CHAPTER

Probabilistic methods

9

CHAPTER OUTLINE

Data Mining. DOI: http://dx.doi.org/10.1016/B978-0-12-804291-5.00009-X

© 2017 Elsevier Inc. All rights reserved.

Probabilistic methods form the basis of a plethora of techniques for data mining and machine learning. In Section 4.2, "Simple probabilistic modeling," we encountered the idea of choosing a model that maximizes the likelihood of an event, and have referred to the general idea of maximizing likelihoods several times since. In this chapter we will formalize the notion of likelihoods, and see how maximizing them underpins many estimation problems. We will look at Bayesian networks, and other types of probabilistic models used in machine learning. Let us begin by establishing the foundations: some fundamental rules of probability.

9.1 FOUNDATIONS

In probability modeling, example data or instances are often thought of as being events, observations, or realizations of underlying *random variables*. Given a discrete random variable A, $P(A)$ is a function that encodes the probabilities for each of the categories, classes, or states that A may be in. Similarly, for continuous random variables x, $p(x)$ is a function that assigns a probability density to all possible values of x. In contrast, $P(A = a)$ is the single probability of observing the specific event $A = a$. This notation is often simplified to $P(a)$, but one needs to be careful to remember whether a was defined as a random variable or as an observation. Similarly for the observation that continuous random variable x has the value x_1: it is common to write this probability density as $p(x_1)$, but this is a simplification of the longer but clearer notation $p(x = x_1)$, which emphasizes that it is the scalar value obtained by evaluating the function at $x = x_1$.

There are a few rules of probability theory that are particularly relevant to this book. They go by various names, but we will refer to them as the product rule,

the sum (or marginalization) rule, and Bayes' rule. As we will see, these seemingly simple rules can take us far.

Discrete or binary events are used below to keep the notation simple. However, the rules can be applied to *binary*, *discrete*, or *continuous* events and variables. For continuous variables, sums over possible states are replaced by integrals.

The *product rule*, sometimes referred to as the "fundamental rule of probability," states that the joint probability of random variables A and B can be written

$$P(A, B) = P(A|B)P(B).$$

The product rule also applies when A and B are groups or subsets of events or random variables.

The *sum rule* states that given the joint probability of variables X_1, X_2,..., X_N, the *marginal probability* for a given variable can be obtained by summing (or integrating) over all the other variables. For example, to obtain the marginal probability of X_1, sum over all the states of all the other variables:

$$P(X_1) = \sum_{x_2} \dots \sum_{x_N} P(X_1, X_2 = x_2, \dots, X_N = x_N),$$

The sums are taken over all possible values for the corresponding variable. This notation can be simplified by writing

$$p(x_1) = \sum_{x_2} \dots \sum_{x_N} P(x_1, x_2, \dots, x_N).$$

For continuous events and variables x_1, x_2,..., x_N, the equivalent formulation is obtained by integrating rather than summing:

$$p(x_1) = \int_{x_2} \dots \int_{x_N} p(x_1, x_2, \dots, x_N) dx_2 \dots dx_N.$$

This can give the marginal distribution of any random variable, or of any subset of random variables.

The famous *Bayes' rule* that was introduced in Chapter 4, Algorithms: the basic methods, can be obtained from the product rule by swapping A and B, observing that $P(B|A)P(A) = P(A|B)P(B)$, and rearranging:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}.$$

Suppose we have models for $P(A|B)$ and $P(B)$, observe that event $A = a$, and want to compute $P(B|A = a)$. $P(A = a|B)$ is referred to as the *likelihood*, $P(B)$ as the *prior* distribution of B, and $P(B|A = a)$ as the *posterior* distribution. $P(A = a)$ is obtained from the sum rule:

$$P(A = a) = \sum_b P(A = a, B = b) = \sum_b P(A = a|B = b)P(B = b).$$

These concepts can be applied to random variables, and also to parameters when they are treated as random quantities.

MAXIMUM LIKELIHOOD ESTIMATION

Consider the problem of estimating a set of parameters θ of a probabilistic model, given a set of *observations* x_1, x_2,. . ., x_n. Assume they are continuous-valued observations—but the same idea applies to discrete data too. Maximum likelihood techniques assume that (1) the examples have no dependence on one another, in that the occurrence of one has no effect on the others, and (2) they can each be modeled in exactly the same way. These assumptions are often summarized by saying that events are *independent and identically distributed* (i.i.d.). Although this is rarely completely true, it is sufficiently true in many situations to support useful inferences. Furthermore, as we will see later in this chapter, dependency structure can be captured by more sophisticated models—e.g., by treating interdependent groups of observations as part of a larger instance.

The i.i.d. assumption implies that a model for the joint probability density function for all observations consists of the product of the same probability model $p(x_i;\theta)$ applied to each observation independently. For n observations, this could be written

$$p(x_1, x_2, \ldots, x_n; \theta) = p(x_1; \theta)p(x_2; \theta)\ldots p(x_n; \theta).$$

Each function $p(x_i;\theta)$ has the same parameter values θ, and the aim of parameter estimation is to maximize a joint probability model of this form. Since the observations do not change, this value can only be changed by altering the choice of the parameters θ. We can think about this value as the *likelihood* of the data, and write it as

$$L(\theta; x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i; \theta).$$

Since the data is fixed, it is arguably more useful to think of this as a likelihood function for the parameters, which we are free to choose.

Multiplying many probabilities can lead to very small numbers, and so people often work with the logarithm of the likelihood, or *log-likelihood*:

$$\log L(\theta; x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} \log p(x_i; \theta),$$

which converts the product into a sum. Since logarithms are strictly monotonically increasing functions, maximizing the log-likelihood is the same as maximizing the likelihood. "Maximum likelihood" learning refers to techniques that search for parameters that do this:

$$\theta_{\mathrm{ML}} = \arg\max_{\theta} \sum_{i=1}^{n} \log p(x_i; \theta).$$

The same formulation also works for conditional probabilities and conditional likelihoods. Given some labels y_i that accompany each x_i, e.g., class labels of

instances in a classification task, maximum conditional likelihood learning corresponds to determining

$$\theta_{\text{MCL}} = \arg\max_{\theta} \sum_{i=1}^{n} \log p(y_i|x_i;\theta).$$

MAXIMUM A POSTERIORI PARAMETER ESTIMATION

Maximum likelihood assumes that all parameter values are equally likely a priori: we do not judge some parameter values to be more likely than others before we have considered the observations. But suppose we have reason to believe that the model's parameters follow a certain prior distribution. Thinking of them as random variables that specify each instance of the model, Bayes' rule can be applied to compute a posterior distribution of the parameters using the joint probability of data and parameters:

$$p(\theta|x_1, x_2, \ldots, x_n) = \frac{p(x_1, x_2, \ldots, x_n|\theta)p(\theta)}{p(x_1, x_2, \ldots, x_n)}.$$

Since we are computing the posterior distribution over the parameters, we have used | or the "given" notation in place of the semicolon. The denominator is a constant, and assuming i.i.d. observations, the posterior probability of the parameters is proportional to the product of the likelihood and prior:

$$p(\theta|x_1, x_2, \ldots, x_n) \propto \prod_{i=1}^{n} p(x_i;\theta)p(\theta).$$

Switching to logarithms again, the maximum a posteriori parameter estimation procedure seeks a value

$$\theta_{\text{MAP}} = \arg\max_{\theta} \left[\sum\nolimits_{i=1}^{n} \log p(x_i;\theta) + \log p(\theta)\right].$$

Again, the same idea can be applied to learn conditional probability models.

We have reverted to the semicolon notation to emphasize that maximum a posteriori parameter estimation involves point estimates of the parameters, evaluating them under the likelihood and prior distributions. This contrasts with fully Bayesian methods (discussed below) that explicitly manipulate the distribution of the parameters, typically by integrating over the parameter's uncertainty instead of optimizing a point estimate. The use of the "given" notation in place of the semicolon is more commonly used with fully Bayesian methods, and we will follow this convention.

9.2 BAYESIAN NETWORKS

The Naïve Bayes classifier of Section 4.2 and the logistic regression models of Section 4.6 both produce probability estimates rather than hard classifications.

For each class value, they estimate the probability that a given instance belongs to that class. Most other types of classifiers can be coerced into yielding this kind of information if necessary. For example, probabilities can be obtained from a decision tree by computing the relative frequency of each class in a leaf and from a decision list by examining the instances that a particular rule covers.

Probability estimates are often more useful than plain predictions. They allow predictions to be ranked, and their expected cost to be minimized (see Section 5.8). In fact, there is a strong argument for treating classification learning as the task of learning class probability estimates from data. What is being estimated is the conditional probability distribution of the values of the class attribute given the values of the other attributes. Ideally, the classification model represents this conditional distribution in a concise and easily comprehensible form.

Viewed in this way, Naïve Bayes classifiers, logistic regression models, decision trees, and so on are just alternative ways of representing a conditional probability distribution. Of course, they differ in representational power. Naïve Bayes classifiers and logistic regression models can only represent simple distributions, whereas decision trees can represent—or at least approximate—arbitrary distributions. However, decision trees have their drawbacks: they fragment the training set into smaller and smaller pieces, which inevitably yield less reliable probability estimates, and they suffer from the replicated subtree problem described in Section 3.4. Rule sets go some way toward addressing these shortcomings, but the design of a good rule learner is guided by heuristics with scant theoretical justification.

Does this mean that we have to accept our fate and live with these shortcomings? No! There is a statistically based alternative: a theoretically well-founded way of representing probability distributions concisely and comprehensibly in a graphical manner. The structures are called *Bayesian networks*. They are drawn as a network of nodes, one for each attribute, connected by directed edges in such a way that there are no cycles—a *directed acyclic graph*.

In this section, to explain how to interpret Bayesian networks and how to learn them from data we will make some simplifying assumptions. We assume that all attributes are nominal—they correspond to discrete random variables—and that there are no missing values, so the data is complete. Some advanced learning algorithms can create new attributes in addition to the ones present in the data—so-called hidden attributes corresponding to latent variables whose values cannot be observed. These can support better models if they represent salient features of the underlying problem, and Bayesian networks provide a good way of using them at prediction time. However, they make both learning and prediction far more complex and time consuming, so we will defer considering them to Section 9.4.

MAKING PREDICTIONS

Fig. 9.1 shows a simple Bayesian network for the weather data. It has a node for each of the four attributes *outlook*, *temperature*, *humidity*, and *windy* and one for the class attribute *play*. An edge leads from the *play* node to each of the

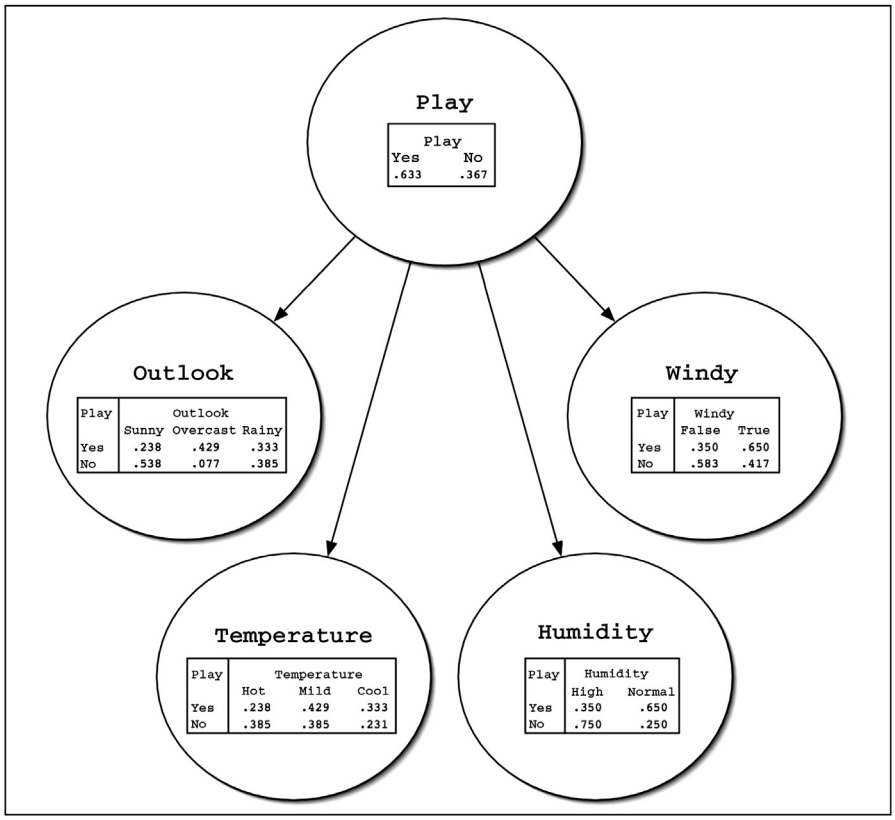

FIGURE 9.1

A simple Bayesian network for the weather data.

other nodes. But in Bayesian networks the structure of the graph is only half the story. Fig. 9.1 shows a table inside each node. The information in the tables defines a probability distribution that is used to predict the class probabilities for any given instance.

Before looking at how to compute this probability distribution, consider the information in the tables. The lower four tables (for *outlook*, *temperature*, *humidity*, and *windy*) have two parts separated by a vertical line. On the left are the values of *play*, and on the right are the corresponding probabilities for each value of the attribute represented by the node. In general, the left side contains a column for every edge pointing to the node, in this case just an edge emanating from the node for the *play* attribute. That is why the table associated with *play* itself does not have a left side: it has no parents. In general, each row of probabilities corresponds to one combination of values of the parent attributes, and the entries in the row show the probability of each value of the node's attribute given this

combination. In effect, each row defines a probability distribution over the values of the node's attribute. The entries in a row always sum to 1.

Fig. 9.2 shows a more complex network for the same problem, where three nodes (*windy*, *temperature*, and *humidity*) have two parents. Again, there is one column on the left for each parent and as many columns on the right as the attribute has values. Consider the first row of the table associated with the

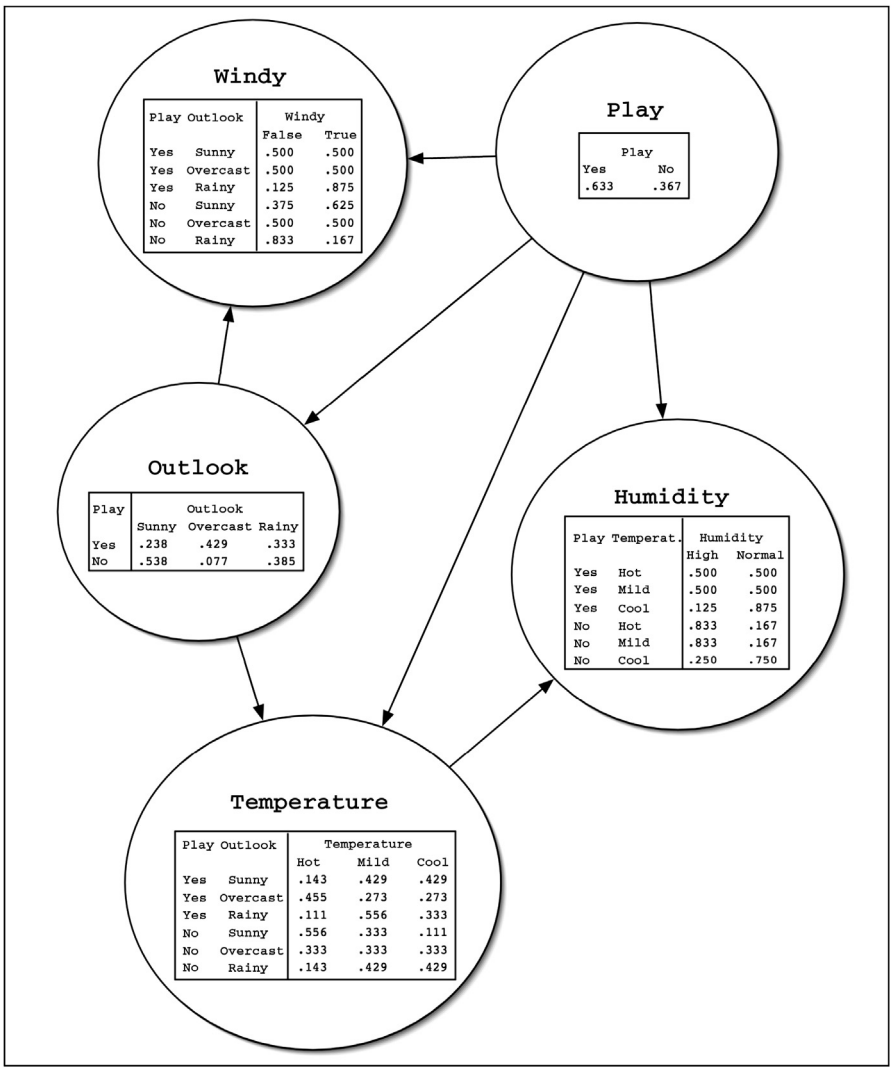

FIGURE 9.2

Another Bayesian network for the weather data.

temperature node. The left side gives a value for each parent attribute, *play* and *outlook*; the right gives a probability for each value of *temperature*. For example, the first number (0.143) is the probability of *temperature* taking on the value *hot*, given that *play* and *outlook* have values *yes* and *sunny*, respectively.

How are the tables used to predict the probability of each class value for a given instance? This turns out to be very easy, because we are assuming that there are no missing values. The instance specifies a value for each attribute. For each node in the network, look up the probability of the node's attribute value based on the row determined by its parents' attribute values. Then just multiply all these probabilities together.

For example, consider an instance with values *outlook* = *rainy*, *temperature* = *cool*, *humidity* = *high*, and *windy* = *true*. To calculate the probability for *play* = *no*, observe that the network in Fig. 9.2 gives probability 0.367 from node *play*, 0.385 from *outlook*, 0.429 from *temperature*, 0.250 from *humidity*, and 0.167 from *windy*. The product is 0.0025. The same calculation for *play* = *yes* yields 0.0077. However, these are clearly not the final answer: the final probabilities must sum to 1, whereas 0.0025 and 0.0077 do not. They are actually the joint probabilities $\mathrm{P}(play = no, E)$ and $\mathrm{P}(play = yes, E)$, where E denotes all the evidence given by the instance's attribute values. Joint probabilities measure the likelihood of observing an instance that exhibits the attribute values in E as well as the respective class value. They only sum to 1 if they exhaust the space of all possible attribute−value combinations, including the class attribute. This is certainly not the case in our example.

The solution is quite simple (we already encountered it in Section 4.2). To obtain the conditional probabilities $P(play = no|E)$ and $P(play = yes|E)$, normalize the joint probabilities by dividing them by their sum. This gives probability 0.245 for *play* = *no* and 0.755 for *play* = *yes*.

Just one mystery remains: Why multiply all those probabilities together? It turns out that the validity of the multiplication step hinges on a single assumption—namely that, given values for each of a node's parents, knowing the values for any other set of nondescendants does not change the probability associated with each of its possible values. In other words, other sets of nondescendants do not provide any information about the likelihood of the node's values over and above the information provided by the parents. This can be written

$$P(\text{node}|\text{parents plus any other nondescendants}) = P(\text{node}|\text{parents}),$$

which must hold for all values of the nodes and attributes involved. In statistics this property is called *conditional independence*. Multiplication is valid provided that each node is conditionally independent of its grandparents, great grandparents, and indeed any other set of nondescendants, given its parents. We have discussed above how the *product rule* of probability can be applied to sets of variables. Applying the product rule recursively between a single variable and the rest of the variables gives rise to another rule known as the *chain rule* which

states that the joint probability of n attributes A_i can be decomposed into the following product:

$$P(A_1, A_2, \ldots, A_n) = P(A_1) \prod_{i=1}^{n-1} P(A_{i+1} | A_i, A_{i-1}, \ldots, A_1).$$

The multiplications of probabilities in Bayesian networks follow as a direct result of the *chain rule*.

The decomposition holds for any order of the attributes. Because our Bayesian network is an acyclic graph, its nodes can be ordered to give all ancestors of a node a_i indices smaller than i. Then, because of the conditional independence assumption all Bayesian networks can be written in the form

$$P(A_1, A_2, \ldots, A_n) = \prod_{i=1}^{n} P(A_i | \text{Parents}(A_i)),$$

where when a variable has no parents, we use the unconditional probability of that variable. This is exactly the multiplication rule that we applied earlier.

The two Bayesian networks in Figs. 9.1 and 9.2 are fundamentally different. The first (Fig. 9.1) makes stronger independence assumptions because for each of its nodes the set of parents is a subset of the corresponding set of parents in the second (Fig. 9.2). In fact, Fig. 9.1 is almost identical to the simple Naïve Bayes classifier of Section 4.2 (The probabilities are slightly different but only because each count has been initialized to 0.5 to avoid the zero-frequency problem.) The network in Fig. 9.2 has more rows in the conditional probability tables and hence more parameters; it may be a more accurate representation of the underlying true probability distribution that generated the data.

It is tempting to assume that the directed edges in a Bayesian network represent causal effects. But be careful! In our case, a particular value of *play* may enhance the prospects of a particular value of *outlook*, but it certainly does not cause it—it is more likely to be the other way round. Different Bayesian networks can be constructed for the same problem, representing exactly the same probability distribution. This is done by altering the way in which the joint probability distribution is factorized to exploit conditional independencies. The network whose directed edges model causal effects is often the simplest one with the fewest parameters. Hence, human experts who construct Bayesian networks for a particular domain often benefit by representing causal effects by directed edges. However, when machine learning techniques are applied to induce models from data whose causal structure is unknown, all they can do is construct a network based on the correlations that are observed in the data. Inferring causality from correlation is always a dangerous business.

LEARNING BAYESIAN NETWORKS

The way to construct a learning algorithm for Bayesian networks is to define two components: a function for evaluating a given network based on the data and a

method for searching through the space of possible networks. The quality of a given network is measured by the probability of the data given the network. We calculate the probability that the network accords to each instance and multiply these probabilities together over all instances. In practice, this quickly yields numbers too small to be represented properly (called *arithmetic underflow*), so we use the sum of the logarithms of the probabilities rather than their product. The resulting quantity is the log-likelihood of the network given the data.

Assume that the structure of the network—the set of edges—is given. It is easy to estimate the numbers in the conditional probability tables: just compute the relative frequencies of the associated combinations of attribute values in the training data. To avoid the zero-frequency problem each count is initialized with a constant as described in Section 4.2. For example, to find the probability that *humidity* = *normal* given that *play* = *yes* and *temperature* = *cool* (the last number of the third row of the *humidity* node's table in Fig. 9.2), observe from Table 1.2 that there are three instances with this combination of attribute values in the weather data, and no instances with *humidity* = *high* and the same values for *play* and *temperature*. Initializing the counts for the two values of *humidity* to 0.5 yields the probability $(3 + 0.5)/(3 + 0 + 1) = 0.875$ for *humidity* = *normal*.

Let us consider more formally how to estimate the conditional and unconditional probabilities in a Bayesian network. The log-likelihood of a Bayesian network with V variables and N examples of complete assignments to the network is

$$\sum_{i=1}^{N} \log P(\{\tilde{A}_1, \tilde{A}_2, \ldots, \tilde{A}_V\}_i) = \sum_{i-1}^{N} \sum_{v=1}^{V} \log P(\tilde{A}_{v,i} \big| \text{Parents}(\tilde{A}_{v,i}); \Theta_v),$$

where the parameters of each conditional or unconditional distribution are given by Θ_v, and we use the $\sim$ to indicate actual observations of a variable. We find the maximum likelihood parameter values by taking derivatives. Since the log-likelihood is a double sum over examples i and variables v, when we take the derivative of it with respect to any given set of parameters Θ_v, all the terms of the sum that are independent of Θ_v will be zero. This means that the estimation problem *decouples* into the problem of estimating the parameters for each conditional or unconditional probability distribution separately. For variables with no parents, we need to estimate an unconditional probability. In Appendix A.2 we give a derivation illustrating why estimating a discrete distribution with parameters given by the probabilities π_k for k classes corresponds to the intuitive formula $\pi_k = n_k/N$, where n_k is the number of examples of class k and N is the total number of examples. This can also be written

$$P(A = a) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}(\tilde{A}_i = a),$$

where $\mathbf{1}(\tilde{A}_i = a)$ is simply an indicator function that returns 1 when the ith observed value for $\tilde{A}_i = a$ and 0 otherwise. The estimation of the entries of a

conditional probability table for $P(B|A)$ can be expressed using a notation similar to the intuitive counting procedures outlined above:

$$P(B = b|A = a) = \frac{P(B = b, A = a)}{P(A = a)} = \frac{\sum_{i=1}^{N} \mathbf{1}(\tilde{A}_i = a, \tilde{B}_i = b)}{\sum_{i=1}^{N} \mathbf{1}(\tilde{A}_i = a)}.$$

This derivation generalizes to the situation where A is a subset of random variables. Note that the above expressions give maximum likelihood estimates, and do not deal with the zero-frequency problem.

The nodes in the network are predetermined, one for each attribute (including the class). Learning the network structure amounts to searching through the space of possible sets of edges, estimating the conditional probability tables for each set, and computing the log-likelihood of the resulting network based on the data as a measure of the network's quality. Bayesian network learning algorithms differ mainly in the way in which they search through the space of network structures. Some algorithms are introduced below.

There is one caveat. If the log-likelihood is maximized based on the training data, it will always be better to add more edges: the resulting network will simply overfit. Various methods can be employed to combat this problem. One possibility is to use cross-validation to estimate the goodness of fit. A second is to add a penalty for the complexity of the network based on the number of parameters, i.e., the total number of independent estimates in all the probability tables. For each table, the number of independent probabilities is the total number of entries minus the number of entries in the last column, which can be determined from the other columns because all rows must sum to 1. Let K be the number of parameters, LL the log-likelihood, and N the number of instances in the data. Two popular measures for evaluating the quality of a network are the *Akaike Information Criterion* (AIC):

$$\text{AIC score} = -LL + K,$$

and the following *MDL metric* based on the MDL principle:

$$\text{MDL score} = -LL + \frac{K}{2}\log N.$$

In both cases the log-likelihood is negated, so the aim is to minimize these scores.

A third possibility is to assign a prior distribution over network structures and find the most likely network by combining its prior probability with the probability accorded to the network by the data. This is the maximum a posteriori approach to network scoring. Depending on the prior distribution used, it can take various forms. However, true Bayesians would average over all possible network structures rather than singling one particular network out for prediction. Unfortunately, this generally requires a great deal of computation. A simplified approach is to average over all network structures that are substructures of a given network. It turns out that this can be implemented very efficiently by changing

the method for calculating the conditional probability tables so that the resulting probability estimates implicitly contain information from all subnetworks. The details of this approach are rather complex and will not be described here.

The task of searching for a good network structure can be greatly simplified if the right metric is used for scoring. Recall that the probability of a single instance based on a network is the product of all the individual probabilities from the various conditional probability tables. The overall probability of the data set is the product of these products for all instances. Because terms in a product are interchangeable, the product can be rewritten to group together all factors relating to the same table. The same holds for the log-likelihood, using sums instead of products. This means that the likelihood can be optimized separately for each node of the network. This can be done by adding, or removing, edges from other nodes to the node that is being optimized—the only constraint is that cycles must not be introduced. The same trick also works if a local scoring metric such as AIC or MDL is used instead of plain log-likelihood because the penalty term splits into several components, one for each node, and each node can be optimized independently.

SPECIFIC ALGORITHMS

Now we move on to actual algorithms for learning Bayesian networks. One simple and very fast learning algorithm, called *K2*, starts with a given ordering of the attributes (i.e., nodes). Then it processes each node in turn and greedily considers adding edges from previously processed nodes to the current one. In each step it adds the edge that maximizes the network's score. When there is no further improvement, attention turns to the next node. As an additional mechanism for overfitting avoidance, the number of parents for each node can be restricted to a predefined maximum. Because only edges from previously processed nodes are considered and there is a fixed ordering, this procedure cannot introduce cycles. However, the result depends on the initial ordering, so it makes sense to run the algorithm several times with different random orderings.

The Naïve Bayes classifier is a network with an edge leading from the class attribute to each of the other attributes. When building networks for classification, it sometimes helps to use this network as a starting point for the search. This can be done in *K*2 by forcing the class variable to be the first one in the ordering and initializing the set of edges appropriately.

Another potentially helpful trick is to ensure that every attribute in the data is in the *Markov blanket* of the node that represents the class attribute. A node's Markov blanket includes all its parents, children, and children's parents. It can be shown that a node is conditionally independent of all other nodes given values for the nodes in its Markov blanket. Hence, if a node is absent from the class attribute's Markov blanket, its value is completely irrelevant to the classification. We show an example of a Bayesian network and its Markov blanket in Fig. 9.3. Conversely, if *K*2 finds a network that does not include a relevant attribute in the

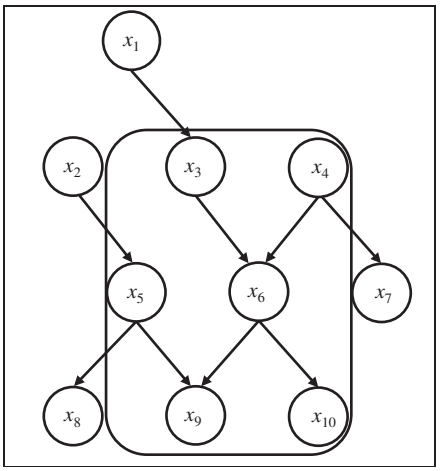

FIGURE 9.3

The Markov blanket for variable x_6 in a 10-variable Bayesian network.

class node's Markov blanket, it might help to add an edge that rectifies this shortcoming. A simple way of doing this is to add an edge from the attribute's node to the class node or from the class node to the attribute's node, depending on which option avoids a cycle.

A more sophisticated but slower version of *K*2 is not to order the nodes but to greedily consider adding or deleting edges between arbitrary pairs of nodes (all the while ensuring acyclicity, of course). A further step is to consider inverting the direction of existing edges as well. As with any greedy algorithm, the resulting network only represents a *local* maximum of the scoring function: it is always advisable to run such algorithms several times with different random initial configurations. More sophisticated optimization strategies such as simulated annealing, tabu search, or genetic algorithms can also be used.

Another good learning algorithm for Bayesian network classifiers is called *tree augmented Naïve Bayes* (TAN). As the name implies, it takes the Naïve Bayes classifier and adds edges to it. The class attribute is the single parent of each node of a Naïve Bayes network: TAN considers adding a second parent to each node. If the class node and all corresponding edges are excluded from consideration, and assuming that there is exactly one node to which a second parent is not added, the resulting classifier has a tree structure rooted at the parentless node—i.e., where the name comes from. For this restricted type of network there is an efficient algorithm for finding the set of edges that maximizes the network's likelihood based on computing the network's maximum weighted spanning tree. This algorithm's runtime is linear in the number of instances and quadratic in the number of attributes.

The type of network learned by the TAN algorithm is called a *one-dependence estimator*. An even simpler type of network is the *super-parent*

one-dependence estimator. Here, exactly one other node apart from the class node is elevated to parent status, and becomes parent of every other nonclass node. It turns out that a simple ensemble of these one-dependence estimators yields very accurate classifiers: in each of these estimators, a different attribute becomes the extra parent node. Then, at prediction time, class probability estimates from the different one-dependence estimators are simply averaged. This scheme is called AODE, for averaged one-dependence estimator. Normally, only estimators with a certain support in the data are used in the ensemble, but more sophisticated selection schemes are possible. Because no structure learning is involved for each super-parent one-dependence estimator, AODE is a very efficient classifier.

AODE makes strong assumptions, but relaxes the even stronger assumption of Naïve Bayes. The model can be relaxed even further by introducing a set of n super parents instead of a single super-parent attribute and averaging across all possible sets, yielding the AnDE algorithm. Increasing n obviously increases computational complexity. There is good empirical evidence that $n = 2$ (A2DE) yields a useful trade-off between computational complexity and predictive accuracy in practice.

All the scoring metrics that we have described so far are likelihood-based in the sense that they are designed to maximize the joint probability $P(a_1, a_2, \ldots, a_n)$ for each instance. However, in classification, what we really want to maximize is the conditional probability of the class given the values of the other attributes—in other words, the conditional likelihood. Unfortunately, there is no closed-form solution for the maximum *conditional*-likelihood probability estimates that are needed for the tables in a Bayesian network. On the other hand, computing the conditional likelihood for a given network and dataset is straightforward—after all, this is what logistic regression does. Hence it has been proposed to use standard maximum likelihood probability estimates in the network, but the conditional likelihood to evaluate a particular network structure.

Another way of using Bayesian networks for classification is to build a separate network for each class value, based on the data pertaining to that class, and combine their predictions using Bayes' rule. The set of networks is called a *Bayesian multinet*. To obtain a prediction for a particular class value, take the corresponding network's probability and multiply it by the class's prior probability. Do this for each class and normalize the result as we did previously. In this case we would not use the conditional likelihood to learn the network for each class value.

All the network learning algorithms we have introduced are score-based. A different strategy, which we will not explain here, is to piece a network together by testing individual conditional independence assertions based on subsets of the attributes. This is known as *structure learning by conditional independence tests*.

DATA STRUCTURES FOR FAST LEARNING

Learning Bayesian networks involves a lot of counting. For each network structure considered in the search, the data must be scanned afresh to obtain the counts needed to fill out the conditional probability tables. Instead, could they be stored

in a data structure that eliminated the need for scanning the data over and over again? An obvious way is to precompute the counts and store the nonzero ones in a table—say, the hash table mentioned in Section 4.5. Even so, any nontrivial data set will have a huge number of nonzero counts.

Again, consider the weather data from Table 1.2. There are five attributes, two with three values and three with two values. This gives $4 \times 4 \times 3 \times 3 \times 3 = 432$ possible counts. Each component of the product corresponds to an attribute, and its contribution to the product is one more than the number of its values because the attribute may be missing from the count. All these counts can be calculated by treating them as item sets, as explained in Section 4.5, and setting the minimum coverage to one. But even without storing counts that are zero, this simple scheme runs into memory problems very quickly. The FP-growth data structure described in Section 6.3 was designed for efficient representation of data in the case of item set mining. In the following, we describe a structure that has been used for Bayesian networks.

It turns out that the counts can be stored effectively in a structure called an *all-dimensions (AD) tree*, which is analogous to the *k*D-trees used for nearest neighbor search described in Section 4.7. For simplicity, we illustrate this using a reduced version of the weather data that only has the attributes *humidity*, *windy*, and *play*. Fig. 9.4A summarizes the data. The number of possible counts is

(A)

Humidity	Windy	Play	Count
High	True	Yes	1
High	True	No	2
High	False	Yes	2
High	False	No	2
Normal	True	Yes	2
Normal	True	No	1
Normal	False	Yes	4
Normal	False	No	0

(B)

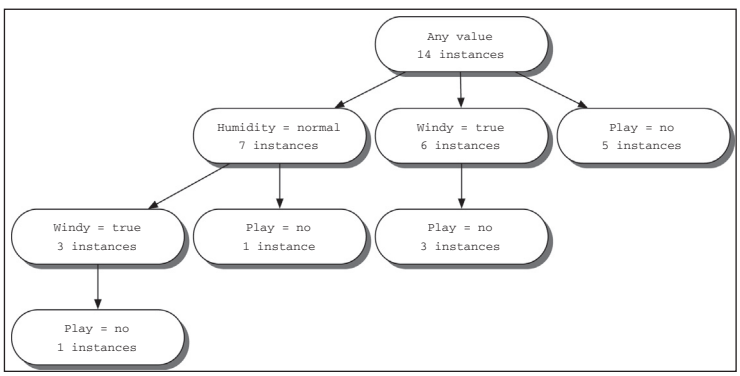

FIGURE 9.4

The weather data: (A) reduced version; (B) corresponding AD tree.

$3 \times 3 \times 3 = 27$, although only 8 of them are shown. For example, the count for *play* = *no* is 5 (count them!).

Fig. 9.4B shows an AD tree for this data. Each node says how many instances exhibit the attribute values that are tested along the path from the root to that node. For example, the leftmost leaf says that there is one instance with values *humidity* = *normal, windy* = *true*, and *play* = *no*, and the rightmost leaf says that there are five instances with *play* = *no*.

It would be trivial to construct a tree that enumerates all 27 counts explicitly. However, that would gain nothing over a plain table and is obviously not what the tree in Fig. 9.4B does, because it contains only 8 counts. There is, e.g., no branch that tests *humidity* = *high*. How was the tree constructed, and how can all counts be obtained from it?

Assume that each attribute in the data has been assigned an index. In the reduced version of the weather data we give *humidity* index 1, *windy* index 2, and *play* index 3. An AD tree is generated by expanding each node corresponding to an attribute i with the values of all attributes that have indices $j > i$, with two important restrictions: the most populous expansion for each attribute is omitted (breaking ties arbitrarily) as are expansions with counts that are zero. The root node is given index 0, so for it all attributes are expanded, subject to the same restrictions.

For example, Fig. 9.4B contains no expansion for *windy* = *false* from the root node because with eight instances it is the most populous expansion: the value *false* occurs more often in the data than the value *true*. Similarly, from the node labeled *humidity* = *normal* there is no expansion for *windy* = *false* because *false* is the most common value for *windy* among all instances with *humidity* = *normal*. In fact, in our example the second restriction—namely that expansions with zero counts are omitted—never kicks in because the first restriction precludes any path that starts with the tests *humidity* = *normal* and *windy* = *false*, which is the only way to reach the solitary zero in Fig. 9.4A.

Each node of the tree represents the occurrence of a particular combination of attribute values. It is straightforward to retrieve the count for a combination that occurs in the tree. However, the tree does not explicitly represent many nonzero counts because the most populous expansion for each attribute is omitted. For example, the combination *humidity* = *high* and *play* = *yes* occurs three times in the data but has no node in the tree. Nevertheless, it turns out that any count can be calculated from those that the tree stores explicitly.

Here's a simple example. Fig. 9.4B contains no node for *humidity* = *normal, windy* = *true,* and *play* = *yes*. However, it shows three instances with *humidity* = *normal* and *windy* = *true*, and one of them has a value for *play* that is different from *yes*. It follows that there must be two instances for *play* = *yes*. Now for a trickier case: how many times does *humidity* = *high, windy* = *true*, and *play* = *no* occur? At first glance it seems impossible to tell because there is no branch for *humidity* = *high*. However, we can deduce the number by calculating the count for *windy* = *true* and *play* = *no* (3) and subtracting the count for *humidity* = *normal, windy* = *true*, and *play* = *no* (1). This gives 2, the correct value.

This idea works for any subset of attributes and any combination of attribute values, but it may have to be applied recursively. For example, to obtain the count for *humidity* = *high*, *windy* = *false*, and *play* = *no*, we need the count for *windy* = *false* and *play* = *no* and the count for *humidity* = *normal*, *windy* = *false*, and *play* = *no*. We obtain the former by subtracting the count for *windy* = *true* and *play* = *no* (3) from the count for *play* = *no* (5), giving 2, and the latter by subtracting the count for *humidity* = *normal*, *windy* = *true*, and *play* = *no* (1) from the count for *humidity* = *normal* and *play* = *no* (1), giving 0. Thus there must be $2 - 0 = 2$ instances with *humidity* = *high*, *windy* = *false*, and *play* = *no*, which is correct.

AD trees only pay off if the data contains many thousands of instances. It is pretty obvious that they do not help on the weather data. The fact that they yield no benefit on small data sets means that, in practice, it makes little sense to expand the tree all the way down to the leaf nodes. Usually, a cutoff parameter k is employed, and nodes covering fewer than k instances hold a list of pointers to these instances rather than a list of pointers to other nodes. This makes the trees smaller and more efficient to use.

This section has only skimmed the surface of the subject of learning Bayesian networks. We left open questions of missing values, numeric attributes, and hidden attributes. We did not describe how to use Bayesian networks for regression tasks. Some of these topics are discussed later in this chapter. Bayesian networks are a special case of a wider class of statistical models called *graphical models*, which include networks with undirected edges (called *Markov networks*). Graphical models have attracted a lot of attention in the machine learning community and we will discuss them in Section 9.6.

9.3 CLUSTERING AND PROBABILITY DENSITY ESTIMATION

An incremental heuristic clustering approach was described in Section 4.8. While it works reasonably well in some practical situations, it has shortcomings: the arbitrary division by k in the category utility formula that is necessary to prevent overfitting, the need to supply an artificial minimum value for the standard deviation of clusters, and the ad hoc cutoff value to prevent every single instance from becoming a cluster in its own right. On top of this is the uncertainty inherent in incremental algorithms. To what extent is the result dependent on the order of examples? Are the local restructuring operations of merging and splitting really enough to reverse the effect of bad initial decisions caused by unlucky ordering? Does the final result represent even a *local* maximum of category utility? Add to this the problem that one never knows how far the final configuration is to a *global* maximum—and, of course, the standard trick of repeating the clustering procedure several times and choosing the best will destroy the incremental nature of the algorithm. Finally, does not the hierarchical nature of the result really beg the question of which are the *best* clusters? There are so many clusters in Fig. 4.21 that it is hard to separate the wheat from the chaff.

A more principled statistical approach to the clustering problem can overcome some of these shortcomings. From a probabilistic perspective, the goal of clustering is to find the most likely set of clusters given the data (and, inevitably, prior expectations). Because no finite amount of evidence is enough to make a completely firm decision on the matter, instances—even training instances—should not be placed categorically in one cluster or the other: instead they have a certain probability of belonging to each cluster. This helps to eliminate the brittleness that is often associated with schemes that make hard and fast judgments.

The foundation for statistical clustering is a statistical model called a *finite mixture* model. A *mixture* is a set of k probability distributions, representing k clusters, that govern the attribute values for members of that cluster. In other words, each distribution gives the probability that a particular instance would have a certain set of attribute values if it were *known* to be a member of that cluster. Each cluster has a different distribution. Any particular instance "really" belongs to one and only one of the clusters, but it is not known which one. Finally, the clusters are not equally likely: there is some probability distribution that reflects their relative populations.

THE EXPECTATION MAXIMIZATION ALGORITHM FOR A MIXTURE OF GAUSSIANS

One of the simplest finite mixture situations is when there is only one numeric attribute, which has a Gaussian or normal distribution for each cluster—but with different means and variances. The clustering problem is to take a set of instances—in this case each instance is just a number—and a prespecified number of clusters, and work out each cluster's mean and variance and the population distribution between the clusters. The mixture model combines several normal distributions, and its probability density function looks like a mountain range with a peak for each component.

Fig. 9.5 shows a simple example. There are two clusters A and B, and each has a normal distribution with means and standard deviations μ_A and σ_A for cluster A, and μ_B and σ_B for cluster B. Samples are taken from these distributions, using cluster A with probability p_A and cluster B with probability p_B (where $p_A + p_B = 1$), resulting in a data set like that shown. Now, imagine being given the data set without the classes—just the numbers—and being asked to determine the five parameters that characterize the model: μ_A, σ_A, μ_B, σ_B, and p_A (the parameter p_B can be calculated directly from p_A). That is the finite mixture problem.

If you knew which of the two distributions each instance came from, finding the five parameters would be easy—just estimate the mean and standard deviation for the cluster A samples and the cluster B samples separately, using the formulas

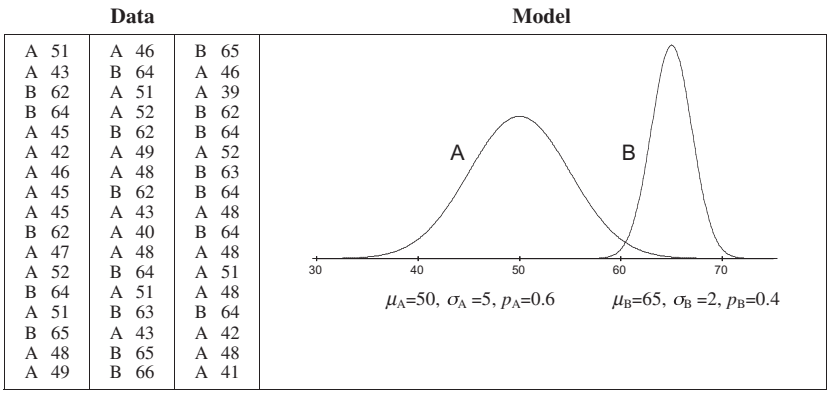

FIGURE 9.5

A two-class mixture model.

$$\mu = \frac{x_1 + x_2 + \cdots + x_n}{n},$$

$$\sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_n - \mu)^2}{n - 1}.$$

(The use of $n-1$ rather than n as the denominator in the second formula ensures an unbiased estimate of the variance, rather than the maximum likelihood estimate: it makes little difference in practice if n is used instead.) Here, x_1, x_2,. . ., x_n are the samples from the distribution A or B. To estimate the fifth parameter p_A, just take the proportion of the instances that are in the A cluster.

If you knew the five parameters, finding the (posterior) probabilities that a given instance comes from each distribution would be easy. Given an instance x_i, the probability that it belongs to cluster A is

$$P(A|x_i) = \frac{P(x_i|\mathrm{A}) \cdot P(A)}{P(x_i)} = \frac{N(x_i; \mu_\mathrm{A}, \sigma_\mathrm{A}) p_\mathrm{A}}{P(x_i)},$$

where $N(x; \mu_\mathrm{A}, \sigma_\mathrm{A})$ is the normal or Gaussian distribution function for cluster A, i.e.:

$$N(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

In practice we calculate the numerators for both $P(\mathrm{A}|x_i)$ and $P(\mathrm{B}|x_i)$, then normalize them by dividing by their sum, which is $P(x_i)$. This whole procedure is just the same as the way numeric attributes are treated in the Naïve Bayes learning scheme of Section 4.2. And the caveat explained there applies here too: strictly speaking, $N(x_i; \mu_\mathrm{A}, \sigma_\mathrm{A})$ is not the probability $\mathrm{P}(x|\mathrm{A})$ because the probability of x being any particular real number x_i is zero. Instead, $N(x_i; \mu_\mathrm{A}, \sigma_\mathrm{A})$ is a probability density, which is turned into a probability by the normalization

process used to compute the posterior. Note that the final outcome is not a particular cluster but rather the (posterior) *probability* with which x_i belongs to cluster A or cluster B.

The problem is that we know neither of these things: not the distribution that each training instance came from nor the five parameters of the mixture model. So we adopt the procedure used for the *k*-means clustering algorithm and iterate. Start with initial guesses for the five parameters, use them to calculate the cluster probabilities for each instance, use these probabilities to re-estimate the parameters, and repeat. (If you prefer, you can start with guesses for the classes of the instances instead.) This is an instance of the *expectation–maximization* or *EM* algorithm. The first step, calculation of the cluster probabilities (which are the "expected" class values) is the "expectation"; the second, calculation of the distribution parameters, is our "maximization" of the likelihood of the distributions given the data.

A slight adjustment must be made to the parameter estimation equations to account for the fact that it is only cluster probabilities, not the clusters themselves, that are known for each instance. These probabilities just act like weights. If w_i is the probability that instance i belongs to cluster A, the mean and standard deviation for cluster A are

$$\mu_A = \frac{w_1 x_1 + w_2 x_2 + \cdots + w_n x_n}{w_1 + w_2 + \cdots + w_n}$$

$$\sigma_A^2 = \frac{w_1 (x_1 - \mu)^2 + w_2 (x_2 - \mu)^2 + \cdots + w_n (x_n - \mu)^2}{w_1 + w_2 + \cdots + w_n}$$

where now the x_i are *all* the instances, not just those belonging to cluster A. (This differs in a small detail from the estimate for the standard deviation given above: if all weights are equal the denominator is n rather than $n-1$, which uses the maximum likelihood estimator rather than the unbiased estimator.)

Now consider how to terminate the iteration. The *k*-means algorithm stops when the classes of the instances do not change from one iteration to the next—a "fixed point" has been reached. In the EM algorithm things are not quite so easy: the algorithm converges toward a fixed point but never actually gets there. We can see how close it is getting by calculating the overall (marginal) *likelihood* that the data came from this model, given the values for the five parameters. The marginal likelihood is obtained by summing (or marginalizing) over the two components of the Gaussian mixture, i,e,

$$\prod_{i=1}^{n} P(x_i) = \prod_{i=1}^{n} \sum_{c_i} P(x_i | c_i) \cdot P(c_i)$$

$$= \prod_{i=1}^{n} (N(x_i; \mu_{\mathrm{A}}, \sigma_{\mathrm{A}}) p_{\mathrm{A}} + N(x_i; \mu_{\mathrm{B}}, \sigma_{\mathrm{B}}) p_{\mathrm{B}}).$$

This is the product of the marginal probability densities of the individual instances, which are obtained from the sum of the probability density under each

normal distribution $N(x; \mu, \sigma)$, weighted by the appropriate (prior) class probability. The cluster membership variable c is a so-called hidden (or latent) variable; we sum it out to obtain the marginal probability density of an instance.

This overall likelihood is a measure of the "goodness" of the clustering and increases at each iteration of the EM algorithm. The above equation and the expressions $N(x_i; \mu_A, \sigma_A)$ and $N(x_i; \mu_B, \sigma_B)$ are probability densities and not probabilities, so they do not necessarily lie between 0 and 1: nevertheless, the resulting magnitude still reflects the quality of the clustering. In practical implementations the log-likelihood is calculated instead: this is done by summing the logarithms of the individual components, avoiding multiplications. But the overall conclusion still holds: you should iterate until the increase in log-likelihood becomes negligible. For example, a practical implementation might iterate until the difference between successive values of log-likelihood is less than 10^{-10} for 10 successive iterations. Typically, the log-likelihood will increase very sharply over the first few iterations and then converge rather quickly to a point that is virtually stationary.

Although the EM algorithm is guaranteed to converge to a maximum, this is a *local* maximum and may not necessarily be the same as the global maximum. For a better chance of obtaining the global maximum, the whole procedure should be repeated several times, with different initial guesses for the parameter values. The overall log-likelihood figure can be used to compare the different final configurations obtained: just choose the largest of the local maxima.

EXTENDING THE MIXTURE MODEL

Now that we have seen the Gaussian mixture model for two distributions, let us consider how to extend it to more realistic situations. The basic method is just the same, but because the mathematical notation becomes formidable we will not develop it in full detail.

Changing the algorithm from two-cluster problems to situations with multiple clusters is completely straightforward, so long as the number k of normal distributions is given in advance.

The model can easily be extended from a single numeric attribute per instance to multiple attributes as long as independence between attributes is assumed. The probabilities for each attribute are multiplied together to obtain the joint probability (density) for the instance, just as in the Naïve Bayes method.

When the dataset is known in advance to contain correlated attributes, the independence assumption no longer holds. Instead, two attributes can be modeled jointly by a bivariate normal distribution, in which each has its own mean value but the two standard deviations are replaced by a "covariance matrix" with four numeric parameters. In Appendix A.2 we show the mathematics for the multivariate Gaussian distribution; the special case of a diagonal covariance model leads to a Naïve Bayesian interpretation. Several correlated attributes can be handled using a multivariate distribution. The number of parameters increases with the

square of the number of jointly varying attributes. With n independent attributes, there are $2n$ parameters, a mean and a standard deviation for each. With n covariant attributes, there are $n + n(n + 1)/2$ parameters, a mean for each, and an $n \times n$ covariance matrix that is symmetric and therefore involves $n(n + 1)/2$ different quantities. This escalation in the number of parameters has serious consequences for overfitting, as we will explain later.

To cater for nominal attributes, the normal distribution must be abandoned. Instead, a nominal attribute with v possible values is characterized by v numbers representing the probability of each one. A different set of numbers is needed for every cluster; kv parameters in all. The situation is very similar to the Naïve Bayes method. The two steps of expectation and maximization correspond exactly to operations we have studied before. Expectation—estimating the cluster to which each instance belongs given the distribution parameters—is just like determining the class of an unknown instance. Maximization—estimating the parameters from the classified instances—is just like determining the attribute–value probabilities from the training instances, with the small difference that in the EM algorithm instances are assigned to classes probabilistically rather than categorically. In Section 4.2 we encountered the problem that probability estimates can turn out to be zero, and the same problem occurs here too. Fortunately, the solution is just as simple—use the Laplace estimator.

Naïve Bayes assumes that attributes are independent—i.e., why it is called "naïve." A pair of correlated nominal attributes with v_1 and v_2 possible values, respectively, can be replaced by a single covariant attribute with $v_1 v_2$ possible values. Again, the number of parameters escalates as the number of dependent attributes increases, and this has implications for probability estimates and overfitting.

The presence of both numeric and nominal attributes in the data to be clustered presents no particular problem. Covariant numeric and nominal attributes are more difficult to handle, and we will not describe them here.

Missing values can be accommodated in various different ways. In principle, they should be treated as unknown and the EM process adapted to estimate them as well as the cluster means and variances. A simple way is to replace them by means or modes in a preprocessing step.

With all these enhancements, probabilistic clustering becomes quite sophisticated. The EM algorithm is used throughout to do the basic work. The user must specify the number of clusters to be sought, the type of each attribute (numeric or nominal), which attributes are to modeled as covarying, and what to do about missing values. Moreover, different distributions can be used. Although the normal distribution is usually a good choice for numeric attributes, it is not suitable for attributes (such as weight) that have a predetermined minimum (zero, in the case of weight) but no upper bound; in this case a "log-normal" distribution is more appropriate. Numeric attributes that are bounded above and below can be modeled by a "log-odds" distribution. Attributes that are integer counts rather than real values are best modeled by the "Poisson" distribution. A comprehensive system might allow these distributions to be specified individually for each

attribute. In each case, the distribution involves numeric parameters—probabilities of all possible values for discrete attributes and mean and standard deviation for continuous ones.

In this section we have been talking about clustering. But you may be thinking that these enhancements could be applied just as well to the Naïve Bayes algorithm too—and you could be right. A comprehensive probabilistic modeler could accommodate both clustering and classification learning, nominal and numeric attributes with a variety of distributions, various possibilities of covariation, and different ways of dealing with missing values. The user would specify, as part of the domain knowledge, which distributions to use for which attributes.

CLUSTERING USING PRIOR DISTRIBUTIONS

However, there is a snag: overfitting. You might say that if we are not sure which attributes are dependent on each other, why not be on the safe side and specify that *all* the attributes are covariant? The answer is that the more parameters there are, the greater the chance that the resulting structure is overfitted to the training data—and covariance increases the number of parameters dramatically. The problem of overfitting occurs throughout machine learning, and probabilistic clustering is no exception. There are two ways that it can occur: through specifying too large a number of clusters and through specifying distributions with too many parameters.

The extreme case of too many clusters occurs when there is one for every data point: clearly, that will be overfitted to the training data. In fact, in the mixture model, problems will occur whenever any one of the normal distributions becomes so narrow that it is centered on just one data point. Consequently, implementations generally insist that clusters contain at least two different data values.

Whenever there are a large number of parameters, the problem of overfitting arises. If you were unsure of which attributes were covariant, you might try out different possibilities and choose the one that maximized the overall probability of the data given the clustering that was found. Unfortunately, the more parameters there are, the larger the overall data probability will tend to be—not necessarily because of better clustering but because of overfitting. The more parameters there are to play with, the easier it is to find a clustering that seems good.

It would be nice if somehow you could penalize the model for introducing new parameters. One principled way of doing this is to adopt a Bayesian approach in which every parameter has a prior probability distribution whose effect is incorporated into the overall likelihood figure. In a sense, the Laplace estimator that we met in Section 4.2, and whose use we advocated earlier to counter the problem of zero probability estimates for nominal values, is just such a device. Whenever there are few observations, it exacts a penalty because it makes probabilities that are zero, or close to zero, greater, and this will decrease the overall likelihood of the data. In fact, the Laplace estimator is tantamount to using a particular prior distribution for the parameter concerned. Making two nominal attributes covariant will exacerbate the problem of sparse data. Instead of $v_1 + v_2$ parameters, where v_1 and

v_2 are the number of possible values, there are now $v_1 v_2$, greatly increasing the chance of a large number of small observed frequencies.

The same technique can be used to penalize the introduction of large numbers of clusters, just by using a prespecified prior distribution that decays sharply as the number of clusters increases.

AutoClass is a comprehensive Bayesian clustering scheme that uses the finite mixture model with prior distributions on all the parameters. It allows both numeric and nominal attributes, and uses the EM algorithm to estimate the parameters of the probability distributions to best fit the data. Because there is no guarantee that the EM algorithm converges to the global optimum, the procedure is repeated for several different sets of initial values. But that is not all. AutoClass considers different numbers of clusters and can consider different amounts of covariance and different underlying probability distribution types for the numeric attributes. This involves an additional, outer level of search. For example, it initially evaluates the log-likelihood for 2, 3, 5, 7, 10, 15, and 25 clusters: after that, it fits a log-normal distribution to the resulting data and randomly selects from it more values to try. As you might imagine, the overall algorithm is extremely computation intensive. In fact, the actual implementation starts with a prespecified time bound and continues to iterate as long as time allows. Give it longer and the results may be better!

A simpler way of selecting an appropriate model—e.g., to choose the number of clusters—is to compute the likelihood on a separate validation set that has not been used to fit the model. This can be repeated with multiple train-validation splits, just as in the case of classification models—e.g., with *k*-fold cross-validation. In practice, the ability to pick a model in this way is a big advantage of probabilistic clustering approaches compared to heuristic clustering methods.

Rather than showing just the most likely clustering to the user, it may be best to present all of them, weighted by probability. Recently, fully Bayesian techniques for *hierarchical* clustering have been developed that produce as output a probability distribution over possible hierarchical structures representing a dataset. Fig. 9.6 is a visualization, known as a *DensiTree*, that shows the set of all trees for a particular dataset in a triangular shape. The tree is best described in terms of its "clades," a biological term from the Greek *klados*, meaning *branch*, for a group of the same species that includes all ancestors. Here there are five clearly distinguishable clades. The first and fourth correspond to a single leaf, while the fifth has two leaves that are so distinct they might be considered clades in their own right. The second and third clades each have five leaves, and there is large uncertainty in their topology. Such visualizations make it easy for people to grasp the possible hierarchical clusterings of their data, at least in terms of the big picture.

CLUSTERING WITH CORRELATED ATTRIBUTES

Many clustering methods make the assumption of independence among the attributes. An exception is AutoClass, which does allow the user to specify in advance

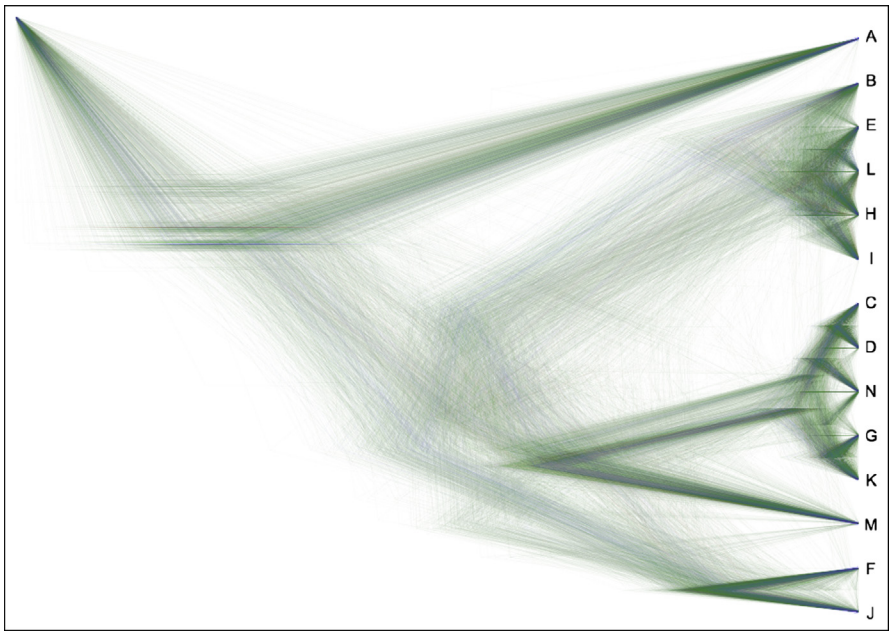

FIGURE 9.6

DensiTree showing possible hierarchical clusterings of a given data set.

that two or more attributes are dependent and should be modeled with a joint probability distribution. (There are restrictions, however: nominal attributes may vary jointly, as may numeric attributes, but not both together. Moreover, missing values for jointly varying attributes are not catered for.) It may be advantageous to preprocess a data set to make the attributes more independent, using statistical techniques such as the independent component transform described in Section 8.3. Note that joint variation that is specific to particular classes will not be removed by such techniques; they only remove overall joint variation that runs across all classes.

If all attributes are continuous, more advanced clustering methods can help capture joint variation on a per-cluster basis, without having the number of parameters explode when there are many dimensions. As discussed above, if each covariance matrix in a Gaussian mixture model is "full" we need to estimate $n(n+1)/2$ parameters per mixture component. However, as we will see in Section 9.6, principal component analysis can be formulated as a probabilistic model, yielding probabilistic principal component analysis (PPCA), and approaches known as *mixtures of principal component analyzers* or *mixtures of factor analyzers* provide ways of using a much smaller number of parameters to represent large covariance matrices. In fact, the problem of estimating $n(n+1)/2$ parameters in a *full* covariance matrix can be transformed into the problem of estimating as few as $n \times d$ parameters in a *factorized covariance matrix*, where d

can be chosen to be small. The idea is to decompose the covariance matrix $\mathbf{M}$ into the form $\mathbf{M} = (\mathbf{W}\mathbf{W}^{\mathrm{T}} + \mathbf{D})$, where $\mathbf{W}$ is typically a long and skinny matrix of size $n \times d$, with as many rows as there are dimensions n of the input, and as many columns d as there are dimensions in the reduced space. Standard PCA corresponds to setting $\mathbf{D} = 0$; PPCA corresponds to using the form $\mathbf{D} = \sigma^2\mathbf{I}$, where σ^2 is a scalar parameter and $\mathbf{I}$ is the identity matrix; and factor analysis corresponds to using a diagonal matrix for $\mathbf{D}$. The mixture model versions give each mixture component this type of factorization.

KERNEL DENSITY ESTIMATION

Mixture models can provide compact representations of probability distributions but do not necessarily fit the data well. In Chapter 4, Algorithms: the basic methods, we mentioned that when the form of a probability distribution is unknown, an approach known as *kernel density estimation* can be used to approximate the underlying distribution more accurately. This estimates the underlying true probability distribution $p(\mathbf{x})$ of data $\mathbf{x}_1$, $\mathbf{x}_2$,..., $\mathbf{x}_n$ using a kernel density *estimator*, which can be written in the following general form

$$\hat{p}(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} K_\sigma(\mathbf{x}, \mathbf{x}_i) = \frac{1}{n\sigma}\sum_{i=1}^{n} K\left[\frac{\mathbf{x} - \mathbf{x}_i}{\sigma}\right],$$

where $K()$ is a nonnegative kernel function that integrates to one. Here, we use the notation $\hat{p}(\mathbf{x})$ to emphasize that this is an estimation of the true (unknown) distribution $p(\mathbf{x})$. The parameter $\sigma > 0$ is the *bandwidth* of the kernel, and serves as a form of smoothing parameter for the approximation. When the kernel function is defined using σ as a subscript it is known as a "scaled" kernel function and is given by $K_\sigma(\mathbf{x}) = 1/\sigma\, K(\mathbf{x}/\sigma)$. Estimating densities using kernels is also known as *Parzen window density estimation*.

Popular kernel functions include the Gaussian, box, triangle, and Epanechnikov kernels. The Gaussian kernel is popular because of its simple and attractive mathematical form. The box kernel implements a windowing function, whereas the triangle kernel implements a smoother, but still conceptually simple, window. The Epanechnikov kernel can be shown to be optimal under a mean squared error metric. The bandwidth parameter affects the smoothness of the estimator and the quality of the estimate. There are several methods for coming up with an appropriate bandwidth, ranging from heuristics motivated by theoretical results on known distributions to empirical choices based on validation sets and cross-validation techniques. Many software packages offer the choice between simple heuristic default values, bandwidth selection through cross-validation methods, and the use of plug-in estimators derived from further analytical analysis.

Kernel density estimation is closely related to k-nearest neighbor density estimation, and it can be shown that both techniques converge to the true distribution $p(\mathbf{x})$ as the amount of data grows towards infinity. This result, combined with the

fact that they are easy to implement, makes kernel density estimators attractive methods in many situations.

Consider, e.g., the practical problem of finding outliers in data, given only positive or only negative examples (or perhaps with just a handful of examples of the other class). One effective approach is to do the best possible job of modeling the probability distribution of the data for the plentiful class using a kernel density estimator, and considering new data to which the model assigns low probability as outliers.

COMPARING PARAMETRIC, SEMIPARAMETRIC AND NONPARAMETRIC DENSITY MODELS FOR CLASSIFICATION

One might view a mixture model as intermediate between two extreme ways of modeling distributions by estimating probability densities. One extreme is a single simple parametric form such as the Gaussian distribution. It is easy to estimate the relevant parameters. However, data often arises from a far more complex distribution. Mixture models use two or more Gaussians to approximate the distribution. In the limit, at the other extreme, one Gaussian is used for each data point. This is kernel density estimation with a Gaussian kernel function.

Fig. 9.7 shows a visual example of this spectrum of models. A density estimate for each class of a 3-class classification problem has been created using three different techniques. Fig. 9.7A uses a single Gaussian distribution for each class, an approach that is often referred to as a "parametric" technique. Fig. 9.7B uses a Gaussian mixture model with two components per class, a "semiparametric" technique in which the number of Gaussians can be determined using a variety of methods. Fig. 9.7C uses a kernel density estimate with a Gaussian kernel on each example, a "nonparametric" method. Here the model complexity grows in proportion to the volume of data.

All three approaches define density models for each class, so Bayes' rule can be used to compute the posterior probability over all classes for any given input.

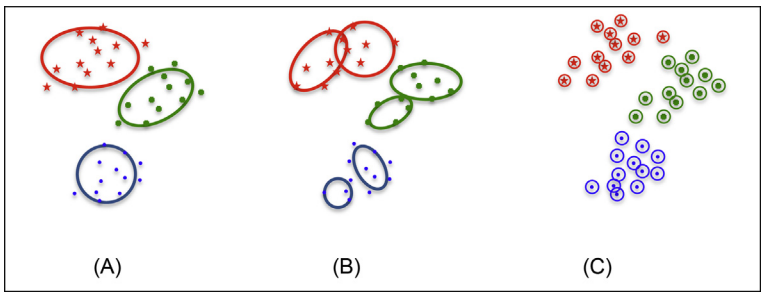

FIGURE 9.7

Probability contours for three types of model, all based on Gaussians.

In this way, density estimators can easily be transformed into classifiers. For simple parametric models, this is quick and easy. Kernel density estimators are guaranteed to converge to the true underlying distribution as the amount of data increases, which means that classifiers constructed from them have attractive properties. They share the computational disadvantages of nearest-neighbor classification, but, just as for nearest-neighbor classification, fast data structures exist that can make them applicable to large datasets.

Mixture models, the intermediate option, give control of model complexity without it growing with the amount of data. For this reason this approach has been standard practice for initial modeling in fields such as in speech recognition, which deal in large datasets. It allows speech recognizers to be created by first clustering data into groups, but in such a way that more complex models of temporal relationships can be added later using a hidden Markov model. (We will consider sequential and temporal probabilistic models, such as hidden Markov models, in Section 9.6.)

9.4 HIDDEN VARIABLE MODELS

We now move on to advanced learning algorithms that can infer new attributes in addition to the ones present in the data—so-called hidden (or latent) variables whose values cannot be observed. As noted in Section 9.3, a quantity called the *marginal likelihood* can be obtained by summing (or integrating) these variables out of the model. It is important not to confuse random variables with observations or hard assignments of random variables. We write $p(x_i = \tilde{x}_i) = p(\tilde{x}_i)$ to denote the probability of the random variable x_i associated with instance i taking the value represented by the observation $\tilde{x}_i$. We use h_i to represent a hidden discrete random variable, and z_i to denote a hidden continuous one. Then, given a model with observations given by $\tilde{x}_i$, the marginal likelihood is

$$L(\theta; \tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n) = \prod_{i=1}^{n} p(\tilde{x}_i; \theta) = \prod_{i=1}^{n} \int_{z_i} \sum_{h_i} p(\tilde{x}_i, z_i, h_i; \theta) dz_i,$$

where the sum is taken over all possible discrete values of h_i and the integral is taken over the entire domain of z_i. The end result of all this integrating and summing is a single number—a scalar quantity—that gives the marginal likelihood for any value of the parameters.

Maximum likelihood based learning for hidden variable models can sometimes be done using marginal likelihood, just as when no hidden variables are present, but the additional variables usually affect the parameterization used to define the model. In fact, these additional variables are often very important: *they are used to represent precisely the things we wish to mine from our data*, be it clusters, topics in a text mining problem, or the factors that underlie variation in the data. By treating parameters as random variables and using functions that are easy to manipulate, marginal likelihoods can also be used to define sophisticated

Bayesian models that involve integrating over parameters. This can create models that are less prone to overfitting.

EXPECTED LOG-LIKELIHOODS AND EXPECTED GRADIENTS

It is not always possible to obtain a form for the marginal likelihood that is easy to optimize. An alternative is to work with another quantity, the *expected log-likelihood*. Writing the set of all observed data as $\tilde{X}$, the set of all discrete hidden variables as H, and the set of all continuous hidden variables as Z, the expected log-likelihood can be expressed as

$$\begin{aligned} E[\log L(\theta; \tilde{X}, Z, H)]_{P(H,Z|\tilde{X};\theta)} &= \sum_{i=1}^{n} \left[\int_{z_i} \sum_{h_i} p(z_i, h_i|\tilde{x}_i; \theta) \log p(\tilde{x}_i, z_i, h_i; \theta) dz_i \right] \\ &= E\left[\sum_{i=1}^{n} \log p(\tilde{x}_i, z_i, h_i; \theta) \right]_{p(z_i,h_i|\tilde{x}_i;\theta)}. \end{aligned}$$

Here, $E[.]_{p(z_i,h_i|\tilde{x}_i;\theta)}$ means that an expectation is performed under the posterior distribution over hidden variables: $p(z_i, h_i|\tilde{x}_i; \theta)$.

It turns out that there is a close relationship between the log marginal likelihood and the expected log-likelihood: the *derivative* of the expected log-likelihood with respect to the parameters of the model equals the *derivative* of the log marginal likelihood. The following derivation, based on applying the chain rule from calculus and considering a single training example for simplicity, demonstrates why this is true:

$$\begin{aligned} \frac{\partial}{\partial\theta} \log p(\tilde{x}_i; \theta) &= \frac{1}{p(\tilde{x}_i; \theta)} \frac{\partial}{\partial\theta} \int_{z_i} \sum_{h_i} p(\tilde{x}_i, z_i, h_i; \theta) dz_i \\ &= \int_{z_i} \sum_{h_i} \frac{p(\tilde{x}_i, z_i, h_i; \theta)}{p(\tilde{x}_i; \theta)} \frac{\partial}{\partial\theta} \log p(\tilde{x}_i, z_i, h_i; \theta) dz_i \\ &= \int_{z_i} \sum_{h_i} p(z_i, h_i|\tilde{x}_i; \theta) \frac{\partial}{\partial\theta} \log p(\tilde{x}_i, z_i, h_i; \theta) dz_i \\ &= E\left[\frac{\partial}{\partial\theta} \log p(\tilde{x}_i, z_i, h_i; \theta) \right]_{p(z_i,h_i|\tilde{x}_i;\theta)} \end{aligned}$$

The final expression is the expectation of the derivative of the log joint likelihood. This relationship is with respect to the derivative of the log marginal likelihood and the expected derivative of the log joint likelihood. However, it is also possible to establish a direct relationship between the log marginal and expected log joint probabilities. The variational analysis in Appendix A.2 shows that

$$\log P(\tilde{x}_i; \theta) = E[\log p(\tilde{x}_i, z_i, h_i; \theta)]_{p(z_i,h_i|\tilde{x}_i;\theta)} + H[p(z_i, h_i|\tilde{x}_i; \theta)],$$

where $H[.]$ is the entropy.

As a consequence of this analysis, to perform learning in a probability model with hidden variables, the marginal likelihood can be optimized using gradient ascent by instead computing and following the gradient of the expected log-likelihood, assuming that the posterior distribution over hidden variables can be computed. This prompts a general approach to learning in a hidden variable model based on following the expected gradient. This can be decomposed into three steps: (1) a **P-step**, which computes the posterior over hidden variables; (2) an **E-step**, which computes the expectation of the gradient given the posterior; and (3) a **G-step**, which uses gradient-based optimization to maximize the objective function with respect to the parameters.

THE EXPECTATION MAXIMIZATION ALGORITHM

Using the expected log joint probability as a key quantity for learning in a probability model with hidden variables is better known in the context of the celebrated "expectation maximization" or EM algorithm, which we encountered in "The expectation maximization algorithm for a mixture of Gaussians" section. We discuss the general EM formulation next. Section 9.6 gives a concrete example comparing and contrasting the expected gradient approach and the EM approach, using the probabilistic formulation of principal component analysis.

The EM algorithm follows the expected gradient approach. However, EM is often used with models in which the M-step can be computed *in closed form*—in other words, when exact parameter updates can be found by setting the derivative of the expected log-likelihood with respect to the parameters to 0. These updates often take the same form as the simple maximum likelihood estimates that one would use to compute the parameters of a distribution, and are essentially just modified forms of the equations used for observed data that involve averages weighted over the posterior distribution in place of observed counts.

The EM algorithm consists of two steps: (1) an **E-step** that computes the expectations used in the expected log-likelihood and (2) an **M-step** in which the objective is maximized—typically using a closed-form parameter update.

In the following, we assume that we have only discrete hidden variables $\mathbf{H}$. The probability of the observed data $\tilde{X}$ can be maximized by maximizing the log-likelihood $\log P(\tilde{X};\theta)$ of the parameters θ arising from an underlying latent variable model $P(X,H;\theta)$ as follows. Initialize the parameters as θ^{old} and repeat the following steps, where convergence is measured in terms of either the change to the log-likelihood or the degree of change to the parameters:

1. **E-step**: Compute required expectations involving $P(H|X;\theta^{\text{old}})$
2. **M-step**: Find $\theta^{\text{new}} = \arg\max_\theta \left[\sum_H P(H|X;\theta^{\text{old}})\log P(X,H;\theta)\right]$
3. If the algorithm has not converged, set $\theta^{\text{old}} = \theta^{\text{new}}$ and return to step 1.

Note that the *M*-step corresponds to maximizing the expected log-likelihood. Although discrete hidden variables are used above, the approach generalizes to continuous ones.

For many latent variable models—Gaussian mixture models, PPCA, and hidden Markov models—the required posterior distributions can be computed exactly, which accounts for their popularity. However, for many other probabilistic models it is simply not possible to compute an exact posterior distribution. This can easily happen with multiple hidden random variables, because the posterior needed in the E-step is the joint posterior of the hidden variables. There is a vast literature on the subject of how to compute approximations to the true posterior distribution over hidden variables in more complex models.

APPLYING THE EXPECTATION MAXIMIZATION ALGORITHM TO BAYESIAN NETWORKS

Bayesian networks capture statistical dependencies between attributes using an intuitive graphical structure, and the EM algorithm can easily be applied to such networks. Consider a Bayesian network with a number of discrete random variables, some of which are observed while others are not. Its marginal probability, in which hidden variables have been integrated out, can be maximized by maximizing the expected log joint probability over the posterior distribution of the hidden variables given the observed data—the expected log-likelihood.

For a network consisting of only discrete variables, this means that the **E-step** involves computing a distribution over hidden variables $\{H\}$ given observed variables $\{\tilde{X}\}$ or $P(\{H\}|\{\tilde{X}\};\theta^{\text{current}})$. If the network is a tree, this can be computed efficiently using the sum-product algorithm, which is explained in Section 9.6. If not, it can be computed efficiently using the junction tree algorithm. However, if the model is large, exact inference algorithms may be intractable, in which case a variational approximation or sampling procedure can be used to approximate the distribution.

The **M-step** seeks

$$\theta^{\text{new}} = \arg\max_{\theta} \left[\sum_{\{H\}} P(\{H\}|\{\tilde{X}\};\theta^{\text{old}}) \log P(\{\tilde{X}\},\{H\};\theta) \right].$$

Recall that the log joint probability given by a Bayesian network decomposes into a sum over functions of subsets of variables. Notice also that the expression above involves an expectation using the joint conditional distribution or posterior over hidden variables. Using the EM algorithm, taking the derivative with respect to any given parameter leaves just terms that involve the marginal expectation over the distribution of variables that participate in the function for the gradient of the relevant parameter. This means, e.g., that to find the unconditional probability of an unobserved variable A in a network, it is necessary to determine the parameters θ_A of $P(A;\theta_A)$ for which

$$\frac{\partial}{\partial \theta_A} \left[\sum_{A} P(A|\{\tilde{X}\};\theta^{old}) \log P(A;\theta_A) \right] = 0,$$

along with the further constraint that the probabilities in the discrete distribution sum to 1. This can be achieved using a Lagrange multiplier (Appendix A.2 gives an example of using this technique to estimate a discrete distribution). Setting the derivative of the constrained objective to 0 gives this *closed form* result:

$$\theta^{\text{new}}_{A=a} = P(A = a) = \frac{1}{N}\sum_{i=1}^{N} P(A_i = a|\{\tilde{X}\}_i; \theta^{\text{old}}).$$

In other words, the unconditional probability distribution is estimated in the same way in which it would be computed if the variables A_i had been observed, but with each observation replaced by its probability. Applying this procedure to the entire data set is tantamount to replacing observed counts with expected counts under the current model settings. If many examples have the same configuration, the distribution need only be computed once, and multiplied by the number of times that configuration has been seen.

Estimating entries in the network's conditional probability tables also has an intuitive form. To estimate the conditional probability of unobserved random variable B given unobserved random variable A in a Bayesian network, simply compute their joint (posterior) probability and the marginal (posterior) probability of A for each example. Just as when the data is observed, the update equation is

$$P(B = b|A = a) = \frac{\sum_{i=1}^{N} P(A_i = a, B_i = b|\{\tilde{X}\}_i; \theta^{\text{old}})}{\sum_{i=1}^{N} P(A_i = a|\{\tilde{X}\}_i; \theta^{\text{old}})}.$$

This is just a ratio of the expected numbers of counts. If some of the variables are fully observed, the expression can be adapted by replacing the inferred probabilities by observed values—effectively assigning the observations a probability of 1. Furthermore, if variable B has multiple parents, A can be replaced by the set of parents.

