Chapter 17: Probabilistic Modelling with Stan

Mark Andrews

Contents

Introduction

In Chapter 8 and subsequent chapters, we described how the aim of Bayesian inference is to infer the posterior distribution of the assumed statistical model's parameters, which we can write as $\mathrm{P}(\theta|\mathcal{D})$, where θ is the set of unknown parameters and $\mathcal{D}$ is the observed data. In general, in all but a small number of cases, this posterior distribution, which we can think of as simply as a function over a (often very high) dimensional space, does not have an analytical description. In other words, in general, there is no formula that will give quantities that we seek such as the posterior's mean, standard deviation, high posterior density interval, posterior predictive distribution, etc. However, all of these quantities are *expectations* of the posterior distribution, which we can express as

$$\langle g(\theta)\rangle = \int g(\theta)\mathrm{P}(\theta|\mathcal{D})d\theta,$$

where $g(\theta)$ is some function of the parameters θ. We can approximate these expectations using the *Monte Carlo integration* method, which is where we draw samples from $\mathrm{P}(\theta|\mathcal{D})$ and then calculate the arithmetic mean of the function g applied to each one:

$$\langle g(\theta)\rangle \approx \frac{1}{n}\sum_{i=1}^{n} g(\tilde{\theta}_i),$$

where $\tilde{\theta}_1, \tilde{\theta}_2 \ldots \tilde{\theta}_n$ are n samples drawn drawn from $\mathrm{P}(\theta|\mathcal{D})$.

In Chapter 8, we also saw that Markov Chain Monte Carlo (MCMC) methods, which include techniques such as the *Metropolis* (or, more general, *Metropolis-Hastings*) sampler, the *Gibbs* sampler, and the *Hamiltonian Monte Carlo* variant of the Metropolis sampler are algorithms from drawing samples from high dimensional probability distributions that can be applied very generally. As useful and as general as these MCMC samplers are, they nonetheless are relatively arduous to implement in practice. Because samplers are computationally very intensive, they need to be implemented in lower level programming languages such as C/C++ or Fortran. Even when we have knowledge and experience with these languages, writing code in these language is more difficult, time consuming, and error prone than writing code in, say, R. In addition, there is often a lot of mathematical details about the models that we need to work out and then implement in code. For example, we must mathematically work out the likelihood function, which for large models may be relatively complex, and then we must implement this in code. For some samplers, some as Gibbs samplers or Hamiltonian Monte Carlo, further mathematical details, such as conditional distributions or the derivative of the posterior distribution must be worked out and implemented. We may then have to fine tune the samplers to maximize their efficiency. In addition, every time we change major aspects of our model, we must often rewrite substantial portions of our code. Overall, compared to any code covered in this book, these are major programming tasks that require time and effort that we simply may not be able to afford.

The aim of a *probabilistic programming language* (PPL) is to automate the implementation of the MCMC sampler. With a PPL, all we need do is specify our probabilistic model, including the priors, in a high level programming language. The sampler is then automatically derived and compiled and executed for us, and samples are then returned to us. The saving in terms of our time and effort can be remarkable. What might have taken days or even weeks of relatively tedious and error prone programming in C++ or Fortran, can now be accomplished in minutes by writing high level code.

In this chapter, we cover the Stan PPL, which is named after the Polish American Mathematician Stanislaw Ulam who was one of the inventors of the Monte Carlo in the late 1940's. The first stable release of Stan was in 2012, and it has grown steadily in popularity since then. Now it is arguably the dominant probabilistic programming language for Bayesian data analysis in statistics. Here, we will attempt to provide a self-contained tutorial introduction to Stan and how it can be used in R by using `rstan`.

Univariate models

Let us begin by considering some simple models, each one being defined essentially by probability distributions over a single observed variable.

Loaded die model

Let us begin with a very simple one parameter problem. Let us imagine that we have a die that is loaded to make an outcome of 6 more likely than other outcome. We throw this die $N = 250$ times and record the resulting face on each occasion. Simulated data of this kind is available in the following data set.

```
dice_df <- read_csv('data/loaded_dice.csv')
```

We can use `table` to count the number of outcomes of each face, and clearly there are more cases of 6 than other outcomes.

```
dice_df %>% table()
## .
##  1  2  3  4  5  6
## 31 38 35 42 31 73
```

We can recode each outcome as a "six" or "not-six" binary outcome.

```
dice_df %<>%
  mutate(is_six = ifelse(outcome == 6, 1, 0))
dice_df
## # A tibble: 250 x 2
##    outcome is_six
##      <dbl>  <dbl>
##  1       6      1
##  2       1      0
##   3       6      1
##   4       2      0
##   5       1      0
##   6       4      0
##   7       5      0
##   8       6      1
##   9       4      0
## 10       1      0
## # ... with 240 more rows
```

Bernoulli model

If we denote these binary outcomes as $y_1, y_2 \ldots y_n$, with each $y_i \in \{0, 1\}$, an assuming that there is a fixed probability θ that each $y_i = 1$, then our model of this data is as follows.

$$y_i \sim \text{Bernoulli}(\theta), \quad \text{for } i \in 1, 2 \ldots n.$$

A Bayesian model places a prior probability distribution over θ. When using a MCMC methods generally, especially with a PPL like Stan, we have practically endless choices for this prior. However, it must be a probability distribution over the real interval $[0, 1]$, and so an easy choice would be a Beta distribution with hyperparameters α and β, which we will assume are known. Thus, the complete Bayesian model is as follows.

$$\begin{aligned} y_i &\sim \text{Bernoulli}(\theta), \quad \text{for } i \in 1, 2 \ldots n, \\ \theta &\sim \text{Beta}(\alpha, \beta). \end{aligned}$$

To implement this model is Stan, we first extract out the `is_six` variable as a standalone vector, which we will name `y`, and record the length of this vector as `N`.

```
y <- dice_df %>% pull(is_six)
N <- length(y)
```

Given that we've assumed the `alpha` and `beta` hyperparameters of the Beta distribution are known, we will set them to be both equal to 1, which gives us a uniform prior distribution over θ.

```
alpha <- 1.0
beta <- 1.0
```

The four variables `y`, `N`, `alpha`, and `beta` are the data that we will send to Stan, and to do so, we must put them into a list.

```
dice_data <- list(y = y,
                  N = N,
                  alpha = alpha,
                  beta = beta)
```

The Stan implementation of this model is written in an external file, namely `loaded_dice.stan`.

```
// loaded_dice.stan
data {
  int<lower=1> N;
  int<lower=0, upper=1> y[N];
  real<lower=0> alpha;
  real<lower=0> beta;
}

parameters {
  real<lower=0, upper=1> theta;
}

model {
  theta ~ beta(alpha, beta);
  y ~ bernoulli(theta);
}
```

