Hey all!

Today I'm going to talk about linear regression. It's only part 1, so stay tuned for the other parts that will accompany this. Turns out there's a lot to say about linear regression!

This tutorial will hopefully cover the basics of Linear Regression with continuous variables and expose you to NIMBLE/JAGS. Future tutorials will go more into depth model selection, categorical variables and random error. But for now, keeping it simple!

I'm assuming for now that everyone has downloaded JAGS and NIMBLE and has them setup on their computers. Before you use NIMBLE make sure R, and Rtools or Xcode are updated on your computer (otherwise a weird "shared library" error can come up). JAGS is downloadable here: https://sourceforge.net/projects/mcmc-jags/ and NIMBLE can be found here: https://r-nimble.org/download.

Please send any questions or suggestions to heather.e.gaya@gmail.com or find me on twitter @doofgradstudent

# Contents

# 1 General Process

When working with models in JAGS and NIMBLE, there's generally a 5 step process:

1. Data cleanup
2. Model Writing
3. Debugging and Running the Model
4. Inspection of Results
5. Visualization/Writeup/Etc.

Let's go through these step by step using a fake scenario.

# 2 A Fake Scenario

Joe goes out in the world and collects 50 frogs. He records each frog's age (in days), weight (in g), left back leg length (cm), the distance the animal was from the road (cm) and what species the frog is (A or B). His data looks like this (but with 50 rows):

| | Frog | Age | Weight | Leg | Dist | Species |
|---|---|---|---|---|---|---|
| 1 | 1 | 191 | 15.44 | 4.90 | 57.61 | A |
| 2 | 2 | 184 | 21.45 | 3.44 | 92.08 | A |
| 3 | 3 | 243 | 21.39 | 4.37 | 54.87 | A |
| 4 | 4 | 770 | 114.28 | 3.42 | 189.96 | A |
| 5 | 5 | 614 | 64.52 | 3.73 | 38.64 | A |
| 6 | 6 | 771 | 79.07 | 3.18 | 13.33 | A |

Joe suspects weight is related to age, back leg length and distance, so he chooses those three variables to model.

Essentially, joe thinks: Expected weight of frog = intercept + age X something + leg X something2 + distance X something3. If this were a perfect predictor, then the actual weight of the frogs would equal the expected weight. But Joe knows frogs are more complex than that.

To deal with the complexity of frogs, he also thinks that maybe expected weight is the mean of a normal distribution, with the true weight falling somewhere on the bell curve.

In mathy math format, Joe's model is:

$$E(weight) = \beta_0 + \beta_1 A + \beta_2 L + \beta_3 D$$
$$Actualweight \sim Normal(\mu = E(weight), \sigma = sd)$$

where $\beta_1$ is the intercept, $\beta_1$ is "something", $\beta_2$ is "something2", etc. Calling it "something" is not really accepted in scientific journals, so we use betas instead. The intercept is often referred to as beta0, as I have done above. In a real journal format you'd also want to call age, legs and distance X1, X2, X3, but this tutorial is far from a real journal :)

# 3 JAGS Model and Running JAGS

If you are using NIMBLE, I suggest reading through this explanation first, as the NIMBLE model will change very little from the JAGS model :)

Our JAGS Model will start with the equation above. However, JAGS will need things to be indexed, so that each frog's data is properly grouped together.

```
for (i in 1:n.frogs){
  meanweight[i] <- beta0 + age[i]*beta1 + leg[i]*beta2 + distance[i]*beta3
  weight[i] ~ dnorm(meanweight[i], prec)
}
```

In JAGS, observations draw from distributions (stochastic nodes) use the $\sim$ symbol, whereas those calculated from an equation (deterministic nodes) use the piping symbol ($<-$). Additionally, the normal distribution uses precision, $\tau$, rather than standard deviation, $\sigma$. For those unfamiliar:

$$\tau = \frac{1}{\sigma^2}$$

This is all well and good, but now we have a bunch of variables in here with no values! We address those next using *priors*. Since we have no idea what the true values are, let's draw them from a uniform distribution. This is called an "uninformative prior" because it's really not constraining the answer at all. We will give uninformative priors to all the betas and to precision. I like to convert precision to standard deviation because it's easier for me to understand, but that's not a requirement.

```
beta0 ~ dunif(-50,50)
beta1 ~ dunif(-50,50)
beta2 ~ dunif(-50,50)
beta3 ~ dunif(-50,50)

prec <- 1/(sd *sd)
sd ~ dunif(0.0001, 100)
```

