Class 4: 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 JAGS/Stan
- Be able to pick a link function for a given data set
- Know how to check model assumptions for a GLM

Revision: linear models

- The simplest version of a linear regression model has:
  - A *response variable* (y) which is what we are trying to predict/understand
  - An *explanatory variable* or *covariate* (x) which is what we are trying to predict the response variable from
  - Some *residual uncertainty* (ϵ) which is the leftover uncertainty that is not accounted for by the explanatory variable
- Our goal is to predict the response variable from the explanatory variable, *or* to try and discover if the explanatory variable *causes* some kind of change in the response

The linear models in maths

- We write the linear model as:

$$y_i = \alpha + \beta x_i + \epsilon_i$$

  where α is the intercept, β the slope, and $i = 1, \ldots, N$ represents each of the N observations

- Usually we make the additional assumption that $\epsilon_i \sim N(0, \sigma^2)$ where σ^2 is the residual standard deviation

- Under this assumption it is common to write $y_i | x_i, \alpha, \beta, \sigma \sim N(\alpha + \beta x_i, \sigma^2)$.

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
```

Multiple covariates

- We can extend LMs 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

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

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
- Let's fit this model in JAGS and Stan and look at the results

Fitting linear regression models in JAGS

Model code:

```
library(R2jags)
dat = read.csv('../data/earnings.csv') # Called dat
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dnorm(alpha + beta1*x1[i] + beta2*x2[i], sigma^-2)
  }
  # Priors
  alpha ~ dnorm(0, 20^-2)
  beta1 ~ dnorm(0, 1^-2)
  beta2 ~ dnorm(0, 10^-2)
  sigma ~ dunif(0, 10)
}
'
jags_run = jags(data = list(N = nrow(dat),
                            y = log(dat$earn),
                            x1 = dat$height_cm,
                            x2 = as.integer(dat$eth ==3)),
                parameters.to.save = c('alpha',
                                       'beta1',
                                       'beta2',
                                       'sigma'),
                model.file = textConnection(jags_code))
```

Output

```
print(jags_run)
```

```
## Inference for Bugs model at "4", fit using jags,
##  3 chains, each with 2000 iterations (first 1000 discarded)
##  n.sims = 3000 iterations saved
##           mu.vect sd.vect     2.5%      25%      50%      75%    97.5%
## alpha       5.848   0.488    4.913    5.517    5.853    6.180    6.802
## beta1       0.022   0.003    0.017    0.020    0.022    0.024    0.028
## beta2       0.102   0.073   -0.045    0.053    0.102    0.153    0.249
## sigma       0.907   0.020    0.868    0.893    0.907    0.921    0.944
## deviance 2797.555   2.820 2794.030 2795.446 2796.908 2799.027 2804.364
##           Rhat n.eff
## alpha    1.001  3000
## beta1    1.001  3000
## beta2    1.001  2500
## sigma    1.002  1400
## deviance 1.002  1900
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 4.0 and DIC = 2801.5
## DIC is an estimate of expected predictive error (lower deviance is better).
```

What do the results actually mean?

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

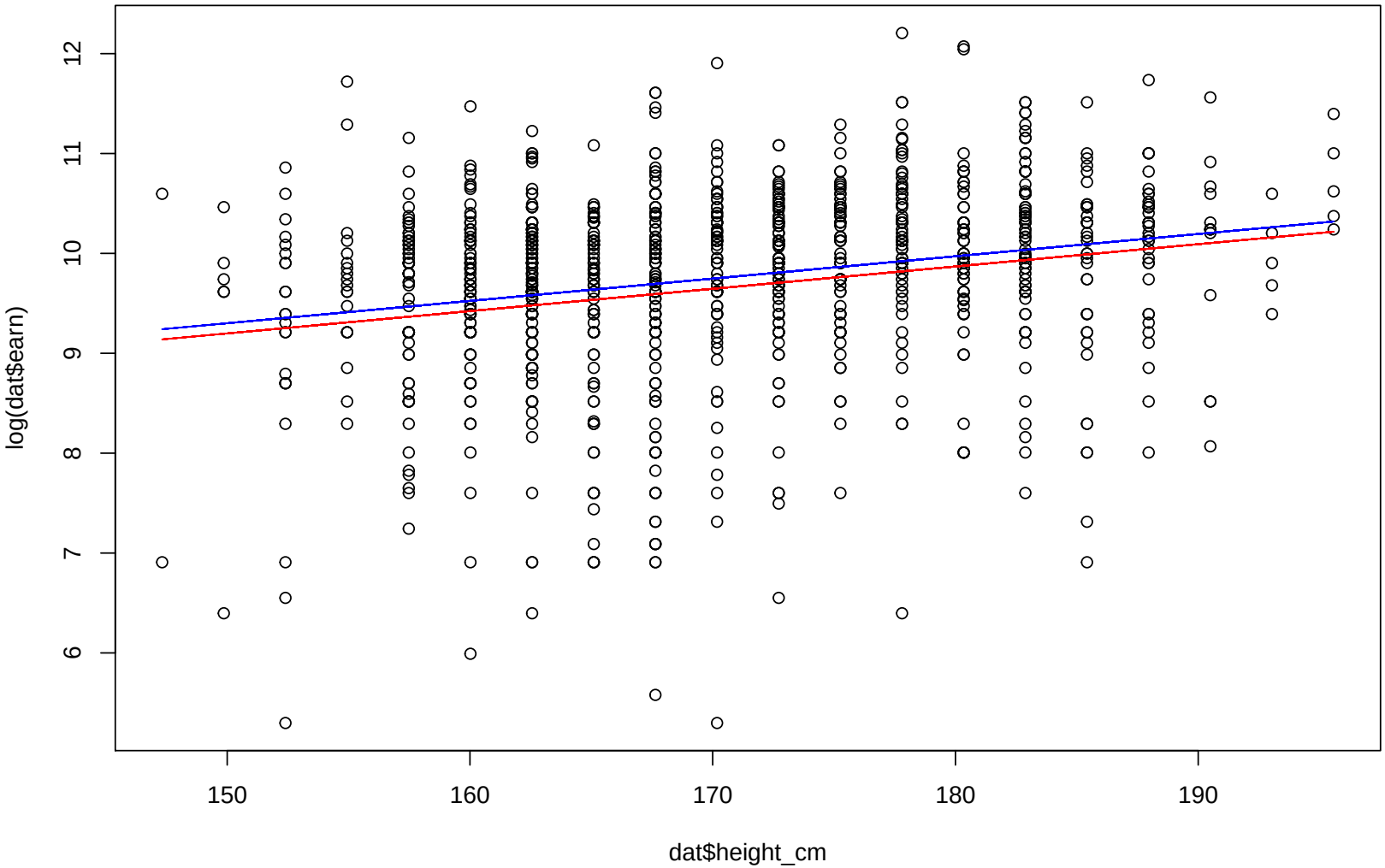
```
post = jags_run$BUGSoutput$sims.matrix
head(post)
```

