DATA 582: Bayesian Inference

Lecture 7: Rstan

Dr. Irene Vrbik

UBCO MDS

The beginning of the end

- In this module we have explored the Bayesian inference paradigm using both conjugate-family models and MCMC simualtions.
- We coded up the important MCMC algorithms that are necessary to fit of realistic, complex, high-dimensional Bayesian models for a few familiar models in order to get a deeper understanding of what was going on “under the hood”.
- A number of packages and libraries exist across the popular coding languages that implement commonly used models in Bayesian Inference.
- Today we'll talk about a few family of programs and point to some R specific packages that implement them.

BUGS

- *Bugs (Bayesian inference using Gibbs sampling)*, is a general-purpose program which requires data input and model specification.

- It performs Bayesian inference using Markov chain Monte Carlo based on, you guessed it, Gibbs sampling[1] to obtain samples from the posterior distribution.

- It was developed during the 1990s and early 2000s by a group of statisticians and programmers, and began as a project at Cambridge in 1989.

[1] Gibbs sampling is a special case of Metropolis-Hastings where the proposed moves are always accepted (the acceptance probability is 1).

- Bugs can be used for the standard problems we've covered in class but also flexible enough to be extended to countless models.
- While it can handle complicated models for small data sets it is slow for large datasets and multivariate structures.
- This is due to the nature of the Gibbs algorithm; namely, since it updates one parameter at a time, in high dimension this could result in slow convergence.
- This is especially true when parameters are highly correlated with one another; see an example here.

- WinBUGS (Lunn et al. 2000) and OpenBUGS (Lunn et al. 2009) are the most widely used software packages for fitting Bayesian models using MCMC, both derived from BUGS.

- WinBUGS and OpenBUGS enable the user to specify a Bayesian model (likelihood and prior(s)) in a simple language that resembles R.

- The software then uses artificial intelligence to devise the proposal density for a Markov chain whose target distribution is the posterior that results from the user's model specification.

Note: There are a large collection of examples using BUGS; see here (3 volumes)

JAGS

- JAGS (Just Another Gibbs Sampler) is a Bayesian engine that was designed to work with R.
- The language JAGS uses to specify Bayesian models is a variation of the basic BUGS language.
- R users access JAGS through the `rjags` package and may used the `coda` package to analyze the MCMC results.

Stan

- The newest Bayesian engine, Stan (named after Stanislaw Ulam of Monte Carlo fame).
- STAN is a open source software that was designed to be faster and handle models that are out of reach for both OpenBugs and JAGS.
- Stan runs on various versions of Windows, Mac OS X and Linux.
- Popular programming languages have interfaces that allow you to access the Stan engine directly from within your program (eg. rstan and pystan for R and Python, respectively).

- As a final consideration in our Bayesian module let's conduct MCMC simulation using the rstan package; R's interface to Stan.

- There are two essential steps to all rstan analyses:
  - Step 1. define the Bayesian model structure in rstan notation
  - Step 2. simulate the posterior.

- Stan models are written in their own *syntax*.

- We will look at a few familiar models to gain some familiarity with the key elements of an rstan simulation.

Note: Examples and explanations were adapted from: the Stan Reference Manual, BayesRules, and RStan YouTube channel

Blocks

- Stan programs are made up of a series of *blocks*
- The blocks are made up of variable *declarations* and *statements*
- Variables must be defined to have some value (as well as declared[2] to have some type) before they are used — if they do not, the behavior is undefined.
- Stan allows local variables to be declared in blocks
- Stan program are executed in the order in which they are written.[3]

[2]This follows the convention of programming languages like C++ and Fortran, not the conventions of scripting languages like Python or R.

[3]this differs from programs like BUGS

- We will be looking at three types of blocks in today's examples

  `data` The *`data` block* declares the required data for the model.

  `parameters` The *`parameters` block* declares the model parameters

  `model` The *`model` block* is where we specify our prior(s) and likelihood

- The models we will look at will all have a similar skeleton; however, there are other types of blocks available; see here for an overview.

Declarations

- Variable in Stan must have a declared data type. For example,
  - `real` for continuous values and `int` for integers,
  - `vector` for column vectors, `row_vector` for row vectors, and `matrix` for matrices.

  An overview of data types here.

- This forces the user to be intentional and also serves as a way of catching value errors as soon as they occur.

- In the `data` block and `parameters` blocks are solely used to declare the data variables for input and the parameter variables for sampling.

