Practical 2 - Using R, JAGS and Stan for fitting GLMs

Andrew Parnell

Introduction

Welcome to Practical 2, an introduction to using R, JAGS and Stan for fitting Generalised Linear Models (GLMs). In this practical we'll:

- Fit some basic regression models
- Have a go at extracting and manipulating the output
- Fit some basic GLMs

Throughout this document you will see code in gray boxes. You should try to understand this code, and you can usually copy it directly from this document into your R script window. At various points you will see a horizontal line in the text which indicates a question you should try to answer, like this:

Exercise X
What words does the following command print to the console?

```
print("Hello World")
```

The two main documents I use when writing Stan/JAGS code are the manuals:

- The JAGS manual contains lots of useful about the function/distribution names, etc, but not many actual examples. If you need examples, Stack Overflow and Google are your friend.
- The Stan language manual by contrast contains loads of examples and is a great source of inspiration for some of the many things that Stan can do. If you get stuck on Stan code, head here first before you start Googling.

Here is a quick warm-up exercise:

Exercise 1

1. Load in the earnings data using `read.csv`:
2. Perform a linear regression with log(earnings) as the response and height in cm as the explanatory variable
3. Have a look at the fit of the model (R-squared, significance, etc)
4. Create a plot of the fitted values (x - axis) vs residuals

---

An introduction to JAGS

The first thing to know about JAGS is that it is a separate language from R which has its own code and structure. It is unlike Stan (which is an R package) in that JAGS is separate software that needs to be installed by itself. Hopefully you already did this when going through the pre-requisites.

To fit a model in JAGS we use this procedure:

1. Load up an R package. We will use `R2Jags` but there are others
2. Write some JAGS code and store it as a character object in R

3. Call the `jags` function, where we additionally provide data, parameters to watch, and (optionally) initial values and other algorithm details
4. Store the output in another R object and (assuming the model ran!) print, plot, or conduct further manipulation of the output

Here's a standard workflow. We're going to fit a simple linear regression using the `earnings` data as in the initial exercise above. We're then going to plot the output. I suggest first just copying and pasting this code to check that it works, then go back through the details line-by-line to understand what's happening.

First load in the data:

```
dat = read.csv('https://raw.githubusercontent.com/andrewcparnell/bhm_course/master/data/earnings.csv')
```

Now load in the package:

```
library(R2jags)
```

Type in our JAGS code and store it in an object. The code is as explained in Class 1.

```
jags_code = '
model{
  # Likelihood
  for(i in 1:N) {
    y[i] ~ dnorm(intercept + slope * x[i],
                 residual_sd^-2)
  }
  # Priors
  intercept ~ dnorm(0, 100^-2)
  slope ~ dnorm(0, 100^-2)
  residual_sd ~ dunif(0, 100)
}
'
```

Run the code by calling `jags`:

```
jags_run = jags(data = list(N = nrow(dat),
                            y = log(dat$earn),
                            x = dat$height_cm),
                parameters.to.save = c('intercept',
                                       'slope',
                                       'residual_sd'),
                model.file = textConnection(jags_code))
```

Above I am giving R a list of data which matches the objects in the `jags_code`, and I'm also telling it which parameters I am interested in via `parameters.to.save`. If you don't provide a specific parameter here, JAGS will not store it in the output object. The `textConnection` function is used to tell JAGS that the model code is stored in an R object.

Once run we can use the print or plot commands:

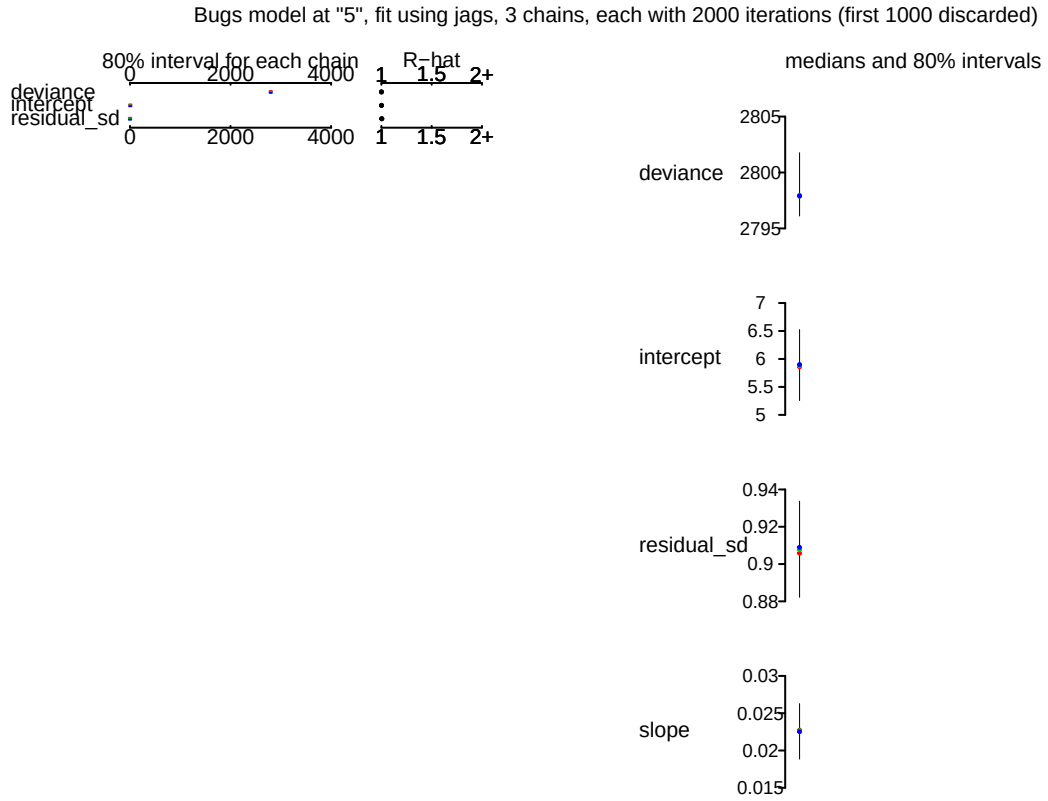
```
print(jags_run)
```

```
## Inference for Bugs model at "5", 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%
## intercept      5.886   0.489    4.949    5.561    5.880    6.204    6.906
## residual_sd    0.908   0.020    0.871    0.894    0.908    0.921    0.947
## slope          0.023   0.003    0.017    0.021    0.023    0.024    0.028
```

```
## deviance    2798.530   2.405 2795.746 2796.731 2797.922 2799.715 2804.645
##               Rhat n.eff
## intercept   1.002  1800
## residual_sd 1.004   570
## slope       1.002  1500
## deviance    1.001  3000
##
## 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 = 2.9 and DIC = 2801.4
## DIC is an estimate of expected predictive error (lower deviance is better).
```

