Linear Models with Stan

Ben Goodrich

April 06, 2022

Basic Matrix Algebra

- A vector can be a column vector (vertical) or row vector (horizontal)
- I use boldface for vectors and matrices and $\top$ for transposition
- A row vector multiplied by a column vector of the same size is a scalar, i.e.

$$\mathbf{x}^\top \boldsymbol{\beta} = \sum_{k=1}^{K} x_k \beta_k$$

- A matrix multiplied by a column vector is a column vector that is obtained by treating each row of the matrix as a row vector, i.e.

$$\boldsymbol{\mu} = \alpha + \mathbf{X}\boldsymbol{\beta}$$

Differentiating the Log Posterior Kernel

- Stan always works with log-PDFs or really log-kernels (in $\boldsymbol{\theta}$)

$$\ln f\left(\boldsymbol{\theta} \mid \mathbf{y}, \ldots\right) = \ln f\left(\boldsymbol{\theta} \mid \ldots\right) + \ln L\left(\boldsymbol{\theta}; \mathbf{y}\right) - \ln f\left(\mathbf{y} \mid \ldots\right)$$

- The gradient of the log posterior PDF is the gradient of the log-kernel

$$\boldsymbol{\nabla} \ln f\left(\boldsymbol{\theta} \mid \mathbf{y}, \ldots\right) = \boldsymbol{\nabla} \ln f\left(\boldsymbol{\theta} \mid \ldots\right) + \boldsymbol{\nabla} \ln L\left(\boldsymbol{\theta}; \mathbf{y}\right) + \mathbf{0}$$

- This gradient is basically exact, and the chain rule can be executed by a C++ compiler without the user having to compute any derivatives

Hamiltonian Monte Carlo

- Stan pairs the J "position" variables $\boldsymbol{\theta}$ with J "momentum" variables $\boldsymbol{\phi}$ and draws from the joint posterior distribution of $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$
- Since the likelihood is NOT a function of ϕ_j, the posterior distribution of ϕ_j is the same as its prior, which is normal with a "tuned" standard deviation. So, at the s-th MCMC iteration, we just draw each $\widetilde{\phi}_j$ from its normal distribution.
- Using physics, the realizations of each $\widetilde{\phi}_j$ at iteration s "push" $\boldsymbol{\theta}$ from iteration $s-1$ for a random amount of time through the parameter space whose topology is defined by the (negated) log-kernel of the posterior distribution
- Although the ODEs must be solved numerically, the integral in "time" is one-dimensional and there are very good customized numerical integrators

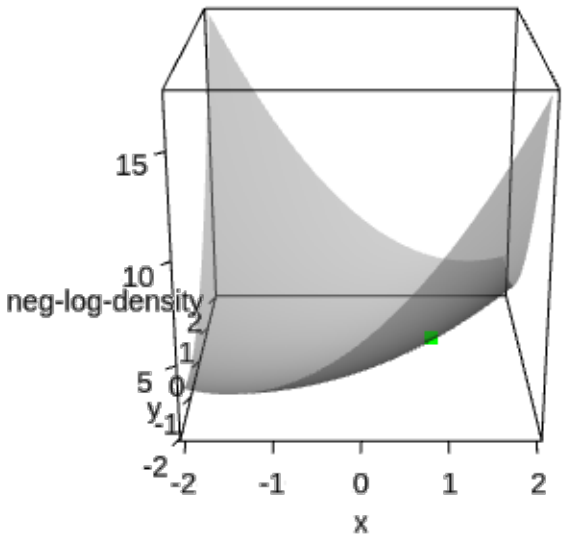
Hamiltonian Monte Carlo

- Instead of simply drawing from the posterior distribution whose PDF is $f\left(\boldsymbol{\theta}|\,\mathbf{y}\ldots\right) \propto f\left(\boldsymbol{\theta}\right) L\left(\boldsymbol{\theta};\mathbf{y}\right)$ Stan augments the "position" variables $\boldsymbol{\theta}$ with an equivalent number of "momentum" variables $\boldsymbol{\phi}$ and draws from

$$f\left(\boldsymbol{\theta}|\,\mathbf{y}\ldots\right) \propto \int_{-\infty}^{\infty} \ldots \int_{-\infty}^{\infty} \prod_{k=1}^{K} \frac{1}{\sigma_k\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\phi_k}{\sigma_k}\right)^2} f\left(\boldsymbol{\theta}\right) L\left(\boldsymbol{\theta};\mathbf{y}\right) d\phi_1 \ldots d\phi_K$$

