CHAPTER 18

Likelihood and Bayesian inference and computation

Most of this book concerns the interpretation of regression models, with the understanding that they can be fit to data fairly automatically using R and Bugs. However, it can be useful to understand some of the theory behind the model fitting, partly to connect to the usual presentation of these models in statistics and econometrics.

This chapter outlines some of the basic ideas of likelihood and Bayesian inference and computation, focusing on their application to multilevel regression. One point of this material is to connect multilevel modeling to classical regression; another is to give enough insight into the computation to allow you to understand some of the practical computational tips presented in the next chapter.

18.1 Least squares and maximum likelihood estimation

We first present the algebra for classical regression inference, which is then generalized when moving to multilevel modeling. We present the formulas here without derivation; see the references listed at the end of the chapter for more.

Least squares

The classical linear regression model is $y_i = X_i\beta + \epsilon_i$, where y and ϵ are (column) vectors of length n, X is a $n \times k$ matrix, and β is a vector of length k. The vector β of coefficients is estimated so as to minimize the errors ϵ_i. If the number of data points n exceeds the number of predictors[1] k, it is not generally possible to find a β that gives a perfect fit (that would be $y_i = X_i\beta$, with no error, for all data points $i = 1, \ldots, n$), and the usual estimation goal is to choose the estimate $\hat{\beta}$ that minimizes the sum of the squares of the residuals $r_i = y_i - X_i\hat{\beta}$. (We distinguish between the *residuals* $r_i = y_i - X_i\hat{\beta}$ and the *errors* $\epsilon_i = y_i - X_i\beta$.) The sum of squared residuals is $SS = \sum_{i=1}^n (y_i - X_i\hat{\beta})^2$; the $\hat{\beta}$ that minimizes it is called the *least squares estimate* and can be written in matrix notation as

$$\hat{\beta} = (X^t X)^{-1} X^t y. \tag{18.1}$$

We rarely work with this expression directly, since it can be computed directly in the computer (for example, using the `lm()` command in R).

The errors ϵ come from a distribution with mean 0 and variance σ^2. This standard deviation can be estimated from the residuals, as

$$\hat{\sigma}^2 = \frac{1}{n-k} SS = \frac{1}{n-k} \sum_{i=1}^n (y_i - X_i\hat{\beta})^2, \tag{18.2}$$

with $n-k$ rather than $n-1$ in the denominator to adjust for the estimation of the

[1] The constant term, if present in the model, counts as one of the predictors; see Section 3.4.

k-dimensional parameter β. (Since β is estimated to minimize the sum of squared residuals, SS will be, on average, lower by a factor of $\frac{n-k}{n}$ than the sum of squared errors.)

Maximum likelihood

As just described, least squares estimation assumes linearity of the model and independence of the errors. If we further assume that the errors are normally distributed, so that $y_i \sim \mathrm{N}(X_i\beta, \sigma^2)$ for each i, the least squares estimate $\hat{\beta}$ is also the maximum likelihood estimate. The *likelihood* of a regression model is defined as the probability of the data given the parameters and inputs; thus, in this example,

$$p(y|\beta, \sigma, X) = \prod_{i=1}^{n} \mathrm{N}(y_i|X_i\beta, \sigma^2), \tag{18.3}$$

where $\mathrm{N}(\cdot|\cdot,\cdot)$ represents the normal probability density function, $\mathrm{N}(y|m, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{y-m}{\sigma}\right)^2\right)$. The model can also be written in vector-matrix notation as $y \sim \mathrm{N}(X\beta, \sigma^2 I_n)$, where I_n is the n-dimensional identity matrix. Giving a diagonal covariance matrix to this multivariate normal distribution implies independence of the errors.

Expression (18.3) is a special case of the general expression for the likelihood of n independent measurements given a vector parameter θ and predictors X:

$$p(y|\theta, X) = \prod_{i=1}^{n} p(y_i|\theta, X_i). \tag{18.4}$$

The maximum likelihood estimate is the vector θ for which this expression is maximized, given data X, y. (In classical least squares regression, θ corresponds to the vector of coefficients β, along with the error scale, σ.) In general, we shall use the notation $p(y|\theta)$ for the likelihood as a function of parameter vector θ, with the dependence on the predictors X implicit.

The likelihood can then be written as

$$p(y|\beta, \sigma, X) = \mathrm{N}(y|X\beta, \sigma^2 I_n). \tag{18.5}$$

Using the standard notation for the multivariate normal distribution with mean vector m and covariance matrix Σ, this becomes

$$\mathrm{N}(y|m, \Sigma) = (2\pi)^{-n/2}|\Sigma|^{-1/2} \exp\left(-\frac{1}{2}(y-m)^t\Sigma^{-1}(y-m)\right).$$

Expressions (18.3) and (18.5) are equivalent and are useful at different times when considering generalizations of the model.

A careful study of (18.3) or (18.5) reveals that maximizing the likelihood is equivalent to minimizing the sum of squared residuals; hence the least squares estimate $\hat{\beta}$ can be viewed as a maximum likelihood estimate under the normal model.

There is a small twist in fitting regression models, in that the maximum likelihood estimate of σ is $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - X_i\hat{\beta})^2}$, with $\frac{1}{n}$ instead of $\frac{1}{n-k}$. The estimate with $\frac{1}{n-k}$ is generally preferred: the maximum likelihood estimate of (β, σ) simply takes the closest fit and needs to be adjusted to account for the fitting of k regression coefficients.

Weighted least squares

The least squares estimate counts all n data points equally in minimizing the sum of squares. If some data are considered more important than others, this can be captured in the estimation by minimizing a *weighted* sum of squares, $WSS = \sum_{i=1}^{n} w_i(y_i - X_i\hat{\beta})^2$, so that points i with larger weights w_i count more in the optimization. The weighted least squares estimate is

$$\hat{\beta}^{\text{WLS}} = (X^t W X)^{-1} X^t W y, \tag{18.6}$$

where W is the diagonal matrix whose elements are the weights w_i.

Weighted least squares is equivalent to maximum likelihood estimation of β in the normal regression model

$$y_i \sim \text{N}(X_i\beta, \sigma^2/w_i), \tag{18.7}$$

with independent errors with variances inversely proportional to the weights. Points with high weights have low error variances and are thus expected to lie closer to the fitted regression function.

Weighted least squares can be further generalized to fit data with correlated errors; if the data are fit by the model $y \sim \text{N}(X\beta, \Sigma)$, then the maximum likelihood estimate is $\hat{\beta} = (X^t\Sigma^{-1}X)^{-1}X^t\Sigma^{-1}y$ and minimizes the expression $(y - X\beta)^t\Sigma^{-1}(y - X\beta)$, which can be seen as a generalization of the "sum of squares" concept.

Generalized linear models

Classical linear regression can be motivated in a purely algorithmic fashion (as "least squares") or as maximum likelihood inference under a normal model. With generalized linear models, the algorithmic justification is usually set aside, and maximum likelihood is the starting point. We illustrate with the two most important examples.

Logistic regression. For binary logistic regression with data $y_i = 0$ or 1, the likelihood is

$$p(y|\beta, X) = \prod_{i=1}^{n} \begin{cases} \text{logit}^{-1}(X_i\beta) & \text{if } y_i = 1 \\ 1 - \text{logit}^{-1}(X_i\beta) & \text{if } y_i = 0, \end{cases}$$

which can be written more compactly, but equivalently, as

$$p(y|\beta, X) = \prod_{i=1}^{n} \left(\text{logit}^{-1}(X_i\beta)\right)^{y_i} \left(1 - \text{logit}^{-1}(X_i\beta)\right)^{1-y_i}.$$

To find the β that maximizes this expression, we can compute the derivative $dp(y|\beta, X)/d\beta$ of the likelihood (or, more conveniently, the derivative of the logarithm of the likelihood), set this derivative equal to 0, and solve for β. There is no closed-form solution, but the maximum likelihood estimate can be found using an iteratively weighted least squares algorithm, each step having the form of a weighted least squares computation, with the weights changing at each step.

Not just a computational trick, iteratively weighted least squares can be understood statistically as a series of steps approximating the logistic regression likelihood by a normal regression model applied to transformed data. We shall not discuss this further here, however. We only mentioned the algorithm to give a sense of how likelihood functions are used in classical estimation.

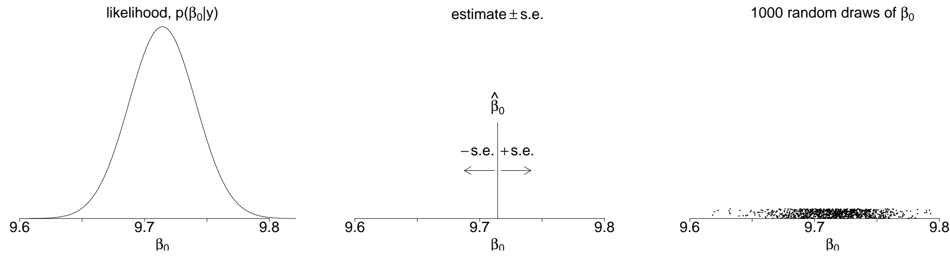

Figure 18.1 *(a) Likelihood function for the parameter β_0 in the trivial linear regression $y = \beta_0 +$ error, of log earnings y_i. (b) Mode of the likelihood function and range indicating ± 1 standard error as computed from the inverse-second-derivative-matrix of the log likelihood at the mode. (c) 1000 random simulation draws from the normal distribution with this mean and standard deviation, representing the distribution of uncertainty in the inference for β_0. The simulations have been vertically jittered to make them visible. (For this one-dimensional problem it would be better to display the simulations as a histogram; we use a dotplot here for compatability with the scatterplot of the two-dimensional simulations in Figures 18.2–18.3.)*

Poisson regression. For Poisson regression (6.3), the likelihood is

$$p(y|\beta, X, u) = \prod_{i=1}^{n} \text{Poisson}\left(y_i \middle| u_i e^{X_i \beta}\right),$$

where each factor has the Poisson probability density function: $\text{Poisson}(y|m) = \frac{1}{y!} m^y e^{-m}$.

18.2 Uncertainty estimates using the likelihood surface

In maximum likelihood estimation, the likelihood function can be viewed as a "hill" with $\hat{\beta}$ identifying the location of the top of the hill—that is, the mode of the likelihood function. We illustrate with two simple regression examples.

