Alternative Model Formulations

Stat 341 – Spring 2017

Campus Crime

```
CampusCrime <- read.file("http://www.calvin.edu/~rpruim/data/CampusCrime.csv") %>%
  select(region, type, enrollment, violent_crimes) %>%
  mutate( region_id = coerce_index(region),
          type_id   = coerce_index(type) )
```

```
head(CampusCrime, 3)
```

```
##   region type enrollment violent_crimes region_id type_id
## 1     SE    U       5590             30         4       2
## 2     SE    C        540              0         4       1
## 3      W    U      35747             23         6       2
```

1. How are these models different and what do they say about crime?

```
crime.stan1 <- map2stan(
  alist(
    violent_crimes ~ dpois(lambda),
    log(lambda) ~ log(enrollment) +  b_type * type_id + b_region[region_id],
    b_type ~ dnorm(0, 1),
    b_region[region_id] ~ dnorm(0, 5)
  ), data = CampusCrime)
```

```
crime.stan2 <- map2stan(
  alist(
    violent_crimes ~ dpois(lambda),
    lambda <- rate * enrollment,
    log(rate) ~ b_type * type_id + b_region[region_id],
    b_type ~ dnorm(0, 1),
    b_region[region_id] ~ dnorm(0, 5)
  ), data = CampusCrime)
```

```
precis(crime.stan1, depth = 2)
```

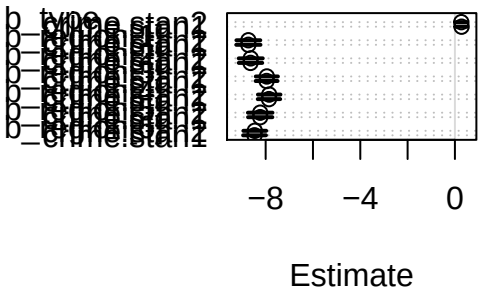
```
##              Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
## b_type       0.28   0.12       0.10       0.48   192    1
## b_region[1] -8.73   0.26      -9.12      -8.29   224    1
## b_region[2] -8.64   0.25      -9.04      -8.24   232    1
## b_region[3] -7.96   0.24      -8.38      -7.61   205    1
## b_region[4] -7.86   0.24      -8.26      -7.49   220    1
## b_region[5] -8.23   0.26      -8.65      -7.84   229    1
## b_region[6] -8.47   0.25      -8.86      -8.06   244    1
```

```
precis(crime.stan2, depth = 2)
```

```
##              Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
## b_type       0.27   0.12       0.09       0.48   274    1
## b_region[1] -8.73   0.26      -9.17      -8.35   294    1
## b_region[2] -8.63   0.26      -9.06      -8.22   298    1
## b_region[3] -7.95   0.24      -8.35      -7.59   290    1
```

```
## b_region[4] -7.85   0.24      -8.24      -7.46   301    1
## b_region[5] -8.23   0.26      -8.70      -7.85   327    1
## b_region[6] -8.46   0.25      -8.91      -8.13   361    1
```

```
plot(coeftab(crime.stan1, crime.stan2))
```



Some Chimpanzee Models

```
data(chimpanzees)
Chimps <- chimpanzees %>%
  mutate(
    recipient = NULL,
    block_id = block    # block is a keyword in Stan
  )
```

This model reports many "diverent transitions".

```
m13.6 <- map2stan(
  alist(
    # likeliood
    pulled_left ~ dbinom(1, p),

    # linear models
    logit(p) <- A + BP * prosoc_left + BPC * condition * prosoc_left,
    A <-     a +   a_actor[actor] +   a_block[block_id],
    BP <-   bp +  bp_actor[actor] +  bp_block[block_id],
    BPC <- bpc + bpc_actor[actor] + bpc_block[block_id],

    # adaptive priors
    c(a_actor, bp_actor, bpc_actor)[actor]    ~ dmvnorm2(0, sigma_actor, Rho_actor),
    c(a_block, bp_block, bpc_block)[block_id] ~ dmvnorm2(0, sigma_block, Rho_block),

    # fixed priors
    c(a, bp, bpc) ~ dnorm(0, 1),
    sigma_actor   ~ dcauchy(0, 2),
    sigma_block   ~ dcauchy(0, 2),
    Rho_actor     ~ dlkjcorr(4),
    Rho_block     ~ dlkjcorr(4)
  ),
  data = Chimps, iter = 3000, warmup = 1000, chains = 3, cores = 3
)
```

This model avoids the "diverent transitions" and converges more efficiently.

