5.3.2 The Binomial Distribution

A good way to understand the Binomial distribution is to consider the following problem. You are given a worn and bent coin and asked to estimate the probability that the coin will turn up heads when tossed. Let us call this unknown probability of heads p. You toss the coin n times and record the number of times r that it turns up heads. A reasonable estimate of p is r/n. Note that if the experiment were rerun, generating a new set of n coin tosses, we might expect the number of heads r to vary somewhat from the value measured in the first experiment, yielding a somewhat different estimate for p. The Binomial distribution describes for each possible value of r (i.e., from 0 to n), the probability of observing exactly r heads given a sample of n independent tosses of a coin whose true probability of heads is p.

Interestingly, estimating p from a random sample of coin tosses is equivalent to estimating $error_{\mathcal{D}}(h)$ from testing h on a random sample of instances. A single toss of the coin corresponds to drawing a single random instance from $\mathcal{D}$ and determining whether it is misclassified by h. The probability p that a single random coin toss will turn up heads corresponds to the probability that a single instance drawn at random will be misclassified (i.e., p corresponds to $error_{\mathcal{D}}(h)$). The number r of heads observed over a sample of n coin tosses corresponds to the number of misclassifications observed over n randomly drawn instances. Thus r/n corresponds to $error_S(h)$. The problem of estimating p for coins is identical to the problem of estimating $error_{\mathcal{D}}(h)$ for hypotheses. The Binomial distribution gives the general form of the probability distribution for the random variable r, whether it represents the number of heads in n coin tosses or the number of hypothesis errors in a sample of n examples. The detailed form of the Binomial distribution depends on the specific sample size n and the specific probability p or $error_{\mathcal{D}}(h)$.

The general setting to which the Binomial distribution applies is:

1. There is a base, or underlying, experiment (e.g., toss of the coin) whose outcome can be described by a random variable, say Y. The random variable Y can take on two possible values (e.g., $Y = 1$ if heads, $Y = 0$ if tails).
2. The probability that $Y = 1$ on any single trial of the underlying experiment is given by some constant p, independent of the outcome of any other experiment. The probability that $Y = 0$ is therefore $(1 - p)$. Typically, p is not known in advance, and the problem is to estimate it.
3. A series of n independent trials of the underlying experiment is performed (e.g., n independent coin tosses), producing the sequence of independent, identically distributed random variables $Y_1, Y_2, \ldots, Y_n$. Let R denote the number of trials for which $Y_i = 1$ in this series of n experiments

$$R \equiv \sum_{i=1}^{n} Y_i$$

4. The probability that the random variable R will take on a specific value r (e.g., the probability of observing exactly r heads) is given by the Binomial distribution

$$\Pr(R = r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r} \tag{5.2}$$

A plot of this probability distribution is shown in Table 5.3.

The Binomial distribution characterizes the probability of observing r heads from n coin flip experiments, as well as the probability of observing r errors in a data sample containing n randomly drawn instances.

5.3.3 Mean and Variance

Two properties of a random variable that are often of interest are its expected value (also called its mean value) and its variance. The expected value is the average of the values taken on by repeatedly sampling the random variable. More precisely

Definition: Consider a random variable Y that takes on the possible values $y_1, \ldots, y_n$. The **expected value** of Y, $E[Y]$, is

$$E[Y] \equiv \sum_{i=1}^{n} y_i \Pr(Y = y_i) \tag{5.3}$$

For example, if Y takes on the value 1 with probability .7 and the value 2 with probability .3, then its expected value is $(1 \cdot 0.7 + 2 \cdot 0.3 = 1.3)$. In case the random variable Y is governed by a Binomial distribution, then it can be shown that

$$E[Y] = np \tag{5.4}$$

where n and p are the parameters of the Binomial distribution defined in Equation (5.2).

A second property, the variance, captures the "width" or "spread" of the probability distribution; that is, it captures how far the random variable is expected to vary from its mean value.

Definition: The **variance** of a random variable Y, $Var[Y]$, is

$$Var[Y] \equiv E[(Y - E[Y])^2] \tag{5.5}$$

The variance describes the expected squared error in using a single observation of Y to estimate its mean $E[Y]$. The square root of the variance is called the *standard deviation* of Y, denoted σ_Y.

Definition: The **standard deviation** of a random variable Y, σ_Y, is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]} \tag{5.6}$$

In case the random variable Y is governed by a Binomial distribution, then the variance and standard deviation are given by

$$Var[Y] = np(1-p)$$
$$\sigma_Y = \sqrt{np(1-p)} \tag{5.7}$$

5.3.4 Estimators, Bias, and Variance

Now that we have shown that the random variable $error_S(h)$ obeys a Binomial distribution, we return to our primary question: What is the likely difference between $error_S(h)$ and the true error $error_{\mathcal{D}}(h)$?

Let us describe $error_S(h)$ and $error_{\mathcal{D}}(h)$ using the terms in Equation (5.2) defining the Binomial distribution. We then have

$$error_S(h) = \frac{r}{n}$$
$$error_{\mathcal{D}}(h) = p$$

where n is the number of instances in the sample S, r is the number of instances from S misclassified by h, and p is the probability of misclassifying a single instance drawn from $\mathcal{D}$.

