An introduction to Bayesian inference

Nate Pope

28 October 2016

Preliminaries

Install the package `rstan` by following the directions here. Note that `rstan` can take a little bit of time to compile, and on OSX and Windows operating systems you might need to install the developer toolchain. The packages that will be used in this tutorial are:

```
library(rstan)
library(shinystan)
library(ggplot2)
library(dplyr)
```

Posterior, prior, and likelihood

Maximum likelihood

Inference by maximum likelihood works by finding the values of the parameters θ that maximize the *likelihood* of a model. The likelihood is just the probability of the data conditional on the parameters of the model, denoted $p(y|\theta)$. For example, a very simple model is that the data y are normally distributed with an unknown mean μ and an unknown standard deviation σ.

For the sake of illustration, here's a very small simulated dataset consisting of three values:

```
# simulate some data
set.seed(101)
fake_data <- rnorm(n = 3, mean = 2, sd = 2)
fake_data # look at it
```

```
## [1] 1.3479270 3.1049237 0.6501123
```

A function to calculate the likelihood for this simple model might look like this:

```
# a function that returns the likelihood for given values of mu and sigma
likelihood <- function(mu, sigma, data_values){
  prod( dnorm(data_values, mean = mu, sd = sigma) )
}

# for example ...
likelihood(mu = 0.5, sigma = 0.3, data_values = fake_data)
```

```
## [1] 1.622847e-18
```

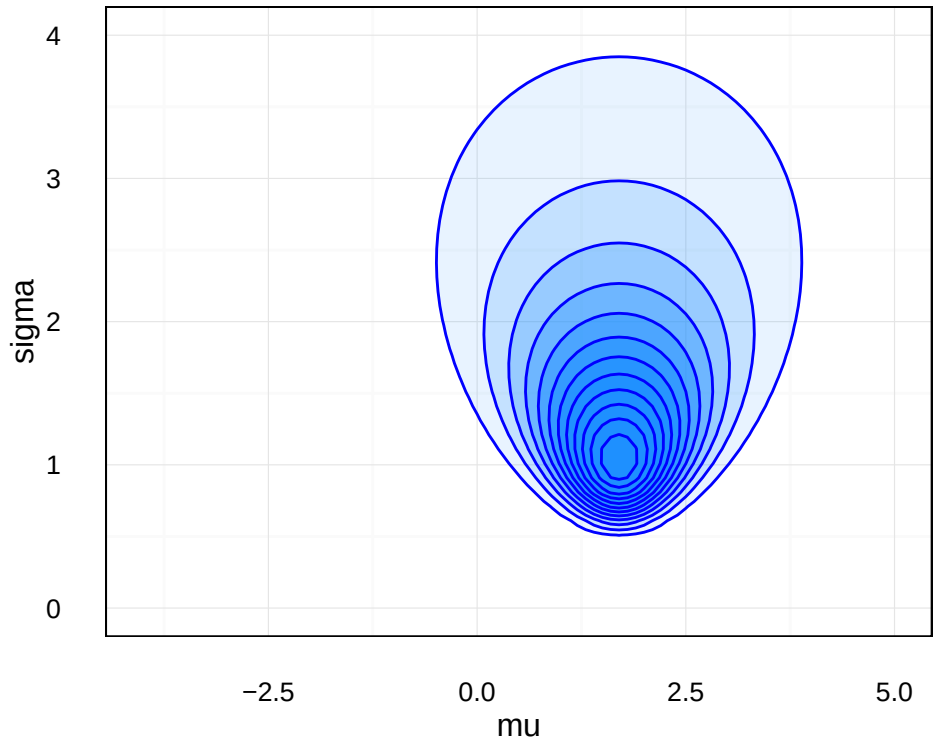
```
likelihood(mu = 0.1, sigma = 0.3, data_values = fake_data)
```

```
## [1] 1.252356e-26
```

```
likelihood(mu = -0.1, sigma = 0.3, data_values = fake_data)
```

```
## [1] 1.488895e-31
```

```
# plot across a grid of values of mu and sigma ...
expand.grid(mu = seq(-4, 5, 0.1), sigma = seq(0.01, 4, 0.1)) %>%
  rowwise() %>%
  mutate(likelihood = likelihood(mu, sigma, fake_data)) %>%
  ggplot(aes(x=mu, y=sigma)) +
  stat_contour(geom="polygon", aes(z = likelihood, alpha = ..level..),
               color="blue", fill="dodgerblue") +
  xlim(-4,5) + ylim(0, 4) + theme_minimal() +
  theme(legend.position = "none", panel.border=element_rect(fill=NA))
```



Note: In practice we'd work with the log-likelihood to prevent numeric problems, but to keep things simple I'll just use the untransformed likelihood.

Bayesian inference

In contrast to maximum likelihood, the goal of Bayesian inference is to estimate the probability of the parameters conditional on the data. This is known as the *posterior distribution* and denoted $p(\theta|y)$. The posterior distribution and the likelihood are related by the following equation:

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)}$$

This formula is called Bayes' rule. Bayes' rule is an essential relation in probability theory, and is not specific to Bayesian statistics (where it plays a crucial role).

What is the intuition behind Bayes rule? Basically, it says that the probability of the parameters given the observed data is *proportional* to: the probability of the data given the parameters ($p(y|\theta)$) weighted by the probability of the parameters regardless of any data ($p(\theta)$). This last term is called the *prior*, and is usually interpreted as our beliefs about feasible values for the parameters.

