Practical 3 - Fitting hierarchical GLMs

Andrew Parnell

Introduction

In practical 3 we are going to:

- Fit some hierarchical regression models
- Create some plots of the posterior predicted values with uncertainty
- Fit some hierarchical GLMs
- Create some prior and posterior predictive pseudo-data

We will mostly be focussing on JAGS for this practical to avoid making it too long. However, I would strongly encourage you to try translating some of this code into Stan as it will be very good practice.

Fitting hierarchical regression models

Let's start with the hierarchical regression model we met in class for the `earnings` data:

```
dat = read.csv('https://raw.githubusercontent.com/andrewcparnell/bhm_course/master/data/earnings.csv')
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dnorm(alpha[eth[i]] +
                   beta[eth[i]]*(x[i] - mean(x)),
                     sigma^-2)
  }
  # Priors
  for(j in 1:N_eth) {
    alpha[j] ~ dnorm(mu_alpha, sigma_alpha^-2)
    beta[j] ~ dnorm(mu_beta, sigma_beta^-2)
  }
  mu_alpha ~ dnorm(11, 2^-2)
  mu_beta ~ dnorm(0, 0.1^-2)
  sigma ~ dunif(0, 5)
  sigma_alpha ~ dunif(0, 2)
  sigma_beta ~ dunif(0, 2)
}
'
```

Some notes/reminders about this code:

- We have four different values that `eth[i]` can take, either 1, 2, 3, or 4. This means that there are four different intercept values `alpha` and four different slope values `beta`
- The explanatory variable `x` here is mean centered. You can either do this in the code as we have done here, or do it by providing JAGS with already mean-centered data
- The prior distributions on `alpha` and `beta` are the clever bits here. They allow us to borrow strength across the different groups.
- The hyper-priors here starting with the text `mu_alpha ~` etc were developed from the prior predictive distribution

We can run it with:

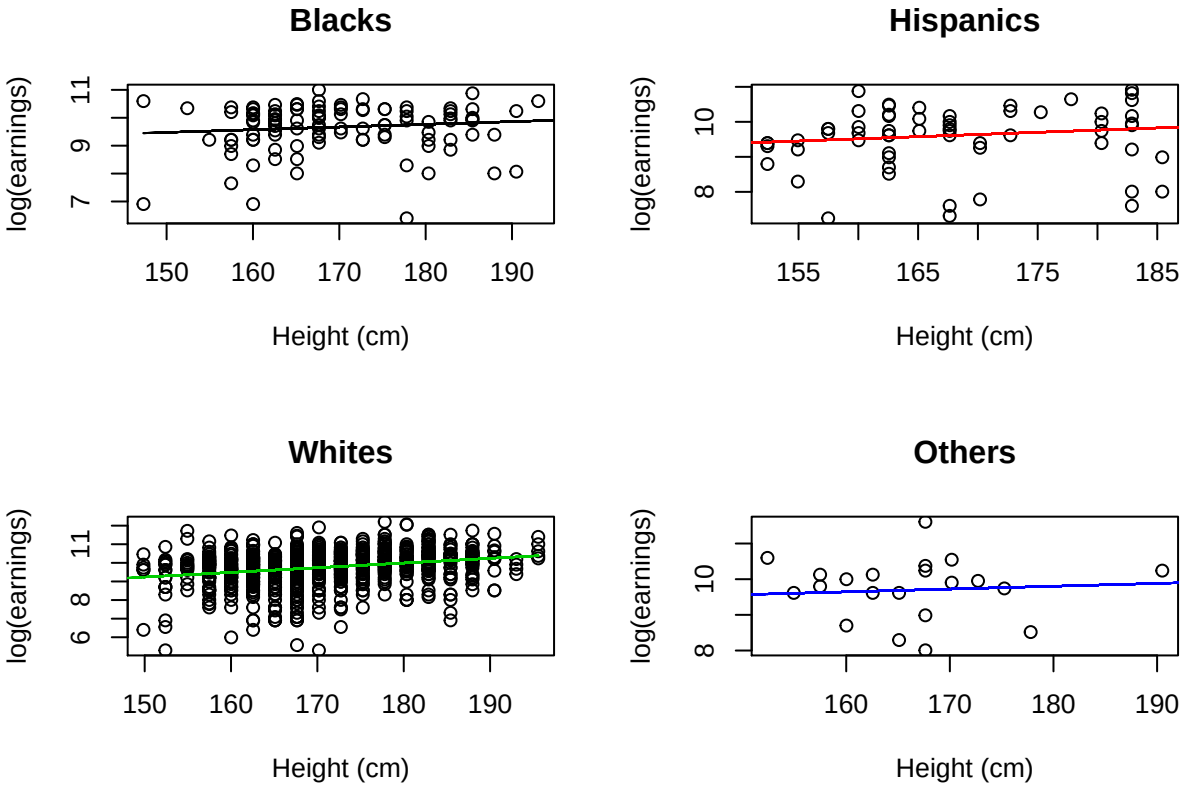
```
jags_run = jags(data = list(N = nrow(dat),
                            y = log(dat$earn),
                            eth = dat$eth,
                            N_eth = length(unique(dat$eth)),
                            x = dat$height_cm),
                parameters.to.save = c('alpha',
                                       'beta',
                                       'sigma',
                                       'mu_alpha',
                                       'mu_beta',
                                       'sigma_alpha',
                                       'sigma_beta'),
                model.file = textConnection(jags_code))
```

Exercise 1

1. Run the above model and use the `print` and `plot` commands on the output. Try to interpret the output as we did in class
2. Re-run the model but this time without mean centering the data. How does your interpretation change?
3. Go back to the mean centered model but start changing the prior distributions on `sigma_alpha` and `sigma_beta`. What happens if you force these values to be very small (e.g. `sigma_alpha ~ dunif(0, 0.01)`)?