Statisticians call $error_S(h)$ an *estimator* for the true error $error_{\mathcal{D}}(h)$. In general, an estimator is any random variable used to estimate some parameter of the underlying population from which the sample is drawn. An obvious question to ask about any estimator is whether on average it gives the right estimate. We define the *estimation bias* to be the difference between the expected value of the estimator and the true value of the parameter.

Definition: The **estimation bias** of an estimator Y for an arbitrary parameter p is

$$E[Y] - p$$

If the estimation bias is zero, we say that Y is an *unbiased estimator* for p. Notice this will be the case if the average of many random values of Y generated by repeated random experiments (i.e., $E[Y]$) converges toward p.

Is $error_S(h)$ an unbiased estimator for $error_{\mathcal{D}}(h)$? Yes, because for a Binomial distribution the expected value of r is equal to np (Equation [5.4]). It follows, given that n is a constant, that the expected value of r/n is p.

Two quick remarks are in order regarding the estimation bias. First, when we mentioned at the beginning of this chapter that testing the hypothesis on the training examples provides an optimistically biased estimate of hypothesis error, it is exactly this notion of estimation bias to which we were referring. In order for $error_S(h)$ to give an unbiased estimate of $error_{\mathcal{D}}(h)$, the hypothesis h and sample S must be chosen independently. Second, this notion of *estimation bias* should not be confused with the *inductive bias* of a learner introduced in Chapter 2. The

estimation bias is a numerical quantity, whereas the inductive bias is a set of assertions.

A second important property of any estimator is its variance. Given a choice among alternative unbiased estimators, it makes sense to choose the one with least variance. By our definition of variance, this choice will yield the smallest expected squared error between the estimate and the true value of the parameter.

To illustrate these concepts, suppose we test a hypothesis and find that it commits $r = 12$ errors on a sample of $n = 40$ randomly drawn test examples. Then an unbiased estimate for $error_{\mathcal{D}}(h)$ is given by $error_S(h) = r/n = 0.3$. The variance in this estimate arises completely from the variance in r, because n is a constant. Because r is Binomially distributed, its variance is given by Equation (5.7) as $np(1-p)$. Unfortunately p is unknown, but we can substitute our estimate r/n for p. This yields an estimated variance in r of $40 \cdot 0.3(1 - 0.3) = 8.4$, or a corresponding standard deviation of $\sqrt{8.4} \approx 2.9$. This implies that the standard deviation in $error_S(h) = r/n$ is approximately $2.9/40 = .07$. To summarize, $error_S(h)$ in this case is observed to be 0.30, with a standard deviation of approximately 0.07. (See Exercise 5.1.)

In general, given r errors in a sample of n independently drawn test examples, the standard deviation for $error_S(h)$ is given by

$$\sigma_{error_S(h)} = \frac{\sigma_r}{n} = \sqrt{\frac{p(1-p)}{n}} \tag{5.8}$$

which can be approximated by substituting $r/n = error_S(h)$ for p

$$\sigma_{error_S(h)} \approx \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \tag{5.9}$$

6.3.1 Brute-Force Bayes Concept Learning

Consider the concept learning problem first introduced in Chapter 2. In particular, assume the learner considers some finite hypothesis space H defined over the instance space X, in which the task is to learn some target concept $c : X \rightarrow \{0, 1\}$. As usual, we assume that the learner is given some sequence of training examples $\langle\langle x_1, d_1\rangle \ldots \langle x_m, d_m\rangle\rangle$ where x_i is some instance from X and where d_i is the target value of x_i (i.e., $d_i = c(x_i)$). To simplify the discussion in this section, we assume the sequence of instances $\langle x_1 \ldots x_m\rangle$ is held fixed, so that the training data D can be written simply as the sequence of target values $D = \langle d_1 \ldots d_m\rangle$. It can be shown (see Exercise 6.4) that this simplification does not alter the main conclusions of this section.

We can design a straightforward concept learning algorithm to output the maximum a posteriori hypothesis, based on Bayes theorem, as follows:

BRUTE-FORCE MAP LEARNING algorithm

1. For each hypothesis h in H, calculate the posterior probability

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

2. Output the hypothesis h_{MAP} with the highest posterior probability

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(h|D)$$

This algorithm may require significant computation, because it applies Bayes theorem to each hypothesis in H to calculate $P(h|D)$. While this may prove impractical for large hypothesis spaces, the algorithm is still of interest because it provides a standard against which we may judge the performance of other concept learning algorithms.

In order specify a learning problem for the BRUTE-FORCE MAP LEARNING algorithm we must specify what values are to be used for $P(h)$ and for $P(D|h)$ (as we shall see, $P(D)$ will be determined once we choose the other two). We may choose the probability distributions $P(h)$ and $P(D|h)$ in any way we wish, to describe our prior knowledge about the learning task. Here let us choose them to be consistent with the following assumptions:

1. The training data D is noise free (i.e., $d_i = c(x_i)$).
2. The target concept c is contained in the hypothesis space H.
3. We have no a priori reason to believe that any hypothesis is more probable than any other.

