Stan Workshop

Daniel Lee

Using RStan v2.2.0

June 13, 2014

http://mc-stan.org/

1. Installation

Cybertorium machines

If you're working on the Windows machines, this is easy.

1. Log in.
2. SSH into `lionxv.rcc.psu.edu`.
3. Load the R module. This will have RStan built in.
   `> module load R/3.0.1`

That's it for now. Skip to the next chapter.

Personal machine (optional)

Prerequisites:

1. C++ compiler. We've found that clang++ is more memory efficient in compilation than g++, but g++ often produces faster executables. This is very version dependent. If you're using g++, you'll need 4.2 or higher.
2. R. Go install it.
3. Packages within R: RCpp and inline. Install from within R; I suggest installing RCpp from source. It will solve a lot of installation issues.

Edit your `~/.R/Makevars` to use your compiler, if it's not standard. If you're on a Mac, you'll also need to add:
`CXX=g++ -arch x86_64 -ftemplate-depth-256 -stdlib=libstdc++`
For more information, go to:
https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started
Install RStan from within R. Type this into your R console:

```
options(repos = c(getOption("repos"), rstan =
   "http://wiki.rstan-repo.googlecode.com/git/"))
install.packages('rstan', type = 'source')
```

2. Lab 1: RStan Basics

Starting RStan

Start R. On the Cybertorium machines, type `R` from the command line. From within R, type `library(rstan)`. That's it, you've now started RStan.

Coin flip example

We'll introduce basic RStan functionality through an example. In this section, we won't focus on the Stan language and what you can do there, but we'll focus on how to drive the software. For the most part, you will be able to cut and paste in this section.

The data

Supposed we flip a coin 10 times. We record each flip separately, `1` represents heads, `0` represents tails. For example, this could be what we observed: `1 0 0 0 1 0 1 1 0 0` (4 heads, 6 tails). Let's call this variable `y`.

The mathematical model

Let's estimate the probability of heads of the coin that generated this data. In case you're not aware, this is a loaded statement and we've already decided that there's a single coin. (An alternative could be that we flipped 10 coins and observed each coin once. Or perhaps 5 coins, each twice.)

We're going to start with the conjugate model:

$$\theta \sim \mathsf{Beta}(1,1)$$
$$y_i \sim \mathsf{Bernoulli}(\theta), \quad \forall i \in N$$

$\mathsf{Beta}(1,1)$ is a uniform prior from 0 to 1. The likelihood is described by the Bernoulli distribution. This is the joint model of our knowns, y, and our unknowns, θ. We are interested in $\Pr(\theta \mid y)$.

Since this is a conjugate model, we can analytically solve for our posterior distribution, which ends up being:

$$\theta \mid y \sim \mathsf{Beta}(n+1, N-n+1), \text{ where } n \text{ is the number of heads}$$

The expectation of the posterior distribution is $E(\theta \mid y) = \frac{n+1}{N+1}$ The classical point estimate is $\hat{\theta} = \frac{n}{N}$, which asymptotically approaches the posterior estimate as N goes to infinity.

The Stan model

This is in `lab-1/bernoulli.stan` Here is the Stan model:

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  theta ~ beta(1, 1);
  for (n in 1:N)
    y[n] ~ bernoulli(theta);
}
```

We'll show how to infer the parameters of the model later in this lab.

Generate data for use within RStan

RStan needs the data to be in a named list. We'll first generate the data, then prep it for RStan. To generate data, you can paste this into R (this is in `lab-1/generate.R`):

```
set.seed(0)

N <- 10
theta <- 0.2
y <- rbinom(N, 1, theta)
```

If you inspect `y`, you'll find the sequence of `1`s and `0`s as in the section above.

To create a named list, we'll call it `data`, type this into R:

```
data <- list(N=N, y=y)
```

Run RStan, basic evaluations

Type: `fit <- stan('lab-1/bernoulli.stan', data=data)`
You've now run Stan. RStan has translated the Stan code into C++, then compiled the C++ into executable, passed the data in, and run the default sampler.
Summary. To see a summary of the the fit object, either type `fit` or `print(fit)`. The result should look something like:

```
Inference for Stan model: bernoulli.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

      mean se_mean  sd  2.5%  25%  50%  75% 97.5% n_eff Rhat
theta  0.4       0 0.1   0.2  0.3  0.4  0.5   0.7  1170    1
lp__  -8.7       0 0.7 -10.8 -8.8 -8.4 -8.2  -8.2  1249    1