Declarations

Constrained Data Types

- Declarations of variables may be provided with constraints.
- All of the basic data types[4] may be given `lower` and/or `upper` bounds using the following syntax:

```
int a = 5; \\ this defines a static local variable
int<lower=1> N;
real<upper=0> log_p;
vector<lower=-1, upper=1>[3] rho; \\ vector of length 3
```

Note: the use of semicolons for end of lines for declarations

[4] other than `complex`

Comments

- Each variable's value is validated against its declaration as it is read. For example,
  - if you try to assign a negative value to a variable declared `real<lower=0>`, an error will be raised,
  - if you provide data of the wrong size to a vector, an error will be raised
- As a result, data type errors will be caught as early as possible.
- Unlike in R and BUGS, variable identifiers in Stan may not contain a period character.

- The variables declared in the `parameters` block correspond directly to the variables being sampled by Stan's samplers.
  - These samplers with HMC (Hamiltonian Monte Carlo) and NUTS[5] (no U-turn sampler).

- The `model` block consists of specifying the likelihood and the priors.

- They typically consist of defining a probability model using a *sampling statement*.

[5] Hoffman, Matthew D., and Andrew Gelman. 2014. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research* 15: 1593–623. [1]

Statements

- In general, a sampling statement has the form[6]

```
y ~ dist(theta1, ..., thetaN);
```

- For instance, we might have a statement like:

```
y ~ binomial(10, theta);
```

or

```
y ~ normal(mu, sigma);
```

[6] `model` blocks allow sampling statements while `data` and `parameter` blocks do not.

Reading input Data

- All of the variables declared in the `data` block must be read into memory. If a variable cannot be read, the program will halt with a message indicating which data variable is missing.
- Stan can read this from a file or read it directly from memory in R.
- If the variable has a declared constraint, it will be validate.
- If the constraint is violated, the program will halt with a warning message indicating which variable contains an illegal value, the value that was read, and the constraint that was declared.

Example

Consider the following Beta-Binomial model:

$$\begin{aligned} Y|\theta &\sim \text{Bin}(10, \pi) \\ \theta &\sim \text{Beta}(2, 2). \end{aligned}$$

We discussed how to decide on a prior for θ in previous lectures, so to save time we won't go into any particularly data story.

Step 1

Defining the Bayesian model structure

We must specify three aspects upon which it depends:

- `data` y the observed number of successes in 10 trials.
  - we know this must be an integer between 0 and 10.
- `parameters` the model depends on one parameter θ
  - we know this must be a real number between 0 and 1
- `model` Here we identify the prior on our parameter θ and the form our likelihood takes on.
  - $Y \mid \theta \sim \text{Binomial}(10, \theta)$; specified using `binomial()`
  - $\theta \sim \text{Beta}(2, 2)$; specified using `beta()`

Step 1

`data` block

Our data, y, is the observed number of successes in 10 trials. We know this must be an integer between 0 and 10.

So our `data` block is given by:

```
     data {
      int<lower = 0, upper = 10> Y;
    }
```

Here we declare any of the data that we are going to pass to stan. The input data is usually stored in list. These declarations tell Stan what this data list should look like.

Step 1

`parameters` block

Our model depends on one parameter θ which we know this must be a real number between 0 and 1

So our `parameters` block is given by:

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

Here we declare what are the parameters that we wish to infer.

Step 1

`model` block

Our prior and likelihood take the form:

- $Y \mid \theta \sim \text{Binomial}(10, \theta)$*; specified using* `binomial()`
- $\theta \sim \text{Beta}(2, 2)$*; specified using* `beta()`

So our `model` block is given by:

```
model {
  Y ~ binomial(10, theta); // likelihood
  theta ~ beta(2, 2);      // prior
}
```

Here we declare the prior and the likelihood

Note: the use of two blackslashes to denote the start of a comment

- We translate this structure into rstan syntax and store it as the *character string*.
- We obviously won't be able to go through all the details on the syntax in the time left if this module, but as you will see, they are pretty intuitive.
- If you are interest in more, see Stat documentation for user guides and reference manuals
- Although it can also be done using a character string within R (as we will demo in this example) Rstan recommends using a separate file with a `.stan` extension. In RStudio:

  New File $\rightarrow$ Stan file

```
# STEP 1: DEFINE the model
# alternatively we can save this to *some_name.stan*
bb_model <- "
  data {
    int<lower = 0, upper = 10> y;
  }

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

  model {
    y ~ binomial(10, pi);   // likelihood
    theta ~ beta(2, 2);     // prior
  }
"
```

Comment

Note that the sampling statement in this model is vectorized meaning

```
Y ~ normal(beta0 + beta1 * X, sigma);
```

applies to all *y*s in the `Y` vector.

