Lecture 4. Introduction to Stan

Zachary Marion

2/05/2017

Introduction to the Stan programming language

So far we have had a conceptual introduction to Bayes, discussed the Bernoulli and Binomial likelihood functions, and talked about how to mathematically specify a beta prior given real-world information. Now it is time to learn how to build a Bayesian model in the Stan programming language.

Stan is a standalone C++ library for Bayesian modeling and inference that interacts with a number of different languages such as python, Julia, Matlab, etc. We will be interacting with Stan through the R interface (duh).

Stan is named for Stanislaw Ulam, the co-inventor of Monte Carlo methods (Metropolis and Ulam, 1949). It uses a No-U-Turn Sampler (NUTS) which is a variant of Hamiltonian Monte Carlo.

To begin, we need to load some packages and set some options:

```r
library(shinystan)
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

The `options(mc.cores = parallel::detectCores())` line detects the number of cores you have on your computer so we can incorporate parallel processing of multiple MCMC chains at the same time. (NOTE: I know this works for Macs. I have absolutely no idea if this is the same thing you need to do for PCs or linux-based OS's. Sorry!)

Also, we need to specify the working directory we will be operating out of:

```r
setwd("YOUR DIRECTORY")
```

RStan allows a Stan program to be coded as a text file (usually with suffix `.stan`) or in a R character vector. I recommend using a separate `.stan` file for the model rather than combining both character vector and execution script together.

A Stan model is composed of up to six "blocks": `data`, `transformed data`, `parameters`, `transformed parameters`, `model`, and `generated quantities`. For now we will concentrate on the `data`, `parameters`, and `model` block.

The Stan language is slightly in flux (will probably change a bit with Stan 3.0), but the language combines aspects of both C++ and R. A few important quirks:

- A completed line must end in a semicolon.
- All data and parameters used in the model must be declared and assigned a type (e.g., real or vector).
- Comments can be set aside with //.

The first section of Stan code is the `data` block. Data are declared as `integer` or `real` and can be `vectors` (column order), `row_vectors` (row order), `matrices`, or (more generally) `arrays`. There are also a host of more specific versions of vectors, matrices, or arrays (e.g., `simplex`, `correlation matrix`).

Below is the `data` block for our simple Bernoulli model:

```
data {
  int<lower=0> nObs;                // Total number of observations
  int<lower=0, upper=1> obs[nObs];  // 1D array of observations
  real<lower=0> alpha;
  real<lower=0> beta;
}
```

Here, we have defined `nObs` as a scalar `integer` with a lower bound at 0.

We have declared `obs` to be an `integer` as well, bounded between 0–1 (the two possible values). `obs` is a 1-D `array` of length `nObs`. `Arrays` are always row-order. It cannot be a `vector` because `vectors` are `real` numbers only.

We are also specifying our beta priors, `alpha` & `beta`, to be data. This means that we don't have to hard-code the priors in the `model` block (see below).

Defining the lower and upper bounds of data in the `data` block serves as a QC check. If anything is amiss (say you have a lower bound of zero but negative data) the model will stop and throw an error. This prevents weird and incorrect posteriors.

Next is the `parameters` block:

```
parameters {
  real<lower=0, upper=1> p;     // prob. of water
}
```

We only have one parameter, `p`, a scalar real number. Because `p` is the probability of water, it is bounded between 0–1.

Bounds in the `parameters` block serve a different function than bounds in the `data` block.

- Stan works on a log-probability gradient and will convert constrained paramters to unconstrained parameters when it works with the log probabilities. It will then back-convert the log-posterior.
- This means that it is very easy to specify folded distributions (e.g., setting `lower=0` for a standard deviation and then assigning a normal, cauchy, or t prior will automatically fold the distribution so the parameter is always positive).

Last, we have the `model` block:

```
model {
  p ~ beta(alpha, beta);                //prior on p
  for(n in 1:nObs) {
    obs[n] ~ bernoulli(p);      // bernoulli likelihood function
    }
}
```

Initially, we will set a flat (uniform) beta prior on `p`.

- By default, if a prior is not specified for a parameter, Stan assigns a uniform prior over the bounds given (if any). This is because a uniform prior essentially does nothing and so drops out mathematically.

Finally, we loop over our data for the Bernoulli likelihood function.

- This is actually slower and unnecessary. Stan allows for vectorization, which means we could just have easily ditched the loops and specified `obs ~ bernoulli(p)`.

The last step is to save our full model as a `.stan` object as `ex1Bernoulli.stan`.