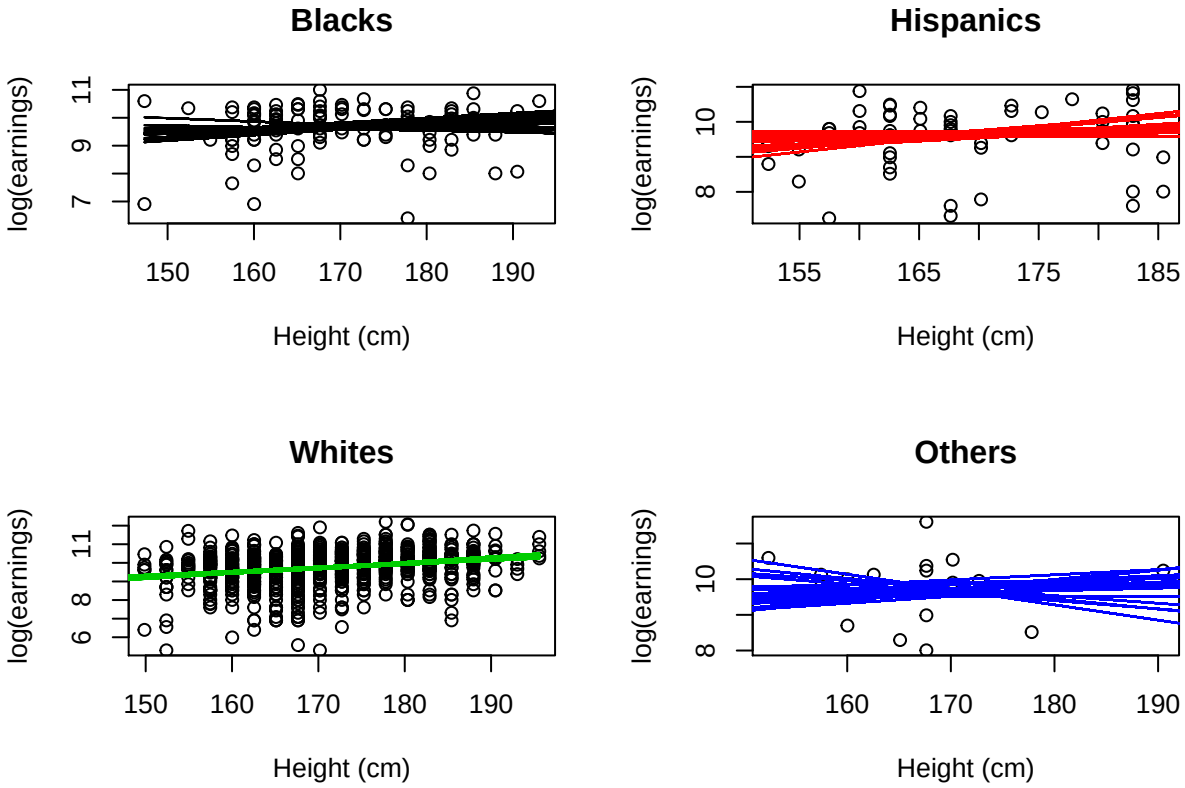
Let's create a much richer plot to that created in class. The original plot we had was created with:

```
pars = jags_run$BUGSoutput$mean
par(mfrow=c(2,2))
eth_names = c('Blacks','Hispanics','Whites','Others')
for(i in 1:4) {
  curr_dat = subset(dat, dat$eth == i)
  plot(curr_dat$height_cm,
       log(curr_dat$earn),
       main = eth_names[i],
       ylab = 'log(earnings)',
       xlab = 'Height (cm)')
  lines(dat$height_cm,
        pars$alpha[i] + pars$beta[i]*(dat$height_cm - mean(dat$height_cm)),
        col = i)
}
```



If you look at the above, all we're creating is the line based on the mean posterior values of `alpha` and `beta`. What would be nice to include on this is the uncertainty in the lines for each plot. Here is some code to produce the same plot but with the first 20 posterior lines to give an idea of the uncertainty:

