data (upper, solid line) for different size decision trees. We stop when the training set error rate asymptotes, and then choose the tree with minimal error on the validation set; in this case the tree of size 7 nodes.

This is the most general formulation of the loss function. Often a simplified version is used, $L(y, \hat{y})$, that is independent of x. We will use the simplified version for the rest of this chapter, which means we can't say that it is worse to misclassify a letter from Mom than it is to misclassify a letter from our annoying cousin, but we can say it is 10 times worse to classify non-spam as spam than vice-versa:

$$L(spam, nospam) = 1, \quad L(nospam, spam) = 10\,.$$

Note that $L(y, y)$ is always zero; by definition there is no loss when you guess exactly right. For functions with discrete outputs, we can enumerate a loss value for each possible misclassification, but we can't enumerate all the possibilities for real-valued data. If $f(x)$ is 137.035999, we would be fairly happy with $h(x) = 137.036$, but just how happy should we be? In general small errors are better than large ones; two functions that implement that idea are the absolute value of the difference (called the L_1 loss), and the square of the difference (called the L_2 loss). If we are content with the idea of minimizing error rate, we can use the $L_{0/1}$ loss function, which has a loss of 1 for an incorrect answer and is appropriate for discrete-valued outputs:

Absolute value loss:	$L_1(y, \hat{y}) = \lvert y - \hat{y} \rvert$
Squared error loss:	$L_2(y, \hat{y}) = (y - \hat{y})^2$
0/1 loss:	$L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1

The learning agent can theoretically maximize its expected utility by choosing the hypothesis that minimizes expected loss over all input–output pairs it will see. It is meaningless to talk about this expectation without defining a prior probability distribution, $\mathbf{P}(X, Y)$ over examples. Let $\mathcal{E}$ be the set of all possible input–output examples. Then the expected **generalization loss** for a hypothesis h (with respect to loss function L) is

GENERALIZATION LOSS

$$GenLoss_L(h) = \sum_{(x,y)\in\mathcal{E}} L(y, h(x))\, P(x, y) ,$$

and the best hypothesis, h^*, is the one with the minimum expected generalization loss:

$$h^* = \operatorname*{argmin}_{h\in\mathcal{H}} GenLoss_L(h) .$$

EMPIRICAL LOSS

Because $P(x, y)$ is not known, the learning agent can only *estimate* generalization loss with **empirical loss** on a set of examples, E:

$$EmpLoss_{L,E}(h) = \frac{1}{N} \sum_{(x,y)\in E} L(y, h(x)) .$$

The estimated best hypothesis $\hat{h}^*$ is then the one with minimum empirical loss:

$$\hat{h}^* = \operatorname*{argmin}_{h\in\mathcal{H}} EmpLoss_{L,E}(h) .$$

There are four reasons why $\hat{h}^*$ may differ from the true function, f: unrealizability, variance, noise, and computational complexity. First, f may not be realizable—may not be in $\mathcal{H}$—or may be present in such a way that other hypotheses are preferred. Second, a learning algorithm will return different hypotheses for different sets of examples, even if those sets are drawn from the same true function f, and those hypotheses will make different predictions on new examples. The higher the variance among the predictions, the higher the probability of significant error. Note that even when the problem is realizable, there will still be random variance, but that variance decreases towards zero as the number of training examples increases. Third, f may be nondeterministic or **noisy**—it may return different values for $f(x)$ each time x occurs. By definition, noise cannot be predicted; in many cases, it arises because the observed labels y are the result of attributes of the environment not listed in x. And finally, when $\mathcal{H}$ is complex, it can be computationally intractable to systematically search the whole hypothesis space. The best we can do is a local search (hill climbing or greedy search) that explores only part of the space. That gives us an approximation error. Combining the sources of error, we're left with an estimation of an approximation of the true function f.

NOISE

SMALL-SCALE LEARNING

LARGE-SCALE LEARNING

Traditional methods in statistics and the early years of machine learning concentrated on **small-scale learning**, where the number of training examples ranged from dozens to the low thousands. Here the generalization error mostly comes from the approximation error of not having the true f in the hypothesis space, and from estimation error of not having enough training examples to limit variance. In recent years there has been more emphasis on **large-scale learning**, often with millions of examples. Here the generalization error is dominated by limits of computation: there is enough data and a rich enough model that we could find an h that is very close to the true f, but the computation to find it is too complex, so we settle for a sub-optimal approximation.

18.4.3 Regularization

In Section 18.4.1, we saw how to do model selection with cross-validation on model size. An alternative approach is to search for a hypothesis that directly minimizes the weighted sum of

empirical loss and the complexity of the hypothesis, which we will call the total cost:

$$Cost(h) = EmpLoss(h) + \lambda\, Complexity(h)$$
$$\hat{h}^* = \operatorname*{argmin}_{h \in \mathcal{H}} Cost(h)\ .$$

Here λ is a parameter, a positive number that serves as a conversion rate between loss and hypothesis complexity (which after all are not measured on the same scale). This approach combines loss and complexity into one metric, allowing us to find the best hypothesis all at once. Unfortunately we still need to do a cross-validation search to find the hypothesis that generalizes best, but this time it is with different values of λ rather than *size*. We select the value of λ that gives us the best validation set score.

REGULARIZATION

This process of explicitly penalizing complex hypotheses is called **regularization** (because it looks for a function that is more regular, or less complex). Note that the cost function requires us to make two choices: the loss function and the complexity measure, which is called a regularization function. The choice of regularization function depends on the hypothesis space. For example, a good regularization function for polynomials is the sum of the squares of the coefficients—keeping the sum small would guide us away from the wiggly polynomials in Figure 18.1(b) and (c). We will show an example of this type of regularization in Section 18.6.

FEATURE SELECTION

Another way to simplify models is to reduce the dimensions that the models work with. A process of **feature selection** can be performed to discard attributes that appear to be irrelevant. χ^2 pruning is a kind of feature selection.

MINIMUM DESCRIPTION LENGTH

It is in fact possible to have the empirical loss and the complexity measured on the same scale, without the conversion factor λ: they can both be measured in bits. First encode the hypothesis as a Turing machine program, and count the number of bits. Then count the number of bits required to encode the data, where a correctly predicted example costs zero bits and the cost of an incorrectly predicted example depends on how large the error is. The **minimum description length** or MDL hypothesis minimizes the total number of bits required. This works well in the limit, but for smaller problems there is a difficulty in that the choice of encoding for the program—for example, how best to encode a decision tree as a bit string—affects the outcome. In Chapter 20 (page 805), we describe a probabilistic interpretation of the MDL approach.

18.5 THE THEORY OF LEARNING

The main unanswered question in learning is this: How can we be sure that our learning algorithm has produced a hypothesis that will predict the correct value for previously unseen inputs? In formal terms, 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. In more recent decades, other questions have emerged: how many examples do we need to get a good h? What hypothesis space should we use? If the hypothesis space is very complex, can we even find the best h, or do we have to settle for a local maximum in the