```
data {
  int<lower=0> nObs;                // Total number of observations
  int<lower=0, upper=1> obs[nObs];  // 1D array of observations
  real<lower=0> alpha;
  real<lower=0> beta;
}

parameters {
  real<lower=0, upper=1> p;     // prob. of water
}

model {
  p ~ beta(alpha, beta);                //prior on p
  for(n in 1:nObs) {
    obs[n] ~ bernoulli(p);      // bernoulli likelihood function
  }
}
```

Running the Stan code:

First we need to set up the data as a list:

```
obs <- rep(c(1, 0), times = c(7, 3))  # our Bernoulli observations
nObs <- length(obs)  # number of observations
alpha <- 1  # Prior for alpha
beta <- 1  # Prior for beta
dat <- list(obs = obs, nObs = nObs, alpha = alpha, beta = beta)
```

Then, to run the code, we use the `stan()` function:

```
mod1 <- stan(file="04.ex1Bernoulli.stan", #path to .stan file
             data=dat,
             iter=2000, # number of MCMC iterations
             chains=4,  # number of independent MCMC chains
             seed=3)    # set the seed so run is repeatable
```

```
hash mismatch so recompiling; make sure Stan code ends with a blank line
```

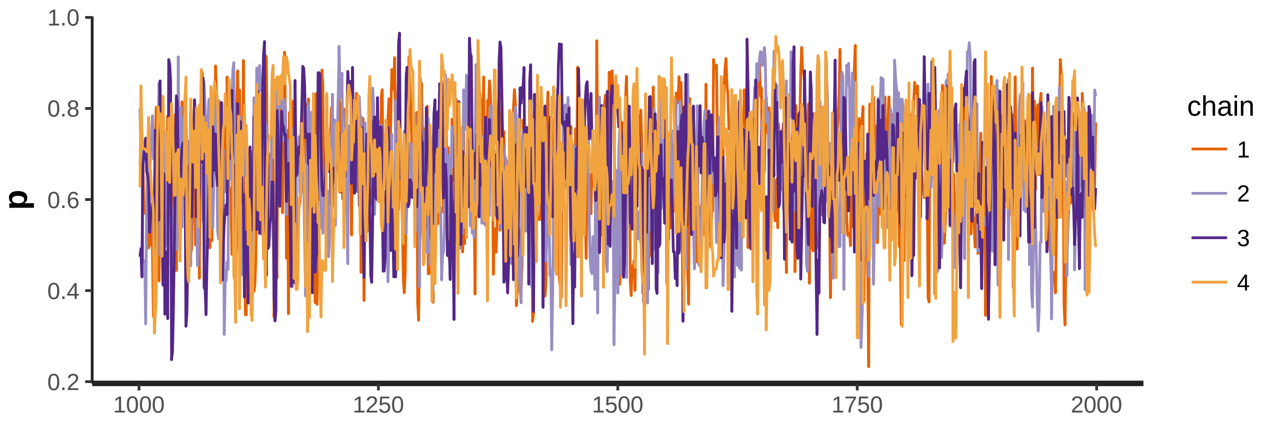
By default, Stan uses the first half of the iterations for each chain as warmup and therefore are discarded. During this time, the HMC sampler is tuned. You can specify the number of warmup iterations with the `warmup` argument.

Before we pay much attention to the results, we need to check some diagnostics to ensure we have reached a stable posterior distribution. We can do this either using the `shinystan` package interface (`launch_shinystan()`) or via command line.

traceplots

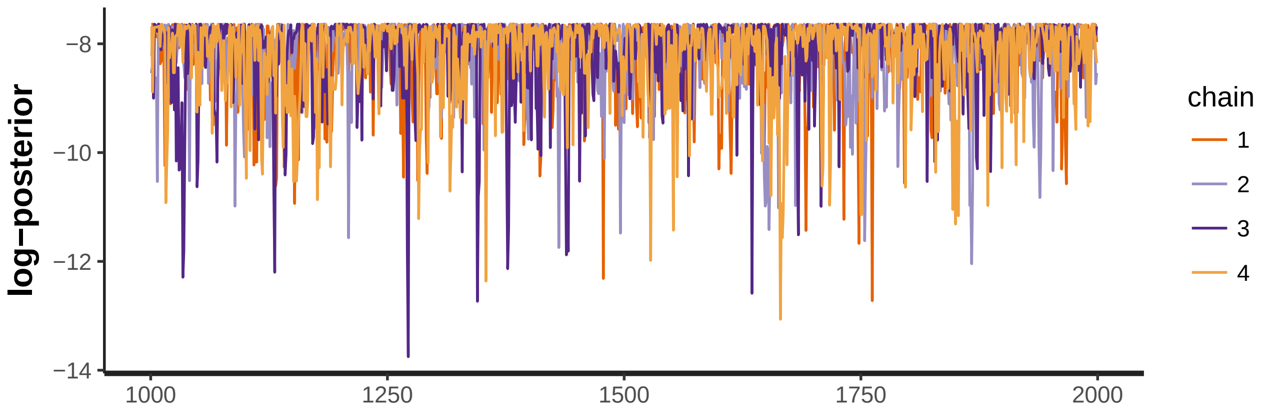
First, we can inspect chain mixing visually for both `theta` and the log-posterior using the `traceplot` function:

```
traceplot(mod1, par="p")
```



The traceplot should look like a hairy caterpillar with no discernable differences among chains. More important is the `traceplot` for the log-posterior, accessed using the `par="lp__"` argument:

```
traceplot(mod1, par="lp__")
```



In general, if the diagnostics for the log-posterior are acceptable, the diagnostics for individual parameters will be as well.

Effective sample size and $\hat{R}$

Two other diagnostics to consider are the effective sample size (ESS) and the Gelman and Rubin statistic.

Ideally, we would like each iteration in the Markov chain to be independent of the previous sample. Unfortunately, MCMC samples are often autocorrelated with each other.

The effective sample size (ESS) quantifies the amount of independent information in autocorrelated chains, specifically the amount of information equivalent to chains with no autocorrelation. It is calculated by dividing the total sample size N by the amount of autocorrelation:

$$ESS = \frac{N}{1 + 2\sum_{k=1}^{\infty} ACF(k)}. \quad (1)$$

The Gelman and Rubin potential scale reduction statistic ($\hat{R}$) measures the ratio of the average variance of samples within each chain to the variance of the pooled samples across chains. If the chains are at equilibrium, the variances will be the same and $\hat{R} = 1$.

- Any value greater than one is suspect, and values above 1.1 indicate poor mixing.

We can access these statistics by using the `print` function:

```r
print(mod1)
```

```
Inference for Stan model: 04.
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
p     0.66    0.00 0.13   0.39  0.58  0.67  0.76  0.89  1517    1
lp__ -8.15    0.02 0.72 -10.25 -8.33 -7.87 -7.69 -7.64  1541    1

Samples were drawn using NUTS(diag_e) at Sun Feb  4 12:02:35 2018.
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).
```

As with the `traceplot` output, a good overall diagnostic of model adequacy is to look at the log-posterior (`lp__`) results. For both the `lp__` and `p`, the ESS and $\hat{R}$ look good.

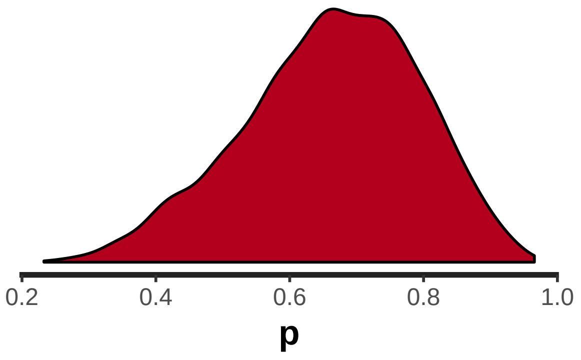
Accessing results

As with the diagnostics, we can get a numerical summary by using the `print` command. We will use the `par` argument to filter the results to return just the `p` parameter (useful when you have estimated hundreds or thousands of parameters).

```r
print(mod1, par="p")
```

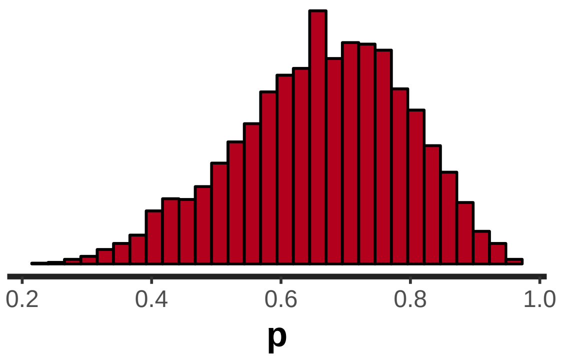
We can also plot the posterior as a density:

```r
stan_dens(mod1, par="p")
```



or a histogram:

```r
stan_dens(mod1, par="p")
```



From the plots we can see that the posterior is skewed. Thus one should be careful about using the mean as a point estimate. The median is probably more appropriate.

When reporting uncertainty intervals, most people use the 95% Highest Density Interval. However, Gelman recommends using the 50% interval for the following reasons:

1. Computational stability (it takes a lot fewer iterations to get good coverage around the 50% interval)
2. More intuitive evaluation (half the 50% intervals should contain the “true” value)
3. In applications, it is best to get a sense of where the parameter and predicted values will be, not to attempt unrealistic near-certainty.