- Since the likelihood is NOT a function of ϕ_k, the posterior distribution of ϕ_k is the same as its prior, which is normal with a "tuned" standard deviation. So, at the s-th MCMC iteration, we just draw each $\widetilde{\phi}_k$ from its normal distribution.
- Using physics, the realizations of each $\widetilde{\phi}_k$ at iteration s "push" $\boldsymbol{\theta}$ from iteration $s-1$ through the parameter space whose topology is defined by the negated log-kernel of the posterior distribution: $-\ln f\left(\boldsymbol{\theta}\right) - \ln L\left(\boldsymbol{\theta};\mathbf{y}\right)$
- See HMC.R demo and next slide

Demo of Hamiltonian Monte Carlo



Reverse | Play | Slower | Faster | Reset

1.00

No U-Turn Sampling (NUTS)

- The location of $\boldsymbol{\theta}$ moving according to Hamiltonian physics at any instant would be a valid draw from the posterior distribution
- But (in the absence of friction) $\boldsymbol{\theta}$ moves indefinitely so when do you stop?
- Hoffman and Gelman (2014) proposed stopping when there is a "U-turn" in the sense the footprints turn around and start to head in the direction they just came from. Hence, the name No U-Turn Sampling.
- After the U-Turn, one footprint is selected with probability proportional to the posterior kernel to be the realization of $\boldsymbol{\theta}$ on iteration s and the process repeates itself
- NUTS discretizes a continuous-time Hamiltonian process in order to solve a system of Ordinary Differential Equations (ODEs), which requires a stepsize that is also tuned during the warmup phase
- Video and R code

Using Stan via R

1. Write the program in a (text) .stan file w/ R-like syntax that ultimately defines a posterior log-kernel. Stan's parser, `rstan::stanc`, does two things:
   - checks that program is syntactically valid and tells you if not
   - writes a conceptually equivalent C++ source file to disk
2. C++ compiler creates a binary file from the C++ source
3. Execute the binary from R (can be concurrent with 2)
4. Analyze the resulting samples from the posterior
   - Posterior predictive checks
   - Model comparison
   - Decision

A Better Model for Vaccine Effectiveness

```
#include quantile_functions.stan
data { /* these are known and passed as a named list from R */
  int<lower = 0> n;              // number of people in clinical trial
  int<lower = 0, upper = n> y;  // number of positives among vaccinated
  real m;
  real<lower = 0> IQR;
  real<lower = -1, upper = 1> asymmetry;
  real<lower =  0, upper = 1> steepness;
}
parameters { /* these are unknowns whose posterior distribution is sought */
  real<lower = 0, upper = 1> p; // CDF of vaccine effectiveness
}
transformed parameters { /* deterministic unknowns that get stored in RAM */
  real VE = gld_qf(p, m, IQR, asymmetry, steepness); // theta = (VE - 1) / (VE - 2)
} // this function ^^^ is defined in the quantile_functions.stan file
model { /* log-kernel of Bayes' Rule that essentially returns "target" */
  target += binomial_lpmf(y | n, (VE - 1) / (VE - 2)); // log-likelihood
} // implicit: p ~ uniform(0, 1) <=> VE ~ gld(m, IQR, asymmetry, steepness)
```

Drawing from a Posterior Distribution with NUTS

```
source(file.path("..", "Week2", "GLD_helpers.R"))
a_s <- GLD_solver_LBFGS(lower_quartile = 0.15, median = 0.3, upper_quartile = 0.55,
                        other_quantile = -0.5, alpha = 0.01)
post <- stan(file.path("..", "Week2", "coronavirus.stan"),
             data = list(n = 94, y = 8, m = 0.3, IQR = 0.4,
                         asymmetry = a_s[1], steepness = a_s[2]),
             seed = 12345, control = list(adapt_delta = .95))


post
```

```
## Inference for Stan model: coronavirus.
## 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     0.86    0.00 0.01  0.84  0.85  0.86  0.86  0.87   690 1.00
## VE    0.89    0.00 0.04  0.79  0.87  0.89  0.92  0.95   740 1.00
## lp__ -4.61    0.03 0.81 -6.99 -4.76 -4.30 -4.11 -4.05   576 1.01
##
## Samples were drawn using NUTS(diag_e) at Wed Apr  6 11:48:03 2022.
## 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).
```

Warnings You Should Be Aware Of

1. Divergent Transitions: This means the tuned stepsize ended up too big relative to the curvature of the log-kernel. Increase `adapt_delta` above its default value (0.8) and / or use more informative priors
2. Hitting the maximum treedepth: This means the tuned stepsize ended up so small that it could not get all the way around the parameter space in one iteration. Increase `max_treedepth` beyond its default value of 10 but each increment will double the wall time, so only do so if you hit the max a lot
3. Bulk / Tail Effective Sample Size too low: This means the tuned stepsize ended up so small that adjacent draws have too much dependence. Increase the number of iterations or chains
4. $\widehat{R} > 1.01$: This means the chains have not converged. You could try running the chains longer, but there is probably a deeper problem.
5. Low Bayesian Fraction of Information: This means that you posterior distribution has really extreme tails. You could try running the chains longer, but there is probably a deeper problem.

Data on 2020 Trump Vote and 2022 Vaccination

```
library(readr); library(dplyr)
# https://docs.google.com/spreadsheets/d/100BFc0VppVL8CIhaNh5ZiTFGBNCnGBdYzfqISAWxln8/
Gabba <- read_csv("Gabba.csv", col_types = c("ccccddddddddddd"), skip = 1, col_names =
                    c("FIPS", "ST", "State", "County", "Trump#", "Votes#", "Trump", "Pop",
                      "Vaccinated#", "Vaccinated", "Death1", "Death2", "Death3", "Death4"))
Gabba <- filter(Gabba, Vaccinated < 100) # some data points were messed up
select(Gabba, State:Vaccinated) %>%
  glimpse # each row is a county
```

```
## Rows: 3,125
## Columns: 8
## $ State         <chr> "Alabama", "Alabama", "Alabama", "Alabama", "Alabama", "Alabama", …
## $ County        <chr> "Autauga", "Baldwin", "Barbour", "Bibb", "Blount", "Bullock", "But…
## $ `Trump#`      <dbl> 19838, 83544, 5622, 7525, 24711, 1146, 5458, 35101, 8753, 10583, 1…
## $ `Votes#`      <dbl> 27770, 109679, 10518, 9595, 27588, 4613, 9488, 50983, 15284, 12301…
## $ Trump         <dbl> 71.44, 76.17, 53.45, 78.43, 89.57, 24.84, 57.53, 68.85, 57.27, 86.…
## $ Pop           <dbl> 58805, 231767, 25223, 22293, 59134, 10357, 19051, 116441, 34772, 2…
## $ `Vaccinated#` <dbl> 24395, 112300, 11070, 7728, 18162, 5305, 7613, 52780, 10354, 7964,…
## $ Vaccinated    <dbl> 41.48, 48.45, 43.89, 34.67, 30.71, 51.22, 39.96, 45.33, 29.78, 31.…
```

Thinking About Priors

- It is usually a good idea to center all predictors so that the intercept can be interpreted as the expected outcome for a unit with “average” predictors
- What are your beliefs about the expected covid vaccination percentage as of March 2022 in a county with an average percentage of Trump voters in 2020?
- What are your beliefs about the expected covid vaccination percentage in a county with 1% more Trump voters than average?
- What are your beliefs about the error standard deviation when predicting vaccination percentage with Trump vote percentage only?
- Do it, using `source(file.path("..", "Week2", "GLD_helpers.R"))`

Prior Hyperparameters

```
m <- c(beta = -0.5, alpha = 50, sigma = 10)
r <- c(beta =  0.4, alpha = 20, sigma = 10)
a_s_beta  <- GLD_solver_bounded(bounds = c(-1, 1),  median = m[1], IQR = r[1])
a_s_alpha <- GLD_solver_bounded(bounds = c(0, 100), median = m[2], IQR = r[2])
a_s_sigma <- GLD_solver_LBFGS(lower_quartile = 5, median = 10, upper_quartile = 15,
                              other_quantile = 0, alpha = 0)
a <- c(beta = a_s_beta[1], alpha = a_s_alpha[1], sigma = a_s_sigma[1])
s <- c(beta = a_s_beta[2], alpha = a_s_alpha[2], sigma = a_s_sigma[2])
```

Primitive Object Types in Stan

- In Stan / C++, variables must declared with types
- In Stan / C++, statements are terminated with semi-colons
- Primitive scalar types: `real x;` or `int K;`
  - Unknowns cannot be `int` because no derivatives and hence no HMC
  - Can condition on integer data because no derivatives are needed
- Implicitly real `vector[K] z;` or `row_vector[K] z;`
- Implicitly real `matrix[N,K] X;` can have 1 column / row
- Arrays are just holders of any other homogenous objects
  - `real x[N]` is similar to `vector[N] x;` but lacks linear algebra functions
  - `vector[N] X[K];` and `row_vector[K] X[N]` are similar to `matrix[N,K] X;` but lack linear algebra functionality, although they have uses in loops
- Vectors and matrices cannot store integers, so instead use possibly multidimensional integer arrays `int y[N];` or `int Y[N,P];`

The `lookup` Function in rstan

- Can input the name of an R function, in which case it will try to find an analagous Stan function
- Can input a regular expression, in which case it will find matching Stan functions that match

```
lookup("^inv.*[^gf]$") # functions starting with inv but not ending with g or f
```

```
#       StanFunction  Arguments ReturnType
# 216 inv_chi_square          ~       real
# 219        inverse (matrix A)     matrix
# 220    inverse_spd (matrix A)     matrix
# 225      inv_gamma          ~       real
# 227      inv_logit      (T x)          R
# 228        inv_phi      (T x)          R
# 229       inv_sqrt      (T x)          R
# 230     inv_square      (T x)          R
# 233    inv_wishart          ~       real
```

Optional `functions` Block of .stan Programs

- Stan permits users to define and use their own functions, which is what we did with `#include quantile_functions.stan`
- If used, must be defined in a leading `functions` block
- Can only validate constraints inside user-defined functions
- Very useful for several reasons:
  - Easier to reuse across different .stan programs
  - Makes subsequent chunks of code more readable
  - Enables posteriors with Ordinary Differential Equations, algebraic equations, and integrals
  - Can be exported to R via `expose_stan_functions()`
- All functions, whether user-defined or build-in, must be called by argument position rather than by argument name, and there are no default arguments
- User-defined functions cannot have the same name as existing functions or keywords and are case-sensitive

Constrained Object Declarations in Stan

Outside of the `functions` block, any primitive object can have bounds:

- `int<lower = 1> K;` `real<lower = -1, upper = 1> rho;`
- `vector<lower = 0>[K] alpha;` and similarly for a `matrix`
- A `vector` (but not a `row_vector`) can be further specialized:
  - `unit_vector[K] x;` implies $\sum_{k=1}^{K} x_k^2 = 1$
  - `simplex[K] x;` implies $x_k \geq 0 \, \forall k$ and $\sum_{k=1}^{K} x_k = 1$
  - `ordered[K] x;` implies $x_j < x_k \, \forall j < k$
  - `positive_ordered[K] x;` implies $0 < x_j < x_k \, \forall j < k$
- A `matrix` can be specialized to enforce constraints:
  - `cov_matrix[K] Sigma;` or better `cholesky_factor_cov[K, K] L;`
  - `corr_matrix[K] Lambda;` or `cholesky_factor_corr[K] C;`

"Required" **data** Block of .stan Programs

- All knowns passed from R to Stan as a NAMED list, such as outcomes $(\mathbf{y})$, covariates $(\mathbf{X})$, constants (K), and / or known hyperparameters
- Basically, everything posterior distribution conditions on
- Can have comments in C++ style (`//` or `/* ... */`)
- Whitespace is essentially irrelevant, except after keywords

```
data {
  int<lower = 0> N; // number of observations
  int<lower = 0> K; // number of predictors
  matrix[N, K] X;   // matrix of predictors
  vector[N] y;      // outcomes
  int<lower = 0, upper = 1> prior_only;   // ignore data?
  vector[K + 2] m;                        // prior medians
  vector<lower = 0>[K + 2] r;             // prior IQRs
  vector<lower = -1, upper = 1>[K + 2] a; // prior asymmetry
  vector<lower =  0, upper = 1>[K + 2] s; // prior steepness
}
```

“Required” `parameters` Block of .stan Programs

- Declare exogenous unknowns whose posterior distribution is sought
- Cannot declare any integer parameters, only real parameters
- Must specify the parameter space but `lower` and `upper` bounds are implicitly $\pm\infty$ if unspecified

```
parameters {
  vector<lower = 0, upper = 1>[K + 2] p;  // CDF values
}
```

- The change-of-variables adjustment due to the transformation from an unconstrained parameter space to the constrained space is handled automatically and added to `target`

Optional `transformed parameters` Block

- Comes after the `parameters` block but before the `model` block
- Need to declare objects before they are assigned
- Calculate endogenous unknowns that are deterministic functions of things declared in earlier blocks
- Used to create interesting intermediate inputs to the log-kernel
- Declared constraints are validated and samples are stored

```
transformed parameters {
  vector[K] beta; // as yet undefined
  real alpha = GLD_qf(p[K + 1], m[K + 1], r[K + 1], a[K + 1], s[K + 1]);
  real<lower = 0> sigma = GLD_qf(p[K + 2], m[K + 2], r[K + 2], a[K + 2], s[K + 2]);
  for (k in 1:K) beta[k] = GLD_qf(p[k], m[k], r[k], a[k], s[k]); // now defined
}
```

"Required" `model` Block of .stan Programs

- Can declare endogenous unknowns, assign to them, and use them
- Constraints cannot be declared / validated and samples not stored
- The `model` block must define (something proportional to)
  $\texttt{target} = \log(f(\boldsymbol{\theta}) \times f(\mathbf{y}|\boldsymbol{\theta}, \cdot)) = \log f(\boldsymbol{\theta}) + \log f(\mathbf{y}|\boldsymbol{\theta}, \cdot)$
- There is an internal reserved symbol called `target` that is initialized to zero (before change-of-variable adjustments) you increment by `target += ...;`
- Functions ending `_lpdf` or `_lpmf` return scalars even if some of their arguments are vectors or one-dimensional arrays, in which case it sums the log density/mass over the presumed conditionally independent elements

```
model { // log likelihood, equivalent to target += normal_lpdf(y | alpha + X * beta, sigma)
  if (!prior_only) target += normal_id_glm_lpdf(y | X, alpha, beta, sigma);
} // implicit: p ~ uniform(0, 1)
```

Entire Stan Program

```
#include quantile_functions.stan
data {
  int<lower = 0> N; // number of observations
  int<lower = 0> K; // number of predictors
  matrix[N, K] X;   // matrix of predictors
  vector[N] y;      // outcomes
  int<lower = 0, upper = 1> prior_only;   // ignore data?
  vector[K + 2] m;                        // prior medians
  vector<lower = 0>[K + 2] r;             // prior IQRs
  vector<lower = -1, upper = 1>[K + 2] a; // prior asymmetry
  vector<lower =  0, upper = 1>[K + 2] s; // prior steepness
}
parameters {
  vector<lower = 0, upper = 1>[K + 2] p;  // CDF values
}
transformed parameters {
  real alpha = gld_qf(p[K + 1], m[K + 1], r[K + 1], a[K + 1], s[K + 1]);
  vector[K] beta; // as yet undefined
  real<lower = 0> sigma = gld_qf(p[K + 2], m[K + 2], r[K + 2], a[K + 2], s[K + 2]);
  for (k in 1:K) beta[k] = gld_qf(p[k], m[k], r[k], a[k], s[k]); // now defined
}
model { // log likelihood, equivalent to target += normal_lpdf(y | alpha + X * beta, sigma)
  if (!prior_only) target += normal_id_glm_lpdf(y | X, alpha, beta, sigma);
} // implicit: p ~ uniform(0, 1)
```

Calling the **stan** Function

```
post <- stan("linear.stan", data = list(N = nrow(Gabba), K = 1, y = Gabba$Vaccinated,
                                        X = as.matrix(Gabba$Trump - mean(Gabba$Trump)),
                                        prior_only = FALSE, m = m, r = r, a = a, s = s))
```

```
post
```

```
## Inference for Stan model: linear.
## 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
## p[1]         0.48    0.00 0.01      0.46      0.47      0.48      0.49      0.51  4540
## p[2]         0.53    0.00 0.00      0.52      0.53      0.53      0.53      0.54  4677
## p[3]         0.43    0.00 0.01      0.42      0.42      0.43      0.43      0.44  4924
## alpha       51.19    0.00 0.16     50.89     51.08     51.19     51.29     51.49  4677
## beta[1]     -0.51    0.00 0.01     -0.53     -0.52     -0.51     -0.51     -0.49  4542
## sigma        8.52    0.00 0.11      8.31      8.45      8.52      8.59      8.73  4924
## lp__    -11131.50    0.03 1.24 -11134.59 -11132.10 -11131.21 -11130.56 -11130.07  1949
##         Rhat
## p[1]       1
## p[2]       1
## p[3]       1
## alpha      1
```

Working with the Marginal Posterior Draws

```
draws <- as.data.frame(post) %>% select(-starts_with("p")) # has 4000 rows
quantile(draws$`beta[1]`, probs = c(.05, .95)) # what people mistake confidence intervals for
```

```
##         5%        95%
## -0.5286308 -0.4974997
```

```
mean(draws$`beta[1]` > -0.5) # what people mistake p-values for
```

```
## [1] 0.0855
```

Working with Posterior Predictive Distributions

```
x <- Gabba$Trump - mean(Gabba$Trump)
mu <- draws$alpha + t(sapply(draws$`beta[1]`, FUN = function(beta) x * beta))
y_ <- matrix(rnorm(length(mu), mean = mu, sd = draws$sigma), nrow(mu), ncol(mu))
dim(y_) # draws from the posterior predictive distribution that INCLUDES the posterior noise
```

```
## [1] 4000 3125
```

```
lower <- apply(y_, MARGIN = 2, FUN = quantile, probs = 0.25)
upper <- apply(y_, MARGIN = 2, FUN = quantile, probs = 0.75)
with(Gabba, c(too_low = mean(Vaccinated < lower), too_high = mean(Vaccinated > upper),
              just_right = mean( (Vaccinated > lower) & (Vaccinated < upper))))
```

```
##    too_low   too_high just_right
##    0.21024    0.22560    0.56416
```

- Ideally, the last line would be `.25 .25 .50` but this is not too bad. The model is not quite making extreme enough predictions.

Republican Governors

```
Gabba <- mutate(Gabba, # Virginia is somewhat ambiguous due to the 2021 election
                GOP_gov = !(ST %in% c("CA", "CO", "CT", "DE", "HI", "IL", "KS", "KY", "LA",
                                      "ME", "MI", "MN", "NV", "NJ", "NM", "NY", "NC", "OR",
                                      "PA", "RI", "VA", "WA", "WI", "DC")))
```

- What are your beliefs about the effect of a state having a Republican governor, conditional on the county's Trump vote percentage?

```
m <- append(m, values = 5,   after = 1)
r <- append(r, values = 4,   after = 1)
a <- append(a, values = 0,   after = 1) # symmetric
s <- append(s, values = 0.5, after = 1) # logistic tails
```

- Here is a good way to make a centered matrix of predictors in R

```
X <- model.matrix(Vaccinated ~ Trump + GOP_gov, data = Gabba)[, -1] # drop column of 1s
X <- sweep(X, MARGIN = 2, STATS = colMeans(X), FUN = `-`)           # center each column
```

Calling the **stan** Function

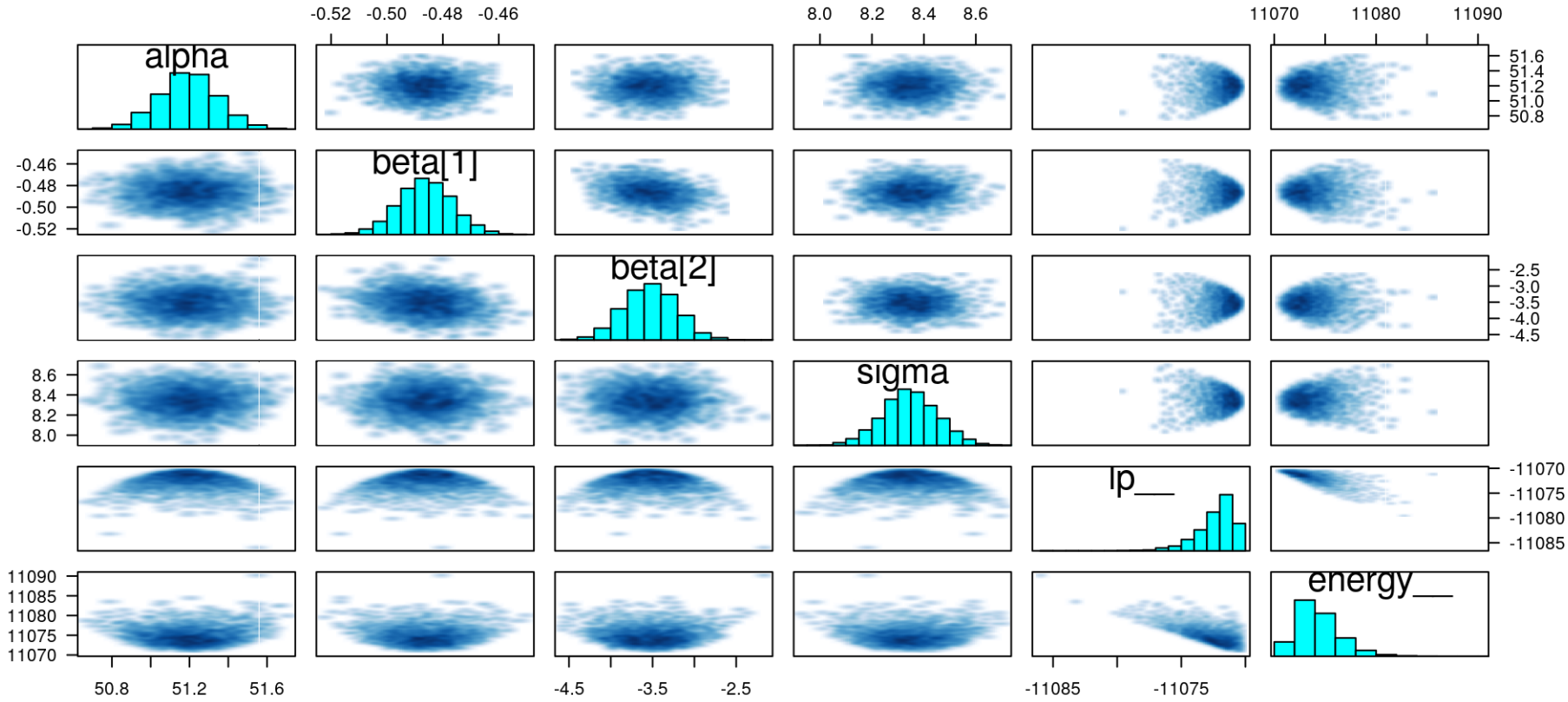
```
post <- stan("linear.stan", data = list(N = nrow(Gabba), K = ncol(X), y = Gabba$Vaccinated,
                                        X = X, prior_only = 0, m = m, r = r, a = a, s = s))
```

```
print(post, pars = c("p", "lp__"), include = FALSE)
```

```
## Inference for Stan model: linear.
## 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
## alpha   51.19    0.00 0.15 50.90 51.09 51.19 51.30 51.49  3868    1
## beta[1] -0.49    0.00 0.01 -0.51 -0.49 -0.49 -0.48 -0.47  3797    1
## beta[2] -3.53    0.01 0.32 -4.14 -3.75 -3.53 -3.31 -2.91  3927    1
## sigma    8.35    0.00 0.11  8.14  8.28  8.34  8.42  8.55  4476    1
##
## Samples were drawn using NUTS(diag_e) at Wed Apr  6 11:48:40 2022.
## 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).
```

Posterior Planes

```
pairs(post, pars = "p", include = FALSE) # this all looks fine in this case
```



ShinyStan

```
library(shinystan)
launch_shinystan(post) # opens in a web browser
```

Utility Function for Predictions of Future Data

- For Bayesians, the log predictive PDF is the most appropriate utility function
- Choose the model that maximizes the expectation of this over FUTURE data

$$\text{ELPD} = \mathbb{E}_Y \ln f\left(y_{N+1}, y_{N+2}, \ldots, y_{2N} \mid y_1, y_2, \ldots, y_N\right) =$$

$$\ln \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f\left(y_{N+1}, y_{N+2}, \ldots, y_{2N} \mid \mathbf{y}\right) dy_{N+1} dy_{N+2} \ldots dy_{2N} \approx$$

$$\sum_{n=1}^{N} \ln f\left(y_n \mid \mathbf{y}_{-n}\right) = \sum_{n=1}^{N} \ln \int_{\Theta} f\left(y_n \mid \boldsymbol{\theta}\right) f\left(\boldsymbol{\theta} \mid \mathbf{y}_{-n}\right) d\theta_1 d\theta_2 \ldots d\theta_K$$

- $f\left(y_n \mid \boldsymbol{\theta}\right)$ is just the n-th likelihood contribution, but can we somehow obtain $f\left(\boldsymbol{\theta} \mid \mathbf{y}_{-n}\right)$ from $f\left(\boldsymbol{\theta} \mid \mathbf{y}\right)$?
- Yes, assuming y_n does not have an outsized influence on the posterior

Optional `generated quantities` Block

- Can declare more endogenous knowns, assign to them, and use them
- Samples are stored
- Can reference anything except stuff in the `model` block
- Can also do this in R afterward, but primarily used for
  - Interesting functions of posterior that don't involve likelihood
  - Posterior predictive distributions and / or functions thereof
  - The log-likelihood for each observation to pass to `loo`

Utilizing Stand-Alone Generated Quantities

```
data {              // saved as "generated_quantities.stan"
  int<lower = 0> N; // number of observations
  int<lower = 0> K; // number of predictors
  matrix[N, K] X;   // matrix of predictors
  vector[N] y;      // outcomes
  /* prior hyperparameters are not needed anymore */
}
parameters {
  real alpha;
  vector[K] beta;
  real<lower = 0> sigma;
}
generated quantities {
  vector[N] log_lik;
  { // mu is not stored because it is in this local block
    vector[N] mu = alpha + X * beta;
    for (n in 1:N) log_lik[n] = normal_lpdf(y[n] | mu[n], sigma);
  }
}
```

Calling Stand-Alone Generated Quantities

```
mod <- stan_model("generated_quantities.stan")
log_lik <- gqs(mod, draws = as.matrix(post),
               data = list(N = nrow(Gabba), K = ncol(X), y = Gabba$Vaccinated, X = X))


loo(log_lik)


##
## Computed from 4000 by 3125 log-likelihood matrix
##
##          Estimate    SE
## elpd_loo -11066.7  66.1
## p_loo         6.4   0.8
## looic     22133.3 132.3
## ------
## Monte Carlo SE of elpd_loo is 0.0.
##
## All Pareto k estimates are good (k < 0.5).
## See help('pareto-k-diagnostic') for details.
```

What about the States?

- Suppose we wanted to include an intercept for each state, rather than merely an indicator for whether the state has a Republican governor
- We could include 50 dummy variables in $\mathbf{X}$ and specify priors on those coefficients, but McElreath prefers the following approach

```
X <- as.matrix(Gabba$Trump - mean(Gabba$Trump))
group <- as.factor(Gabba$State)
nlevels(group) # size N but only 51 unique values
```

```
## [1] 51
```

- We can also utilize normal priors if we prefer with means and standard deviations as

```
m     <- c(beta = -0.5, alpha = 50, sigma = 10)
scale <- c(beta = 0.25, alpha = 10, sigma = 3)
```

```
data {              // saved as "groups.stan"
  int<lower = 0> N; // number of observations
  int<lower = 0> K; // number of predictors
  matrix[N, K] X;   // matrix of predictors
  vector[N] y;      // outcomes
  int<lower = 1> J; // number of groups
  int<lower = 1, upper = J> group[N];     // group membership
  int<lower = 0, upper = 1> prior_only;   // ignore data?
  vector[K + 2] m;                        // prior means
  vector<lower = 0>[K + 2] scale;         // prior scales
}
parameters {
  vector[K] beta;
  vector[J] alpha;
  real<lower = 0> sigma;
}
model {
  if (!prior_only) target += normal_id_glm_lpdf(y | X, alpha[group], beta, sigma);
  target += normal_lpdf(beta  | m[1:K],   scale[1:K]); // ^ important
  target += normal_lpdf(alpha | m[K + 1], scale[K + 1]);
  target += normal_lpdf(sigma | m[K + 2], scale[K + 2]); // actually half normal
}
generated quantities {
  vector[N] log_lik;
  {
    vector[N] mu = alpha[group] + X * beta;
```

Calling **stan** for the grouped model

```
states <- stan("groups.stan", data = list(N = nrow(Gabba), K = ncol(X), y = Gabba$Vaccinated,
                                          X = X, J = nlevels(group), group = as.integer(group),
                                          prior_only = 0, m = m, scale = scale)) # ^ important
```

```
states # only 6 states could fit on the screen but all 51 intercepts were estimated
```

```
## Inference for Stan model: groups.
## 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%
## beta[1]         -0.46    0.00 0.01     -0.48     -0.47     -0.46     -0.46     -0.44
## alpha[1]        42.92    0.01 0.89     41.21     42.31     42.90     43.53     44.57
## alpha[2]        55.20    0.02 1.42     52.46     54.24     55.18     56.16     57.99
## alpha[3]        57.27    0.03 2.06     53.20     55.86     57.26     58.64     61.35
## alpha[4]        48.18    0.01 0.87     46.46     47.61     48.19     48.77     49.90
## alpha[5]        51.37    0.01 1.00     49.43     50.69     51.39     52.05     53.29
## alpha[6]        51.58    0.01 0.96     49.68     50.93     51.58     52.24     53.44
## alpha[7]        62.77    0.03 2.65     57.76     60.96     62.74     64.63     67.95
## alpha[8]        52.55    0.05 4.07     44.68     49.75     52.60     55.31     60.36
## alpha[9]        48.53    0.07 5.88     37.12     44.48     48.61     52.44     60.11
## alpha[10]       53.13    0.01 0.93     51.27     52.49     53.12     53.77     54.94
## alpha[11]       42.94    0.01 0.62     41.72     42.53     42.94     43.36     44.13
```

Model Comparison

```
library(loo)
loo_compare(list(GOP_govs = loo(log_lik), states = loo(states)))
```

```
##          elpd_diff se_diff
## states      0.0       0.0
## GOP_govs -278.6      26.6
```

```
loo_model_weights(list(GOP_govs = loo(log_lik), states = loo(states)))
```

```
## Method: stacking
## ------
##          weight
## GOP_govs 0.076
## states   0.924
```

Leverage Diagnostic Plot

```
plot(loo(states), label_points = TRUE) # not too bad, 318 is D.C.
```

```
## Warning: Some Pareto k diagnostic values are slightly high. See help('pareto-k-diagnostic')
```