Setting up and getting started with Stan

Luke Miratrix (prev versions from Jameson Quin)

Fall, 2016

Intro

This document will point you to instructions for installing Stan on your computer. We will also walk through a very simple example of how to fit a model in Stan, after which we'll create some basic numerical and visual summaries of your model fitting results. The main purpose of this write-up is to make sure you can get Stan installed and running on your computer. We will cover more details about Stan and its Bayesian modeling framework in upcoming lectures, section, and homework assignments.

What is Stan?

Stan is a probabilistic programming language for specifying statistical models and implementing

1) full Bayesian statistical inference via Markov chain Monte Carlo methods,
2) penalized maximum likelihood estimation with optimization, and
3) approximate Bayesian inference with variational inference.

The Stan creators' original motivation for building the programming software was to be able to apply full Bayesian inference to the sort of hierarchical generalized linear models discussed in Part II of *Data Analysis Using Regression and Multilevel/Hierarchical Models* (Gelman and Hill 2007). These HGLMs have grouped and interacted predictors at multiple levels, hierarchical covariance priors, nonconjugate coefficient priors, latent effects as in item-response models, and varying output link functions and distributions. The Stan developers didn't find any good existing general purpose inference software for handling these complicated models, so they ended up developing one themselves and released the first version of Stan on August 30, 2012. The open-source software is coded in C++ and runs on Windows, Mac OS X, and Linux. There are at least six different interfaces to this body of code: R, Python, the command line shell, MATLAB, Julia, and Stata.

Why use Stan

In this class so far, we've been learning how to extend simple linear models by adding more structure, mostly focusing on the capabilities of `lmer()` in the `lme4` R package. (We recently also introduced the `nlme` R package for fitting multilevel models with fancier correlation structures.) `lmer` can do a lot of things, but eventually you might want to fit a model that is more complicated than it can handle.

When that happens, you generally have at least two options (besides giving up). One is to find specialized packages to deal with whatever specific model we are fitting; the other is to deal with a general modeling framework that allows you to iteratively write down whatever model you wish to fit, and then the program will fit your custom model. We are going to take the second approach, because we believe it allows for greater flexibility, and avoids the need of continually tracking down packages, learning how to use them, and deciding whether to trust them or not. Stan offers such a general modeling framework.

It is important to note that Stan is a Bayesian modeling framework and we have not been Bayesian so far.

However, if we use Stan in a specific way, we essentially get uncertainty intervals (called credible intervals) that are asymptotically equivalent to what we would get from a maximum likelihood framework. So we can use Stan output similar to how we use the output of `lmer`.

In this class so far, we've been learning how to extend simple linear models by adding more structure, mostly focusing on the capabilities of "lmer" in R. Lmer can do a lot of things, but eventually you will want to fit a model that is more complicated than it can handle.

When that happens, you generally have two options

This avenue is the modeling framework called Stan. It is important to note, however, that Stan is a bayesian modeling framework and we have not been bayesian so far.

However, if we use Stan in a specific way, we essentially get uncertainty intervals that are asymptotically equivalent to what we would get from a maximum likelihood framework. So we can use Stan output just like we use the output of lmer. (Check with Peng for references)

How do you use Stan?

1. Write down a model, with parameters, as we've already learned to do.
2. Specify prior distributions for these parameters. This is Bayesian thinking; the prior combines with the likelihood to give posterior credible intervals for your parameters or for any other quantifiable estimand.
3. Run Stan in one of 3 different ways, described below.

The three different things you can ask Stan to do:

- **MCMC (Markov chain Monte Carlo)**. This allows you to sample from the full posterior distributions of your parameters. When an MCMC procedure works well, it can tell you a lot about the underlying correlation structure of your parameters and data, and allows you to get realistic credible intervals for any estimand you might write down for your model. The downside of MCMC is that it can be very computationally expensive. Also, in some cases it is finicky when your model has parameters whose distributions are hard to explore such that you don't get good "mixing" between your Markov chains.
- **Optimization**. Stan supports optimization-based inference for models. Given a posterior distribution, Stan can find the posterior mode — the value at which the posterior distribution of the parameter is maximized — which is the Bayesian version of the MLE. The posterior mode is sometimes also called the maximum a posterior (MAP) estimate. This optimization procedure is relatively fast, but only gives you the simplest information about the underlying correlation structure of your parameters. Sometimes there may be problems with finding only a local maximum rather than a global maximum.
- **Variational inference**. This is a kind of compromise between MCMC and optimization in terms of speed and faithfulness to the underlying distribution of parameters. It uses an EM-like method to find a simplified approximation of the underlying distribution. It can be a bit better at accounting for all sources of uncertainty than MCMC is, but still is unable to really probe the more complex aspects of the distribution of parameters.

