Bayesian Statistics: Laboratory 3 - Introduction to Stan

Vincenzo Gioia

DEAMS

University of Trieste

vincenzo.gioia@units.it

Building D, room 2.13

Office hour: Friday, 15 - 17

28/04/2024

Stan - ABC

- C++ library for Bayesian modeling and inference that
  - primarily uses the No-U-Turn sampler (NUTS, Hoffman and Gelman, 2012), that is a variant of Hamiltonian Monte Carlo, to obtain posterior simulations given a user-specified model and data
  - alternatively, can utilize the LBFGS optimization algorithm to maximize an objective function, such as a log-likelihood
- The R package **rstan** provides RStan. Take a look to: https://cran.r-project.org/web/packages/rstan/vignettes/rstan.html (see also http://mc-stan.org/rstan/)

Stan - ABC

- Info and guidelines to install **rstan** and set up your pc are available at the following link: https://mc-stan.org/users/interfaces/rstan
- Remember to verify that C++ Toolchain is properly configured: https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started
- Take a look to:
  - Reference manual https://mc-stan.org/docs/reference-manual/index.html
  - Stan website: https://mc-stan.org/
  - Stan user's guide https://mc-stan.org/docs/stan-users-guide/index.html

Stan

A stan file can be created in RStudio from File -> New File -> Stan file. It contains a simple normal model that we use to explore the syntax.

```
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  y ~ normal(mu, sigma);
}
```

Note: comments can be added using the double forward slash // or /* something */ for multiline comments

Stan file structure

Stan programs are organized into blocks (delimited by curl brakets). See https://mc-stan.org/docs/2_18/reference-manual/blocks-chapter.html

You can define up to 7 blocks

- **functions**
- **data**
- **transformed data**
- **parameters**
- **transformed parameters**
- **model**
- **generated quantities**

Example:

- We will see the Stan file structure by means of the following statistical model of interest

$$y_j \sim \mathcal{N}(\theta_j, \sigma_j), \quad j = 1, \dots, 8$$

$$\theta_j \sim \mathcal{N}(\mu, \tau)$$

$$\pi(\mu, \tau) \propto 1$$

  where each σ_j is assumed known.

- It corresponds to the Eight Schools example (from https://cran.r-project.org/web/packages/rstan/vignettes/rstan.html)

Stan file structure: user defined functions block

```
functions{
// user defined functions (language similar to c++)
}
```

- We will see an example of such block in the Cockroaches' example

Stan file structure: data block

- Data block creates objects that are passed in input from the stan function through a list that must have the same objects names
- Note: no statements are allowed here, only declarations

```
data {
  int<lower=0> J;             // number of schools
  real y[J];                  // estimated treatment effects
  real<lower=0> sigma[J];  // s.e. of effect estimates
}
```

Stan file structure: transformed data block

- Here you can transform the data in input (square root and so on)

```
transformed data{
// transformed quantities from the data block
}
```

Stan file structure: parameters block

- Here one can define parameters of the model for sampling: these are the mean (mu) and standard deviation (tau) of the school effects, plus the standardized school-level effects (eta)
- As for the data block no statements are allowed, only declarations

```
parameters {
  real mu;                // population treatment effect
  real<lower=0> tau;      // s.d. in treatment effects
  vector[J] eta;          // unsc. dev. from mu by school
}
```

Stan file structure: transformed parameters block

- Here, you can transform the quantities in the parameter block
- In this model, we let the unstandardized school-level effects (theta) be a transformed parameter constructed by scaling the standardized effects by τ and shifting them by μ rather than directly declaring θ as a parameter. This trick allows sampling more efficiently.

```
transformed parameters{
  vector[J] theta;
  theta = mu + tau * eta;
}
```

Stan file structure: Model block

- The priors and likelihood of your model can be specified in two ways:
  - using the sampling notation, e.g. ‘y $\sim$ normal(mu, sigma)’
  - using the target statement: target is not a variable. It evaluates the log density of the posterior up to an additive constant. It is initialized at 0.
  - You can mix the two notations, e.g. for the prior you can use statements, and for the likelihood target
- The difference between the sampling statement and target is that the sampling drops all the constants, so it can be faster.

```
model {
  // priors (flat, uniform, if omitted)
  eta ~ normal(0,1);          // prior
  y ~ normal(theta, sigma);   // likelihood
}
```

Example

The model block

```
model {
  eta ~ normal(0,1);            // prior
  y ~ normal(theta, sigma);    //likelihood
}
```

can be equivalntly written as

```
model {
  target += normal_lpdf(eta | 0, 1); //log prior
  target += normal_lpdf(y | theta, sigma); //log-likelihood
}
```