Alternatively, we could have specified it using a for loop

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

Note: In addition to being more concise, the vectorized form is much faster.

Step 2

- In **step 2**, we simulate the posterior
- We do this using the `stan` function from the **rstan** library.
- Very loosely speaking, `stan()` designs and runs an MCMC algorithm to produce an approximate sample from the Beta-Binomial posterior.

```
# STEP 2: SIMULATE the posterior
bb_sim <- stan(model_code = bb_model, data = list(y = 9),
               chains = 4, iter = 5000*2, seed = 84735)
```

If we saved our model to a file called `bbmod.stan` would use:

```
# STEP 2: SIMULATE the posterior
bb_sim <- stan(file = "bbmod.stan", data = list(y = 9),
               chains = 4, iter = 5000*2, seed = 84735)
```

(assuming `bbmod.stan` is located in our working directory).

Note: note the change in argument (`model_code` vs `file`)

stan

Like with `rstan`, this function requires:

1. model information
   - `model_code` the *character string* defining the model (here `bb_model`, or `bbmod.stan`[7])
   - `data` = a *list* of the observed data (here Y = 9).

The data must always be prepared into a named list with the names corresponding to the variable names declared in our `data` block in the stan model.

[7] to be specified with argument `file` = path to the .stan file

`stan`

2. Markov chain information

`chains` how many parallel Markov chains to run (Stan will run a default of 4 chains)

`iter` the desired number of iterations in, or length of, each Markov chain[8]

`seed` To set the random number generating seed for an rstan simulation (good practice for reproducibility).

[8] By default, the first half of these iterations are thrown out as burn-in

- The result, stored in `bb_sim`, is a `stanfit` object.
- This object includes four parallel Markov chains run for 10,000 iterations each.
- After tossing out the first 5,000 iterations of all four chains, we end up with four separate Markov chain samples of size 5,000, or a combined Markov chain sample size of 20,000.

Note: On Windows you may get a warning about g++ not being found but this appears to be harmless.

- `print` provides a summary for the parameter of the model

- For more detailed statistics you can alternatively use `summary`.[9]

- For diagnosing convergence we can look at our effective sample size which is sufficiently high ($6536/20000 = 0.33$, $5902/20000 = 0.3$) and our $\hat{R}$ (`Rhat`) is 1 (which implies all chains at equilibrium).

Note: the `lp__` is like a log-likelihood, but for the Bayesian framework, up to a constant of proportionality. None of the resources seem to spend much time on this output

[9] For more methods and details of class stanfit, see the help of class stanfit, i.e `?stanfit`

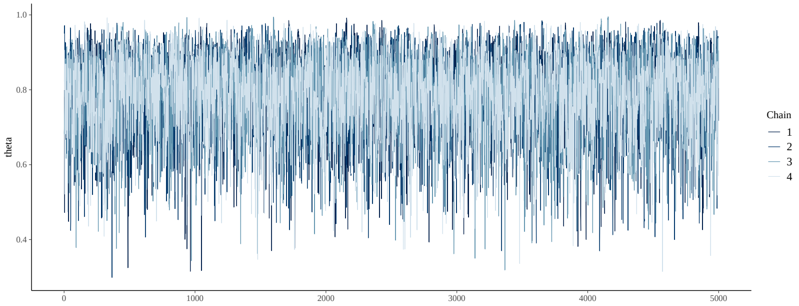
```
> print(bb_sim)
Inference for Stan model: 98850af8240eadd144be26250e6cd1f9.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

       mean se_mean   sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
theta  0.79    0.00 0.11   0.55  0.72  0.80  0.87  0.95  6536    1
lp__  -7.81    0.01 0.78 -10.01 -7.98 -7.52 -7.33 -7.27  5902    1

Samples were drawn using NUTS(diag_e) at Sun Apr 24 11:44:46 2022.
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).
```

While the numeric diagnostics look good, you should also have a look at diagnostic plots as talked about in previous lectures.

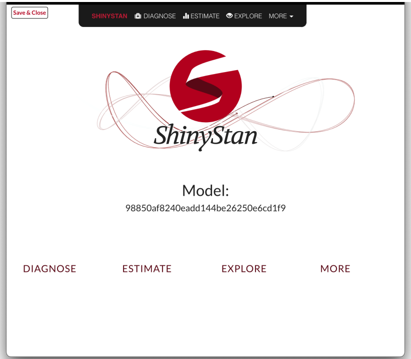
```
library(bayesplot)
mcmc_trace(bb_sim, pars = "theta")
```



