RETHINKING
AN R PACKAGE FOR FITTING AND MANIPULATING BAYESIAN MODELS
VERSION 1.57

RICHARD MCELREATH

Contents

1. Overview

The rethinking R package began as an instructional aid for a PhD course on applied Bayesian statistics. It has since grown into much more. Its motivation is to provide (1) a very flexible modeling syntax that closely resembles typical mathematical notation and (2) access to both *maximum a posteriori* (MAP) and Hamiltonian Monte Carlo (HMC, provided by rstan) for model fitting. As a result, rethinking can specify many more types of models than a package like MCMCglmm can, while also using much better sampling algorithms, as provided by rstan.

The package also provides tools for extracting and processing posterior samples and predictions, computing WAIC and DIC criteria for model comparison, and construction of posterior prediction model ensembles. Finally, there is a function `glimmer` that translates mixed model formulas in `lme4`'s `glmer` syntax into the model definitions needed to use `rethinking`'s HMC model fitting functions. The main purpose of `glimmer` is to help the user understand the model implied by a `lme4` formula. Specifying models directly with `rethinking` formulas is more flexible.

This manual is an introduction to the `rethinking` package, but not a guide to doing Bayesian statistical inference. If you are not yet familiar with conventions of applied Bayesian statistics, beginning with a textbook is a better idea.

1.1. **Installation.** The `rethinking` package is not on CRAN. Luckily it is still very easy to install `rethinking`. There are two steps.

First, install `rstan`. It is available at `mc-stan.org`. Many users will have to install a C compiler, before they can successfully install `rstan`. The Stan website has platform specific instructions for acquiring and installing a C compiler. In my experience, there are two common platform-specific obstacles.

- Windows: After downloading the `RTools` installer wizard, run it by right-clicking on it and choosing "Run as administrator." Then near the end of the install, be sure to check the box for setting the system `PATH` variable. This will allow your R install to see the compiler.
- Mac OS X: After installing Xcode, be sure to run it once and agree to the license. Some users experience an error later when trying to compile `rstan` that states "`llvm-g++-4.2: command not found`." If you get this message, it may indicate that R is looking for the wrong compiler; it should be using `clang++`, not `llvm-g++`. You can trick R into seeing the correct compiler by opening the Terminal and issuing these three lines of code, one after another:

```
cd /usr/bin
sudo ln -fs clang llvm-gcc-4.2
sudo ln -fs clang++ llvm-g++-4.2
```

Once the compiler is installed, continue with the instructions on the `mc-stan.org` website. After `rstan` finishes compiling, it's a good idea to close R and restart it.

The second step is to install `rethinking` itself. You have two options. You can download it from Github or from the author's repository. To install from Github, use these commands within R:

R code 1.1
```
install.packages(c('devtools','coda','mvtnorm'))
library(devtools)
install_github("rmcelreath/rethinking")
```

Or to install from the repository:

R code 1.2
```
install.packages(c('coda','mvtnorm'))
options(repos=c(getOption('repos'),rethinking='http://xcelab.net/R'))
install.packages('rethinking',type='source')
```

Once both `rstan` and `rethinking` are installed, you can access the in-system help files with:

R code 1.3
```
library(rethinking)
help(package=rethinking)
```

Like most R documentation, those help pages provide technical reference but do not really stand on their own. The textbook that accompanies the `rethinking` package, *Statistical Rethinking: A Bayesian Course with R Examples*, provides many examples of its use. The remainder of this manual provides additional technical documentation and concise examples of different model families that can be specified and fit.

1.2. **`map`: *Maximum a posteriori* fitting.** The `map` function provides an interface for *maximum a posteriori* fitting, usually abbreviated "MAP." MAP fitting uses optimization to search for the minimum of the log-posterior, as defined by a model. The parameter values that minimize the log-posterior possess maximum posterior probability, so these are the MAP parameter values. `map` finds the MAP values, using R's built-in `optim`, and then approximates the curvature near them using the Hessian. This effectively assumes a multivariate Gaussian posterior distribution.

1.2.1. *Model formulas.* A statistical model is specified for `map` by using a list of formulas. For example, here is a simple linear regression:

R code 1.4
```
a_map_formula <- alist(
    y ~ dnorm(mu,sigma),
    mu <- a + b*x,
    a ~ dnorm(0,100),
    b ~ dnorm(0,1),
    sigma ~ dunif(0,10)
)
```

The above corresponds to this model, in standard mathematical notation:

$$
\begin{aligned}
y_i &\sim \text{Normal}(\mu_i, \sigma) \\
\mu_i &= \alpha + \beta x_i \\
\alpha &\sim \text{Normal}(0, 100) \\
\beta &\sim \text{Normal}(0, 1) \\
\sigma &\sim \text{Uniform}(0, 10)
\end{aligned}
$$

`map` uses the formula list to build a function that computes the log-posterior for any combination of parameter values. It then passes that function to `optim`, which searches for the MAP.

For example:

R code 1.5
```
data(cars)

# define model
cars_formula <- alist(
    dist ~ dnorm(mu,sigma),
```

```
    mu <- a + b*speed,
    a ~ dnorm(0,100),
    b ~ dnorm(0,10),
    sigma ~ dunif(0,50)
)

# pass to map
cars_map <- map( cars_formula , data=cars )

# default show information
show(cars_map)
```

```
Maximum a posteriori (MAP) model fit

Formula:
dist ~ dnorm(mu, sigma)
mu <- a + b * speed
a ~ dnorm(0, 100)
b ~ dnorm(0, 10)
sigma ~ dunif(0, 50)

MAP values:
         a          b      sigma
-17.400714   3.921415  15.068462

Log-likelihood: -206.58
```

1.2.2. *Initial values.* As with many optimization algorithms, `map` needs initial values for all parameters in order to begin searching. If explicit initial values are omitted, as in the above example, then `map` will sample random initial values from the priors. Initial values can be specified with a named `start` list.

It is possible to omit priors for parameters. In that case, a flat (usually improper) prior is implied and the user must also specify a starting value for the parameter. For example:

R code 1.6

```
cars_formula2 <- alist(
    dist ~ dnorm(mu,sigma),
    mu <- a + b*speed
)
start_list <- list(
    a=mean(cars$dist),
    b=0,
    sigma=sd(cars$dist)
)
cars_mle <- map( cars_formula2 , data=cars , start=start_list )
summary(cars_mle)
```

```
        Mean StdDev   5.5% 94.5%
a     -17.59   6.62 -28.17 -7.01
b       3.93   0.41   3.28  4.58
sigma  15.07   1.51  12.66 17.48
```

Since the priors are flat in the example above, these estimates correspond to a maximum likelihood solution with quadratic standard errors.

1.2.3. *Link functions.* `map` understands link functions in conventional form. For example, here is a binomial model with logit link:

R code 1.7

```
data(UCBadmit)
UCBadmit$male <- ifelse(UCBadmit$applicant.gender=="male",1,0)
ucb_formula <- alist(
    admit ~ dbinom(applications,p),
    logit(p) <- a + b*male,
    a ~ dnorm(0,10),
    b ~ dnorm(0,1)
)
ucb_map <- map( ucb_formula , data=UCBadmit )
```

1.2.4. *Vector parameters.* `map` can construct vectors of parameters, using index variables. For example:

R code 1.8

```
# convert dept factor to integer index variable
UCBadmit$dept_id <- coerce_index(UCBadmit$dept)

# formula
ucb_formula2 <- alist(
    admit ~ dbinom(applications,p),
    logit(p) <- a[dept_id] + b*male,
    a[dept_id] ~ dnorm(0,10),
    b ~ dnorm(0,1)
)

ucb_map2 <- map( ucb_formula2 , data=UCBadmit )

precis(ucb_map2,depth=2)
```

```
      Mean StdDev  5.5% 94.5%
a[1]  0.68   0.10  0.52  0.84
a[2]  0.64   0.12  0.45  0.82
a[3] -0.58   0.07 -0.70 -0.46
a[4] -0.61   0.09 -0.75 -0.48
a[5] -1.06   0.10 -1.22 -0.90
a[6] -2.62   0.16 -2.88 -2.37
b    -0.10   0.08 -0.23  0.03
```

A unique intercept estimate is return for each unique value in `dept_id`.

1.2.5. *Custom densities.* Since `map` constructs a function for the log-posterior directly from the formula list, any R density function can, in principle, be used for a likelihood or prior. This includes custom functions written by the user.

1.3. **`map2stan`: Hamiltonian Monte Carlo.** The other model fitting interface is `map2stan`, which compiles the same type of formula lists into Stan models that can be fit using Hamiltonian Monte Carlo. This allows a wide range of multilevel and mixed effects models to be fit.

Good practice when compiling models through `map2stan` includes:

(1) Do not use any variable names with dots "." in them. Stan does not allow dots in variable or parameter names. `map2stan` will convert any dots to underscores "_", but mismatches between internal names and the original data will persist and lead to potential errors down the pipeline. Better to rename all variables and parameters at the start.

(2) All data must be numeric. Remove any character or factor data from the data frame, or rather pass a minimal list of data to `map2stan` that omits any character or factor data.

For example, here is a varying intercepts binomial regression:

R code 1.9

```
data(UCBadmit)

# construct 'male' dummy variable
UCBadmit$male <- ifelse(UCBadmit$applicant.gender=="male",1,0)

# convert dept factor to integer index variable
UCBadmit$dept_id <- coerce_index(UCBadmit$dept)

# formula
ucb_formula3 <- alist(
    admit ~ dbinom(applications,p),
    logit(p) <- a[dept_id] + b*male,
    a[dept_id] ~ dnorm(a_bar,sigma),
    b ~ dnorm(0,1),
    a_bar ~ dnorm(0,10),
    sigma ~ dcauchy(0,2.5)
)

ucb_stan <- map2stan( ucb_formula3 , data=UCBadmit ,
    iter=2000 , warmup=1000 , chains=1 ,
    refresh=500 )
```

```
TRANSLATING MODEL 'admit ~ dbinom(applications, p)' FROM Stan CODE TO C++ CODE NOW.
COMPILING THE C++ CODE FOR MODEL 'admit ~ dbinom(applications, p)' NOW.

SAMPLING FOR MODEL 'admit ~ dbinom(applications, p)' NOW (CHAIN 1).

Iteration:    1 / 2000 [  0%]  (Warmup)
Iteration:  500 / 2000 [ 25%]  (Warmup)
Iteration: 1000 / 2000 [ 50%]  (Warmup)
Iteration: 1001 / 2000 [ 50%]  (Sampling)
Iteration: 1500 / 2000 [ 75%]  (Sampling)
Iteration: 2000 / 2000 [100%]  (Sampling)
#  Elapsed Time: 0.049535 seconds (Warm-up)
```

```
#                0.043325 seconds (Sampling)
#                0.09286 seconds (Total)


#  Elapsed Time: 3e-06 seconds (Warm-up)
#                2.8e-05 seconds (Sampling)
#                3.1e-05 seconds (Total)

Computing WAIC
Constructing posterior predictions
[ 1000 / 1000 ]
Aggregated binomial counts detected. Splitting to 0/1 outcome for WAIC calculation.
Warning messages:
1: In map2stan(ucb_formula3, data = UCBadmit, iter = 2000, warmup = 1000,  :
  Renaming variable 'applicant.gender' to 'applicant_gender' internally.
You should rename the variable to remove all dots '.'
2: In FUN(c("dept", "applicant_gender", "admit", "reject", "applications",  :
  data with name dept is not numeric and not used
3: In FUN(c("dept", "applicant_gender", "admit", "reject", "applications",  :
  data with name applicant_gender is not numeric and not used
4: In FUN(c("dept", "applicant_gender", "admit", "reject", "applications",  :
  data with name dept is not numeric and not used
5: In FUN(c("dept", "applicant_gender", "admit", "reject", "applications",  :
  data with name applicant_gender is not numeric and not used
```

The model has sampled correctly, but the warnings at the end indicate both the renaming of internal dots to underscores and the complaints about non-numeric data.

1.3.1. *Stan distribution names.* You can use Stan's internal distribution names, rather than R names, when defining `map2stan` models. For example, the previous model can be defined equivalently:

R code 1.10

```
# formula
ucb_formula4 <- alist(
    admit ~ binomial(applications,p),
    logit(p) <- a[dept_id] + b*male,
    a[dept_id] ~ normal(a_bar,sigma),
    b ~ normal(0,1),
    a_bar ~ normal(0,10),
    sigma ~ cauchy(0,2.5)
)
```

1.3.2. *Accessing samples.* Extract the samples into a named list of parameters with:

R code 1.11

```
post <- extract.samples(ucb_stan)
str(post)
```

```
 $ a    : num [1:1000, 1:6] 0.682 0.601 0.86 0.649 0.695 ...
 $ b    : num [1:1000(1d)] -0.1716 -0.0766 -0.1984 -0.0691 -0.1067 ...
 $ a_bar: num [1:1000(1d)] -1.282 -0.303 -0.416 -0.928 -0.568 ...
 $ sigma: num [1:1000(1d)] 1.815 2.536 1.1 0.851 1.148 ...
```

Note that the vector of parameters `a` is returned as a matrix, with samples in rows and parameters in columns. Matrix parameters (not shown) are similarly returned as arrays, with samples in the first dimension.

1.3.3. *Accessing the internal Stan code.* The Stan code that was compiled for sampling is available with `stancode`:

R code 1.12
```
stancode(ucb_stan)
```

```
data{
    int<lower=1> N;
    int<lower=1> N_dept_id;
    int admit[N];
    int applications[N];
    int dept_id[N];
    real male[N];
}
parameters{
    vector[N_dept_id] a;
    real b;
    real a_bar;
    real<lower=0> sigma;
}
model{
    vector[N] p;
    sigma ~ cauchy( 0 , 2.5 );
    a_bar ~ normal( 0 , 10 );
    b ~ normal( 0 , 1 );
    a ~ normal( a_bar , sigma );
    for ( i in 1:N ) {
        p[i] <- a[dept_id[i]] + b * male[i];
    }
    admit ~ binomial_logit( applications , p );
}
generated quantities{
    vector[N] p;
    real dev;
    dev <- 0;
    for ( i in 1:N ) {
        p[i] <- a[dept_id[i]] + b * male[i];
    }
    dev <- dev + (-2)*binomial_logit_log( admit , applications , p );
}
```

The generated quantities block is used to automatically compute DIC for each model. WAIC is also computed, but all calculations are done outside of Stan.

1.3.4. *Diagnostics.* Summary information for a `map2stan` model fit includes the per-parameter chain diagnostics `n_eff`, the number of effective samples, and `Rhat`, the Gelman-Rubin convergence diagnostic.

```
precis(ucb_stan,depth=2)
```

R code 1.13

```
       Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
a[1]   0.68   0.10       0.53       0.83   552    1
a[2]   0.63   0.11       0.43       0.78   481    1
a[3]  -0.58   0.07      -0.70      -0.46   848    1
a[4]  -0.62   0.09      -0.77      -0.49   633    1
a[5]  -1.06   0.10      -1.22      -0.90  1000    1
a[6]  -2.61   0.15      -2.83      -2.36   847    1
b     -0.09   0.08      -0.21       0.04   413    1
a_bar -0.58   0.63      -1.58       0.38   483    1
sigma  1.48   0.57       0.73       2.18   447    1
```

You can also check for divergent iterations:

```
divergent(ucb_stan)
```

R code 1.14

```
[1] 0
```

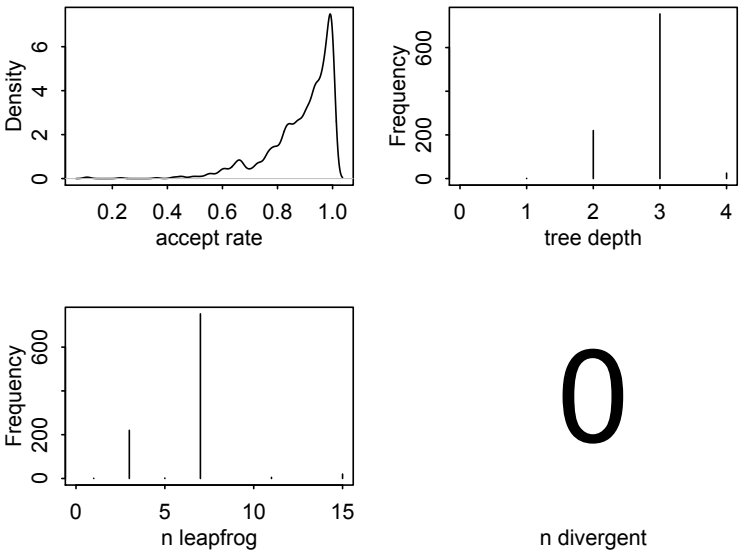
There were no divergent iterations in this example, but `map2stan` automatically monitors divergent iterations and warns the user if any occur.

In addition to `divergent`, you can use `dashboard` to get a more complete report of sampler parameters. For example:

```
sp <- dashboard(ucb_stan)
```

R code 1.15

This plot is shown:

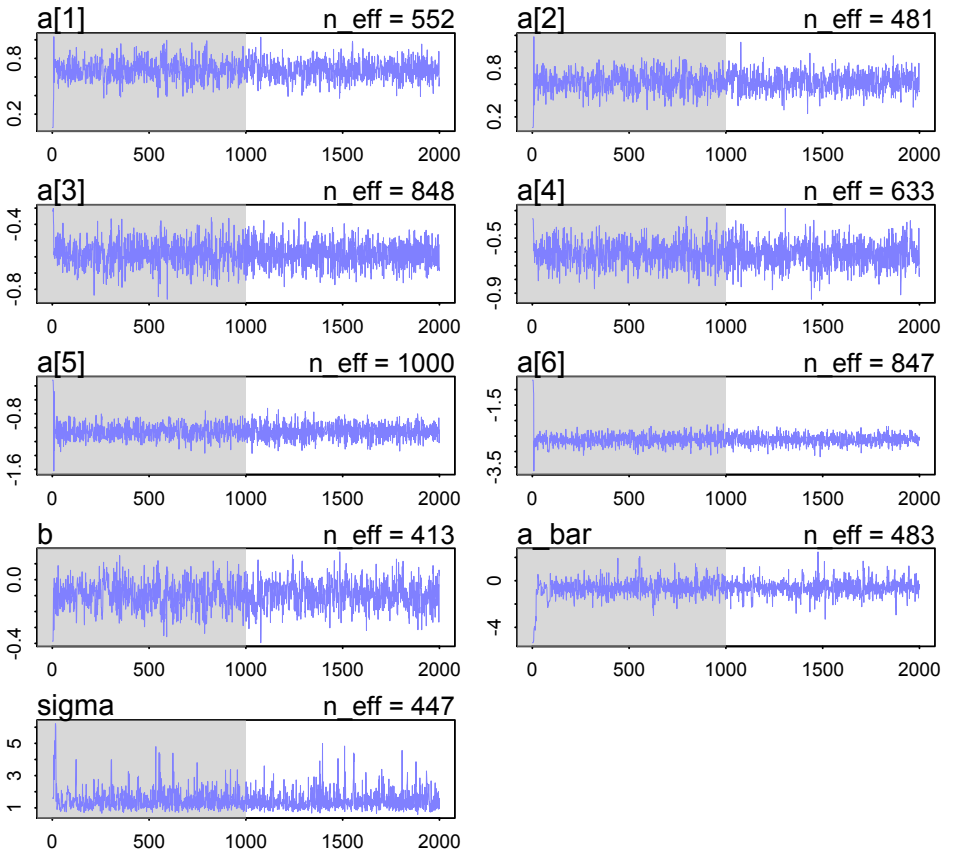


And the object `sp` contains a matrix of the per-sample diagnostics.

1.3.5. *Control parameters.* Stan's control parameters can be modified by using an optional named `control` list. For example, `control=list(adapt_delta=0.95)` sets the target acceptance rate to 95%. See `?stan` for details.

1.3.6. *Trace plots.* To view the trace plot for all parameters:

R code 1.16
```
plot(ucb_stan)
```



Gray regions indicate warmup samples. Multiple chains are display in different colors (not shown). See `plot.map2stan` for more details and additional options. See also `pairs.map2stan` for a pairs plot method.

1.3.7. *Parameter constraints.* `map2stan` will guess some parameter constraints. For example, the standard deviation for a Gaussian likelihood or prior is automatically set to a lower bound of zero, "`<lower=0>`" in Stan's language.

Additional custom parameter constraints can be specified using the `constraints` named list. For example, to redundantly specify a positive constraint on a parameter `sigma`, use `constraints=list(sigma="lower=0")`. To specify both a lower and upper bound, use `constraints=list(sigma="lower=0,upper=10")`.

As always with Stan models, it is important that the constraints on a parameter match the constraints on its prior. Otherwise, you might open a black hole or a portal to another dimension.

1.3.8. *Parameter types.* Stan is a statically typed language, and `map2stan` guesses the correct type for each parameter. If you need to manually specify a parameter type, you can use a named `types` list. For example, to define a matrix `Rho` as a correlation matrix, you could use `types=list(Rho="corr_matrix")`.

1.3.9. *Parallel chains.* To run multiple chains in parallel on multiple cores, use the optional `cores` argument. For example:

R code 1.17

```
ucb_stan_4chains <- map2stan( ucb_formula3 , data=UCBadmit ,
    iter=2000 , warmup=1000 , chains=4 , cores=3 ,
    refresh=-1 )
```

The model will be pre-compiled and then the `parallel` package will be used to parallelize the chains across the number of specified cores. A single fit model object with samples from all chains will be returned. The `refresh=-1` argument above turns off the sampling progress display.

If you are experiencing crashes when parallelizing chains this way, it's probably because you are running R in a graphical interface, either the application from CRAN or RStudio. If you run R instead from the Terminal/Command Prompt, not only will those crashes stop, but it will use less memory, be more responsive, and allow you to run any number of different R sessions on your computer simultaneously. Graphical interfaces are so 20th century.

1.4. **Posterior predictions with `link` and `sim`.** Once you have a model fit with `map` or `map2stan`, you can process the samples directly by using `extract.samples`. For a `map` model, this draws samples from the multivariate Gaussian posterior defined by the MAP values and the variance-covariance matrix defined by the Hessian. For a `map2stan` model, this just copies the Stan samples (post-warmup) into a named list. Note that the Stan samples will be permuted, such that the chains are mixed and sequential samples are not autocorrelated. If you want the samples in the order they were drawn and split by chain (how they appear in the trace plots), see `?rstan::extract`.

There are two convenience tools, however, that make working with model predictions easier. The first is `link`, which produces posterior values of linear models. The second is `sim`, which simulates posterior predictions. This section provides an introduction to each.

1.4.1. `link`. This function computes the value of each linear model at each sample from the posterior. Inverse link functions are applied, so that the value of each left-hand symbol is returned. For example, in a logistic regression, the linear model will appear like this:

```
logit(p) <- a + b * x
```

The function `link` will return the value of `p` for each sample from the posterior, not `logit(p)`. By default, the data used to fit model are used in calculation. But an optional `data` argument allows computing linear model values over any new prediction set you like.

Here's an example, using a varying intercepts logistic regression:

R code 1.18

```
data(chimpanzees)
d <- list(
    pulled_left = chimpanzees$pulled_left ,
    prosoc_left = chimpanzees$prosoc_left ,
    condition = chimpanzees$condition ,
    actor = as.integer( chimpanzees$actor )
```

```
)
m <- map2stan(
    alist(
        pulled_left ~ dbinom(1,p),
        logit(p) <- a_actor[actor] + (bp + bpc*condition)*prosoc_left,
        a_actor[actor] ~ dnorm( a , sigma_actor ) ,
        a ~ dnorm(0,10),
        bp ~ dnorm(0,10),
        bpc ~ dnorm(0,10),
        sigma_actor ~ dcauchy(0,1)
    ) ,
    data=d, iter=6000 , chains=3 , cores=3 , refresh=-1 )

p <- link(m,n=2000)
str(p)
```

```
 num [1:2000, 1:504] 0.199 0.373 0.265 0.28 0.382 ...
```

The result is a matrix, with samples on rows and observations in columns. This set of data consists of 504 observations. By using `n=2000` in the call to `link`, we ask for 2000 samples from the posterior. If you use `n=0`, you will get as many samples as Stan returned (9000 in this case).

To compute values for new data, just construct a new data frame. You need to specify values for all of the predictor variables. This includes all index variables. But outcome variables are not needed. So for example to compute posterior predictions for actor #4 across all four treatments:

R code 1.19

```
new_data <- data.frame(
    actor = 4,
    prosoc_left = c(1,0,1,0),
    condition = c(1,1,0,0)
)
p <- link(m,data=new_data)
str(p)
```

```
 num [1:1000, 1:4] 0.501 0.244 0.509 0.542 0.354 ...
```

Now what do you do with all of these linear model values? Anything you like. But the most common task is to summarize them. To compute means and 90% highest posterior density intervals for each observation (column):

R code 1.20

```
p_mean <- apply(p,2,mean)
p_HPDI <- apply(p,2,HPDI,prob=0.9)
rbind(p_mean,p_HPDI)
```

```
            [,1]      [,2]      [,3]      [,4]
p_mean 0.4222182 0.2680967 0.4516272 0.2680967
|0.9   0.2964644 0.1792636 0.3289030 0.1792636
0.9|   0.5415335 0.3519525 0.5679774 0.3519525
```

When there is more than one linear model, `link` will return a named list of matrices, one for each linear model symbol. For example, recoding the previous model so that it has two linear models:

R code 1.21

```
m2 <- map2stan(
    alist(
        pulled_left ~ dbinom(1,p),
        logit(p) <- a_actor[actor] + BP*prosoc_left,
        BP <- bp + bpc*condition,
        a_actor[actor] ~ dnorm( a , sigma_actor ) ,
        a ~ dnorm(0,10),
        bp ~ dnorm(0,10),
        bpc ~ dnorm(0,10),
        sigma_actor ~ dcauchy(0,1)
    ) ,
    data=d, iter=6000 , chains=3 , cores=3 , refresh=-1 )

p <- link(m2,n=2000)
str(p)
```

```
List of 2
 $ p : num [1:2000, 1:504] 0.316 0.347 0.472 0.409 0.43 ...
 $ BP: num [1:2000, 1:504] 0.964 0.221 0.667 0.376 0.852 ...
```

The first matrix `p` is just like before, because it uses the values of the second, `BP`, in its own calculations.

1.4.2. `sim`. The function `link` produces posterior distributions of linear models. To simulate observations that average over these posterior distributions, use `sim`. Here's an example, continuing with the model from the previous section.

R code 1.22

```
sim_pulls <- sim(m,data=new_data)
str(sim_pulls)
```

```
 int [1:1000, 1:4] 0 0 1 0 0 0 0 0 0 1 ...
```

Rows are individual simulations and columns are observations. There are four here, because again we used the four treatments in `new_data` from the previous section. All of these simulations produced 0 or 1, because this is a logistic regression. The average value in each column should be pretty close to corresponding value in `p_mean` from the previous section:

R code 1.23

```
sim_mean <- apply(sim_pulls,2,mean)
cbind(sim_mean,p_mean)
```

```
     sim_mean    p_mean
[1,]    0.441 0.4222182
[2,]    0.283 0.2680967
[3,]    0.420 0.4516272
[4,]    0.254 0.2680967
```

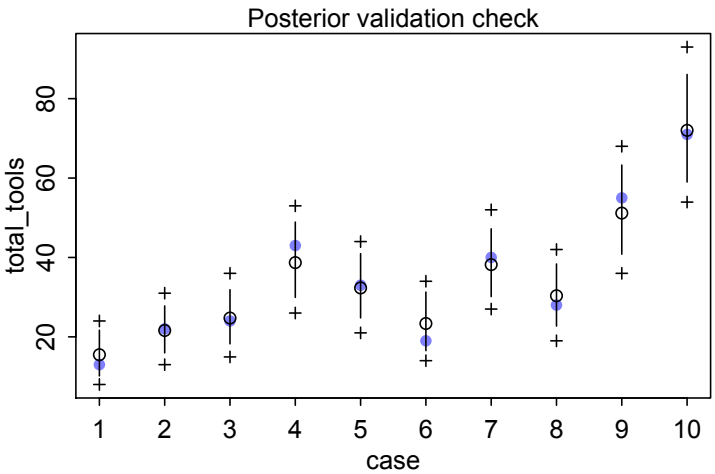
What `sim` does is identify the likelihood function in the model formula and then use the corresponding random distribution function to simulate from it. So in this case the `dbinom` likelihood is converted to `rbinom`. So if there is no random distribution function corresponding the likelihood you named in the formula definition, `sim` will not work.

1.4.3. *Posterior validation with* `postcheck`. A quick but unattractive comparison of raw data with posterior predictive distributions can be plotted with `postcheck`. This function just calls `link` and `sim` for a fit model and then plots observations on the horizontal axis and values on the vertical. It uses automatic paging, in the event there are many observations to show. The number of observations on each page is controlled by the `window` argument. See `?postcheck` for details.

Here's an example, using a varying intercepts (over-dispersed) Poisson model with log-link:

R code 1.24

```
data(Kline)
Kline$log_pop <- log(Kline$population)
Kline$id <- 1:10
m <- map2stan(
    alist(
        total_tools ~ dpois(lambda),
        log(lambda) <- a + a_society[id] + b*log_pop,
        a_society[id] ~ dnorm(0,sigma),
        a ~ dnorm(0,10),
        b ~ dnorm(0,1),
        sigma ~ dcauchy(0,2.5)
    ),
    data=Kline , chains=3 , core=3 , refresh=-1 )
postcheck(m)
```



Each case on the horizontal is a row in the data. The vertical axis is a count scale, implied by the Poisson likelihood in the model definition. Each blue point is a raw data value. Each open circle is a posterior mean prediction. The vertical line segments are 89% posterior intervals of the mean prediction. Finally, the + marks show the 89% interval of simulated observations (from `sim` output).

1.4.4. *Counter-factual predictions with* `link` *and* `sim`. It is often more useful to plot smooth counter-factual posterior predictions across a range of a predictor variable. Here's an example, using the same data as the example just above, but with a simpler model:

R code 1.25

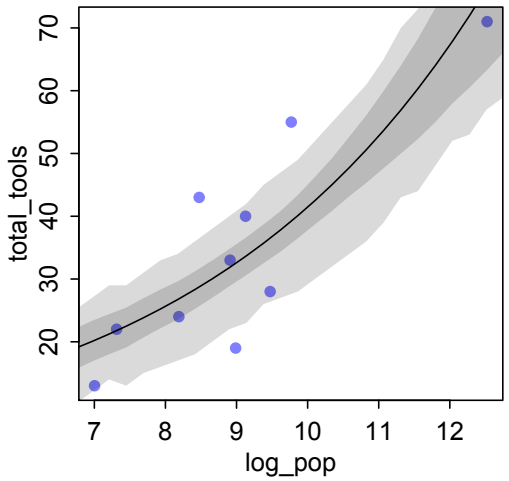
```
m <- map2stan(
    alist(
        total_tools ~ dpois(lambda),
        log(lambda) <- a + b*log_pop,
        a ~ dnorm(0,10),
        b ~ dnorm(0,1)
    ),
    data=Kline , chains=1 , warmup=500 , iter=1e4 )

log_pop.seq <- seq(from=6,to=13,length.out=30)
new_data <- data.frame(log_pop=log_pop.seq)

lambda <- link(m,data=new_data,n=9000)
lambda.mean <- apply(lambda,2,mean)
lambda.HPDI <- apply(lambda,2,HPDI)
tt.sim <- sim(m,data=new_data,n=9000)
tt.HPDI <- apply(tt.sim,2,HPDI)

plot( total_tools ~ log_pop , data=Kline , pch=16 , col=rangi2 )
lines( log_pop.seq , lambda.mean )
shade( lambda.HPDI , log_pop.seq )
shade( tt.HPDI , log_pop.seq )
```

This is the result:



Note that the lighter shaded region, the simulated Poisson observation interval, is jagged. This is because it is a count outcome, so only integers are simulated. You can see this in the raw simulations:

R code 1.26
```
str(tt.sim)
```

```
 int [1:9000, 1:30] 12 8 13 16 25 14 15 17 12 16 ...
```

One thing that is easy to appreciate from the counterfactual plot above is that the predictions are under-dispersed, relative to the data.

1.5. **WAIC/LOO/DIC model comparison and averaging.** The `rethinking` package provides `DIC`, `WAIC`, and `LOO` methods to compute the Deviance Information Criterion (DIC), Widely Applicable Information Criterion (WAIC), and Pareto-smoothed importance sampling Leave-One-Out cross-validation (LOO) from posterior samples for both `map` and `map2stan` model fits. The convenience functions `compare` and `ensemble` then simplify the process of comparing sets of models and constructing prediction ensembles using model weights based upon any of these criteria. Both `compare` and `ensemble` default to use WAIC. The textbook contains a lot more detail on these metrics and how the ensembles are constructed. Examples follow.

1.5.1. `compare`. The `compare` function accepts a series of fit `map` or `map2stan` models. It computes WAIC (or DIC) for each and then constructs a table that expedites comparison. Here's an example, constructing three models fit to the same data. These data and models are detailed in Chapter 7 of the textbook.

R code 1.27
```
data(rugged)
# remove cases with missing values
d <- rugged[ complete.cases(rugged$rgdppc_2000) , ]
d$log_gdp <- log(d$rgdppc_2000)
dlist <- list(
    log_gdp = d$log_gdp,
    rugged = d$rugged,
    africa = d$cont_africa
)

# intercept-only model
m1 <- map2stan(
    alist(
        log_gdp ~ dnorm(mu,sigma),
        mu <- a,
        a ~ dnorm(0,100),
        sigma ~ dcauchy(0,2.5)
    ),
    data=dlist ,
    warmup=1000 , iter=5000 , chains=2 , cores=2 , refresh=-1 )

# model with terrain ruggedness
m2 <- map2stan(
    alist(
        log_gdp ~ dnorm(mu,sigma),
        mu <- a + bR*rugged,
        a ~ dnorm(0,100),
```

```
        bR ~ dnorm(0,10),
        sigma ~ dcauchy(0,2.5)
    ),
    data=dlist ,
    warmup=1000 , iter=5000 , chains=2 , cores=2 , refresh=-1 )

# model with Africa indicator
m3 <- map2stan(
    alist(
        log_gdp ~ dnorm(mu,sigma),
        mu <- a + bA*africa,
        a ~ dnorm(0,100),
        bA ~ dnorm(0,10),
        sigma ~ dcauchy(0,2.5)
    ),
    data=dlist ,
    warmup=1000 , iter=5000 , chains=2 , cores=2 , refresh=-1 )

# model with both terrain ruggedness and Africa indicator
# note vector prior notation for both coefficients
m4 <- map2stan(
    alist(
        log_gdp ~ dnorm(mu,sigma),
        mu <- a + bR*rugged + bA*africa,
        a ~ dnorm(0,100),
        c(bR,bA) ~ dnorm(0,10),
        sigma ~ dcauchy(0,2.5)
    ),
    data=dlist ,
    warmup=1000 , iter=5000 , chains=2 , cores=2 , refresh=-1 )

# model with interaction of terrain ruggedness and Africa indicator
m5 <- map2stan(
    alist(
        log_gdp ~ dnorm(mu,sigma),
        mu <- a + bR*rugged + bA*africa + bRA*rugged*africa,
        a ~ dnorm(0,100),
        c(bR,bA,bRA) ~ dnorm(0,10),
        sigma ~ dcauchy(0,2.5)
    ),
    data=dlist ,
    warmup=1000 , iter=5000 , chains=2 , cores=2 , refresh=-1 )
```

You may want to verify that the chains for each model sampled correctly. Now we can invoke `compare`:

```
compare(m1,m2,m3,m4,m5)
```

R code 1.28

```
    WAIC pWAIC dWAIC weight    SE   dSE
```

```
m5 469.6   5.3   0.0   0.88 14.89    NA
m3 474.2   2.5   4.6   0.09 14.70  7.48
m4 476.2   4.3   6.5   0.03 15.01  6.86
m1 537.1   1.4  67.5   0.00 12.88 15.48
m2 539.6   2.6  69.9   0.00 12.99 15.47
```

The columns are:

- `WAIC`: The WAIC value of each model, sorted from small (better) to large (worse)
- `pWAIC`: The "effective" number of parameters for each model. This is a measure of the model's flexibility in fitting the sample.
- `dWAIC`: Difference between each WAIC and the smallest WAIC in the set.
- `weight`: The Akaike weight of each model in the set, conditional on the set.
- `SE`: The estimated standard error of each WAIC.
- `dSE`: The standard error of the different between each WAIC and the smallest WAIC in the set. Often useful to compare these values to each corresponding `dWAIC`.

The full `dSE` matrix, for a pairs of models, is available from the `dSE` slot of the invisible result that `compare` produces:

R code 1.29

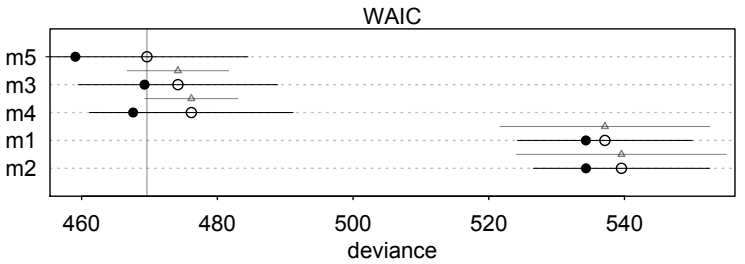
```
compare_table <- compare(m1,m2,m3,m4,m5)
compare_table@dSE
```

```
           m1         m2        m3        m4        m5
m1         NA  0.5317339 14.918201 15.154642 15.482533
m2  0.5317339         NA 14.819779 15.080354 15.473666
m3 14.9182006 14.8197790        NA  3.041996  7.483212
m4 15.1546423 15.0803543  3.041996        NA  6.855100
m5 15.4825326 15.4736656  7.483212  6.855100        NA
```

A crude visual representation of the information in the `compare` table is available using `plot`:

R code 1.30

```
plot(compare(m1,m2,m3,m4,m5))
```



The solid points are in-sample deviance values (not shown in the table, but implicitly equal to `WAIC`−2`pWAIC`). The open points are the `WAIC` values. Dark line segments on each row show plus-and-minus one standard error of WAIC. The points and gray line segments between rows show the `dWAIC` and `dSE` values.

1.5.2. `ensemble`. Model selection is often unwise, because of high variance in information criteria and the loss of model uncertainty. Model averaging often performs better. This

means however averaging predictions, not parameter values. One way to do this is to construct a mix of posterior predictions weighted by Akaike weights. The `ensemble` function operates similarly to `link` and `sim` and automates this procedure.

Continuing with the example from just above, let's construct an ensemble of counterfactual posterior predictions. We need to visualize predictions for a nation in Africa and for a nation not in Africa. I'll show the code for one of these. Changing the value of the `africa` indicator in the data will produce the alternative plot.

R code 1.31

```
rugged.seq <- seq(from=0,to=7,length.out=30)
new_data <- data.frame(
    africa = 1, # imaginary nation in Africa
    rugged = rugged.seq
)

# change n argument to change number of samples used and smooth results
m1to5_ensemble <- ensemble(m1,m2,m3,m4,m5,data=new_data,n=3000)
```

The result, stored here in `m1to5_ensemble`, contains two named matrices, `link` and `sim`, each corresponding to the model-averaged output of each simpler function. These matrices are summarized and plotted just like the simpler examples.

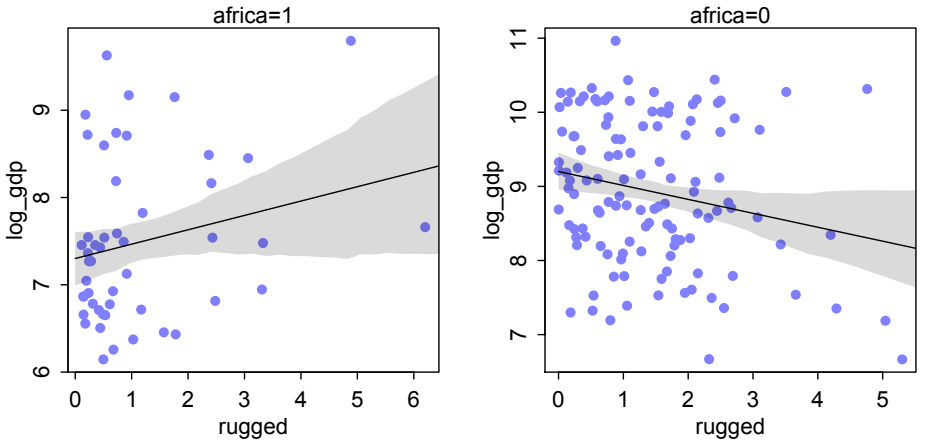
R code 1.32

```
mu.mean <- apply( m1to5_ensemble$link , 2 , mean )
mu.HPDI <- apply( m1to5_ensemble$link , 2 , HPDI )

# extract just raw data with same africa value as new_data
africa_val <- new_data$africa[1]
plot_data <- d[ d$cont_africa==africa_val , c("log_gdp","rugged") ]
plot( log_gdp ~ rugged , data=plot_data , pch=16 , col=rangi2 )
mtext( concat("africa=",africa_val) )

# overlay counter-factual predictions
lines( rugged.seq , mu.mean )
shade( mu.HPDI , rugged.seq )
```

Here are both results, for `africa=1` and `africa=0`:



Since model m5 has more than 75% of the Akaike weight, it dominates predictions. But notice the asymmetric uncertainty in the slope in the righthand plot. This arise from including the lower-ranked models in the ensemble. In particular, model m3 is second-ranked, and it ignores rugged completely.

1.6. **Missing data imputation with `map2stan`.** map2stan supports basic automated Bayesian imputation, much like BUGS and JAGS. Normally, data with missing values (NA) will prevent a model from running. Here's an example:

R code 1.33

```
data(milk)
d <- list(
    kcal = milk$kcal.per.g,
    log_mass = log(milk$mass),
    neocortex = milk$neocortex.perc/100
)
m <- map2stan(
    alist(
        kcal ~ dnorm(mu,sigma),
        mu <- a + bM*log_mass + bN*neocortex,
        a ~ dnorm(0,100),
        c(bM,bN) ~ dnorm(0,10),
        sigma ~ dcauchy(0,2.5)
    ),
    data=d , warmup=500 , iter=2500 , chains=3 , cores=3 , refresh=-1 )
```

```
Error in map2stan(alist(kcal ~ dnorm(mu, sigma), mu <- a + bM * log_mass +  :
  Variable 'neocortex' has missing values (NA) in:
a + bM * log_mass[i] + bN * neocortex
Either remove cases with NA or declare a distribution to use for imputation.
```

As the error message states, there are missing values in neocortex. We must either remove all the cases with missing values (this is what most of the automated tools in R do automatically and dangerously) or either provide a distribution for imputation. The second option means to state a distribution for the predictor variable neocortex. Like this:

R code 1.34

```
m2 <- map2stan(
    alist(
        kcal ~ dnorm(mu,sigma),
        mu <- a + bM*log_mass + bN*neocortex,
        neocortex ~ dnorm(mu_N,sigma_N),
        mu_N ~ dnorm(0,10),
        sigma_N ~ dcauchy(0,1),
        a ~ dnorm(0,100),
        c(bM,bN) ~ dnorm(0,10),
        sigma ~ dcauchy(0,2.5)
    ),
    data=d , warmup=500 , iter=2500 , chains=3 , cores=3 , refresh=-1 )
```

After a bunch of messages, the posterior distribution will contain one parameter for each missing value in neocortex:

R code 1.35

```
precis(m2,depth=2)
```

```
                      Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
neocortex_impute[1]   0.63   0.05       0.55       0.71  4864    1
neocortex_impute[2]   0.63   0.05       0.54       0.70  3855    1
neocortex_impute[3]   0.62   0.05       0.53       0.70  3391    1
neocortex_impute[4]   0.65   0.05       0.58       0.73  4291    1
neocortex_impute[5]   0.70   0.05       0.62       0.78  5253    1
neocortex_impute[6]   0.66   0.05       0.58       0.73  5285    1
neocortex_impute[7]   0.69   0.05       0.61       0.76  5339    1
neocortex_impute[8]   0.70   0.05       0.62       0.77  5533    1
neocortex_impute[9]   0.71   0.05       0.64       0.79  4730    1
neocortex_impute[10]  0.65   0.05       0.57       0.72  4356    1
neocortex_impute[11]  0.66   0.05       0.58       0.73  4216    1
neocortex_impute[12]  0.70   0.05       0.61       0.77  4786    1
mu_N                  0.67   0.01       0.65       0.69  4182    1
sigma_N               0.06   0.01       0.04       0.08  2313    1
a                    -0.55   0.47      -1.30       0.19  1212    1
bM                   -0.07   0.02      -0.11      -0.03  1705    1
bN                    1.93   0.74       0.74       3.07  1199    1
sigma                 0.13   0.02       0.09       0.17  1911    1
```

For details on the algorithm used, either inspect the raw Stan code (`stancode(m2)`) or see Chapter 14 in the textbook, where this example is explained in depth.

1.7. **`glimmer`: From `glmer` to `map2stan`.** For users who are new to full model specifications of the sort needed to define Bayesian models, the function `glimmer` translates mixed model formulas of the kind used by `glm` or `glmer` (in the `lme4` package) into full formula lists of the kind used by both `map` and `map2stan`. See `?glimmer` for details and examples.

2. Model specification examples

The sections to follow present examples of different standard model types. Each example contains a mathematical specification of the model, as well as working R code to implement it using `map2stan`. Many of these models will work equally well in `map`, provided there are no hyperparameters that MAP estimation cannot handle correctly. All of these model types are explained in more detail in the textbook. The material here is a technical reference.

[These sections will be filled out with examples eventually. Please bear with me as I find time to work on it.]

2.1. **Linear models.** Linear models are simple and universal. In mathematical form:

$$
\begin{aligned}
Y_i &\sim \text{Normal}(\mu_i, \sigma) \\
\mu_i &= \alpha + \beta X_i \\
\alpha &\sim \text{Normal}(0, 100) \\
\beta &\sim \text{Normal}(0, 1) \\
\sigma &\sim \text{HalfCauchy}(0, 2.5)
\end{aligned}
$$

And this is the corresponding `map2stan` implementation:

R code 2.1

```
library(rethinking)
data(Howell1)
f_lm <- alist(
    height ~ dnorm(mu,sigma),
    mu <- a + b*weight,
    a ~ dnorm(170,100),
    b ~ dnorm(0,1),
    sigma ~ dcauchy(0,2.5)
)
set.seed(1) # just for the example, so results repeatable
m_lm <- map2stan( f_lm , data=Howell1 , refresh=-1 )
precis(m_lm)
```

```
       Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
a     75.54   1.00      73.85      76.92   258    1
b      1.76   0.03       1.72       1.81   241    1
sigma  9.38   0.27       8.93       9.79   367    1
```

2.2. **Multilevel (mixed effects) models.** The simplest multilevel model includes varying intercepts that are adaptively regularized by a Gaussian prior learned from the data. In mathematical form:

$$\begin{aligned} Y_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha_{\text{GROUP}[i]} + \beta X_i \\ \alpha_{\text{GROUP}} &\sim \text{Normal}(\alpha, \tau) \\ \alpha &\sim \text{Normal}(0, 100) \\ \beta &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{HalfCauchy}(0, 2.5) \\ \tau &\sim \text{HalfCauchy}(0, 2.5) \end{aligned}$$

And this is the corresponding `map2stan` implementation:

R code 2.2

```
f_lmm <- alist(
    height ~ dnorm(mu,sigma),
    mu <- a_group[group] + b*weight,
    a_group[group] ~ dnorm(a,tau),
    a ~ dnorm(170,100),
    b ~ dnorm(0,1),
    sigma ~ dcauchy(0,2.5),
    tau ~ dcauchy(0,2.5)
)
```

The `group` variable should comprise a contiguous set of index values for the groups in the data.

More than one type of grouping variable can be added in the same way. But be cautious about the overall mean intercept. Here I take the mean out of the prior and place it in the linear model, to avoid identification issues:

R code 2.3

```
f_lmm2 <- alist(
    height ~ dnorm(mu,sigma),
    mu <- a + a_group1[group1] + a_group2[group2] + b*weight,
    a_group1[group1] ~ dnorm(0,tau1),
    a_group2[group2] ~ dnorm(0,tau2),
    a ~ dnorm(170,100),
    b ~ dnorm(0,1),
    sigma ~ dcauchy(0,2.5),
    tau1 ~ dcauchy(0,2.5),
    tau2 ~ dcauchy(0,2.5)
)
```

To introduce varying slopes, a multivariate Gaussian prior can be used. Here is the recommended form, using separate priors for the standard deviations and the correlation matrix:

R code 2.4

```
f_lmm3 <- alist(
    height ~ dnorm(mu,sigma),
    mu <- a_group[group] + b_group[group]*weight,
    c(a_group,b_group)[group] ~ dmvnorm2(c(a,b),tau,Rho),
    a ~ dnorm(170,100),
    b ~ dnorm(0,1),
    sigma ~ dcauchy(0,2.5),
    tau ~ dcauchy(0,2.5),
    Rho ~ dlkjcorr(2)
)
```

It is sometimes more efficient in sampling to move the means `a` and `b` to the linear model:

R code 2.5

```
f_lmm4 <- alist(
    height ~ dnorm(mu,sigma),
    mu <- a + a_group[group] + (b + b_group[group])*weight,
    c(a_group,b_group)[group] ~ dmvnorm2(0,tau,Rho),
    a ~ dnorm(170,100),
    b ~ dnorm(0,1),
    sigma ~ dcauchy(0,2.5),
    tau ~ dcauchy(0,2.5),
    Rho ~ dlkjcorr(2)
)
```

This change means that the parameters `a_group` and `b_group` will now be offsets from the mean.

A fully non-centered parameterization is available by using the `dmvnormNC` density:

R code 2.6

```
f_lmm5 <- alist(
    height ~ dnorm(mu,sigma),
    mu <- a + a_group[group] + (b + b_group[group])*weight,
```

```
    c(a_group,b_group)[group] ~ dmvnormNC(tau,Rho),
    a ~ dnorm(170,100),
    b ~ dnorm(0,1),
    sigma ~ dcauchy(0,2.5),
    tau ~ dcauchy(0,2.5),
    Rho ~ dlkjcorr(2)
)
```

Internally, `dmvnormNC` uses a Cholesky factor `L_Rho` and a matrix of z-scores to perform sampling. Take a look at the raw Stan code with `stancode` for details.

2.3. **Logistic and binomial models.** Models that use Bernoulli and binomial likelihoods typically employ a logit link function. For bernoulli outcomes, a simple GLM takes this form:

R code 2.7

```
f_glm_bernoulli <- alist(
    y ~ dbinom(1,p),
    logit(p) <- a + b*x,
    a ~ dnorm(0,1),
    b ~ dnorm(0,1)
)
```

And aggregated binomial outcomes take this form:

R code 2.8

```
f_glm_binomial <- alist(
    y ~ dbinom(n,p),
    logit(p) <- a + b*x,
    a ~ dnorm(0,1),
    b ~ dnorm(0,1)
)
```

The data vector `n` should hold the number of trials for each observation. Varying effects can be added as usual.

2.4. **Poisson models.** Poisson outcome models take the form:

R code 2.9

```
f_glm_pois <- alist(
    y ~ dpois(lambda),
    log(lambda) <- a + b*x,
    a ~ dnorm(0,10),
    b ~ dnorm(0,1)
)
```

This uses a conventional log link function.

To add an offset (exposure), just add the data vector to the linear model:

R code 2.10

```
f_glm_pois <- alist(
    y ~ dpois(lambda),
    log(lambda) <- offset + a + b*x,
```

```
    a ~ dnorm(0,10),
    b ~ dnorm(0,1)
)
```

Here `offset` should be a vector of offsets, one for each value in the outcome `y`.

2.5. **Beta-binomial and gamma-Poisson models.** Both beta-binomial and gamma-Poisson likelihoods are straightforward, but be aware that the scale parameters in each case are often weakly identified. Using good initial values and appropriately regularizing priors are typically needed to get sensible results.

Here's a template for beta-binomial:

R code 2.11

```
f_glm_bbinom <- alist(
    y ~ dbetabinom(n,p,theta),
    logit(p) <- a + b*x,
    a ~ dnorm(0,10),
    b ~ dnorm(0,1),
    theta ~ dexp(2)
)
```

For a `map2stan` fit, the scale parameter `theta` will be constrained to positive reals, unless a custom constraint is provided. When using `map`, including a log link inline is recommended. For example:

R code 2.12

```
f_glm_bbinom2 <- alist(
    y ~ dbetabinom(n,p,exp(log_theta)),
    logit(p) <- a + b*x,
    a ~ dnorm(0,10),
    b ~ dnorm(0,1),
    log_theta ~ dnorm(0,1)
)
```

Note the impact on the prior, as the parameter is now defined for all reals, and so needs a prior defined for all reals. The example here is not a recommended prior. You'll need to choose wisely for each context.

Here's a template for gamma-Poisson:

R code 2.13

```
f_glm_gampois <- alist(
    y ~ dgampois(mu,scale),
    log(mu) <- a + b*x,
    a ~ dnorm(0,10),
    b ~ dnorm(0,1),
    scale ~ dexp(2)
)
```

For a `map2stan` fit, the scale parameter `scale` will be constrained to positive reals, unless a custom constraint is provided. When using `map`, including a log link inline is recommended.

2.6. **Gamma and exponential models.** Here is a simple gamma distributed GLM:

R code 2.14

```
f_glm_gamma <- alist(
    y ~ dgamma2(mu,scale),
    log(mu) <- a + b*x,
    a ~ dnorm(0,10),
    b ~ dnorm(0,1),
    scale ~ dexp(2)
)
```

And an exponential outcome model is very similar:

R code 2.15

```
f_glm_exp <- alist(
    y ~ dexp(lambda),
    log(lambda) <- a + b*x,
    a ~ dnorm(0,10),
    b ~ dnorm(0,1)
)
```

2.7. **Zero-inflated Poisson and binomial models.** x

2.8. **Zero-augmented gamma models.** x

2.9. **Multinomial logit models.** x

2.10. **Ordered logit models.** x

2.11. **Gaussian processes.** x

2.12. **Item response theory.** x

2.13. **Factor analysis.** x