Given these assumptions, what values should we specify for $P(h)$? Given no prior knowledge that one hypothesis is more likely than another, it is reasonable to assign the same prior probability to every hypothesis h in H. Furthermore, because we assume the target concept is contained in H we should require that these prior probabilities sum to 1. Together these constraints imply that we should choose

$$P(h) = \frac{1}{|H|} \quad \text{for all } h \text{ in } H$$

What choice shall we make for $P(D|h)$? $P(D|h)$ is the probability of observing the target values $D = \langle d_1 \ldots d_m \rangle$ for the fixed set of instances $\langle x_1 \ldots x_m \rangle$, given a world in which hypothesis h holds (i.e., given a world in which h is the correct description of the target concept c). Since we assume noise-free training data, the probability of observing classification d_i given h is just 1 if $d_i = h(x_i)$ and 0 if $d_i \neq h(x_i)$. Therefore,

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases} \tag{6.4}$$

In other words, the probability of data D given hypothesis h is 1 if D is consistent with h, and 0 otherwise.

Given these choices for $P(h)$ and for $P(D|h)$ we now have a fully-defined problem for the above BRUTE-FORCE MAP LEARNING algorithm. Let us consider the first step of this algorithm, which uses Bayes theorem to compute the posterior probability $P(h|D)$ of each hypothesis h given the observed training data D.

Recalling Bayes theorem, we have

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

First consider the case where h is inconsistent with the training data D. Since Equation (6.4) defines $P(D|h)$ to be 0 when h is inconsistent with D, we have

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0 \text{ if } h \text{ is inconsistent with } D$$

The posterior probability of a hypothesis inconsistent with D is zero.

Now consider the case where h is consistent with D. Since Equation (6.4) defines $P(D|h)$ to be 1 when h is consistent with D, we have

$$\begin{aligned} P(h|D) &= \frac{1 \cdot \frac{1}{|H|}}{P(D)} \\ &= \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}} \\ &= \frac{1}{|VS_{H,D}|} \text{ if } h \text{ is consistent with } D \end{aligned}$$

where $VS_{H,D}$ is the subset of hypotheses from H that are consistent with D (i.e., $VS_{H,D}$ is the version space of H with respect to D as defined in Chapter 2). It is easy to verify that $P(D) = \frac{|VS_{H,D}|}{|H|}$ above, because the sum over all hypotheses of $P(h|D)$ must be one and because the number of hypotheses from H consistent with D is by definition $|VS_{H,D}|$. Alternatively, we can derive $P(D)$ from the theorem of total probability (see Table 6.1) and the fact that the hypotheses are mutually exclusive (i.e., $(\forall i \neq j)(P(h_i \wedge h_j) = 0)$)

$$\begin{aligned} P(D) &= \sum_{h_i \in H} P(D|h_i)P(h_i) \\ &= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} + \sum_{h_i \notin VS_{H,D}} 0 \cdot \frac{1}{|H|} \\ &= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} \\ &= \frac{|VS_{H,D}|}{|H|} \end{aligned}$$

To summarize, Bayes theorem implies that the posterior probability $P(h|D)$ under our assumed $P(h)$ and $P(D|h)$ is

$$P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|} \text{ if } h \text{ is consistent with } D \\ 0 \quad \text{otherwise} \end{cases} \tag{6.5}$$

where $|VS_{H,D}|$ is the number of hypotheses from H consistent with D. The evolution of probabilities associated with hypotheses is depicted schematically in Figure 6.1. Initially (Figure 6.1*a*) all hypotheses have the same probability. As training data accumulates (Figures 6.1*b* and 6.1*c*), the posterior probability for inconsistent hypotheses becomes zero while the total probability summing to one is shared equally among the remaining consistent hypotheses.

The above analysis implies that under our choice for $P(h)$ and $P(D|h)$, every *consistent* hypothesis has posterior probability $(1/|VS_{H,D}|)$, and every inconsistent hypothesis has posterior probability 0. Every consistent hypothesis is, therefore, a MAP hypothesis.

6.3.2 MAP Hypotheses and Consistent Learners

The above analysis shows that in the given setting, every hypothesis consistent with D is a MAP hypothesis. This statement translates directly into an interesting statement about a general class of learners that we might call *consistent learners*. We will say that a learning algorithm is a *consistent learner* provided it outputs a hypothesis that commits zero errors over the training examples. Given the above analysis, we can conclude that *every consistent learner outputs a MAP hypothesis, if we assume a uniform prior probability distribution over H (i.e., $P(h_i) = P(h_j)$ for all i, j), and if we assume deterministic, noise-free training data (i.e., $P(D|h) = 1$ if D and h are consistent, and 0 otherwise).*

Consider, for example, the concept learning algorithm FIND-S discussed in Chapter 2. FIND-S searches the hypothesis space H from specific to general hypotheses, outputting a maximally specific consistent hypothesis (i.e., a maximally specific member of the version space). Because FIND-S outputs a consistent hypothesis, we know that it will output a MAP hypothesis under the probability distributions $P(h)$ and $P(D|h)$ defined above. Of course FIND-S does not explicitly manipulate probabilities at all—it simply outputs a maximally specific member

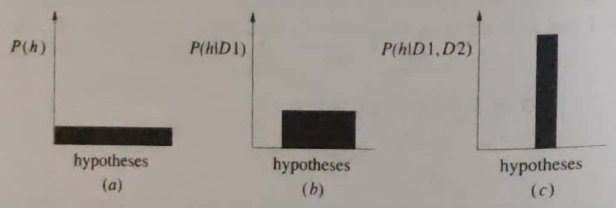

FIGURE 6.1
Evolution of posterior probabilities $P(h|D)$ with increasing training data. (*a*) Uniform priors assign equal probability to each hypothesis. As training data increases first to $D1$ (*b*), then to $D1 \wedge D2$ (*c*), the posterior probability of inconsistent hypotheses becomes zero, while posterior probabilities increase for hypotheses remaining in the version space.

of the version space. However, by identifying distributions for $P(h)$ and $P(D|h)$ under which its output hypotheses will be MAP hypotheses, we have a useful way of characterizing the behavior of FIND-S.