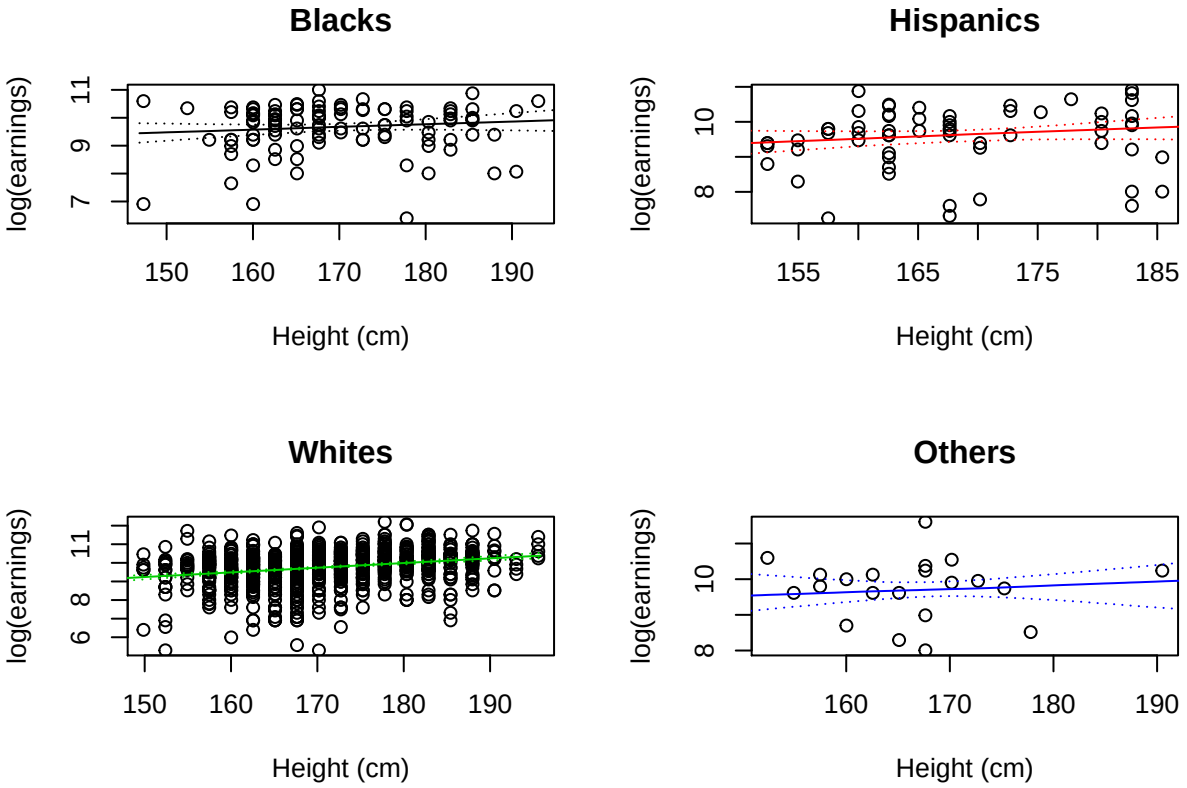
```
pars = jags_run$BUGSoutput$sims.list
par(mfrow=c(2,2))
for(i in 1:4) {
  curr_dat = subset(dat, dat$eth == i)
  plot(curr_dat$height_cm, log(curr_dat$earn),
       main = eth_names[i], ylab = 'log(earnings)',
       xlab = 'Height (cm)')
  for(j in 1:20) {
    lines(dat$height_cm,
          pars$alpha[j,i] +
            pars$beta[j,i]*(dat$height_cm - mean (dat$height_cm)),
          col = i)
  }
}
```



You can start to see that the 'Whites' group is much less uncertain due to the quantity of data.

You can go even further than this by creating thousands of predicted lines and then summarising them to produce, e.g. a 90% posterior confidence interval:

```
n_sims = jags_run$BUGSoutput$n.sims
x_grid = pretty(dat$height_cm - mean(dat$height_cm), n = 100)
par(mfrow=c(2,2))
for(i in 1:4) {
  curr_dat = subset(dat, dat$eth == i)
  plot(curr_dat$height_cm, log(curr_dat$earn), main = eth_names[i], ylab = 'log(earnings)', xlab = 'Hei
  all_lines = matrix(NA, ncol = length(x_grid), nrow = n_sims)
  for(j in 1:n_sims) {
    all_lines[j,] = pars$alpha[j,i] + pars$beta[j,i]*x_grid
  }
  all_lines_summary = apply(all_lines,2,'quantile',probs = c(0.05, 0.5, 0.95))
  lines(x_grid + mean(dat$height_cm), all_lines_summary[2,], col = i)
  lines(x_grid + mean(dat$height_cm), all_lines_summary[1,], col = i, lty = 'dotted')
  lines(x_grid + mean(dat$height_cm), all_lines_summary[3,], col = i, lty = 'dotted')
}
```



This code is pretty complicated so have a good careful read through and please ask questions if some lines don't make sense. Notice that I am creating an `x_grid` here to produce a neat set of 100 `x` values to create the lines but am then adding back on the mean of the height to match the plots.

Exercise 2a

1. Experiment with the code which produces the 20 posterior samples. Try changing the number of simulations. Why do you think we wouldn't plot 3000 posterior lines?
2. Try changing the colours of the posterior sample lines in the above. One possible solution to the above problem is to make them slightly transparent. You can do this by setting the alpha value of the `rgb` function, e.g. `col = rgb(1,0,0,alpha=0.02)`. See if you can create a really nice plot
3. For the model which creates confidence intervals. See if you can add in the 25th and 75th percentile lines in addition to the above.