Now we've defined everything except n.frogs, age, leg, distance and weight. These variables are all data, so we'll give them to JAGS before we run the model. We can now combine everything and stick it together in a modelstring to pass to JAGS (via runjags in R or another similar package).

```
modelstring.Frogs = "
  model
{
for (i in 1:n.frogs){
  meanweight[i] <- beta0 + age[i]*beta1 + leg[i]*beta2 + distance[i]*beta3
  weight[i] ~ dnorm(meanweight[i], prec)
}

beta0 ~ dunif(-50,50)
beta1 ~ dunif(-50,50)
beta2 ~ dunif(-50,50)
beta3 ~ dunif(-50,50)

```

```
prec <- 1/(sd *sd)
sd ~ dunif(0.0001, 100)
}
"
```

So now we've made our model! We need to tell JAGS a little bit more info before we can actually run the model. First, we need to tell JAGS what variables we're interested in. For instance, since we're giving the model all the frog ages, we aren't interested in having the model tell us the frog ages.

For this model, we're going to monitor the betas and the sd, since those will tell us things about the relationship of weight to the other variables.

```
params <- c("beta0", "beta1", "beta2", "beta3","sd")
```

Next we give JAGS the data as a list object.

```
data <- list(weight = Frogs$Weight, distance = Frogs$Dist, age = Frogs$Age
    , leg = Frogs$Leg, n.frogs = 50)
```

(Reminder that in R, you can select columns in dataframes using the $ symbol and the name of the column)

The other thing we want to give JAGS are initial values. If you have some idea of the expected values, you can specify them here too. But if you don't know anything, you can choose random values. Make sure these random values make sense! You can't start the model with beta3 = 500 because above we said beta3 was between -50 and 50.

SIDE NOTE: If you ever get an error about "invalid parent node" it means the starting values are outside the prior you set for that node. It comes up a lot.

```
inits <- function(){list(beta0 = runif(1, -10, 10), beta1 = runif
    (1,-10,10), beta2 = runif(1,-10,10), beta3 = runif(1,-10, 10))}
```

And now we can run the model! You can use whatever package you want, but I like runjags. The arguments are fairly straightforward. Model = the name of your model from above, monitor = what parameters are you interested in, inits = the initial values to start the model at and data = your data. Some less familiar ones follow. n.chains = how many independent runs of the model do you want? This is important for estimating convergence, which we will discuss (briefly) in a minute. Sample is how many times you want to run the MCMC to give you an estimate of the true value of your parameters. I like to start with a smallish number and add more as needed if my model didn't converge. Lastly, method = "parallel" allows your computer to run the chains on separate cores of your computer for faster processing.

```
Frog.mod <- run.jags(model = modelstring.Frogs,
                     monitor = params, inits = inits,
                     data = data, n.chains = 3,
                     sample = 10000, method = "parallel")
```

# 4 A Very Brief Explanation of MCMC

So why do we do MCMC? And what's all this about chains???

The short answer is - calculus is hard. And sometimes unsolvable across the entire domain of a distribution. In theory, and for easy distribution combinations, we could go to the trouble of taking integrals of distributions to estimate the distribution of the parameters we are interested in. But when you have a ton of distributions all combining in one model, it gets to be a headache. So instead, the MCMC "wanders around" through different possible values. So at each iteration, the MCMC chain picks a new possible value to go to (and in most samplers, this is somewhat near its current value) and evaluates the distribution there. If it does this enough times, eventually the "accepted" samples will look fairly similar to the true distribution! And then we can ask the chains what percent of the time they spent at various values, which gives us the credible interval of the parameters we are interested in. Now then, if we just have one chain, it's hard to tell if it's been running for long enough. Imagine that the true distribution looks like two hills with a valley in between. Maybe the chain is just exploring one peak, when the other peak is still out there somewhere, unrepresented in the iterations already run. This is why we use multiple chains. If multiple chains, starting at different points and bouncing around the mathematical space end up telling us the same answer, there's a much higher chance that the answer is correct! Most people use 2 or 3 chains - you need at least two to estimate convergence (AKA, the chains are agreeing on the same answer).

