Generalised Linear Models in STAN follow along

Edwin Yánez

2025-04-27

Table of contents

Purpose

This is a follow-along document reporting my engagement with Coding Club's Generalised linear models in Stan tutorial. Not everything in the tutorial is expected to be replicated here.

The following libraries are used in this tutorial:

```
library(rio)
library(tidyverse)
library(rstanarm)
library(brms)  # for models
library(bayesplot)
library(ggplot2)
library(dplyr)
library(tidybayes)
library(modelr)
```

```
# Custom function to get a sense of the data as a dataframe:
my_glimpse <- function(df, nn = 7) {
  df <- df %>%
    mutate(across(everything(), ~ if_else(is.character(.) & str_detect(.,
     ↪  "^\\s*$"), NA, .)))

  tibble::tibble(
  Variable = names(df),
  N_distinct = unname(purrr::map_int(df, dplyr::n_distinct)),
  NAs = unname(purrr::map_int(df, ~ sum(is.na(.)))),
  Types = unname(purrr::map_chr(df, ~ paste(class(.), collapse = ', '))),
  Content = unname(purrr::map_chr(df, ~ {
    vals_unique <- unique(.)
    n_vals_unique <- length(vals_unique)
    if (n_vals_unique == 0)
      ''
    else if (n_vals_unique > nn)
      paste(paste0(head(vals_unique, nn), collapse = ', '), ',...')
    else
      paste(vals_unique, collapse = ', ')
  }))
  )
}
```

Introduction

Finding answers to our research questions often requires statistical models. Designing models, choosing what variables to include, which data distribution to use are all worth thinking about carefully. In this tutorial, we will continue exploring different model structures in search of the best way to find the answers to our research questions. We will build on the Coding Club tutorials on how to design a model, and on Bayesian Modelling in `MCMCglmm` for key background information on model design and Bayesian statistics.

Statistical models can be fit in a variety of packages in `R` or other statistical languages. But sometimes the perfect model that you can design conceptually is very hard or impossible to implement in a package or programme that restricts the distributions and complexity that you can use. This is when you may want to move to a statistical programming language such as `Stan`. For an introduction to `Stan`, you can check out our intro tutorial here.

In this tutorial, we will learn about two packages, `rstanarm` and `brms` which allow us to fit `Stan` models using syntax similar to packages like `lme4`, `nlme` and `MCMCglmm`. We will use these packages to fit models that test how species richness has changed over time near Toolik Lake Field Station.

Use the `rstanarm` package to run a `Poisson` model

Research question: How has plant species richness changed over time at Toolik Lake?

Hypothesis: Plant species richness has increased over time at Toolik Lake.

```
# Load data
toolik_richness <- import('toolik_richness.csv') %>% as_tibble()
s_toolik_richness <- my_glimpse(toolik_richness)
s_toolik_richness
```

```
# A tibble: 11 x 5
   Variable       N_distinct   NAs Types     Content
   <chr>               <int> <int> <chr>     <chr>
 1 V1                  17493     0 integer   1, 2, 3, 4, 5, 6, 7 ,...
 2 Year                    4     0 integer   2012, 2011, 2010, 2008
 3 Site                    5     0 character MAT, 06MAT, DH, MNT, SAG
 4 Treatment              19     0 character NFCT, NFNP, CT, NP, 06CT, 06F0.5, ~
 5 Block                   9     0 character 1, 2, 3, 4, 6, 12, 13 ,...
 6 Plot                    8     0 integer   1, 2, 3, 4, 5, 6, 7 ,...
 7 Species               115     0 character moss, lichen, litter, Bet nan, Car~
 8 Relative.Cover       4337    41 numeric   0.016, 0.008, 0.048, 0.176, 0.288,~
 9 Mean.Temp               4     0 numeric   -9.45, -8.1, -8.566666667, -8.2833~
10 SD.Temp                 4     0 numeric   16.82598099, 14.61051925, 15.37887~
11 Richness               28     0 integer   15, 14, 16, 13, 17, 12, 18 ,...
```

```
liaison <- toolik_richness
```