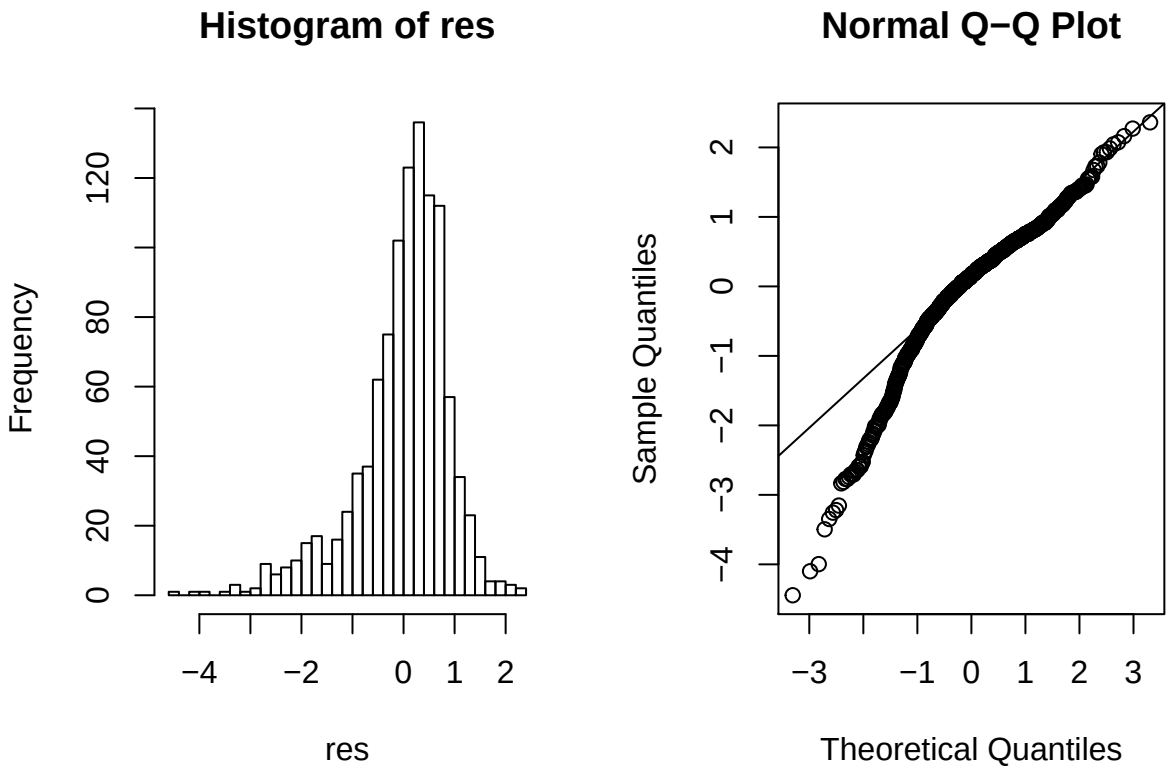
Checking residual assumptions

We should also check the residuals of these models. Again we can do this directly from the JAGS output. Let's start by creating the residuals for the mean values of the parameters:

```
par_means = jags_run$BUGSoutput$mean
fits = par_means$alpha[dat$eth] + par_means$beta[dat$eth] * (dat$height_cm - mean(dat$height_cm))
res = log(dat$earn) - fits
```

Now let's create a histogram and a qq-plot of the mean residuals:

```r
par(mfrow=c(1,2))
hist(res, breaks = 30)
qqnorm(res)
qqline(res)
```



These look perhaps a little bit skewed at the low end.

Exercise 2b

1. Another common plot is that of the residuals vs the fitted values. Creat a nice plot of this too. Interpret the plot.
2. Go back and re-fit the model to the raw earnings data (rather than log earnings - make sure to check your priors). Do they residual plots improve?

Fitting Hierarchical GLMs

We earlier met lots of different hierarchical GLM models. The code for all of these models is in the Class 6 `.Rmd` file. Rather than reproduce all the code here, I suggest working through these exercises:

Exercise 3

1. Go through each model in turn in the Class 6 notes. Make sure it will run and check the convergence. Run the model for longer if you don't think it has converged.

2. For each model try to understand the JAGS code. If there are lines that are confusing try looking them up in the manual, or ask for help.
3. Try to create posterior summary plots like those we created above for the earnings data. See if you can beat the plots that I created in the slides. Make them neater, more colourful, more expository, etc
4. Change the prior distributions. Look at the effect on the models

If you're feeling really brave try translating them into Stan. Some will be quite easy (with help from the manual). Some will be really tough.

Creating prior and posterior predictive distributions

At the beginning of Class 5 we created a prior predictive distribution for the earnings data. To do this we went through the following steps:

1. We hypothesised a model and wrote down the maths
2. We simulated a possible value(s) from the prior distribution
3. For each simulated set of prior values we simulated some possible data values
4. We looked at the resulting data set and decided whether it was reasonable or not and changed our prior distribution accordingly
5. We then ran the model properly using these prior distributions

All of this is independent of using JAGS or Stan. We run all these commands in R.

Let's now create a prior predictive distribution for the over-dispersed Poisson example of Class 6. The JAGS code for a starting model might be:

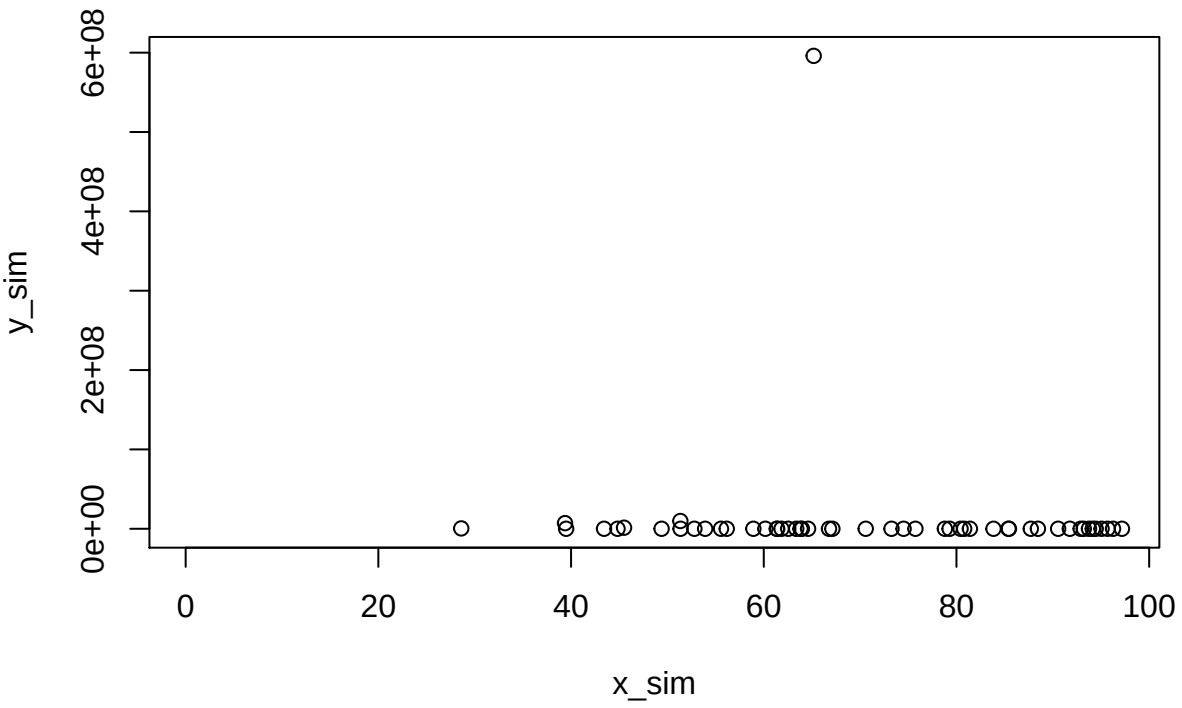
```
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dpois(exp(log_lambda[i]))
    log_lambda[i] ~ dnorm(alpha + beta * (x[i] - mean(x)), sigma^-2)
  }
  alpha ~ dnorm(0, 10^-2)
  beta ~ dnorm(0, 10^-2)
  sigma ~ dt(0,10,1)T(0,)
}
'
```

If you recall, the response variable `y` here represents the number of birds found, whilst `x` is the percentage of forest cover (a value from 0 to 100). If you were the person running the experiment you might know that it's very unlikely to get much more than 5 or 6 birds in a particular sampling region, and that you're not sure how forest cover affects the number of birds. Let's start by generating a single sample data set of say, size 100, from this model:

```
N = 100
x_sim = runif(N, 0, 100)
sigma_sim = abs(rt(1, df = 1))*10 # *10 needed as t-dist in R doesn't have sd
beta_sim = rnorm(1, 0, 10)
alpha_sim = rnorm(1, 0, 10)
log_lambda_sim = rnorm(100, alpha_sim + beta_sim * (x_sim - mean(x_sim)),
                       sigma_sim)
y_sim = rpois(N, exp(log_lambda_sim))
```

```
## Warning in rpois(N, exp(log_lambda_sim)): NAs produced
```

```
plot(x_sim, y_sim)
```



