Stan!

Locally Sourced, Non-GMO, Organic Bayesian Models

DS6040 Fall 2024
Teague R. Henry

University of Virginia | School of Data Science

Outline

- Review
- What is a probabilistic programming language?
- Stan Setup Details
- Basic Stan Structure
- A basic analysis in Stan
    - The humble 2 sample t-test…

Review

if you don't provide a prior in STAN, one will be provided for you, but this isn't always a great choice

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}$$

Given our likelihood (model) and priors (you decide), we need to determine the posterior (what we all want).

- If our models + priors are nicely structured (conjugate), we already know how to get to the posterior.
- If not, we can sample or use mode approximation/variational inference.

variational inference: we don't care about being super accurate in uncertainty, but we want to get our point estimates.

But like all good data scientists, we don't want to do the math...

Enter probabilistic programming languages STAN! :D

Probabilistic Programming Language

A PPL is a way of systematically specifying a model, priors, and other bits, so that the underlying software can decide on the appropriate estimating techniques.

- PPLs typically have all the same control logic as in a regular programming language (for-loops, if/then, etc). But these are not used to create "software".
- You can think of PPLs as a dedicated scripting language for running models.
- PPLs take care of all the difficult math, you just need to specify well formed models.

Stan

stan compiles into C++ :D

Stan is a PPL named after Stanislaw Ulam, the developer of Monte Carlo simulations.

- Stan has its own syntax and structure.
- Interfaces for R, Python, Ruby, Matlab, Stata exist.
- Extremely well documented. Any questions you might have will be answered here: [Stan - Documentation (mc-stan.org)]

Stan Installation

We will be using rstan, which is the R interface with the Stan system.

- You must have the C++ toolchain configured for R. (See next slide for info)
- After you have the toolchain configured, all you need to do is install the rstan library:
- install.packages("rstan", repos = "https://cloud.r-project.org/", dependencies = TRUE)

C++ Toolchain Configuration

- Many R packages have C++ code that needs to be compiled. In most cases, R packages are already compiled into binaries.
- Stan dynamically generates C++ code based on your model, so you need the ability to compile C++ code.
- Fortunately, enabling this capability is simple. You need to install Rtools.
  - Windows: Rtools43 for Windows (r-project.org)
  - Mac: Configuring C Toolchain for Mac · stan-dev/rstan Wiki · GitHub
  - Linux: You should be able to figure this out if you are rolling with a Linux distro.

Follow the directions for installation. Let me/TAs know if you are having any issues.

Stan Basic Syntax

A Stan model is defined by a .stan file (you can write these inside of RStudio).

- Stan is a strongly typed language. This means you need to be very explicit as to what each thing in the model means.
- Note the ending ; on each line, you need to have those.
- Indentation doesn't matter.

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

Stan Basic Syntax

Stan has specific blocks where you define different components of a model:

- Data – where you tell Stan very explicitly what data you are putting into the model.
- Parameters – define each parameter
- Model – where you define the priors and likelihood.

Other blocks exist, more later!

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

our priors here are missing. In stan, if we don't specify, we are putting in FLAT PRIORS.
If we were to run our model without the parameter constraints, it would start crashing

Stan Data Block

When you run Stan code using rstan, you submit data in the form of a list:

- `data_list = list(N = 100, y = rnorm(100,0,1)))`
  these data objects are put into the stan call
- Always use primitive data objects:
  - Numerics
  - Vectors
  - Matrices
- You cannot just submit a data.frame. stan won't know what to do with this.

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

Stan Data Block

Because Stan is a strongly typed language, you need to tell it exactly what each element in your data list is.

- Here N is a non-negative integer.
- y is a numeric vector of length N.

Other types:

- `real` – real numbers
- `matrix[i,k]` – A matrix of size i by k.

Constraints:

- `type<upper=X, lower=Y>`
- Constraints are important!

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

Stan Parameters Block

In the parameters block, you define what you want Stan to estimate.

- What parameters you have are all determined by the model, so you would typically write the model block first.
- The parameters block is just for listing each parameter and providing the type.
- Here, mu is a real number
- Sigma is a positive real number.

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

Stan Parameters Block

In the parameters block, you **cannot** do any calculations or transformations.

- The parameters block is just a list of parameters.

Why does this matter?

- Sometimes we want parameters that are complex functions of other parameters.
- We can do this in a different block, but that is for a future day.

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

Stan Model Block

Okay, so you've defined what data is going into the model. You've defined what parameters are going to be estimated.

Now we need to define our priors and model.

The model block –

- Standard probabilistic notation:
  - x ~ blahblah means the object x is distributed as blahblah.
  - This applies to both data and parameters.
  - Other mathematical functions are available.

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

the model block is for you to define your priors and likelihoods.

Specify priors before likelihoods.

Stan Model Block - Priors

What are the priors for the example model?

- We haven't specified them!

If you do not tell Stan what the prior is for a given parameter, it defaults to a flat prior over the given constraints.

- Is this good? Bad? Depends.

So, we need to define our priors explicitly in the model.

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

Stan Model Block - Priors

All we need to do here is tell Stan what we want the parameters to be distributed as:

- mu (our mean) has a uninformative normal prior.
- sigma (our standard deviation) has an uninformative gamma distribution.

As long as the priors are defined as a distribution, this is the way of specifying priors.

- For things like Jeffreys priors, this is a bit more difficult...

```
data {
	int<lower=0> N;
	vector[N] y;
}
parameters {
	real mu;
	real<lower=0> sigma;
}
model {
	mu ~ normal(0, 1000);
	sigma ~ gamma(1,1);
	y ~ normal(mu, sigma);
}
```

Stan Model Block - Priors

You can hardcode hyperparameter values. But it can be more useful to define them as data so you can quickly change them.

- You can also define hierarchical priors, where the hyperparameters of one parameter are a function of other parameters.
  - We will use this later.
- Specifying hyperparameters in your data input makes your model a bit more flexible and easy to work with. But it requires more typing!

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

Stan Model Block - Model

After you've specified your priors, you need to connect them to your data.

- Here, you need to specify the likelihood of the model.
- In the example, the variable y is modeled as a normal distribution.

This is a complete model. We have priors, and a likelihood defined.

```
data {
        int<lower=0> N;
        vector[N] y;
}
parameters {
        real mu;
        real<lower=0> sigma;
}
model {
        mu ~ normal(0, 1000);
        sigma ~ gamma(1,1);
        y ~ normal(mu, sigma);
}
```

Stan Model Block - Model

```r
rstan_options(auto_write = TRUE)

y = rnorm(100, 0, 1)
N = 100
dat_list = list(y = y, N=N)

results = stan(
      file = "Example_1.stan",
      data = dat_list,
     verbose = T)
```

```stan
data {
      int<lower=0> N;
      vector[N] y;
}
parameters {
      real mu;
      real<lower=0> sigma;
}
model {
      mu ~ normal(0, 1000);
      sigma ~ gamma(1,1);
      y ~ normal(mu, sigma);
}
```

Examining Your Stan Fit

So, you've fit your model, and you want to examine the parameter estimates.

- Your starting point is the `summary()` function.
- This gives you the summary statistics across all chains.
- The `extract()` function gives you all the posterior samples (useful if you want to calculate something).

But what about diagnostics?

```
library(rstan)
rstan_options(auto_write = TRUE)

y = rnorm(100, 0, 1)
N = 100
dat_list = list(y = y, N=N)

results = stan(
        file = "Example_1.stan",
        data = dat_list,
       verbose = T)
summary(results)$summary
```

Examining Your Stan Fit

For diagnostics, we can use the `shinystan` library!

- Opens an interactive webpage with all the necessary plots for diagnostics.

To an example!

```
library(rstan)
library(shinystan)
rstan_options(auto_write = TRUE)

y = rnorm(100, 0, 1)
N = 100
dat_list = list(y = y, N=N)

results = stan(
        file = "Example_1.stan",
        data = dat_list,
       verbose = T)
summary(results)$summary

launch_shinystan(results)
```

Stan options

The stan function has a number of options to change number of chains/iterations.

- `chains` – number of chains, defaults to 4
- `iter`- number of draws after burnin
- `warmup` – number of draws for burnin
- `thin` – number of samples to thin (default to 1, use default…)
- `init` – different ways of determining starting values

```
library(rstan)
library(shinystan)
rstan_options(auto_write = TRUE)

y = rnorm(100, 0, 1)
N = 100
dat_list = list(y = y, N=N)

results = stan(
        file = "Example_1.stan",
        data = dat_list,
       verbose = T)
summary(results)$summary

launch_shinystan(results)
```

Stan options

For variational inference, you use a different function: vb

- You need to compile your model separately to use `vb`.
- Summary still works, but won't give you precisely the same information.
- One very nice way of working with vb is to use it to get starting values for a full sampler.

```
library(rstan)
library(shinystan)
rstan_options(auto_write = TRUE)

y = rnorm(100, 0, 1)
N = 100
dat_list = list(y = y, N=N)

mod = stan_model(file =
"Example_1.stan")
results = vb(mod,data = dat_list)

summary(results)$summary

launch_shinystan(results)
```

The 2 sample t-test

Let's say I want to test the difference in means between two groups.

- This is the 2 sample t-test in frequentist statistics.

$$x \sim N(\mu_x, \sigma_x)$$
$$y \sim N(\mu_y, \sigma_y)$$

- Our parameter of interest is a function: $\mu_x - \mu_y$
  - We will use the transformed parameters block!
- Let's put standard priors on the means and standard deviations.

```
data {
        int<lower=0> N1;
        vector[N1] y;
        int<lower=0> N2;
        vector[N2] x;
}
parameters {
        real mu_x;
        real<lower=0> sigma_x;
        real mu_y;
        real<lower=0> sigma_y;
}
transformed parameters {
        real mu_diff;
        mu_diff = mu_x - mu_y;
}
model {
        mu_x ~ normal(0, 1000);
        sigma_x ~ gamma(1,1);
        mu_x ~ normal(0, 1000);
        sigma_x ~ gamma(1,1);
        x ~ normal(mu_x, sigma_x);
        y ~ normal(mu_y, sigma_y);
}
```

The 2 sample t-test

I generated y as N(.25, 1.2) and x as N(0, 1). 1000 samples in each group.

- These results look reasonable. Mean difference of approximately .20, which tracks with the sampling variability.

	mean	se_mean
mu_x	2.035303e-01	0.0005304942
sigma_x	1.185839e+00	0.0003966925
mu_y	6.647023e-03	0.0004509130
sigma_y	1.009210e+00	0.0003243735
mu_diff	1.968832e-01	0.0007157718
lp__	-1.181506e+03	0.0304703884

lp is log posterior

Next Time

More Stan!

- Transformed parameters!
- Generated quantities!
- Example models like regression!