It's hard not to make the specification of the prior somewhat arbitrary, and this is the most common criticism of Bayesian inference. For the toy 'normal sample' model I described above, I'll use priors which emphasise how the prior and likelihood combine to form the posterior. I'll use a normal prior for the mean (with location -1 and scale 1), and a half-normal prior for the std deviation (with scale 1).

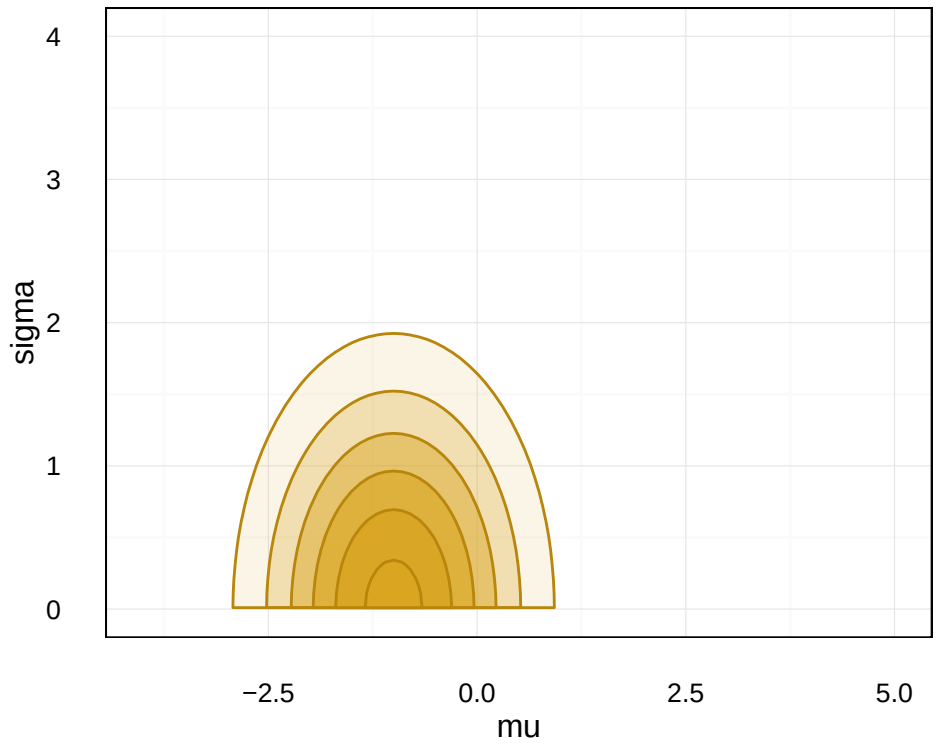
```
# a function that returns the prior for given values of mu and sigma
prior <- function(mu, sigma){
  # normal prior on mu, half-normal prior on sigma
  dnorm(mu, mean = -1, sd = 1) *
    2*dnorm(abs(sigma), mean = 0, sd = 1)
}

# for example ...
prior(mu = 0.5, sigma = 0.3)
```

```
## [1] 0.09879287
```

```
prior(mu = 1, sigma = 1)
```

```
## [1] 0.02612847
```

```
# across a grid of values of mu and sigma ...
expand.grid(mu = seq(-4, 5, 0.1), sigma = seq(0.01, 4, 0.1)) %>%
  rowwise() %>%
  mutate(likelihood = prior(mu, sigma)) %>%
  ggplot(aes(x=mu, y=sigma)) +
  stat_contour(geom="polygon", aes(z = likelihood, alpha = ..level..),
               color="darkgoldenrod", fill="goldenrod") +
  xlim(-4,5) + ylim(0, 4) + theme_minimal() +
  theme(legend.position = "none", panel.border=element_rect(fill=NA))
```



Notice that only the numerator of Bayes' rule is a function of the parameters. This means that the numerator – also called the "target" or "unnormalized posterior" – has exactly the same *shape* as the posterior distribution.

Example 1.

Here is an example of forming the target – the unnormalized posterior – to visualize the posterior distribution of the mean and variance of a sample (assuming normality). Specifically, the model is: $y \sim \mathcal{N}(\mu, \sigma^2)$. In English, the observed data y are normally distributed with unknown mean μ and variance σ^2.

The parameters are μ and σ, and these both need priors, which I defined above.