```
##          alpha      beta1       beta2 deviance     sigma
## [1,] 5.458428 0.02348271 0.232862196 2811.409 0.8687041
## [2,] 6.475766 0.01930220 0.004728883 2797.960 0.9141370
## [3,] 6.460258 0.01835601 0.223561773 2799.307 0.9106671
## [4,] 5.959597 0.02120780 0.194524481 2795.484 0.8949676
## [5,] 5.317468 0.02550939 0.097068875 2796.385 0.8816301
## [6,] 5.949902 0.02147783 0.176270295 2794.928 0.9065328
```

Plots of output

```
alpha_mean = mean(post[,'alpha'])
beta1_mean = mean(post[,'beta1'])
beta2_mean = mean(post[,'beta2'])
plot(dat$height_cm, log(dat$earn))
lines(dat$height_cm, alpha_mean +
        beta1_mean * dat$height_cm)
lines(dat$height_cm, alpha_mean +
        beta1_mean * dat$height_cm, col = 'red')
lines(dat$height_cm, alpha_mean +
        beta1_mean * dat$height_cm + beta2_mean,
      col = 'blue')
```

Plots



Fitting in Stan

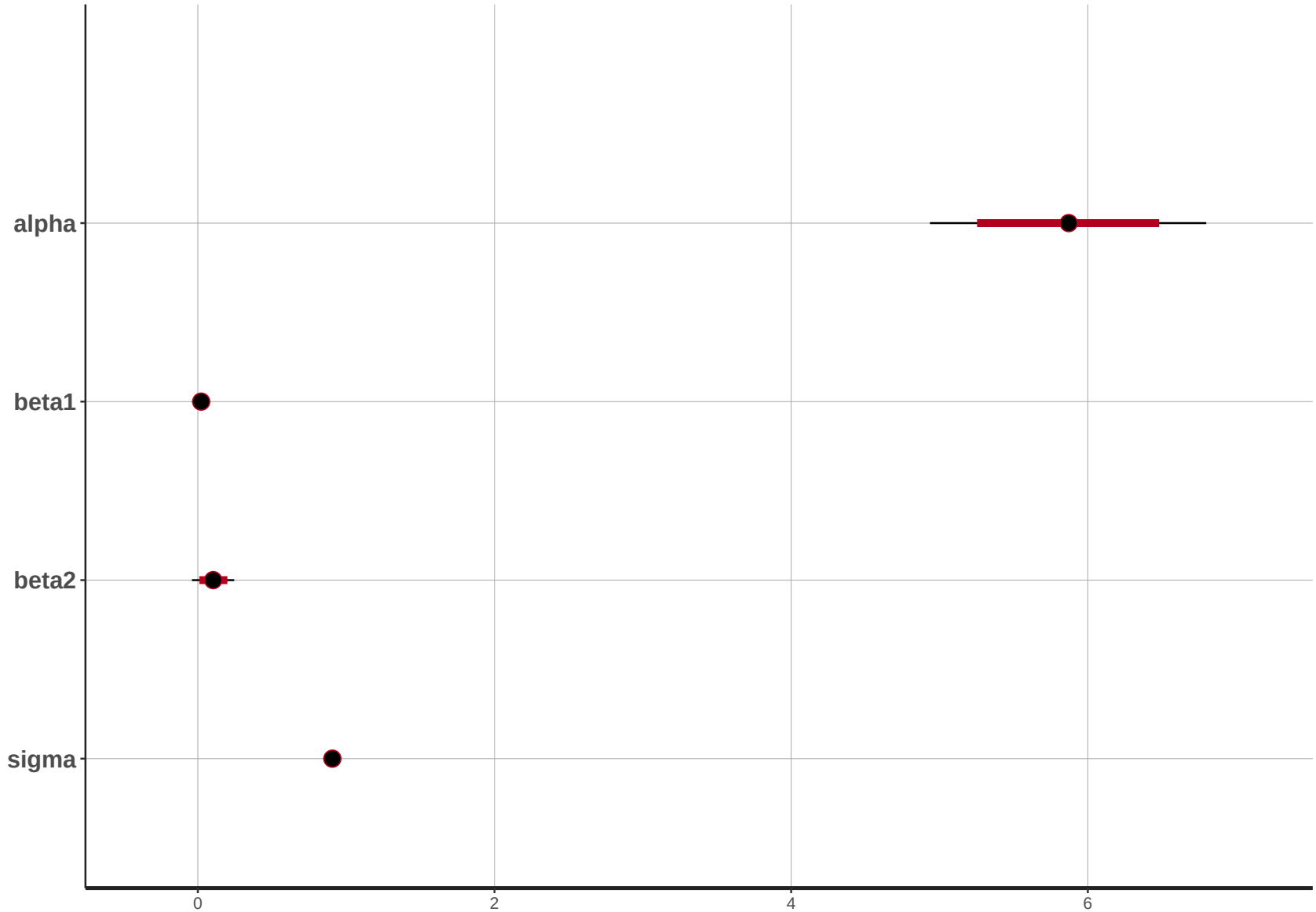
```
stan_code = '
data {
  int<lower=0> N;
  vector[N] y;
  vector[N] x1;
  vector[N] x2;
}
parameters {
  real alpha;
  real beta1;
  real beta2;
  real<lower=0> sigma;
}
model {
  y ~ normal(alpha + x1 * beta1  + x2 * beta2, sigma);
}
'
```

Running the Stan version

