17.19 A Dutch auction is similar in an English auction, but rather than starting the bidding at a low price and increasing, in a Dutch auction the seller starts at a high price and gradually lowers the price until some buyer is willing to accept that price. (If multiple bidders accept the price, one is arbitrarily chosen as the winner.) More formally, the seller begins with a price p and gradually lowers p by increments of d until at least one buyer accepts the price. Assuming all bidders act rationally, is it true that for arbitrarily small d, a Dutch auction will always result in the bidder with the highest value for the item obtaining the item? If so, show mathematically why. If not, explain how it may be possible for the bidder with highest value for the item not to obtain it.

17.20 Imagine an auction mechanism that is just like an ascending-bid auction, except that at the end, the winning bidder, the one who bid b_{max}, pays only $b_{max}/2$ rather than b_{max}. Assuming all agents are rational, what is the expected revenue to the auctioneer for this mechanism, compared with a standard ascending-bid auction?

17.21 Teams in the National Hockey League historically received 2 points for winning a game and 0 for losing. If the game is tied, an overtime period is played; if nobody wins in overtime, the game is a tie and each team gets 1 point. But league officials felt that teams were playing too conservatively in overtime (to avoid a loss), and it would be more exciting if overtime produced a winner. So in 1999 the officials experimented in mechanism design: the rules were changed, giving a team that loses in overtime 1 point, not 0. It is still 2 points for a win and 1 for a tie.

a. Was hockey a zero-sum game before the rule change? After?

b. Suppose that at a certain time t in a game, the home team has probability p of winning in regulation time, probability $0.78 - p$ of losing, and probability 0.22 of going into overtime, where they have probability q of winning, $.9 - q$ of losing, and .1 of tying. Give equations for the expected value for the home and visiting teams.

c. Imagine that it were legal and ethical for the two teams to enter into a pact where they agree that they will skate to a tie in regulation time, and then both try in earnest to win in overtime. Under what conditions, in terms of p and q, would it be rational for both teams to agree to this pact?

d. Longley and Sankaran (2005) report that since the rule change, the percentage of games with a winner in overtime went up 18.2%, as desired, but the percentage of overtime games also went up 3.6%. What does that suggest about possible collusion or conservative play after the rule change?

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.

INDUCTIVE LEARNING
DEDUCTIVE LEARNING

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:

UNSUPERVISED LEARNING
CLUSTERING

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
>
> $$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N) ,$$
>
> where each y_j was generated by an unknown function $y = f(x)$,
> discover a function h that approximates the true function f.

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

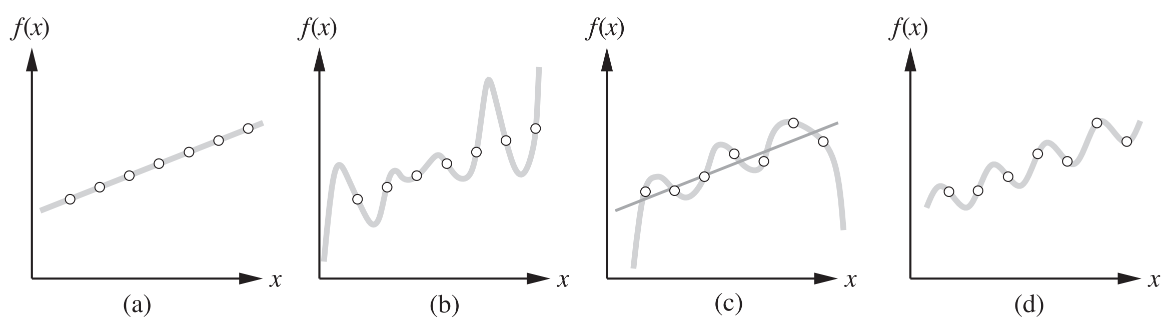

Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data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

GENERALIZATION

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

CLASSIFICATION

REGRESSION

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

HYPOTHESIS SPACE

CONSISTENT

OCKHAM'S RAZOR

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.

Figure 18.1(c) shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial for an exact fit. There are just 7 data points, so a polynomial with 7 parameters does not seem to be finding any pattern in the data and we do not expect it to generalize well. A straight line that is not consistent with any of the data points, but might generalize fairly well for unseen values of x, is also shown in (c). *In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better.* In Figure 18.1(d) we expand the hypothesis space $\mathcal{H}$ to allow polynomials over both x and $\sin(x)$, and find that the data in (c) can be fitted exactly by a simple function of the form $ax + b + c\sin(x)$. This shows the importance of the choice of hypothesis space. We say that a learning problem is **realizable** if the hypothesis space contains the true function. Unfortunately, we cannot always tell whether a given learning problem is realizable, because the true function is not known.

REALIZABLE

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

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

By Bayes' rule this is equivalent to

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

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

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

We will see that the expressiveness–complexity tradeoff is not as simple as it first seems: it is often the case, as we saw with first-order logic in Chapter 8, that an expressive language makes it possible for a *simple* hypothesis to fit the data, whereas restricting the expressiveness of the language means that any consistent hypothesis must be very complex. For example, the rules of chess can be written in a page or two of first-order logic, but require thousands of pages when written in propositional logic.

18.3 LEARNING DECISION TREES

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