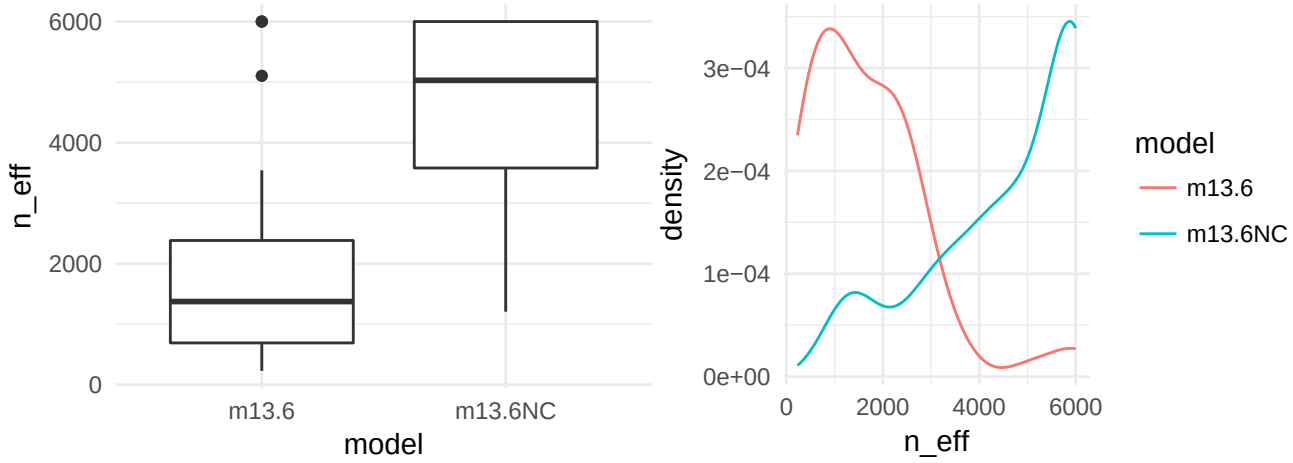
```
m13.6NC <- map2stan(
  alist(
    pulled_left ~ dbinom(1, p),
    logit(p) <- A + BP * prosoc_left  + BPC * condition * prosoc_left,
    A <-    a +    a_actor[actor] +   a_block[block_id],
    BP <-  bp +   bp_actor[actor] +  bp_block[block_id],
    BPC <- bpc + bpc_actor[actor] + bpc_block[block_id],
    # adaptive NON-CENTERED priors
    c(a_actor, bp_actor, bpc_actor)[actor]    ~ dmvnormNC(sigma_actor, Rho_actor),
    c(a_block, bp_block, bpc_block)[block_id] ~ dmvnormNC(sigma_block, Rho_block),
    c(a, bp, bpc) ~ dnorm(0, 1),
    sigma_actor ~ dcauchy(0, 2),
    sigma_block ~ dcauchy(0, 2),
    Rho_actor ~ dlkjcorr(4),
    Rho_block ~ dlkjcorr(4)
  ),
  data = Chimps, iter = 3000, warmup = 1000, chains = 3, cores = 3)
```

A coparison of effective number of samples for parameters in each model.

```
Neff <- data_frame(
  n_eff = precis(m13.6, depth = 2)@output$n_eff,
  model = "m13.6") %>%
  bind_rows(
    data_frame(
      n_eff = precis(m13.6NC, depth = 2)@output$n_eff,
      model = "m13.6NC")
  )
```

```
## Warning in precis(m13.6, depth = 2): There were 305 divergent iterations during sampling.
## Check the chains (trace plots, n_eff, Rhat) carefully to ensure they are valid.
```

```
gf_boxplot(n_eff ~ model, data = Neff)
gf_dens( ~ n_eff + color::model, data = Neff)
```




```
precis(m13.6NC,
       depth = 2,
       pars = c("sigma_actor", "sigma_block"))
```

```
##                Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
## sigma_actor[1] 2.32   0.89       1.12       3.47  2027    1
```

```
## sigma_actor[2] 0.44   0.34       0.00       0.85  3076    1
## sigma_actor[3] 0.52   0.48       0.00       1.07  2992    1
## sigma_block[1] 0.22   0.21       0.00       0.46  3436    1
## sigma_block[2] 0.57   0.40       0.00       1.05  2004    1
## sigma_block[3] 0.51   0.42       0.00       1.02  2884    1
```

```
m13.6NC_link <- link(m13.6NC)
glimpse(m13.6NC_link)
```

```
## List of 4
##  $ p  : num [1:1000, 1:504] 0.218 0.294 0.28 0.195 0.335 ...
##  $ A  : num [1:1000, 1:504] -1.279 -0.877 -0.942 -1.42 -0.685 ...
##  $ BP : num [1:1000, 1:504] 1.148 0.736 0.242 0.904 0.732 ...
##  $ BPC: num [1:1000, 1:504] -0.0315 0.1818 -0.866 -0.0693 -0.9818 ...
```

What is dlkjcorr()?

