

Class 10: Moving from `rstanarm` to `rstan`

Andrew Parnell
andrew.parnell@mu.ie



Learning outcomes:

- ▶ Start using `rstan` instead of `rstanarm`
- ▶ Be able to fit more flexible models
- ▶ Interpret output from `rstan` models
- ▶ Do some model comparison using LOO and WAIC

1 / 35

2 / 35

Main differences

- ▶ `rstanarm` fits most models in one line very quickly, but it only fits a few of the main types of models (mainly regression models)
- ▶ `rstan` can fit a much wider variety of models
- ▶ `rstan` gives you much more control over prior distributions
- ▶ `rstan` takes a long time to compile each model before it starts running

Modelling set-up in `rstan`

1. Write some stan code and save it in a `rstan` file or in a text string
2. Save your data in a list with all the named components matching the data part of the stan code
3. Use the `stan` function to fit the model
4. Use `plot`, `summary`, etc to look at the output

3 / 35

4 / 35

Linear regression in rstan

```
stan_code = '  
data {  
  int N;  
  vector[N] x;  
  vector[N] y;  
}  
parameters {  
  real intercept;  
  real slope;  
  real<lower=0> residual_sd;  
}  
model {  
  // Likelihood  
  y ~ normal(intercept + slope * x, residual_sd);  
  // Priors  
  intercept ~ normal(0, 100);  
  slope ~ normal(0, 100);  
  residual_sd ~ uniform(0, 100);  
}  
'
```

5 / 35

Key features of rstan code

- ▶ Three blocks for most models
 - ▶ data must declare all the objects that are fixed throughout the code
 - ▶ parameters can only include objects which are given prior distributions
 - ▶ model contains the priors and the likelihoods
- ▶ Other blocks we will use later

6 / 35

Fitting the models

```
earnings = read.csv('../data/earnings.csv')  
library(rstan)  
#options(mc.cores = parallel::detectCores())  
stan_run = stan(data = list(N = nrow(earnings),  
                             y = earnings$y,  
                             x = earnings$x_centered),  
                 model_code = stan_code)
```

7 / 35

Looking at output

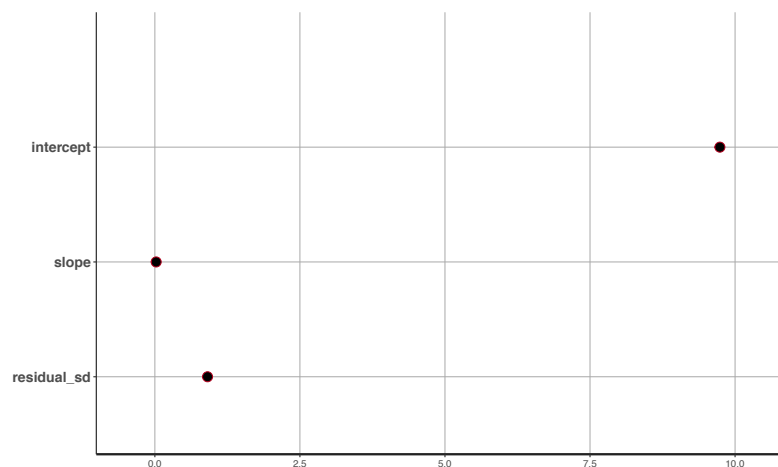
```
print(stan_run)
```

```
## Inference for Stan model: ec2a1d3ccd2ba93b4b1df3b0cecf11b9.  
## 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%  
## intercept    9.74    0.00 0.03   9.68   9.72   9.74   9.76   9.79  
## slope        0.02    0.00 0.00   0.02   0.02   0.02   0.02   0.03  
## residual_sd  0.91    0.00 0.02   0.87   0.89   0.91   0.92   0.95  
## lp__         -426.20  0.03 1.18 -429.21 -426.75 -425.90 -425.33 -424.84  
##               n_eff Rhat  
## intercept    2720     1  
## slope        4000     1  
## residual_sd  2939     1  
## lp__         2205     1  
##  
## Samples were drawn using NUTS(diag_e) at Mon Oct  8 15:03:04 2018.  
## 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).
```

8 / 35

Looking at output 2

```
plot(stan_run)
```



9 / 35

An alternative way of setting the model up

```
stan_code = '  
data {  
  int N;  
  vector[N] x;  
  vector[N] y;  
}  
parameters {  
  real intercept;  
  real slope;  
  real<lower=0> residual_sd;  
}  
transformed parameters {  
  vector[N] fits;  
  for (i in 1:N) {  
    fits[i] = intercept + slope * x[i];  
  }  
}  
model {  
  // Likelihood  
  y ~ normal(fits, residual_sd);  
  // Priors  
  intercept ~ normal(0, 100);  
  slope ~ normal(0, 100);  
  residual_sd ~ uniform(0, 100);  
}  
'
```

10 / 35

Flexibility in prior distributions

- ▶ Because we are writing out the model directly we can change the priors exactly how we want them
- ▶ For example, if we wanted to force the slope to be positive we could put a `gamma` prior on the slope, or change the declaration to `real<lower=0> slope;`
- ▶ A popular prior for standard deviation parameters is the half-cauchy. You will see this lots in the `rstanarm` and `rstan` examples

11 / 35

Quirks of the stan language

- ▶ Each line must finish with a semi-colon
- ▶ The declarations are a minefield. There seems to be at least 2 ways to specify vectors, and multiple ways to specify matrices. Hopefully they will tidy up in a future version
- ▶ However you can declare other variables on the fly in e.g. the `model` or `transformed parameters` sections
- ▶ Within block it doesn't seem to matter hugely the order the code is in, but the declarations need to be at the top
- ▶ Unlike most or R, everything is strongly typed. You cannot miss anything out of the `parameters` or `data` parts
- ▶ If you can vectorise the likelihood stan will run much faster

12 / 35

Mixed effects models in rstan

```
stan_code_mm = '
data {
  int N;
  int N_eth;
  vector[N] x;
  vector[N] y;
  int eth[N];
}
parameters {
  real intercept[N_eth];
  real slope;
  real mean_intercept;
  real<lower=0> residual_sd;
  real<lower=0> sigma_slope;
}
model {
  // Likelihood
  for (i in 1:N) {
    y[i] ~ normal(intercept[eth[i]] + slope * x[i], residual_sd);
  }
  // Priors
  slope ~ normal(0, 0.1);
  for (j in 1:N_eth) {
    intercept[j] ~ normal(mean_intercept, sigma_slope);
  }
  mean_intercept ~ normal(11, 2);
  sigma_slope ~ cauchy(0, 10);
  residual_sd ~ cauchy(0, 10);
}
'
```

13 / 35

Running the hierarchical model

```
stan_run_2 = stan(data = list(N = nrow(earnings),
                              y = earnings$y,
                              x = earnings$x_centered,
                              eth = earnings$eth,
                              N_eth = length(unique(earnings$eth))),
                  model_code = stan_code_mm)
```

14 / 35

Output 1

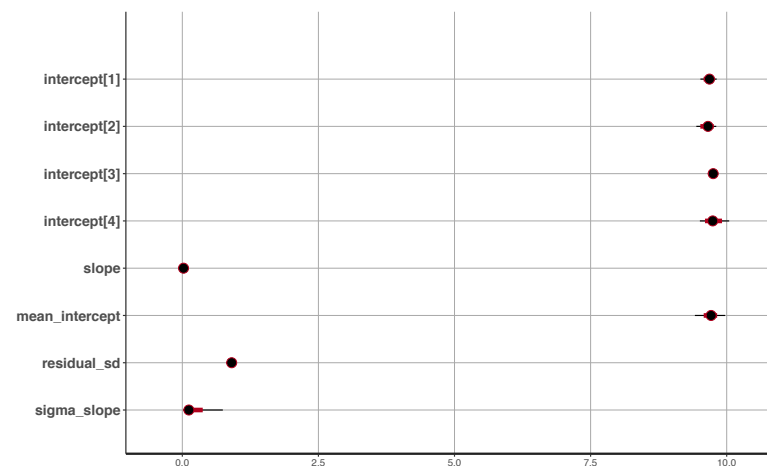
```
print(stan_run_2)
```

```
## Inference for Stan model: 90c73485e3cd793d3c411fc380ed2c1f.
## 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%
## intercept[1]    9.68    0.00 0.08   9.52   9.63   9.68   9.73
## intercept[2]    9.65    0.00 0.10   9.44   9.59   9.66   9.71
## intercept[3]    9.75    0.00 0.03   9.69   9.73   9.75   9.77
## intercept[4]    9.75    0.00 0.13   9.51   9.67   9.74   9.81
## slope           0.02    0.00 0.00   0.02   0.02   0.02   0.02
## mean_intercept  9.71    0.00 0.13   9.42   9.65   9.71   9.77
## residual_sd     0.91    0.00 0.02   0.87   0.89   0.91   0.92
## sigma_slope     0.18    0.01 0.23   0.02   0.07   0.12   0.22
## lp__           -421.10   0.18 2.94 -427.43 -422.93 -420.94 -419.09
##
##          97.5% n_eff Rhat
## intercept[1]    9.82  1100 1.01
## intercept[2]    9.81   650 1.01
## intercept[3]    9.81  1539 1.00
## intercept[4]   10.05 2281 1.00
## slope           0.03   732 1.00
## mean_intercept  9.97 1552 1.00
## residual_sd     0.94  1941 1.00
## sigma_slope     0.74   930 1.00
## lp__           -415.69   275 1.01
##
## Samples were drawn using NUTS(diag_e) at Mon Oct  8 15:03:33 2018.
## 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).
```

15 / 35

Output 2

```
plot(stan_run_2)
```



16 / 35

Getting directly at the posterior distribution

```
post = as.data.frame(stan_run_2)
head(post)
```

```
##   intercept[1] intercept[2] intercept[3] intercept[4]
## 1    9.587808    9.618257    9.752213    9.704414 0.
## 2    9.603065    9.550718    9.785447    9.840273 0.
## 3    9.652886    9.561738    9.752084    9.721791 0.
## 4    9.717656    9.761773    9.762135    9.784466 0.
## 5    9.698116    9.693098    9.763895    9.747191 0.
## 6    9.698116    9.693098    9.763895    9.747191 0.
##   mean_intercept residual_sd sigma_slope    lp__
## 1    9.680109    0.8852871  0.11053914 -423.2668
## 2    9.730371    0.8659221  0.19622963 -422.2908
## 3    9.660946    0.8897362  0.15178171 -419.9416
## 4    9.700114    0.9228407  0.08869739 -418.9574
## 5    9.767824    0.9171940  0.04259593 -418.8286
## 6    9.767824    0.9171940  0.04259593 -418.8286
```

17 / 35

Creating predictions by hand

```
stan_code_3 = '
data {
  int N;
  int N_pred;
  vector[N] x;
  vector[N] y;
  vector[N_pred] x_pred;
}
parameters {
  real intercept;
  real slope;
  real<lower=0> residual_sd;
}
model {
  // Likelihood
  y ~ normal(intercept + slope * x, residual_sd);
  // Priors
  intercept ~ normal(0, 100);
  slope ~ normal(0, 100);
  residual_sd ~ uniform(0, 100);
}
generated quantities {
  vector[N_pred] y_pred;
  for (j in 1:N_pred)
    y_pred[j] = intercept + slope * x[j];
}
'
```

18 / 35

Fitting the new model

```
stan_run_3 = stan(data = list(N = nrow(earnings),
                              N_pred = 5,
                              y = earnings$y,
                              x = earnings$x_centered,
                              x_pred = seq(-3,3, length = 5),
                              model_code = stan_code_3)
```

19 / 35

Extract out the predictions

```
preds = extract(stan_run_3, 'y_pred')
head(preds$y_pred)
```

```
##
## iterations      [,1]      [,2]      [,3]      [,4]      [,5]
##      [1,] 10.18138  9.693234  9.571198  9.510179  9.571198
##      [2,] 10.13474  9.680057  9.566386  9.509551  9.566386
##      [3,] 10.06657  9.659494  9.557726  9.506842  9.557726
##      [4,] 10.18562  9.623705  9.483225  9.412985  9.483225
##      [5,] 10.08883  9.675997  9.572789  9.521185  9.572789
##      [6,] 10.16509  9.658379  9.531702  9.468364  9.531702
```

20 / 35

Creating posterior predictive values by hand

```
stan_code_4 = '
data {
  int N;
  vector[N] x;
  vector[N] y;
  vector[N_pred] y_pred;
}
parameters {
  real intercept;
  real slope;
  real<lower=0> residual_sd;
}
model {
  // Likelihood
  y ~ normal(intercept + slope * x, residual_sd);
  // Priors
  intercept ~ normal(0, 100);
  slope ~ normal(0, 100);
  residual_sd ~ uniform(0, 100);
}
generated quantities {
  vector[N] y_pred;
  for (j in 1:N)
    y_pred[j] = normal_rng(intercept + slope * x[j], residual_sd);
}
'
```

21 / 35

A stan glmm

```
stan_code_od_pois = '
data {
  int<lower=0> N;
  int<lower=0> N_trt;
  int<lower=0> y[N];
  int trt[N];
}
parameters {
  real beta_trt[N_trt];
  real trt_mean;
  real<lower=0> trt_sd;
}
model {
  for (i in 1:N)
    y[i] ~ poisson_log(beta_trt[trt[i]]);

  // Priors on coefficients
  for(j in 1:N_trt)
    beta_trt[j] ~ normal(trt_mean, trt_sd);

  trt_mean ~ normal(0, 10);
  trt_sd ~ cauchy(0, 5);
}
'
```

22 / 35

Model comparison in rstan (and rstanarm)

- ▶ These two have their own model comparison criteria called WAIC (Widely Applicable Information Criterion) and LOO (Leave one out)
- ▶ We will use both. Philosophically WAIC is the more satisfactory but practically LOO seems to work better
- ▶ WAIC falls under the framework of *Information Criterion*, LOO is slightly different

23 / 35

Model comparison: an introduction

- ▶ We can come up with the fanciest model in the world but if it does not meet our desired goals (either prediction or causation) then we cannot publish or use it
- ▶ You can broadly split model comparison into two parts: *absolute* model comparison and *relative* model comparison
- ▶ In absolute model comparison, we are looking at how well a specific model fits the data at hand
- ▶ In relative model comparison, we can only look at how well a set of models performs on the same data with the goal of choosing the best one (or group)

24 / 35

Relative model comparison: model information criteria

- ▶ You might have come across these before: Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC)
- ▶ The general idea is that the score on the likelihood is a good measure of model fit, except for the fact that more complex models will generally have higher likelihood scores
- ▶ If we penalise these scores by some measure of the complexity of the model then we can compare models across complexities
- ▶ The usual measure of complexity is some function of the number of parameters
- ▶ Because these are relative model comparisons, the best model according to an IC might still be useless!

25 / 35

Different types of information criteria

- ▶ For various historical reasons, people tend to transform the likelihood score into the *deviance*, which is minus twice the log-likelihood score
- ▶ They then add a model complexity term onto it
- ▶ The two most common ICs are:

$$\text{AIC} : -2 \log L + 2p$$

$$\text{BIC} : -2 \log L + p \log n$$

where p is the number of parameters and n is the number of observations

- ▶ We usually pick the smallest values of these across different models

26 / 35

Information criteria for Hierarchical models

- ▶ For Bayesian models it's hard to know which value of L to use, seeing as at each iteration we get a different likelihood score.
- ▶ Two specific versions of IC have been developed for these situations
- ▶ The first, called the *Deviance Information Criteria* (DIC) is calculated via:

$$\text{DIC} : -2 \log L_{\max} + 2p_D$$

where p_D is the *effective number of parameters*

- ▶ The second called the Widely Applicable Information Criterion (WAIC) which is calculated as:

$$\text{WAIC} : -2 \log L_{\max} + p_{\text{WAIC}}$$

- ▶ Here p_{WAIC} is a measure of the variability of the likelihood scores

27 / 35

Which information criterion should I use?

- ▶ WAIC and DIC are built for Bayesian hierarchical models
- ▶ DIC was traditionally used everywhere but has fallen out of favour
- ▶ WAIC is included in the `loo` package which is installed alongside Stan
- ▶ WAIC is considered superior as it also provides uncertainties on the values. Most of the others just give a single value
- ▶ More generally there is a philosophical argument about whether we ever want to choose a single best model

28 / 35

An alternative: cross validation

- ▶ Cross validation (CV) works by:
 1. Removing part of the data,
 2. Fitting the model to the remaining part,
 3. Predicting the values of the removed part,
 4. Comparing the predictions with the true (left-out) values
- ▶ It's often fitted repeatedly, as in k-fold CV where the data are divided up into k groups, and each group is left out in turn
- ▶ In smaller data sets, people perform leave-one-out cross-validation (LOO-CV)

29 / 35

Pros and cons of CV

- ▶ We might also run the 5-fold CV on the previous slide for different complexity models and see which had the smallest root mean square error of prediction (RMSEP), i.e. use it as a relative criteria
- ▶ CV is great because it actually directly measures the performance of the model on real data, based on data the model hasn't seen
- ▶ However, it's computationally expensive, and problems occur in hierarchical models if some groups are small, and therefore might get left out of a fold
- ▶ The `loo` function (in the `loo` package) gives an approximation of LOO-CV

30 / 35

Absolute model comparison

- ▶ We've already met posterior predictive distributions, which is essentially leave none out CV.
- ▶ Another popular one is something called the *Bayes Factor*. This is created by first calculating the posterior distribution of a model given the data, a measure of absolute model fit. The ratios of these can be compared for different models to create a relative model criteria.
- ▶ However, Bayes Factors are really hard to calculate and often overly sensitive to irrelevant prior choices

31 / 35

Continuous model expansion

- ▶ There are lots of clever ways to set up prior distributions so that a model choice step is part of the model fit itself
- ▶ One way is partial pooling, by which we force e.g. varying slope and intercept parameters to the same value (or not)
- ▶ Another way is to put shrinkage or selection priors on the parameters in the model, possibly setting them to zero
- ▶ More on all of these later in the course

32 / 35

Example: computing loo and waic on an rstanarm regression model

```
library(loo)
prostate = read.csv('../data/prostate.csv')
mod_1 = stan_lmer(lpsa ~ lcavol + (1 | gleason), data = prostate)
mod_2 = stan_lmer(lpsa ~ (lcavol | gleason), data = prostate)

loo_1 = loo(mod_1)
loo_2 = loo(mod_2)
compare_models(loo_1, loo_2)
```

```
##
## Model comparison:
## (negative 'elpd_diff' favors 1st model, positive favors 2nd)
##
## elpd_diff      se
##      -0.3      1.8
```

33 / 35

Now with WAIC

```
library(loo)
prostate = read.csv('../data/prostate.csv')
waic_1 = waic(mod_1)
waic_2 = waic(mod_2)
compare_models(waic_1, waic_2)
```

```
##
## Model comparison:
## (negative 'elpd_diff' favors 1st model, positive favors 2nd)
##
## elpd_diff      se
##      -0.3      1.8
```

34 / 35

Summary

- ▶ We have now seen a number of different types of hierarchical GLM in rstan
- ▶ Many of the ideas of hierarchical linear models transfer over, but we can explore richer behaviour with hierarchical GLMs
- ▶ These have all used the normal, binomial or Poisson distribution at the top level, and have allowed for over-dispersion, robustness, and ordinal data, to name just three