13/10/2019

The Gompertz model is one of the most used sigmoid distributions. Sigmoid distributions are commonly fitted to growth data, in the same way as a logistic growth model would be used. (first published by Verhulst in 1845?). It is now conventionally accepted that the Gompertz model can be reparameterised in the form of the “generalised logistic model” as with other common growth models (e.g. *cite*).

We use a generalised Gompertz model to parameterise our population model. Here we look at the effect of variance in the observation models when predicting the population dynamics of mice. This document steps through the state-space model construction to look at the effect of variance around the observation model and the predicted fit from the model output. The Gompertz model parameterisation for the Bayesian Model can be written out as:

This equation describes the logistic growth of a population, where is equivalent to the rate of increase and is the parameter that describes the strength of density-dependence in the population abundance at time where the carrying capacity is:

Almost all Gompertz models have been divided into two groups based on the type of location parameter (Tjorve and Tjorve 2017). A location parameter is. …….This distinction is the key difference between the type 1 and type 2 variants of the Gompertz model.

|  |  |
| --- | --- |
| Type 1 | Type 2 |
| Type one models have a single parameter that controls the time (i.e x-value) at the point on the curve. | The other group has the single parameter controlling the starting point (i.e the inter-section with the y-axis) at each time point |

For this paper we will focus on what type two as for each time point we have an estimate for different ’s for each observation process??

Method

A state-space modelling approach allows for the estimation of process dynamics.

* For ecological questions focused on linking the observed abundance in a particular species with the underlying interactions with other species, density and environmental factors
* By informing the dynamics as the visualisations of the process model we can understand the quality of observation (abundance estimates) are needed to capture the true underlying population dynamics.

Method scope

1. Simulate everything from start as a simple process model
2. Varying observational data from the following common observational models (CR abundance ) for the data () as:
3. Use the observed data from mice and beech:

**Another bit of method scope??**

We will start with very simple model of mice CR data as the base data and augmentation 5 ,10 ,100 ,500 ,1000 zeros to the longest line of captures.

1. And then we will augmentation the same numbers of zeros but even independent of the number of captures in the trip.
2. Modify p
3. Use observed CR data and augment across same range

JAGS simulation from Olivier Gimenez

Here, I illustrate the possibility to use JAGS to simulate data with two examples that might be of interest to population ecologists: first a linear regression, second a ‘Cormack-Jolly-Seber’ capture-recapture model to estimate animal survival (formulated as a state-space model). The code is available from [GitHub](https://github.com/oliviergimenez/simul_with_jags).

Recently, I have been struggling with simulating data from complex hierarchical models. After several unsuccessful attempts in R, I remembered the good old times when I was using WinBUGS (more than 10 years already!) and the possibility to simulate data with it. I’m using JAGS now, and a quick search in Google with ‘simulating data with jags’ led me to [a complex example](https://www.georg-hosoya.de/wordpress/?p=799) and [a simple example](https://stackoverflow.com/questions/38295839/simulate-data-in-jags-r2jags).

Simulating data with JAGS is convenient because you can use (almost) the same code for simulation and inference, and you can carry out simulation studies (bias, precision, interval coverage) in the same environment (namely JAGS).

Linear regression example

We first load the packages we need for this tutorial:

library(R2jags)

## Loading required package: rjags

## Loading required package: coda

## Linked to JAGS 4.3.0

## Loaded modules: basemod,bugs

##   
## Attaching package: 'R2jags'

## The following object is masked from 'package:coda':  
##   
## traceplot

library(runjags)  
library(mcmcplots)

## Registered S3 method overwritten by 'mcmcplots':  
## method from   
## as.mcmc.rjags R2jags

Then straight to the point, let’s generate data from a linear regression model. The trick is to use a data block, have the simplest model block you could think of and pass the parameters as if they were data. Note that it’d be possible to use only a model block, see comment [here](https://stackoverflow.com/questions/38295839/simulate-data-in-jags-r2jags).

txtstring <- '  
data{  
# Likelihood:  
for (i in 1:N){  
y[i] ~ dnorm(mu[i], tau) # tau is precision (1 / variance)  
mu[i] <- alpha + beta \* x[i]  
}  
}  
model{  
fake <- 0  
}  
'

Here, alpha and beta are the intercept and slope, tau the precision or the inverse of the variance, y the response variable and x the explanatory variable.