`Site` and `Species` are strings (letters) and categorical data (factors) - they are names. `Year`, `Cover`, `Mean.Temp` and `SD.Temp` are numeric and continuous data - they are numbers. `Cover` shows the relative cover (out of 1) for different plant species, `Mean.Temp` is the mean annual temperature at Toolik Lake Station and `SD.Temp` is the standard deviation of the mean annual temperature. Then we have `Treatment`, another categorical variable that refers to different chemical treatments, e.g. some plots received extra nitrogen, others extra phosphorus. Finally, we have `Block` and `Plot`, which give more detailed information about where the measurements were taken.

The plot numbers are currently coded as numbers - 1, 2,...8 and they are a numerical variable. We should make them a categorical variable, since just like `Site` and `Block`, the numbers represent the different categories, not actual count data.

> **Note**
>
> Note from Edwin: In essence, what the tutorial is saying is that the following variables should be a factor: `Site`, `Species`, `Treatment`, `Block`, and `Plot`.

```
toolik_richness <- liaison
toolik_richness <- toolik_richness %>%
  mutate(
    Site = as.factor(Site),
    Species = as.factor(Species),
    Treatment = as.factor(Treatment),
    Block = as.factor(Block),
    Plot = as.factor(as.character(Plot))
  )
```

Now, let's think about the distribution of the data, specifically our response variable, species richness (`Richness`).

```
p <- ggplot(toolik_richness, aes(x = Richness)) +
    geom_histogram() +
    theme_classic()
p
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

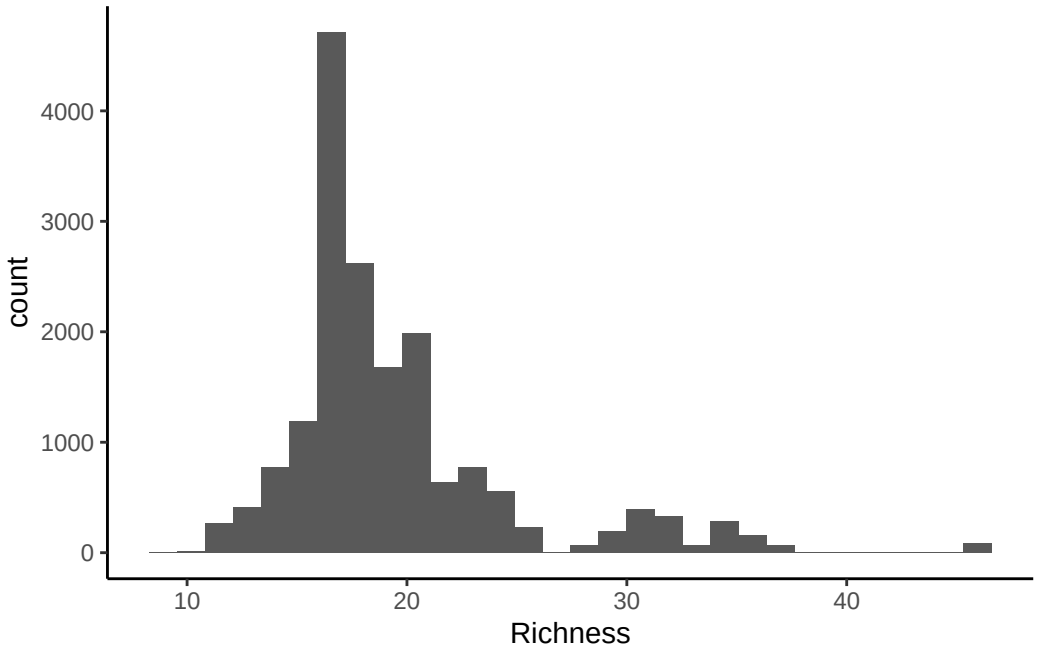


Figure 1: Histogram of Richness

We are working with integers (similar to count data, species only come in whole numbers) and the data are right-skewed, that is because we have a few data points on the extreme

right end, they are pulling the mean and median or our data to the right. For ecological data, this is quite common - across all of the sampling plots, we expect that they won't all have lots of different species within them.

This means that a `Poisson` distribution might be suitable for our model.

One advantage to fitting models in `Stan` is that it's easy to fully take advantage of your computing power. In `Stan`, we usually run two or more **chains** - different iterations of our model which we can then compare - if they are massively different, something is not right. If your computer has two or more cores, you can split the chains, i.e., run a chain on each core, saving you some time, as the code will then finish running quicker. This means that you'll be running the code in parallel.

To enable parallel computing, you can run this line of code and then later on in the model code, you can specify how many cores you want to use.

```
options(mc.cores = parallel::detectCores())
```

Now we are all set up for our first model. Remember that the data have a hierarchical structure - species richness is measured in plots, which fall within blocks that are then part of different sites. To derive inferences about changes species richness through time, our models should take this complexity of the data structure into account.

One disadvantage to `Stan` models is that the code can take a while to finish running, even if we use all of our computer cores, it can still be hours before you can see a summary of your model. Of course, one should use the most suitable type of model for a research question, regardless of how long the model takes to run (within reason…), but in our case, for teaching purposes and so that you can see some model outputs today, we will focus on species richness change within just one of the plots. In that case, the model does not need to include random effects, because on the plot level, there is no replication.

Just so that you know what the syntax looks like and if you have time to wait for this code to finish running, this is how the model for species richness change around Toolik Lake in general would look like:

```
# # Note - this code could take hours to run!
# # We are running the model using the default weakly informative priors.
# # More about priors coming later in the tutorial!
# stan_lm <- stan_glmer(
#   Richness ~ I(Year-2007) + (1|Site/Block/Plot),
#   data = toolik_richness,
#   family = poisson,
#   chains = 4,
#   cores = 4
# )
#
# # Assess converge by looking at the trace plots
```

```
# plot(stan_lm, plotfun = "trace")
#
# # Explore the summary output
# summary(stan_lm)
```

> **Note**
>
> **Note from Edwin**: I did end up running the model. The longest chain ended up taking x minutes. I have saved the environment up to that point for further reference as `long-model.RData`.

> **Note**
>
> For teaching purposes only, we will proceed with a model without random effects - that way you can see the outputs of the models as you are going through the tutorial. Note that we do not advise doing this when you are analysing your data for real - of course a model that takes into account the hierarchical structure of the data would be better.

Advantages and disadvantages of using `Stan`

`Stan` models can take a long time to compile. One of their key advantage is that you have a lot of freedom to specify the priors (your prior knowledge and expectations of how a given parameter will behave) for your model parameters and you can really account for the complex structure your data might have. There is a time cost to this, as we've seen above. One way to approach this is to first make sure your code works on a small chunk of your data, and only afterwards, you can start running the code on the full data (and do other things while you wait for the models to compile).

Now we can run our simplified model. First, let's check how many years of data we have:

```
unique(toolik_richness$Year)
```

```
[1] 2012 2011 2010 2008
```

There are four years of data from 2008 to 2012. For modelling purposes, it helps to transform the year variable into a continuous variable where the first year is year one, i.e., 2008 is 1, 2009 is 2. That way, the model won't have to estimate richness in the year 500, the year 501, etc., it will start straight from our first monitoring year.

```
# Note how now we are using stan_glm because
# there are no random effects
stan_glm1 <- stan_glm(
  Richness ~ I(Year-2007),
  data = toolik_richness,
  family = poisson,
  chains = 4,
  cores = 4,
  seed = 123
)
```

If you find this code still takes a long time, you can change the `chains` argument to only two chains, but note that it's better to run models with more than two chains - then you have more room for comparison. If one or more of the four chains is behaving really differently from the rest, then the model might not have converged. What is model convergence? In brief, if a model hasn't converged, you can't trust the estimates it gives you. You can find more details in the model design tutorial here.

Assessing model convergence

One way to assess model convergence is by visually examining the trace plots. They should be fuzzy with no big gaps, breaks or gigantic spikes.

```
plot(stan_glm1, plotfun = "trace")
```

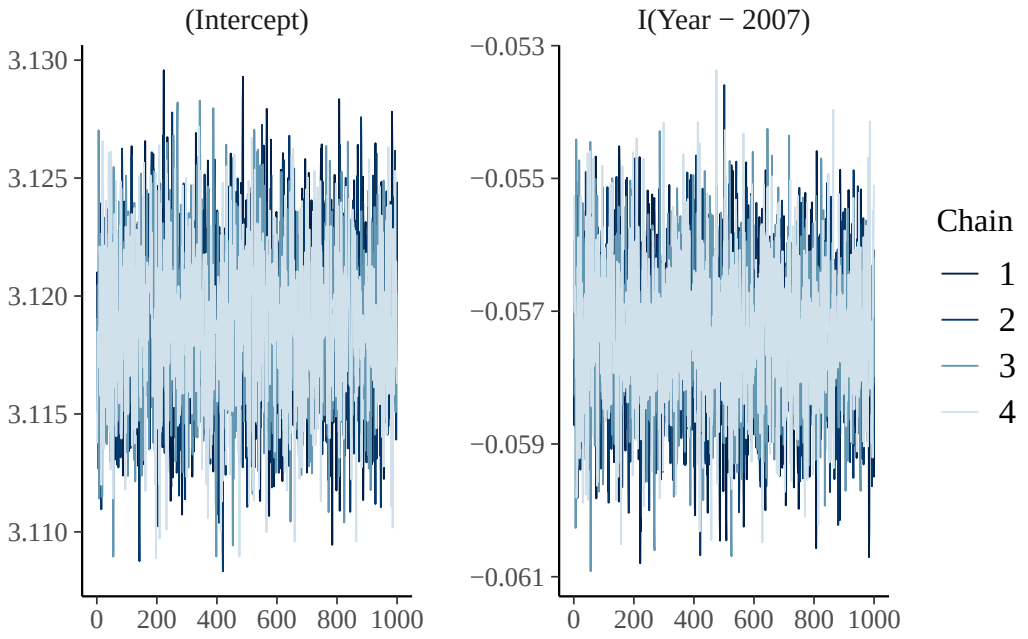


Figure 2: Traceplots for the simple model.

Here, the trace plots look fine.

Next we can look at the summary output.

```
summary(stan_glm1)
```

```
Model Info:
 function:     stan_glm
 family:       poisson [log]
 formula:      Richness ~ I(Year - 2007)
 algorithm:    sampling
 sample:       4000 (posterior sample size)
 priors:       see help('prior_summary')
 observations: 17493
 predictors:   2

Estimates:
                 mean   sd   10%   50%   90%
(Intercept)      3.1    0.0  3.1   3.1   3.1
I(Year - 2007) -0.1    0.0 -0.1  -0.1  -0.1

Fit Diagnostics:
           mean   sd   10%   50%   90%
mean_PPD 19.4    0.0 19.4  19.4  19.5
```

```
The mean_ppd is the sample average posterior predictive distribution of the outcome variab

MCMC diagnostics
               mcse Rhat n_eff
(Intercept)    0.0  1.0  2710
I(Year - 2007) 0.0  1.0  3451
mean_PPD       0.0  1.0  1930
log-posterior  0.0  1.0  1746

For each parameter, mcse is Monte Carlo standard error, n_eff is a crude measure of effect
```

We can see that the effective sample size is alright (there is no hard cut threshold, but more than around 1000 is usually a good sign), another diagnostic metric, the Rhat value is also indicating convergence (an Rhat of 1 is a good sign, more than 1 could indicate trouble).

Posterior predictive checks

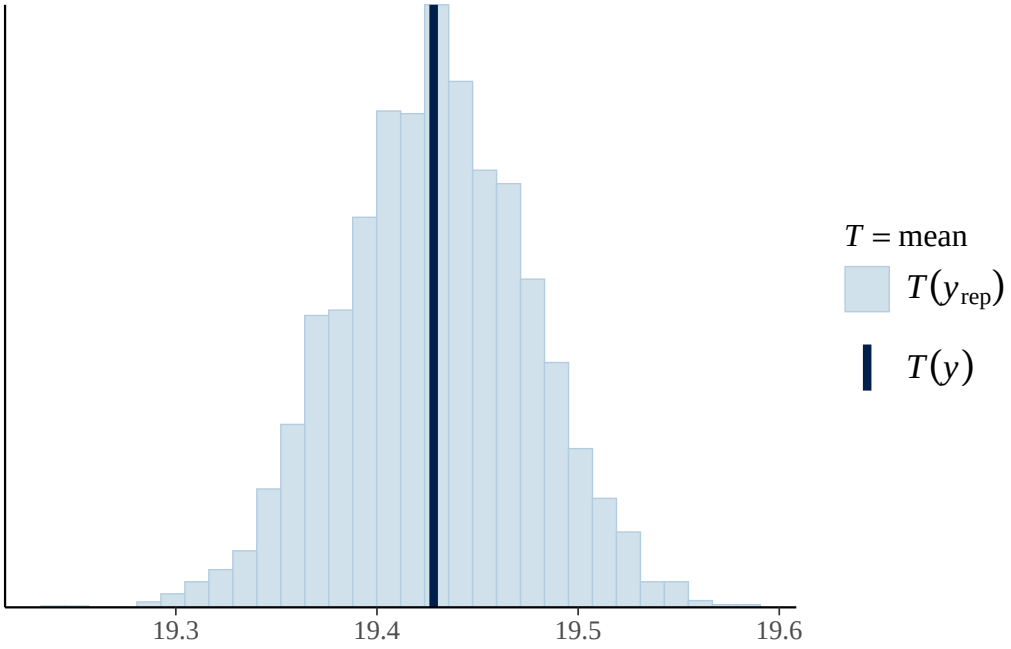
The pre-compiled models in `rstanarm` already include a `y_rep` variable (our model predictions) in the generated quantities block (your posterior distributions). We can use the `pp_check` function from the `bayesplot` package to see how the model predictions compare to the raw data, i.e., is the model behaving as we expect it to be?

```
pp_check(stan_glm1, plotfun = "stat", stat = "mean")
```

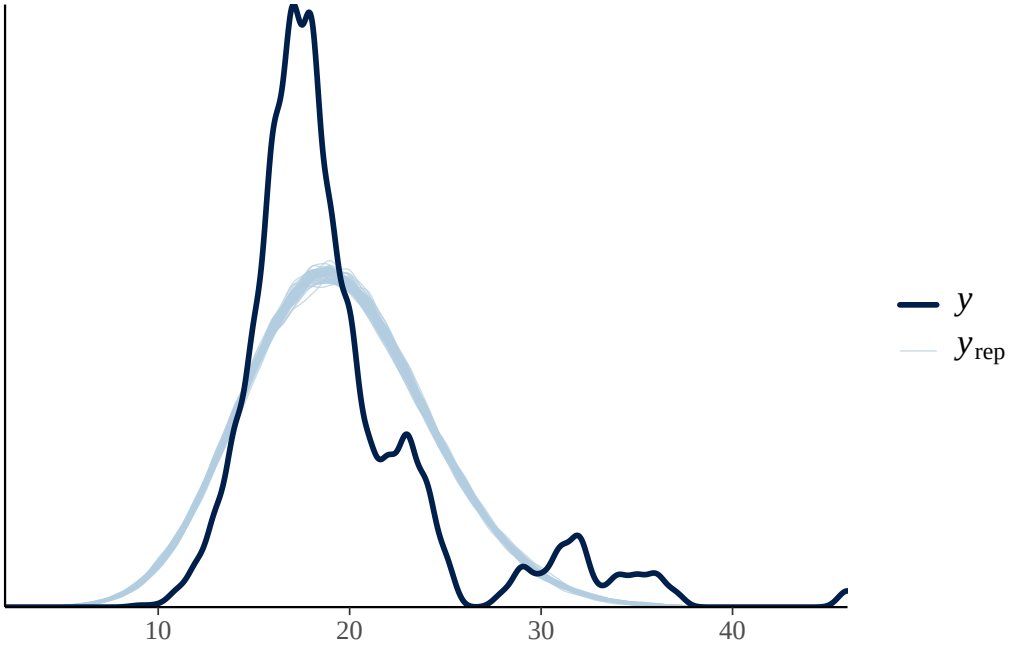
```
Note: in most cases the default test statistic 'mean' is too weak to detect anything of in

`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
pp_check(stan_glm1, plotfun = "dens_overlay")
```



(a) Histogram



(b) Density overlay

Figure 3: Posterior overlay checks.

What do you think? How does the density distribution of the model predictions compare with that of the raw data?

From the histogram, we can see that the posterior mean (the dark blue line) aligns well with the raw data. From the density plot, we can see that the model is underpredicting the

low species richness value. This is a common problem when working with left-skewed or zero/low number-inflated data. Overall though, the model predictions follow the underlying data, so in summary, we can say that this model fit is acceptable. Note that these decisions can vary based on your question, data and even your statistical philosophy.

Model diagnostics using `shinystan`

For an interactive exploration of `Stan` models, you can use the `shinystan` app. The app is compatible with `Stan` models generated using the `rstan`, `rstanarm` and `brms` packages, so regardless of how you chooose to run your `Stan` models, you can still use `shinystan` to assess whether or not they've converged and are behaving properly.

We can launch the app using the code below. That will open a window in our internet browser, where we can further explore our model.

```
if (FALSE) {
  launch_shinystan(stan_glm1)
}
```

Have a go at exploring the various options - you'll probably spot some of the plots we've already made in `R`, but there are many others as well. Here, for example, if there were any divergent chains (i.e. a chain that is behaving in a very weird unexpected way), we would have seen red points. In our case, we don't have any divergent chains.

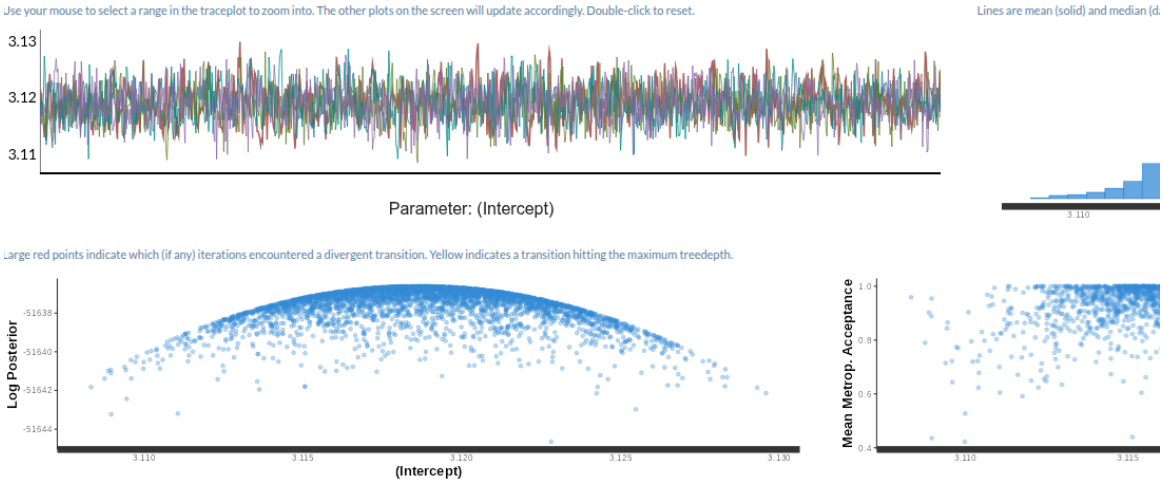


Figure 4: Example plots from Shiny Stan