

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.

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

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:

TRAINING SET

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

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

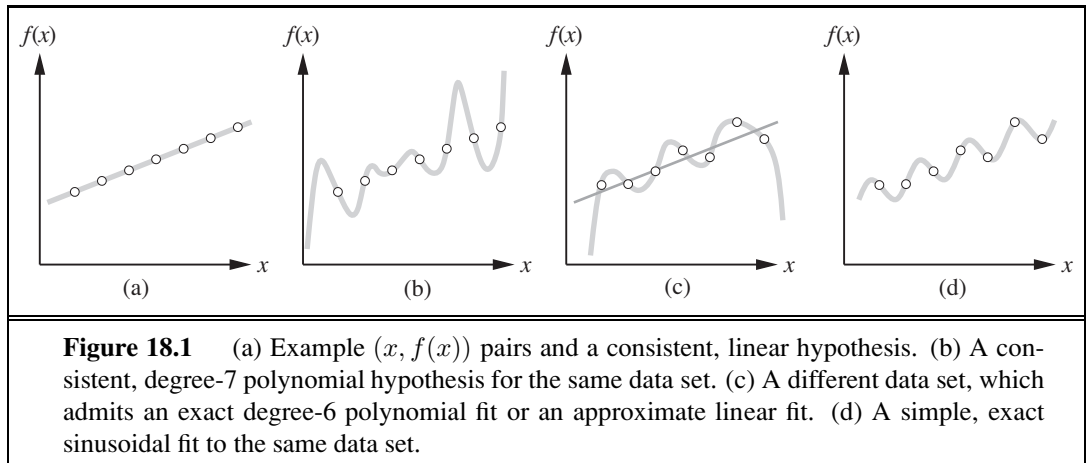
where each y_j was generated by an unknown function $y = f(x)$,
discover a function h that approximates the true function f .

HYPOTHESIS

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

TEST 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 j th example and we will use i to index the n attributes of each example. The elements of \mathbf{x}_j are written $x_{j,1}, x_{j,2}, \dots, x_{j,n}$.



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, $\mathbf{P}(Y | x)$.

CLASSIFICATION

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.)

REGRESSION

HYPOTHESIS SPACE

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?* 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.

CONSISTENT



OCKHAM'S RAZOR



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

REALIZABLE

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.

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:

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

By Bayes' rule this is equivalent to

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

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

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

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

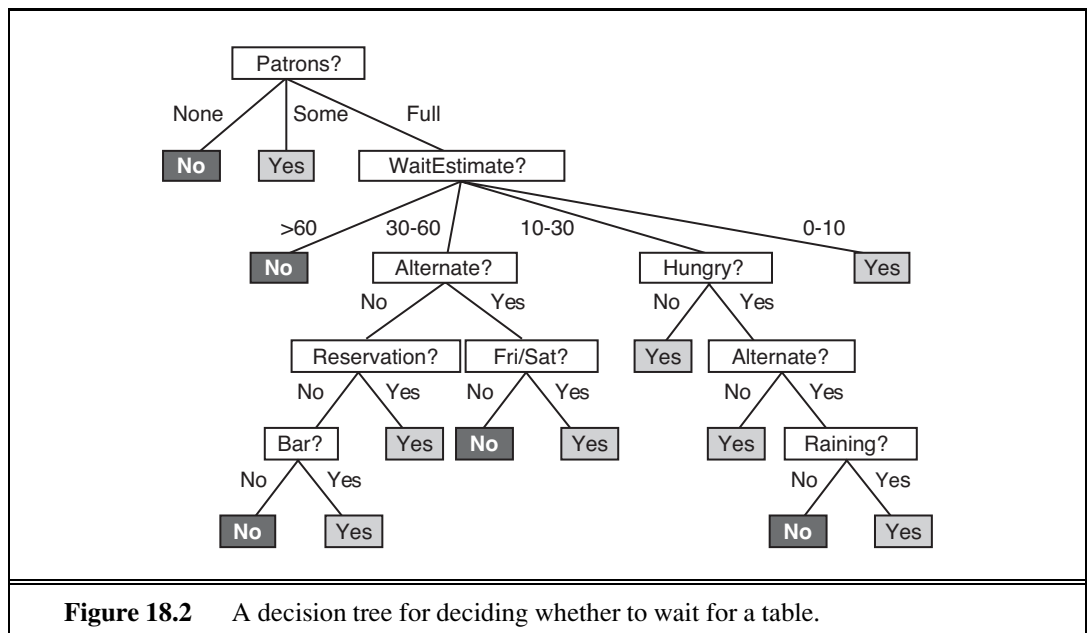
that any function in propositional logic can be expressed as a decision tree. As an example, the rightmost path in Figure 18.2 is

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

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



Example	Input 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	<i>Yes</i>	<i>No</i>	<i>No</i>	<i>Yes</i>	<i>Some</i>	<i>\$\$\$</i>	<i>No</i>	<i>Yes</i>	<i>French</i>	<i>0-10</i>	$y_1 = \text{Yes}$
x_2	<i>Yes</i>	<i>No</i>	<i>No</i>	<i>Yes</i>	<i>Full</i>	<i>\$</i>	<i>No</i>	<i>No</i>	<i>Thai</i>	<i>30-60</i>	$y_2 = \text{No}$
x_3	<i>No</i>	<i>Yes</i>	<i>No</i>	<i>No</i>	<i>Some</i>	<i>\$</i>	<i>No</i>	<i>No</i>	<i>Burger</i>	<i>0-10</i>	$y_3 = \text{Yes}$
x_4	<i>Yes</i>	<i>No</i>	<i>Yes</i>	<i>Yes</i>	<i>Full</i>	<i>\$</i>	<i>Yes</i>	<i>No</i>	<i>Thai</i>	<i>10-30</i>	$y_4 = \text{Yes}$
x_5	<i>Yes</i>	<i>No</i>	<i>Yes</i>	<i>No</i>	<i>Full</i>	<i>\$\$\$</i>	<i>No</i>	<i>Yes</i>	<i>French</i>	<i>>60</i>	$y_5 = \text{No}$
x_6	<i>No</i>	<i>Yes</i>	<i>No</i>	<i>Yes</i>	<i>Some</i>	<i>\$\$</i>	<i>Yes</i>	<i>Yes</i>	<i>Italian</i>	<i>0-10</i>	$y_6 = \text{Yes}$
x_7	<i>No</i>	<i>Yes</i>	<i>No</i>	<i>No</i>	<i>None</i>	<i>\$</i>	<i>Yes</i>	<i>No</i>	<i>Burger</i>	<i>0-10</i>	$y_7 = \text{No}$
x_8	<i>No</i>	<i>No</i>	<i>No</i>	<i>Yes</i>	<i>Some</i>	<i>\$\$</i>	<i>Yes</i>	<i>Yes</i>	<i>Thai</i>	<i>0-10</i>	$y_8 = \text{Yes}$
x_9	<i>No</i>	<i>Yes</i>	<i>Yes</i>	<i>No</i>	<i>Full</i>	<i>\$</i>	<i>Yes</i>	<i>No</i>	<i>Burger</i>	<i>>60</i>	$y_9 = \text{No}$
x_{10}	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>	<i>Full</i>	<i>\$\$\$</i>	<i>No</i>	<i>Yes</i>	<i>Italian</i>	<i>10-30</i>	$y_{10} = \text{No}$
x_{11}	<i>No</i>	<i>No</i>	<i>No</i>	<i>No</i>	<i>None</i>	<i>\$</i>	<i>No</i>	<i>No</i>	<i>Thai</i>	<i>0-10</i>	$y_{11} = \text{No}$
x_{12}	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>	<i>Full</i>	<i>\$</i>	<i>No</i>	<i>No</i>	<i>Burger</i>	<i>30-60</i>	$y_{12} = \text{Yes}$

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 (x_1, x_3, \dots); the negative examples are the ones in which it is false (x_2, x_5, \dots).

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-

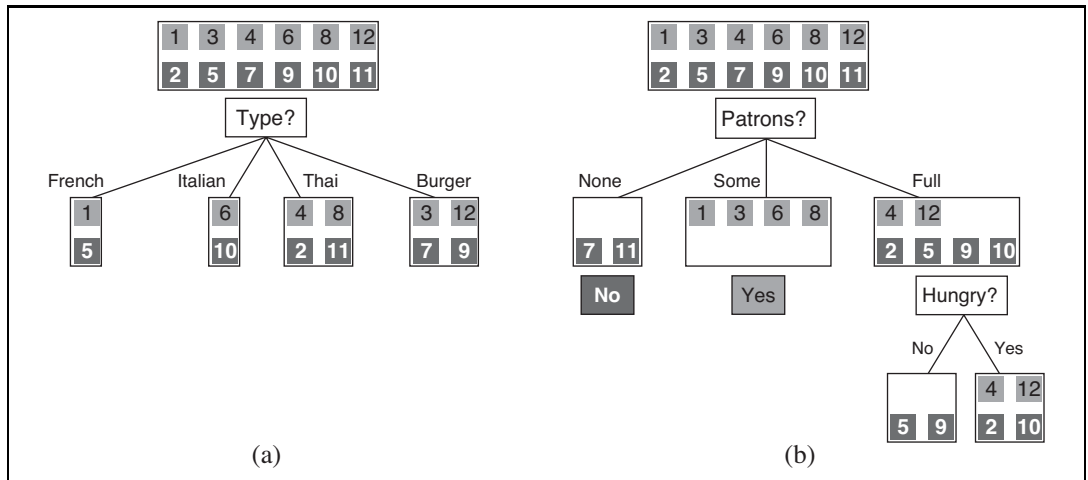


Figure 18.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of 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
     $A \leftarrow \operatorname{argmax}_{a \in \text{attributes}} \text{IMPORTANCE}(a, \text{examples})$ 
    tree  $\leftarrow$  a new decision tree with root test A
    for each value  $v_k$  of A do
      exs  $\leftarrow \{e : e \in \text{examples} \text{ and } e.A = v_k\}$ 
      subtree  $\leftarrow$  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.

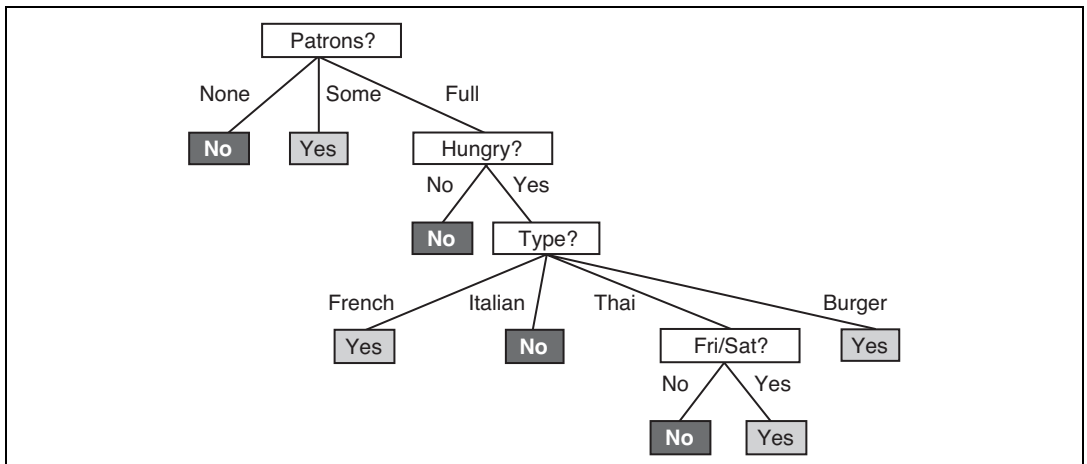
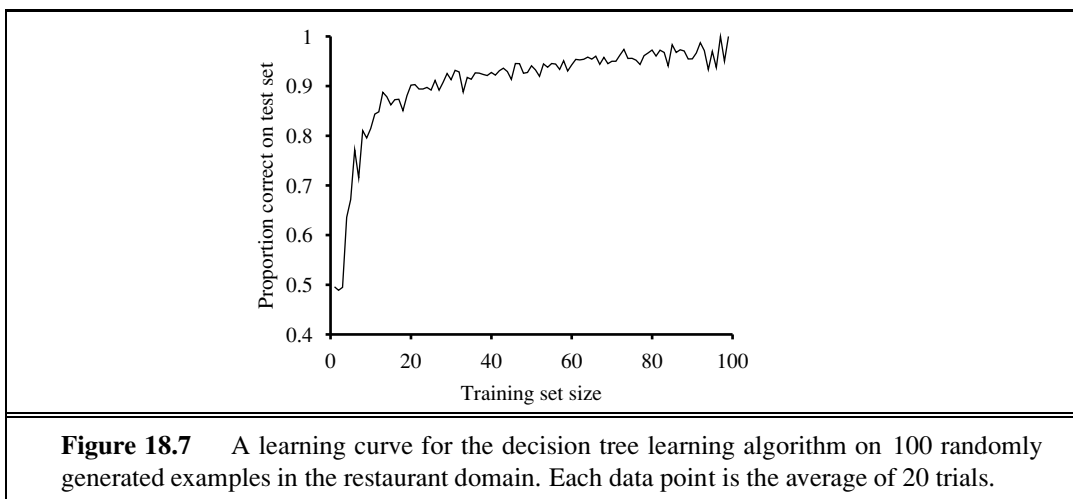


Figure 18.6 The decision tree induced from the 12-example training set.

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.

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



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.

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

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

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

$$H(\text{Fair}) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1 .$$

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

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

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, $H(\text{Loaded}) = B(0.99) \approx 0.08$. 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

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

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, \dots, 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 k th value for the attribute with probability $(p_k + n_k)/(p + n)$, so the expected entropy remaining after testing attribute A is

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

INFORMATION GAIN

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

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

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

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

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

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:

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

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., $\text{Price} = \$$ when it was actually $\text{Price} = \$\$$) 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.

EARLY STOPPING

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.

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.

GAIN RATIO

- **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.

SPLIT POINT

- **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.

REGRESSION TREE

- **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.

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:

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

and each example has an identical prior probability distribution:

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

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 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.

HOLDOUT
CROSS-VALIDATION

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.

VALIDATION 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.

MODEL SELECTION

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.

OPTIMIZATION

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.

WRAPPER

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  $\infty$  do
    errT[size], errV[size]  $\leftarrow$  CROSS-VALIDATION(Learner, size, k, examples)
    if errT has converged then do
      best_size  $\leftarrow$  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  $\leftarrow$  0; fold_errV  $\leftarrow$  0
  for fold = 1 to k do
    training_set, validation_set  $\leftarrow$  PARTITION(examples, fold, k)
    h  $\leftarrow$  Learner(size, training_set)
    fold_errT  $\leftarrow$  fold_errT + ERROR-RATE(h, training_set)
    fold_errV  $\leftarrow$  fold_errV + ERROR-RATE(h, validation_set)
  return fold_errT/k, fold_errV/k

