Stan Workshop

Peter Gao and Serge Aleshin-Guendel

November 21 2017

Set Up

To run a model in Stan in R, you need two files: a **.stan** file and a **.R file**. The **.stan** file declares the model, the **.R file** performs inference on the declared model.

Beta-Binomial Model

Suppose that

$$Y \mid \theta \sim \mathrm{Binomial}(N, \theta)$$

where $\theta \sim \mathrm{Beta}(1, 1)$.

[1] Example courtesy of Aki Vehtari.

binom.stan

```
data {
  int<lower=0> N;
  int<lower=0> y;
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);
  y ~ binomial(N,theta);
}
```

[1] Example courtesy of Aki Vehtari.

binom.R

```
library(rstan)
set.seed(1121)
N = 50
y <- rbinom(1, N, .35) # y = 18
results <- stan(file="binom.stan",
                data = list(N = N, y = y),
                iter = 1000, chains = 1)
```

[1] Example courtesy of Aki Vehtari.

Results

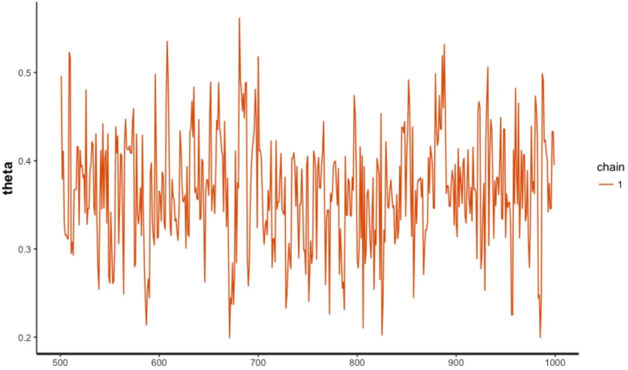
```
results
```

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

        mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
theta   0.37    0.00 0.06   0.24   0.32   0.36   0.41   0.49   195    1
lp__  -34.62    0.05 0.68 -36.60 -34.76 -34.35 -34.18 -34.14   206    1

Samples were drawn using NUTS(diag_e) at Sun Nov 19 09:41:06 2017.
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).
```

[1] Example courtesy of Aki Vehtari.

Results

```
traceplot(results)
```



[1] Example courtesy of Aki Vehtari.

Gaussian Linear Regression Model

Example courtesy of Aki Vehtari.
Suppose that

$$
\begin{aligned}
Y \mid X, \alpha, \beta &\sim \mathrm{Norm}(\alpha + \beta X, \sigma) \\
\alpha &\sim \mathrm{Norm}(\mu_\alpha, \sigma_\alpha) \\
\beta &\sim \mathrm{Norm}(\mu_\beta, \sigma_\beta).
\end{aligned}
$$

More about σ in a second!

lin.R

```
d_lin <- c(list(
  N = N,
  x = x,
  xpred = xpred,
  y = y,
  pmualpha = 0,
  psalpha = 1,
  pmubeta = 0,
  psbeta = 1))
fit_lin <- stan(file = ’lin.stan’, data = d_lin)
```

lin.stan

```
data {
  int<lower=0> N; // number of data points
  vector[N] x; //
  vector[N] y; //
  real xpred; // input location for prediction
  real pmualpha; // prior mean for alpha
  real psalpha;  // prior std for alpha
  real pmubeta;  // prior mean for beta
  real psbeta;   // prior std for beta
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
transformed parameters {
  vector[N] mu;
  mu = alpha + beta*x;
}
```

lin.stan

```
model {
  alpha ~ normal(pmualpha, psalpha);
  beta ~ normal(pmubeta, psbeta);
  // No prior defined on sigma!
  // Default prior is uniform (possibly improper)
  y ~ normal(mu, sigma);
}
generated quantities {
  real ypred;
  vector[N] log_lik;
  ypred = normal_rng(alpha + beta*xpred, sigma);
  for (i in 1:N)
    log_lik[i] = normal_lpdf(y[i] | mu[i], sigma);
}
```

ICAR Model

Given observations about n regions, we model the spatial dependence between regions n_i and n_j conditionally via a random vector $\phi = (\phi_1, \ldots, \phi_n)^T$. We can represent the spatial relationships between our n regions with an adjacency matrix W.

[1] Example courtesy of Mitzi Morris.

ICAR Model

In an ICAR model, we specify the conditional distributions of each ϕ_i in terms of a Markov random field:

$$p(\phi_i \mid \phi_j; j \neq i, \tau_i^{-1}) = N\left(\frac{\sum_{i\sim j}\phi_j}{d_{i,i}}, \frac{1}{d_{i,i}}\tau_i\right)$$

where $i \sim j$ if i is a neighbor of j, and $d_{i,i}$ is the number of neighbors for region n_i. Thus, the mean for ϕ_i is the average of its neighbors.

This gives us the joint distribution

$$\phi \sim N_n(\mathbf{0}, [\tau(D - W)]^{-1})$$

where $D = \mathrm{diag}(d_{1,1}, \ldots, d_{n,n})$.

[1] Example courtesy of Mitzi Morris.

ICAR Model

Now, we add an ICAR component to a Stan model. It turns out (see Morris for derivation) that we can rewrite the log probability density of ϕ in terms of pairwise differences:

$$\log p(\phi) = -\frac{1}{2}\left(\sum_{i\sim j}(\phi_i - \phi_j)^2\right) + \text{const.}$$

This will allow us to store the adjacency matrix (generally sparse) in a less computationally way.

[1] Example courtesy of Mitzi Morris.

simple_iar.stan

