Linear Regression 1

Heather Gaya

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(at)gmail.com or find me on twitter: doofgradstudent

Contents

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.

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	191	15.44	4.90	57.61	A
2	184	21.45	3.44	92.08	A
3	243	21.39	4.37	54.87	A
4	770	114.28	3.42	189.96	A
5	614	64.52	3.73	38.64	A
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 β_1 is the intercept, β_1 is "something", β_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 :)

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)
}
```

For those unfamiliar with for loops - this says "hey, for each from, from 1 to however many I have (AKA n.frogs), please take that frog's age, leg length and distance and put it into this equation to evaluate that frog's mean weight. Also, that mean weight is the mean of a normal distribution, so whatever value you choose for the betas, they have to make sense so that the data I give you makes sense! Thanks!"

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, τ, rather than standard deviation, σ. 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. This is just to get the MCMC chain working, it doesn't impact your final result 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 = 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. 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 = 3000, method = "parallel")
```

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. 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).

The other thing about MCMC is that it isn't just randomly wandering around the state space without any pattern at all. Each time it chooses a new possible value to test, that value is *near* the value it just had in the previous iteration. So naturally, between chains there's some autocorrelation. This is just important to remember for later, when we try to assess if the MCMC is doing what we want.

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!

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:

```
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Thu Oct 15 17:01:28 2020
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 5
##    Total graph size: 412
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## ++++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************** 100%
## . . . . . . Updating 3000
## -------------------------------------------------| 3000
## ************************************************** 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 5 variables....
## Finished running the simulation
```

	Lower95	Median	Upper95	Mean	SD	Mode	MCerr	MC%ofSD	SSeff	AC.10	psrf
beta0	-18.76	-7.93	2.92	-7.89	5.57	NA	0.65	11.7	73	0.85	1.04
beta1	0.12	0.12	0.13	0.12	0.00	NA	0.00	3.7	720	0.20	1.00
beta2	-4.62	-2.11	0.20	-2.10	1.23	NA	0.13	10.5	91	0.83	1.04
beta3	0.14	0.17	0.20	0.17	0.01	NA	0.00	2.9	1183	0.06	1.00
sd	4.37	5.40	6.61	5.44	0.59	NA	0.01	1.7	3409	0.04	1.00