We notice that in this Stan program, as with most Stan programs, we have multiple code blocks, specifically `data`, `parameters`, and `model`. The `data` block defines the input data.

```
data {
  int<lower=1> N;
  int<lower=0, upper=1> y[N];
  real<lower=0> alpha;
  real<lower=0> beta;
}
```

Notice that we must not only declare the names of the variables that we will be passing in to the program as data, but we must also declare their size and type. For example, we declare than `N` is a positive integer, that `y` is a vector of `N` integers which are bounded between 0 and 1, and so `y` is a binary vector, and `alpha` and `beta` are declared as non-negative real numbers. The `parameters` block declares the (free) parameters of the model.

```
parameters {
  real<lower=0, upper=1> theta;
}
```

In this example, we have just one parameter, `theta`, which corresponds to θ in the above mathematical description. This has a real value bounded between 0 and 1. The next block is `model` and is where we define the model itself.

```
model {
  theta ~ beta(alpha, beta);
  y ~ bernoulli(theta);
}
```

This code corresponds almost perfectly to the mathematical description of the model. First, we state that `theta` is distributed as a Beta distribution with hyperparameters `alpha` and `beta`. Next, we state that each element of `y` is distributed as a Bernoulli distribution with parameter `theta`. Here, we are using vectorized notation. In other words, the statement

```
y ~ bernoulli(theta);
```

is equivalent to

```
for (i in 1:N){
  y[i] ~ bernoulli(theta);
}
```

While the second form maps on identically to the mathematical description, it is less concise notation and also less efficient.

We can execute this Stan program in R via commands provided by the `rstan` package.

```
library(rstan)
```

It should be noted, however, that Stan is a program that is completely independent of R, and can be interfaced with many other programming languages and environments such as Python, Matlab, Stata, Julia, Mathematica, Scala, and others. The following command from the `rstan` package will compile a sampler based on the specifications in `loaded_dice.stan` and the data in `dice_data`, and then draw samples from it.

```
M_dice <- stan(file = 'loaded_dice.stan',
               data = dice_data)
```

Typing the name `M_dice` gives us the following output.

```
M_dice
## Inference for Stan model: loaded_dice.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##          mean se_mean   sd    2.5%     25%     50%     75%   97.5% n_eff Rhat
## theta    0.29    0.00 0.03    0.24    0.27    0.29    0.31    0.35  1397    1
## lp__  -153.09    0.02 0.75 -155.30 -153.24 -152.80 -152.61 -152.56  1458    1
##
## Samples were drawn using NUTS(diag_e) at Sun May  2 20:06:13 2021.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

There we see output that is similar to that of a `brm` based model, which of course is another interface to Stan. The number of chains, total number of iterations, and number of warmup iterations, are all presented at the top of this output. In the summary of the samples themselves, we are given information concerning `theta`, which is our one unknown variable. From this, we have the mean, the standard deviation, and quantiles of the posterior distribution's samples. In addition, we have the `Rhat` convergence diagnostic, the effective number of samples `n_eff`. The `se_mean` is the standard deviation divided by the square root of the `n_eff`. As we can see, in addition to the information concerning `theta`, we have the same information for `lp__`. This the logarithm of the (unnormalized) posterior density evaluated at the posterior samples of `theta`. This information is not often of direct interest in itself but is used in the calculation of various model fit statistics.

If we apply the generic `summary` command to `M_dice`, we are given a list with two objects: `summary` and `c_summary`.

```
summary(M_dice) %>% class()
## [1] "list"
summary(M_dice) %>% names()
## [1] "summary"   "c_summary"
```

The `summary` object in this list is a matrix that summarizes the samples from all chains together. The `c_summary` is a multidimensional array that gives a separate summary for each chain. With the main `summary` command, we can pass in a vector of parameters using the keyword `pars` and a vector of quantiles using the keyword `probs`. For example, to get the 2.5th and 97.5th percentile values of `theta`, and obtain these summaries for all chains together, we can do the following.

```
summary(M_dice, pars = 'theta', probs = c(0.025, 0.975))$summary
##            mean      se_mean         sd      2.5%     97.5%    n_eff     Rhat
## theta 0.2929358 0.0007865121 0.02939633 0.2359161 0.3541264 1396.933 1.002714
```

For convenience, we can create a custom function `stan_summary` to return the summary matrix as a tibble.

```
stan_summary <- function(stan_model, pars, probs = c(0.025, 0.975)){
  summary(stan_model, pars = pars, probs = probs)$summary %>%
    as_tibble(rownames = 'par')
}
```

```
stan_summary(M_dice, pars = 'theta')
## # A tibble: 1 x 8
##   par    mean  se_mean     sd `2.5%` `97.5%` n_eff  Rhat
##   <chr> <dbl>    <dbl>  <dbl>  <dbl>   <dbl> <dbl> <dbl>
## 1 theta 0.293 0.000787 0.0294  0.236   0.354 1397.  1.00
```

Binomial model

Now let us consider a variant on the Bernoulli model for the loaded die. Because each of the n throws of the die and hence each six or not-six binary outcome is independent of every other and dependent only the value of θ, the mathematical model defined above is also identical to the following binomial model.

$$\begin{aligned} m &\sim \text{Binomial}(n, \theta),\\ \theta &\sim \text{Beta}(\alpha, \beta), \end{aligned}$$

where m is the total number of observations where the binary outcome was equal to six. A Stan program for this model is in `loaded_dice_binomial.stan`.

```
// loaded_dice_binomial.stan
data {
  int<lower=1> N;
  int<lower=1, upper=N> m;
  real<lower=0> alpha;
  real<lower=0> beta;
}

parameters {
  real<lower=0, upper=1> theta;
}

model {
  theta ~ beta(alpha, beta);
  m ~ binomial(N, theta);
}
```

As we can see, in the model block, we now have the line

```
m ~ binomial(N, theta);
```

rather than this line from the previous Stan program

```
y ~ bernoulli(theta);
```

As such, we no longer need to pass in `y` as data but need to pass in `m = sum(y)` instead. The value of `m` must be bounded between `1` and `N`, and hence we also include the following new line, which replaces the line declaring `y`.

```
int<lower=1, upper=N> m;
```

We then call the model as before.

```
M_dice_2 <- stan('loaded_dice_binomial.stan',
                 data = list(m = sum(y),
                             N = length(y),
                             alpha = alpha,
                             beta = beta)
)
```

The summary results are almost identical to those of the Bernoulli model.