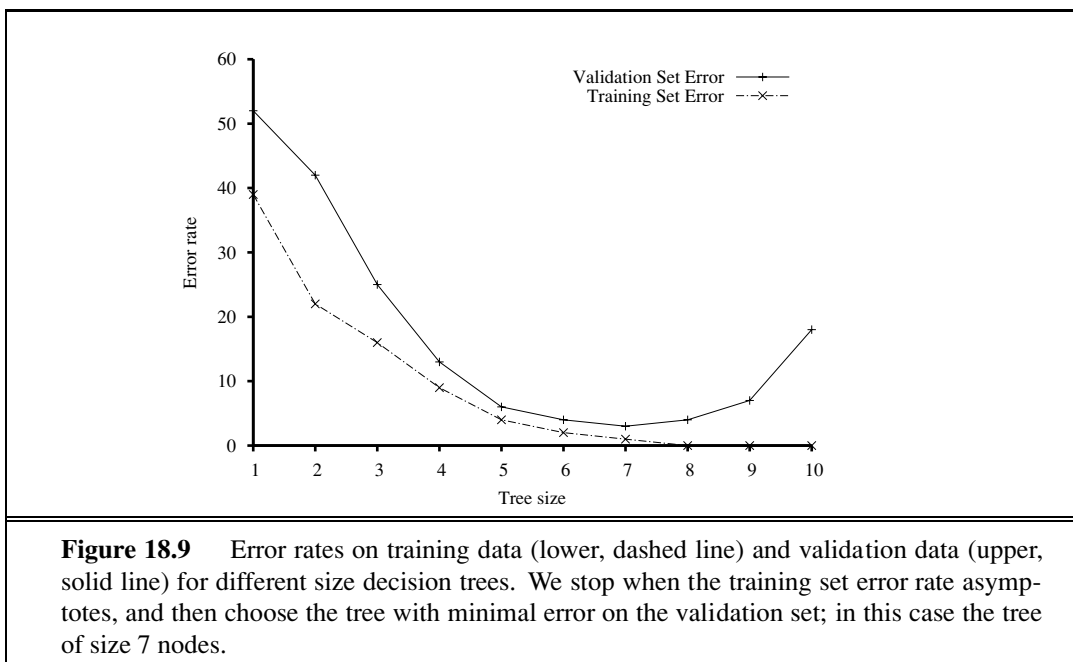
```

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) than 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 $f(x) = y$:

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



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:

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

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:

$$\text{Absolute value loss: } L_1(y, \hat{y}) = |y - \hat{y}|$$

$$\text{Squared error loss: } L_2(y, \hat{y}) = (y - \hat{y})^2$$

$$0/1 \text{ loss: } L_{0/1}(y, \hat{y}) = 0 \text{ if } y = \hat{y}, \text{ else } 1$$

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

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

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

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

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

EMPIRICAL LOSS

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

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

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

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

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.

SMALL-SCALE
LEARNINGLARGE-SCALE
LEARNING

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:

$$\begin{aligned} \text{Cost}(h) &= \text{EmpLoss}(h) + \lambda \text{Complexity}(h) \\ \hat{h}^* &= \underset{h \in \mathcal{H}}{\operatorname{argmin}} \text{Cost}(h) . \end{aligned}$$

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

space of hypotheses? How complex should h be? How do we avoid overfitting? This section examines these questions.

We'll start with the question of how many examples are needed for learning. We saw from the learning curve for decision tree learning on the restaurant problem (Figure 18.7 on page 703) that improves with more training data. Learning curves are useful, but they are specific to a particular learning algorithm on a particular problem. Are there some more general principles governing the number of examples needed in general? Questions like this are addressed by **computational learning theory**, which lies at the intersection of AI, statistics, and theoretical computer science. The underlying principle is that *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***. Any learning algorithm that returns hypotheses that are probably approximately correct is called a **PAC learning** algorithm; we can use this approach to provide bounds on the performance of various learning algorithms.

PAC-learning theorems, like all theorems, are logical consequences of axioms. When a *theorem* (as opposed to, say, a political pundit) states something about the future based on the past, the axioms have to provide the “juice” to make that connection. For PAC learning, the juice is provided by the stationarity assumption introduced on page 708, which says that future examples are going to be drawn from the same fixed distribution $\mathbf{P}(E) = \mathbf{P}(X, Y)$ as past examples. (Note that we do not have to know what distribution that is, just that it doesn't change.) In addition, to keep things simple, we will assume that the true function f is deterministic and is a member of the hypothesis class \mathcal{H} that is being considered.

The simplest PAC theorems deal with Boolean functions, for which the 0/1 loss is appropriate. The **error rate** of a hypothesis h , defined informally earlier, is defined formally here as the expected generalization error for examples drawn from the stationary distribution:

$$\text{error}(h) = \text{GenLoss}_{L_{0/1}}(h) = \sum_{x,y} L_{0/1}(y, h(x)) P(x, y) .$$

In other words, $\text{error}(h)$ is the probability that h misclassifies a new example. 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 ϵ is a small constant. We will show that we can find an N such 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 **ϵ -ball** around the true function f . The hypothesis space outside this ball is called \mathcal{H}_{bad} .

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

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

COMPUTATIONAL
LEARNING THEORY



PROBABLY
APPROXIMATELY
CORRECT
PAC LEARNING

ϵ -BALL

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

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

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

$$|\mathcal{H}|(1 - \epsilon)^N \leq \delta.$$

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

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

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 ϵ . In other words, it is probably approximately correct. The number of required examples, as a function of ϵ and δ , is called the **sample complexity** of the hypothesis space.

SAMPLE
COMPLEXITY

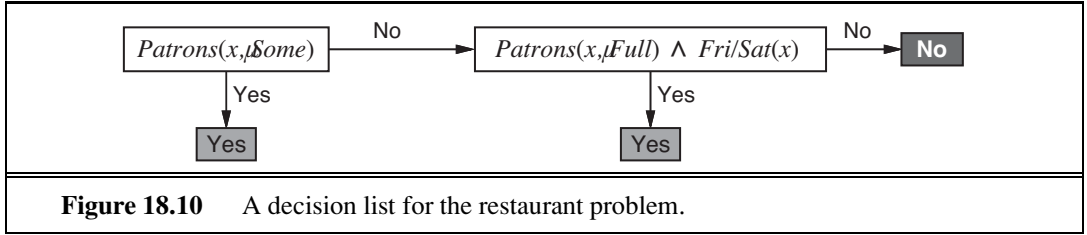
As we saw earlier, if \mathcal{H} is the set of all Boolean functions on n attributes, then $|\mathcal{H}| = 2^{2^n}$. Thus, the sample complexity of the space grows as 2^n . Because the number of possible examples is also 2^n , this suggests that PAC-learning in the class of all Boolean functions requires seeing all, or nearly all, of the possible examples. A moment's thought reveals the reason for this: \mathcal{H} contains enough hypotheses to classify any given set of examples in all possible ways. In particular, for any set of N examples, the set of hypotheses consistent with those examples contains equal numbers of hypotheses that predict x_{N+1} to be positive and hypotheses that predict x_{N+1} to be negative.

To obtain real generalization to unseen examples, then, it seems we need to restrict the hypothesis space \mathcal{H} in some way; but of course, if we do restrict the space, we might eliminate the true function altogether. There are three ways to escape this dilemma. The first, which we will cover in Chapter 19, is to bring prior knowledge to bear on the problem. The second, which we introduced in Section 18.4.3, is to insist that the algorithm return not just any consistent hypothesis, but preferably a simple one (as is done in decision tree learning). In cases where finding simple consistent hypotheses is tractable, the sample complexity results are generally better than for analyses based only on consistency. The third escape, which we pursue next, is to focus on learnable subsets of the entire hypothesis space of Boolean functions. This approach relies on the assumption that the restricted language contains a hypothesis h that is close enough to the true function f ; the benefits are that the restricted hypothesis space allows for effective generalization and is typically easier to search. We now examine one such restricted language in more detail.

18.5.1 PAC learning example: Learning decision lists

DECISION LISTS

We now show how to apply PAC learning to a new hypothesis space: **decision lists**. A decision list 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. Decision lists resemble decision trees, but their overall structure is simpler: they branch only in one



direction. In contrast, the individual tests are more complex. Figure 18.10 shows a decision list that represents the following hypothesis:

$$WillWait \Leftrightarrow (Patrons = Some) \vee (Patrons = Full \wedge Fri/Sat).$$

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, 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 18.10 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, that any function in k -DL can be approximated accurately after training on 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 because 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 :

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

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