One-parameter example: linear regression with just a constant term

Figure 18.1 demonstrates likelihood estimation for the simple problem of regression with only a constant term; that is, inference for β_0 in the model $y_i = \beta_0 + \epsilon_i,\ i = 1, \ldots, n$, for the earnings data from Chapter 2. In this example, β_0 corresponds to the average log earnings in the population represented by the survey. For simplicity, we shall assume that σ, the standard deviation of the errors ϵ_i, is known and equal to the sample standard deviation of the data.

Figure 18.1a plots the likelihood function for β_0. The peak of the function is the maximum likelihood estimate, which in this case is simply $\bar{y}$, the average log earnings reported in the sample. The range of the likelihood function tells us that it would be extremely unlikely for these data to occur if the true β_0 were as low as 9.6 or as high as 9.8. Figure 18.1b shows the maximum likelihood estimate ± 1 standard error, and Figure 18.1c displays 1000 random draws from the normal distribution representing uncertainty in β_0.

Two-parameter example: linear regression with two coefficients

Figure 18.2 illustrates the slightly more complicated case of a linear regression model with two coefficients (corresponding to a constant term and a linear predic-

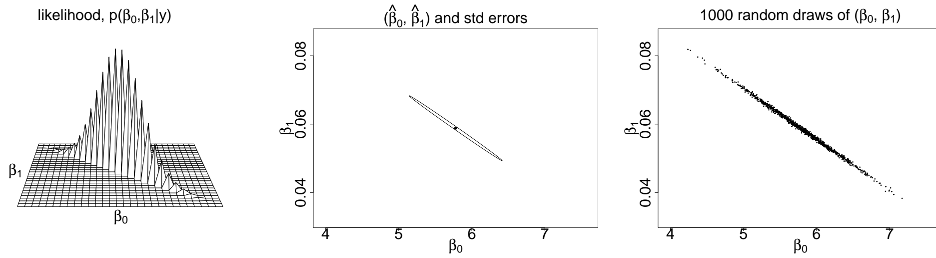

Figure 18.2 *(a) Likelihood function for the parameters β_0, β_1 in the linear regression $y = \beta_0 + \beta_1 x +$ error, of log earnings, y_i, on heights, x_i. (The spiky pattern of the three-dimensional plot is an artifact of the extreme correlation of the distribution.) (b) Mode of the likelihood function (that is, the maximum likelihood estimate $(\hat\beta_0, \hat\beta_1)$) and ellipse summarizing the inverse-second-derivative-matrix of the log likelihood at the mode. (c) 1000 random simulation draws from the normal distribution centered at $(\hat\beta_0, \hat\beta_1)$ with variance matrix equal to the inverse of the negative second derivative of the log likelihood.*

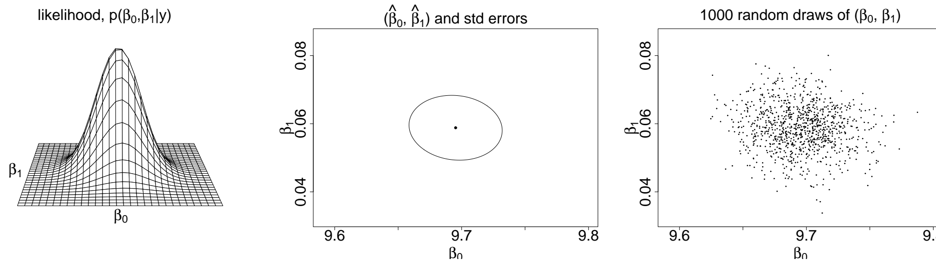

Figure 18.3 *(a) Likelihood function, (b) mode and uncertainty ellipse, and (c) 1000 simulation draws of the regression coefficients for the model $y = \beta_0 + \beta_1 z +$ error, of log earnings, y_i, on mean-centered heights, $z_i = x_i - \bar{x}$. The inferences for the parameters β_0, β_1 are now independent. Compare to Figure 18.2.*

tor). (Strictly speaking this model has three parameters—β_0, β_1, and σ—but for simplicity we display the likelihood of β_0, β_1 conditional on the estimated $\hat\sigma$.)

Figure 18.2a shows the likelihood as a function of (β_0, β_1). The area with highest likelihood surrounding the peak can be represented by an ellipse as is shown in Figure 18.2b. Figure 18.2c displays 1000 random draws from the normal distribution with covariance matrix represented by this ellipse. The shape of the uncertainty ellipse, or equivalently the correlation of the simulation draws, tells us something about the information available about the two parameters. For example, the data are consistent with β_0 being anywhere between 4.5 and 7.2, and with β_1 being anywhere between 0.04 and 0.08. However, the inferences for these two parameters are correlated: if β_0 is 4.5, then β_1 must be near 0.08, and if β_0 is 7, then β_1 must be near 0.04. To understand this inferential correlation, see Figure 4.1 on page 54: the regression line must go through the cloud of points, which is far from the y-axis. Lines of higher slope (for which β_1 is higher) intersect the y-axis at a lower value (and thus have lower values of β_0), and vice versa.

It is convenient to reparameterize the model so that the inferences for the intercept and slope coefficients are uncorrelated. We can do this by replacing the predictor x_i by its mean-centered values, $z_i = x_i - \overline{x}$—that is, height relative to the average height in the sample. Figure 18.3 shows the likelihood function and simulations for the coefficients in the regression of $y = \beta_0 + \beta_1 z +$ error.

Nonidentified parameters and the likelihood function

In maximum likelihood inference, parameters in a model are nonidentified if they can be changed without affecting the likelihood. Continuing with the "hill" analogy, nonidentifiability corresponds to a "ridge" in the likelihood—that is, a direction in parameter space in which the likelihood is flat. This occurs, for example, when predictors in a classical regression are collinear.

Summarizing uncertainty about β and σ using the variance matrix from a fitted regression

We summarize the fit of a model $y = X\beta + \epsilon$ by a least squares estimate $\hat{\beta} = (X^tX)^{-1}X^ty$ and a *variance matrix* (or *covariance matrix*) of estimation,

$$V_\beta = (X^tX)^{-1}\sigma^2. \tag{18.8}$$

We represent the uncertainty in the estimated β vector using the normal distribution with mean $\hat{\beta}$ and variance matrix V_β. Figures 18.1–18.3 show examples of the estimated $\mathrm{N}(\hat{\beta}, V_\beta)$ distribution in one and two dimensions.

Expression (18.8) depends on the unknown σ^2, which we can estimate most simply with $\hat{\sigma}^2$ from (18.2) on page 387. To better capture uncertainty, we first compute $\hat{\sigma}^2$ and then sample $\sigma^2 = \hat{\sigma}^2(n-k)/X^2_{n-k}$, where X^2_{n-k} represents a random draw from the χ^2 distribution with $n-k$ degrees of freedom. These steps are performed by the `sim()` function we have written in R, as we describe next.

18.3 Bayesian inference for classical and multilevel regression

Bayesian inference for classical regression

In Bayesian inference, the likelihood is multiplied by a prior distribution, and inferences are typically summarized by random draws from this product, the *posterior distribution.*

The simplest form of Bayesian inference uses a uniform prior distribution, so that the posterior distribution is the same as the likelihood function (when considered as a function of the parameters), as pictured, for example, in the left graphs in Figures 18.1–18.3. The random draws shown in the rightmost graphs in these figures correspond to random draws from the posterior distribution, assuming a uniform prior distribution. In this way, informal Bayesian inference is represented as discussed in Section 7.2, using the simulations obtained from the `sim()` function in R (which draws from the normal distribution with mean $\hat{\beta}$ and standard deviation V_β). This is basically a convenient way to summarize classical regression, especially for propagating uncertainty for predictions.

Informative prior distributions in a single-level regression

Bayesian inference can also be used to add numerical information to a regression model. Usually we shall do this using a multilevel model, but we illustrate here with the simpler case of a specified prior distribution—the regression of log earnings on height, shown in Figures 18.2 and 18.3. Suppose we believed that β_1 was probably between 0 and 0.05—that is, a predicted difference of between 0 and 5% in earnings per inch of height. We could code this as a normal prior distribution with mean 2.5% and standard deviation 2.5%, that is, $\beta_1 \sim \mathrm{N}(0.025, 0.025^2)$.

Mathematically, this prior distribution can be incorporated into the regression

by treating it as an additional "data point" of 0.025, measured directly on β_2, with a standard deviation of 0.025. This in turn can be computed using a weighted regression of an augmented data vector y_* on an augmented predictor matrix X_* with augmented weight vector w_*. These are defined as follows:

$$y_* = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ 0.025 \end{pmatrix}, \quad X_* = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \\ 0 & 1 \end{pmatrix}, \quad w_* = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ \sigma_y^2/0.025^2 \end{pmatrix}. \qquad (18.9)$$

We have added the prior information as a new data point and given it a weight of the data variance (which can be estimated from the classical regression) divided by the prior variance. This weighting makes sense:

- If $\sigma_y > 0.025$, then the prior distribution is *more* informative than any data point, and so the prior "data point" will be given a high weight.
- If $\sigma_y = 0.025$, then the prior distribution has the same information as one data point and so is given equal weight.
- If $\sigma_y < 0.025$, then the prior distribution has *less* information than a single data point and so gets a lower weight.

We rarely use formulation (18.9) directly, but similar ideas apply with multilevel models, in which the group-level model for parameters α_j and β_j can be interpreted as prior information.

Collinearity and Bayesian regression

In the matrix-algebra language of (18.1) and (18.6), the Bayesian estimate—the least squares estimate of the augmented regression based on (18.9)—contains the expression $X_*^t \mathrm{Diag}(w_*) X_*$, which is simply $X^t X$ with an added term corresponding to the prior information. With collinear predictors, the original $X^t X$ is noninvertible, but the new $X_*^t \mathrm{Diag}(w_*) X_*$ might be invertible, depending on the structure of the new information.

For example, in a classical varying-intercept model, we would include only $J-1$ group indicators as predictors, because a regression that included the constant term along with indicators for all J groups would be collinear and nonidentifiable. But the multilevel model has the effect of adding a prior distribution for the J coefficients of the group indicators, thus adding a term to $X^t X$ which makes the new matrix invertible, even with the constant term included as well.