```
library(rstan)
stan_run = stan(data = list(N = nrow(dat),
                            y = log(dat$earn),
                            x1 = dat$height_cm,
                            x2 = as.integer(dat$eth==3)),
                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
- Stan seems to struggle with regression models where the data are not standardised
- 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)

From LM to GLM

- We use a *generalised linear model* (GLM) when the normal distribution is not longer appropriate for the data
- This probability distribution should match the type of data (e.g. count data, binary, etc) and will have its own parameters
- We often have to transform the parameters if we still want to use a linear regression type relationship with a covariate. The transformation is called a *link function*
- In a Bayesian generalised linear model we just compute a likelihood and combine it with a prior distribution just like every other model we fit

The data generating process for a logistic regression

- What if the response variable was binary? Clearly the linear regression simulation 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 `NA`s 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 0 1 0 1 1 1 1 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 $(-\infty, \infty)$
- This type of model is known as *logistic-Binomial* regression (or just *logistic regression*) and the logit is known as the *link function*
- It's also possible to generate data with maximum value bigger than 1 by changing the `size` parameter

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 1 0 2 1 1 3 2 7 4
```

- 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

Example: 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 dur.2 dur.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:

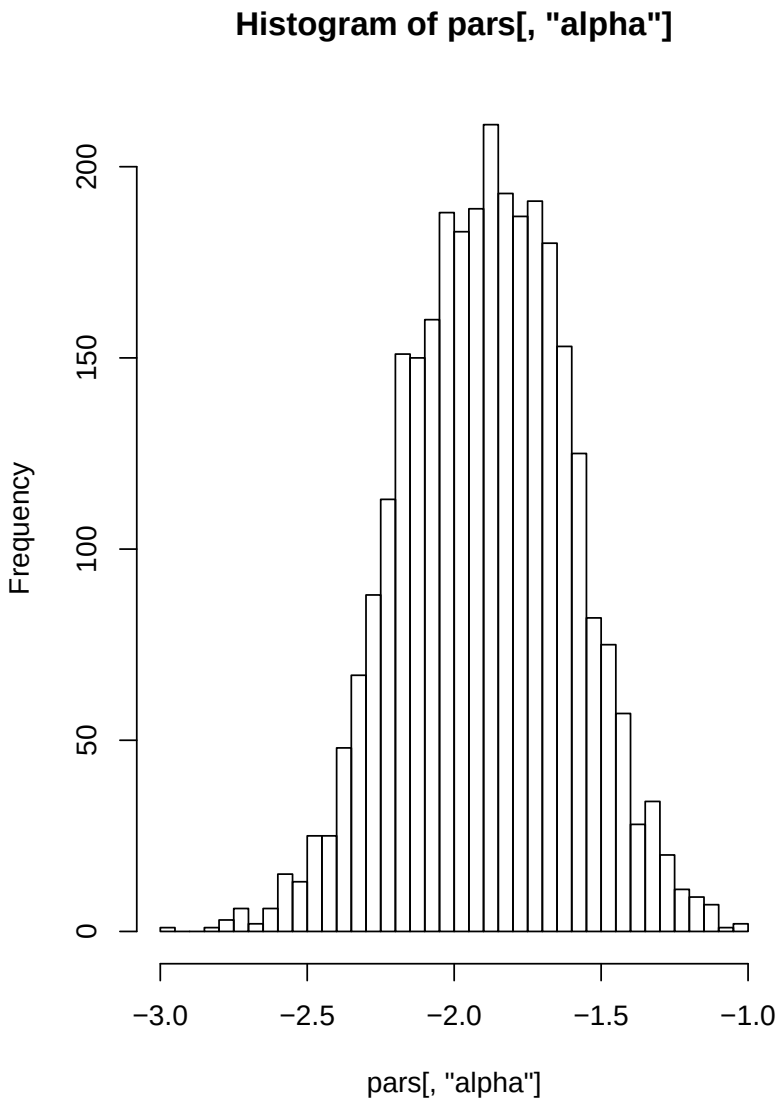
$$y_i \sim Bin(1, p_i), logit(p_i) = \alpha + \beta x_i$$

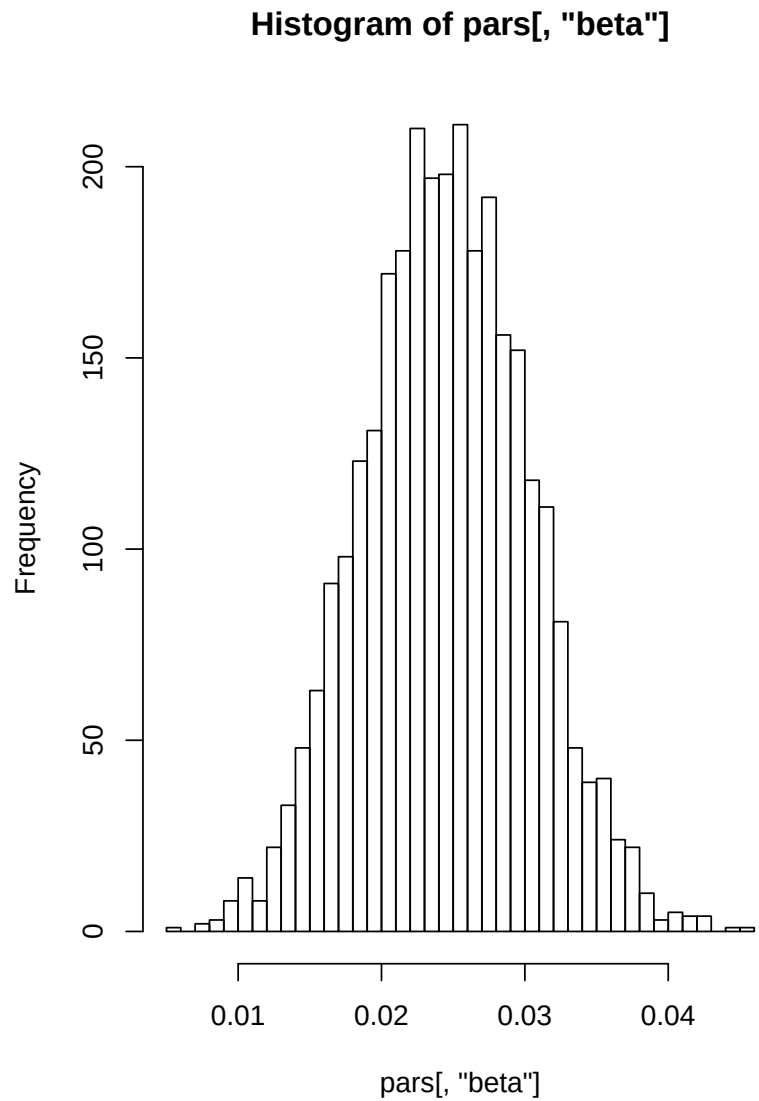
- Note that there is no residual standard deviation parameter here. This is because the variance of the binomial distribution depends only on the number of counts (here 1) and the probability, i.e. $Var(y_i) = p_i(1 - p_i)$

Fitting the model in JAGS

```
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dbin(p[i], 1)
    logit(p[i]) <- alpha + beta*x[i]
  }
  # Priors
  alpha ~ dnorm(0, 20^-2)
  beta ~ dnorm(0, 20^-2)
}
'
jags_run = jags(data = list(N = nrow(swt),
                            y = swt$rep.1,
                            x = swt$forest),
                parameters.to.save = c('alpha',
                                       'beta'),
                model.file = textConnection(jags_code))
```

Looking at the output

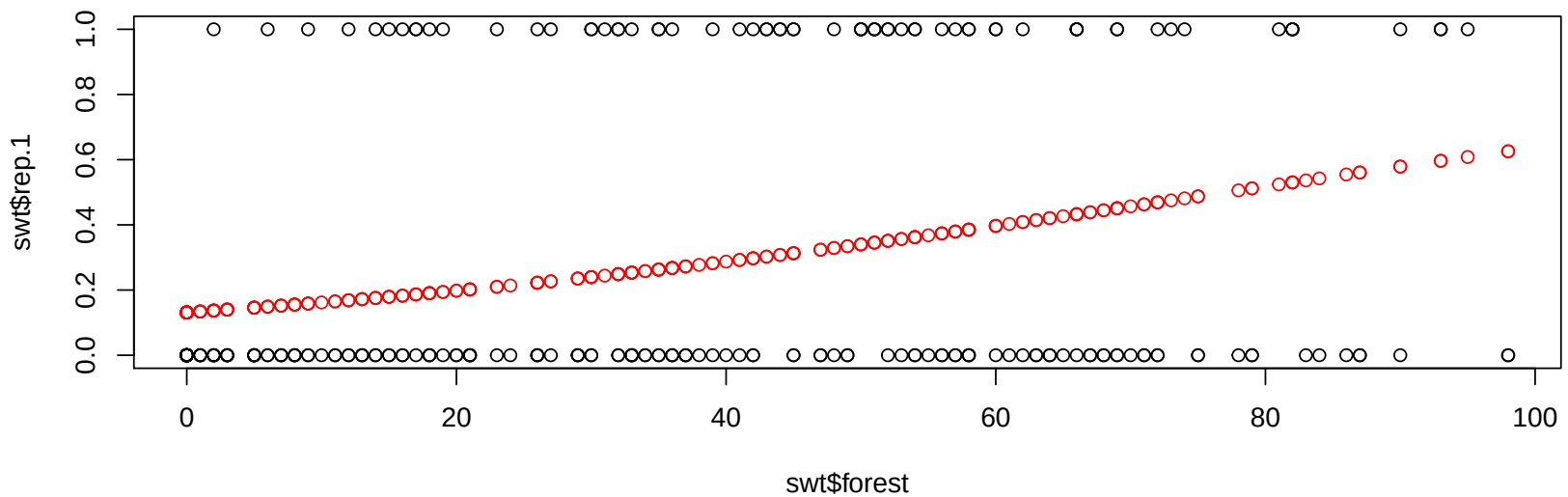




Plotting the fits

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

```
plot(swt$forest, swt$rep.1)
points(swt$forest,
       inv.logit(mean(pars[,'alpha']) +
                   mean(pars[,'beta'])*swt$forest ),
       col = 'red')
```



Poisson models

- Here's some JAGS code for a Poisson model:

```
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dpois(lambda[i])
    log(lambda[i]) <- alpha + beta*x[i]
  }
  # Priors
  alpha ~ dnorm(0, 20^-2)
  beta ~ dnorm(0, 20^-2)
}
'
```

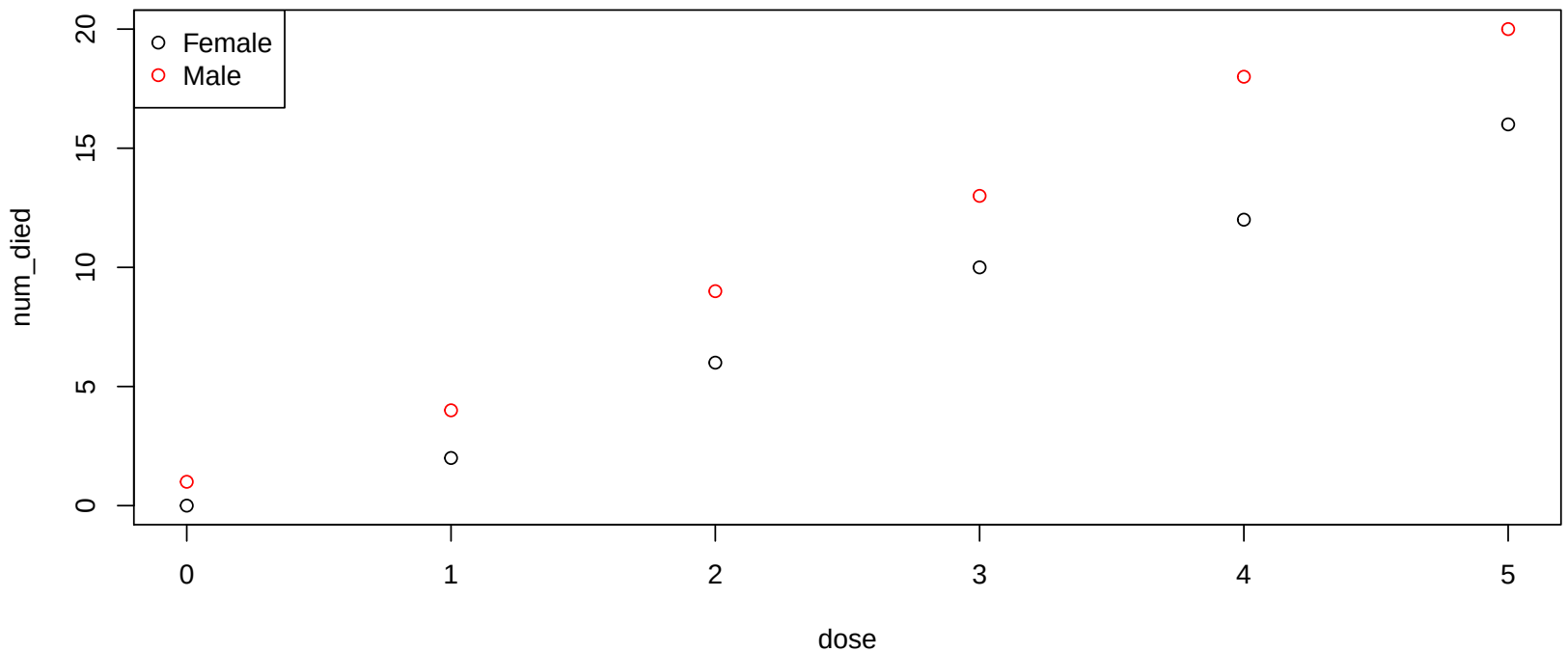
Offsets

- 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

```
y[i] ~ dpois(offset * lambda[i])
```

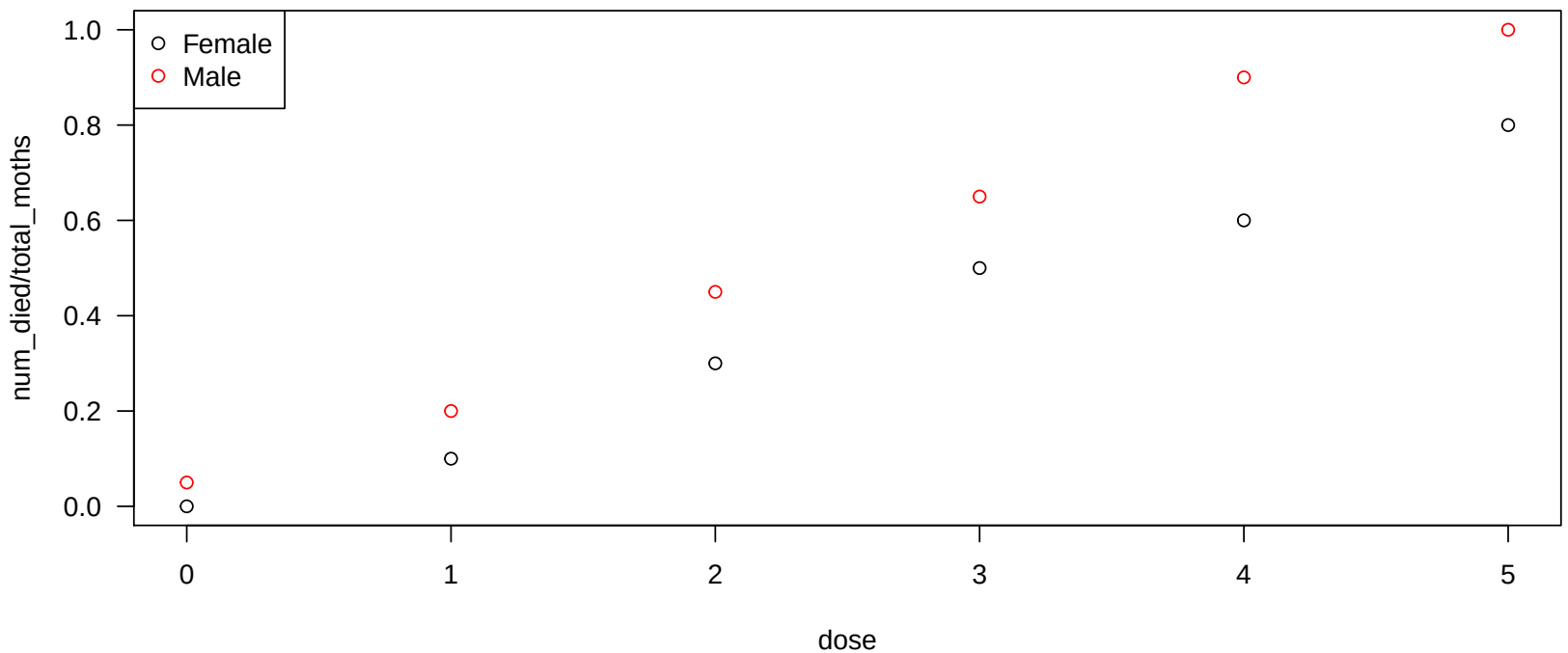
Example of a full binomial regression model