The important things to look at here are the estimated posterior means and standard deviations, the quantiles, and also the Rhat values. You can get a slightly useless plot with:

```
plot(jags_run)
```



Exercise 2

Once you are happy that the above runs for you, try the following:

1. Try changing the `x` variable in the data list from `height_cm` to `height`. How does the model change?
2. Try changing some of the prior distributions in the JAGS code. What happens if you make the prior standard deviations, or the `residual_sd` upper limit very small?

3. Try removing one or more of the parameters given in the `parameters.to.save` argument of the JAGS call. What happens to the resulting `print/plot` commands? Do any of the values change?

Manipulating output in JAGS

The object created from the `jags` call is a huge list with lots of things in it, most of them useless. A good command to explore the structure of a list is `str`:

```
str(jags_run)
```

There are two useful objects to extract out of the model. The first is the list of all the parameter values. You can get this with:

```
pars = jags_run$BUGSoutput$sims.list
str(pars)
```

```
## List of 4
##  $ deviance   : num [1:3000, 1] 2802 2798 2796 2798 2802 ...
##  $ intercept  : num [1:3000, 1] 4.96 5.3 5.55 6.11 6.53 ...
##  $ residual_sd: num [1:3000, 1] 0.924 0.931 0.903 0.921 0.866 ...
##  $ slope      : num [1:3000, 1] 0.0278 0.026 0.0246 0.0215 0.0189 ...
```

After this run we can get at, e.g. the first 10 sample values of the `intercept` parameter with:

```
pars$intercept[1:10]
```

```
##  [1] 4.959564 5.304473 5.554114 6.108635 6.525872 6.402061 5.869030
##  [8] 6.063816 5.844860 6.230177
```

From this we can calculate the posterior mean with, e.g. `mean(pars$intercept)` but we can also get this more directly with:

```
par_means = jags_run$BUGSoutput$mean
str(par_means)
```

```
## List of 4
##  $ deviance   : num [1(1d)] 2799
##  $ intercept  : num [1(1d)] 5.89
##  $ residual_sd: num [1(1d)] 0.908
##  $ slope      : num [1(1d)] 0.0226
```

The other things you might want to get out are also stored in the `BUGSoutput` bit:

```
str(jags_run$BUGSoutput)
```

Exercise 3

1. See if you can extract the posterior medians. What is the command?
2. See if you can extract the posterior 95% confidence interval for a given parameter (hint: look in `$summary`)
3. Try creating some plots of the output, e.g. histograms of the posterior parameter values (hint: use the `pars` object created above)

An introduction to Stan

If you can fit models in JAGS then it's pretty easy to move them over to Stan. The structure of the workflow is very similar, as are the commands. Stan code tends to be slightly longer as you have to declare all of the data and parameters, and this has its disadvantages (longer code), and advantages (easier to follow).

When you load up stan, it's best not just to call the package, but also to run two additional lines:

```
library(rstan)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

These two lines allow Stan to run in parallel on a multi-core machine which can really speed up workflow. We now specify our model code:

```
stan_code = '
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
parameters {
  real intercept;
  real slope;
  real<lower=0> residual_sd;
}
model {
  y ~ normal(intercept + slope * x, residual_sd);
}
'
```

It's a good idea to compare and contrast the code between JAGS and Stan. The key differences are:

1. Stan has different blocks of code. Here they are `data`, `parameters`, and `model`. There are others we will come to later
2. Stan requires you to specify the dimension of all these values. By contrast JAGS guesses the dimensions for you
3. Stan doesn't require you to put prior distributions on all the parameters, JAGS does. Stan thus assumes flat prior distributions for all parameters by default. It's much better practice to always put prior distributions in your model code
4. Stan parameterises the normal distribution as `(mean, standard deviation)`. By contrast, JAGS uses `(mean, precision)`. Remember precision is the reciprocal of the variance, so precision = 1 / sd^2
5. Stan ends each line with a semi-colon. JAGS does not
6. Stan uses `=` to assign variables, JAGS uses `<-`

Finally we run the model with:

```
stan_run = stan(data = list(N = nrow(dat),
                            y = log(dat$earn),
                            x = dat$height_cm),
                model_code = stan_code)
```

In contrast to JAGS, we do not need to specify the parameters to watch. Stan watches all of them by default. We also don't need a `textConnection` command to provide the `stan` function with the stan code. The `data` call is identical to JAGS.

I often find with Stan that I get lots of warnings when I run a model, especially with more complicated models (which we haven't met yet). Most of them tend to be ignorable and are unhelpful. Usually Stan gives

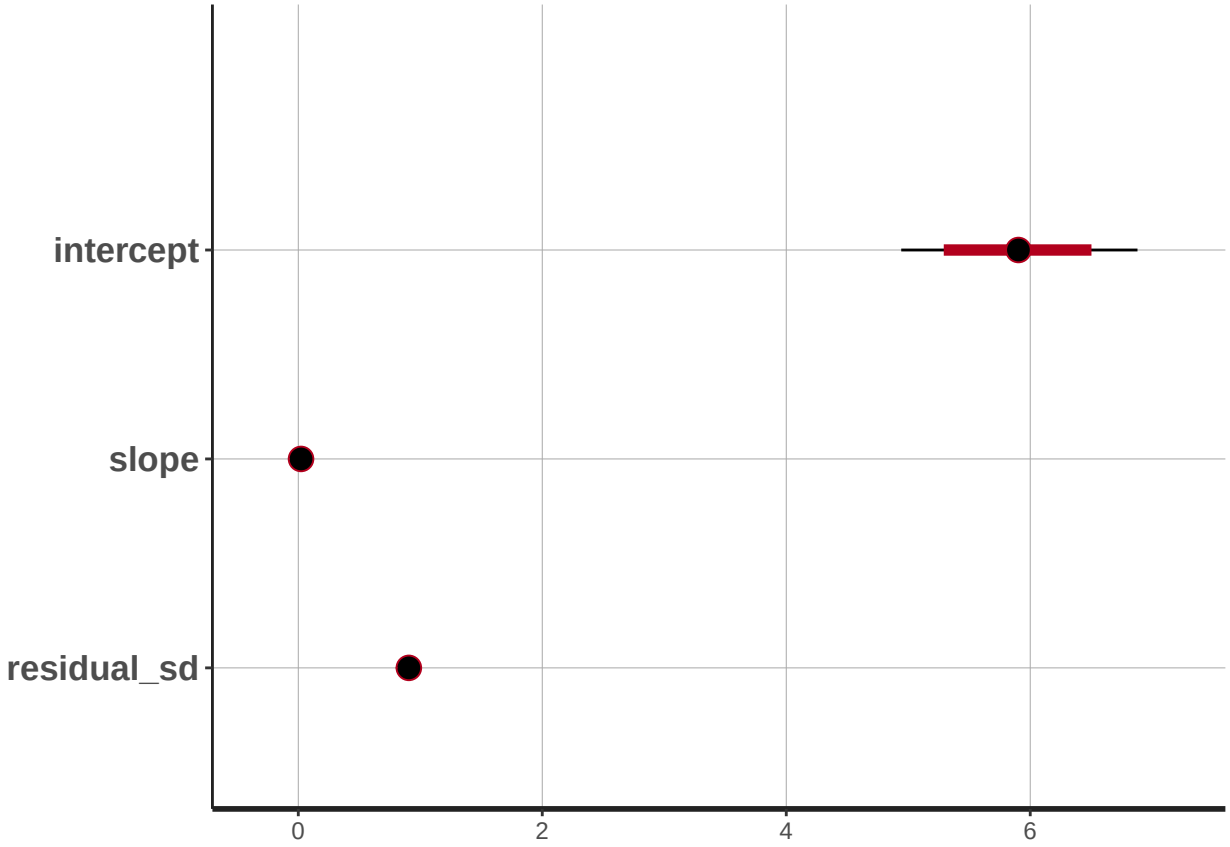
you a count of how many 'bad things' happened in each chain. However, there doesn't seem to be much guidance on what to change, if anything, in response. If you start to see lots of warnings in your Stan output, let me know!

Just like JAGS, we can print or plot the output. This is also a pretty useless plot:

```
plot(stan_run)
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```



Exercise 4

Let's do the same exercises again for Stan:

1. Try changing the `x` variable in the data list to `height`. How does the model change?
2. Try adding in some prior distributions to the Stan code. For example, add `intercept ~ normal(0, 100);`. Again experiment with making these ranges small or large

---

Manipulating output in Stan

Stan by default saves all the parameters, but they are not easily accessible directly from the `stan_run` object we just created. Instead there is a special function called `extract` to get at the posterior samples.

```
pars = extract(stan_run)
str(pars)
```

```
## List of 4
##  $ intercept  : num [1:4000(1d)] 5.91 6.25 5.94 5.27 5.53 ...
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ iterations: NULL
##  $ slope      : num [1:4000(1d)] 0.0223 0.0205 0.0226 0.0262 0.0246 ...
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ iterations: NULL
##  $ residual_sd: num [1:4000(1d)] 0.936 0.933 0.902 0.905 0.896 ...
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ iterations: NULL
##  $ lp__       : num [1:4000(1d)] -427 -426 -426 -426 -425 ...
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ iterations: NULL
```

If we want the first 10 samples of the intercept we can run:

```
pars$intercept[1:10]
```

```
##  [1] 5.905598 6.253130 5.937122 5.265238 5.534437 5.543804 5.799504
##  [8] 5.565497 6.910885 5.106497
```