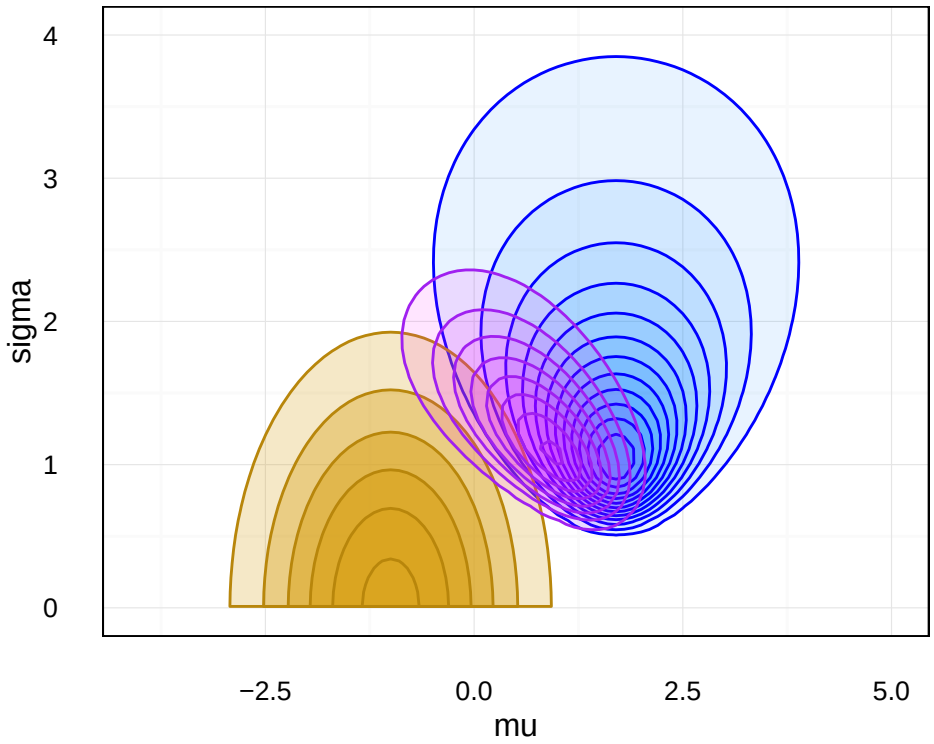
```
# a function that computes the unnormalized posterior
# given the functions likelihood(), prior() defined above
posterior <- function(mu, sigma, data_values){
  likelihood(mu, sigma, data_values) * prior(mu, sigma)
}

# for example ...
posterior(0, 0.5, fake_data)
```

```
## [1] 4.15249e-12
```

```
posterior(0.5, 0.5, fake_data)
```

```
## [1] 1.342178e-08
```

```r
# across a grid of values ...
# visualize posterior (purple), prior (yellow), likelihood (blue) together
expand.grid(mu = seq(-4, 5, 0.1), sigma = seq(0.01, 4, 0.1)) %>%
  rowwise() %>%
  mutate(likelihood = likelihood(mu, sigma, fake_data),
         prior = prior(mu, sigma),
         posterior = posterior(mu, sigma, fake_data)) %>%
  ggplot(aes(x=mu, y=sigma)) +
  stat_contour(geom="polygon", aes(z = likelihood, alpha = ..level..),
               color="blue", fill="dodgerblue") +
  stat_contour(geom="polygon", aes(z = prior, alpha = ..level..),
              color="darkgoldenrod", fill="goldenrod") +
  stat_contour(geom="polygon", aes(z = posterior, alpha = ..level..),
              color="purple", fill="magenta") +
  xlim(-4,5) + ylim(0, 4) + theme_minimal() +
  theme(legend.position = "none", panel.border=element_rect(fill=NA))
```



Exercise 1:

The prior on μ can be interpreted as follows: the mean of the prior is our best guess as to the value of μ, while standard deviation of the prior represents the amount of uncertainty we have regarding this guess. In this case our prior guess about μ is -1, far from the true value of 2, and the disagreement between the prior and likelihood will result in a posterior that is a compromise between the two. Copy the functions above and modify them to answer the following questions:

1. What happens to the posterior if you decrease the standard deviation of the prior on μ to 0.1? Why?
2. What happens to the posterior if you change the mean of the prior on μ to 2? Why?

3. What happens to the posterior if you simulate new data, but increase the number of data points to 10? Why?

```
# your code goes here
```

Denominator of Bayes' rule

The denominator in Bayes' rule is harder to understand. This is termed the *marginal likelihood* and is defined as the probability of the data, averaged (integrated) across all possible values of the parameters. Notice that the marginal likelihood is *not* a function of the parameters. What is the role of the marginal likelihood?

By definition, probability distributions must integrate (sum) to one. This is equivalent to saying that the probabilities of all possible parameter values sum to 1. For the posterior distribution to be a probability distribution, it must to satisfy this criterion. This is the role of the marginal likelihood: if you add up (integrate) the numerator of Bayes' rule across all possible values of the parameters, the result is the marginal likelihood.

When there's more than a few parameters, the marginal likelihood can be very hard to calculate. For this reason, most Bayesian analyses proceed by forming the numerator in Bayes' rule – sometimes called the "unnormalized posterior" or "target" – and then use simulation to numerically approximate the posterior distribution (see the last section of this document for more information).

In summary

1. The goal of Bayesian inference is to calculate the posterior. This is accomplished with Bayes' rule.
2. The shape of the posterior is determined by the numerator of Bayes' rule: the product of likelihood and prior.
3. The role of the denominator of Bayes' rule (the marginal likelihood) is to make the posterior integrate (sum) to 1.
4. The posterior is a compromise between likelihood and prior. As the amount of data increases, the likelihood dominates the prior.

Stan

How is Bayesian inference accomplished in practice, if the marginal likelihood (and thus posterior) cannot be computed? Essentially, if we can simulate many samples from the posterior distribution, we can approximate the posterior distribution to an arbitrary accuracy. By far, the most common method of simulation involves what is known as Markov chain Monte Carlo (see the last section of this document for a worked example).

As far as software that implements simulation for Bayesian analysis, there's a spectrum where flexibility is traded against ease-of-use. In a nutshell:

- *Specific models*: easy to use, efficiency high, flexibility low (ie. MCMCglmm, BayesLogit)
- *Modelling language*: moderate difficulty, efficiency usually high, flexibility high (ie. Stan, JAGS/BUGS, Nimble)
- *Do-it-yourself*: hard to do, efficiency depends, flexibility limitless

Here I'll focus on the middle category – 'modelling language' – as I think this provides the best introduction to Bayesian inference.

