8 *Search and Optimization Methods*

8.1 Introduction

In chapter 6 we saw that there are broad classes of *model structures* or *representations* that can be used to represent knowledge in structured form. Subsequently, in chapter 7 we discussed the principles of how such structures (in the form of models and patterns) can be *scored* in terms of how well they match the observed data. This chapter focuses on the *computational methods* used for model and pattern-fitting in data mining algorithms; that is, it focuses on the procedures for searching and optimizing over parameters and structures guided by the available data and our score functions. The importance of effective search and optimization is often underestimated in the data mining, statistical and machine learning algorithm literatures, but successful applications in practice depend critically on such methods.

We recall that a *score function* is the function that numerically expresses our preference for one model or pattern over another. For example, if we are using the sum of squared errors, S_{SSE}, we will prefer models with lower S_{SSE}—this measures the error of our model (at least on the training data). If our algorithm is searching over multiple models with different representational power (and different complexities), we may prefer to use a penalized score function such as S_{BIC} (as discussed in chapter 7) whereby more complex models are penalized by adding a penalty term related to the number of parameters in the model.

Regardless of the specific functional form of our score function S, once it has been chosen, our goal is to optimize it. (We will usually assume without loss of generality in this chapter that we wish to *minimize* the score function, rather than maximize it). So, let $S(\theta|D, M) = S(\theta_1, \ldots, \theta_d|D, M)$ be the score function. It is a scalar function of a d-dimensional parameter vector θ and a model structure M (or a pattern structure ρ), conditioned on a specific set of observed data D.

This chapter examines the fundamental principles of how to go about finding the values of parameter(s) that minimize a general score function S. It is useful in practical terms, although there is no high-level conceptual difference, to distinguish between two situations, one referring to parameters that can only take discrete values (discrete parameters) and the other to parameters that can take values from a continuum (continuous parameters).

Examples of discrete parameters are those indexing different classes of models (so that 1 might correspond to trees, 2 to neural networks, 3 to polynomial functions, and so on) and parameters that can take only integral values (for example, the number of variables to be included in a model). The second example indicates the magnitude of the problems that can arise. We might want to use, as our model, a regression model based on a subset of variables chosen from a possible p variables. There are $K = 2^p$ such subsets, which can be very large, even for moderate p. Similarly, in a pattern context, we might wish to examine patterns that are probabilistic rules involving some subset of p binary variables expressed as a conjunction on the left-hand side (with a fixed right-hand side). There are $J = 3^p$ possible conjunctive rules (each variable takes value 1, 0, or is not in the conjunction at all). Once again, this can easily be an astronomically large number. Clearly, both of these examples are problems of *combinatorial optimization,* involving *searching* over a set of possible solutions to find the one with minimum score.

Examples of continuous parameters are a parameter giving the mean value of a distribution or a parameter vector giving the centers of a set of clusters into which the data set has been partitioned. With continuous parameter spaces, the powerful tools of differential calculus can be brought to bear. In some special but very important special cases, this leads to closed form solutions. In general, however, these are not possible and iterative methods are needed. Clearly the case in which the parameter vector θ is unidimensional is very important, so we shall examine this first. It will give us insights into the multidimensional case, though we will see that other problems also arise in this situation. Both unidimensional and multidimensional situations can be complicated by the existence of local minima: parameter vectors with values smaller than any other similar vectors, but are not the smallest values that can be achieved. We shall explore ways in which such problems can be overcome.

Very often, the two problems of searching over a set of possible model structures and optimizing parameters within a given model go hand in hand;

that is, since any single model or pattern structure typically has unknown parameters then, as well as finding the best model or pattern structure, we will also have to find the best parameters for each structure considered during the search. For example, consider the set of models in which $\hat{y}$ is predicted as a simple linear combination of some subset of the three predictor variables x_1, x_2, and x_3. One model would be $\hat{y}(i) = ax_1(i) + bx_2(i) + cx_3(i)$, and others would have the same form but merely involving pairs of the predictor variables or single predictor variables. Our search will have to roam over all possible subsets of the x_j variables, as noted above, but for each subset, it will also be necessary to find the values of the parameters (a, b, and c in the case with all three variables) that minimize the score function.

This description suggests that one possible design choice, for algorithms that minimize score functions over both model structures and parameter estimates, is to nest a loop for the latter in a loop for the former. This is often used since it is relatively simple, though it is not always the most efficient approach from a computational viewpoint.

It is worth remarking at this early stage that in some data mining algorithms the focus is on finding *sets* of models, patterns, or regions within parameter space, rather than just the single *best* model, pattern, or parameter vector, according to the chosen score function. This occurs, for example, in Bayesian averaging techniques and in searching for sets of patterns. Usually (although, as always, there are exceptions) in such frameworks similar general principles of search and optimization will arise as in the single model/pattern/parameter case and, so in the interests of simplicity of presentation we will focus primarily on the problem of finding the single best model, pattern, and/or parameter-vector.

Section 2 focuses on general search methods for situations where there is no notion of continuity in the model space or parameter space being searched. This section includes discussion of the combinatorial problems that typically prevent exhaustive examination of all solutions, the general state-space representation for search problems, discussion of particular search strategies, as well as methods such as branch and bound that take advantage of properties of the parameter space or score function to reduce the number of parameter vectors that must be explicitly examined. Section 3 turns to optimization methods for continuous parameter spaces, covering univariate and multivariate cases, and problems complicated by constraints on the permissible parameter values. Section 4 describes a powerful class of methods that apply to problems that involve (or can usefully be regarded as involving) missing values. In many data mining situations, the data sets are so large that multiple passes through the data have to be avoided. Section 5 describes algorithms aimed at this. Finally, since many problems involve score functions that have multiple minima (and maxima), stochastic search methods have been developed to improve the chances of finding the global optimum (and not merely a rather poor local optimum). Some of these are described in section 6.

8.2 Searching for Models and Patterns

8.2.1 Background on Search

This subsection discusses some general high level issues of search. In many practical data mining situations we will not know ahead of time what particular model structure M or pattern structure ρ is most appropriate to solve our task, and we will search over a *family* of model structures $\mathcal{M} = \{M_1, \ldots, M_K\}$ or pattern structures $\mathcal{P} = \{\rho_1, \ldots, \rho_J\}$. We gave some examples of this earlier: finding the best subset of variables in a linear regression problem and finding the best set of conditions to include in the left-hand side of a conjunctive rule. Both of these problems can be considered "best subsets" problems, and have the general characteristic that a combinatorially large number of such solutions can be generated from a set of p "components" (p variables in this case). Finding "best subsets" is a common problem in data mining. For example, for predictive classification models in general (such as nearest neighbor, naive Bayes, or neural network classifiers) we might want to find the subset of variables that produces the lowest classification error rate on a validation data set.

A related model search problem, that we used as an illustration earlier in chapter 5, is that of finding the best tree-structured classifier from a "pool" of p variables. This has even more awesome combinatorial properties. Consider the problem of searching over all possible binary trees (that is, each internal node in the tree has two children). Assume that all trees under consideration have depth p so that there are p variables on the path from the root node to any leaf node. In addition, let any variable be eligible to appear at any node in the tree, remembering that each node in a classification tree contains a test on a single variable, the outcomes of which define which branch is taken from that node. For this family of trees there are on the order of p^{2^p} different tree structures—that is, p^{2^p} classification trees that differ from each other in the specification of at least one internal node. In practice, the number of possible tree structures will in fact be larger since we also want to consider various subtrees of the full-depth trees. Exhaustive search over all possible trees is clearly infeasible!

We note that from a purely mathematical viewpoint one need not necessarily distinguish between different model structures in the sense that all such model structures could be considered as special cases of a single "full" model, with appropriate parameters set to zero (or some other constant that is appropriate for the model form) so that they disappear from the model. For example, the linear regression model $y = ax_1 + b$ is a special case of $y = ax_1 + cx_2 + dx_3 + b$ with $c = d = 0$. This would reduce the model structure search problem to the type of parameter optimization problem we will discuss later in this chapter. Although mathematically correct, this viewpoint is often not the most useful way to think about the problem, since it can obscure important structural information about the models under consideration.

In the discussion that follows we will often use the word *models* rather than the phrase *models or patterns* to save repetition, but it should be taken as referring to both types of structure: the same general principles that are outlined for searching for models are also true for the problem of searching for patterns.

Some further general comments about search are worth making here:

- We noted in the opening section that finding the model or structure with the optimum score from a family $\mathcal{M}$ necessarily involves finding the best parameters θ_k for each model structure M_k within that family. This means that, conceptually and often in practice, a nested loop search process is needed, in which an optimization over parameter values is nested within a search over model structures.