Note: for continuous (discrete) distributions: name_lpdf (name_lpmf)

Stan file structure: generated quantities block

```
generated quantities{
// quantities to make inference, e.g. posterior predictive,
// or to simulate pseudo-random
// generated quantities related to the posterior
}
```

- We will see an example of such block in the Cockroaches' example

Stan

Everything you use in the model need to be declared:

- Data
- Parameters
- Other related quantities

Advantage:

- programs are easier to comprehend and debug
- you can't assign the same variable to objects of different types

Note:

- indexing starts from 1
- each line must end with a semicolon ;

Data types

https://mc-stan.org/docs/reference-manual/data-types.html

Primitive types: continuous (*real*) and integer (*int*) values

```
real x; // real[for continuous values]
int x;  // int[for integer values]
```

Vector and matrix types: column vector (*vector*), row vector (*row_vector*), matrix (*matrix*)

```
vector[10] x; // x is a column vector of reals of size 10
row_vector[10] x; //x is a row vector of reals of size 10
matrix[2,3] X; // X is a matrix with 2 rows,
               //3 columns
```

Note: Vectors and matrices cannot be typed to return integer values

Data types

Array types

```
// 1-dim array of size 5 with integer values
array[5] int a;
// 2-dim array of real values with 3 rows and 4 columns
array[3, 4] real a;
/* 3-dim array of real values with 5 rows,
4 columns and 2 shelves */
array[5, 4, 2] real a;
//array of size 3 containing vectors with 7 elements (real)
array[3] vector[7] a;
//15 by 12 array of 7×2 matrices
array[15, 12] matrix[7, 2] a;
```

Constraints

- The constraints are very important and useful for debugging and to make the code more readable.
- If you know that some objects can't assume certain values you should define constraints on them.
- Some common examples are: counts (the size of a sample can't be negative so define a lower bound at 1 or 0), standard deviation or variance is always non negative.

```
int<lower=0> N;
real<upper=1> x;
vector<lower=0, upper=1>[3] a;
```

Constraints

There are some pre-specified data types for vectors and matrices:

```
// For vectors
simplex[10] x; //unit simplex (elements sum up to 1)
unit_vector[5] y; //vector with norm equal to 1
positive_ordered[8] z;

// For matrices
//symmetric, positive definite and unit diagonal
corr_matrix[2,2];
// symmetric, positive definite
cov_matrix[3,3];
```

Exercise

- Write a binary variable z that can be 0 or 1
- Write an object to store the correlation coefficient rho;

Constraints

Other variables can be used to define constraints or object dimensions, but they need to be declared before their use:

```
int<lower=1> i = 5;
int<lower=1> j = 10;
matrix[i,j] x;

real  y[10];
int<lower=1>  N;
vector<lower=min(y)>[N] x;
```

Elements selection

- Try to comment the following line of code (see https://mc-stan.org/docs/2_25/reference-manual/language-multi-indexing-section.html)

```
vector[10] x;
x[2:];
x[2:5];

matrix[10,10] X;
X[2:,];
X[,4:10];
```

Arithmetic operations

Arithmetic operations (like matrix multiplication) or linear algebra functions (eigenvalues) are allowed only among vectors or matrices (not arrays).

```
matrix[2,2] M;
M'; // transpose M matrix
*   // Multiplication
.*   // Elementwise multiplication
/   // Division
./   // Elementwise division
```

Take a look to: https://mc-stan.org/docs/functions-reference/index.html

Example

Consider the Eight Schools example (from https://cran.r-project.org/web/packages/rstan/vignettes/rstan.html) and check that everything works smoothly

Workflow:

- Write your model in a .stan file and check it through the dedicated button
- Define the list of data
- Run your model using the function stan in R

Default numbers of simulations and chains are 2000 and 4, respectively

The algorithm has two phases: warm-up and sampling

Example

- After setting up the correct working directory, run the following lines of code

```
library("rstan")
schools_dat <- list(J = 8,
                    y = c(28,  8, -3,  7, -1,  1, 18, 12),
                    sigma = c(15, 10, 16, 11,  9, 11, 10, 18))

fit <- stan(file = 'schools.stan', data = schools_dat)
```

Example

```
data {
  int<lower=0> J;          // number of schools
  real y[J];               // estimated treatment effects
  real<lower=0> sigma[J]; // s.e. of effect estimates
}
parameters {
  real mu;                // population treatment effect
  real<lower=0> tau;      // s.d. in treatment effects
  vector[J] eta;          // unsc. dev. from mu by school
}
transformed parameters {
  vector[J] theta = mu + tau * eta; // school treat. eff.
}
model {
  eta ~ normal(0,1);            // prior
  y ~ normal(theta, sigma);     //likelihood
}
```