Bayes Computational Methods

Evan Miyakawa

Introduction

The Bayes Paradigm

In Bayesian statistical inference, we are interested in finding the posterior distribution of certain parameters, given a prior distribution and a likelihood for those parameters our data. Much of Bayesian theory is built upon the definition of conditional probability, which states that we can find the probability of a variable θ, conditioned on x, by

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)}.$$

In any Bayesian inference, the first step is to find the joint probability distribution of all the observed and unobserved quantities in the problem. Using the definition of conditional probability, we can then find the posterior distribution of the parameters of interest. Bayes theorem states that the posterior distribution of a parameter θ is equal to

$$p(\theta|x_1, x_2, \ldots, x_n) = \frac{\ell(\theta|x_1, x_2, \ldots, x_n)\pi(\theta)}{\int \ell(\theta|x_1, x_2, \ldots, x_n)\pi(\theta)d\theta} = \frac{\ell(\theta|\mathbf{x})\pi(\theta)}{\int \ell(\theta|\mathbf{x})\pi(\theta)d\theta},$$

where $\ell(\theta|\mathbf{x})$ is proportional to the likelihood of θ, conditioned on the data, and $\pi(\theta)$ is the prior distribution of θ.

For example, suppose that $X_i \sim \text{Bern}(\theta)$, and we are interested in inference on θ. A reasonable prior to assign to θ would be a Beta(α, β), distribution, given that θ is constrained to be between 0 and 1, which is a property of the Beta distribution. Therefore, given that $\theta \sim \text{Beta}(\alpha, \beta)$ for some known values of α and β, then

$$\begin{aligned}
p(\theta|x) &= \frac{\ell(\theta|\mathbf{x})\pi(\theta)}{\int \ell(\theta|\mathbf{x})\pi(\theta)d\theta} \\
&= \frac{\theta^{\sum_{i=1}^n x_i}(1-\theta)^{n-\sum_{i=1}^n x_i}\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)+\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}}{\int \theta^{\sum_{i=1}^n x_i}(1-\theta)^{n-\sum_{i=1}^n x_i}\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)+\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}d\theta} \\
&\propto \theta^{\sum(x_i)+\alpha-1}(1-\theta)^{n-\sum(x_i)+\beta-1},
\end{aligned}$$

which is the kernel of a Beta $\left(\sum(x_i) + \alpha, n - \sum(x_i) + \beta\right)$ distribution. Therefore, we can analytically calculate the posterior distribution of θ as

$$p(\theta|x) \sim \text{Beta}\left(\sum(x_i) + \alpha, n - \sum(x_i) + \beta\right).$$

One challenge in bayesian inference comes in evaluating the integral in the denominator of Bayes theorem, which is used to normalize the posterior distribution. If we don't have distributions for the likelihood and prior that are conjugate (or in other words, play nice together), then it is impossible to analytically calculate the posterior distribution.

Bayesian Computational Methods

The need for bayesian computational methods arises in our search for an alternative approach to finding the posterior if we cannot do so analytically. Most of the time, the goal of any bayesian analysis is to not to find the posterior distribution, but to calculate expectations with respect to the posterior. However, in order for these estimates to be accurate, we often need to know about the full behavior of that posterior distribution. In bayesian computing, the aim is to somehow obtain individual draws of values that would come from the posterior, and then aggregate a lot of them to compute the expectations we need.

Markov Chain Monte Carlo

Markov Chain Monte Carlo (referred to as MCMC), is a common approach that allows for drawing samples from a target distribution, which in our case is the posterior. MCMC is an iterative algorithm, where we obtain one draw at a time from the parameter space from which we are trying to sample. In any Markov chain, the marginal distribution of each draw is conditioned on the values of our chain from the previous draw. As we keep obtaining draws, these marginal distributions asymptotically converge to the true target distribution. Therefore, if we run the markove chain for many iterations, we can expect to see draws coming from the true posterior distribution we are trying to estimate. MCMC is the only suitable method for obtaining the posterior distribution when we don't have conjugate priors.

Metropolis-Hastings Algorithm

Gibbs Sampling

Gibbs Sampling is one of the most common Markov Chain Monte Carlo Algorithms used for Bayesian computation. A special case of the Metropolis-Hastings algorithm, a Gibbs sampler works by generating a multi-dimensional markov chain, with one dimension for each unknown variable of interest. By splitting the vector of random unknown variables, the algorithm samples each subvector one by one, conditional on the most recent values of the other variables.

Lunn et. all outlines the Gibbs sampler algorithm as follows: If we have a vector of unknown parameters θ, where $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$, then

1. Choose arbitrary starting values $\theta_1^{(0)}, \theta_2^{(0)}, \ldots, \theta_k^{(0)}$
2. Complete one iteration of the sampler by getting new values for θ using the following:
    - Sample a new value for θ_1, from the full conditional distribution of θ_1, given the most recent values of all other elements of θ and the data:

$$\theta_1^{(1)} \sim p(\theta_1 | \theta_2^{(0)}, \theta_3^{(0)}, \ldots, \theta_k^{(0)}, y).$$

    - Sample a new value $\theta_2^{(1)}$ for the second component of θ, from the full conditional distribution $p(\theta_2 | \theta_1^{(1)}, \theta_3^{(0)}, \ldots, \theta_k^{(0)}, y)$.
    - Continue this process for the remainder of the k elements of θ.
3. Repeat step 2 for the total decided number of iterations that the sampler will run.

The beauty of the Gibbs sampling strategy is that, no matter how complex the space of the posterior distribution is, sampling from these high-dimensional spaces can be broken down into more straightforward draws from one-dimensional distributions. Gibbs sampling is also easier to understand and implement, which makes it very versatile. However, full conditional distributions for each draw are needed in order to do the method.

Because the Gibbs sampler is a random-walk algorithm, if several variables are highly correlated, traversing the space can take a very high amount of iterations because the step-size needs to be small in order to not step outside the target distribution. A two-dimensional example of this is seen in Figure 1, where the joint

distribution of θ_1 and θ_2 is highlgy correlated. Since each Markov transition only allows us to move in the direction of one dimension at a time, it may take many steps in order to travel up and down the parameter space, as demonstrated by the red arrows. In reality, the movement of the chains will be much more random since the algorithm does not know which direction will allow the chain to most efficiently travel through the posterior.

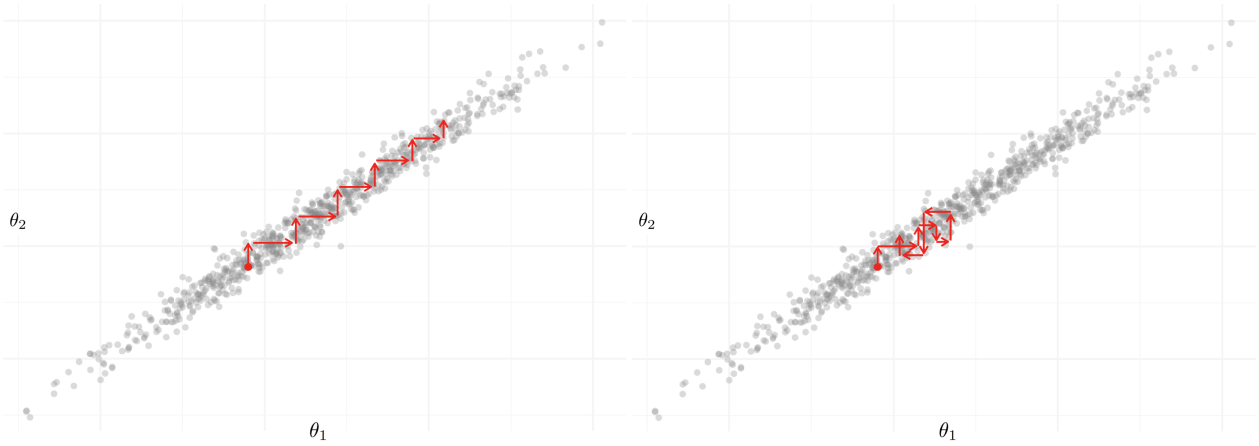

Figure 1: *This is a demonstration of a Gibbs sampler in a highly correlated two-dimensional parameter space. On the left, the ideal behavior of the markov chain is shown, moving as quickly as possible from the bottom-left to the top-right to efficiently sample from the target distribution without stepping outside of it. On the right, we see the more realistic behavior of the markov chain, as the random-walk behavior of the Gibbs sampler moves the chain around in directions that aren't always preferred.*

Hamiltonian Monte Carlo

A main of objective of Hamiltonian Monte Carlo is to use the geometry of the typical set to help us know where to sample from, instead of blindly using a guess-and-check method (Betancourt HMC Youtube talk). This method utilizes Hamiltonian Dynamics, which is a physics system that has three elements: position, momentum, and potential energy variables. To use Hamiltonian Dynamics to traverse the typical set we are interested in, each parameters of interest is transformed into a position variable, which can be visualized as a frictionless puck sliding over a surface of varying height. Each position variable p has an associated momentum variable q that defines the direction and magnitude of the movement of the "puck". At any given time, the value of the parameter can be found as the position of p in some dimension.

We define the joint probability of the position p and momentum q as $\pi(q,p) = \pi(q|p)\pi(p)$. This structure allows us at anytime to throw out q and recover the value of p, which holds the position values of the parameter(s). The Hamiltonian is defined as

$$\begin{aligned} H(q,p) &= -\log \pi(p|q)\pi(q) \\ &= -\log \pi(p|q) - \log \pi(q) \\ &= -K - V, \end{aligned}$$

where K is potential energy and V is kinetic energy. Taking partial derivatives of the Hamiltonian yields Hamilton's equations, which is where we obtain our vector field.

$$\frac{dq}{dt} = \frac{\partial K}{\partial p}$$
$$\frac{dp}{dt} = -\frac{\partial K}{\partial q} - \frac{\partial V}{\partial q}.$$

Hamiltonian Dynamics can be used to sample from the target distribution by translating the density function of the distribution to a potential energy function, and obtaining position variables (and associated momentum variables) for parameters of interest. We can simulate a markov chain where each iteration consists of the following steps:

1. New values for the momentum variables are randomly drawn from their Gaussian distribution, independent of the current values of the position variables
2. A metropolis update is performed, using hamiltonian dynamics to propose a new state for the position variables. The proposed state is either accepted or rejected based on the Hamiltonian Monte Carlo probability function used. If the proposed state is rejected, the next state is the same as the current one.

Hamiltonian Monte Carlo methods are very robust, in that the markov chains seldom get caught in complex regions of the posterior compared to other common MCMC methods. When the chains do get caught, the HMC chains break so obviously that errors can be easily spotted and reported to the user.

STAN

STAN 2015 - Based on a probabilistic programming language that is more flexible and expressive than BUGS/JAGS. - Based on HMC - Works in command line, Python, and R - "STAN is a platform for Bayesian inference that tries to abstract as much computation away as possible, so users can focus on building complex models"

Bayesian Samplers

BUGS

The BUGS project, which stands for Bayesian inference Using Gibbs Sampling, began in 1989 in an effort to create a program that utilized MCMC to approximate draws from the posterior distribution in a Bayesian framework. The BUGS language has functions and distributions that allow for specifying relationship between different "nodes" in a bayesian model. While the language itself has a somewhat limited syntax, it is able to describe a very diverse and complex set of models. The BUGS language is declarative, meaning that the order of statements in a model script does not matter since the entire script is evaluated at once, instead of line-by-line in a procedural language.

Many different programs have been developed that utilize the BUGS syntax to accomplish bayesian sampling, and while these programs have continually evolved, the BUGS syntax has remained very consistent. The Windows program WinBUGS was created by the inventors of the BUGS language, and was a more popular choice for bayesian computation until it became deprecated in 2007. OpenBUGS is now more commonly used in place of WinBUGS, although it's last stable release was in 2014. OpenBUGS communicates better with other software than WinBUGS and is more portable.

In general, there are six steps needed to run bayesian models in WinBUGS/OpenBUGS:

1. Specify the model through BUGS syntax and check to make sure the model code is valid.
2. Load the appropriate data.
3. Compiling the BUGS model.

4. Start the simulation at appropriate initial values for the Markov chains.
5. Simulate draws from the posterior distributions of the unknown parameters of interest.
6. Gather the results and report summary statistics, along with other appropriate output.

OpenBUGS

OpenBugs is the oldest of the Bayesian samplers that we use. OpenBugs is designed for Windows machines, but there is a workaround to run it on MacOS and Linux, using WINE, a program designed to run Windows software on other operating systems. OpenBugs can be used through the actual click-and-point application, or through `R`, using the `R2OpenBugs` package, which allows for the user to send model scripts and data to OpenBugs without needing to leave the `R` interface. The `R` command to run OpenBugs is `bugs()`. Using this method causes OpenBugs to run invisibly in the background and return the results, unless an argument is specified to debug code in the actual application itself, which can be very useful. When using R to run OpenBugs, there is little way to know what went wrong if the code errors somewhere without interacting with the OpenBugs program itself. Using the `debug = T` argument within `bugs()` will cause the application to run in the forefront as if it were actually being used directly by the user. This way, any messages given by the application while running can be viewed directly. It is recommend to use `debug = T` only when debugging, as it does slow down the sampling process slightly. OpenBugs has many tools to visualize the performance the sampler which are easy to use. The biggest drawback of using the OpenBugs application directly is that specifiying the data can be tedious if the data is not very small in size or design. Using R to specify the data is much easier. One unique feature in OpenBugs is the Doodle tool, which allows for model specification through the creation of a diagram of nodes representing the parameters. Though using Doodle can be helpful for illustration, it is not used often in practice.

JAGS

In terms of model specification, JAGS is very similar to OpenBugs, but works more seemlessly in R using the package `rjags`. JAGS was developed independently of OpenBUGS, but uses the BUGS language for most of its model specification syntax. The `runjags` is an optional package that allows for the JAGS model script to be inputted as a string, and run using the `run.jags()` function. Since JAGS runs inside `R` itself, it is very easy to work with. Helpful errors are given when there is an issue with the model code or sampling process. JAGS is often a very fast sampler, especially for less complicated models.

Nimble

Nimble is an extension of Bugs that allows for bayesian computation within `R`, but uses C++ under the hood to compile models and algorithms for speed. Using the `nimble` package, Nimble can perform MCMC bayesian computations, but also allows for other types of algorithms to be loaded or custom-built. It can also compile other code without BUGS models into C++ for other uses. For the purpose of bayesian sampling using MCMC, Nimble is used very similarly to BUGS in practicality. The flexibility of Nimble is not needed in most cases when doing MCMC. The model specification language is almost identical to OpenBugs. A model can be both compiled and run using the function `nimbleMCMC()`. Because the model is compiled using C++ each time, this can drastically increase the total computation time as compared to other sampling methods. Occasionally, Nimble can also freeze up or fail, which can take a bit of time to fix.

STAN

STAN is a language in-and-of itself used for statistcal modeling and data analysis. STAN can be used in several programming languages, including `R` and Python. While many bayesian samplers use Gibbs sampling, STAN uses Hamiltonian Monte Carlo under the hood, a more computationally efficient sampler that is

based on an advanced physics simulation. This can greatly reduce the computational time for many types of statistical models, especially for more complicated ones. Like Nimble, STAN compiles each model, but one big advantage is that it can store each compiled model in a .Rds file, which allows for repeated use without the need to re-compile each time. In `R`, STAN is used through the `rstan` package, which allows for data to be loaded through normal `R` data structures. The biggest practical difference in using STAN compared to other bayesian computational methods is in the model specification process. STAN requires a model object in the form of a .stan file, which is created in a very unique language that is more formal than in BUGS. A .stan file, either created through a character string in R or in a seperate text file, can be used by the `stan()` function, which takes the file along with the data to perform the sampling. There is a very robust set of manuals for STAN users, compared to other methods.

Greta

Greta, a very recent addition to the list of bayes computational methods, is made specifically for use in `R`, making the model and data specification process as easy as possible. Not only does it utilize Hamiltonian Monte Carlo methods for sampling, but it also uses Google TensorFlow to perform its calculations, and allows for more flexible use of CPUs and GPUs. Unfortunately, as of the writing of this, it is still in need of lots of development, and lacks an appropriate amount of documentation and support for users. Computationally, it is very slow compared to the other bayes methods, especially in simple cases. There are several types of semi-common models that greta seems unable to properly handle in model specfication at the moment. Greta also attempts to make model specification very user-friendly, but uses some very common function names in the process that might override other often-used `R` functions. Greta does offer a model visualization tool that does not exist in any of the other Bayesian samplers, in that it can provide a graphical representation of the specified model through the use of nodes and arrows. This tool can be very useful in trying to explain how all the components of a model work together.

Example Model

To illustrate how each bayesian computational method is used in practice, we will demonstrate how to construct a sampler for a simple Beta-Binomial model for each of the five methods we are studying.

Data Preparation

First, we need to prepare the data that we will be using.

```
p <- .25 # binomial p
n <-  10 # binomial n

set.seed(1)

(y <- rbinom(1, n, p))

model_data <- list(
  "y" = y,
  "N" = n
)
```

Model Configuration

Next, we will pick some basic model parameters. Here, we will use four chains, a warmup of 1,000 iterations, and a total iteration count of 10,000.

```
n_chains <- 4L
n_iter <- 1e4L
n_warmup <- 1e3L
```

OpenBUGS / R2OpenBUGS

When using OpenBUGS through R, we will use the `R2OpenBUGS` library in order to pipe the data and model code into the program rather than using manual entry. This requires the creation of a temporary file that holds the model text that will be used.

```
## load library
library(R2OpenBUGS)

## model script code
bugs_model <- function() {
  y ~ dbin(p,N)
  p ~ dbeta(1,1)
}

## parameters to monitor
bugs_monitor <- "p"

## temporary model script text file creation
bugs.file <- file.path(tempdir(), "model.txt")
write.model(bugs_model, bugs.file)

## run the model
bugs_fit <- bugs(
    "model.file" = bugs.file, "data" = model_data, "parameters.to.save" = bugs_monitor,
    "inits" = NULL, "n.chains" = n_chains, "n.iter" = n_iter, "n.burnin" = n_warmup
)
```

JAGS

In JAGS, the model script simply needs to be inputted as a string to the `run.jags()` function.

```
## load libraries
library("rjags"); library("runjags")

## model script code
jags_model <- "
  model{
    y ~ dbin(p,n)
    p ~ dbeta(1,1)
  }
"

## parameters to monitor
```

```
jags_monitor <- c("p")

## run the model
jags_fit <- run.jags(
    "model" = jags_model, "data" = model_data, "monitor" = jags_monitor,
    "n.chains" = n_chains, "sample" = n_iter, "burnin" = n_warmup
)
```

Nimble

An additional step needed when using Nimble, which is to specify initial values for all parameters of interest, since the sampler cannot generate initial values on its own.

```
## load library
library(nimble)

## model script code
nimble_model <- nimbleCode({
  y ~ dbin(p,n)
  p ~ dbeta(1,1)
})

## parameters to monitor
nimble_monitor <- c("p")

## specify initial values of parameters
nimble_inits <- list(
  "p" = rbeta(1,1,1)
)

## run the model
nimble_fit <- nimbleMCMC(
    "code" = nimble_model, "data" = model_data,
    "inits" = nimble_inits, "monitors" = nimble_monitor, "nchains" = n_chains,
    "niter" = n_iter, "nburnin" = n_warmup, "summary" = TRUE
)
```

STAN

Running models in STAN requires a .stan file which contains the model script code, written in the unique STAN syntax. This can either be a seperate file, or can be inputted as a character string. It is generally recommended to use a seperate .stan file, espcially when using RStudio, which can do syntax highlighting and point out errors in STAN code.

Here is the .stan file for this example, which is saved as "stan_file.stan". Each STAN file contains a data block, parameters block, and model block at a minimum.

```
data {
  int<lower=0> y;
  int<lower=0> n;
}

parameters {
```

```
  real p;
}

model {
  y ~ binomial(n,p);
  p ~ beta(1,1);
}
```

The sampler can then be run simply using the `stan()` function call, with the .stan file location specified.

```
## load library
library("rstan")

## run the model
stan_fit <- stan(
    "file" = "stan_file.stan", "data" = model_data,
    "chains" = n_chains, "iter" = n_iter, "warmup" = n_warmup
)
```

Greta

Each data variable needs to be converted to a greta array before it can be used in the model.

```
## convert data into greta arrays
y <- greta::as_data(y)
n <- greta::as_data(n)
```

Next, the priors for each parameter of interested are defined using functions that describe the prior distribution paramter values. The distribution of the response variable given in the data is then described using the `distribution()` function. The model is compiled using the `model()` function, before being sampled by `mcmc()`.

```
## load library
library(greta)

## specify prior for parameter(s) of interest
p <- beta(1,1)

## define distribution of data variable and relationship with parameter
distribution(y) <- binomial(n,p)

## compile model and run
greta_model <- model(p)
greta_fit <- mcmc(
    "model" = greta_model, "n_samples" = n_iter,
    "warmup" = n_warmup,"chains" = n_chains
)
```

Common Bayesian Computation Settings

Chains

When using Markov Chain Monte Carlo methods to estimate Bayesian posteriors of parameters, the number of independent Markov chains to be run needs to be specified. It is recommended to use several chains to explore the space of the target distribution, as a single chain may not fully traverse the entire region on its own. For some simpler models that don't have irregular features in the target distribution, using one chain can suffice. Each sampler has an argument in its function call that specifies the number of chains. Some of the samplers have a default value if the number of chains isn't given.

Iterations

The number of steps that each Markov chain will take is a very important setting to get right. If each chain is not set up to run long enough, the parameter space may not be fully explored and the estimates will be inaccurate. Checking convergence statistics can be helpful in determining if the number of iterations is sufficient. An argument specifiying the number of iterations for each chain can be inputted in the function call for each sampler.

Warmup

Since our inferences are based on the assmumption that our draws are from the target distribution, it is wise to discard the samples at the beginning of each chain, since it takes time for the sampler to "settle down". The terms "warmup" and "burn-in" are often used to indicate the number of iterations that we will discard at the beginning of each chain when calculating statistics about our posterior distribution. We can choose this number in the function call for each sampler. In `OpenBUGS`, `Nimble`, and `Jags`, the argument is referred to as burnin, but in `STAN` and `greta` it is called warmup. Some samplers (`greta`, `Jags, Nimble`) treat this warmup argument as a number of iterations to run before starting to track the sampling results, but others (`STAN`, `BUGS`) have this be the actual number of iterations to discard out of the total number of specified iterations, so it is important to know which designation a sampler uses.

Thinning

In some cases, there is high correlation between sequential samples in a given chain, which can bias the resulting posterior inference, which can be diagnosed using some graphical procedures. In this situation, it is wise to "thin" the results by only looking at every ith observation from a chain, such as every 10th observation. While each sampler defaults to including every simulated posterior draw, a user can specify a "thinning rate", which is the rate at which a sampler will keep memory of the observations. A thinning rate of 1 means that every observation will be kept, and a rate of 10 indicates that only every 10th observation will be used.

Initial Value Specification

In order to start sampling, each Markov chain needs to have an initial value at which to start. These initial values can be either be specified, or can often be randomly generated based on the prior distributions for each parameter of interest. In many cases, it is easiest to have the initial values randomly generated by the sampler. However, in cases where the target distribution is very complex, it is possible that poorly chosen initial values can lead to a chain getting "stuck" in a difficult region, or diverging in the wrong direction. In these cases, a more strategic choice of initial values is advised. `Nimble` unfortunately requires for these values to be specified instead of randomly generated. Using `R` to randomly generate the values from the prior distributions can be used in this situation:

```
nimble_inits <- list(
  "alpha" = rnorm(1,0,1000),
  "beta" = rnorm(1,0,1000),
  "sigma" = runif(1,0,10)
)

nimbleMCMC("inits" = nimble_inits, ... )
```

If multiple chains are being used, a list containing a number of lists equalling number of chains can be used:

```
nimble_inits <- list(
  list(
    "alpha" = rnorm(1,0,1000),
    "beta" = rnorm(1,0,1000),
    "sigma" = runif(1,0,10)
  ),
    list(
    "alpha" = rnorm(1,0,1000),
    "beta" = rnorm(1,0,1000),
    "sigma" = runif(1,0,10)
  )
)
```

Alternatively, if a list containing the initial values for one chain is used, these values will be recycled for the remaining chains.

Common Tricks for Model and Data Specification

When describing a bayesian model through code, there are instances were slightly different implementations are needed based on the sampler being used. Both in model and data specification, the user needs to understand the intricacies of the specific sampling tool being used in order to get the analysis to run properly. This section provides some helpful tips for implementing models in each of the different samplers being examined.

Model Specification

The most important distinction between each of the samplers is the way that each allows for model specification text to be inputted.

OpenBUGS

In `OpenBUGS`, the earliest method created of the lot, the model specification code must be wrapped inside the expression `model{}`:

```
model{
  ...model code here...
}
```

Since we are using the `R2OpenBUGS` library to run `OpenBUGS` through an `R` session, a text file with the model code is passed to the `bugs()` function. We can specify the model in `R` and create the appropriate text file:

```
bugs_model <- function() {
  ...model code here...
```

```
}

bugs.file <- file.path(tempdir(), "model.txt")
write.model(bugs_model, bugs.file)

bugs("model" = bugs.file, ... )
```

JAGS

The `JAGS` sampler was intended to allow for model specification code to be almost identical to `BUGS`, so the implementation is very similar. There are two common approaches, one using the `rjags` package, and one using the `runjags` package. We use the latter, which allows for the model code to be specified in a multi-line character string:

```
jags_model <- "
  model{
  ...model code here...
  }
"

run.jags("model" = jags_model, ... )
```

Nimble

Model specification in `Nimble` is extremely similar to `BUGS` as well. Model code is wrapped inside the `nimble` package function `nimbleCode()`, and then fed into `nimbleMCMC()` sampler function:

```
nimble_model <- nimbleCode({
  ...model code here...
})

nimbleMCMC("code" = nimble_model, ... )
```

STAN

The STAN model specification language is significantly different than other samplers, in that a .stan file is fed to the `C++` compiler. A .stan file has a very unique model specification format, which divides code into "data", "parameter", and "model" blocks at a minimum. When needed, "transformed data" and "transformed parameters" blocks are also used. The basic format for model specification is provided below:

```
data {
  ...data specified here...
}

parameters {
  ...parameters specified here...
}

model {
  ...model code here...
}
```

Each data variable and and parameter variable needs to be initialized in the appropriate block, along with associated domain limits. For example, if one were initializing a rate parameter for an exponential distribution

called “lambda”, it would be initialized with a lower bound of 0:

```
parameters {
  <lower=0> lambda;
}
```

A character string specifying the path to the .stan file is then passed to `stan()`.

```
stan_file <- "file.stan"
stan("file" = stan_file, ... )
```

Data Specification and Input

Specifying data for the bayesian samplers in `R` is very straightforward. In `OpenBUGS`, `JAGS`, and `STAN`, a list containing each data object or constant can be passed to the function that calls the sampler. For example, in `JAGS`, the data might constructed like this:

```
n <-  10 # binomial n
y <- rbinom(1, n, p)

jags_data <- list(
  "y" = y,
  "n" = n
)

run.jags("data" = jags_data, ... )
```

In `Nimble`, the constants need to be specified in a separate list from the data.

```
n <-  10 # binomial n
y <- rbinom(1, n, p)

nimble_data <- list(
  "y" = y
)

nimble_constants <- list(
  "n" = n
)


nimbleMCMC("data" = nimble_data, "constants" = nimble_constants, ... )
```

Monitoring Parameters

In `OpenBUGS`, `JAGS`, `Nimble`, and `STAN`, specifying parameters for the sampler to monitor is as easy as providing a character vector with the names of the parameters to monitor:

```
jags_monitors <- jags_monitor <- c("alpha", "beta")

run.jags("monitor" = jags_monitor, ... )
```

STAN Probabilty and Sampling Statements

Inside the model block of a stan file, probability statements can be written in the same convention as in other bayesian sampling methods

```
model{
  y ~ binomial(n,p);
  p ~ beta(1,1);
}
```

This kind of probability statement does not actually perform any sampling, but is instead tranformed into a statement about the incremental log probability density of a target distribution. The log probability density function can be specified directly.

```
model {
  target += binomial_lpdf(y | n, p);
  target += beta_lpdf(p | 1, 1);
}
```

Either method yields the same result, but in some cases it is advantageous to use the incremental log probability density (or cumulative density) function to get exact log probability values for the model. In general, using the standard sampling statement syntax is preferred since it is easier to understand.

Greta's Unique Specification Language

Implementing models in `greta` is very unique in that each data variable or parameter is initialized seperately with its own line of `R` code. Each data object in `R` is redefined using the `as_data()` function in the `greta` package:

```
y <- as_data(rpois(5, theta))
```

Each parameter is defined using a function indicating the prior distribution and the chosen prior parameters.

```
theta <- gamma(3,1)
```

Once all the variables have been defined, the relationship between the data and the unknown parameters is established using the `distribution()` function:

```
distribution(y) <- poisson(theta)
```

Finally, the greta model is constructed using `model()` function, which takes each of the monitored parameters as arguments:

```
greta_model <- model(theta)

mcmc("model" = greta_model, ... )
```

How to Specify Tricky Models

Each sampler has its own unique methods for using certain bayesian model details. We provide illustrations for some of the most common ones.

Truncated Distribution

Using a truncated distribution is a common practice in bayesian modeling, especially when specifying priors for parameters with positive domains. The syntax for specifying a truncated distribution is slightly different

for each sampling method. Below we are using a truncated normal distribution with a lower bound of zero and no upper bound:

```
## Truncated normal distribution in OpenBUGS
tau ~ dnorm(0,1) I(0, )

## Model Text for OpenBUGS using R2OpenBUGS
tau ~ dnorm(0,1) %_% I(0, )

## JAGS
tau ~ dnorm(0,1) I(0, )

## Nimble
tau ~ T(dnorm(0,1), 0,)

## Greta
tau <- normal(0,1, truncation = c(0,Inf))

## STAN
# Parameter block initialization
real<lower=0> tau;

# Model block specification
tau ~ normal(0,1);

# Alternative solution
tau ~ normal(0,1) T[0,]
```

Flat Priors

Sometimes using a flat, diffuse prior is appropriate. Most of the samplers allow for improper flat priors, although `JAGS` does not. In `STAN`, a parameter has is given a flat, improper prior by default if it doesn't have a distribution assigned. Simply declaring the parameter will do the job.

```
## Flat prior in OpenBUGS
theta ~ dflat()

## Nimble
theta ~ dflat()

## Greta
theta <- variable()

## STAN
# Parameter block initialization
real theta;
```

Distribution Parameter Order

Some samplers use different parameter orders for certain distributions.

Some use the σ parameterization of the normal distribution, where σ is the standard deviation, and others use τ instead, where $\tau = 1/\sigma^2$. Among bayesian samplers, `JAGS`, `OpenBUGS`, and `Nimble` use the τ parameteriztion, while `STAN` and `greta` use σ.

Normal: The $N(\mu, \sigma)$ parameterization is used by `STAN` and `greta`, where σ is the standard deviation. The $N(\mu, \tau)$ parameterization is used by `JAGS`, `OpenBUGS`, and `Nimble`, which uses the precision parameter $\tau = 1/\sigma^2$. The same parameterization guidelines are the same for the student-t distribution as well.

Binomial: The $Bin(n, p)$ parameterization is used by `OpenBUGS`, `JAGS`, and `Nimble`, while the $Bin(p, n)$ parameterization is used by `STAN` and `greta`.

Gamma: The $Gam(\alpha, \lambda)$ parameterization, where α is a shape parameter and λ is a rate parameter, is used by all five implementations.

Weibull: The $Weibull(v, \lambda)$ parameterization is used by `OpenBUGS`, `JAGS`, `Nimble`, and `greta`, where λ is a shape parameter. The $Weibull(v, \sigma)$ parameterization is used by `STAN`, where $\sigma = 1/\lambda$.

Equations inside distribution parameters

In order to utilized calculated parameters inside `OpenBUGS`, a variable must be defined using the assignment operator that is the result of the calculation:

```
y_hat <- alpha + beta * x
y ~ dnorm(y_hat, tau)
```

Fortunately, the other samplers allow for variable calculations inside the actual parameter definition of a distribution function. Here is an example in `JAGS`:

```
y ~ dnorm(alpha + beta * x, tau)
```