But first, some history

In the late 1990's, a program called BUGS made Bayesian inference vastly more accessible by creating a computational platform that would automatically carry out the hard part of Bayesian inference – approximating the posterior distribution by Markov chain Monte Carlo. All that one had to do was specify the model, in a language that mixed a pseudo-mathematical notation with programming constructs like loops and if/else statements.

BUGS is no longer under development, and the computational methods that it employs are pretty inefficient by today's standards. A number of other 'general purpose' Bayesian modelling platforms are spiritual successors to BUGS. The one I'm teaching today is called **Stan**. It currently seems to have the most diverse user base, the largest and most active development team, and the best documentation (the manual is incredible, at 500+ pages, with many example models on github). It's also exceptionally efficient and seems to have new features popping out by the month. *Note: there is a very important limitation to Stan: it can only handle continuous parameters. However, it will handle any type of data (observed values).*

Stan syntax

Stan interfaces with R via a package called `rstan`. The workflow is as follows:

1. The model is written out in Stan's language (which is kinda like R).
2. The data is put into a list.
3. Both data and model are fed to an R function which simulates from the posterior
4. Use a Shiny app (shinystan) to look at diagnostics, summaries, etc.

The model code for Stan requires some explanation. It's a mish-mash of R and C++ syntax. Here's a summary of some important similarities/dissimilarities with R:

- Lines must end with a semi-colon `;` (C++)
- All variables/parameters must have defined types/dimensions (C++)
- Indexing starts at 1 (R)
- The "for-loop" syntax is `for(iterator in range) { do stuff }` (R)
- Supports "incrementing operators": for example, the operator `left_side += right_side` adds `right_side` to `left_side` (C++)
- Lines are commented out by `//` (C++)

The structure of a coded Stan model is quite strict. There are multiple blocks, in each of which an important piece of the model is defined. The blocks are structured like:

```
block_name {
  // important stuff goes here
}
```

The three essential blocks are `data`, `parameters`, and `model`. I'll describe each of these in turn.

The `data` block

The purpose of the `data` block is the define the types, dimensions, and names of inputs into the model. Inputs usually consist of observed data (i.e. response and covariates) and indices which map each data point onto groups. The general syntax for a type declaration is:

```
type_name<lower=number, upper=number>[type dimension] variable_name[dimension];
```

That's a lot, so let's unpack it with some examples. The only pieces that are **always** required are what I called `type_name` and `variable_name`. For example, if I want to input an integer called `N`, I would use the following type declaration:

```
int N;
```

If I wanted to have 10 integers stored in N, I can do this by providing what I called `dimension`:

```
int N[10];
```

Sometimes I might want a lower or upper bound on a variable. For example, if `N` is a count of expressed protein reads, I might want to set the lower bound to be 0. I can do this by providing what I called `<lower=number>`:

```
int<lower=0> N[10];
```

Some types–for example vectors–have a dimensionality of their own. For example, if I create a `vector` called `Y`, I would need to specify the size of the vector using what I called `type_dimension`:

```
vector[10] Y;
```

If I wanted Y to consist of 5 vectors, each of length 10, I would do:

```
vector[10] Y[5];
```

Everything that goes in the `data` block **must** be supplied by the user–no unobserved values go in this block.

The `parameters` block

The parameters block is very similar to the data block in that it consists of type declarations, but holds only unobserved quantities – i.e. quantities we would like to estimate. For example, if the parameters of my model are a continuous mean and standard deviation (like in the "normal sample" example of the first section), I could do something like:

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

The `model` block

This is where the action happens. The point of this block is to **tell Stan how to put together the target (the log unnormalized posterior distribution)**. Declarative code goes in this block, and it looks much like a typical program. There are a few different ways to tell Stan how you want to put together the target, but the clearest uses the following syntax:

```
target += name_of_distribution_function( random variable | parameters );
```

This code reads as: evaluate the log probability distribution of a variable conditional on some parameters, and add it to a running sum of the target. The keyword `target` is reserved (you can't use `target` as the name of a variable).

That might seem like gibberish, so let's unpack it with some examples. The code:

```
target += normal_lpdf(mu|0, 2);
target += inv_gamma_lpdf(sigma|0, 2);
target += normal_lpdf(Y|mu, sigma);
```

First takes a parmeter called `mu`, calculates the log probability density of `mu` under a normal distribution with mean 0 and variance 2, and adds the resulting quantity to a running tally of the log unnormalized posterior distribution. Second, takes a parameter called `sigma`, calculates the log probability density of `sigma` under a inverse gamma distribution with shape 0.1 and rate 0.1, and adds the resulting quantity to a running tally of the unnormalized posterior. Third, takes a vector of observe data called `Y`, evaluates the log probability density of `Y` under a normal distribution with mean `mu` and standard deviation `sigma`, and adds the resulting quantity to a running tally of the unnormalized posterior. How do I know that these are observed/unobserved or data/parameters? Because I declared them in the `data` and `parameter` blocks!

Some examples of log probability distribution functions in Stan (see the manual for more): `normal_lpdf(variable | mean, sd)` is the normal distribution with parameters for mean and standard deviation; `inv_gamma_lpdf(variable|shape, rate)` is the inverse gamma distribution with shape and rate parameters; `cauchy_lpdf(variable|location, scale)` is the Cauchy distribution with location and scale parameters; `poisson_lpmf(variable|mean)` is the Poisson distribution with a mean parameter.