9.5 BAYESIAN ESTIMATION AND PREDICTION

If there is reason to believe that a certain parameter has been drawn from a particular distribution, we can adopt a more Bayesian perspective. A common strategy is to employ a hyperparameter α to represent that distribution. Define the *joint distribution* of data and parameters as

$$p(x_1, x_2, \ldots, x_n, \theta; \alpha) = \prod_{i=1}^{n} p(x_i|\theta)p(\theta; \alpha)$$

Bayesian-style predictions use a quantity known as the *posterior predictive distribution*, which consists of the probability model for a new observation marginalized over the posterior probability inferred for the parameters given the observations so far. Again using a notation that explicitly differentiates variables x_i from their observations $\tilde{x}_i$, the posterior predictive distribution is

$$p(x_{\text{new}}|\tilde{x}_1,\ \tilde{x}_2,\ldots,\tilde{x}_n;\alpha) = \int_\theta p(x_{\text{new}}|\theta)p(\theta|\tilde{x}_1,\ \tilde{x}_2,\ldots,\tilde{x}_n;\alpha)d\ \theta.$$

Given a Bayesian model that uses distributions over parameters, so-called "empirical Bayesian" methods can be employed to find a suitable value for the hyperparameter α. One such approach is obtained by maximizing the log marginal likelihood with respect to the model's hyperparameters:

$$\alpha_{\text{MML}} = \arg\max_\alpha \left[\log \int \prod_{i=1}^{n} p(x_i|\theta)p(\theta;\alpha)d\theta\right].$$

The remainder of this section demonstrates several techniques for creating complex structured probabilistic models.

PROBABILISTIC INFERENCE METHODS

With complex probability models—and even with some seemingly simple ones—computing quantities such as posterior distributions, marginal distributions, and the maximum probability configuration, often require specialized methods to achieve results efficiently—even approximate ones. This is the field of *probabilistic inference*. Below we review some widely used probabilistic inference methods, including probability propagation, sampling and simulated annealing, and variational inference.

Probability propagation

Structured probability models like the Bayesian networks and Markov random fields discussed in Section 9.6 decompose a joint probability distribution into a factorized structure consisting of products of functions over subsets of variables. Then the task of computing marginal probabilities to find a maximum-probability configuration can be computationally demanding using brute force computation. In some cases even a naïve approach is completely infeasible in practice. However, it is sometimes possible to take advantage of the model's structure to perform inference more efficiently. When Bayesian networks and related graphical models have an underlying tree connectivity structure, then *belief propagation* (also known as *probability propagation*) based on the sum-product and max-product algorithms presented in Section 9.6 can be applied to compute exact marginals, and hence the most probable model configuration.

Sampling, simulated annealing, and iterated conditional modes

With fully Bayesian methods that use distributions on parameters, or graphical models with cyclic structures, sampling methods are popular in both statistics and machine learning. Markov chain Monte Carlo methods are widely used to generate random samples from probability distributions that are difficult to compute. For example, as we have seen above, posterior distributions are often needed for expectations required during learning, but in many settings these can be difficult

to compute. *Gibbs sampling* is a popular special case of the more general Metropolis–Hastings algorithm that allows one to generate samples from a joint distribution even when the true distribution is a complex continuous function. These samples can then be used to approximate expectations of interest, and also to approximate marginal distributions of subsets of variables by simply ignoring parts pertaining to other variables.

Gibbs sampling is conceptually very simple. Assign an initial set of states to the random variables of interest. With n random variables, this initial assignment or set of samples can be written $x_1 = x_1^{(0)}, \ldots, x_n = x_n^{(0)}$. We then iteratively update each variable by *sampling* from its conditional distribution given the others:

$$\begin{aligned} x_1^{(i+1)} &\sim p(x_1 | x_2 = x_2^{(i)}, \ldots, x_n = x_n^{(i)}), \\ &\vdots \\ x_n^{(i+1)} &\sim p(x_n | x_1 = x_1^{(i)}, \ldots, x_{n-1} = x_{n-1}^{(i)}). \end{aligned}$$

In practice, these conditional distributions are often easy to compute. Furthermore, the idea of a "Markov blanket" introduced in Section 9.2 can often be used to reduce the number of variables necessary, because conditionals in structured models may depend on a much smaller subset of the variables.

To ensure an unbiased sample it is necessary to cycle through the data discarding samples in a process known as "burn-in." The idea is to allow the Markov chain defined by the sampling procedure to approach its stationary distribution, and it can be shown that in the limit one will indeed obtain a sample from this distribution, and that the distribution corresponds to the underlying joint probability we wish to sample from. There is considerable theory concerning how much burn-in is required, but in practice it is common to discard samples arising from the first 100–1000 cycles. Sometimes, if more than one sample configuration is desired, averages are taken over k additional configurations of the sampler obtained after periods of about 100 cycles. We see how this procedure is used in practice in Section 9.6, under latent Dirichlet allocation.

Simulated annealing is a procedure that seeks an approximate most probable configuration or explanation. It adapts the Gibbs sampling procedure, described above, to include an iteration-dependent "temperature" term t_i. Starting with an initial assignment $x_1 = x_1^{(0)}, \ldots, x_n = x_n^{(0)}$, subsequent samples take the form

$$\begin{aligned} x_1^{(i+1)} &\sim p(x_1 | x_2 = x_2^{(i)}, \ldots, x_n = x_n^{(i)})^{\frac{1}{t_i}}, \\ &\vdots \\ x_n^{(i+1)} &\sim p(x_n | x_1 = x_1^{(i)}, \ldots, x_{n-1} = x_{n-1}^{(i)})^{\frac{1}{t_i}}, \end{aligned}$$

Where the temperature is decreased at each iteration: $t_{i+1} < t_i$. If the schedule is slow enough, this process will converge to the true global minimum. But therein lies the catch: the temperature may have to decrease very slowly. However, this is often possible, particularly with an efficient implementation of the sampler.

Another well-known algorithm is the *iterated conditional modes* procedure, consisting of iterations of the form

$$x_1^{(i+1)} \sim \arg\max_{x_1} p(x_1 | x_2 = x_2^{(i)}, \ldots, x_n = x_n^{(i)}),$$
$$\vdots$$
$$x_n^{(i+1)} \sim \arg\max_{x_n} p(x_n | x_1 = x_1^{(i)}, \ldots, x_{n-1} = x_{n-1}^{(i)}).$$

This can be very fast, but prone to local minima. It can be useful when constructing more interesting graphical models and optimizing them quickly in an analogous, greedy way.

Variational inference

Rather than sampling from a distribution that is difficult to manipulate, the distribution can be approximated by a simpler, more tractable, function. Suppose we have a probability model with a set H of hidden variables and a set X of observed variables. Let $p = p(H|\tilde{X}; \theta)$ be the exact posterior distribution of the model and $q = q(H|\tilde{X}; \Phi)$ be an approximation to it, where Φ is a set of so-called "variational parameters." Variational methods for probability models commonly involve defining a simple form for q that makes it easy to optimize Φ in a way that brings q closer to p. The theory of variation EM optimizes latent variable models by maximizing a lower bound on the log marginal likelihood. This so-called "variational bound" is described in Appendix A.2, where we see how the EM algorithm can be viewed through a variational analysis. This allows one to create EM algorithms using either exact or approximate posterior distributions. While statisticians often prefer sampling methods to variational ones, variational methods are popular in machine learning because they can be faster—and can also be combined with sampling methods.

9.6 GRAPHICAL MODELS AND FACTOR GRAPHS

Bayesian networks give an intuitive picture of a probabilistic model that corresponds directly with how we have decided to decompose the joint probability of the random variables representing attributes into a product of conditional and unconditional probability distributions. Mixture models, such as the Gaussian mixture models of Section 9.3, are alternative ways of approximating joint distributions. This section shows how such models can be illustrated using Bayesian networks, and introduces a generalization of Bayesian networks, the so-called "plate notation," that allows one to visualize the result of techniques that treat parameters as random quantities. A further generalization, "factor graphs," can represent and visualize an even wider class of probabilistic graphical models. As before, we view attributes as random variables and instances as observations of them; we also represent things like the label of a cluster by a random variable in a graph.

GRAPHICAL MODELS AND PLATE NOTATION

Consider a simple two-cluster Gaussian mixture model. It can be illustrated in the form of a Bayesian network with a binary random variable C for the cluster membership and a continuous random variable x for the real-valued attribute. In the mixture model, the joint distribution $P(C, x)$ is the product of the prior $P(C)$ and the conditional probability distribution $P(x|C)$. This structure is illustrated by the Bayesian network in Fig. 9.8A, where for each state of C a different Gaussian is used for the conditional distribution of the continuous variable x.

Multiple Bayesian networks can be used to visualize the underlying joint likelihood that results when parameter estimation is performed. The probability model for N observations $x_1 = x_1$, $x_2 = x_2$, and $x_N = x_N$ can be conceptualized as N Bayesian networks, one for each variable x_i observed or instantiated to the value x_i. Fig. 9.8B illustrates this, using shaded nodes to indicate which random variables are observed.

A "plate" is simply a box around a Bayesian network that denotes a certain number of replications of it. The plate in Fig. 9.8C indicates $i = 1,\ldots,N$ networks, each with an observed value for x_i. Plate notation captures a model for the joint probability of the entire data with a simple picture.

Bayesian networks, and more complex models comprising plates of Bayesian networks, are known as *generative models* because the probabilistic definition of the model can be used to randomly generate data governed by the probability distribution that the model represents. Bayesian hierarchical modeling involves defining a hierarchy of levels for the *parameters* of a model and using the rules of probability arising from the application of Bayesian methods to infer the parameter values given observed data. These can be drawn with graphical models in which both random variables and parameters are treated as random quantities. The section on latent Dirichlet allocation below gives an example of this technique.

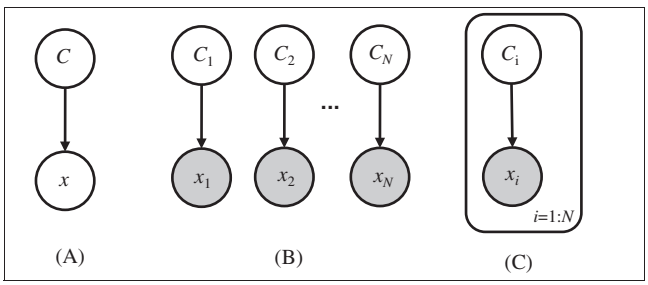

FIGURE 9.8

(A) Bayesian network for a mixture model; (B) multiple copies of the Bayesian network, one for each observation; (C) plate notation version of (B).

PROBABILISTIC PRINCIPAL COMPONENT ANALYSIS

Principal component analysis can be viewed as the consequence of performing parameter estimation in a special type of linear Gaussian hidden variable model. This connects the traditional view of this standard technique, presented in Chapter 8, Data transformations, with the probabilistic formulation discussed here, and leads to more advanced methods based on Boltzmann machines and autoencoders that are introduced in Chapter 10, Deep learning. The probabilistic formulation also helps deal with missing values. The underlying idea is to represent data as being generated by a Gaussian distributed continuous hidden latent variable that is linearly transformed under a Gaussian model. As a result, the principal components of a given data set correspond to an underlying factorized covariance model for a corresponding multivariate Gaussian distribution model for the data. This factorized model becomes apparent when the hidden variables of the underlying latent variable model are integrated out.

More specifically, a set of hidden variables is used to represent the input data in a space of reduced dimensionality. Each dimension corresponds to an independent random variable drawn from a Gaussian distribution with zero mean and unit variance. Let $\mathbf{x}$ be a random variable corresponding to the d-dimensional vectors of observed data, and $\mathbf{h}$ be a k-dimensional vector of hidden random variables. k is typically smaller than d (but need not be). Then the underlying joint probability model has this linear Gaussian form:

$$\begin{aligned} p(\mathbf{x}, \mathbf{h}) &= p(\mathbf{x}|\mathbf{h})p(\mathbf{h}) \\ &= N(\mathbf{x}; \mathbf{W}\mathbf{h} + \boldsymbol{\mu}, \mathbf{D})N(\mathbf{h}; \mathbf{0}, \mathbf{I}), \end{aligned}$$

where the zero vector $\mathbf{0}$ and identity matrix $\mathbf{I}$ denote the mean and covariance matrix for the Gaussian distribution used for $p(\mathbf{h})$, and $p(\mathbf{x}|\mathbf{h})$ is Gaussian with mean $\mathbf{W}\mathbf{h} + \boldsymbol{\mu}$ and a diagonal covariance matrix $\mathbf{D}$. (The expression for the mean is where the term "linear" in "linear Gaussian" comes from.) The mean $\boldsymbol{\mu}$ is included as a parameter, but it would be zero if we first mean-centered the data. Fig. 9.9A shows a Bayesian network for PPCA; it reveals intuitions about the

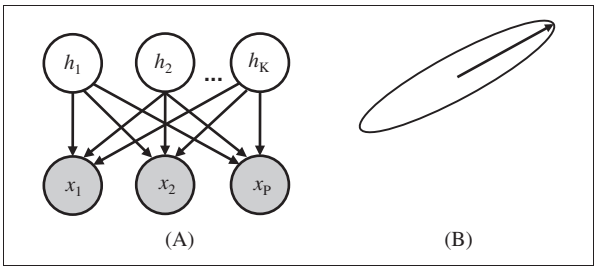

FIGURE 9.9

(A) Bayesian network for probabilistic PCA; (B) equal-probability contour for a Gaussian distribution along with its covariance matrix's principal eigenvector.

probabilistic interpretation of principal component analysis based on hidden variables that will help in understanding other models discussed later.

Probabilistic PCA is a form of generative model, and Fig. 9.9A visualizes the associated underlying generative process. Data is generated by sampling each dimension of $\mathbf{h}$ from an independent Gaussian distribution and using the matrix $\mathbf{Wh} + \boldsymbol{\mu}$ to project this lower dimensional representation of the data into the observed, higher dimensional representation. Noise, specified by the diagonal covariance matrix $\mathbf{D}$, is added separately to each dimension of the higher dimensional representation.

If the noise associated with the conditional distribution of $\mathbf{x}$ given $\mathbf{h}$ is the same in each dimension (i.e., isotropic) and infinitesimally small (such that $\mathbf{D} = \lim_{\sigma^2 \to 0} \sigma^2\mathbf{I}$), a set of estimation equations can be derived that give the same principal components as those obtained by conventional principal component analysis. Restricting the covariance matrix $\mathbf{D}$ to be diagonal produces a model known as *factor analysis*). If $\mathbf{D}$ is isotropic (i.e., has the form $\mathbf{D} = \sigma^2\mathbf{I}$), then after optimizing the model and learning σ^2 the columns of $\mathbf{W}$ will be scaled and rotated principal eigenvectors of the covariance matrix of the data. The contours of equal probability for a multivariate Gaussian distribution can be drawn with an ellipse whose principal axis corresponds to the principal eigenvector of the covariance matrix, as illustrated in Fig. 9.9B.

Because of the nice properties of Gaussian distributions, the marginal distributions of $\mathbf{x}$ in these models are also Gaussian, with parameters that can be computed *analytically* using simple algebraic expressions. For example, a model with $\mathbf{D} = \sigma^2\mathbf{I}$ has $p(\mathbf{x}) = N(\mathbf{x}; \boldsymbol{\mu}, \mathbf{WW^T} + \sigma^2\mathbf{I})$, which is the marginal distribution of $\mathbf{x}$ under the joint probability model, obtained by integrating out the uncertainty associated with $\mathbf{h}$ from the joint distribution $p(\mathbf{x},\mathbf{h})$ defined earlier. Integrating out the hidden variables $\mathbf{h}$ from a PPCA model defines a Gaussian distribution whose covariance matrix $\mathbf{M}$ has the special form $\mathbf{WW}^{\mathrm{T}} + \mathbf{D}$. Appendix A.2 relates this factorization to an eigenvector analysis of the covariance matrix, resulting in the standard matrix factorization view of principal component analysis.

Inference with PPCA

As a result of the underlying linear Gaussian formulation, various other quantities needed for making inferences and performing parameter estimation can also be obtained analytically. For example, given $\mathbf{M} = (\mathbf{WW}^T + \sigma^2\mathbf{I})$, the *posterior distribution* for $\mathbf{h}$ can be obtained from Bayes' rule, along with some Gaussian identities given in Appendix A.2. The posterior can be written

$$p(\mathbf{h}|\mathbf{x}) = N(\mathbf{h}; \mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu}), \sigma^2\mathbf{M}^{-1}). \tag{9.1}$$

Once the model parameters have been estimated, the mean of the posterior for a new example can be calculated, which can then serve as a reduced dimensional representation for it. The mathematics (though still perhaps daunting) is greatly simplified by the fact that marginalizing, multiplying and dividing

Gaussians distributions produces other Gaussian distributions that are functions of observed values, mean vectors, and covariance matrices. In fact, many more sophisticated methods and models for data analysis rely on the ease with which key quantities can be computed when the underlying model is based on linear Gaussian forms.

Marginal log-likelihood for PPCA

Given a probabilistic model with hidden variables, the Bayesian philosophy is to integrate out the uncertainly associated with them. As we have just seen, linear Gaussian models make it possible to obtain the marginal probability of the data, $p(\mathbf{x})$, which takes the form of a Gaussian distribution. Then, the learning problem is tantamount to maximizing the probability of the given data under the model. The probability that variable $\mathbf{x}$ is observed to be $\tilde{\mathbf{x}}$ is indicated by $p(\tilde{\mathbf{x}}) = p(\mathbf{x} = \tilde{\mathbf{x}})$. The log (marginal) likelihood of the parameters given all the observed data $\tilde{\mathbf{X}}$ can be maximized using this objective function:

$$L(\tilde{\mathbf{X}}; \theta) = \log\left[\prod_{i=1}^{N} P(\tilde{\mathbf{x}}_i; \theta)\right] = \sum_{i=1}^{N} \log[N(\tilde{\mathbf{x}}_i; \boldsymbol{\mu},\ \mathbf{W}\mathbf{W}^{\mathbf{T}} + \sigma^2\mathbf{I})],$$

where the parameters $\theta = \{\mathbf{W}, \boldsymbol{\mu}, \sigma^2\}$, consist of a matrix, a vector, and a scalar. We will assume henceforth that the data has been mean-centered: $\boldsymbol{\mu} = \mathbf{0}$. (However, when creating generalizations of this approach there can be advantages to retaining the mean as an explicit parameter of the model.)

Expected log-likelihood for PPCA

Section 9.4 showed that the derivative of the log marginal likelihood of a hidden variable model with respect to its parameters equals the derivative of the expected log joint likelihood, where the expectation is taken with respect to the exact posterior distribution over the model's hidden variables. The underlying Gaussian formulation of PPCA means that an exact posterior can be computed, which provides an alternative way to optimize the model based on the expected log-likelihood. The log joint probability of all data and all hidden variables $\mathbf{H}$ under the model is

$$L(\tilde{\mathbf{X}}, \mathbf{H}; \theta) = \log\left[\prod_{i=1}^{N} p(\tilde{\mathbf{x}}_i, \mathbf{h}_i; \theta)\right] = \sum_{i=1}^{N} \log[p(\tilde{\mathbf{x}}_i|\mathbf{h}_i; \mathbf{W})p(\mathbf{h}_i; \sigma^2)].$$

Notice that while the data $\tilde{\mathbf{X}}$ is observed, the hidden variables $\mathbf{h}_i$ are unknown, so for a given parameter value $\theta = \tilde{\theta}$ this expression does not evaluate to a scalar quantity. It can be converted into a scalar-valued function using the expected log-likelihood, which can then be optimized. As we saw in Section 9.4, the expected log-likelihood of the data using the posterior distribution of each example for the expectations is given by

$$E[L(\tilde{\mathbf{X}}, \mathbf{H}; \theta)]_{p(\mathbf{H}|\tilde{\mathbf{X}})} = \sum_{i=1}^{N} E[\log[p(\tilde{\mathbf{x}}_i, \mathbf{h}_i; \theta)]]_{p(\mathbf{h}_i|\tilde{\mathbf{x}}_i)}.$$

Expected gradient for PPCA

Gradient descent can be used to learn the parameter matrix $\mathbf{W}$ using the expected log-likelihood as the objective, an example of the expected gradient approach discussed in Section 9.4. The gradient is a sum over examples, and a fairly lengthy derivation shows that each example contributes the following term to this sum:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{W}} E[L(\tilde{\mathbf{x}}, \mathbf{h})] &= E[\mathbf{W}\mathbf{h}\mathbf{h}^{\mathrm{T}}] - E[\tilde{\mathbf{x}}\mathbf{h}^{T}] \\ &= \mathbf{W}E[\mathbf{h}\mathbf{h}^{\mathrm{T}}] - \tilde{\mathbf{x}}E[\mathbf{h}]^{T},\end{aligned} \tag{9.2}$$

where in all cases the expectations are taken with respect to the posterior, $p(\mathbf{h}|\tilde{\mathbf{x}})$, using the current settings of the model parameters. This partial derivative has a natural interpretation as a difference between two expectations. The second term creates a matrix the same size as $\mathbf{W}$, consisting of an observation and the hidden variable representation. The first term simply replaces the observation with a similar $\mathbf{Wh}$ factor, which could be thought of as the model's prediction of the input. (We will revisit this interpretation when we examine conditional probability models in Section 9.7 and restricted Boltzmann machines in Section 10.5.)

This shows that the model reconstructs the data as it ascends the gradient. If the optimization converges to a model that perfectly reconstructs the data, the derivative in (9.2) will be zero. Other probabilistic models examined show similar forms for the gradients for key parameters, consisting of differences of expectations—but different types of expectation are involved.

To compute these quantities, note that the expected value $E[\mathbf{h}] = \mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu})$ can be obtained from the mean of the posterior distribution for each example given by (9.1). The $E[\mathbf{h}\mathbf{h}^{\mathrm{T}}]$ term can be found using the fact that $E[\mathbf{h}\mathbf{h}^{\mathrm{T}}] = \mathrm{cov}[\mathbf{h}] + E[\mathbf{h}]E[\mathbf{h}]^{\mathrm{T}}$, where $\mathrm{cov}[\mathbf{h}]$ is the posterior covariance matrix, which we also know from (9.1) is $\sigma^2\mathbf{M}^{-1}$.

EM for PPCA

An alternative to gradient ascent using the expected log-likelihood is to formulate the model as an expected gradient-based learning procedure using the classical EM algorithm. The M-step update can be obtained by setting the derivative in (9.2) to zero and solving for $\mathbf{W}$. This can be expressed in closed form: $\mathbf{W}$ in the first term is independent of the expectation and can therefore be moved out of the expectation and terms re-arranged into a closed-form M-step. In a zero mean model with $\mathbf{D} = \sigma^2\mathbf{I}$ and some small value for σ^2, the E and M steps of the PPCA EM algorithm can be rewritten

$$\textbf{E-Step: } \mathrm{E}[\mathbf{h}_i] = \mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}\tilde{\mathbf{x}}_i,\ \mathrm{E}[\mathbf{h}_i\mathbf{h}_i^{T}] = \sigma^2\mathbf{M}^{-1} + \mathrm{E}[\mathbf{h}_i]\mathrm{E}[\mathbf{h}_i^{T}],$$

$$\textbf{M-Step: } \mathbf{W}^{\mathrm{New}} = \left[\sum_{i=1}^{N} \tilde{\mathbf{x}}_i\, \mathrm{E}[\mathbf{h}_i]^{\mathbf{T}}\right]\left[\sum_{i=1}^{N} \mathrm{E}[\mathbf{h}_i\mathbf{h}_i{}^{\mathrm{T}}]\right]^{-1},$$

where all expectations are taken with respect to each example's posterior distribution, $p(\mathbf{h}_i|\tilde{\mathbf{x}}_i)$, and, as above, $\mathbf{M} = (\mathbf{W}\mathbf{W}^{T} + \sigma^2\mathbf{I})$.

The EM algorithm can be further simplified by taking the limit as σ^2 approaches zero. This is the *zero input noise* case, and it implies $\mathbf{M} = \mathbf{W}\mathbf{W}^T$. Define the matrix $\mathbf{Z} = E[\mathbf{H}]$, each column of which contains the expected vector from $P(\mathbf{h}_i|\check{\mathbf{x}}_i)$ for one of the hidden variables $\mathbf{h}_i$, so that $E[\mathbf{H}\mathbf{H}^T] = E[\mathbf{H}]E[\mathbf{H}^T] = \mathbf{Z}\mathbf{Z}^T$. This yields simple and elegant equations for the E and M steps:

E-Step: $\mathbf{Z} = E[\mathbf{H}] = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\tilde{\mathbf{X}}$,
M-Step: $\mathbf{W}^{\text{New}} = \tilde{\mathbf{X}}\mathbf{Z}^T[\mathbf{Z}\mathbf{Z}^T]^{-1}$.

where both equations operate on the entire data matrix $\tilde{\mathbf{X}}$.

The probabilistic formulation of principal component analysis permits traditional likelihoods to be defined, which supports maximum likelihood based learning and probabilistic inference. This in turn leads to natural methods for dealing with missing data. It also leads on to the other models discussed below.

LATENT SEMANTIC ANALYSIS

Chapter 8, Data transformations, introduced principal component analysis, which, as we have seen above, can be viewed as a form of linear Gaussian latent variable model. We now discuss an influential early form of data driven document analysis known as "latent semantic analysis" (LSA), which uses singular value decomposition to decompose each document in a collection into *topics*. If documents and terms are projected into the topic space, comparisons can be made that capture semantic structure in the documents resulting from the cooccurrence of words across the collection. Characterizing documents in this way is called "latent semantic indexing." Probabilistic LSA (pLSA) addresses similar goals but applies a statistical model based on multinomial distributions. Latent Dirichlet allocation is a related model that uses a hierarchical Bayesian approach in which Dirichlet priors are placed on the underlying multinomial distributions.

To motivate the introduction of these techniques, let us examine the relationship between the original LSA method and singular value decomposition. Imagine a term by document matrix $\mathbf{X}$, with t rows and d columns, each element of which contains the number of times the word associated with its row occurs in the document associated with its column. LSA decomposes $\mathbf{X}$ into a product $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T$, where $\mathbf{U}$ and $\mathbf{V}$ have orthogonal columns and $\mathbf{S}$ is a diagonal matrix containing the singular values, which are conventionally sorted in decreasing order. This factorization is known as the singular value decomposition, and has the property that for every value k, if all but the k largest singular values are discarded the data matrix can be reconstructed in a way that is optimal in a least squares sense. For any given approximation level k, we can write $\mathbf{X} \approx \tilde{\mathbf{X}} = \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T$.

Fig. 9.10 illustrates how this works. The $\mathbf{U}_k$ matrix can be thought of as k orthogonal "topics" that are combined according to the appropriate proportions for each document, encoded in the $k \times d$ matrix $\mathbf{V}_k^T$. The matrix $\mathbf{A} = \mathbf{S}_k\mathbf{V}_k^T$ represents

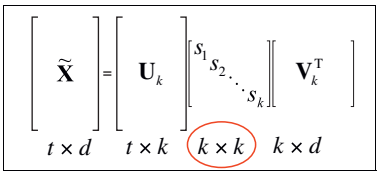

FIGURE 9.10

The singular value decomposition of a t by d matrix.

the activity level of the topics associated with each document. Thus the learning phase of LSA simply performs singular value decomposition on the data matrix.

The dot (or scalar) product of any two columns of the approximated data matrix $\tilde{\mathbf{X}}$ provides a measure of the similarity of term usage in the two documents. The dot products between all pairs of documents used to compute the singular value decomposition are

$$\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} = \mathbf{V}_k\mathbf{S}_k^2\mathbf{V}_k^T.$$

To analyze a new document (or query) $\mathbf{x}_q$ that is not in the original collection used for the decomposition, it can be projected into the semantic space of topic activity defined by the model using

$$\mathbf{a}_q = \mathbf{S}_k^{-1}\mathbf{U}_k^T\mathbf{x}_q. \tag{9.3}$$

USING PRINCIPAL COMPONENT ANALYSIS FOR DIMENSIONALITY REDUCTION

The singular value decomposition is widely used to project data into a space of reduced dimensions, often before applying other analysis techniques. For instance, data can be projected into a lower dimensional space in order to effectively apply nearest neighbor techniques, which tend to break down in high dimensional spaces.

LSA is in fact a form of principal component analysis. When viewed as a probability model, the projection of a document into a lower dimensional semantic space is, in effect, a *latent variable representation for the document*. The equivalent quantity in PPCA is given by the expected value of the posterior distribution over the hidden variables. This gives an intuitive view of what it means to project a document or query vector into a lower dimensional space.

For a PPCA model with no noise in the observed variables $\mathbf{x}$, we have seen that an input vector $\mathbf{x}$ can be projected into a reduced dimensional random vector $\mathbf{z}$ by computing the mean value of the posterior for the latent variables using $\mathbf{z} = E[\mathbf{h}] = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^{\mathrm{T}}\mathbf{x}$. Appendix A.2 details the relationships between PCA, PPCA, the singular value decomposition and eigendecompositions, and shows

that for mean-centered data the $\mathbf{U}$ matrix in (9.3) equals the matrix of eigenvectors $\mathbf{\Phi}$ of the eigendecomposition of the corresponding covariance matrix of the data, i.e. $\mathbf{U} = \mathbf{\Phi}$. The diagonal matrix $\mathbf{S}$ in (9.3) is related to the diagonal matrix of eigenvalues $\mathbf{\Lambda}$ by $\mathbf{S} = \mathbf{\Lambda}^{\frac{1}{2}}$. Furthermore, the eigendecomposition of the covariance matrix implies that $\mathbf{W} = \mathbf{\Phi}\mathbf{\Lambda}^{\frac{1}{2}}$ in the corresponding PPCA. Thus, under the probabilistic interpretation of principal component analysis with no noise for observed variables,$\mathbf{W} = \mathbf{US}$. Using the fact that $\mathbf{U}$ is orthogonal and $\mathbf{S}$ is diagonal, this means that computing a principal component analysis of mean-centered data using the singular value decomposition and interpreting the result as a linear Gaussian hidden variable model yields the following expression for projecting results into the reduced dimensional space based on the mean of the posterior:

$$\begin{aligned}\mathbf{z} &= (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{x} \\ &= (\mathbf{SU}^T\mathbf{US})^{-1}\mathbf{SU}^T\mathbf{x} \\ &= (\mathbf{S}^2)^{-1}\mathbf{SU}^T\mathbf{x} \\ &= \mathbf{S}^{-1}\mathbf{U}^T\mathbf{x}.\end{aligned}$$

This is the same expression as the LSA projection in (9.3) to compute $\mathbf{a}_q$, the semantic representation of a document, and it represents a general expression for dimensionality reduction using principal component analysis. This means that principal component analysis can be performed using the singular value decomposition of the data matrix, or an eigendecomposition of the covariance matrix, the EM algorithm, or even expected gradient descent. When working with large datasets or data with missing values there are advantages and disadvantages to each method.

The input data $\mathbf{X}$ need not arise from documents: LSA is just a concrete example of applying singular value decomposition to a real problem. Indeed, the general idea of computing such projections is widely used across machine learning and data mining. Because of the relationships discussed above the methods are often discussed using different terminology, even when they refer to the same underlying analysis.

PROBABILISTIC LSA

PPCA is based on continuous valued representations of data and an underlying Gaussian model. In contrast, the pLSA approach, also known as an "aspect model," is based on a formulation using the multinomial distribution; it was originally applied to the cooccurrences of words and documents. The multinomial distribution is a natural distribution for modeling word occurrence counts. In the pLSA framework one considers the index of each document as being encoded using observations of discrete random variables d_i for $i = 1,\ldots,n$ documents. Each variable d_i has n states, and over the document corpus there is one observation of the variable for each state. Topics are represented with discrete variables z_{ij}, while words are represented with random variables w_{ij}, where m_i words are associated

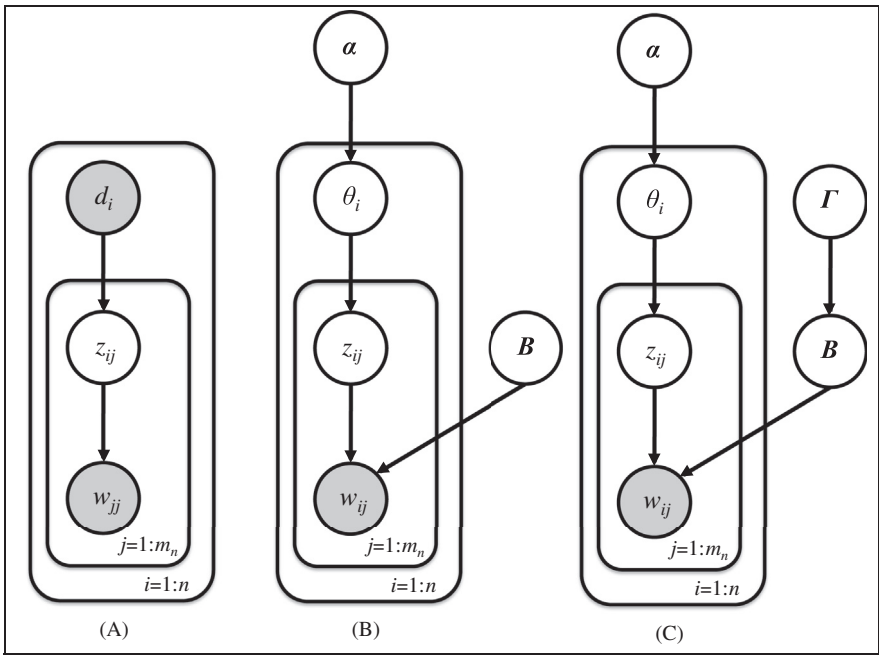

FIGURE 9.11

Graphical models for (A) pLSA, (B) LDA^b, and (C) smoothed LDA^b.

with each document and each word is associated with a topic. There are two variants: an asymmetric and a symmetric formulation.

Fig. 9.11A illustrates the asymmetric formulation; the symmetric formulation reverses the arrow from d to z. D is a set of random variables for the document index observations, and W is a set of random variables for all the words observed across the documents. The asymmetric formulation corresponds to

$$P(W,D) = \prod_{i=1}^{n} P(d_i) \prod_{j=1}^{m_n} \sum_{z_{ij}} P(z_{ij}|d_i) P(w_{ij}|z_{ij}).$$

Because d, an index over the training documents, is a random variable in the graph, pLSA is not a generative model for new documents. However, it does correspond to a valid probabilistic model with hidden variables, so the EM algorithm can be used to estimate parameters and obtain a representation of each document in a corpus in terms of its distribution over the topic variables.

LATENT DIRICHLET ALLOCATION

pLSA can be extended into a hierarchical Bayesian model with three levels, known as *latent Dirichlet allocation*. We refer to this as LDA^b ("b" for Bayesian)

to distinguish it from linear discriminant analysis which is commonly referred to as LDA. LDA[b] was proposed in part to reduce the overfitting that has been observed with pLSA, and has been extended in many ways. Extensions of LDA[b] can be used to determine trends over time and identify "hot" and "cold" topics. Such analyses are particularly interesting today, with the recent explosion of social media and the interest in analyzing it.

Latent Dirichlet allocation is a hierarchical Bayesian model that reformulates pLSA by replacing the document index variables d_i with the random parameter $\boldsymbol{\theta}_i$, a vector of multinomial parameters for the documents. The distribution of $\boldsymbol{\theta}_i$ is influenced by a Dirichlet prior with hyperparameter $\boldsymbol{\alpha}$, which is also a vector. (Appendix A.2 explains Dirichlet distributions and their use as priors for the parameters of the discrete distribution.) Finally, the relationship between the discrete topic variables z_{ij} and the words w_{ij} is also given an explicit dependence on a hyperparameter, namely, the matrix $\mathbf{B}$. Fig. 9.11B shows the corresponding graphical model. The probability model for the set of all observed words W is

$$P(\mathrm{W}|\boldsymbol{\alpha},\mathbf{B}) = \prod_{i=1}^{n}\int P(\boldsymbol{\theta}_i|\boldsymbol{\alpha})\left[\prod_{j=1}^{m_n}\sum_{z_{ij}} P(z_{ij}|\boldsymbol{\theta}_i)P(w_{ij}|z_{ij},\mathbf{B})\right]d\boldsymbol{\theta}_i$$
$$= \prod_{i=1}^{n}\int P(\boldsymbol{\theta}_i|\boldsymbol{\alpha})\left[\prod_{j=1}^{m_n} P(w_{ij}|\boldsymbol{\theta}_i,\mathbf{B})\right]d\boldsymbol{\theta}_i,$$

which marginalizes out the uncertainty associated with each $\boldsymbol{\theta}_i$ and z_{ij}. $P(\theta_i|\alpha)$ is given by a k-dimensional Dirichlet distribution, which also leads to k-dimensional topic variables z_{ij}. For a vocabulary of size V, $P(w_{ij}|z_{ij},\mathbf{B})$ encodes the probability of each word given each topic, and prior information is therefore captured by the $k \times V$ dimensional matrix $\mathbf{B}$.

The marginal log-likelihood of the model can be optimized using an empirical Bayesian method by adjusting the hyperparameters $\boldsymbol{\alpha}$ and $\mathbf{B}$ via the variational EM procedure. To perform the E-step of EM, the posterior distribution over the unobserved random quantities is needed. For the model defined by the above equation, with a random $\boldsymbol{\theta}$ for each document, word observations $\mathbf{w}$, and hidden topic variables $\mathbf{z}$, the posterior distribution is

$$P(\boldsymbol{\theta},z|\mathbf{w},\boldsymbol{\alpha},\mathbf{B}) = \frac{P(\boldsymbol{\theta},z,\mathbf{w}|\boldsymbol{\alpha},\mathbf{B})}{P(\mathbf{w}|\boldsymbol{\alpha},\mathbf{B})}$$

which, unfortunately, is intractable. For the M-step it is necessary to update hyperparameters $\boldsymbol{\alpha}$ and $\mathbf{B}$, which can be done by computing the maximum likelihood estimates using the expected sufficient statistics from the E-step. The variational EM procedure amounts to computing and using a separate approximate posterior for each $\boldsymbol{\theta}_i$ and each z_{ij}.

A method called "collapsed Gibbs sampling" turns out to be a particularly effective alternative to variational methods for performing LDA[b]. Consider first that the model in Fig. 9.11B can be expanded to that shown in Fig. 9.11C,

Table 9.1 Highest Probability Words and User Tags From a Sample of Topics Extracted From a Collection of Scientific Articles

Topic 2	Topic 39	Topic 102	Topic 201	Topic 210
Species	Theory	Tumor	Resistance	Synaptic
Global	Time	Cancer	Resistant	Neurons
Climate	Space	Tumors	Drug	Postsynaptic
Co_2	Given	Human	Drugs	Hippocampal
Water	Problem	Cells	Sensitive	Synapses
Geophysics, geology, ecology	Physics, math, applied math	Medical sciences	Pharmacology	Neurobiology

which was originally cast as the smoothed version of LDA[b]. Then add another Dirichlet prior with parameters given by $\boldsymbol{\Gamma}$ on the topic parameters of $\mathbf{B}$, a formulation that further reduces the effects of overfitting. Standard Gibbs sampling involves iteratively sampling the hidden random variables z_{ij}, the $\boldsymbol{\theta}_i$'s and the elements of matrix $\mathbf{B}$. Collapsed Gibbs sampling is obtained by integrating out the $\boldsymbol{\theta}_i$'s and $\mathbf{B}$ analytically, which deals with these distributions exactly. Consequently, conditioned by the current estimates of $\boldsymbol{\Gamma}$, $\boldsymbol{\alpha}$, and the observed words of a document corpus, the Gibbs sampler proceeds by simply iteratively updating each z_{ij} to compute the required approximate posterior. Using either samples or variational approximations it is then relatively straightforward to obtain estimates for the $\boldsymbol{\theta}_i$'s and $\mathbf{B}$.

The overall approach of using a smoothed, collapsed LDA[b] model to extract topics from a document collection can be summarized as follows: first define a hierarchical Bayesian model for the joint distribution of documents and words following the structure of Fig. 9.11C. We could think of there being a Bayesian **E-step** that performs approximate inference using Gibbs' sampling to sample from the *joint posterior* over *all* topics for all documents in the model, or $P(z_{ij}|w_{ij}, \boldsymbol{\Gamma}, \boldsymbol{\alpha})$, where the $\boldsymbol{\theta}_i$'s and $\mathbf{B}$ have been integrated out. This is followed by an **M-step** that uses these samples to update the estimates of the $\boldsymbol{\theta}_i$'s and $\mathbf{B}$, using update equations that are functions of $\boldsymbol{\Gamma}$, $\boldsymbol{\alpha}$, and the samples. This procedure is performed within a hierarchical Bayesian model, so the updated parameters can be used to create a Bayesian predictive distribution over new words and new topics given the observed words.

Table 9.1 shows the highest probability words from a sampling of topics mined by Griffiths and Steyvers (2004) through applying LDA[b] to 28,154 abstracts of papers published in the *Proceedings of the National Academy of Science* from 1991 to 2001 and tagged by authors with subcategory information. Analyzing the distribution over these tags identifies the highest probability user tags for each topic, which are shown at the bottom of Table 9.1. Note that the user tags were not used to create the topics, but we can see how well the extracted topics match with human labels.

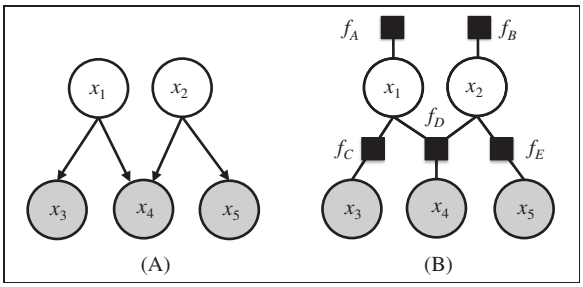

FIGURE 9.12

(A) Bayesian network and (B) corresponding factor graph.

FACTOR GRAPHS

Bayesian networks are a special kind of probability model that factorize a joint probability distribution into the product of conditional and unconditional distributions. Factor graphs provide an even more general framework for representing general functions by factoring them into the product of local functions, each of which acts on a subset of the full argument set:

$$F(x_1, \ldots, x_n) = \prod_{j=1}^{S} f_j(X_j),$$

where X_j is a subset of the original set of arguments $\{x_1,\ldots,x_n\}$, $f_j(X_j)$ is a function of X_j, and $j = 1,\ldots,S$ enumerates the argument subsets. A *factor graph* consists of variable nodes—circles—for each variable x_k and factor nodes—rectangles—for each function, with edges that connect each factor node to its variables.

Fig. 9.12A and B shows a Bayesian network and its factor graph corresponding to the factorization

$$\begin{aligned} F(x_1, \ldots, x_5) &= f_A(x_1) f_B(x_2) f_C(x_3, x_1) f_D(x_4, x_1, x_2) f_E(x_5, x_2) \\ &= P(x_1)P(x_2)P(x_3|x_1)P(x_4|x_1, x_2)P(x_5|x_2). \end{aligned}$$

Factor graphs make concepts such as the Markov blanket for a given variable in a Bayesian network easy to identify. For example, Fig. 9.13 shows the Markov blanket for variable x_6 in a factor graph that corresponds to the Bayesian network in Fig. 9.3: it consists of all nodes that are connected to it through a factor. Factor graphs are more powerful than Bayesian networks because they can represent a wider class of factorizations and models. These include Markov random fields, which we will meet shortly.

Factor graphs, Bayesian networks, and the logistic regression model

It is instructive to compare the factor graph for a naïvely constructed Bayesian model with the factor graph for a Naïve Bayes model of the same set of variables (and, later, with the factor graph for a logistic regression formulation of the same

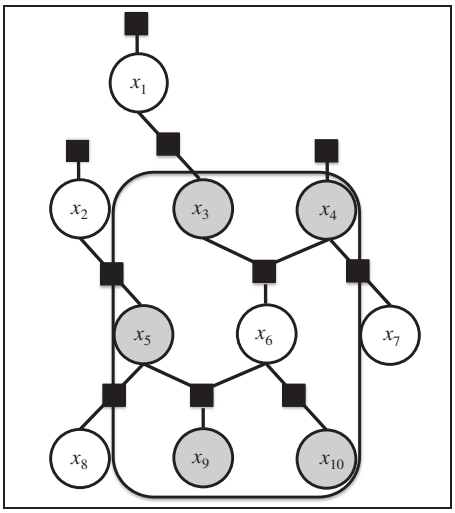

FIGURE 9.13

The Markov blanket for variable x_6 in a 10-variable factor graph.

problem). Fig. 9.14A and B shows the Bayesian network and its factor graph for a network with a child node y that has several parents x_i, $i = 1,\ldots,n$. Fig. 9.14B involves a large conditional probability table for $P(y|x_1,\ldots,x_n)$, with many parameters that must be estimated or specified, because in

$$P(y,x_1,\ldots,x_n) = P(y|x_1,\ldots,x_n)\prod_{i-1}^{n} P(x_i)$$

the number of parameters increases exponentially with the number of parent variables. In contrast, Fig. 9.14C and D shows the Bayesian network and its factor graph for the Naïve Bayes model. Here the number of parameters is linear in the number of children because the model breaks down into the product of functions involving y and just one x_i, because the underlying factorization is

$$P(y,x_1,\ldots,x_n) = P(y)\prod_{i-1}^{n} P(x_i|y).$$

The factor graphs show the different complexities very clearly. The graph in Fig. 9.14B has a factor involving $n+1$ variables, while the factors in Fig. 9.14D involve no more than two variables.

Factor graphs can be extended to clarify an important distinction for conditional models. The Bayesian network of Fig. 9.15A involves a large table for the conditional distribution of y given many x_i's, but a logistic regression model could be used to reduce the number of parameters for $P(y|x_1,\ldots,x_n)$ from exponential to linear, depicted in Fig. 9.15B.

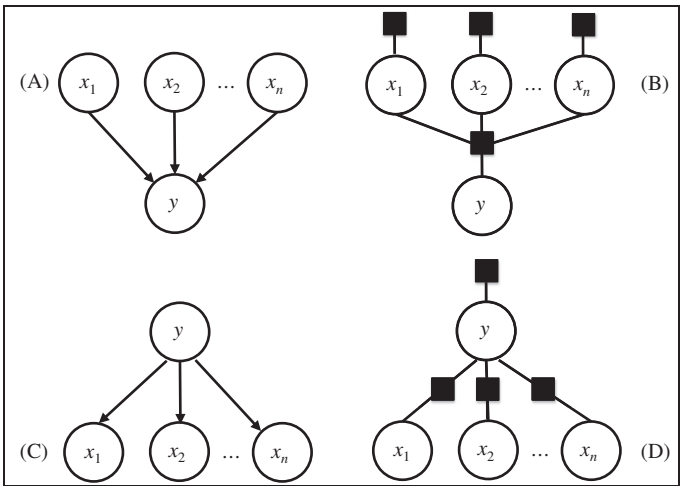

FIGURE 9.14

(A) and (B) Bayesian network and corresponding factor graph; (C) and (D) Naïve Bayes model and corresponding factor graph.

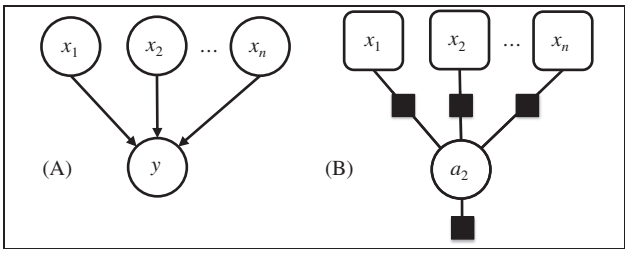

FIGURE 9.15

(A) Bayesian network representing the joint distribution of y and its parents; (B) factor graph for a logistic regression for the conditional distribution of y given its parents.

Let us assume that all variables are binary. Given a separate function $f_i(y,x_i)$ for each binary variable x_i, the conditional distribution defined by a logistic regression model has the form

$$P(y|x_1,\ldots,x_n) = \frac{1}{Z(x_1,\ldots,x_n)}\exp\left(\sum_{i=1}^{n} w_i f_i(y,x_i)\right)$$
$$= \frac{1}{Z(x_1,\ldots,x_n)}\prod_{i=1}^{n}\phi_i(x_i,y).$$

where the denominator Z is a data-dependent normalization term that makes the conditional distribution sum to 1, and $\phi_i(x_i,y) = \exp(w_i f_i(y,x_i))$. This corresponds

to a factor graph that resembles the Naïve Bayes model, but with the factorized conditional distribution shown in Fig. 9.15B. Here, curved rectangles represent variables that are not explicitly defined as random variables. This graph represents the conditional probability function $P(y|x_1, \ldots, x_n)$, and the number of parameters scales linearly because each function is connected to just a pair of variables.

MARKOV RANDOM FIELDS

Markov random fields define another factorized model for a set of random variables X, where these variables are divided into so-called "cliques" X_c and a factor $\Psi_c(X_c)$ is defined for each clique:

$$P(X) = \frac{1}{Z} \prod_{c=1}^{C} \Psi_c(X_c),$$

A clique is a group of nodes in an undirected graph that all connect to every other node in the clique. Z, known as the partition function, normalizes the model to ensure that it is a probability distribution, and consists of a sum over all possible values for all variables in the model. It could be written

$$Z = \sum_{x \in X} \prod_{c=1}^{C} \Psi_c(X_c).$$

Fig. 9.16A and B shows an undirected graph corresponding to a Markov random field, and its factor graph. Again the factor graph makes explicit the nature of the underlying functions used to create the model. For example, it shows that functions are associated with each node, which is not clear from the undirected graph notation. The Markov random field structure in Fig. 9.16 has been widely used for images: this general structure is typically repeated over an entire image, with each node representing a property of a pixel—e.g., a label, or its depth.

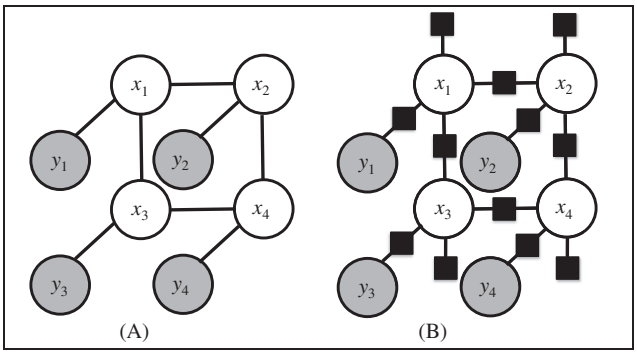

FIGURE 9.16

(A) Undirected graph representing a Markov random field structure; (B) corresponding factor graph.

Fig. 9.16 factorizes the joint probability for four variables as follows:

$$P(x_1, x_2, x_3, x_4) = \frac{1}{Z} f_A(x_1) f_B(x_2) f_C(x_1) f_D(x_2) f_E(x_1, x_2) f_F(x_2, x_3) f_G(x_3, x_4) f_H(x_4, x_1)$$

$$= \frac{1}{Z} \prod_{u=1}^{U} \phi_u(X_u) \prod_{v=1}^{V} \Psi_v(X_v),$$

where $\phi_u(X_u) = \phi_u(x_i)$ represents a set of unary functions of just one variable, while $\Psi_v(X_v) = \Psi_v(x_i, x_j)$ represents a set of pairwise functions of two variables. Subscripts u and v index both the functions and the sets of single variables $X_u = \{x_i\}_u$ and variable pairs $X_v = \{x_i, x_j\}_v$ that serve as their arguments.

This representation can equivalently be expressed using an energy function $F(X)$ of this form:

$$F(X) = \sum_{u=1}^{U} U(X_u) + \sum_{v=1}^{V} V(X_v).$$

Then a Markov random field can be written

$$P(X) = \frac{1}{Z} \exp(-F(X)) = \frac{1}{Z} \exp\left(-\sum_{u=1}^{U} U(X_u) - \sum_{v=1}^{V} V(X_v) \right).$$

Since Z is constant for any assignment of the variables X, the negative log probability under the model can be written

$$-\log P(x_1, x_2, x_3, x_4) = -\log \left[\prod_{u=1}^{U} \phi_u(X_u) \prod_{V=1}^{V} \psi_V(X_V) \right] - \log Z$$
$$\alpha \sum_{u=1}^{U} U(X_u) + \sum_{V=1}^{V} V(X_V),$$

This leads to the commonly used strategy of minimizing an energy function in this form to perform tasks such as image segmentation and entity resolution in text documents. When such minimization tasks are "submodular," a term that denotes a particular category of optimization problems, an exact minimum can be found using algorithms based on graph-cuts; otherwise methods such as tree-reweighted message passing are used.

COMPUTING USING THE SUM-PRODUCT AND MAX-PRODUCT ALGORITHMS

Key quantities of interest for any probability model are the marginal probabilities and the most probable explanation of the model. For tree-structured graphical models, exact solutions for these can be found efficiently by the sum-product and max-product algorithms. When applied to the hidden Markov models discussed in Section 9.8, these are known as the *forwards-backwards* and *Viterbi* algorithms, respectively. We begin with some simple examples for motivation, and then present the algorithms themselves.

Marginal probabilities

Given a Bayesian network, an initial step is to determine the marginal probability of each node given no observations whatsoever. These *single node marginals* differ from the conditional and unconditional probabilities that were used to specify the network. Indeed, software packages for manipulating Bayesian networks often take the definition of a network in terms of the underlying conditional and unconditional probabilities and show the user the single node marginals for each node in a visual interface. The marginal for variable x_i is

$$P(x_i) = \sum_{x_{j \neq i}} P(x_1, \ldots, x_n),$$

where the sum is over the states of all variables $x_j \neq x_i$, and can be computed by the sum-product algorithm. In fact, the same algorithm serves in many other situations, such as when some variables are observed and we wish to compute the belief of others, and also for finding the posterior distributions needed for learning—e.g., using the EM algorithm.

Consider the task of computing the marginal probability of variable x_3 given the observation $x_4 = \tilde{x}_4$ from the Bayesian network in Fig. 9.12A. Since we are conditioning on a variable, we need to compute a marginal *conditional* probability. This corresponds to the practical notion of posing a query, where the model is used to infer an updated belief about x_3 given the state of variable x_4.

Since other variables in the graph have not been observed, they should be integrated out of the graphical model to obtain the desired result:

$$P(x_3|\tilde{x}_4) = \frac{P(x_3, \tilde{x}_4)}{P(\tilde{x}_4)} = \frac{P(x_3, \tilde{x}_4)}{\sum_{x_3}(x_3, \tilde{x}_4)},$$

Here the key probability of interest is

$$\begin{aligned} P(x_3, \tilde{x}_4) &= \sum_{x_1} \sum_{x_2} \sum_{x_5} P(x_1, x_2, x_3, \tilde{x}_4, x_5) \\ &= \sum_{x_1} \sum_{x_2} \sum_{x_5} P(x_1) P(x_2) P(x_3|x_1) P(\tilde{x}_4|x_1, x_2) P(x_5|x_2). \end{aligned}$$

However, this sum involves a large data structure containing the joint probability, composed of the products over the individual probabilities. The *sum-product algorithm* refers to a much better solution: simply *push the sums as far as possible to the right* before computing products of probabilities. Here, the required marginalization can be computed by

$$\begin{aligned} P(x_3, \tilde{x}_4) &= \sum_{x_1} P(x_3|x_1) P(x_1) \sum_{x_2} P(\tilde{x}_4|x_1, x_2) P(x_2) \sum_{x_5} P(x_5|x_2) \\ &= \sum_{x_1} P(x_3|x_1) P(x_1) P(\tilde{x}_4|x_1) \\ &= \sum_{x_1} P(x_1, x_3, \tilde{x}_4). \end{aligned}$$

The sum-product algorithm

The approach illustrated by this simple example can be generalized into an algorithm for computing marginals that can be transformed into conditional marginals if desired. Conceptually, it is based on sending messages between the variables and functions defined by a factor graph.

Begin with variable or function nodes that have only one connection. Function nodes send the message $\mu_{f\to x}(x) = f(x)$ to the variable connected to them, while variable nodes send $\mu_{x\to f}(x) = 1$. Each node waits until it has received a message from all neighbors except the one it sent its message to. Then function nodes send messages of the following form to variable x:

$$\mu_{f\to x}(x) = \sum_{x_1,\ldots,x_K} f(x, x_1, \ldots, x_K) \prod_{k\in N(f)\, x} \mu_{x_k\to f}(x_k),$$

where $N(f)\backslash x$ represents the set of the function node f's neighbors, excluding the recipient variable x; we write these variables of the K other neighboring nodes as $x_1,\ldots,x_K$. If a variable is observed, messages for functions involving it no longer need a sum over the states of the variable, the function is evaluated with the observed state. One could think of the associated variable node as being transformed into the new modified function. There is then no variable to function message for the observed variable.

Variable nodes send messages of this form to functions:

$$\mu_{x\to f}(x) = \mu_{f_1\to x}(x)\ldots\mu_{f_K\to x}(x) = \prod_{k\in N(x)\, f} \mu_{f_k\to x}(x),$$

where the product is over the messages from all neighboring functions $N(x)$ other than the recipient function f; i.e., $fk \in N(x)\, f$. When the algorithm terminates, the marginal probability of each node is the product over all incoming messages from all functions connected to the variable:

$$P(x_i) = \mu_{f_1\to x}(x)\ldots\mu_{f_K\to x}(x)\mu_{f_{K+1}\to x}(x) = \prod_{k=1}^{K+1} \mu_{f_k\to x}(x),$$

This is written as a product over $K + 1$ function messages to emphasize its similarity to the variable-to-function-node messages. After sending a message to any given function f consisting of the product of K messages, the variable simply needs to receive one more incoming message back from f to compute its marginal.

If some of the variables in the graph are observed, the algorithm yields the marginal probability of each variable *and* the observations. The marginal conditional distribution for each variable can be obtained by normalizing the result by the probability of the observation, obtainable from any node by summing over x_i in the resulting distributions, which have the form $P(x_i, \{\tilde{x}_{j\in O}\})$ where O is the set of indices of the observed variables.

As is often the case with probability models, multiplying many probabilities quickly leads to very small numbers. The sum-product algorithm is often implemented with rescaling. Alternatively, the computations can be performed in log

space (as they are in the max-product algorithm; see below), leading to computations of the form $c = \log(\exp(a) + \exp(b))$. To help prevent loss of precision when computing the exponents, note that

$$c = \log(e^a + e^b) = a + \log(1 + e^{b-a}) = b + \log(1 + e^{a-b}),$$

and pick the expression with the smaller exponent.

Sum-product algorithm example

The idea behind the sum-product algorithm is to push sums as far to the right as possible, and this is done efficiently for all variables simultaneously. When the algorithm is used to compute $P(x_3, \tilde{x}_4)$ from the Bayesian network in Fig. 9.12A and the corresponding factor graph in Fig. 9.17, the key messages involved are:

$$P(x_3, \tilde{x}_4) = \sum_{x_1} P(x_3|x_1) \underbrace{P(x_1)}_{1d} \sum_{x_2} P(\tilde{x}_4|x_1, x_2) \underbrace{P(x_2)}_{1c} \underbrace{\sum_{x_5} P(x_5|x_2)}_{2a} \cdot \underbrace{1}_{1a}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{3a}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{4a}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{5a}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{6a}$$

These numbered messages can be written

1a: $\mu_{x_5 \to f_E}(x_5) = 1$, 1c: $\mu_{f_B \to x_2}(x_2) = f_B(x_2)$, 1d: $\mu_{f_A \to x_1}(x_1) = f_A(x_1)$
2a: $\mu_{f_E \to x_2}(x_2) = \sum_{x_5} f_E(x_5, x_2)$
3a: $\mu_{x_2 \to f_D}(x_5) = \mu_{f_B \to x_2}(x_2)\mu_{f_E \to x_2}(x_2)$
4a: $\mu_{f_D \to x_1}(x_1) = \sum_{x_2} f_D(\tilde{x}_4|x_1, x_2)\mu_{x_2 \to f_D}(x_5)$
5a: $\mu_{x_1 \to f_C}(x_1) = \mu_{f_A \to x_1}(x_1)\mu_{f_D \to x_1}(x_1)$
6a: $\mu_{f_C \to x_3}(x_3) = \sum_{x_1} f_C(x_3, x_1)\mu_{x_1 \to f_C}(x_1)$

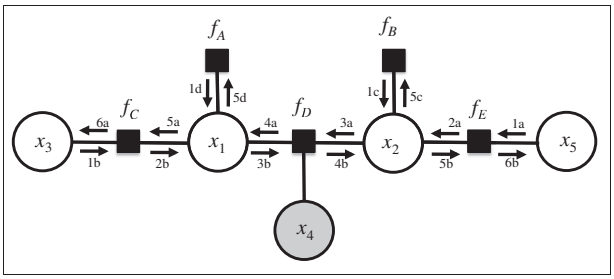

FIGURE 9.17

Message sequence in an example factor graph.

Keep in mind that the complete algorithm will yield all single-variable marginals in the graph using the other messages shown in Fig. 9.17, but not enumerated above. This simple example is based on a Bayesian network, which transforms into a chain-structured factor graph when $\tilde{x}_4$ is observed, and the message passing structure resembles the computations used for the hidden Markov models and conditional random fields discussed below. For long chains or large tree-structured networks, such computations are essential for computing the necessary quantities efficiently, without recourse to approximate methods.

Most probable explanation example

Finding the most probable configuration of all other variables in our example given $x_4 = \tilde{x}_4$ involves searching for

$$\{x_1^*, x_2^*, x_3^*, x_5^*\} = \arg\max_{x_1,x_2,x_3,x_5} P(x_1, x_2, x_3, x_5|\tilde{x}_4),$$

for which

$$P(x_1^*, x_2^*, x_3^*, x_5^*|\tilde{x}_4) = \max_{x_1,x_2,x_3,x_5} P(x_1, x_2, x_3, x_5|\tilde{x}_4).$$

The joint probability is related to the conditional by a constant, so one could equally well find the maximum of $P(x_1, x_2, x_3, \tilde{x}_4, x_5)$. Because max behaves in a similar way to sum, we can take a tip from the sum-product and push the max operations as far to the right as possible, noting that $\max(ab, ac) = a \max(b, c)$. Here,

$$\begin{aligned}&\max_{x_1}\max_{x_2}\max_{x_3}\max_{x_5} P(x_1, x_2, x_3, \tilde{x}_4, x_5)\\ &\quad = \max_{x_3}\max_{x_1} P(x_1)P(x_3|x_1) \max_{x_2} P(x_2)P(\tilde{x}_4|x_1, x_2) \max_{x_5} P(x_5|x_2).\end{aligned}$$

Consider the max over x_5 at the far right, which seeks the greatest probability among the possible states of x_5 for each of the possible states of x_2. This involves creating a table giving the maximum values of x_5 for each configuration of x_2. The next operation to the left, the max over x_2, involves multiplying the current maximum values in the table for x_2 by the corresponding probabilities for $P(x_2)P(\tilde{x}_4|x_1, x_2)$. We thus need to find the maximum for each state of x_2 for each state of x_1, which can be modeled by producing a message over the states of x_1 with the greatest values obtained for each possible state based on all information propagated so far from the right.

This process continues until eventually we have a final max over x_3. This gives a single value, the probability of the most probable explanation, which corresponds to the entry in the data structure with scores for each state of x_3. By changing which variable is used to compute the final max, we can extract it from any variable, because this will lead to the same maximum value. However, taking the *arg max* for each variable will yield the desired most probable explanation $x_1^*, x_2^*, x_3^*, x_5^*$—the configuration of all variables which has the greatest probability. Generalizing these ideas and performing them efficiently leads to the max-product algorithm, a general template for performing such computations exactly in arbitrary tree-structured graphs.

The max-product or max-sum algorithm

The max-product algorithm can be used to find the most probable explanation in a tree-structured probability model. It is generally implemented in logarithmic space to alleviate problems with numerical stability, where it is better characterized as the max-sum algorithm. Since the log function increases monotonically, $\log(\max_x p(x)) = \max_x \log p(x)$, and, as mentioned above, $\max(c + a, c + b) = c + \max(a, b)$. These properties allow the maximum probability configuration in a tree-structured probability model to be computed as follows.

As in the sum-product algorithm, variables or factors that have only one connection in the graph begin by sending either a function-to-variable message consisting of $\mu_{x\to f}(x) = 0$, or a variable-to-function message consisting of $\mu_{f\to x}(x) = \log f(x)$. Each function and variable node in the graph waits until it has received a message from all neighbors other than the node that will receive its message. Then function nodes send messages of the following form to variable x

$$\mu_{f\to x}(x) = \max_{x_1,\ldots,x_K} \left[\log f(x, x_1, \ldots, x_K) + \sum_{k\in N(f)\,x} \mu_{x_k\to f}(x_k)\right],$$

where the notation $N(f)\backslash x$ is the same as for the sum-product algorithm above. Likewise, variables send messages of this form to functions:

$$\mu_{x\to f}(x) = \sum_{k\in N(x)\,f} \mu_{f_k\to x}(x),$$

where the sum is over the messages from all functions other than the recipient function. When the algorithm terminates, the probability of the most probable configuration can be extracted from any node using

$$p^* = \max_x \left[\sum_{k\in N(x)} \mu_{f_k\to x}(x)\right].$$

The most probable configuration itself can be obtained by applying this computation to each variable:

$$x^* = \arg\max_x \left[\sum_{k\in N(x)} \mu_{f_k\to x}(x)\right].$$

To understand how this works for a concrete example, follow the sequence of messages illustrated in Fig. 9.17, but use the max-product messages defined above and the final computations for the max and arg max in place of the sum-product messages we examined earlier.

The max-product algorithm is widely used to make final predictions for tree-structured Bayesian networks, as well as for sequences of labels in conditional random fields and hidden Markov models, discussed in Sections 9.7 and 9.8, respectively.

9.7 CONDITIONAL PROBABILITY MODELS

You may be surprised to learn that the regression models of Section 4.6 correspond to the simplest and most popular types of conditional probability model. The Linear and Polynomial Regression as Probability Models section views linear regression through the lens of probability, and we go on to examine the multiclass extension to logistic regression, expressing this in both scalar and matrix-vector forms. The leap to the matrix-vector form reveals that key aspects of modeling and learning can be expressed in compact and elegant ways. This allows computer implementations to be accelerated by using libraries—or hardware—for matrix-vector manipulation, which are highly optimized and exploit modern computing hardware. Graphics processing units (GPUs) can yield execution speeds that are orders of magnitude faster than standard implementations.

LINEAR AND POLYNOMIAL REGRESSION AS PROBABILITY MODELS

Suppose the conditional probability distribution for the observations of a continuous variable y_i given observations of another variable x_i is a linear Gaussian:

$$p(y_i|x_i) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{\{y_i-(\theta_0+\theta_1 x_i)\}^2}{2\sigma^2}\right],$$

where the parameters θ_0 and θ_1 represent the slope and intercept. The conditional distribution for the observed y_i's given the corresponding observed x_i's can be defined as

$$p(y_1,\ldots,y_N|x_1,\ldots,x_N) = \prod_{i=1}^{N} p(y_i|x_i).$$

As usual, we work with the log-likelihood instead:

$$L_{y|x} = \log\prod_{i=1}^{N} p(y_i|x_i) = \sum_{i=1}^{N}\log p(y_i|x_i).$$

This simplifies to

$$\begin{aligned} L_{y|x} &= \sum_{i=1}^{N}\log\left\{\frac{1}{\sigma\sqrt{2\pi}}\exp\left[-\frac{\{y_i-(\theta_0+\theta_1 x_i)\}^2}{2\sigma^2}\right]\right\} \\ &= -N\log\left[\sigma\sqrt{2\pi}\right] - \sum_{i=1}^{N}\frac{\{y_i-(\theta_0+\theta_1 x_i)\}^2}{2\sigma^2}. \end{aligned}$$

The first term is independent of the data. Thus to find the parameters that maximize the log-likelihood, it suffices to find the parameters that minimize the squared error:

$$\arg\max_{\theta_0,\theta_1}(L_{y|x}) = \arg\min_{\theta_0,\theta_1}\left(\sum_{i=1}^{N}\{y_i-(\theta_0+\theta_1 x_i)\}^2\right).$$

This is ordinary linear regression!

Although x_i is a scalar here, the method generalizes to a vector $\mathbf{x}_i$. Categorical variables can be encoded into a subset of the dimensions of $\mathbf{x}$ using the so-called "one-hot" method, placing a 1 in the dimension corresponding to the category label and 0s in all other dimensions allocated to the variable. If all input variables are categorical, this corresponds to the classic *analysis of variance* (ANOVA) method.

USING PRIORS ON PARAMETERS

Placing a Gaussian prior on the parameters $\mathbf{w}$ leads to the method of ridge regression (Section 7.2)— also called "weight decay." Consider a regression that uses a D-dimensional vector $\mathbf{x}$ to make predictions. The regression's bias term can be represented by defining the first dimension of $\mathbf{x}$ as the constant 1 for every example. Defining $[\theta_1 \ldots \theta_D] = \mathbf{w}^T$ and using the usual $N(x;\mu,\sigma)$ notation for scalar Gaussians, the underlying probability model is

$$\prod_{i=1}^{N} p(y_i|x_i;\theta)p(\theta;\tau) = \left[\prod_{i=1}^{N} N(y_i;\mathbf{w}^T x_i,\sigma^2)\right]\left[\prod_{d=1}^{D} N(w_d;0,\tau^2)\right],$$

where τ is the hyperparameter specifying the prior. Setting $\lambda \equiv \sigma^2/\tau^2$, it is possible to show that maximum a posteriori parameter estimation based on the log conditional likelihood is equivalent to minimizing the squared error loss function

$$F(\mathbf{w}) = \sum_{i=1}^{N}\{y_i-\mathbf{w}^T x_i\}^2 + \lambda \mathbf{w}^T\mathbf{w}, \tag{9.4}$$

which includes an L_2-based regularization term given by $R_{L_2}(\mathbf{w}) = \mathbf{w}^T\mathbf{w} = ||\mathbf{w}||_2^2$. ("Regularization" is another term for overfitting avoidance.)

Using a Laplace prior for the distribution over weights and taking the log of the likelihood function yields an L_1-based regularization term $R_{L_1}(\mathbf{w}) = ||\mathbf{w}||_1$. To see why, note that the Laplace distribution has the form

$$P(w;\mu,b) = L(w;\mu,b) = \frac{1}{2b}\exp\left(-\frac{|w-\mu|}{b}\right),$$

where μ and b are parameters. Modeling the prior probability for each weight by a Laplace distribution with $\mu_j = 0$ yields

$$-\log\left[\prod_{d=1}^{D} L(w_d; 0, b)\right] = \log(2b) + \frac{1}{b}\sum_{d=1}^{D} |w_d| \propto ||\mathbf{w}||_1.$$

Since the Laplace distribution places more probability at zero than the Gaussian distribution does, it can provide both regularization and variable selection in regression problems. This technique has been popularized as a regression approach known as the LASSO, "Least Absolute Shrinkage and Selection Operator."

An alternative approach known as the "elastic net" combines L_1 and L_2 regularization techniques using

$$\lambda_1 R_{L_1}(\boldsymbol{\theta}) + \lambda_2 R_{L_2}(\boldsymbol{\theta}) = \lambda_1 ||\mathbf{w}||_1 + \lambda_2 ||\mathbf{w}||_2^2.$$

This corresponds to a prior distribution that consist of the product of a Gaussian and a Laplacian distribution. The result leads to convex optimization problems—i.e., problems where any local minimum must be a global minimum—if the loss is convex, which applies to models like logistic or linear regression.

Matrix vector formulations of linear and polynomial regression

This section formulates linear regression using matrix operations. Observe that the loss in (9.4)—without the penalty term—can be written

$$\begin{aligned}
&\sum_{i=1}^{N}\{y_i-(\theta_0+\theta_1 x_{1i}+\theta_2 x_{2i}+\ldots+\theta_D x_{Di})\}^2 \\
&= \left(\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} 1 & x_{11} & x_{21} & \cdots & x_{D1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{1N} & x_{2N} & & x_{DN} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_D \end{bmatrix}\right)^T \left(\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} 1 & x_{11} & x_{21} & \cdots & x_{D1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{1N} & x_{2N} & & x_{DN} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_D \end{bmatrix}\right) \\
&= (\mathbf{y}-\mathbf{A}\mathbf{w})^T(\mathbf{y}-\mathbf{A}\mathbf{w}).
\end{aligned}$$

where the vector $\mathbf{y}$ is just the individual y_i's stacked up, and $\mathbf{w}$ is the vector of parameters (or weights) for the model. Taking the partial derivative with respect to $\mathbf{w}$ and setting the result to zero yields a closed form expression for the parameters:

$$\begin{aligned}
&\frac{\partial}{\partial \mathbf{w}}(\mathbf{y}-\mathbf{A}\mathbf{w})^T(\mathbf{y}-\mathbf{A}\mathbf{w}) = 0 \\
&\Rightarrow \mathbf{A}^T\mathbf{A}\mathbf{w} = \mathbf{A}^T\mathbf{y} \\
&\mathbf{w} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}.
\end{aligned} \tag{9.5}$$

These are famous equations. $\mathbf{A}^T\mathbf{A}\mathbf{w} = \mathbf{A}^T\mathbf{y}$ is known as the *normal equations*, and the quantity $\mathbf{A}^+ = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ is known as the *pseudoinverse*. Note that $\mathbf{A}^T\mathbf{A}$ is not always invertible, but this problem can be addressed using regularization.

For ridge regression, a prior is added to make the objective function

$$F(\mathbf{w}) = (\mathbf{y} - \mathbf{A}\mathbf{w})^T(\mathbf{y} - \mathbf{A}\mathbf{w}) + \lambda \mathbf{w}^T\mathbf{w}, \tag{9.6}$$

for a suitably defined matrix $\mathbf{A}$ with vectors $\mathbf{x}_i$ in the rows, and λ as defined above. Again, setting the partial derivative of $F(\mathbf{w})$ to zero yields a closed form solution:

$$\frac{\partial}{\partial \mathbf{w}} F(\mathbf{w}) = 0$$

$$\Rightarrow \mathbf{A}^T\mathbf{A}\mathbf{w} + \lambda \mathbf{w} = \mathbf{A}^T\mathbf{y}$$
$$\mathbf{w} = (\mathbf{A}^T\mathbf{A} + \lambda \mathbf{I})^{-1}\mathbf{A}^T\mathbf{y}.$$

This modification to the pseudoinverse equation allows solutions to be found that would otherwise not exist, often using a very small λ. It is often presented as a regularization method that avoids numerical instability, but the analysis in terms of a prior on parameters gives more insight. For example, it may be appropriate to use Gaussian priors of different strengths for different terms. In fact, it is common practice to impose no penalty at all on the bias weight. This could be implemented by replacing the $\lambda \mathbf{w}^T\mathbf{w}$ term in (9.6) by $\mathbf{w}^T\mathbf{D}\mathbf{w}$, where $\mathbf{D}$ is a diagonal matrix containing the λ_i's to be used for each weight, transforming the solution into $\mathbf{w} = (\mathbf{A}^T\mathbf{A} + \mathbf{D})^{-1}\mathbf{A}^T\mathbf{y}$. While the above expression for the partial derivative is fairly simple, care must still be taken during implementation to avoid numerically instable results.

The linear regression model can be transformed into a nonlinear polynomial model. Although the polynomial regression model yields nonlinear predictions, the estimation problem is linear in the parameters. To see this, express the problem in matrix form, using a suitably defined matrix $\mathbf{A}$ to encode the polynomial's higher order terms and a vector $\mathbf{c}$ to encode the coefficients, including those used for the higher order terms:

$$\begin{aligned}
&\sum_{i=1}^{N} \{y_i - (\theta_0 + \theta_1 x_i + \theta_2 x_i^2 + \cdots + \theta_K x_i^K)\}^2 \\
&= \left(\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^K \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & & x_N^K \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_K \end{bmatrix} \right)^{\mathbf{T}} \left(\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^K \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & & x_N^K \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_K \end{bmatrix} \right) \\
&= (\mathbf{y} - \mathbf{A}\mathbf{c})^T(\mathbf{y} - \mathbf{A}\mathbf{c}).
\end{aligned}$$

Eq. (9.5) can be used to solve for the parameters in closed form.

This trick of transforming a linear prediction method into a nonlinear one while keeping the underlying estimation problem linear can be generalized. The general approach goes by the name of *basis function expansion*. For polynomial regression, the polynomial basis given by the rows of the matrix $\mathbf{A}$ (the powers of x) is used. However, any nonlinear function of the inputs $\phi(\mathbf{x})$ could be used to define models of the form

$$p(y|\mathbf{x}) = N(y; \mathbf{w}^T\phi(\mathbf{x}), \sigma^2),$$

which also give closed form solutions for a linear parameter estimation problem. We will return to this when discussing kernelizing probabilistic models below.

MULTICLASS LOGISTIC REGRESSION

Binary logistic regression was introduced in Section 4.6. Consider now a *multi-class classification problem* where class values are encoded as instances of the random variable $y \in \{1, \ldots, N\}$, and, as before, feature vectors are instances of a variable $\mathbf{x}$. Assume there is no significance to the order of the classes.

A simple linear probabilistic classifier can be created using this parametric form:

$$p(y|\mathbf{x}) = \frac{\exp\left(\sum_{k=1}^{K} w_k f_k(y, \mathbf{x})\right)}{\sum_y \exp\left(\sum_{k=1}^{K} w_k f_k(y, \mathbf{x})\right)} = \frac{1}{Z(\mathbf{x})} \exp\left(\sum_{k=1}^{K} w_k f_k(y, \mathbf{x})\right), \tag{9.7}$$

which employs K *feature functions* $f_k(y,\mathbf{x})$ and K weights, w_k as the parameters of the model. This is one way of formulating *multinomial logistic regression*. The feature functions could encode complex features extracted from the input vector $\mathbf{x}$. To perform learning using maximum conditional likelihood and observations that are instances of y and $\mathbf{x}$, $\{\tilde{y}_1, \ldots, \tilde{y}_N, \tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_N\}$, write the objective function as

$$L_{y|x} = \log \prod_{i=1}^{N} p(\tilde{y}_i|\tilde{\mathbf{x}}_i) = \sum_{i=1}^{N} \log p(\tilde{y}_i|\tilde{\mathbf{x}}_i).$$

Unfortunately there is no closed form solution for the parameter values that maximize this conditional likelihood, and optimization is usually performed using a gradient based procedure. On taking the partial derivative with respect to one of the weights of the log conditional probability for just one observation we have

$$\begin{aligned}
\frac{\partial}{\partial w_j} p(\tilde{y}|\tilde{\mathbf{x}}) &= \frac{\partial}{\partial w_j} \left\{ \log \left(\frac{1}{Z(\tilde{\mathbf{x}})} \exp \left(\sum_{k=1}^{K} w_k f_k(\tilde{y}, \tilde{\mathbf{x}}) \right) \right) \right\} \\
&= \frac{\partial}{\partial w_j} \left\{ \left[\sum_{k=1}^{K} \underbrace{w_k f_k(\tilde{y}, \tilde{\mathbf{x}})}_{\text{easy part}} \right] - \underbrace{\log Z(\tilde{\mathbf{x}})}_{\text{cool part}} \right\} \\
&= f_{k=j}(\tilde{y}, \tilde{\mathbf{x}}) - \frac{\partial}{\partial w_j} \left\{ \log \left[\sum_y \exp \left(\sum_{k=1}^{K} w_k f_k(y, \tilde{\mathbf{x}}) \right) \right] \right\}.
\end{aligned}$$

The derivative breaks apart into two terms. The first is easy because all terms involving weights where $w_k \neq w_j$ are 0, leaving the sum with just one term. The second, while seemingly daunting, has a derivative that yields an intuitive and interpretable result:

$$
\begin{aligned}
-\frac{\partial}{\partial w_j}\{\log Z(\tilde{\mathbf{x}})\} &= -\frac{\partial}{\partial w_j}\left\{\log\left[\sum_y \exp\left(\sum_{k=1}^{K} w_k f_k(y,\tilde{\mathbf{x}})\right)\right]\right\} \\
&= -\frac{\sum_y \exp\left(\sum_{k=1}^{K} w_k f_k(y,\tilde{\mathbf{x}})\right) f_j(y,\tilde{\mathbf{x}})}{\sum_y \exp\left(\sum_{k=1}^{K} w_k f_k(y,\tilde{\mathbf{x}})\right)} \\
&= -\sum_y p(y|\tilde{\mathbf{x}}) f_j(y,\tilde{\mathbf{x}}) = -\mathrm{E}[f_j(y,\tilde{\mathbf{x}})]_{p(y|\tilde{\mathbf{x}})},
\end{aligned}
$$

which corresponds to the expectation $E[.]_{P(y|\tilde{\mathbf{x}})}$ of the feature function $f_j(y|\tilde{\mathbf{x}})$ under the probability distribution $p(y|\tilde{\mathbf{x}})$ given by the model with the current parameter settings. Write the feature functions as a function $\mathbf{f}(\tilde{y}|\tilde{\mathbf{x}})$ in vector form. Then for a vector of weights $\mathbf{w}$, the partial derivative of the conditional log-likelihood for the entire dataset with respect to $\mathbf{w}$ is

$$
\frac{\partial}{\partial \mathbf{w}} L_{y|x} = \sum_{i=1}^{N} [\mathbf{f}(\tilde{y}_i|\tilde{\mathbf{x}}_i) - E[\mathbf{f}(y_i|\tilde{\mathbf{x}}_i)]_{P(y_i|\tilde{\mathbf{x}}_i)}].
$$

This consists of the sum of the differences between the observed feature vector for a given example and the expected value of the feature vector under the current model settings. If the model were perfect, classifying each example correctly with probability 1, the partial derivative would be zero. Intuitively, the learning procedure adjusts the model parameters so as to produce predictions that are closer to the observed data.

Matrix vector formulation of multiclass logistic regression

The model in (9.7) for a multiclass linear probabilistic classifier using vectors $\mathbf{w}_c$ for the weights associated with each class can be written

$$
p(y = c|\mathbf{x}) = \frac{\exp(\mathbf{w}_c^T \mathbf{x})}{\sum_{\mathbf{y}} \exp(\mathbf{w}_y^T \mathbf{x})},
$$

where y is an index, the weights are encoded into a vector of length K, and the features $\mathbf{x}$ have been re-defined as the result of evaluating the feature functions $f_k(y,\mathbf{x})$ in such a way that there is no difference between the features given by $f_k(y = i,\mathbf{x})$ and $f_k(y = j,\mathbf{x})$. This form is widely used for the last layer in neural network models, where it is referred to as the *softmax* function.

The information concerning class labels can be encoded into a *multinomial vector* $\mathbf{y}$, which is all 0s except for a single 1 in the dimension that represents the correct class label—e.g., $\mathbf{y} = [0 \quad 1 \quad 0 \ldots \quad 0]^T$ for the second class. The weights form a matrix $\mathbf{W} = [\mathbf{w}_1 \quad \mathbf{w}_2 \quad \ldots \quad \mathbf{w}_K]^T$, and the biases form a vector $\mathbf{b} = [b_1 \quad b_2 \quad \ldots \quad b_K]^T$. Then the model yields *vectors of probabilities*

$$p(\mathbf{y}|\mathbf{x}) = \frac{\exp(\mathbf{y}^T\mathbf{W}\mathbf{x} + \mathbf{y}^T\mathbf{b})}{\sum_{\mathbf{y}\in Y}\exp(\mathbf{y}^T\mathbf{W}\mathbf{x} + \mathbf{y}^T\mathbf{b})},$$

where the denominator sums over each possible label $\mathbf{y}\in Y$, which is $Y = \{[1\ 0\ 0\ \ldots\ 0]^T, [0\ 1\ 0\ \ldots\ 0]^T, \ldots, [0\ 0\ 0\ \ldots\ 1]^T\}$. Redefining $\mathbf{x}$ as $\mathbf{x} = [\mathbf{x}^{\mathrm{T}}\ 1]^{\mathrm{T}}$ and the parameters as a matrix of the form

$$\boldsymbol{\theta} = [\mathbf{W}\ \mathbf{b}] = \begin{bmatrix} \mathbf{w}_1^T & b_1 \\ \mathbf{w}_2^T & b_2 \\ \vdots & \vdots \\ \mathbf{w}_k^T & b_k \end{bmatrix},$$

the conditional model can be written in this compact matrix-vector form:

$$p(\mathbf{y}|\mathbf{x}) = \frac{\exp(\mathbf{y}^T\boldsymbol{\theta}\mathbf{x})}{\sum_{\mathbf{y}\in Y}\exp(\mathbf{y}^T\boldsymbol{\theta}\mathbf{x})} = \frac{1}{Z(\mathbf{x})}\exp(\mathbf{y}^T\boldsymbol{\theta}\mathbf{x}).$$

Then the gradient of the log-conditional-likelihood with respect to the parameter matrix $\boldsymbol{\theta}$ can be expressed as

$$\begin{aligned}\frac{\partial}{\partial\boldsymbol{\theta}}\log\prod_i p(\tilde{\mathbf{y}}_i|\tilde{\mathbf{x}}_i;\boldsymbol{\theta}) &= \sum_{i=1}^{N}\left[\frac{\partial}{\partial\boldsymbol{\theta}}(\tilde{\mathbf{y}}_i^T\boldsymbol{\theta}\tilde{\mathbf{x}}_i) - \frac{\partial}{\partial\boldsymbol{\theta}}\log Z(\tilde{\mathbf{x}}_i)\right] \\ &= \sum_{i=1}^{N}\tilde{\mathbf{y}}_i\tilde{\mathbf{x}}_i^T - \sum_{i=1}^{N}\sum_{\mathbf{y}\in Y}P(\mathbf{y}|\tilde{\mathbf{x}}_i)\mathbf{y}\tilde{\mathbf{x}}_i^T \\ &= \sum_{i=1}^{N}\tilde{\mathbf{y}}_i\tilde{\mathbf{x}}_i^T - \sum_{i=1}^{N}E[\mathbf{y}\tilde{\mathbf{x}}_i^T]_{P(\mathbf{y}|\tilde{\mathbf{x}}_i)},\end{aligned}$$

where the notation $E[\mathbf{y}\tilde{\mathbf{x}}_i^T]_{P(\mathbf{y}|\tilde{\mathbf{x}}_i)}$ denotes the expectation of the random vector $\mathbf{y}$ under $P(\mathbf{y}|\tilde{\mathbf{x}}_i)$, the vector of conditional probabilities that the model yields for the observed input $\tilde{\mathbf{x}}_i$ under the current parameter settings. The first term corresponds to a matrix that is computed just once, when the procedure commences. The second corresponds to a matrix that approaches the first term more closely as the model learns to make predictions that match the observed data.

Formulating these equations as vector and matrix operations allows highly optimized numerical libraries to be applied. Fast libraries for vector and matrix operations are key facilitators of big data techniques. In particular, state-of-the-art methods for deep learning with large datasets rely heavily on the dramatic performance improvements enabled by GPUs.

Note that we have not taken account of the fact that the final prediction need only yield probabilities for $N-1$ of the N classes, the remaining one being inferred from the fact that they must all sum to 1. Multinomial logistic regression models can be formulated that exploit this fact and involve fewer parameters.

Priors on parameters, and the regularized loss function

Logistic regression is frequently performed with some regularizer or prior on the parameters to combat overfitting. From a probabilistic perspective, this means

that the conditional probability for the set of all labels Y given the set of all input vectors X can be rewritten

$$p(Y, \boldsymbol{\theta}|X) = p(\boldsymbol{\theta}; \sigma) \prod_{i=1}^{N} p(y_i|\mathbf{x}_i, \boldsymbol{\theta}),$$

where $p(\boldsymbol{\theta}; \sigma)$ is a prior distribution for the parameters. Given observed data $\tilde{Y}, \tilde{X}$, finding the value of $\boldsymbol{\theta}$ that maximizes this expression is an instance of maximum a posteriori parameter estimation with a conditional probability model. The goal, then, is to minimize

$$-\log p(\tilde{Y}, \boldsymbol{\theta}|\tilde{X}) = -\sum_{i=1}^{N} \log p(\tilde{y}_i|\tilde{\mathbf{x}}_i) - \log p(\boldsymbol{\theta}; \lambda)$$

The first term is the negative log-likelihood, corresponding to the loss function, and the second is the negative log of the prior for the parameters, also known as the "regularization" term. L_2 regularization is often used for the weights in a logistic regression model. A prior could be applied to the bias too; but in practice it is often better not to do this (or, equivalently, to use a uniform distribution as the prior).

So-called "L_2 regularization" is based on the L_2 norm, which is just the Euclidean distance $||\mathbf{w}||_2 = \sqrt{\mathbf{w}^T\mathbf{w}}$. If a Gaussian distribution with zero mean and a common variance for each weight is used as the prior for the weights, the corresponding regularization term is the squared L_2 distance weighted by λ, plus a constant:

$$-\log p(\theta; \sigma) = \lambda||\mathbf{w}||_2^2 + const.$$

The constant can be ignored during the optimization and is often omitted from the regularized loss function. However, using $\lambda/2$ or $1/2\ \sigma^2$ as the regularization term gives a more direct correspondence with the σ^2 parameter of the Gaussian distribution.

Solving a regularized multiclass logistic regression with M weight parameters corresponds to finding the weights and biases that minimize

$$-\sum_{i=1}^{N} \log p(\tilde{y}_i|\tilde{\mathbf{x}}_i; \mathbf{W}, \mathbf{b}) + \lambda \sum_{j=1}^{M} w_j^2,$$

the regularization term will encourage the weights to be small. In the context of multilayer perceptrons, this sort of regularization is known as weight decay; in statistics it is known as ridge regression.

In regression, the elastic net regularization introduced earlier combines L_1 and L_2 regularization using $\lambda_2 R_{L_2}(\boldsymbol{\theta}) + \lambda_1 R_{L_1}(\boldsymbol{\theta})$, which corresponds to a prior on weights consisting of the product of Laplacian and Gaussian distributions. In terms of a matrix representation, given a weight matrix $\mathbf{W}$ with row and column entries given by $w_{r,c}$, this can be written

$$R_{L_2}(\boldsymbol{\theta}) = \sum_r \sum_c (w_{r,c})^2, \frac{\partial}{\partial \mathbf{W}} R_{L_2}(\boldsymbol{\theta}) = 2\mathbf{W},$$

$$R_{L_1}(\boldsymbol{\theta}) = \sum_r \sum_c |w_{r,c}|, \frac{\partial}{\partial w_{r,c}} R_{L_1}(\boldsymbol{\theta}) = \begin{cases} 1, & w_{r,c} > 0 \\ 0, & w_{r,c} = 0 \\ -1, & w_{r,c} < 0 \end{cases},$$

where we have set the derivative of the L_1 prior to zero at zero despite the fact that it is technically undefined at this point. This is common practice, since the goal of this type of regularization is to induce sparsity: once a weight is zero the gradient will be zero. Notice that regularization has not been applied to the bias terms. This regularizer leads to convex optimization problems if the loss is convex, which is the case for logistic regression and linear regression.

GRADIENT DESCENT AND SECOND-ORDER METHODS

By formulating maximum likelihood learning with a prior on parameters as the problem of minimizing the negative log probability, gradient descent can be used to optimize the model's parameters. Given a conditional probability model $p(y|\mathbf{x};\boldsymbol{\theta})$ with parameter vector $\boldsymbol{\theta}$ and data $\tilde{y}_i, \tilde{\mathbf{x}}_i, i = 1, \ldots, N$, along with a prior on parameters given by $p(\boldsymbol{\theta};\lambda)$, with hyperparameter λ, the gradient descent procedure with learning rate η is:

$$\begin{array}{l} \boldsymbol{\theta} = \boldsymbol{\theta}_o // \text{initialize parameters} \\ \text{while converged} == \text{FALSE} \\ \quad \mathbf{g} = \dfrac{\partial}{\partial \boldsymbol{\theta}} \left[-\sum_{i=1}^{N} \log p(\tilde{y}_i|\tilde{\mathbf{x}}_i;\boldsymbol{\theta}) - \log p(\boldsymbol{\theta};\lambda) \right] \\ \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{g} \end{array}$$

Convergence is usually determined by monitoring the change in the loss or the parameters and terminating when one of them stabilizes. Appendix A.1 shows how a Taylor series expansion can be used to interpret and justify the learning rate parameter η.

Alternatively, gradient descent can be based on the second derivative by computing the Hessian matrix $\mathbf{H}$ at each iteration and replacing the above update by

$$\begin{array}{l} \mathbf{H} = \dfrac{\partial^2}{\partial \boldsymbol{\theta}^2} \left[-\sum_{i=1}^{N} \log p(\tilde{y}_i|\tilde{\mathbf{x}}_i;\boldsymbol{\theta}) - \log p(\boldsymbol{\theta};\lambda) \right], \\ \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{H}^{-1}\mathbf{g}. \end{array}$$

GENERALIZED LINEAR MODELS

Linear regression and logistic regression are special cases of a family of conditional probability models known in statistics as "generalized linear models," which were proposed to unify and generalize linear and logistic regression. In this

formulation, linear models may be related to a response variable using distributions other than the Gaussian distribution used for linear regression. Generalized linear models can be created for any distribution in the exponential family (Appendix A.2 introduces exponential-family distributions).

In this context, the data can be thought of in terms of response variables y_i and explanatory variables organized as p-dimensional vectors $\mathbf{x}_i$, $i = 1,\ldots, n$. Response variables can be expressed in different ways, ranging from binary to categorical or ordinal data. A model is then defined where the expected value $E[y]$ of the distribution used for the response variable consists of an initial linear prediction $\eta_i = \boldsymbol{\beta}^T \mathbf{x}_i$ using a parameter vector $\boldsymbol{\beta}$, which is then subjected to a smooth, invertible, and potentially nonlinear transformation using the *mean function* g^{-1}:

$$\mu_i = E[y_i] = g^{-1}(\boldsymbol{\beta}^T \mathbf{x}_i).$$

The mean function is the inverse of the *link function* g. In generalized linear modeling the entire set of explanatory variables for all the observations is arranged as an $n \times p$ matrix $\mathbf{X}$, so that a vector of linear predictions for the entire data set is $\boldsymbol{\eta} = \mathbf{X}\boldsymbol{\beta}$. The variance of the underlying distribution can also be modeled; typically as a function of the mean. Different distributions, link functions, and corresponding mean functions give a great deal of flexibility in defining probabilistic models; Table 9.2 shows some examples.

The multiclass extension of logistic regression discussed above is another example of a generalized linear model that uses the multinomial distribution for the response variable y. Because this model is defined in terms of probability distributions, the parameters can be estimated using maximum likelihood techniques.

Since these models are fairly simple, the coefficients β_j are interpretable. Applied statisticians are often interested not only in the estimated values but in other information such as the standard error of the estimates and statistical significance tests.

Table 9.2 Link Functions, Mean Functions, and Distributions Used in Generalized Linear Models

Link Name	Link Function $\eta = \boldsymbol{\beta}^T\mathbf{x} = g(\mu)$	Mean Function $\mu = g^{-1}(\boldsymbol{\beta}^T\mathbf{x}) = g^{-1}(\eta)$	Typical Distribution
Identity	$\eta = \mu$	$\mu = \eta$	Gaussian
Inverse	$\eta = \mu^{-1}$	$\mu = \eta^{-1}$	Exponential
Log	$\eta = \log_e \mu$	$\mu = \exp(\eta)$	Poisson
Log-log	$\eta = -\log(-\log_e \mu)$	$\mu = \exp(-\exp(-\eta))$	Bernoulli
Logit	$\eta = \log_e \frac{\mu}{1-\mu}$	$\mu = \frac{1}{1+\exp(-\eta)}$	Bernoulli
Probit	$\eta = \Phi^{-1}(\mu)$	$\mu = \Phi(\eta)$	Bernoulli

Note: $\Phi(\cdot)$ is the cumulative normal distribution.

MAKING PREDICTIONS FOR ORDERED CLASSES

In many situations class values are categorical but possess a natural ordering. To deal with ordinal class attributes, the class probabilities can be expressed in terms of cumulative distributions, which are then modeled to construct an underlying probability distribution function for each class. To define a model with M ordinal categories, $M-1$ cumulative probability models of the form $P(Y_i \leq j)$ are used for the random variable Y_i that represents the category of a given instance i. Models for $P(Y_i = j)$ are then obtained using differences between the cumulative distribution models. Here we will use complementary cumulative probabilities, known as *survival functions*, of the form $P(Y_i > j) = 1 - P(Y_i \leq j)$, because this sometimes simplifies the interpretation of parameters. Then the class probabilities are obtained by:

$$P(Y_i = 1) = 1 - P(Y_i > 1)$$
$$P(Y_i = j) = P(Y_i > j - 1) - P(Y_i > j)$$
$$P(Y_i = M) = P(Y_i > M - 1).$$

The generalized linear models discussed above can be further generalized to ordinal categorical data. In fact, the general approach can be applied to various model classes by modeling complementary cumulative probabilities with a smooth and invertible link function that transforms them nonlinearly and equates the result to a linear predictor. For binary predictions, the models often take this form:

$$\text{logit}(\gamma_{ij}) = \log \frac{\gamma_{ij}}{1 - \gamma_{ij}} = b_j + \mathbf{w}^T \mathbf{x}_i,$$

where $\mathbf{w}$ is a vector of weights, $\mathbf{x}_i$ are vectors of features, and γ_{ij} represent the probability that example i is greater than the discretized, ordinal category j. Such models are called "proportional odds" models, or "ordered logit" models. The model above uses a different bias for each inequality, but the same set of weights. This guarantees a consistent set of probabilities.

CONDITIONAL PROBABILISTIC MODELS USING KERNELS

Linear models can be transformed into nonlinear ones by applying the "kernel trick" mentioned in Section 7.2 under kernel regression, or, alternatively, through basis expansion as mentioned in the section above on matrix formulations of linear and polynomial regression. This can be applied to both kernel regression and kernel logistic regression.

Suppose the features $\mathbf{x}$ are replaced by the vector $\mathbf{k}(\mathbf{x})$ whose elements are determined using a kernel function $k(\mathbf{x},\mathbf{x}_j)$ for every training example (or for some subset of training vectors):

$$\mathbf{k}(\mathbf{x}) = \begin{bmatrix} k(\mathbf{x}, \mathbf{x}_1) \\ \vdots \\ k(\mathbf{x}, \mathbf{x}_V) \\ 1 \end{bmatrix}$$

A "1" has been appended to this vector to implement the bias term in the parameter matrix. A kernel regression is then

$$p(y|\mathbf{x}) = N(y; \mathbf{w}^T\mathbf{k}(\mathbf{x}), \sigma^2).$$

For classification, an analogous kernel logistic regression is

$$p(\mathbf{y}|\mathbf{x}) = \frac{\exp(\mathbf{y}^T\ \boldsymbol{\theta}\mathbf{k}(\mathbf{x}))}{\sum_{\mathbf{y}} \exp(\mathbf{y}^T\ \boldsymbol{\theta}\mathbf{k}(\mathbf{x}))} = \frac{1}{Z(\mathbf{k}(\mathbf{x}))}\exp(\mathbf{y}^T\boldsymbol{\theta}\mathbf{k}(\mathbf{x})).$$

Since every example in the training set can have a different kernel vector, it is necessary to compute a *kernel matrix* $\mathbf{K}$ with entries given by $\mathbf{K}_{ij} = k(\mathbf{x}_i,\mathbf{x}_j)$. For large datasets this operation can be time- and memory-intensive.

Support vector machines have a similar form, although they are not formulated probabilistically, and multiclass support vector machines are not as easily formulated as multiclass kernel logistic regression. Because they use an underlying hinge loss and weight regularization term, support vector machines often assign zero weights to many of the terms; they also have the appealing feature of placing nonzero weights only on vectors that reside at the boundary of the decision surface. In many applications this results in a significant reduction in the number of kernel evaluations needed at test time. Neither kernel logistic regression nor kernel ridge regression produce such sparse solutions, even when a Gaussian or (squared) L_2 regularization is used: in general there is a nonzero weight for every exemplar. However, kernel logistic regression can outperform support vector machines, and several techniques have been proposed to make these methods sparse. They are mentioned in the *Further Reading* section at the end of the chapter.

9.8 SEQUENTIAL AND TEMPORAL MODELS

Consider the task of creating a probability model for a sequence of observations. If they correspond to words, random variables could be defined with as many states as there are words in the vocabulary. If they are continuous, some parametric form is needed to create suitable continuous distributions.

MARKOV MODELS AND *N*-GRAM METHODS

One simple and effective probabilistic model for discrete sequential data is known as a "Markov model." A first-order Markov model assumes that each symbol in a sequence can be predicted using its conditional probability given the preceding symbol. (For the very first symbol, an unconditional probability is used.) Given observation variables $\mathrm{O} = \{O_1,\ldots,O_T\}$, this can be written

$$P(\mathrm{O}) = P(O_1)\prod_{t=1}^{T} P(O_{t+1}|O_t).$$

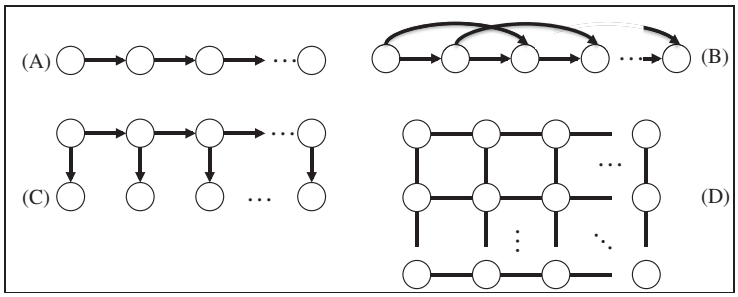

FIGURE 9.18

(A) and (B) First- and second-order Markov models for a sequence of variables; (C) Hidden Markov model; (D) Markov random field.

Usually, every conditional probability used in such models is the same. The corresponding Bayesian networks consist simply of a linear chain of variables with directed edges between each successive pair; Fig. 9.18A shows an example. The method generalizes naturally to second-order models (Fig. 9.18B), and to higher orders. *N*-gram models correspond to the use of an $(N-1)$-th order Markov model. For example, first-order models involve the use of 2-grams (or "bigrams"); third-order models use trigrams, and zero-th order models correspond to single observations or "unigrams." Such models are widely used for modeling biological sequences such as DNA, as well as for text mining and computational linguistics applications.

All probabilistic models raise the issue of what to do when there is no data for certain configurations of variables. Zero-valued parameters cause problems, as we saw in Section 4.2. This is particularly severe with high-order Markov models, and smoothing techniques—such as Laplace or Dirichlet smoothing, which can be derived from a Bayesian analysis—become critical. See the *Further Reading* section for some pointers to some specialized methods for smoothing n-grams.

Large *n*-gram models are extremely useful for applications ranging from machine translation and speech recognition to spelling correction and information extraction. Indeed, Google has made available English language word counts for a billion five-word sequences that appear at least 40 times in a trillion-word corpus After discarding words that appear less than 200 times, there remain 13 million unique words (unigrams), 300 million bigrams; and around a billion each of trigrams, four-grams, and five-grams.

HIDDEN MARKOV MODELS

Hidden Markov models have been widely used for pattern recognition since at least the 1980s. Until recently most of the major speech recognition systems have consisted of large Gaussian mixture models combined with hidden Markov

models. Many problems in biological sequence analysis can also be formulated in terms of hidden Markov models, with various extensions and generalizations.

A hidden Markov model is a joint probability model of a set of discrete observed variables $\mathrm{O} = \{O_1,\ldots,O_T\}$ and discrete hidden variables $\mathrm{H} = \{H_1,\ldots, H_T\}$ for T observations that factors the joint distribution as follows:

$$P(\mathrm{O},\mathrm{H}) = P(H_1)\prod_{t=1}^{T} P(H_{t+1}|H_t)\prod_{t=1}^{T} P(O_t|H_t),$$

Each O_t is a discrete random variable with N possible values, and each H_t is a discrete random variable with M possible values. Fig. 9.18C illustrates a hidden Markov model as a type of Bayesian network that is known as a "dynamic" Bayesian network because variables are replicated dynamically over the appropriate number of time steps. They are an obvious extension of first-order Markov models, and it is common to use "time-homogeneous" models where the transition matrix $P(H_{t+1}|H_t)$ is the same at each time step. Define $\mathbf{A}$ to be a transition matrix whose elements encode $P(H_{t+1} = j|H_t = i)$, and $\mathbf{B}$ to be an emission matrix $\mathbf{B}$ whose elements b_{ij} correspond to $P(O_t = j|H_t = i)$. For the special $t = 1$ the initial state probability distribution is encoded in a vector π with elements $\pi_i = P(H_t = i)$. The complete set of parameters is $\theta = \{\mathbf{A}, \mathbf{B}, \boldsymbol{\pi}\}$, a set containing two matrices and one vector. We write a particular observation sequence as a set of observations $\tilde{\mathrm{O}} = \{O_1 = o_1, \ldots, O_T = o_T\}$.

Hidden Markov models pose three key problems:

1. Compute $P(\tilde{\mathrm{O}};\theta)$, the probability of a sequence under the model with parameters θ.
2. Find the most probable explanation—the best sequence of states $H^* = \{H_1 = h_1, \ldots, H_T = h_T\}$ that explains an observation.
3. Find the best parameters θ for the model given a data set of observed sequences.

The first problem can be solved using the sum-product algorithm, the second using the max-product algorithm, and the third using the EM algorithm for which the required expectations are computed using the sum-product algorithm. If there is labeled data for the sequences of hidden variables that correspond to observed sequences, the required conditional probability distributions can be computed from the corresponding counts in the same way as we did with Bayesian networks.

The only difference between the parameter estimation task when using an hidden Markov model viewed as a dynamic Bayesian network and the updates used for learning the conditional probability tables in a Bayesian network is that one can average over the statistics obtained at each time step, because the same emission and transition matrices are used at each step. The basic hidden Markov model formulation serves as a point of reference when coupling more complex probabilistic models over time using more general dynamic Bayesian network models.

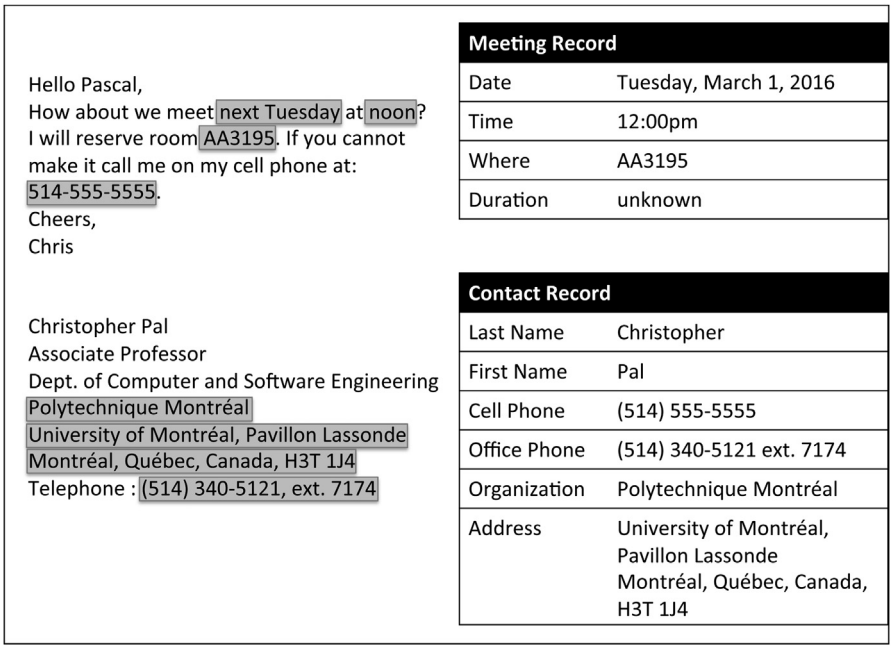

FIGURE 9.19

Mining emails for meeting details.

CONDITIONAL RANDOM FIELDS

Hidden Markov models are not the only models that are useful when working with sequence data. *Conditional random fields* are a statistical modeling technique that takes context into account, and are often structured as linear chains. They are widely used for sequence processing tasks in data mining, but are also popular in image processing and computer vision. Fig. 9.19 illustrates the problem of extracting locations and dates of meetings from email text, which can be addressed using chain-structured conditional random fields. A major advantage is that these models can be given arbitrarily complex features of the input sequence to be used for making predictions—e.g., matching words to lists of organizations and processing variations of known abbreviations.

Inference for chain-structured conditional random fields can be performed efficiently using the sum- and max-product algorithms discussed above. The sum-product algorithm can be used to compute the expected gradients needed to learn them, and the max-product algorithm serves to label new sequences—with tags such as "where" indicating the location or room and "time" labels associated with a meeting.

We begin with a general definition, and then focus on the simpler case of linear conditional random fields. Fig. 9.20D shows a chain-structured conditional

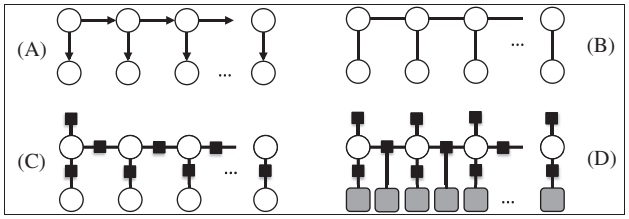

FIGURE 9.20

(A) Dynamic Bayesian network representation of a hidden Markov model; (B) similarly structured Markov random field; (C) factor graph for (A); and (D) factor graph for a linear chain conditional random field.

random field, illustrated as a factor graph, which can be contrasted with the Markov random field depicted in Fig. 9.20B and the hidden Markov model in Fig. 9.20C, also shown as a factor graph. Circles are not used to represent the observed variables in Fig. 9.20D because the underlying conditional random field model does not explicitly encode these as random variables in the graph.

From Markov random fields to conditional random fields

Whereas both Bayesian networks and Markov random fields define a joint probability model for data, conditional random fields (also known as "structured prediction" techniques) define a joint *conditional* distribution for multiple predictions. For a set X of random variables, we have already seen how a Markov random field factorizes the joint distribution for X using an exponentiated energy function $F(X)$:

$$P(X) = \frac{1}{Z}\exp(-F(X)),$$

$$Z = \sum_{X}\exp(-F(X)),$$

where the sum is over all states of all variables in X. Conditional random fields condition on some observations, yielding a conditional distribution:

$$P(Y|X) = \frac{1}{Z(X)}\exp(-F(Y,X)),$$

$$Z(X) = \sum_{Y}\exp(-F(Y,X)),$$

where the sum is over all states of all variables in Y. Both Markov and conditional random fields can be defined for general model structures, but the energy functions usually include just one or two variables—unary and pairwise potentials. Conceptually, to create a conditional random field for $P(Y|X)$ based on U unary and V pairwise functions of variables in Y, the energy function takes the form

$$F(Y,\tilde{X}) = \sum_{u=1}^{U} U(Y_u,\tilde{X}) + \sum_{v=1}^{V} V(Y_v,\tilde{X}).$$

Such energy functions can be transformed into so-called potential functions by negation, exponentiation, and normalization. The conditional probability model takes this form:

$$P(Y|\tilde{X}) = \frac{1}{Z(\tilde{X})}\exp\left[\sum_{u=1}^{U} U(Y_u, \tilde{X}) + \sum_{v=1}^{V} V(Y_v, \tilde{X})\right] = \frac{1}{Z(\tilde{X})}\prod_{u=1}^{U}\phi_u(Y_u, \tilde{X})\prod_{v=1}^{V}\Psi_v(Y_v, \tilde{X}). \tag{9.8}$$

Lattice-structured models like this are used in image processing applications, as mentioned earlier. Processing sequences with chain-structured conditional random fields is even simpler. Note that logistic regression could be thought of as a simple conditional random field, one that arises naturally from a conditional version of a Markov random field with a factorization shown in Fig. 9.15B.

Linear chain conditional random fields

Consider observations $\tilde{Y} = \{y_1 = \tilde{y}_1, \ldots, y_N = \tilde{y}_N\}$ of a sequence of discrete random variables $Y = \{y_1, \ldots, y_N\}$, using integers to encode the relevant states, and an observed input sequence $\tilde{X} = \{\tilde{x}_1, \ldots, \tilde{x}_N\}$, which can be of any data type. A conditional random field defines the conditional probability $P(Y|X)$ of a label sequence given an input sequence. This contrasts with hidden Markov models, which treat both Y and X as random variables and defines a joint probability model for $P(Y, X)$. Note that X above was intentionally not defined as a sequence of random variables because we will define the conditional distribution of Y given X and need not have an explicit model for $P(X)$. Of course, an *implicit* model for $P(X)$ could be defined as the empirical distribution of the data, or the distribution produced by placing a Dirac or Kronecker delta function on each observation and normalizing by the number of examples. Not defining X as a sequence leaves open what type of variables these are, and whether it is just a set of variables or a set of formally defined *random* variables. Fig. 9.20D uses shaded rectangles to emphasize this point. In this chain model, for a given sequence of length N, Eq. (9.8) could be re-written

$$P(Y|\tilde{X}) = \frac{1}{Z(\tilde{X})}\prod_{u=1}^{N}\phi_u(y_u, \tilde{X})\prod_{v=1}^{N}\Psi_v(y_v, y_{v+1}, \tilde{X}).$$

Focusing on a linear chain structure reveals some analogies with the emission and transition matrices of hidden Markov models. There are two types of features: a set of J *single variable* (state) features $u_j(y_i, X, i)$ that are a function of a single y_i, and which are computed for each y_i in the sequence $i = 1, \ldots, N$; and a set of K *pairwise* (transition) features $v_k(y_{i-1}, y_i, X, i)$ for $i > 1$. Each type has associated unary weights θ_j^u and pairwise weights θ_k^v. Note that these features can be a function of the entire observed sequence $\tilde{X}$, or of some subset. This global dependence

on the input is a major advantage of conditional random fields over hidden Markov models. The analog of the Markov model transition matrix are the *per-position pairwise potential functions*: these can be written as a set of matrices that depend upon the observation sequence and consist of a sum over the product of pairwise weights θ_k^v and pairwise features

$$\Psi_i[y_i, y_{i+1}] = \exp\left[\sum_{k=1}^{K} \theta_k^v v_k(y_i, y_{i+1}, \tilde{X}, i)\right].$$

The unary potential functions play a similar role to the terms arising from the hidden Markov model emission matrices, and can be written as the following set of vectors that involve exponentiated weighted combinations of unary features

$$\phi_i[y_i] = \exp\left[\sum_{j=1}^{J} \theta_j^\mu u_j(y_i, \tilde{X}, i)\right].$$

Sometimes, rather than keeping the unary and pairwise features and their parameters separate, it is useful to work with all features for a given position i. To do this, define $\mathbf{f}(y_i, y_{i+1}, X, i)$ as a length-L vector containing all single-variable and pairwise features, and define the global feature vector to be the sum over each of these position dependent feature vectors:

$$\mathbf{g}(Y, X) = \sum_{i=1}^{N} \mathbf{f}(y_i, y_{i+1}, X, i).$$

Now a conditional random field can be written in a particularly compact form

$$P(Y|X) = \frac{\exp(\boldsymbol{\theta}^T \mathbf{g}(Y, X))}{\sum_Y \exp(\boldsymbol{\theta}^T \mathbf{g}(Y, X))}.$$

If we turn to the problem of learning based on maximizing the conditional likelihood for this model, an analogy can be drawn to the simpler case of logistic regression. The gradient of the log-likelihood of a conditional random field for a set of M input sequences $A = \{\tilde{X}_1, \ldots, \tilde{X}_M\}$ and corresponding output sequences $B = \{\tilde{Y}_1, \ldots, \tilde{Y}_M\}$, is

$$\frac{\partial}{\partial \boldsymbol{\theta}} \log P(B|A) = \sum_{m=1}^{M} [\mathbf{g}(\tilde{Y}_m, \tilde{X}_m) - E_m[\mathbf{g}(Y_m, \tilde{X}_m)]].$$

Here, $E_m[.]$ is an expectation taken with respect to $P(Y_m|\tilde{X}_m)$. It is standard practice to include a Gaussian prior or L_2 regularization on parameters by adding a regularization term to this expression. Unlike logistic regression, this expectation involves the joint distribution of the label sequence and not just the distribution for a single label variable. However, in chain-structured graphs it can be computed efficiently and exactly using the sum-product algorithm.

Learning for chain-structured conditional random fields

To compute the gradient for a linear chain conditional random field, it is useful to expose each parameter in the L_2 regularized log-likelihood in scalar form:

$$\log P(B|A) = \sum_{m=1}^{M} \sum_{i=1}^{N_m} \sum_{l=1}^{L} \theta_l g_l(\tilde{y}_{m,i}, \tilde{y}_{m,i+1}, \tilde{X}_m) - \sum_{m=1}^{M} \log Z(\tilde{X}_m) - \sum_{l=1}^{L} \frac{\theta_l^2}{2\sigma^2}.$$

where σ is the regularization parameter. Taking the derivative with respect to a single parameter and training example, the contribution to the gradient of each example is

$$\frac{\partial}{\partial \theta_l} \log P(\tilde{Y}_m|\tilde{X}_m) = \sum_{i=1}^{N_m} g_l(\tilde{y}_i, \tilde{y}_{i+1}, \tilde{X}) - \sum_{i=1}^{N_m} \sum_{y_i} \sum_{y_{i+1}} g_l(y_i, y_{i+1}, \tilde{X}) P(y_i, y_{i+1}|\tilde{X}) - \frac{\theta_l}{\sigma^2}.$$

This is simply the difference between the observed occurrence of the feature and its expectation under the current prediction of the model, taken with respect to $P(y_i, y_{i+1}|\tilde{X})$, minus the partial derivative of the regularization term. Similar terms arise for unary functions, but the expectation is taken with respect to $P(y_i|\tilde{X})$. These distributions are the single- and pairwise-variable marginal conditional distributions, and can be computed efficiently using the sum-product algorithm.

Using conditional random fields for text mining

The text information extraction scenario in Fig. 9.19 is just one example of applying data mining to extract information from natural language. Such information might be other *named entities* such as *locations, personal names, organizations, money, percentages, dates*, and *times*; or fields in a seminar announcement such as *speaker name, seminar room, start time*, and *end time*. In such tasks, input features often consist of current, previous, and next word; character *n*-grams; part-of-speech tag sequences; the presence of certain key words in windows to the left or right of the current position. Other features can be defined using lists of known words, such as first and last names and honorifics; locations and organizations. Features such as capitalization and alphanumeric characters can be defined using regular expressions and integrated into an underlying probabilistic model based on conditional random fields.

9.9 FURTHER READING AND BIBLIOGRAPHIC NOTES

The field of probabilistic machine learning and data mining is enormous: it essentially subsumes all classical and modern statistical techniques. This chapter has focused on foundational concepts and some widely used probabilistic techniques in data mining and machine learning. Excellent books that focus on statistical and probabilistic methods include Hastie, Tibshirani, and Friedman (2009), and

Murphy (2012). Koller and Friedman (2009)'s excellent book specializes in advanced techniques and principles of probabilistic graphical models.

The *K*2 algorithm for learning Bayesian networks was introduced by Cooper and Herskovits (1992). Bayesian scoring metrics are covered by Heckerman et al. (1995). Friedman, Geiger, and Goldszmidt (1997) introduced the tree augmented Naïve Bayes algorithm, and also describe multinets. Grossman and Domingos (2004) show how to use the conditional likelihood for scoring networks. Guo and Greiner (2004) present an extensive comparison of scoring metrics for Bayesian network classifiers. Bouckaert (1995) describes averaging over subnetworks. AODEs are described by Webb, Boughton, and Wang (2005), and AnDEs by Webb et al. (2012). AD trees were introduced and analyzed by Moore and Lee (1998)—the same Andrew Moore whose work on *k*D-trees and ball trees was mentioned in Section 4.10. Komarek and Moore (2000) introduce AD trees for incremental learning that are also more efficient for data sets with many attributes.

The AutoClass program is described by Cheeseman and Stutz (1995). Two implementations have been produced: the original research implementation, written in LISP, and a follow-up public implementation in C that is 10 or 20 times faster but somewhat more restricted—e.g., only the normal distribution model is implemented for numeric attributes. DensiTrees were developed by Bouckaert (2010).

Kernel density estimation is an effective and conceptually simple probabilistic model. Epanechnikov (1969) showed the optimality of the Epanechnikov kernel under the mean-squared error metric. Jones, Marron, and Sheather (1996) recommends using a so-called "plug-in" estimate to select the kernel bandwidth. Duda and Hart (1973) and Bishop (2006) show theoretically that kernel density estimation converges to the true distribution as the amount of data grows.

The EM algorithm, which originates in the work of Dempster, Laird, and Rubin (1977), is the key to learning in hidden or latent variable models. The modern variational view provides a solid theoretical justification for the use of approximate posterior distributions, as discussed in Appendix A.2 and in Bishop (2006). This perspective originated in the 1990s with work by Neal and Hinton (1998), Jordan, Ghahramani, Jaakkola, and Saul (1998), and others. Salakhutdinov, Roweis and Ghahramani (2003) explore the EM approach and compare it with the expected gradient, including the more sophisticated expected conjugate gradient based optimization.

Markov chain Monte Carlo methods are popular in Bayesian statistical modeling; see, e.g., Gilks (2005). Geman and Geman (1984) first described the Gibbs sampling procedure, naming it after the physicist Josiah Gibbs because of the analogy between sampling, the underlying functional forms of random fields and statistical physics. Hastings' (1970) generalization of the Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953) algorithm has been influential in laying the foundation for present-day methods. The *iterated conditional modes* approach for finding an approximate most probable explanation was proposed by Besag (1986).

Plate notation has been widely used in artificial intelligence (Buntine, 1994), machine learning (Blei, Ng, & Jordan, 2003) and computational statistics (Lunn, Thomas, Best, & Spiegelhalter, 2000) to define complex probabilistic graphical models, and forms the basis of the BUGS (Bayesian inference Using Gibbs Sampling) software project (Lunn et al., 2000). Our presentation of factor graphs and the sum-product algorithm follows their origins in Kschischang, Frey, and Loeliger (2001) and Frey (1998). The sum- and max-product algorithms only apply to trees. However, Bayesian networks and other models that contain cycles can be manipulated into a structure known as a *junction tree* by clustering variables, and Lauritzen and Spiegelhalter (1988)'s junction tree algorithm permits exact inference. Ripley (1996) covers the junction tree algorithm, with practical examples; Huang and Darwiche (1996)'s procedural guide is an excellent resource for those who need to implement the algorithm. Probability propagation in a junction tree yields exact results, but is sometimes infeasible because the clusters become too large—in which case one must resort to sampling or variational methods.

Roweis (1998) gives an early EM formulation for PPCA: he examines the zero input noise case and provides the elegant mathematics for the simplified EM algorithm presented above. Tipping and Bishop (1999a, 1999b) give further analysis, and show that after optimizing the model and learning the variance of the observation noise the columns of the matrix $\mathbf{W}$ are scaled and rotated principal eigenvectors of the covariance matrix of the data. The probabilistic formulation of principal component analysis opens the door to probabilistic formulations of further generalizations, such as mixtures of principal component analyzers (Dony & Haykin, 1995; Tipping & Bishop, 1999a, 1999b) and mixtures of factor analyzers (Ghahramani & Hinton, 1996). Of particular utility is the ability of PPCA and factor analysis to easily deal with missing data: providing the data is missing at random one can marginalize over the distribution associated with unobserved values, as detailed by Ilin and Raiko (2010).

PPCA corresponds to factorizing a covariance matrix. Another way to reduce the number of parameters in a continuous Gaussian model is to use sparse inverse covariance models, which, when combined with mixture models and EM, yield another form of clustering with correlated attributes. Edwards (2012) provides a nice introduction to graphical modeling, including mixed models with discrete and continuous components. Unlike any other treatments, he also examines graphical Gaussian models and delves further into the correspondence between a graphical model and the sparsity structure of an inverse covariance matrix. These concepts are better grasped using the "canonical" parameterization of the Gaussian distribution in terms of $\beta = \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}$, and $\Omega = \boldsymbol{\Sigma}^{-1}$, rather than the usual "moment" parameterization that uses the mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$.

LSA was introduced by Deerwester, Dumais, Landauer, Furnas, and Harshman (1990). pLSA has its origins in Hofmann (1999). Latent Dirichlet allocation (LDA^b) was proposed in Blei et al (2003). The highly effective "collapsed Gibbs sampling" approach for LDA^b was proposed by Teh et al.

(2006), who also extended the concept to variational methods. Rather than applying LDA[b] naïvely when looking for trends over time, Blei and Lafferty (2006)'s *dynamic topic models* treat the temporal evolution of topics explicitly; they examined topical trends in the journal *Science*. Griffiths and Steyvers (2004) used Bayesian model selection to determine the number of topics in their LDA[b] analysis of *Proceedings of the National Academy of Science* abstracts. Griffiths and Steyvers (2004) and Teh et al. (2006) give more details on the collapsed Gibbs sampling and variational approaches to LDA[b]. Hierarchical Dirichlet processes (Teh et al., 2006) and related techniques offer alternatives to the problem of determining the number of topics or clusters in hierarchical Bayesian models. These technically sophisticated methods are quite popular, and high-quality implementations are available online.

Logistic regression is sometimes referred to as the workhorse of applied statistics; Hosmer and Lemeshow (2004) is a useful resource. Nelder and Wedderburn (1972)'s work led to the generalized linear modeling framework. McCullagh (1980) developed proportional odds models for ordinal regression, which are sometimes called ordered logit models because they use the generalized logit function. Frank and Hall (2001) showed how to adapt arbitrary machine learning techniques to ordered predictions. McCullagh and Nelder (1989)'s widely cited monograph is another good source for additional details on the framework of generalized linear models.

Tibshirani (1996) developed the famous "Least Absolute Shrinkage and Selection Operator," also known as the LASSO; while Zou and Hastie (2005) developed the "elastic net" regularization approach.

Kernel logistic regression transforms a linear classifier into a nonlinear one, and probabilistic sparse kernel techniques are attractive alternatives to support vector machines. Tipping (2001) proposed a "relevance vector machine" that manipulates priors on parameters in a way that encourages kernel weights to become zero during learning. Lawrence, Seeger, and Herbrich (2003) proposed an "informative vector machine," which tackles the problem with a fast, sparse Gaussian process method in the sense of Williams and Rasmussen (2006). Zhu and Hastie (2005) formulated sparse kernel logistic regression as an "import vector machine" that uses greedy search methods. However, none of these approach the popularity of Cortes and Vapnik (1995)'s support vector machines, perhaps because their objective functions are not convex, in contrast to that of an SVM (and L_2 regularized kernel logistic regression). Convex optimization problems have a single minimum in the loss function (maximum in the likelihood). When probabilities are needed from an SVM, Platt (1999) shows how to fit a logistic regression to the classification scores.

Some techniques for smoothing *n*-grams arise from applying prior distributions to the parameters of the model's conditional probabilities. Others come from different perspectives—such as interpolation techniques, where weighted combinations of lower order *n*-grams are used. Good-Turing discounting (Good, 1953) (coinvented by Alan Turing, one of the fathers of computing), and Witten−Bell

smoothing (Witten & Bell, 1991) are based on these ideas. Brants and Franz (2006) discuss the massive Google *n*-gram collections mentioned in Section 9.8; they are available as a 24 GB compressed text file from the Linguistic Data Consortium.

Rabiner and Juang (1986) and Rabiner (1989) give a classic introduction and tutorial respectively on Hidden Markov Models. They have been used extensively for decades in speech recognition systems, and are widely applicable to many other problems. The human genome sequencing project stretched from around 1990 to the early 2000s (International Human Genome Sequencing Consortium, 2001; Venter et al., 2001), and spawned a surge of activity in recognizing and modeling genes in genomes using hidden Markov models (Burge & Karlin, 1997; Kulp, Haussler, Rees, & Eeckman, 1996)—a particularly impressive and important application of data mining. Murphy (2002) is an excellent source for details on how dynamic Bayesian networks extend hidden Markov models.

Lafferty, McCallum, and Pereira (2001) is a seminal paper on conditional random fields; Sutton and McCallum (2006) is an excellent source of further details. Sha and Pereira (2003) present the global feature vector view of conditional random fields. Our presentation synthesizes these perspectives. The original application was to sequence labeling problems, but they have since become widely used for many sequence processing tasks in data mining. Kristjansson, Culotta, Viola, and McCallum (2004) examined the specific problem of extracting information from email text. The Stanford Named Entity Recognizer is based on conditional random fields; Finkel, Grenager, and Manning (2005) give details of the implementation.

Markov logic networks (Richardson & Domingos, 2006) provide a way to create dynamically instantiated Markov random fields from programs encoded using weighted clauses in first-order logic. This approach has been used for collective or structured classification, link or relationship prediction, entity and identity disambiguation, among many others, as described in Domingos and Lowd (2009)'s textbook.

SOFTWARE PACKAGES AND IMPLEMENTATIONS

Implementations of principal component analysis, Gaussian mixture models, and hidden Markov models are widely available in many software packages. MatLab's statistics toolbox, e.g., has implementations of principal component analysis and its probabilistic variant based on the methods we have discussed, and also contains implementations of Gaussian mixture models and all the canonical hidden Markov model manipulations.

Kevin Murphy's MatLab-based Probabilistic Modeling Toolkit is a large, open source collection of MatLab functions and tools. The software contains implementations for many of the methods we have discussed here, including code for Bayesian network manipulation and inference methods.

The Hugin software package from Hugin Expert A/S and the Netica software from Norsys are well-known commercial software implementations for manipulating Bayesian networks. They contain excellent graphical user interfaces for interacting with these networks.

The BUGS (Bayesian inference Using Gibbs Sampling) project has created a variety of software packages for the Bayesian analysis of complex statistical models using Markov chain Monte Carlo methods. WinBUGS (Lunn et al., 2000) is a stable version of the software, but the more recent OpenBUGS project is an open source version of the core BUGS implementation (Lunn, Spiegelhalter, Thomas, & Best, 2009).

The VIBES software package (Bishop, Spiegelhalter, & Winn, 2002) allows inference in graphical models using variational methods. Microsoft Research has created a programming language known as infer.net that allows one to define graphical models and perform inference in them using variational methods, Gibbs sampling or another message passing method known as expectation propagation (Minka, 2001). Expectation propagation generalizes belief propagation to distributions and models beyond the typical discrete and binary models used to create Bayesian networks. John Winn and Tom Minka at Microsoft Research have been leading the infer.net project.

The R programming language and software environment was created for statistical computing and visualization (Ihaka & Gentleman, 1996). It has its origins at the University of Auckland, and provides an open source implementation of the S programming language from Bell Labs. It is comparable to well known commercial packages such as SAS, SPSS, and Stata, and contains implementations of many classical statistical methods such as generalized linear models and other regression techniques. Since it is a general purpose programming language there are many extensions and implementations of the models discussed in this chapter available online. Historically Brian D. Ripley has overseen the development of R. Ripley is now retired from Oxford University where he was a statistic Professor. He is the coauthor of a number of books on S programming (Venables & Ripley, 2000, 2002) and an older but very high-quality textbook on pattern recognition and neural networks (Ripley, 1996).

The MALLET Machine Learning for Language Toolkit (McCallum, 2002) provides excellent Java implementations of latent Dirichlet allocation and conditional random fields. It also provides many other statistical natural language-processing methods ranging from document classification and clustering to topic modeling, information extraction, and other machine learning techniques frequently used for text processing.

The open source Alchemy software package is widely used for Markov logic networks (Richardson & Domingos, 2006).

Scikit-learn (Pedregosa et al., 2011) is a rapidly growing Python-based set of implementations of many machine learning methods. It contains implementations of many probabilistic and statistical methods for classification, regression, clustering, dimensionality reduction (including factor analysis and PPCA), model selection and preprocessing.

9.10 WEKA IMPLEMENTATIONS

- Bayesian networks
  - *BayesNet* (Bayesian networks without hidden variables for classification)
  - *A1DE* and *A2DE* (in the *AnDE* package)
- Conditional probability models
  - *LatentSemanticAnalysis* (in the *latentSemanticAnalysis* package)
  - *ElasticNet* (in the *elasticNet* package)
  - *KernelLogisticRegression* (in the *kernelLogisticRegression* package)
- Clustering
  - *EM* (clustering and density estimation using the EM algorithm)