```

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

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

```

Figure 18.11 An algorithm for learning decision lists.

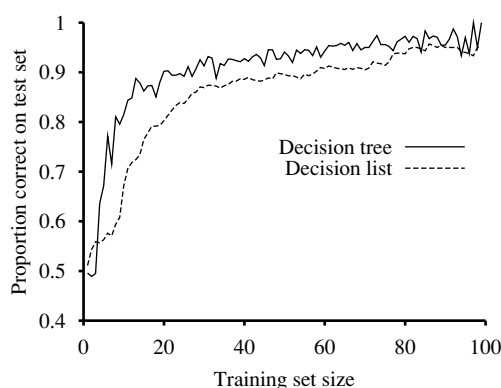


Figure 18.12 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

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 18.11.

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, 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 18.12 suggests.

18.6 REGRESSION AND CLASSIFICATION WITH LINEAR MODELS

Now it is time to move on from decision trees and lists to a different hypothesis space, one that has been used for hundred of years: the class of **linear functions** of continuous-valued

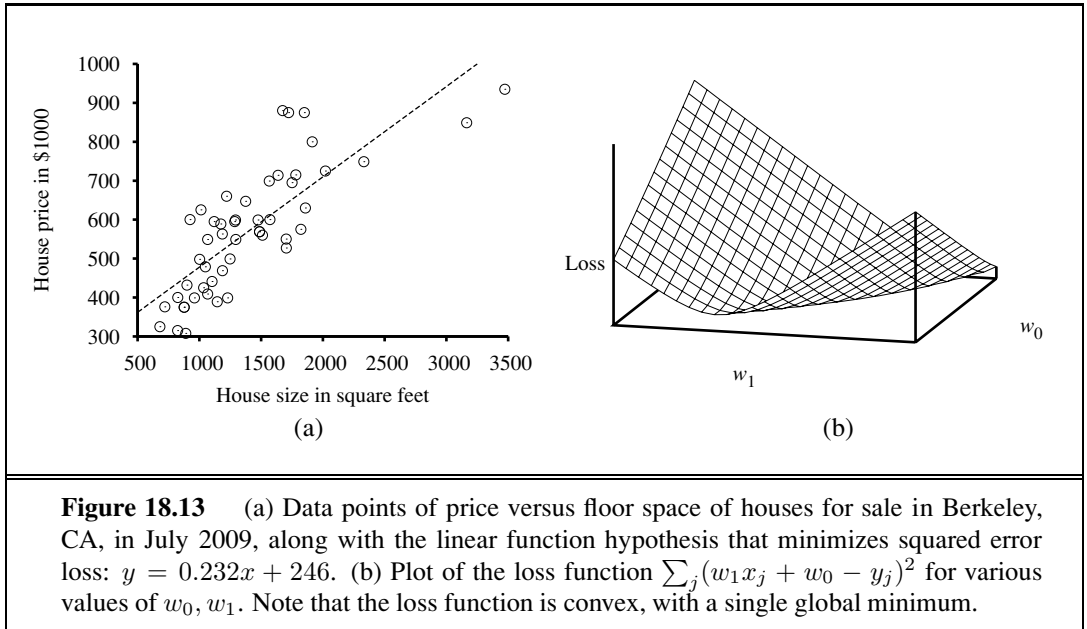


Figure 18.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1 . Note that the loss function is convex, with a single global minimum.

inputs. We'll start with the simplest case: regression with a univariate linear function, otherwise known as “fitting a straight line.” Section 18.6.2 covers the multivariate case. Sections 18.6.3 and 18.6.4 show how to turn linear functions into classifiers by applying hard and soft thresholds.

18.6.1 Univariate linear regression

A univariate linear function (a straight line) with input x and output y has the form $y = w_1 x + w_0$, where w_0 and w_1 are real-valued coefficients to be learned. We use the letter w because we think of the coefficients as **weights**; the value of y is changed by changing the relative weight of one term or another. We'll define \mathbf{w} to be the vector $[w_0, w_1]$, and define

$$h_{\mathbf{w}}(x) = w_1 x + w_0.$$

Figure 18.13(a) shows an example of a training set of n points in the x, y plane, each point representing the size in square feet and the price of a house offered for sale. The task of finding the $h_{\mathbf{w}}$ that best fits these data is called **linear regression**. To fit a line to the data, all we have to do is find the values of the weights $[w_0, w_1]$ that minimize the empirical loss. It is traditional (going back to Gauss³) to use the squared loss function, L_2 , summed over all the training examples:

$$\text{Loss}(h_{\mathbf{w}}) = \sum_{j=1}^N L_2(y_j, h_{\mathbf{w}}(x_j)) = \sum_{j=1}^N (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^N (y_j - (w_1 x_j + w_0))^2.$$

³ Gauss showed that if the y_j values have normally distributed noise, then the most likely values of w_1 and w_0 are obtained by minimizing the sum of the squares of the errors.

We would like to find $\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} \operatorname{Loss}(h_{\mathbf{w}})$. The sum $\sum_{j=1}^N (y_j - (w_1 x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^N (y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1} \sum_{j=1}^N (y_j - (w_1 x_j + w_0))^2 = 0. \quad (18.2)$$

These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N. \quad (18.3)$$

For the example in Figure 18.13(a), the solution is $w_1 = 0.232$, $w_0 = 246$, and the line with those weights is shown as a dashed line in the figure.

WEIGHT SPACE

Many forms of learning involve adjusting weights to minimize a loss, so it helps to have a mental picture of what's going on in **weight space**—the space defined by all possible settings of the weights. For univariate linear regression, the weight space defined by w_0 and w_1 is two-dimensional, so we can graph the loss as a function of w_0 and w_1 in a 3D plot (see Figure 18.13(b)). We see that the loss function is **convex**, as defined on page 133; this is true for *every* linear regression problem with an L_2 loss function, and implies that there are no local minima. In some sense that's the end of the story for linear models; if we need to fit lines to data, we apply Equation (18.3).⁴

GRADIENT DESCENT

To go beyond linear models, we will need to face the fact that the equations defining minimum loss (as in Equation (18.2)) will often have no closed-form solution. Instead, we will face a general optimization search problem in a continuous weight space. As indicated in Section 4.2 (page 129), such problems can be addressed by a hill-climbing algorithm that follows the **gradient** of the function to be optimized. In this case, because we are trying to minimize the loss, we will use **gradient descent**. We choose any starting point in weight space—here, a point in the (w_0, w_1) plane—and then move to a neighboring point that is downhill, repeating until we converge on the minimum possible loss:

$\mathbf{w} \leftarrow$ any point in the parameter space

loop until convergence **do**

for each w_i **in** \mathbf{w} **do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \operatorname{Loss}(\mathbf{w}) \quad (18.4)$$

LEARNING RATE

The parameter α , which we called the **step size** in Section 4.2, is usually called the **learning rate** when we are trying to minimize loss in a learning problem. It can be a fixed constant, or it can decay over time as the learning process proceeds.

For univariate regression, the loss function is a quadratic function, so the partial derivative will be a linear function. (The only calculus you need to know is that $\frac{\partial}{\partial x} x^2 = 2x$ and $\frac{\partial}{\partial x} x = 1$.) Let's first work out the partial derivatives—the slopes—in the simplified case of

⁴ With some caveats: the L_2 loss function is appropriate when there is normally-distributed noise that is independent of x ; all results rely on the stationarity assumption; etc.

only one training example, (x, y) :

$$\begin{aligned}\frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}) &= \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x))^2 \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x)) \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - (w_1 x + w_0)) ,\end{aligned}\tag{18.5}$$

applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} \text{Loss}(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)); \quad \frac{\partial}{\partial w_1} \text{Loss}(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \times x$$

Then, plugging this back into Equation (18.4), and folding the 2 into the unspecified learning rate α , we get the following learning rule for the weights:

$$w_0 \leftarrow w_0 + \alpha (y - h_{\mathbf{w}}(x)); \quad w_1 \leftarrow w_1 + \alpha (y - h_{\mathbf{w}}(x)) \times x$$

These updates make intuitive sense: if $h_{\mathbf{w}}(x) > y$, i.e., the output of the hypothesis is too large, reduce w_0 a bit, and reduce w_1 if x was a positive input but increase w_1 if x was a negative input.

The preceding equations cover one training example. For N training examples, we want to minimize the sum of the individual losses for each example. The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)); \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \times x_j .$$

BATCH GRADIENT
DESCENT

These updates constitute the **batch gradient descent** learning rule for univariate linear regression. Convergence to the unique global minimum is guaranteed (as long as we pick α small enough) but may be very slow: we have to cycle through all the training data for every step, and there may be many steps.

STOCHASTIC
GRADIENT DESCENT

There is another possibility, called **stochastic gradient descent**, where we consider only a single training point at a time, taking a step after each one using Equation (18.5). Stochastic gradient descent can be used in an online setting, where new data are coming in one at a time, or offline, where we cycle through the same data as many times as is necessary, taking a step after considering each single example. It is often faster than batch gradient descent. With a fixed learning rate α , however, it does not guarantee convergence; it can oscillate around the minimum without settling down. In some cases, as we see later, a schedule of decreasing learning rates (as in simulated annealing) does guarantee convergence.

18.6.2 Multivariate linear regression

MULTIVARIATE
LINEAR REGRESSION

We can easily extend to **multivariate linear regression** problems, in which each example \mathbf{x}_j is an n -element vector.⁵ Our hypothesis space is the set of functions of the form

$$h_{sw}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i} .$$

⁵ The reader may wish to consult Appendix A for a brief summary of linear algebra.

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{sw}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i}.$$

The best vector of weights, \mathbf{w}^* , minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j).$$

Multivariate linear regression is actually not much more complicated than the univariate case we just covered. Gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight w_i is

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}(y_j - h_{\mathbf{w}}(\mathbf{x}_j)). \quad (18.6)$$

It is also possible to solve analytically for the \mathbf{w} that minimizes loss. Let \mathbf{y} be the vector of outputs for the training examples, and \mathbf{X} be the **data matrix**, i.e., the matrix of inputs with one n -dimensional example per row. Then the solution

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

minimizes the squared error.

With univariate linear regression we didn't have to worry about overfitting. But with multivariate linear regression in high-dimensional spaces it is possible that some dimension that is actually irrelevant appears by chance to be useful, resulting in **overfitting**.

Thus, it is common to use **regularization** on multivariate linear functions to avoid overfitting. Recall that with regularization we minimize the total cost of a hypothesis, counting both the empirical loss and the complexity of the hypothesis:

$$\operatorname{Cost}(h) = \operatorname{EmpLoss}(h) + \lambda \operatorname{Complexity}(h).$$

For linear functions the complexity can be specified as a function of the weights. We can consider a family of regularization functions:

$$\operatorname{Complexity}(h_{\mathbf{w}}) = L_q(\mathbf{w}) = \sum_i |w_i|^q.$$

As with loss functions,⁶ with $q=1$ we have L_1 regularization, which minimizes the sum of the absolute values; with $q=2$, L_2 regularization minimizes the sum of squares. Which regularization function should you pick? That depends on the specific problem, but L_1 regularization has an important advantage: it tends to produce a **sparse model**. That is, it often sets many weights to zero, effectively declaring the corresponding attributes to be irrelevant—just as DECISION-TREE-LEARNING does (although by a different mechanism). Hypotheses that discard attributes can be easier for a human to understand, and may be less likely to overfit.

⁶ It is perhaps confusing that L_1 and L_2 are used for both loss functions and regularization functions. They need not be used in pairs: you could use L_2 loss with L_1 regularization, or vice versa.

DATA MATRIX

SPARSE MODEL

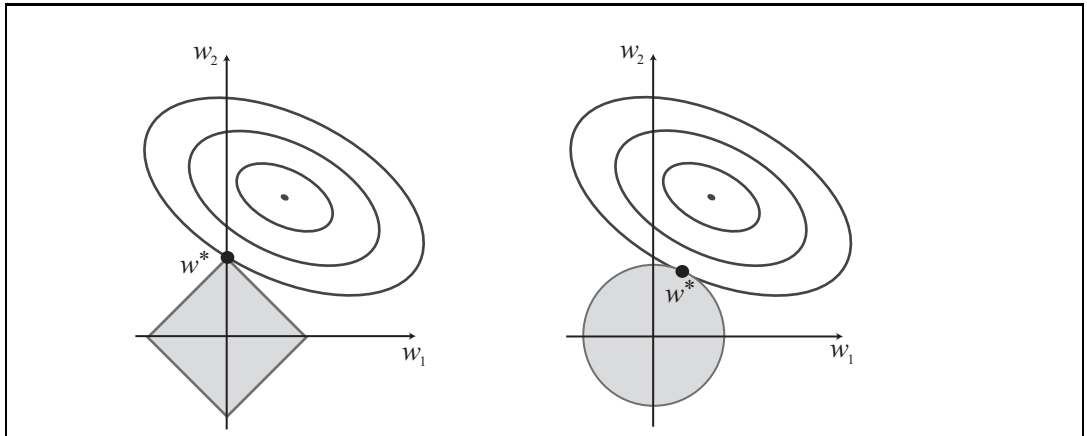


Figure 18.14 Why L_1 regularization tends to produce a sparse model. (a) With L_1 regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. (b) With L_2 regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

Figure 18.14 gives an intuitive explanation of why L_1 regularization leads to weights of zero, while L_2 regularization does not. Note that minimizing $Loss(\mathbf{w}) + \lambda Complexity(\mathbf{w})$ is equivalent to minimizing $Loss(\mathbf{w})$ subject to the constraint that $Complexity(\mathbf{w}) \leq c$, for some constant c that is related to λ . Now, in Figure 18.14(a) the diamond-shaped box represents the set of points \mathbf{w} in two-dimensional weight space that have L_1 complexity less than c ; our solution will have to be somewhere inside this box. The concentric ovals represent contours of the loss function, with the minimum loss at the center. We want to find the point in the box that is closest to the minimum; you can see from the diagram that, for an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum, just because the corners are pointy. And of course the corners are the points that have a value of zero in some dimension. In Figure 18.14(b), we've done the same for the L_2 complexity measure, which represents a circle rather than a diamond. Here you can see that, in general, there is no reason for the intersection to appear on one of the axes; thus L_2 regularization does not tend to produce zero weights. The result is that the number of examples required to find a good h is linear in the number of irrelevant features for L_2 regularization, but only logarithmic with L_1 regularization. Empirical evidence on many problems supports this analysis.

Another way to look at it is that L_1 regularization takes the dimensional axes seriously, while L_2 treats them as arbitrary. The L_2 function is spherical, which makes it rotationally invariant: Imagine a set of points in a plane, measured by their x and y coordinates. Now imagine rotating the axes by 45° . You'd get a different set of (x', y') values representing the same points. If you apply L_2 regularization before and after rotating, you get exactly the same point as the answer (although the point would be described with the new (x', y') coordinates). That is appropriate when the choice of axes really is arbitrary—when it doesn't matter whether your two dimensions are distances north and east; or distances north-east and

south-east. With L_1 regularization you'd get a different answer, because the L_1 function is not rotationally invariant. That is appropriate when the axes are not interchangeable; it doesn't make sense to rotate "number of bathrooms" 45° towards "lot size."

18.6.3 Linear classifiers with a hard threshold

Linear functions can be used to do classification as well as regression. For example, Figure 18.15(a) shows data points of two classes: earthquakes (which are of interest to seismologists) and underground explosions (which are of interest to arms control experts). Each point is defined by two input values, x_1 and x_2 , that refer to body and surface wave magnitudes computed from the seismic signal. Given these training data, the task of classification is to learn a hypothesis h that will take new (x_1, x_2) points and return either 0 for earthquakes or 1 for explosions.

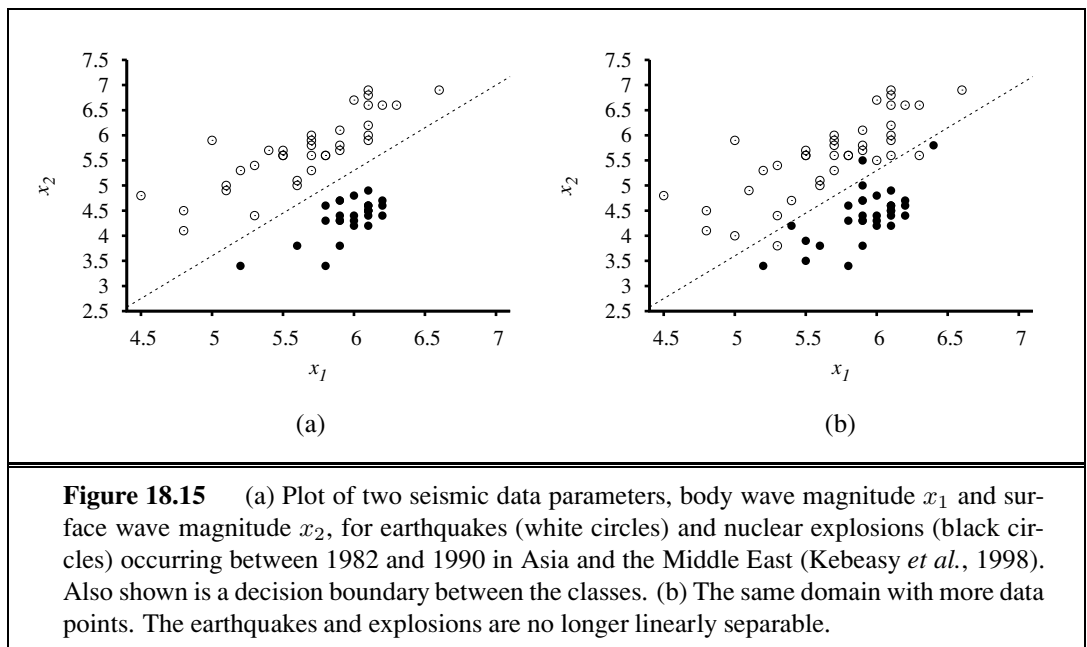


Figure 18.15 (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2 , for earthquakes (white circles) and nuclear explosions (black circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

DECISION
BOUNDARY

LINEAR SEPARATOR
LINEAR
SEPARABILITY

A **decision boundary** is a line (or a surface, in higher dimensions) that separates the two classes. In Figure 18.15(a), the decision boundary is a straight line. A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**. The linear separator in this case is defined by

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0.$$

The explosions, which we want to classify with value 1, are to the right of this line with higher values of x_1 and lower values of x_2 , so they are points for which $-4.9 + 1.7x_1 - x_2 > 0$, while earthquakes have $-4.9 + 1.7x_1 - x_2 < 0$. Using the convention of a dummy input $x_0 = 1$, we can write the classification hypothesis as

$$h_{\mathbf{w}}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \text{ and } 0 \text{ otherwise.}$$

THRESHOLD
FUNCTION

Alternatively, we can think of h as the result of passing the linear function $\mathbf{w} \cdot \mathbf{x}$ through a **threshold function**:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x}) \text{ where } \text{Threshold}(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise.}$$

The threshold function is shown in Figure 18.17(a).

Now that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ has a well-defined mathematical form, we can think about choosing the weights \mathbf{w} to minimize the loss. In Sections 18.6.1 and 18.6.2, we did this both in closed form (by setting the gradient to zero and solving for the weights) and by gradient descent in weight space. Here, we cannot do either of those things because the gradient is zero almost everywhere in weight space except at those points where $\mathbf{w} \cdot \mathbf{x} = 0$, and at those points the gradient is undefined.

There is, however, a simple weight update rule that converges to a solution—that is, a linear separator that classifies the data perfectly—provided the data are linearly separable. For a single example (\mathbf{x}, y) , we have

$$w_i \leftarrow w_i + \alpha (y - h_{\mathbf{w}}(\mathbf{x})) \times x_i \quad (18.7)$$

PERCEPTRON
LEARNING RULE

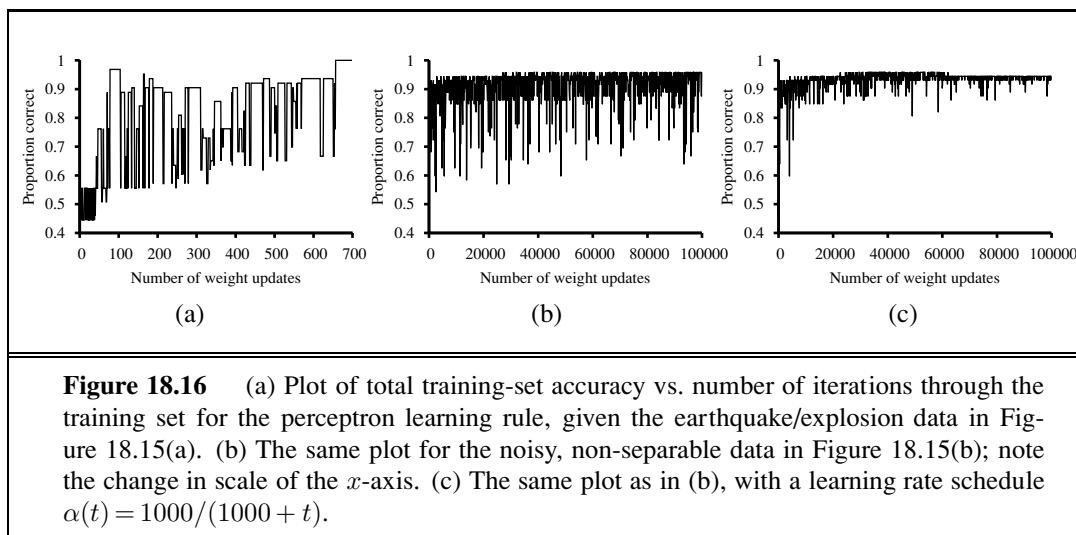
which is essentially identical to the Equation (18.6), the update rule for linear regression! This rule is called the **perceptron learning rule**, for reasons that will become clear in Section 18.7. Because we are considering a 0/1 classification problem, however, the behavior is somewhat different. Both the true value y and the hypothesis output $h_{\mathbf{w}}(\mathbf{x})$ are either 0 or 1, so there are three possibilities:

- If the output is correct, i.e., $y = h_{\mathbf{w}}(\mathbf{x})$, then the weights are not changed.
- If y is 1 but $h_{\mathbf{w}}(\mathbf{x})$ is 0, then w_i is *increased* when the corresponding input x_i is positive and *decreased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ bigger so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 1.
- If y is 0 but $h_{\mathbf{w}}(\mathbf{x})$ is 1, then w_i is *decreased* when the corresponding input x_i is positive and *increased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ smaller so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 0.

TRAINING CURVE

Typically the learning rule is applied one example at a time, choosing examples at random (as in stochastic gradient descent). Figure 18.16(a) shows a **training curve** for this learning rule applied to the earthquake/explosion data shown in Figure 18.15(a). A training curve measures the classifier performance on a fixed training set as the learning process proceeds on that same training set. The curve shows the update rule converging to a zero-error linear separator. The “convergence” process isn’t exactly pretty, but it always works. This particular run takes 657 steps to converge, for a data set with 63 examples, so each example is presented roughly 10 times on average. Typically, the variation across runs is very large.

We have said that the perceptron learning rule converges to a perfect linear separator when the data points are linearly separable, but what if they are not? This situation is all too common in the real world. For example, Figure 18.15(b) adds back in the data points left out by Kebeasy *et al.* (1998) when they plotted the data shown in Figure 18.15(a). In Figure 18.16(b), we show the perceptron learning rule failing to converge even after 10,000 steps: even though it hits the minimum-error solution (three errors) many times, the algorithm keeps changing the weights. In general, the perceptron rule may not converge to a