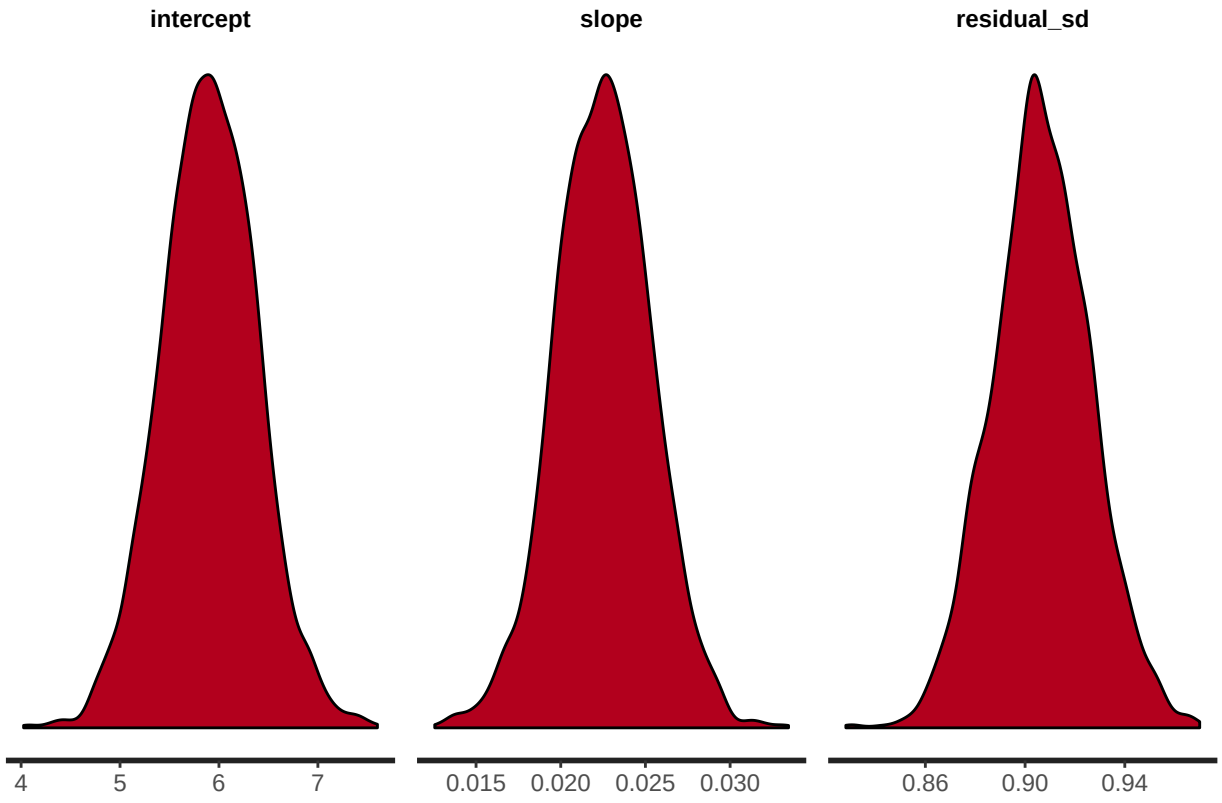
Alternatively if you want to get at the summary statistics you can run:

```
pars_summ = summary(stan_run)$summary
pars_summ['intercept','mean']
```

```
## [1] 5.901651
```

In addition, stan has some useful plotting commands, e.g.

```
stan_dens(stan_run)
```



You can find a bigger list with `?stan_plot`

Exercise 5

As with JAGS:

1. See if you can extract the posterior medians. What is the command?
2. See if you can extract the posterior 95% confidence interval for a given parameter
3. Try creating some plots of the output, e.g. histograms of the posterior parameter values

---

Fitting GLMs

Once you have the basic workflow in place, and the ability to manipulate output, the only hard thing that remains is the ability to write your own JAGS/Stan code to fit the model you want.

Recall that for a generalised linear model (GLM) we don't have a normally distributed likelihood, instead we have a probability distribution that matches the data (e.g. binomial for restricted counts, Poisson for unrestricted) and a *link function* which transforms the key parameter (usually the mean parameter) from its restricted range into something we can use in a linear regression type model.

Here is some JAGS code to fit a Binomial-logit model to the Swiss willow tit data:

```
swt = read.csv('https://raw.githubusercontent.com/andrewcparnell/bhm_course/master/data/swt.csv')
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))
```

Most of the above steps you have already seen. The key things to note are:

- In the jags code we have `y[i] ~ dbin(p[i], 1)` to represent the binomial likelihood with probability p_i and number of trials set to 1. (If you've forgotten the details of the binomial model please ask!)
- The link function here is the `logit`. Both JAGS and Stan are slightly different from other programming languages in that you can write the link function on the left hand side. In most programming languages you would have to write something like `p[i] <- logit_inverse(alpha + beta*x[i])`
- I'm using `alpha` and `beta` here, rather than `slope` and `intercept`

Once run, you can manipulate the object in any way you like. Let's create a simple plot of the data with the fitted line going through it. A useful package to call here is the `boot` package which contains the `logit` and `inv.logit` functions. You can try them out with:

```
library(boot)
logit(0.4) # Convert from 0-to-1 space to -Infinity to +Infinity
```

```
## [1] -0.4054651
```

```
inv.logit(3) # ... and the opposite
```

```
## [1] 0.9525741
```

To create the plot we will first get the posterior means of the slopes:

```
post_means = jags_run$BUGSoutput$mean
alpha_mean = post_means$alpha
beta_mean = post_means$beta
```