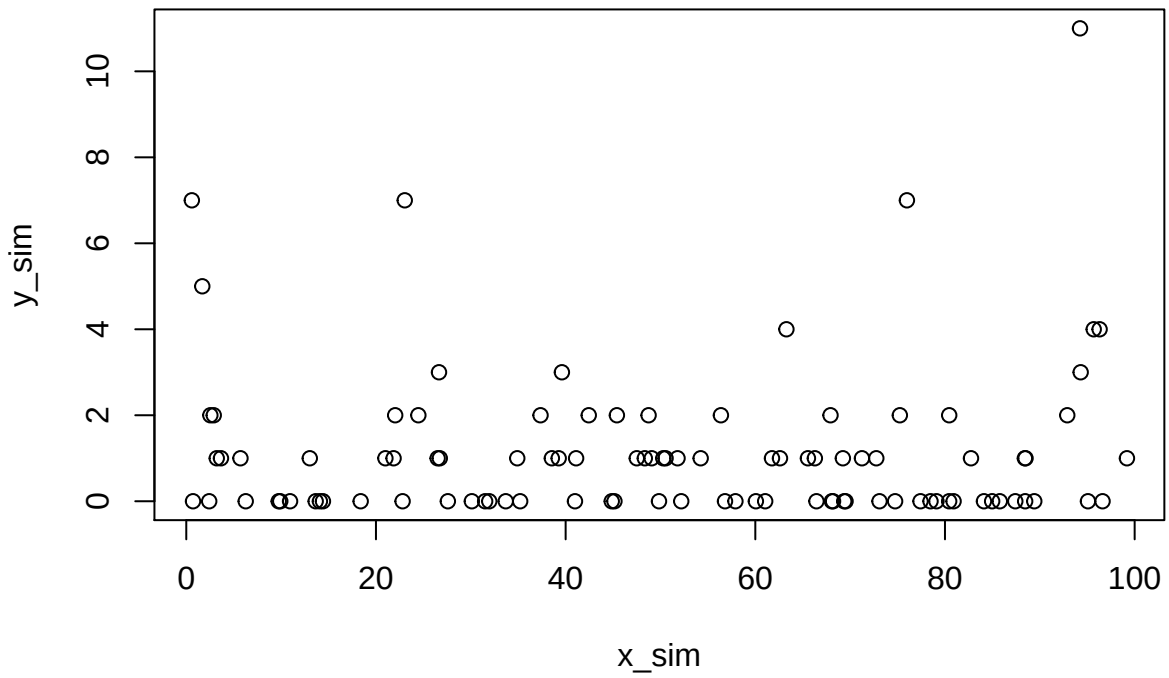
I got lots of warnings when I ran this - some of the `y_sim` values are infinite! This must be a crazy set of prior values. Let's improve:

- For a start, if the mean number of birds per region is about 2 or 3, then this suggests that `alpha` should be about $log(2.5) \approx 1$, so perhaps $N(1, 1)$ is a better prior.
- Similarly, if we expect the number of birds to be not much more than, say, 20 in a region, then a value for `beta_sim` of about +/- 0.02 seems reasonable (because $\exp(1 + 0.02 * 100) \approx 20$). Perhaps $N(0, 0.02)$ would be a good prior.
- The value of `sigma_sim` is perhaps the hardest, but given that we're on the log scale I can't imagine that values of $\exp(0.1) \approx 1$ are very likely. I would suggest `sigma ~ dt(0,0.1,1)T(0,)` as a reasonable prior.

You are welcome to argue against these and make your own choices, especially on data sets where you are the subject expert!

Let's simulate from this model:

```
N = 100
x_sim = runif(N, 0, 100)
sigma_sim = abs(rt(1, df = 1))*0.1
beta_sim = rnorm(1, 0, 0.02)
alpha_sim = rnorm(1, 1, 1)
log_lambda_sim = rnorm(100, alpha_sim + beta_sim * (x_sim - mean(x_sim)),
                       sigma_sim)
y_sim = rpois(N, exp(log_lambda_sim))
plot(x_sim, y_sim)
```



This still produces some extreme values but in general is much more reasonable.

Exercise 4

1. Try running the above code a few times and check it matches with my pre-conceptions about the data. You could embed it in a loop and plot lots of realisations.
2. Suppose I had not mean-corrected the explanatory variable. Try re-doing the exercise. You should find it harder to specify reasonable values of e.g. `alpha`

Let's now fit this model. This time we're going to include an extra line which allows us to get pseudo-data from the posterior - our posterior predictive distribution.

First load in the data and create the sum of `rep.1`, `rep.2`, and `rep.3` ignoring the missing values, as discussed in Class 6:

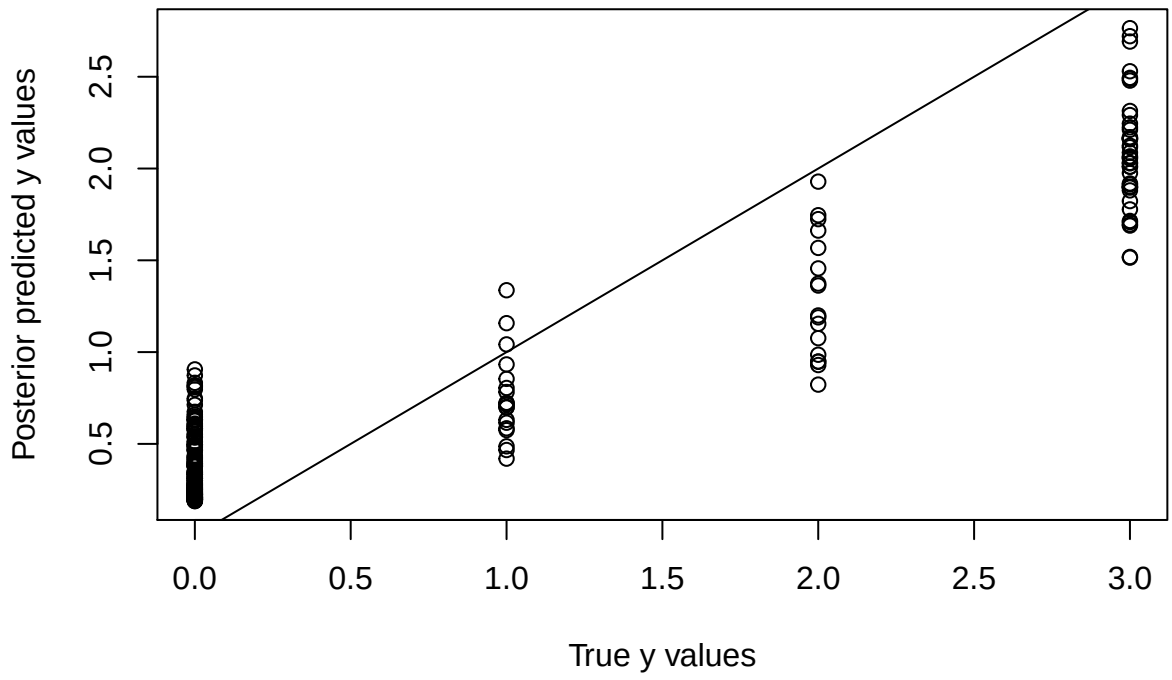
```
swt = read.csv('https://raw.githubusercontent.com/andrewcparnell/bhm_course/master/data/swt.csv')
sum_fun = function(x) {
  s = ifelse(is.na(x[1]),0,x[1]) + ifelse(is.na(x[2]),0,x[2]) + ifelse(is.na(x[3]),0,x[3])
  N = ifelse(is.na(x[1]),0,1) + ifelse(is.na(x[2]),0,1) + ifelse(is.na(x[3]),0,1)
  return(c(s,N))
}
y = apply(swt[,1:3],1,sum_fun)[1,]
```

Now run the model

```
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dpois(exp(log_lambda[i]))
    log_lambda[i] ~ dnorm(alpha + beta * (x[i] - mean(x)), sigma^-2)
    y_sim[i] ~ dpois(exp(log_lambda[i]))
  }
  alpha ~ dnorm(1, 1^-2)
  beta ~ dnorm(0, 0.02^-2)
  sigma ~ dt(0,0.1,1)T(0,)
}
'
jags_run = jags(data = list(N = nrow(swt),
                            y = y,
                            x = swt$forest),
                parameters.to.save = c('alpha','beta','sigma','y_sim'),
                model.file = textConnection(jags_code))
```

We can now do a naive plot of the posterior predictive distribution:

```
y_pp_mean = jags_run$BUGSoutput$mean$y_sim
plot(y, y_pp_mean, xlab = 'True y values', ylab = 'Posterior predicted y values')
abline(a=0, b=1)
```



This looks pretty good to me. I think it might be improved by putting `jitter(y)` in the first argument. When the data values are continuous (unlike here where they are discrete) you can also include the confidence

intervals on `y_sim` values as they can give you an idea about the uncertainty in the posterior predictive. We'll leave this for the next practical though.

Re-parameterising the model

A final useful thing to note about this model is the fact that you can re-write it slightly differently. Take a look at the following code and see if you can spot the difference:

```
jags_code_2 = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dpois(exp(log_lambda[i]))
    log_lambda[i] <- alpha + beta * (x[i] - mean(x)) + b[i]
    b[i] ~ dnorm(0, sigma^-2)
    y_sim[i] ~ dpois(exp(log_lambda[i]))
  }
  alpha ~ dnorm(1, 1^-2)
  beta ~ dnorm(0, 0.02^-2)
  sigma ~ dt(0,0.1,1)T(0,)
}
'
jags_run_2 = jags(data = list(N = nrow(swt),
                              y = y,
                              x = swt$forest),
                  parameters.to.save = c('alpha','beta','sigma','y_sim'),
                  model.file = textConnection(jags_code_2))
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 237
##    Unobserved stochastic nodes: 477
##    Total graph size: 1601
##
## Initializing model
```

Did you spot the difference? We've now added an explicit set of parameters for the over-dispersion, called `b[i]`. It's actually exactly the same model and if you compare you should get pretty much exactly the same posterior distributions for `alpha`, `beta`, and `sigma`. Some people prefer to write the model this way as it means they can see exactly the size of the random effect for each observation by watching the values of `b[i]`.

Exercise 5

1. Check that you can run both models and that they produce approximately the same posterior distributions for the parameters and DIC values.

2. To to create a model *without* over dispersion (OD). Get the DIC values for this new model (via e.g. `jags_run$BUGSoutput$DIC`). Does DIC suggest there's OD in this data set?
3. Create the posterior predictive plot for the earnings data used at the start of this practical. See if you can add in the confidence intervals on the plot of the posterior predictive.