- As we have already noted, there is typically no notion of the score function S being a "smooth" function in "model space," and thus, many of the traditional optimization techniques that rely on smoothness information (for example, gradient descent) are not applicable. Instead we are in the realm of *combinatorial optimization* where the underlying structure of the problem is inherently discrete (such as an index over model structures) rather than a continuous function. Most of the combinatorial optimization problems that occur in data mining are inherently intractable in the sense that the only way to guarantee that one will find the best solution is to visit all possible solutions in an exhaustive fashion.

- For some problems, we will be fortunate in that we will not need to perform a full new optimization of parameter space as we move from one model structure to the next. For example, if the score function is *decomposable,* then the score function for a new structure will be an additive function of the score function for the previous structure as well as a term accounting for the change in the structure. For example, adding or deleting an internal node in a classification tree only changes the score for data points belonging to the subtree associated with that node. However, in many cases, changing the structure of the model will mean that the old parameter values are no longer optimal in the new model. For example, suppose that we want to build a model to predict y from x based on two data points $(x, y) = (1, 1)$ and $(x, y) = (3, 3)$. First let us try very simple models of the form $y = a$, that is y is a constant (so that all our predictions are the same). The value of a that minimizes the sum of squared errors $(1 - a)^2 + (3 - a)^2$ is 2. Now let us try the more elaborate model $y = bx + a$. This adds an extra term into the model. Now the values of a and b that minimize the sum of squared errors (this is a standard regression problem, although a particularly simple example) are, respectively, 0 and 1. We see that the estimate of a depends upon what else is in the model. It is possible to formalize the circumstances in which changing the model will leave parameter estimates unaltered, in terms of *orthogonality* of the data. In general, it is clearly useful to know when this applies, since much faster algorithms can then be developed (for example, if variables are orthogonal in a regression case, we can just examine them one at a time). However, such situations tend to arise more often in the context of designed experiments than in the secondary data occurring in data mining situations. For this reason, we will not dwell on this issue here.

For linear regression, parameter estimation is not difficult and so it is straightforward (if somewhat time-consuming) to recalculate the optimal parameters for each model structure being considered. However, for more complex models such as neural networks, parameter optimization can be both computationally demanding as well as requiring careful "tuning" of the optimization method itself (as we will see later in this chapter). Thus, the "inner loop" of the model search algorithm can be quite taxing computationally. One way to ease the problem is to leave the existing parameters in the model fixed to their previous values and to estimate only the values of parameters added to the model. Although this strategy is clearly suboptimal, it permits a trade-off between highly accurate parameter estimation of just a few models or approximate parameter estimation of a much larger set of models.

- Clearly for the best subsets problem and the best classification tree problem, exhaustive search (evaluating the score function for all candidate models in the model family $\mathcal{M}$) is intractable for any nontrivial values of p since there are 2^p and p^{2^p} models to be examined in each case. Unfortunately, this combinatorial explosion in the number of possible model and pattern structures will be the norm rather than the exception for many data mining problems involving search over model structure. Thus, without even taking into account the fact that for each model one may have to perform some computationally complex parameter optimization procedure, even simply enumerating the models is likely to become intractable for large p. This problem is particularly acute in data mining problems involving very high-dimensional data sets (large p).

- Faced with inherently intractable problems, we must rely on what are called *heuristic search* techniques. These are techniques that experimentally (or perhaps provably on average) provide good performance but that cannot be guaranteed to provide the best solution always. The *greedy heuristic* (also known as *local improvement*) is one of the better known examples. In a model search context, greedy search means that, given a current model M_k we look for other models that are "near" M_k (where we will need to define what we mean by "near") and move to the best of these (according to our score function) if indeed any are better than M_k.

8.2.2 The State-Space Formulation for Search in Data Mining

A general way to describe a search algorithm for discrete spaces is to specify the problem as follows:

1. **State Space Representation:** We view the search problem as one of moving through a discrete set of states. For model search, each model structure M_k consists of a state in our state space. It is conceptually useful to think of each state as a vertex in a graph (which is potentially very large). An abstract definition of our search problem is that we start at some particular node (or state), say M_1, and wish to move through the state space to find the node corresponding to the state that has the highest score function.

2. **Search Operators:** Search operators correspond to legal "moves" in our search space. For example, for model selection in linear regression the operators could be defined as either adding a variable to or deleting a

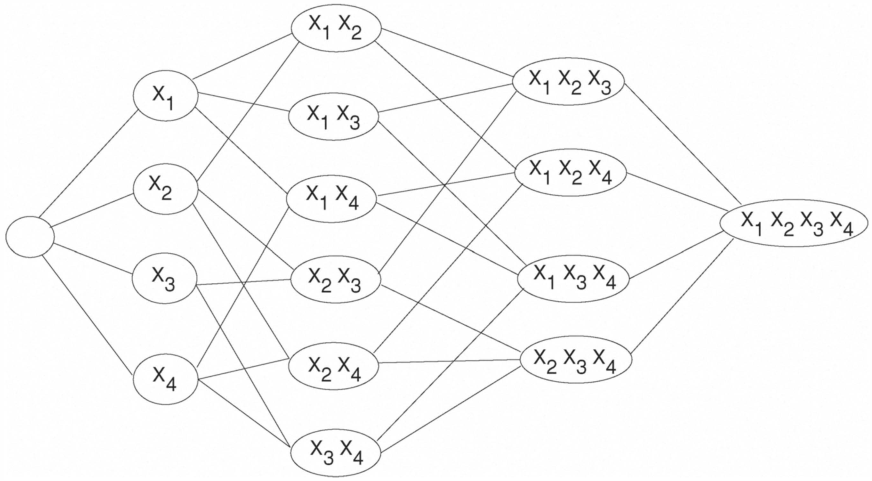

Figure 8.1 An example of a simple state-space involving four variables X_1, X_2, X_3, X_4. The node on the left is the null set—i.e., no variables in the model or pattern.

variable from the current model. The search operators can be thought of as defining directed edges in the state space graph. That is, there is a directed edge from state M_i to M_j if there is an operator that allows one to move from one model structure M_i to another model structure M_j.

A simple example will help illustrate the concept. Consider the general problem of selecting the best subset from p variables for a particular classification model (for example, the nearest neighbor model). Let the score function be the cross-validated classification accuracy for any particular subset. Let M_k denote an individual model structure within the general family we are considering, namely all $K = 2^p - 1$ different subsets containing at least one variable. Thus, the state-space has $2^p - 1$ states, ranging from models consisting of subsets of single variables $M_1 = \{x_1\}, M_2 = \{x_2\}, \ldots$ all the way through to the full model with all p variables, $M_K = \{x_1, \ldots, x_p\}$. Next we define our operators. For subset selection it is common to consider simple operators such as adding one variable at a time and deleting one variable at a time. Thus, from any state with p' variables (model structure) there are two "directions" one can "move" in the model family: add a variable to move to a state with $p' + 1$ variables, or delete a variable to move to a state with $p' - 1$ variables (figure 8.1 shows a state-space for subset selection for 4 variables with these two operators). We can easily generalize these operators to adding or deleting r variables at a time. Such "greedy local" heuristics are embedded in many data mining algorithms. Search algorithms using this idea vary in terms of what state they start from: *forward selection* algorithms work "forward" by starting with a minimally sized model and iteratively adding variables, whereas *backward selection* algorithms work in reverse from the full model. Forward selection is often the only tractable option in practice when p is very large since working backwards may be computationally impractical.

It is important to note that by representing our problem in a state-space with limited connectivity we have *not* changed the underlying intractability of the general model search problem. To find the optimal state it will still be necessary to visit all of the exponentially many states. What the state-space/operator representation does is to allow us to define systematic methods for *local exploration* of the state-space, where the term "local" is defined in terms of which states are adjacent in the state-space (that is, which states have operators connecting them).

8.2.3 A Simple Greedy Search Algorithm

A general iterative greedy search algorithm can be defined as follows:

1. **Initialize:** Choose an initial state $M^{(0)}$ corresponding to a particular model structure M_k.

2. **Iterate:** Letting $M^{(i)}$ be the current model structure at the ith iteration, evaluate the score function at all possible adjacent states (as defined by the operators) and move to the best one. Note that this evaluation can consist of performing parameter estimation (or the change in the score function) for each neighboring model structure. The number of score function evaluations that must be made is the number of operators that can be applied to the current state. Thus, there is a trade-off between the number of operators available and the time taken to choose the next model in state-space.

3. **Stopping Criterion:** Repeat step 2 until no further improvement can be

attained in the local score function (that is, a local minimum is reached in state-space).

4. **Multiple Restarts:** (optional) Repeat steps 1 through 3 from different initial starting points and choose the best solution found.

This general algorithm is similar in spirit to the local search methods we will discuss later in this chapter for parameter optimization. Note that in step 2 that we must explicitly evaluate the effect of moving to a neighboring model structure in a discrete space, in contrast to parameter optimization in a continuous space where we will often be able to use explicit gradient information to determine what direction to move. Step 3 helps avoid ending at a local minimum, rather than the global minimum (though it does not guarantee it, a point to which we return later). For many structure search problems, greedy search is provably suboptimal. However, in general it is a useful heuristic (in the sense that for many problems it will find quite good solutions on average) and when repeated with multiple restarts from randomly chosen initial states, the simplicity of the method makes it quite useful for many practical data mining applications.