stable solution for fixed learning rate α , but if α decays as $O(1/t)$ where t is the iteration number, then the rule can be shown to converge to a minimum-error solution when examples are presented in a random sequence.⁷ It can also be shown that finding the minimum-error solution is NP-hard, so one expects that many presentations of the examples will be required for convergence to be achieved. Figure 18.16(b) shows the training process with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$: convergence is not perfect after 100,000 iterations, but it is much better than the fixed- α case.

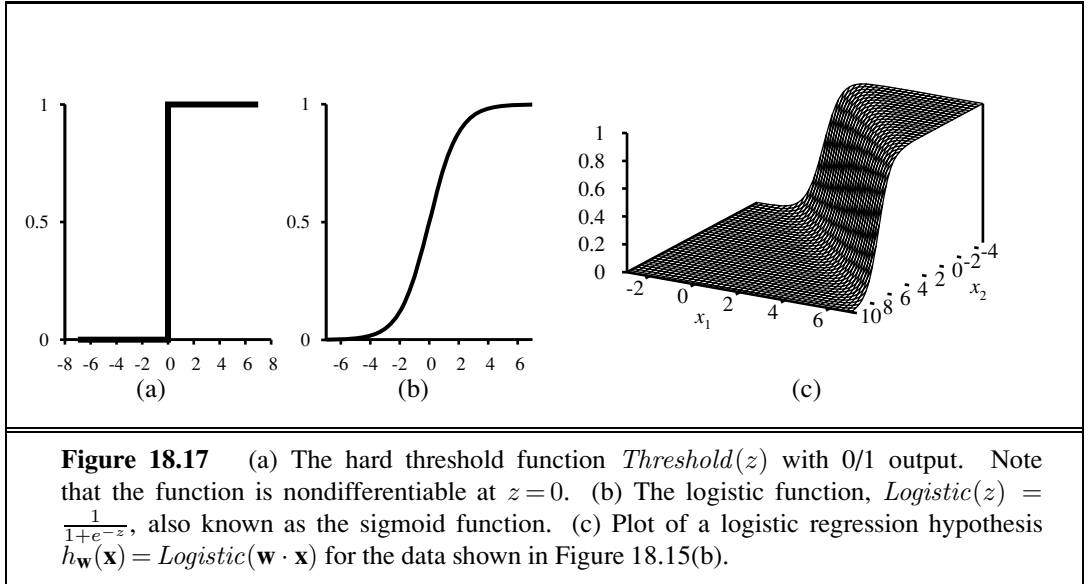
18.6.4 Linear classification with logistic regression

We have seen that passing the output of a linear function through the threshold function creates a linear classifier; yet the hard nature of the threshold causes some problems: the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is in fact a discontinuous function of its inputs and its weights; this makes learning with the perceptron rule a very unpredictable adventure. Furthermore, the linear classifier always announces a completely confident prediction of 1 or 0, even for examples that are very close to the boundary; in many situations, we really need more gradated predictions.

All of these issues can be resolved to a large extent by softening the threshold function—approximating the hard threshold with a continuous, differentiable function. In Chapter 14 (page 522), we saw two functions that look like soft thresholds: the integral of the standard normal distribution (used for the probit model) and the logistic function (used for the logit model). Although the two functions are very similar in shape, the logistic function

$$\text{Logistic}(z) = \frac{1}{1 + e^{-z}}$$

⁷ Technically, we require that $\sum_{t=1}^{\infty} \alpha(t) = \infty$ and $\sum_{t=1}^{\infty} \alpha^2(t) < \infty$. The decay $\alpha(t) = O(1/t)$ satisfies these conditions.



has more convenient mathematical properties. The function is shown in Figure 18.17(b). With the logistic function replacing the threshold function, we now have

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}.$$

An example of such a hypothesis for the two-input earthquake/explosion problem is shown in Figure 18.17(c). Notice that the output, being a number between 0 and 1, can be interpreted as a *probability* of belonging to the class labeled 1. The hypothesis forms a soft boundary in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary.

The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**. There is no easy closed-form solution to find the optimal value of \mathbf{w} with this model, but the gradient descent computation is straightforward. Because our hypotheses no longer output just 0 or 1, we will use the L_2 loss function; also, to keep the formulas readable, we'll use g to stand for the logistic function, with g' its derivative.

For a single example (\mathbf{x}, y) , the derivation of the gradient is the same as for linear regression (Equation (18.5)) up to the point where the actual form of h is inserted. (For this derivation, we will need the **chain rule**: $\partial g(f(x))/\partial x = g'(f(x)) \partial f(x)/\partial x$.) We have

$$\begin{aligned} \frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(\mathbf{x}))^2 \\ &= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(\mathbf{x})) \\ &= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i} \mathbf{w} \cdot \mathbf{x} \\ &= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i. \end{aligned}$$

LOGISTIC
REGRESSION

CHAIN RULE

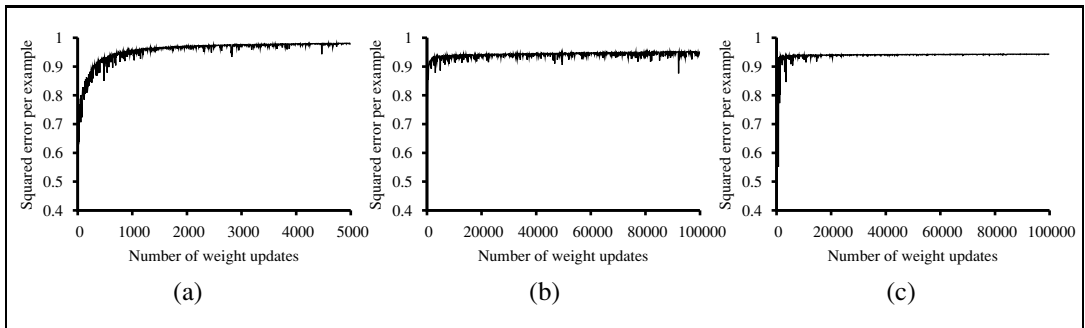


Figure 18.18 Repeat of the experiments in Figure 18.16 using logistic regression and squared error. The plot in (a) covers 5000 iterations rather than 1000, while (b) and (c) use the same scale.

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

so the weight update for minimizing the loss is

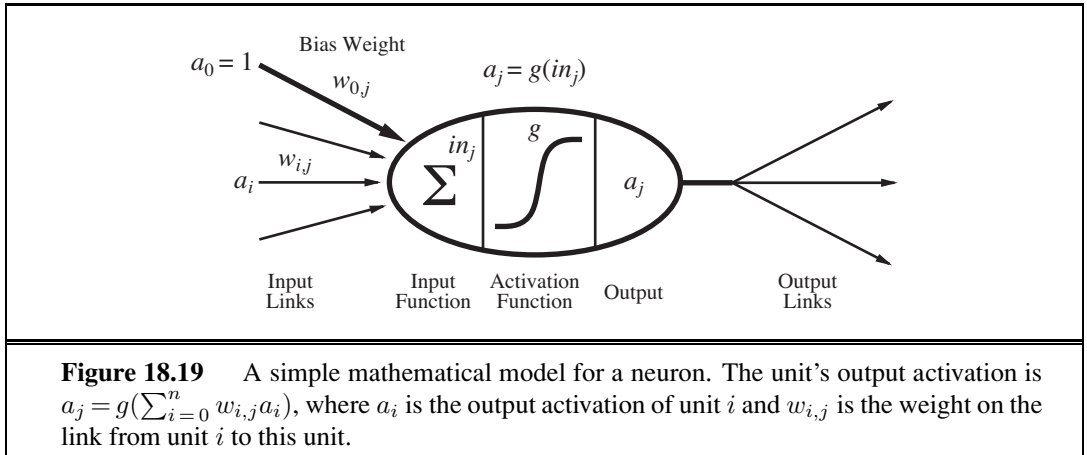
$$w_i \leftarrow w_i + \alpha (y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i. \quad (18.8)$$

Repeating the experiments of Figure 18.16 with logistic regression instead of the linear threshold classifier, we obtain the results shown in Figure 18.18. In (a), the linearly separable case, logistic regression is somewhat slower to converge, but behaves much more predictably. In (b) and (c), where the data are noisy and nonseparable, logistic regression converges far more quickly and reliably. These advantages tend to carry over into real-world applications and logistic regression has become one of the most popular classification techniques for problems in medicine, marketing and survey analysis, credit scoring, public health, and other applications.

18.7 ARTIFICIAL NEURAL NETWORKS

We turn now to what seems to be a somewhat unrelated topic: the brain. In fact, as we will see, the technical ideas we have discussed so far in this chapter turn out to be useful in building mathematical models of the brain's activity; conversely, thinking about the brain has helped in extending the scope of the technical ideas.

Chapter 1 touched briefly on the basic findings of neuroscience—in particular, the hypothesis that mental activity consists primarily of electrochemical activity in networks of brain cells called **neurons**. (Figure 1.2 on page 11 showed a schematic diagram of a typical neuron.) Inspired by this hypothesis, some of the earliest AI work aimed to create artificial **neural networks**. (Other names for the field include **connectionism**, **parallel distributed processing**, and **neural computation**.) Figure 18.19 shows a simple mathematical model of the neuron devised by McCulloch and Pitts (1943). Roughly speaking, it “fires” when a linear combination of its inputs exceeds some (hard or soft) threshold—that is, it implements



a linear classifier of the kind described in the preceding section. A neural network is just a collection of units connected together; the properties of the network are determined by its topology and the properties of the “neurons.”

Since 1943, much more detailed and realistic models have been developed, both for neurons and for larger systems in the brain, leading to the modern field of **computational neuroscience**. On the other hand, researchers in AI and statistics became interested in the more abstract properties of neural networks, such as their ability to perform distributed computation, to tolerate noisy inputs, and to learn. Although we understand now that other kinds of systems—including Bayesian networks—have these properties, neural networks remain one of the most popular and effective forms of learning system and are worthy of study in their own right.

18.7.1 Neural network structures

Neural networks are composed of nodes or **units** (see Figure 18.19) connected by directed **links**. A link from unit i to unit j serves to propagate the **activation** a_i from i to j .⁸ Each link also has a numeric **weight** $w_{i,j}$ associated with it, which determines the strength and sign of the connection. Just as in linear regression models, each unit has a dummy input $a_0 = 1$ with an associated weight $w_{0,j}$. Each unit j first computes a weighted sum of its inputs:

$$in_j = \sum_{i=0}^n w_{i,j} a_i .$$

Then it applies an **activation function** g to this sum to derive the output:

$$a_j = g(in_j) = g \left(\sum_{i=0}^n w_{i,j} a_i \right) . \quad (18.9)$$

⁸ A note on notation: for this section, we are forced to suspend our usual conventions. Input attributes are still indexed by i , so that an “external” activation a_i is given by input x_i ; but index j will refer to internal units rather than examples. Throughout this section, the mathematical derivations concern a single generic example \mathbf{x} , omitting the usual summations over examples to obtain results for the whole data set.

PERCEPTRON
SIGMOID
PERCEPTRON

The activation function g is typically either a hard threshold (Figure 18.17(a)), in which case the unit is called a **perceptron**, or a logistic function (Figure 18.17(b)), in which case the term **sigmoid perceptron** is sometimes used. Both of these nonlinear activation function ensure the important property that the entire network of units can represent a nonlinear function (see Exercise 18.22). As mentioned in the discussion of logistic regression (page 725), the logistic activation function has the added advantage of being differentiable.

FEED-FORWARD
NETWORK

RECURRENT
NETWORK

Having decided on the mathematical model for individual “neurons,” the next task is to connect them together to form a network. There are two fundamentally distinct ways to do this. A **feed-forward network** has connections only in one direction—that is, it forms a directed acyclic graph. Every node receives input from “upstream” nodes and delivers output to “downstream” nodes; there are no loops. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A **recurrent network**, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior. Moreover, the response of the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory. This makes them more interesting as models of the brain, but also more difficult to understand. This section will concentrate on feed-forward networks; some pointers for further reading on recurrent networks are given at the end of the chapter.

LAYERS

HIDDEN UNIT

Feed-forward networks are usually arranged in **layers**, such that each unit receives input only from units in the immediately preceding layer. In the next two subsections, we will look at single-layer networks, in which every unit connects directly from the network’s inputs to its outputs, and multilayer networks, which have one or more layers of **hidden units** that are not connected to the outputs of the network. So far in this chapter, we have considered only learning problems with a single output variable y , but neural networks are often used in cases where multiple outputs are appropriate. For example, if we want to train a network to add two input bits, each a 0 or a 1, we will need one output for the sum bit and one for the carry bit. Also, when the learning problem involves classification into more than two classes—for example, when learning to categorize images of handwritten digits—it is common to use one output unit for each class.

18.7.2 Single-layer feed-forward neural networks (perceptrons)

PERCEPTRON
NETWORK

A network with all the inputs connected directly to the outputs is called a **single-layer neural network**, or a **perceptron network**. Figure 18.20 shows a simple two-input, two-output perceptron network. With such a network, we might hope to learn the two-bit adder function, for example. Here are all the training data we will need:

x_1	x_2	y_3 (carry)	y_4 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The first thing to notice is that a perceptron network with m outputs is really m separate networks, because each weight affects only one of the outputs. Thus, there will be m separate training processes. Furthermore, depending on the type of activation function used, the training processes will be either the **perceptron learning rule** (Equation (18.7) on page 724) or gradient descent rule for the **logistic regression** (Equation (18.8) on page 727).

If you try either method on the two-bit-adder data, something interesting happens. Unit 3 learns the carry function easily, but unit 4 completely fails to learn the sum function. No, unit 4 is not defective! The problem is with the sum function itself. We saw in Section 18.6 that linear classifiers (whether hard or soft) can represent linear decision boundaries in the input space. This works fine for the carry function, which is a logical AND (see Figure 18.21(a)). The sum function, however, is an XOR (exclusive OR) of the two inputs. As Figure 18.21(c) illustrates, this function is not linearly separable so the perceptron cannot learn it.

The linearly separable functions constitute just a small fraction of all Boolean functions; Exercise 18.20 asks you to quantify this fraction. The inability of perceptrons to learn even such simple functions as XOR was a significant setback to the nascent neural network

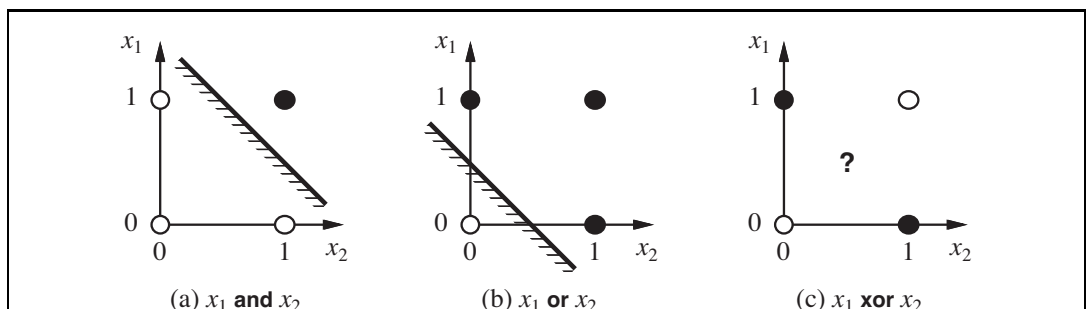
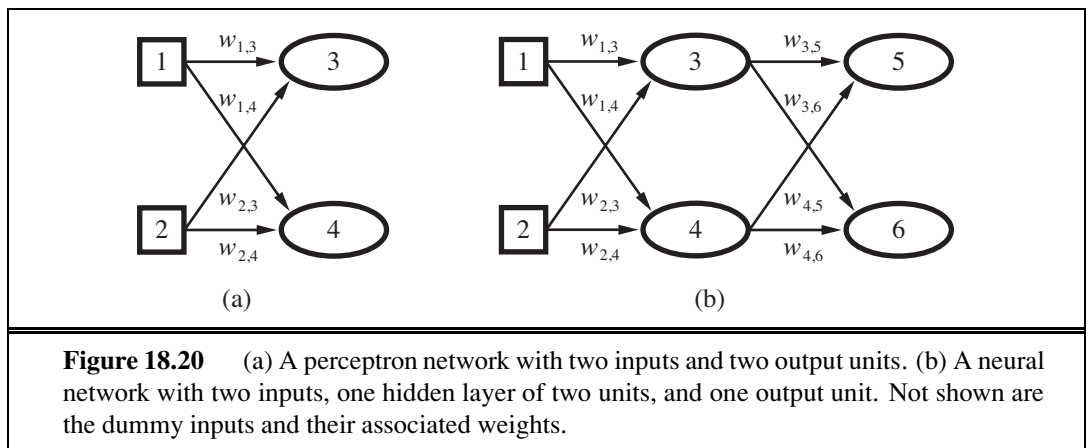
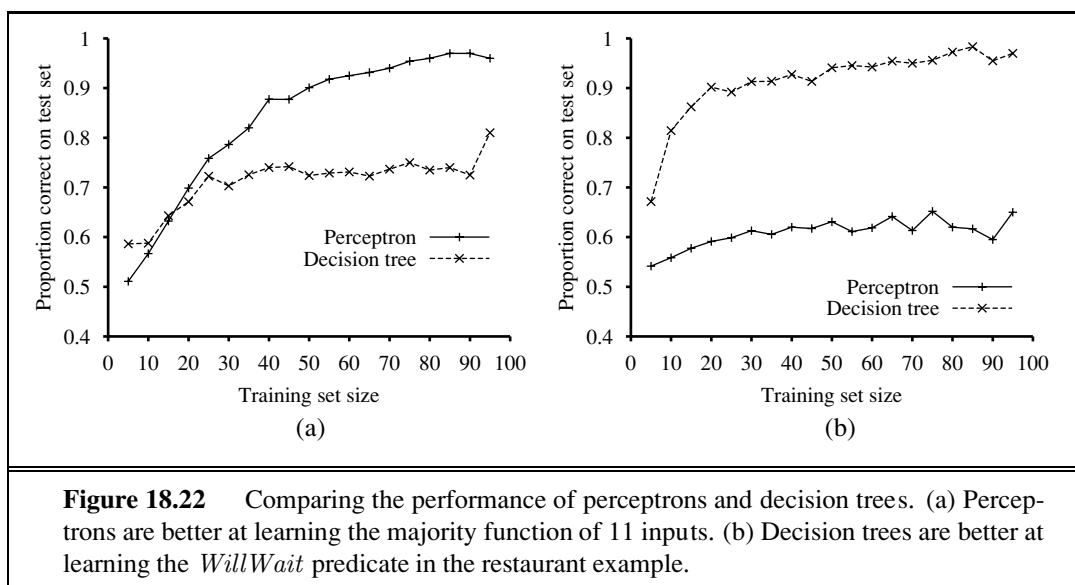


Figure 18.21 Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.



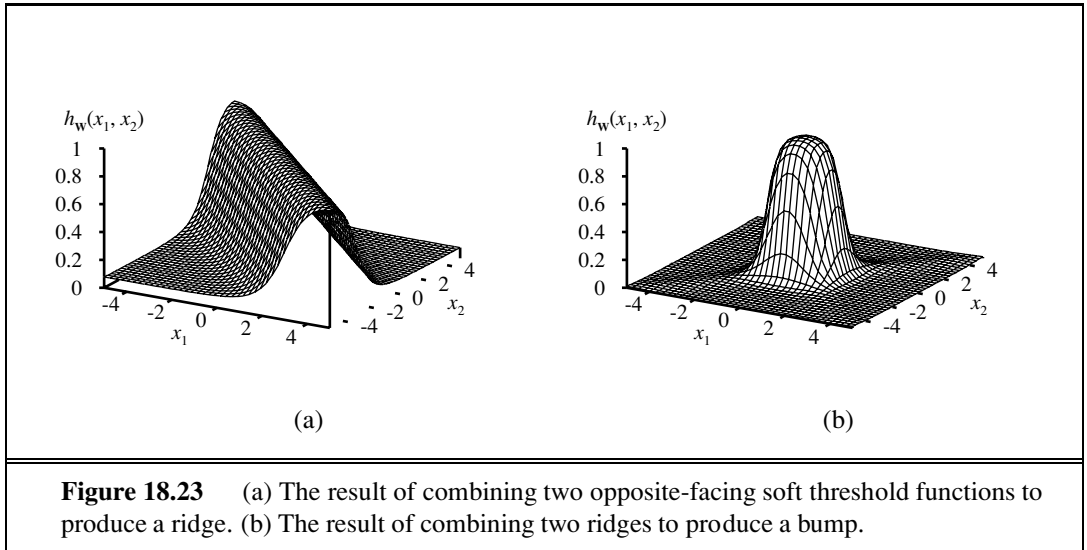
community in the 1960s. Perceptrons are far from useless, however. Section 18.6.4 noted that logistic regression (i.e., training a sigmoid perceptron) is even today a very popular and effective tool. Moreover, a perceptron can represent some quite “complex” Boolean functions very compactly. For example, the **majority function**, which outputs a 1 only if more than half of its n inputs are 1, can be represented by a perceptron with each $w_i = 1$ and with $w_0 = -n/2$. A decision tree would need exponentially many nodes to represent this function.

Figure 18.22 shows the learning curve for a perceptron on two different problems. On the left, we show the curve for learning the majority function with 11 Boolean inputs (i.e., the function outputs a 1 if 6 or more inputs are 1). As we would expect, the perceptron learns the function quite quickly, because the majority function is linearly separable. On the other hand, the decision-tree learner makes no progress, because the majority function is very hard (although not impossible) to represent as a decision tree. On the right, we have the restaurant example. The solution problem is easily represented as a decision tree, but is not linearly separable. The best plane through the data correctly classifies only 65%.

18.7.3 Multilayer feed-forward neural networks

(McCulloch and Pitts, 1943) were well aware that a single threshold unit would not solve all their problems. In fact, their paper proves that such a unit can represent the basic Boolean functions AND, OR, and NOT and then goes on to argue that any desired functionality can be obtained by connecting large numbers of units into (possibly recurrent) networks of arbitrary depth. The problem was that nobody knew how to train such networks.

This turns out to be an easy problem if we think of a network the right way: as a function $h_{\mathbf{w}}(\mathbf{x})$ parameterized by the weights \mathbf{w} . Consider the simple network shown in Figure 18.20(b), which has two input units, two hidden units, and two output unit. (In addition, each unit has a dummy input fixed at 1.) Given an input vector $\mathbf{x} = (x_1, x_2)$, the activations



of the input units are set to $(a_1, a_2) = (x_1, x_2)$. The output at unit 5 is given by

$$\begin{aligned}
 a_5 &= g(w_{0,5} + w_{3,5} a_3 + w_{4,5} a_4) \\
 &= g(w_{0,5} + w_{3,5} g(w_{0,3} + w_{1,3} a_1 + w_{2,3} a_2) + w_{4,5} g(w_{0,4} + w_{1,4} a_1 + w_{2,4} a_2)) \\
 &= g(w_{0,5} + w_{3,5} g(w_{0,3} + w_{1,3} x_1 + w_{2,3} x_2) + w_{4,5} g(w_{0,4} + w_{1,4} x_1 + w_{2,4} x_2)).
 \end{aligned}$$

Thus, we have the output expressed as a function of the inputs and the weights. A similar expression holds for unit 6. As long as we can calculate the derivatives of such expressions with respect to the weights, we can use the gradient-descent loss-minimization method to train the network. Section 18.7.4 shows exactly how to do this. And because the function represented by a network can be highly nonlinear—composed, as it is, of nested nonlinear soft threshold functions—we can see neural networks as a tool for doing **nonlinear regression**.

Before delving into learning rules, let us look at the ways in which networks generate complicated functions. First, remember that each unit in a sigmoid network represents a soft threshold in its input space, as shown in Figure 18.17(c) (page 726). With one hidden layer and one output layer, as in Figure 18.20(b), each output unit computes a soft-thresholded linear combination of several such functions. For example, by adding two opposite-facing soft threshold functions and thresholding the result, we can obtain a “ridge” function as shown in Figure 18.23(a). Combining two such ridges at right angles to each other (i.e., combining the outputs from four hidden units), we obtain a “bump” as shown in Figure 18.23(b).

With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.⁹ Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

⁹ The proof is complex, but the main point is that the required number of hidden units grows exponentially with the number of inputs. For example, $2^n/n$ hidden units are needed to encode all Boolean functions of n inputs.

18.7.4 Learning in multilayer networks

First, let us dispense with one minor complication arising in multilayer networks: interactions among the learning problems when the network has multiple outputs. In such cases, we should think of the network as implementing a vector function $\mathbf{h}_{\mathbf{w}}$ rather than a scalar function $h_{\mathbf{w}}$; for example, the network in Figure 18.20(b) returns a vector $[a_5, a_6]$. Similarly, the target output will be a vector \mathbf{y} . Whereas a perceptron network decomposes into m separate learning problems for an m -output problem, this decomposition fails in a multilayer network. For example, both a_5 and a_6 in Figure 18.20(b) depend on all of the input-layer weights, so updates to those weights will depend on errors in both a_5 and a_6 . Fortunately, this dependency is very simple in the case of any loss function that is *additive* across the components of the error vector $\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})$. For the L_2 loss, we have, for any weight w ,

$$\frac{\partial}{\partial w} \text{Loss}(\mathbf{w}) = \frac{\partial}{\partial w} |\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})|^2 = \frac{\partial}{\partial w} \sum_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w} (y_k - a_k)^2 \quad (18.10)$$

where the index k ranges over nodes in the output layer. Each term in the final summation is just the gradient of the loss for the k th output, computed as if the other outputs did not exist. Hence, we can decompose an m -output learning problem into m learning problems, provided we remember to add up the gradient contributions from each of them when updating the weights.

The major complication comes from the addition of hidden layers to the network. Whereas the error $\mathbf{y} - \mathbf{h}_{\mathbf{w}}$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the hidden nodes should have. Fortunately, it turns out that we can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient. First, we will describe the process with an intuitive justification; then, we will show the derivation.

At the output layer, the weight-update rule is identical to Equation (18.8). We have multiple output units, so let Err_k be the k th component of the error vector $\mathbf{y} - \mathbf{h}_{\mathbf{w}}$. We will also find it convenient to define a modified error $\Delta_k = Err_k \times g'(in_k)$, so that the weight-update rule becomes

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k. \quad (18.11)$$

To update the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes. Here is where we do the error back-propagation. The idea is that hidden node j is “responsible” for some fraction of the error Δ_k in each of the output nodes to which it connects. Thus, the Δ_k values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for the Δ values is the following:

$$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k. \quad (18.12)$$

```

function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
           network, a multilayer network with  $L$  layers, weights  $w_{i,j}$ , activation function  $g$ 
  local variables:  $\Delta$ , a vector of errors, indexed by network node

  repeat
    for each weight  $w_{i,j}$  in network do
       $w_{i,j} \leftarrow$  a small random number
    for each example (x, y) in examples do
      /* Propagate the inputs forward to compute the outputs */
      for each node  $i$  in the input layer do
         $a_i \leftarrow x_i$ 
      for  $\ell = 2$  to  $L$  do
        for each node  $j$  in layer  $\ell$  do
           $in_j \leftarrow \sum_i w_{i,j} a_i$ 
           $a_j \leftarrow g(in_j)$ 
      /* Propagate deltas backward from output layer to input layer */
      for each node  $j$  in the output layer do
         $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$ 
      for  $\ell = L - 1$  to  $1$  do
        for each node  $i$  in layer  $\ell$  do
           $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$ 
      /* Update every weight in network using deltas */
      for each weight  $w_{i,j}$  in network do
         $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ 
  until some stopping criterion is satisfied
  return network

```

Figure 18.24 The back-propagation algorithm for learning in multilayer networks.

Now the weight-update rule for the weights between the inputs and the hidden layer is essentially identical to the update rule for the output layer:

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j .$$

The back-propagation process can be summarized as follows:

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
 - Propagate the Δ values back to the previous layer.
 - Update the weights between the two layers.

The detailed algorithm is shown in Figure 18.24.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. The derivation is quite similar to the gradient calculation for logistic

regression (leading up to Equation (18.8) on page 727), except that we have to use the chain rule more than once.

Following Equation (18.10), we compute just the gradient for $Loss_k = (y_k - a_k)^2$ at the k th output. The gradient of this loss with respect to weights connecting the hidden layer to the output layer will be zero except for weights $w_{j,k}$ that connect to the k th output unit. For those weights, we have

$$\begin{aligned} \frac{\partial Loss_k}{\partial w_{j,k}} &= -2(y_k - a_k) \frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k) \frac{\partial g(in_k)}{\partial w_{j,k}} \\ &= -2(y_k - a_k) g'(in_k) \frac{\partial in_k}{\partial w_{j,k}} = -2(y_k - a_k) g'(in_k) \frac{\partial}{\partial w_{j,k}} \left(\sum_j w_{j,k} a_j \right) \\ &= -2(y_k - a_k) g'(in_k) a_j = -a_j \Delta_k, \end{aligned}$$

with Δ_k defined as before. To obtain the gradient with respect to the $w_{i,j}$ weights connecting the input layer to the hidden layer, we have to expand out the activations a_j and reapply the chain rule. We will show the derivation in gory detail because it is interesting to see how the derivative operator propagates back through the network:

$$\begin{aligned} \frac{\partial Loss_k}{\partial w_{i,j}} &= -2(y_k - a_k) \frac{\partial a_k}{\partial w_{i,j}} = -2(y_k - a_k) \frac{\partial g(in_k)}{\partial w_{i,j}} \\ &= -2(y_k - a_k) g'(in_k) \frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k \frac{\partial}{\partial w_{i,j}} \left(\sum_j w_{j,k} a_j \right) \\ &= -2\Delta_k w_{j,k} \frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k} \frac{\partial g(in_j)}{\partial w_{i,j}} \\ &= -2\Delta_k w_{j,k} g'(in_j) \frac{\partial in_j}{\partial w_{i,j}} \\ &= -2\Delta_k w_{j,k} g'(in_j) \frac{\partial}{\partial w_{i,j}} \left(\sum_i w_{i,j} a_i \right) \\ &= -2\Delta_k w_{j,k} g'(in_j) a_i = -a_i \Delta_j, \end{aligned}$$

where Δ_j is defined as before. Thus, we obtain the update rules obtained earlier from intuitive considerations. It is also clear that the process can be continued for networks with more than one hidden layer, which justifies the general algorithm given in Figure 18.24.

Having made it through (or skipped over) all the mathematics, let's see how a single-hidden-layer network performs on the restaurant problem. First, we need to determine the structure of the network. We have 10 attributes describing each example, so we will need 10 input units. Should we have one hidden layer or two? How many nodes in each layer? Should they be fully connected? There is no good theory that will tell us the answer. (See the next section.) As always, we can use cross-validation: try several different structures and see which one works best. It turns out that a network with one hidden layer containing four nodes is about right for this problem. In Figure 18.25, we show two curves. The first is a training curve showing the mean squared error on a given training set of 100 restaurant examples

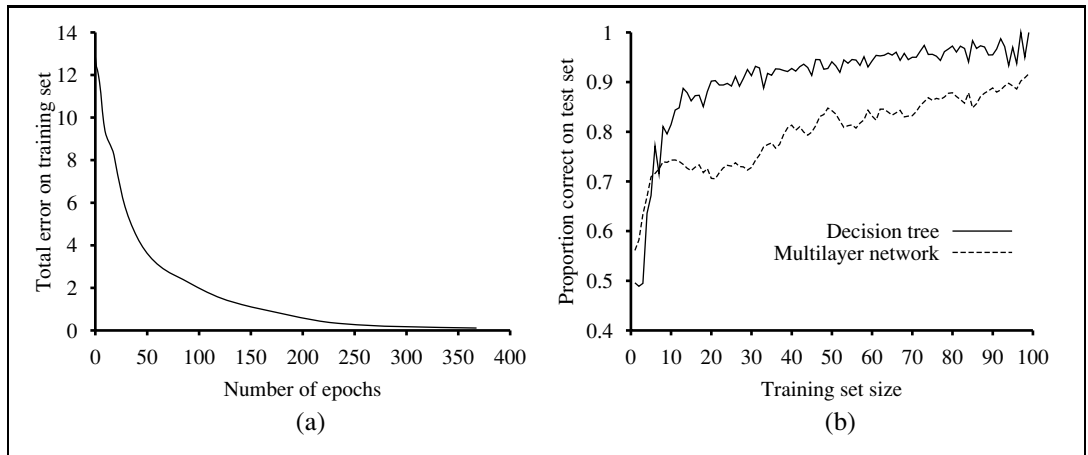


Figure 18.25 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better on the restaurant problem than back-propagation in a multilayer network.

during the weight-updating process. This demonstrates that the network does indeed converge to a perfect fit to the training data. The second curve is the standard learning curve for the restaurant data. The neural network does learn well, although not quite as fast as decision-tree learning; this is perhaps not surprising, because the data were generated from a simple decision tree in the first place.

Neural networks are capable of far more complex learning tasks of course, although it must be said that a certain amount of twiddling is needed to get the network structure right and to achieve convergence to something close to the global optimum in weight space. There are literally tens of thousands of published applications of neural networks. Section 18.11.1 looks at one such application in more depth.

18.7.5 Learning neural network structures

So far, we have considered the problem of learning weights, given a fixed network structure; just as with Bayesian networks, we also need to understand how to find the best network structure. If we choose a network that is too big, it will be able to memorize all the examples by forming a large lookup table, but will not necessarily generalize well to inputs that have not been seen before.¹⁰ In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters in the model. We saw this in Figure 18.1 (page 696), where the high-parameter models in (b) and (c) fit all the data, but might not generalize as well as the low-parameter models in (a) and (d).

If we stick to fully connected networks, the only choices to be made concern the number

¹⁰ It has been observed that very large networks *do* generalize well *as long as the weights are kept small*. This restriction keeps the activation values in the *linear* region of the sigmoid function $g(x)$ where x is close to zero. This, in turn, means that the network behaves like a linear function (Exercise 18.22) with far fewer parameters.

of hidden layers and their sizes. The usual approach is to try several and keep the best. The **cross-validation** techniques of Chapter 18 are needed if we are to avoid **peeking** at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

OPTIMAL BRAIN
DAMAGE

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies. The **optimal brain damage** algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

TILING

Several algorithms have been proposed for growing a larger network from a smaller one. One, the **tiling** algorithm, resembles decision-list learning. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

18.8 NONPARAMETRIC MODELS

Linear regression and neural networks use the training data to estimate a fixed set of parameters \mathbf{w} . That defines our hypothesis $h_{\mathbf{w}}(\mathbf{x})$, and at that point we can throw away the training data, because they are all summarized by \mathbf{w} . A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a **parametric model**.

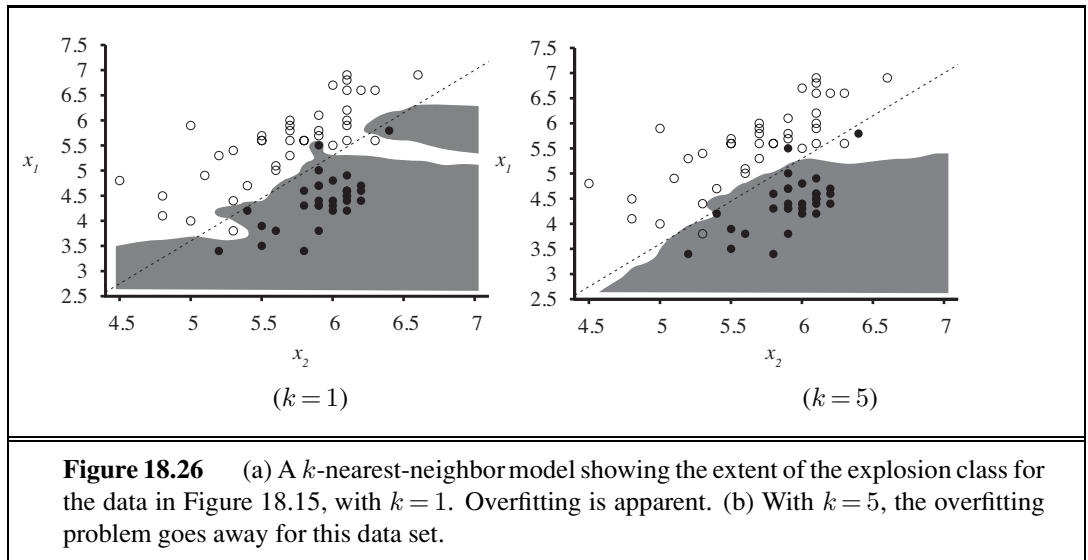
PARAMETRIC MODEL

No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs. When data sets are small, it makes sense to have a strong restriction on the allowable hypotheses, to avoid overfitting. But when there are thousands or millions or billions of examples to learn from, it seems like a better idea to let the data speak for themselves rather than forcing them to speak through a tiny vector of parameters. If the data say that the correct answer is a very wiggly function, we shouldn't restrict ourselves to linear or slightly wiggly functions.

NONPARAMETRIC
MODEL

A **nonparametric model** is one that cannot be characterized by a bounded set of parameters. For example, suppose that each hypothesis we generate simply retains within itself all of the training examples and uses all of them to predict the next example. Such a hypothesis family would be nonparametric because the effective number of parameters is unbounded—it grows with the number of examples. This approach is called **instance-based learning** or **memory-based learning**. The simplest instance-based learning method is **table lookup**: take all the training examples, put them in a lookup table, and then when asked for $h(\mathbf{x})$, see if \mathbf{x} is in the table; if it is, return the corresponding y . The problem with this method is that it does not generalize well: when \mathbf{x} is not in the table all it can do is return some default value.

INSTANCE-BASED
LEARNING
TABLE LOOKUP



18.8.1 Nearest neighbor models

NEAREST
NEIGHBORS

We can improve on table lookup with a slight variation: given a query \mathbf{x}_q , find the k examples that are *nearest* to \mathbf{x}_q . This is called **k -nearest neighbors** lookup. We'll use the notation $NN(k, \mathbf{x}_q)$ to denote the set of k nearest neighbors.

To do classification, first find $NN(k, \mathbf{x}_q)$, then take the plurality vote of the neighbors (which is the majority vote in the case of binary classification). To avoid ties, k is always chosen to be an odd number. To do regression, we can take the mean or median of the k neighbors, or we can solve a linear regression problem on the neighbors.

In Figure 18.26, we show the decision boundary of k -nearest-neighbors classification for $k = 1$ and 5 on the earthquake data set from Figure 18.15. Nonparametric methods are still subject to underfitting and overfitting, just like parametric methods. In this case 1-nearest neighbors is overfitting; it reacts too much to the black outlier in the upper right and the white outlier at (5.4, 3.7). The 5-nearest-neighbors decision boundary is good; higher k would underfit. As usual, cross-validation can be used to select the best value of k .

MINKOWSKI
DISTANCE

The very word “nearest” implies a distance metric. How do we measure the distance from a query point \mathbf{x}_q to an example point \mathbf{x}_j ? Typically, distances are measured with a **Minkowski distance** or L^p norm, defined as

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = \left(\sum_i |x_{j,i} - x_{q,i}|^p \right)^{1/p}.$$

HAMMING DISTANCE

With $p = 2$ this is Euclidean distance and with $p = 1$ it is Manhattan distance. With Boolean attribute values, the number of attributes on which the two points differ is called the **Hamming distance**. Often $p = 2$ is used if the dimensions are measuring similar properties, such as the width, height and depth of parts on a conveyor belt, and Manhattan distance is used if they are dissimilar, such as age, weight, and gender of a patient. Note that if we use the raw numbers from each dimension then the total distance will be affected by a change in scale in any dimension. That is, if we change dimension i from measurements in centimeters to

NORMALIZATION

miles while keeping the other dimensions the same, we'll get different nearest neighbors. To avoid this, it is common to apply **normalization** to the measurements in each dimension. One simple approach is to compute the mean μ_i and standard deviation σ_i of the values in each dimension, and rescale them so that $x_{j,i}$ becomes $(x_{j,i} - \mu_i)/\sigma_i$. A more complex metric known as the **Mahalanobis distance** takes into account the covariance between dimensions.

MAHALANOBIS
DISTANCE

In low-dimensional spaces with plenty of data, nearest neighbors works very well: we are likely to have enough nearby data points to get a good answer. But as the number of dimensions rises we encounter a problem: the nearest neighbors in high-dimensional spaces are usually not very near! Consider k -nearest-neighbors on a data set of N points uniformly distributed throughout the interior of an n -dimensional unit hypercube. We'll define the k -neighborhood of a point as the smallest hypercube that contains the k -nearest neighbors. Let ℓ be the average side length of a neighborhood. Then the volume of the neighborhood (which contains k points) is ℓ^n and the volume of the full cube (which contains N points) is 1. So, on average, $\ell^n = k/N$. Taking n th roots of both sides we get $\ell = (k/N)^{1/n}$.

CURSE OF
DIMENSIONALITY

To be concrete, let $k = 10$ and $N = 1,000,000$. In two dimensions ($n = 2$; a unit square), the average neighborhood has $\ell = 0.003$, a small fraction of the unit square, and in 3 dimensions ℓ is just 2% of the edge length of the unit cube. But by the time we get to 17 dimensions, ℓ is half the edge length of the unit hypercube, and in 200 dimensions it is 94%. This problem has been called the **curse of dimensionality**.

