Introduction to Stan

Cameron Bracken
University of Colorado Boulder

February 2015

What is Stan?

"A probabilistic programming language implementing full ***Bayesian statistical inference with MCMC sampling*** *(NUTS, HMC) and penalized maximum likelihood estimation with Optimization (L-BFGS)"*

"Stanislaw Ulam, namesake of Stan and co-inventor Monte Carlo methods shown here holding the Fermiac, Enrico Fermi's physical Monte Carlo simulator for neutron diffusion." (image from the Stan manual)

Bayesian Statistics

By Bayesian data analysis, we mean practical methods for making inferences from data using probability models for quantities we observe and about which we wish to learn.

The essential characteristic of Bayesian methods is their explict use of probability for quantifying uncertainty in inferences based on statistical analysis.

[Gelman et al., Bayesian Data Analysis, 3rd edition, 2013]

Background on Bayesian Statistics

From Bayes' rule, supposing the data is fixed (observed):

$$
\begin{aligned}
p(\theta|y) = \frac{p(y,\theta)}{p(y)} &= \frac{p(y|\theta)p(\theta)}{p(y)} \\
&= \frac{p(y|\theta)p(\theta)}{\int p(y,\theta)d\theta} \\
&= \frac{p(y|\theta)p(\theta)}{\int p(y|\theta)p(\theta)d\theta} \\
p(\theta|y) &\propto p(y|\theta)p(\theta) = p(y,\theta)
\end{aligned}
$$

- Everyone: Model data as random
- Bayesians: Model parameters as random

Background on Bayesian Statistics

From Bayes' rule:

$$\underbrace{p(\theta|y,x)}_{\text{Posterior}} \propto \underbrace{p(y|\theta,x)}_{\text{Liklihood}}\,\underbrace{p(\theta,x)}_{\text{Prior}}$$

- θ Parameters
- y Dependent data (response)
- x Independent data (covariates/predictors/constants)

Posterior: The answer, probability distributions of parameters

Liklihood: A computable function of the parameters, model specific

Prior: "Initial guess", incorporates existing knowledge of the system

The key to building Bayesian models is specifying the likelihood function, same as frequentist.

Monte Carlo Markov Chain (MCMC) in a nutshell

- We want to generate random draws from a target distribution (the posterior). We then identify a way to construct a 'nice' markov chain such that its equilibrium probability distribution is our target distribution.
- If we can construct such a chain then we arbitrarily start from some point and iterate the markov chain many times (like how we forecasted the weather n times). Eventually, the draws we generate would appear as if they are coming from our target distribution.
- There are several ways to construct 'nice' markov chains (e.g., gibbs sampler, Metropolis-Hastings algorithm).

(explanation from Cross Validated)
- MCMC is really a way to solve integrals that are impossible to solve analytically.

What does stan do?

- Samples from the posterior distribution (if your model is specified correctly)
- "Fits" bayesian models
- Empowers you to write your own Bayesian models, it's much easier than you think!

No U-Turn Sampler

Automatic Step Size and Number Adaptation

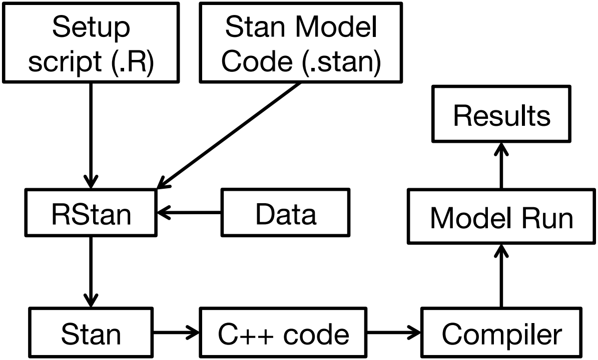
Why Stan?

There are tons of other “black-box” MCMC samplers out there (BUGS, JAGS, Church, PyMC, many many more, `http://probabilistic-programming.org/wiki/Home`)

- Stan is open source
- Built to be fast (about 10 times faster then BUGS according Gelman)
- “Stan can handle problems that choke BUGS and JAGS” – Andrew Gelman

Using Stan

Stan is a library with a number of interfaces, we will use the R interface called RStan.