8.2.4 Systematic Search and Search Heuristics

The generic algorithm described above is often described as a "hill-climbing" algorithm because (when the aim is to maximize a function) it only follows a single "path" in state-space to a local maximum of the score function. A more general (but more complex) approach is to keep track of multiple models at once rather than just a single current model. A useful way to think about this approach is to think of a *search tree,* a data structure that is dynamically constructed as we search the state-space to keep track of the states that we have visited and evaluated. (This has nothing to do with classification trees, of course.) The search tree is not equivalent to the state-space; rather, it is a representation of how a particular search algorithm moves through a state-space.

An example will help to clarify the notion of a search tree. Consider again the problem of finding the best subset of variables to use in a particular classification model. We start with the "model" that contains no variables at all and predicts the value of the most likely class in the training data as its prediction for all data points. This is the *root node* in the search tree. Assume that we have a forward-selection algorithm that is only allowed to add variables one at a time. From the root node, there are p variables we can add to

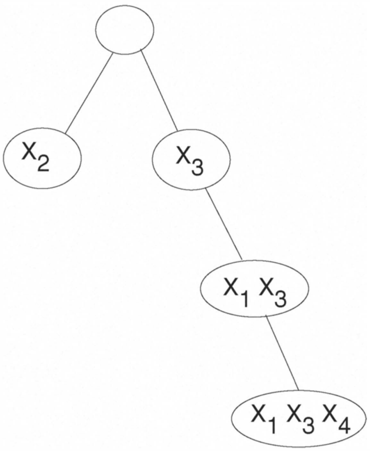

Figure 8.2 An example of a simple search tree for the state-space of figure 8.1.

the model with no variables, and we can represent these p new models as p children of the original root node. In turn, from each of these p nodes we can add p variables, creating p children for each, or p^2 in total (clearly, $p^2 - \binom{p}{2}$ are redundant, and in practice we need to implement a duplicate-state detection scheme to eliminate the redundant nodes from the tree).

Figure 8.2 shows a simple example of a search tree for the state space of figure 8.1. Here the root node contains the empty set (no variables) and only the two best states so far are considered at any stage of the search. The search algorithm (at this point of the search) has found the two best states (as determined by the score function) to be X_2 and X_1, X_3, X_4.

Search trees evolve dynamically as we search the state-space, and we can imagine (hypothetically) keeping track of all of the leaf nodes (model structures) as candidate models for selection. This quickly becomes infeasible since at depth k in the tree there will be p^k leaf nodes to keep track of (where the root node is at depth zero and we have branching factor p). We will quickly run out of memory using this brute-force method (which is essentially *breadth-first* search of the search tree). A memory-efficient alternative is *depth-first* search, which (as its name implies) explores branches in the search tree to some maximum depth before backing up and repeating the depth-first search in a recursive fashion on the next available branch.

Both of these techniques are examples of *blind search*, in that they simply order the nodes to be explored lexicographically rather than by the score function. Typically, improved performance (in the sense of finding higher quality models more quickly) can be gained by exploring the more promising nodes first. In the search tree this means that the leaf node with the highest score is the one whose children are next considered; after the children are added as leaves, the new leaf with the highest score is examined. Again, this strategy can quickly lead to many more model structures (nodes in the tree) being generated than we will be feasibly able to keep in memory. Thus, for example, one can implement a *beam search* strategy that uses a *beam width* of size b to "track" only the b best models at any point in the search (equivalently to only keep track of the b best leaves on the tree). (In figure 8.2 we had $b = 2$.) Naturally, this might be suboptimal if the only way to find the optimal model is to first consider models that are quite suboptimal (and thus, might be outside the "beam"). However, in general, beam search can be quite effective. It is certainly often much more effective than simple hill-climbing, which is similar to depth-first search in the manner in which it explores the search tree: at any iteration there is only a single model being considered, and the next model is chosen as the child of the current model with the highest score.

8.2.5 Branch-and-Bound

A related and useful idea in a practical context is the notion of *branch-and-bound*. The general idea is quite simple. When exploring a search tree, and keeping track of the best model structure evaluated so far, it may be feasible to calculate analytically a lower bound on the best possible score function from a particular (as yet unexplored) branch of the search tree. If this bound is greater than the score of the best model so far, then we need not search this subtree and it can be pruned from further consideration. Consider, for example, the problem of finding the best subset of k variables for classification from a set of p variables where we use the training set error rate as our score function. Define a tree in which the root node is the set of all p variables, the immediate child nodes are the p nodes each of which have a single variable dropped (so they each have $p - 1$ variables), the next layer has two variables dropped (so there are $\binom{p}{2}$ unique such nodes, each with $p - 2$ variables), and so on down to the $\binom{p}{k}$ leaves that each contain subsets of k variables (these are the candidate solutions). Note that the training set error rate cannot *decrease* as we work down any branch of the tree, since lower nodes are based on fewer variables.

Now let us begin to explore the tree in a depth-first fashion. After our depth-first algorithm has descended to visit one or more leaf nodes, we will have calculated scores for the models (leaves) corresponding to these sets of k variables. Clearly the smallest of these is our best candidate k-variable model so far. Now suppose that, in working down some other branch of the tree, we encounter a node that has a score larger than the score of our smallest k-variable node so far. Since the score cannot decrease as we continue to work down this branch, there is no point in looking further: nodes lower on this branch cannot have smaller training set error rate than the best k-variable solution we have already found. We can thus save the effort of evaluating nodes further down this branch. Instead, we back up to the nearest node above that contained an unexplored branch and begin to investigate that. This basic idea can be improved by ordering the tree so that we explore the most promising nodes first (where "promising" means they are likely to have low training set error rate). This can lead to even more effective pruning. This type of general branch and bound strategy can significantly improve the computational efficiency of model search. (Although, of course, it is not a guaranteed solution—many problems are too large even for this strategy to provide a solution in a reasonable time.)

These ideas on searching for model structure have been presented in a very general form. More effective algorithms can usually be designed for specific model structures and score functions. Nonetheless, general principles such as iterative local improvement, beam search, and branch-and-bound have significant practical utility and recur commonly under various guises in the implementation of many data mining algorithms.

8.3 Parameter Optimization Methods

8.3.1 Parameter Optimization: Background

Let $S(\theta) = S(\theta|D, M)$ be the score function we are trying to optimize, where θ are the parameters of the model. We will usually suppress the explicit dependence on D and M for simplicity. We will now assume that the model structure M is fixed (that is, we are temporarily in the inner loop of parameter estimation where there may be an outer loop over multiple model structures). We will also assume, again, that we are trying to minimize S, rather than maximize it. Notice that S and $g(S)$ will be minimized for the same value of θ if g is a monotonic function of S (such as $\log S$).

In general θ will be a d-dimensional vector of parameters. For example, in a regression model θ will be the set of coefficients and the intercept. In a tree model, θ will be the thresholds for the splits at the internal nodes. In an artificial neural network model, θ will be a specification of the weights in the network.

In many of the more flexible models we will consider (neural networks being a good example), the dimensionality of our parameter vector can grow very quickly. For example, a neural network with 10 inputs and 10 hidden units and 1 output, could have $10 \times 10 + 10 = 110$ parameters. This has direct implications for our optimization problem, since it means that in this case (for example) we are trying to find the minimum of a nonlinear function in 110 dimensions.

Furthermore, the shape of this potentially high-dimensional function may be quite complicated. For example, except for problems with particularly simple structure, S will often be multimodal. Moreover, since $S = S(\theta|D, M)$ is a function of the observed data D, the precise structure of S for any given problem is data-dependent. In turn this means that we may have a completely different function S to optimize for each different data set D, so that (for example) it may be difficult to make statements about how many local minima S has in the general case.

As discussed in chapter 7, many commonly used score functions can be written in the form of a sum of local error functions (for example, when the training data points are assumed to be independent of each other):

$$S(\theta) = \sum_{i=1}^{N} e\Big(y(i), \hat{y}_\theta(i)\Big) \tag{8.1}$$

where $\hat{y}_\theta(i)$ is our model's estimate of the target value $y(i)$ in the training data, and e is an error function measuring the distance between the model's prediction and the target (such as square error or log-likelihood). Note that the complexity in the functional form S (as a function of θ) can enter both through the complexity of the model structure being used (that is, the functional form of $\hat{y}$) and also through the functional form of the error function e. For example, if $\hat{y}$ is linear in θ and e is defined as squared error, then S will be quadratic in θ, making the optimization problem relatively straightforward since a quadratic function has only a single (global) minimum or maximum. However, if $\hat{y}$ is generated by a more complex model or if e is more complex as a function of θ, S will not necessarily be a simple smooth function of θ with a single easy-to-find extremum. In general, finding the parameters

θ that minimize $S(\theta)$ is usually equivalent to the problem of minimizing a complicated function in a high-dimensional space.