```
plot(dose, num_died, col = sex_male + 1)
legend('topleft', legend = c('Female', 'Male'),
       col = 1:2, pch = 1)
```



Looking at the data differently - proportion died

```
plot(dose, num_died/total_moths, col = sex_male + 1, las =
legend('topleft', legend = c('Female', 'Male'),
       col = 1:2, pch = 1)
```

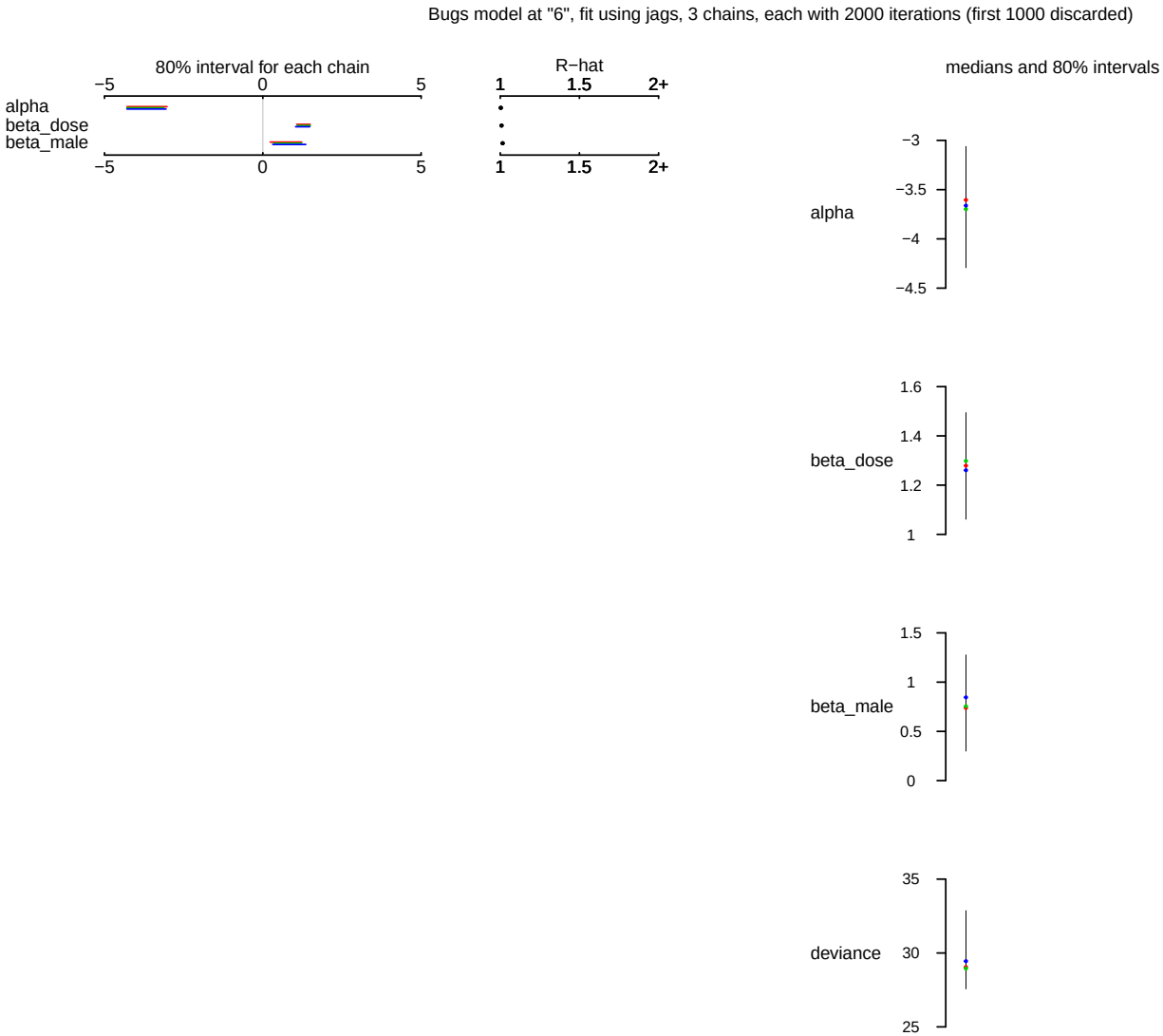


Fit with JAGS

```
model_code = '
model
{
  # Likelihood
  for (i in 1:N) {
    y[i] ~ dbin(p_died[i], K)
    logit(p_died[i]) <- alpha + beta_male * x_male[i] + beta_dose * x_dose[i]
  }
  # Priors
  alpha ~ dnorm(0, 10^-2)
  beta_male ~ dnorm(0, 10^-2)
  beta_dose ~ dnorm(0, 10^-2)
}
'
model_data = list(N = N, y = num_died, x_male = sex_male, x_dose = dose, K = total_moths)
model_parameters =  c("alpha", "beta_male", "beta_dose")
model_run = jags(data = model_data,
                 parameters.to.save = model_parameters,
                 model.file = textConnection(model_code))
```

Model output

```
plot(model_run)
```



Further examples of GLM-type data

- Later in 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

What are JAGS and Stan doing in the background?

- JAGS and Stan run a stochastic algorithm called Markov chain 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 Guessing *new parameter values* which may or may not be similar to the previous values
     - 2.2 Seeing whether the new values match the data and the prior by calculating *new scores*
     - 2.3 If the scores for the new parameters are higher, keep them. If they are lower, keep them with some probability depending on how close the scores are, otherwise discard them and keep the old values
- What you end up with is a set of parameter values for however many iterations you chose.

How many iterations?

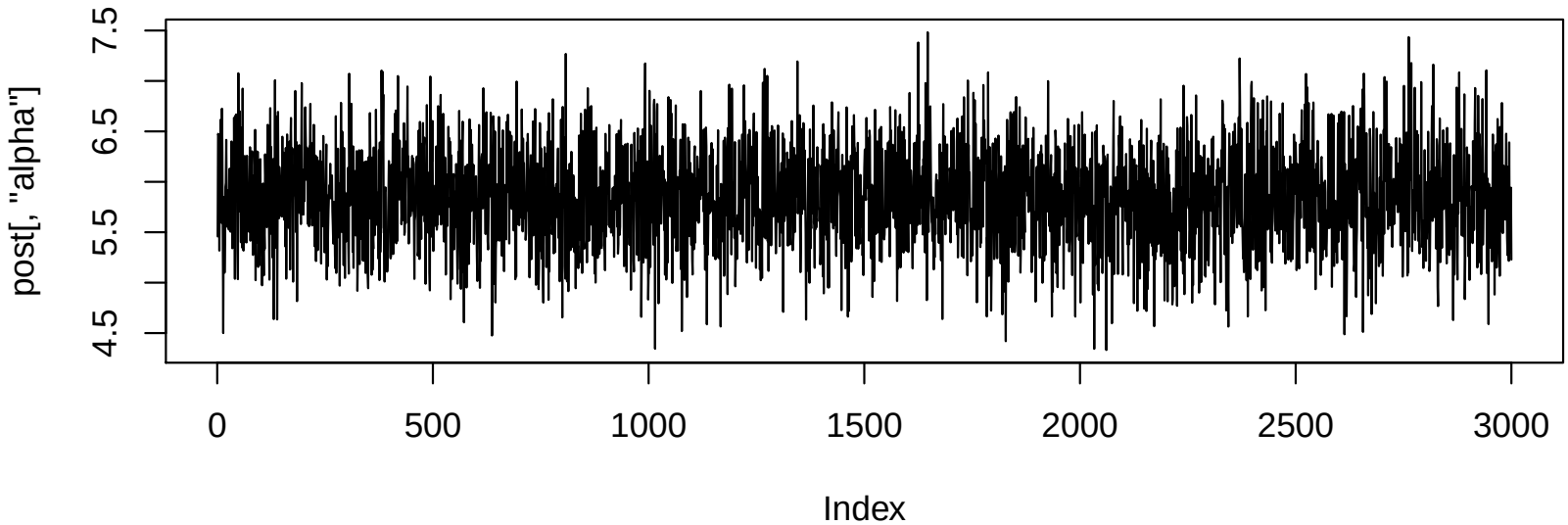
- Ideally you want a set of posterior parameter samples that are independent across iterations and is of sufficient size that you can get decent estimates of uncertainty
- There are three key parts of the algorithm that affect how good the posterior samples are:
    1. The starting values you chose. 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

JAGS and Stan 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 `traceplot`, or for just one with e.g.

```
plot(post[,'alpha'], type = 'l')
```



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, JAGS and Stan automatically run *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.
- Both JAGS and Stan report 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

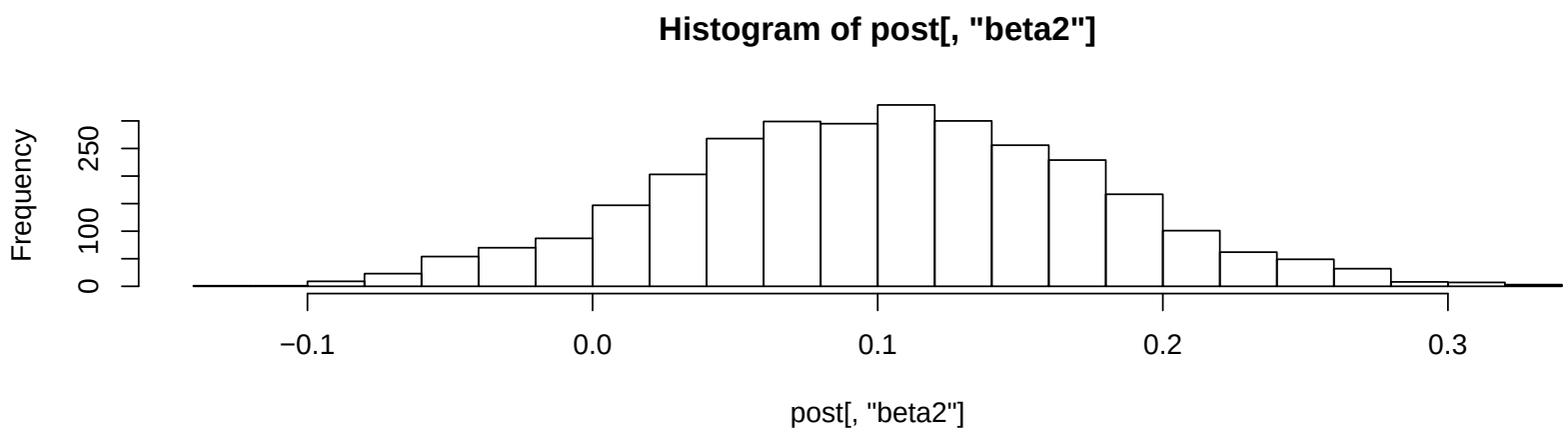
- We could create *credible intervals* (Bayesian confidence intervals):