We pick some values for the model parameters that we will use as data:

# parameters for simulations   
N = 30 # nb of observations  
x <- 1:N # predictor  
alpha = 0.5 # intercept  
beta = 1 # slope  
sigma <- .1 # residual sd  
tau <- 1/(sigma\*sigma) # precision  
# parameters are treated as data for the simulation step  
data<-list(N=N,x=x,alpha=alpha,beta=beta,tau=tau)

Now call JAGS; note that we monitor the response variable instead of parameters as we would do when conducting standard inference:

# run jags  
out <- run.jags(txtstring, data = data,monitor=c("y"),sample=1, n.chains=1, summarise=FALSE)

## Compiling rjags model...  
## Calling the simulation using the rjags method...  
## Note: the model did not require adaptation  
## Burning in the model for 4000 iterations...  
## Running the model for 1 iterations...  
## Simulation complete  
## Finished running the simulation

The output is a bit messy and needs to be formatted appropriately:

# reformat the outputs  
Simulated <- coda::as.mcmc(out)  
Simulated

## Markov Chain Monte Carlo (MCMC) output:  
## Start = 5001   
## End = 5001   
## Thinning interval = 1   
## y[1] y[2] y[3] y[4] y[5] y[6] y[7]  
## 5001 1.538065 2.323131 3.521898 4.479218 5.538601 6.441434 7.690092  
## y[8] y[9] y[10] y[11] y[12] y[13] y[14]  
## 5001 8.451348 9.598716 10.60291 11.52204 12.63125 13.60879 14.54123  
## y[15] y[16] y[17] y[18] y[19] y[20] y[21]  
## 5001 15.54145 16.55213 17.58122 18.45127 19.5789 20.43211 21.47933  
## y[22] y[23] y[24] y[25] y[26] y[27] y[28]  
## 5001 22.50246 23.34913 24.43771 25.23727 26.38228 27.38148 28.50089  
## y[29] y[30]  
## 5001 29.62597 30.69498

dim(Simulated)

## [1] 1 30

dat = as.vector(Simulated)  
dat

## [1] 1.538065 2.323131 3.521898 4.479218 5.538601 6.441434 7.690092  
## [8] 8.451348 9.598716 10.602911 11.522043 12.631246 13.608794 14.541234  
## [15] 15.541451 16.552128 17.581219 18.451271 19.578899 20.432112 21.479327  
## [22] 22.502465 23.349131 24.437713 25.237270 26.382276 27.381482 28.500894  
## [29] 29.625970 30.694983

Now let’s fit the model we used to simulate to the data we just generated. I won’t go into the details and assume that the reader is familiar with JAGS and linear regression.

# specify model in BUGS language  
model <-   
paste("   
model {  
# Likelihood:  
for (i in 1:N){  
y[i] ~ dnorm(mu[i], tau) # tau is precision (1 / variance)  
mu[i] <- alpha + beta \* x[i]  
}  
# Priors:  
alpha ~ dnorm(0, 0.01) # intercept  
beta ~ dnorm(0, 0.01) # slope  
sigma ~ dunif(0, 100) # standard deviation  
tau <- 1 / (sigma \* sigma)   
}  
")  
writeLines(model,"lin\_reg.jags")   
  
# data  
jags.data <- list(y = dat, N = length(dat), x = x)  
  
# initial values  
inits <- function(){list(alpha = rnorm(1), beta = rnorm(1), sigma = runif(1,0,10))}   
  
# parameters monitored  
parameters <- c("alpha", "beta", "sigma")  
  
# MCMC settings  
ni <- 10000  
nt <- 6  
nb <- 5000  
nc <- 2  
  
# call JAGS from R  
res <- jags(jags.data, inits, parameters, "lin\_reg.jags", n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb, working.directory = getwd())