```
stan_summary(M_dice_2, pars = 'theta')
## # A tibble: 1 x 8
##   par    mean  se_mean     sd `2.5%` `97.5%` n_eff  Rhat
##   <chr> <dbl>    <dbl>  <dbl>  <dbl>   <dbl> <dbl> <dbl>
## 1 theta 0.295 0.000741 0.0283  0.242   0.353 1457.  1.00
```

Logistic Bernoulli model

Another variant on the Bernoulli model above is the following logit model.

$$\begin{aligned} y_i &\sim \text{Bernoulli}(\theta), \quad \text{for } i \in 1, 2 \ldots n, \\ \log\left(\frac{\theta}{1-\theta}\right) &= \mu, \quad \mu \sim N(0, \sigma^2). \end{aligned}$$

Here, the prior is a normal distribution, with a zero mean and variance of σ^2, over the log odds of θ. This is simply an alternative prior over θ that is known as *logit normal* distribution. Again, we will assume that σ is known. Setting $\sigma = 1.0$ gives a relatively diffuse prior over θ, albeit one that is unimodal and centered at $\theta = 0.5$. See Figure 1 for an illustration of this distribution.

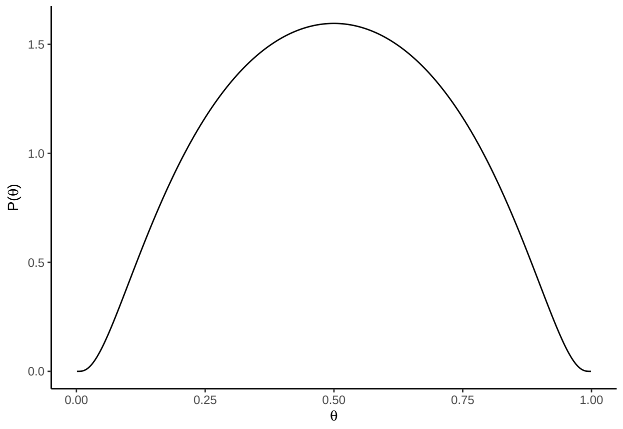

Figure 1: A logit-normal prior over θ whose hyperparameters are $\mu = 0$ and $\sigma = 1$.

The logit-normal based model is defined in the `loaded_dice_logit.stan` file.

```
// loaded_dice_logit.stan
data {
  int<lower=1> N;
  int<lower=0, upper=1> y[N];
  real<lower=0> sigma;
}

parameters {
  real mu;
}

model {
  mu ~ normal(0, sigma);
  y ~ bernoulli_logit(mu);
}
```

```
generated quantities{
  real<lower=0, upper=1> theta;
  theta = inv_logit(mu);
}
```

In this program, we use the `bernoulli_logit` probability distribution that takes the normally distributed variable `mu` as a parameter. Given that we want to view samples from `theta` instead of, or at least in addition to, those of `mu`, we include the following `generated quantities` code block.

```
generated quantities{
  real<lower=0, upper=1> theta;
  theta = inv_logit(mu);
}
```

We can compile, execute, and draw samples from this program using `rstan::stan` as usual.

```
dice_data_3 <- list(y = y, N = N, sigma = 1)
M_dice_3 <- stan('loaded_dice_logit2.stan',
                 data = dice_data_3)
```

The summary results are almost identical to those of the Bernoulli model.

```
stan_summary(M_dice_3, pars = c('mu', 'theta'))
## # A tibble: 2 x 8
##   par     mean  se_mean     sd `2.5%` `97.5%` n_eff  Rhat
##   <chr>  <dbl>    <dbl>  <dbl>  <dbl>   <dbl> <dbl> <dbl>
## 1 mu    -0.867 0.00381  0.138  -1.14   -0.595 1305.  1.00
## 2 theta  0.297 0.000791 0.0286  0.242   0.355 1309.  1.00
```

Clearly, the posterior distribution over `theta` is practically identical in this model than in the model with the Beta prior.

Normal models

Now let us consider models where the observed variable is assumed to be normally distributed. In the following data set, we have data concerning mathematical achievement scores in a sample of US university students.

```
math_df <- read_csv('data/MathPlacement.csv')
```

A histogram for this data is shown in the mathematical SAT (Scholastic Aptitude Test) (`SATM`) scores is shown in Figure 2.

```
math_df %>%
  ggplot(aes(x = SATM)) +
  geom_histogram(binwidth = 2, col='white')
```

Clearly, this data is unimodal and roughly bell-shaped but also with a negative skew. Despite is lack of symmetry, a simple and almost default model of this data would be as follows.

$$y_i \sim N(\mu, \sigma^2), \quad \text{for } i \in 1 \ldots n,$$

where y_i is the maths SAT score of student i and where there are n students in total. Obviously, we have two unknowns, μ and σ, and so in a Bayesian model, we first put priors over these two variables. As with the previous examples above, we have an almost endless variety of priors to use in this case. Given the simplicity of the model, and the fact that we have 1236 observations, excluding missing values, any prior that is not extremely precise will be dominated by the likelihood function when determining the posterior distribution, and thus most common choices are not likely to make much practical differences to the posterior distribution.

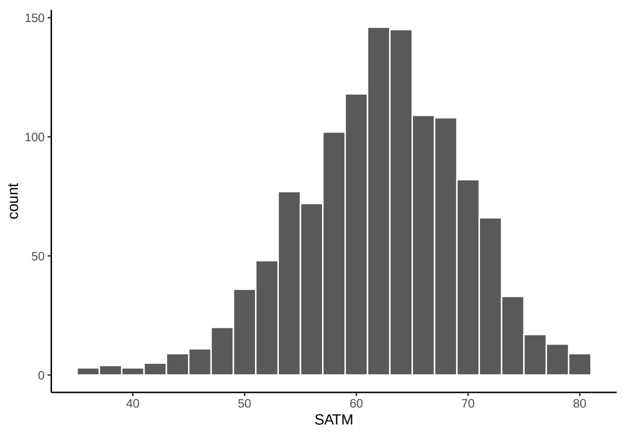

Figure 2: Histogram of mathematical SAT scores in a sample of student in a US university.

Common choices for a prior on the μ parameter of the normal distribution is another normal distribution. These can be set to have a relatively high value for the variance (hyper)-parameter to get a vague and hence weakly informative prior. For the prior over σ, Gelman and others (2006) generally recommends heavy tailed distributions over the positive real values such as a half-Cauchy or half-t distribution. Following these suggestions, our Bayesian model becomes, for example:

$$y_i \sim N(\mu, \sigma^2), \quad \text{for } i \in 1 \ldots n,$$
$$\mu \sim N(\nu, \tau^2), \quad \sigma \sim \text{Student}_+(\kappa, \phi, \omega),$$

where Student_+ is the upper half of the (nonstandard) Student t-distribution centered at ϕ, with scale parameter ω, and with degrees of freedom κ. For this choice of prior, we therefore have in total 5 hyper-parameters ν, τ, ϕ, ω and κ.

A Stan program implementing this model is in the file `normal.stan`.