Example 1 – Fit Normal Distribution – Model code

Download from: `http://bechtel.colorado.edu/~bracken/stan/example_models.zip`
Example files in `1-normal`: `normal.stan/normal.R`

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

Example 1 – Fit Normal Distribution – RStan code

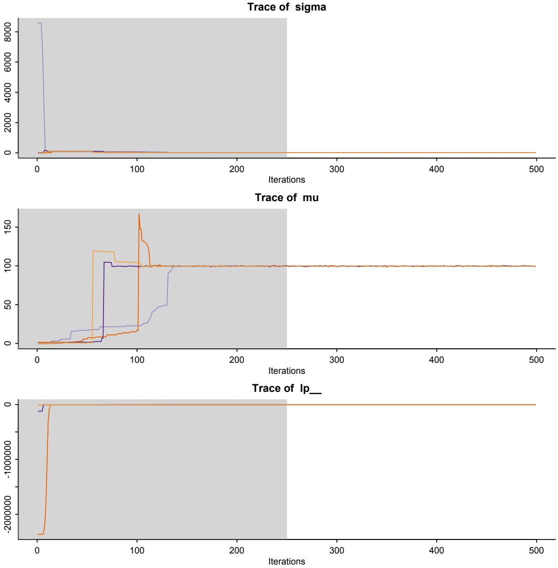
```
library(RStan)
model_file = 'normal.stan'
iterations = 500
N = 1000
mu = 100
sigma = 10
y = rnorm(N, mu,sigma) # simulate data

stan_data = list(N=N, y=y) # data passed to stan
    # set up the model
stan_model = stan(model_file, data = stan_data, chains = 0)
stanfit = stan(fit = stan_model, data = stan_data,
    iter=iterations) # run the model
print(stanfit,digits=2)
```

Diagnostics - Text output

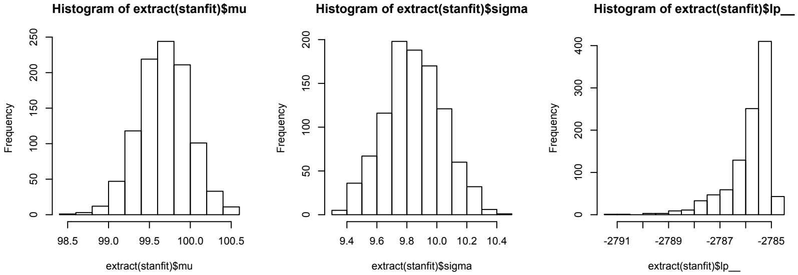
```
Inference for Stan model: normal.
4 chains, each with iter=500; warmup=250; thin=1;
post-warmup draws per chain=250, total post-warmup draws=1000.

          mean se_mean   sd     2.5%      25%      50%      75%    97.5% n_eff Rhat
sigma     9.84    0.01 0.19     9.47     9.72     9.84     9.98    10.23   366 1.01
mu       99.68    0.01 0.31    99.08    99.48    99.69    99.88   100.30   617 1.00
lp__  -2785.83    0.04 0.87 -2788.07 -2786.19 -2785.58 -2785.19 -2784.97   395 1.00

Samples were drawn using NUTS(diag_e) at Thu Feb 19 21:04:56 2015.
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).
```

- thin, warmup
- n_{eff}
- $\hat{R}$
- `lp__`

Diagnostics - Traceplots



Diagnostics - Posterior plots



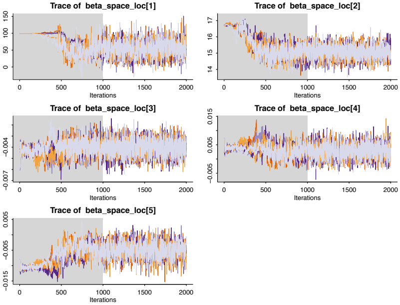
Diagnostics - shinyStan

Old package (don't use it):
`https://github.com/jgabry/SHINYstan`
Changing to: `https://github.com/shinyStan/shinyStan`

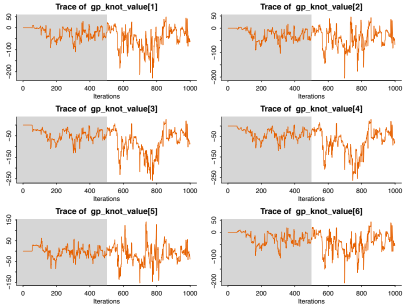
```
library("shinyStan")
launch_shinystan(stanfit)
```