A note about creating posterior predictive distributions in Stan

Creating posterior predictive distributions in JAGS is very easy; just one extra line of code. In Stan things are a little bit more complicated as you need to add an extra block to the model code. Here's an example of creating the posterior predictive for the above over-dispersed Poisson model:

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
stan_code = '
data {
  int<lower=0> N;
  vector[N] x;
  int y[N];
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
  vector[N] b;
} model {
  y ~ poisson_log(alpha + beta * (x - mean(x)) + b);
  for(i in 1:N)
    b[i] ~ normal(0, sigma);
  sigma ~ cauchy(0, 0.1);
  alpha ~ normal(1, 1);
  beta ~ normal(0, 0.02);
}
generated quantities {
  int<lower=0> y_sim[N];
  for (i in 1:N) {
    y_sim[i] = poisson_log_rng(alpha + beta * (x[i] - mean(x)) + b[i]);
  }
}
'
stan_run = stan(data = list(N = nrow(swt),
                            y = y,
                            x = swt$forest),
                init = 0,
                model_code = stan_code)
```

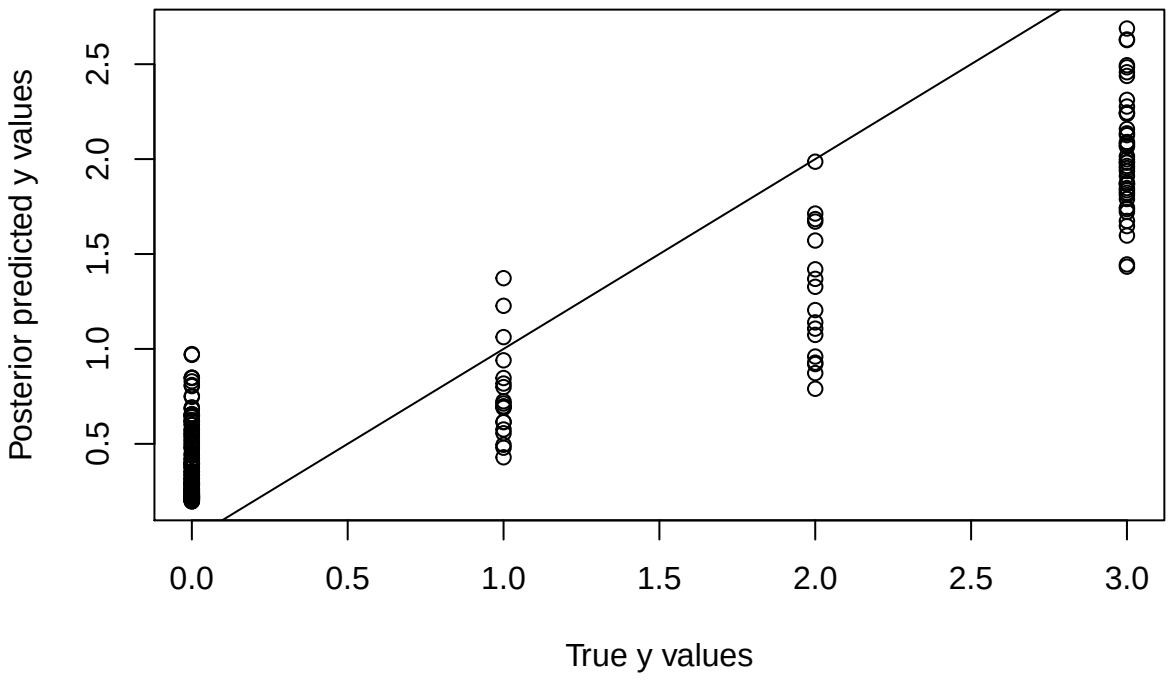
You can see the fiddly extra bit we have had to include at the bottom of the `stan_code`. We now have a `generated quantities` block which contains our `y_sim` object using a special function called `poisson_log_rng` which generates sample data from the Poisson distribution when the rate is the log of what is in the brackets.

The other interesting thing that occurred when I added the `generated quantities` block is that Stan started throwing warnings and errors at me. It turned out (after half an hour of debugging!) that the default starting values Stan uses are Uniform between -2 and 2. The beta values were getting crazily too big and it was producing infinite `y_sim` values. Calling the `stan` function with all starting values set to zero with `init=0` solved the problem.

The posterior predictive should look much the same as the JAGS script:

```
y_sim_pp_stan = extract(stan_run, pars = 'y_sim')$y_sim
y_sim_pp_mean = apply(y_sim_pp_stan,2,'mean')
```

```r
plot(y, y_sim_pp_mean, xlab = 'True y values', ylab = 'Posterior predicted y values')
abline(a=0, b=1)
```



Debugging code in JAGS and Stan

Unless you are running a very simple model, almost always you will run into problems with error statements because you have written your model incorrectly. In recent times the error messages coming out of JAGS and Stan have improved immeasurably but they are still written by people who are so familiar with the internal workings of the code that it is very hard for them to explain what you have done wrong!

Here are some general tips if you run into problems:

- Simplify your model. Go back to a model that you know works and run it on your data even if it's ludicrously simple. Try to work from that model back to the one you want to fit
- Pay really close attention to the error message. Usually you just haven't read it closely enough and it is actually telling you exactly what went wrong
- Sometimes it's nothing that you have done wrong, but just that your data and your model don't match and the model cannot start. Try specifying initial values (covered in next practical)
- Stan supports debug-by-print which means you can plug in e.g. `print(alpha);` statements throughout your code to get to the bottom of problems. This can be really helpful

Harder exercises

- We haven't covered cross-validation in this code at all. You might like to try and code up a cross-validation version of some of these models. You can copy the example from Class 6

- Try and create posterior predictive distributions for all of the models fitted in Class 6.