```
// normal.stan
data {
  int<lower=0> N;
  real nu;
  real<lower=0> tau;
  real phi;
  real<lower=0> omega;
  int<lower=0> kappa;
  vector[N] y;
}

parameters {
  real mu;
  real<lower=0> sigma;
}

model {
  sigma ~ student_t(kappa, phi, omega);
  mu ~ normal(nu, tau);
  y ~ normal(mu, sigma);
}
```

This program is much the same as before with its three main code blocks. One important general feature of Stan not seen before is that the truncated Student t-distribution is defined by the general Student t-distribution. However, because `sigma` is defined as having only positive values, the resulting prior distribution over `sigma` is the half t-distribution. In general, regardless of the prior distribution that we use, it will be truncated based on the variable's defined limits.

We can run this program with `rstan::stan` as follows.

```
math_data <- list(y = y, N = N, nu = 50, tau = 25, phi = 0, omega = 10, kappa = 5)
M_math <- stan('normal.stan', data = math_data)
```

As we can see, we have the hyperparameters for the normal distribution on `mu` to be `nu = 50` and `tau = 25`. This places a relatively diffuse normal distribution over μ with its center at 50 and with 95% of its mass from approximately 0 to 100. The half Student-t distribution has its lower bound at 0, and with its scale being `omega = 10`, this entails that 95% of its mass extends as far as 30.

As before, we can view the summary of the results with `stan_summary`.

```
stan_summary(M_math, pars = c('mu', 'sigma'))
## # A tibble: 2 x 8
##   par    mean se_mean    sd `2.5%` `97.5%` n_eff  Rhat
##   <chr> <dbl>   <dbl> <dbl>  <dbl>   <dbl> <dbl> <dbl>
## 1 mu    62.6  0.00371 0.212  62.2    63.0  3257.  1.00
## 2 sigma  7.50 0.00259 0.151   7.22    7.81 3384.  1.00
```

Clearly, this reveals a very precise estimates of both the mean `mu` and standard deviation `sigma` of the normal distribution of maths SAT scores.

Regression models

Normal linear regression models are extensions of the normal distribution based model just described in the previous section.

Simple linear regression

As an example, using the `math_df` data, we will model how the score on a math placement exam `PlcmtScore` varies as a function of `SATM`. A scatterplot of this data is shown in Figure 3.

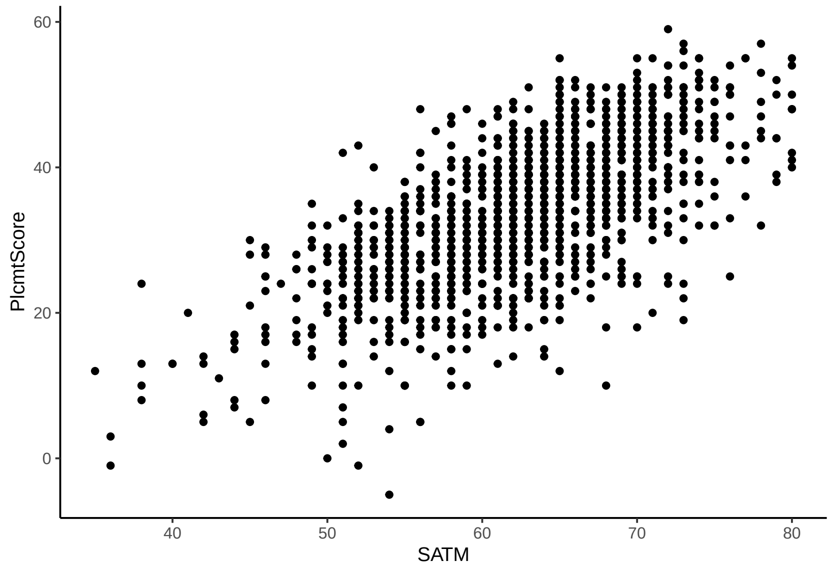

Figure 3: A scatterplot of scores on a mathematics placement exam against maths SAT scores.

Denoting the `PlcmtScore` by y and `SATM` by x, the model can be written as follows.

$$\text{for } i \in 1 \ldots n \quad y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \beta_0 + \beta_1 x_i.$$

There are now three parameters in the model: β_0, β_1, σ. We will place normal priors on β_0 and β_1, and half t-distribution on σ. As such the full Bayesian model is as follows.

$$\begin{aligned} y_i &\sim N(\mu_i, \sigma^2), \quad \mu_i = \beta_0 + \beta_1 x_i, \\ \beta_0 &\sim N(\nu_0, \tau_0^2), \quad \beta_1 \sim N(\nu_1, \tau_1^2), \quad \sigma \sim \text{Student}_+(\kappa, \phi, \omega) \end{aligned}$$

The Stan code for this model is in `normallinear.stan`.

```
// normlinear.stan
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;

  // hyperparameters
  real<lower=0> tau;
  real<lower=0> omega;
  int<lower=0> kappa;
}

parameters {
  real beta_0;
  real beta_1;
  real<lower=0> sigma;
}

model {
  // priors
  sigma ~ student_t(kappa, 0, omega);
  beta_0 ~ normal(0, tau);
  beta_1 ~ normal(0, tau);
  // data model
  y ~ normal(beta_0 + beta_1 * x, sigma);
}
```

For this example, we will choose the hyperparameters to lead to effectively uninformative priors on β_0 and β_1. Specifically, the normal distributions will be centered on zero, i.e. $\nu_0 = \nu_1 = 0$, and will be sufficiently wide, i.e., $\tau_0 = \tau_1 = 50$, so as to be effectively uniform over all practically possible values for β_0 and β_1. For the prior on σ, as above, we will use the upper half of Student's t-distribution centered at 0 and with a relatively low degrees of freedom and with a scale ω equal to the MAD of the outcome variable y. If we place the `x` and `y` data vectors and the values of the hyperparameters in the list `math_data_2`, we can call the Stan program as using `rstan::stan` as we did above.

```
math_df_2 <- math_df %>%
  select(SATM, PlcmtScore) %>%
  na.omit()

x <- pull(math_df_2, SATM)
y <- pull(math_df_2, PlcmtScore)

math_data_2 <- list(
  x = x,
  y = y,
  N = length(x),
  tau = 50, omega = mad(y), kappa = 3
)

M_math_2 <- stan('normlinear.stan', data = math_data_2)
```

As before, we can view the results with `stan_summary`

```
stan_summary(M_math_2, pars = c('beta_0', 'beta_1', 'sigma'))
## # A tibble: 3 x 8
##   par       mean  se_mean     sd  `2.5%` `97.5%` n_eff  Rhat
##   <chr>    <dbl>    <dbl>  <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 beta_0 -23.6   0.0550   1.83   -27.2   -20.0   1109.  1.00
## 2 beta_1   0.913 0.000878 0.0291   0.857   0.970 1099.  1.00
## 3 sigma    7.62  0.00383  0.154    7.32    7.92  1608.  1.00
```

Multiple regression and categorical predictors

In general, we may have any number of predictors in a regression model. As we have seen previously, if any of these predictors are categorical variables, each one is recoded using a binary dummy code. For example, a categorical variable with L distinct levels can be recoded using $L-1$ binary variables. The values of the predictors for all observations can be arranged as an $n \times K+1$ matrix X, known as the design matrix, where n is the number of observations, K is the total final number of predictors after all categorical predictors have been recoded, and the first column of X is a vector of n 1's. Therefore, in general, any normal linear model can be described as follows.

$$\vec{y} \sim N(\vec{\mu}, \sigma^2), \quad \vec{\mu} = X\vec{\beta},$$

where $\vec{y}$ is the n dimensional vector of all observations of the outcome variable, $\vec{\beta}$ is a $K+1$ dimensional vector of coefficients, and the first value of $\vec{\beta}$ is the intercept term. A full Bayesian version of this model is implemented in the program `mlreg.stan`.

