# R Stan: First Examples

## *ICPSR Summer Program at York University*

***Georges Monette***

***July 17 to 21, 2017***

(Updated: July 16 2017 14:36)

Load packages

```r
library(rstan)
```

```
## Loading required package: ggplot2
```

```
## Loading required package: StanHeaders
```

```
## rstan (Version 2.15.1, packaged: 2017-04-19 05:03:57 UTC, GitRev: 2e1f913d3ca3)
```

```
## For execution on a local, multicore CPU with excess RAM we recommend calling
## rstan_options(auto_write = TRUE)
## options(mc.cores = parallel::detectCores())
```

```r
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
# windowsFonts(Arial=windowsFont("TT Arial"))
library(spida2)
```

```
## 
## Attaching package: 'spida2'
```

```
    The following object is masked from 'package:ggplot2':

        labs
```

```
# Install the loo package if necessary
# install.packages('loo')
library(loo)
```

```
    This is loo version 1.1.0
```

## Height and Weight Example —-

Artificial data set with archetypal outliers

We use the subset with no outliers to start then we look at things we can do with a data set with an outlier.

```
data(hw)
head(hw)
```

```
     Height Weight Health Type Outlier
   1 0.6008 0.3355  1.280    0    none
   2 0.9440 0.6890  1.208    0    none
   3 0.6150 0.6980  1.036    0    none
   4 1.2340 0.7617  1.395    0    none
   5 0.7870 0.8910  0.912    0    none
   6 0.9150 0.9330  1.175    0    none
```

```
dd <- subset(hw, Type == 0)  # no outliers
if(interactive) {
  library(p3d)
  Init3d()
  Plot3d(Health ~ Weight + Height | Outlier, hwoutliers)
}
```

Fit the model using data with no outliers

```
fit <- lm(Health ~ Weight + Height, dd)
summary(fit)
```

```
Call:
lm(formula = Health ~ Weight + Height, data = dd)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.29163 -0.05797  0.01300  0.07795  0.17900 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   1.0151     0.1158   8.770 1.45e-06 ***
Weight       -0.7856     0.1680  -4.676 0.000536 ***
Height        0.8360     0.1851   4.516 0.000706 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1368 on 12 degrees of freedom
Multiple R-squared:  0.6577, Adjusted R-squared:  0.6006 
F-statistic: 11.53 on 2 and 12 DF,  p-value: 0.00161
```

```
if(interactive) {
  Fit3d(fit, alpha = .5)
  fg()
}
```

# First Stan model —-

Generic Stan model for regression with improper uniform prior on betas and uniform on sigma

## Step 0: Scale data —-

A major reason for posterior sampling to go poorly is a posterior with a complex geometry.

A first step is to scale predictors so they have similar variability. As we know this prevents the confidence ellipsoids from becoming too eccentric just as a result of variable scaling. They might be eccentric because of high collinearity but at least rescaling reduces unnecessary eccentricity.

Unless you run into serious problems, you shouldn't use automated methods. Just rescale to units that produce a similar variability. It can also be helpful to centre variable to a meaningful value.

## Step 1: Write Model in Stan —-

Write the model using the Stan Modeling Language and save it to a file called 'first.stan' Three basic blocks

- data block (optional)
- parameters block (optional)
- model block (required)

Later:

- transformed data
- transformed parameters
- generated quantities

- functions (maybe)

Here we're using 'cat' but we could have writen the Stan program directly in the 'first.stan' file. That's considered a better practice but would have been unwieldy for a workshop.

```
cat( c("
  data {
    int N;   // number of observations
    vector[N] weight;   // vector of weights
    vector[N] height;   // etc.
    vector[N] health;
  }
  parameters {
    real b_0;      // intercept parameter
    real b_weight;
    real b_height;
    real<lower=0> sigma_y; // non-negative standard deviation
  }
  model {
    health ~ normal(
      b_0 + b_weight * weight + b_height * height,
      sigma_y);   // model of form y ~ normal(mean, sd)
  }
"), file = 'first.stan')
```

Notes:

- Every command must end with a semicolon
- Every variable is declared.
- We didn't specify any priors in the 'model' statement so the default priors are uniform, in this case improper
- We can specify lower and upper bounds for parameters

## Step 2: Compile the Model —-

Compile the Stan program to create an object module (Dynamic Shared Object) with C++

```
first.dso <-
  stan_model('first.stan', model_name = 'First Model')
```

The parser is amazingly generous and informative with its error messages.

Compiling takes a while because it produces optimized code that will be used to:

1. compute the height of the bowl as the skateboard moves around
2. the gradient of the bowl

which needs to be done very fast to minimize sampling time.

## Step 3: Prepare a Data List for Stan —-

Every variable declared in the 'data' step needs to be fed to Stan through a list in R

```
dat <- list(
  N = nrow(dd),
  weight = dd$Weight,
  height = dd$Height,
  health = dd$Health
)
dat
```

```
   $N
   [1] 15

   $weight
    [1] 0.3355 0.6890 0.6980 0.7617 0.8910 0.9330 0.9430 1.0060 1.0200 1.2150
   [11] 1.2230 1.2360 1.3530 1.3770 2.0734

   $height
    [1] 0.6008 0.9440 0.6150 1.2340 0.7870 0.9150 1.0490 1.1840 0.7370 1.0770
   [11] 1.1280 1.5000 1.5310 1.1500 1.9340

   $health
    [1] 1.280 1.208 1.036 1.395 0.912 1.175 1.237 1.048 1.003 0.943 0.912
   [12] 1.311 1.411 0.603 1.073
```

## Step 4: Sample From the Posterior —-

We give 4 skateboards (chains) a random shove and let them sample using HMC with NUTS.

```
first.stanfit <- sampling(first.dso, dat)
```

## Step 5: Check whether HMC worked —-

These are diagnostics to check whether Stan worked, no so much whether the model is good, although problems with Stan are often a consequence of a poor model.

- Did some chains (skateboards) get stuck far from the others
- Is there low correlation within each chain, or do they look like slow local random walks?

Initial diagnostics by printing the fit:

- Rhat: Total variability / Variability within Chains
  - values much greater 1 show that chains are not in agreement and not exploring the same regions of parameter space.
  - I have seen a suggested requirement that Rhat < 1.01 for all parameter. I get 1.02 or a bit larger on problems that I think are okay.
- n_eff: effective sample size taking serial correlation in chains in to account.
  - With default 4 chains of 2000 (1000 post-warmup), the total sample has size 4,000. So 1,000 is pretty good. If it isn't greater than 1,000, run MCMC for more iterations, especially if you are reporting results. Stan's best practices page (https://github.com/stan-dev/stan/wiki/Stan-Best-Practices) recommends that if `N_eff / N < 0.001` you should be suspicious of the calculation of `N_eff`. In our example, the two predictors are stongly correlated which results in a lower `N_eff/N`.

```
first.stanfit
```

```
   Inference for Stan model: First Model.
   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
   b_0       1.01    0.00 0.13  0.75  0.93  1.01  1.09  1.29  2015    1
   b_weight -0.79    0.00 0.20 -1.17 -0.91 -0.79 -0.66 -0.40  1650    1
   b_height  0.84    0.01 0.22  0.41  0.70  0.84  0.98  1.28  1520    1
   sigma_y   0.15    0.00 0.04  0.10  0.13  0.15  0.17  0.24  1647    1
   lp__     19.55    0.05 1.63 15.53 18.72 19.90 20.77 21.64  1252    1

   Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:36:13 2017.
   For each parameter, n_eff is a crude measure of effective sample size,
   and Rhat is the potential scale reduction factor on split chains (at
   convergence, Rhat=1).
```

compare with ‘lm’:

```
lm(Health ~ Weight + Height, dd) %>% summary
```

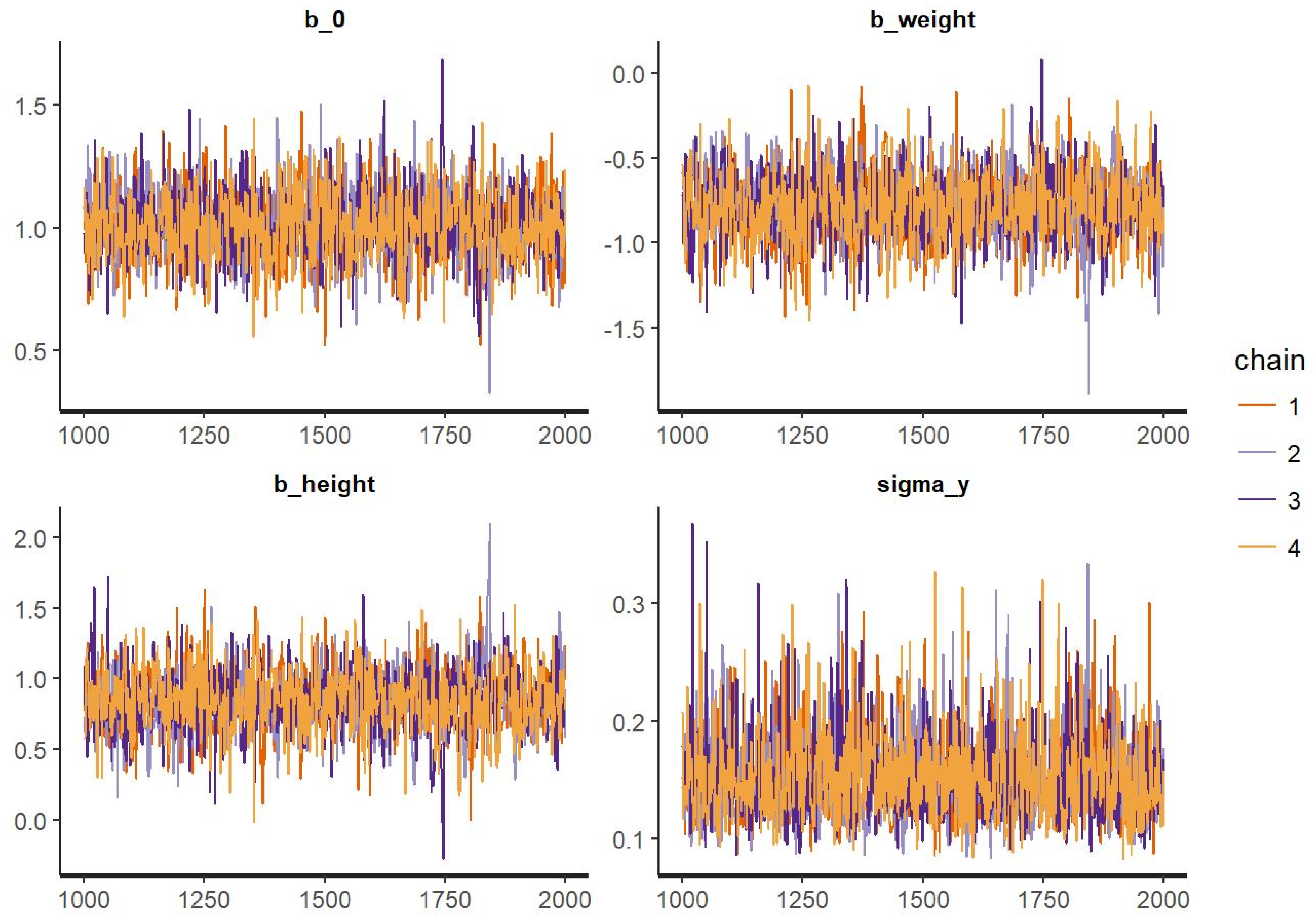
```

Call:
lm(formula = Health ~ Weight + Height, data = dd)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.29163 -0.05797  0.01300  0.07795  0.17900 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   1.0151     0.1158   8.770 1.45e-06 ***
Weight       -0.7856     0.1680  -4.676 0.000536 ***
Height        0.8360     0.1851   4.516 0.000706 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1368 on 12 degrees of freedom
Multiple R-squared:  0.6577, Adjusted R-squared:  0.6006 
F-statistic: 11.53 on 2 and 12 DF,  p-value: 0.00161
```

```
first.stanfit  %>%  print(digits=4)
```

```
Inference for Stan model: First Model.
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%
b_0       1.0111  0.0030 0.1342  0.7480  0.9271  1.0119  1.0922  1.2882
b_weight -0.7860  0.0049 0.1978 -1.1705 -0.9110 -0.7903 -0.6583 -0.3982
b_height  0.8396  0.0056 0.2175  0.4054  0.7009  0.8365  0.9750  1.2753
sigma_y   0.1540  0.0009 0.0364  0.0997  0.1280  0.1484  0.1734  0.2397
lp__     19.5515  0.0460 1.6276 15.5313 18.7249 19.8964 20.7683 21.6426
         n_eff   Rhat
b_0       2015 1.0005
b_weight  1650 1.0008
b_height  1520 1.0017
sigma_y   1647 1.0005
lp__      1252 1.0016

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:36:13 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

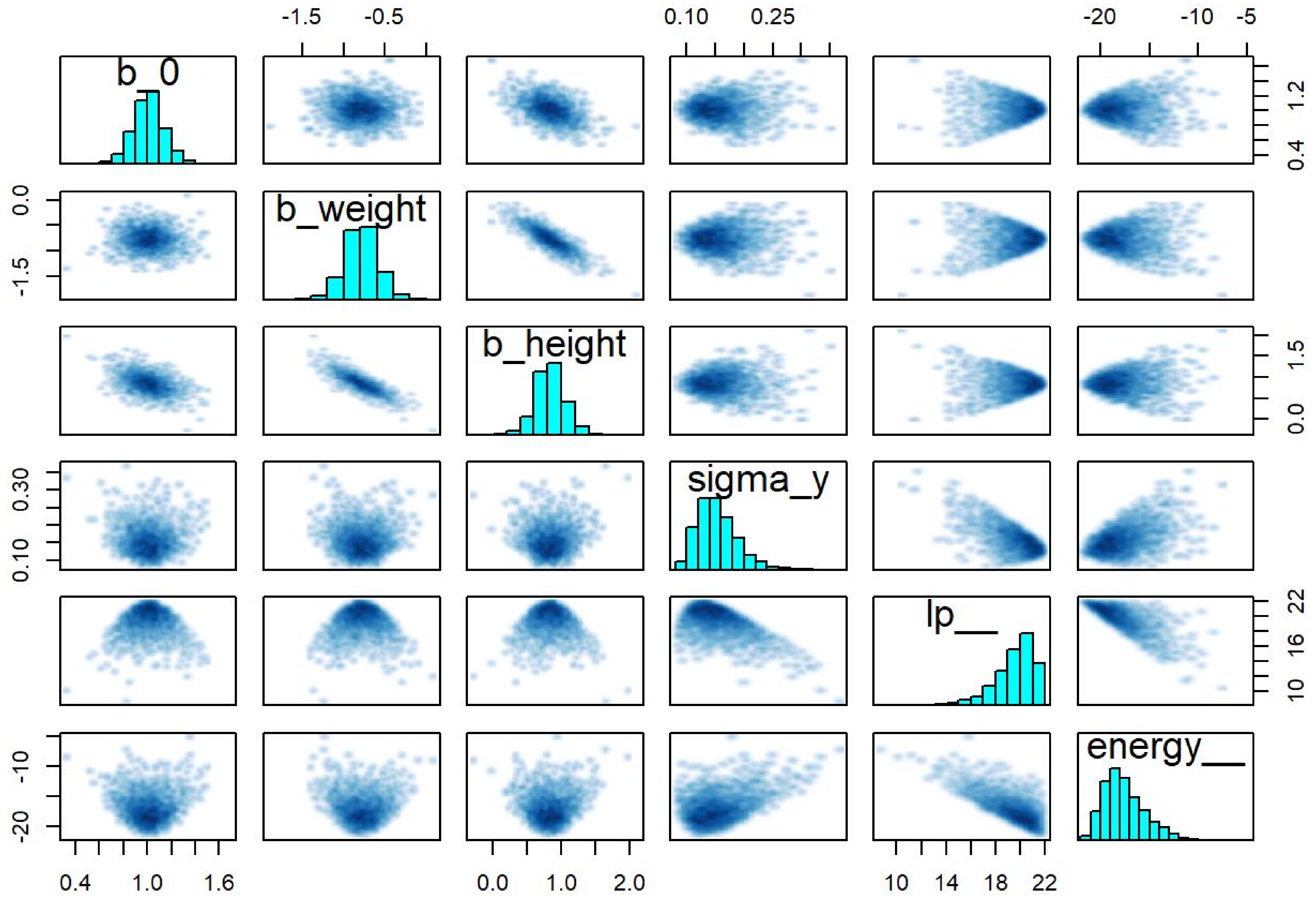
Graphical diagnostics:

```
traceplot(first.stanfit)
```



Note: skew in sigma_y, as expected

```
pairs(first.stanfit)
```



- Lower diagonal contains draws with acceptance probability below the median.
- Points leading to divergent transitions are in red
- Yellow points reached maximum tree depth without a U-turn

## Generic regression model with Stan —-

Here’s a a Stan model that can do OLS regression given a Y vector and an X matrix

```
cat( c("
data {
  int N;   // number of observations
  int P;   // number of columns of X matrix (including intercept)
  matrix[N,P] X;   // X matrix including intercept
  vector[N] Y;  // response
}
parameters {
  vector[P] beta; // default uniform prior if nothing specied in model
  real <lower=0> sigma;
}
model {
  Y ~ normal( X * beta, sigma );
    // note that * is matrix mult.
    // For elementwise multiplication use .*

  // To do ridge regression, throw in  a prior
  // You can vary the variance parameter through data
  // or you can turn it into a parameter with a prior:

  //  beta ~ normal(0, 3);

}
"), file = 'ols.stan')
```

Create reg_model 'dynamic shared object module' which is compiled C++ code that generates HMC samples from the posterior distribution

```
system.time(
ols.dso <- stan_model('ols.stan')
)
```

```
      user  system elapsed
      0.19    0.00    0.22
```

```
#
# Prepare the data list
# striplevels
X <- cbind(1, dd$Weight, dd$Height)
dat <- list(
  N = nrow(X),
  P = ncol(X),
  X = X,
  Y = dd$Health)
dat
```

```
$N
[1] 15

$P
[1] 3

$X
      [,1]   [,2]   [,3]
 [1,]    1 0.3355 0.6008
 [2,]    1 0.6890 0.9440
 [3,]    1 0.6980 0.6150
 [4,]    1 0.7617 1.2340
 [5,]    1 0.8910 0.7870
 [6,]    1 0.9330 0.9150
 [7,]    1 0.9430 1.0490
 [8,]    1 1.0060 1.1840
 [9,]    1 1.0200 0.7370
[10,]    1 1.2150 1.0770
[11,]    1 1.2230 1.1280
[12,]    1 1.2360 1.5000
[13,]    1 1.3530 1.5310
[14,]    1 1.3770 1.1500
[15,]    1 2.0734 1.9340

$Y
```

```
 [1] 1.280 1.208 1.036 1.395 0.912 1.175 1.237 1.048 1.003 0.943 0.912
[12] 1.311 1.411 0.603 1.073
```

```
ols.fit <- sampling(ols.dso, dat)
ols.fit
```

```
Inference for Stan model: ols.
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
beta[1]  1.01    0.00 0.13  0.76  0.93  1.02  1.10  1.26  1995    1
beta[2] -0.78    0.00 0.19 -1.16 -0.90 -0.78 -0.66 -0.38  1586    1
beta[3]  0.83    0.01 0.21  0.40  0.69  0.84  0.97  1.24  1486    1
sigma    0.15    0.00 0.04  0.10  0.13  0.15  0.17  0.24  1673    1
lp__    19.65    0.05 1.59 15.62 18.85 20.00 20.83 21.62  1225    1

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:36:33 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

ols.dso could be used with any X matrix

Define a function that returns the posterior mean prediction

```
fun <- function(stanfit) {
  post <- get_posterior_mean(stanfit)
  beta <- post[,ncol(post)]  # use last column (all chains)
  function(Weight, Height) beta[1] + beta[2] * Weight + beta[3] * Height
}
fun(ols.fit)  # this is a closure
```

```
   function(Weight, Height) beta[1] + beta[2] * Weight + beta[3] * Height
   <environment: 0x0000000019f9df18>
```

```
fun(ols.fit)(2,3)
```

```
    beta[1]
   1.947643
```

```r
if(interactive) {
  Init3d()
  Plot3d(Health ~ Weight + Height, subset(hw, Type ==3 ))
  Fit3d(fun(ols.fit), col = 'grey')
}
#
# Including Type 3 outlier
#
hw3 <- subset(hw, Type == 3)
hw3
```

```
     Height Weight Health Type Outlier
  48 0.6008 0.3355  1.280    3  Type 3
  49 0.9440 0.6890  1.208    3  Type 3
  50 0.6150 0.6980  1.036    3  Type 3
  51 1.2340 0.7617  1.395    3  Type 3
  52 0.7870 0.8910  0.912    3  Type 3
  53 0.9150 0.9330  1.175    3  Type 3
  54 1.0490 0.9430  1.237    3  Type 3
  55 1.1840 1.0060  1.048    3  Type 3
  56 0.7370 1.0200  1.003    3  Type 3
  57 1.0770 1.2150  0.943    3  Type 3
  58 1.1280 1.2230  0.912    3  Type 3
  59 1.5000 1.2360  1.311    3  Type 3
  60 1.5310 1.3530  1.411    3  Type 3
  61 1.1500 1.3770  0.603    3  Type 3
  62 0.2000 1.9000  1.900    3  Type 3
  63 1.9340 2.0734  1.073    3  Type 3
```

```
head( Xmat3 <- model.matrix(Health ~ Weight + Height, hw3) )
```

```
   (Intercept) Weight Height
48           1 0.3355 0.6008
49           1 0.6890 0.9440
50           1 0.6980 0.6150
51           1 0.7617 1.2340
52           1 0.8910 0.7870
53           1 0.9330 0.9150
```

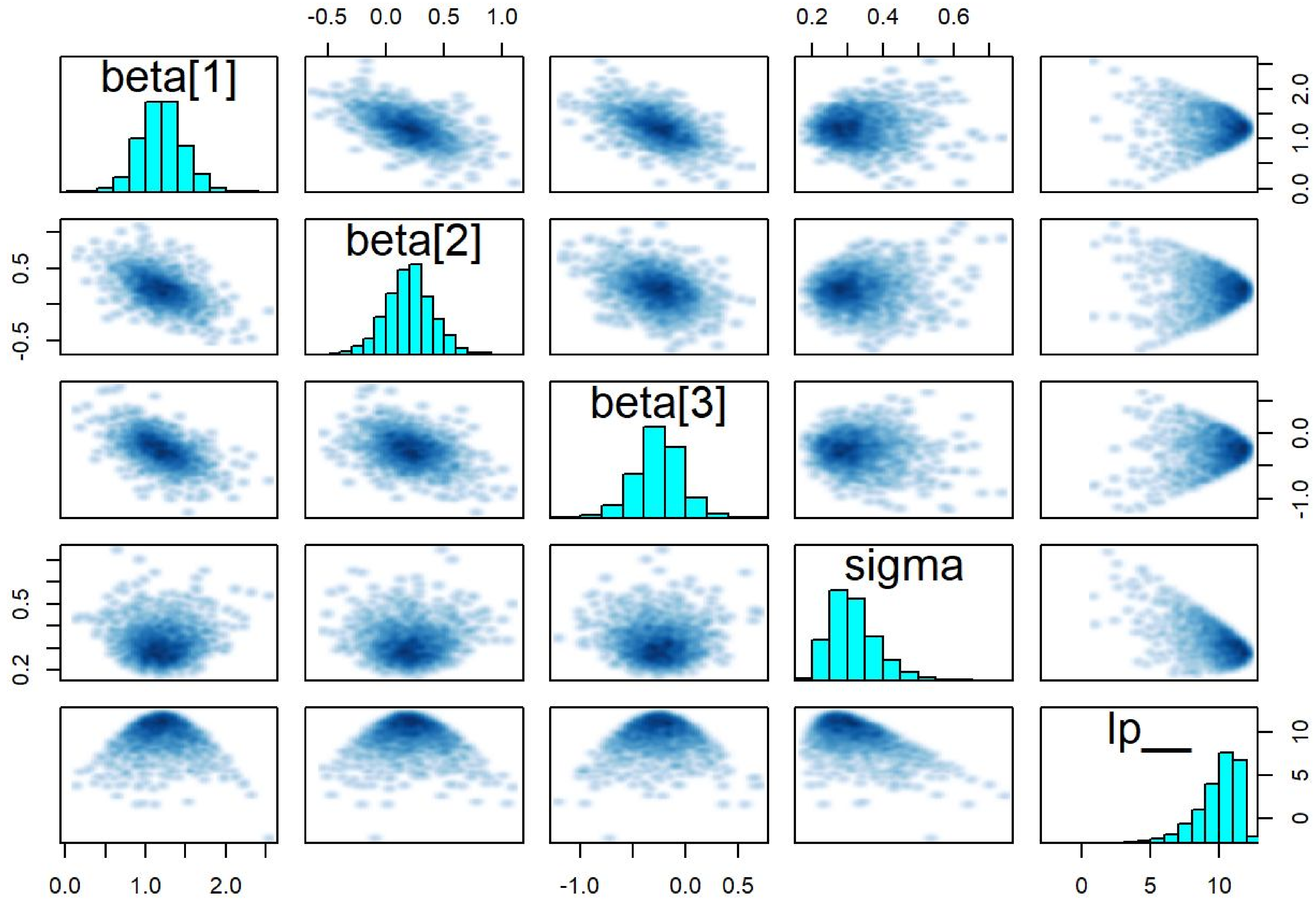
```
hw3_list <- list(N = nrow(Xmat3), P = ncol(Xmat3),
                 X = Xmat3, Y = hw3$Health)

system.time(
  fit3_stan <- sampling(ols.dso, hw3_list)  # same dso
)
```

```
   user  system elapsed
   0.31    0.19    6.05
```

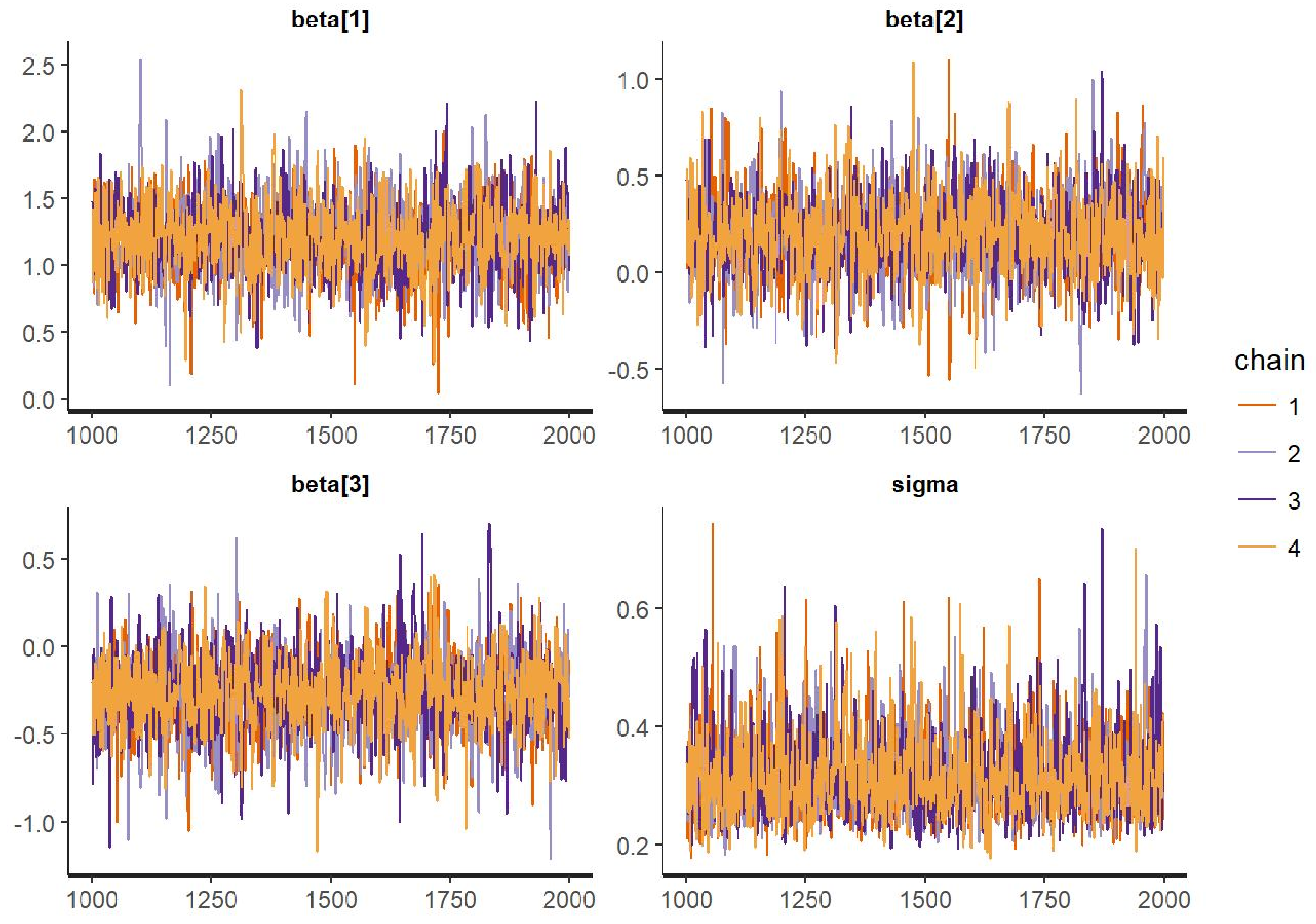
```
print(fit3_stan)
```

```
Inference for Stan model: ols.
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
beta[1]  1.20    0.01 0.28  0.67  1.02  1.20  1.38  1.75  1922    1
beta[2]  0.20    0.00 0.21 -0.22  0.07  0.20  0.33  0.61  2056    1
beta[3] -0.26    0.01 0.22 -0.72 -0.39 -0.26 -0.13  0.17  1855    1
sigma    0.32    0.00 0.07  0.22  0.27  0.31  0.36  0.49  1918    1
lp__     9.92    0.05 1.67  5.55  9.13 10.30 11.13 11.99  1102    1

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:36:39 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
pairs(fit3_stan,pars=c('beta','sigma','lp__'))
```



```
traceplot(fit3_stan)
```



```
get_posterior_mean(fit3_stan)[,5]
```

```
      beta[1]    beta[2]    beta[3]      sigma       lp__ 
    1.2024885  0.2000206 -0.2621058  0.3179320  9.9238155 
```

```
if(interactive) {
  Fit3d(fun(fit3_stan), col = 'magenta')
}
```

## Robust fits with a heavy-tailed error distribution —-

So far quite boring - nothing new, MCMC with normal error and uniform prior give results like OLS - but we can easily change the error distribution

It’s as easy as pi to use a different family of distributions for error.

Exactly the same except for the error distribution and add nu for degrees for freedom for t distribution

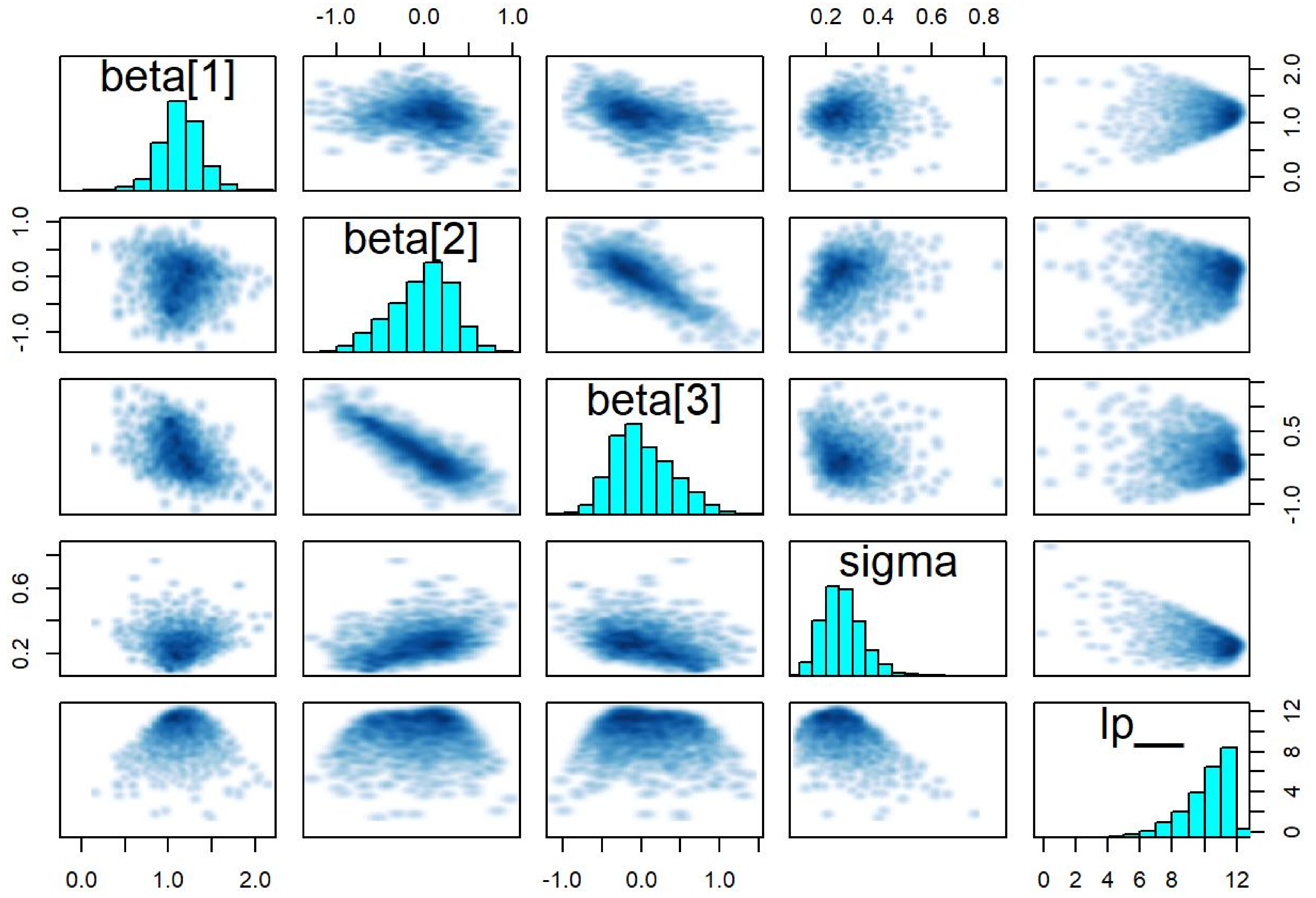
```
cat(c(
"
data {
  int N;   // number of observations
  int P;   // number of columns of X matrix (including intercept)
  matrix[N,P] X;   // X matrix including intercept
  vector[N] Y;  // response
  int nu;   // degrees for freedom for student_t
}
parameters {
  vector[P] beta;   // default uniform prior if nothing specied in model
  real <lower=0> sigma;
}
model {
  Y ~ student_t(nu, X * beta, sigma);
}
"), file = 'robust.stan')

system.time(
  robust_model_dso <- stan_model('robust.stan')
)
```

```
      user  system elapsed 
      0.25    0.00    0.28 
```

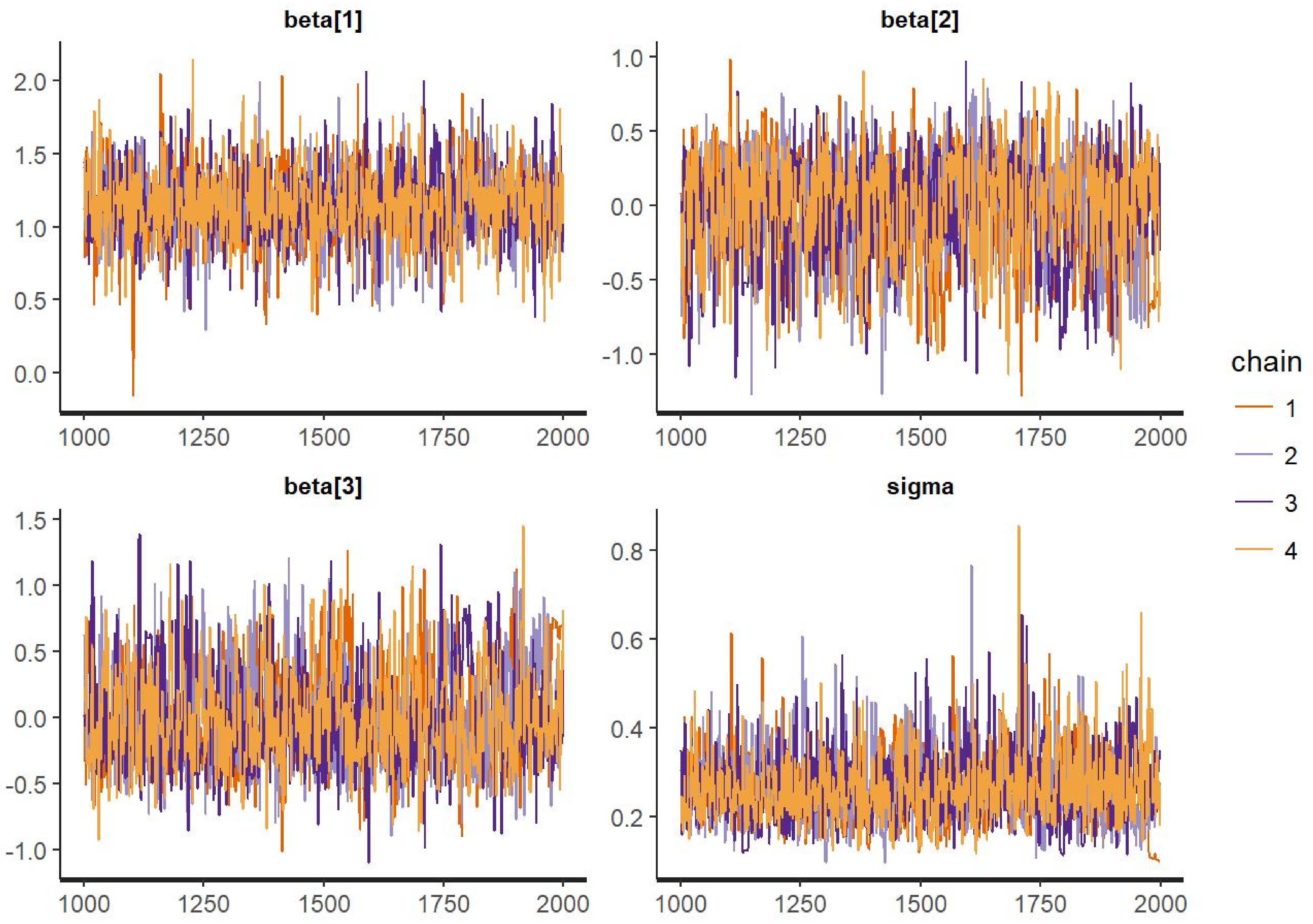
```
fit3_stan_6 <- sampling(robust_model_dso, c(hw3_list, nu = 6))
fit3_stan_6
```

```
   Inference for Stan model: robust.
   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
   beta[1]  1.14    0.01 0.24  0.63  1.00  1.14  1.29  1.61  1785    1
   beta[2] -0.03    0.01 0.35 -0.79 -0.26  0.01  0.22  0.57   690    1
   beta[3]  0.02    0.01 0.38 -0.61 -0.26 -0.03  0.27  0.82   746    1
   sigma    0.26    0.00 0.08  0.13  0.21  0.26  0.31  0.44   787    1
   lp__    10.13    0.05 1.62  6.12  9.35 10.54 11.33 12.05  1195    1

   Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:36:56 2017.
   For each parameter, n_eff is a crude measure of effective sample size,
   and Rhat is the potential scale reduction factor on split chains (at
   convergence, Rhat=1).
```

```
pairs(fit3_stan_6, pars = c('beta','sigma','lp__'))
```



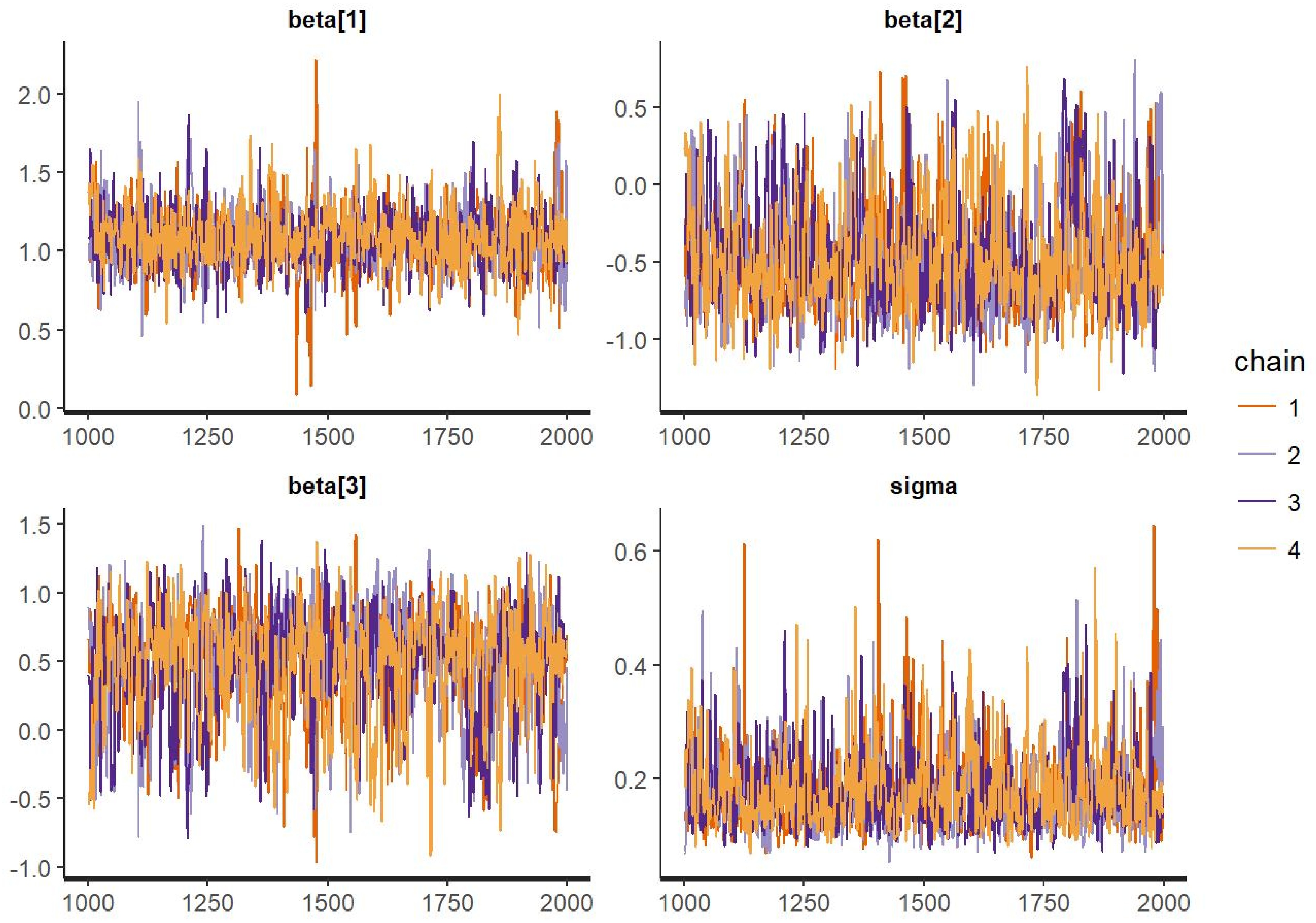
```
traceplot(fit3_stan_6)
```


beta[1]
beta[2]
beta[3]
sigma
chain
1
2
3
4
1000
1250
1500
1750
2000

```
if(interactive) {
  Fit3d(fun(fit3_stan_6), col = 'purple')
}
```

Let’s try more kurtosis

```
fit3_stan_3 <- sampling(robust_model_dso, c(hw3_list, nu = 3))

traceplot(fit3_stan_3)
```



```
fit3_stan_3
```

```
Inference for Stan model: robust.
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
beta[1]  1.08    0.01 0.19  0.75  0.97  1.06  1.17  1.49  1269    1
beta[2] -0.46    0.01 0.34 -1.00 -0.70 -0.52 -0.28  0.34   550    1
beta[3]  0.48    0.02 0.38 -0.40  0.30  0.55  0.73  1.10   548    1
sigma    0.18    0.00 0.07  0.09  0.13  0.16  0.21  0.35   668    1
lp__    12.62    0.08 1.92  7.86 11.60 13.00 14.08 15.14   621    1

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:37:13 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```r
if(interactive) {
  Fit3d(fun(fit3_stan_3), col = 'pink')
}
```

## Using proper priors —-

Using proper priors will often help with models that don’t work with improper uniform priors.

How ‘informative’ priors should be is a question that is both pragmatic and philosophical.

See prior choice recommendations (https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations) and prior distributions (https://github.com/stan-dev/rstanarm/wiki/Prior-distributions)

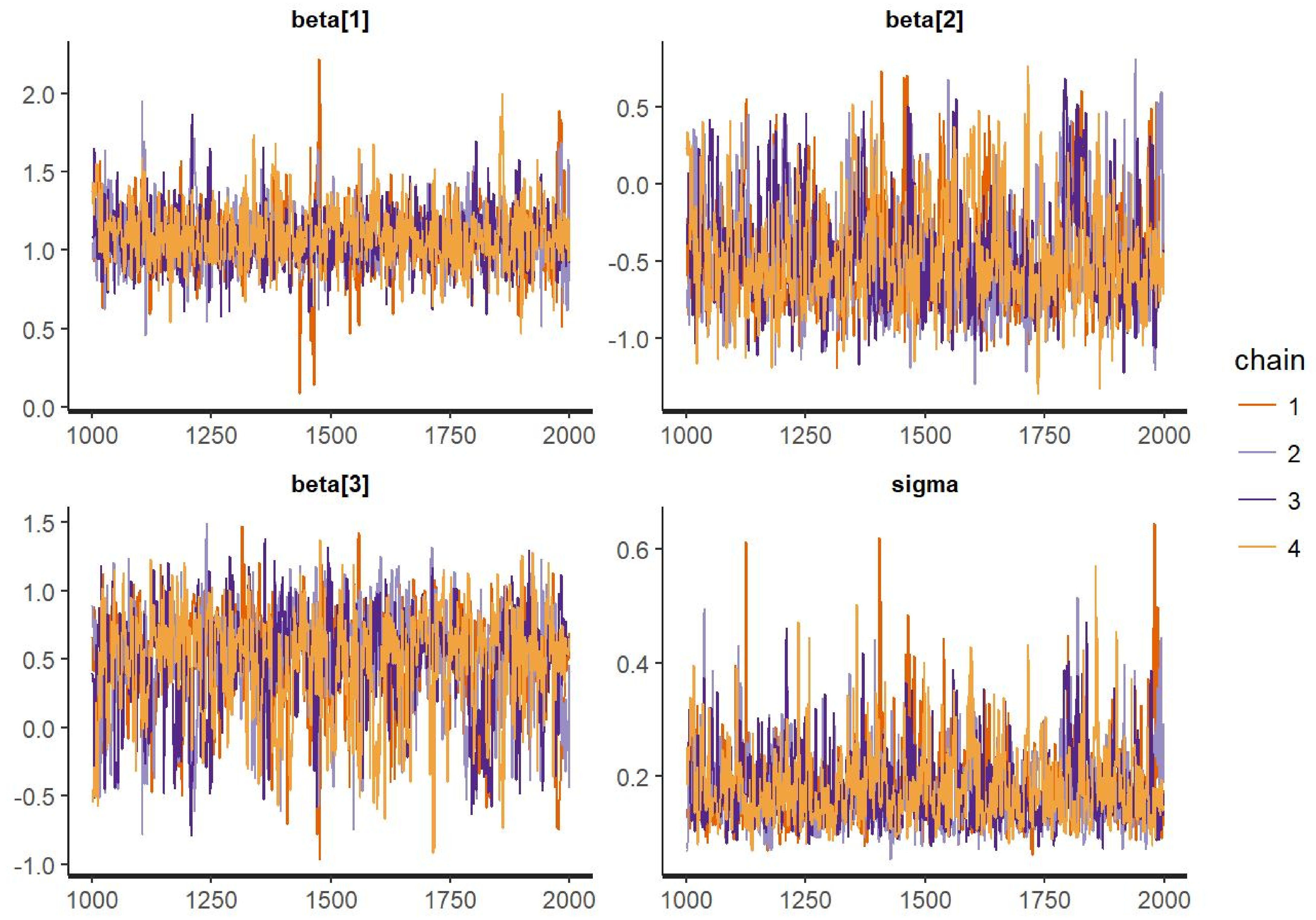
Although it's easy to specify a prior with this generic regression, it usually does not make sense to do so. The prior should be formulated in a way that is reasonable for the structure of the data. See the example using the Prestige data set to illustrate a way of handling a catetorical variable.

```
cat(c(
"
  data {
    int N;   // number of observations
    int P;   // number of columns of X matrix (including intercept)
    matrix[N,P] X;   // X matrix including intercept
    vector[N] Y;  // response
    int nu;   // degrees for freedom for student_t
  }
  parameters {
    vector[P] beta;   // default uniform prior if nothing specied in model
    real <lower=0> sigma;
  }
  model {

  // prior distributions:

    beta ~ student_t(6,0,10); // semi-informative prior
    sigma ~ student_t(6,0,10);      // folded t

  // model:

    Y ~ student_t(nu, X * beta, sigma);
  }

"), file = 'proper_prior.stan')
```

```
system.time(
  proper_prior_dso <- stan_model('proper_prior.stan')
)
```

```
   user  system elapsed 
   0.19    0.00    0.25 
```

```
fit_prior_3 <- sampling(proper_prior_dso, c(hw3_list, nu = 3))

traceplot(fit3_stan_3)
```



```
fit3_stan_3
```

```
Inference for Stan model: robust.
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
beta[1]  1.08    0.01 0.19  0.75  0.97  1.06  1.17  1.49  1269    1
beta[2] -0.46    0.01 0.34 -1.00 -0.70 -0.52 -0.28  0.34   550    1
beta[3]  0.48    0.02 0.38 -0.40  0.30  0.55  0.73  1.10   548    1
sigma    0.18    0.00 0.07  0.09  0.13  0.16  0.21  0.35   668    1
lp__    12.62    0.08 1.92  7.86 11.60 13.00 14.08 15.14   621    1

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:37:13 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
if(interactive) {
  Fit3d(fun(fit3_stan_3), col = 'cyan')
}
```

## Fit Indices for Bayesian models: WAIC and LOO —-

See Fox (2015 pp 669ff) for a review of information criteria such AIC and BIC

Except from Vektari et al. (2016) Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC (https://arxiv.org/abs/1507.04544)

Leave-one-out cross-validation (LOO) and the widely applicable information criterion (WAIC) are methods for estimating pointwise out-of-sample prediction accuracy from a fitted Bayesian model using the log-likelihood evaluated at the posterior simulations of the parameter values. LOO and WAIC have various advantages over simpler estimates of predictive error such as AIC and DIC but are less used in practice because they involve additional computational steps.

Here we lay out fast and stable computations for LOO and WAIC that can be performed using existing simulation draws. We introduce an efficient computation of LOO using Pareto-smoothed importance sampling (PSIS), a new procedure for regularizing importance weights. Although WAIC is asymptotically equal to LOO, we demonstrate that PSIS-LOO is more robust in the finite case with weak priors or influential observations. As a byproduct of our calculations, we also obtain approximate standard errors for estimated predictive errors and for comparing of predictive errors between two models. We implement the computations in an R package called 'loo' and demonstrate using models fit with the Bayesian inference package Stan.

Also see Vektari and Gelman (2014) WAIC and cross-validation in Stan (http://www.stat.columbia.edu/~gelman/research/unpublished/waic_stan.pdf)

The 'generated quantities' block evaluates the log-likelihood at each observed point for each model.

```
cat(c("
data {
  int N;   // number of observations
  int P;   // number of columns of X matrix (including intercept)
  matrix[N,P] X;   // X matrix including intercept
  vector[N] Y;  // response
  int nu;   // degrees for freedom for student_t
}
parameters {
  vector[P] beta;   // default uniform prior if nothing specied in model
  real <lower=0> sigma;
}
model {
  Y ~ student_t(nu, X * beta, sigma);
}
generated quantities {
  // compute the point-wise log likelihood
  // at each point to compute WAIC
  vector[N] log_lik;
  for(n in 1:N) {  // index n for loop need not be declared
    log_lik[n] = student_t_lpdf(Y[n] | nu, X[n,] * beta , sigma);
  }
}
"), file = 'robust_loo.stan')

system.time(
  robust_loo_dso <-
    stan_model('robust_loo.stan', model_name = 'robust with LOO')
)
```

```
      user  system elapsed 
      0.23    0.00    0.33 
```

See description of 'student_t_lpdf' in stan documentation. (lpdf = log probability density function)

Fit models with different error kurtoses

```
fitlist <- list(
  fit3_stan_3 = sampling(robust_loo_dso, c(hw3_list, nu = 3)),
  fit3_stan_6 = sampling(robust_loo_dso, c(hw3_list, nu = 6)),
  fit3_stan_100 = sampling(robust_loo_dso, c(hw3_list, nu = 100))
)

fitlist %>%
  lapply(print, pars = c('beta','sigma')) %>%
  invisible
```

```
Inference for Stan model: robust with LOO.
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
beta[1]  1.07    0.00 0.18  0.73  0.96  1.06  1.16  1.46  1301 1.00
beta[2] -0.49    0.01 0.32 -1.00 -0.69 -0.54 -0.34  0.28   783 1.00
beta[3]  0.52    0.01 0.34 -0.32  0.36  0.57  0.74  1.07   714 1.01
sigma    0.17    0.00 0.07  0.09  0.13  0.16  0.20  0.34   753 1.00

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:37:38 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Inference for Stan model: robust with LOO.
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
beta[1]  1.15    0.01 0.23  0.72  1.00  1.15 1.29  1.60  2074    1
beta[2] -0.05    0.01 0.34 -0.75 -0.29  0.00 0.22  0.53  1058    1
beta[3]  0.03    0.01 0.37 -0.58 -0.25 -0.02 0.28  0.80  1074    1
sigma    0.26    0.00 0.07  0.14  0.21  0.25 0.30  0.43  1415    1

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:37:47 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Inference for Stan model: robust with LOO.
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
beta[1]  1.20    0.01 0.27  0.66  1.03  1.19  1.37  1.73  2056    1
beta[2]  0.19    0.00 0.21 -0.21  0.06  0.19  0.32  0.60  2421    1
beta[3] -0.24    0.00 0.22 -0.68 -0.38 -0.24 -0.11  0.20  2215    1
sigma    0.32    0.00 0.07  0.21  0.27  0.31  0.36  0.50  1565    1

Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:37:59 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
fitlist  %>%
  lapply(extract_log_lik) %>%
  lapply(loo)
```

```
Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-
diagnostic') for details.

Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-
diagnostic') for details.
```

```
$fit3_stan_3
Computed from 4000 by 16 log-likelihood matrix

         Estimate   SE
elpd_loo     -4.8  7.9
p_loo         8.0  4.2
looic         9.6 15.7

All Pareto k estimates are good (k < 0.5)
See help('pareto-k-diagnostic') for details.
$fit3_stan_6
Computed from 4000 by 16 log-likelihood matrix

         Estimate   SE
elpd_loo    -11.9 10.1
p_loo        11.8  8.7
looic        23.8 20.3

Pareto k diagnostic values:
                         Count  Pct
(-Inf, 0.5]   (good)     15    93.8%
 (0.5, 0.7]   (ok)        0     0.0%
   (0.7, 1]   (bad)       0     0.0%
   (1, Inf)   (very bad)  1     6.2%
See help('pareto-k-diagnostic') for details.
$fit3_stan_100
```

```
Computed from 4000 by 16 log-likelihood matrix

         Estimate  SE
elpd_loo     -7.1 3.6
p_loo         4.8 2.6
looic        14.2 7.2

Pareto k diagnostic values:
                         Count  Pct
(-Inf, 0.5]   (good)     13    81.2%
 (0.5, 0.7]   (ok)        2    12.5%
   (0.7, 1]   (bad)       0     0.0%
   (1, Inf)   (very bad)  1     6.2%
See help('pareto-k-diagnostic') for details.
```

```
fitlist  %>%
  lapply(extract_log_lik) %>%
  lapply(loo) ->
  loolist
```

```
Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-
diagnostic') for details.

Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-
diagnostic') for details.
```

Re Pareto k diagnostic (https://rdrr.io/cran/loo/man/pareto-k-diagnostic.html)

Importance sampling is likely to work less well if the marginal posterior p(θ^s | y) and LOO posterior p(θ^s | y_{-i}) are much different, which is more likely to happen with a non-robust model and highly influential observations.

A robust model may reduce the sensitivity to highly influential observations.

## Pairwise comparisons of LOO with SEs —-

See the help file and references on

```
?loo::compare
```

```
   starting httpd help server ...
```

```
    done
```

Pairwise comparisons with SEs:

Note that the only 'significant' comparison is betweeen the two 'student_t' models

```
loolist[1:2] %>% compare(x = .)
```

```
   elpd_diff        se
        -7.1       2.4
```

```
loolist[c(1,3)] %>% compare(x = .)
```

```
   elpd_diff        se
        -2.3       4.8
```

```
loolist[c(2,3)] %>% compare(x = .)
```

```
   elpd_diff        se
         4.8       7.1
```

## Identifying outliers —-

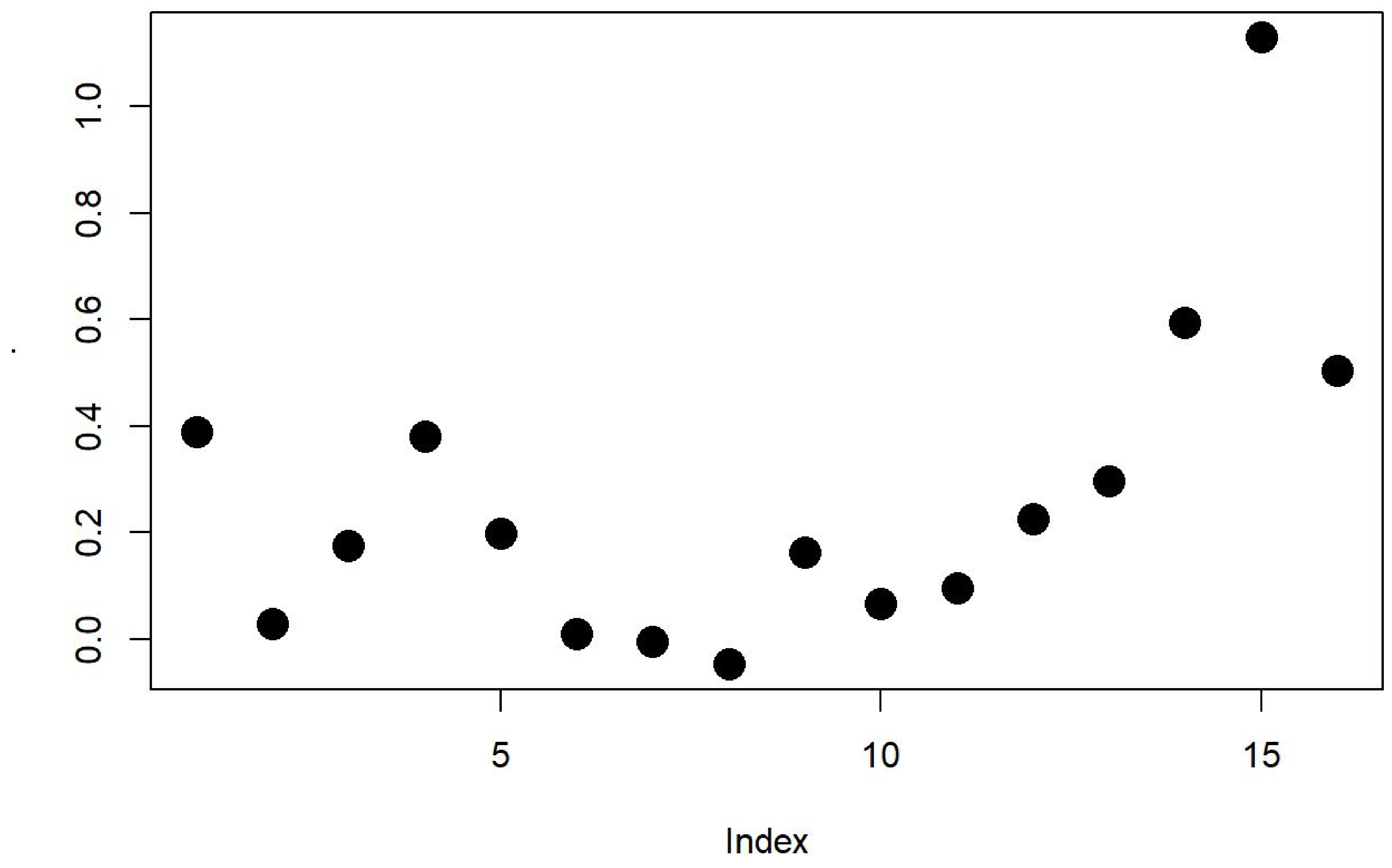
A large Pareto-k parameter indicates an unusual point with large weight in Pareto-Smoothed Importance Sampling (PSIS).

For the quasi-normal model (nu = 100):

```
loo3 <- loo(extract_log_lik(fitlist[[3]]))
```

```
   Warning: Some Pareto k diagnostic values are too high. See help('pareto-k-
   diagnostic') for details.
```

```
loo3$pareto_k %>% plot(pch=16, cex =2)
```



correctly identifies observation 15 as influential. See Vehtari, Gelman, and Gabry (2017) or online preprint (http://www.stat.columbia.edu/~gelman/research/unpublished/loo_stan.pdf)

Also: cho2009bayesian, Zhu, Ibrahim, and Tang (2011) and zhu2012bayesian.

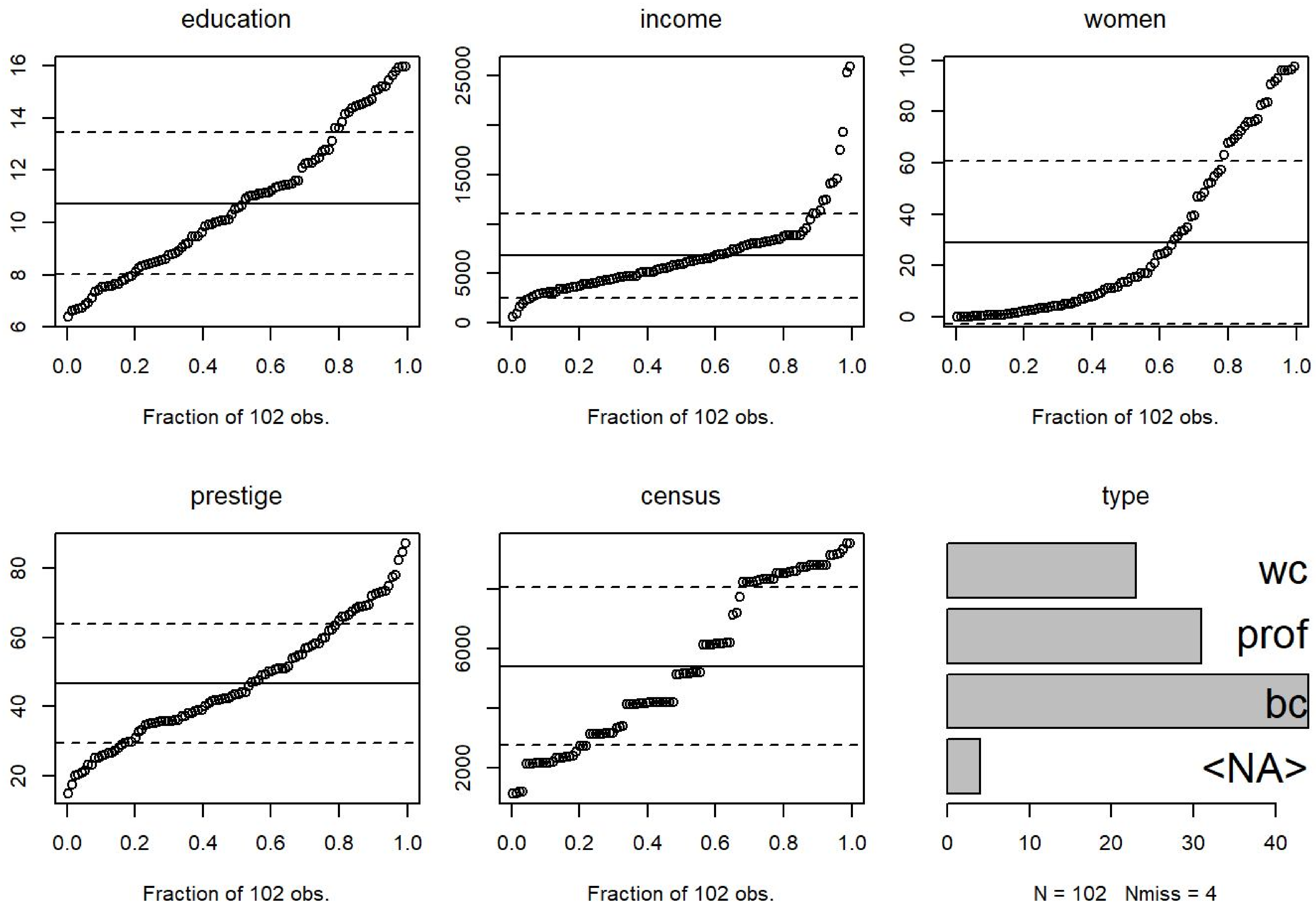
EXERCISE: - See what happens with the Student_t models. Does observation 15 look more 'normal' even though its residual is larger?

## Categorical Predictors —-

```
library(car)
head(Prestige)
```

```
                        education income women prestige census type
   gov.administrators       13.11  12351 11.16     68.8   1113 prof
   general.managers         12.26  25879  4.02     69.1   1130 prof
   accountants              12.77   9271 15.70     63.4   1171 prof
   purchasing.officers      11.42   8865  9.11     56.8   1175 prof
   chemists                 14.62   8403 11.68     73.5   2111 prof
   physicists               15.64  11030  5.13     77.6   2113 prof
```

```
xqplot(Prestige)
```



Using complete cases

```
Prestige %>%
  subset(!is.na(type)) %>%
  droplevels ->    # often good practice if dropping levels of a factor
  dd
tab(dd, ~type)
```

```
   type
      bc  prof    wc Total
      44    31    23    98
```

Note that `type` has 3 levels.

We will regress ‘prestige’ on ‘type’ and ‘women’ (percentage of women).

```
cat(c("

  data{
    int N;
    int Ntype;
    int type[N]; // type will be coded as an integer from 1 to 3
    vector[N] women;
    vector[N] prestige;
  }
  parameters{
    real m_prestige;
    real b_women;
    vector[Ntype] u_type;
    real<lower=0> sigma;
  }
  transformed parameters{
    vector[Ntype] m_type;
    m_type = m_prestige + u_type;
  }
  model{
    // uniform on m_prestige, b_women sigma
    u_type ~ normal(0,100);  // proper prior on deviations
           // -- a proper Bayesian hierarchical model for
           // type would use a hyperparameter instead of 100
           // and the hyperparameter would help determine
           // appropriate amount of pooling between types
    prestige ~ normal(
      m_prestige +
        u_type[type] +     // note how this works using array indexing
                           // -- a key technique for hierarchical modeling
        b_women * women,
      sigma);
  }

"), file = "prestige.stan")

prestige_dso <- stan_model("prestige.stan")
```

Data

```
dat <-
  with(dd,
       list( N = nrow(dd),
             Ntype = length(unique(type)),
             type = as.numeric(as.factor(type)), # to ensure integers from 1 to 3
             women = women,
             prestige = prestige
       )
  )

prestige.stanfit <- sampling(prestige_dso, dat)
prestige.stanfit
```

```
Inference for Stan model: prestige.
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%
m_prestige   48.40    2.26 57.89  -65.34   10.63   49.15   86.07  167.20
b_women      -0.08    0.00  0.03   -0.14   -0.10   -0.08   -0.06   -0.01
u_type[1]   -11.38    2.26 57.88 -129.55  -49.23  -11.71   26.41  103.27
u_type[2]    21.42    2.26 57.90  -97.90  -16.05   20.80   59.26  135.62
u_type[3]    -2.08    2.26 57.89 -121.53  -39.47   -2.38   35.52  112.06
sigma         9.42    0.02  0.72    8.17    8.91    9.39    9.89   10.94
m_type[1]    37.02    0.03  1.55   33.94   35.97   37.06   38.08   40.00
m_type[2]    69.83    0.03  1.94   66.00   68.54   69.79   71.13   73.66
m_type[3]    46.32    0.05  2.71   41.21   44.51   46.33   48.07   51.80
lp__       -266.36    0.05  1.85 -271.05 -267.30 -266.02 -265.00 -263.88
           n_eff Rhat
m_prestige   654 1.01
b_women     2049 1.00
u_type[1]    654 1.01
u_type[2]    657 1.01
u_type[3]    655 1.01
sigma       1836 1.00
m_type[1]   3500 1.00
m_type[2]   3534 1.00
m_type[3]   2565 1.00
lp__        1185 1.00
```

```
Samples were drawn using NUTS(diag_e) at Sun Jul 16 14:38:54 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
wald(prestige.stanfit, diag(3), pars = 'm_type' )
```

```
  numDF denDF  F.value p.value
1     3   Inf 535.4555 <.00001

    Estimate Std.Error  DF  t-value p-value Lower 0.95 Upper 0.95
  1 37.01976  1.552590 Inf 23.84387 <.00001   33.97673   40.06278
  2 69.82807  1.942824 Inf 35.94153 <.00001   66.02021   73.63594
  3 46.32079  2.711620 Inf 17.08233 <.00001   41.00611   51.63546
```

```
Ldiff <- rbind(
  '2-1' = c(-1,1,0),
  '3-1' = c(-1,0,1),
  '3-2' = c(0,-1,1)
)
wald(prestige.stanfit, Ldiff, pars = 'm_type' )
```

```
    numDF denDF  F.value p.value
  1     2   Inf 108.1827 <.00001

          Estimate Std.Error  DF   t-value p-value Lower 0.95 Upper 0.95
    2-1  32.808315  2.259771 Inf 14.518425 <.00001  28.379245   37.23738
    3-1   9.301029  2.697021 Inf  3.448630 0.00056   4.014965   14.58709
    3-2 -23.507285  2.767078 Inf -8.495346 <.00001 -28.930659  -18.08391
```

```
summary(fitlm <- lm(prestige ~ type + women, dd))
```

```

Call:
lm(formula = prestige ~ type + women, data = dd)

Residuals:
     Min       1Q   Median       3Q      Max 
-17.1119  -7.1401  -0.3717   6.1882  27.0299 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 36.97658    1.53718  24.055  < 2e-16 ***
typeprof    32.82082    2.19002  14.987  < 2e-16 ***
typewc       9.30277    2.64428   3.518 0.000672 ***
women       -0.07640    0.03335  -2.291 0.024194 *  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.293 on 94 degrees of freedom
Multiple R-squared:  0.7136, Adjusted R-squared:  0.7045 
F-statistic: 78.08 on 3 and 94 DF,  p-value: < 2.2e-16
```

```
wald(fitlm, Ldiff(fitlm, 'type'))
```

```
    numDF denDF  F.value p.value
  1     2    94 115.1268 <.00001

                 Estimate Std.Error DF   t-value p-value Lower 0.95
    prof - <ref>  32.820824  2.190019 94 14.986547 <.00001  28.472490
    wc - <ref>     9.302767  2.644275 94  3.518078 0.00067   4.052496
    wc - prof    -23.518057  2.714843 94 -8.662770 <.00001 -28.908441

                Upper 0.95
    prof - <ref>   37.16916
    wc - <ref>     14.55304
    wc - prof     -18.12767
```

EXERCISE:

Try to specify a hyperparameter for the variability between 'u_type'. Experiment with priors for the hyperparameter.

## Notes —-

- Some links:
  - Stan documentation (http://mc-stan.org/users/documentation/index.html)
  - Manual: download as a pdf (https://github.com/stan-dev/stan/releases/download/v2.16.0/stan-reference-2.16.0.pdf)
  - Stan tutorials (http://mc-stan.org/users/documentation/tutorials)
  - Stan Development Team (2017c) Stan Forums (http://discourse.mc-stan.org/)
    - Old list – now 'deprecated' (https://groups.google.com/forum/#!forum/stan-users)

    - Betancourt (2017) A conceptual introduction to HMC (https://arxiv.org/pdf/1701.02434v1.pdf)
- Note from the documentation (Stan Development Team 2017d)
    - An example of the distributions available in the model block for **both priors and models**. The only difference is that the 'random variable' is data for the model and a parameter for a prior.
        - `y ~ normal(mu, sigma);` – Note `sigma` is the standard deviation
        - `y ~ student(nu, mu, sigma)` – Note: `sigma` is a scale parameter, not the SD
        - `y ~ cauchy(mu, sigma)` – Note: same as cauchy with `nu = 0`
        - `y ~ exponential(beta)` – Note: beta is the waiting time. Expected value is 1/beta
        - `y ~ poisson(lambda)`
        - `y ~ bernoulli(prob)` – y is 0 or 1
        - `y ~ bernoulli_logit(theta)` – where `theta` is the logit
    - To use distributions in Stan, you need to understand the type declarations for the `_lpmf` (for a discrete distribution) or for the `_lpf` version of the corresponding function. For example, for the `bernouilli_logit` sampling function, search for `bernouilli_logit` in the manual (**???**) and look at declarations for `bernoulli_logit_lpmf` which are:
        - `real bernoulli_logit_lpmf(ints y | reals alpha)`
        - This means that the random variable `y` must be an integer, hence a scalar or an array, and that the logit parameter must be a 'real', i.e. a real scalar, a vector or an array.
    - See section 24.3 for a description of sampling:
        - Simulating a skateboard with leapfrog steps:
            - simple models: ok to use a few big steps
            - complex models: use many small steps
            - or default NUTS: keep going until U-turn, adjust step size adaptively
            - with 'perfect simulation' always accept proposal because probability: sum of potential and kinetic energy always constant, but not with simulation. Trade-off between step size and probability of acceptance.

- See section 25 for a very good description of data types.Notably:
    - ints are promoted to reals where necessary. But avoid.
    - bounds may be expressions but must use previously declared variables (data + parameters). If bounds are parameter, log_Jacobian is adjusted appropriately.
    - vectors, row vectors and matrices are reals, arrays anything.
    - index starts at 1, in contrast with C++
    - vectors are column vectors. Dimension shown in declaration: `vector[3] u;` . Bounds apply to all terms, e.g. `vector<lower=0>[3] u;`
        - size expressions may use data, transformed data or local variables, not parameters, transformed parameters or generated quantities.
    - Fancy constraints:
        - unit simplex: `simplex[5] theta;` non-neg, sum to 1, looks like a vector
        - ordered vector: `ordered[5] c;`
        - row vector: `row_vector[5] u;`
        - matrices: `matrix[M,N] A;` ( `M` , `N` must be integers)
            - assign to row: `A[1] = b` assigns vector to row
            - `corr_matrix[6] B;` constrains matrix
            - `cholesky_factor_corr[6] L;`
            - `cov_matrix[6] S;`
            - `cholesky_factor_cov[6] L;`
    - Assignments and constraint checks:
        - runtime size checks at end of blocks, not compiler
        - data: when read or after transformed data block
        - parameters: enforced by transform
        - transformed parameters: at end of block
        - Single index to matrix refers to row of type `row_vector`

        - Arrays of anything, arbitrary dimension, follow name:
            - e.g. `matrix[3,4] A[2,3];` a 2x3 array of 3x4 matrices
            - `A[1]` is a a sub-array: a 3-array of 3x4 matrices.
            - Arrays and subarrays can be manipulated and assigned
            - Array of matrices: array indices go first.
        - Assigment: arrays, vectors, row_vectors, matrices are not inter-assignable unless at the scalar level with a loop or with sub-arrays/vectors of corresponding size and type.
        - Reserved names: `for,in,while,repeat,until,if,then,else,true,false` + a bunch more – see the manual
        - operators:
            - use apostrophe for vector or matrix transpose, e.g. `A'B`
            - use `*`` for matrix multiplication
            - use `.*`` for element-wise multiplication
            - vectorization of functions and operators: see manual, this is expanding
        - Indexing:
            - indices can be integers or integer arrays or symbols, e.g. `:` denoting all, `5:` denoting 5 up or `:6` denoting 1 to 6.
            - If `x = [11,12,13,14]` is a row vector and `ii = (1,2,1,2,3)` is an integer array (note that this is not correct Stan notation to assign vectors), then `x[ii] = [11,12,11,12,13]` is a row vector.
        - Type signatures for Object-Oriented dispatching:
            - `real mean(real[])` and `real mean(vector)` are different methods of the `mean` generic function
        - Constants: `pi(), e(), 1.0` are real, `1` is int.
- References:
    - McElreath (2015) is considered a good introduction to Stan, highly reccommended by the developers of Stan, although it, unfortunately, avoids coding in Stan by providing a front end in an R package.
    - Carpenter et al. (2016) is a recent article with an up-to-date introduction to Stan.

- gelman2015stan
- Stan Development Team (2017a) is the on-line documentation that is a paragon of intelligible information.
- Stan Development Team (2017d) needs to be downloaded as a pdf file. Go to Stan Development Team (2017a) to download the latest version.
- Stan Development Team (2017b) gives some recommendations for priors.
- HMC and Banana Distributions (http://nross626.math.yorku.ca/SCS_Longitudinal/files/Orbits.R.html)
- Stan Development Team (2016) Brief Guide to Stan's Warnings (http://mc-stan.org/misc/warnings.html)

## References

Carpenter, Bob, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2016. "Stan: A Probabilistic Programming Language." *Journal of Statistical Software* 20: 1–37.

Fox, John. 2015. *Applied Regression Analysis and Generalized Linear Models, 3rd Ed.* Sage Publications.

McElreath, Richard. 2015. "Statistical Rethinking." CRC Press.

Stan Development Team. 2016. "Brief Guide to Stan's Warnings." http://mc-stan.org/misc/warnings.html (http://mc-stan.org/misc/warnings.html).

———. 2017a. "Documentation." http://mc-stan.org/users/documentation/ (http://mc-stan.org/users/documentation/).

———. 2017b. "Prior Choice Recommendations." https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations (https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations).

———. 2017c. "Stan Forums." http://discourse.mc-stan.org/ (http://discourse.mc-stan.org/).

———. 2017d. “Stan Modeling Language: User’s Guide and Reference Manual.” https://github.com/stan-dev/stan/releases/download/v2.16.0/stan-reference-2.16.0.pdf (https://github.com/stan-dev/stan/releases/download/v2.16.0/stan-reference-2.16.0.pdf).

Vehtari, Aki, Andrew Gelman, and Jonah Gabry. 2017. “Practical Bayesian Model Evaluation Using Leave-One-Out Cross-Validation and Waic.” *Statistics and Computing* 27 (5). Springer: 1413–32.

Zhu, Hongtu, Joseph G Ibrahim, and Niansheng Tang. 2011. “Bayesian Influence Analysis: A Geometric Approach.” *Biometrika* 98 (2). Oxford University Press: 307–23. doi:doi: 10.1093/biomet/asr009 (https://doi.org/doi: 10.1093/biomet/asr009).