This allows us to sample correlation matrices. The argument `eta` controls the distribution of off-diagonal elements. A larger value of `eta` implies less correlation.

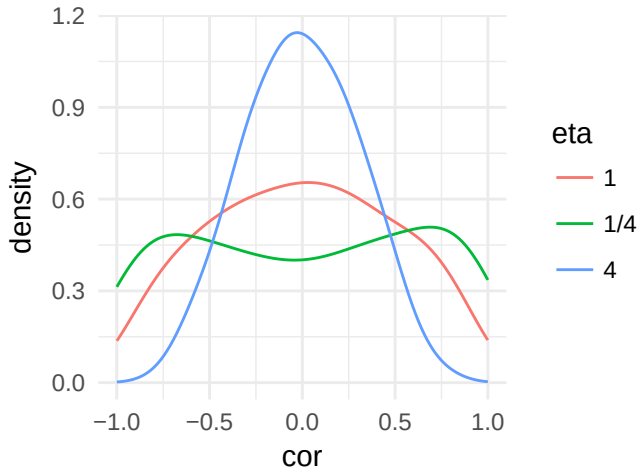
```
rlkjcorr(1, K = 3, eta = 4)
```

```
##           [,1]        [,2]        [,3]
## [1,] 1.0000000  0.35288234  0.19718120
## [2,] 0.3528823  1.00000000 -0.09614385
## [3,] 0.1971812 -0.09614385  1.00000000
```

```
rlkjcorr(1, K = 3, eta = 1/4)
```

```
##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.5670915 0.9993622
## [2,] 0.5670915 1.0000000 0.5373966
## [3,] 0.9993622 0.5373966 1.0000000
```

```
DD <-
  bind_rows(
    data_frame(cor = rlkjcorr(1000, K=3, eta = 1) %>% as.vector, eta = "1"),
    data_frame(cor = rlkjcorr(1000, K=3, eta = 4) %>% as.vector, eta = "4"),
    data_frame(cor = rlkjcorr(1000, K=3, eta = 1/4) %>% as.vector, eta = "1/4")
  )
gf_dens( ~ cor + color::eta, data = DD %>% filter(cor < .99999), adjust = 2)
```



What's the magic potion?

The situation here is similar to the Poisson model above, but all the work of re-expressing the model is hidden in `dmvnormNC()` vs. `dmvnorm2()`.

The basic idea is that the following are equivalent:

- $y \sim \text{Norm}(\mu, \sigma)$
- $y = \mu + z\sigma$, and $z \sim \text{Norm}(0, 1)$

In our example, however,

- We are dealing with multivariate distributions, so μ is a vector of means and σ is replaced by a matrix of covariances.
- We were using a model that had $\mu = 0$, so all the action is in the σ part – basically we are separating the covariance matrix into standard deviations and a correlation matrix (and then pulling the corellation matrix out of the priors too).
- `dmvnormNC()` does all the work of pulling things apart and moving expressions out of priors and into the linear models (like we did by hand with the Poisson model above).

