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

Andrew Parnell
`andrew.parnell@mu.ie`

Maynooth University
National University of Ireland Maynooth

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

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

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);
}
'
```

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

Fitting the models

```r
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)
```

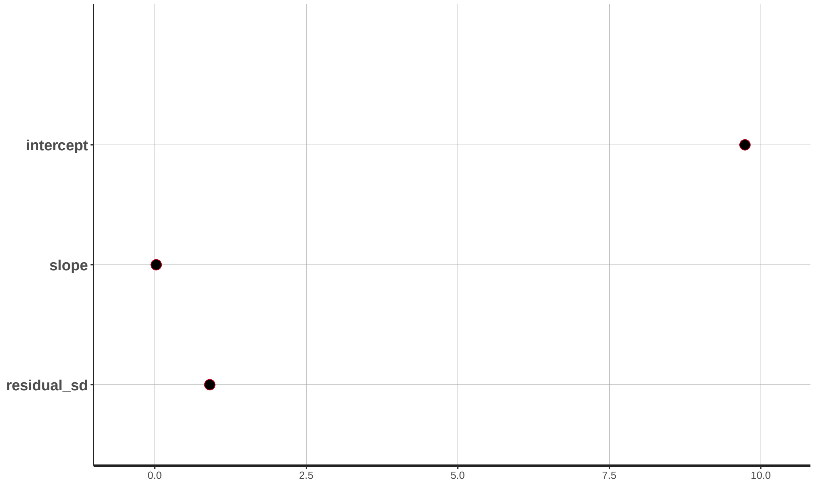
Looking at output

```r
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__        -426.19    0.03 1.22 -429.27 -426.75 -425.87 -425.30 -424.81
##             n_eff Rhat
## intercept    2958    1
## slope        4000    1
## residual_sd  2853    1
## lp__         1996    1
##
## Samples were drawn using NUTS(diag_e) at Thu Oct 11 11:58:41 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).
```

Looking at output 2

```r
plot(stan_run)
```



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);
}
'
```

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-cuachy. You will see this lots in the `rstanarm` and `rstan` examples

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 of 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

Mixed effects models in rstan

```
stan_code_mm = '
data {
  int N;
  int N_eth;
  vector[N] x;
  vector[N] y;
  int eth[N];
}
parameters {
  vector[N_eth] intercept;
  real slope;
  real mean_intercept;
  real<lower=0> residual_sd;
  real<lower=0> sigma_intercept;
}
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_intercept);
  }
  mean_intercept ~ normal(11, 2);
  sigma_intercept ~ cauchy(0, 10);
  residual_sd ~ cauchy(0, 10);
}
'
```

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(earnin
                  model_code = stan_code_mm)
```

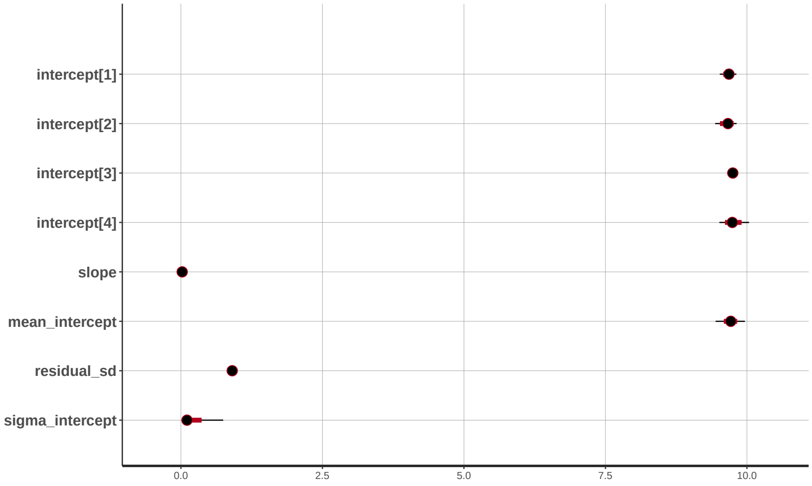
Output 1

```r
print(stan_run_2)
```

```
## Inference for Stan model: b5c22af963cb15d8d6f60b78c007c3e1.
## 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 0.07    9.52    9.63    9.68    9.73
## intercept[2]       9.65    0.00 0.10    9.44    9.59    9.66    9.72
## 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    9.68    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    9.45    9.66    9.71    9.76
## residual_sd        0.91    0.00 0.02    0.87    0.89    0.91    0.92
## sigma_intercept    0.18    0.01 0.24    0.02    0.06    0.11    0.20
## lp__            -421.03    0.18 3.05 -427.53 -422.90 -420.72 -418.93
##                   97.5% n_eff Rhat
## intercept[1]       9.81  1413 1.00
## intercept[2]       9.82  1211 1.00
## intercept[3]       9.81  1547 1.00
## intercept[4]      10.04  1551 1.00
## slope              0.03  4000 1.00
## mean_intercept     9.96  1349 1.00
## residual_sd        0.95  1838 1.00
## sigma_intercept    0.75   716 1.00
## lp__            -415.71   292 1.02
##
## Samples were drawn using NUTS(diag_e) at Thu Oct 11 11:59:10 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).
```

Output 2

```r
plot(stan_run_2)
```



Getting directly at the posterior distribution

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

```
##   intercept[1] intercept[2] intercept[3] intercept[4]
## 1     9.643149     9.637511     9.705220     9.609573 0.
## 2     9.649707     9.628031     9.719524     9.606112 0.
## 3     9.705975     9.795285     9.764903     9.734626 0.
## 4     9.705975     9.795285     9.764903     9.734626 0.
## 5     9.718001     9.769074     9.767792     9.719854 0.
## 6     9.668102     9.654408     9.730741     9.714518 0.
##   mean_intercept residual_sd sigma_intercept      lp__
## 1       9.672665   0.8858085      0.09026650 -419.7149
## 2       9.641047   0.8842357      0.06725221 -418.7544
## 3       9.755040   0.8966953      0.04013150 -417.1831
## 4       9.755040   0.8966953      0.04013150 -417.1831
## 5       9.698637   0.8983558      0.05460703 -418.3659
## 6       9.647617   0.8942897      0.05606405 -417.6471
```

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_pred[j];
}
'
```

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)
```

Extract out the predictions

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

```
##
## iterations     [,1]     [,2]     [,3]     [,4]     [,5]
##       [1,] 9.685709 9.722660 9.759610 9.796561 9.833512
##       [2,] 9.675222 9.710491 9.745761 9.781030 9.816299
##       [3,] 9.662664 9.695727 9.728790 9.761854 9.794917
##       [4,] 9.650314 9.685256 9.720199 9.755142 9.790084
##       [5,] 9.650852 9.684599 9.718345 9.752091 9.785837
##       [6,] 9.687121 9.715935 9.744749 9.773562 9.802376
```

Creating posterior predictive values by hand

```
stan_code_4 = '
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);
}
generated quantities {
  vector[N] y_pred;
  for (j in 1:N)
    y_pred[j] = normal_rng(intercept + slope * x[j], residual_sd);
}
'
```

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);
}
'
```

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. Philosphically WAIC is the more satisfactory but practically LOO seems to work better
- WAIC falls under the framework of *Information Criterion*, LOO is slightly different

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)

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 acording to an IC might still be useless!

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

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:

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

  where p_D is the *effective number of parameters*
- The second called the Widely Applicable Information Criterion (WAIC) which is calculated as:

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

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

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

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)

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

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

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

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 + (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       1.1
```

Now with WAIC

```r
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
##
## elpd_diff        se
##       0.3       1.1
```

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