Example 2:

A good introductory example to Stan is a simple linear regression model. Let y_i, x_i be the dependent and independent variables for data point i. The model is: y_i are normally distributed with mean $\alpha + \beta x_i$ for the ith datapoint, and standard deviation σ common to all datapoints. The parameters α, β are the intercept and slope of the regression line. A mathematical shorthand for this is:

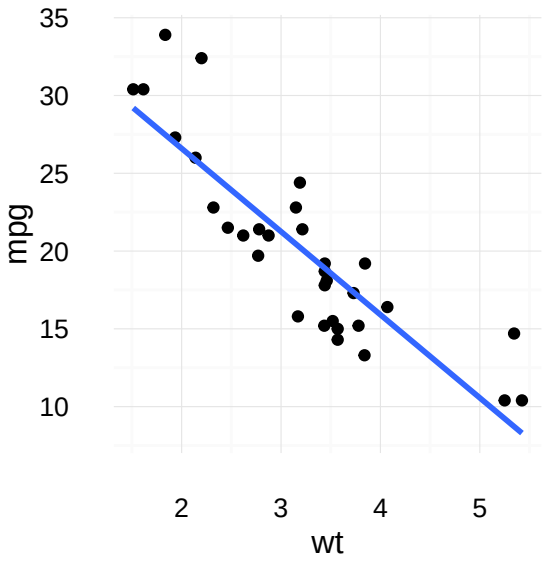
$$y_i \sim \text{Normal}(\alpha + \beta x_i, \sigma)$$

The parameters to estimate are α, β, and σ. We'll use a built-in dataset, `mtcars`, to regression miles per gallon against weight of different car models.

```
# we'll use the mtcars data
data(mtcars)
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
# regression of miles per gallon against weight
mtcars %>% ggplot(aes(x=wt,y=mpg)) +
  geom_point() + geom_smooth(method="lm", se=F) +
  theme_minimal()
```



Now, we construct a model in Stan, with the blocks that I described previously.

First, we need to define the **type**, **size**, and **name** of each kind of data we are putting in to the model. *Note that this includes indices (ie. the number of data points)!*

```
data_block <- "
data {
  int N; // the number of data points
  vector[N] Y; // the response variable is an N-dimensional vector
  vector[N] X; // the covariate is also an N-dimensional vector
}
"
```

Second, we need to define the type, size, and name of each unknown parameter, as well as contraints on their range.

```
param_block <- "
parameters {
  real alpha; // the intercept
  real beta; // the slope
  real<lower=0> sigma; // the standard deviation
}
"
```

Finally, we need to specify how to put together the 'target' – the log unnormalized posterior – from the prior and the likelihood.

```
model_block <- "
model {
  // priors
  target += cauchy_lpdf(sigma | 0, 5); // Cauchy prior distribution on std. deviation
  target += normal_lpdf(alpha | 0, 10); // Normal(0,10) prior on intercept
  target += normal_lpdf(beta | 0, 10); // Normal(0,10) prior on slope

  // likelihood
```

```
  target += normal_lpdf(Y | alpha + X*beta, sigma); // Y ~ Normal(alpha+X*beta, sigma^2)
}
"
```

Now we assemble the blocks into a single string, and also assemble the 'inputs' to the model as a named list. The names must correspond to those in the 'data' block.

```
# piece together model blocks into single string
regression_model <- paste(data_block, param_block, model_block)
cat(regression_model) # take a look at it
```

```
##
## data {
##  int N; // the number of data points
##  vector[N] Y; // the response variable is an N-dimensional vector
##  vector[N] X; // the covariate is also an N-dimensional vector
## }
##
## parameters {
##  real alpha; // the intercept
##  real beta; // the slope
##  real<lower=0>   sigma; // the standard deviation
## }
##
## model {
##  // priors
##  target += cauchy_lpdf(sigma | 0, 5); // Cauchy prior distribution on std. deviation
##  target += normal_lpdf(alpha | 0, 10); // Normal(0,10) prior on intercept
##  target += normal_lpdf(beta | 0, 10); // Normal(0,10) prior on slope
##
##  // likelihood
##  target += normal_lpdf(Y | alpha + X*beta, sigma); // Y ~ Normal(alpha+X*beta, sigma^2)
## }
```

```
# piece together inputs (ie. data) into a list
regression_inputs <- list(N=nrow(mtcars), Y=mtcars$mpg, X=mtcars$wt)
```

It's time to run the model. Load the package `rstan`, and use the function `stan()` on the model and inputs. You can specify the number of iterations with argument `iter`, the thinning interval with argument `thin`, the number of Markov chains with argument `chains`, and the amount of "burn-in" (the number of initial iterations to discard) with argument `warmup`; and so on and so forth. The total number of samples that you'll simulate from the posterior is `chains * (iter - warmup) / thin`.

```
library(rstan)
# fit model with stan
regression_fit <- stan(model_name = "regression tutorial",
                       model_code = regression_model,
                       data = regression_inputs,
                       chains = 3, iter = 1000,
                       thin = 1, warmup = 300)
```

```
##
```

```
## SAMPLING FOR MODEL 'regression tutorial' NOW (CHAIN 1).
##
## Chain 1, Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1, Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1, Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1, Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1, Iteration: 301 / 1000 [ 30%]  (Sampling)
## Chain 1, Iteration: 400 / 1000 [ 40%]  (Sampling)
## Chain 1, Iteration: 500 / 1000 [ 50%]  (Sampling)
## Chain 1, Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1, Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1, Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1, Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1, Iteration: 1000 / 1000 [100%]  (Sampling)
##  Elapsed Time: 0.01487 seconds (Warm-up)
##                0.022856 seconds (Sampling)
##                0.037726 seconds (Total)
##
##
## SAMPLING FOR MODEL 'regression tutorial' NOW (CHAIN 2).
##
## Chain 2, Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2, Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2, Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2, Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2, Iteration: 301 / 1000 [ 30%]  (Sampling)
## Chain 2, Iteration: 400 / 1000 [ 40%]  (Sampling)
## Chain 2, Iteration: 500 / 1000 [ 50%]  (Sampling)
## Chain 2, Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2, Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2, Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 2, Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2, Iteration: 1000 / 1000 [100%]  (Sampling)
##  Elapsed Time: 0.014456 seconds (Warm-up)
##                0.030427 seconds (Sampling)
##                0.044883 seconds (Total)
##
##
## SAMPLING FOR MODEL 'regression tutorial' NOW (CHAIN 3).
##
## Chain 3, Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3, Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3, Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3, Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3, Iteration: 301 / 1000 [ 30%]  (Sampling)
## Chain 3, Iteration: 400 / 1000 [ 40%]  (Sampling)
## Chain 3, Iteration: 500 / 1000 [ 50%]  (Sampling)
## Chain 3, Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3, Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3, Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3, Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3, Iteration: 1000 / 1000 [100%]  (Sampling)
##  Elapsed Time: 0.013112 seconds (Warm-up)
##                0.028856 seconds (Sampling)
```

```
##                0.041968 seconds (Total)
```

