Class 8: Bayesian Linear and generalised linear models (GLMs)

Andrew Parnell
andrew.parnell@mu.ie

Maynooth University
National University of Ireland Maynooth

Learning outcomes

- Understand the basic formulation of a GLM in a Bayesian context
- Understand the code for a GLM in `rstanarm`
- Be able to pick a link function for a given data set
- Know how to check model assumptions for a GLM

Aside: thinking about the data generating process for a standard LM

If we believe that a linear model is appropriate for our data, there are several ways we could generate data from the model. Here is one way:

```
N = 10
x = 1:N
y = rnorm(N, mean = -2 + 0.4 * x, sd = 1)
```

Here is another:

```
eps = rnorm(N, mean = 0, sd = 1)
y = -2 + 0.4 * x + eps
```

The data generating process for a logistic regression

- What if the response variable was binary? Clearly the previous code will not produce binary values - Instead we could simulate from the binomial distribution:

```
y = rbinom(N, size = 1, prob = -2 + 0.4 * x)
```

. . . but this will produce NAs as the prob argument needs to be between 0 and 1. We need to transform the values involving the covariate

- A popular way is to use the inverse logit function. Look!

```
-2 + 0.4 * x
```

```
##  [1] -1.6 -1.2 -0.8 -0.4  0.0  0.4  0.8  1.2  1.6  2.0
```

```
exp(-2 + 0.4 * x)/(1 + exp(-2 + 0.4 * x))
```

```
##  [1] 0.1679816 0.2314752 0.3100255 0.4013123 0.5000000 0.5986877 0.6899745
##  [8] 0.7685248 0.8320184 0.8807971
```

- In fact you can take any number a from $-\infty$ to ∞ and create $\exp(a)/(1+\exp(a))$ and it will always lie between 0 and 1

Generating binomial data

- Thus a way to generate binary data which allows for covariates is:

```
library(boot)
p = inv.logit(-2 + 0.4 * x)
y = rbinom(N, size = 1, prob = p)
y
```

```
##  [1] 0 1 1 1 0 0 1 0 0 1
```

- The logit function itself is $\log\left(\frac{p}{1-p}\right)$ and will turn the probabilities form the range (0,1) to the range $(-\infty, \infty)$
- Using this type of model is known as *logistic-Binomial* regression and the logit is known as the *link function*

Generating other types of data

- Once we have discovered link functions, we can use them to generate other types of data, e.g. Poisson data via the log link:

```
lambda = exp(-2 + 0.4 * x)
y = rpois(N, lambda)
y
```

```
##  [1] 0 0 1 1 0 2 2 3 8 2
```

- The rate (λ) of the Poisson distribution has to be positive, so taking the log of it changes its range to $(-\infty, \infty)$ as before. The inverse-link (exp) turns the unrestricted ranges into something that must be positive

From LM to GLM

- In general, a *generalised linear model* (GLM) can be written out as:

$$y \sim Distribution(f(\theta, x))$$

  where *Distribution* is some probability distribution, θ are some parameters, and f is a link function that transforms the parameters into a range so that we can incorporate x in an unrestricted way
- The above allows us to simulate from the model, given some parameters θ and some covariates x we can use the probability distribution to get simulated data
- It also allows us to calculate the *likelihood* as we can get a score for how likely it is to see the data we have observed given some values of the parameters

Multiple covariates

- We can extend LMs and GLMs to have multiple covariates if we want, e.g.

```
y = rnorm(N, mean = -2 + 0.4 * x1 - 0.3 * x2, sd = 1)
p = inv.logit(-2 + 0.4 * x1 - 0.3 * x2)
y = rbinom(N, size = 1, prob = p)
```

- Alternatively we can incorporate multiplicative interactions...

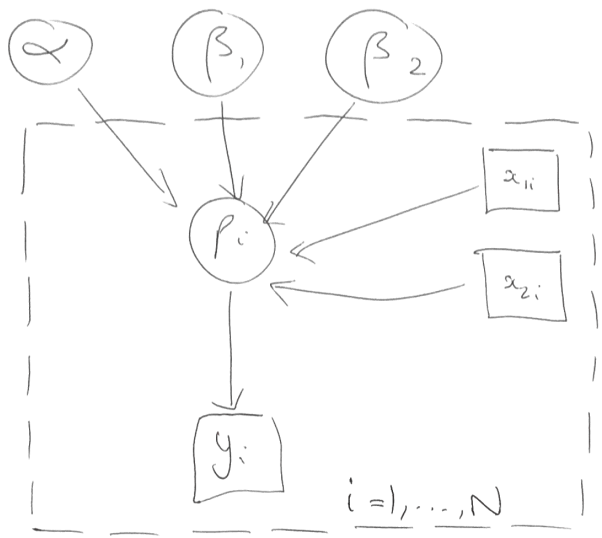
```
y = rnorm(N, mean = -2 + 0.4 * x1 - 0.3 * x2 +
              0.05 * x1 * x2, sd = 1)
```

- ... or non-linear effects

```
p = inv.logit(-2 + 0.4 * x1 - 0.3 * x2 - 0.02 * x1^2)
y = rbinom(N, size = 1, prob = p)
```

Directed Acyclic Graphs

- Once we have decided on a model, it is often a good idea to draw a picture of it to make it clear how it works
- In Bayesian statistics, this is commonly done using a Directed Acyclic Graph or DAG which tells us how to simulate from the model. Circles indicate parameters, squares data, and the dotted lines indicate loops
- Here is a DAG for the logistic regression model with two covariates:



Example: earnings data

- Going back to the earnings data, suppose we want to fit a model to predict log earnings based on sex and whether respondent is white (`eth==3`) or not
- The model is:

$$\log(\text{earnings}) \sim N(\alpha + \beta_1 \text{height} + \beta_2 \text{white}, \sigma^2)$$

- We want to get the posterior distribution of α, β_1, β_2 and σ given the data
- What prior distributions could we set on these parameters?

Fitting linear regression models in rstanarm

Model code:

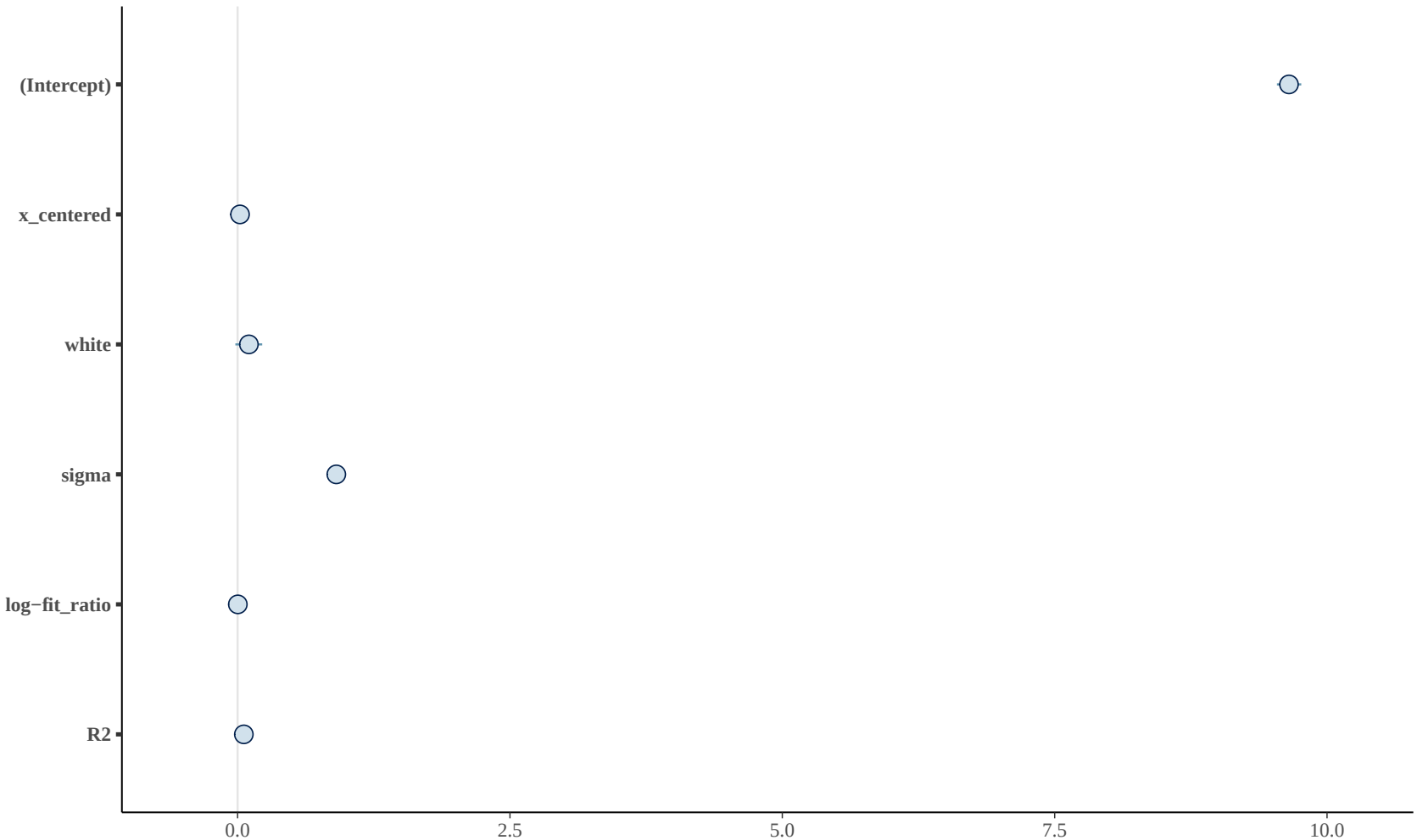
```
earnings = read.csv('data/earnings.csv')
earnings$white = as.integer(earnings$eth == 3)
mod_1 = stan_lm(y ~ x_centered + white, data = earnings,
                prior = R2(location = 0.5, 'mean'))
```

```
round(as.data.frame(summary(mod_1)), 2)
```

```
##                   mean mcse   sd     2.5%      25%      50%      75%
## (Intercept)       9.65 0.00 0.07     9.52     9.61     9.65     9.70
## x_centered        0.02 0.00 0.00     0.02     0.02     0.02     0.02
## white             0.10 0.00 0.07    -0.04     0.06     0.10     0.15
## sigma             0.91 0.00 0.02     0.87     0.89     0.91     0.92
## log-fit_ratio     0.00 0.00 0.02    -0.04    -0.01     0.00     0.02
## R2                0.06 0.00 0.01     0.03     0.05     0.06     0.07
## mean_PPD          9.74 0.00 0.04     9.66     9.71     9.74     9.76
## log-posterior -1402.70 0.06 1.82 -1407.13 -1403.66 -1402.36 -1401.36
##                  97.5% n_eff Rhat
## (Intercept)       9.78  1785    1
## x_centered        0.03  2479    1
## white             0.25  1792    1
## sigma             0.95  2620    1
## log-fit_ratio     0.05  2422    1
## R2                0.09  2548    1
## mean_PPD          9.81  4000    1
## log-posterior -1400.21  1038    1
```

Plot the posterior values

```
plot(mod_1)
```



Using other priors

```
mod_2 = stan_lm(y ~ x_centered + white, data = earnings,
                prior = R2(location = 0.5, 'mean'),
                prior_intercept = normal(0, 10))
```

```
round(as.data.frame(summary(mod_2)), 2)
```

```
##                   mean mcse   sd     2.5%      25%      50%      75%
## (Intercept)       9.65 0.00 0.07     9.52     9.61     9.65     9.70
## x_centered        0.02 0.00 0.00     0.02     0.02     0.02     0.02
## white             0.10 0.00 0.07    -0.04     0.05     0.10     0.15
## sigma             0.91 0.00 0.02     0.87     0.89     0.91     0.92
## log-fit_ratio     0.00 0.00 0.02    -0.04    -0.01     0.00     0.02
## R2                0.06 0.00 0.01     0.03     0.05     0.06     0.07
## mean_PPD          9.74 0.00 0.04     9.66     9.71     9.74     9.76
## log-posterior -1404.11 0.06 1.75 -1408.59 -1405.05 -1403.79 -1402.80
##                  97.5% n_eff Rhat
## (Intercept)       9.78  1446    1
## x_centered        0.03  1880    1
## white             0.25  1605    1
## sigma             0.95  2443    1
## log-fit_ratio     0.04  2333    1
## R2                0.09  1974    1
## mean_PPD          9.81  4000    1
## log-posterior -1401.70  1014    1
```

What do the results actually mean?

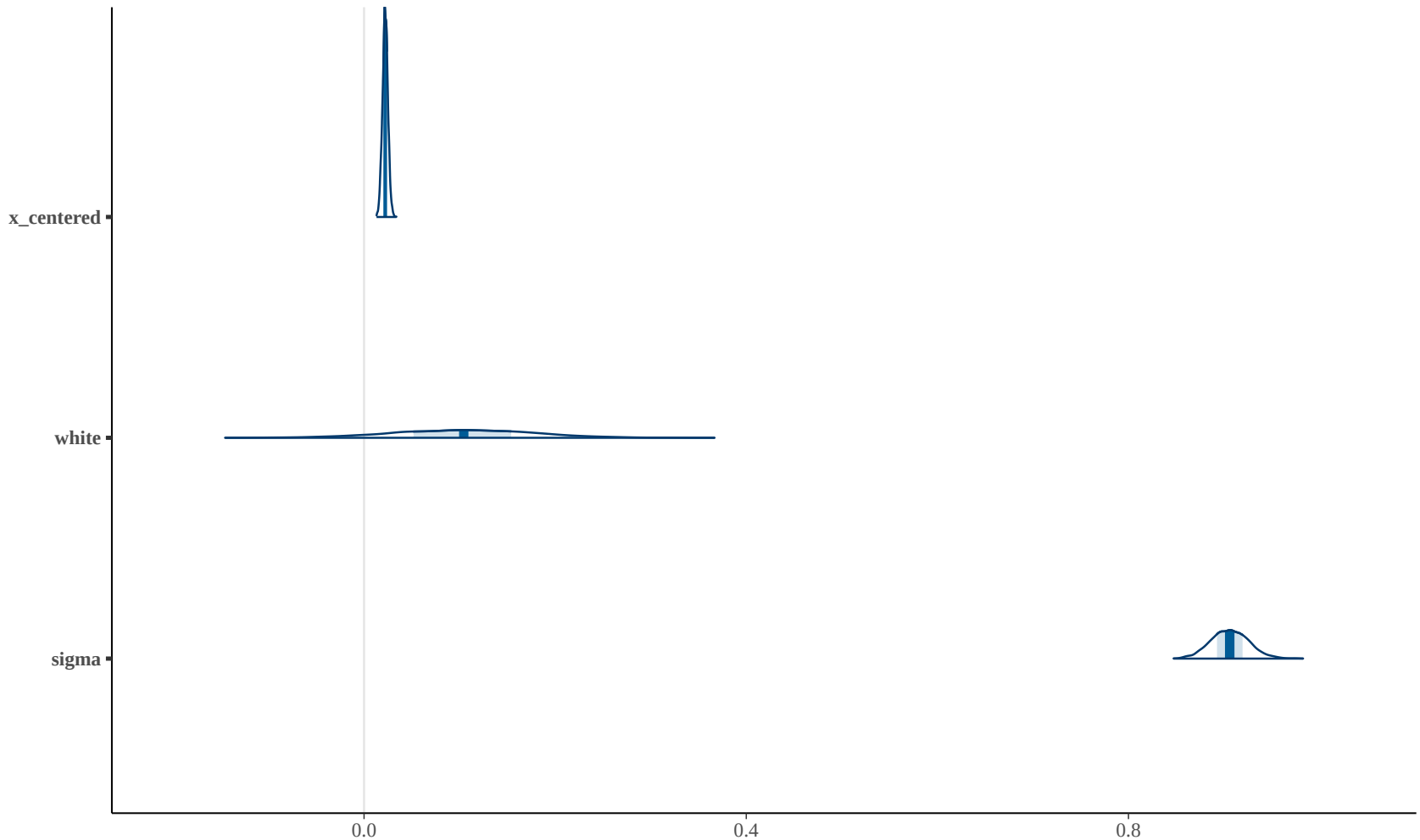
- We now have access to the posterior distribution of the parameters:

```
post = as.data.frame(mod_2)
head(post)
```

```
##   (Intercept) x_centered      white     sigma log-fit_ra
## 1    9.592836 0.02507382 0.21633742 0.9301058  0.036097
## 2    9.551875 0.02282112 0.17018098 0.9139646  0.012325
## 3    9.588871 0.02163994 0.15578411 0.8900420 -0.016006
## 4    9.589588 0.02174304 0.20310271 0.9158380  0.012860
## 5    9.599199 0.02180863 0.19532988 0.9138107  0.010628
## 6    9.680555 0.01996215 0.04551595 0.9121066  0.000715
```