Simple multilevel model with no predictors

Bayesian inference achieves partial pooling for multilevel models by treating the group-level model as defining prior distributions for varying intercepts and slopes.

We begin by working through the algebra for the radon model with no individual- or group-level predictors, simply measurements within counties:

$$\begin{aligned} y_i &\sim \mathrm{N}(\alpha_{j[i]}, \sigma_y^2) \ \text{ for } i = 1, \ldots, n \\ \alpha_j &\sim \mathrm{N}(\mu_\alpha, \sigma_\alpha^2) \ \text{ for } j = 1, \ldots, J. \end{aligned} \qquad (18.10)$$

We label the number of houses in county j as n_j (so that Lac Qui Parle County has $n_j = 2$, Aitkin County has $n_j = 4$, and so forth; see Figure 12.2 on page 255).

Sample size in group, n_j	Estimate, $\hat{\alpha}_j$
$n_j = 0$	$\hat{\alpha}_j = \mu_\alpha$ (complete pooling)
$n_j < \sigma_y^2/\sigma_\alpha^2$	$\hat{\alpha}_j$ closer to μ_α
$n_j = \sigma_y^2/\sigma_\alpha^2$	$\hat{\alpha}_j = \frac{1}{2}\bar{y}_j + \frac{1}{2}\mu_\alpha$
$n_j > \sigma_y^2/\sigma_\alpha^2$	$\hat{\alpha}_j$ closer to $\bar{y}_j$
$n_j = \infty$	$\hat{\alpha}_j = \bar{y}_j$ (no pooling)

Figure 18.4 *Summary of partial pooling of multilevel estimates as a function of group size.*

Complete-pooling and no-pooling estimates. As usual, we begin with the classical estimates. In complete pooling, all counties are considered to be equivalent, so that $\alpha_1 = \alpha_2 = \cdots = \alpha_J = \mu_\alpha$, and the model reduces to $y_i \sim \mathrm{N}(\mu_\alpha, \sigma_y^2)$ for all measurements y. The estimate of μ_α, and thus of all the individual α_j's, is then simply $\bar{y}$, the average of the n measurements in the data.

In the no-pooling model, each county is estimated alone, so that each α_j is estimated by $\bar{y}_j$, the average of the measurements in county j.

Multilevel inference if the hyperparameters were known. The multilevel model (18.10) has *data-level regression coefficients* $\alpha_1, \ldots, \alpha_J$ and *hyperparameters* μ_α, σ_y, and σ_α. In multilevel estimation, we perform inference for both sets of parameters. To explain how to do this, we first work out the inferences for each set of parameters separately.

The key step of multilevel inference is estimation of the data-level regression coefficients given the data and hyperparameters—that is, acting as if the hyperparameters were known. As discussed in the regression context in Section 12.2, the estimate of each α_j will be a compromise between $\bar{y}_j$ and μ_α, the unpooled estimate in county j and the average over all the counties.

Given the hyperparameters, the inferences for the α_j's follow independent normal distributions, which we can write as

$$\alpha_j | y, \mu_\alpha, \sigma_y, \sigma_\alpha \sim \mathrm{N}(\hat{\alpha}_j, V_j), \quad \text{for } j = 1, \ldots, J, \tag{18.11}$$

where the estimate and variance of estimation are

$$\hat{\alpha}_j = \frac{\frac{n_j}{\sigma_y^2}\bar{y}_j + \frac{1}{\sigma_\alpha^2}\mu_\alpha}{\frac{n_j}{\sigma_y^2} + \frac{1}{\sigma_\alpha^2}}, \quad V_j = \frac{1}{\frac{n_j}{\sigma_y^2} + \frac{1}{\sigma_\alpha^2}}. \tag{18.12}$$

The notation "$\alpha_j | y, \mu_\alpha, \sigma_y, \sigma_\alpha \sim$" in (18.11) can be read as, "α_j, given data, μ_α, σ_y, and σ_α, has the distribution ...," indicating that this is the estimate with the hyperparameters assumed known.

The estimate $\hat{\alpha}_j$ in (18.12) can be interpreted as a *weighted average* of $\bar{y}_j$ and μ_α, with relative weights depending on the sample size in the county and the variance at the data and group levels. As shown in Figure 18.4, the key parameter is the variance ratio, $\sigma_y^2/\sigma_\alpha^2$. For counties j for which $n_j = \sigma_y^2/\sigma_\alpha^2$, then the weights in (18.12) are equal, and $\hat{\alpha}_j = \frac{1}{2}\bar{y}_j + \frac{1}{2}\mu_\alpha$. If n_j is greater than the variance ratio, then $\hat{\alpha}_j$ is closer to $\bar{y}_j$; and if n_j is less then the variance ratio, then $\hat{\alpha}_j$ is closer to μ_α.

Crude inference for the hyperparameters. Given the data-level regression coefficients α_j, how can we estimate the hyperparameters, $\sigma_y, \mu_\alpha, \sigma_\alpha$ in the multilevel model (18.10)?

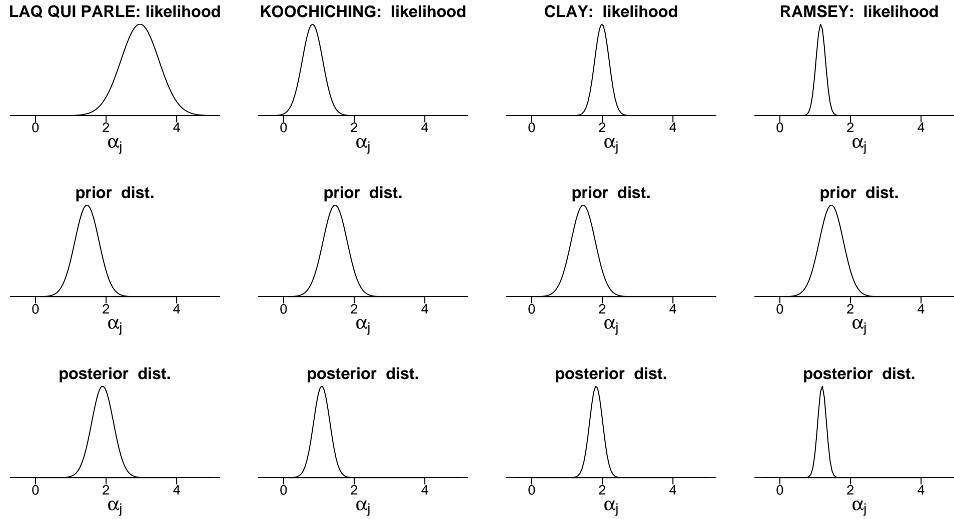

Figure 18.5 *Likelihood, prior distribution, and posterior distribution for the intercept parameter α_j for the simple radon model (with no county-level predictors) in four different counties j in Minnesota with a range of sample sizes in the data. As the sample size in the county increases, the likelihood becomes more informative (see Figure 12.4 on page 257).*

- The natural estimate of the data variance σ_y^2 is simply the residual variance:

$$\hat{\sigma}_y^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \alpha_{j[i]})^2. \tag{18.13}$$

- The mean μ_α from the group-level model in (18.10) can be estimated by the average of the county intercepts α_j:

$$\hat{\mu}_\alpha = \frac{1}{J}\sum_{j=1}^{J}\alpha_j, \tag{18.14}$$

  with an estimation variance of $\frac{1}{J}\sigma_\alpha^2$.
- The group-level variance σ_α^2 can be estimated by

$$\hat{\sigma}_\alpha^2 = \frac{1}{J}\sum_{j=1}^{J}(\alpha_j - \mu_\alpha)^2. \tag{18.15}$$

Unfortunately, the county parameters α_j are not themselves known, so we cannot directly apply the above formulas. We can, however, use an iterative algorithm that alternately estimates the α_j's and the hyperparameters, as we describe next.

Individual predictors but no group-level predictors

We next consider the varying-intercept model (12.2) from page 256: $y_i \sim \mathrm{N}(\alpha_{j[i]} + \beta x_i, \sigma_y^2)$, where $j[i]$ is the county containing house i. The basic varying-intercept model (12.3) is $\alpha_j \sim \mathrm{N}(\mu_\alpha, \sigma_\alpha^2)$—that is, a normal prior distribution for each α_j that is common to all counties j. (The hyperparameters $\mu_\alpha, \sigma_\alpha$ must themselves be estimated from the data, but we shall set this issue aside for a moment and just treat them as known.)

The top row of Figure 18.5 shows the likelihood for α_j in four of the counties.

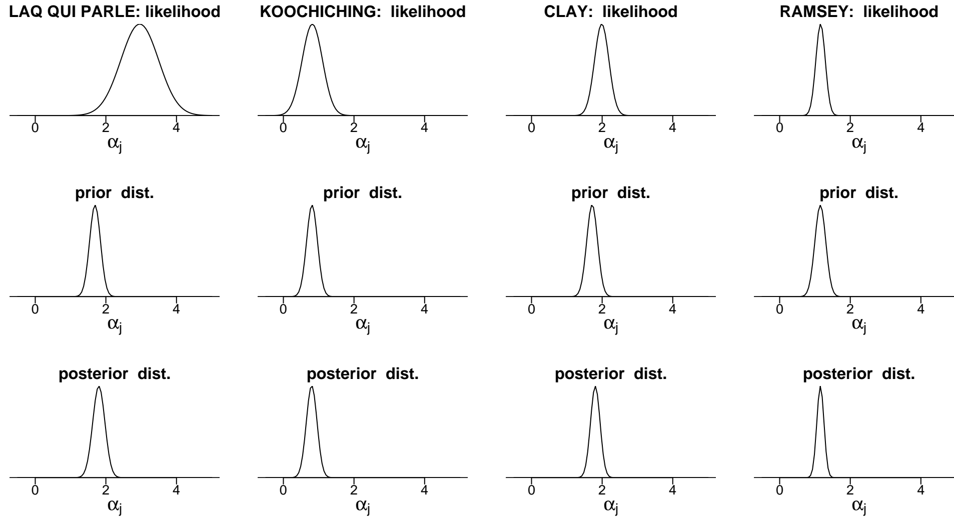

Figure 18.6 *Likelihood, prior distribution, and posterior distribution for the intercept parameter α_j in four counties for the radon model that includes uranium as a county-level predictor. The prior distributions for the counties now differ because of their varying uranium levels (see Figure 12.6 on page 266). Compare to Figure 18.5.*

For each county j, the likelihood indicates the range of values of α_j that are most consistent with the data in that county. The four counties are displayed in increasing order of sample size; the likelihood is more informative as sample size increases. The second row of Figure 18.5 shows the $\mathrm{N}(\mu_\alpha, \sigma_\alpha^2)$ prior distribution, which is the same for the four counties. In this context the "prior distributions" do not represent information occurring before the data have been seen; rather, they convey the information about the distribution of the α_j's among the counties, which is relevant for estimating each individual α_j.