```
data {
  int<lower=0> N;
  int<lower=0> N_edges;
  int<lower=1, upper=N> node1[N_edges];
  int<lower=1, upper=N> node2[N_edges];
   // node1[i] adjacent to node2[i]
   // and node1[i] < node2[i]
}
parameters {
  vector[N-1] phi_raw_std;
}
transformed parameters {
  vector[N] phi;
  phi[1:(N - 1)] = phi_raw_std;
  phi[N] = -sum(phi_raw_std);
}
model {
  target += -0.5 * dot_self(phi[node1] - phi[node2]);
}
```

[1] Example courtesy of Mitzi Morris.

fit_simple_iar.R

```
library(rstan)
options(mc.cores = parallel::detectCores())

source("mungeCARdata4stan.R")
source("scotland_data.R")

iter = 10000;
mfile = "simple_iar.stan";
ofile = "simple_iar_stan_010K_iters.txt";

nbs = mungeCARdata4stan(data$adj, data$num);
N = data$N;
node1 = nbs$node1;
node2 = nbs$node2;
N_edges = nbs$N_edges;

fit_stan = stan(mfile, data=list(N,N_edges,node1,node2), iter=iter);
```

[1] Example courtesy of Mitzi Morris.

Results

```
stanfit = extract(fit_stan);
```

```
> # cov neighbors
> cov(stanfit$phi[,6],stanfit$phi[,8]);
[1] 2.716187
> cov(stanfit$phi[,6],stanfit$phi[,3]);
[1] 1.761167
> cov(stanfit$phi[,10],stanfit$phi[,22]);
[1] 0.5112239
> # cov non-neighbors
> cov(stanfit$phi[,6],stanfit$phi[,54]);
[1] -0.2371438
> cov(stanfit$phi[,8],stanfit$phi[,54]);
[1] -0.2612349
> cov(stanfit$phi[,2],stanfit$phi[,55]);
[1] -0.1943578
```

[1] Example courtesy of Mitzi Morris.

Gaussian Process Regression Model

Will be following a tutorial by Rob Trangucci.
Suppose we're in the regression setting again. We'll consider the univariate case for simplicity. We'll place a Gaussian Process prior on the regression function, which is parameterized by a mean function $\mu(X)$ and a positive semi-definite kernel function $k_\theta(X)$ parameterized by θ:

$$\begin{aligned}
Y \mid X, f, \sigma &\sim \mathrm{Norm}(f(X), \sigma) \\
f(X) &\sim \mathrm{GP}(\mu(X), k_\theta(X)) \\
\theta &\sim p(\theta) \\
\sigma &\sim p(\sigma).
\end{aligned}$$

Finite Dimensional Generative Model of GP

The key property of a GP is that any finite sample from a GP is jointly multivariate normal, with the mean vector and covariance matrix being the finite sample realizations of the mean and kernel function. Thus for a finite sample of N points, we can specify our model as

$$\begin{aligned}
Y_i \mid X_i, f, \sigma &\sim \mathrm{Norm}(f_i, \sigma) \; \forall i \in \{1, \cdots, N\} \\
f &\sim \mathrm{MVN}(0, K_\theta(X)) \\
\theta &\sim p(\theta) \\
\sigma &\sim p(\sigma),
\end{aligned}$$

where $K_\theta(X)_{ij} = k_\theta(x_i, x_j)$, and the mean function is taken to be 0 (common in practice).

Choice of Kernel

We'll use the exponentiated quadratic/squared exponential/Gaussian/radial basis function kernel:

$$k_\theta(x_i, x_j) = \alpha^2 \exp\left(\frac{-(x_i - x_j)^2}{2\ell^2}\right),$$

where α^2 is the marginal variance of the process f, and ℓ is the length scale of the process.
Currently one of the only kernels already implemented in Stan (can implement your own though).

Cholesky Decomposition/ Non-Centered MVN Parameterization

Since $K_\theta(X)$ is a covariance matrix, there exists a Cholesky decomposition

$$LL^T = K_\theta(X),$$

where L is a lower triangular matrix. We can use this decomposition to write out the finite sample realization of f as

$$\begin{aligned}
&f \sim \mathrm{MVN}(0, K_\theta(X)) \\
&f = L\eta \\
&\eta \sim \mathrm{MVN}(0, I_N).
\end{aligned}$$

When working with a MVN, going to have to invert, find determinant, etc. Fastest way to do that in Stan is to use this decomposition.

gp_simple_latent.stan