```
// mlreg.stan
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K+1] X;
  vector[N] y;

  // hyper parameters
  real<lower=0> tau;
  real<lower=0> omega;
  int<lower=0> kappa;
}

parameters {
  vector[K+1] beta;
  real<lower=0> sigma;
}

transformed parameters {
  vector[N] mu;
  mu = X * beta;
}
```

```
model {
  // priors
  beta ~ normal(0.0, tau);
  sigma ~ student_t(kappa, 0, omega);

  // data model
  y ~ normal(mu, sigma);
}
```

In this program, we have a normal prior on $\vec{\beta}$ and a half Student t-distribution prior on σ as before. Note that the prior over $\vec{\beta}$ is a specified in the Stan code as follows.

```
beta ~ normal(0.0, tau);
```

This places the same $N(0, \tau^2)$ prior over each element of $\vec{\beta}$. Note also that this program has a new command block that we have not used previously: `transformed parameters`. This is used to create parameters that are transformations of the original ones. Here, we have used it for the μ vector, which is a deterministic function of β and X.

To use this Stan program, the base R command `model.matrix` and the corresponding `model_matrix` command from the `modelr` package can used to create the design matrix X. As an example, in the following data set, we have the height, weight, gender, and race, amongst other variables, for a set of over 6000 individuals.

```
weight_df <- read_csv('data/weight.csv')
weight_df
## # A tibble: 6,068 x 7
##    subjectid gender height weight handedness   age race
##        <dbl> <chr>   <dbl>  <dbl> <chr>      <dbl> <chr>
##  1     10027 male     178.   81.5 right         41 white
##  2     10032 male     170.   72.6 left          35 white
##  3     10033 male     174.   92.9 left          42 black
##  4     10092 male     166.   79.4 right         31 white
##  5     10093 male     191.   94.6 right         21 black
##  6     10115 male     172    80.2 right         39 white
##  7     10117 male     181   116.  right         32 black
##  8     10237 male     185    95.4 right         23 white
##  9     10242 male     178.   99.5 right         36 white
## 10     10244 male     181.   70.2 left          23 white
## # ... with 6,058 more rows
```

The `gender` variable has two values, male and female. The race variable has 7 values, of which `white`, `black`, `hispanic` make up around 95% of cases, and so we will limit our focus to them.

```
weight_df %<>% filter(race %in% c('black', 'white', 'hispanic'))
```

The design matrix to predict `weight` from `height`, `gender`, and `race` is

```
library(modelr)
model_matrix(weight_df, weight ~ height + gender + race)
## # A tibble: 5,769 x 5
##    `(Intercept)` height gendermale racehispanic racewhite
##            <dbl>  <dbl>      <dbl>        <dbl>     <dbl>
##  1             1   178.          1            0         1
##  2             1   170.          1            0         1
##  3             1   174.          1            0         0
##  4             1   166.          1            0         1
##  5             1   191.          1            0         0
##  6              1   172           1             0          1
##  7              1   181           1             0          0
##  8              1   185           1             0          1
##  9              1   178.          1             0          1
## 10              1   181.          1             0          1
## # ... with 5,759 more rows
```

Note that here, the `gender` variable has been coded with a single binary dummy code as follows.

female	0
male	1

On the other hand, the `race` variable has been coded by the following dummy code with two binary variables.

black	0	0
hispanic	1	0
white	0	1

The design matrix X and outcome vector $\vec{y}$ can now be obtained simply as follows.

```
X <- model_matrix(weight_df, weight ~ height + gender + race) %>%
  as.matrix()
y <- pull(weight_df, weight)
```

With this, the data list can be set up as follows.

```
weight_data <- list(
  X = X,
  y = y,
  N = length(y),
  K = ncol(X)  - 1,
  tau = 100, kappa = 3, omega = mad(y)
)
```

We may then execute the Stan program with `stan`.

```
M_weight <- stan('mlreg.stan', data = weight_data)
```

We may view the summary of the posterior samples for $\vec{\beta}$ and σ using `stan_summary`.

```
stan_summary(M_weight, pars = c('beta', 'sigma'))
## # A tibble: 6 x 8
##   par        mean  se_mean     sd  `2.5%` `97.5%` n_eff  Rhat
##   <chr>     <dbl>    <dbl>  <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 beta[1] -85.9   0.106    3.85   -93.6   -78.3   1316. 1.00
## 2 beta[2]   0.952 0.000649 0.0235   0.906   0.998 1307. 1.00
## 3 beta[3]   6.08  0.0116   0.463    5.18    6.98  1600. 1.00
## 4 beta[4]  -0.427 0.0119   0.563   -1.54    0.671 2228. 1.00
## 5 beta[5]  -2.30  0.00814  0.390   -3.06   -1.53  2300. 0.999
## 6 sigma    11.6   0.00193  0.106   11.4    11.8   3032. 1.00
```

Generalized linear models

Extending the multiple linear regression example just described to a generalized linear model is very straightforward. Just as in the case of linear models, in generalized linear models, our predictor variables, including categorical predictor variables that have been recoded using a dummy binary code, can be represented as $K + 1$ design matrix X. If our outcome variable vector $\vec{y}$ is a binary vector, a binary logistic regression model of this data would be as follows.

$$\vec{y} \sim \text{Bernoulli}(\vec{\theta}), \quad \text{logit}(\vec{\theta}) = X\vec{\beta}.$$

On the other hand, if $\vec{y}$ is a vector of counts, we could use the following Poisson regression model for this data.

$$\vec{y} \sim \text{Poisson}(\vec{\lambda}), \quad \log(\vec{\lambda}) = X\vec{\beta}.$$

Alternatively, our model for the count outcome variable could be a negative binomial regression model as follows.

$$\vec{y} \sim \text{NegBinomial}(\vec{\lambda}, \phi), \quad \log(\vec{\lambda}) = X\vec{\beta},$$

where ϕ is the inverse dispersion parameter.

Fully Bayesian versions of all of these models would be straightforward extensions of the multiple linear regression model in the previous section. For example, a Bayesian binary logistic regression with multiple predictors can be found in the `logitreg.stan`.

