Simple Regression in NIMBLE

Adam B. Smith

2023-09-26

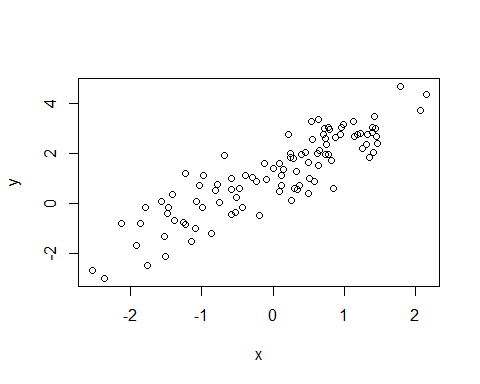
This tutorial walks you through how to do a simple Bayesian linear regression using the **nimble** package for **R**. linear regression using the nimble package for R. It also explores basic model diagnostics. We’ll try to estimate the “true” values we use to generate the data using simple linear regression. We will also explores basic model diagnostics.

### Setup

library(nimble) # the workhorse  
library(coda) # for model diagnostics  
library(bayesplot) # for graphing

## Create some fake data

N <- 100 # number of data points  
x <- rnorm(N) # predictor  
beta0 <- 1 # intercept  
beta1 <- 1.2 # slope  
sigma <- 0.8 # variation around trendline  
y <- beta0 + beta1 \* x + rnorm(N, 0, sigma)  
  
plot(x, y)



## Organize data into formats needed by nimble

Data is supplied to nimble using lists:

* “Data” list: Supplies values that were measured. The distinction between data and “constants” (the next list), is that data have distributions so, in theory, could have been different values (i.e., if you were to repeat your sampling again).
* “Constants” list: This list has values that do not change. For example, the number of data points or number of statistical groups is a constant. It can also include predictor variables (which are assumed to be “fixed”.) However, in some models, even the predictors are assumed to have error, so have distributions and therefore should go into the “data” list.
* “Initialization” list: This list is optional, but **nimble** works much more reliably if you provide some initial starting values for each parameter.

data <- list(y = y)  
  
constants <- list(  
 x = x,  
 N = N  
)  
  
# note use of "\_hat"  
inits <- list(  
 beta0\_hat = 0,  
 beta1\_hat = 0,  
 sigma\_hat = 1  
)

In the equations below, we’ll be using the variables beta0\_hat, beta1\_hat, and sigma\_hat to distingish them from the values we defined to simulate “reality” in the steps above.

## What we’re about to do

You are supremely familiar with the slope-intercept formula for a line:

Assume that we’ve constructed a line in the plane defined by x and y. Now, the equation for the location of data point i is:

where is assumed to be a value drawn from a normal distribution with a mean of 0 and a standard deviation of . This means that the data has a mean of , and an “error” around this of (a normal distribution with a mean of 0 and standard deviation of ).

This is the “standard” way of thinking about regression. In Bayes world, we think about the distribution first, then the properties it has. In other words, we can think of a regression line as a normal distribution with a mean given by :

So, for a given data point i, we can calculate its likelihood as:

So, in nimble, we’ll be writing the likelihood of each data point as

y[i] ~ dnorm(beta0\_hat + beta1\_hat \* x[i], sigma\_hat)

which is the nimble equivalent of the equation immediately above this.

In practice, we often split this up so it reads more easily (which can be important for complex models):

y[i] ~ dnorm(y\_mean[i], sigma\_hat)

y\_mean[i] <- beta0\_hat + beta1\_hat \* x[i]

Note also that we use ~ for indicating a value “is distributed as” and <- for when something is not. When do we use ~? Simply put, when something occurs in the “data” list, it can only appear on the left-hand side of an equation, and it can only be followed by a ~, then one of the d...() functions like dnorm().

## Writing the nimble model

Now, we’ll write out our nimble model. The code consists of a statement about the likelihood of each datum, plus the prior distributions for each parameter. These include beta0\_hat, beta1\_hat, and sigma\_hat.

A very funny thing about this kind of code is that the order in which things are written does not matter… the software knows where to go to get what it needs. In other words, we could write:

y[i] ~ dnorm(y\_mean[i], sigma\_hat)

y\_mean[i] <- beta0\_hat + beta1\_hat \* x[i]

or

y\_mean[i] <- beta0\_hat + beta1\_hat \* x[i]

y[i] ~ dnorm(y\_mean[i], sigma\_hat)

and the result would be the same. For clarity though, it is usually best to keep “like” things together (e.g., likelihoods for data points together and likelihoods for priors together).

We write the model inside a function called nimbleCode().

code <- nimbleCode({  
  
 # likelihood of each datum  
 for (i in 1:N) {  
  
 y[i] ~ dnorm(y\_mean[i], sigma\_hat)  
 y\_mean[i] <- beta0\_hat + beta1\_hat \* x[i]  
  
 }  
  
 # likelihoods of priors  
 # beta0\_hat and beta1\_hat: "broad" normals  
 # important: use argument "sd"... don't leave as blank!  
 # sigma\_hat: uniform >0 (after Gelman 2006 Bayesian Analysis)  
  
 beta0\_hat ~ dnorm(0, sd = 10)  
 beta1\_hat ~ dnorm(0, sd = 10)  
 sigma\_hat ~ dunif(0, 10)  
  
})

## Process the model code

This step checks the code for errors and does other arcane things. Note that this is where we supply the three lists we created.

model <- nimbleModel(  
 code = code, # our model  
 constants = constants, # constants  
 data = data, # data  
 inits = inits, # initialization values  
 check = TRUE, # any errors?  
 calculate = FALSE  
)

## Defining model

## Building model

## Setting data and initial values

## Checking model sizes and dimensions

## [Note] This model is not fully initialized. This is not an error.  
## To see which variables are not initialized, use model$initializeInfo().  
## For more information on model initialization, see help(modelInitialization).

## Checking model calculations

You probably got a warning, “This model is not fully initialized. . . To see which variables are not initialized, use model$initializeInfo().” So, let’s do that.

model$initializeInfo()

## [Note] All model variables are initialized.

Welcome to Bayesland.

## Checking for errors

This next step is optional, but it can help to check for errors before running the model. This is useful because it can take a long time to “run” the model before an error stops it.

model$calculate()

## [1] -299.5412

This is the log-likelihood of the model. We would have received something else (i.e., a non-numerical value), plus maybe messages, had there been errors.

## Creating the MCMC confugration for the model

We now create an MCMC “configuration” for the model. This creates a sampler, which we can later adjust to get more stable results.

We also need to tell it what variables to keep track of. In our case, we are interested in estimates of beta0\_hat, beta1\_hat and sigma\_hat. Keeping track of a parameter uses memory, and in some cases we use intermediate variables in which we are not interested. An example of this is our y\_mean value in the code.

monitors <- c('beta0\_hat', 'beta1\_hat', 'sigma\_hat')  
  
conf <- configureMCMC(  
 model,  
 monitors = monitors,  
 print = TRUE,  
 enableWAIC = FALSE # Watanabe's AIC... useful, but takes time  
)

## ===== Monitors =====  
## thin = 1: beta0\_hat, beta1\_hat, sigma\_hat  
## ===== Samplers =====  
## RW sampler (1)  
## - sigma\_hat  
## conjugate sampler (2)  
## - beta0\_hat  
## - beta1\_hat

The output indicates we are monitoring the variables we said to monitor. It also indicates that a random walk (RW) sampler is used for sigma\_hat, and conjugate samplers for the betas.

If we wanted to change samplers, we could do that in this step. But for this simple problem, these are good. We would want to change samplers to make model estimates more stable, or to make the MCMC run more efficiently.

Lastly, we must “build” and compile the model. Compiling the model creates code in C, which runs much faster than R code. However, compiling can take a long time (hours). So, if you are developing a model and want to simply test that it works, it can help to subset your data (e.g., use the first ten data points) to reduce the compilation time.

build <- buildMCMC(conf)  
compiled <- compileNimble(model, build, showCompilerOutput = FALSE)

## Compiling  
## [Note] This may take a minute.  
## [Note] Use 'showCompilerOutput = TRUE' to see C++ compilation details.

## Run the MCMC sampler for the model

The last step is running the MCMC sampler using runMCMC(). This actually “implements” the model (estimates coefficients). We need to provide the function:

* The total number of MCMC iterations (we’ll use 1100 because the problem of fitting a regression line to linear data is simple–in some cases you may need to use many thousands or even millions).
* The number of burn-in iterations to exclude before “recording” MCMC samples begins (100 in this example–the number of iterations includes the burn-in, so in our case we’ll get 1000 estimates of each coeffiecnt in our final output).
* The chains, or number of times to run the model. We’ll use 4, which is fairly standard (you could go down to 2 in special cases, and usually more than 4 is overkill).
* The “thinning” rate, or how often to record coefficient estimates across MCMC iterations. The default is 1, meaning we record each MCMC step. But if our chains mix slowly, or coefficients are correlated, recording each step takes a lot of memory for not a lot of new information gained. So we might want to increase the thinning rate so as not to include so many values. To get a sufficent sample, we would also need then to increase the total number of iterations and burn-in iterations.

chains <- runMCMC(  
 compiled$build,  
 niter = 1100,  
 nburnin = 100,  
 thin = 1,  
 nchains = 4,  
 inits = inits, # initialization values  
 progressBar = TRUE,  
 samplesAsCodaMCMC = TRUE, # for using coda packagge for plots  
 summary = TRUE, # calculate summaries across chains  
 WAIC = FALSE, # Watanabe's AIC... useful, but takes time  
 perChainWAIC = FALSE  
)

## running chain 1...

## |-------------|-------------|-------------|-------------|  
## |-------------------------------------------------------|

## running chain 2...

## |-------------|-------------|-------------|-------------|  
## |-------------------------------------------------------|

## running chain 3...

## |-------------|-------------|-------------|-------------|  
## |-------------------------------------------------------|

## running chain 4...

## |-------------|-------------|-------------|-------------|  
## |-------------------------------------------------------|

What did we get? The output of runMCMC() is a list object. We can peer at this list using str():

str(chains, 2)

## List of 2  
## $ samples:List of 4  
## ..$ chain1: 'mcmc' num [1:1000, 1:3] 1.07 1.13 1.12 1.22 1.21 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## .. ..- attr(\*, "mcpar")= num [1:3] 1 1000 1  
## ..$ chain2: 'mcmc' num [1:1000, 1:3] 1.11 1.24 1.11 1.18 1.17 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## .. ..- attr(\*, "mcpar")= num [1:3] 1 1000 1  
## ..$ chain3: 'mcmc' num [1:1000, 1:3] 1.24 1.29 1.31 1.1 1.34 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## .. ..- attr(\*, "mcpar")= num [1:3] 1 1000 1  
## ..$ chain4: 'mcmc' num [1:1000, 1:3] 1.23 1.05 1 1.21 1.14 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## .. ..- attr(\*, "mcpar")= num [1:3] 1 1000 1  
## ..- attr(\*, "class")= chr "mcmc.list"  
## $ summary:List of 5  
## ..$ chain1 : num [1:3, 1:5] 1.17 1.3 1.81 1.17 1.3 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## ..$ chain2 : num [1:3, 1:5] 1.17 1.3 1.76 1.17 1.3 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## ..$ chain3 : num [1:3, 1:5] 1.18 1.3 1.78 1.17 1.3 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## ..$ chain4 : num [1:3, 1:5] 1.17 1.3 1.78 1.17 1.3 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## ..$ all.chains: num [1:3, 1:5] 1.17 1.3 1.78 1.17 1.3 ...  
## .. ..- attr(\*, "dimnames")=List of 2

This list has two main components. The first is named “samples”, and it has one item per chain we ran (4 chains, in our example). Each chain is a matrix with 1000 rows (one per MCMC iteration), and three columns (one each for beta0\_hat, beta1\_hat, and sigma\_hat).

The second main component of the output is names summary, and it reports the mean, standard deviation, etc. for each of the coefficients for each chain, then for all chains together. This last summary is called all.chains in the object. Let’s take a look.

chains$summary$all.chains

## Mean Median St.Dev. 95%CI\_low 95%CI\_upp  
## beta0\_hat 1.172488 1.173910 0.07532162 1.019610 1.320766  
## beta1\_hat 1.299998 1.298844 0.07105920 1.162294 1.443332  
## sigma\_hat 1.779987 1.767955 0.24862549 1.320937 2.305369

These are our estimates for the parameters of interest. Usually, we use the “Mean” to report central tendency, and the lower and upper 95% confidene interval bounds to report uncertainty. How well did we do?

beta0

## [1] 1

beta1

## [1] 1.2

sigma

## [1] 0.8

When I ran my model, I got good estimates of beta0 and beta1, but the real value of sigma, 0.8, was outside the estimated 95% CI. I could improve this by, for example, increasing the number of MCMC iterations, using a different prior for sigma\_hat, or changing the sampler.

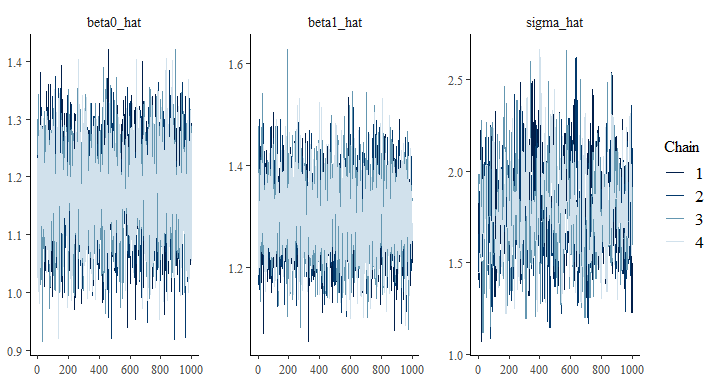
# Model diagnostics

It’s *always* a good idea to examine model diagnostics. We’ll do that here graphically using the **coda** and **bayesplot** packages.

## Chain mixing and convergence

Before we look at the parameter estimates, we should see how well the MCMC sampler actually sampled the available parameter space and if it tended to converge between and within the 4 chains. Chains should “mix” well, meaning they should sample the same portions of parameter space. To assess that, we use “trace” plots, which graph the estimated value of each coefficient for each MCMC iteration. What we want to see is what’s been described as “grass, as seen from a bug’s perspective”.

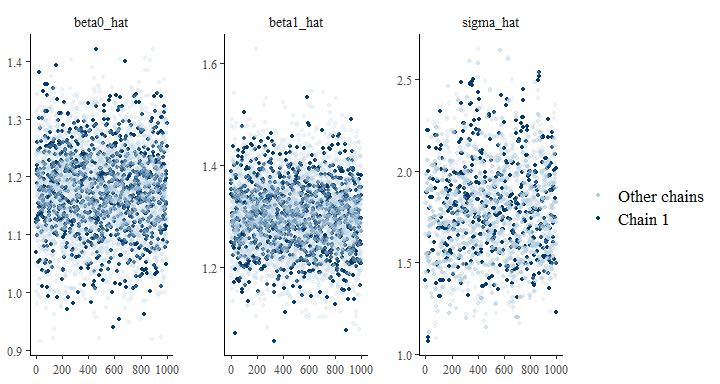
mcmc\_trace(chains$samples)



These chains look fairly good. But (at least on my case), you can see the one for sigma\_hat looks a little different than the ones for the betas. Peering at it, you can see that the chains mix a bit more slowly. This indicates that the coefficient estimates in each chain are highly correlated with values immediately preceeding them. The chain is mixing more slowly as a result. The problem with this is that the chain can linger is a particular part of parameter space for too long of a time, providing biased estimates of parameters.

This issue is better highlighted using the mcmc\_trace\_highlight() method of plotting chains.

mcmc\_trace\_highlight(chains$samples, highlight =1)



This graph highlights estimates from chain 1 versus the other chains. You can readily see that this chain looks less well-mixed for sigma\_hat compared to the betas. The same holds for the other chains (not shown–but you can show yourself!).

When you have a lot of coefficients, examining their trace plots can be tedious (and subjective). Alternatively, you can calculate the “Gelman-Rubin” statistic, which is also called (“R-hat”). measures the ratio of variation among the chains. It’s ideal value for a well-mixed model is 1, an in practice, values of <1.1 are acceptible. The statistic is almost always reported in publications using MCMC.

gelman.diag(chains$samples)

## Potential scale reduction factors:  
##   
## Point est. Upper C.I.  
## beta0\_hat 1 1  
## beta1\_hat 1 1  
## sigma\_hat 1 1  
##   
## Multivariate psrf  
##   
## 1

In my case, the values are all very close to 1, suggesting that the chains mixed well. This is a little surprising given what we’ve seen, but it underscores the fact that relying on just one diagnostic metric may be unreliable. Part of the “problem” is that measures how far apart the chains started from where they ended up. We started with the same initialization values for each chain, and they were not too far from their end values, so the statistic was “doomed” to be close to 1.

So, in retrospect, we should have chosen different initialization values for each chain. However, I’d advise against selecting widly different (implausible) values… it can cause chains to “run off” from the pack, leaving your model even more unstable. This would be evident if one of the chains in the traceplot wandered off on its own.

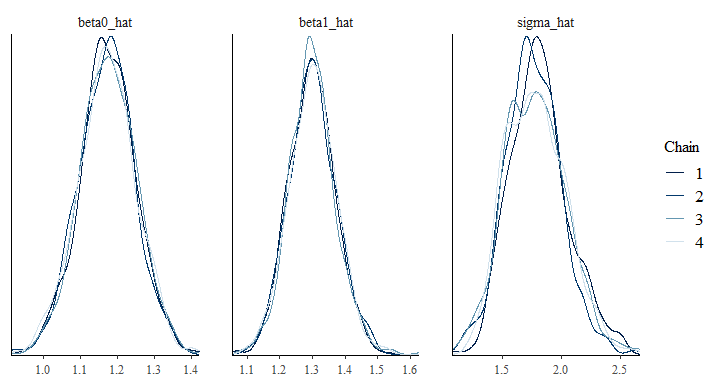
Each chain should provide an independent estimate of coefficients. If you see that a chain has a wildly different density, there may be something amiss. You can often fix this by running the model again (it’s stochastic, after all!), using different initialization values, or running it for more iterations. If failure to mix is persistent, you may have a dimensionality/identifability issue and so need to reformulate your model (i.e., maybe use fewer coefficients or reduce correlation between predictors).

Back in reality, we would probably want to run our model again for more iterations to stabilize parameter estimates. But we’ll continue with this model for expediency.

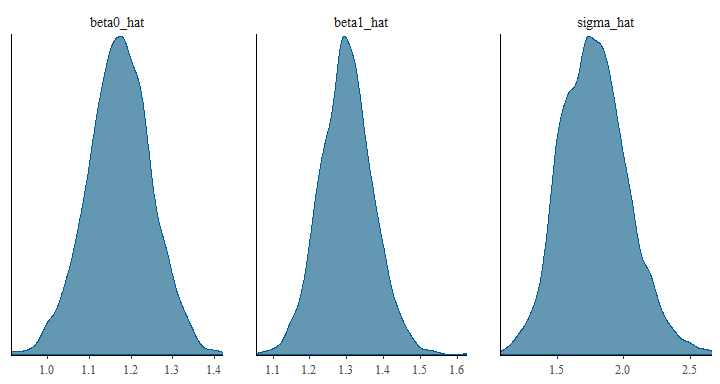
## Parameter estimates

Let’s examine the posterior estimates of our model coefficients.

# densities of estimates by chain  
mcmc\_dens\_overlay(chains$samples)

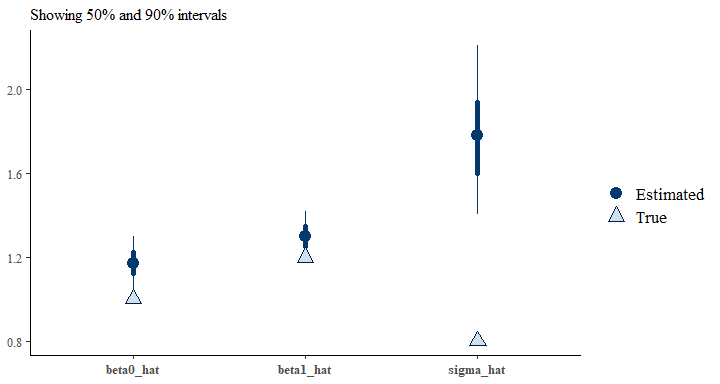


# densities across all chains  
mcmc\_dens(chains$samples)



We can also graphically compare the posterior distributions to their “true” values.

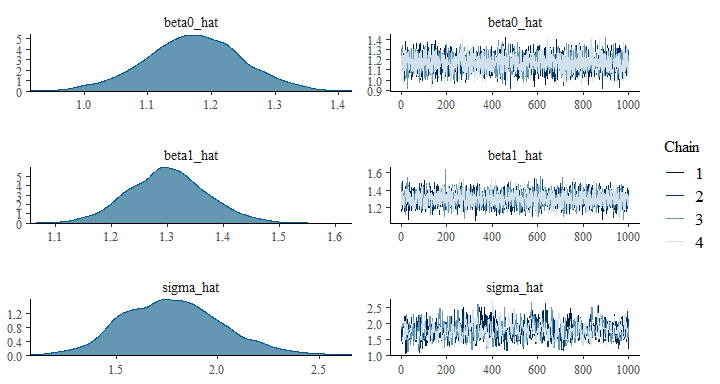
truth <- c(beta0, beta1, sigma)  
mcmc\_recover\_intervals(chains$samples, true = truth, point\_est = 'mean')



Here, thick part of each line represents the inner 50%CI, and the thin part the inner 90% CI. Circles represent the mean across the posterior. You can see In my case, we’re OK at estimating beta0, very good at estimating beta1, and poor at sigma.

Finally, we can efficiently exmaine trace plots and posterior densities using “combo” plots:

mcmc\_combo(chains$samples)



### The easier way

Note that we could have gotten a regression simply by using lm() to do a “standard” linear regression:

model <- lm(y ~ x)  
coefficients(model)

## (Intercept) x   
## 1.170818 1.300619

sigma(model)

## [1] 0.7573939

Note that the estimates for the betas are nearly the same, and that it got sigma correct! It also finishes faster. Later, we will see how Bayesian models can be better than “standard” frequentist models when data and models are complex. However, please keep in mind that life is too short to always be Bayesian.

*Finis!*