Let us define the gradient function of S as

$$\mathrm{g}(\theta) = \nabla_{\theta} S(\theta) = \left(\frac{\partial S(\theta)}{\partial \theta_1}, \frac{\partial S(\theta)}{\partial \theta_2}, \ldots, \frac{\partial S(\theta)}{\partial \theta_d} \right), \tag{8.2}$$

which is a d-dimensional vector of partial derivatives of S evaluated at θ. In general, $\nabla_{\theta} S(\theta) = 0$ is a necessary condition for an extremum (such as a minumum) of S at θ. This is a set of d simultaneous equations (one for each partial derivative) in d variables. Thus, we can search for solutions θ (that correspond to extrema of $S(\theta)$) of this set of d equations.

We can distinguish two general types of parameter optimization problems:

1. The first is when we can solve the minimization problem in *closed form*. For example, if $S(\theta)$ is quadratic in θ, then the gradient $\mathrm{g}(\theta)$ will be linear in θ and the solution of $\nabla S(\theta) = 0$ involves the solution of a set of d linear equations. However, this situation is the exception rather than the rule in practical data mining problems.

2. The second general case occurs when $S(\theta)$ is a smooth nonlinear function of θ such that the set of d equations $\mathrm{g}(\theta) = 0$ does not have a direct closed form solution. Typically we use iterative improvement search techniques for these types of problems, using local information about the curvature of S to guide our local search on the surface of S. These are essentially hill-climbing or descending methods (for example, steepest descent). The backpropagation technique used to train neural networks is an example of such a steepest descent algorithm.

Since the second case relies on local information, it may end up converging to a local minimum rather than the global minimum. Because of this, such methods are often supplemented by a stochastic component in which, to take just one example, the optimization procedure starts several times from different randomly chosen starting points.

8.3.2 Closed Form and Linear Algebra Methods

Consider the special case when $S(\theta)$ is a *quadratic* function of θ. This is a very useful special case since now the gradient $\mathrm{g}(\theta)$ is linear in θ and the minimum of S is the unique solution to the set of d linear equations $\mathrm{g}(\theta) = 0$ (assuming the matrix of second derivatives of S at these solutions satisfies the condition of being positive definite). This is illustrated in the context of multiple regression (which usually uses a sum of squared errors score function) in chapter 11. We showed in chapter 4 how the same result was obtained if likelihood was adopted as the score function, assuming Normal error distributions. In general, since such problems can be framed as solving for the inverse of an $d \times d$ matrix, the complexity of solving such linear problems tends to scale in general as $O(nd^2 + d^3)$, where it takes order of nd^2 steps to construct the original matrix of interest and order of d^3 steps to invert it.

8.3.3 Gradient-Based Methods for Optimizing Smooth Functions

In general of course, we often face the situation in which $S(\theta)$ is not a simple function of θ with a single minimum. For example, if our model is a neural network with nonlinear functions in the hidden units, then S will be a relatively complex nonlinear function of θ with multiple local minima. We have already noted that many approaches are based on iteratively repeating some local improvement to the model.

The typical iterative local optimization algorithm can be broken down into four relatively simple components:

1. **Initialize:** Choose an initial value for the parameter vector $\theta = \theta^0$ (this is often chosen randomly).

2. **Iterate:** Starting with $i = 0$, let

$$\theta^{i+1} = \theta^i + \lambda^i \mathbf{v}^i \tag{8.3}$$

   where $\mathbf{v}$ is the direction of the next step (relative to θ^i in parameter space) and λ^i determines the distance. Typically (but not necessarily) $\mathbf{v}^i$ is chosen to be in a direction of improving the score function.

3. **Convergence:** Repeat step 2 until $S(\theta^i)$ appears to have attained a local minimum.

4. **Multiple Restarts:** Repeat steps 1 through 3 from different initial starting points and choose the best minimum found.

Particular methods based on this general structure differ in terms of the chosen direction $\mathbf{v}^i$ in parameter space and the distance λ^i moved along the chosen direction, amongst other things. Note that this is this algorithm has essentially the same design as the one we defined in section 8.2 for local search among a set of discrete states, except that here we are moving in continuous d-dimensional space rather than taking discrete steps in a graph.

The direction and step size must be determined from local information gathered at the current point of the search—for example, whether first derivative or second derivative information is gathered to estimate the local curvature of S. Moreover, there are important trade-offs between the quality of the information gathered and the resources (time, memory) required to calculate this information. No single method is universally superior to all others; each has advantages and disadvantages.

All of the methods discussed below require specification of initial starting points and a convergence (termination) criterion. The exact specifications of these aspects of the algorithm can vary from application to application. In addition, all of the methods are used to try to find a *local extremum* of $S(\theta)$. One must check in practice that the found solution is in fact a minimum (and not a maximum or saddlepoint). In addition, for the general case of a nonlinear function S with multiple minima, little can be said about the quality of the local minima relative to the global minima without carrying out a brute-force search over the entire space (or using sophisticated probabilistic arguments that are beyond this text). Despite these reservations, the optimization techniques that follow are extremely useful in practice and form the core of many data mining algorithms.

8.3.4 Univariate Parameter Optimization

Consider first the special case in which we just have a single unknown parameter θ and we wish to minimize the score function $S(\theta)$ (for example, figure 8.3). Although in practical data mining situations we will usually be optimizing a model with more than just a single parameter, the univariate case is nonetheless worth looking at, since it clearly illustrates some of the general principles that are relevant to the more general multivariate optimization problem. Moreover, univariate search can serve as a *component* in a multivariate search procedure, in which we first find the *direction* of search using the gradient and then decide how far to move in that direction using univariate search for a minimum along that direction.

Letting $g(\theta) = S'(\theta) = \frac{\partial S(\theta)}{\partial \theta}$, the minimum of S occurs wherever $g(\theta) = 0$ and the second derivative $g'(\theta) > 0$. If a closed form solution is possible, then we can find it and we are done. If not, then we can use one of the methods below.

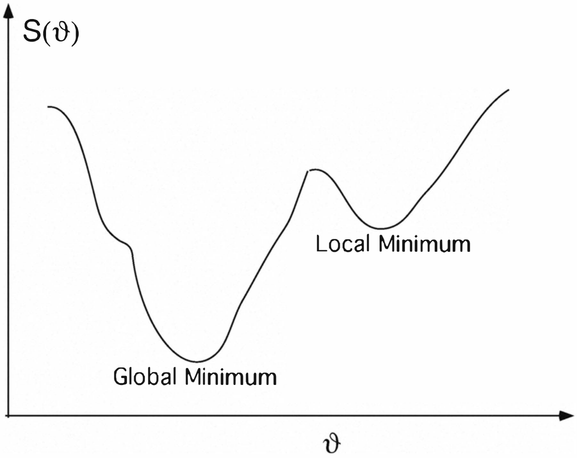

Figure 8.3 An example of a score function $S(\theta)$ of a single univariate parameter θ with both a global minimum and a local minimum.

The Newton-Raphson Method

Suppose that the solution occurs at some unknown point θ^s; that is, $g(\theta^s) = 0$. Now, for points θ^* not too far from θ^s we have, by using a Taylor series expansion

$$g(\theta^s) \approx g(\theta^*) + (\theta^s - \theta^*)g'(\theta^*), \tag{8.4}$$

where this linear approximation ignores terms of order $(\theta^s - \theta^*)^2$ and above. Since θ^s satisfies $g(\theta^s) = 0$, the left-hand side of this expression is zero. Hence, by rearranging terms we get

$$\theta^s \approx \theta^* - \frac{g(\theta^*)}{g'(\theta^*)}. \tag{8.5}$$

In words, this says that given an initial value θ^*, then an approximate solution of the equation $g(\theta^s) = 0$ is given by adjusting θ^* as indicated in equation 8.5. By repeatedly iterating this, we can in theory get as close to the solution as we like. This iterative process is the *Newton-Raphson* (NR) iterative update for univariate optimization based on first and second derivative information. The *i*th step is given by

$$\theta^{i+1} = \theta^i - \frac{g(\theta^i)}{g'(\theta^i)}. \tag{8.6}$$

The effectiveness of this method will depend on the quality of the linear approximation in equation 8.4. If the starting value is close to the true solution θ^s then we can expect the approximation to work well; that is, we can locally approximate the surface around $S(\theta^*)$ as parabolic in form (or equivalently, the derivative $g(\theta)$ is linear near θ^* and θ^s). In fact, when the current θ is close to the solution θ^s, the convergence rate of the NR method is *quadratic* in the sense that the error at step i of the iteration $e_i = |\theta^i - \theta^s|$ can be recursively written as

$$e_i \propto e_{i-1}^2. \tag{8.7}$$

To use the Newton-Raphson update, we must know both the derivative function $g(\theta)$ and the second derivative $g'(\theta)$ in closed form. In practice, for complex functions we may not have closed-form expressions, necessitating numerical approximation of $g(\theta)$ and $g'(\theta)$, which in turn may introduce more error into the determination of where to move in parameter space. Generally speaking, however, if we can evaluate the gradient and second derivative accurately in closed form, it is advantageous to do so and to use this information in the course of moving through parameter space during iterative optimization.

The drawback of NR is, of course, that our initial estimate θ^i may not be sufficiently close to the solution θ^s to make the approximation work well. In this case, the NR step can easily overshoot the true minimum of S and the method need not converge at all.

The Gradient Descent Method

An alternative approach, which can be particularly useful early in the optimization process (when we are potentially far from θ^s), is to use only the gradient information (which provides at least the correct direction to move in for a 1-dimensional problem) with a heuristically chosen step size λ:

$$\theta^{i+1} = \theta^i - \lambda g(\theta^i). \tag{8.8}$$

The multivariate version of this method is known as *gradient* (or *steepest*) descent. Here λ is usually chosen to be quite small to ensure that we do not step too far in the chosen direction. We can view gradient descent as a special case of the NR method, whereby the second derivative information $\frac{1}{g'(\theta^i)}$ is replaced by a constant λ.

Momentum-Based Methods

There is a practical trade-off in choosing λ. If it is too small, then gradient descent may converge very slowly indeed, taking very small steps at each iteration. On the other hand, if λ is too large, then the guarantee of convergence is lost, since we may overshoot the minimum by stepping too far. We can try to accelerate the convergence of gradient descent by adding a *momentum* term:

$$\theta^{i+1} = \theta^i + \triangle^i \tag{8.9}$$

where $\triangle^i$ is defined recursively as

$$\triangle^i = -\lambda g(\theta^i) + \mu\triangle^{i-1} \tag{8.10}$$

and where μ is a "momentum" parameter, $0 \leq \mu \leq 1$. Note that $\mu = 0$ gives us the standard gradient descent method of equation 8.8, and $\mu > 0$ adds a "momentum" term in the sense that the current direction $\triangle^i$ is now also a function of the previous direction $\triangle^{i-1}$. The effect of μ in regions of low curvature in S is to accelerate convergence (thus, improving standard gradient descent, which can be very slow in such regions) and fortunately has little effect in regions of high curvature. The momentum heuristic and related ideas have been found to be quite useful in practice in training models such as neural networks.

Bracketing Methods

For functions which are not well behaved (if the derivative of S is not smooth, for example) there exists a different class of scalar optimization methods that do not rely on any gradient information at all (that is, they work directly on the function S and not its derivative g). Typically these methods are based on the notion of *bracketing*—finding a bracket $[\theta_1, \theta_2]$ that provably contains an extremum of the function. For example, if there exists a "middle" θ value θ_m, such that $\theta_1 > \theta_m > \theta_2$ and $S(\theta_m)$ is less than both $S(\theta_1)$ and $S(\theta_2)$, then clearly a local minimum of the function S must exist between θ_1 and θ_2 (assuming that S is continuous). One can use this idea to fit a parabola through the three points θ_1, θ_m, and θ_2 and evaluate $S(\theta_p)$ where θ_p is located at the minimum value of parabola. Either θ_p is the desired local minimum, or else we can narrow the bracket by eliminating θ_1 or θ_2 and iterating with another parabola. A variety of methods exist that use this idea with varying degrees of sophistication (for example, a technique known as Brent's method is widely used). It will be apparent from this outline that bracketing methods are really a search strategy. We have included them here, however, partly because of their importance in finding optimal values of parameters, and partly because they rely on the parameter space having a connected structure (for example, ordinality) even if the function being minimized is not continuous.

8.3.5 Multivariate Parameter Optimization

We now move on to the much more difficult problem we are usually faced with in practice, namely, finding the minimum of a scalar score function S of a *multivariate* parameter vector θ in d-dimensions. Many of the methods used in the multivariate case are analogous to the scalar case. On the other hand, d may be quite large for our models, so that the multidimensional optimization problem may be significantly more complex to solve than its univariate cousin. It is possible, for example, that local minima may be much more prevalent in high-dimensional spaces than in lower-dimensional spaces. Moreover, a problem similar (in fact, formally equivalent) to the combinatorial explosion that we saw in the discussion of search also manifests itself in multidimensional optimization; this is the curse of dimensionality that we have already encountered in chapter 6. Suppose that we wish to find the d dimensional parameter vector that minimizes some score function, and where each parameter is defined on the unit interval, $[0, 1]$. Then the multivariate parameter vector θ is defined on the unit d-dimensional hypercube. Now suppose we know that at the optimal solution none of the components of θ lie in $[0, 0.5]$. When $d = 1$, this means that half of the parameter space has been eliminated. When $d = 10$, however, only $\left(\frac{1}{2}\right)^{10} \simeq \frac{1}{1000}$ of the parameter space has been eliminated, and when $d = 20$ only $\left(\frac{1}{2}\right)^{20} \simeq \frac{1}{1000000}$ of the parameter space has been eliminated. Readers can imagine—or do the arithmetic themselves—to see what happens when really large numbers of parameters are involved. This shows clearly why there is a real danger of missing a global minimum, with the optimization ending on some (suboptimal) local minimum.

Following the pattern of the previous subsection, we will first describe methods for optimizing functions continuous in the parameters (extensions of the Newton-Raphson method, and so on) and then describe methods that can be applied when the function is not continuous (analogous to the bracketing method).

The iterative methods outlined in the preceding subsection began with some initial value, and iteratively improved it. So suppose that the parameter vector takes the value θ^i at the ith step. Then, to extend the methods outlined in the preceding subsection to the multidimensional case we have to answer two questions:

1. In which direction should we move from θ^i?

2. How far should we step in that direction?

The local iterations can generally be described as

$$\theta^{i+1} = \theta^i + \lambda^i \mathbf{v}^i \tag{8.11}$$

where θ^i is the parameter estimate at iteration i and $\mathbf{v}$ is the d-dimensional vector specifying the next direction to move (specified in a manner dependent on the particular optimization technique being used).

For example, the *multivariate gradient descent* method is specified as

$$\theta^{i+1} = \theta^i - \lambda \mathrm{g}(\theta^i) \tag{8.12}$$

where λ is the scalar *learning rate* and $\mathrm{g}(\theta)$ is a d-dimensional gradient function (as defined in equation 8.2). This method is also known as *steepest descent*, since $-\mathrm{g}(\theta^i)$ will point in the direction of steepest slope from θ^i. Provided λ is chosen to be sufficiently small then gradient descent is guaranteed to converge to a local minimum of the function S.

The *backpropagation* method for parameter estimation popular with neural networks is really merely a glorified steepest descent algorithm. It is somewhat more complicated than the standard approach only because of the multiple layers in the network, so that the derivatives required above have to be derived using the chain rule.

Note that the gradient in the steepest descent algorithm need not necessarily point directly towards the minimum. Thus, as shown in figure 8.4, being limited to take steps only in the direction of the gradient can be an extremely inefficient way to find the minimum of a function. A more sophisticated class of multivariate optimization methods uses local second derivative information about θ to decide where in the parameter space to move to next. In particular, *Newton's method* (the multivariate equivalent of univariate NR) is defined as:

$$\theta^{i+1} = \theta^i - H^{-1}(\theta^i)g(\theta^i) \tag{8.13}$$

where $H^{-1}(\theta^i)$ is the inverse of the $d \times d$ matrix of second derivatives of S (known as the *Hessian matrix*) evaluated at θ^i. The Hessian matrix has entries defined as:

$$h_{lm} = \frac{\partial S(\theta)}{\partial \theta_l \partial \theta_m}, \qquad 1 \leq l, m \leq d. \tag{8.14}$$

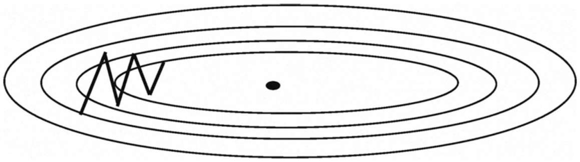

Figure 8.4 An example of a situation in which we minimize a score function of two variables and the shape of the score function is a parabolic "bowl" with the minimum in the center. Gradient descent does not point directly to the minimum but instead tends to point "across" the bowl (solid lines on the left), leading to a series of indirect steps before the minimum is reached.