```
// logitreg.stan
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K+1] X;
  int<lower=0, upper=1> y[N];

  // hyper parameters
  real<lower=0> tau;
}

parameters {
  vector[K+1] beta;
}

transformed parameters {
  vector[N] mu;
  mu = X * beta;
}

model {
  // priors
  beta ~ normal(0.0, tau);

  // data model
  y ~ bernoulli_logit(mu);
}

generated quantities{
  vector[N] theta;
  theta = inv_logit(mu);
}
```

This program is obviously similar to the normal linear regression model. The principal difference comes down to the model of the outcome variables, specified in the following line.

```
y ~ bernoulli_logit(mu);
```

Although not strictly necessary, we also include the following `generated quantities` block to calculate $\vec{\theta} = \text{ilogit}(\vec{\mu})$, where $\vec{\mu} = X\vec{\beta}$.

```
generated quantities{
  vector[N] theta;
  theta = inv_logit(mu);
}
```

We can use this model with the following data set.

```
biochem_df <- read_csv('data/biochemist.csv')
```

This data gives us data from 915 PhD students. For each one, we have the number of peer reviewed articles they have published (`publications`), as well their gender (`gender`), whether they are married or not (`married`), how many children they have (`children`), a measure of the prestige of their institution (`prestige`), and the number of publications of their research mentor (`mentor`). We can also create a new variable that indicates if the PhD student obtained a publication or not.

```
biochem_df %<>% mutate(published = publications > 0)
```

This binary variable can be the outcome variable in a logistic regression analysis that analyses how the probability of being published or not varies as a function of a set of predictor variables. We will use `gender`, `married`, `prestige`, and `mentor` as predictors, and we will create a binary variable that indicates whether the number of children that the PhD student has is greater than zero or not. As above, we will create the design matrix for these predictors using `modelr::model_matrix`.

```
X <- model_matrix(~ gender + married + I(children > 0) + prestige + mentor,
                  data = biochem_df) %>%
  as.matrix()
```

Here, `gender` will be coded such that `Women` is coded as `1` and `Men` is coded as `0`. For the `married` variable, `Married` is coded by `0` and `Single` is coded by `1`. Now, we can create the necessary data for the Stan model.

```
y <- biochem_df %>% pull(published)

biochem_data <- list(y = y,
                     X = X,
                     N = nrow(X),
                     K = ncol(X) - 1,
                     tau = 100)
```

Here, with `tau = 100`, we set the standard deviation τ for the normal prior distribution over $\beta_0, \beta_1 \ldots \beta_k \ldots \beta_K$. Now, we can call the Stan model with `stan`.

```
M_biochem <- stan('logitreg.stan', data = biochem_data)
```

We may view the summary of the posterior samples for $\vec{\beta}$ using `stan_summary`.

```
stan_summary(M_biochem, pars = 'beta')
## # A tibble: 6 x 8
##   par        mean  se_mean    sd  `2.5%` `97.5%` n_eff  Rhat
##   <chr>     <dbl>    <dbl> <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 beta[1]  0.570  0.00625  0.279  0.0293  1.11   1998.  1.00
## 2 beta[2] -0.236  0.00298  0.160 -0.561   0.0734 2876.  1.00
## 3 beta[3] -0.344  0.00364  0.187 -0.711   0.0147 2641.  1.00
## 4 beta[4] -0.436  0.00364  0.187 -0.801  -0.0582 2631.  1.00
## 5 beta[5]  0.0203 0.00175  0.0807 -0.136   0.172  2119.   1.00
## 6 beta[6]  0.0819 0.000221 0.0131  0.0575  0.109  3516.   1.00
```

A Bayesian Poisson regression model can be implemented as a relatively minor extension of the logistic regression model, as we see in the following program from the file `poisreg.stan`.

```
// poisreg.stan
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K+1] X;
  int<lower=0> y[N];

  // hyper parameters
  real<lower=0> tau;
}

parameters {
  vector[K+1] beta;
}

transformed parameters {
  vector[N] mu;
  mu = X * beta;
}

model {
  // priors
  beta ~ normal(0.0, tau);

  // data model
  y ~ poisson_log(mu);
}

generated quantities{
  vector[N] lambda;
  lambda = exp(mu);
}
```

The difference between this program and that of the logistic regression program occurs in three places. First, in the following line in the `data` block, we indicate that the values of `y` are integers that are bounded by zero but have no upper limit.

```
int<lower=0> y[N];
```

Second, in the following line in the `model` block, we indicate that `y` is modelled as a Poisson distribution.

```
y ~ poisson_log(mu);
```

Note that here, the distribution is `poisson_log`. This entails that the input argument vector, denoted by `mu`, is the logarithm of the rate of the Poisson distribution. In other words, `mu` is $\vec{\mu} = \log(\lambda)$ from the mathematical description above. To obtain the rate itself, we use the following `generated quantities` block.

```
generated quantities{
  vector[N] lambda;
  lambda = exp(mu);
}
```

We will use the `X` design matrix as before, but set `y` to be `publications`, which gives the number of publications for each student. Therefore, our input data list is as follows.

```
y <- biochem_df %>% pull(publications)

biochem_data_count <- list(y = y,
                           X = X,
                           N = nrow(X),
                           K = ncol(X) - 1,
                           tau = 100)
```

We then execute the program as follows.

```
M_biochem_pois <- stan('poisreg.stan', data = biochem_data_count)
```

We may then view the summary of the posterior samples for $\vec{\beta}$ using `stan_summary` as before.

```
stan_summary(M_biochem_pois, pars = 'beta')
## # A tibble: 6 x 8
##   par        mean   se_mean      sd  `2.5%` `97.5%` n_eff  Rhat
##   <chr>     <dbl>     <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 beta[1]  0.456  0.00228   0.0967   0.268   0.642  1799.  1.00
## 2 beta[2] -0.218  0.00115   0.0565  -0.334  -0.108  2397.  1.00
## 3 beta[3] -0.152  0.00137   0.0650  -0.277  -0.0245 2243.  1.00
## 4 beta[4] -0.250  0.00135   0.0635  -0.373  -0.129  2201.  1.00
## 5 beta[5]  0.0105 0.000588  0.0263  -0.0408  0.0609 2008.  1.00
## 6 beta[6]  0.0258 0.0000264 0.00203  0.0218  0.0296 5943.  1.00
```

As a final example of a generalized linear model, let us consider a negative binomial model, which is suitable for overdispersed count data. A Stan program implementing this is in `negbinreg.stan`.