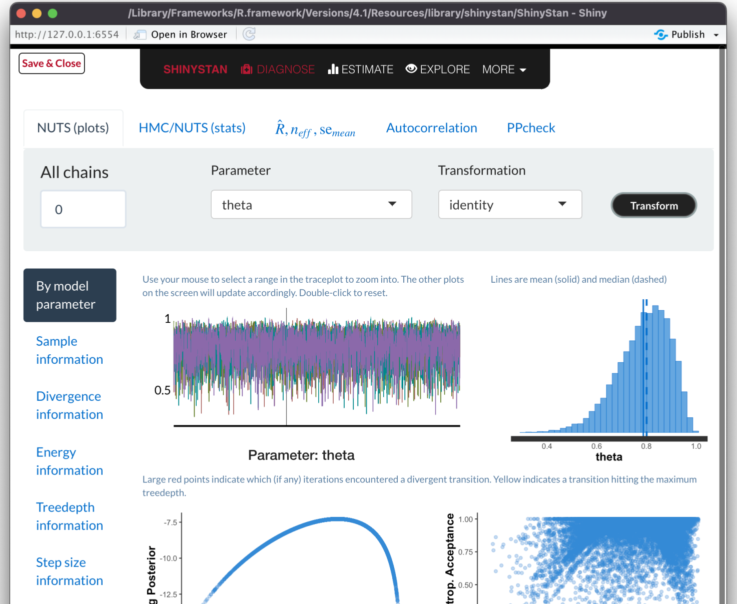
One really cool function is found in the **shinystan** library

```
library(shinystan)
launch_shinystan(stan_bike_sim)
```

The will launches an interactive app in your default browser



Model:
98850af8240eadd144be26250e6cd1f9



0

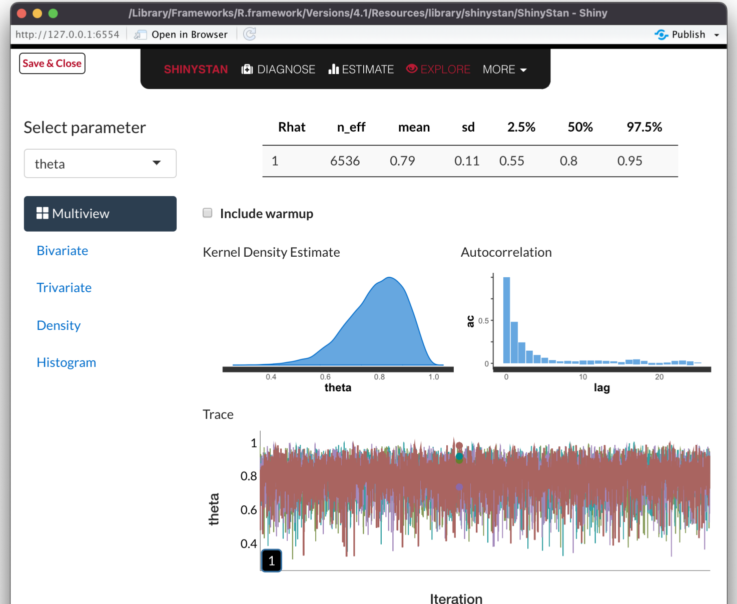
theta

identity

Use your mouse to select a range in the traceplot to zoom into. The other plots on the screen will update accordingly. Double-click to reset.

Lines are mean (solid) and median (dashed)

Large red points indicate which (if any) iterations encountered a divergent transition. Yellow indicates a transition hitting the maximum treedepth.



Rhat	n_eff	mean	sd	2.5%	50%	97.5%
1	6536	0.79	0.11	0.55	0.8	0.95

We can use the extract function on stanfit objects to obtain the samples.

```
> samp <- extract(bb_sim)
> class(samp)
[1] "list"
> names(samp)
[1] "theta" "lp__"
> length(samp$theta)
[1] 20000
> head(samp$theta) # appends all the chains together
[1] 0.8307124 0.7918663 0.6831837 0.7383745 0.8570146 0.8174098
```

We can then do all the kinds of plotting and summaries to these chains as done in previous lectures (more on this in your final lab)

Linear Regression

- We have already seen how to run a Bayesian Normal regression model in R using `rstanarm`.
- Instead of using this “shortcut” function, however, we could have coded this with the `stan` function directly.
- As an exercise, let’s see how this would be done[10]

[10] also see section 9.3.2 in the BR

Linear Regression

- We'll use the bikes ride share example from last class.

$$\begin{aligned} Y_i|\beta_0, \beta_1, \sigma &\overset{ind}{\sim} N\left(\mu_i, \sigma^2\right) \quad \text{with} \quad \mu_i = \beta_0 + \beta_1 X_i \\ \beta_{0c} &\sim N\left(5000, 1000^2\right) \\ \beta_1 &\sim N\left(100, 40^2\right) \\ \sigma &\sim \text{Exp}(0.0008). \end{aligned} \tag{1}$$

Normal Regression using `rstan`

The important pieces of information we must communicate in step 1 are:

`data` : The data on variables Y (response) and X (predictor), `rides` and `temperature`, will be vectors of length n.