Unlike many other programs for Bayesian computation, Stan is quite efficient: it converges very quickly and doesn't require much thinning. We'll go over how to assess if these settings are sufficient below.

You can get a summary of the posterior by typing the name of the fitted model object. Note that these estimates are approximated by simulation, and so will change a little bit every time you run the model.

```
regression_fit
```

```
## Inference for Stan model: regression tutorial.
## 3 chains, each with iter=1000; warmup=300; thin=1;
## post-warmup draws per chain=700, total post-warmup draws=2100.
##
##         mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
## alpha  35.85    0.07 1.88  32.11  34.62  35.97  37.19  39.20   772    1
## beta   -4.93    0.02 0.56  -5.95  -5.32  -4.98  -4.56  -3.79   787    1
## sigma   3.17    0.01 0.42   2.49   2.87   3.11   3.40   4.13   908    1
## lp__  -96.73    0.04 1.19 -99.76 -97.24 -96.41 -95.88 -95.39   757    1
##
## Samples were drawn using NUTS(diag_e) at Fri Oct 28 11:52:25 2016.
## 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).
```

The `Rhat` values in this summary are an indicator of successful convergence: you want to see `Rhat` which are very close to 1. There's also a convenient (if overwhelming) way to look at the posterior summary (and various diagonostics) using the package `shinystan`:

```
## (will open web browser)
launch_shinystan(regression_fit)
```

Exercise 2.

Modify the regression model to include the covariate `drat` in addition to `wt`. You'll need to change all three blocks to include: a new covariate, a new parameter, a new prior and likelihood. How might you write a model that could include an arbitrary number of covariates?

```
# your code goes here
```

Exercise 3.

Modify the regression model to use the non-negative, integer-valued variable `hp` as a response variable instead of `mpg`. Use a Poisson distribution for the likelihood. The Poisson has a single parameter: the mean, which is constrained to be positive. To respect this constraint, we exponentiate the regression equation $\alpha + x_i\beta$ (this type of transformation – to respect contraints – is often called a "link function" in the statistical literature). The model is

$$y_i \sim \text{Poisson}(\exp(\alpha + x_i\beta))$$

Note that the function to implement the Poisson distribution in Stan is `poisson_lpmf( variable | mean )`.

```
# your code goes here
```

Exercise 4.

An optionial block is called `generated quantities {}`. If you wanted to calculate functions of the parameters (and save the results across MCMC samples), you would do so inside this block. An example of such a function is the fitted values given by the equation `alpha + beta*X`. Add a `generated quantities` block to the regression model, and store the fitted values in a vector called `fitted_values`.

```
# your code goes here
```

Exercise 5.

Modify the model above to include a categorical covariate, the number of gears `gear`. To do so, you'll have to allow `alpha` to vary across data points in the likelihood. `alpha` will have to be defined as a vector of length 3.

```
# your code goes here
```

Hint: there are several ways to approach this problem, but one involves the following indexing 'trick': if `alpha` is a vector of three numbers, and `indices` is a set of `N` integers that only include 1,2,3, then `alpha[indices]` is a vector with `N` elements (the elements of `alpha` 'repeated' corresponding to the pattern given in `indices`. This is similar to how R works, and can be demonstrated (in R) as follows:

```
alpha <- c(-0.5, 3, -18)
indices <- c(1,1,3,2,2,1,2,3)
alpha[indices]
```

```
## [1]  -0.5  -0.5 -18.0   3.0   3.0  -0.5   3.0 -18.0
```

Worked example of Markov chain Monte Carlo (for the curious)

At this point, Stan probably seems like a bit of a black box. Like many programs for Bayesian inference, Stan draws samples from the posterior distribution using a variant of 'Markov chain Monte Carlo' (MCMC for short). MCMC is a very clever way to sample from a probability distribution known only up to a constant: this is exactly the case with the posterior distribution! (The normalizing constant in this case is the "marginal likelihood"–the denominator of Bayes' rule.)

It's hard to overstate the importance of MCMC to Bayesian inference. Although developed in the late 1940s by physicists, MCMC wasn't generally adopted by the Bayesian community until the late 1980s/early 1990s. Prior to this time, easy solutions to Bayesian inference problems were few and far between. MCMC opened the doors of Bayesian inference to general practitioners, and is largely responsible for the explosion of Bayesian-flavoured statistics in the past couple decades.