```
data {
  int<lower=1> N;
  int<lower=1> N_pred;
  vector[N] y;
  real x[N];
  vector[N] zeros;
  real x_pred[N_pred];
}
```

gp_simple_latent.stan

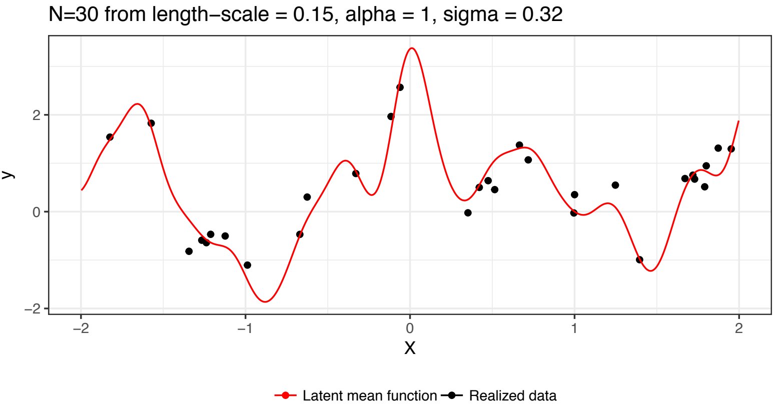
```
parameters {
  real<lower=0> length_scale;
  real<lower=0> alpha;
  real<lower=0> sigma;
  vector[N] f_eta;
}
transformed parameters {
  vector[N] f;
  {
    matrix[N, N] L_cov;
    matrix[N, N] cov;
    cov = cov_exp_quad(x, alpha, length_scale);
    for (n in 1:N)
      cov[n, n] = cov[n, n] + 1e-12; // For numerical stability
    L_cov = cholesky_decompose(cov);
    f = L_cov * f_eta;
  }
}
```

gp_simple_latent.stan

```
model {
  length_scale ~ gamma(2, 2);
  alpha ~ normal(0, 1); // Truncated normal
  sigma ~ normal(0, 1); // Truncated normal
  f_eta ~ normal(0, 1);
  y ~ normal(f, sigma);
}
generated quantities {
  vector[N_pred] f_pred;
  vector[N_pred] y_pred;

  f_pred = gp_pred_rng(x_pred, y, x, alpha,
                       length_scale, sigma);
  for (n in 1:N_pred)
    y_pred[n] = normal_rng(f_pred[n], sigma);
}
```

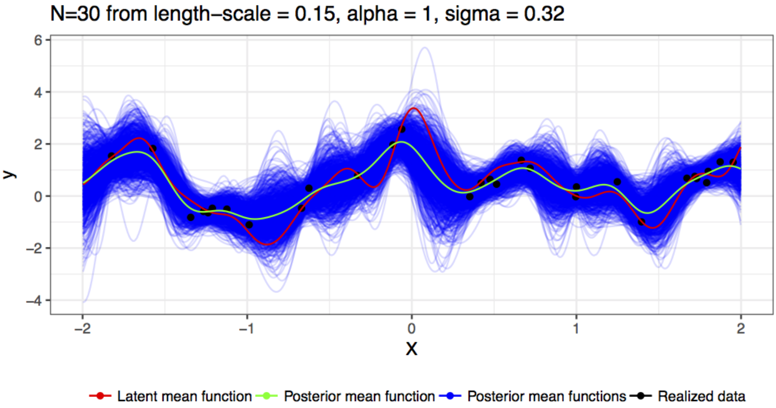
Data

Simulated from the described generative model with $\sigma = 0.32$, $\alpha = 1$, and $\ell = 0.15$:



Posterior

We arrive at the posterior. Note that “Latent mean function” is the true latent GP mean function, “Posterior mean function” is the (pointwise) mean of the posterior mean function, and “Posterior mean functions” are draws of the mean function from the posterior GP.


N=30 from length–scale = 0.15, alpha = 1, sigma = 0.32
y
x
Latent mean function
Posterior mean function
Posterior mean functions
Realized data