Obviously there's a lot more to it than that, but I have found that for the most part you don't need to full understand MCMC to actually use it. Just know that people far far smarter than me have developed this lovely mechanism to allow us to estimate the probability that some parameter has a specific value. It's really quite nifty!

# 5 JAGS Output

Anyway, back to JAGS. To view our model output, we look at a summary of the chains we've run. It will look something like this:

| | Lower95 | Median | Upper95 | Mean | SD |
|---|---|---|---|---|---|
| beta0 | -17.71 | -8.09 | 3.01 | -7.83 | 5.32 |
| beta1 | 0.12 | 0.12 | 0.13 | 0.12 | 0.00 |
| beta2 | -4.50 | -2.04 | 0.13 | -2.09 | 1.20 |
| beta3 | 0.14 | 0.17 | 0.19 | 0.17 | 0.01 |
| sd | 4.36 | 5.38 | 6.63 | 5.44 | 0.59 |

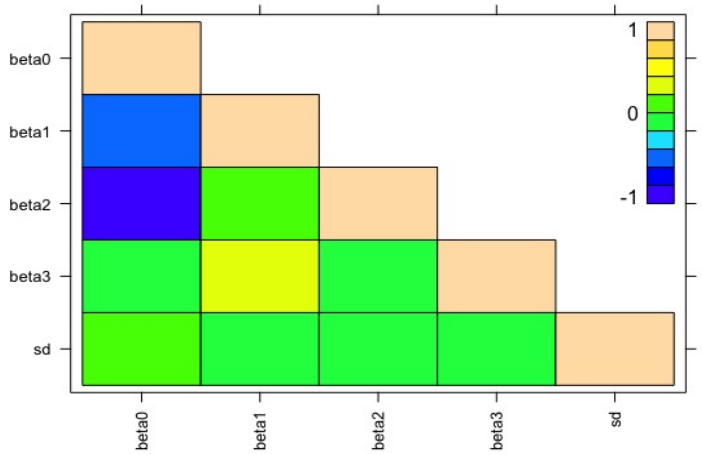
| | Mode | MCerr | MC%ofSD | SSeff | AC.10 | psrf |
|---|---|---|---|---|---|---|
| beta0 | | 0.34 | 6.30 | 250.00 | 0.85 | 1.01 |
| beta1 | | 0.00 | 2.00 | 2462.00 | 0.18 | 1.00 |
| beta2 | | 0.07 | 5.80 | 295.00 | 0.83 | 1.01 |
| beta3 | | 0.00 | 1.60 | 4013.00 | 0.09 | 1.00 |
| sd | | 0.01 | 0.90 | 12502.00 | 0.01 | 1.00 |

The first 3 columns are the credible interval. Unlike confidence intervals, credible intervals are easy to make sense of. 95% of our samples from our chains fell between the upper and

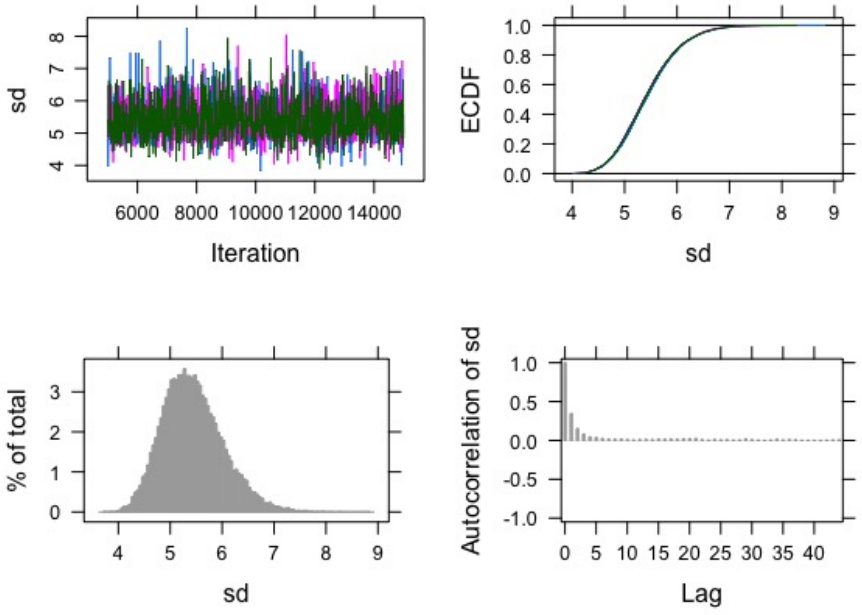
lower bounds. The mean is the point estimate for that variable. So for our frogs, beta0 is between (-17.71,3.01) with a mean value of (-7.83). SD is the standard deviation, the mode is the mode (blank for continuous variables).