```
// negbinreg.stan
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K+1] X;
  int<lower=0> y[N];

  // hyper parameters
  real<lower=0> tau;
}

parameters {
  vector[K+1] beta;
  real<lower=0> phi;
}

transformed parameters {
  vector[N] mu;
  mu = X * beta;
}

model {
  // priors
  beta ~ normal(0.0, tau);
  phi ~ cauchy(0, 10);

  // data model
  y ~ neg_binomial_2_log(mu, phi);
}

generated quantities{
  vector[N] lambda;
  lambda = exp(mu);
}
```

The outcome variable is modelled a negative binomial with the key line.

```
y ~ neg_binomial_2_log(mu, phi);
```

Note that, as described above in the mathematical description, the mean of the negative binomial is given by $\vec{\lambda}$ where $\log(\vec{\lambda}) = X\vec{\beta}$. Thus, in the Stan code, the `mu` corresponds to $X\vec{\beta}$. The negative binomial distribution also has an additional parameter, ϕ. The higher the inverse of ϕ, the greater the overdispersion in the distribution. Here, we put a Cauchy prior with a scale of 10 on ϕ.

```
phi ~ cauchy(0, 10);
```

We will use the same `biochem_data_count` as we used in the case of the Poisson distribution and then execute the program as follows.

```
M_biochem_nb <- stan('negbinreg.stan', data = biochem_data_count)
```

We may then view the summary of the posterior samples for $\vec{\beta}$ using `stan_summary` as before.

```
stan_summary(M_biochem_nb, pars = 'beta')
## # A tibble: 6 x 8
##   par        mean   se_mean      sd  `2.5%` `97.5%` n_eff  Rhat
##   <chr>     <dbl>     <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 beta[1]  0.391  0.00287   0.130    0.143   0.646  2044.  1.00
## 2 beta[2] -0.207  0.00131   0.0729  -0.353  -0.0660 3077.  1.00
## 3 beta[3] -0.144  0.00160   0.0849  -0.316   0.0234 2799.  1.00
## 4 beta[4] -0.230  0.00174   0.0867  -0.396  -0.0622 2474.  1.00
## 5 beta[5]  0.0157 0.000747  0.0352  -0.0511  0.0849 2228.  1.00
## 6 beta[6]  0.0294 0.0000512 0.00356  0.0226  0.0365 4838.  1.00
```

Multilevel models

A multilevel linear regression model, also known as a linear mixed effects model, can be written as follows:

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \vec{x}_i \vec{\beta}_{z_i}, \quad \text{for } i \in 1 \ldots n$$

where each $z_i \in 1 \ldots J$ and each $\vec{\beta}_j \sim N(\vec{b}, \Sigma)$. In other words, as we have explained in Chapter 11, each observation i is a member of subgroup or cluster z_i, each cluster has its own set of regression coefficients, e.g. cluster j has coefficients vector $\vec{\beta}_j$, and the set of J coefficients vectors are each drawn from a multivariate normal distribution when mean vector $\vec{b}$ and covariance matrix Σ.

For simplicity here, we will consider a linear mixed effects model with one predictor variable. This can be written as follows.

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \beta_{0z_i} + \beta_{1z_i} x_i, \quad \text{for } i \in 1 \ldots n,$$

where each $z_i \in 1, 2, \ldots J$, and

$$\vec{\beta}_j = \begin{bmatrix} \beta_{0j}, \\ \beta_{1j} \end{bmatrix} \sim N(\vec{b}, \Sigma)$$

where $\vec{b} = [b_0, b_1]^\intercal$. The covariance matrix Σ can be written

$$\Sigma = \begin{bmatrix} \tau_0^2 & \tau_0\tau_1\rho \\ \tau_0\tau_1\rho & \tau_1^2 \end{bmatrix},$$

where τ_0 and τ_1 are the standard deviations of the group level intercepts and slopes, respectively, and ρ is their correlation coefficient.

In this model, we must specify priors for $\vec{b}$, Σ and the residual standard deviation σ. Clearly, there are other parameters in the model, namely $\vec{\beta}_1, \vec{\beta}_2 \ldots \vec{\beta}_J$. However, the prior for these is determined by the values of $\vec{b}$ and Σ. A commonly used, even default, prior family for $\vec{b}$ is the normal distribution. If this is centered at zero and has a relatively wide variance, then this is effectively an uninformative prior. For Σ, on the other hand, we have more choices. One generally useful prior for Σ is the LJK (named after Lewandowski, Kurowicka, and Joe (2009)) prior on its corresponding correlation matrix, and a separate prior on the variance terms. In this example, however, because there is only one correlation coefficient term in the matrix, namely ρ, we will put prior on that and then separate priors on τ_0 and τ_0. Finally, we will put a similar prior on σ.

A Stan program for this model is in the file `lmm.stan`.

```
// lmm.stan
data {
  int<lower=1> N; // no. observations
  int<lower=1> J; // no. groups

  vector[N] y;    // outcome
  vector[N] x;    // predictor
  int<lower=0, upper=J> z[N]; // group index
}

parameters {
  vector[2] b;
  vector[2] beta[J];
  real<lower=-1, upper=1> rho;
  vector<lower=0>[2] tau;
  real<lower=0> sigma;
}

transformed parameters {
  cov_matrix[2] Sigma;
  corr_matrix[2] Omega;
  Omega[1, 1] = 1;
  Omega[1, 2] = rho;
  Omega[2, 1] = rho;
  Omega[2, 2] = 1;
  Sigma = quad_form_diag(Omega, tau);
}

model {

  rho ~ uniform(-1, 1);
  tau ~ cauchy(0, 10);
  sigma ~ cauchy(0, 10);
  b ~ normal(0, 100);

  beta ~ multi_normal(b, Sigma);

  for (i in 1:N)
    y[i] ~ normal(beta[z[i], 1] + x[i] * beta[z[i], 2], sigma);

}
```

In the `model` block, we specify Cauchy priors on `sigma` and `tau`, where the latter corresponds to the vector $[\tau_0, \tau_1]^\intercal$, a uniform prior on ρ, and very diffuse normal distribution prior on the `b` vector. We also see that `beta`, which corresponds to the set of J vectors $\beta_1, \beta_2 \ldots \beta_J$, is distributed as a multivariate normal distribution. Each individual observation, for $i \in 1 \ldots n$, the model is $y_i \sim N(\mu_i, \sigma^2)$, $\mu_i = \beta_{0z_i} + \beta_{1z_i} x_i$. In the Stan program, this is implemented in the following lines.

```
for (i in 1:N)
  y[i] ~ normal(beta[z[i], 1] + x[i] * beta[z[i], 2], sigma);
```

This model can be tested using the `sleepstudy` data set from `lme4`, which we explored in the previous chapter.

```
sleepstudy <- lme4::sleepstudy

y <- sleepstudy$Reaction
x <- sleepstudy$Days
z <- sleepstudy$Subject %>% as.numeric()

sleep_data <- list(N = length(y),
                   J = length(unique(z)),
                   y = y,
                   x = x,
                   z = z)

M_lmm <- stan('lmm.stan', data = sleep_data)
```

The summary of the main paramters of this model is as follows, which are comparable to the results obtained from the non-Bayesian `lmer` analysis of the same model.

```
stan_summary(M_lmm, pars = c('b', 'tau', 'rho', 'sigma'))
## # A tibble: 6 x 8
##   par       mean se_mean    sd  `2.5%` `97.5%` n_eff  Rhat
##   <chr>    <dbl>   <dbl> <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 b[1]    250.    0.108   6.91  237.    264.   4110. 1.00
## 2 b[2]     10.5   0.0250  1.64    7.15   13.6  4311. 0.999
## 3 tau[1]   24.4   0.141   6.32   13.8    38.2  2011. 1.00
## 4 tau[2]    6.31  0.0273  1.41    4.05    9.52 2686. 1.00
## 5 rho       0.129 0.00803 0.291  -0.419   0.698 1310. 1.01
## 6 sigma    25.9   0.0285  1.55   23.1    29.2  2939. 1.00
```

Posterior expectations

As mentioned in the introduction to this chapter, quantities of interest from a Bayesian model can be expressed as *posterior expectations* that can be approximated using Monte Carlo integration:

$$\langle g(\theta) \rangle = \int g(\theta) \mathrm{P}(\theta|\mathcal{D}) d\theta \approx \langle g(\theta) \rangle \approx \frac{1}{n} \sum_{i=1}^{n} g(\tilde{\theta}_i),$$

where $\tilde{\theta}_1, \tilde{\theta}_2 \ldots \tilde{\theta}_n$ are posterior samples of the unknown variables in the model. In general, any quantity of interest from a Bayesian model can be expressed in this way. For this reason, when we have the posterior samples, any question of interest concerning the model may be addressed.

We can obtain each sample from each chain for any variables using `rstan::extract` as follows.

```
rstan::extract(M_dice, pars='theta', permuted=F, inc_warmup=T) %>%
  magrittr::extract(,,1) %>%
  as_tibble()
## # A tibble: 2,000 x 4
##    `chain:1` `chain:2` `chain:3` `chain:4`
##        <dbl>     <dbl>     <dbl>     <dbl>
##  1     0.345     0.211    0.0784     0.233
##  2     0.345     0.211    0.0784     0.233
##  3     0.345     0.211    0.0784     0.233
##  4     0.345     0.332    0.367      0.307
##  5     0.342     0.332    0.341      0.307
##  6     0.346     0.325    0.253      0.293
##  7     0.276     0.332    0.277      0.311
##  8     0.328     0.291    0.324      0.324
##  9     0.295     0.291    0.288      0.312
## 10     0.262     0.297    0.279      0.312
## # ... with 1,990 more rows
```

(In this command, we use the `extract` function from both `rstan` and `magrittr` and so we use their namespaces to distinguish between them). With `rstan::extract`, by using `permute = F` we obtain an array for each parameter that we specify in `pars`, and get all samples including the warmup samples by `inc_warmup`. This function returns a multi-dimensional array whose first dimension indexes the samples, the second indexes the chains, and the third indexes the parameters.

The package `tidybayes` provides many useful functions from working with Stan based models, including functions from extracting samples. For example, using the `M_math_2` regression model described above, the following extracts the (post-warmup) samples into a data frame with one row for each sample from each chain.

```
library(tidybayes)
spread_draws(M_math_2, beta_0, beta_1, sigma)
## # A tibble: 4,000 x 6
##    .chain .iteration .draw beta_0 beta_1 sigma
##     <int>      <int> <int>  <dbl>  <dbl> <dbl>
##  1      1          1     1  -23.1  0.907  7.69
##  2      1          2     2  -22.1  0.889  7.65
##  3      1          3     3  -21.9  0.885  7.55
##  4      1          4     4  -21.3  0.876  7.56
##  5      1          5     5  -21.6  0.882  7.57
##  6      1          6     6  -22.3  0.893  7.52
##  7      1          7     7  -24.6  0.928  7.50
##  8      1          8     8  -24.2  0.921  7.72
##  9      1          9     9  -22.2  0.893  7.63
## 10      1         10    10  -22.2  0.890  7.62
## # ... with 3,990 more rows
```

We these samples in this format, we may now easily compute quantities of interest. For example, let us imagine we are interested in knowing the probability that someone could score greater than 50 on `PlcmtScore` given that their `SATM` score was exactly 75. If we knew the true values of β_0, β_1, and σ, we would calculate this as follows.

$$\mathrm{P}(y > 50|x = 75, \beta_0, \beta_1, \sigma) = \int_{50}^{\infty} N(y|\mu = \beta_0 + \beta_1 \times 75, \sigma)dy.$$

Integrating over the posterior distribution of β_0, β_1, and σ is as follows.

$$\mathrm{P}(y > 50|x = 75) = \int \mathrm{P}(y > 50|x = 75, \beta_0, \beta_1, \sigma)\mathrm{P}(\beta_0, \beta_1, \sigma|\mathcal{D})d\beta_0, d\beta_1, d\sigma.$$

Using Monte Carlo integration, this integral is approximated as follows.

$$\mathrm{P}(y > 50|x = 75) \approx \frac{1}{S}\sum_{i=1}^{S} \mathrm{P}(y > 50|x = 75, \tilde{\beta_0}_s, \tilde{\beta_1}_s, \tilde{\sigma}_s),$$

where $\tilde{\beta_0}_s$, $\tilde{\beta_1}_s$, $\tilde{\sigma}_s$, for $s \in 1 \ldots S$, are S samples from the posterior distribution. This calculation can be easily performed using R. First, we write a function to calculate $\mathrm{P}(y > 50|x = 75, \beta_0, \beta_1, \sigma)$.

```
f <- function(beta_0, beta_1, sigma){
  pnorm(50,
        mean = beta_0 + beta_1 * 75,
        sd = sigma,
        lower.tail = F
  )
}
```

We then compute this function for each sample from the posterior and calculate the average.

```
spread_draws(M_math_2, beta_0, beta_1, sigma) %>%
  mutate(p = f(beta_0, beta_1, sigma)) %>%
  summarise(prob = mean(p))
## # A tibble: 1 x 1
##    prob
##   <dbl>
## 1 0.251
```

References

Gelman, Andrew, and others. 2006. "Prior Distributions for Variance Parameters in Hierarchical Models (Comment on Article by Browne and Draper)." *Bayesian Analysis* 1 (3): 515–34.

Lewandowski, Daniel, Dorota Kurowicka, and Harry Joe. 2009. "Generating Random Correlation Matrices Based on Vines and Extended Onion Method." *Journal of Multivariate Analysis* 100 (9): 1989–2001.