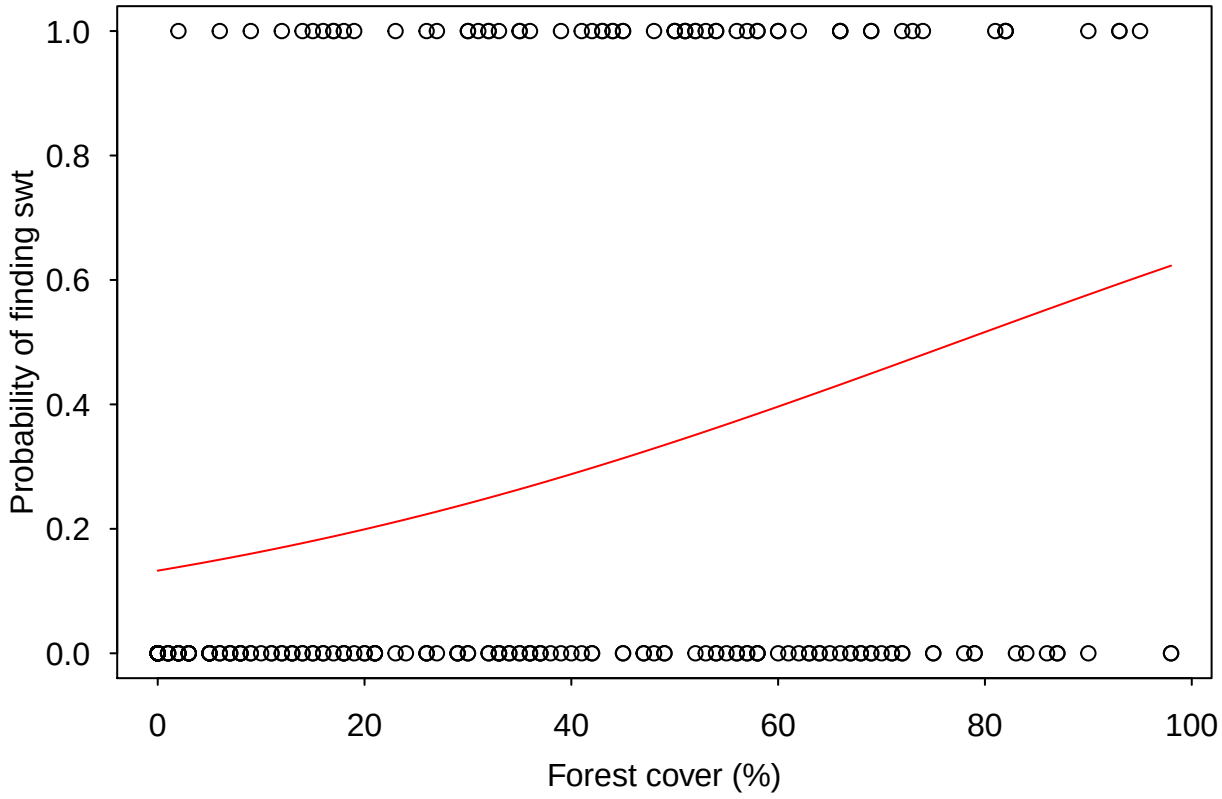
Now plot the data and add in the predicted values of the probability by creating new explanatory variables on a grid:

```
par(mar=c(3,3,2,1), mgp=c(2,.7,0), tck=-.01, las=1) # Prettify plots a bit
with(swt, plot(forest, rep.1,
     xlab = 'Forest cover (%)',
     ylab = 'Probability of finding swt'))
forest_grid = pretty(swt$forest, n = 100)
lines(forest_grid, inv.logit(alpha_mean + beta_mean * forest_grid), col = 'red')
```

```
## Warning in beta_mean * forest_grid: Recycling array of length 1 in array-vector arithmetic is deprec
##   Use c() or as.vector() instead.
```

```
## Warning in alpha_mean + beta_mean * forest_grid: Recycling array of length 1 in array-vector arithme
```

```
##   Use c() or as.vector() instead.
```



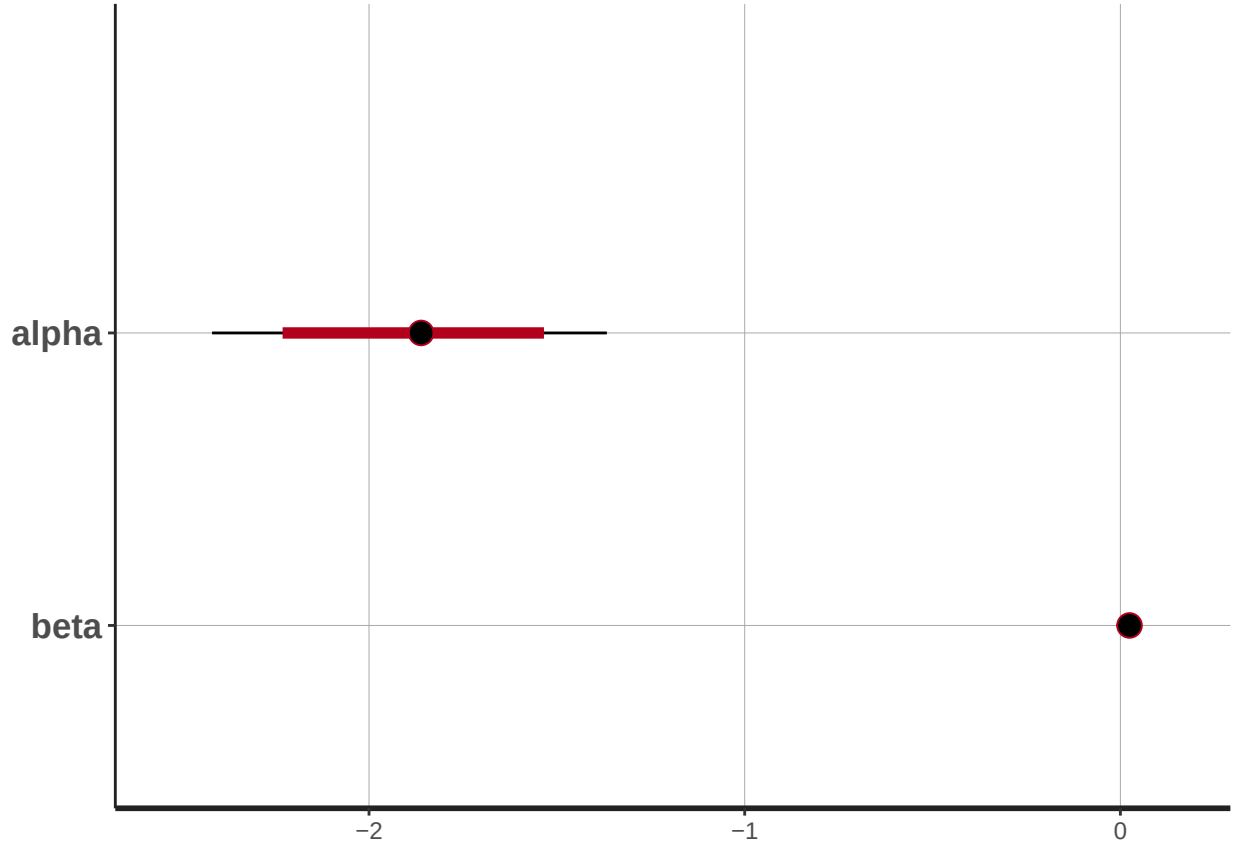
At this point if you're feeling really brave, you could try and code the above model in Stan without looking any further down the page. I'd suggest looking in the manual for the `binomial_logit` function (unfortunately there aren't currently any examples in the Stan manual for Binomial-logit regression).

If you're not so confident (or you've finished and want to check), here is the code for a Stan Binomial logit model.

```
stan_code = '
data {
  int<lower=0> N;
  vector[N] x;
  int y[N];
}
parameters {
  real alpha;
  real beta;
} model {
  y ~ binomial_logit(1, alpha + beta * x);
  alpha ~ normal(0, 10);
  beta ~ normal(0, 10);
}
'
stan_run = stan(data = list(N = nrow(swt),
                            y = swt$rep.1,
                            x = swt$forest),
                model_code = stan_code)
```

```
## In file included from file9fbb7d55eff3.cpp:8:
## In file included from /Users/andrewparnell/Library/R/3.4/library/StanHeaders/include/src/stan/model/
## In file included from /Users/andrewparnell/Library/R/3.4/library/StanHeaders/include/stan/math.hpp:4
## In file included from /Users/andrewparnell/Library/R/3.4/library/StanHeaders/include/stan/math/rev/ma
## In file included from /Users/andrewparnell/Library/R/3.4/library/StanHeaders/include/stan/math/rev/co
## In file included from /Users/andrewparnell/Library/R/3.4/library/StanHeaders/include/stan/math/rev/co
## In file included from /Users/andrewparnell/Library/R/3.4/library/StanHeaders/include/stan/math/rev/co
## In file included from /Users/andrewparnell/Library/R/3.4/library/BH/include/boost/math/tools/config.h
## In file included from /Users/andrewparnell/Library/R/3.4/library/BH/include/boost/config.hpp:39:
## /Users/andrewparnell/Library/R/3.4/library/BH/include/boost/config/compiler/clang.hpp:200:11: warning
## #  define BOOST_NO_CXX11_RVALUE_REFERENCES
##           ^
## <command line>:6:9: note: previous definition is here
## #define BOOST_NO_CXX11_RVALUE_REFERENCES 1
##         ^
## 1 warning generated.
```

```
plot(stan_run)
```



The things to note about this code are:

1. I've set `x` to be a `vector` and `y` to be an `int`. This is because `x` is continuous (it's our percentage of forest cover) but `y` must be a whole number to be a valid binomial random variable. If you try to set `y` as a vector too Stan will complain
2. One of the oddities of Stan is that you write `vector[N] x` but `int y[N]`. So the `[N]` appears in different places for vectors than for integers. I can never remember which way round is correct!

3. The likelihood term here using the function `binomial_logit` does the two lines that JAGS uses in one go. It calculates the binomial probability and creates the logit of `alpha + beta * x` for you.
4. Stan automatically vectorises most things so you don't need a `for` loop like JAGS
5. If you don't give the prior distributions to Stan here you will end up with flat prior distributions on the parameters. This is bad practice but appears frequently in the Stan manual and examples.

Exercise 6

Pick your favourite so far (either JAGS or Stan) and try these:

1. Change the explanatory variable from `forest` to `elev`. Re-create the plots and interpret your findings.
2. Try and adapt the model to include both `forest` and `elev`. How do the results change?

Now try and swap programmes and try the above again! Then compare the results of JAGS vs Stan on the GLM. They should look similar, though the algorithm they use is stochastic so the results won't be identical.

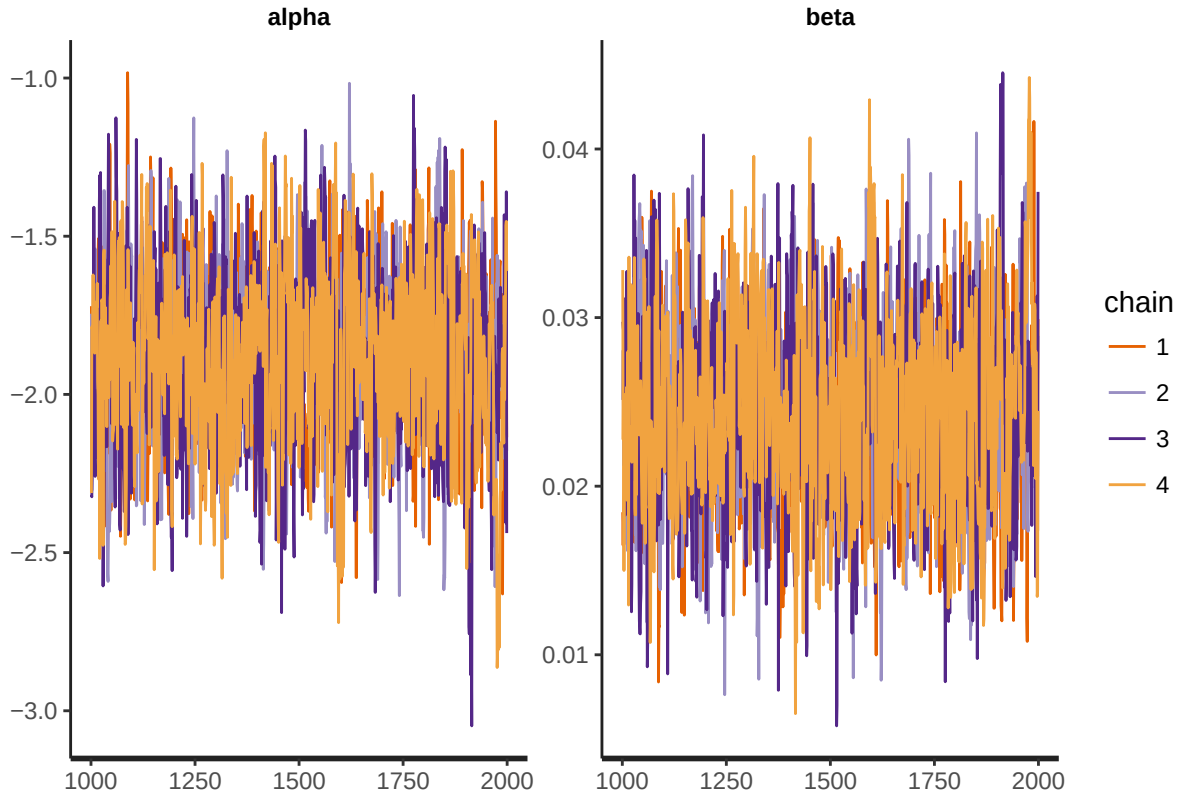
Checking and changing the model run

Both JAGS and Stan provide the Rhat value with their default `print` output. Remember the rule of thumb is that the model run is satisfactory if all values are less than 1.1. If the Rhat values are above 1.1 then you have two possibilities:

1. Something about your model is mathematically wrong (often I find that I'm missing something like an intercept term in my model). You need to look carefully at your model, and perhaps get some help from a statistician.
2. You need to run the model for longer

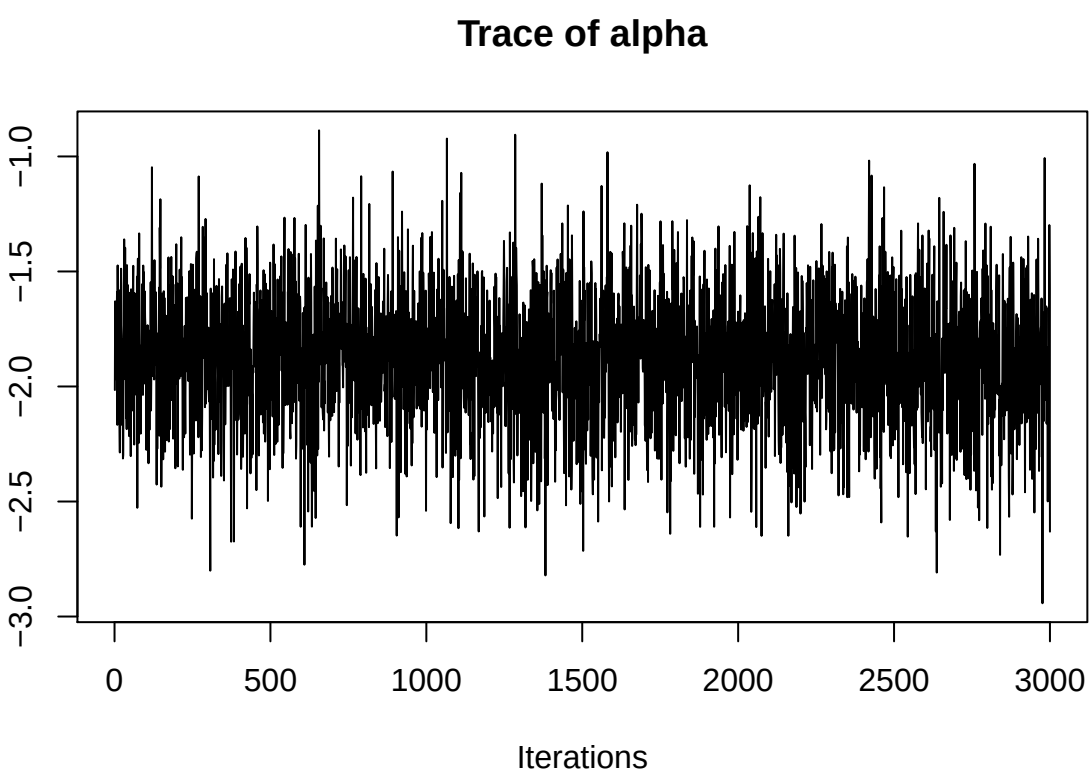
If the Rhat values are above 1.1 then the first thing I do is create a *trace plot* of some of the parameters. This is the plot of the parameter values at each iteration. You can get it in Stan from:

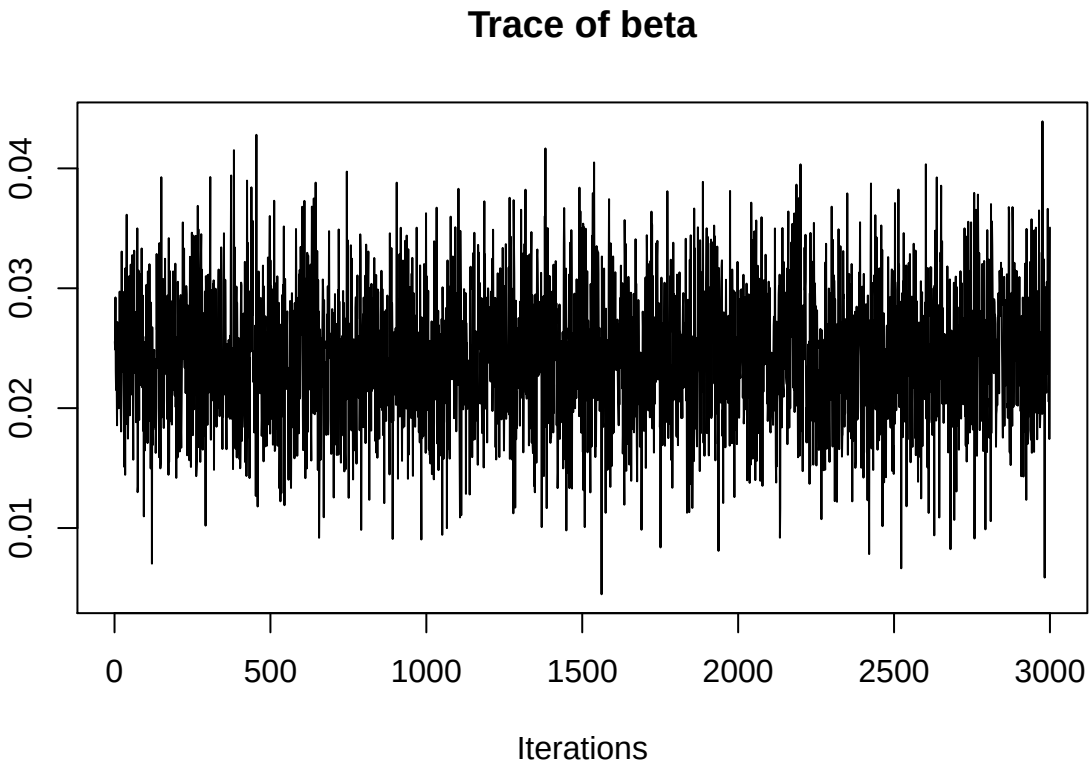
```
rstan::traceplot(stan_run)
```

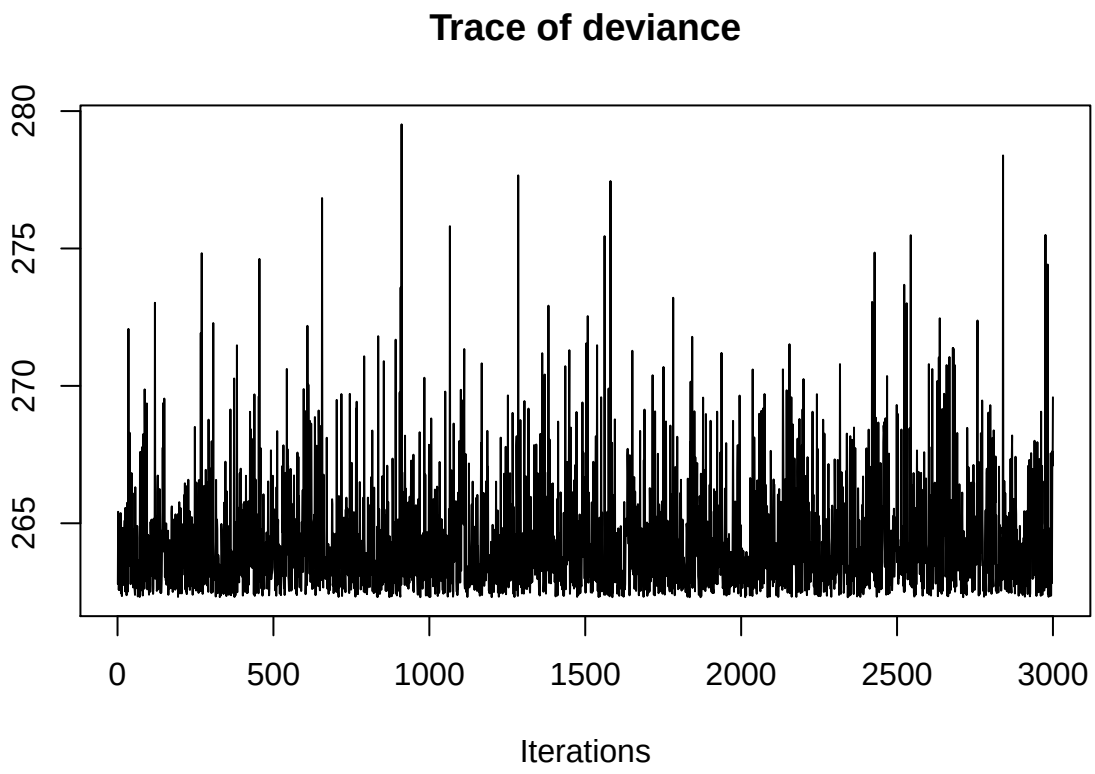


It's a bit more fiddly in JAGS:

```
coda::traceplot(as.mcmc(jags_run$BUGSoutput$sims.matrix))
```


Trace of alpha
−1.0
−1.5
−2.0
−2.5
−3.0
0
500
1000
1500
2000
2500
3000
Iterations


Trace of beta
0.01
0.02
0.03
0.04
0
500
1000
1500
2000
2500
3000
Iterations



The reason for the `stan::` and the `coda::` bit is because JAGS and Stan each have a function called `traceplot`. You have to specify the package to get the correct one. The traceplots should look like hairy caterpillars and shouldn't be wandering around or stuck in a location.

If the traceplots look bad there are five things you can change in a JAGS/Stan run to help. These were all covered in Class 2 so go back to the slides if you have forgotten. They are:

1. The number of iterations (JAGS/Stan default 2000). This is the first thing to increase
2. The size of the burn-in (JAGS/Stan default 1000). Increase this if it looks like the parameter values settled after a while but you need to remove an initial chunk
3. The amount of thinning (JAGS/Stan default 1). Increase this if the parameters are wandering around slowly
4. The starting parameter guesses. By default JAGS and Stan come up with their own, but you can provide a function which generates good starting values and provides it to JAGS/Stan. We will see an example of this in one of the later classes. It's usually only necessary for very complex models that struggle to start.
5. The number of chains (JAGS default 3, Stan default 4). There's usually no need to increase this unless you have a lot of spare processors and want to be *really sure* of convergence

Here's an example of Stan code with the iterations/burn-in/thinning values doubled (Stan calls the burn-in the `warmup`):

```
stan_run = stan(data = list(N = nrow(swt),
                            y = swt$rep.1,
                            x = swt$forest),
                iter = 4000,
                warmup = 2000,
                thin = 2,
```

```
                model_code = stan_code)
```

and here's the same example in JAGS

```
jags_run = jags(data = list(N = nrow(swt),
                            y = swt$rep.1,
                            x = swt$forest),
                parameters.to.save = c('alpha',
                                       'beta'),
                n.iter = 4000,
                n.burnin = 2000,
                n.thin = 2,
                model.file = textConnection(jags_code))
```

For the vast majority of problems we will cover (up until the last few classes) you shouldn't need to change any of the defaults but it is useful to know.

Harder exercises

1. We previously met a Poisson GLM but we didn't fit it to any data. Try fitting it using to the `whitefly.csv` data set using the `imm` variable as the response and the `trt` variable as a covariate. Don't worry about interpreting the output, just get the model to run and converge.
2. Next see if you can fit the above model in Stan. You'll need to look up the `poisson_log` function in the Stan manual
3. The variable `n` in this data set is a potential offset. Use the notes on offsets in Class 2 to fit a model that uses the `n` variable as an offset. Again, don't worry about interpretation. Just get the models to run!