In this class, we focus on the first of the three above methods. In broader statistical practice, MCMC is probably the most popular use of Stan. If you have a fast enough computer, it is really the gold standard. As for variational inference, it has its supporters, but as a compromise method, it rarely offers particular advantages that you can't get from one or the other of the first two methods.

Installing RStan

RStan is the R Project interface for Stan, and is what we will be using for this course. RStan is coded in C++ and runs under Windows, Mac OS X, and Linux, and is open source licensed under the GNU Public License, version 3. Follow the instructions on the RStan Quick Start Guide to download and install RStan on your computer. Since you already have R, you should be able to skip the Prerequisites section.

NOTE: The Stan creators recommend installing RStan through RStudio. If this doesn't work you can install the program in R (version 3.2.2) via the R GUI. After completing this installation, you should be able to load the `stan` package in RStudio.

```
# Run this once to install everything.
install.packages("rstan", dependencies = TRUE)

# In your files you then load the installed package:
library(rstan)
# Read startup messages and decide if you want to follow the recommendations
```

Stan Documentation

Go to the Stan documentation webpage to download the Stan User's Guide and Reference Manual. We *highly* recommend you carefully read through relevant parts of the manual as you begin to learn Stan. Otherwise, you will almost certainly be lost. Like other R commands, you can type `?stan_model` in the R console to quickly look up help documentation for `stan_model`. The Stan webpage also has links to example code, video tutorials, and research papers. (Warning: Some of the links are broken. Stan is still a very new software, so while it has come a long way since its initial release in 2012, it is still actively undergoing revisions and improvements.)

Also see this Library of sample models from the Gelman and Hill textbook.

Example: Running a simple linear regression in Stan

This is material from Chap 5 of Stan User's Guide and Reference Manual.

Let's start with a very simple model: simple linear regression. In practice, we wouldn't go through the trouble of using Stan to fit such a model, but we proceed for illustration purposes. We're familiar with the High School and Beyond data set, so let's use that dataset and model math achievement on SES.

Stan requires modeling files that specify the model we are going to fit. For our first model, please look through the `model1code.stan` file accompanying this tutorial. You can open '.stan' files inside RStudio.

To load stan in R, simply type

```
library(rstan)    # For running Stan
options(digits=4) # Avoid excessive precision in displayed values
```

Prepare data structure for Stan

We need to make the data structure that our model will use. In the "data" block of `model1code.stan` we defined three variables:

- nstudents (an integer)
- mathach (real vector)
- ses (real vector)

Thus, we need to feed into Stan a list containing values for these three variables. We do that as follows:

```
# Load data into R as usual
hsstud <- read.csv( "hsb1.csv", header=TRUE )

# Create data list we will feed into Stan
```

```
model1dat <- list(
  nstudents = nrow(hsstud),
  ses = hsstud$ses,
  mathach = hsstud$mathach
)
```

Run Stan to build and fit our model

We next tell Stan to make our model from the model file. This can take a moment.

```
model1stan <- stan_model(file="model1code.stan")
```

We then fit the model to our data using MCMC to get full posterior distributions for each of our model parameters. This can take a long time for complex models, and it will print out a bunch of stuff.

```
model1mcmc <- sampling(model1stan, data = model1dat, chains=4, iter=1000)
```

When you run this, it will print out a lot of stuff.