Convergence - Traceplots

Converged:

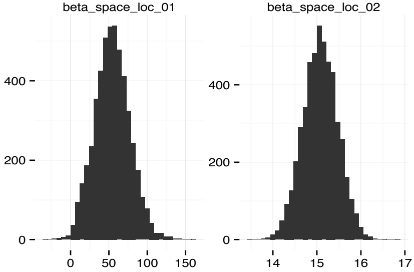


Not Converged:

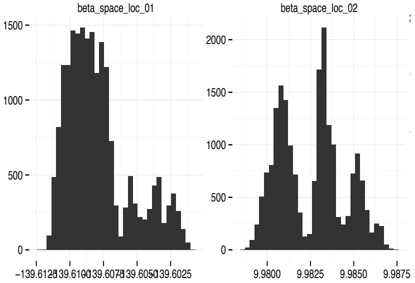


Convergence - Posterior plots

Converged:



Not Converged:



Example 2 – Fit normal distribution (fancier)

$$y_n \sim \text{Normal}(\mu, \sigma)$$

The likelihood of observing a normally distributed data value is the normal density of that point given the parameter values.

Example files in `2-normal`: `normal2.stan`/`normal2.R`

- priors
- initial values

Example 3 – fit GEV distribution

Generalized extreme value distribution

$$y \sim GEV(\mu, \sigma, \xi)$$

μ: location, σ: scale, ξ: shape

Example files in `3-gev`: `gev.stan/gev.R`

- Random seed
- constraints
- variable constraints
- Initial values

Example 4 - multiple linear regression

$$y_n = \alpha + \beta x_n + \varepsilon_n$$

where $\varepsilon_n \sim \text{Normal}(0, \sigma)$, which can be written:

$$y_n - (\alpha + \beta x_n) \sim \text{Normal}(0, \sigma)$$

$$y_n \sim \text{Normal}(\alpha + \beta x_n, \sigma)$$

The likelihood of observing a given y_n is just the normal density with mean $\alpha + \beta x_n$ and standard deviation σ.

Example files in `4-multiple-linear-regression`:
`mlr.stan`/`mlr.R`

Example 5 - Binomial regression (glm)

$$y_n = \alpha + \beta g^{-1}(x_n) + \varepsilon_n$$

$$y_n \sim \text{Normal}(g^{-1}(x_n), 1)$$

Example files in `5-binomial-logit`:
`binomial-logit.stan`/`binomial-logit.R`

- Hierarchical

Stan tips and tricks

#1 tip: Read the Manual! It is excellent

Other things we didn't really talk about:

- Local variables in the model block, can be used to store intermediate results
- Matrices vs arrays, Column vector vs row vector
- Constrained data types
- Transformed parameters
- Functions
- Logical operations/Other types of looping
- Elementwise operators
- Built-in functions
- Print statements
- Missing data
- Prediction
- Discrete variables

Stan tips and tricks

No need to truncate priors, do that in the parameter bounds

- **BAD**: setting constraints on parameters but using a prior with other constraints

```
parameters{
    real alpha; //implies no constraints
}
model{
    alpha ~ uniform(0,1);
}
```

- **GOOD**::

```
parameters{
    real <lower=0,upper=1> alpha;
}
model{
    #alpha ~ uniform(0,1); // default uniform priors
}
```

Stan tips and tricks

- No need to use conjugate priors
- Unlike BUGS (or other Gibbs based samplers), avoid super vauge priors if you can, i.e. `inv_gamma(0.1,0.1)`
- When in doubt, use a normal prior, or google it
- The Stan mailing list is very active

Speeding up Stan models

- Avoid repeated operations

```
// 1/alpha is repeated
for(n in 1:N)
    y[n] ~ exponential(1/alpha * x[n]);
```

- Vectorization is always faster

```
// not vectorized
for(n in 1:N)
    y[n] ~ normal(beta0 + beta1 * x[n], sigma);
//vectorized
y ~ normal(beta0 + beta1 * x, sigma);
```

- Priors: More informative the better (think better initial conditions), use MLE to get initial estimates
- Parallization: can run multiple chains if you have multiple cores, but each chain is still serial
- More advanced: Access `increment_log_prob` directly