In simulation-based Bayesian inference, our goal is to get samples from the probability distribution $p(\theta|y)$. Ideally, we'd have some nice routine like that implemented in `rnorm()` to quickly draw *independent* samples from the distribution of interest. Unfortunately, this isn't possible most of the time: we don't have a tailoured algorithm that can efficiently draw independent samples from the posterior.

The next best thing would be to draw samples from the posterior that are *not independent*, but have some way of thinning–or transforming–them so that they are independent. It turns out that this is possible for a wide variety of cases, by constructing a Markov chain. A Markov chain is a type of stochastic process – a sequence of random variables moving forward in time – where the state at a current time t depends only on the the state from the previous time $t-1$. The "stationary distribution" of the Markov chain simply refers to the long-run frequency with which the chain stays at a given state – in other words, the proportion of time the Markov chain occupies a given state.

It turns out that it's rather easy to construct a Markov chain that targets the posterior as a stationary distribution. To do this we must specify a proposal distribution: this is a probability distribution that 'picks' a new state θ_{t+1} conditional on the current state θ_t, and is denoted as $q(\theta_{t+1}|\theta_t)$. The proposal distribution must also be valid the "backwards" transition $q(\theta_t|\theta_{t+1})$.

With the proposal distribution, and the unnormalized posterior (as constructed in the first section of this document), we can construct what is known as the 'transition kernel': this is the probability of *actually moving* to the new state from the current state. This has a particular form: $1 \wedge \frac{p(\theta_{t+1}|y)p(\theta_{t+1})q(\theta_t|\theta_{t+1})}{p(\theta_t|y)p(\theta_t)q(\theta_{t+1}|\theta_t)}$ Where $1 \wedge x$ means take 1 or x, whichever is smaller. The chain will move to the new state with this probability, and stay at the current state otherwise.

The simplest form of the algorithm (known as Metropolis-Hastings) is as follows:

1. Propose new values θ^* from distribution $q(\theta_{t+1}|\theta_t)$
2. Calculate acceptance ratio $r = \frac{p(\theta^*|y)p(\theta^*)q(\theta_t|\theta^*)}{p(\theta_t|y)p(\theta_t)q(\theta^*|\theta_t)}$
3. Set $\theta_{t+1} = \theta^*$ with probability $1 \wedge r$, set $\theta_{t+1} = \theta_t$ otherwise

Example 3

Let's put the Metropolis-Hastings algorithm into practice for the regression problem. The likelihood is:

```
log_likelihood <- function(alpha, beta, sigma, response, covariate){
  sum( dnorm(response, mean = alpha + beta*covariate, sd = sigma, log=TRUE) )
}
```

We'll use normal(0,10) priors for the intercept and slope, and a half-cauchy(0, 5) prior for the standard deviation.

```
log_prior <- function(alpha, beta, sigma){
  dnorm(alpha, 0, 10, log=TRUE) +
    dnorm(beta, 0, 10, log=TRUE) +
    dcauchy(sigma, 0, 5, log=TRUE)
}
```

We'll use a normal proposal distribution for the slope/intercept (with mean equal to the current state and std deviation 0.7 and 2.5 respectively), and a log-normal proposal distribution for σ (with mean equal to the log of the current state and std deviation 0.5). These 'tuning parameters' for the proposal distribution will strongly affect how quickly the chain moves through parameter space, and in this case I found them through trial and error.

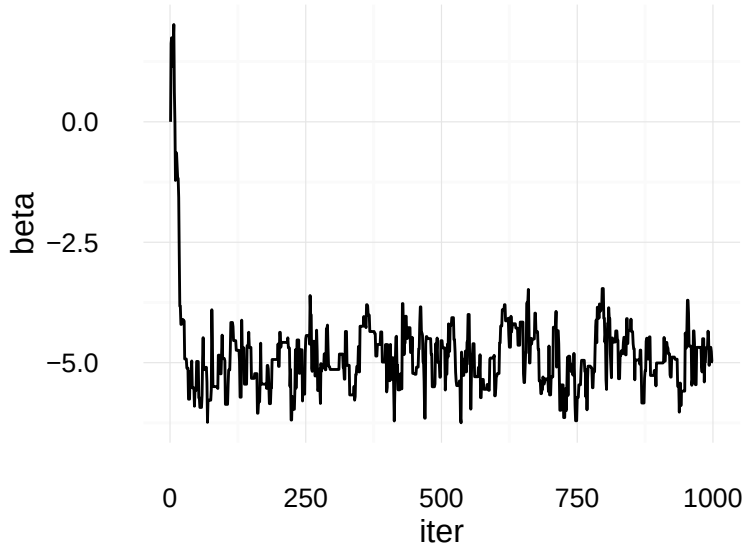
```
log_proposal <- function(alpha_new, beta_new, sigma_new, alpha, beta, sigma){
  dnorm(alpha_new, alpha, 2.5, log=TRUE) +
    dnorm(beta_new, beta, 0.7, log=TRUE) +
    dnorm(log(sigma_new), log(sigma), 0.5, log=TRUE)
}
```

It turns out that this particular form of proposal distribution cancels out in the transition kernel, but I'll include it below so you can convice yourself!

```
# MCMC algorithm (Metropolis-Hastings)
set.seed(101)
niter <- 1000 # number of iterations
nthin <- 10 # thinning interval
stored_iterations <- matrix(NA, niter, 3) # storage for simulations
colnames(stored_iterations) <- c("alpha", "beta", "sigma")
alpha <- 0; beta <- 0; sigma <- 1
stored_iterations[1,] <- c(alpha,beta,sigma) # starting values
for(i in 2:niter){
  for(j in 1:nthin){
    proposed_alpha <- rnorm(1, alpha, 2.5)
    proposed_beta <- rnorm(1, beta, 0.7)
    proposed_sigma <- exp(rnorm(1, log(sigma), 0.5))
    acceptance_prob <-
      log_likelihood(proposed_alpha, proposed_beta, proposed_sigma,
                     response = mtcars$mpg, covariate = mtcars$wt) +
      log_prior(proposed_alpha, proposed_beta, proposed_sigma) +
      log_proposal(alpha, beta, sigma, proposed_alpha,
                   proposed_beta, proposed_sigma) -
      log_likelihood(alpha, beta, sigma,
                     response = mtcars$mpg, covariate = mtcars$wt) -
      log_prior(alpha, beta, sigma) -
      log_proposal(proposed_alpha, proposed_beta, proposed_sigma,
                   alpha, beta, sigma)
    if( log(runif(1)) < acceptance_prob ){
      # accept new values with probability 'acceptance_prob'
      alpha <- proposed_alpha
      beta <- proposed_beta
      sigma <- proposed_sigma
    }
    stored_iterations[i,] <- c(alpha, beta, sigma)
  }
}
```

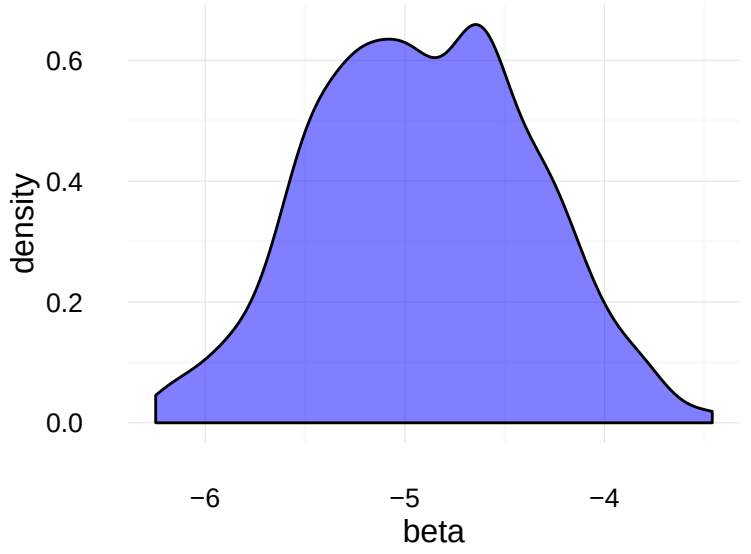
We can look at the trajectory of the Markov chain for the slope:

```
# plot Markov chain for 'beta'
data.frame(iter=1:niter, stored_iterations) %>%
  ggplot(aes(x = iter, y = beta)) + geom_line() +
  theme_minimal()
```



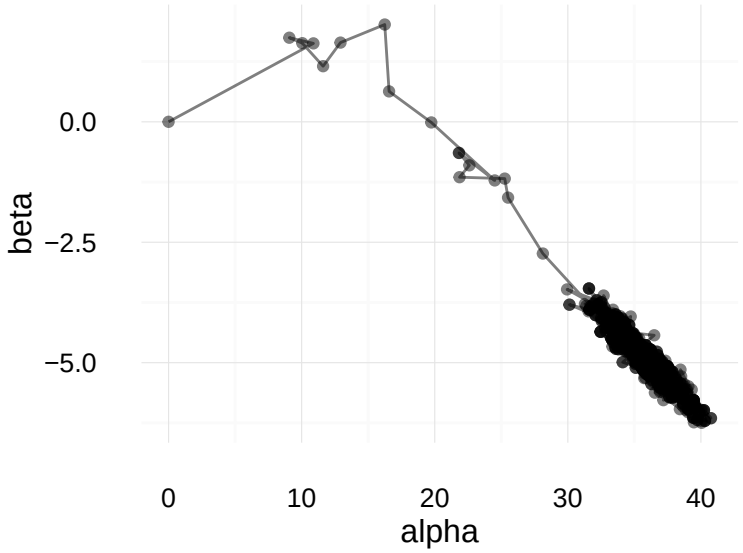
Or, a smoothed density estimate for the slope:

```
# approximate marginal distribution of `beta`
data.frame(iter=1:niter, stored_iterations) %>%
  filter(iter > 100) %>% ggplot(aes(x = beta)) +
  geom_density(fill="blue", alpha=0.5) + theme_minimal()
```



It can also be instructive to look at the trajectory in "2d", for example, with the slope on the y-axis and the intercept on the x-axis.

```
# plot Markov chain in 2d state space for 'beta' and 'alpha'
data.frame(iter=1:niter, stored_iterations) %>%
  ggplot(aes(x = alpha, y = beta)) +
  geom_path(alpha=0.5) + geom_point(alpha=0.5) +
  theme_minimal()
```



This is a very simple type of MCMC algorithm, and one that is not very efficient in high dimensions! Stan uses a different approach to generate proposals called Hamiltonian Monte Carlo (HMC). HMC results in chains that move relatively quickly through parameter space but have a high acceptance rate (so they don't get stuck on the same value for a long time). A good introduction to HMC is outside the scope of this tutorial. Bayesian computation is a very active area of statistical research: the current challenge is developing methods that scale to massive datasets.