Two important attributes you need to specify for an MCMC procedure are the number of Markov chains you want to run, and the number of iterations for each chain. Type `?sampling` in the R console to see details

Look at MCMC results

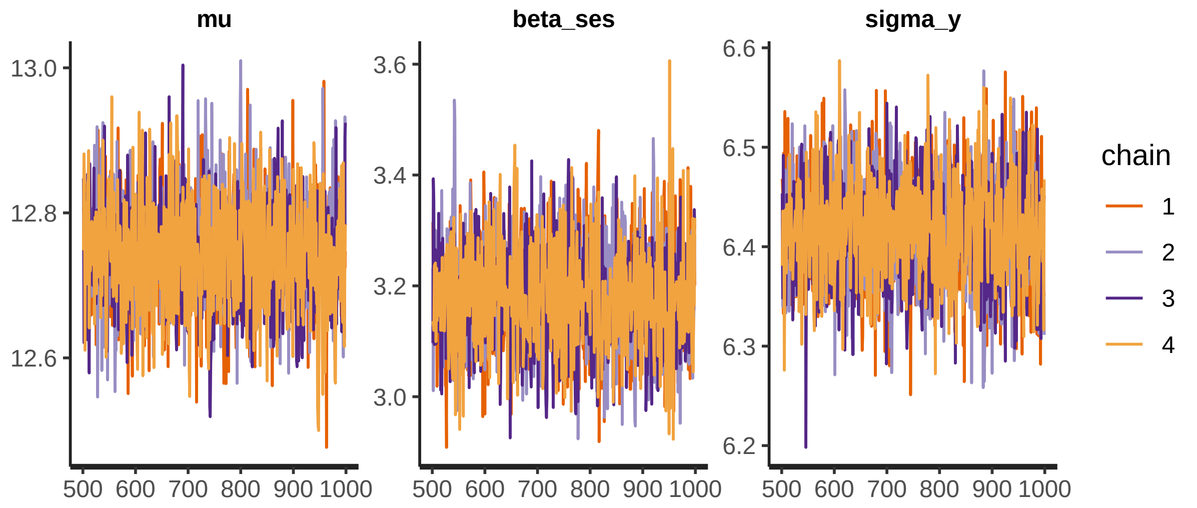
```
model1mcmc
```

```
## Inference for Stan model: model1code.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##                mean se_mean   sd      2.5%       25%       50%       75%
## mu            12.75    0.00 0.07     12.60     12.70     12.75     12.79
## beta_ses       3.18    0.00 0.09      3.00      3.12      3.18      3.24
## sigma_y        6.41    0.00 0.05      6.31      6.38      6.41      6.45
## lp__      -16946.07    0.04 1.24 -16949.01 -16946.60 -16945.74 -16945.19
##               97.5% n_eff Rhat
## mu            12.89  2043    1
## beta_ses       3.37  2324    1
## sigma_y        6.52  2567    1
## lp__      -16944.71   995    1
##
## Samples were drawn using NUTS(diag_e) at Wed Dec  4 14:47:28 2019.
## 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).
```

The `lp__` is the log posterior density, up to a constant.

Traceplots give us an idea of how well our chains are 'mixing' (i.e. Do the chains cross?), and whether or not we've reached convergence (i.e. Have each of the chains for a parameter settled down to fluctuating around a certain value?). We get the traceplots like this:

```
traceplot(model1mcmc)
```



Messy noise is what you want to see.