Another way to look at it: consider the points that fall within a thin shell making up the outer 1% of the unit hypercube. These are outliers; in general it will be hard to find a good value for them because we will be extrapolating rather than interpolating. In one dimension, these outliers are only 2% of the points on the unit line (those points where $x < .01$ or $x > .99$), but in 200 dimensions, over 98% of the points fall within this thin shell—almost all the points are outliers. You can see an example of a poor nearest-neighbors fit on outliers if you look ahead to Figure 18.28(b).

The $NN(k, \mathbf{x}_q)$ function is conceptually trivial: given a set of N examples and a query \mathbf{x}_q , iterate through the examples, measure the distance to \mathbf{x}_q from each one, and keep the best k . If we are satisfied with an implementation that takes $O(N)$ execution time, then that is the end of the story. But instance-based methods are designed for large data sets, so we would like an algorithm with sublinear run time. Elementary analysis of algorithms tells us that exact table lookup is $O(N)$ with a sequential table, $O(\log N)$ with a binary tree, and $O(1)$ with a hash table. We will now see that binary trees and hash tables are also applicable for finding nearest neighbors.

18.8.2 Finding nearest neighbors with k-d trees

K-D TREE

A balanced binary tree over data with an arbitrary number of dimensions is called a **k-d tree**, for k-dimensional tree. (In our notation, the number of dimensions is n , so they would be n -d trees. The construction of a k-d tree is similar to the construction of a one-dimensional balanced binary tree. We start with a set of examples and at the root node we split them along the i th dimension by testing whether $x_i \leq m$. We chose the value m to be the median of the examples along the i th dimension; thus half the examples will be in the left branch of the tree

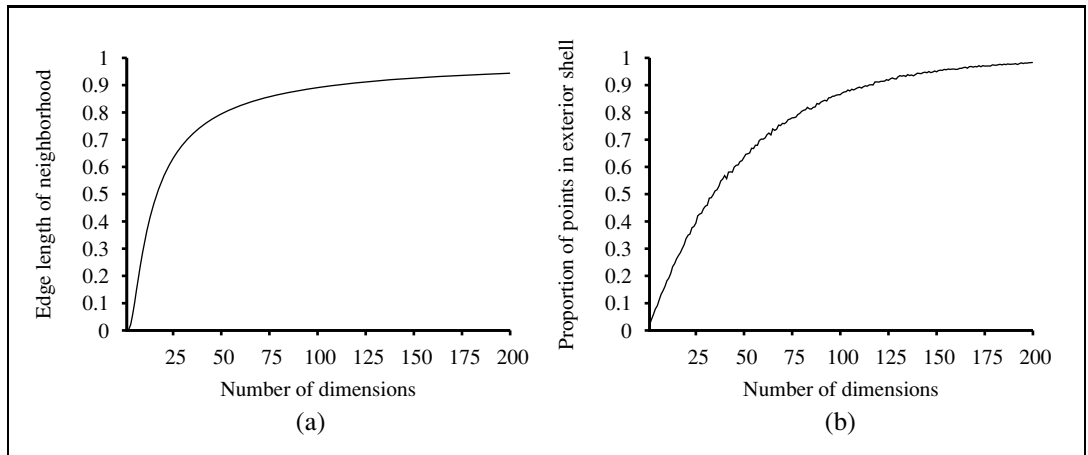


Figure 18.27 The curse of dimensionality: (a) The length of the average neighborhood for 10-nearest-neighbors in a unit hypercube with 1,000,000 points, as a function of the number of dimensions. (b) The proportion of points that fall within a thin shell consisting of the outer 1% of the hypercube, as a function of the number of dimensions. Sampled from 10,000 randomly distributed points.

and half in the right. We then recursively make a tree for the left and right sets of examples, stopping when there are fewer than two examples left. To choose a dimension to split on at each node of the tree, one can simply select dimension $i \bmod n$ at level i of the tree. (Note that we may need to split on any given dimension several times as we proceed down the tree.) Another strategy is to split on the dimension that has the widest spread of values.

Exact lookup from a k-d tree is just like lookup from a binary tree (with the slight complication that you need to pay attention to which dimension you are testing at each node). But nearest neighbor lookup is more complicated. As we go down the branches, splitting the examples in half, in some cases we can discard the other half of the examples. But not always. Sometimes the point we are querying for falls very close to the dividing boundary. The query point itself might be on the left hand side of the boundary, but one or more of the k nearest neighbors might actually be on the right-hand side. We have to test for this possibility by computing the distance of the query point to the dividing boundary, and then searching both sides if we can't find k examples on the left that are closer than this distance. Because of this problem, k-d trees are appropriate only when there are many more examples than dimensions, preferably at least 2^n examples. Thus, k-d trees work well with up to 10 dimensions with thousands of examples or up to 20 dimensions with millions of examples. If we don't have enough examples, lookup is no faster than a linear scan of the entire data set.

18.8.3 Locality-sensitive hashing

Hash tables have the potential to provide even faster lookup than binary trees. But how can we find nearest neighbors using a hash table, when hash codes rely on an *exact* match? Hash codes randomly distribute values among the bins, but we want to have near points grouped together in the same bin; we want a **locality-sensitive hash** (LSH).

We can't use hashes to solve $NN(k, \mathbf{x}_q)$ exactly, but with a clever use of randomized algorithms, we can find an *approximate* solution. First we define the **approximate near-neighbors** problem: given a data set of example points and a query point \mathbf{x}_q , find, with high probability, an example point (or points) that is near \mathbf{x}_q . To be more precise, we require that if there is a point \mathbf{x}_j that is within a radius r of \mathbf{x}_q , then with high probability the algorithm will find a point $\mathbf{x}_{j'}$ that is within distance cr of q . If there is no point within radius r then the algorithm is allowed to report failure. The values of c and “high probability” are parameters of the algorithm.

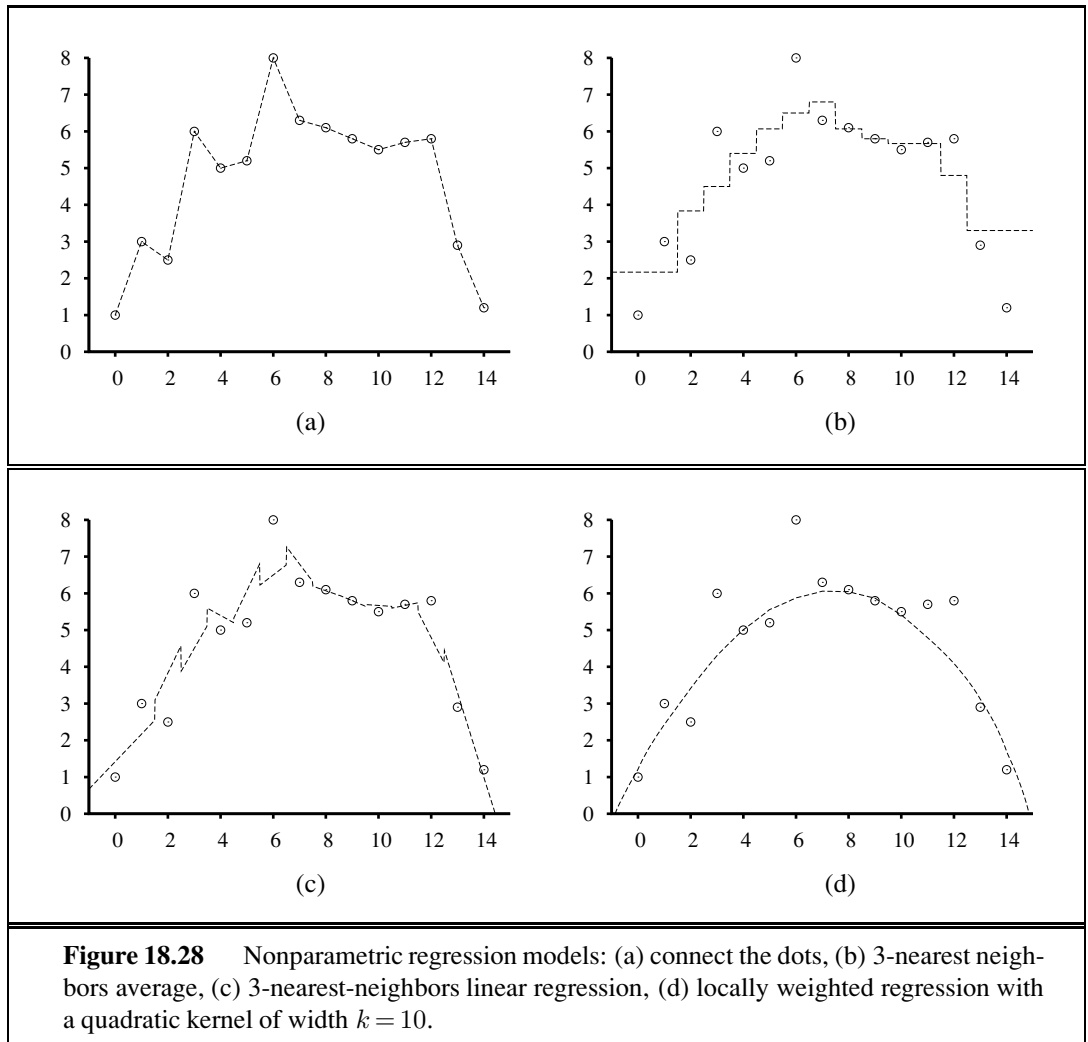
To solve approximate near neighbors, we will need a hash function $g(\mathbf{x})$ that has the property that, for any two points \mathbf{x}_j and $\mathbf{x}_{j'}$, the probability that they have the same hash code is small if their distance is more than cr , and is high if their distance is less than r . For simplicity we will treat each point as a bit string. (Any features that are not Boolean can be encoded into a set of Boolean features.)

The intuition we rely on is that if two points are close together in an n -dimensional space, then they will necessarily be close when projected down onto a one-dimensional space (a line). In fact, we can discretize the line into bins—hash buckets—so that, with high probability, near points project down to exactly the same bin. Points that are far away from each other will tend to project down into different bins for most projections, but there will always be a few projections that coincidentally project far-apart points into the same bin. Thus, the bin for point \mathbf{x}_q contains many (but not all) points that are near to \mathbf{x}_q , as well as some points that are far away.

The trick of LSH is to create *multiple* random projections and combine them. A random projection is just a random subset of the bit-string representation. We choose ℓ different random projections and create ℓ hash tables, $g_1(\mathbf{x}), \dots, g_\ell(\mathbf{x})$. We then enter all the examples into each hash table. Then when given a query point \mathbf{x}_q , we fetch the set of points in bin $g_k(q)$ for each k , and union these sets together into a set of candidate points, C . Then we compute the actual distance to \mathbf{x}_q for each of the points in C and return the k closest points. With high probability, each of the points that are near to \mathbf{x}_q will show up in at least one of the bins, and although some far-away points will show up as well, we can ignore those. With large real-world problems, such as finding the near neighbors in a data set of 13 million Web images using 512 dimensions (Torralba *et al.*, 2008), locality-sensitive hashing needs to examine only a few thousand images out of 13 million to find nearest neighbors; a thousand-fold speedup over exhaustive or k-d tree approaches.

18.8.4 Nonparametric regression

Now we'll look at nonparametric approaches to *regression* rather than classification. Figure 18.28 shows an example of some different models. In (a), we have perhaps the simplest method of all, known informally as “connect-the-dots,” and superciliously as “piecewise-linear nonparametric regression.” This model creates a function $h(x)$ that, when given a query x_q , solves the ordinary linear regression problem with just two points: the training examples immediately to the left and right of x_q . When noise is low, this trivial method is actually not too bad, which is why it is a standard feature of charting software in spreadsheets.



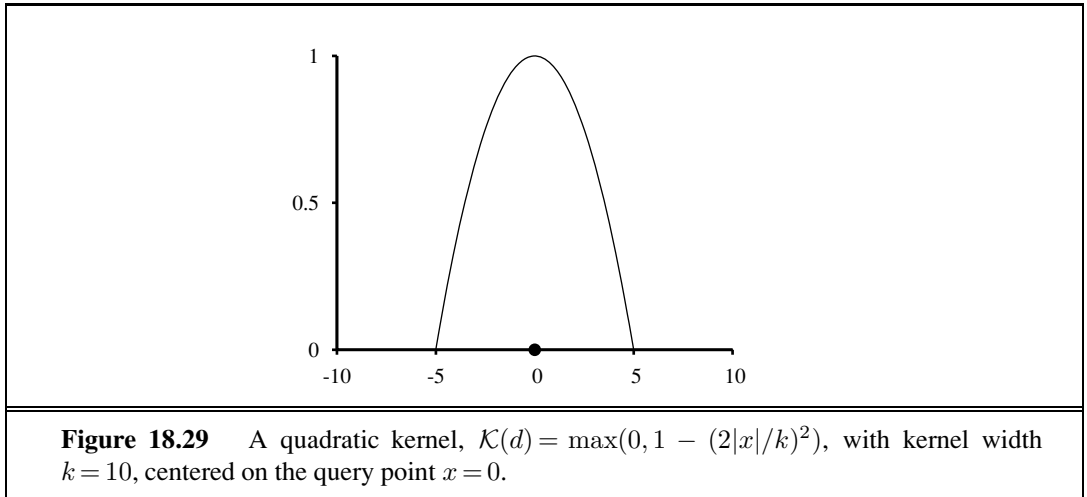
But when the data are noisy, the resulting function is spiky, and does not generalize well.

k -nearest-neighbors regression (Figure 18.28(b)) improves on connect-the-dots. Instead of using just the two examples to the left and right of a query point x_q , we use the k nearest neighbors (here 3). A larger value of k tends to smooth out the magnitude of the spikes, although the resulting function has discontinuities. In (b), we have the k -nearest-neighbors average: $h(x)$ is the mean value of the k points, $\sum y_j/k$. Notice that at the outlying points, near $x = 0$ and $x = 14$, the estimates are poor because all the evidence comes from one side (the interior), and ignores the trend. In (c), we have k -nearest-neighbor linear regression, which finds the best line through the k examples. This does a better job of capturing trends at the outliers, but is still discontinuous. In both (b) and (c), we're left with the question of how to choose a good value for k . The answer, as usual, is cross-validation.

Locally weighted regression (Figure 18.28(d)) gives us the advantages of nearest neighbors, without the discontinuities. To avoid discontinuities in $h(x)$, we need to avoid disconti-

NEAREST-
NEIGHBORS
REGRESSION

LOCALLY WEIGHTED
REGRESSION



nities in the set of examples we use to estimate $h(x)$. The idea of locally weighted regression is that at each query point x_q , the examples that are close to x_q are weighted heavily, and the examples that are farther away are weighted less heavily or not at all. The decrease in weight over distance is always gradual, not sudden.

KERNEL

We decide how much to weight each example with a function known as a **kernel**. A kernel function looks like a bump; in Figure 18.29 we see the specific kernel used to generate Figure 18.28(d). We can see that the weight provided by this kernel is highest in the center and reaches zero at a distance of ± 5 . Can we choose just any function for a kernel? No. First, note that we invoke a kernel function \mathcal{K} with $\mathcal{K}(\text{Distance}(\mathbf{x}_j, \mathbf{x}_q))$, where \mathbf{x}_q is a query point that is a given distance from \mathbf{x}_j , and we want to know how much to weight that distance. So \mathcal{K} should be symmetric around 0 and have a maximum at 0. The area under the kernel must remain bounded as we go to $\pm\infty$. Other shapes, such as Gaussians, have been used for kernels, but the latest research suggests that the choice of shape doesn't matter much. We do have to be careful about the width of the kernel. Again, this is a parameter of the model that is best chosen by cross-validation. Just as in choosing the k for nearest neighbors, if the kernels are too wide we'll get underfitting and if they are too narrow we'll get overfitting. In Figure 18.29(d), the value of $k = 10$ gives a smooth curve that looks about right—but maybe it does not pay enough attention to the outlier at $x = 6$; a narrower kernel width would be more responsive to individual points.

Doing locally weighted regression with kernels is now straightforward. For a given query point \mathbf{x}_q we solve the following weighted regression problem using gradient descent:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j \mathcal{K}(\text{Distance}(\mathbf{x}_q, \mathbf{x}_j)) (y_j - \mathbf{w} \cdot \mathbf{x}_j)^2,$$

where Distance is any of the distance metrics discussed for nearest neighbors. Then the answer is $h(\mathbf{x}_q) = \mathbf{w}^* \cdot \mathbf{x}_q$.

Note that we need to solve a new regression problem for *every* query point—that's what it means to be *local*. (In ordinary linear regression, we solved the regression problem once, globally, and then used the same $h_{\mathbf{w}}$ for any query point.) Mitigating against this extra work

is the fact that each regression problem will be easier to solve, because it involves only the examples with nonzero weight—the examples whose kernels overlap the query point. When kernel widths are small, this may be just a few points.

Most nonparametric models have the advantage that it is easy to do leave-one-out cross-validation without having to recompute everything. With a k -nearest-neighbors model, for instance, when given a test example (\mathbf{x}, y) we retrieve the k nearest neighbors once, compute the per-example loss $L(y, h(\mathbf{x}))$ from them, and record that as the leave-one-out result for every example that is not one of the neighbors. Then we retrieve the $k + 1$ nearest neighbors and record distinct results for leaving out each of the k neighbors. With N examples the whole process is $O(k)$, not $O(kN)$.

18.9 SUPPORT VECTOR MACHINES

SUPPORT VECTOR MACHINE

The **support vector machine** or SVM framework is currently the most popular approach for “off-the-shelf” supervised learning: if you don’t have any specialized prior knowledge about a domain, then the SVM is an excellent method to try first. There are three properties that make SVMs attractive:

1. SVMs construct a **maximum margin separator**—a decision boundary with the largest possible distance to example points. This helps them generalize well.
2. SVMs create a linear separating hyperplane, but they have the ability to embed the data into a higher-dimensional space, using the so-called **kernel trick**. Often, data that are not linearly separable in the original input space are easily separable in the higher-dimensional space. The high-dimensional linear separator is actually nonlinear in the original space. This means the hypothesis space is greatly expanded over methods that use strictly linear representations.