As in the univariate case, if S is quadratic the step taken by the Newton iteration in parameter space points directly toward the minimum of S. We might reasonably expect that for many functions the shape of the function is approximately locally quadratic in θ about its local minima (think of approximating the shape of the top of a "smooth" mountain by a parabola), and hence, that at least near the minima, the Newton strategy will be making the correct assumption about the shape of S. In fact, this assumption is nothing more than the multivariate version of Taylor's series expansion. Of course, since the peak will usually not be exactly quadratic in shape, it is necessary to apply the Newton iteration recursively until convergence. Again, as in the univariate case, the use of the Newton method may diverge rather than converge (for example, if the Hessian matrix $H(\theta^i)$ is singular; that is, the inverse H^{-1} does not exist at θ^i).

The Newton scheme comes at a cost. Since H is a $d \times d$ matrix, there will be $O(nd^2 + d^3)$ computations required per step to estimate H and invert it. For models with large numbers of parameters (such as neural networks) this may be completely impractical. Instead, we could, for example, approximate H by its diagonal (giving $O(nd)$ complexity per step). Even though the diagonal approximation will clearly be incorrect (since we can expect that parameters will exhibit dependence on each other), the approximation may nonetheless be useful as a linear cost alternative to the full Hessian calculation.

An alternative approach is to build an approximation to H^{-1} iteratively based on gradient information as we move through parameter space. These techniques are known as *quasi-Newton* methods. Initially we take steps in the direction of the gradient (assuming an initial estimate of $\hat{H} = I$ the identity matrix) and then take further steps in the direction $-\hat{H}_i^{-1}(\theta^i)g(\theta^i)$, where

$\hat{H}_i^{-1}(\theta^i)$ is the estimate of H^{-1} at iteration i. The BFGS (Broyden-Fletcher-Goldfarb-Shanno) method is a widely used technique based on this general idea.

Of course, sometimes special methods have been developed for special classes of models and score functions. An example is the *iteratively weighted least squares method* for fitting generalized linear models, as described in chapter 11.

The methods we have just described all find a "good" direction for the step at each iteration. A simple alternative would be merely to step in directions parallel to the axes. This has the disadvantage that the algorithm can become stuck—if, for example there is a long narrow valley diagonal to the axes. If the shape of the function in the vicinity of the minimum is approximated by a quadratic function, then the principal axes of this will define directions (probably not parallel to the axes). Adopting these as an alternative coordinate system, and then searching along these new axes, will lead to a quicker search. Indeed, if the function to be minimized really is quadratic, then this procedure will find the minimum exactly in d steps. These new axes are termed *conjugate directions*.

Once we have determined the direction $\mathbf{v}$ in which we are to move, we can adopt a "line search" procedure to decide how far to move; that is, we simply apply one of the one-dimensional methods discussed above, in the chosen direction. Often a fast and approximate method of choosing the size of the univariate steps may be sufficient in multivariate optimization problems, since the choice of direction itself will itself be based on various approximations.

The methods described so far are all based on, or at least derived from, finding the local direction for the "best" step and then moving in that direction. The *simplex search* method (not to be confused with the *simplex algorithm* of linear programming) evaluates $d+1$ points arranged in a simplex (a "hypertetrahedron") in the d-dimensional parameter space and uses these to define the best direction in which to step. To illustrate, let us take the case of $d = 2$. The function is evaluated at three ($= d + 1$ when $d = 2$) points, arranged as the vertices of an equilateral triangle, which is the simplex in two dimensions. The triangle is then reflected in the side opposite the vertex with the largest function value. This gives a new vertex, and the process is repeated using the triangle based on the new vertex and the two that did not move in the previous reflection. This is repeated until oscillation occurs (the triangle just flips back and forth, reflecting about the same side). When this happens the sides of the triangle are halved, and the process continues.

This basic simplex search method has been extended in various ways. For example, the Nelder and Mead variant allows the triangle to increase as well as decrease in size so as to accelerate movement in appropriate situations. There is evidence to suggest that, despite its simplicity, this method is comparable to the more sophisticated methods described above in high-dimensional spaces. Furthermore, the method does not require derivatives to be calculated (or even to exist).

A related search method, called *pattern search,* also carries out a local search to determine the direction of step. If the step reduces the score function, then the step size is increased. If it does poorly, the step size is decreased (until it hits a minimum value, at which point the search is terminated). (The word *pattern* in the phrase *pattern search* has nothing to do with the patterns of data mining as discussed earlier.)

8.3.6 Constrained Optimization

Many optimization problems involve constraints on the parameters. Common examples include problems in which the parameters are probabilities (which are constrained to be positive and to sum to 1), and models that include the variance as a parameter (which must be positive). Constraints often occur in the form of inequalities, requiring that a parameter θ satisfy $c_1 \leq \theta \leq c_2$, for example, with c_1 and c_2 being constants, but more complex constraints are expressed as functions: $g\left(\theta_1, ..., \theta_d\right) \leq 0$ for example. Occasionally, constraints have the form of equalities. In general, the region of parameter vectors that satisfy the constraints is termed the *feasible region.*

Problems that have linear constraints and convex score functions can be solved by methods of *mathematical programming*. For example, *linear programming* methods have been used in supervised classification problems, and *quadratic programming* is used in suport vector machines. Problems in which the score functions and constraints are nonlinear are more challenging.

Sometimes constrained problems can be converted into unconstrained problems. For example, if the feasible region is restricted to positive values of the parameters $(\theta_1, ..., \theta_d)$, we could, instead, optimize over $(\phi_1, ..., \phi_d)$, where $\theta_i = \phi_i^2$, $i = 1, ..., d$. Other (rather more complicated) transformations can remove constraints of the form $c_1 \leq \theta \leq c_2$.

A basic strategy for removing equality constraints is through *Lagrange multipliers.* A necessary condition for θ to be a local minimum of the score function $S = S\left(\theta\right)$ subject to constraints $h_j\left(\theta\right) = 0$, $j = 1, ..., m$, is that it satisfies $\nabla S(\theta) + \sum_j \lambda_j \nabla h_j(\theta) = 0$, for some scalars, λ_j. These equations and the constraints yield a system of $(d + m)$ simultaneous (nonlinear) equations, that can be solved by standard methods (often by using a least squares routine to minimize the sum of squares of the left hand sides of the $(d + m)$ equations). These ideas are extend to inequality constraints in the *Kuhn-Tucker conditions* (see Further Reading).

Unconstrained optimization methods can be modified to yield constrained methods. For example, penalties can be added to the score function so that the parameter estimates are repelled if they should approach boundaries of the feasible region during the optimization process.

8.4 Optimization with Missing Data: The EM Algorithm

In this section we consider the special but important problem of maximizing a likelihood score function when some of the data are missing, that is, there are variables in our data set whose values are unobserved for some of the cases. It turns out that a large number of problems in practice can effectively be modeled as missing data problems. For example, measurements on medical patients where for each patient only a subset of test results are available, or application form data where the responses to some questions depends on the answers to others.

More generally, any model involving a hidden variable (i.e., a variable that cannot be directly observed) can be modeled as a missing data problem, in which the values of this variable are unknown for all n objects or individuals. Clustering is a specific example; in effect we assume the existence of a discrete-valued hidden cluster variable C taking values $\{c_1, \ldots, c_k\}$ and the goal is to estimate the values of C (that is, the cluster labels) for each observation $\mathbf{x}(i)$, $1 \leq i \leq n$.

The Expectation-Maximization (EM) algorithm is a rather remarkable algorithm for solving such missing data problems in a likelihood context. Specifically, let $D = \{\mathbf{x}(1), \ldots, \mathbf{x}(n)\}$ be a set of n observed data vectors. Let $H = \{z(1), \ldots, z(n)\}$ represent a set of n values of a hidden variable Z, in one-to-one correspondence with the observed data points D; that is, $z(i)$ is associated with data point $\mathbf{x}(i)$. We can assume Z to be discrete (this is not necessary, but is simply convenient for our description of the algorithm), in which case we can think of the unknown $z(i)$ values as class (or cluster) labels for the data, that are hidden.

We can write the log-likelihood of the observed data as

$$l(\theta) = \log p(D|\theta) = \log \sum_H p(D, H|\theta) \tag{8.15}$$

where the term on the right indicates that the observed likelihood can be expressed as the likelihood of both the observed and hidden data, summed over the hidden data values, assuming a probabilistic model in the form $p(D, H|\theta)$ that is parametrized by a set of unknown parameters θ. Note that our optimization problem here is doubly complicated by the fact that *both* the parameters θ and the hidden data H are unknown.

Let $Q(H)$ be any probability distribution on the missing data H. We can then write the log-likelihood in the following fashion:

$$\begin{aligned} l\,(\theta) &= \log \sum_H p(D, H|\theta) \\ &= \log \sum_H Q(H) \frac{p(D, H|\theta)}{Q(H)} \\ &\geq \sum_H Q(H) \log \frac{p(D, H|\theta)}{Q(H)} \\ &= \sum_H Q(H) \log p(D, H|\theta) + \sum_H Q(H) \log \frac{1}{Q(H)} \\ &= F(Q, \theta) \end{aligned} \tag{8.16}$$

where the inequality is a result of the concavity of the log function (known as Jensen's inequality).

