BDA - Assignment 5

Anonymous

Contents


```
# To install aaltobda, see the General information in the assignment.
library(aaltobda)
library(markmyassignment)
library("ggplot2")
assignment_path <-
  paste("https://github.com/avehtari/BDA_course_Aalto/", "blob/master/assignments/tests/assignment5.yml
set_assignment(assignment_path)
```

```
## Assignment set:
## assignment5: Bayesian Data Analysis: Assignment 5
## The assignment contain the following task:
## - density_ratio
```

```
show_tasks()
```

```
## [1] "density_ratio"
```

Generalized linear model: Bioassay with Metropolis

1.

a)

Start by implementing a function called density_ratio to compute the density ratio function, r in Eq. (11.1) in BDA3.

```
library(aaltobda)
data("bioassay")
set.seed(05102020)

mu_0 = c(0, 10)
sigma_0 =  matrix(c(4, 10, 10, 100), 2, 2)
```

```
density_ratio <- function(alpha_propose,
                          alpha_previous,
                          beta_propose,
                          beta_previous,
                          x,
                          y,
                          n){

  # Compute the log-density of a multivariate normal prior
  log_prior_prop = dmvnorm(matrix(c(alpha_propose, beta_propose), 1, 2), mu_0, sigma_0, log = TRUE)
  log_prior_prev = dmvnorm(matrix(c(alpha_previous, beta_previous), 1, 2), mu_0, sigma_0, log = TRUE)
  log_likelihood_prop = bioassaylp(alpha_propose, beta_propose, x, y, n)
  log_likelihood_prev = bioassaylp(alpha_previous, beta_previous, x, y, n)


  # The unnormalized log posterior is simply the sum of log-likelihood and log-prior
  log_posterior_prop = (log_prior_prop + log_likelihood_prop)
  log_posterior_prev = (log_prior_prev + log_likelihood_prev)

  # Calculate the ratio as suggested in the BDA3 eq. 11.1
  # NOTE: Remember that p1/p0 = exp(log(p1) - log(p0))!
  exp(log_posterior_prop - log_posterior_prev)
}

mark_my_assignment("density_ratio")
```

```
## v |  OK F W S | Context
##
/ |   0       | density_ratio()
v |   6       | density_ratio()
##
## == Results =====================================================================================
## OK:       6
## Failed:   0
## Warnings: 0
## Skipped:  0
```

b)

Now implement a function called Metropolis_bioassay() which implements the Metropolis algorithm using the density_ratio().

```
Metropolis_bioassay <- function(nof_iters, nof_chains, scale){

  nof_iters = nof_iters - 1

  # 1. Draw a starting point
  res = array(dim = c(nof_chains, nof_iters+1, 2))
  chain = array(dim = c(nof_iters+1, 2))
  chain[1,] = rmvnorm(1, mu_0, sigma_0)

  for (i in 1:nof_chains){
    # 2. For t = 1, 2, . . . :
```

```
    for (t in 1:nof_iters){
      # a) sample prop
      alpha_prop = rnorm(1, chain[t, 1], (scale*2)^2)
      beta_prop = rnorm(1, chain[t, 2], (scale*10)^2)

      # b) Calculate the ratio of the densities
      r = density_ratio(alpha_prop, chain[t, 1],
                        beta_prop, chain[t, 2],
                        bioassay$x, bioassay$y, bioassay$n)

      # c) Set next chain value based on ratio
      if (runif(1) <= r){
        chain[t+1,] = c(alpha_prop, beta_prop)
      } else {
        chain[t+1,] = chain[t, ]
      }
    }

    # Save iterated chain and initialize the chain
    res[i,,] = chain
    chain = array(dim = c(nof_iters+1, 2))
    chain[1,] = rmvnorm(1, mu_0, sigma_0)

  }

  res

}
```

After testing more in console, we are satisfied with what we see and state that the algorithm works as it should. Continuing in to part 2.

2.

a)

The basic idea of the Metropolis algorithm is to take random jumps and on each jump either accept or reject a value drawn from a symmetric proposal distribution as the next member of the chain. The basic idea is that each new member of the chain is drawn from a distribution that is not further away from the target distribution compared to distribution of the previous member of the chain. That is, it is allowed that the new member can be drawn from the exact same distribution as the previous member. The acception/rejection is based on the calculated density ratio of the proposal and the previous member's distribution. The initial starting point of the chain is drawn from a starting distribution, which can be based on an approximation. In general, the algorithm is useful for sampling from Bayesian posterior distributions.

b)

As a proposal distribution I used a simple normal distribution, as it was proposed in the ex. description. Normal distribution is a symmetric distribution, which is a requirement for the jumping distribution in Metropolis algorithm. I set the scale to be an argument of the function, so that it can changed later when visualizing the chains. The proposal distribution from where the new member candidates to the chain are

drawn randomly, is of form

$$N\left(\begin{bmatrix}\alpha_{previous}\\ \beta_{previous}\end{bmatrix}, \begin{bmatrix}(2s)^2\\ (10s)^2\end{bmatrix}\right)$$

, where s is the scale.

So the main choosing for the proposal distribution is the scale factor as in this case, it is already decided that the distribution type is normal. Finding an optimal scale is hard when relying only on visual analysis, but the main idea is that the scale cannot be too small, so that we more often with each iteraton, get closer to the target distribution, but the speed and amount of iterations needed for convergence (depending of course on the starting point) can be a really high number. Also, if the scale is too high, we always return in the same point as previously as we just either jump further away from the convergence or even to the other side of the distribution, and so the member of the chain never changes and we might never reach the target distribution.

Based on playing with the plots in the later parts of the report, I ended up choosing scale factor 0.4.

c)

The starting points α_0, β_0 are drawn randomly from a bivariate normal prior distribution $N(\mu_0, \Sigma_0)$, where

$$\mu_0 = \begin{bmatrix}0\\ 10\end{bmatrix}, \Sigma_0 = \begin{bmatrix}4 & 10\\ 10 & 100\end{bmatrix}$$

.

d)

For each chain, I used 1000 iterations. The number was decided based on the visual experiments about the convergence. It was also suggested on BDA3 page 283 that the total number of iterations saved is no more than 1000, so that we don't run in to storage problems, especially when the are more parameters present-.

e)

Warm-up length is used to diminish the influence of the early values of the chain. Even when simulations have reached approximate convergence, early iterations still reflect the starting approximation rather than the target distribution.

I used the general warm-up length that was proposed in the book p. 282, so in other words, discarded the first half of the chains.

f)

There are two main challenges in iterative simulation inference: First, if the iterations have not proceeded long enough, and second is the within-sequence correlation of the simulaton draws.

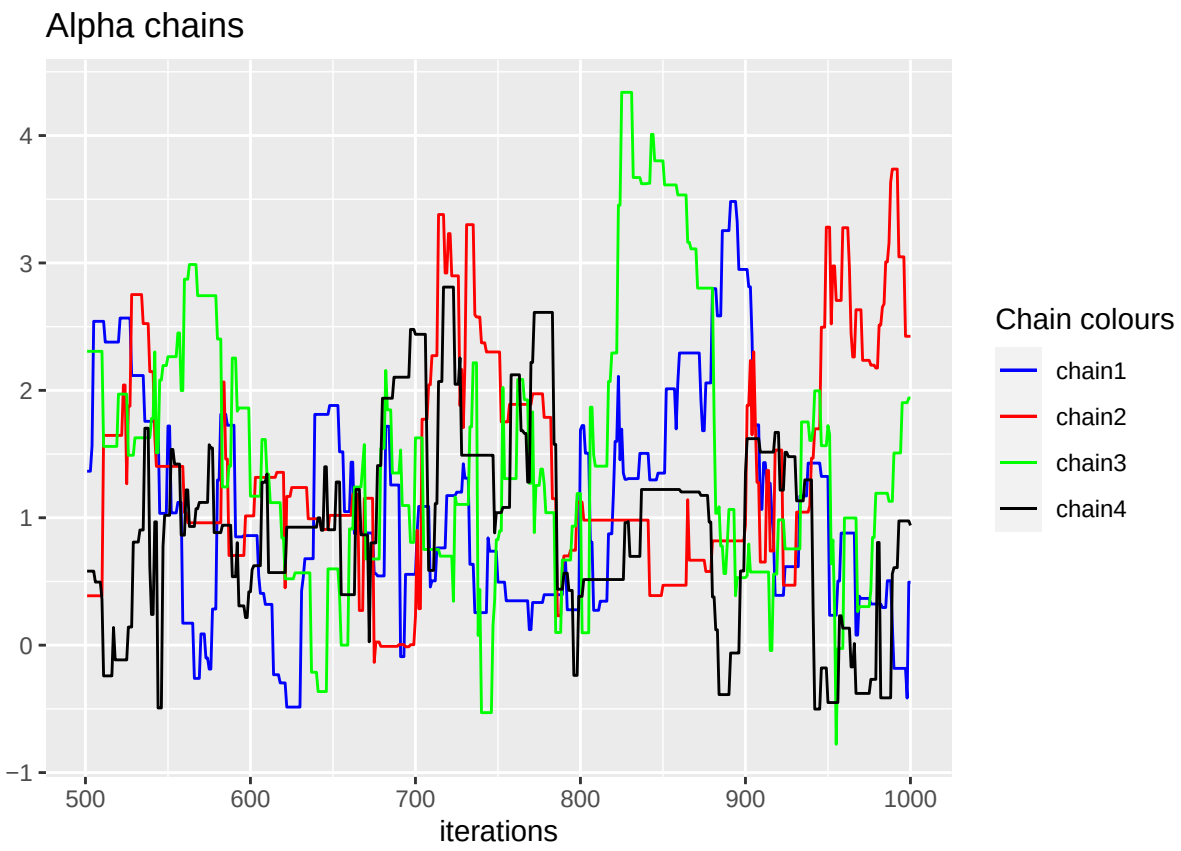
We handle the special problems of iterative simulation in three ways. First, we attempt to design the simulation runs to allow effective monitoring of convergence, in particular by simulating multiple sequences with starting points dispersed throughout parameter space. Second, we monitor the convergence of all quantities of interest by comparing variation between and within simulated sequences until 'within' variation roughly equals 'between' variation. Only when the distribution of each simulated sequence is close to the distribution of all the sequences mixed together can they all be approximating the target distribution. (BDA3 p.282)

Due to the reasons stated above, multiple chains are needed to infer from iterative simulations. I used 4 as the number of chains because for the convergence diagnostics, at least 4 chains are recommended.

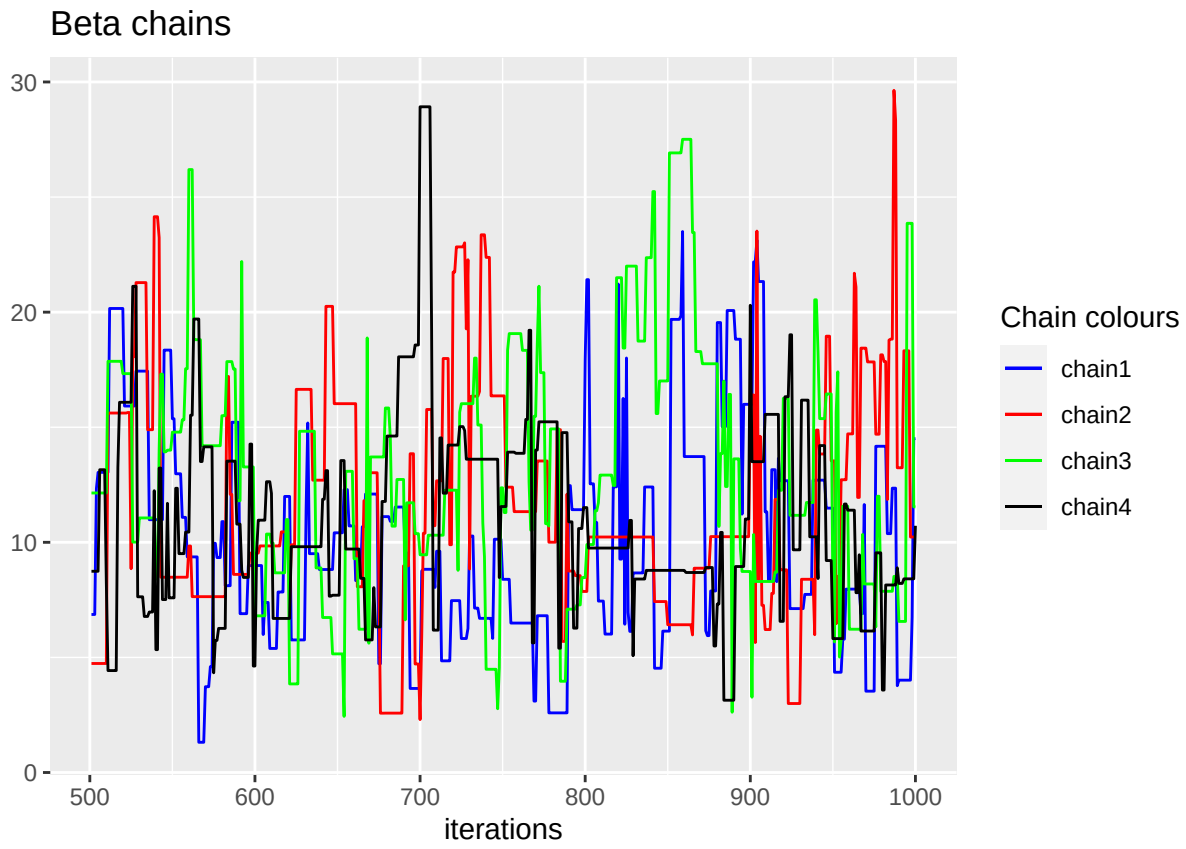
g & h)

Plot the chains for alpha and beta.

```
number_of_chains = 4
number_of_iterations = 1000
scale = 0.4

chains = Metropolis_bioassay(number_of_iterations, number_of_chains, scale)

# Plot for alpha
alpha_chains = chains[,,1]

alpha_data = data.frame(iterations = c(1:dim(alpha_chains)[2]),
                        chain1 = alpha_chains[1,],
                        chain2 = alpha_chains[2,],
                        chain3 = alpha_chains[3,],
                        chain4 = alpha_chains[4,])

# Discard a part of the chains based on the the warm-up length
warmup = number_of_iterations / 2 + 1
alpha_data = alpha_data[warmup:dim(alpha_data)[1],]

p_alpha = ggplot(alpha_data, aes(x=iterations)) +
  geom_line(aes(y=chain1, color = 'chain1')) +
  geom_line(aes(y=chain2, color = 'chain2')) +
  geom_line(aes(y=chain3, color = 'chain3')) +
  geom_line(aes(y=chain4, color = 'chain4')) +
  labs(title = 'Alpha chains', y = '') +
  scale_color_manual(name = "Chain colours",
                     breaks = c('chain1','chain2', 'chain3', 'chain4'),
                     values = c("chain1" = "blue", "chain2" = "red", "chain3" = "green", "chain4" = "bla
p_alpha
```




```
# Plot for beta
beta_chains = chains[,,2]

beta_data = data.frame(iterations = c(1:dim(alpha_chains)[2]),
                       chain1 = beta_chains[1,],
                       chain2 = beta_chains[2,],
                       chain3 = beta_chains[3,],
                       chain4 = beta_chains[4,])

# Discard a part of the chains based on the the warm-up length
warmup = number_of_iterations / 2 + 1
beta_data = beta_data[warmup:dim(beta_data)[1],]

p_beta = ggplot(beta_data, aes(x=iterations)) +
  geom_line(aes(y=chain1, color = 'chain1')) +
  geom_line(aes(y=chain2, color = 'chain2')) +
  geom_line(aes(y=chain3, color = 'chain3')) +
  geom_line(aes(y=chain4, color = 'chain4')) +
  labs(title = 'Beta chains', y = '') +
  scale_color_manual(name = "Chain colours",
                     breaks = c('chain1','chain2', 'chain3', 'chain4'),
                     values = c("chain1" = "blue", "chain2" = "red", "chain3" = "green", "chain4" = "bla
p_beta
```



Discussion on the convergence of the chains

From the plots we can see that all 4 chains for both α and β mix nicely with each other and the within and the between variances seem (at least based on the plots) to be pretty much equal after 1000 simulations made for each of the 4 chains. Also, all the plotted chains appear to be pretty much stationary. So, based on these inferences made based on the visual proofs, we could say the chains are pretty much converged. But if we want to get more precise analysis, we need to proceed further than just trying to see if the convergence happen solely based on visual assesment. More precise results about the convergence can be reached with numerical analysis of between-sequence and within-sequence information.

3.

Using the Rhat function from package rstan for calculating $\hat{R}$ for α and β.

```
library(rstan)
```

```
## Loading required package: StanHeaders
```

```
## rstan (Version 2.21.2, GitRev: 2e1f913d3ca3)
```

```
## For execution on a local, multicore CPU with excess RAM we recommend calling
## options(mc.cores = parallel::detectCores()).
## To avoid recompilation of unchanged Stan programs, we recommend calling
## rstan_options(auto_write = TRUE)
```

```r
# Compute the R_hat convergence diagnostic for alpha and beta, respectively

# For alpha
alpha_sims = subset(alpha_data, select = -c(iterations) )
Rhat(data.matrix(alpha_sims))
```

```
## [1] 1.038671
```

```r
# For beta
beta_sims = subset(beta_data, select = -c(iterations) )
Rhat(data.matrix(beta_sims))
```

```
## [1] 1.042879
```

a)

The basic idea of $\hat{R}$ is to serve as a convergence diagnostic that compares the between- and within-chain estimates for model parameters. With this estimand one can monitor convergence of the iterative simulation of the current proposal distribution. If this potential scale reduction estimand gets high values, conducting further simulations can improve our inference about the target distribution.

b)

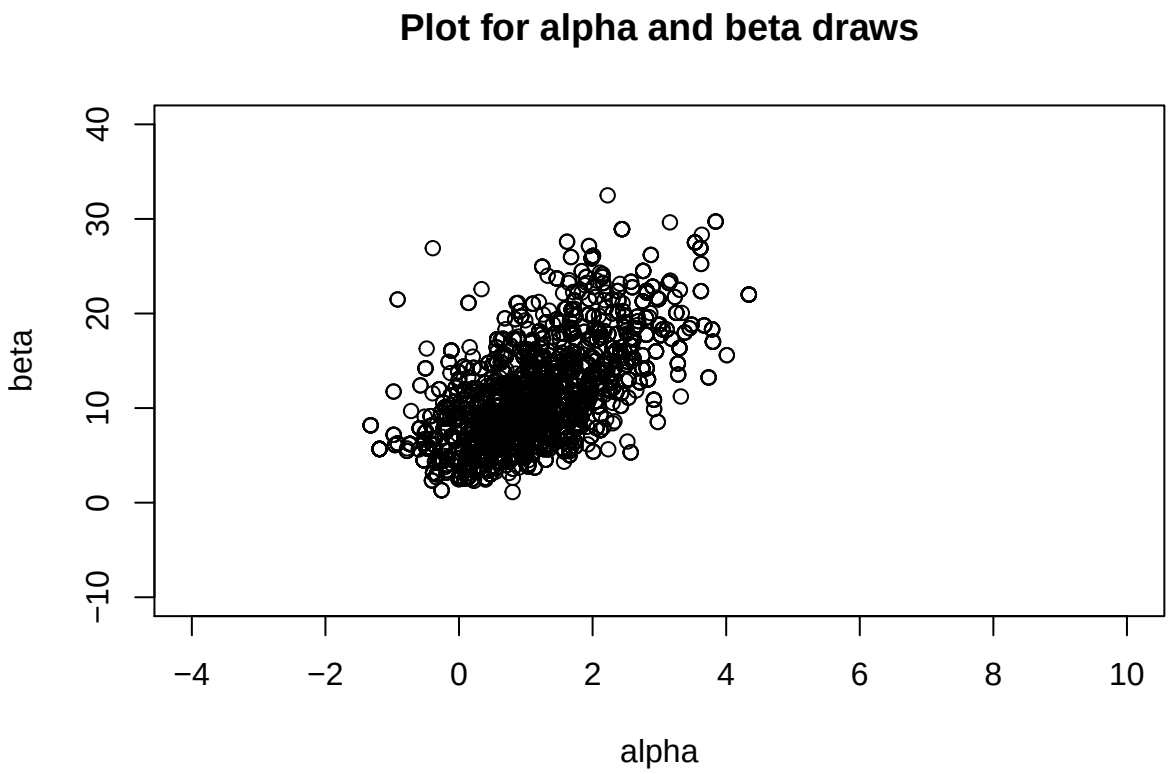
The convergence diagnostics aquired were good with first try.

The value of $\hat{R}$ is close to 1 for both parameters alpha and beta drawn from the initial proposal distribution. This means that we are satisfied with our convergence and do not need to proceed simulating.

4.

Plotting the draws for α and β and comparing it to Figure 3.3b in BDA3. Scale the plot in similar manner as the plot 3.3b in BDA3 for easier visual comparison.

```r
alpha = alpha_chains
beta = beta_chains

# Scale the plot in similar manner as the plot 3.3b in BDA3 for easier visual comparison.
plot(alpha, beta, ylim = c(-10, 40), xlim = c(-4, 10), main = "Plot for alpha and beta draws")
```



Compared to the plot 3.3b in BDA3, the plots do indeed look fairly similar.