The bottom row of Figure 18.5 displays the posterior distributions, which combine the information from the likelihoods and prior distributions. The posterior distribution for each county is centered at a point between the maximum likelihood estimate and the maximum of the prior distribution—a weighted average of likelihood and prior estimates—falling closer to the prior distribution when sample sizes are small and closer to the likelihood when sample sizes are large.

Including group-level predictors

We now move to the radon model including uranium as a county-level predictor. Figure 18.6 displays the likelihood, prior distributions, and posterior distributions for four counties. In this case, the prior distributions for county j is normal with mean $\gamma_0 + \gamma_1 u_j$ and variance σ_α^2. The county uranium levels u_j vary, and so the prior distributions vary also, as can be seen in the second row of Figure 18.6.

Multilevel regression as least squares with augmented data

To understand the matrix algebra of multilevel regression, we continue with the data-augmentation idea illustrated in (18.9) on page 393. Starting with classical weighted least squares with data vector $y = (y_1, \ldots, y_n)$, an $n \times k$ predictor matrix X, and data weights $w = (w_1, \ldots, w_n)$, we define $W_y = \mathrm{Diag}(w_1, \ldots, w_n)$, which is

a matrix proportional to the inverse data variances in the model. The model is

$$y \sim \mathrm{N}(X\beta, \Sigma_y),$$

where $\Sigma_y = \sigma^2 W_y^{-1}$. The vector of regression coefficients is estimated by weighted least squares as $\hat{\beta}_{\mathrm{wls}} = (X^t W_y X)^{-1} X^t W_y y$, and the corresponding variance matrix is $V_\beta = (X^t W_y X)^{-1}\sigma^2$. We refer to this as "the regression of y on X with weight matrix W_y." (This reduces to classical unweighed regression if the weights are all equal to 1, so that W_y is the identity matrix.)

To fit multilevel models in this framework, we work with the formulation as a large regression model, as in the discussion following (12.10) on page 264. We illustrate with the flight simulator model (13.9) on page 289. Here, β is a vector of length 14: the mean parameter, followed by 5 treatment effects and 8 airport effects. In the notation of (13.9), $\beta = (\alpha_0, \gamma_1, \ldots, \gamma_5, \delta_1, \ldots, \delta_8)$. We define μ_β and Σ_β as the mean and variance of β in the prior distribution: $\beta \sim \mathrm{N}(\mu_\beta, \Sigma_\beta)$. Finally, we define the weight matrix W_β as the inverse-variance of β, scaled by the data variance: $W_\beta = \sigma_y^2 \Sigma_\beta^{-1}$.

For the flight simulator example, μ_β is a vector of 14 zeroes, and W_β is a diagonal matrix with diagonal entries , followed by $\sigma_y^2/\sigma_\gamma^2$ five times, followed by $\sigma_y^2/\sigma_\delta^2$ eight times. The first element of the diagonal of W_β is zero because our model specifies no information about the parameter α_0; the other elements indicate the information in the model about each multilevel parameter, compared to the information in each data point.

The multilevel model can be expressed as a least squares regression of y_* on X_* with weight matrix W_*, where

$$y_* = \begin{pmatrix} y \\ \mu_\beta \end{pmatrix}, \quad X_* = \begin{pmatrix} X \\ I_k \end{pmatrix}, \quad W_* = \begin{pmatrix} W_y & 0 \\ 0 & W_\beta \end{pmatrix}. \tag{18.16}$$

The augmented data correspond to the extra information in the model for β. The augmentation has the effect of partially pooling the least squares estimate of β in the direction of its mean vector μ_β, and can be viewed as a matrix generalization of (18.12).

18.4 Gibbs sampler for multilevel linear models

Gibbs sampling is the name given to a family of iterative algorithms that are used by Bugs ("Bayesian inference using Gibbs sampling") and other programs to fit Bayesian models. The basic idea of Gibbs sampling is to partition the set of unknown parameters and then estimate them one at a time, or one group at a time, with each parameter or group of parameters estimated conditional on all the others. The algorithm is effective because, in a wide range of problems, estimating separate parts of a model is relatively easy, even if it is difficult to see how to estimate all the parameters at once.

Figure 18.7 illustrates the Gibbs sampler for a simple example. In general, the algorithm proceeds as follows:

1. Choose some number n_{chains} of parallel simulation runs (typically a small number such as 3). For each of the chains:

   (a) Start with *initial values* for all the parameters. These should be dispersed (as pictured by the solid squares in Figure 18.7); for convenience we typically use simple random numbers, as discussed in Chapter 16 in the context of implementing models in Bugs.

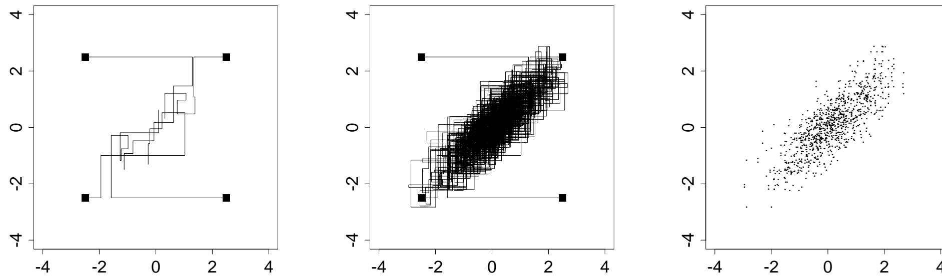

Figure 18.7 *Four independent sequences of the Gibbs sampler for a simple example with two parameters. Initial values of the $n_{\text{chains}} = 4$ sequences are indicated by solid squares. (a) First 10 iterations, showing the component-by-component updating of the Gibbs iterations. (b) After 500 iterations, when the sequences have reached approximate convergence. (c) The points from the second halves of the sequences.*

(b) Choose some number n_{iter} of iterations (typically a somewhat large number such as 1000). For each iteration:

Update the parameters, or batches of parameters, one at a time. For each parameter or batch, take a random simulation draw given the data and the current estimate of all the other parameters. (We illustrate with some examples below.)

2. Evaluate the mixing of the simulated chains using the $\hat{R}$ summary, which we have already discussed in Section 16.4 in the context of interpreting the output from Bugs models.
3. If convergence is poor, run longer or alter the model, following the advice in Section 16.9.

The key part of this algorithm is the sequential updating step. Bugs performs it automatically, but here we will show how to compute Gibbs updates "manually" in R for multilevel linear regressions. Our purpose is not to set you up to program these yourself but rather to give enough insight that you can understand roughly how Bugs works, and thus better diagnose and fix problems when Bugs is not working so well.

We present in this section the steps of Gibbs sampling for a series of multilevel linear regressions: first a model with no predictors, then including a predictor at the individual level, then adding one at the group level.

The basic Gibbs sampler structure described here works for multilevel regressions, with the new twist that the regression coefficients can be estimated using an adaptation of classical least squares regression. (Model (18.10) can be considered as a special case of regression with only an intercept and no slope parameters, but this case is so simple that least squares matrix computations were not required.)

Gibbs sampler for a multilevel model with no predictors

We first go through the steps of the Gibbs sampler—mathematically and as programmed in R—for the multilevel model (18.10) with data in groups and no predictors.

The Gibbs sampler starts with initial values for all the parameters and then updates the parameters in turn, giving each a random estimate based on the data and the current guess of the other parameters in the model. For the simple model we are considering here, the Gibbs updating steps are:

1. Update α: For $j = 1, \dots, J$, compute $\hat{\alpha}_j$ and V_j from (18.12) and then draw α_j from the normal distribution with mean $\hat{\alpha}_j$ and variance V_j.
2. Update μ_α: Compute $\hat{\mu}_\alpha$ from (18.14) and then draw μ_α from the normal distribution with mean $\hat{\mu}_\alpha$ and variance σ_α^2/J.
3. Update σ_y: Compute $\hat{\sigma}_y^2$ from (18.13) and then draw $\sigma_y^2 = \hat{\sigma}_y^2/X_{n-1}^2$, where X_{n-1}^2 is a random draw from a χ^2 distribution with $n-1$ degrees of freedom.
4. Update σ_α: Compute $\hat{\sigma}_\alpha^2$ from (18.15) and then draw $\sigma_\alpha^2 = \hat{\sigma}_\alpha^2/X_{J-1}^2$, where X_{J-1}^2 is a random draw from a χ^2 distribution with $J-1$ degrees of freedom.

Each of these steps should seem reasonable; however, the details (such as the χ^2 distributions and their degrees of freedom) are not particularly intuitive and must be derived using probability calculations that are beyond the scope of this book. Each step uses random simulations rather than point estimates so that the procedure captures the inferential uncertainty about the parameters.

Iterating the above four steps produces a “chain” of simulation draws—a sequence of simulations $\alpha_1, \dots, \alpha_J$; σ_y; μ_α; $\alpha_1, \dots, \alpha_J$; σ_y; μ_α; and so forth. Looking at any single one of these parameters, we have a sequence of simulations that, if the chain is run long enough, captures the range of uncertainty in the estimation of that parameter. We start several chains with random initial values and then run until the chains have mixed (see Figure 16.2 on page 357).

Programming the Gibbs sampler in R

When Bugs fits model (18.10), it performs a series of computations that are similar to the steps just given. To understand in more detail, we program them here in R. For many applications, we can simply use Bugs, but when computational speed is a concern (for example, with large datasets), or for some complicated models (for example, the social networks model in Section 15.3), it can be necessary to code the Gibbs sampler directly.

We program the Gibbs sampler in three steps: setting up the data, writing functions for the individual parameter updates, and writing a loop for the actual computation. For the radon example, we have already set up the data vector `y` and the vector of county indexes `county`, and we are ready to program the parameter updates.

R code

```
a.update <- function() {
  a.new <- rep (NA, J)
  for (j in 1:J){
    n.j <- sum (county==j)
    y.bar.j <- mean (y[county==j])
    a.hat.j <- ((n.j/sigma.y^2)*ybar.j + (1/sigma.a^2)*mu.a)/
               (n.j/sigma.y^2 + 1/sigma.a^2)
    V.a.j <- 1/(n.j/sigma.y^2 + 1/sigma.a^2)
    a.new[j] <- rnorm (1, a.hat.j, sqrt(V.a.j))
  }
  return (a.new)
}
mu.a.update <- function() {
  mu.a.new <- rnorm (1, mean(a), sigma.a/sqrt(J))
  return (mu.a.new)
}
sigma.y.update <- function() {
  sigma.y.new <- sqrt(sum((y-a[county])^2)/rchisq(1,n-1))
```

```
  return (sigma.y.new)
}
sigma.a.update <- function() {
  sigma.a.new <- sqrt(sum((a-mu.a)^2)/rchisq(1,J-1))
  return (sigma.a.new)
}
```

These functions have empty argument lists (for example, `sigma.a.update` uses α, μ_α, and J, but these variables are not passed to as arguments in the function call) because we find it convenient to define all variables globally when putting the functions together. Passing functions through argument lists is cleaner in a programming sense but in this context can lead to confusion when models get altered, with parameters added and removed.

Another approach to programming these Gibbs updates would be to write general updating functions for the normal and inverse-χ^2 distributions and to call these by passing arguments through the functions.

In any case, having created the updating functions, we now create the space for three independent chains of length 1000 and give names to the parameters that will be saved in a large array, `sims`, that will contain posterior simulation draws for $\alpha, \mu_\alpha, \sigma_y, \sigma_\alpha$:

R code

```
n.chains <- 3
n.iter <- 1000
sims <- array (NA, c(n.iter, n.chains, J+3))
dimnames (sims) <- list (NULL, NULL,
  c (paste ("a[", 1:J, "]", sep=""), "mu.a", "sigma.y", "sigma.a"))
```

This last bit looks confusing; after running the command in R, it is helpful to type `dimnames(sims)` to see what this name object looks like, and to type `sims[1:5,1,]` to see how the names attach themselves to the `sims` object (or, in this case, the first five steps of the first chain of the `sims` object).

We are now ready to run the Gibbs sampler, first initializing $\mu_\alpha, \sigma_y, \sigma_\alpha$ with random values set crudely based on the range of the data—we need not initialize α because it is updated in the first step in the loop—and then simulating three chains for 1000 iterations each:

R code

```
for (m in 1:n.chains){
  mu.a <- rnorm (1, mean(y), sd(y))
  sigma.y <- runif (1, 0, sd(y))
  sigma.a <- runif (1, 0, sd(y))
  for (t in 1:n.iter){
    a <- a.update ()
    mu.a <- mu.a.update ()
    sigma.y <- sigma.y.update ()
    sigma.a <- sigma.a.update ()
    sims[t,m,] <- c (a, mu.a, sigma.y, sigma.a)
  }
}
```

We then summarize the simulations—view the inferences and check convergence—by using the `as.bugs.array` function to convert them to a Bugs object:

R code

```
sims.bugs <- as.bugs.array (sims)
plot (sims.bugs)
```

Gibbs sampler for a multilevel model with regression predictors

We can easily adapt the above algorithm to include predictors at the individual and group levels. Consider the model

$$\begin{aligned} y_i &\sim \mathrm{N}(\alpha_{j[i]} + X_i\beta, \sigma_y^2) \text{ for } i = 1, \dots, n \\ \alpha_j &\sim \mathrm{N}(U_j\gamma, \sigma_\alpha^2) \text{ for } j = 1, \dots, J, \end{aligned}$$

where X is a matrix of individual-level predictors (without a constant term), U is a matrix of group-level predictors (including a constant term), and β and γ are vectors of coefficients.

After initializing the parameters $\alpha, \beta, \gamma, \sigma_y, \sigma_\alpha$ with random numbers (constraining the σ parameters to be positive), the Gibbs sampler can be implemented as follows:

1. Update α: It is simplest to use the reexpression, $\alpha_j = U_j + \eta_j$; the η_j's are group-level errors that are partially pooled toward their mean of 0. We apply (18.12) to suitably adjusted data y, correcting for individual- and group-level predictors. For each data point, compute $y_i^{\text{temp}} = y_i - X_i\beta - U_{j[i]}\gamma$. Then for $j = 1, \dots, J$, compute $\hat{\eta}_j$ and V_j from (18.12)—but using y^{temp} in place of y—and draw η_j from the normal distribution with mean $\hat{\eta}_j$ and variance V_j. We complete the updating step by setting each α_j to $U_j\gamma + \eta_j$.
2. Update β: For each data point, compute $y_i^{\text{temp}} = y_i - \alpha_{j[i]}$. Then regress y^{temp} on X to obtain an estimate $\hat{\beta}$ and covariance matrix V_β, inserting σ_y for σ in equation (18.8). Now draw β from the $\mathrm{N}(\hat{\beta}, V_\beta)$ distribution.
3. Update γ: Regress α on U (this is a regression with J data points) to obtain an estimate $\hat{\gamma}$ and covariance matrix V_γ, inserting σ_α for σ in (18.8). Now draw γ from the $\mathrm{N}(\hat{\gamma}, V_\gamma)$ distribution.
4. Update σ_y: Compute $\hat{\sigma}_y^2 = \frac{1}{n}\sum_{i=1}^n (y_i - \alpha_{j[i]} - X_i\beta)^2$ and then draw $\sigma_y^2 = \hat{\sigma}_y^2 / X_{n-1}^2$, where X_{n-1}^2 is a random draw from a χ^2 distribution with $n-1$ degrees of freedom.
5. Update σ_α: Compute $\hat{\sigma}_\alpha^2 = \frac{1}{J}\sum_{i=1}^J (\alpha_j - U_j\gamma)^2$ and then draw $\sigma_\alpha^2 = \hat{\sigma}_\alpha^2 / X_{J-1}^2$, where X_{J-1}^2 is a random draw from a χ^2 distribution with $J-1$ degrees of freedom.

This algorithm combines partial pooling (step 1) with classical regression estimation of coefficients (steps 2 and 3) and standard errors (steps 4 and 5). It all works to summarize uncertainty because the parameters are updated iteratively, leading to an inference that includes all aspects of the model.

Programming in R involves writing functions for each of the five steps:

R code

```
a.update <- function() {
  y.temp <- y - X%*%b - U[county]%*%g
  eta.new <- rep (NA, J)
  for (j in 1:J){
    n.j <- sum (county==j)
    y.bar.j <- mean (y.temp[county==j])
    eta.hat.j <- ((n.j/sigma.y^2)*y.bar.j/
                  (n.j/sigma.y^2 + 1/sigma.a^2))
    V.eta.j <- 1/(n.j/sigma.y^2 + 1/sigma.a^2)
    eta.new[j] <- rnorm (1, eta.hat.j, sqrt(V.eta.j))
  }
  a.new <- U%*%g + eta.new
  return (a.new)
```

```
  }
  b.update <- function() {
    y.temp <- y - a[county]
    lm.0 <- lm (y.temp ~ X)
    b.new <- sim (lm.0, n.sims=1)
    return (b.new)
  }
  g.update <- function() {
    lm.0 <- lm (a ~ U)
    g.new <- sim (lm.0, n.sims=1)
    return (g.new)
  }
  sigma.y.update <- function() {
    sigma.y.new <- sqrt(sum((y-a[county]-X%*%b)^2)/rchisq(1,n-1))
    return (sigma.y.new)
  }
  sigma.a.update <- function() {
    sigma.a.new <- sqrt(sum((a-U%*%g)^2)/rchisq(1,J-1))
    return (sigma.a.new)
  }
```

(In the calls to `lm()` in the `b.update` and `g.update` functions, we have specified the predictors in matrix form rather than as a formula listing the individual predictor names.)

Now that the updating functions have been written, the Gibbs sampler can be programmed and run as in the example earlier in this section, simply expanding to include the new parameters.

The Gibbs sampler as a general way of working with multilevel models

On page 239 we discussed the simple two-step procedure of first regressing y on X and group indicators to estimate β and the α_j's, then regressing the estimated α_j's on U to estimate γ. Multilevel modeling, in its Gibbs implementation, can be seen as a generalization of two-step regression in which the α_j's are estimated more accurately using partial pooling. Similarly, in more complicated multilevel structures, it often makes sense to program a Gibbs sampler in a way that alternately performs group-level regressions and separate inferences within each group.

If we were to start with a simple no-pooling, complete-pooling, or two-step analysis, and then gradually improve it to account for estimation uncertainty in each step, iterating to allow inferences to be based on the latest estimate for each parameter, then we would end up with a Gibbs sampler.

18.5 Likelihood inference, Bayesian inference, and the Gibbs sampler: the case of censored data

We illustrate some of the ideas of likelihood and Bayesian inference for a *censored-data* model. We begin with a regression of weight (in pounds) on height (in inches), using data from a random sample of Americans. Before fitting, we center the height variable (`c.height <- height - mean(height)`) so that we can better interpret the intercept as well as the slope of the regression:

R output

```
lm(formula = weight ~ c.height)
            coef.est coef.se
(Intercept)    156.1     0.6
```

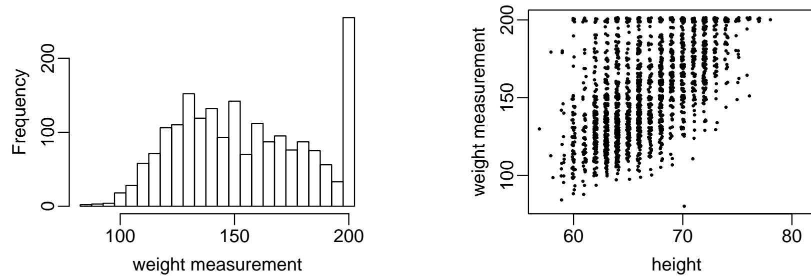

Figure 18.8 *(a) Histogram of weights as recorded on a hypothetical scale that censors measurements at 200; (b) Plot of (jittered values of) measured weight versus height in a sample of adults. The relation between height and weight is clear for low heights but becomes more difficult to follow at the high end, where the censoring becomes more frequent.*

```
c.height          4.9     0.2
  n = 1984, k = 2
  residual sd = 28.6, R-Squared = 0.30
```

Censoring

Now imagine that weights had been measured on a scale that was limited to a maximum of 200, so that any weights greater than 200 were recorded as "200^+" with the superscript indicating the censoring. We artificially perform this censoring on the survey data:

R code

```
C <- 200
censored <- weight >= C
y <- ifelse (censored, C, weight)
```

From here on, we suppose that the censored variable, `y`, is what was observed, with the true weight only known if $y < 200$. Figure 18.8a shows the measured weights y, and Figure 18.8b shows weight plotted against height.

Naive regression estimate excluding the censored data

The two simple (but wrong) analyses of these data are to ignore the censored measurements, or to include them as measurements of 200. Here is the regression discarding the measurements of 200^+:

R output

```
lm(formula = y ~ c.height, subset = y<200)
            coef.est coef.se
(Intercept)    148.7     0.5
c.height         3.8     0.1
  n = 1739, k = 2
  residual sd = 20.5, R-Squared = 0.31
```

Both the intercept and slope are too low, which makes sense given Figure 18.8b. This analysis excludes the largest values of weight and thus underestimates the average weight in the population. Also, more data are censored at the high end of heights, so the slope is underestimated too.

Naive regression estimate imputing the censoring point

Another simple but wrong approach is to simply code the 200^+ measurements as 200, which yields the following regression fit:

R output

```
lm(formula = y ~ c.height)
            coef.est coef.se
(Intercept)    153.6     0.5
c.height         4.3     0.1
  n = 1984, k = 2
  residual sd = 23.8, R-Squared = 0.32
```

Once again, this underestimates both the intercept (by using $y = 200$ in cases where we know the true weight is *at least* 200) and also the slope (because more of this bias occurs for taller persons).

Likelihood function accounting for the censoring. A better way to account for the censoring is to include it explicitly in the likelihood function. We write the censoring formally as

$$y_i = \begin{cases} z_i & \text{if } z_i \leq 200 \\ 200^+ & \text{if } z_i > 200, \end{cases} \tag{18.17}$$

with a linear regression for the true weights z_i given heights x_i:

$$y_i \sim \mathrm{N}(a + bx_i, \sigma^2). \tag{18.18}$$

For the uncensored data points, the likelihood is simply the normal distribution, $\mathrm{N}(y_i|a + bx_i, \sigma^2)$, as in (18.3) on page 388. For a censored measurement, the likelihood is

$$\Pr(y = 200^+) = \Pr(z_i \geq 200) = \int_{200}^{\infty} \mathrm{N}(z_i \,|\, a + bx_i, \sigma^2) = \Phi\left(\frac{a + bx_i - z_i}{\sigma}\right),$$

where Φ is the normal cumulative distribution function (which can be computed using the `pnorm()` function in R).

The likelihood of all the data is then

$$p(y|\beta, \sigma, x) = \prod_{i=1}^{n} p(y_i|\beta, \sigma, x_i), \tag{18.19}$$

where the individual factors of the likelihood are

$$p(y_i|\beta, \sigma, X) = \begin{cases} \mathrm{N}(y_i|a + bx_i, \sigma^2) & \text{if } y_i < 200 \\ \Phi((a + bx_i - 200)/\sigma) & \text{if } y_i = 200^+. \end{cases} \tag{18.20}$$

We shall clarify this expression (we hope) by programming it in R.

Maximum likelihood estimate using R

We shall first program the likelihood function in R and then call an optimization routine to find the maximum. In programming the likelihood, it is convenient to express the unknown parameters (in this case, a, b, and σ) as a vector, and then include the data and censoring point as additional arguments to the function. The following function computes the logarithm of the likelihood by evaluating (18.20) one data point at a time and then adding these values (which, on the log scale, is equivalent to the multiplication in (18.19)):

R code

```
Loglik <- function (parameter.vector, x, y, C) {
  a <- parameter.vector[1]
  b <- parameter.vector[2]
  sigma <- parameter.vector[3]
  ll.vec <- ifelse (y<C, dnorm (y, a + b*x, sigma, log=TRUE),
    pnorm ((a + b*x - C)/sigma, log=TRUE))
```

```
    return (sum (ll.vec))
  }
```

We have used the `log=TRUE` options of the `dnorm()` and `pnorm()` functions so that R automatically computes the logarithms of these probabilities. It is more computationally stable to compute probabilities on the log scale and only exponentiate at the end of the calculations.

To find the values of a, b, σ that maximize the log likelihood, we use the `optim()` function in R, which requires initial values (for which we simply use uniformly distributed random numbers) and some specifications:[2]

R code

```
inits <- runif (3)
mle <- optim (inits, Loglik,  lower=c(-Inf,-Inf,1.e-5),
  method="L-BFGS-B", control=list(fnscale=-1), x=c.height,
  y=weight.censored, C=200)
```

We check by typing `print(mle$convergence)` (which should take on the value 0; type `?optim` in R for more details) and then typing

R code

```
mle$par
```

to find the vector of maximum likelihood estimates, which in this case are 155, 4.8, and 26.5 (corresponding to $\hat{a}$, $\hat{b}$, and $\hat{\sigma}$, respectively).

Fitting the censored-data model using Bugs

Bugs model. Another way of fitting the censoring model, more consistent with the general approach of this book, is to write it in Bugs. The trick here is to express the model in terms of the true weights, z_i as defined in (18.17), which follow the regression model (18.18). For the censored data (the measurements $y_i = 200^+$, the true weights are unobserved but are constrained to fall in the range $(200, \infty)$. In Bugs, we express this constraint as a lower bound of 200 as follows:

Bugs code

```
model {
  for (i in 1:n){
    z.lo[i] <- C*equals(y[i],C)
    z[i] ~ dnorm (z.hat[i], tau.y) I(z.lo[i],)
    z.hat[i] <- a + b*x[i]
  }
  a ~ dnorm (0, .0001)
  b ~ dnorm (0, .0001)
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)
}
```

The `I(z.lo[i],)` factor constrains the distribution for $z[i]$ to be above `z.lo[i]`,[3] and this lower bound has been defined using `equals` to equal C (that is, 200) for censored observations and 0 otherwise.[4]

[2] In addition to specifying the vector of initial values and the name of the function to be optimized, we need to constrain σ to be positive—this is done by assigning a vector of lower limits to all three parameters, with empty $-\infty$ limits set for a and b. We then must set `method="L-BFGS-B"`, which is the "box constraint" algorithm that allows for bounds on the parameters. Setting `control=list(fnscale=-1)` tells `optim()` to find a maximum, rather than a minimum, of the specified function. Finally, we must specify the values of the other inputs to the `Loglik()` function, which in this case are x, y, and C.

[3] The factor `I(,z.hi[i])` would restrict the distribution to be *lower* than some value `z.hi[i]`, and `I(z.lo[i],z.hi[i])` would constrain to a finite range; see the models in Sections 6.5 and 17.7 for other examples of constrained distributions.

[4] A lower bound of zero is fine given that true weights are always positive. If there were no such natural bound—for example, if we were modeling log(weights)—then one could use an

Fitting the Bugs model from R. To fit the model in Bugs, we must first define z, which equals the weight when observed and is missing otherwise:

R code
```
z <- ifelse (censored, NA, weight.censored)
```

We then set up the `bugs()` call as usual:

R code
```
data <- list (x=c.height, y=weight.censored, z=z, n=n, C=C)
inits <- function() {
  list (a=rnorm(1), b=rnorm(1), sigma.y=runif(1))}
params <- c ("a", "b", "sigma.y")
censoring.1 <- bugs (data, inits, params, "censoring.bug", n.iter=100)
```

Reproducing some of the output from `print(censoring.1)`:

R output
```
          mean  sd   2.5%    25%    50%    75%   97.5% Rhat n.eff
a        155.3 0.6  154.2  154.9  155.4  155.8  156.4    1   150
b          4.8 0.2    4.5    4.7    4.7    4.9    5.2    1   150
sigma.y   26.5 0.5   25.7   26.1   26.5   26.8   27.3    1   150
```

This inference is essentially the same as the maximum likelihood estimate—which makes sense, given that the sample size is large and the number of parameters is small—but are clearly different from the naive estimates excluding the censored data ($\hat{b} = 3.8 \pm 0.1$) and ($\hat{b} = 4.3 \pm 0.1$). The censoring model appropriately imputes the missing values, which we know lie above 200.

Gibbs sampler

Yet another way to fit this model is by programming a Gibbs sampler in R, as follows:

1. Impute crude starting values for the missing data—the true weights z_i corresponding to measurements $y = 200^+$.
2. Iterate the following two steps:
   (a) Run a regression of z (including the imputed values) on x and take a random draw from the uncertainty distribution of the parameters a, b, σ.
   (b) Use the estimated a, b, σ to create random imputations of the missing data.

For this problem, there is no real reason to program these steps; as we have just seen, the model is easy to fit in Bugs. This is, however, a good example to illustrate the way the Gibbs sampler handles uncertainty about missing data. We shall give the algebra and R code for each of the above steps.

Crude starting values. We simply impute a random value between C and $2C$ (in our example, 200 and 400 pounds) for each of the missing weights:

R code
```
n.censored <- sum (censored)
z[censored] <- runif (n.censored, C, 2*C)
```

Regression if the exact weights were known. We fit a regression and then draw one simulation value for the parameters a, b, σ:

R code
```
x <- c.height
lm.1 <- lm (z ~ x)
sim.1 <- sim (lm.1, n.sims=1)
a <- sim.1$beta[1]
b <- sim.1$beta[2]
sigma <- sim.1$sigma
```

assignment such as `z.lo[i] <- -1.E5 + (1.E5+C)*equals(y[i],C)` to set an extremely low bound for uncensored cases.

Imputing the missing values given the fitted regression. The predictive distribution for any particular censored value i is normal with mean $a + bx_i$ and standard deviation σ, but constrained to be at least 200. We can write an R function to draw from this constrained distribution:

R code
```
rnorm.trunc <- function (n, mu, sigma, lo=-Inf, hi=Inf) {
  p.lo <- pnorm (lo, mu, sigma)
  p.hi <- pnorm (hi, mu, sigma)
  u <- runif (n, p.lo, p.hi)
  return (qnorm (u, mu, sigma))
}
```

This function first locates the constraint points (set by default to $(-\infty, \infty)$ if no constraints are given) in the specified distribution, then draws a sample within these probabilities, and finally transforms back to the original scale. We have written the function to take n independent draws, by analogy to the `rnorm()` function (type `?rnorm` in R for details); this is not the same n that is the length of the data vector y.

We can then use this function to sample the missing z_i's given their predictors x_i:

R code
```
z[censored] <- rnorm.trunc (n.censored, a + b*x[censored], sigma, lo=C)
```

Gibbs sampler: putting it together in a loop. We can now produce a Gibbs sampler. We first set up a space for 3 chains of 100 iterations each, saving $3 + n_{\text{censored}}$ parameters corresponding to a, b, σ, and the unobserved z_i's:

R code
```
n.chains <- 3
n.iter <- 100
sims <- array (NA, c(n.iter, n.chains, 3 + n.censored))
dimnames (sims) <- list (NULL, NULL,
  c ("a", "b", "sigma", paste ("z[", (1:n)[censored], "]", sep="")))
```

We then program a Gibbs sampler, looping over the 3 chains: each chain starts with random initial values, then a loop through 100 iterations, first updating a, b, σ and then updating the censored components of z, and saving all these parameters at the end of each iteration.

R code
```
for (m in 1:n.chains){
  z[censored] <- runif (n.censored, C, 2*C)  # random initial values
  for (t in 1:n.iter){
    lm.1 <- lm (z ~ x)
    sim.1 <- sim (lm.1, n.sims=1)
    a <- sim.1$beta[1]
    b <- sim.1$beta[2]
    sigma <- sim.1$sigma
    z[censored] <- rnorm.trunc (n.censored, a + b*x[censored], sigma, lo=C)
    sims[t,m,] <- c (a, b, sigma, z[censored])
  }
}
```

Finally, we check the convergence:

R code
```
sims.bugs <- as.bugs.array (sims)
print (sims.bugs)
```

yielding:

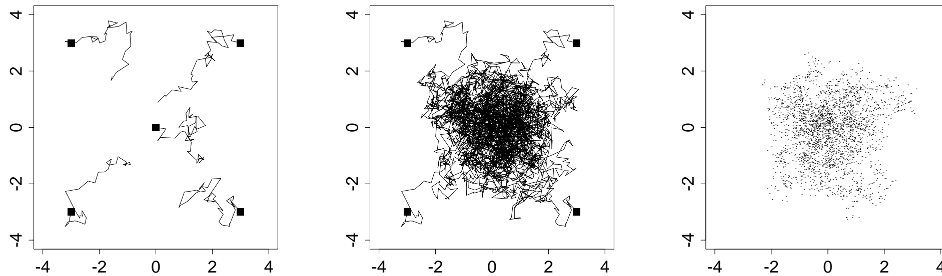

Figure 18.9 *Five independent sequences of a Metropolis algorithm, with overdispersed starting points indicated by solid squares. (a) After 50 iterations, the sequences are still far from convergence. (b) After 1000 iterations, with the sequences nearer to convergence. (c) The iterates from the second halves of the sequences, jittered so that steps in which the random walks stood still are not hidden.*

R output

```
          mean   sd  2.5%   25%   50%   75% 97.5% Rhat n.eff
a        155.3  0.6 154.1 154.9 155.3 155.7 156.3  1.0    53
b          4.8  0.2   4.5   4.7   4.8   4.9   5.1  1.0   150
sigma     26.5  0.5  25.7  26.2  26.5  26.9  27.6  1.0    69
z[3]     216.9 13.0 200.7 206.1 215.4 223.7 247.7  1.0    87
z[6]     210.9  9.3 200.3 203.5 208.9 214.9 234.1  1.0   150
z[11]    208.5  7.1 200.3 203.4 206.6 211.7 226.2  1.0   150
. . .
```

which is essentially identical to the results from the Bugs run (as it should be, given that we are fitting the same model).

Section 25.6 briefly describes a more realistic and complicated example of censoring that arises in a study of reversals of the death penalty, in which cases are censored that are still under consideration by appellate courts.

18.6 Metropolis algorithm for more general Bayesian computation

Moving to even more general models, the Gibbs sampler is a special case of a larger class of *Markov chain simulation algorithms* that can be used to iteratively estimate parameters in any statistical model. Markov chain simulation in general (and the Gibbs sampler in particular) can be thought of as iterative imputation of unknown parameters, or as a random walk through parameter space.

The Gibbs sampler updates the parameters one at a time (or in batches) using their conditional distributions. It can also be efficient to use the *Metropolis algorithm*, which takes a random walk through the space of parameters.

The Gibbs sampler and Metropolis algorithms are special cases of Markov chain simulation (also called *Markov chain Monte Carlo*, or MCMC), a general method based on drawing values of θ from approximate distributions and then correcting those draws to better approximate the target posterior distribution, $p(\theta|y)$. The samples are drawn sequentially, with the distribution of the sampled draws depending on the last value drawn; hence, the draws form a Markov chain. (As defined in probability theory, a *Markov chain* is a sequence of random variables $\theta^{(1)}, \theta^{(2)}, \ldots$, for which, for any t, the distribution of $\theta^{(t)}$ given all previous θ's depends only on the most recent value, $\theta^{(t-1)}$.) The key to the method's success, however, is not the Markov property but rather that the approximate distributions are improved at each step in the simulation, in the sense of converging to the target distribution.

Figure 18.9 illustrates a simple example of a Markov chain simulation—in this

case, a Metropolis algorithm in which θ is a vector with only two components, with a bivariate unit normal posterior distribution, $\theta \sim \mathrm{N}(0, I)$. First consider Figure 18.9a, which portrays the early stages of the simulation. The space of the figure represents the range of possible values of the multivariate parameter, θ, and each of the five jagged lines represents the early path of a random walk starting near the center or the extremes of the target distribution and jumping through the distribution according to an appropriate sequence of random iterations. Figure 18.9b represents the mature stage of the same Markov chain simulation, in which the simulated random walks have each traced a path throughout the space of θ, with a common stationary distribution that is equal to the target distribution. From a simulation such as 18.9b, we can perform inferences about θ using points from the second halves of the Markov chains we have simulated, as displayed in Figure 18.9c.

It is useful to have some sense of how the Metropolis algorithm works, because it is a key part of Bugs and other programs that perform iterative simulation. For further details on programming the Metropolis algorithm, see Gelman et al. (2003).

18.7 Specifying a log posterior density, Gibbs sampler, and Metropolis algorithm in R

To compute Markov chain simulation, the posterior density function and Gibbs sampler steps must be given. Bugs sets up these specification automatically (determining them from the model file) but for certain applications in which Bugs does not run or is too slow, it is necessary to program the log posterior density and Gibbs sampler steps explicitly.

We illustrate for the overdispersed Poisson regression model for social networks from Section 15.3, in which the large number of parameters (more than 1400) makes Bugs too slow to be practical. Instead, we use Umacs (universal Markov chain sampler), a program under development that performs Gibbs and Metropolis sampling given a specified posterior distribution. We go through the steps here, partly to complete the fitting of the social network model and partly to illustrate Markov chain sampling on a relatively complicated example.

The joint posterior density

The joint posterior density of the model in Section 15.3 can be written as

$$p(\alpha, \beta, \omega, \mu_\alpha, \mu_\beta, \sigma_\alpha, \sigma_\beta | y) \propto \prod_{i=1}^{n} \prod_{k=1}^{K} \binom{y_{ik} + \xi_{ik} - 1}{\xi_{ik} - 1} \left(\frac{1}{\omega_k}\right)^{\xi_{ik}} \left(\frac{\omega_k - 1}{\omega_k}\right)^{y_{ik}} \times \prod_{i=1}^{n} \mathrm{N}(\alpha_i | \mu_\alpha, \sigma_\alpha^2) \prod_{k=1}^{K} \mathrm{N}(\beta_k | \mu_\beta, \sigma_\beta^2) \prod_{k=1}^{K} \omega_k^{-2}, \quad (18.21)$$

where $\xi_{ik} = e^{\alpha_i + \beta_k}/(\omega_k - 1)$, from the definition of the negative binomial distribution. The first factor in the posterior density is the likelihood—the probability density function of data given the parameters—and the remaining factors are the population distributions for each of the α_j's, β_k's, and ω_k's. The prior $p(\omega_k) \propto \omega_k^{-2}$ is equivalent to a uniform prior distribution on $1/\omega$, using the "Jacobian" from probability theory to transform from $1/\omega$ to ω.[5]

In computing, we actually work with the logarithm of the posterior density func-

[5] See, for example, Gelman et al. (2003, p. 24) for an explanation of the Jacobian.

tion because then computations are more stable and less likely to result in overflows or underflows than when using the density function itself. We typically only specify the density up to a multiplicative constant (note the proportionality sign in (18.21)) or, equivalently, the log density up to an additive constant—but this is all that is needed for Gibbs/Metropolis calculations.

The simulation algorithm

Our Markov chain simulation for the social network model requires the following steps:

- Gibbs sampler on the hyperparameters $\mu_\alpha, \sigma_\alpha, \mu_\beta, \sigma_\beta$.
- Metropolis jumping for each component of α, β, γ. Jump one vector at once for computational convenience.
- Constraining the components of γ to keep them above 1.
- Renormalization at each step because we are working with an overspecified model. (As discussed in Section 15.3, the model is unchanged if a constant is added to all the components of α and β. This constant thus needs to be specified in some way to stop the simulations from drifting aimlessly.)
- Adaptive Metropolis updating to keep acceptance rates near the target of 44%.
- Simulation of three parallel chains.
- After burn-in: stop adaptation, run awhile, and check convergence.
- Summarize with random simulation draws.

We program the first four of the above items; the others are performed automatically by Umacs.

We obtain posterior simulations using a Gibbs-Metropolis algorithm, iterating the following steps:

1. For each i, update α_i using a Metropolis step with jumping distribution, $\alpha_i^* \sim \mathrm{N}(\alpha_i^{(t-1)}, (\text{jumping scale of } \alpha_i)^2)$.
2. For each k, update β_k using a Metropolis step with jumping distribution, $\beta_k^* \sim \mathrm{N}(\beta_i^{(t-1)}, (\text{jumping scale of } \beta_k)^2)$.
3. Update $\mu_\alpha \sim \mathrm{N}(\hat{\mu}_\alpha, \sigma_\alpha^2/n)$, where $\hat{\mu}_\alpha = \frac{1}{n}\sum_{i=1}^n \alpha_i$.
4. Update $\sigma_\alpha^2 \sim \text{Inv-}\chi^2(n-1, \hat{\sigma}_\alpha^2)$, where $\hat{\sigma}_\alpha^2 = \frac{1}{n}\sum_{i=1}^n (\alpha_i - \mu_\alpha)^2$.
5. Update $\mu_\beta \sim \mathrm{N}(\hat{\mu}_\beta, \sigma_\beta^2/n)$, where $\hat{\mu}_\beta = \frac{1}{K}\sum_{k=1}^K \beta_k$.
6. Update $\sigma_\beta^2 \sim \text{Inv-}\chi^2(K-1, \hat{\sigma}_\beta^2)$, where $\hat{\sigma}_\beta^2 = \frac{1}{K}\sum_{k=1}^K (\beta_k - \mu_\beta)^2$.
7. For each k, update ω_k using a Metropolis step with jumping distribution, $\omega_k^* \sim \mathrm{N}(\omega_k^{(t-1)}, (\text{jumping scale of } \omega_k)^2)$.
8. Rescale the α's and β's by computing the adjustment term C described on page 336 and adding it to all the α_i's and μ_α and subtracting it from all the β_k's and μ_β.

We construct starting points for the algorithm by fitting a classical Poisson regression (the null model, $y_{ik} \sim \text{Poisson}(\lambda_{ik})$, with $\lambda_{ik} = a_i b_k$) and then estimating the overdispersion for each subpopulation k using the statistic (6.5) on page 114.

Programming in R and Umacs

Setting up the model in Umacs requires several steps, which we include here as illustration of the details required to fully specify multilevel computation in R.

Log likelihood function. We take advantage of the matrix representation of the data $y = (y_{jk})$ to write a function that computes the log likelihood in parallel for all data points at once. In these expressions, y is a 1370×32 matrix; α is a vector of length 1370; β and ω are vectors of length 32, and `data.n`= 1370, the number of survey respondents.

R code
```
f.loglik <- function (y, a, b, o, data.n) {
  theta.mat <- exp (outer (a, b, "+"))
  O.mat <- outer (rep (1, data.n), o, "*")
  A.mat <- theta.mat/(O.mat-1)     # the "alpha" and "beta" parameters
  B.mat <- 1/(O.mat-1)             # of the negative binomial distribution
  loglik <- lgamma(y+A.mat) - lgamma(A.mat) - lgamma(y+1) +
    (log(B.mat)-log(B.mat+1))*A.mat - log(B.mat+1)*y
  return (loglik)
}
```

The expression for `loglik` is the logarithm of the negative binomial density function.

Log posterior density functions. Our next step is to write functions that compute the log posterior density for each vector of parameters; these log densities are formed by summing the log likelihood by row or column and then adding the log prior distribution. We write different log posterior density functions for each parameter vector in order to make computations more efficient (for example, in updating α, we only need to include factors that depend on this parameter):

R code
```
f.logpost.a <- function() {
  loglik <- f.loglik (y, a, b, o, data.n)
  rowSums (loglik, na.rm=TRUE) + dnorm (a, mu.a, sigma.a, log=TRUE)
}
f.logpost.b <- function() {
  loglik <- f.loglik (y, a, b, o, data.n)
  colSums (loglik, na.rm=TRUE) + dnorm (b, mu.b, sigma.b, log=TRUE)
}
f.logpost.o <- function() {
  reject <- !(o>1)          # reject if omega is not greater than 1
  o[reject] <- 2            # set rejected omega's to arbitrary value of 2
  loglik <- f.loglik (y, a, b, o, data.n)
  loglik <- colSums (loglik, na.rm=TRUE) - 2*log(o)
  loglik[reject] <- -Inf    # set loglik to zero for rejected values
  return (loglik)
}
```

We constrain the components of ω to be greater than 1.01 because the model restricts this parameter to be greater than 1, and we want to avoid potential numerical difficulties when any ω_k is exactly 1.

Data and initial values. Next, we load in the data:

R code
```
library ("foreign")
y <- as.matrix (read.dta ("all.dta"))
```

and define initial values for the parameters:

R code
```
a.init       <- function() {rnorm (data.n)}
b.init       <- function() {rnorm (data.j)}
o.init       <- function() {runif (data.j, 1.01, 50)}
mu.a.init    <- function() {rnorm (1)}
mu.b.init    <- function() {rnorm (1)}
sigma.a.init <- function() {runif (1)}
sigma.b.init <- function() {runif (1)}
```

The initial values (as well as the log posterior densities defined above) are set up as functions with no arguments because Umacs uses the variables in the workspace rather than passing data and parameters back and forth among functions.[6]

Gibbs sampler steps. Having defined the model, data, and inital values, we write the functions for the Gibbs samplers for the hyperparameters:

R code
```
mu.a.update <- function() {
  rnorm (1, mean(a), sigma.a/sqrt(data.n))
}
mu.b.update <- function() {
  rnorm (1, mean(b), sigma.b/sqrt(data.j))
}
sigma.a.update <- function() {
  sqrt (sum((a-mu.a)^2)/rchisq(1, data.n-1))
}
sigma.b.update <- function() {
  sqrt (sum((b-mu.b)^2)/rchisq(1, data.j-1))
}
```

Renormalization step. We next write a function for the renormalization of the α's and β's in terms of the frequencies of the names in the population:

R code
```
renorm.network <- function() {
  const <- log (sum(exp(b[c(2,4,12)]))/.00357) +
    .5*log (sum(exp(b[c(3,7)]))/.00760) -
    .5*log (sum(exp(b[c(6,8,10)]))/.00811)
  a <- a + const
  mu.a <- mu.a + const
  b <- b - const
  mu.b <- mu.b - const
}
```

Setting up the Umacs sampler function. We are now ready to set up the series of steps for Metropolis and Gibbs sampling for the social network model. We update each of the vectors α, β, and ω using the `PSMetropolis()` routine, which stands for "scalar parallel Metropolis"—that is, separately updating each component using Metropolis jumping, automatically tuning these to jump efficiently.[7]

R code
```
s.network <- Sampler (
  y = y,
  data.n = nrow(y),
  data.j = ncol(y),
  a = PSMetropolis (f.logpost.a, a.init),
  b = PSMetropolis (f.logpost.b, b.init),
  o = PSMetropolis (f.logpost.o, o.init),
  mu.a = Gibbs (mu.a.update, mu.a.init),
  mu.b = Gibbs (mu.b.update, mu.b.init),
  sigma.a = Gibbs (sigma.a.update, sigma.a.init),
  sigma.b = Gibbs (sigma.b.update, sigma.b.init),
  renorm.network)
```

This call to `Sampler()` creates a function, `s.network()`, which we can then call to perform the actual sampling.

[6] As discussed on page 400, this "global variable" structure makes it easier for us to expand models without having to worry about keeping track of the parameters used in each function call. Other programming strategies are also possible.

[7] Umacs also includes vector Metropolis jumping, in which several components of a vector are altered at once, but in this case the posterior density for each vector of parameters factors into its components, so these components can be efficiently updated in parallel.

Running Umacs and saving the simulations. Finally, we run the sampler for three parallel chains for 2000 iterations, keeping the last 1000. We save the output as a Bugs object and plot it.

R code
```
network.1 <- s.network (n.iter=2000, n.sims=1000, n.chains=3)
network.1.bugs <- as.bugs.array (network.1)
plot (network.1)
```

We can then check convergence (by looking at the values of $\widehat{R}$ in the plot), access the simulations using `attach.bugs(network.1.bugs)`, and make the plots shown in Section 15.3.

18.8 Bibliographic note

For a fuller presentation of our perspective on likelihood and Bayesian data analysis, see Gelman et al. (2003). Other presentations of Bayesian inference include Box and Tiao (1973), Bernardo and Smith (1994), and Carlin and Louis (2001).

For more on prior distributions, see Jeffreys (1961), Jaynes (1983), Box and Tiao (1973), and Meng and Zaslavsky (2002). Many of the concerns in this literature are less urgent in multilevel models, in which most parameters are themselves modeled at the group level—but the prior distribution can still be relevant for the few remaining hyperparameters of any model. Our approach of prior distributions as placeholders or "reference models" follows Bernardo (1979); see also Kass and Wasserman (1996).

Full Bayesian analysis for multilevel models was first performed by Hill (1965), Tiao and Tan (1965, 1966), and Tiao and Box (1967). Important later work includes Lindley and Smith (1972), Efron and Morris (1975), Dempster, Rubin, and Tustakawa (1981), Gelfand and Smith (1990), and Pauler, Wakefield, and Kass (1999).

See Gilks, Richardson, and Spiegelhalter (1996) for more on the Gibbs sampler and the Metropolis algorithm. For an introduction to Bayesian inference for censoring and truncation, see Gelman et al. (2003, section 7.8).

The social network example comes from Zheng, Salganik, and Gelman (2006). Umacs is described by Kerman (2006) and Kerman and Gelman (2006).

18.9 Exercises

1. Linear regression algebra: show that weighted least squares is maximum likelihood estimation for the model (18.7).
2. Bayesian inference: take a multilevel linear model that you have already fit, and make a graph such as in Figure 18.5 or 18.6 showing likelihood, prior distribution, and posterior distribution, in each of several groups.
3. Maximum likelihood estimation: consider the logistic regression you set up in Exercise 5.8(a) for predicting presence of rodents in an apartment given ethnic group.
   (a) Write the likelihood for this model.
   (b) Program the likelihood function in R and use `optim()` to find the maximum likelihood estimate. Check that your estimate is the same as you obtained in Exercise 5.8(a).
4. Censored data: take the data on beauty and teaching evaluations data described in Exercise 3.5 and artificially censor by reporting all course evaluations below 3.0 simply as "*"

(a) Take one of the models from that earlier exercise and write the likelihood function given this mix of observed and censored data.

(b) Find the maximum likelihood estimate in R using the `optim()` function.

(c) Fit the model using Bugs, accounting for the censoring.

(d) Compare the censored-data inferences from the estimates using the complete data.