Plots of output

```
library(bayesplot)
mcmc_areas(post,
           pars = c("x_centered", "white", "sigma"))
```



rstan version

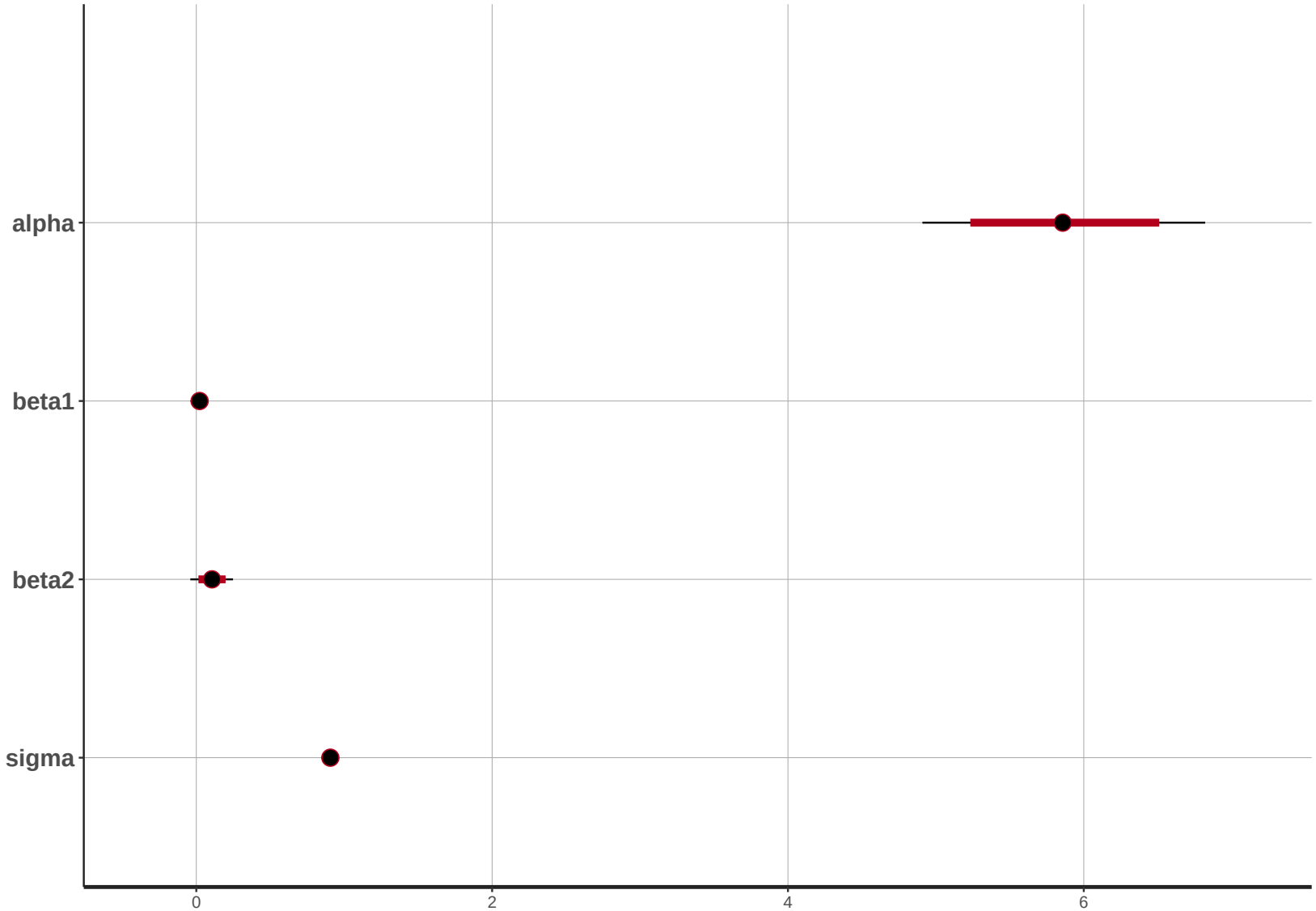
```
stan_code = '
data {
  int<lower=0> N;
  vector[N] y;
  vector[N] x1;
  vector[N] x2;
}
parameters {
  real alpha;
  real beta1;
  real beta2;
  real<lower=0> sigma;
}
model {
  y ~ normal(alpha + x1 * beta1  + x2 * beta2, sigma);
}
'
```

Running the Stan version

```
library(rstan)
stan_run = stan(data = list(N = nrow(earnings),
                            y = earnings$y,
                            x1 = earnings$x,
                            x2 = earnings$white),
                model_code = stan_code)
```

Stan output

```
plot(stan_run)
```



To standardise or not?

- Most regression models work better if the covariates are standardised (subtract the mean and divide by the standard deviation) before you run the model
- `rstan` can struggle with regression models where the data are not standardised. `rstanarm` does a much better job
- The advantage of standardising is that you get more numerically stable results (this is true of `R`'s `lm` function too), and that you can directly compare between the different slopes
- The disadvantage is that the slope values are no longer in the original units (e.g. cm)

What is stan doing in the background?

- Stans run a stochastic algorithm called Hamiltonian Monte Carlo to create the samples from the posterior distribution
- This involves:
  1. Guessing at *initial values* of the parameters. Scoring these against the likelihood and the prior to see how well they match the data
  2. Then iterating:
     - 2.1 Working out which *directions* to try to generate new good parameter values
     - 2.2 Sampling *new parameter values* which may or may not be similar to the previous values
     - 2.3 Repeating these steps to build up a posterior sample of parameter values
- What you end up with is a set of parameter values for however many iterations you chose.

How many iterations?