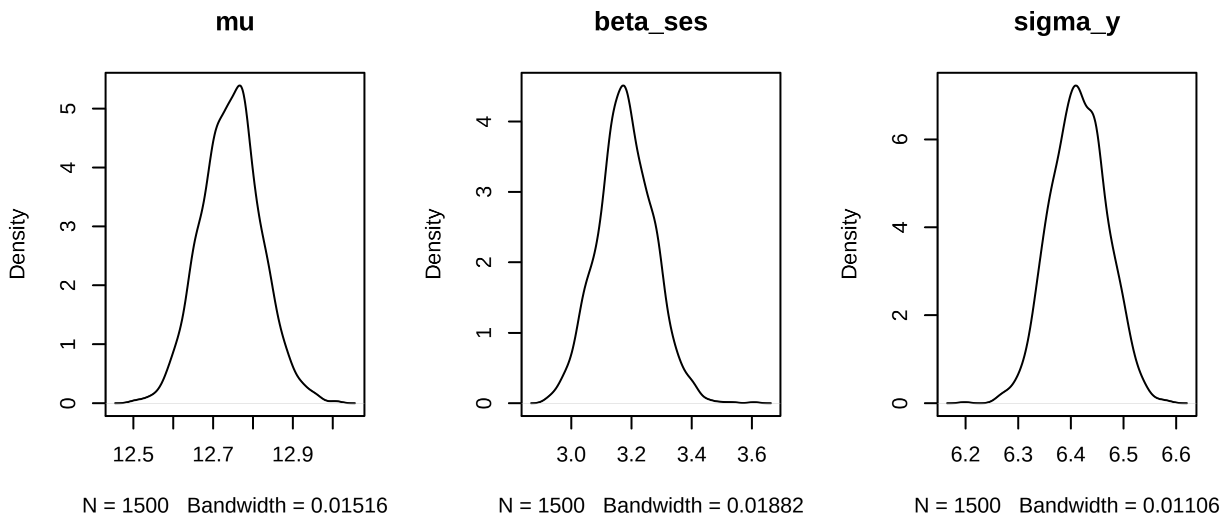
We can extract the values out of the "stanfit" object that Stan returned and then work with it like we do for regular dataframes:

```
model1samps_with_warmup_draws <- as.data.frame(model1mcmc)

# Exclude the MCMC warm-up draws, which is set in sampling() by default
# to be the first iter/2 draws
model1samps <- model1samps_with_warmup_draws[-c(1:500), ]
str(model1samps)
```

```
## 'data.frame':    1500 obs. of  4 variables:
##  $ mu      : num  12.8 12.7 12.7 12.8 12.7 ...
##  $ beta_ses: num  3.23 3.01 3.24 3.17 3.16 ...
##  $ sigma_y : num  6.43 6.34 6.37 6.47 6.36 ...
##  $ lp__    : num  -16945 -16947 -16945 -16945 -16945 ...
```

```
# Plot the posterior distributions of the model parameters
par(mfrow=c(1,3))
for(param in 1:3){
  plot(density(model1samps[ ,param]), main=names(model1samps)[param])
}
```



These look normal-ish. They show us what values of the parameters are most likely, given the data and model.

In Bayesian inference we calculate Credible intervals intead of confidence intervals. For large data, they are basically the same thing. Here we calculate 90% credible intervals for each parameter. These are simply the percentiles of each parameter's MCMC samples:

```r
for(param in 1:3){
  cat(paste(names(model1samps)[param], ":"),
      quantile(model1samps[ ,param], probs=c(0.05, 0.95) ), '\n')
}
```

```
## mu : 12.62 12.87
## beta_ses : 3.03 3.331
## sigma_y : 6.33 6.501
```

Something to consider: can you write more efficient code using the plyr package to generate both numerical and visual summaries of your MCMC samples?

Exercises you should try

- Go back to `model1codeMin` and include the `female` indicator from `hsstud` as a predictor in your linear regression model. You will need to add code to the data block, parameters block, and model block of `model1codeMin`. You will also need to add the vector of `female` indicator values to your `model1dat` list.
- Use Stan's MCMC method to fit your multiple linear regression from above. Use 3 chains and 1000 iterations per chain. Plot the posterior distributions of your four model parameters, `mu`, `beta_ses`, `beta_female`, and `sigma_y`. Calculate the posterior median (50 percentile) for each of these parameters.
- Hint: see the link to some Stan example code.

A discussion of priors

A core aspect of Bayesian modeling is you have to state some "prior distributions" for your parameters, which amount to beliefs about to what values your parameters might have a priori, i.e. before you see any data. These beliefs can influence your final estimates of your parameters, which gets some people very upset. Four kinds of priors people talk about for dealing with this issue are called "uninformative", "minimally informative", "weakly informative", and "informative" (or "strongly informative").

People rarely want to use "strongly informative" priors, because the ideal is that "data speaks". That is, analogously to partial pooling, if you have data saying this value is located here, you believe your data. If you don't have data, you turn to similar units or the prior to decide what the value is likely to be.

"Uninformative priors" can refer to prior distributions, or "improper" things that aren't actually distributions but nonetheless act like priors. In many cases uninformative priors turn bayesian estimation into likelihood estimation exactly. One class of uninformative prior is the Jeffreys prior. These have several problems for our purposes. First, because they usually aren't actually proper probability distributions, in order to use them with Stan you would have to use some programming hacks that don't always work. Second, even if you could use them, they tend to not scale nicely to scenarios with many different associated parameters. So we'll avoid Jeffreys priors.