The function $F(Q, \theta)$ is a lower bound on the function we wish to maximize (the likelihood $l(\theta)$). The EM algorithm alternates between maximizing F with respect to the distribution Q with the parameters θ fixed, and then maximizing F with respect to the parameters θ with the distribution $Q = p(H)$ fixed. Specifically:

$$\text{E-step:} \qquad Q^{k+1} = \arg\max_Q F(Q^k, \theta^k) \tag{8.17}$$

$$\text{M-step:} \qquad \theta^{k+1} = \arg\max_\theta F(Q^{k+1}, \theta^k) \tag{8.18}$$

It is straightforward to show that the maximum in the E-step is achieved when $Q^{k+1} = p(H|D, \theta^k)$, a term that can often be calculated explicitly in a relatively straightforward fashion for many models. Furthermore, for this value of Q the bound becomes tight, i.e., the inequality becomes an equality above and $l(\theta^k) = F(Q, \theta^k)$.

The maximization in the M-step reduces to maximizing the first term in F (since the second term does not depend on θ), and can be written as

$$\theta^{k+1} = \arg\max_{\theta} \sum_{H} p(H|D, \theta^k) \log p(D, H|\theta^k). \tag{8.19}$$

This expression can also fortunately often be solved in closed form.

Clearly the E and M steps as defined cannot decrease $l(\theta)$ at each step: at the beginning of the M-step we have that $l(\theta^k) = F(Q^{k+1}, \theta^k)$ by definition, and the M-step further adjusts θ to maximize this F.

The EM steps have a simple intuitive interpretation. In the E-step we estimate the distribution on the hidden variables Q, conditioned on a particular setting of the parameter vector θ^k. Then, keeping the Q function fixed, in the M-step we choose a new set of parameters θ^{k+1} so as to maximize the expected log-likelihood of observed data (with expectation defined with respect to $Q = p(H)$). In turn, we can now find a new Q distribution given the new parameters θ^{k+1}, then another application of the M-step to get θ^{k+2}, and so forth in an iterative manner. As sketched above, each such application of the E and M steps is guaranteed not to decrease the log-likelihood of the observed data, and under fairly general conditions this in turn implies that the parameters θ will converge to at least a local maximum of the log-likelihood function.

To specify an actual algorithm we need to pick an initial starting point (for example, start with either an initial randomly chosen Q or θ) and a convergence detection method (for example, detect when any of Q, θ, or $l(\theta)$ do not change appreciably from one iteration to the next). The EM algorithm is essentially similar to a form of local hill-climbing in multivariate parameter space (as discussed in earlier sections of this chapter) where the direction and distance of each step is implicitly (and automatically) specified by the E and M steps. Thus, just as with hill-climbing, the method will be sensitive to initial conditions, so that different choices of initial conditions can lead to different local maxima. Because of this, in practice it is usually wise to run EM from different initial conditions (and then choose the highest likelihood solution) to decrease the probability of finally settling on a relatively poor local maximum. The EM algorithm can converge relatively slowly to the final parameter values, and for example, it can be combined with more traditional optimization techniques (such as Newton-Raphson) to speed up convergence in the later iterations. Nonetheless, the standard EM algorithm is widely used given the broad generality of the framework and the relative ease with which an EM algorithm can be specified for many different problems.

The computational complexity of the EM algorithm is dictated by both the number of iterations required for convergence and the complexity of each of the E and M steps. In practice it is often found that EM can converges relatively slowly as it approaches a solution, although the actual rate of convergence can depend on a variety of different factors. Nonetheless, for simple models at least, the algorithm can often converge to the general vicinity of the solution after only a few (say 5 or 10) iterations. The complexity of the E and M steps at each iteration depends on the nature of the model being fit to the data (that is, the likelihood function $p(D, H|\theta)$). For many of the simpler models (such as the mixture models discussed below) the E and M steps need only take time linear in n, i.e., each data point need only be visited once during each iteration.

Examples 8.1 and 8.2 illustrate the application of the EM algorithm in estimating the parameters of a normal mixture and a Poisson mixture (respectively) for one-dimensional measurements x. In each case, the data are assumed to have arisen from a mixture of K underlying component distributions (normal and Poisson, respectively). However, the component labels are unobserved, and we do not know which component each data point arose from. We will discuss the estimation of these types of mixture models in more detail again in chapter 9.

Example 8.1 We wish to fit a normal mixture distribution

$$f(x) = \sum_{k=1}^{K} \pi_k f_k(x; \mu_k, \sigma_k) \tag{8.20}$$

where μ_k is the mean of the kth component, σ_k is the standard deviation of the kth component, and π_k is the prior probability of a data point belonging to component k ($\sum_K \pi_k = 1$). Hence, for this problem, we have that the parameter vector $\theta = \{p_1, \ldots, p_K, \mu_1, \ldots, \mu_K, \sigma_1, \ldots, \sigma_K\}$. Suppose for the moment that we knew the values of θ. Then, the probability that an object with measurement vector x arose from the kth class would be

$$\hat{P}(k \mid x) = \frac{\pi_k f_k(x; \mu_k, \sigma_k)}{f(x)} \tag{8.21}$$

This is the basic E-step.

From this, we can then estimate the values of π_k, μ_k, and σ_k as

$$\hat{\pi}_k = \frac{1}{n} \sum_{i=1}^{n} \hat{P}(k \mid x(i)) \tag{8.22}$$

$$\hat{\mu}_k = \frac{1}{n\hat{\pi}_k} \sum_{i=1}^{n} \hat{P}(k \mid x(i))x(i) \tag{8.23}$$

$$\hat{\sigma}_k = \frac{1}{n\hat{\pi}_k} \sum_{i=1}^{n} \hat{P}(k \mid x(i))(x(i) - \hat{\mu}_k)^2 \tag{8.24}$$

where the summations are over the n points in the data set. These three equations are the M-steps. This set of equations leads to an obvious iterative procedure. We pick starting values for μ_k, σ_k, and π_k, plug them into equation 8.21 to yield estimates $\hat{P}(k \mid x)$, use these estimates in equations 8.22, 8.23, and 8.24, and then iterate back using the updated estimates of μ_k, σ_k, and π_k, cycling around until a convergence criterion (usually convergence of the likelihood or model parameters to a stable point) has been satisfied.

Note that equations 8.23 and 8.24 are very similar to those involved in estimating the parameters of a single normal distribution, except that the contribution of each point are split across the separate components, in proportion to the estimated size of that component at the point. In essence, each data point is weighted by the probability that it belongs to that component. If we actually knew the class labels the weights for data point $x(i)$ would be 1 for the class to which the data point belongs and 0 for the other $K - 1$ components (in the standard manner).

Example 8.2 The Poisson model can be used to model the rate at which individual events occur, for example, the rate at which a consumer uses a telephone calling card. For some cards, there might be multiple individuals (within a single family for example) on the same account (with copies of the card), and in theory each may have a different rate at which they use it (for example, the teenager uses the card frequently, the father much less frequently, and so on). Thus, with K individuals, we would observe event data generated by a mixture of K Poisson processes:

$$f(x) = \sum_{k=1}^{K} \pi_k \frac{(\lambda_k)^x e^{-\lambda_k}}{x!}, \tag{8.25}$$

the equations for the iterative estimation procedure analogous to example 8.4 take the form

$$\hat{P}(k \mid x\,(i)) = \frac{\pi_k P(x\,(i) \mid k)}{f(x\,(i))} = \frac{\pi_k \frac{(\lambda_k)^{x(i)} e^{-\lambda_k}}{x(i)!}}{f\,(x\,(i))} \tag{8.26}$$

$$\hat{\pi}_k = \frac{1}{n} \sum_{i=1}^{n} \hat{P}(k \mid x\,(i)) \tag{8.27}$$

$$\hat{\lambda}_k = \frac{1}{n\hat{\pi}_k} \sum_{i=1}^{n} \hat{P}(k \mid x(i)) x(i) \tag{8.28}$$

8.5 Online and Single-Scan Algorithms

All of the optimization methods we have discussed so far implicitly assume that the data are all resident in main memory and, thus, that each data point can be easily accessed multiple times during the course of the search. For very large data sets we may be interested in optimization and search algorithms that see each data point only once at most. Such algorithms may be referred to as *online* or *single-scan* and clearly are much more desirable than "multiple-pass" algorithms when we are faced with a massive data set that resides in secondary memory (or further away).