The other useful value to look at is "psrf" which is a statistic relating to convergence. This tells you about how different the chains are and is a proxy for convergence. Convergence is generally reached at values less than 1.1, but visual inspection of the chains is always important too! All our values seem to have converged, but let's check the plots to make sure.

```
plot(Frog.mod)
```




For runjags, the first plot represents correlation. Generally you want to avoid correlated variables if at all possible. But our plot doesn't look tooooo terrible, so we'll roll with it. The other plots will all have 4 parts, as seen below:



Panel 1 (top left) shows the chains mixing together. They look kind of like a blur of green/pink/blue. This is what we want to see. This shows the MCMC exploring the statespace and the 3 chains coming together in agreement of values. The mean is also pretty steady - if you were to draw a trend line through the chains, it would be fairly flat. This is what we want to see.

Panel 2 (top right) shows the ECDF - a cumulative density plot of all the values for sd. Again, this plot should look like almost one color if the chains are mixing well.

Panel 3 shows a histogram of the parameter values (in this case, sd) from all the chains. This helps give you an idea of the spread of values that parameter can take. A nice visual way to quickly see values. This is also a good time to check if your prior was reasonable. If we see the bars are "bumping up" against an xlimit, that's a sign that we should maybe rethink our model a little bit.

SIDE NOTE: When people say "the posterior distribution of my parameter indicated" etc etc, this is the posterior distribution they are talking about.

Panel 4 is autocorrelation. During MCMC, or at least during the MCMC algorithm used by JAGS, the values of the chain from one iteration to the next are correlated. The autocorrelation tells us how many iterations apart we have to be before we have independent draws from the distribution. Generally we just want this to drop down fairly quickly - we don't want to see values 100 iterations apart still being autocorrelated with each other. If this happens, we will want to think more deeply about our MCMC process and maybe consider running more iterations or thinning (only looking at every xth value of the chain) to get a better idea of the posterior distribution.

So now we've looked at all our chains and the summary stats and everything looks pretty good! We can now update our equation to reflect the mean parameter values we found:

$$E(weight) = -7.83 + 0.12A - 2.09L + .17D$$
$$Actualweight \sim Normal(\mu = E(weight), \sigma = 5.44)$$

And that's it for Linear Regression in JAGS Part 1! Down below I explain how to do it in NIMBLE (which I encourage everyone to learn, as I suspect it is the way of the future). Stayed tuned for Part 2 - categorical variables and Part 3 - graphing the results with credible intervals!

# 6 NIMBLE Model and Running NIMBLE

The NIMBLE version of this model will require only a few tiny changes, and only one of them is to the actual model structure. Most of the changes are just coding things and the wrappers used to tell NIMBLE what is what.

Take a look at the model in NIMBLE:

```
nimbleFrogs <-
  nimbleCode({ #so this is a new change
for (i in 1:n.frogs){
  meanweight[i] <- beta0 + age[i]*beta1 + leg[i]*beta2 + distance[i]*beta3
  weight[i] ~ dnorm(meanweight[i], sd = sd) # the other change
}
```

```
7
8  beta0 ~ dunif(-50,10)
9  beta1 ~ dunif(-50,10)
10 beta2 ~ dunif(-50,10)
11 beta3 ~ dunif(-50,10)
12
13 sd ~ dunif(0.0001, 100)
14 })
```

There are really only 2 changes. The first is that instead of writing "model {", we write "NimbleCode({". Not too big a deal.