You can see all the ugly coding with

```
m13.6NC %>% stancode()
```

```
## data{
##     int<lower=1> N;
##     int<lower=1> N_actor;
##     int<lower=1> N_block_id;
##     int pulled_left[N];
##     int condition[N];
##     int prosoc_left[N];
##     int actor[N];
##     int block_id[N];
## }
## parameters{
##     matrix[3,N_block_id] z_N_block_id;
##     cholesky_factor_corr[3] L_Rho_block;
##     matrix[3,N_actor] z_N_actor;
##     cholesky_factor_corr[3] L_Rho_actor;
##     real a;
##     real bp;
##     real bpc;
##     vector<lower=0>[3] sigma_actor;
##     vector<lower=0>[3] sigma_block;
## }
## transformed parameters{
##     matrix[N_block_id,3] v_N_block_id;
##     vector[N_block_id] a_block;
##     vector[N_block_id] bp_block;
##     vector[N_block_id] bpc_block;
##     matrix[3,3] Rho_block;
##     matrix[N_actor,3] v_N_actor;
##     vector[N_actor] a_actor;
##     vector[N_actor] bp_actor;
##     vector[N_actor] bpc_actor;
##     matrix[3,3] Rho_actor;
##     v_N_block_id = (diag_pre_multiply(sigma_block,L_Rho_block)*z_N_block_id)';
```

```
##     a_block = col(v_N_block_id,1);
##     bp_block = col(v_N_block_id,2);
##     bpc_block = col(v_N_block_id,3);
##     Rho_block = L_Rho_block * L_Rho_block';
##     v_N_actor = (diag_pre_multiply(sigma_actor,L_Rho_actor)*z_N_actor)';
##     a_actor = col(v_N_actor,1);
##     bp_actor = col(v_N_actor,2);
##     bpc_actor = col(v_N_actor,3);
##     Rho_actor = L_Rho_actor * L_Rho_actor';
## }
## model{
##     vector[N] BPC;
##     vector[N] BP;
##     vector[N] A;
##     vector[N] p;
##     L_Rho_block ~ lkj_corr_cholesky( 4 );
##     L_Rho_actor ~ lkj_corr_cholesky( 4 );
##     sigma_block ~ cauchy( 0 , 2 );
##     sigma_actor ~ cauchy( 0 , 2 );
##     bpc ~ normal( 0 , 1 );
##     bp ~ normal( 0 , 1 );
##     a ~ normal( 0 , 1 );
##     to_vector(z_N_block_id) ~ normal( 0 , 1 );
##     to_vector(z_N_actor) ~ normal( 0 , 1 );
##     for ( i in 1:N ) {
##         BPC[i] = bpc + bpc_actor[actor[i]] + bpc_block[block_id[i]];
##     }
##     for ( i in 1:N ) {
##         BP[i] = bp + bp_actor[actor[i]] + bp_block[block_id[i]];
##     }
##     for ( i in 1:N ) {
##         A[i] = a + a_actor[actor[i]] + a_block[block_id[i]];
##     }
##     for ( i in 1:N ) {
##         p[i] = A[i] + BP[i] * prosoc_left[i] + BPC[i] * condition[i] * prosoc_left[i];
##     }
##     pulled_left ~ binomial_logit( 1 , p );
## }
## generated quantities{
##     vector[N] BPC;
##     vector[N] BP;
##     vector[N] A;
##     vector[N] p;
##     real dev;
##     dev = 0;
##     for ( i in 1:N ) {
##         BPC[i] = bpc + bpc_actor[actor[i]] + bpc_block[block_id[i]];
##     }
##     for ( i in 1:N ) {
##         BP[i] = bp + bp_actor[actor[i]] + bp_block[block_id[i]];
##     }
##     for ( i in 1:N ) {
##         A[i] = a + a_actor[actor[i]] + a_block[block_id[i]];
##     }
```

```
##     for ( i in 1:N ) {
##         p[i] = A[i] + BP[i] * prosoc_left[i] + BPC[i] * condition[i] * prosoc_left[i];
##     }
##     dev = dev + (-2)*binomial_logit_lpmf( pulled_left | 1 , p );
## }
```

Roughly, the result of this is to change the shape of the poasterior, which can make sampling more or less efficient. (Generally, sampling works poorly if there are regions that are too flat or two curved, and prefers to have a moderate amount of curvature.)

There is not rule to say which of two paramterizations might work best, but knowing that the paramterization can matter, when one paramterization fails, we should try another before giving up on the model.

Here is a version that is not quite as fancy as what `dmvnormNC()` is doing, but shows the kinds of things that are going on.

```
m13.6nc1 <- map2stan(
  alist(
    pulled_left ~ dbinom(1, p),

    # linear models
    logit(p) <- A + BP * prosoc_left + BPC * condition * prosoc_left,
    A <-     a +   za_actor[actor] * sigma_actor[1] +   za_block[block_id] * sigma_block[1],
    BP <-   bp +  zbp_actor[actor] * sigma_actor[2] +  zbp_block[block_id] * sigma_block[2],
    BPC <- bpc + zbpc_actor[actor] * sigma_actor[3] + zbpc_block[block_id] * sigma_block[3],

    # adaptive priors
    c(za_actor, zbp_actor, zbpc_actor)[actor]    ~ dmvnorm(0, Rho_actor),
    c(za_block, zbp_block, zbpc_block)[block_id] ~ dmvnorm(0, Rho_block),

    # fixed priors
    c(a, bp, bpc) ~ dnorm(0, 1),
    sigma_actor ~ dcauchy(0, 2),
    sigma_block ~ dcauchy(0, 2),
    Rho_actor ~ dlkjcorr(4),
    Rho_block ~ dlkjcorr(4)
  ),
  data = Chimps,
  start = list(sigma_actor = c(1, 1, 1), sigma_block = c(1, 1, 1)),
  constraints = list(sigma_actor = "lower=0", sigma_block = "lower=0"),
  types = list(Rho_actor = "corr_matrix", Rho_block = "corr_matrix"),
  iter = 2000, warmup = 1000, chains = 1, cores = 3 )
```

```
##
## SAMPLING FOR MODEL 'pulled_left ~ dbinom(1, p)' NOW (CHAIN 1).
##
## Gradient evaluation took 0.000298 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 2.98 seconds.
## Adjust your expectations accordingly!
##
##
## Iteration:    1 / 2000 [  0%]  (Warmup)
## Iteration:  200 / 2000 [ 10%]  (Warmup)
## Iteration:  400 / 2000 [ 20%]  (Warmup)
## Iteration:  600 / 2000 [ 30%]  (Warmup)
## Iteration:  800 / 2000 [ 40%]  (Warmup)
```

```
## Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Iteration: 2000 / 2000 [100%]  (Sampling)
##
##  Elapsed Time: 10.6933 seconds (Warm-up)
##                8.77084 seconds (Sampling)
##                19.4641 seconds (Total)

## The following numerical problems occurred the indicated number of times on chain 1

##                                                                          count
## Exception thrown at line 46: lkj_corr_lpdf: y is not positive definite.     1

## When a numerical problem occurs, the Hamiltonian proposal gets rejected.

## See http://mc-stan.org/misc/warnings.html#exception-hamiltonian-proposal-rejected

## If the number in the 'count' column is small, there is no need to ask about this message on stan-use

##
## SAMPLING FOR MODEL 'pulled_left ~ dbinom(1, p)' NOW (CHAIN 1).
##
## Gradient evaluation took 0.00023 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 2.3 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: No variance estimation is
##          performed for num_warmup < 20
##
## Iteration: 1 / 1 [100%]  (Sampling)
##
##  Elapsed Time: 1e-06 seconds (Warm-up)
##                0.000609 seconds (Sampling)
##                0.00061 seconds (Total)

## Computing WAIC

## Constructing posterior predictions
```

What do we learn from the models?

The main story seems to be that there is a lot of variability from chimp to chimp in left vs. right preference. The other factors (`condition` and `prosoc_left`) seem to make much less difference.

```
m12.5 <- map2stan(
  alist(
    pulled_left ~ dbinom(1, p),
    logit(p) <- a + a_actor[actor] + a_block[block_id] +
      (bp + bpc * condition) * prosoc_left,
    a_actor[actor] ~ dnorm(0, sigma_actor),
    a_block[block_id] ~ dnorm(0, sigma_block),
    c(a, bp, bpc) ~ dnorm(0, 10),
    sigma_actor ~ dcauchy(0, 1),
```

```
    sigma_block ~ dcauchy(0, 1)
  ),
  data = Chimps,
  warmup = 1000, iter = 3000, chains = 3, cores = 3, refresh = 0)
```

```
## Warning: There were 1 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help.
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

```
## Warning: Examine the pairs() plot to diagnose sampling problems
```

```
##
## SAMPLING FOR MODEL 'pulled_left ~ dbinom(1, p)' NOW (CHAIN 1).
##
## Gradient evaluation took 0.000157 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 1.57 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: No variance estimation is
##          performed for num_warmup < 20
##
## Iteration: 1 / 1 [100%]  (Sampling)
##
##  Elapsed Time: 0 seconds (Warm-up)
##                0.000294 seconds (Sampling)
##                0.000294 seconds (Total)
```

```
## Computing WAIC
```

```
## Constructing posterior predictions
```

```
## Warning in map2stan(alist(pulled_left ~ dbinom(1, p), logit(p) <- a + a_actor[actor] + : There were
## Check the chains (trace plots, n_eff, Rhat) carefully to ensure they are valid.
```

```
compare(m13.6NC, m12.5)
```

```
##          WAIC pWAIC dWAIC weight    SE  dSE
## m12.5   532.6  10.4   0.0   0.75 19.67   NA
## m13.6NC 534.8  18.3   2.1   0.25 19.88 4.03
```