In general, it is usually possible to modify the search algorithms above directly to deal with data points one at a time. For example, consider simple gradient descent methods for parameter optimization. As discussed earlier, for the "offline" (or batch) version of the algorithm, one finds the gradient $g(\theta)$ in parameter space, evaluates it at the current location θ^k, and takes a step proportional to distance λ in that direction. Now moving in the direction of the gradient $g(\theta)$ is only a heuristic, and it may not necessarily be the optimal direction. In practice, we may do just as well (at least, in the long run) if we move in a direction approximating that of the gradient. This idea is used in practice in an *online approximation* to the gradient, that uses the current best estimate based both on the current location and the current and (perhaps) "recent" data points. The online estimates can be viewed as *stochastic* (or "noisy") estimates of the full gradient estimate that would be produced by the batch algorithm looking at all of the data points. There exists a general theory in statistics for this type of search technique, known as *stochastic approximation*, which is beyond the scope of this text but that is relevant to online parameter estimation. Indeed, in using gradient descent to find weight parameters for neural networks (for example) stochastic online search has been found to be useful in practice. The stochastic (data-driven) nature of the search is even thought to sometimes improve the quality of the solutions found by allowing the search algorithm to escape from local minima in a manner somewhat reminiscent of simulated annealing (see below).

More generally, the more sophisticated search methods (such as multivariate methods based on the Hessian matrix) can also be implemented in an online manner by appropriately defining online estimators for the required search directions and step-sizes.

8.6 Stochastic Search and Optimization Techniques

The methods we have presented thus far on model search and parameter optimization rely heavily on the notion of taking local greedy steps near the current state. The main disadvantage is the inherent myopia of this approach. The quality of the solution that is found is largely a function of the starting point. This means that, at least with a single starting position, there is the danger that the minimum (or maximum) one finds may be a nonglobal local optimum. Because of this, methods have been developed that adopt a more global view by allowing large steps away from the current state in a nondeterministic (stochastic) manner. Each of the methods below is applicable to either the parameter optimization or model search problem, but for simplicity we will just focus here on model search in a state-space.

- **Genetic Search:** Genetic algorithms are a general set of heuristic search techniques based on ideas from evolutionary biology. The essential idea is to represent states (models in our case) as chromosomes (often encoded as binary strings) and to "evolve" a *population* of such chromosomes by selectively pairing chromosomes to create new offspring. Chromosomes (states) are paired based on their "fitness" (their score function) to encourage the fitter chromosomes to survive from one generation to the next (only a limited number of chromosomes are allowed to survive from one generation to the next). There are many variations on this general theme, but the key ideas in genetic search are:

  – Maintenance of a set of candidate states (chromosomes) rather than just a single state, allowing the search algorithm to explore different parts of the state space simultaneously

  – Creating new states to explore based on combinations of existing states, allowing in effect the algorithm to "jump" to different parts of the state-space (in contrast to the local improvement search techniques we discussed earlier)

  Genetic search can be viewed as a specific type of heuristic, so it may work well on some problems and less well on others. It is not always clear that it provides better performance on specific problems than a simpler method such as local iterative improvement with random restarts. A practical drawback of the approach is the fact that there are usually many *algorithm parameters* (such as the number of chromosomes, specification of how chromosomes are combined, and so on) that must be specified and it may not be clear what the ideal settings are for these parameters for any given problem.

- **Simulated Annealing:** Just as genetic search is motivated by ideas from evolutionary biology, the approach in *simulated annealing* is motivated by ideas from physics. The essential idea is to not to restrict the search algorithm to moves in state-space that decrease the score function (for a score function we are trying to minimize), but to also allow (with some probability) moves that can *increase* the score function. In principle, this allows a search algorithm to escape from a local minimum. The probability of such non-decreasing moves is set to be quite high early in the process and gradually decreased as the search progresses. The decrease in this probability is analogous to the process of gradually decreasing the *temperature* in the physical process of annealing a metal with the goal of obtaining a low-energy state in the metal (hence the name of the method).

  For the search algorithm, higher temperatures correspond to a greater probability of large moves in the parameter space, while lower temperatures correspond to greater probability of only small moves that decrease the function being taken. Ultimately, the *temperature schedule* reduces the temperature to zero, so that the algorithm by then only moves to states that decrease the score function. Thus, at this stage of the search, the algorithm will inevitably converge to a point at which no further decrease is possible. The hope is that the earlier (more random) moves have led the algorithm to the deepest "basin" in the score function surface. In fact, one of the appeals of the approach is that it can be mathematically proved that (under fairly general conditions) this will happen if one is using the appropriate temperature schedule. In practice, however, there is usually no way to specify the optimal temperature schedule (and the precise details of how to select the possible nondecreasing moves) for any specific problem. Thus, the practical application of simulated annealing reduces to (yet another) heuristic search method with its own set of algorithm parameters that are often chosen in an ad hoc manner.

  We note in passing that the idea of *stochastic search* is quite general, where the next set of parameters or model is chosen stochastically based on a probability distribution on the quality of neighboring states conditioned on the current state. By exploring state-space in a stochastic fashion, a search algorithm can in principle spend more time (on average) in the higher quality states and build up a model on the distribution of the quality (or score) function across the state-space. This general approach has become very popular in Bayesian statistics, with techniques such as Monte Carlo Markov Chain (MCMC) being widely used. Such methods can be viewed as generalizations of the basic simulated annealing idea, and again, the key ideas originated in physics. The focus in MCMC is to find the *distribution* of scores in parameter or state-space, weighted by the probability of those parameters or models given the data, rather than just finding the location of the single global minimum (or maximum).

It is difficult to make general statements about the practical utility of methods such as simulated annealing and genetic algorithms when compared to a simpler approach such as iterative local improvement with random restarts, particularly if we want to take into account the amount of time taken by each method. It is important when comparing different search methods to compare not only the *quality* of the final solution but also the *computational resources* expended to find that solution. After all, if time is unlimited, we can always use exhaustive enumeration of all models to find the global optimum. It is fair to say that since stochastic search techniques typically involve considerable extra computation and overhead (compared to simpler alternatives) that they tend to be used in practice on specialized problems involving relatively small data sets, and are often not practical from a computational viewpoint for very large data sets.

8.7 Further Reading

Papadamitriou and Steiglitz (1982) is a classic (although now a little outdated) text on combinatorial optimization. Cook et al. (1998) is an authoritative and more recent text on the topic. Pearl (1984) deals specifically with the topic of search heuristics. The CN2 rule-finding algorithm of Clark and Niblett (1989) is an example of beam search in action.

Press et al. (1988) is a useful place to start for a general introduction and some sound practical advice on numerical optimization techniques, particularly chapters 9 and 10. Other texts such as Gill, Murray, and Wright (1981) and Fletcher (1987) are devoted specifically to optimization and provide a wealth of practical advice as well as more details on specific methods. Luenberger (1984) and Nering and Tucker (1993) discuss linear programming and related constrained optimization techniques in detail. Mangasarian (1997) describes the application of constrained optimization techniques to a variety problems in data mining, including feature selection, clustering, and robust model selection. Bradley, Fayyad, and Mangasarian (1999) contain further discussion along these lines.

Thisted (1988) is a very useful and comprehensive reference on the application of optimization and search methods specifically to statistical problems. Lange (1999) is more recent text on the same topic (numerical methods for statistical optimization) with a variety of useful techniques and results. Bishop (1995, chapter 7) has an extensive and well-written account of optimization in the context of parameter estimation for neural networks, with specific reference to online techniques.

The seminal paper on the EM algorithm is Dempster, Laird, and Rubin (1977) which first established the general theoretical framework for the procedure. This paper had been preceded by almost a century of work in the general spirit of EM, including Newcomb (1886) and McKendrick (1926). The work of Baum and Petrie (1966) was an early development of a specific EM algorithm in the context of hidden Markov models. McLachlan and Krishnan (1998) provide a comprehensive treatment of the many recent advances in the theory and application of EM. Meilijson (1989) introduced a general technique for speeding up EM convergence, and Lange (1995) discusses the use of gradient methods in an EM context. A variety of computational issues concerning EM in a mixture modeling context are discussed in Redner and Walker (1984). Neal and Hinton (1998) discuss online versions of EM that can be particularly useful in the context of massive data sets.

Online learning in a regression context can be viewed theoretically as a special case of the general technique of stochastic approximation of Robbins and Monro (1951)—see Bishop (1995, chapter 2) for a discussion in the context of neural networks.

Mitchell (1997) is a comprehensive introduction to the ideas underlying genetic algorithms. Simulated annealing was introduced by Kirkpatrick, Gelatt, and Vecchi (1983) but has its origins in much earlier work in statistical physics. Van Laarhoven and Aarts (1987) provide a general overview of the field. Brooks and Morgan (1995) contains a systematic comparison between simulated annealing and more conventional optimization techniques (such as Newton-based methods), as well as hybrids of the two. They conclude that hybrid methods appear better than either traditional methods or simulated annealing on their own. Gilks, Richardson, and Spiegelhalter (1996) is an edited volume containing a good sampling of recent work in statistics using stochastic search and MCMC methods in a largely Bayesian context.