`parameters` : Our two regression coefficients β_0 (`beta0`) and β_1 (`beta1`) both can be any real number and standard deviation parameter σ (`sigma`) which must be non-negative.

`model` : Specify the likelihood and priors ....

`model`: Specify the likelihood and priors ....

- data Y is normal with mean `beta0+beta1*X`, and sd `sigma`
- the priors for β_1 and σ will be specified according to the normal model on the previous slide
- Using rstan, we must directly express our prior understanding of the intercept β_0, not the centered intercept β_{0c}
- Hence we must adjust the β_{0c} prior to a prior on β_0 by extrapolating our "prior line" (for lack of better terminolgy).

Starting with our general equation of the line:

$$\texttt{rides} = \beta_0 + \beta_1 \times \texttt{temp}$$

sub the mean β_1 prior

$$\texttt{rides} = \beta_0 + 100 \times \texttt{temp}$$

sub a "typical day" of 5000 riders on a 70 degree day

$$\begin{aligned} 5000 &= \beta_0 + 100 \times 70 \\ \implies \beta_0 &= 5000 - 100 \times 70 \\ &= -2000 \end{aligned}$$

So we instead use the prior $\beta_0 \sim N\left(-2000, 1000^2\right)$

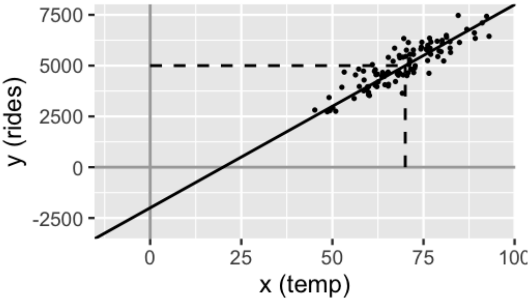


Figure: A simulated set of ridership data with intercept β_0 = - 2000 and centered intercept β_{0c}= 5000 at an average temperature of 70 degrees. Source: Ch9 of BR

```
# STEP 1: DEFINE the model
stan_bike_model <- "
  data {
    int<lower = 0> n;
    vector[n] Y;
    vector[n] X;
  }
  parameters {
    real beta0;
    real beta1;
    real<lower = 0> sigma;
  }
  model {
    Y ~ normal(beta0 + beta1 * X, sigma);
    beta0 ~ normal(-2000, 1000);
    beta1 ~ normal(100, 40);
    sigma ~ exponential(0.0008);
  }
"
```

We can prepare our data for **step 2** by putting them in a list with the same names as the variables declare in the stan model

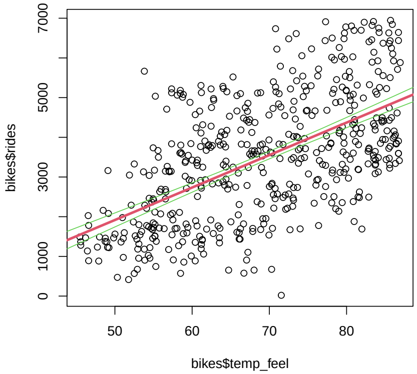
```
bikedata = list(n=nrow(bikes), Y=bikes$rides, X=bikes$temp_feel)
```

Now we simulate the posterior model of $(\beta_0, \beta_1, \sigma)$ using the stan()

```
# STEP 2: SIMULATE the posterior
stan_bike_sim <-
  stan(model_code = stan_bike_model,
       data = bikedata,
       chains = 4, iter = 5000*2, seed = 84735)
```

Note: Alternatively, we can just create that list within the function and not create an explicit R object

Again, we can use these chain values as we did before and more[11]! For example, we can create prediction intervals:



[11] Usually this posterior analysis falls under the heads of estimation, hypothesis testing, predicting)

Comments

- The Stan code we talked about is not specific to Rstan (for instance, we'd use the same stan model if we were using Python's interface with Stan)

- Stan is very flexible and you could define your own functions (if you are using a uncommon pdf for your likelihood for example)

- There are things that can be tweaked (eg. of specifying user-defined initializations (see this chapter) which you can read more about in the Reference Manual.[12]

[12] Also see Stan's User Guide for example models and programming techniques for coding statistical models in Stan.