Cite as: Majoros WH (2008) Advanced Methods for Parameter Estimation. Online supplement to: Methods for Computational Gene Prediction, Cambridge University Press, 2007.

supplement S2

Advanced Methods for Parameter Estimation

We have described a number of predictive models in this book, all of which incorporate parameters that must be determined before the models can be used for prediction. For each type of model we have outlined typical methods for *parameter estimation*—i.e., for *training* the model—specific to that model type. In this chapter we give a more thorough treatment of parameter estimation techniques for sequence analysis models, with an emphasis on the use of gradient-based methods as well as the popular e*xpectation maximization* algorithm; we also consider (briefly) the possibility of estimating parameters via sampling. This chapter should prove most useful to those contemplating a new type of gene-finding model which extends in some way the traditional models discussed in earlier chapters. It will also be seen to be relevant to the training of *conditional random fields* (described in supplementary chapter S1) and to the discriminative training of generative models such as HMMs and their many variants.

S2.1 Parameter Estimation as Optimization

It may be reasonably argued[1] that in the training of any model, our goal will

[1] ...as long as you are not arguing with a Bayesian, who will likely tell you that you should instead perform parameter estimation via *sampling*. We will consider this option very briefly near the end of the chapter.

always be to maximize some function f of the training data T and the model parameters θ. One example of such an *objective function* is the likelihood $f(\theta)=P(T|\theta)$, in which case our training objective is the popular *maximum likelihood* criterion commonly employed for generative models:

$$\theta_{MLE} = \arg\max_{\theta} P(T|\theta) \tag{S2.1}$$

Another objective function is the *expected prediction accuracy* on unseen data; we will consider this case more fully in section S2.5.

More generally, we may conceive of the process of parameter estimation as that of finding a global peak on some landscape defined in a multi-dimensional space, in which the individual dimensions of the space correspond to individual model parameters, and the contour of the landscape in that space is defined by the objective function f. More simply put, we wish to find the "best" model parameters θ, where the notion of "model goodness" is formalized via some objective function $f(\theta)$. In the case of training an HMM from fully observable data, we saw in section 6.3 that the optimal parameters could be directly computed via simple counting. For cases in which the training data are not *fully observable* (such as when the model has hidden states which are not annotated in the training data), or when the objective function is too complicated to permit a simple, analytical solution, we will typically not be able to compute the optimal parameters directly. In these cases we must resort to the more laborious process of searching the optimization landscape for the desired optimum. This scenario is illustrated very crudely in Figure S2.1.

As noted in Chapter 2, the problem of optimization is often confounded by the existence of local optima. Whereas the goal of optimization is generally to find a global *optimum* (i.e., global *maximum* or global *minimum*), many optimization methods are formulated so as to find a *local* optimum, which may or may not also be a global optimum. We will revisit the issue of local-versus-global optima in section S2.4.

In the following sections we will consider several methods for finding a local optimum of a given objective function, including the popular *Expectation Maximization* (*EM*) technique as well as more general "hill-climbing" and "gradient ascent" methods. As noted in section 2.5, hill-climbing methods seek to iteratively improve on a random initial solution so as to find successively better solutions which ultimately converge on an optimum. Although hill-climbing methods are well known to be susceptible to the problem of local optima, the hill-climbing approach remains at the heart of most of the popular optimization techniques used for model training, including even the venerable EM method (see section S2.3.1).

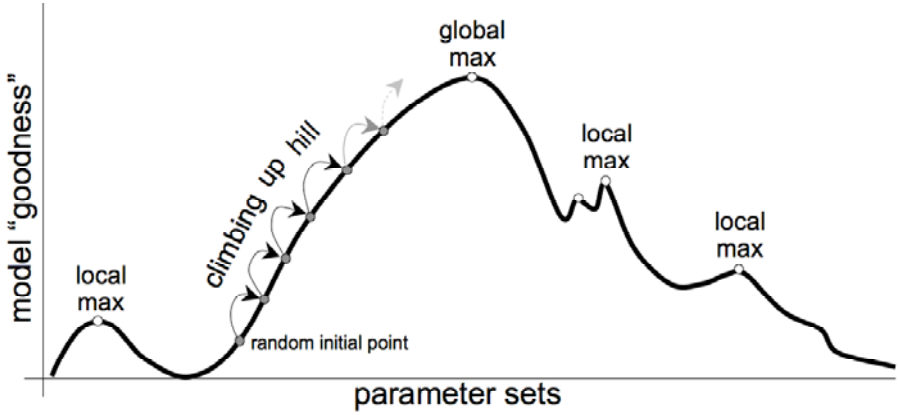

Figure S2.1 *Crude illustration of hill-climbing. Sets of model parameters are organized along the x-axis, with similar parameter sets situated near each other. The objective function provides evaluations for individual parameter sets, thereby defining a landscape over which an optimizer may search for a local maximum by iteratively choosing the best neighboring parameter sets in a series of neighborhoods.*

It will be useful in what follows to make reference to a scenario referred to as *black-box optimization* (Figure S2.2). In this scenario, a general-purpose optimizer $\mathcal{A}$ is utilized for the task of optimizing some function *f*. It should be noted that the optimizer has no intimate knowledge of the inner workings of *f*, and likewise *f* remains ignorant of the inner workings of the optimizer $\mathcal{A}$. The optimizer and the objective function interact only through a strict interface known as *generate-and-test*. According to this paradigm, the optimizer generates bit strings (or strings in some other general-purpose encoding) which are interpreted by the objective function as parameterizations θ. The objective function evaluates the training data *T* under this encoded parameterization and returns a single "quality" value to the optimizer, which we may conceptualize as the height of the landscape at the point θ. The sending of a single bit string and the return of a single quality value constitutes one *function evaluation*. It should be intuitively clear that, through the sending of many alternative bit strings and the sorting of their returned quality values, the optimizer may be able to forge a path of successively better parameterizations, so long as the objective function is sufficiently *smooth*—i.e., as long as parameterizations which differ only slightly in their parameter values also differ only slightly in their quality values. This was the basis for the rudimentary hill-climbing algorithm given in Chapter 2.

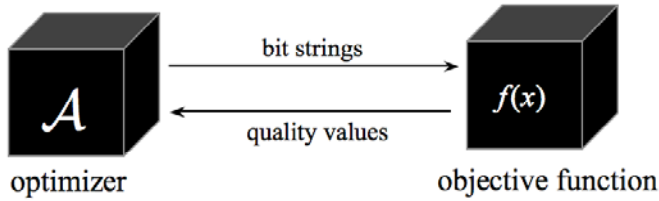

Figure S2.2 *The paradigm of generate-and-test. A black-box optimizer sends bit strings encoding values of x to a function evaluator, which is itself a black box. The function evaluator returns the value of f(x) to the optimizer, which then utilizes this information in choosing a new x to be sent as the next query.*

Optimizers which perform hill-climbing using only the quality values returned by the objective function are sometimes called "blind" hill-climbers, because they cannot directly observe the contour of the landscape which they are searching—they instead have to "feel their way" blindly in search of local improvements over their current location on the landscape (see Figure S2.3). We say that such methods are of *zeroth order*.

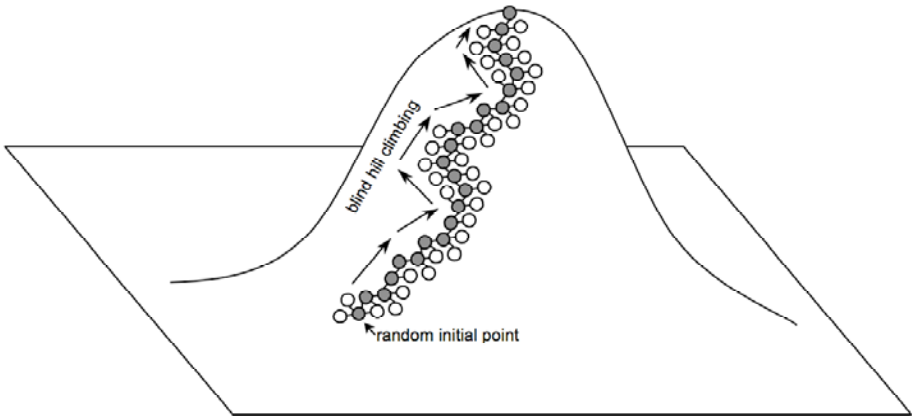

Figure S2.3 *Blind hill-climbing. A blind optimizer has no sense for the overall shape of the landscape, but must test local points in the immediate neighborhood, choosing the best local step at each point. When the step size is small, reaching the optimum can take a very large number of steps.*

A potentially more efficient alternative is to modify the generate-and-test paradigm so that at each step the optimizer obtains not only a quality value $f(x)$ for the current point x (i.e., the "height" of the landscape at that point), but also a precise description of the slope of the landscape at that point. Since the "slope" of a multi-dimensional function is described by the function's *gradient* (i.e., vector of first derivatives), we say that such methods are

performing *gradient-based search*, and are of *first order*. Later we will consider *second-order* methods which utilize not only the gradient of the function, but also the curvature as defined by the function's second derivatives.

Although we will not dwell on the issue, it should be fairly obvious that parameter estimation can become rather more difficult when our model includes some variables which are *unobservable*—i.e., variables whose values we neither know nor wish to estimate. These are sometimes called "nuissance variables" since in most cases one would simply like to eliminate them, to the extent possible. Two common strategies for accommodating unobservables are (1) to integrate them out, and (2) to repeatedly sample likely values for them. The act of integrating over the unobservables can sometimes be accomplished internally within the objective function; an example is the summation over ancestral states in a PhyloHMM (section 9.6.1). For continuous variables this can be more difficult. When the appropriate integrals cannot be solved analytically, one might resort to numerical integration (see, e.g., Press *et al*., 1992). A sometimes more computationally amenable solution is to sample values for the unobservables. Sampling is discussed in section S2.6. Another technique which directly accommodates missing data is *EM*, which we describe in section S2.3.1.

S2.2 Optimization with Gradients

Because the gradient provides information about how fast the function is increasing or decreasing in any given direction at a particular location on the landscape, gradients can inform the hill-climbing process by indicating the most promising direction at any given point—i.e., the direction in which we can expect to most rapidly approach the optimum. Without gradient information a hill-climber typically needs to query a number of points surrounding the current point in order to identify a promising direction for further search. Thus, gradient methods can reduce the number of function evaluations needed during hill-climbing. Unfortunately, these methods require that we be able to evaluate the function's derivative, which for some functions is difficult to do analytically. We will return to this issue shortly.

S2.2.1 Gradient Search in One Dimension

Optimization via gradients is most easily explained for the case of a univariate function $f(x)$—i.e., for a curve in a single-dimensional space indexed by $x \in \mathbb{R}$. For the more complex case of multidimensional optimization, x will be replaced by a vector $\mathbf{x}$ of parameters; we will consider this case in the next section.

Depending on whether our objective function is to be maximized or minimized, we may describe gradient-based methods as either gradient *ascent* or gradient *descent*, respectively. Since maximization of a function

$f(x)$ is effectively the same as minimization of its additive inverse, $-f(x)$, we will, without loss of generality, consider only the case of gradient ascent, and will therefore restrict our attention to the maximization of the objective function.

The simplest gradient-based method, termed *steepest ascent* (or steepest *descent* for minimization problems) utilizes only the first derivative of the objective function; shortly we will consider methods which utilize the second derivative as well. Starting from a randomly chosen initial point x_k, for $k=0$, we compute the gradient $f'(x_k)$ at that point, and then apply the following update rule:

$$x_{k+1} = x_k + \eta f'(x_k) \tag{S2.2}$$

for some fixed (and typically very small) *learning rate*, η, to arrive at the next point x_{k+1}, which it is hoped will be closer to the maximum than was x_k. The process continues via iterative application of Eq. (S2.2) to produce successive values x_k, resulting in a "walk" $w=(x_0, x_1, ..., x_N)$ over the optimization landscape. The process may be terminated when either a fixed N is reached, or when the improvement between successive steps in the walk becomes negligible—i.e.,

$$f(x_{k+1}) - f(x_k) < \varepsilon \tag{S2.3}$$

for some small, fixed value ε. Other termination criteria may be applied as deemed appropriate.

One shortcoming of the steepest ascent approach to hill climbing derives from the use of a fixed learning rate, η. The justification for the update rule given by Eq. (S2.2) is that the derivative $f'(x)$ indicates the direction of increasing values of $f(x)$—i.e., when $f'(x)>0$, then in the immediate vicinity of x we have $f(x+dx)>f(x)$, whereas when $f'(x)<0$ then $f(x-dx)>f(x)$, for sufficiently small $dx>0$. Thus, we desire an update rule of the form $x_{k+1}=x_k+\delta$ where the *step size*, δ, has the same sign as $f'(x)$ and is small—i.e., $|\delta| \ll 1$. The method prescribed by Eq. (S2.2) satisfies these two requirements by setting $\delta=\eta f'(x)$. However, leaving η fixed during the optimization process creates two potential difficulties.

First, the user (or creator) of the optimization software must guess a value of η which is appropriately scaled for the function being optimized. Returning to our analogy of landscapes, if the optimization function is very *smooth* in the sense that changes in elevation occur only very slowly (as measured by the horizontal distance along the landscape), then there should be no harm in taking moderately large steps, since any single step is unlikely to inadvertantly pass by a peak hidden between two successive points in the optimizer's walk. Indeed, on such a landscape we would prefer large learning

rates over small ones, since an η which is too small may result in the optimizer having to take excessively many (small) steps in order to reach an optimum. Conversely, if the landscape is very *rugged*, in the sense that the elevation can change very abruptly, then smaller steps would be safer, since a large step may skip right over the optimum. Unfortunately, the most appropriate step size (and therefore the most appropriate learning rate) for a function depends on the function itself, and prior to performing the optimization it can be difficult to guess an optimal learning rate. In practice, users often run the optimization several times with different learning rates, finding a suitable learning rate by trial and error.

A second difficulty derives from the fact that in Eq. (S2.2), the step size $\delta=\eta f'(x)$ is proportional to $f'(x)$. That is, the step size will be largest when the function is changing most rapidly and smallest when it is changing most slowly. Thus, in the vicinity of a steep landform the optimizer may inadvertently take overly large steps and thereby miss a nearby optimum, whereas in regions of the landscape where the elevation is changing very slowly the optimizer may take steps which are so small that its rate of progress is painfully slow.

For these reasons, steepest-descent methods are generally eschewed in favor of more sophisticated algorithms based on *Newton's method* for finding roots, which the reader may remember from grade school. We will review the method very briefly.

Consider the problem of finding a *root* of the function $f(x)$—that is, of finding a value x^* such that $f(x^*)=0$. This is the problem addressed by Newton's method. Figure S2.4 illustrates the scenario. The function $f(x)$ is shown as a bold curve which intersects the x-axis; at this intersection $f(x)$ obviously assumes the value 0, and so our aim is find that x^* at which the curve crosses the x-axis.

Suppose we know *a priori* that the coordinate x_0 is in the vicinity of a root of f. Then if we can identify the line $y=mx+b$ which is tangent to the function at the point (x_0, y_0) for $y_0=f(x_0)$, we can solve algebraically for the coordinate x_1 where this tangent line intersects the x-axis. The intersection of this tangent with the x-axis should be a better approximation for x^* than x_0, as can be seen from Figure S2.4. We can now repeat this process by finding the tangent line at $(x_1, f(x_1))$ and solving for x_2, the x-value at the intersection of this second tangent line with the x-axis. As suggested by the figure, iterating this process should produce a series of x_k values which converge on the desired root x^* for the objective function.

Deriving Newton's update rule for obtaining x_{k+1} from x_k is simple. Since the derivative of the function gives the slope of the tangent, and since the slope of the tangent is given by the ratio of the change in y to the change in x between $(x_k, f(x_k))$ and $(x_{k+1},0)$, we have:

$$f'(x_k) = \frac{\Delta y}{\Delta x} = \frac{0 - f(x_k)}{x_{k+1} - x_k} \tag{S2.4}$$

From this we can trivially see that:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \tag{S2.5}$$

giving us Newton's update rule. Applying this rule iteratively, starting from x_0, we should converge on the root or "zero" x^* of the function. Keep in mind that we assumed that the initial value x_0 was already near the desired root. We will return to this important point when we consider methods for avoiding local optima.

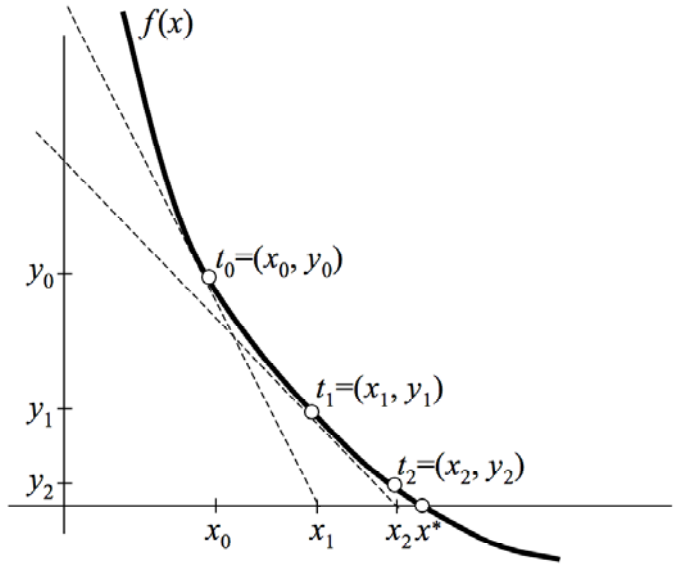

Figure S2.4 *Graphical illustration of Newton's method for finding the roots of a univariate function. The function f(x) is shown in bold. Dashed lines show the tangent at points $t_k=(x_k,y_k)$. The series $(x_0, x_1, x_2, \ldots)$ converges to x^*, the location on the x-axis where the function evaluates to zero.*

Now let us consider how we may use Newton's method to find not the zero of a function, but rather the function's maximum. Recall from elementary calculus that the optima of a function occur at either the boundaries of the function's domain (or any other place where its derivative is undefined) or at the *critical points*—i.e., the points x^* where $f'(x^*) = 0$.

Ignoring the issue of boundaries, and assuming we have a starting point x_0 which is already near a maximum (so that we can rule out the possibility of the nearest critical point actually being a minimum or a *saddle point*), we can apply Newton's method to the function's *derivative* (instead of to the function itself), giving us the following update rule:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{S2.6}$$

Thus, given (1) a function's first derivative, (2) the function's second derivative, and (3) a point x_0 known to be close to a local maximum x^* for the function (and assuming both derivatives are defined in the vicinity of x^*), we can use Newton's method to find x^* to any desired precision, simply by iteratively applying the update equation until $f(x_{k+1}) - f(x_k) < \varepsilon$, for termination threshold ε.

Ignoring for now the issue of picking a suitable initial point x_0 near the unknown maximum, we see that use of the method described above presents us with a problem—namely, that for many real-world functions there is no known closed-form expression for the function's first or second derivatives. In these cases, we can estimate the derivative of the function numerically at individual points, by making use of repeated evaluation of the function at points very close in the function's domain. Although in principle this approach could be extended to evaluation of the second derivative, there are other methods for approximating the second derivative; we will postpone a discussion of numerical estimation of second derivatives to the next section.

Recall the definition of a derivative from elementary calculus:

$$f'(x) = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx} \tag{S2.7}$$

This immediately suggests a simple approximation:

$$f'(x) \approx \frac{f(x + dx) - f(x)}{dx} \tag{S2.8}$$

where the constant dx is ideally chosen to be very small, perhaps on the order of 1×10^{-6}. This approximation can be motivated by performing a *Taylor expansion* of $f'(x)$ and discarding all but the lowest-order terms:

$$f'(x) = \frac{f(x + dx) - f(x)}{dx} + \mathbb{O}(dx) \tag{S2.9}$$

where $\mathbb{O}(dx)$ denotes the higher-order terms which we discard in making the

approximation given by Equation (S2.8). By discarding these terms we are accepting an error in our approximation on the order of dx, as denoted by $\mathbb{O}(dx)$. In some cases this can result in a fairly large error, even for values of dx as small as 1×10^{-6}. The magnitude of this error can be reduced by simply retaining more of the terms from the Taylor expansion. A more accurate approximation is given by:

$$f'(x) \approx \frac{f(x+dx) - f(x-dx)}{2dx} \qquad \text{(S2.10)}$$

which has a smaller error of size $\mathbb{O}(dx^2)$. Although this latter formula involves the same number of function evaluations as Equation (S2.8) in the univariate case, we will see in the next section that in the multivariate case this latter formula incurs a significant computational cost over that of Equation (S2.8).

An even smaller error of order $\mathbb{O}(dx^4)$ can be achieved by utilizing even more of the terms from the Taylor expansion:

$$f'(x) = \frac{f(x-2dx) - 8f(x-dx) + 8f(x+dx) - f(x+2dx)}{12dx} + \mathbb{O}(dx^4) \qquad \text{(S2.11)}$$

though again we see that in doing so we have significantly increased the number of function evaluations needed to estimate the derivative. In the majority of real-world optimization tasks, it is the function evaluations which, by far, make up the great bulk of the computational cost of the optimization process. Gene-finding applications generally fit this trend, with likelihood computations for complex models and large training sets often incurring significant computational costs, especially when these operations must be performed very many times (i.e., hundreds or thousands) in optimizing a gene-finding system.

Returning to our two-point approximations given by Equations (S2.8) and (S2.10), it is important to note that the latter formula not only provides a smaller theoretical bound on the error of the approximation—i.e., $\mathbb{O}(dx^2)$ rather than $\mathbb{O}(dx)$, for $dx \ll 1$—but it also results in a formula involving what is known as a "central difference" rather than a "forward difference." That is, whereas the difference $f(x+dx)-f(x)$ involves an asymmetric interval $[x, x+dx]$ about x, the difference $f(x+dx)-f(x-dx)$ instead utilizes a symmetric interval $[x-dx, x+dx]$ about x.

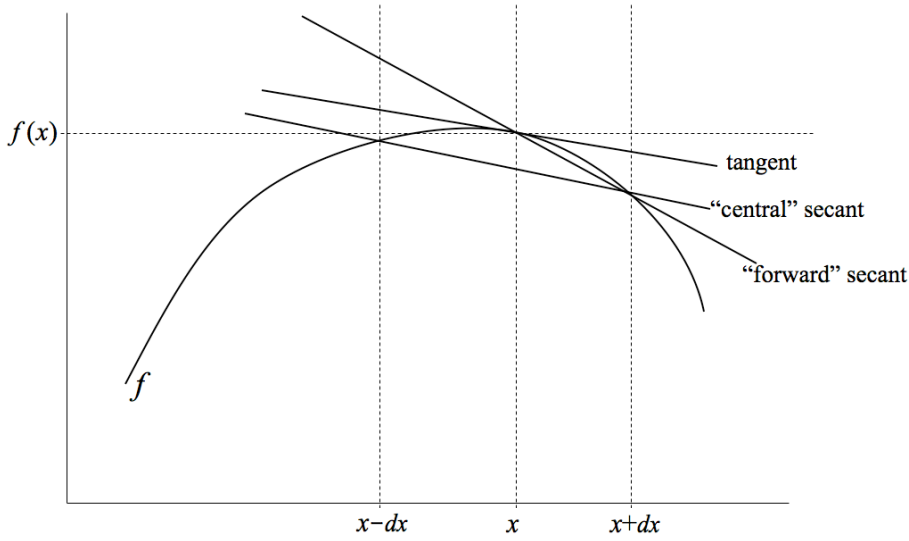

Figure S2.5 *Central versus forward secants. The central secant is defined by points (x-dx, f(x-dx)) and (x+dx, f(x+dx)), whereas the forward secant is defined by points (x, f(x)) and (x+dx, f(x+dx)). Although both secants approach the tangent in the limit dx→0, the central secant provides a more accurate approximation to the tangent for fixed dx.*

In Figure S2.5 we illustrate the potential advantage of central differences over forward differences. From the figure it can be seen that the secant line corresponding to the central differences more closely parallels the true tangent at x than does the secant for the forward differences. Although both secants converge on the true tangent as dx approaches zero, even for the relatively small values of dx typically used in numerical software one can see appreciable differences in approximation error between the forward and central secants.

As an example, we show in Table S2.1 the results of training several states of a *PhyloHMM* (see chapters 9 and S3) for the task of predicting the extents of 3′ UTRs in human protein-coding genes. As can be seen from the table, not only did the models trained using the central secants achieve a better fit to the training data, but when re-trained repeatedly over numerous training runs the central method produced a much smaller variation in the resulting log-likelihood.

We now move on to the more general case of multidimensional optimization.

	forward		central	
state	mean	std dev	mean	std dev
UTR	-199105	19238	-174709	0
DSE	-191300	15468	-181964	0
intergenic	-134059	9240	-129077	23
spacer	-158084	9219	-154525	0
USE	-173737	14929	-164994	6
cleavage	-122384	8404	-118764	0
hexamers	-55633	972	-55439	2

Table S2.1 *Log-likelihood values of optimal points for an evolutionary model of various genomic elements in 3′ UTR. Forward and central secants were used in estimating gradients.*

S2.2.2 Gradient Search in Multiple Dimensions

When the objective function $f(\mathbf{x})$ is defined on points $\mathbf{x}$ encapsulating several variables, we have the problem of *multivariate gradient search*. In this case the derivatives of the function are *partial derivatives*—i.e., derivatives $\partial f/\partial \mathbf{x}_i$ taken with respect to individual variables $\mathbf{x}_i$ in vector $\mathbf{x}$. Collecting these partial derivatives together into a vector gives us the *gradient*, $\nabla f(\mathbf{x})$:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial \mathbf{x}_0}, \frac{\partial f}{\partial \mathbf{x}_1}, \ldots, \frac{\partial f}{\partial \mathbf{x}_{n-1}}\right] \tag{S2.12}$$

for dimensionality n. Similarly, we can collect the *second partial derivatives* of f into a matrix $H=[z_{ij}]$ in which $z_{ij}=\partial^2 f/(\partial \mathbf{x}_i \partial \mathbf{x}_j)$. Such a matrix is called a *Hessian*, and is denoted $\nabla^2 f$. When the Hessian is evaluated at a particular point $\mathbf{x}$ we write $\nabla^2 f(\mathbf{x})$. Note that we will denote by $\mathbf{x}_i$ the i^{th} component of vector $\mathbf{x}$, and by $\mathbf{x}_{(k)}$ the k^{th} vector in a series of vectors.

By substituting the gradient for the first derivative and the Hessian for the second derivative, we can generalize Newton's update equation (Eq. S2.6) to the case of multivariate gradient ascent as follows:

$$\mathbf{x}_{(k+1)} = \mathbf{x}_{(k)} - \left(\nabla^2 f(\mathbf{x}_{(k)})\right)^{-1} \nabla f(\mathbf{x}_{(k)}), \tag{S2.13}$$

where $(\nabla^2 f(\mathbf{x}_{(k)}))^{-1}$ is the inverse of the Hessian matrix evaluated at point $\mathbf{x}_{(k)}$. The effect of using a Hessian in optimization is to allow a more accurate description of the local contour of the optimization landscape, since the Hessian corrects our estimate of the gradient to account for non-independence between dimensions of the search space.

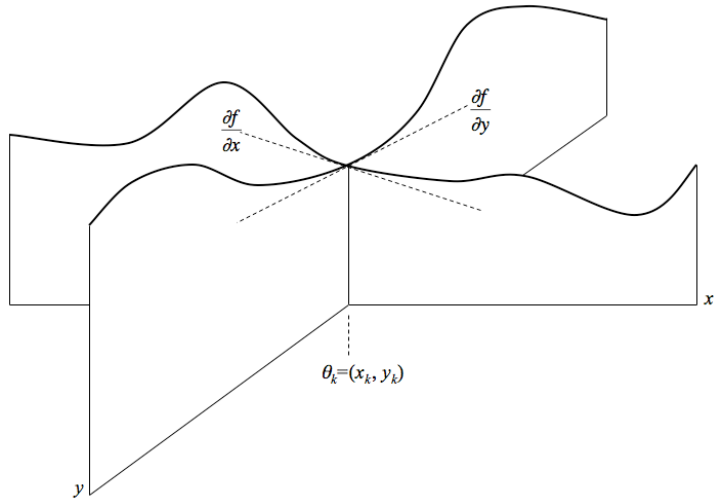

Figure S2.6 *Gradients in two dimensions. Each partial derivative can be thought of as a slope in one dimension. Taken together the gradient defines a hyperplane in multiple dimensions. Utilizing a Hessian matrix allows this flat hyperplane to be replaced with a gradient surface having curvature, and can result in more accurate steps during optimization.*

Because the Hessian can be difficult to compute in practice, it is generally approximated with a matrix **B** which is initialized to the identity matrix **I** and iteratively updated during the gradient-ascent process, as we will describe below. Furthermore, since inverting this approximate Hessian matrix can be computational expensive (and in some cases even impossible, due to *singularity* issues—see, e.g., Gill and King, 2004), in practice one generally approximates $\mathbf{B}^{-1}$ directly, rather than estimating **B** and then explicitly computing its inverse. We thus arrive at the following update equation:

$$\mathbf{x}_{(k+1)} = \mathbf{x}_{(k)} - \mathbf{B}_{(k)}^{-1} \nabla f(\mathbf{x}_{(k)}) \tag{S2.14}$$

Methods which utilize an approximate Hessian in this way are referred to as *quasi-Newton* methods. Note that $\mathbf{B}_{(k)}^{-1}$ is the k^{th} approximation to the inverse Hessian; this approximation is updated (i.e., refined) at each iteration of the optimization process, so that as optimization proceeds we can expect the approximation to become somewhat more accurate.

Since $\mathbf{B}_{(k)}^{-1}$ is an approximation, we can expect it (especially during the early steps of the optimization) to be fairly erroneous in many cases. Intuitively, the purpose of incorporating the second derivative into the gradient-ascent process was to provide a slight correction in the gradient

direction as indicated by the first derivative(s). Assuming that the direction indicated by $\mathbf{B}_{(k)}^{-1}\nabla f(x_{(k)})$ is roughly accurate (even if the magnitude is not exactly correct), it is reasonable to consider rescaling this term with a coefficient α, which effectively adjusts our *step-size* to account for errors in the approximation of $\mathbf{B}_{(k)}^{-1}$:

$$\mathbf{x}_{(k+1)} = \mathbf{x}_{(k)} - \alpha\mathbf{B}_{(k)}^{-1}\nabla f(\mathbf{x}_{(k)}) \tag{S2.15}$$

This coefficient is generally found by performing a *line maximization*—i.e., by performing a one-dimensional optimization of α along the direction of $\mathbf{B}_{(k)}^{-1}\nabla f(x_{(k)})$:

$$\alpha = \arg\max_{\alpha} f\left(\mathbf{x}_{(k)} - \alpha\mathbf{B}_{(k)}^{-1}\nabla f(\mathbf{x}_{(k)})\right) \tag{S2.16}$$

in which only α is permitted to vary during the line maximization. Once α has been determined to an acceptable precision (as defined by yet another threshold or "metaparameter") the original optimization problem is resumed by applying Eq. (S2.15) with the chosen α, to compute the next point, $\mathbf{x}_{(k+1)}$. Note that the line maximization process involves repeated evaluation of the objective function, and can therefore have an appreciable computational cost.

A popular instance of the above technique is the *BFGS* algorithm (named after its inventors: Broyden, Fletcher, Goldfarb, and Shanno). In this particular algorithm, the Hessian update rule (which is applied directly to the inverse of **B** rather than to **B** itself, thereby obviating the need to invert **B** numerically) is given by:

$$\begin{aligned}\mathbf{B}_{(k+1)}^{-1} = \mathbf{B}_{(k)}^{-1} &+ \frac{\mathbf{w}_{(k)} \otimes \mathbf{w}_{(k)}}{\mathbf{w}_{(k)} \cdot \mathbf{z}_{(k)}} - \frac{(\mathbf{B}_{(k)}^{-1} \cdot \mathbf{z}_{(k)}) \otimes (\mathbf{B}_{(k)}^{-1} \cdot \mathbf{z}_{(k)})}{\mathbf{z}_{(k)} \cdot \mathbf{B}_{(k)}^{-1} \cdot \mathbf{z}_{(k)}} \\ &+ (\mathbf{z}_{(k)} \cdot \mathbf{B}_{(k)}^{-1} \cdot \mathbf{z}_{(k)}) \cdot (\mathbf{r}_{(k)} \otimes \mathbf{r}_{(k)})\end{aligned} \tag{S2.17}$$

(Press *et al.*, 1992), where $\mathbf{M}=\mathbf{x}\otimes\mathbf{y}$ denotes the *outer product* of vectors **x** and **y** (i.e., $\mathbf{M}_{ij}=\mathbf{x}_i\mathbf{y}_j$), and $\mathbf{w}_{(k)}$, $\mathbf{z}_{(k)}$, and $\mathbf{r}_{(k)}$ are defined as:

$$\begin{aligned}\mathbf{z}_{(k)} &= \nabla f(\mathbf{x}_{(k+1)}) - \nabla f(\mathbf{x}_{(k)}), \\ \mathbf{w}_{(k)} &= \mathbf{x}_{(k+1)} - \mathbf{x}_{(k)}, \\ \mathbf{r}_{(k)} &= \frac{\mathbf{w}_{(k)}}{\mathbf{w}_{(k)} \cdot \mathbf{z}_{(k)}} - \frac{\mathbf{B}_{(k)}^{-1} \cdot \mathbf{z}_{(k)}}{\mathbf{z}_{(k)} \cdot \mathbf{B}_{(k)}^{-1} \cdot \mathbf{z}_{(k)}}\end{aligned} \tag{S2.18}$$

(Press *et al.*, 1992).

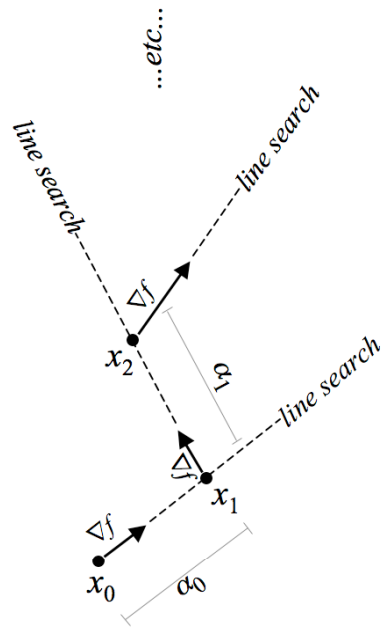

Figure S2.7 *Line maximization in quasi-Newton optimizers. At each new point a gradient provides a search direction. A line search is performed in that direction to find the optimal "step size"* α *for the current step.*

Given this particular Hessian update formula, the resulting procedure is known as the *BFGS algorithm*, and is very popular due to its demonstrated efficiency. Several variants of this algorithm are known: *L-BFGS*, called *limited-memory BFGS*, utilizes a memory-efficient heuristic in representing large Hessian matrices (important when working in high-dimensional parameter spaces); and *L-BFGS-B*, called *limited memory BFGS with boundary constraints* (Byrd *et al.*, 1995), which enforces arbitrary sets of boundary constraints of the form:

$$\forall_{0 \le i < n} \mathbf{a}_i \le \mathbf{x}_i \le \mathbf{b}_i \qquad \text{(S2.19)}$$

We will consider simple boundary constraints under BFGS for probabilistic (and other) models below.

First, let us investigate the advantage of higher-order methods such as BFGS over simple first-order methods. In Figure S2.8 we show the number of function evaluations required to train a PhyloHMM for an exon-recognition task, along with the log-likelihood value achieved after that many evaluations. As can be seen from the figure, not only was the computational cost of steepest-ascent generally higher than that of BFGS, but also the cost

of steepest-ascent increased more rapidly as the optimum was approached, with steepest ascent appearing to require an exponentially-increasing amount of computation while BFGS appeared to require only a linear or sub-linear increase in computation in approaching the optimum.

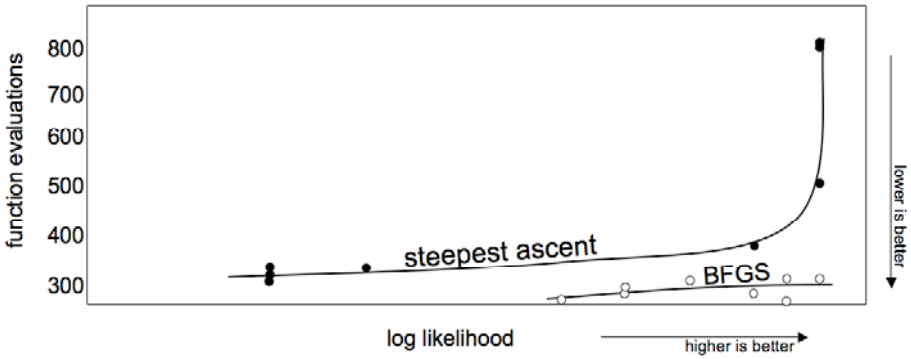

Figure S2.8 *Log-likelihood (x-axis) versus computational cost (y-axis) in terms of function evaluations for steepest-ascent (black dots) and BFGS (white dots) on a particular training task.*

Thus, BFGS appears to perform much more efficiently for this problem. However, on this particular problem even BFGS proved to be susceptible to the problem of local optima, as suggested by Figure S2.9. In this graph we show the maximum log-likelihood achieved during 25 separate runs of the optimizer from random initial points. As can be seen from the figure, there was wide variation in the likelihood of the model to which the optimizer converged, suggesting that for the problem of training a PhyloHMM on real genomic data, the optimization landscape may indeed be very rugged, with many local optima.

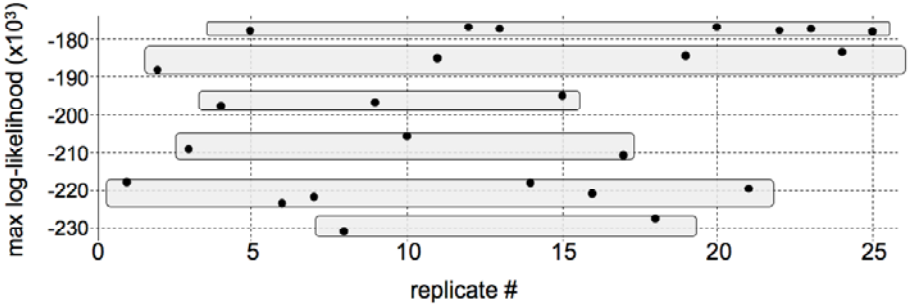

Figure S2.9 *Log-likelihoods of trained models during twenty-five replicates of a training run for a PhyloHMM, using a BFGS optimizer. Values in a similar range are grouped together to suggest possible local optima.*

Optimization software based on the foregoing methods is readily available from various sources. The distribution due to *Press et al.* (1992) is very popular but may be subject to some restrictions due to copyright and/or licensing issues. The Gnu Scientific Library, or GSL, described in section 10.15, includes implementations of BFGS and several other second-order methods. Another library, ALGLIB (Bochkanov and Bystritsky, 2007), provides implementations of L-BFGS and L-BFGS-B.

S2.2.3 Constrained Optimization

As noted earlier, special modifications can be made to existing optimization procedures so as to respect constraints on the allowable values taken by parameters of a function to be optimized. In the case of probabilistic gene-finders, this is clearly a necessary refinement, since we will generally require $0 \leq p \leq 1$ for all parameters p which are interpreted as probabilities. When using a general-purpose (unconstrained) optimizer, special measures must be taken whenever parameter values provided by the optimizer to the objective function are outside their legal range (i.e., probabilities less than 0 or greater than 1) since the objective function will generally be undefined at those points.

Enforcing boundary constraints on probabilistic parameters with an unconstrained optimizer can be achieved quite simply by employing a *squashing function*, as depicted in Figure S2.10.

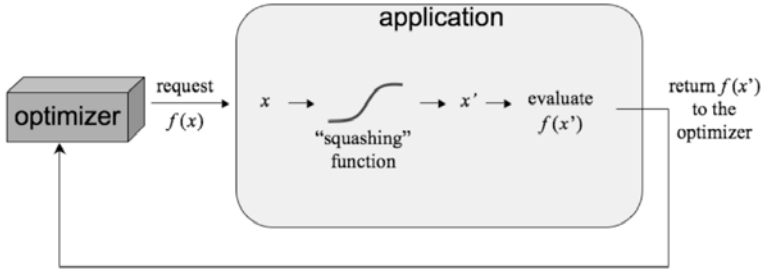

Figure S2.10 *Constraining parameters via a sigmoidal function.*

Given a univariate objective function $f(x)$ and a squashing function $\sigma(x): \mathbb{R} \mapsto [0,1]$ mapping arbitrary real values into the interval [0,1], we can simply adopt the strategy of, when asked by the optimizer to evaluate $f(x)$, instead evaluating $y=f(\sigma(x))$ and returning y to the optimizer in place of the requested $f(x)$. Likewise, when asked by a quasi-Newton optimizer for the derivative $f'(x)$ we can instead return the value $f'(\sigma(x))$. Note that such a transformation is useful even when the given x value is within its legal range, since the values $f(x+dx)$ and $f(x-dx)$ might otherwise be undefined when x is 1 or 0. Upon convergence of the optimizer, the chosen optimum x returned

by the optimizer can again be mapped to x^* via $x^*=\sigma(x)$. The generalization of this strategy to the multivariate case is trivial: we merely apply these methods separately to the individual components of the parameter vector **x**.

A useful σ function for constraining probabilistic parameters is given by $x'=tan^{-1}(x)/\pi+\frac{1}{2}$, which is illustrated in Figure S2.11.

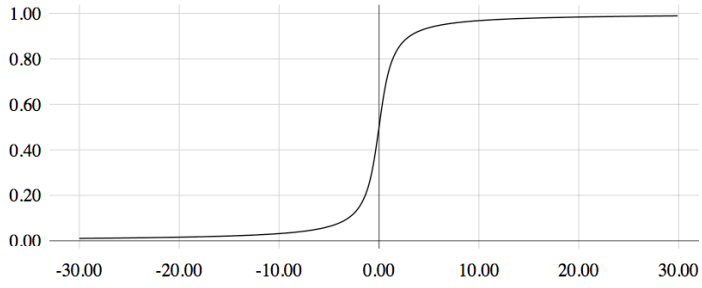

Figure S2.11 *Sigmoidal function* $tan^{-1}(x)/\pi+\frac{1}{2}$.

Other σ functions may be easily derived for intervals other than [0,1], though in practice one should strive to adopt a function which is smooth, to avoid any adverse effects on the transformed optimization landscape. It is also important to keep in mind that the optimizer will be working only with the transformed landscape, so that meta-parameters such as ε which are provided to the optimizer should be appropriately scaled for that transformed landscape. Note that if the optimizer is to be provided an initial starting point, then σ will need to be invertible, so that the desired initial point x' can be transformed into $x=\sigma^{-1}(x')$ to be provided to the optimizer.

Other options in constraining the optimizer include the use of *Lagrange multipliers*, *penalty functions*, and *barrier functions*. We refer the interested reader to the ample literature on these topics (e.g., Boyd and Vandenberghe, 2004).

S2.2.4 Tips to Improve Search Efficiency

Black-box optimization can be painfully slow. In practice there are a number of ways that an optimizer might be accelerated. We have found the following methods to substantially reduce the time required to train a model when using black-box optimizers.

The most effective way to reduce the time required for optimization (apart from adopting a more efficient optimization algorithm) is to either reduce the number of function evaluations, or to reduce the amount of time required for each function evaluation (i.e., to increase the efficiency of the software which evaluates the function being optimized). We will consider several methods aimed at achieving both of these goals.

Reducing the number of function evaluations can, surprising, be achieved in many cases by simply employing a *cache*. Many optimization software packages do not employ an internal cache of their own, so that the optimizer can (and often will) request function evaluations for points **x** that have already been evaluated previously by the objective function (Wolpert and Macready, 1997). Once $y=f(\mathbf{x})$ has been evaluated, y can be stored in a cache indexed by **x**, so that the next time the optimizer requests $f(\mathbf{x})$ we can simply return the corresponding value y found in the cache rather than evaluating $f(\mathbf{x})$ again. Figure S2.12 illustrates this trivially simple scheme.

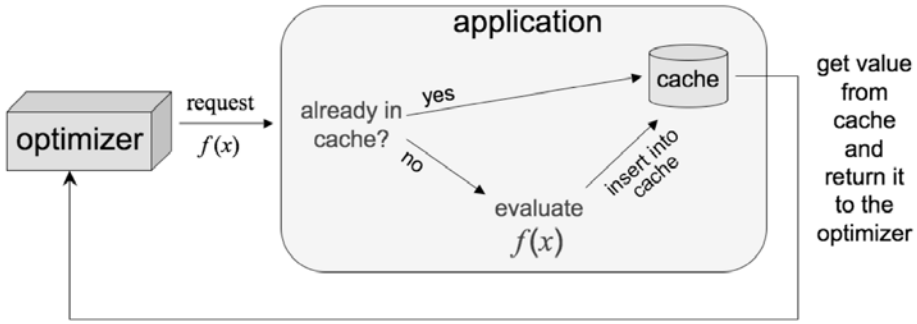

Figure S2.12 *Employing a cache to reduce the number of function evaluations.*

As an example, during one run of a PhyloHMM training process using the L-BFGS-B optimizer from the ALGLIB library, out of 157 340 evaluation requests from the optimizer, 26 985 of the requested **x** values were found to be present already in the cache, allowing a 17% reduction in the number of function evaluations which needed to be performed.

Another method which can sometimes reduce the number of required function evaluations quite substantially is that of *hierarchical training*. We first define the notion of *model equivalence*. Let us suppose two models M_1 and M_2 having parameterizations θ_1 and θ_2 have the property that whenever $\theta_1=\theta_2$, the two models so parameterized will produce identical behavior—i.e., $M_1(\mathbf{x};\theta_1)=M_2(\mathbf{x};\theta_2)$ for all **x** in the domain of M_2. We say models M_1 and M_2 are *equivalent*. Now consider the case in which $\theta_1=(p_1, p_2, \ldots, p_n)$ and $\theta_2=(q_1, q_2, \ldots, q_m)$, for $n<m$. If it is possible to fix $m-n$ of the parameters in M_2 so as to arrive at a model M_2' having n free parameters such that M_2' is equivalent to M_1, then we say that M_1 is *nested within* M_2, or that M_2 *generalizes* M_1. Given such a pair of untrained models M_1 and M_2, an obvious approach to training M_2 is to first train M_1 and then to use the estimates for M_1's parameters to "seed" M_2 prior to training the remaining parameters of M_2:

$$\text{step 1: } (p_1^*,...,p_n^*) = \arg\max_{p_1,...,p_n} f(M_1(p_1,...,p_n))$$

$$\text{step 2: } (q_{n+1}^*,...,q_m^*) = \arg\max_{q_{n+1},...,q_m} f(M_2(p_1^*,...,p_n^*,q_{n+1},...,q_m))$$

$$\text{step 3: } \theta_2^* = (p_1^*,...,p_n^*,q_{n+1}^*,...,q_m^*)$$

with θ_2^* constituting the final model parameters to be used when deploying model M_2.

Alternatively, we can use M_1's parameter estimates as initial values for the corresponding parameters in M_2, with the remaining parameters being initialized randomly as usual. Training of M_2 would then proceed normally—i.e., allowing all parameters of M_2 to vary during gradient ascent.

Generalizing this scenerio, it is possible to have a hierarchy of models M_1, M_2, ... , M_N such that M_i is nested within M_{i+1}. In such cases we may apply the above strategy sequentially to successively more general models, resulting in what may be termed *hierarchical training* (see Fig. S2.13)

Hierarchical training can be applied to a wider variety of models if we relax our definition of nested models somewhat, so that M_1 is considered to be nested within M_2 if for any θ_1 there exists a θ_2 such that the models so parameterized are equivalent. Under this relaxed definition it should be apparent that higher-order Markov chains (section 6.7) constitute hierarchical models, as do some varieties of context-dependent PhyloHMMs as described in supplementary chapter S3. In the case of context-dependent PhyloHMMs, we have found that hierarchical training can sometimes accelerate the training process by as much as 35%, as compared to training the composite model from scratch.

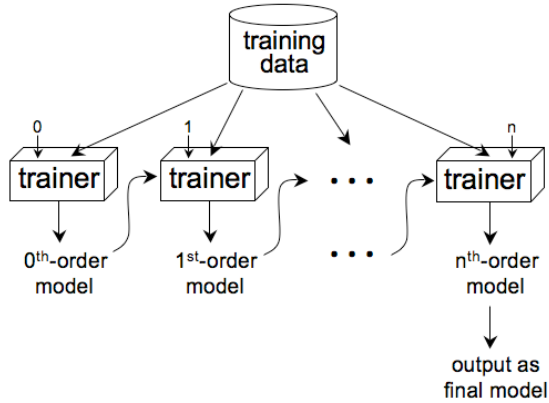

Figure S2.13 *Hierarchical training of nested models. Each successive model serves as a "seed" or starting point for the optimization of the next higher model in the hierarchy.*

Finally, we consider the problem of improving the efficiency of individual function evaluations. Although this is largely a function-specific task, some general advice can be given here as well. Parallelization of the objective function (section 12.5) is one approach that can often produce substantial efficiency gains. For functions which are not easily parallelizable but which have a fairly high *arity* (i.e., number of parameters), a significant speed-up can be achieved during gradient evaluation by assigning evaluation of each partial derivative in the gradient to a separate processor. Alternatively, for objective functions which can be reasonably factored according to position in the input DNA sequence, work can be divided between processors at the level of individual training sequences—i.e., individual contigs or portions of contigs (with the latter possibly involving independence assumptions when recombining scores from the same contig).

Apart from parallelization, other approaches to improving efficiency include the obvious methods employed in any software—e.g., avoiding dynamic memory allocation within loops, utilizing efficient data structures, etc.

S2.3 Optimization without Gradients

The use of gradients during optimization involves an unfortunate tradeoff—namely, that while the gradients can inform the optimization process and thereby (in many cases) accelerate that process by reducing the number of steps needed to converge (as compared to blind hill-climbing), evaluating the gradients can be very time-consuming, especially for functions taking many parameters (since a partial derivative must be evaluated for each parameter).

A number of non-gradient-based optimization techniques have been shown to be effective for many applications, and in some cases these alternative methods can be somewhat less susceptible to the problem of converging to a non-global optimum. Two such methods were mentioned in chapter 10: *genetic algorithms* and *simulated annealing*. The former method maintains a population $Q=\{\theta_1,...,\theta_n\}$ of model parameterizations and attempts to "evolve" that population (via appropriately-chosen mutation and recombination operators) so as to improve the average population "fitness" (i.e., the average value of the objective function). Simulated annealing instead maintains a single θ which is successivly perturbed so as to improve the value of the objective function (i.e., climbing uphill), while occasionally accepting a perturbation which results in a somewhat worse evaluation under the objective function (i.e., occasionally stepping downhill). Both methods differ from strict hill-climbing and gradient ascent by sometimes allowing movements over the optimization landscape which are not strictly uphill, in hopes that these occasional movements may allow the optimizer to escape a local optimum so as to find more promising regions of the optimization landscape.

A method which is vaguely reminiscent of a genetic algorithm is the *downhill simplex* method (not to be confused with the well-known "simplex" method for linear programming) due to Nelder and Mead (1965). Rather than computing an n-dimensional gradient at the current point, this method maintains $n+1$ current points and utilizes the relative differences in their objective-function values to infer a promising direction for further search. Yet another approach is *Powell's method* (not to be confused with the *Davidon-Fletcher-Powell* method, which is similar to BFGS), in which each dimension is searched separately via simple line-maximization—i.e., the algorithm performs a separate line maximization along each axis in the parameter space, in turn, cycling repeatedly through the set of all axes until convergence. The simplex and Powell methods tend to require more function evaluations than gradient-based methods, though they are simpler to implement and can perform surprisingly well for some problems (Press *et al.*, 1992).

For bioninformatic applications, the most popular non-gradient method for optmization is the *expectation maximization* procedure for finding an MLE parameterization of a probabilistic model. We describe this next in some detail.

S2.3.1 Expectation Maximization

Expectation Maximization, or *EM*, is especially popular in bioinformatics applications. EM is in fact a form of hill-climbing, and although the method generally does not require explicit computation of gradients, it can be very efficient. The method is explicitly geared toward probabilistic models, and indeed it is well-known as a form of maximum likelihood estimation (MLE). Like any hill-climbing method, EM is guaranteed only to converge to a local optimum. Furthermore, the method places some requirements on the objective function; in particular, it is necessary before applying the method to derive a set of so-called *update equations*, algebraically, from the objective function. Since gradients need not be explicitly computed (under ideal circumstances), each iteration of the optimization process can often be faster than with a general-purpose gradient-ascent algorithm. Furthermore, each iteration of EM is guaranteed to improve the likelihood of the model being trained, unlike the case with general gradient-ascent procedures, in which an inappropriately large step size can quite easily result in function evaluations at points with poorer likelihoods than points evaluated earlier in sequence. Indeed, an especially convenient feature of EM is the virtual lack of any *meta-parameters* such as, e.g., step size, dx, etc. The only meta-parameter required in the general case is a convergence threshold, ε. In chapter 6 we considered a special case of EM: the Baum-Welch algorithm for MLE estimation of HMM parameters. A slightly modified version of EM can also be used for MAP estimation (see Dempster *et al*., 1977).

A common scenario in problems for which EM is typically employed

involves models with *unobservables*—i.e., variables having an unknown value. In the case of HMM's, the DNA sequence would constitute the series of *observables*, while the labels of the individual states emitting each nucleotide would form the series of unobservables. A particularly attractive feature of EM is that it provides a method for estimating model parameters even in the presence of unobservable variables.

Formally, define the log-likelikood function $L(\theta)$ for a single training example as:

$$L(\theta) = \log P(\mathbf{X}|\theta) = \log \sum_{\mathbf{Y}} P(\mathbf{X}, \mathbf{Y}|\theta), \tag{S2.20}$$

for observables **X**, unobservables **Y**, and model θ. That is, given a vector **X** of observable variables (such as nucleotides emitted by an HMM) and their fixed values, and a vector **Y** of unobservable variables (such as state labels in an HMM) for a single training instance, we can compute the likelihood of the observables under the model θ by marginalizing over all possible assignments to the unobservables. This can obviously be generalized via simple summation to include multiple training examples, though we omit the extra summation for simplicity.

Dempster *et al*. (1977) introduced the Q function[2]:

$$Q(\theta_{k+1}|\theta_k) = \sum_{\mathbf{Y}} P(\mathbf{Y}|\mathbf{X}, \theta_k) \log P(\mathbf{X}, \mathbf{Y}|\theta_{k+1}), \tag{S2.21}$$

which we use to define another useful function, V:

$$V(\theta_{k+1}|\theta_k) = L(\theta_k) + Q(\theta_{k+1}|\theta_k) - Q(\theta_k|\theta_k). \tag{S2.22}$$

Two useful properties of V are:

$$L(\theta_{k+1}) \geq V(\theta_{k+1}|\theta_k) \tag{S2.23}$$

(via *Jensen's inequality* and the convexity of the *log* function—see Borman, 2006) and:

$$L(\theta_k) = V(\theta_k|\theta_k) \tag{S2.24}$$

(via simple algebra). That is, V is bounded by L. Thus, if we can find a θ_{k+1} for which $V(\theta_{k+1}|\theta_k) > V(\theta_k|\theta_k)$, we will have:

[2] In the original paper, Dempster *et al.* omit the **X** (which they call **y**) from the final term, since it is implicit in their definition of **Y**. Our formulation is thus equivalent.

$$L(\theta_{k+1}) \geq V(\theta_{k+1}|\theta_k) > V(\theta_k|\theta_k) = L(\theta_k), \tag{S2.25}$$

and therefore by transitivity:

$$L(\theta_{k+1}) > L(\theta_k). \tag{S2.26}$$

Thus, by increasing V (i.e., between iterations k and $k+1$) we are guaranteed to increase L. Furthermore, if we iteratively *maximize* V rather than merely increasing it (for successive values of k), we will be effectively hill-climbing on L:

$$\theta_{k+1} = \arg\max_{\theta} V(\theta|\theta_k), \tag{S2.27}$$

which it can be seen from Eq. (S2.22) is equivalent to:

$$\theta_{k+1} = \arg\max_{\theta} Q(\theta|\theta_k), \tag{S2.28}$$

since $L(\theta_k)$ and $Q(\theta_k|\theta_k)$ do not change over the argmax.

EM is therefore defined as an iterative algorithm in which at each step we compute:

$$\theta_{k+1} = \arg\max_{\theta} \sum_{\mathbf{Y}} P(\mathbf{Y}|\mathbf{X},\theta_k) \log P(\mathbf{X},\mathbf{Y}|\theta). \tag{S2.29}$$

Figure S2.14 illustrates the relationship between $L(\theta)$ and $V(\theta|\theta_k)$ for the case of a single-dimensional θ, which is associated with the x-axis. The solid curve shows $L(\theta)$ and the dashed curve shows $V(\theta|\theta_k)$. Note that $V(\theta|\theta_k)$ is considered a function of θ here, since θ_k is fixed. The figure can be thought of as a snapshot of the EM algorithm at step k. Note that $L(\theta)$ is higher than $V(\theta|\theta_k)$ at all points except θ_k.

More importantly, we see that for the point θ_{k+1} which maximizes $V(\theta|\theta_k)$—and therefore also maximizes $Q(\theta|\theta_k)$—we have $L(\theta_{k+1})>L(\theta_k)$, so that by maximizing $Q(\theta|\theta_k)$ we are guaranteed to increase L (or at least not to decrease it if we are already at the maximum). Furthermore, note that the shape of $V(\theta|\theta_k)$ is determined by the fixed parameter θ_k, so that it will have a different shape (and a different maximum) on each successive iteration. From this it should be intuitively obvious that the algorithm should converge to a local maximum θ^* of the likelihood function; formal convergence proofs are easily found in the literature and will not be reproduced here.

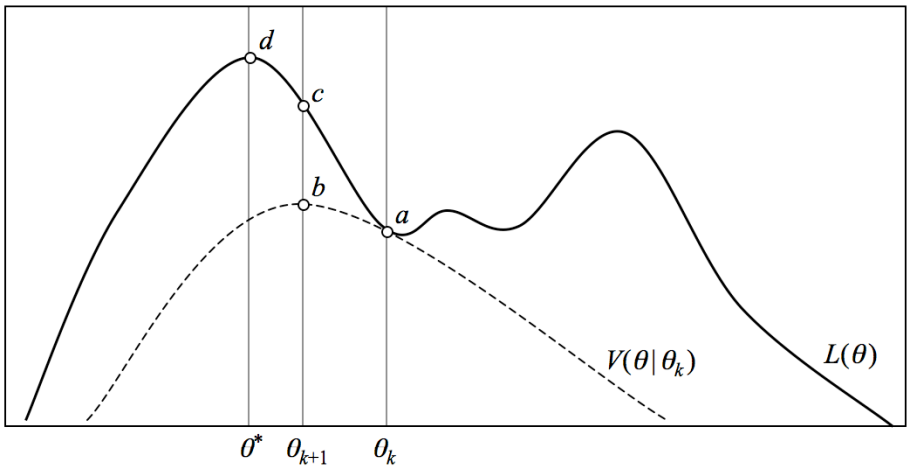

Figure S2.14 *The relationship between L and V (adapted from Borman, 2006) during the k^{th} iteration of EM. Point c is at least as high as point b, which is at least as high as point a, showing that by maximizing Q (and therefore V) we are guaranteed to take a non-decreasing step on L. The maximum of L, at point d, is the ultimate goal.*

Algorithm S2.1 Expectation-Maximization. A model θ is constructed which maximizes the likelihood of data set **X** to within a tolerance of ε. The problem-specific function Q must be provided as well.

```
procedure EM(X,ε,Q)
1. θ0←random();
2. k←0;
3. repeat:
4.    θk+1=argmaxθ(Q(θ|θk));
5.    k=k+1;
6. until P(X|θk)-P(X|θk-1)<ε;
7. return θk;
```

The exact form of the P terms in Eq. (S2.21) will depend on the type of model being trained. Eq. (S2.28) is known as the *EM update equation*. Starting with θ_0 initialized to some random parameterization[3], we apply the

[3] Technically, initializing to a random point could result in convergence to a minimum or a saddle point instead of a maximum, though these cases appear to be exceedingly rare (see, e.g., Carreira-Perpiñán, 2007).

EM update equation to compute θ_{k+1} for successively higher k. When the log-likelihood of the model is no longer changing appreciably, we stop, taking θ_{k+1} as the trained model (for the largest k+1 computed). Algorithm S2.1 summarizes the procedure.

It is sometimes lamented by those newly introduced to EM that the procedure merely replaces one maximization with another—namely, the maximization of L with the maximization of Q (i.e., line 4 in Algorithm S2.1). The standard reply to this objection is that Q is often easier to maximize analytically than L.

As an example, let us consider the case of HMMs. The following is adapted directly from (Durbin *et al.*, 1998):

$$\begin{aligned} Q(\theta_{k+1}|\theta_k) &= \sum_{\phi} P(\phi|S,\theta_k)\log P(S,\phi|\theta_{k+1}) \\ &= \sum_{i=1}^{N-1}\sum_{x\in\alpha} E_{i,x}\log e_{i,x} + \sum_{i=0}^{N-1}\sum_{h=0}^{N-1} A_{i,h}\log a_{i,h} \end{aligned} \tag{S2.30}$$

where $A_{i,j}$ and $E_{i,j}$ are the expected emission and transition counts computed via *forward-backward* (Algorithm 6.7 in section 6.6.3), $a_{i,j}$ and $e_{i,j}$ are the corresponding emission and transition probabilities, and N is the number of states in the model (q_0 being the silent initial/final state).

It can be shown that this Q is maximized by simply choosing:

$$\theta_{k+1} = \left\{ e_{i,x}^{k+1}, a_{i,j}^{k+1} \middle| x\in\alpha, 0\le i\le N, 0\le j\le N \right\}$$

for:

$$e_{i,j}^{k+1} = \frac{E_{i,j}}{\sum_{h=0}^{|\alpha|-1} E_{i,h}}, \qquad a_{i,j}^{k+1} = \frac{A_{i,j}}{\sum_{h=0}^{N-1} A_{i,h}} \tag{S2.31}$$

(see, e.g., Durbin *et al.*, 1998). Thus, in the case of HMM's, maximizing Q is trivial, whereas directly maximizing L could be very difficult. To see that this choice of parameters will indeed maximize Q, consider the difference between the chosen θ_{k+1} and an alternative θ:

$$\begin{aligned} &Q(\theta_{k+1}|\theta_k) - Q(\theta|\theta_k) = \\ &\sum_{i=1}^{N-1}\left(\sum_{y\in\alpha} E_{i,y}^{k}\right)\sum_{x\in\alpha} e_{i,x}^{k+1}\log\frac{e_{i,x}^{k+1}}{e_{i,x}} + \sum_{i=0}^{N-1}\left(\sum_{j=0}^{N-1} A_{i,j}^{k}\right)\sum_{h=0}^{N-1} a_{i,h}^{k+1}\log\frac{a_{i,h}^{k+1}}{a_{i,h}} \end{aligned} \tag{S2.32}$$

(Durbin *et al.*, 1998) for $\theta=\{e_{\cdot,\cdot}, a_{\cdot,\cdot}\}$. The log-sum terms:

$$\sum_{x \in \alpha} e_{i,x}^{k+1} \log \frac{e_{i,x}^{k+1}}{e_{i,x}} \tag{S2.33}$$

and

$$\sum_{h=0}^{N-1} a_{i,h}^{k+1} \log \frac{a_{i,h}^{k+1}}{a_{i,h}} \tag{S2.34}$$

are *relative entropies* (section 2.10), which are always non-negative; the $E_{i,y}^{k}$ and $A_{i,j}^{k}$ terms are also non-negative, and are constant for any fixed θ_k. Thus, $Q(\theta_{k+1}|\theta_k) > Q(\theta|\theta_k)$ for any θ other than θ_{k+1}, so $Q(\theta_{k+1}|\theta_k)$ is maximal for fixed θ_k.

Deriving EM update equations for other types of models can sometimes be accomplished analytically by setting the derivative (or gradient) of Q equal to zero and solving for those critical points θ^* where the second derivative (or Hessian) is negative:

$$\nabla_\theta Q(\theta|\theta_k) = \frac{d}{d\theta} \sum_{\mathbf{Y}} P(\mathbf{Y}|\mathbf{X},\theta_k) \log P(\mathbf{X},\mathbf{Y}|\theta) = \mathbf{0} \tag{S2.35}$$

$$\nabla_\theta^2 Q(\theta|\theta_k) < \mathbf{0} \tag{S2.36}$$

When this maximization is not easy to accomplish analytically, it is possible to do the maximization numerically, using a generic gradient ascent procedure. This is the approach taken by Siepel and Haussler (2004c) in the training of the PhyloHMM *ExoniPhy*.

Note that it is not necessary to *maximize* Q at each step; just *increasing* Q at each step will ensure that the log-likelihood also increases. When Q is only increased (rather than maximized) at each step, the resulting algorithm is termed a *Generalized EM* algorithm, or *GEM* (Dempster *et al.*, 1977).

At this juncture is it worthwhile for us to draw attention to a common misconception regarding the EM algorithm. The algorithm itself is often described as a two-step process (steps "E" and "M") which "*alternates between estimating the unobserved variables given the current model and refitting the model using the estimated, complete data"* (Redner and Walker, 1984). While this description might be applicable in *specific* cases, it is not true in *all* cases. In the case of HMM's the math does in fact work out such that the "E-step" involves finding the expectations of the emission and transition counts over all possible parses of the input sequence. Likewise, in the example problem given in Dempster *et al.*'s original paper, an "E-step" was identifiable as the estimation of certain unobserved quantities (i.e.,

"sufficient statistics" for an exponential distribution).

Recall that EM is stated simply as a maximization:

$$\theta_{k+1} = \arg\max_{\theta} \sum_{\mathbf{Y}} P(\mathbf{Y}|\mathbf{X},\theta_k) \log P(\mathbf{X},\mathbf{Y}|\theta) \qquad \text{(S2.37)}$$

Depending on the detailed form of Q, it is *sometimes* possible to factor this equation so that the right-hand side is transformed into an expression involving the expected values of easily-computed quantities (such as emission and transition counts in an HMM); however, this is not necessarily always the case.

Furthermore, the simplistic (though intuitively appealing) strategy of repeatedly alternating between estimating fixed values for unobservable data and then re-fitting the model parameters to the combined observed and unobserved data is not (at least in all cases) equivalent to EM, and may in some cases not even converge to a local optimum. In the case of HMM's, *Viterbi training* (section 6.6.1) can be seen to constitute an instance of the former strategy by inferring the unobserved state labels and then estimating the model parameters from these; differences in the efficacy of Viterbi training versus EM for HMM's have been well-documented (e.g., Rodríguez and Torres, 2003).

Quite a bit has been written concerning the relationship between EM and general gradient ascent. According to Redner and Walker (1984), EM "takes discrete steps in parameter space similar to a first order method operating on the gradient of a locally reshaped likelihood function." Similarly, Salakhutdinov *et al.* (2004) state that for "most objective functions, the EM step ... in parameter space and true gradient can be related by a transformation matrix ... that changes at each iteration." From this one is tempted to conclude that EM should be inferior to second-order methods (such as those described earlier in this chapter). However, Xu and Jordan (1995) state that EM "can be viewed as a variable metric gradient ascent algorithm for which the projection matrix ... changes at each iteration as a function of the current parameter value[s]" and, further, that "the EM algorithm can be viewed as a special form of quasi-Newton method in which the projection matrix ... playes the role of [the estimate of the inverse Hessian matrix]," implying that the method is in fact more similar to second-order methods such as BFGS. Note, however, that whereas quasi-Newton methods typically need to perform a line search in the gradient direction before re-computing the gradient, EM requires no such line search at each step. Indeed, since EM requires no user-defined "step size" metaparameter, one can think intuitively of the optimization of Q at each iteration as automatically choosing an "optimal" step size at each step.

S2.4 Avoiding Local Optima

Many real-world optimization landscapes are rugged—that is, they have many local optima which are not globally optimal. Gradient ascent methods, as well as those based on EM, are all susceptible to the problem of local optima. We will briefly consider several methods for rectifying this problem.

Recall from section S2.2 that gradient-based methods generally assume that we are starting from a point x_0 which is known to be near the desired optimum x^*. Unfortunately, this is rarely a valid assumption. One common technique aimed at rectifying this problem is to run a given optimization procedure multiple times, each from a different random starting point. For a given degree of ruggedness in the landscape and a given sampling density (i.e., number of random re-starts), we might reasonably expect to find the global optimum of the function being optimized. An alternative to random restarting is (obviously) that of systematic restarting, in which the initial points of the successive searches are chosen to exhaustively cover a lattice which samples the parameter space at discrete intervals. For high-dimensional parameter spaces this is simply not feasible for most problems of practical significance. As we noted earlier, an alternative to sampling the parameter space during initialization of the search is to allow occasional "lateral jumps" or "large-scale jumps" over the optimization landscape during hill-climbing, rather than strictly climbing uphill at every iteration. This approach is embodied to varying degrees in simulated annealing, genetic algorithms, and several other, less popular methods. Unfortunately, the effectiveness of any of these approaches will depend to a very large degree on the specific optimization landscape, so that concrete guidance in this regard is difficult to offer for the general case.

A somewhat related issue is the avoidance of *over-training*. Although it is tempting to think of the process of optimization as one of exploring the optimization landscape defined by the problem space, in reality we can only explore the landscape induced by the training set, which is only an approximation to the true landscape of the problem space. Given a reasonably large but finite training set, we can expect that the induced landscape will resemble the true landscape in its overall contour, but that minute details may differ to some degree. Thus, an optimal point on the *induced* landscape may not perfectly coincide with an optimum on the *true* landscape, and in some cases such an induced optimum may correspond to a grossly sub-optimal point on the real landscape.

One strategy aimed at accommodating these issues is called *early stopping*, in which the optimizer is terminated somewhat before it has converged to a local optimum. The hope is that by terminating the optimizer before it has converged, we will be stopping it before it has had a chance to finely adapt the trained model to the minute details of the induced landscape, since those minute details will often reflect sampling error rather than

reflecting minute details of the true landscape. To the extent that the overall contour of the induced landscape matches the true landscape, a point visited by the optimizer shortly before it has fully converged to a local optimum may be a fairly good point on the true landscape.

A common early-stopping approach involves the use of a "hold-out" set, H. During training, whenever the value of the objective function is observed to increase on the training set T but to decrease on the hold-out set H, we can decide to halt the training under the assumption that further iterations will only result in over-training of the model to the minute details of T which are not representative of the problem space in general.

S2.5 The Importance of Choosing an Appropriate Objective Function

As noted earlier, there is more than one option for the objective function when estimating parameters for probabilistic models. Although maximum likelihood is very popular for generative models, when the model is to be utilized for a prediction task we generally wish to maximize the expected accuracy of the resulting model on unseen data. In section 12.4 we described this issue as the distinction between generative and discriminative modeling. Here we briefly provide an example illustrating the difference for a simple parsing model.

In Figure 6.3 we showed a rudimentary HMM (model H_3) for gene parsing; this four-state HMM includes states for coding exons, introns, and intergenic regions, as well as a silent initial/final state. For the purposes of the present discussion we trained this model on a set of *Arabidopsis thaliana* genes, using the maximum likelihood techniques described in chapter 6. We denote this trained model M_{MLE}. We then used M_{MLE} to generate (i.e., by running the model and observing its emissions) a large quantity (several Mb) of synthetic DNA sequence, denoted S_{synth}. Because we also observed the transitions utilized by M_{MLE} during its generation of S_{synth}, we know the "correct" parse ϕ_{true} for S_{synth}—i.e., the precise coordinates within S_{synth} of the coding exons generated by M_{MLE}'s coding exon state.

We then used M_{MLE} to parse S_{synth} (via Viterbi decoding), producing a predicted parse ϕ_{MLE}. Because M_{MLE} generated S_{synth}, we expect that ϕ_{MLE} should be quite accurate, since the model used to compute the Viterbi parse ϕ_{MLE} was the very same model used to generate the sequence being parsed; indeed, the parses ϕ_{MLE} and ϕ_{true} were both produced from M_{MLE}—the former via Viterbi decoding and the latter by observing the state sequence during generation of S_{synth}. Comparing ϕ_{MLE} to ϕ_{true}, we found that the predicted parse was 78% accurate that the nucleotide level.

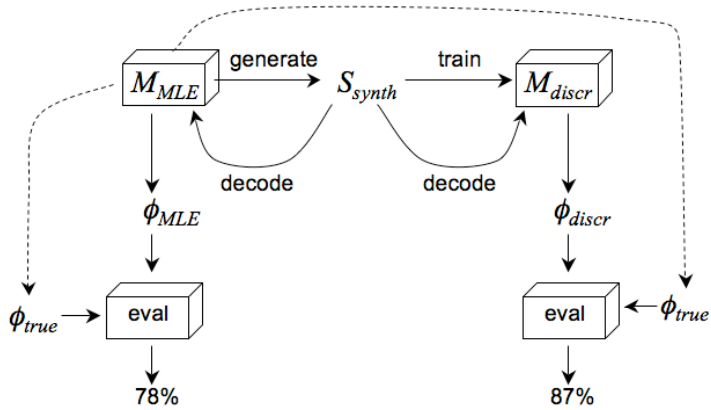

Figure S2.15 *A simple experiment demonstrating the difference between objective functions for training a parsing model. The MLE model, though used to both generate and parse the same data, was unable to match the parsing accuracy of the model trained with a more discriminative objective function.*

We then trained an identically structured model, M_{discr}, using a discriminative approach—i.e., we sought the parameters θ_{discr} which would maximize the nucleotide-level parsing accuracy of M_{discr} when the model was used to parse S_{synth}. Thus, while M_{discr} has an identical structure (i.e., number of states and pattern of transitions) to M_{MLE}, the transition and emission parameters of the two models differ. To our surprise, we found that the parsing accuracy of M_{discr} on S_{synth} was 87%—a full 9% higher than that of the model which actually generated the sequence.

Tables S2.2 and S2.3 show the differences between model parameters for the two models. Interestingly, these differences are consistent with the notion that superior discrimination requires emphasis on those features which most reliably separate the classes of interest (see, e.g., the discussion of *support vector machines* in section 10.14). In this case, the *Arabidopsis* training genes showed both lower `T` content and higher `C` and `G` content in the exonic regions versus the other two regions, and these biases, which were already present in the same proportions in the MLE model are obviously exaggerated in the discriminative model. The most extreme exaggeration, the -12% for `A` in the intergenic state, is among the most subtle differences in the training data, in which the intergenic regions showed ~29% `A` versus ~27% in the exonic and intronic regions; in this way, the discriminative trainer has identified a subtle but *consistent* difference and boosted the "weight" of this feature in the model by exaggerating the corresponding emission probabilities.

		symbol			
		A	C	G	T
state	1	−12%	+5%	−3%	+10%
	2	+1%	+2%	+3%	−6%
	3	+3%	−1%	−6%	+4%

Table S2.2 *Changes to the emission probabilities selected by the discriminative trainer.*

		To state			
		0	1	2	3
from state	0	·	·	·	·
	1	·	−1.5%	+1.5%	·
	2	·	+0.4%	−0.8%	+0.4%
	3	·	·	+0.9%	−0.9%

Table S2.3 *Changes to the transition probabilities selected by the discriminative trainer. A dot (.) means no change.*

Although M_{MLE} generated the dataset in this example, we could just as easily have trained (via MLE) a separate model M_{MLE}' from S_{synth} and compared the parsing accuracy of M_{MLE}' to that of M_{discr}. However, in the limit as the amount of training data approaches infinity, we expect the model parameters of M_{MLE}' to converge to the same values as the parameters of M_{MLE}. Thus, we see that the use of a different objective function for optimization of the model parameters (i.e., likelihood of the training data, versus nucleotide-level accuracy under Viterbi decoding) can impact the parsing accuracy of the resulting model, so that the choice of objective function can be as important as the choice of optimizer or even the choice of model type.

S2.6 Parameter Estimation as Sampling

As we remarked at the beginning of this chapter, parameter estimation can be formulated as a *sampling*—rather than *optimization*—problem. Perhaps the most popular family of sampling methods are those that fall into the category of *Markov chain Monte Carlo* (*MCMC*), which we will consider only briefly, since its use for training gene finders is not yet widespread (though that may change).

The primary motivation for estimating parameters via sampling is to find the *maximum a posteriori* (*MAP*) parameter settings in the presence of missing data. The "missing data" may consist, for example, of the unknown sequences and parses of ancestral taxa, in the case of comparative gene finding using related informant genomes—i.e., using a PhyloHMM. Since

the ancestral sequences account for the non-independence of the informants, knowledge of those ancestral sequences (and their parses) is important for accurate estimation of substitution rates and other parameters of the evolutionary model underlying the PhyloHMM. Sampling provides a way to infer this missing data, in a principled way, as we will explain shortly.

The ability of sampling to provide MAP estimates—rather than ML estimates—of parameters is another advantage of this technique. Recall that the objective function in MAP estimation is equal to the ML objective function weighted by a prior:

$$\arg\max_{\theta} P(\theta|X) = \arg\max_{\theta} \frac{P(\theta, X)}{P(X)} = \arg\max_{\theta} P(X|\theta)P(\theta) \qquad \text{(S2.38)}$$

for data set X. In the rightmost term you can see that we are maximizing the product of the *likelihood*—$P(X|\theta)$—together with the *prior* probability of each parameter set—$P(\theta)$. The use of a prior in parameter estimation can be a very powerful technique, since it gives the modeler great freedom in incorporating biological insight into the training process. Often, however, there is no obvious way to derive a prior probability distribution on θ, and in these cases we can simply use a *uniform* distribution—i.e., an "uninformative prior." The effect is that the $P(\theta)$ term drops out of the above equation and we are left with maximum likelihood. Thus, parameter estimation via sampling can be formulated as a MAP estimation or as an ML estimation, showing that the technique is very flexible indeed.

The basic idea behind sampling is very simple. If finding the θ which maximizes $P(\theta|X)$ is computationally intractible, but we are at least able to *sample* from this distribution, then we can use sampling as a means to approximate $argmax_{\theta} P(\theta|X)$. This can be done by merely sampling some number of parameter sets $\Theta = (\theta_0, \theta_1, \ldots, \theta_n)$, and then choosing the parameter set θ for which $P(\theta|X)$ is maximal over the sampled set Θ:

$$\theta^* = \arg\max_{\theta \in \Theta} P(\theta|X) \qquad \text{(S2.39)}$$

In cases where finding the maximimal $P(\theta|X)$ over all possible parameterizations θ is computationally intractible, accepting the maximum over all *sampled* parameterizations Θ should be a good approximation, so long as the sample size is sufficiently large.

As an alternative to taking the *maximum* over the sampled set, we can instead take an *average*:

$$\theta_i^* = \frac{1}{|\Theta|}\sum_{\theta\in\Theta}\theta_i \tag{S2.40}$$

where θ_i denotes the i^{th} parameter, p_i, in $\theta=(p_0, p_1, ...)$. Eq. (S2.40) can be shown to provide an estimate of the *expectation* of the corresponding parameter, as long as the sample points θ are indeed drawn according to the posterior distribution, $P(\theta|X)$:

$$\theta_i^* \approx \sum_{\theta} P(\theta)\theta_i \tag{S2.41}$$

Although this will in almost all cases produce a selected θ^* which is not maximally probable, taking an average can sometimes be advantageous when our procedure for sampling from $P(\theta|X)$ lacks accuracy; in such cases, the act of averaging over the sampled points can help to guard against noise.

MCMC sampling can be accomplished by applying the *Metropolis-Hastings* algorithm (Metropolis *et al.*, 1953; Hastings, 1970), as follows. Given an initial parameterization $\theta^{(0)}$, which may be produced either by random initialization or some other simple method, we iteratively apply the following algorithm, increasing k each time, starting at k=0:

1. $\theta' \sim P(\theta|\theta^{(k)}, X)$
2. $r_H = \min\left(1, \frac{P(\theta'|X)P(\theta^{(k)}|\theta', X)}{P(\theta^{(k)}|X)P(\theta'|\theta^{(k)}, X)}\right)$
3. $r \sim U(0,1)$
4. $\theta^{(k+1)} = \begin{cases}\theta' & \text{if } r \le r_H \\ \theta^{(k)} & \text{otherwise}\end{cases}$

In line 1, we sample θ' from the distribution $P(\theta|\theta^{(i)}, X)$, where $\theta^{(k)}$ is the most recent sample produced via this algorithm (or the random initial point, $\theta^{(0)}$). The function $P(\theta|\theta^{(k)}, X)$ is known as the *proposal distribution*, though it should be noted that these this not be a properly normalized probability density[4]; the only important requirement for this function is that it allow us to get from any sample point to any other sample point (possibly via intermediate points) with zero probability—a property called *ergodicity*. On line 2 we compute what is known as a *Hastings ratio*, which will serve as an

[4] If $P(\theta|\theta^{(k)}, X)$ is a properly normalized distribution, then r_H will always evaluate to 1, so that the move is accepted; this is equivalent to Gibbs sampling.

acceptance distribution. On line 3 we pick a real number $r \in [0,1]$ uniformly at random, and on line 4 we test this value against the threshold value r_H computed via the Hastings ratio. If $r \leq r_H$, then the new parameterization θ' sampled from the proposal distribution is accepted and emitted by the algorithm as the next sampled point, $\theta^{(k+1)}$. Otherwise, θ' is rejected, and we instead assign $\theta^{(k+1)} = \theta^{(k)}$.

The sampling of θ' in line 1 may be performed in a component-wise manner if desired—that is, we may randomly (or systematically) choose a component index i, and then sample a new value for just that component:

$$\forall_j \; \theta'_j = \begin{cases} p \sim P(\theta_i \left| \theta^{(k)}, X \right.) & \text{if } j = i \\ \theta_j^{(k)} & \text{otherwise} \end{cases} \tag{S2.42}$$

Similarly, we may sample a whole block of components all at once, if that happens to be convenient for the problem at hand. If $P(\theta_i | \theta^{(k)}, X)$ is a properly-normalized probability distribution, then an MCMC sampler utilizing Eq. (S2.42) is called a *Gibbs sampler*; Gibbs samplers often exhibit better convergence behavior than non-Gibbs MCMC samplers. If a block of components are sampled simultaneously from the properly-normalized posterior, we call the sampler a *blocked Gibbs sampler*. Note that θ need not be conditioned on $\theta^{(k)}$; in this case we have what is known as an *independence sampler* (Tierney, 1994).

In the case of missing data—i.e., where there are unobservable variables in our model—sampling can provide a very convenient solution, via *data augmentation*. By incorporating the unobservable variables into the set of parameters θ and then performing sampling as usual, we allow the sampler to assign likely values to the unobservables during the parameter estimation process. For any given sample point, since the unobservables will be assigned particular values, evaluation of the model likelihood and other relevant quantities will not be impeded by the (normally) unobservable nature of these *nuissance parameters*. At the end of the sampling process, the actual model parameters may be extracted in the usual way from θ, while the sampled values for the unobservables are discarded.

An MCMC sampler generates a stream of data points which, in the limit as $k \to \infty$, converge to the *stationary distribution*, $P(\theta | X)$. The initial stream of samples, prior to convergence, is called the *burn-in*. It is important to ignore the samples obtained prior to burn-in, since they will not, in general, appear with frequency proportional to their true stationary probability. Rigorous methods for establishing the exact length of the burn-in period have been proposed in the literature, but none have yet been widely accepted. In practice, researchers typically plot successive values of a parameter and subjectively identify some initial portion of the resulting curve which appears

to differ qualitatively from the rest of the plot.

Another difficult decision regards the number of sample points to collect after the burn-in period. One method is to run several copies of the sampling algorithm simultaneously (from different initial points) and to stop the ensemble of samplers only when the averages from the different runs are sufficiently similar (see, e.g., Raferty and Lewis, 1995).

In addition to the notion of *convergence* (i.e., that length of time needed for the sampler to begin sampling from the true posterior distribution), there is the equally important notion of *mixing*. A sampler with *slow mixing* tends to produce successive sample points which are highly correlated to each other. Even after such a sampler has converged to the posterior distribution, it may be necessary to collect a very large number of samples before an unbiased average can be computed, since the exploration of the sample space conducted by the sampler may be very slow. General advice regarding convergence and mixing can be found in (Gilks *et al.*, 1995).

Training of gene-finders via sampling is not a widespread practice, though as gene-finding systems become more complex and incorporate larger numbers of parameters, the use of sampling may very well become more popular. The method is already seeing some use in the training of the evolution models underlying PhyloHMMs (e.g., Lunter and Hein, 2004)

S2.7 Summary

Many techniques have been devised over the years for estimating the parameters of quantitative models; we have considered only a few of these, concentrating on those which are either most generally applicable or have proved most popular for training predictive models in the machine-learning and/or bioinformatics fields. As we have indicated several times in this book, we believe that issues related to training may often have a more significant impact on predictive accuracy than other modeling issues, and so we believe that further improvements in the predictive accuracy of gene-finding models will in many cases derive primarily from further advances in approaches to their training.

In this chapter we have concentrated on the use of numerical optimizers, with an emphasis on gradient-based approaches. Figure S2.1 presents a simple taxonomy of parameter estimation methods and places within it some of the algorithms we have considered here.

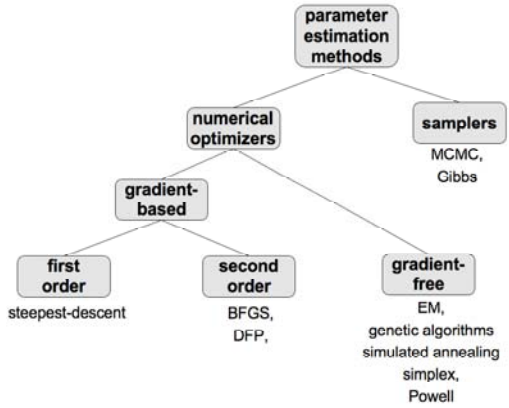

Figure S2.16 *A taxonomy of parameter estimation methods.*

Additional References

See the print edition for the full list of references.

Blimes JA (1998) A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. *Technical Report TR-97-021*, Department of Electrical Engineering and Computer Science, U.C. Berkeley.

Borman S (2006) The expectation maximization algorithm: A short tutorial. *Unpub. manuscript*. URL: http://www.seanborman.com/publications/EM_algorithm.pdf.

Bochkanov S, Bystritsky V (2007). ALGLIB, http://www.alglib.net/.

Boyd SP, Vandenberghe L (2004) *Convex Optimization*. Cambridge University Press.

Byrd RH, Lu P, Nocedal J (1995) A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Stat. Comp.* 16:1190-1208.

Carreira-Perpiñán MA (2007) Gaussian mean-shift is an EM algorithm. *IEEE Trans Patt Anal Mach Intel* 29:767-776.

Dempster AP, Laird NM, Rubin DB (1977) Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Stat. Soc. B*:1-38.

Gilks WR, Richardson S, Spiegelhalter DJ (1995) *Markov Chain Monte Carlo in Practice*. Chapman &Hall.

Gill J, King G (2004) Numerical issues involved in inverting Hessian matrices. In: Altman M, Gill J, McDonald MP (eds.) *Numerical Issues in Statistical Computing for the Social Scientist.* Wiley.

Hastings, WK (1070) Monte carlo sampling methods using Markov chains and their applications, *Biometrika* 57:97-109.

Lunter G, Hein J (2004) A nucleotide substitution model with nearest-

neighbor interactions. *Bioinformatics* 20:i216-i23.
Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953) Equations of state calculations by fast computing machines. *J Chem Phys* 21:1087-1092.
Nelder JA, Mead R (1965) A simplex method for function minimization. *Comput J* 7:308-313.
Raferty AE, Lewis SM (1995) Implementing MCMC. In: Gilks WR, Richardson S, Spiegelhalter DJ (eds.), *Markov Chain Monte Carlo in Practice*. Chapman &Hall.
Redner RA, Walker HF (1984) Mixture densities, maximum likelihood and the EM algorithm. *SIAM Review* 26:195-239.
Rodríguez LJ, Torres I (2003) Comparative study of the Baum-Welch and Viterbi training algorithms applied to read and spontaneous speech recognition. In: Goos G, Hartmanis J, van Leeuwen J (Eds.) *Pattern Recognition and Image Analysis*, pp. 847-857. Springer.
Salakhutdinov R, Roweis S, Ghahramani Z (2003) Optimization with EM and Expectation-Conjugate-Gradient. *Proc. of the Twentieth International Conf. on Machine Learning (ICML)*.
Tierney L (1994) Markov chains for exploring posterior distributions (with discussion). *Ann Stat* 22:1701-1762.
Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. *IEEE Trans Evol Comp* 1:67-82.
Xu L, Jordan MI (1995) On convergence properties of the EM algorithm for Gaussian mixtures. *A.I. Memo #1520*, Massachusetts Institute of Technology.