3. SVMs are a nonparametric method—they retain training examples and potentially need to store them all. On the other hand, in practice they often end up retaining only a small fraction of the number of examples—sometimes as few as a small constant times the number of dimensions. Thus SVMs combine the advantages of nonparametric and parametric models: they have the flexibility to represent complex functions, but they are resistant to overfitting.

You could say that SVMs are successful because of one key insight and one neat trick. We will cover each in turn. In Figure 18.30(a), we have a binary classification problem with three candidate decision boundaries, each a linear separator. Each of them is consistent with all the examples, so from the point of view of 0/1 loss, each would be equally good. Logistic regression would find some separating line; the exact location of the line depends on *all* the example points. The key insight of SVMs is that some examples are more important than others, and that paying attention to them can lead to better generalization.

Consider the lowest of the three separating lines in (a). It comes very close to 5 of the black examples. Although it classifies all the examples correctly, and thus minimizes loss, it

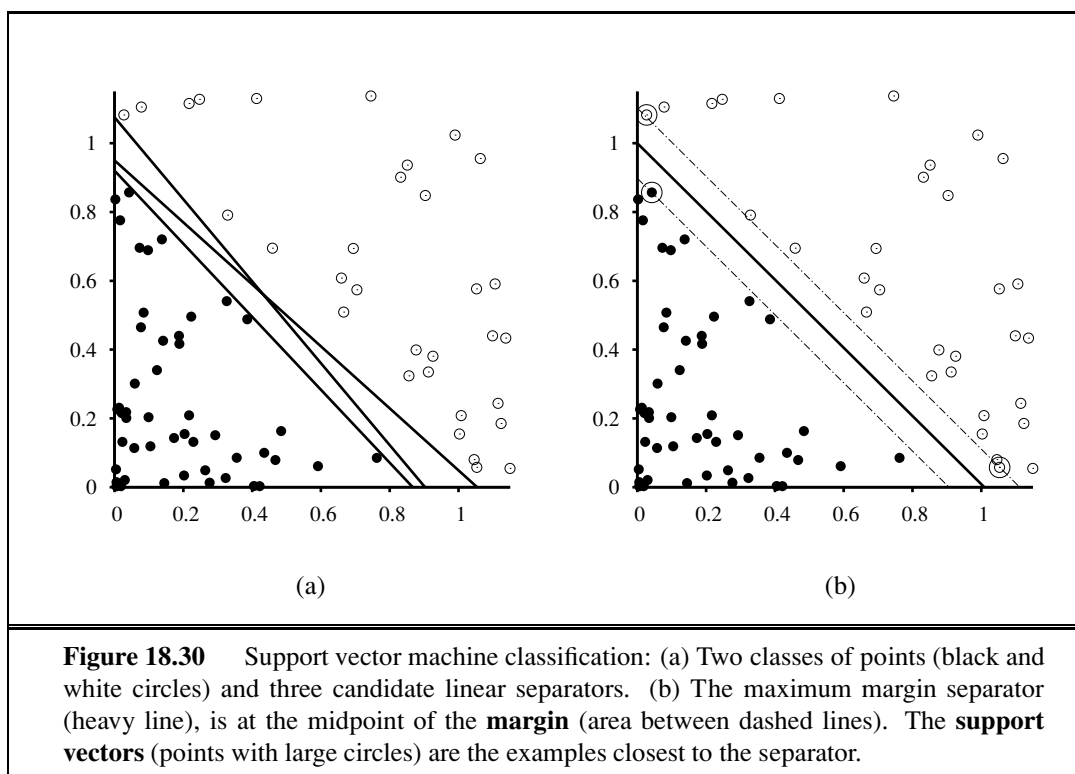


Figure 18.30 Support vector machine classification: (a) Two classes of points (black and white circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points with large circles) are the examples closest to the separator.

should make you nervous that so many examples are close to the line; it may be that other black examples will turn out to fall on the other side of the line.

SVMs address this issue: Instead of minimizing expected *empirical loss* on the training data, SVMs attempt to minimize expected *generalization loss*. We don't know where the as-yet-unseen points may fall, but under the probabilistic assumption that they are drawn from the same distribution as the previously seen examples, there are some arguments from computational learning theory (Section 18.5) suggesting that we minimize generalization loss by choosing the separator that is farthest away from the examples we have seen so far. We call this separator, shown in Figure 18.30(b) the **maximum margin separator**. The **margin** is the width of the area bounded by dashed lines in the figure—twice the distance from the separator to the nearest example point.

Now, how do we find this separator? Before showing the equations, some notation: Traditionally SVMs use the convention that class labels are +1 and -1, instead of the +1 and 0 we have been using so far. Also, where we put the intercept into the weight vector \mathbf{w} (and a corresponding dummy 1 value into $x_{j,0}$), SVMs do not do that; they keep the intercept as a separate parameter, b . With that in mind, the separator is defined as the set of points $\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + b = 0\}$. We could search the space of \mathbf{w} and b with gradient descent to find the parameters that maximize the margin while correctly classifying all the examples.

However, it turns out there is another approach to solving this problem. We won't show the details, but will just say that there is an alternative representation called the dual

MAXIMUM MARGIN
SEPARATOR
MARGIN

representation, in which the optimal solution is found by solving

$$\operatorname{argmax}_{\alpha} \sum_j \alpha_j - \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k (\mathbf{x}_j \cdot \mathbf{x}_k) \quad (18.13)$$

QUADRATIC
PROGRAMMING



subject to the constraints $\alpha_j \geq 0$ and $\sum_j \alpha_j y_j = 0$. This is a **quadratic programming** optimization problem, for which there are good software packages. Once we have found the vector α we can get back to \mathbf{w} with the equation $\mathbf{w} = \sum_j \alpha_j \mathbf{x}_j$, or we can stay in the dual representation. There are three important properties of Equation (18.13). First, the expression is convex; it has a single global maximum that can be found efficiently. Second, *the data enter the expression only in the form of dot products of pairs of points*. This second property is also true of the equation for the separator itself; once the optimal α_j have been calculated, it is

$$h(\mathbf{x}) = \operatorname{sign} \left(\sum_j \alpha_j y_j (\mathbf{x} \cdot \mathbf{x}_j) - b \right). \quad (18.14)$$

SUPPORT VECTOR

A final important property is that the weights α_j associated with each data point are *zero* except for the **support vectors**—the points closest to the separator. (They are called “support” vectors because they “hold up” the separating plane.) Because there are usually many fewer support vectors than examples, SVMs gain some of the advantages of parametric models.

What if the examples are not linearly separable? Figure 18.31(a) shows an input space defined by attributes $\mathbf{x} = (x_1, x_2)$, with positive examples ($y = +1$) inside a circular region and negative examples ($y = -1$) outside. Clearly, there is no linear separator for this problem. Now, suppose we re-express the input data—i.e., we map each input vector \mathbf{x} to a new vector of feature values, $F(\mathbf{x})$. In particular, let us use the three features

$$f_1 = x_1^2, \quad f_2 = x_2^2, \quad f_3 = \sqrt{2}x_1x_2. \quad (18.15)$$

We will see shortly where these came from, but for now, just look at what happens. Figure 18.31(b) shows the data in the new, three-dimensional space defined by the three features; the data are *linearly separable* in this space! This phenomenon is actually fairly general: if data are mapped into a space of sufficiently high dimension, then they will almost always be linearly separable—if you look at a set of points from enough directions, you’ll find a way to make them line up. Here, we used only three dimensions;¹¹ Exercise 18.16 asks you to show that four dimensions suffice for linearly separating a circle anywhere in the plane (not just at the origin), and five dimensions suffice to linearly separate any ellipse. In general (with some special cases excepted) if we have N data points then they will always be separable in spaces of $N - 1$ dimensions or more (Exercise 18.25).

Now, we would not usually expect to find a linear separator in the input space \mathbf{x} , but we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_j \cdot \mathbf{x}_k$ in Equation (18.13) with $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$. This by itself is not remarkable—replacing \mathbf{x} by $F(\mathbf{x})$ in *any* learning algorithm has the required effect—but the dot product has some special properties. It turns out that $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$ can often be computed without first computing F

¹¹ The reader may notice that we could have used just f_1 and f_2 , but the 3D mapping illustrates the idea better.

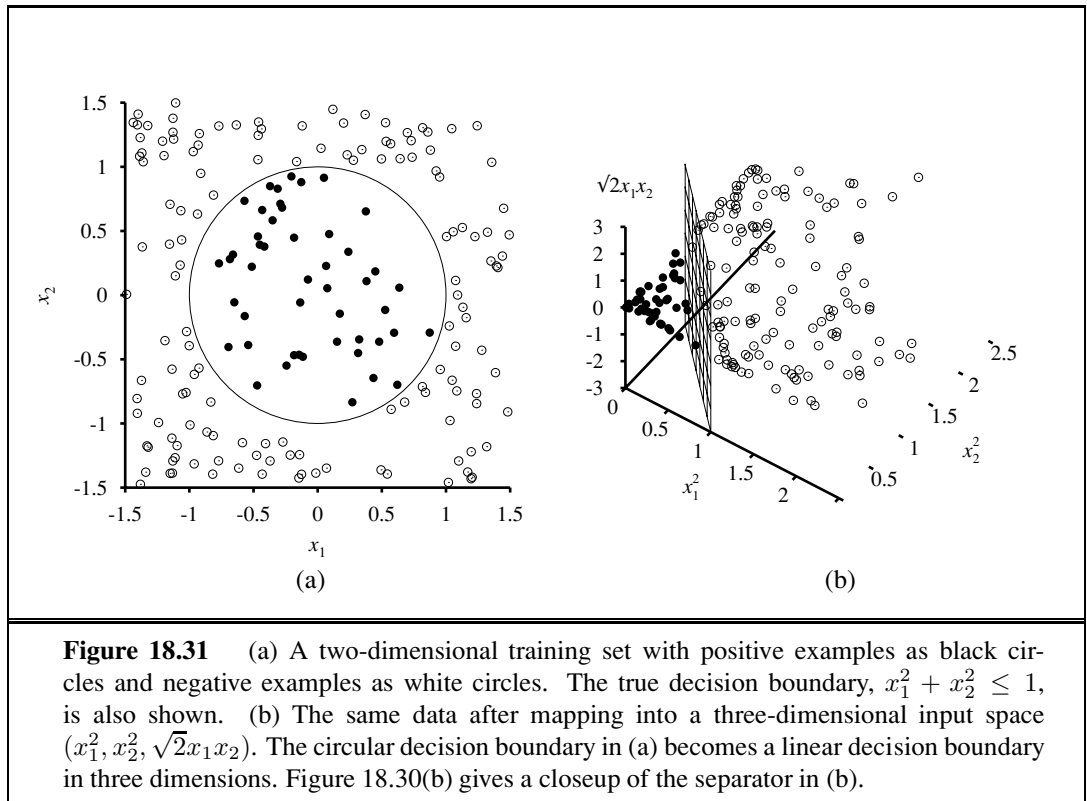


Figure 18.31 (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 18.30(b) gives a closeup of the separator in (b).

for each point. In our three-dimensional feature space defined by Equation (18.15), a little bit of algebra shows that

$$F(\mathbf{x}_j) \cdot F(\mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2.$$

KERNEL FUNCTION

(That's why the $\sqrt{2}$ is in f_3 .) The expression $(\mathbf{x}_j \cdot \mathbf{x}_k)^2$ is called a **kernel function**,¹² and is usually written as $K(\mathbf{x}_j, \mathbf{x}_k)$. The kernel function can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can find linear separators in the higher-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_j \cdot \mathbf{x}_k$ in Equation (18.13) with a kernel function $K(\mathbf{x}_j, \mathbf{x}_k)$. Thus, we can learn in the higher-dimensional space, but we compute only kernel functions rather than the full list of features for each data point.

MERCER'S THEOREM

POLYNOMIAL
KERNEL

The next step is to see that there's nothing special about the kernel $K(\mathbf{x}_j, \mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2$. It corresponds to a particular higher-dimensional feature space, but other kernel functions correspond to other feature spaces. A venerable result in mathematics, **Mercer's theorem** (1909), tells us that any “reasonable”¹³ kernel function corresponds to *some* feature space. These feature spaces can be very large, even for innocuous-looking kernels. For example, the **polynomial kernel**, $K(\mathbf{x}_j, \mathbf{x}_k) = (1 + \mathbf{x}_j \cdot \mathbf{x}_k)^d$, corresponds to a feature space whose dimension is exponential in d .

¹² This usage of “kernel function” is slightly different from the kernels in locally weighted regression. Some SVM kernels are distance metrics, but not all are.

¹³ Here, “reasonable” means that the matrix $\mathbf{K}_{jk} = K(\mathbf{x}_j, \mathbf{x}_k)$ is positive definite.

KERNEL TRICK



This then is the clever **kernel trick**: Plugging these kernels into Equation (18.13), *optimal linear separators can be found efficiently in feature spaces with billions of (or, in some cases, infinitely many) dimensions*. The resulting linear separators, when mapped back to the original input space, can correspond to arbitrarily wiggly, nonlinear decision boundaries between the positive and negative examples.

SOFT MARGIN

In the case of inherently noisy data, we may not want a linear separator in some high-dimensional space. Rather, we'd like a decision surface in a lower-dimensional space that does not cleanly separate the classes, but reflects the reality of the noisy data. That is possible with the **soft margin** classifier, which allows examples to fall on the wrong side of the decision boundary, but assigns them a penalty proportional to the distance required to move them back on the correct side.

KERNELIZATION

The kernel method can be applied not only with learning algorithms that find optimal linear separators, but also with any other algorithm that can be reformulated to work only with dot products of pairs of data points, as in Equations 18.13 and 18.14. Once this is done, the dot product is replaced by a kernel function and we have a **kernelized** version of the algorithm. This can be done easily for k -nearest-neighbors and perceptron learning (Section 18.7.2), among others.

18.10 ENSEMBLE LEARNING

ENSEMBLE
LEARNING

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

The motivation for ensemble learning is simple. Consider an ensemble of $K = 5$ 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_k in the ensemble has an error of p —that is, the probability that a randomly chosen example is misclassified by h_k 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 18.18) shows that using an ensemble of five hypotheses reduces an error rate of 1 in 10 down to an error rate of less than 1 in 100. Now, obviously 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. But if the hypotheses are at least a little bit different, thereby reducing the correlation between their errors, then ensemble learning can 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

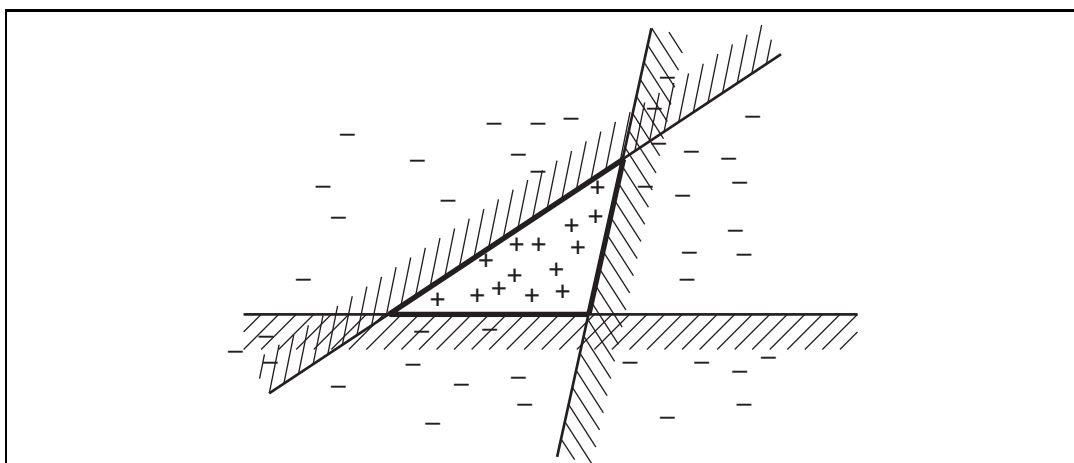


Figure 18.32 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the unshaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

space as the set of all possible ensembles constructable from hypotheses in the original space. Figure 18.32 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.

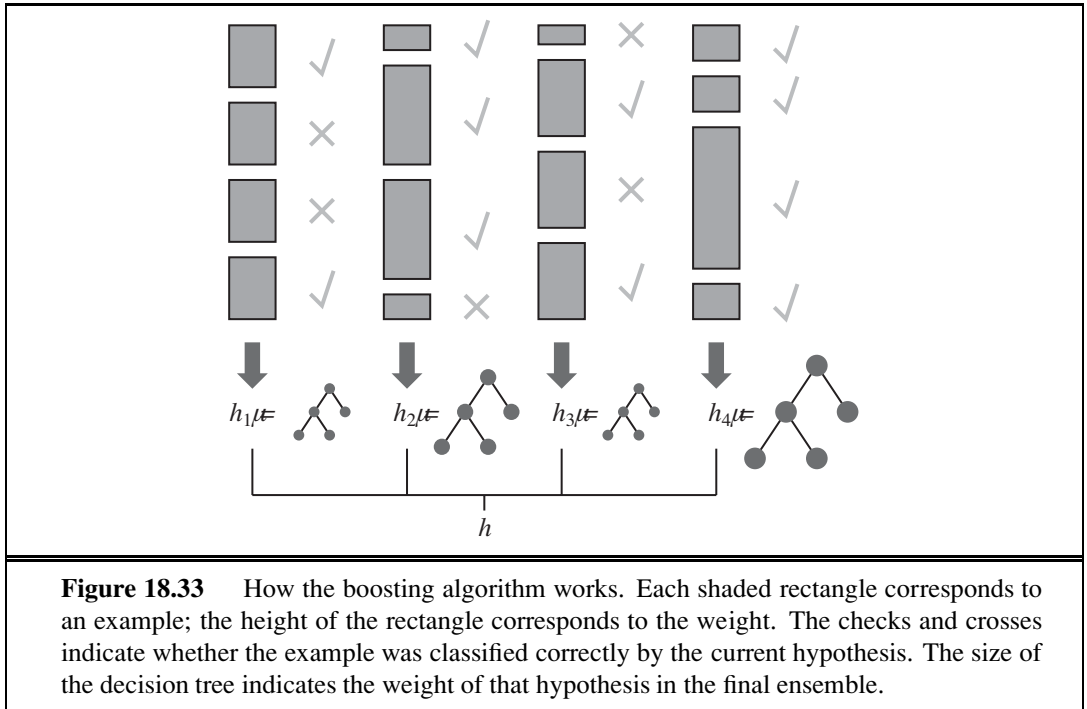
BOOSTING
WEIGHTED TRAINING
SET

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 $w_j \geq 0$. The higher the weight of an example, the higher is the importance attached to it during the learning of a hypothesis. It is straightforward to modify the learning algorithms we have seen so far to operate with weighted training sets.¹⁴

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 K hypotheses, where K is an input to the boosting algorithm. The final ensemble hypothesis is a weighted-majority combination of all the K hypotheses, each weighted according to how well it performed on the training set. Figure 18.33 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 18.34. ADABOOST has a very important property: if the input learning algorithm L is a **weak learning** algorithm—which

WEAK LEARNING

¹⁴ For learning algorithms in which this is not possible, one can instead create a **replicated training set** where the j th example appears w_j times, using randomization to handle fractional weights.



means that L always returns a hypothesis with accuracy on the training set that is slightly better than random guessing (i.e., $50\% + \epsilon$ for Boolean classification)—then ADABOOST will return a hypothesis that *classifies the training data perfectly* for large enough K . 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.

DECISION STUMP

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 18.35(a) shows that unboosted decision stumps are not very effective for this data set, reaching a prediction performance of only 81% on 100 training examples. When boosting is applied (with $K = 5$), the performance is better, reaching 93% after 100 examples.

An interesting thing happens as the ensemble size K increases. Figure 18.35(b) shows the training set performance (on 100 examples) as a function of K . Notice that the error reaches zero when K is 20; that is, a weighted-majority combination of 20 decision stumps suffices to fit the 100 examples exactly. As more stumps are added to the ensemble, the error remains at zero. The graph also shows that *the test set performance continues to increase long after the training set error has reached zero*. At $K = 20$, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as $K = 137$, before gradually dropping to 0.95.

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