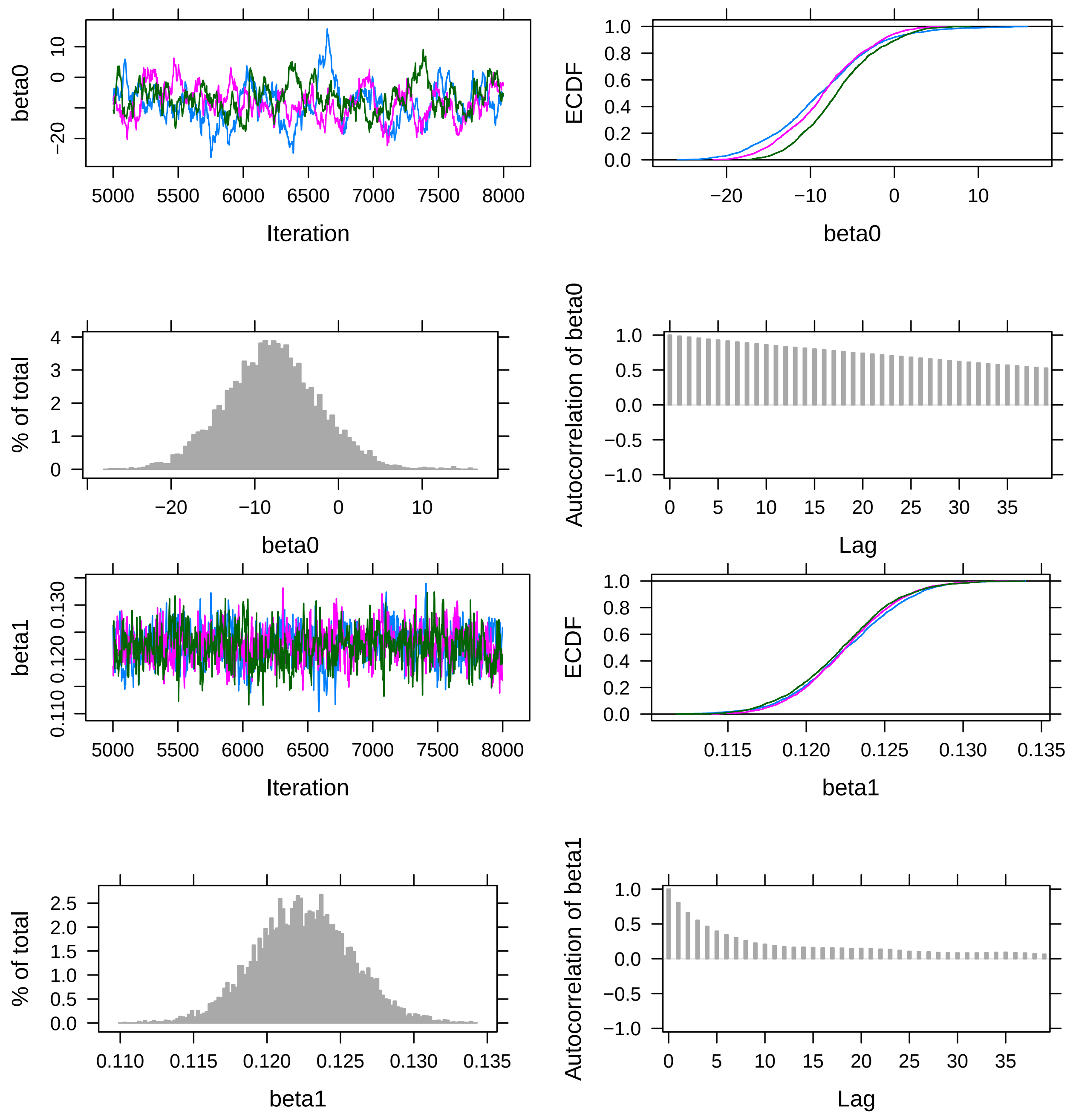
The first 3 columns are the credible interval. Unlike confidence intervals, credible intervals are easy to make sense of. 95% of our samples from our chains fell between the upper and lower bounds. The mean is the point estimate for that variable. SD is the standard deviation, the mode is the mode (blank for continuous variables).

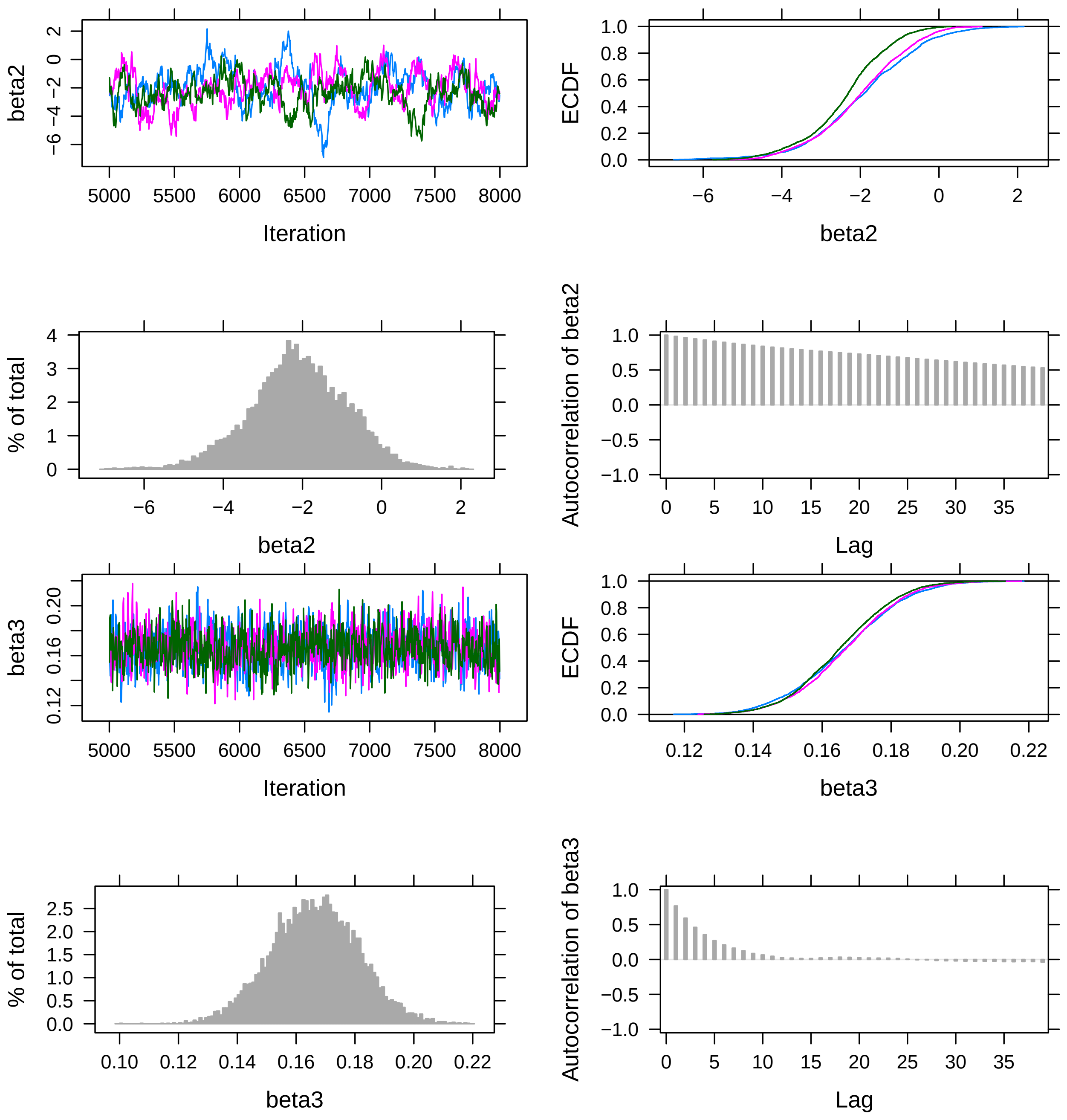
There is no Mode for any of our variables since all of them are continuous. MCerr refers to Monte Carlo error - an estimate of how much of the variation in your result is due to the method of getting your result (aka estimation via MCMC chains). Similarly, MC%ofSD tells you what percent of your standard deviation is due to Monte Carlo error. There's no exact metric for this, but generally you want it to be $< 5\%$. In our case, we see two of our variables have relatively high MC error. We'll look into that more in a minute.

AC.10 refers to the amount of auto-correlation between iterations of your MCMC chains and SSeff stands for the effective sample size of your run. In general, the farther away iterations from eachother, the less autocorrelation you want. If it drops off quickly, that means you're doing a good job of exploring your state space for likely solutions.

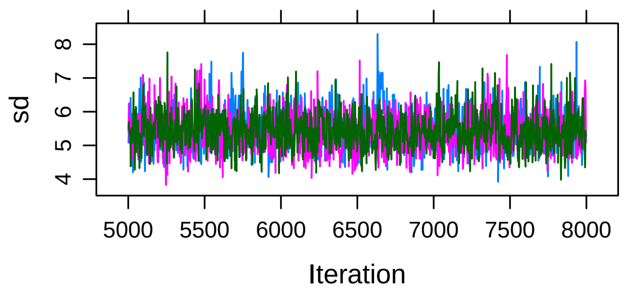
The other useful value to look at is "psrf" which is a statistic relating to convergence. This tells you about how different the chains are and is a proxy for convergence. Convergence is generally reached at values less than 1.1, but visual inspection of the chains is always important too! All our values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

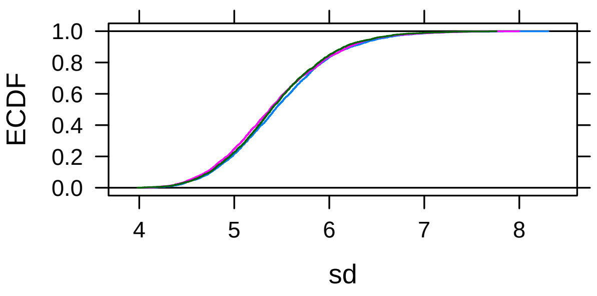
```
plot(Frog.mod)
## Generating plots...
```

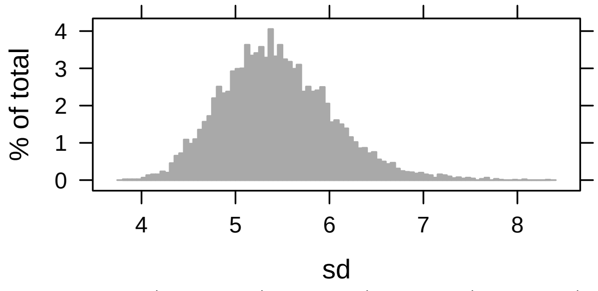

beta0
Iteration
ECDF
beta0
% of total
beta0
Autocorrelation of beta0
Lag
beta1
Iteration
ECDF
beta1
% of total
beta1
Autocorrelation of beta1
Lag

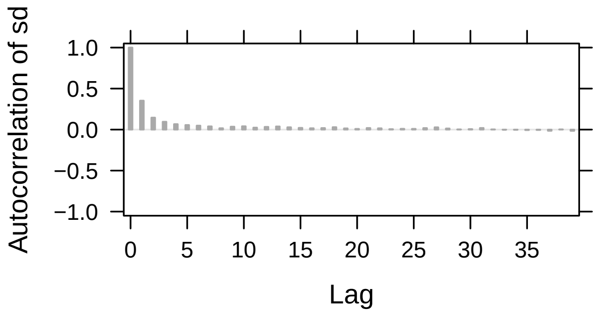


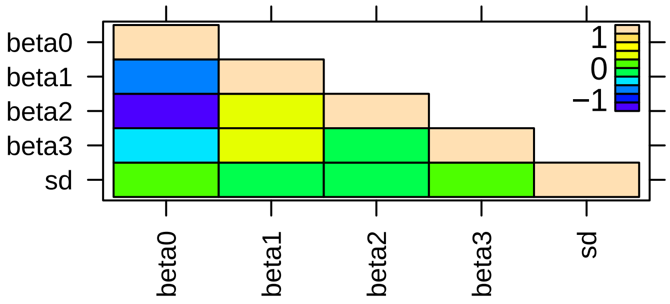
beta2
Iteration
ECDF
beta2
% of total
beta2
Autocorrelation of beta2
Lag
beta3
Iteration
ECDF
beta3
% of total
beta3
Autocorrelation of beta3
Lag











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. As previously mentioned, 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. In this case, our autocorrelation for beta0 tells us that we might need to run more chains to get a clear answer of this value. In a real analysis, that level of autocorrelation would be a little worrying.

These panels are repeated for each variable we monitored in our “params” argument.

The final plot represents correlation. Generally you want to avoid correlated variables if at all possible. We can see in this case that beta0 and beta2 have a correlation of -1. That’s NOT what we want. In a real analysis, we’d want to find a way to avoid having both of those variables at the same time - either remove distance from the road or the intercept or find some other variables to model. In this case, the data is completely made up so it doesn’t really matter.

Putting The Equation Together

So now we’ve looked at all our chains and the summary stats and most of it looked okay… mostly. If we’re satisfied with our answer, we can now update our equation to reflect the mean parameter values we found:

$$E(weight) = -7.89 + 0.12A - -2.1L + 0.17D$$
$$Actualweight \sim Normal(\mu = E(weight), \sigma = 5.44)$$

`~ ta-dah! ~`

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). Check out Part 2 on my website: categorical variables and graphing the results with credible intervals!

NIMBLE Model

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:

```r
library(nimble)
## Warning: package 'nimble' was built under R version 3.6.2
## nimble version 0.9.1 is loaded.
## For more information on NIMBLE and a User Manual,
## please visit http://R-nimble.org.
##
## Attaching package: 'nimble'
## The following object is masked from 'package:stats':
##
##     simulate
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
}

beta0 ~ dunif(-50,10)
beta1 ~ dunif(-50,10)
beta2 ~ dunif(-50,10)
beta3 ~ dunif(-50,10)

sd ~ dunif(0.0001, 100)
})
```

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.

```
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.

```
constants <- list(n.frogs = 50)
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 seem to like it if you provide initial values as a function, so we can just leave it as a list.

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

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. There are two ways to send code to NIMBLE. Option 1 is great for debugging and modifications/customization of the MCMC. Option 2 is great if you don't want to think about anything and just want NIMBLE to do its own thing.

Running NIMBLE Option 1

First we send the model code to NIMBLE.

```
prepfrogs <- nimbleModel(code = nimbleFrogs, constants = constants, data = data.frogs, inits = inits)
## defining model...
## building model...
## setting data and initial values...
## running calculate on model (any error reports that follow may simply reflect missing values in model
## checking model sizes and dimensions...
## model building finished.
```

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

```
prepfrogs$initializeInfo()
## All model variables are initialized.
```

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!

```
mcmcfrogs <- configureMCMC(prepfrogs, monitors = params)
## ===== Monitors =====
## thin = 1: beta0, beta1, beta2, beta3, sd
## ===== Samplers =====
## RW sampler (5)
##   - beta0
##   - beta1
##   - beta2
##   - beta3
##   - sd
frogsMCMC <- buildMCMC(mcmcfrogs)
Cmodel <- compileNimble(prepfrogs)
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
Compfrogs <- compileNimble(frogsMCMC, project = prepfrogs)
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
```

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)
## running chain 1...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 2...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 3...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

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)

summary(frog.out)
```

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.

Fast Way to Run NIMBLE

If you don't want to do each step of the above individually, you can also use the "basic" version of NIMBLE, which works just as well.

```r
Frog.mod.nimble <- nimbleMCMC(code = nimbleFrogs,
                   constants = constants,
                   data = data, inits = inits, monitors = params,
                   niter = 15000, thin = 1, nchains =3,
                   nburnin = 1000, samplesAsCodaMCMC = TRUE)
```

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):

	Mean	SD	Naive SE	Time-series SE
beta0	-7.80	5.54	0.03	0.49
beta1	0.12	0.00	0.00	0.00
beta2	-2.11	1.24	0.01	0.11
beta3	0.17	0.01	0.00	0.00
sd	5.44	0.59	0.00	0.01

	2.5%	25%	50%	75%	97.5%
beta0	-18.96	-11.41	-7.71	-4.15	2.85
beta1	0.12	0.12	0.12	0.12	0.13
beta2	-4.46	-2.93	-2.14	-1.30	0.43
beta3	0.14	0.16	0.17	0.18	0.20
sd	4.42	5.03	5.39	5.81	6.72

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 (-18.96, 2.85) with a mean value of (-7.71). We can also check our convergence based on the Gelman diagnostic function in the coda package.

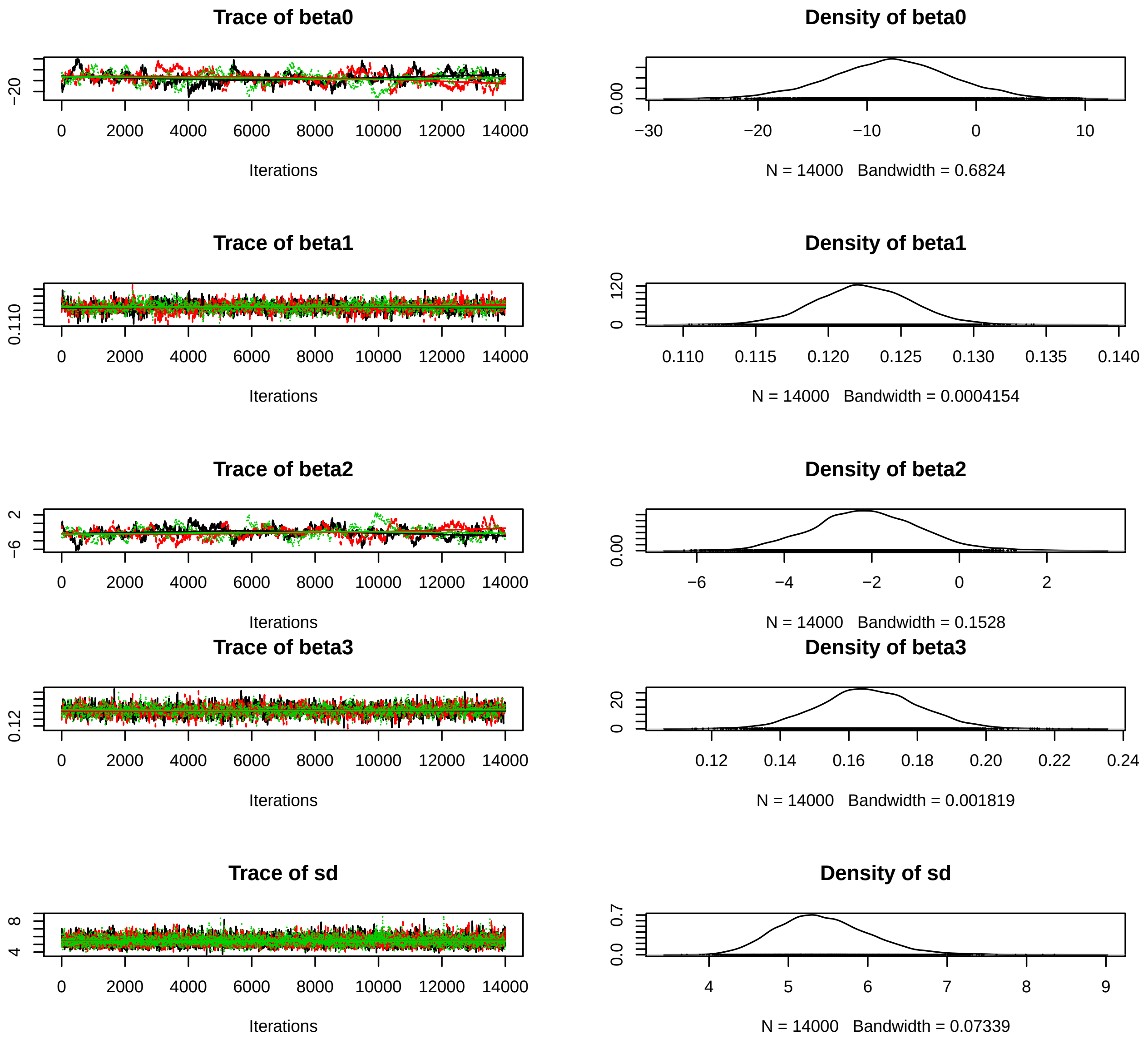
```r
gelman.diag(Frog.mod.nimble)
## Potential scale reduction factors:
##
##       Point est. Upper C.I.
## beta0       1.04       1.12
## beta1       1.01       1.03
## beta2       1.04       1.11
## beta3       1.00       1.00
## sd          1.00       1.00
##
## Multivariate psrf
##
## 1.03
```

In general, we want all these values to be below 1.1 and in this case they all are. This isn't a perfect diagnostic, but it's a good rule of thumb when combined with visual inspection of the chains. 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 just for funsies.

```r
gelman.plot(Frog.mod.nimble)
```

We can also plot our summary information, just like we did with JAGS.

```r
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, just like with JAGS.

$$E(weight) = -7.71 + 0.12A + `round(rsummary(Frog.mod.nimble)[[2]][3,3], digits = 2)`L + 0.17D$$
$$Actual weight \sim Normal(\mu = E(weight), \sigma = 5.39)$$

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