## module glm loaded

## Compiling model graph  
## Resolving undeclared variables  
## Allocating nodes  
## Graph information:  
## Observed stochastic nodes: 30  
## Unobserved stochastic nodes: 3  
## Total graph size: 130  
##   
## Initializing model

Let’s have a look to the results and compare with the parameters we used to simulate the data (see above):

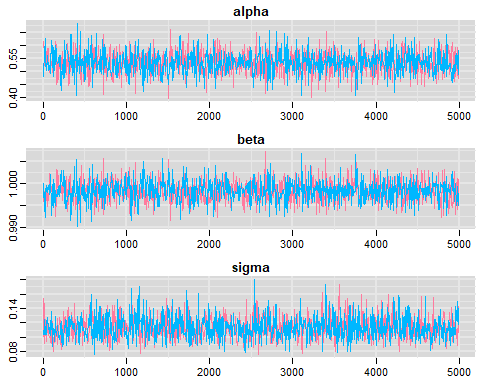
# summarize posteriors  
print(res, digits = 3)

## Inference for Bugs model at "lin\_reg.jags", fit using jags,  
## 2 chains, each with 10000 iterations (first 5000 discarded), n.thin = 6  
## n.sims = 1666 iterations saved  
## mu.vect sd.vect 2.5% 25% 50% 75% 97.5% Rhat  
## alpha 0.535 0.043 0.451 0.507 0.535 0.564 0.620 1.001  
## beta 0.998 0.002 0.994 0.997 0.998 1.000 1.003 1.001  
## sigma 0.113 0.016 0.087 0.102 0.112 0.123 0.148 1.000  
## deviance -47.432 2.639 -50.555 -49.430 -48.124 -46.136 -40.707 1.002  
## n.eff  
## alpha 1700  
## beta 1700  
## sigma 1700  
## deviance 1700  
##   
## 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 = 3.5 and DIC = -43.9  
## DIC is an estimate of expected predictive error (lower deviance is better).

Pretty close!

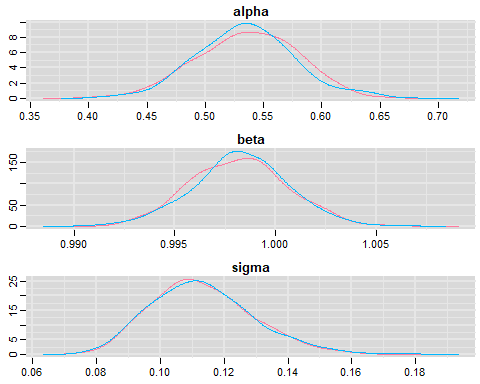
Check convergence:

# trace plots  
traplot(res,c("alpha", "beta", "sigma"))



Plot the posterior distribution of the regression parameters and residual standard deviation:

# posterior distributions  
denplot(res,c("alpha", "beta", "sigma"))



Capture-recapture example

I now illustrate the use of JAGS to simulate data from a Cormack-Jolly-Seber model with constant survival and recapture probabilities. I assume that the reader is familiar with this model and its formulation as a state-space model.

Let’s simulate!

txtstring <- '  
data{  
# Constant survival and recapture probabilities  
for (i in 1:nind){  
 for (t in f[i]:(n.occasions-1)){  
 phi[i,t] <- mean.phi  
 p[i,t] <- mean.p  
 } #t  
 } #i  
# Likelihood   
for (i in 1:nind){  
 # Define latent state and obs at first capture  
 z[i,f[i]] <- 1  
 mu2[i,1] <- 1 \* z[i,f[i]] # detection is 1 at first capture ("conditional on first capture")  
 y[i,1] ~ dbern(mu2[i,1])  
 # then deal w/ subsequent occasions  
 for (t in (f[i]+1):n.occasions){  
 # State process  
 z[i,t] ~ dbern(mu1[i,t])  
 mu1[i,t] <- phi[i,t-1] \* z[i,t-1]  
 # Observation process  
 y[i,t] ~ dbern(mu2[i,t])  
 mu2[i,t] <- p[i,t-1] \* z[i,t]  
 } #t  
 } #i  
}  
model{  
fake <- 0  
}  
'

Let’s pick some values for parameters and store them in a data list:

# parameter for simulations   
n.occasions = 10 # nb of occasions  
nind = 100 # nb of individuals  
mean.phi <- 0.8 # survival  
mean.p <- 0.6 # recapture  
f = rep(1,nind) # date of first capture  
data<-list(n.occasions = n.occasions, mean.phi = mean.phi, mean.p = mean.p, f = f, nind = nind)

Now run JAGS:

out <- run.jags(txtstring, data = data,monitor=c("y"),sample=1, n.chains=1, summarise=FALSE)

## Compiling rjags model...  
## Calling the simulation using the rjags method...  
## Note: the model did not require adaptation  
## Burning in the model for 4000 iterations...  
## Running the model for 1 iterations...  
## Simulation complete  
## Finished running the simulation

Format the output:

Simulated <- coda::as.mcmc(out)  
dim(Simulated)

## [1] 1 1000

dat = matrix(Simulated,nrow=nind)  
head(dat)

## [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]  
## [1,] 1 1 0 0 0 0 0 0 0 0  
## [2,] 1 1 0 1 1 1 0 1 1 0  
## [3,] 1 1 0 0 1 1 0 0 1 0  
## [4,] 1 1 1 0 0 0 0 0 0 0  
## [5,] 1 1 0 0 1 0 0 0 0 0  
## [6,] 1 0 0 1 1 1 0 0 0 0

## [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]  
## [1,] 1 1 0 0 0 0 0 0 0 0  
## [2,] 1 1 1 1 0 0 0 0 0 0  
## [3,] 1 0 0 0 0 0 0 0 0 0  
## [4,] 1 0 0 0 0 0 0 0 0 0  
## [5,] 1 0 0 0 0 0 0 0 0 0  
## [6,] 1 1 1 1 0 0 1 0 1 1

Here I monitored only the detections and non-detections, but it is also possible to get the simulated values for the states, i.e. whether an individual is alive or dead at each occasion. You just need to amend the call to JAGS with monitor=c("y","x") and to amend the output accordingly.

Now we fit a Cormack-Jolly-Seber model to the data we’ve just simulated, assuming constant parameters:

model <-   
paste("   
model {  
# Priors and constraints  
for (i in 1:nind){  
 for (t in f[i]:(n.occasions-1)){  
 phi[i,t] <- mean.phi  
 p[i,t] <- mean.p  
 } #t  
 } #i  
mean.phi ~ dunif(0, 1) # Prior for mean survival  
mean.p ~ dunif(0, 1) # Prior for mean recapture  
# Likelihood   
for (i in 1:nind){  
 # Define latent state at first capture  
 z[i,f[i]] <- 1  
 for (t in (f[i]+1):n.occasions){  
 # State process  
 z[i,t] ~ dbern(mu1[i,t])  
 mu1[i,t] <- phi[i,t-1] \* z[i,t-1]  
 # Observation process  
 y[i,t] ~ dbern(mu2[i,t])  
 mu2[i,t] <- p[i,t-1] \* z[i,t]  
 } #t  
 } #i  
}  
")  
writeLines(model,"cjs.jags")

Prepare the data:

# vector with occasion of marking  
get.first <- function(x) min(which(x!=0))  
f <- apply(dat, 1, get.first)  
# data  
jags.data <- list(y = dat, f = f, nind = dim(dat)[1], n.occasions = dim(dat)[2])

# Initial values  
known.state.cjs <- function(ch){  
 state <- ch  
 for (i in 1:dim(ch)[1]){  
 n1 <- min(which(ch[i,]==1))  
 n2 <- max(which(ch[i,]==1))  
 state[i,n1:n2] <- 1  
 state[i,n1] <- NA  
 }  
 state[state==0] <- NA  
 return(state)  
 }  
inits <- function(){list(mean.phi = runif(1, 0, 1), mean.p = runif(1, 0, 1), z = known.state.cjs(dat))}

We’d like to carry out inference about survival and recapture probabilities:

parameters <- c("mean.phi", "mean.p")

Standard MCMC settings:

ni <- 10000  
nt <- 6  
nb <- 5000  
nc <- 2

Ready to run JAGS!

