Bayes Computational Methods

Evan Miyakawa

Bayesian Samplers

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`. 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. 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.

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, ... )
```

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);
```

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 `OpenBUGS`, `JAGS`, `Nimble`, and `greta`. The $Gam(\alpha, \beta)$ parameterization is used by `STAN`, using the scale parameter $\beta = 1/\lambda$.

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)
```