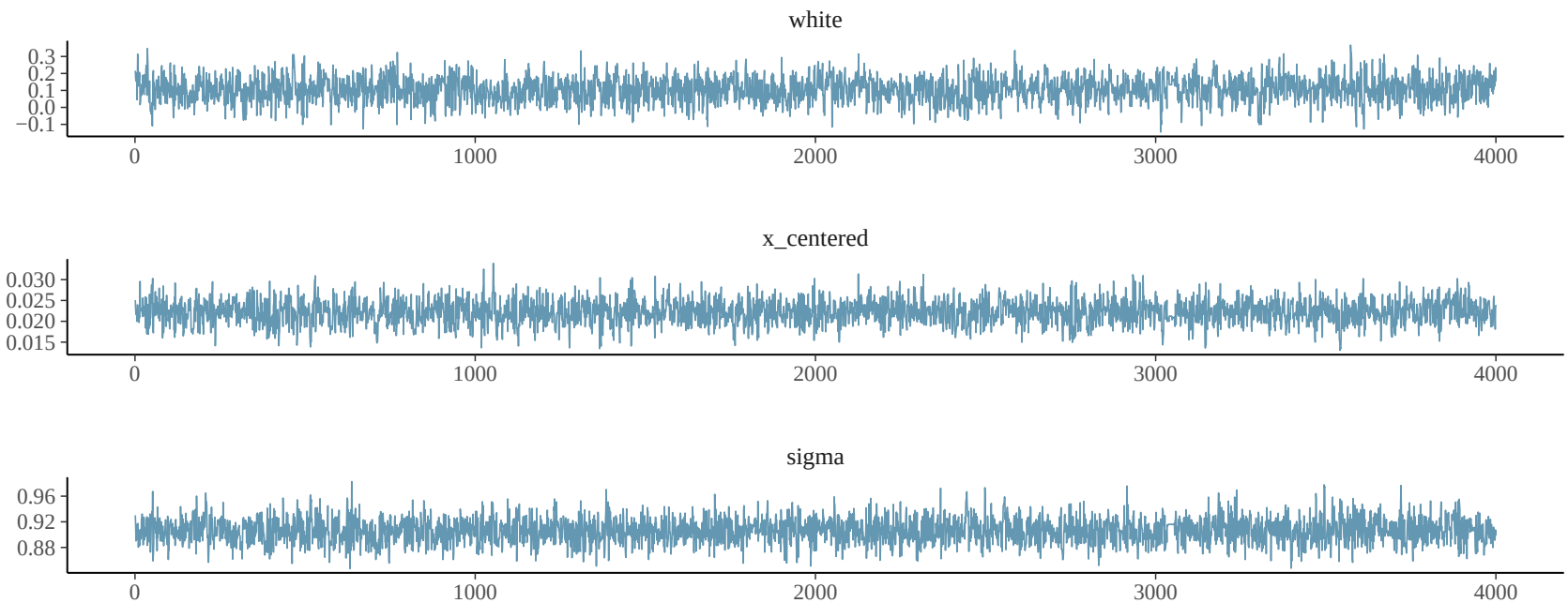
- Ideally you want a set of posterior parameter samples that are independent across iterations and is of sufficient size that you can get decent estimates of uncertainty
- There are three key parts of the algorithm that affect how good the posterior samples are:
    1. The starting values you chose. If you chose bad starting values, you might need to discard the first few thousand iterations. This is known as the *burn-in* period
    2. The way you choose your new parameter values. If they are too close to the previous values the MCMC might move too slowly so you might need to *thin* the samples out by taking e.g. every 5th or 10th iteration
    3. The total number of iterations you choose. Ideally you would take millions but this will make the run time slower

`rstanarm` and `rstan` have good default choices for these but for complex models you often need to intervene

Plotting the iterations

You can plot the iterations for all the parameters with `mcmc_trace`, e.g.

```
mcmc_trace(post, pars = c('white', 'x_centered', 'sigma'),
           facet_args = list(nrow = 3))
```



A good trace plot will show no patterns or runs, and will look like it has a stationary mean and variance

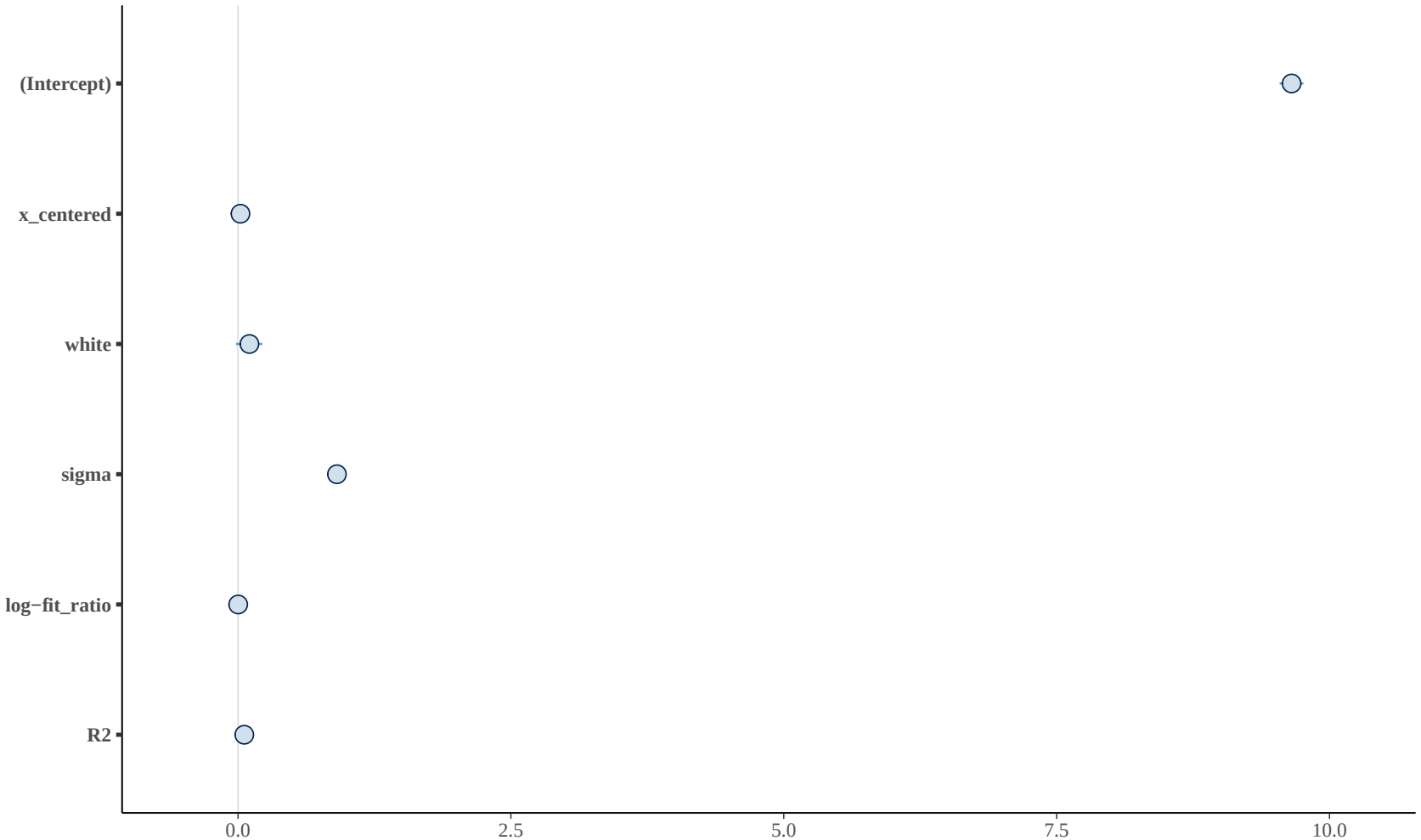
How many chains?

- Beyond increasing the number of iterations, thinning, and removing a burn-in period, Stan automatically runs *multiple chains*
- This means that they start the algorithm from 3 or 4 different sets of starting values and see if each *chain* converges to the same posterior distribution
- If the MCMC algorithm has converged then each chain should have the same mean and variance.
- Stan reports the `Rhat` value, which is close to 1 when all the chains match
- It's about the simplest and quickest way to check convergence. If you get `Rhat` values above 1.1, run your MCMC for more iterations

What else can I do with the output

- We could create *credible intervals* (Bayesian confidence intervals):

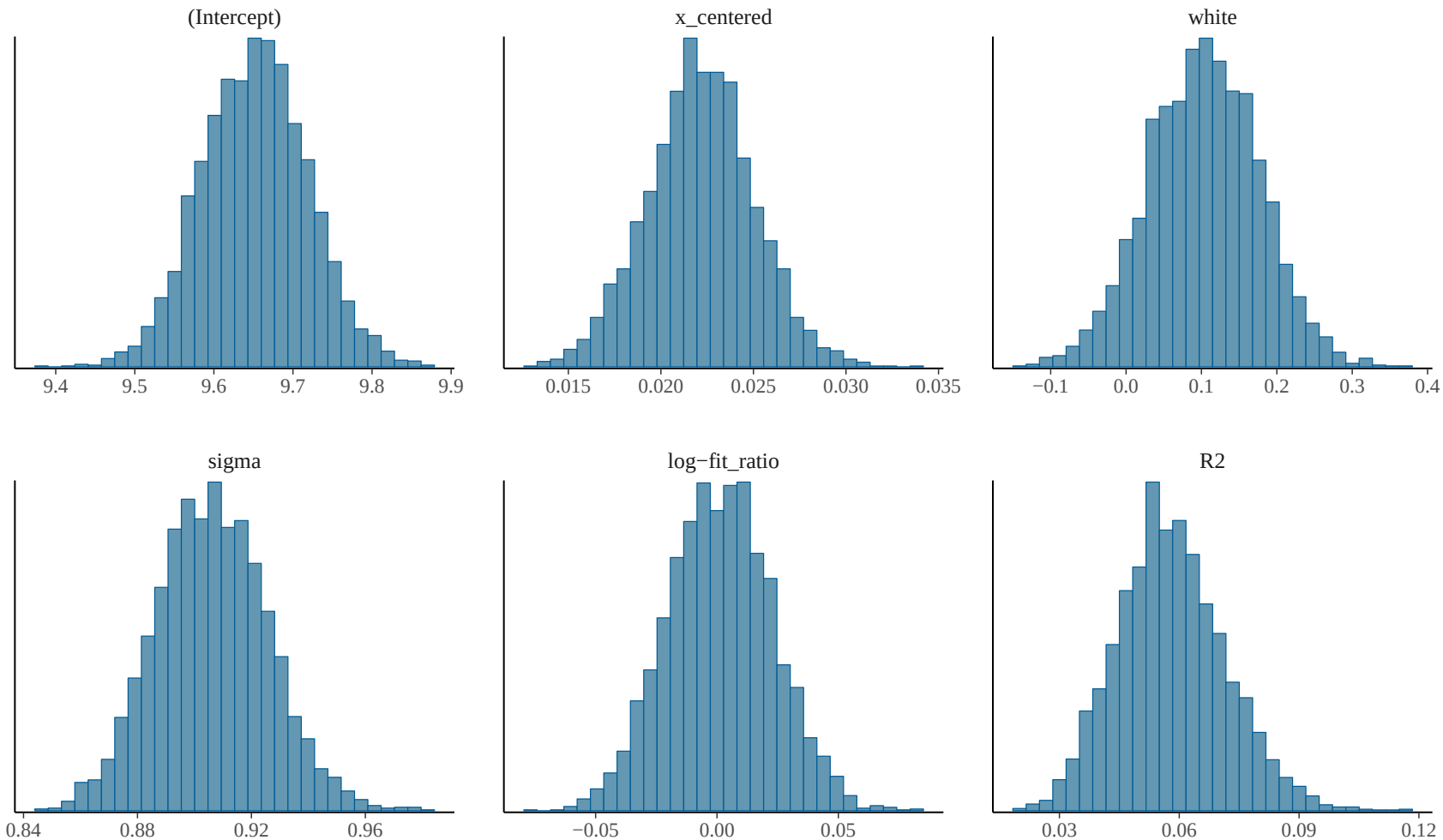
```
mcmc_intervals(post)
```



Or histograms

```
mcmc_hist(post)
```

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



Checking model fit

- How do we know if this model fits the data well or not?
- One way is to simulate from the posterior distribution of the parameters, and subsequently simulate from the likelihood to see if the these data match the real data we observed
- This is known as a *posterior predictive check*

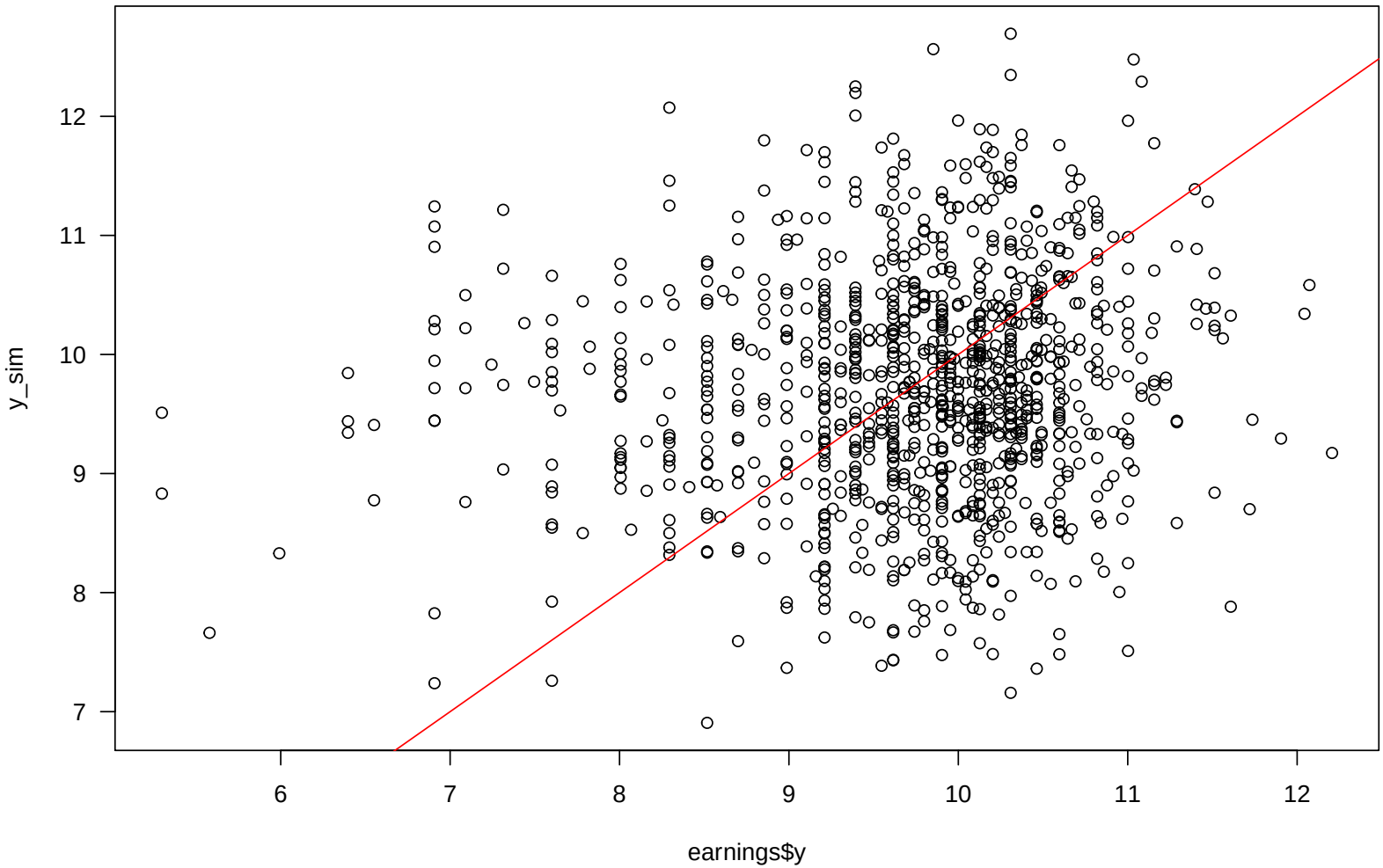
Posterior predictive: the long way

- The long way of doing this is in R after running the model
- For each value sampled from the posterior, compute:

```
y_sim = rnorm(nrow(earnings),
              post$`(Intercept)`[1] +
                post$x_centered[1] * earnings$x_centered +
                post$white[1] * earnings$white,
              sd = post$sigma[1])
plot(earnings$y, y_sim)
abline(a = 0, b = 1, col = 'red')
```

If the model is good, these should form a straight line!

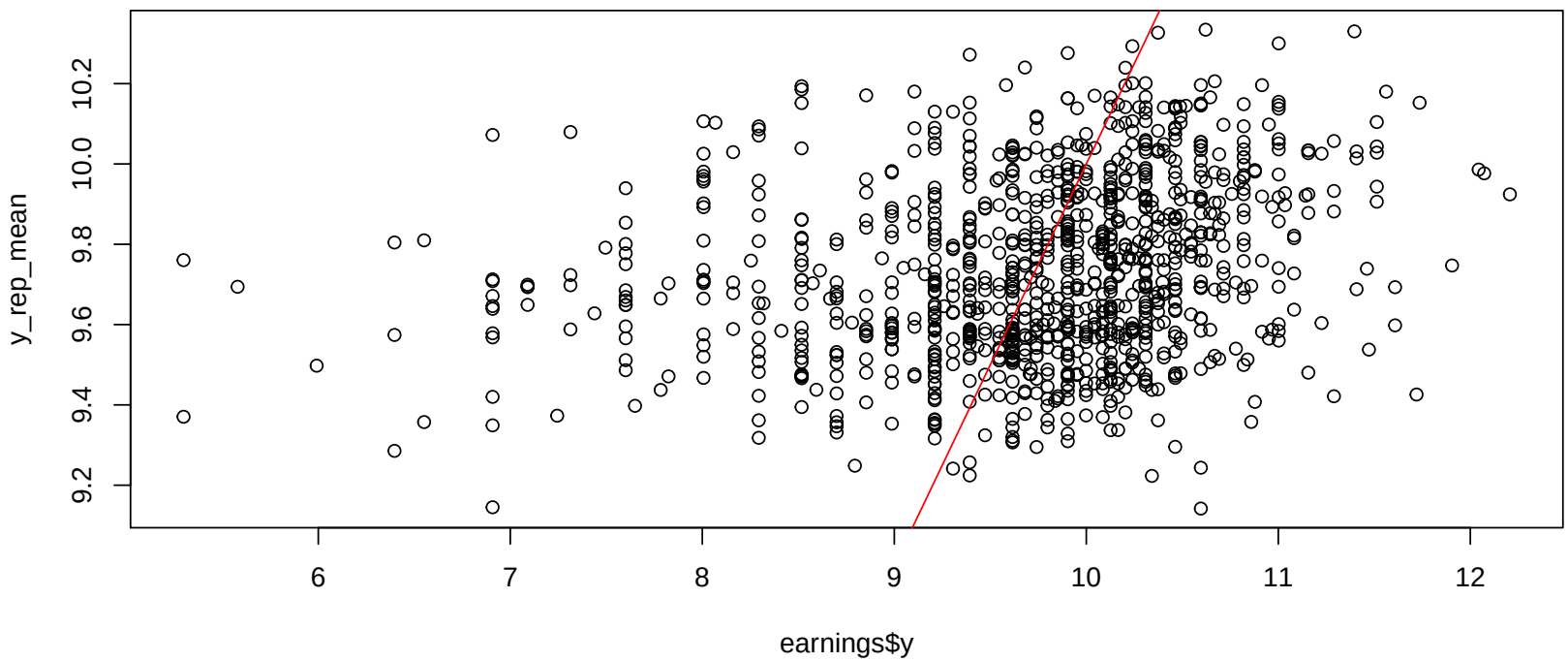
Posterior predictive plot for one iteration



Easier posterior predictive distributions

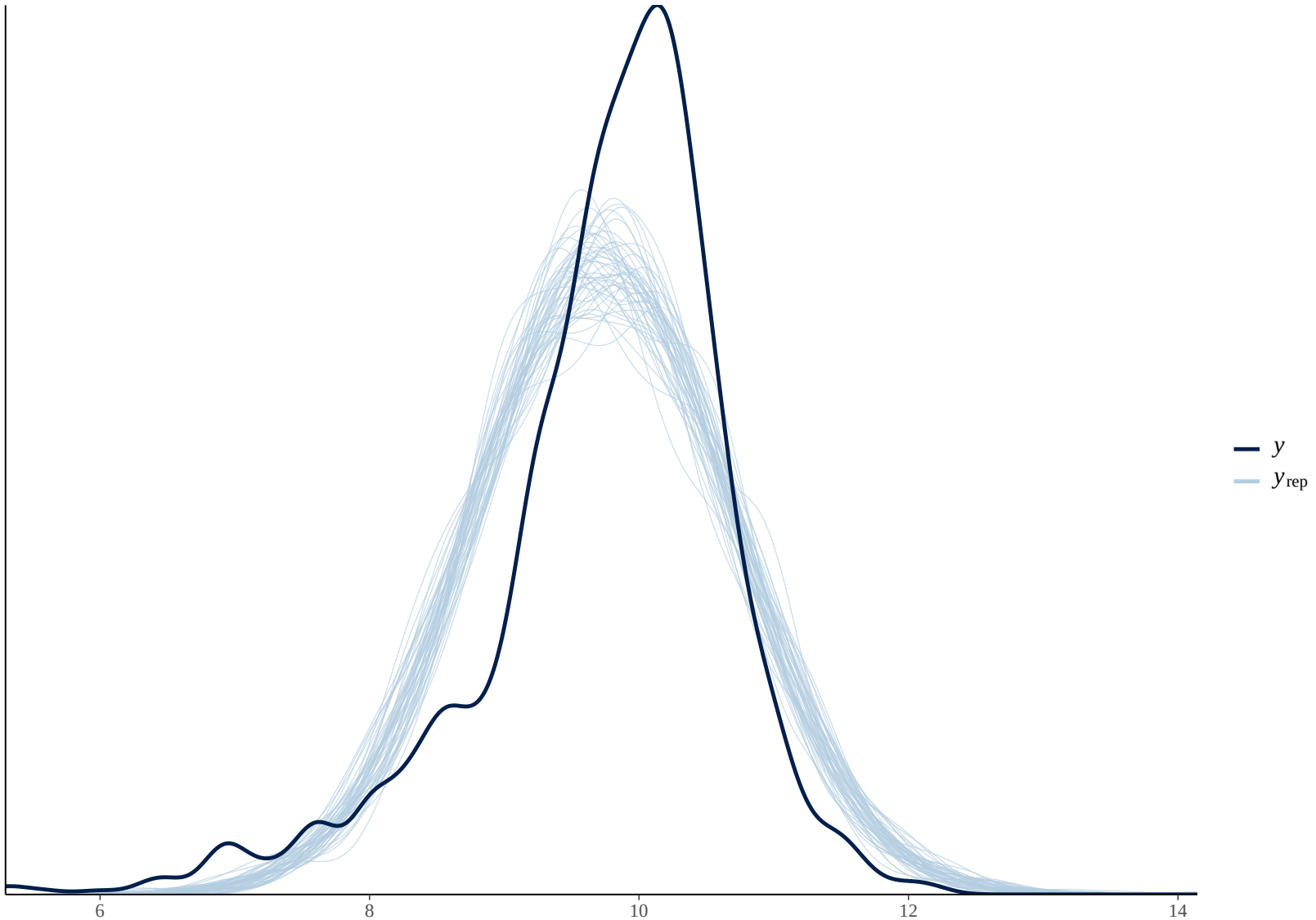
- The easier way is to use the `posterior_predict` command in `rstanarm`:

```
y_rep = posterior_predict(mod_2)
y_rep_mean = apply(y_rep, 2, 'mean')
plot(earnings$y, y_rep_mean)
abline(a = 0, b = 1, col = 'red')
```



More posterior predictive checks

```
pp_check(mod_2)
```



rstanarm GLMs: Swiss Willow tit data

Recall the Willow tit data:

```
swt = read.csv('data/swt.csv')
head(swt)
```

```
##   rep.1 rep.2 rep.3 c.2 c.3 elev forest dur.1 day.2 day.3 length alt
## 1     0     0     0   0   0  420      3   240    58    73    6.2 Low
## 2     0     0     0   0   0  450     21   160    39    62    5.1 Low
## 3     0     0     0   0   0 1050     32   120    47    74    4.3 Med
## 4     0     0     0   0   0 1110     35   180    44    71    5.4 Med
## 5     0     0     0   0   0  510      2   210    56    73    3.6 Low
## 6     0     0     0   0   0  630     60   150    56    73    6.1 Low
```

Fitting a Binomial-logistic model

- Suppose we want to fit a Binomial-logistic model to the first binary replicate with forest cover as a covariate
- The model is:

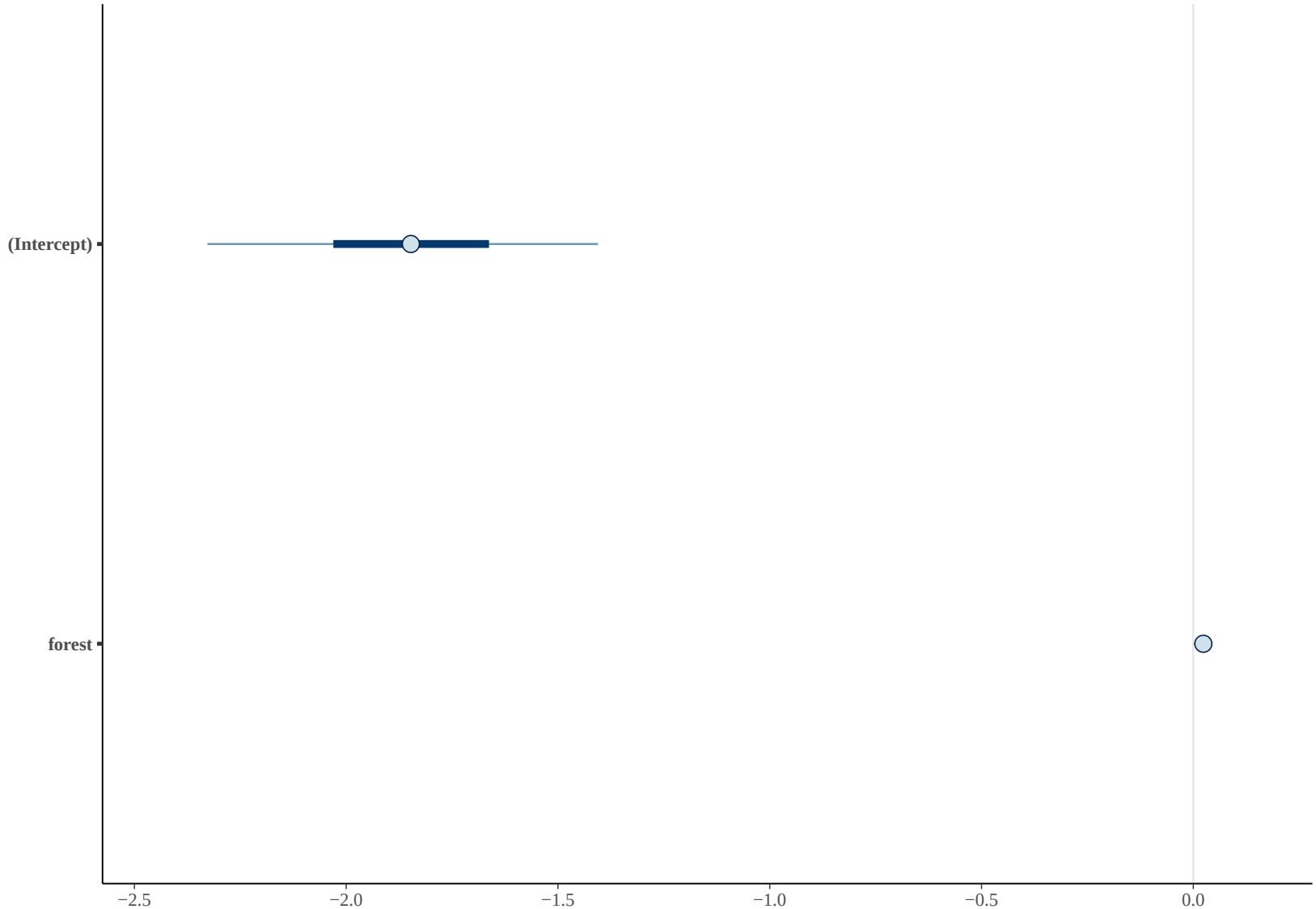
$$y_i \sim Bin(1, p_i), logit(p_i) = \alpha + \beta x_i$$

- Note that there is no residual standard deviation parameter here. This is because the variance of the binomial distribution depends only on the number of counts (here 1) and the probability, i.e. $Var(y_i) = p_i(1 - p_i)$

Fitting the model in rstanarm

```
mod_3 = stan_glm(rep.1 ~ forest,
                 data = swt,
                 family = binomial(link = 'logit'),
                 prior = normal(0, 1),
                 prior_intercept = normal(0, 5))
```

Looking at the output



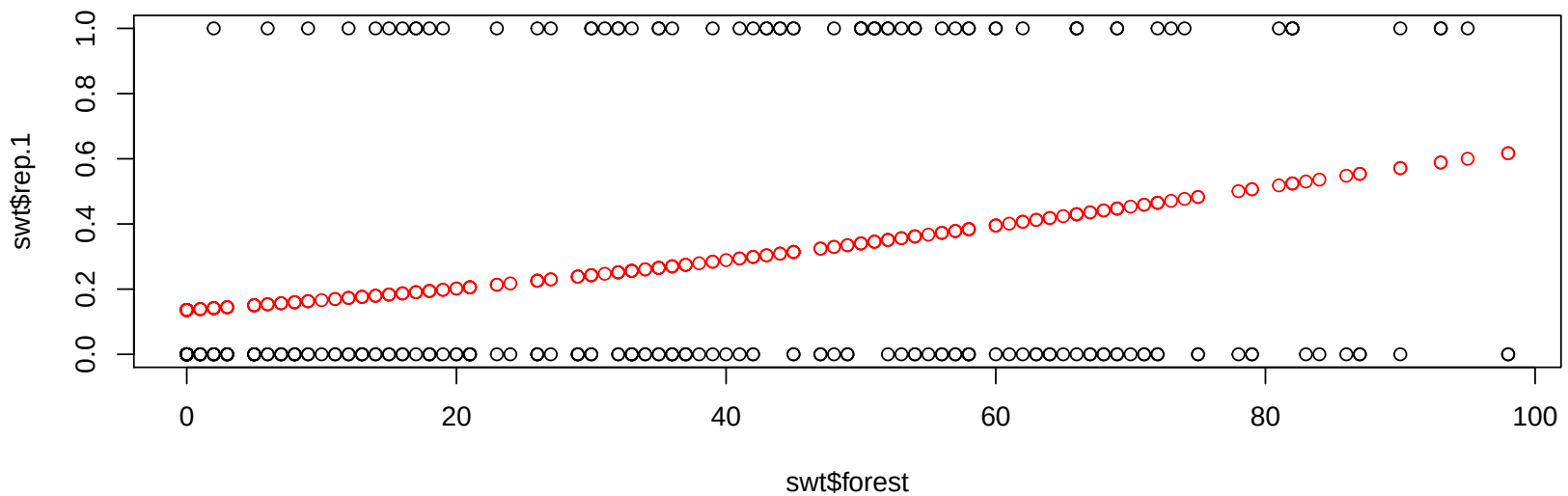
Looking at the output

```
##                  mean mcse   sd    2.5%     25%     50%     75%   97.5%
## (Intercept)     -1.85 0.01 0.28   -2.41   -2.03   -1.85   -1.66   -1.33
## forest           0.02 0.00 0.01    0.01    0.02    0.02    0.03    0.03
## mean_PPD         0.28 0.00 0.04    0.21    0.26    0.28    0.31    0.37
## log-posterior -135.87 0.02 1.01 -138.55 -136.31 -135.56 -135.14 -134.86
##               n_eff Rhat
## (Intercept)    2120    1
## forest         2395    1
## mean_PPD       3217    1
## log-posterior  1752    1
```

Plotting the fits

- It's not as easy to plot a fitted line in a Binomial regression model, but we can plot the probabilities:

```
post = as.data.frame(mod_3)
plot(swt$forest, swt$rep.1)
points(swt$forest,
       inv.logit(mean(post$`(Intercept)`) +
                   mean(post$forest)*swt$forest ),
       col = 'red')
```

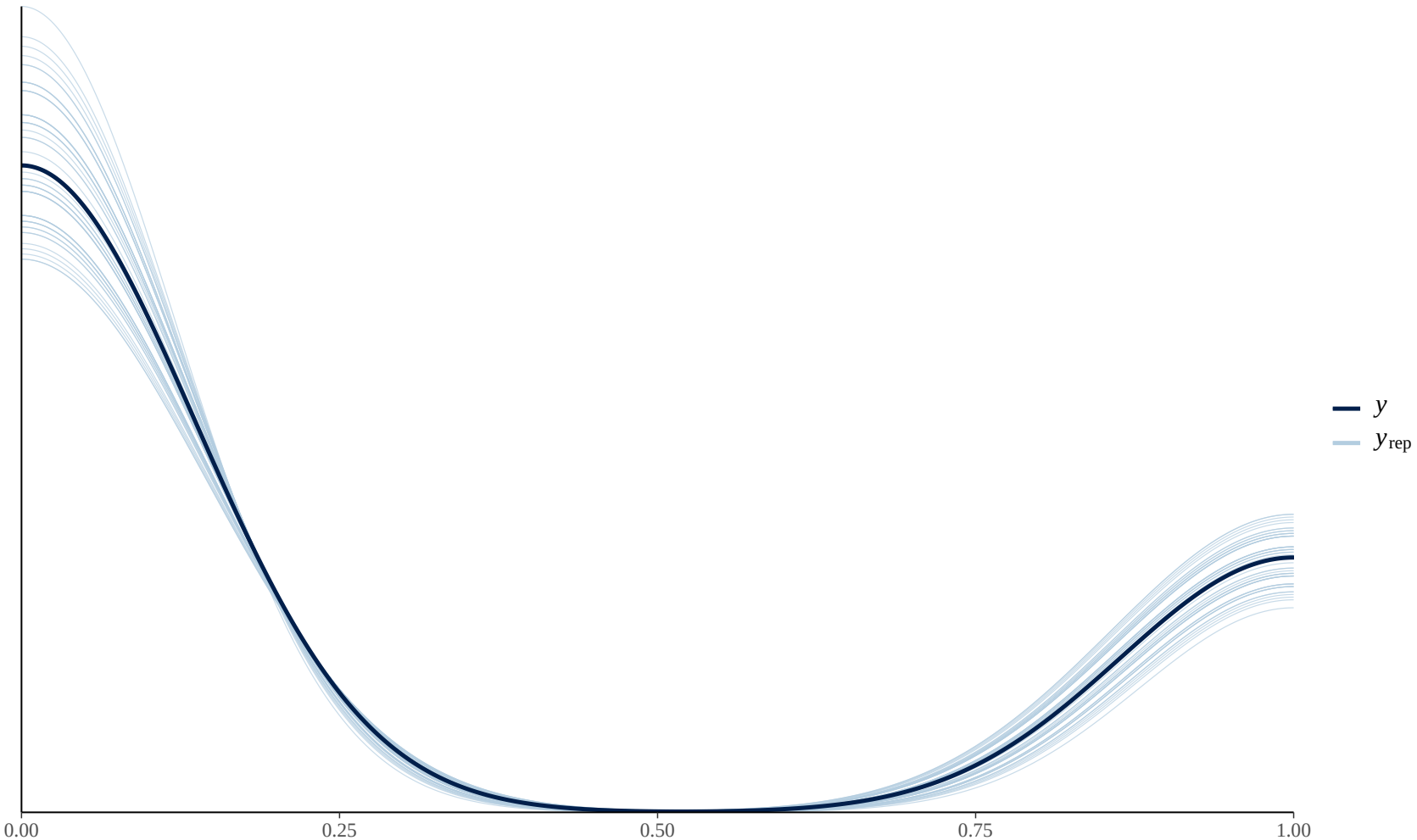


Checking model assumptions

- Just like the linear regression example, we can create posterior predictive distributions for the binary data from the binomial distribution
- However, it isn't as easy to plot as the regression situation as all the true values are 0 and 1.
- Instead people often use *classification metrics* which we do not cover in this course (but can discuss if required)

Posterior predictive check for Binomial data

```
pp_check(mod_3)
```



Binomial modelling as latent data

- The most common way of using binomial or binary data is using the logit link function
- An alternative way of fitting binomial data is via a cut-off normal distribution:

$$y_i = \begin{cases} 1 & \text{if } z_i \geq 0 \\ 0 & \text{if } z_i < 0 \end{cases}$$

with

$$z_i \sim N(\alpha + \beta x_i, 1)$$

- This is known as probit regression, with z_i a *latent parameter*

Poisson models

- Here's some `rstanarm` code for a Poisson model:

```
mod_4 = stan_glm(rep.1 ~ forest,
                 data = swt,
                 family = poisson(link = 'log'),
                 prior = normal(0, 1),
                 prior_intercept = normal(0, 5))
```

Offsets

- For Poisson data it's quite common for the counts to be dependent on the amount of effort required to collect the data
- If there is a variable that quantifies this amount of effort it should be included in the model, as it will be directly linked to the size of the counts
- These variables are often called an *offset*, and are included in the model likelihood via

```
stan_glm(formula, data,
         family = poisson(link = 'log'),
         offset = offset, ...)
```

Further examples of GLM-type data

- As we go through the course we will talk about different types of models for count data
- The Poisson is a bit restrictive, in that the variance and the mean of the counts should be the same, which is rarely satisfied by data
- We'll extend to over-dispersed and zero-inflated data
- We'll also discuss multivariate models using e.g. the multinomial distribution

Summary

- GLMs are very easy to fit in `rstanarm` once you get the hang of link functions and extracting the output
- It takes a bit of care to get the posterior distribution out of the model and to decide what you want to do with that
- There are lots of different types of GLM so pick the one that matches your data best
- Don't forget to check model assumptions via e.g. a posterior predictive check. We'll cover more checks later in the course