Samples were drawn using NUTS(diag_e) at Thu Jun 12 00:57:23 2014.
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).
```

Traceplots. To see traceplots, type: `traceplot(fit)`

If you only want to see the traceplot of a particular variable, in this case `theta`, provide a vector of variable names to the `traceplot()` function. `traceplot(fit, 'theta')`
Summary plot. To see a plot, type: `plot(fit)`. This will display medians for all chains, 80% (empirical) intervals, and the split R hat statistic for convergence as a color.
Extract samples. To extract the samples drawn, type: `theta <- extract(fit)`. To only extract a specific variable, type something like: `theta <- extract(fit, 'theta')`

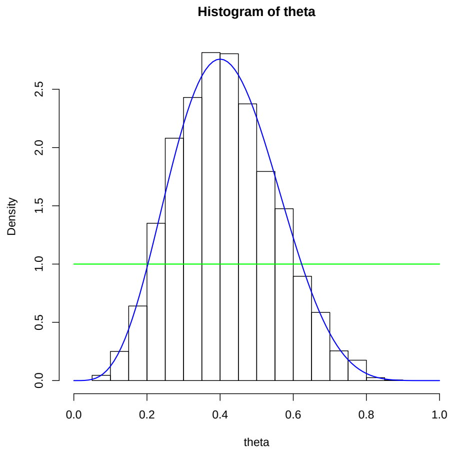

Figure 2.1: *Posterior distribution of a beta bernoulli conjugate model with the prior distribution in green, the posterior in blue for 10 observations, 4 of those being heads.*

Comparing to the analytic solution

Let's start by looking at the samples. Type:

```
theta <- extract(fit, 'theta')
theta <- unlist(theta, use.names=FALSE)
head(theta)
```

(You need the `unlist()` step because `extract()` returns a list as a data type.)

Let's start by showing a histogram of the samples:

```
hist(theta, xlim=c(0,1), freq=FALSE)
```

Let's add the analytic posterior to this plot:

```
curve(dbeta(x, 1, 1),
      from=0, to=1,
      add=TRUE, col='green', lwd=1.5)
curve(dbeta(x, sum(y)+1, N-sum(y)+1),
      from=0, to=1,
      add=TRUE, col='blue', lwd=1.5)
```

You should end up with a figure like Figure 2.1. To find the posterior mean, which is an estimate of the expected value of the posterior distribution, you can just take mean of the samples: `mean(theta)`. You should notice that it's closer to 0.5 than the classical estimate.

Reruning the model without recompiling the model

If we wanted to run the model again without waiting to recompile, we can call `stan()` like this:

```
fit2 <- stan(fit=fit, data=data)
```

This should have run much, much faster! For these simple examples, Stan takes a lot longer to compile than to run.

We can now do the same analysis with the new run, which is in `fit2`.

Optimization

You can also run an optimizer on the same model. Currently, from RStan, this isn't easy to do, but still possible:

```
point_estimate <- optimizing(fit@stanmodel, data=data)
point_estimate
```

For the data we've been using, this should show the point estimate at 0.4.

Exercise 1: change the data

Generate new data. Feed it into the existing model without recompiling. If you generate a lot of data, say 10000 instances, we should be able just put it into the model without recompiling.

Exercise 2: use a non-conjugate prior

Since we're changing the model, we'll need to recompile the model. Go ahead and start playing with the prior. See how changing the model changes the estimates for the data. Also compare to the conjugate model, which we had an analytic solution.

Recap

In this section, you should have learned how to start RStan, get data into a form that can be passed into Stan, compile and run a model, rerun a model without recompiling, some basic graphing.

3. Lab 2: Non-identifiability and Modeling

Simple Gaussian mixture model

Suppose we have data that comes from two Gaussians with some mixture probability, θ. Here's one way to generate the data:

```
set.seed(100)

N <- 1000
theta <- 0.3
mu1 <- -10
mu2 <- 10
sd <- 1

y <- ifelse(runif(N) < theta, rnorm(N, mu1, sd), rnorm(N, mu2, sd))
```

If you look at the histogram, you should data that's separated into two groups.

Estimating the mixture probability and the means

Here's the joint model we wish to estimate:

$$\begin{aligned}
\theta &\sim \mathsf{Uniform}(0,1)\\
\mu_1 &\sim \mathsf{Normal}(0,10)\\
\mu_2 &\sim \mathsf{Normal}(0,10)\\
y_i &\sim \theta * \mathsf{Normal}(\mu_1,1) + (1-\theta) * \mathsf{Normal}(\mu_2,1)
\end{aligned}$$

This model, in Stan, is located in `lab-2/normal_mixture.stan`.

Using increment_log_prob

In the Stan model, we had to increment the log probability directly with the mixture model. That's because Stan doesn't have this mixture built into the language. All we are doing within the loop is computing the likelihood defined above. For complex models, you can use this function to build arbitrary priors and likelihood functions.

Run the model

To run the model:

```
data <- list(N=N, y=y)
fit <- stan('lab-2/normal_mixture.stan', data=data)
```

Check the model results to what was used to generate the data. Check the traceplot of `theta`. You might notice that different chains ended up at different locations. This model is non-identified. By swapping `theta` with `(1-theta)` and swapping `mu[1]` and `mu[2]`, we get the same log probability value.

Non-identifiability is an issue in modeling. If we wanted to sample using MCMC correctly, we should wait until all chains swap between both modes many times. But, even if we did that, our posterior mean for the `theta` would be 0.5, which isn't sensible.

So, what's the solution? In this model, we want to break the symmetry of the model so we can identify it.

Exercise 1: add a different prior

Use a non-symmetric prior on `theta`. This should make the model more identifiable. For example, give it a half-normal distribution with mean 0, standard deviation 0.1. (If this isn't actually identifying the model, you'll find that given the amount of data, this shift in the prior doesn't change the log probabilty enough, as seen from the traceplot of the `lp__` parameter.)

Instead of a prior on `theta`, we could just put different priors on each `mu`.

Exercise 2: restrict theta

One other way to deal with this non-identifiability is to restrict the parameter value itself. By changing the parameter declaration from: `real<lower=0,upper=1>` to setting an upper bound of 0.5, we've now restricted `theta` and the model will be identified.

Exercise 3: estimate scale parameters for Gaussians

Extend the model with parameters for the standard deviation of each of those normals.

Exercise 4: create a k-mixture model

Create a model to accept `k` as data, the number of mixture components.

Recap

In this section, you should have seen a non-identifiability in a very simple model and thought about how to deal with it. Having non-identifiability is an issue with the computation and the posterior analysis and we are often better off addressing the model.

4. Lab 3: Hierarchical model

Eight Schools

This example is covered in *Bayesian Data Analysis*. For a more in-depth treatment, please look at the BDA.

The Educational Testing Service had a study to analyze the effect of SAT coaching. Eight high schools participated. There was no prior belief that any single program was more effective than the others and no prior belief that some were more similiar to each other.

Setup

To make it easy to start, here's the data that can be easily used in RStan:

```
J <- 8
y <- c(28,  8, -3,  7, -1,  1, 18, 12)
sigma <- c(15, 10, 16, 11,  9, 11, 10, 18)

data <- list(J=J, y=y, sigma=sigma)
```

And here's a skeleton model that has the data block already defined.

```
data {
  int<lower=0> J;          // # schools
  real y[J];               // estimated treatment effect
  real<lower=0> sigma[J];  // std err of effect
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
}
```

You'll find this in `lab-3/eight_schools.stan`.

School	Estimated Treatment Effect	Standard Error of Treatment Effect
A	28	15
B	8	10
C	-3	16
D	7	11
E	-1	9
F	1	11
G	18	10
H	12	18

Figure 4.1: *The data for SAT coaching for eight schools.*

No pooling model

Let's start with a silly model. We want to estimate the true school effect, `theta`, for each school.

$$y_i \sim \text{Normal}(\theta_i, \sigma_i)$$

Here, y_i and σ_i are observed, θ_i are our parameters.

Here is the full model:

```
data {
  int<lower=0> J;          // # schools
  real y[J];               // estimated treatment effect
  real<lower=0> sigma[J];  // std err of effect
}
parameters {
  real theta[J];           // school effect
}
model {
  y ~ normal(theta, sigma);
}
```

We'll take this example and expand on it. If you notice, we've used a vectorized call to the normal distribution. Run the model, take a look at it, make sure the estimates line up with the data.

Exercise 1: complete pooling

Let's go to the other extreme. Assume that there's a combined pooled effect, that is, instead of having 8 `thetas`, we have a single `theta`. Change the model so we're estimating the global effect of the program. (This should be quick and pretty unsatisfying.)

Here's the model we're looking for:

$$y_i \sim \text{Normal}(\theta, \sigma_i)$$

Exercise 2: partial pooling

Next up in terms of complexity is to have partial pooling for the school effects, but with the amount of pooling fixed. Here's the model:

$$\theta_i \sim \text{Normal}(\mu, 25)$$
$$y_i \sim \text{Normal}(\theta_i, \sigma_i)$$

We've now added a hierarchical parameter for the global effect of coaching, μ. We are interested in estimating that in addition to what we believe is the true effect of each individual school. We've also fixed the amount of pooling to `25`. As that value goes to 0, the more we believe that all schools are the same.

Exercise 3: full hierarchical model

We've reached the model we really wish to fit. We not only want to estimate the mean effect of the program, but also the group level variance.

$$\theta_i \sim \text{Normal}(\mu, \tau)$$
$$y_i \sim \text{Normal}(\theta_i, \sigma_i)$$

MCMC is necessary for this model

For this model, the variance component's maximum likelihood estimate is 0. In RStan, you can optimize this model and see that the estimate for τ approaches 0. Why is this a problem? This is our complete pooling model. We don't believe that all schools are identical. We are interested in the Bayesian estimate, the posterior density and often summarized by the posterior expectation, of the variance component.

Just to convince yourself, take a look at the histogram of the samples of τ.

Recap

In this section, you should have built up a hierarchical model and explored why you need MCMC for hierarchical models.