The other change is that normal distribution in NIMBLE can use standard deviation, precision or variance! So we have to tell NIMBLE which one we want. If we don't specify, it will assume precision.

As with JAGS, we need to provide parameters to monitor.

```
1 params <- c("beta0", "beta1", "beta2", "beta3","sd")
```

We also need to provide data. However, in NIMBLE we need to split constants (non-stochastic values) from "data". In this case, we only have one constant - the number of frogs (n.frogs) - and everything else is data.

```
1 constants <- list(n.frogs = 50)
2 data.frogs <- list(weight = Frogs$Weight, distance = Frogs$Dist, age =
    Frogs$Age, leg = Frogs$Leg)
```

We can also give NIMBLE initial values. NIMBLE doesn't like it if you provide initial values as a function, so we can just leave it as a list.

```
1 inits <- list(beta0 = runif(1, -10, 10), beta1 = runif(1,-10,10), beta2 =
    runif(1,-10,10), beta3 = runif(1,-10, 10), sd = runif(1,0, 100))
```

So now we can run the model! This requires more steps than JAGS, but this is something that can be really useful as you get more advanced with NIMBLE. It also helps with debugging.

In the background, NIMBLE is taking the code and sending it to C++. This allows NIMBLE to be super speedy, but this is why there are more steps than with JAGS. First we send the model code to NIMBLE.

```
1 prepfrogs <- nimbleModel(code = nimbleFrogs, constants = constants, data =
    data.frogs, inits = inits)
```

Next we ask the model if it needs any more info from us. Have we provided enough initial value?

```
1 prepfrogs$initializeInfo()
```

Yay, everything is initialized! We then ask NIMBLE to configure the MCMC, build the code for the samplers, and compile the model. For some of these steps it may say "this may take a minute" but don't be fooled - a minute can be anywhere from 1 minute to HOURS. Luckily this example should be fast!

```
1 mcmcfrogs <- configureMCMC(prepfrogs, monitors = params)
2 frogsMCMC <- buildMCMC(mcmcfrogs)
3 Cmodel <- compileNimble(prepfrogs)
4 Compfrogs <- compileNimble(frogsMCMC, project = prepfrogs)
```

Now we have compiled and are happy campers! It is time to finally run the iterations of the model.

```
Frog.mod.nimble <- runMCMC(Compfrogs, niter = 15000, thin = 1, nchains =3,
    nburnin = 1000, samplesAsCodaMCMC = TRUE)
```

This will run the chains one after the other then combine them for you for analysis. Note that unlike JAGS, the samples you get out = niter - nburnin. So in this case we're expected 14000 iterations back out. The "samplesAsCodaMCMC" just makes the output nice for later.

You can also run NIMBLE in parallel, via the "makeCluster" and "clusterExport" commands. Like so:

```
cl <- makeCluster(3)
clusterExport(cl = cl, varlist = c("constants", "data.frogs", "inits", "
   params", "nimbleFrogs"))
system.time(frog.out <- clusterEvalQ(cl = cl,{
  library(nimble) #you're now in a totally different environment so have
    to load the package again
  prepfrogs <- nimbleModel(code = nimbleFrogs, constants = constants,
                           data = data.frogs, inits = inits)
  prepfrogs$initializeInfo()
  mcmcfrogs <- configureMCMC(prepfrogs, monitors = params, print = T )
  frogsMCMC <- buildMCMC(mcmcfrogs) #actually build the code for those
    samplers
  Cmodel <- compileNimble(prepfrogs) #compiling the model itself in C++;
  Compfrogs <- compileNimble(frogsMCMC, project = prepfrogs) # compile the
    samplers next
  Frog.mod.nimble <- runMCMC(Compfrogs, niter = 15000, thin = 1, nburnin =
    1000, samplesAsCodaMCMC = TRUE)
}))
Frog.mod.nimble <- mcmc.list(frog.out)
stopCluster(cl)
```

This is less satisfying in that the output is suppressed until it's all done, but this is really helpful if you have something that takes a long time to run. I suggest doing all the running and debugging with just one chain first and then running it in parallel once you know everything is working properly.