```
apply(post, 2, quantile, probs = c(0.025, 0.975))
```

```
##           alpha      beta1       beta2 deviance     sigma
## 2.5%  4.912925 0.01680622 -0.04465226 2794.030 0.8683218
## 97.5% 6.801527 0.02788807  0.24872565 2804.364 0.9442827
```

- Or histograms

```
hist(post[,'beta2'], breaks = 30)
```



Checking model fit

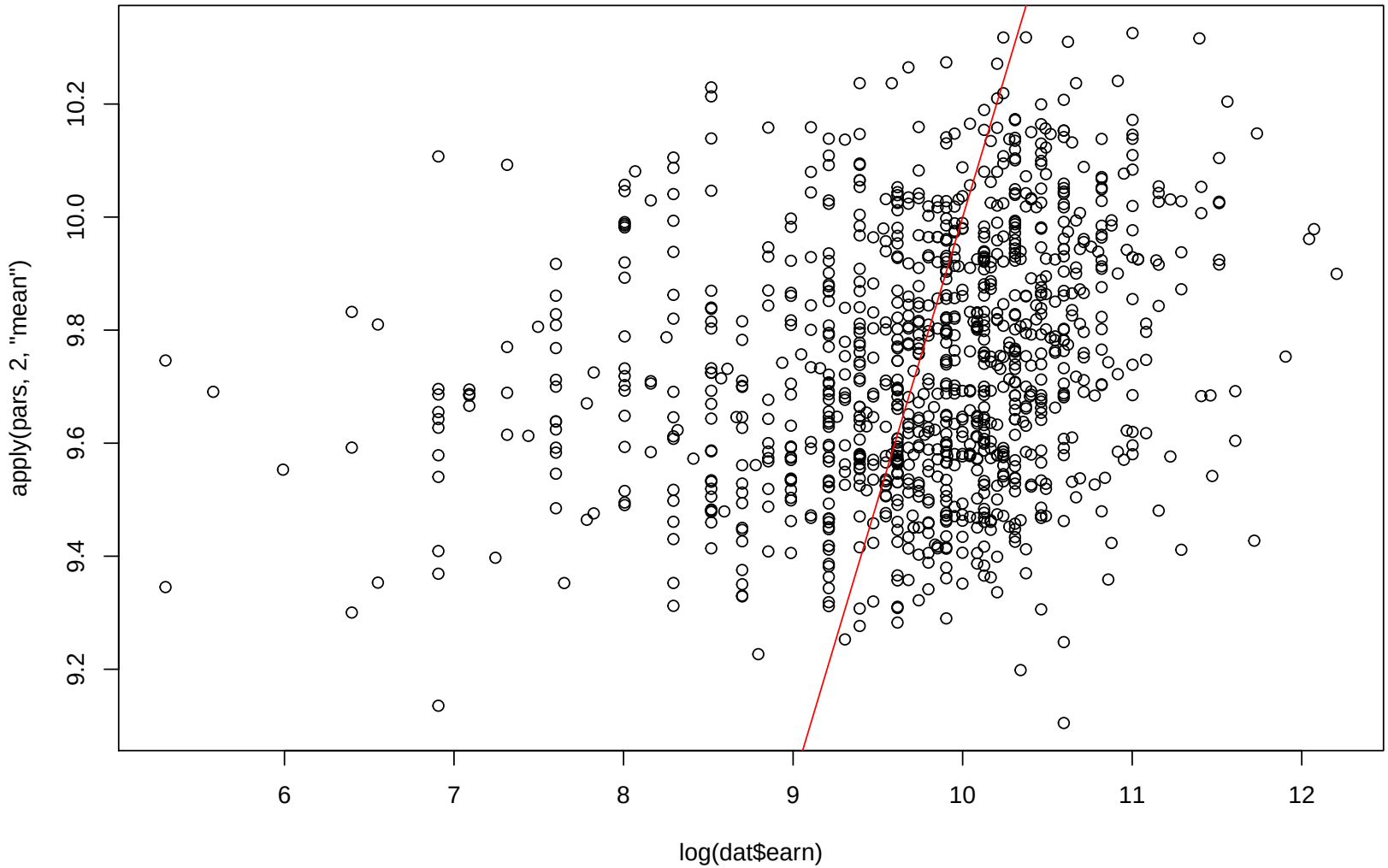
- How do we know if this model fits the data well or not?
- One way is to simulate from the posterior distribution of the parameters, and subsequently simulate from the likelihood to see if the these data match the real data we observed
- This is known as a *posterior predictive check*

Simple posterior predictive distributions

- The easier way is to put an extra line in the JAGS code:

```
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dnorm(alpha + beta1*x1[i] + beta2*x2[i],
                 sigma^-2)
    y_sim[i] ~ dnorm(alpha + beta1*x1[i] + beta2*x2[i],
                     sigma^-2)
  }
  # Priors
  alpha ~ dnorm(0, 20^-2)
  beta1 ~ dnorm(0, 1^-2)
  beta2 ~ dnorm(0, 10^-2)
  sigma ~ dunif(0, 10)
}
'
```

Posterior predictive outputs



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)

Summary

- GLMs are very easy to fit in JAGS/Stan once you get the hang of link functions
- 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