```

function ADABOOST(examples, L, K) returns a weighted-majority hypothesis
  inputs: examples, set of N labeled examples  $(x_1, y_1), \dots, (x_N, y_N)$ 
           L, a learning algorithm
           K, the number of hypotheses in the ensemble
  local variables: w, a vector of N example weights, initially  $1/N$ 
                    h, a vector of K hypotheses
                    z, a vector of K hypothesis weights

  for k = 1 to K do
    h[k]  $\leftarrow L(\text{examples}, \mathbf{w})$ 
    error  $\leftarrow 0$ 
    for j = 1 to N do
      if h[k](xj)  $\neq y_j$  then error  $\leftarrow \text{error} + \mathbf{w}[j]$ 
    for j = 1 to N do
      if h[k](xj) = yj then  $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot \text{error} / (1 - \text{error})$ 
     $\mathbf{w} \leftarrow \text{NORMALIZE}(\mathbf{w})$ 
     $\mathbf{z}[k] \leftarrow \log(1 - \text{error}) / \text{error}$ 
  return WEIGHTED-MAJORITY(h, z)

```

Figure 18.34 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples. 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**.

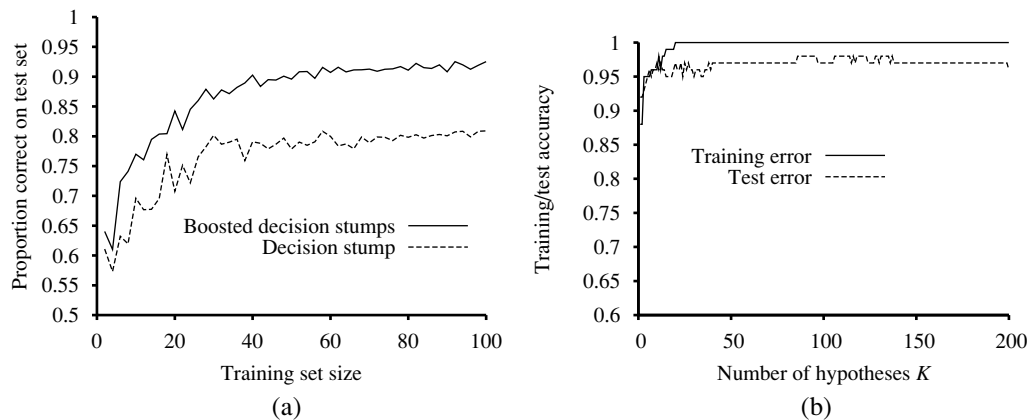


Figure 18.35 (a) Graph showing the performance of boosted decision stumps with $K = 5$ versus unboosted decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of K , the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

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 learning** (see Chapter 20), which can be shown to be an optimal learning algorithm, and the approximation improves as more hypotheses are added. Another possible explanation is that the addition of further hypotheses enables the ensemble to be *more definite* in its distinction between positive and negative examples, which helps it when it comes to classifying new examples.

18.10.1 Online Learning

So far, everything we have done in this chapter has relied on the assumption that the data are i.i.d. (independent and identically distributed). On the one hand, that is a sensible assumption: if the future bears no resemblance to the past, then how can we predict anything? On the other hand, it is too strong an assumption: it is rare that our inputs have captured all the information that would make the future truly independent of the past.

ONLINE LEARNING

In this section we examine what to do when the data are not i.i.d.; when they can change over time. In this case, it matters *when* we make a prediction, so we will adopt the perspective called **online learning**: an agent receives an input x_j from nature, predicts the corresponding y_j , and then is told the correct answer. Then the process repeats with x_{j+1} , and so on. One might think this task is hopeless—if nature is adversarial, all the predictions may be wrong. It turns out that there are some guarantees we can make.

RANDOMIZED
WEIGHTED
MAJORITY
ALGORITHM

Let us consider the situation where our input consists of predictions from a panel of experts. For example, each day a set of K pundits predicts whether the stock market will go up or down, and our task is to pool those predictions and make our own. One way to do this is to keep track of how well each expert performs, and choose to believe them in proportion to their past performance. This is called the **randomized weighted majority algorithm**. We can describe it more formally:

1. Initialize a set of weights $\{w_1, \dots, w_K\}$ all to 1.
2. Receive the predictions $\{\hat{y}_1, \dots, \hat{y}_K\}$ from the experts.
3. Randomly choose an expert k^* , in proportion to its weight: $P(k) = w_k / (\sum_{k'} w_{k'})$.
4. Predict \hat{y}_{k^*} .
5. Receive the correct answer y .
6. For each expert k such that $\hat{y}_k \neq y$, update $w_k \leftarrow \beta w_k$

Here β is a number, $0 < \beta < 1$, that tells how much to penalize an expert for each mistake.

REGRET

We measure the success of this algorithm in terms of **regret**, which is defined as the number of additional mistakes we make compared to the expert who, in hindsight, had the best prediction record. Let M^* be the number of mistakes made by the best expert. Then the number of mistakes, M , made by the random weighted majority algorithm, is bounded by¹⁵

$$M < \frac{M^* \ln(1/\beta) + \ln K}{1 - \beta}.$$

¹⁵ See (Blum, 1996) for the proof.

This bound holds for *any* sequence of examples, even ones chosen by adversaries trying to do their worst. To be specific, when there are $K = 10$ experts, if we choose $\beta = 1/2$ then our number of mistakes is bounded by $1.39M^* + 4.6$, and if $\beta = 3/4$ by $1.15M^* + 9.2$. In general, if β is close to 1 then we are responsive to change over the long run; if the best expert changes, we will pick up on it before too long. However, we pay a penalty at the beginning, when we start with all experts trusted equally; we may accept the advice of the bad experts for too long. When β is closer to 0, these two factors are reversed. Note that we can choose β to get asymptotically close to M^* in the long run; this is called **no-regret learning** (because the average amount of regret per trial tends to 0 as the number of trials increases).

Online learning is helpful when the data may be changing rapidly over time. It is also useful for applications that involve a large collection of data that is constantly growing, even if changes are gradual. For example, with a database of millions of Web images, you wouldn't want to train, say, a linear regression model on all the data, and then retrain from scratch every time a new image is added. It would be more practical to have an online algorithm that allows images to be added incrementally. For most learning algorithms based on minimizing loss, there is an online version based on minimizing regret. It is a bonus that many of these online algorithms come with guaranteed bounds on regret.

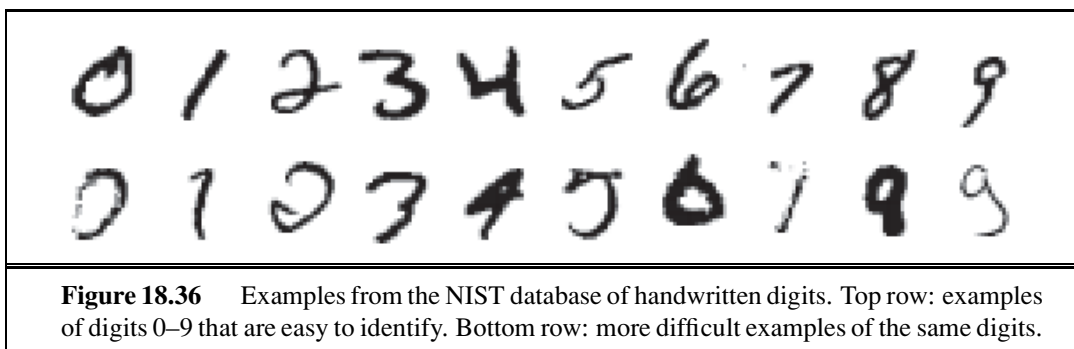
To some observers, it is surprising that there are such tight bounds on how well we can do compared to a panel of experts. To others, the really surprising thing is that when panels of human experts congregate—predicting stock market prices, sports outcomes, or political contests—the viewing public is so willing to listen to them pontificate and so unwilling to quantify their error rates.

18.11 PRACTICAL MACHINE LEARNING

We have introduced a wide range of machine learning techniques, each illustrated with simple learning tasks. In this section, we consider two aspects of practical machine learning. The first involves finding algorithms capable of learning to recognize handwritten digits and squeezing every last drop of predictive performance out of them. The second involves anything but—pointing out that obtaining, cleaning, and representing the data can be at least as important as algorithm engineering.

18.11.1 Case study: Handwritten digit recognition

Recognizing handwritten digits is an important problem with many applications, including automated sorting of mail by postal code, automated reading of checks and tax returns, and data entry for hand-held computers. It is an area where rapid progress has been made, in part because of better learning algorithms and in part because of the availability of better training sets. The United States National Institute of Science and Technology (**NIST**) has archived a database of 60,000 labeled digits, each $20 \times 20 = 400$ pixels with 8-bit grayscale values. It has become one of the standard benchmark problems for comparing new learning algorithms. Some example digits are shown in Figure 18.36.



Many different learning approaches have been tried. One of the first, and probably the simplest, is the **3-nearest-neighbor** classifier, which also has the advantage of requiring no training time. As a memory-based algorithm, however, it must store all 60,000 images, and its run time performance is slow. It achieved a test error rate of 2.4%.

A **single-hidden-layer neural network** was designed for this problem with 400 input units (one per pixel) and 10 output units (one per class). Using cross-validation, it was found that roughly 300 hidden units gave the best performance. With full interconnections between layers, there were a total of 123,300 weights. This network achieved a 1.6% error rate.

A series of **specialized neural networks** called LeNet were devised to take advantage of the structure of the problem—that the input consists of pixels in a two-dimensional array, and that small changes in the position or slant of an image are unimportant. Each network had an input layer of 32×32 units, onto which the 20×20 pixels were centered so that each input unit is presented with a local neighborhood of pixels. This was followed by three layers of hidden units. Each layer consisted of several planes of $n \times n$ arrays, where n is smaller than the previous layer so that the network is down-sampling the input, and where the weights of every unit in a plane are constrained to be identical, so that the plane is acting as a feature detector: it can pick out a feature such as a long vertical line or a short semi-circular arc. The output layer had 10 units. Many versions of this architecture were tried; a representative one had hidden layers with 768, 192, and 30 units, respectively. The training set was augmented by applying affine transformations to the actual inputs: shifting, slightly rotating, and scaling the images. (Of course, the transformations have to be small, or else a 6 will be transformed into a 9!) The best error rate achieved by LeNet was 0.9%.

A **boosted neural network** combined three copies of the LeNet architecture, with the second one trained on a mix of patterns that the first one got 50% wrong, and the third one trained on patterns for which the first two disagreed. During testing, the three nets voted with the majority ruling. The test error rate was 0.7%.

A **support vector machine** (see Section 18.9) with 25,000 support vectors achieved an error rate of 1.1%. This is remarkable because the SVM technique, like the simple nearest-neighbor approach, required almost no thought or iterated experimentation on the part of the developer, yet it still came close to the performance of LeNet, which had had years of development. Indeed, the support vector machine makes no use of the structure of the problem, and would perform just as well if the pixels were presented in a permuted order.

A **virtual support vector machine** starts with a regular SVM and then improves it with a technique that is designed to take advantage of the structure of the problem. Instead of allowing products of all pixel pairs, this approach concentrates on kernels formed from pairs of nearby pixels. It also augments the training set with transformations of the examples, just as LeNet did. A virtual SVM achieved the best error rate recorded to date, 0.56%.

Shape matching is a technique from computer vision used to align corresponding parts of two different images of objects (Belongie *et al.*, 2002). The idea is to pick out a set of points in each of the two images, and then compute, for each point in the first image, which point in the second image it corresponds to. From this alignment, we then compute a transformation between the images. The transformation gives us a measure of the distance between the images. This distance measure is better motivated than just counting the number of differing pixels, and it turns out that a 3-nearest neighbor algorithm using this distance measure performs very well. Training on only 20,000 of the 60,000 digits, and using 100 sample points per image extracted from a Canny edge detector, a shape matching classifier achieved 0.63% test error.

Humans are estimated to have an error rate of about 0.2% on this problem. This figure is somewhat suspect because humans have not been tested as extensively as have machine learning algorithms. On a similar data set of digits from the United States Postal Service, human errors were at 2.5%.

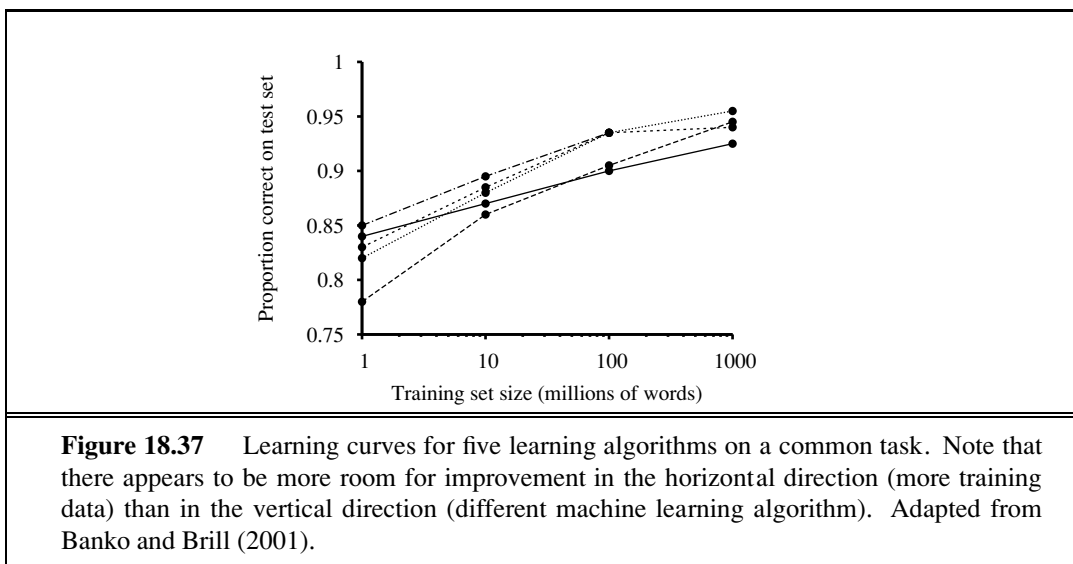
The following figure summarizes the error rates, run time performance, memory requirements, and amount of training time for the seven algorithms we have discussed. It also adds another measure, the percentage of digits that must be rejected to achieve 0.5% error. For example, if the SVM is allowed to reject 1.8% of the inputs—that is, pass them on for someone else to make the final judgment—then its error rate on the remaining 98.2% of the inputs is reduced from 1.1% to 0.5%.

The following table summarizes the error rate and some of the other characteristics of the seven techniques we have discussed.

	3 NN	300 Hidden	LeNet	Boosted LeNet	SVM	Virtual SVM	Shape Match
Error rate (pct.)	2.4	1.6	0.9	0.7	1.1	0.56	0.63
Run time (millisec/digit)	1000	10	30	50	2000	200	
Memory requirements (Mbyte)	12	.49	.012	.21	11		
Training time (days)	0	7	14	30	10		
% rejected to reach 0.5% error	8.1	3.2	1.8	0.5	1.8		

18.11.2 Case study: Word senses and house prices

In a textbook we need to deal with simple, toy data to get the ideas across: a small data set, usually in two dimensions. But in practical applications of machine learning, the data set is usually large, multidimensional, and messy. The data are not handed to the analyst in a prepackaged set of (\mathbf{x}, y) values; rather the analyst needs to go out and acquire the right data. There is a task to be accomplished, and most of the engineering problem is deciding what data are necessary to accomplish the task; a smaller part is choosing and implementing an



appropriate machine learning method to process the data. Figure 18.37 shows a typical real-world example, comparing five learning algorithms on the task of word-sense classification (given a sentence such as “The bank folded,” classify the word “bank” as “money-bank” or “river-bank”). The point is that machine learning researchers have focused mainly on the vertical direction: Can I invent a new learning algorithm that performs better than previously published algorithms on a standard training set of 1 million words? But the graph shows there is more room for improvement in the horizontal direction: instead of inventing a new algorithm, all I need to do is gather 10 million words of training data; even the *worst* algorithm at 10 million words is performing better than the *best* algorithm at 1 million. As we gather even more data, the curves continue to rise, dwarfing the differences between algorithms.

Consider another problem: the task of estimating the true value of houses that are for sale. In Figure 18.13 we showed a toy version of this problem, doing linear regression of house size to asking price. You probably noticed many limitations of this model. First, it is measuring the wrong thing: we want to estimate the selling price of a house, not the asking price. To solve this task we’ll need data on actual sales. But that doesn’t mean we should throw away the data about asking price—we can use it as one of the input features. Besides the size of the house, we’ll need more information: the number of rooms, bedrooms and bathrooms; whether the kitchen and bathrooms have been recently remodeled; the age of the house; we’ll also need information about the lot, and the neighborhood. But how do we define neighborhood? By zip code? What if part of one zip code is on the “wrong” side of the highway or train tracks, and the other part is desirable? What about the school district? Should the *name* of the school district be a feature, or the *average test scores*? In addition to deciding what features to include, we will have to deal with missing data; different areas have different customs on what data are reported, and individual cases will always be missing some data. If the data you want are not available, perhaps you can set up a social networking site to encourage people to share and correct data. In the end, this process of

deciding what features to use, and how to use them, is just as important as choosing between linear regression, decision trees, or some other form of learning.

That said, one *does* have to pick a method (or methods) for a problem. There is no guaranteed way to pick the best method, but there are some rough guidelines. Decision trees are good when there are a lot of discrete features and you believe that many of them may be irrelevant. Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features (as long as there are fewer than 20 or so). However, nonparametric methods usually give you a function h that is more expensive to run. Support vector machines are often considered the best method to try first, provided the data set is not too large.

18.12 SUMMARY

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

- Learning takes many forms, depending on the nature of the agent, the component to be improved, and the available feedback.
- If the available feedback provides the correct answer for example inputs, then the learning problem is called **supervised learning**. The task is to learn a function $y = h(x)$. Learning a discrete-valued function is called **classification**; learning a continuous function is called **regression**.
- Inductive learning involves finding a hypothesis that agrees well 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 a learning algorithm is measured by the **learning curve**, which shows the prediction accuracy on the **test set** as a function of the **training-set** size.
- When there are multiple models to choose from, **cross-validation** can be used to select a model that will generalize well.
- Sometimes not all errors are equal. A **loss function** tells us how bad each error is; the goal is then to minimize loss over a validation set.
- **Computational learning theory** analyzes the sample complexity and computational complexity of inductive learning. There is a tradeoff between the expressiveness of the hypothesis language and the ease of learning.
- **Linear regression** is a widely used model. The optimal parameters of a linear regression model can be found by gradient descent search, or computed exactly.
- A linear classifier with a hard threshold—also known as a **perceptron**—can be trained by a simple weight update rule to fit data that are **linearly separable**. In other cases, the rule fails to converge.

- **Logistic regression** replaces the perceptron's hard threshold with a soft threshold defined by a logistic function. Gradient descent works well even for noisy data that are not linearly separable.
- **Neural networks** represent complex nonlinear functions with a network of linear-threshold units. Multilayer feed-forward neural networks can represent any function, given enough units. The **back-propagation** algorithm implements a gradient descent in parameter space to minimize the output error.
- **Nonparametric models** use all the data to make each prediction, rather than trying to summarize the data first with a few parameters. Examples include **nearest neighbors** and **locally weighted regression**.
- **Support vector machines** find linear separators with **maximum margin** to improve the generalization performance of the classifier. **Kernel methods** implicitly transform the input data into a high-dimensional space where a linear separator may exist, even if the original data are non-separable.
- Ensemble methods such as **boosting** often perform better than individual methods. In **online learning** we can aggregate the opinions of experts to come arbitrarily close to the best expert's performance, even when the distribution of the data is constantly shifting.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Chapter 1 outlined the history of philosophical investigations into inductive learning. William of Ockham¹⁶ (1280–1349), 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 beyond necessity.” Unfortunately, this laudable piece of advice is nowhere to be found in his writings in precisely these words (although he did say “Pluralitas non est ponenda sine necessitate,” or “plurality shouldn’t be posited without necessity”). A similar sentiment was expressed by Aristotle in 350 B.C. in *Physics* book I, chapter VI: “For the more limited, if adequate, is always preferable.”

The first notable use of decision trees was in EPAM, the “Elementary Perceiver And Memorizer” (Feigenbaum, 1961), which was a simulation of human concept learning. ID3 (Quinlan, 1979) added the crucial idea of choosing the attribute with maximum entropy; it is the basis for the decision tree algorithm in this chapter. Information theory 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 χ^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 exists in the statistical literature. *Classification and Regression Trees* (Breiman *et al.*, 1984), known as the “CART book,” is the principal reference.

¹⁶ The name is often misspelled as “Occam,” perhaps from the French rendering, “Guillaume d’Occam.”