Are there other probability distributions for $P(h)$ and $P(D|h)$ under which FIND-S outputs MAP hypotheses? Yes. Because FIND-S outputs a *maximally specific* hypothesis from the version space, its output hypothesis will be a MAP hypothesis relative to any prior probability distribution that favors more specific hypotheses. More precisely, suppose $\mathcal{H}$ is any probability distribution $P(h)$ over H that assigns $P(h_1) \geq P(h_2)$ if h_1 is more specific than h_2. Then it can be shown that FIND-S outputs a MAP hypothesis assuming the prior distribution $\mathcal{H}$ and the same distribution $P(D|h)$ discussed above.

To summarize the above discussion, the Bayesian framework allows one way to characterize the behavior of learning algorithms (e.g., FIND-S), even when the learning algorithm does not explicitly manipulate probabilities. By identifying probability distributions $P(h)$ and $P(D|h)$ under which the algorithm outputs optimal (i.e., MAP) hypotheses, we can characterize the implicit assumptions under which this algorithm behaves optimally.

Using the Bayesian perspective to characterize learning algorithms in this way is similar in spirit to characterizing the inductive bias of the learner. Recall that in Chapter 2 we defined the inductive bias of a learning algorithm to be the set of assumptions B sufficient to *deductively* justify the inductive inference performed by the learner. For example, we described the inductive bias of the CANDIDATE-ELIMINATION algorithm as the assumption that the target concept c is included in the hypothesis space H. Furthermore, we showed there that the output of this learning algorithm follows deductively from its inputs plus this implicit inductive bias assumption. The above Bayesian interpretation provides an alternative way to characterize the assumptions implicit in learning algorithms. Here, instead of modeling the inductive inference method by an equivalent deductive system, we model it by an equivalent *probabilistic reasoning* system based on Bayes theorem. And here the implicit assumptions that we attribute to the learner are assumptions of the form "the prior probabilities over H are given by the distribution $P(h)$, and the strength of data in rejecting or accepting a hypothesis is given by $P(D|h)$." The definitions of $P(h)$ and $P(D|h)$ given in this section characterize the implicit assumptions of the CANDIDATE-ELIMINATION and FIND-S algorithms. A probabilistic reasoning system based on Bayes theorem will exhibit input-output behavior equivalent to these algorithms, provided it is given these assumed probability distributions.

The discussion throughout this section corresponds to a special case of Bayesian reasoning, because we considered the case where $P(D|h)$ takes on values of only 0 and 1, reflecting the deterministic predictions of hypotheses and the assumption of noise-free training data. As we shall see in the next section, we can also model learning from noisy training data, by allowing $P(D|h)$ to take on values other than 0 and 1, and by introducing into $P(D|h)$ additional assumptions about the probability distributions that govern the noise.

6.4 MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

As illustrated in the above section, Bayesian analysis can sometimes be used to show that a particular learning algorithm outputs MAP hypotheses even though it may not explicitly use Bayes rule or calculate probabilities in any form.

In this section we consider the problem of learning a continuous-valued target function—a problem faced by many learning approaches such as neural network learning, linear regression, and polynomial curve fitting. A straightforward Bayesian analysis will show that *under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis.* The significance of this result is that it provides a Bayesian justification (under certain assumptions) for many neural network and other curve fitting methods that attempt to minimize the sum of squared errors over the training data.

Consider the following problem setting. Learner L considers an instance space X and a hypothesis space H consisting of some class of real-valued functions defined over X (i.e., each h in H is a function of the form $h : X \rightarrow \Re$, where $\Re$ represents the set of real numbers). The problem faced by L is to learn an unknown target function $f : X \rightarrow \Re$ drawn from H. A set of m training examples is provided, where the target value of each example is corrupted by random noise drawn according to a Normal probability distribution. More precisely, each training example is a pair of the form $\langle x_i, d_i \rangle$ where $d_i = f(x_i) + e_i$. Here $f(x_i)$ is the noise-free value of the target function and e_i is a random variable representing the noise. It is assumed that the values of the e_i are drawn independently and that they are distributed according to a Normal distribution with zero mean. The task of the learner is to output a maximum likelihood hypothesis, or, equivalently, a MAP hypothesis assuming all hypotheses are equally probable a priori.

A simple example of such a problem is learning a linear function, though our analysis applies to learning arbitrary real-valued functions. Figure 6.2 illustrates

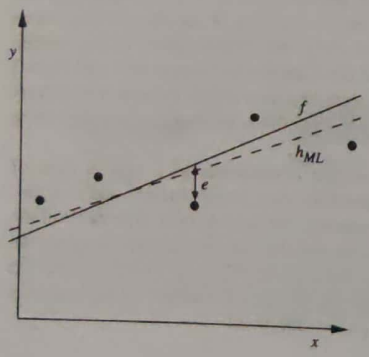

FIGURE 6.2
Learning a real-valued function. The target function f corresponds to the solid line. The training examples $\langle x_i, d_i \rangle$ are assumed to have Normally distributed noise e_i with zero mean added to the true target value $f(x_i)$. The dashed line corresponds to the linear function that minimizes the sum of squared errors. Therefore, it is the maximum likelihood hypothesis h_{ML}, given these five training examples.

a linear target function f depicted by the solid line, and a set of noisy training examples of this target function. The dashed line corresponds to the hypothesis h_{ML} with least-squared training error, hence the maximum likelihood hypothesis. Notice that the maximum likelihood hypothesis is not necessarily identical to the correct hypothesis, f, because it is inferred from only a limited sample of noisy training data.

Before showing why a hypothesis that minimizes the sum of squared errors in this setting is also a maximum likelihood hypothesis, let us quickly review two basic concepts from probability theory: probability densities and Normal distributions. First, in order to discuss probabilities over continuous variables such as e, we must introduce probability *densities*. The reason, roughly, is that we wish for the total probability over all possible values of the random variable to sum to one. In the case of continuous variables we cannot achieve this by assigning a finite probability to each of the infinite set of possible values for the random variable. Instead, we speak of a probability *density* for continuous variables such as e and require that the integral of this probability density over all possible values be one. In general we will use lower case p to refer to the probability density function, to distinguish it from a finite probability P (which we will sometimes refer to as a probability *mass*). The probability density $p(x_0)$ is the limit as ϵ goes to zero, of $\frac{1}{\epsilon}$ times the probability that x will take on a value in the interval $[x_0, x_0 + \epsilon)$.

Probability density function:

$$p(x_0) \equiv \lim_{\epsilon \rightarrow 0} \frac{1}{\epsilon} P(x_0 \leq x < x_0 + \epsilon)$$

Second, we stated that the random noise variable e is generated by a Normal probability distribution. A Normal distribution is a smooth, bell-shaped distribution that can be completely characterized by its mean μ and its standard deviation σ. See Table 5.4 for a precise definition.

Given this background we now return to the main issue: showing that the least-squared error hypothesis is, in fact, the maximum likelihood hypothesis within our problem setting. We will show this by deriving the maximum likelihood hypothesis starting with our earlier definition Equation (6.3), but using lower case p to refer to the probability density

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}}\, p(D|h)$$

As before, we assume a fixed set of training instances $\langle x_1 \ldots x_m \rangle$ and therefore consider the data D to be the corresponding sequence of target values $D = \langle d_1 \ldots d_m \rangle$. Here $d_i = f(x_i) + e_i$. Assuming the training examples are mutually independent given h, we can write $P(D|h)$ as the product of the various $p(d_i|h)$

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} p(d_i|h)$$

6.9 NAIVE BAYES CLASSIFIER

One highly practical Bayesian learning method is the naive Bayes learner, often called the *naive Bayes classifier*. In some domains its performance has been shown to be comparable to that of neural network and decision tree learning. This section introduces the naive Bayes classifier; the next section applies it to the practical problem of learning to classify natural language text documents.

The naive Bayes classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target function $f(x)$ can take on any value from some finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values $\langle a_1, a_2 \ldots a_n \rangle$. The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach to classifying the new instance is to assign the most probable target value, v_{MAP}, given the attribute values $\langle a_1, a_2 \ldots a_n \rangle$ that describe the instance.

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

We can use Bayes theorem to rewrite this expression as

$$\begin{aligned} v_{MAP} &= \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)} \\ &= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j) \end{aligned} \tag{6.19}$$

Now we could attempt to estimate the two terms in Equation (6.19) based on the training data. It is easy to estimate each of the $P(v_j)$ simply by counting the frequency with which each target value v_j occurs in the training data. However, estimating the different $P(a_1, a_2 \ldots a_n | v_j)$ terms in this fashion is not feasible unless we have a very, very large set of training data. The problem is that the number of these terms is equal to the number of possible instances times the number of possible target values. Therefore, we need to see every instance in the instance space many times in order to obtain reliable estimates.

The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the conjunction $a_1, a_2 \ldots a_n$ is just the product of the probabilities for the individual attributes: $P(a_1, a_2 \ldots a_n | v_j) = \prod_i P(a_i | v_j)$. Substituting this into Equation (6.19), we have the approach used by the naive Bayes classifier.

Naive Bayes classifier:

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j) \tag{6.20}$$

where v_{NB} denotes the target value output by the naive Bayes classifier. Notice that in a naive Bayes classifier the number of distinct $P(a_i | v_j)$ terms that must

be estimated from the training data is just the number of distinct attribute values times the number of distinct target values—a much smaller number than if we were to estimate the $P(a_1, a_2 \ldots a_n|v_j)$ terms as first contemplated.

To summarize, the naive Bayes learning method involves a learning step in which the various $P(v_j)$ and $P(a_i|v_j)$ terms are estimated, based on their frequencies over the training data. The set of these estimates corresponds to the learned hypothesis. This hypothesis is then used to classify each new instance by applying the rule in Equation (6.20). Whenever the naive Bayes assumption of conditional independence is satisfied, this naive Bayes classification v_{NB} is identical to the MAP classification.

One interesting difference between the naive Bayes learning method and other learning methods we have considered is that there is no explicit search through the space of possible hypotheses (in this case, the space of possible hypotheses is the space of possible values that can be assigned to the various $P(v_j)$ and $P(a_i|v_j)$ terms). Instead, the hypothesis is formed without searching, simply by counting the frequency of various data combinations within the training examples.

6.9.1 An Illustrative Example

Let us apply the naive Bayes classifier to a concept learning problem we considered during our discussion of decision tree learning: classifying days according to whether someone will play tennis. Table 3.2 from Chapter 3 provides a set of 14 training examples of the target concept *PlayTennis*, where each day is described by the attributes *Outlook*, *Temperature*, *Humidity*, and *Wind*. Here we use the naive Bayes classifier and the training data from this table to classify the following novel instance:

$$\langle Outlook = sunny, Temperature = cool, Humidity = high, Wind = strong \rangle$$

Our task is to predict the target value (*yes* or *no*) of the target concept *PlayTennis* for this new instance. Instantiating Equation (6.20) to fit the current task, the target value v_{NB} is given by

$$\begin{aligned} v_{NB} &= \underset{v_j \in \{yes, no\}}{\text{argmax}} \; P(v_j) \prod_i P(a_i|v_j) \\ &= \underset{v_j \in \{yes, no\}}{\text{argmax}} \; P(v_j) \; P(Outlook = sunny|v_j) P(Temperature = cool|v_j) \\ &\qquad P(Humidity = high|v_j) P(Wind = strong|v_j) \end{aligned} \tag{6.21}$$

Notice in the final expression that a_i has been instantiated using the particular attribute values of the new instance. To calculate v_{NB} we now require 10 probabilities that can be estimated from the training data. First, the probabilities of the different target values can easily be estimated based on their frequencies over the 14 training examples

$$P(PlayTennis = yes) = 9/14 = .64$$
$$P(PlayTennis = no) = 5/14 = .36$$

Similarly, we can estimate the conditional probabilities. For example, those for $Wind = strong$ are

$$P(Wind = strong | PlayTennis = yes) = 3/9 = .33$$
$$P(Wind = strong | PlayTennis = no) = 3/5 = .60$$

Using these probability estimates and similar estimates for the remaining attribute values, we calculate v_{NB} according to Equation (6.21) as follows (now omitting attribute names for brevity)

$$P(yes)\ P(sunny|yes)\ P(cool|yes)\ P(high|yes)\ P(strong|yes) = .0053$$
$$P(no)\ P(sunny|no)\ P(cool|no)\ P(high|no)\ P(strong|no) = .0206$$

Thus, the naive Bayes classifier assigns the target value $PlayTennis = no$ to this new instance, based on the probability estimates learned from the training data. Furthermore, by normalizing the above quantities to sum to one we can calculate the conditional probability that the target value is *no*, given the observed attribute values. For the current example, this probability is $\frac{.0206}{.0206+.0053} = .795$.

8.2 *k*-NEAREST NEIGHBOR LEARNING

The most basic instance-based method is the *k*-NEAREST NEIGHBOR algorithm. This algorithm assumes all instances correspond to points in the n-dimensional space $\Re^n$. The nearest neighbors of an instance are defined in terms of the standard

Euclidean distance. More precisely, let an arbitrary instance x be described by the feature vector

$$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle$$

where $a_r(x)$ denotes the value of the rth attribute of instance x. Then the distance between two instances x_i and x_j is defined to be $d(x_i, x_j)$, where

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

In nearest-neighbor learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions of the form $f : \Re^n \rightarrow V$, where V is the finite set $\{v_1, \ldots v_s\}$. The k-NEAREST NEIGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1. As shown there, the value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of f among the k training examples nearest to x_q. If we choose $k = 1$, then the 1-NEAREST NEIGHBOR algorithm assigns to $\hat{f}(x_q)$ the value $f(x_i)$ where x_i is the training instance nearest to x_q. For larger values of k, the algorithm assigns the most common value among the k nearest training examples.

Figure 8.1 illustrates the operation of the k-NEAREST NEIGHBOR algorithm for the case where the instances are points in a two-dimensional space and where the target function is boolean valued. The positive and negative training examples are shown by "+" and "−" respectively. A query point x_q is shown as well. Note the 1-NEAREST NEIGHBOR algorithm classifies x_q as a positive example in this figure, whereas the 5-NEAREST NEIGHBOR algorithm classifies it as a negative example.

What is the nature of the hypothesis space H implicitly considered by the k-NEAREST NEIGHBOR algorithm? Note the k-NEAREST NEIGHBOR algorithm never forms an explicit general hypothesis $\hat{f}$ regarding the target function f. It simply computes the classification of each new query instance as needed. Nevertheless,

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

TABLE 8.1
The k-NEAREST NEIGHBOR algorithm for approximating a discrete-valued function $f : \Re^n \rightarrow V$.

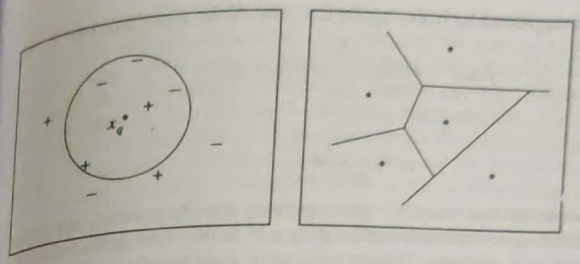

FIGURE 8.1
k-NEAREST NEIGHBOR. A set of positive and negative training examples is shown on the left, along with a query instance x_q to be classified. The 1-NEAREST NEIGHBOR algorithm classifies x_q positive, whereas 5-NEAREST NEIGHBOR classifies it as negative. On the right is the decision surface induced by the 1-NEAREST NEIGHBOR algorithm for a typical set of training examples. The convex polygon surrounding each training example indicates the region of instance space closest to that point (i.e., the instances for which the 1-NEAREST NEIGHBOR algorithm will assign the classification belonging to that training example).

we can still ask what the implicit general function is, or what classifications would be assigned if we were to hold the training examples constant and query the algorithm with every possible instance in X. The diagram on the right side of Figure 8.1 shows the shape of this decision surface induced by 1-NEAREST NEIGHBOR over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the *Voronoi diagram* of the set of training examples.

The k-NEAREST NEIGHBOR algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have the algorithm calculate the mean value of the k nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function $f : \Re^n \rightarrow \Re$ we replace the final line of the above algorithm by the line

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{8.1}$$

8.2.1 Distance-Weighted NEAREST NEIGHBOR Algorithm

One obvious refinement to the k-NEAREST NEIGHBOR algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight to closer neighbors. For example, in the algorithm of Table 8.1, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q.

CHAPTER 13

REINFORCEMENT LEARNING

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. This very generic problem covers tasks such as learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games. Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward. This chapter focuses on an algorithm called Q learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment. Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

13.1 INTRODUCTION

Consider building a learning robot. The robot, or *agent*, has a set of sensors to observe the *state* of its environment, and a set of *actions* it can perform to alter this state. For example, a mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn." Its task is to learn a control strategy, or *policy*, for choosing actions that achieve its goals. For example, the robot may have a goal of docking onto its battery charger whenever its battery level is low.

This chapter is concerned with how such agents can learn successful control policies by experimenting in their environment. We assume that the goals of the agent can be defined by a *reward* function that assigns a numerical value—an immediate payoff—to each distinct action the agent may take from each distinct state. For example, the goal of docking to the battery charger can be captured by assigning a positive reward (e.g., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition. This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot. The task of the robot is to perform sequences of actions, observe their consequences, and learn a control policy. The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent. This general setting for robot learning is summarized in Figure 13.1.

As is apparent from Figure 13.1, the problem of learning a control policy to maximize cumulative reward is very general and covers many problems beyond robot learning tasks. In general the problem is one of learning to control sequential processes. This includes, for example, manufacturing optimization problems in which a sequence of manufacturing actions must be chosen, and the reward to be maximized is the value of the goods produced minus the costs involved. It includes sequential scheduling problems such as choosing which taxis to send for passengers in a large city, where the reward to be maximized is a function of the wait time of the passengers and the total fuel costs of the taxi fleet. In general, we are interested in any type of agent that must learn to choose actions that alter the state of its environment and where a cumulative reward function is used to define the quality of any given action sequence. Within this class of problems we will consider specific settings, including settings in which the actions have deterministic or nondeterministic outcomes, and settings in which the agent

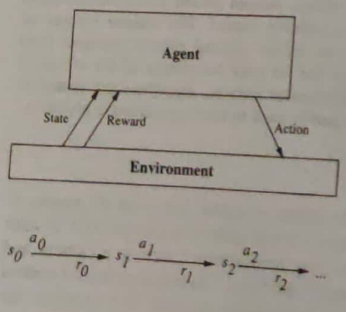

FIGURE 13.1
An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. It can perform any of a set of possible actions A. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the immediate value of this state-action transition. This produces a sequence of states s_i, actions a_i, and immediate rewards r_i as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

has or does not have prior knowledge about the effects of its actions on the environment.

Note we have touched on the problem of learning to control sequential processes earlier in this book. In Section 11.4 we discussed explanation-based learning of rules to control search during problem solving. There the problem is for the agent to choose among alternative actions at each step in its search for some goal state. The techniques discussed here differ from those of Section 11.4, in that here we consider problems where the actions may have nondeterministic outcomes and where the learner lacks a domain theory that describes the outcomes of its actions. In Chapter 1 we discussed the problem of learning to choose actions while playing the game of checkers. There we sketched the design of a learning method very similar to those discussed in this chapter. In fact, one highly successful application of the reinforcement learning algorithms of this chapter is to a similar game-playing problem. Tesauro (1995) describes the TD-GAMMON program, which has used reinforcement learning to become a world-class backgammon player. This program, after training on 1.5 million self-generated games, is now considered nearly equal to the best human players in the world and has played competitively against top-ranked players in international backgammon tournaments.

The problem of learning a control policy to choose actions is similar in some respects to the function approximation problems discussed in other chapters. The target function to be learned in this case is a control policy, $\pi : S \rightarrow A$, that outputs an appropriate action a from the set A, given the current state s from the set S. However, this reinforcement learning problem differs from other function approximation tasks in several important respects.

- *Delayed reward.* The task of the agent is to learn a target function π that maps from the current state s to the optimal action $a = \pi(s)$. In earlier chapters we have always assumed that when learning some target function such as π, each training example would be a pair of the form $(s, \pi(s))$. In reinforcement learning, however, training information is not available in this form. Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of *temporal credit assignment*: determining which of the actions in its sequence are to be credited with producing the eventual rewards.
- *Exploration.* In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a tradeoff in choosing whether to favor *exploration* of unknown states and actions (to gather new information), or *exploitation* of states and actions that it has already learned will yield high reward (to maximize its cumulative reward).
- *Partially observable states.* Although it is convenient to assume that the agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. For example, a robot with a forward-pointing camera cannot see what is

behind it. In such cases, it may be necessary for the agent to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

- *Life-long learning.* Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

13.3 Q LEARNING

How can an agent learn an optimal policy π^* for an arbitrary environment? It is difficult to learn the function $\pi^* : S \rightarrow A$ directly, because the available training data does not provide training examples of the form $\langle s, a \rangle$. Instead, the only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2, \ldots$. As we shall see, given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

What evaluation function should the agent attempt to learn? One obvious choice is V^*. The agent should prefer state s_1 over state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from s_1. Of course the agent's policy must choose among actions, not among states. However, it can use V^* in certain settings to choose among actions as well. The optimal action in state s is the action a that maximizes the sum of the immediate reward $r(s, a)$ plus the value V^* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s, a) + \gamma V^*(\delta(s, a))] \tag{13.3}$$

(recall that $\delta(s, a)$ denotes the state resulting from applying action a to state s.) Thus, the agent can acquire the optimal policy by learning V^*, *provided it has perfect knowledge of the immediate reward function* r *and the state transition function* δ. When the agent knows the functions r and δ used by the environment to respond to its actions, then it can then use Equation (13.3) to calculate the optimal action for any state s.

Unfortunately, learning V^* is a useful way to learn the optimal policy *only* when the agent has perfect knowledge of δ and r. This requires that it be able to perfectly predict the immediate result (i.e., the immediate reward and immediate successor) for every possible state-action transition. This assumption is comparable to the assumption of a perfect domain theory in explanation-based learning, discussed in Chapter 11. In many practical problems, such as robot control, it is impossible for the agent or its human programmer to predict in advance the exact outcome of applying an arbitrary action to an arbitrary state. Imagine, for example, the difficulty in describing δ for a robot arm shoveling dirt when the resulting state includes the positions of the dirt particles. In cases where either δ or r is unknown, learning V^* is unfortunately of no use for selecting optimal actions because the agent cannot evaluate Equation (13.3). What evaluation function should the agent use in this more general setting? The evaluation function Q, defined in the following section, provides one answer.

13.3.1 The Q Function

Let us define the evaluation function $Q(s, a)$ so that its value is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action. In other words, the value of Q is the reward received immediately upon executing action a from state s, plus the value (discounted by γ) of following the optimal policy thereafter.

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \tag{13.4}$$

Note that $Q(s, a)$ is exactly the quantity that is maximized in Equation (13.3) in order to choose the optimal action a in state s. Therefore, we can rewrite Equation (13.3) in terms of $Q(s, a)$ as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a) \tag{13.5}$$

Why is this rewrite important? Because it shows that if the agent learns the Q function instead of the V^* function, it will be able to select optimal actions *even when it has no knowledge of the functions r and δ*. As Equation (13.5) makes clear, it need only consider each available action a in its current state s and choose the action that maximizes $Q(s, a)$.

It may at first seem surprising that one can choose globally optimal action sequences by reacting repeatedly to the local values of Q for the current state. This means the agent can choose the optimal action without ever conducting a lookahead search to explicitly consider what state results from the action. Part of the beauty of Q learning is that the evaluation function is defined to have precisely this property—the value of Q for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s.

To illustrate, Figure 13.2 shows the Q values for every state and action in the simple grid world. Notice that the Q value for each state-action transition equals the r value for this transition plus the V^* value for the resulting state discounted by γ. Note also that the optimal policy shown in the figure corresponds to selecting actions with maximal Q values.

13.3.2 An Algorithm for Learning Q

Learning the Q function corresponds to learning the optimal policy. How can Q be learned?

The key problem is finding a reliable way to estimate training values for Q, given only a sequence of immediate rewards r spread out over time. This can be accomplished through iterative approximation. To see how, notice the close relationship between Q and V^*,

$$V^*(s) = \max_{a'} Q(s, a')$$

which allows rewriting Equation (13.4) as

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \tag{13.6}$$

This recursive definition of Q provides the basis for algorithms that iteratively approximate Q (Watkins 1989). To describe the algorithm, we will use the symbol $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual Q function. In this algorithm the learner represents its hypothesis $\hat{Q}$ by a large table with a separate entry for each state-action pair. The table entry for the pair $\langle s, a \rangle$ stores the value for $\hat{Q}(s, a)$—the learner's current hypothesis about the actual but unknown value $Q(s, a)$. The table can be initially filled with random values (though it is easier to understand the algorithm if one assumes initial values of zero). The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$. It then updates the table entry for $\hat{Q}(s, a)$ following each such transition, according to the rule:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \tag{13.7}$$

Note this training rule uses the agent's current $\hat{Q}$ values for the new state s' to refine its estimate of $\hat{Q}(s, a)$ for the previous state s. This training rule is motivated by Equation (13.6), although the training rule concerns the agent's approximation $\hat{Q}$, whereas Equation (13.6) applies to the actual Q function. Note although Equation (13.6) describes Q in terms of the functions $\delta(s, a)$ and $r(s, a)$, the agent does not need to know these general functions to apply the training rule of Equation (13.7). Instead it executes the action in its environment and then observes the resulting new state s' and reward r. Thus, it can be viewed as sampling these functions at the current values of s and a.

The above Q learning algorithm for deterministic Markov decision processes is described more precisely in Table 13.1. Using this algorithm the agent's estimate $\hat{Q}$ converges in the limit to the actual Q function, provided the system can be modeled as a deterministic Markov decision process, the reward function r is

---

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

TABLE 13.1
Q learning algorithm, assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$.