"Minimally informative" priors refers to priors which are very close to being Jeffreys priors, but which are still proper distributions, and which are often chosen to have convenient mathematical properties such as being "conjugate" to the model's data distributions.

As an example: for a mean parameter, the Jeffreys prior would commonly be the flat prior, which assigns the same “probability” to the parameter being in any real interval of the same size. We have to use scare quotes on the word “probability” there, because there is a problem with dividing by infinity; with the flat prior, the probability of the parameter being in any actual interval would work out to be exactly zero. In order to create a minimally informative prior to approximate that flat prior, we could use a Normal distribution with a very high standard deviation. If the standard deviation is a million, while the parameter values we expect are nowhere near that big, the prior distribution is for all intents and purposes flat over the area we care about.

Unfortunately, minimally informative priors can still lead to Stan having difficulty in the actual estimation process. This is similar to the flat likelihood surface issues we’ve been seeing before with failure to converge.

With this framework, however, we can impose weakly informative priors, i.e. we state that parameters are unlikely to be any sort of crazy value. This stabilizes the process and makes getting things to converge much easier.

This also has the happy benefit of pulling our model away from crazy combinations of perimeter values that would seem to give relatively good likelihood to your data. By ruling out the extreme values, these combinations cannot occur which means your estimation will tend to produce estimates of “sane” or “plausible” values.

Doing this ruins the frequentist properties of your interval estimates, which means you’re technically not allowed to call them “confidence” intervals any more. So you call them “credible intervals”. But if you’ve done a good job with your priors, they should be pretty close to what the confidence intervals would have been, and asymptotically they will be the same. In fact, insofar as they’re different from the confidence intervals, and insofar as you believe that the values your priors say are unlikely are actually unlikely, credible intervals could be better than confidence intervals in practice.

The .stan file

For reference the stan file is this:

```
// This is Stan code.
// It is a simple OLS model for the regression of
// math achievement on ses.
// It is for the "getting started with stan" tutorial and miniassignment.
//
// Each complete line of code ends with a semicolon.
// As for the rest of the structure, it's best to explain by example:
// Also, see section 27.2 of Stan manual for overview of Stan's program blocks

// Define variables in data:
data {

    int<lower=0> nstudents;  // Total number of students, constrained to be non-negative integers

    // student covariates
    vector[nstudents] ses;   // Define ses as a vector of length nstudents

    // outcome
    vector[nstudents] mathach;   // Define mathach as a vector of length nstudents
}

// Define parameters that will be estimated:
parameters {
    // Population intercept
    real mu;   // 'real' specifies that mu takes on real-valued numbers

    // Population slope for SES
    real beta_ses;

    // Error term
    real<lower=0, upper=100> sigma_y;  // Specify sigma_y to be a non-negative
    // real number less than or equal to 100
}

transformed parameters  {
    // In some cases, it can be convenient to calculate derived values, such as
    // predicted means or residuals, here. This can help make the code in the 'model'
    // section below simpler. However, there is a tradeoff; any variables you create here
    // will show up in the raw output of various Stan functions, so you'd probably have to
    // filter it out.
}

model {
    // Priors for parameters
    // The default in Stan is to provide uniform (or 'flat') priors on parameters
    // over their legal values as determined by their declared constraints.
    // A parameter declared without constraints is thus given a uniform prior on
    // (-inf, inf) by default, whereas a scale parameter declared with a lower bound
    // of zero gets an improper uniform prior on (0, inf).

    // Here, we'll use default uniform priors for mu and beta_ses,
    // but placing a uniform(-inf, inf) prior on the SD is usually a bad idea
    // so we've constrained it to have a finite support in the next line:

    sigma_y ~ uniform(0, 100) ;
    // As stated on p. 46 of the Stan manual, it's important
    // that the support and constraints of parameters match.
    // See the definition of the sigma_y parameter above under 'parameters'


    // Finally, our simple linear regression model
    mathach ~ normal(mu + ses * beta_ses, sigma_y);
    // This says that mathach_i is Normally distributed with mean (mu + ses_i*beta_ses) and
    // variance sigma_y^2
    // See Chap 41 of Stan manual for parameterizations of different distributions.
}
```