# 7 NIMBLE Output

We can inspect the NIMBLE results using the coda package in R. As we did with JAGS, we can look at the summary of the chain(s):

```
summary(Frog.mod.nimble)
```

This gives us the following tables:

| | Mean | SD | Naive SE | Time-series SE |
|---|---|---|---|---|
| beta0 | -7.65 | 5.35 | 0.03 | 0.47 |
| beta1 | 0.12 | 0.00 | 0.00 | 0.00 |
| beta2 | -2.12 | 1.21 | 0.01 | 0.10 |
| beta3 | 0.17 | 0.01 | 0.00 | 0.00 |
| sd | 5.42 | 0.58 | 0.00 | 0.01 |

| | 2.5% | 25% | 50% | 75% | 97.5% |
|---|---|---|---|---|---|
| beta0 | -17.87 | -11.27 | -7.66 | -3.98 | 2.82 |
| beta1 | 0.12 | 0.12 | 0.12 | 0.12 | 0.13 |
| beta2 | -4.51 | -2.93 | -2.13 | -1.30 | 0.18 |
| beta3 | 0.14 | 0.16 | 0.17 | 0.18 | 0.19 |
| sd | 4.43 | 5.01 | 5.38 | 5.77 | 6.69 |

As we might expect, the Mean and SD are the mean and standard deviation for the value of each parameter in our output. The second table gives us the quantile spread (AKA the credible interval). So for our frogs, beta0 is likely between (-17.87, 2.82) with a mean value of (-7.65). We can also check our convergence based on the Gelman diagnostic function in the coda package.

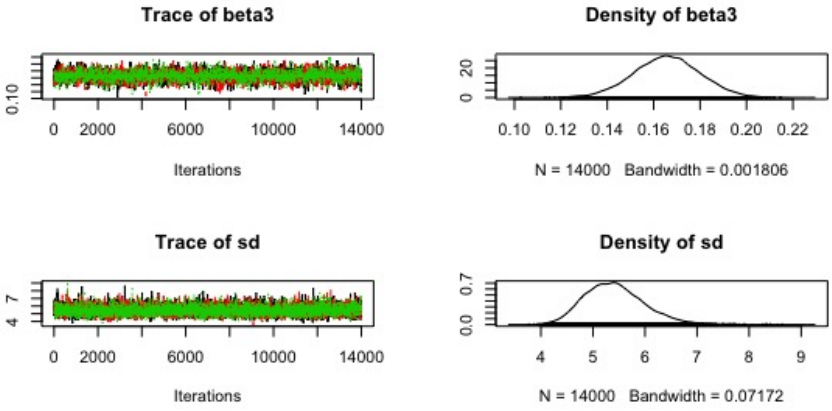
```
gelman.diag(Frog.mod.nimble)
```

| | Point est. | Upper C.I. |
|---|---|---|
| beta0 | 1.01 | 1.04 |
| beta1 | 1.00 | 1.01 |
| beta2 | 1.01 | 1.03 |
| beta3 | 1.00 | 1.01 |
| sd | 1.00 | 1.00 |

We want all these values to be below 1.1 and in this case they all are. You can also use gelman.plot to look at a visual version of this information. We don't really need to in this case, but it's something to try out if you're interested.

We can also plot our summary information.

```
plot(mcmc.list(Frog.mod.nimble))
```




The plot of our parameters shows us the chains on the left panels and a smoothed density plot of the values of the parameter from our chains on the right. We can see from our plot that the chains seem to have converged for our parameters of interest, just as the gelman diagnostic suggested.

We can now update our equation to reflect the mean parameter values we found:

$$E(weight) = -7.65 + 0.12A - 2.12L + .17D$$
$$Actualweight \sim Normal(\mu = E(weight), \sigma = 5.42)$$

And that's it for Linear Regression Part 1! Stayed tuned for Part 2 - categorical variables and Part 3 - graphing the results with credible intervals!

Feel free to email any questions or suggestions for future topics to heather.e.gaya(at)gmail.com or contact me on twitter @doofgradstudent !

Here's a bonus photo of my cat for making it this far: