Generalized Linear Regression in NIMBLE

Adam B. Smith

2023-10-13

This tutorial walks you through how to do generalized linear regression. Specifically, we will explore logistic regression (where the response is 1s and 0s). We’ll use simulated data for which we know the “true” answers. We’ll also compare a “correct” model that uses the form of the function we used to generate the data, and an “incorrect” model that uses a spurious, extra term.

Again, the [online manual](https://r-nimble.org/html_manual/cha-welcome-nimble.html) for nimble is very useful. The [main webpage](https://r-nimble.org/) for nimble also links to a host of useful information.

## Setup

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

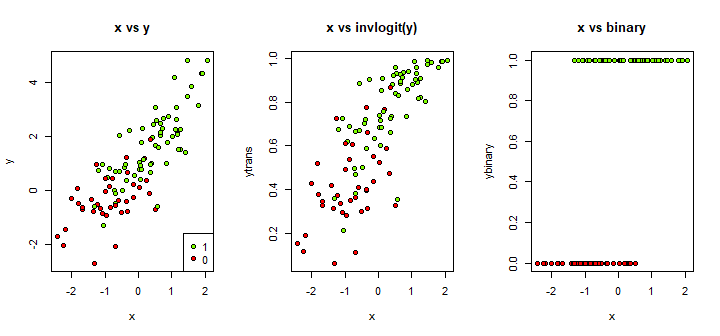
## Create some fake data

We need to create a response vector that has just 1s and 0s. We’ll do this by simulating some data, y, that responds linearly to x. Values of y will follow a normal distribution. We’ll then use the inverse logit transform to convert y to the range (0, 1) (like a probability). Finally, we’ll impose a probabilistic threshold to convert these values to 1s and 0s.

N <- 100 # number of data points  
x <- rnorm(N) # predictor  
beta0 <- 1 # intercept  
beta1 <- 1.2 # slope  
sigma <- 0.8 # variation around trendline  
  
noise <- rnorm(N, mean = 0, sd = sigma)  
y <- beta0 + beta1 \* x + noise  
  
# function for inverse logit  
invlogit <- function(x) exp(x) / (1 + exp(x))  
  
# convert to range of (0, 1)  
ytrans <- invlogit(y)  
  
# convert to 0/1 probabilistically  
ybinary <- as.integer(runif(N) < ytrans)

Let’s look at the data through the transformation process.

par(mfrow = c(1, 3))  
  
# x vs y  
bg <- ifelse(ybinary == 1, "chartreuse", "red")  
plot(x, y, pch = 21, bg = bg, main = 'x vs y')  
  
legend('bottomright', legend = c(1, 0), pt.bg = c('chartreuse', 'red'), pch = 21)  
  
# x vs inverse-logit transform of y  
plot(x, ytrans, pch = 21, bg = bg, main = 'x vs invlogit(y)')  
  
# x vs binary y  
plot(x, ybinary, pch = 21, bg = bg, main = 'x vs binary')



Finally, we’ll rescale our predictor (x) to have a 0 mean and unit variance.

x\_scaled <- scale(x) # output is a 1-column matrix  
x\_scaled <- c(x\_scaled) # convert back to a vector

## Organize data into formats needed by nimble

data <- list(ybinary = ybinary)  
  
constants <- list(  
 x\_scaled = x\_scaled,  
 N = N  
)  
  
# two sets of initialization values, one for correct and one for incorrect model  
inits\_correct <- list(  
 beta0\_hat = 0,  
 beta1\_hat = 0,  
 y\_est = ybinary  
)  
  
inits\_incorrect <- list(  
 beta0\_hat = 0,  
 beta1\_hat = 0,  
 beta2\_hat = -1,  
 y\_est = ybinary  
)

## What we’re about to do

If we were interested in the relationship between y and x (neither of them transformed), we could use a linear model like this:

where, together, all the s for all follow a normal distribution with a mean of 0 and standard deviation of . Ergo,

.

We’ve learned that this can be written like:

meaning thaty has a normal distribution that has a mean given by the estimated regression line.

For binary (logistic) regression, though, we have 0s and 1s, which cannot be described by a normal distribution. Rather, we use the Bernoulli distribution, which returns 0 or 1, and takes as an argument , the probability that the value is 1. A Bernoulli distribution is essentially a single coin toss that returns heads or tails (or, equivalently, a binomial distribution with just one trial). So, we have:

We can now write an equation for , as a function of things we think drive the probability of getting a 1 or 0.

Note that: \* We use the logit() function to transform , which has the range (0, 1) to unbounded values (i.e., between negative and positive infinity), just like a normal distribution.

* The right hand side is our linear regression equation from before. We’re positing that the probability of getting a 1 or 0 depends on x and the coefficients.
* We don’t include the part in the linear predictor. That’s because the randomness is taken account of by the distribution function.

NIMBLE has functions for the Bernoulli distribution and for the logit transform.

## Writing the nimble model

In addition to specifying our model, we will also ask NIMBLE to do a little extra, interesting work to help us assess which of our models is the correct one. Let’s say we want to find the mean value of ybinary. This is easy to do using mean(ybinary), but let’s say we want to calculate the mean of the values of ybinary that NIMBLE *thinks* the values should be, based on its model of ybinary. We can then compare mean(ybinary) with NIMBLE’s estimate of the mean of ybinary.

This is a kind of check we can do to see if we correctly specified the model. There are much more sophisticated ways of doing this (e.g., simulating all the ybinarys).

Below, the first model is “correct” because it says that the relationship between y (untransformed) is just a linear function of x. The second one is incorrect, because it adds a squared term which we did not include when we generated the data.

code\_correct <- nimbleCode({  
  
 # likelihood of each datum  
 for (i in 1:N) {  
  
 # note the "sd = " part!  
 ybinary[i] ~ dbern(psi[i])  
 logit(psi[i]) <- beta0\_hat + beta1\_hat \* x\_scaled[i]  
  
 }  
  
 # likelihoods of priors  
 # beta0\_hat and beta1\_hat: "broad" normals  
 # sigma\_hat: uniform >0 (after Gelman 2006 Bayesian Analysis)  
  
 beta0\_hat ~ dnorm(0, sd = 10)  
 beta1\_hat ~ dnorm(0, sd = 10)  
  
 # simulate the data and calculate an auxillary variable  
 # this does not affect the likelihood  
 for (j in 1:N) {  
 y\_est[j] ~ dbern(psi[j])  
 }  
 y\_est\_mean <- mean(y\_est[1:N])  
  
})  
  
code\_incorrect <- nimbleCode({  
  
 # likelihood of each datum  
 for (i in 1:N) {  
 # note the "sd = " part!  
 ybinary[i] ~ dbern(psi[i])  
 logit(psi[i]) <- beta0\_hat + beta1\_hat \* x\_scaled[i] + beta2\_hat \* (x\_scaled[i])^2  
 }  
  
 # likelihoods of priors  
 # beta0\_hat and beta1\_hat: "broad" normals  
 # sigma\_hat: uniform >0 (after Gelman 2006 Bayesian Analysis)  
  
 beta0\_hat ~ dnorm(0, sd = 10)  
 beta1\_hat ~ dnorm(0, sd = 10)  
 beta2\_hat ~ dnorm(0, sd = 10)  
  
 # simulate the data and calculate an auxillary variable  
 # this does not affect the likelihood  
 for (j in 1:N) {  
 y\_est[j] ~ dbern(psi[j])  
 }  
 y\_est\_mean <- mean(y\_est[1:N])  
  
})

## Process the model code

First, check the code for errors and do some other arcane things.

# correct model  
model\_correct <- nimbleModel(  
 code = code\_correct, # our model  
 constants = constants, # constants  
 data = data, # data  
 inits = inits\_correct, # 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

model\_correct$initializeInfo()

## [Note] All model variables are initialized.

model\_correct$calculate()

## [1] -145.0725

# incorrect model  
model\_incorrect <- nimbleModel(  
 code = code\_incorrect, # our model  
 constants = constants, # constants  
 data = data, # data  
 inits = inits\_incorrect, # 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

model\_incorrect$initializeInfo()

## [Note] All model variables are initialized.

model\_incorrect$calculate()

## [1] -204.5082

Note that the starting log-likelihood (given the initialization values we used) is smaller for the incorrect model, which is telling.

We now create an MCMC “configuration” for the model. We also need to tell it what variables to keep track of.

# corect model  
monitors\_correct <- c('beta0\_hat', 'beta1\_hat', 'y\_est\_mean')  
  
conf\_correct <- configureMCMC(  
 model\_correct,  
 monitors = monitors\_correct,  
 print = TRUE,  
 enableWAIC = FALSE # Watanabe's AIC  
)

## ===== Monitors =====  
## thin = 1: beta0\_hat, beta1\_hat, y\_est\_mean  
## ===== Samplers =====  
## RW sampler (2)  
## - beta0\_hat  
## - beta1\_hat  
## posterior\_predictive sampler (100)  
## - y\_est[] (100 elements)

# incorrect model  
monitors\_incorrect <- c('beta0\_hat', 'beta1\_hat', 'beta2\_hat', 'y\_est\_mean')  
  
conf\_incorrect <- configureMCMC(  
 model\_incorrect,  
 monitors = monitors\_incorrect,  
 print = TRUE,  
 enableWAIC = FALSE # Watanabe's AIC  
)

## ===== Monitors =====  
## thin = 1: beta0\_hat, beta1\_hat, beta2\_hat, y\_est\_mean  
## ===== Samplers =====  
## RW sampler (3)  
## - beta0\_hat  
## - beta1\_hat  
## - beta2\_hat  
## posterior\_predictive sampler (100)  
## - y\_est[] (100 elements)

Lastly, we “build” and compile the model.

build\_correct <- buildMCMC(conf\_correct)  
build\_incorrect <- buildMCMC(conf\_incorrect)  
  
compiled\_correct <- compileNimble(model\_correct, build\_correct)

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

compiled\_incorrect <- compileNimble(model\_incorrect, build\_incorrect)

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

## Run the MCMC sampler!

chains\_correct <- runMCMC(  
 compiled\_correct$build,  
 niter = 1100, # iterations  
 nburnin = 100, # burn-in  
 thin = 1, # thinning rate  
 nchains = 4, # number of chains  
 inits = inits\_correct, # initialization values  
 progressBar = TRUE,  
 samplesAsCodaMCMC = TRUE, # for using coda package for plots  
 summary = TRUE, # calculate summaries across chains  
 WAIC = FALSE, # Watanabe's AIC  
 perChainWAIC = FALSE  
)

## running chain 1...

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

## running chain 2...

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

## running chain 3...

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

## running chain 4...

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

chains\_incorrect <- runMCMC(  
 compiled\_incorrect$build,  
 niter = 1100, # iterations  
 nburnin = 100, # burn-in  
 thin = 1, # thinning rate  
 nchains = 4, # number of chains  
 inits = inits\_incorrect, # initialization values  
 progressBar = TRUE,  
 samplesAsCodaMCMC = TRUE, # for using coda package for plots  
 summary = TRUE, # calculate summaries across chains  
 WAIC = FALSE, # Watanabe's AIC  
 perChainWAIC = FALSE  
)

## running chain 1...

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

## running chain 2...

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

## running chain 3...

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

## running chain 4...

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

# Examine output

Let’s look at the summary of our estimates:

chains\_correct$summary$all.chains

## Mean Median St.Dev. 95%CI\_low 95%CI\_upp  
## beta0\_hat 0.8044549 0.7952253 0.28458041 0.2902359 1.388757  
## beta1\_hat 1.9249530 1.8966507 0.38971600 1.2467187 2.757742  
## y\_est\_mean 0.6181225 0.6200000 0.05437729 0.5100000 0.720000

chains\_incorrect$summary$all.chains

## Mean Median St.Dev. 95%CI\_low 95%CI\_upp  
## beta0\_hat 0.7783726 0.7772996 0.3389844 0.1101078 1.465629  
## beta1\_hat 2.1361259 2.1023053 0.4679839 1.3359975 3.220641  
## beta2\_hat 0.1844698 0.1614406 0.3956241 -0.5620953 1.014437  
## y\_est\_mean 0.6189550 0.6200000 0.0533073 0.5100000 0.720000

# correct values  
c(beta0 = beta0, beta1 = beta1, y\_mean = mean(y))

## beta0 beta1 y\_mean   
## 1.0000000 1.2000000 0.9085741

These are our estimates for the parameters of interest.

# Model diagnostics

## Chain convergence

First, calculate the the Gelman-Rubin diagnostic values for assessing convergence of our chains:

gelman.diag(chains\_correct$samples)

## Potential scale reduction factors:  
##   
## Point est. Upper C.I.  
## beta0\_hat 1 1.01  
## beta1\_hat 1 1.02  
## y\_est\_mean 1 1.00  
##   
## Multivariate psrf  
##   
## 1.01

gelman.diag(chains\_incorrect$samples)

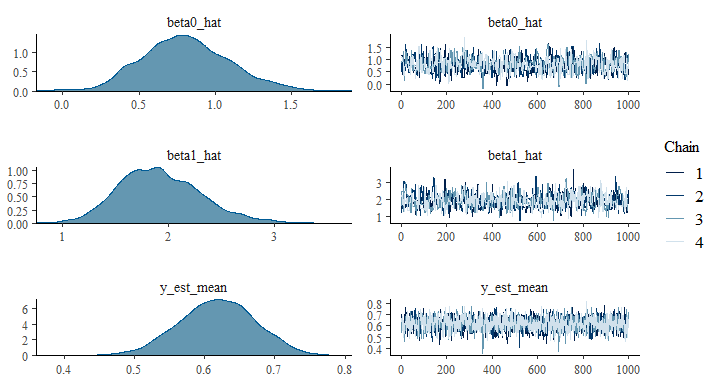
## Potential scale reduction factors:  
##   
## Point est. Upper C.I.  
## beta0\_hat 1.02 1.06  
## beta1\_hat 1.01 1.03  
## beta2\_hat 1.01 1.02  
## y\_est\_mean 1.01 1.03  
##   
## Multivariate psrf  
##   
## 1.02

In my run, these are all <1.1 so look OK.

## Chain mixing, convergence, and parameter estimates

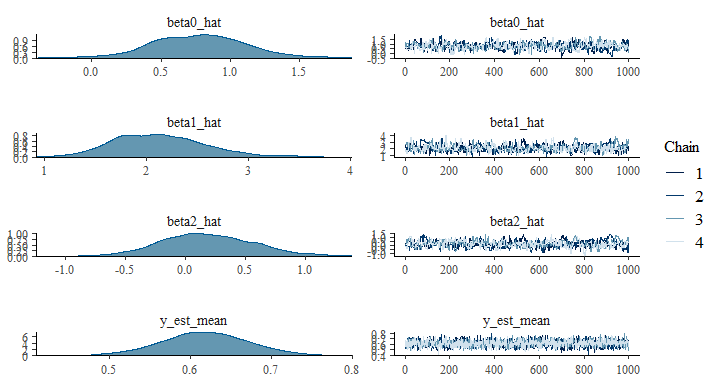
Now, let’s confirm what we saw in the G-R values by examining the trace plots and parameter estimates, first for the correct model.

mcmc\_combo(chains\_correct$samples)



Compare these to the incorrect model.

mcmc\_combo(chains\_incorrect$samples)



You can probably see that the chain estimates for the incorrect model seem less well-mixed than those for the correct model. This isn’t necessarily a tell-tale sign of model misspecification. The incorrect model is more complex, so it is likely to take more time to mix sufficiently.

What about parameter estimates? They distributions are really very similar, and there’s no clear winner. Our simulated version of ybinary, and its mean, also does not indicate which model is the correct one because both do a good job of recreating the data. Part of the reason is because the squared term (the beta2\_hat coefficient) in the incorrect model was correctly estimated to be ~0, on average. Thus, it had little effect on simulated values of ybinary.

All the same, we get very equally-performing models. Given that, based on the principle of parsimony, we should probably use the simpler model.

### The easier way

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

model <- glm(ybinary ~ x\_scaled, family = binomial())  
coefficients(model)

## (Intercept) x\_scaled   
## 0.79376 1.86586

*Finis!*