# Call JAGS from R (BRT 1 min)  
cjs <- jags(jags.data, inits, parameters, "cjs.jags", n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb, working.directory = getwd())

## Compiling model graph  
## Resolving undeclared variables  
## Allocating nodes  
## Graph information:  
## Observed stochastic nodes: 900  
## Unobserved stochastic nodes: 902  
## Total graph size: 3707  
##   
## Initializing model

Summarize posteriors and compare to the values we used to simulate the data:

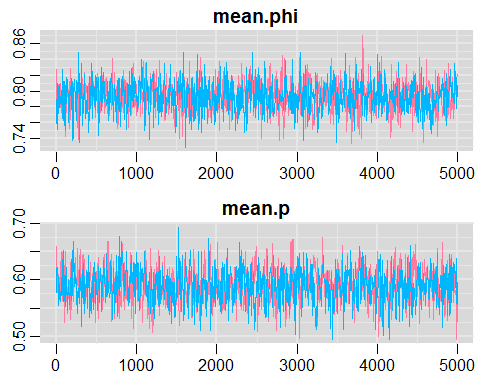
print(cjs, digits = 3)

## Inference for Bugs model at "cjs.jags", fit using jags,  
## 2 chains, each with 10000 iterations (first 5000 discarded), n.thin = 6  
## n.sims = 1666 iterations saved  
## mu.vect sd.vect 2.5% 25% 50% 75% 97.5% Rhat  
## mean.p 0.590 0.032 0.528 0.568 0.590 0.612 0.651 1.000  
## mean.phi 0.793 0.021 0.752 0.779 0.793 0.807 0.834 1.000  
## deviance 467.760 18.737 432.318 454.861 467.449 480.260 506.237 1.003  
## n.eff  
## mean.p 1700  
## mean.phi 1700  
## deviance 540  
##   
## 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 = 175.3 and DIC = 643.1  
## DIC is an estimate of expected predictive error (lower deviance is better).

Again pretty close!

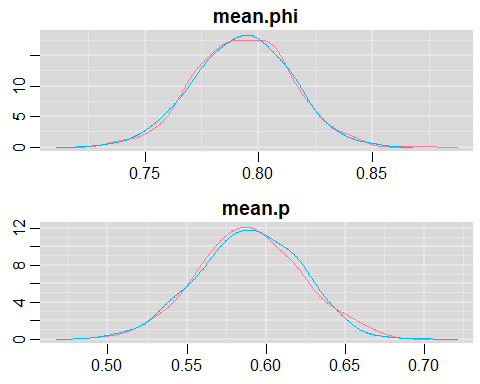
Trace plots

traplot(cjs,c("mean.phi", "mean.p"))



Posterior distribution plots:

denplot(cjs,c("mean.phi", "mean.p"))



Original from mice

One

Two

Three

Four

JAGS model

Observation model

For each grid and year we estimated the observation process to be:

which was assumed to be draw from an underlying normal distribution with a mean = 0 and sd = 1 (prior?).

where the model fit is later estimated from the simX distribution for R-squared maybe??

Process model

Model fit (black line)

Predicted fit (red line)

Priors

# for(i in 2:nYears) {  
#   
# X[i] ~ dnorm(predX[i], tau[2])  
# predX[i] <- X[i-1] + b + beta.rain \* rain[i-lag] - d \* X[i-1]  
#   
# simX[i] ~ dnorm(predsimX[i], tau[2]) # fitted model  
# predsimX[i] <- simX[i-1] + b + beta.rain \* rain[i-lag] - d \* simX[i-1]  
# }  
#   
# #priors?  
# b ~ dnorm(0, 0.001)T(0, )  
# beta.rain ~ dnorm(0, 0.01)  
# d ~ dnorm(0, 0.01)T(0, )  
# mu.r ~ dnorm(0, 0.001)  
#   
# for(i in 1:2) {  
# tau[i] <- 1 / (sigma[i] \* sigma[i])  
# sigma[i] ~ dunif(0, 10)  
# }

Assumptions

* Gompertz model 1 with same parameters for each grid.
* Here carrying capacity is: