Doing Markov Chain Monte Carlo by Hand

Inglis, A., Ali, A., Wundervald, B. and Prado, E.

December, 2018

Contents

1 Introduction

Statistical inference is a tool that allows us to make decisions based on collected data through findings about the paramters of statistical models. It can be conducted under two main different approaches: classical and Bayesian. The classical approach uses only the data to find estimates about the parameters, while the Bayesian approach makes use also of our prior information about the problem in hand. Along with that, from the Bayesian perspective, there is no fundamental distinction between observations and parameters in a model, as they are all considered random variables (Gasparini et al. (1997)). Consider that $\mathbf{Y}$ is random variable, $P(\mathbf{Y} = \mathbf{y})$ representing its probability distribution and $P(\mathbf{Y} = \mathbf{y}|\theta)$ the probability distribution conditioned on the value of the parameter θ, which is unknown. The Bayesian approach assumes a distribution for both the data and the parameter, including a $P(\theta)$ in the analysis, which has a prior (to the data) and a posterior form. Then, the assumed distribution for the data and the prior distribution for the parameters are combined through the Bayes' theorem, resulting in the posterior distribution of the parameters, obtained with

$$P(\theta|\mathbf{y}) = \frac{P(\mathbf{y}|\theta)P(\theta)}{P(\mathbf{y})}. \tag{1}$$

It is important to notice that $P(\mathbf{y})$, even if unknown, can be written as

$$P(\mathbf{y}) = \int P(\mathbf{y}|\theta)P(\theta)d\theta. \tag{2}$$

Oftentimes, the posterior distribution has no tractable form. This means that the complete formula of the posterior is not entirely known nor it is possible to work with directly. The most common case is when the posterior is obtained up to the Normalization constant, which might not be easy to evaluate (Gelman et al. (2014)).

The Markov Chain Monte Carlo (MCMC) simulation methods are used to overcome the intractability problem. The methods consist, in summary, on drawing values of θ from approximate, known distributions and then correcting the draws in order to get a distribution closer to the target one, the $P(\theta|\mathbf{y})$. This class of algorithms performs sequential simulation, once the distribution of each simulated value depends on the previous one, and hence, the draws form a Markov Chain. This property is actually helpful for proving that the approximate distributions are improved at each step of the simulation, in the sense that they converge to the target one (Gelman et al. (2014), Brooks et al. (2011)).

The key for MCMC methods to work is to create a Markov process whose stationary distribution is the target $P(\theta|\mathbf{y})$ and running the simulation long enough that the current distribution is very close to the stationary distribution (Gelman et al. (2014)) .

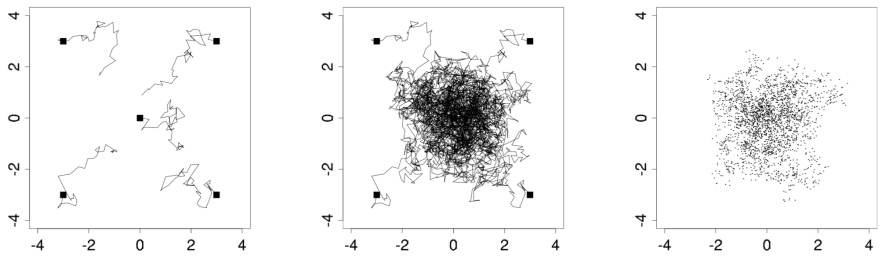

Figure 1: Illustration of a Markov Chain simulation (Gelman et al., 2014)

It is always necessary to check for the convergence of the algorithm. Let us consider, for example, Figure 1, extracted from Gelman et al. (2014) . The three plots represent five independent sequences of a Markov chain simulation targeting a bivariate Normal distribution. In the first plot, on the left of Figure 1, 50 iterations

have already happened but with no sign of probable convergence. In the plot in the middle, after 1000 iterations, the sequences are now close to the target distribution. The last plot represents a set of correlated draws from the target distribution. So, this is a case where fair convergence was achieved within a reasonable number of iterations, but this is not always the instance. If convergence is too slow, the algorithm might require a bit of improvement.

There is a few statistical programs that perform MCMC efficiently. The `BUGS` (Using Gibbs Sampling) (Thomas et al. (2003)) program was introduced in 1989, providing a platform to perform Bayesian analysis from a script. `BUGS` was followed by the `Win-BUGS` (Ntzoufras (2008)) program, initially available just for Microsoft Windows, which made it no longer possible to compile from a script. The open-source software `JAGS`(Just Another Gibbs Sampler) (Plummer (2003)) was introduced in 2003, allowing users to test and compare different Bayesian methodologies. Nowadays, we also have `Stan` (Carpenter et al. (2017)), that deals with both univariate and multivariate marginal distributions and utilizes Hamiltonian Monte Carlo, providing a faster convergence to the target distribution by using the gradient of its log.

The goal of this report is to explain and provide the implementation of a few bayesian models using Markov Chain Monte Carlo methods. The selected method for this case is the Gibbs sampler. Our aim is to understand how the models are specified and how the structure of the algorithm is constructed. As writing algorithms is usually a good way to learn how theory actually works, this work is a way of clarifying the mathematics of the methods. As it is well known that Bayesian analysis can be conducted using different kinds of software, our implementations can be compared with a standard one, giving us a notion of how efficient they are.

In this report, first we introduce the Markov Chain Monte Carlo methods in Section 1, giving a brief explanation and algorithm for the Gibbs sampling, the Metropolis and the Metropolis-Hastings algorithm. After that, Section 2 discusses and shows the applications of three MCMC models in their Bayesian form: the linear regression, the logistic regression and the Beta regression model. For all three models, Section 3 presents the theory, also, the raw implementation of MCMC in R and python and a comparison with the `JAGS` program using its respective usual interface in R (`R2jags`). Section 4 finishes the document with the final remarks, comprising the main achievements of the project, a few lines about how the method can be extended to other models and reflections on the key challenges and learnings of the group.

2 Markov Chain Monte Carlo Methods

Monte Carlo Markov Chain is a simulation method that draws values of θ from approximate distributions then correcting the draws towards the target distribution $p(\theta \mid y)$ (Gelman et al. (2014)). A Markov chain is created whose stationary distribution is $p(\theta \mid y)$.

he main theory behind the MCMC methods is that any chain that is irreducible and aperiodic will have a stationary distribution and at the t step the transition kernel will converge to that stationary distribution as $\mathrm{t} \to \infty$ (Brooks et al. (2011)).

The concepts of aperiodictiy and irreducibility are important properties for the chain to converge to the stationary distribution. As in Andrieu et al. (2003), irreducibility is when the chain has a positive probability of visiting other states. Aperiodicty means that the chain should not get trapped in cycles. Another important property is reversibility where

$$p(\theta^t)p(\theta^{t-1} \mid \theta^t) = p(\theta^{t-1})p(\theta^t \mid \theta^{t-1}).$$

On this section we give the explanation about the theory and algorithm of the selected MCMC methods. The following pages are dedicated to the Gibbs sampling, Metropolis and Metropolis-Hastings methods.

2.1 Gibbs sampling

Suppose that a parameter vector θ has been divided into d componentes or subvectors, $\theta = (\theta_1, \ldots, \theta_d)$. Each iteration of the Gibbs sampler cycles through the subvectors of θ, drawing each subset conditional on the

value of all the others, making it a iteration with d steps. An ordering of the d subvectors of θ is chosen and, in turn, θ_j^t is sampled from the conditional distribution given all the other components of θ:

$$p(\theta_j|\theta_{-j}^{t-1}, y)$$

where θ_{-j}^{t-1} represents all the components of θ except for θ_j. Each subvector θ_j is updated conditional on the latest values of the other components of θ, which are the iteration t values for the components already updated and the iteration *t-1* values for the others (Gelman et al. (2014)). In Gibbs sampling the proposal is from a conditional distribution of the desired equilibrium distribution which is always accepted (Brooks et al. (2011)).

The idea is to successively sample from each conditional distribution of θ_j in order to obtain samples from the joint posterior distribution. The Gibbs Sampling algorithm is described as following:

Algorithm Gibbs Sampling

1. Initialize $t = 1$ and define initial values $\theta_0^{(0)}, \theta_1^{(0)}, ..., \theta_{j-1}^{(0)}$ for the vector $\boldsymbol{\theta} = (\theta_0, \theta_2, ..., \theta_{j-1})^\top$.
2. Sample

$$\begin{aligned}
\theta_0^{(t)} &\sim p(\theta_0|\theta_1^{(t-1)}, \theta_2^{(t-1)}, \theta_3^{(t-1)}, ..., \theta_{d-1}^{(t-1)}, \mathbf{y}); \\
\theta_1^{(t)} &\sim p(\theta_1|\theta_0^{(t)}, \theta_2^{(t-1)}, \theta_3^{(t-1)}, ..., \theta_{d-1}^{(t-1)}, \mathbf{y}); \\
&\vdots \\
\theta_{d-1}^{(t)} &\sim p(\theta_{j-1}|\theta_1^{(t)}, \theta_2^{(t)}, \theta_3^{(t)}, ..., \theta_{j-2}^{(t)}, \mathbf{y});
\end{aligned}$$

3. Take $t = t + 1$ and return to the step 2 until the desired sample has been obtained for each θ_j.

2.2 Metropolis Algorithm

Metropolis-Hastings is actually a general term for a family of Markov Chain simulation methods used in Bayesian modelling to sample from the posterior of the parameters. The Gibbs sampler can be viewed of a particular case of this family.

The basic Metropolis algorithm is a generalization of the Metropolis-Hastings. It is an adaptation of a random walk with an acceptance/rejection rule to converge to the specific target distribution (Gelman et al. (2014)).

The method makes use of a proposal distribution $J_t(\theta^* \mid \theta_{t-1})$ from where it samples from. This proposal distribution always conditioned on the values of the last iteration and must be symmetric, satisfying $J_t(\theta_a \mid \theta_b) = J_t(\theta_b \mid \theta_a)$ for all θ_a, θ_b, and t. If the ratio of the densities of the current value and of the previous value is bigger than sample from the unit uniform distribution, the θ^* is accepted as the new value θ^t (Gelman et al. (2014)). The Metropolis algorithm can be written as:

Algorithm Metropolis

1. Initialize $t = 1$ and define the initial values $\theta_1^{(0)}, \theta_2^{(0)}, ..., \theta_d^{(0)}$ for the vector $\boldsymbol{\theta} = (\theta_0, \theta_1, ..., \theta_d)$;
2. Sample $\boldsymbol{\theta}^*$ from the proposal distribution $q(\boldsymbol{\theta}^{(t-1)})$;
   (a) Compute

$$\alpha(\boldsymbol{\theta}^{(t-1)}, \boldsymbol{\theta}^*) = \min\left\{1, \frac{p(\boldsymbol{\theta}^*|\mathbf{y})}{p(\boldsymbol{\theta}^{(t-1)}|\mathbf{y})}\right\}, \tag{3}$$

   (b) Compute $u \sim U[0, 1]$. If $u < \alpha(\boldsymbol{\theta}^{(t-1)}, \boldsymbol{\theta}^*)$, then $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^*$, otherwise, $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)}$;
3. Take $t = t + 1$ and return to the step 2 until the desired posterior sample has been obtained.

2.3 Metropolis-Hastings algorithm

The Metropolis-Hastings algorithm is a generalization of the Metropolis algorithm. In this case, the difference is that the proposal distribution $J(\theta^* \mid \theta)$ does not need to be symmetric (Gelman et al. (2014)). The Metropolis-Hastings algorithm, with a stationary distribution $p(\theta|y)$ and proposal distribution $q(\theta^* \mid \theta)$, involves sampling a value θ^* given θ from the proposal distribution $q(\theta^* \mid \theta)$. The Markov chain then moves towards θ^* with an acceptance probability

$$\alpha(\theta^{(t-1)}, \theta^*) = \min\left\{1, \frac{p(\theta^* \mid y)q(\theta^{(t-1)})}{p(\theta^{(t-1)} \mid y)q(\theta^*)}\right\},$$

otherwise it remains at θ.

The speed of the random walk process is improved by the loose of the symmetry constraint and the convergence to the target distribution is proved in the same way as for the basic Metropolis algorithm (Gelman et al. (2014)). Nevetheless, to correct for the assymmetry in the jumping rule, the acception value is replaced by a ratio of ratios, as shown in the following algorithm:

Algorithm Metropolis-Hastings

1. Initialize $t = 1$ and define the initial values $\theta_1^{(0)}, \theta_2^{(0)}, ..., \theta_d^{(0)}$ for the vector $\boldsymbol{\theta} = (\theta_0, \theta_1, ..., \theta_d)$;
2. Sample $\boldsymbol{\theta}^*$ from the proposal distribution $q(\boldsymbol{\theta}^{(t-1)})$;
   (a) Compute
   $$\alpha(\boldsymbol{\theta}^{(t-1)}, \boldsymbol{\theta}^*) = \min\left\{1, \frac{p(\boldsymbol{\theta}^*|\mathbf{y})q(\boldsymbol{\theta}^{(t-1)})}{p(\boldsymbol{\theta}^{(t-1)}|\mathbf{y})q(\boldsymbol{\theta}^{(*)})}\right\}, \tag{4}$$
   (b) Compute $u \sim U[0,1]$. If $u < \alpha(\boldsymbol{\theta}^{(t-1)}, \boldsymbol{\theta}^*)$, then $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^*$, otherwise, $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)}$;
3. Take $t = t + 1$ and return to the step 2 until the desired posterior sample has been obtained.

---

3 MCMC Models by hand

On this section we present the theory, the raw implementation in R and in python for three models: Linear Regression, Logistic Regression and Beta Regression.

3.1 Linear Regression

In order to illustrate a Bayesian linear model where Metropolis-Hastings may be helpful, consider a random variable $\mathbf{y}$, being assumed that $\mathbf{y} \sim \mathbf{N}_n(\mathbf{X}\boldsymbol{\beta}, \sigma^2\mathbf{I})$, assuming that both $\boldsymbol{\beta}$ and σ^2 are unknown. First, consider a Normal distribution as prior for each $\beta_j|\sigma^2$ such as

$$f(\beta_j|m_j, v_j) = (2\pi v_j)^{-1/2} \exp\Big\{-\frac{1}{2v_j}(\beta_j - m_j)^2\Big\},$$

where $\mathbf{X}$ is the design matrix, $\beta_j \in \mathbb{R}$, $m_j \in \mathbb{R}$ and $v_j > 0$. In this case, the posterior conditional distribution for $\boldsymbol{\beta}|\sigma^2$ is given by

$$f(\boldsymbol{\beta}|\mathbf{y},\mathbf{X},\sigma^2) \propto f_{\mathrm{y}}(\mathbf{y}|\mathbf{X},\boldsymbol{\beta}) \prod_{i=1}^{d} f(\beta_j|m_j,v_j),$$
$$\propto \exp\Big\{ -\frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^\top(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) - \sum_{j=1}^{d}\frac{1}{2v_j}(\beta_j - m_j)^2\Big\}.$$

For σ^2, consider as prior distribution an Uniform, denoted by $\sigma^2|a,b \sim U(a,b)$, in the form of

$$f(\sigma^2|a,b) = \frac{1}{b-a},$$

where $-\infty < a < b < \infty$. Hence, the posterior conditional distribution for σ^2 is given by

$$f(\sigma^2|\mathbf{y},\mathbf{X},\boldsymbol{\beta}) \propto f_{\mathrm{y}}(\mathbf{y}|\mathbf{X},\boldsymbol{\beta}) f(\sigma^2|a,b),$$
$$\propto (\sigma^2)^{-(n/2+a+1)} \exp\Big\{ -\frac{1}{2\sigma^2}[(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^\top(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})]\Big\}\frac{1}{b-a},$$

3.1.1 R Code

The code below is the the implementation of the respective linear regression model using MCMC and the programming language R.

```
for(i in 1:TotIter){

  # Proposal distribution for sigma2
  sigma2c = rgamma(1, (sigma2_^2)/var,  scale=1/(sigma2_ / var))


  # Computing the ratio in the Metropolis-Hastings
  ratio_sigma2 = (((-N/2) * log(sigma2c) - 0.5/sigma2c *
  (t(y - (X%*%MCMCBetasI))%*%(y-(X%*%MCMCBetasI))) - log(b - a) + 1/sigma2_ +
  dgamma(sigma2_, (sigma2c^2)/var, scale = 1/(sigma2c / var))) -
  ((-N/2) * log(sigma2_) - 0.5/sigma2_ * (t(y - (X%*%MCMCBetasI))%*%(y-(X%*%MCMCBetasI))) -
  log(b - a) + 1/sigma2c + dgamma(sigma2c, (sigma2_^2)/var, scale = 1/(sigma2_ / var))))

  # Accept/Reject step for sigma2
  if(runif(1) < min(1, exp(ratio_sigma2)))
  {
    sigma2_ = sigma2c
    js = js +1
  }

  # Proposal distribution for beta
  MCMCBetasC <- mvrnorm(1, MCMCBetasI, V)

  # Computing the ratio:
  ratio_beta <-  ((-0.5/sigma2_* (t(y - (X %*% MCMCBetasC)) %*%
  (y - (X %*% MCMCBetasC))  - 0.5/vi*sum(abs(MCMCBetasC - mi))^2)) -
```

```
  (-0.5/sigma2_* (t(y - (X %*% MCMCBetasI)) %*%
  (y - (X %*% MCMCBetasI))  - 0.5/vi*sum(abs(MCMCBetasI - mi))^2)))

  # Accept/Reject step for beta
  if(runif(1) < min(1, exp(ratio_beta)))
  {
    MCMCBetasI <- MCMCBetasC
    jb <- jb+1
  }

  if (i > BurnIn){
  # Saving results
    SaveResults[AuxBurnIn,] <- c(AuxBurnIn, MCMCBetasI, sigma2_)
    AuxBurnIn <- AuxBurnIn + 1
  }
}
```

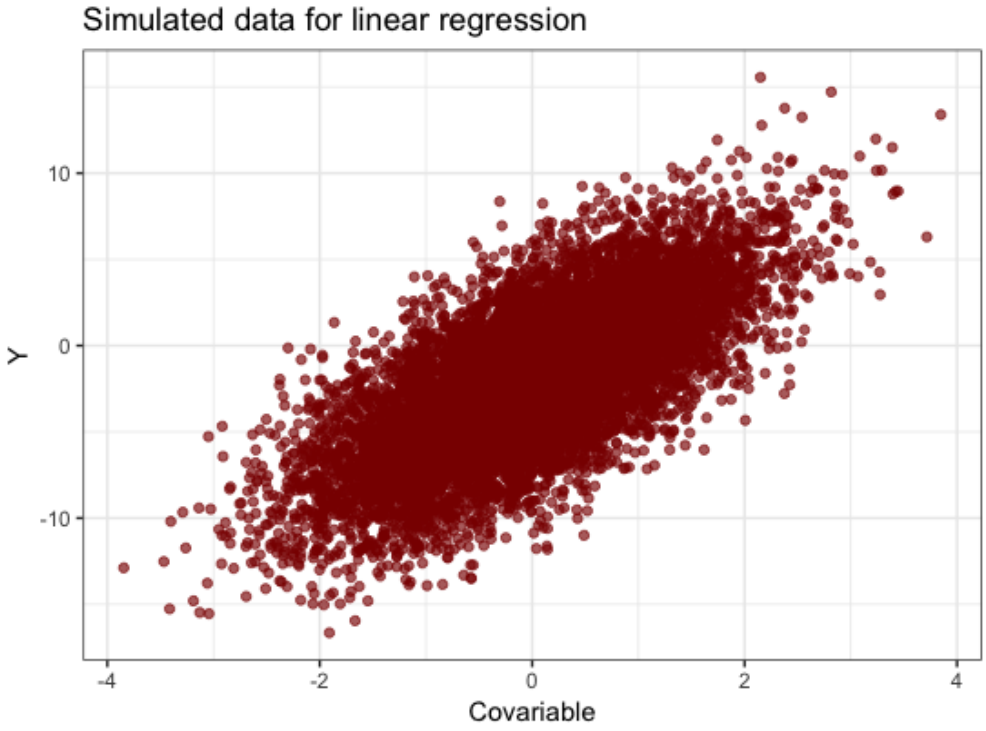

Figure 2: Simulated data for the linear regression

On Figure 2 we see the simulated data used for the `R` and python `code`.

3.1.2 Results - R Code

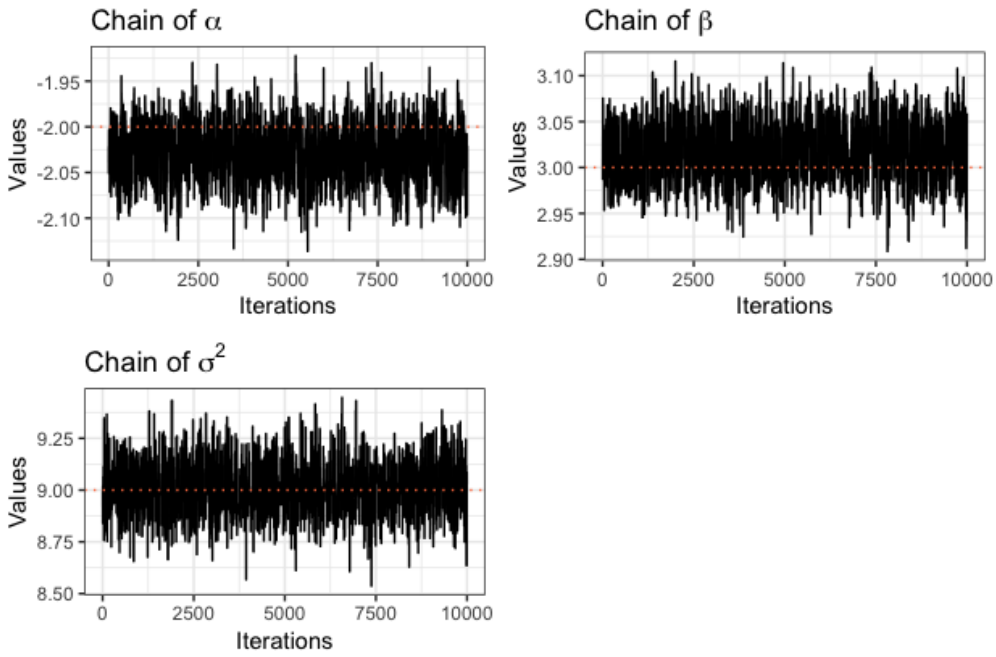

Figure 3: Results for the linear model using MCMC by hand done in R

In Figure 3, we have the results of the implementation for the linear regression in `R`. We can observe that for the regression parameters α and β, the chains are a little far from the true values. Surprisingly, for the dispersion parameter σ^2, the implementation did very well.

3.1.3 Python Code

The code below is the implementation of the respective linear regression model using MCMC and the programming language `python`.

```
for i in tqdm(range(TotIter)):

    # Proposal distribution for sigma2
    sigma2c = gamma.rvs((sigma2_**2)/var, scale=1/(sigma2_ / var), size=1, random_state=None)

    # Computing the ratio in the Metropolis-Hastings
    ratio_sigma2 = (((-N/2) * np.log(sigma2c) - 0.5/sigma2c*(y - (X @ beta_)).T @(y - (X @ beta_))
    - np.log(b - a) + 1/sigma2_ + gamma.pdf(sigma2_, (sigma2c**2)/var, scale = 1/(sigma2c / var))) -
    ((-N/2) * np.log(sigma2_) - 0.5/sigma2_*(y - (X @ beta_)).T @(y - (X @ beta_))
    - np.log(b - a) + 1/sigma2c + gamma.pdf(sigma2c, (sigma2_**2)/var,
    scale = 1/(sigma2_ / var))))

    # Accept/reject step for sigma2c (sigma2 candidate)
    if (np.random.uniform(0, 1, 1) < np.exp(ratio_sigma2)):
        sigma2_ = sigma2c
        if (i > Burn):
            js = js + 1
```

```
# Proposal distribution for beta (multivariate Normal)
betac = np.random.multivariate_normal(list(itertools.chain(*beta_)), mprop,1).T

# Computing the ratio stated in the Metropolis-Hastings algorithm
ratio_beta =  ((-0.5/sigma2_*(y - (X @ betac)).T @(y - (X @ betac))  -
0.5/vi*sum(abs(betac - mi)**2)) - (-0.5/sigma2_*(y - (X @ beta_)).T @(y - (X @ beta_))  -
0.5/vi*sum(abs(beta_ - mi)**2)))

# Accept/reject step for betac (beta candidate)
if (np.random.uniform(0, 1, 1) < np.exp(ratio_beta)):
    beta_ = betac
    if (i > Burn):
        jb = jb + 1

# Storing the results
if (i >= Burn):
    sbeta.append(beta_.tolist())
    ssigm.append(sigma2_.tolist())
```

3.1.4 Results - Python Code

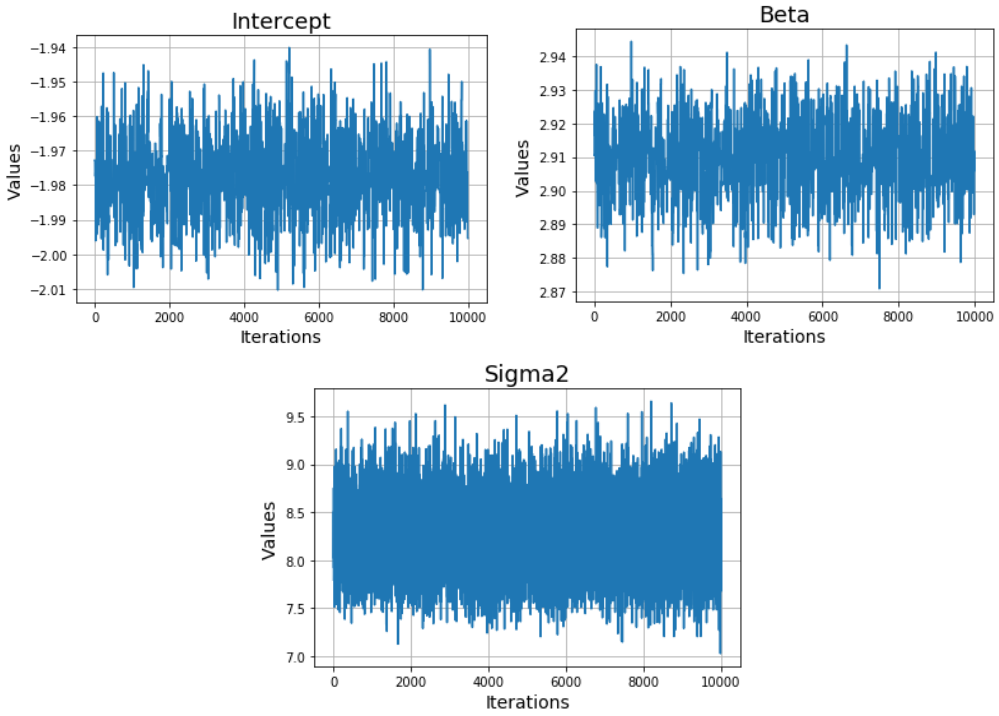

Figure 4: Results for the linear model using MCMC by hand done in python

Figure 4 shows the results of the implementation for the linear regression in `python`. In this case, all of the chains are close to the true values of the parameters.

3.2 Logistic Regression

The logistic regression model assumes that a sequence of independent and identically distributed random variables $\{Y_i\}_1^n$ has a Binomial distribution, denoted by $Y_i \sim \text{Binomial}(k_i, p_i)$, in the form of

$$f(Y_i = y_i|k_i, p_i) = \binom{k_i}{Y_i} p_i^{y_i}(1-p_i)^{k_i - y_i},$$

where $k_i \in \mathbb{N}$, $Y_i = 1, 2, ..., k_i$, $\log(\frac{p_i}{1-p_i}) = \mathbf{x}_i\boldsymbol{\beta}$, $p_i = \exp\{\mathbf{x}_i\boldsymbol{\beta}\}/(1+\exp\{\mathbf{x}_i\boldsymbol{\beta}\})$, $\mathbf{x}_i = (1, x_{i,1}, ..., x_{i,1})$ is the line vector of covariates associated to the individual i and $\boldsymbol{\beta}$ is the vector of unknown parameters. In our simulated examples we considered $k_i = 1$ for all i, which means that $Y_i \in \{0, 1\}$. Below we present the likelihood function, which is needed to compute the posterior conditional distribution for $\boldsymbol{\beta}$.

$$\begin{aligned}
\prod_{i=1}^{n} f(Y_i = y_i|p_i) &= \prod_{i=1}^{n} \binom{k_i}{y_i} p_i^{y_i}(1-p_i)^{k_i - y_i}, \\
&= \prod_{i=1}^{n} \binom{k_i}{y_i} \left(\frac{p_i}{1-p_i}\right)^{y_i} (1-p_i)^{k_i}, \\
&= \prod_{i=1}^{n} \binom{k_i}{y_i} \frac{\exp\{y_i\mathbf{x}_i\boldsymbol{\beta}\}}{(1+\exp\{\mathbf{x}_i\boldsymbol{\beta}\})^{k_i}}.
\end{aligned}$$

As prior distribution for $\boldsymbol{\beta}$, we consider a Normal distribution with mean 0 and large variance, denoted by $\boldsymbol{\beta} \sim N(\mathbf{m}, \mathbf{V})$. Thus, the joint posterior distribution is given by

$$\begin{aligned}
f(\boldsymbol{\beta}|\mathbf{X}, \mathbf{y}) &\propto \prod_{i=1}^{n} f(y_i|p_i) \times f(\boldsymbol{\beta}), \\
&\propto \prod_{i=1}^{n} \binom{k_i}{y_i} \frac{\exp\{y_i\mathbf{x}_i\boldsymbol{\beta}\}}{(1+\exp\{\mathbf{x}_i\boldsymbol{\beta}\})^{k_i}} \sqrt{(2\pi)^{-d/2}|\mathbf{V}|^{-1/2}} \exp\{-\frac{1}{2}(\boldsymbol{\beta}-\mathbf{m})^\top\mathbf{V}^{-1}(\boldsymbol{\beta}-\mathbf{m})\}, \\
&\propto \prod_{i=1}^{n} \frac{\exp\{y_i\mathbf{x}_i\boldsymbol{\beta}\}}{(1+\exp\{\mathbf{x}_i\boldsymbol{\beta}\})^{k_i}} \exp\{-\frac{1}{2}(\boldsymbol{\beta}-\mathbf{m})^\top\mathbf{V}^{-1}(\boldsymbol{\beta}-\mathbf{m})\}, \\
&\propto \left(\prod_{i=1}^{n} \frac{1}{(1+\exp\{\mathbf{x}_i\boldsymbol{\beta}\})^{k_i}}\right) \exp\{\sum_{i=1}^{n} y_i\mathbf{x}_i\boldsymbol{\beta} - \frac{1}{2}(\boldsymbol{\beta}-\mathbf{m})^\top\mathbf{V}^{-1}(\boldsymbol{\beta}-\mathbf{m})\}.
\end{aligned}$$

3.2.1 R Code

The code below is the implementation of the respective logistic regression model using MCMC and the programming language `R`.

```
for(i in 1:TotIter){

  # Proposal distribution for beta
  MCMCBetasC <- mvrnorm(1, MCMCBetasI, V)

  # Metropolis ratio
  ratio <- ((-k * sum(log(1 + exp(X%*%MCMCBetasC)))  + sum(y * X%*%MCMCBetasC) - (1/2) *
  t(MCMCBetasC - mu)%*%Sigma_inv%*%(MCMCBetasC - mu)) - ((-k * sum(log(1 + exp(X%*%MCMCBetasI)))) +
  sum(y * X%*%MCMCBetasI) - (1/2) * t(MCMCBetasI - mu)%*%Sigma_inv%*%(MCMCBetasI - mu)))

  # Accept/reject step
  if(runif(1) < min(1, exp(ratio)))
  {MCMCBetasI <- MCMCBetasC
```

```
  j <- j+1}

  # Saving results
  if (i > BurnIn){
    SaveResults[AuxBurnIn,] <- c(AuxBurnIn, MCMCBetasI)
    AuxBurnIn <- AuxBurnIn + 1
  }
}
```

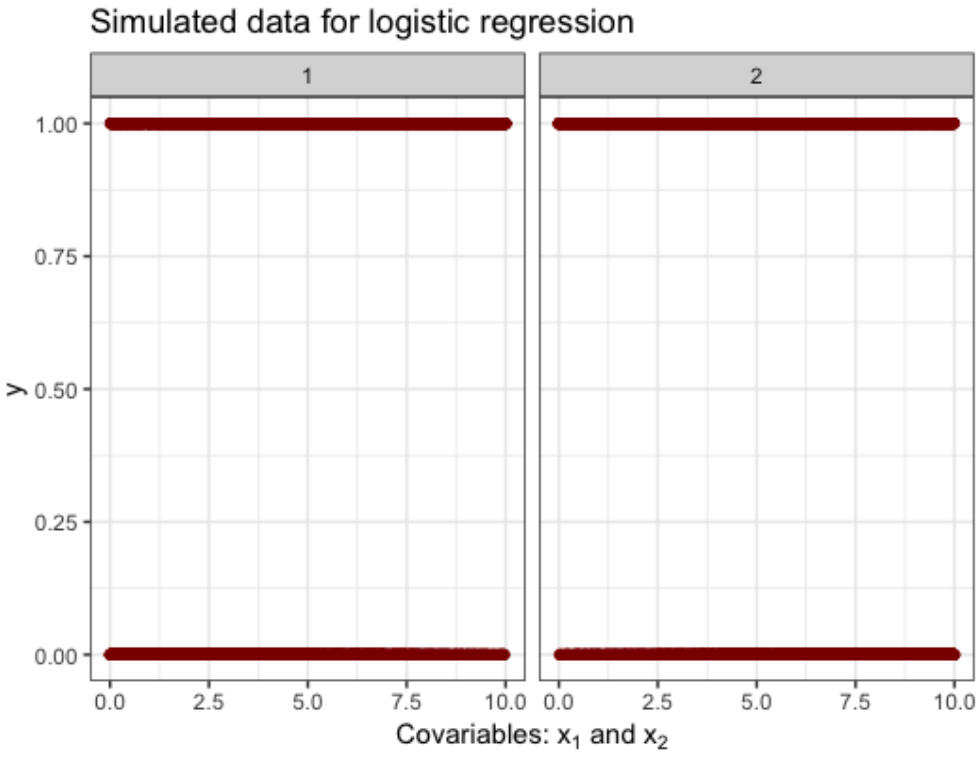

Figure 5: Simulated data for the logistic regression

On Figure 5 we see the simulated data used for the `R` and python `code` for the logistic regression model.

3.2.2 Results - R Code

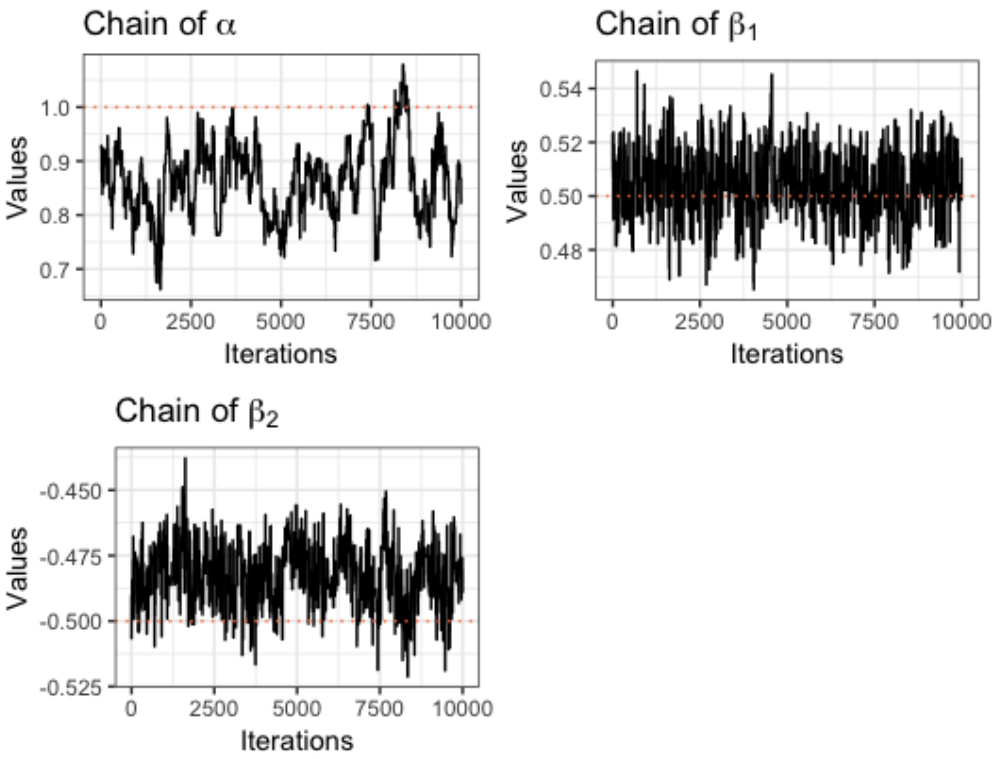

Figure 6: Results for the logistic model using MCMC by hand done in R

For the logistic regression, accordingly to the results shown in Figure 6, the chains are good enough for β_1, but not for the other two regressions parameters α and β_2.

3.2.3 Python Code

The code below is the implementation of the respective logistic regression model using MCMC and the programming language `python`.

```
for i in tqdm(range(TotIter)):

    # Proposal distribution for sigma2
    sigma2c = gamma.rvs((sigma2_**2)/var, scale=1/(sigma2_ / var), size=1, random_state=None)

    # Computing the ratio in the Metropolis-Hastings
    ratio_sigma2 = (((-N/2) * np.log(sigma2c) - 0.5/sigma2c*(y - (X @ beta_)).T @(y - (X @ beta_)) -
    np.log(b - a) + 1/sigma2_ + gamma.pdf(sigma2_, (sigma2c**2)/var, scale = 1/(sigma2c / var))) -
    ((-N/2) * np.log(sigma2_) - 0.5/sigma2_*(y - (X @ beta_)).T @(y - (X @ beta_)) -
    np.log(b - a) + 1/sigma2c + gamma.pdf(sigma2c, (sigma2_**2)/var, scale = 1/(sigma2_ / var))))

    # Accept/reject step for sigma2c (sigma2 candidate)
    if (np.random.uniform(0, 1, 1) < np.exp(ratio_sigma2)):
        sigma2_ = sigma2c
        if (i > Burn):
            js = js + 1
```

```
# Proposal distribution for beta (multivariate Normal)
betac = np.random.multivariate_normal(list(itertools.chain(*beta_)), mprop,1).T

# Computing the ratio stated in the Metropolis-Hastings algorithm
ratio_beta =  ((-0.5/sigma2_*(y - (X @ betac)).T @(y - (X @ betac))  -
0.5/vi*sum(abs(betac - mi)**2)) -(-0.5/sigma2_*(y - (X @ beta_)).T @(y - (X @ beta_))
- 0.5/vi*sum(abs(beta_ - mi)**2)))

# Accept/reject step for betac (beta candidate)
if (np.random.uniform(0, 1, 1) < np.exp(ratio_beta)):
    beta_ = betac
    if (i > Burn):
        jb = jb + 1

# Storing the results
if (i >= Burn):
    sbeta.append(beta_.tolist())
    ssigm.append(sigma2_.tolist()
```

3.2.4 Results - Python Code

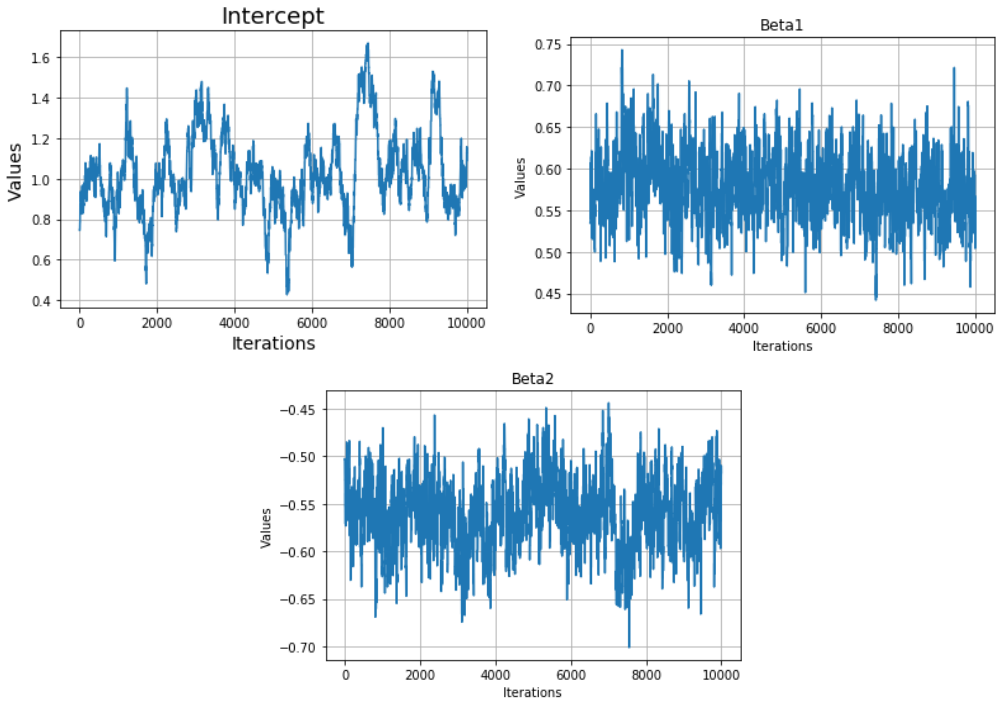

Figure 7: Results for the logistic model using MCMC by hand done in python

The results in Figure 7 show that for the implementation in `python`, the chains for the parameters of the logistic regression are very close to true values. Despite that, we can notice an unconstant behavior for the chain of α (the intercept).

3.3 Beta Regression

Let $\{Y_i\}_{i=1}^n$ be independent and identically distributed random variables and $\mathbf{x}_i = (1, x_{i,1}, ..., x_{i,1})$ a line vector with all covariates of the individual i. We assume that Y_i is distributed according to a Beta distribution, denoted by $Y_i \sim \text{Beta}(\mu_i, \phi)$, where the Beta distribution may be written in the form

$$f(Y_i|\mu_i, \phi) = \frac{\Gamma(\phi)}{\Gamma(\mu_i\phi)\Gamma((1-\mu_i)\phi)} Y_i^{\mu_i\phi-1}(1-Y_i)^{(1-\mu_i)\phi-1},$$

where $0 < Y_i < 1$, $\mathbb{E}(Y_i) = \mu_i$, $\mathbb{V}(Y_i) = \mu_i(1-\mu_i)/(1+\phi)$, $0 < \mu_i < 1$ and $\phi > 0$. Thus, it is possible to model $g(\mu_i) = \mathbf{x}_i\boldsymbol{\beta}$, where $\boldsymbol{\beta}$ is the vector of unknown parameters and $g(\cdot)$ is the link function that maps the unit interval into $\mathbb{R}$, which must be strictly monotonic and twice differentiable (Ferrari and Cribari-Neto (2004)). Considering the link function as logit, i.e., $g(\mu_i) = \log(\mu_i/(1-\mu_i)) = \mathbf{x}_i\boldsymbol{\beta} \Rightarrow \mu_i = \exp\{\mathbf{x}_i\boldsymbol{\beta}\}/(1+\exp\{\mathbf{x}_i\boldsymbol{\beta}\})$, the likelihood function has the form of

$$\begin{aligned} f(\mathbf{y}|\mu_i, \phi) = \prod_{i=1}^n f(Y_i|\mu_i, \phi) &= \prod_{i=1}^n \frac{\Gamma(\phi)}{\Gamma(\mu_i\phi)\Gamma((1-\mu_i)\phi)} Y_i^{\mu_i\phi-1}(1-Y_i)^{(1-\mu_i)\phi-1}, \\ &= \left(\frac{\Gamma(\phi)^n}{\prod_{i=1}^n \Gamma(\mu_i\phi)\Gamma((1-\mu_i)\phi)}\right) \prod_{i=1}^n Y_i^{\mu_i\phi-1} \prod_{i=1}^n (1-Y_i)^{(1-\mu_i)\phi-1}. \end{aligned}$$

To find the posterior conditional distributions for $\boldsymbol{\beta}$ and ϕ, we set the following priors: $\boldsymbol{\beta} \sim N(\mathbf{m}, \mathbf{V})$ and $\phi \sim \text{U}(a, b)$. Firstly, we introduce the posterior conditional distribution for $\boldsymbol{\beta}$.

$$\begin{aligned} f(\boldsymbol{\beta}|\mathbf{y}, \mathbf{X}, \phi) &\propto f(\mathbf{y}|\mu_i, \phi) \times f(\boldsymbol{\beta}), \\ &\propto \frac{1}{\prod_{i=1}^n \Gamma(\mu_i\phi)\Gamma((1-\mu_i)\phi)} \prod_{i=1}^n Y_i^{\mu_i\phi-1} \prod_{i=1}^n (1-Y_i)^{(1-\mu_i)\phi-1} \times \\ &\times \exp\Big\{-\frac{1}{2}(\boldsymbol{\beta}-\mathbf{m})^\top \mathbf{V}^{-1}(\boldsymbol{\beta}-\mathbf{m})\Big\}. \end{aligned}$$

Second, we present the posterior conditional distribution for ϕ, which is proportional to the multiplication of the likelihood function and its prior distribution.

$$\begin{aligned} f(\phi|\mathbf{y}, \mathbf{X}, \phi) &\propto f(\mathbf{y}|\mu_i, \phi) \times f(\phi), \\ &\propto \left(\frac{\Gamma(\phi)^n}{\prod_{i=1}^n \Gamma(\mu_i\phi)\Gamma((1-\mu_i)\phi)}\right) \prod_{i=1}^n Y_i^{\mu_i\phi-1} \prod_{i=1}^n (1-Y_i)^{(1-\mu_i)\phi-1} \times \\ &\times \frac{1}{b-a}\Big\}. \end{aligned}$$

The link https://www.ime.usp.br/~sferrari/beta.pdf contains the paper (Ferrari and Cribari-Neto (2004)). On the pages 4 and 6 there are equations that can be helpful.

3.3.1 R Code

The code below is the implementation of the respective Beta regression model using MCMC and the programming language `R`.

```
for(i in 1:TotIter){

  MCMCBetasC <- mvrnorm(1, MCMCBetasI, mprop)
  mu_I <- func_mu(MCMCBetasI)
  muC  <- func_mu(MCMCBetasC)

  # Computing the ratio stated in the Metropolis-Hastings algorithm
```

```
  ratio_beta = ((- sum(log(gamma(muC * phi_I))) - sum(log(gamma((1 - muC) *
  phi_I))) + sum((muC * phi_I - 1) * log(y)) + sum(((1 - muC) * phi_I - 1) *  log(1 - y)) -
  0.5 * (t(MCMCBetasC - m) %*% V_1 %*%(MCMCBetasC - m))) - (- sum(log(gamma(mu_I * phi_I))) -
  sum(log(gamma((1 - mu_I) * phi_I))) + sum((mu_I * phi_I - 1) * log(y)) + sum(((1 - mu_I) * phi_I - 1)
  log(1 - y)) - 0.5 * (t(MCMCBetasI - m) %*% V_1 %*%(MCMCBetasI - m))))

   # Accept/Reject step for beta
  if(runif(1) < min(1, exp(ratio_beta)))
  {MCMCBetasI <- MCMCBetasC
  j_beta <- j_beta +1}

  # Proposal distribution for phi
  phiC = rgamma(1, (phi_I^2)/var, scale=1/(phi_I / var))

  # Computing the ratio stated in the Metropolis-Hastings algorithm
  ratio_phi = ((N * log(gamma(phiC)) - sum(log(gamma(mu_I * phiC))) -
  sum(log(gamma((1 - mu_I) * phiC))) + sum((mu_I* phiC - 1) * log(y)) +
  sum(((1 - mu_I) * phiC - 1) * log(1 - y)) - log(b - a) +
  dgamma(phi_I, (phiC^2)/var,  scale = 1/(phiC / var))) - (N * log(gamma(phi_I)) -
  sum(log(gamma(mu_I * phi_I)))
  -sum(log(gamma((1 - mu_I) * phi_I))) +sum((mu_I* phi_I - 1) * log(y)) +
  sum(((1 - mu_I) * phi_I - 1) * log(1 - y)) - log(b - a) +
  dgamma(phiC,  (phi_I^2)/var, scale = 1/(phi_I / var))))

  # Accept/Reject step for phi
  if(runif(1) < min(1, exp(ratio_phi)))
  {phi_I <- phiC
  j_phi <- j_phi +1}

  # Saving results
  if (i > BurnIn){
    SaveResults[AuxBurnIn,] <- c(AuxBurnIn, MCMCBetasI, phi_I)
    AuxBurnIn <- AuxBurnIn + 1
  }
}
```

Figure 8: Simulated data for the Beta regression

On Figure 8 we see the simulated data used for the `R` and `python` code for the Beta regression model.

3.3.2 Results - R Code

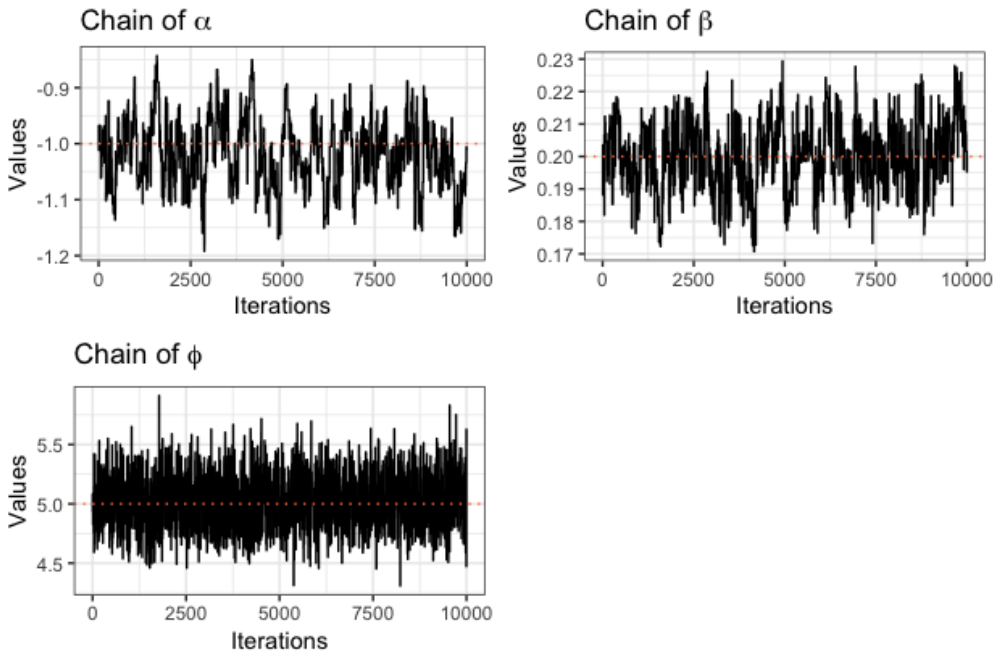

Figure 9: Results for the Beta model using MCMC by hand done in R

Figure 9, by its turn, shows that the chains for the Beta regression model have a very good behaviour with the implementation done in `R`.

3.3.3 Python Code

The code below is the implementation of the respective Beta regression model using MCMC and the programming language `python`.

```
for i in tqdm(range(TotIter)):

    # Proposal distribution for beta (multivariate Normal)
    betac = np.random.multivariate_normal(list(itertools.chain(*beta_)), mprop, 1).T

    mu_ = func_mu(beta_)
    muc = func_mu(betac)

    # Computing the ratio stated in the Metropolis-Hastings algorithm
    ratio_beta = ((- sum(np.log(Gamma(muc * phi_))) - sum(np.log(Gamma((1 - muc) * phi_))) +
    sum((muc * phi_ - 1) * np.log(y)) + sum(((1 - muc) * phi_ - 1) * np.log(1 - y)) - 0.5 *
    (betac - m).T @ V_1 @(betac - m)) - (- sum(np.log(Gamma(mu_ * phi_))) -
    sum(np.log(Gamma((1 - mu_) * phi_))) + sum((mu_ * phi_ - 1) * np.log(y)) +
    sum(((1 - mu_) * phi_ - 1) * np.log(1 - y)) - 0.5 * (beta_ - m).T @ V_1 @(beta_ - m)))

    # Accept/reject step for betac (beta candidate)
    if (np.random.uniform(0, 1, 1) < np.exp(ratio_beta)):
        beta_ = betac
        if (i > Burn):
```

```
        j_beta = j_beta + 1

# Proposal distribution for phi
phic = gamma.rvs((phi_**2)/var, scale=1/(phi_ / var), size=1, random_state=None)

# Computing the ratio stated in the Metropolis-Hastings algorithm
ratio_phi = ((N * np.log(Gamma(phic)) - sum(np.log(Gamma(mu_ * phic))) -
sum(np.log(Gamma((1 - mu_) * phic))) + sum((mu_* phic - 1) * np.log(y)) +
sum(((1 - mu_) * phic - 1) * np.log(1 - y)) - np.log(b-a) +
gamma.pdf(phi_, (phic**2)/var, scale = 1/(phic / var))) - (N * np.log(Gamma(phi_)) -
sum(np.log(Gamma(mu_ * phi_))) - sum(np.log(Gamma((1 - mu_) * phi_))) +
sum((mu_* phi_ - 1) * np.log(y)) + sum(((1 - mu_) * phi_ - 1) *
np.log(1 - y)) - np.log(b-a) + gamma.pdf(phic, (phi_**2)/var, scale = 1/(phi_ / var))))

# Accept/reject step for phic (phi candidate)
if (np.random.uniform(0, 1, 1) < np.exp(ratio_phi)):
    phi_ = phic
    if (i > Burn):
        j_phi = j_phi + 1

# Storing the results
if (i >= Burn):
    sbeta.append(beta_.tolist())
    sphi.append(phi_)
```

3.3.4 Results - Python Code

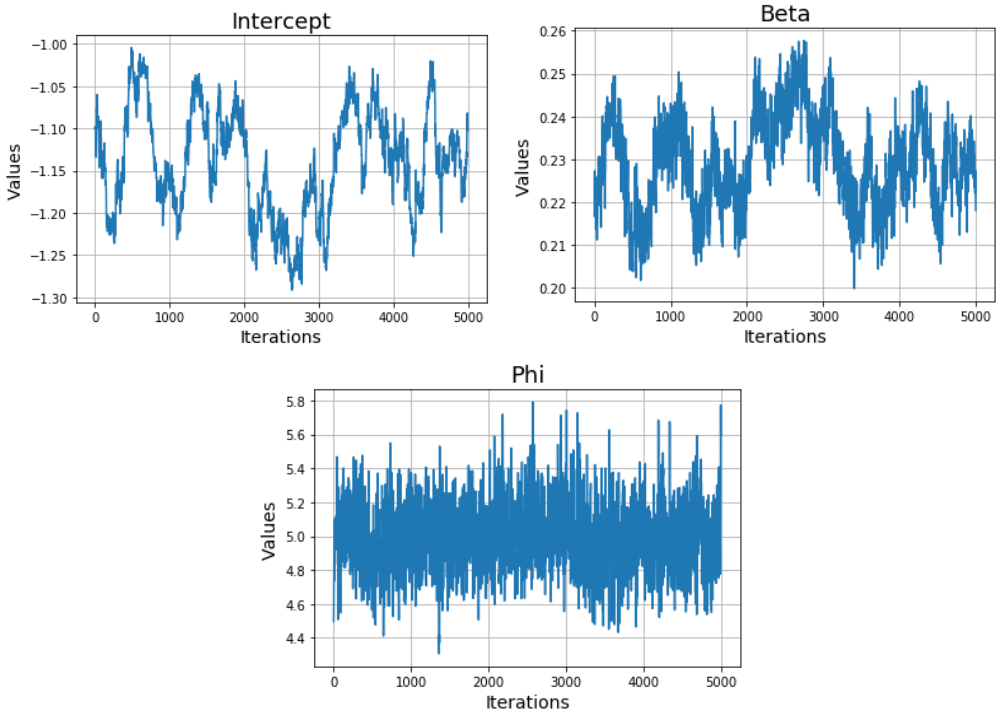

Figure 10: Results for the Beta model using MCMC by hand done in python

The results shown in Figure 10 about the Beta regression model indicate that the chains are far from the true values of the parameters for the implementation done in `python`.

3.4 Comparisons with JAGS

After running the code for both `R` and `python`, we can make some comparisons with the `JAGS` (Just Another Gibbs Sampler) program. The comparisons are performed with the results generated through the interface provided by the `R` package `R2jags`. The interface for `python`, the `pyjags` package, was not considered for pratical reasons. Under the assumption that `JAGS` should have the same results regardless of the interface, we can proceed without much concern.

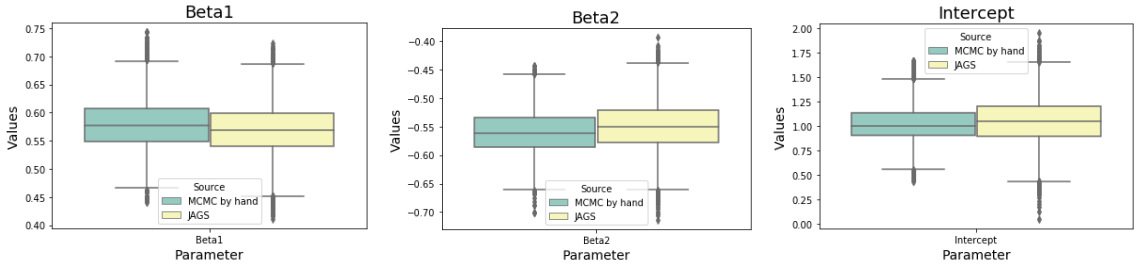

Figure 11: Box-plots of the chains for the parameters of the Logistic regression: comparison between MCMC by hand done in python and JAGS.

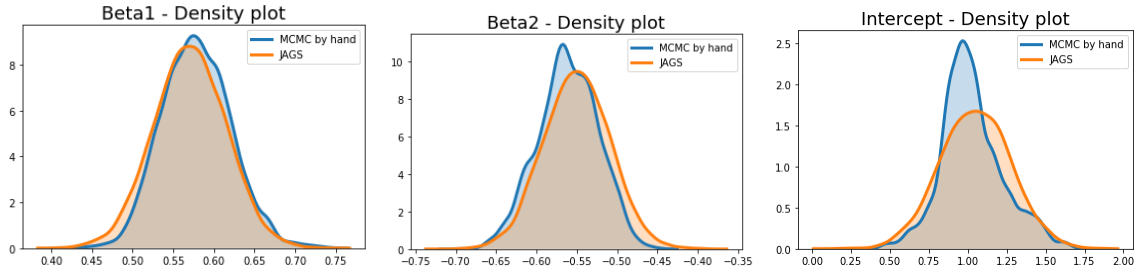

Figure 12: Posteriors for the parameters of the Logistic regression: comparison between MCMC by hand done in python and JAGS.

In Figure 11, we observe that the results obtained with the implementation of MCMC in `python` do not differ much from `JAGS` for the Logistic regression case. For the $\beta = (\beta_1, \beta_2)$, the chains are similar, but `JAGS` is still closer to the true parameters. For the intercept, both results are really good, but `JAGS` values have a bigger variance. The densities shown in Figure 12 represent the porteriors of the parameters.

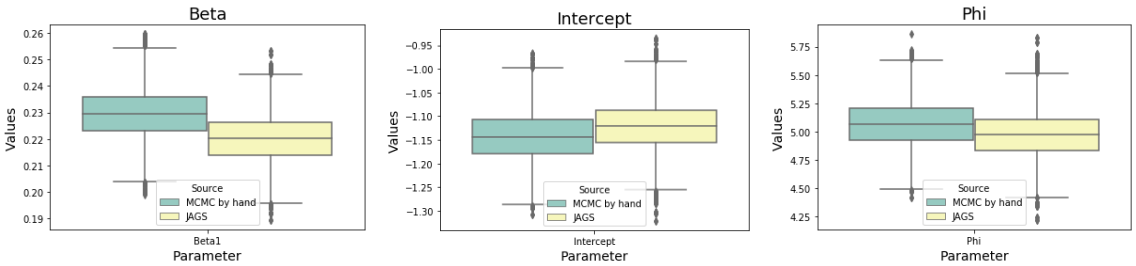

Figure 13: Box-plots of the chains for the parameters of the Beta regression: comparison between MCMC by hand done in python and JAGS

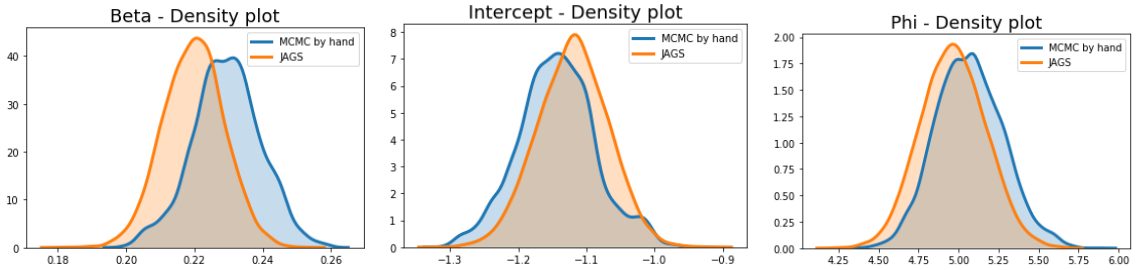

Figure 14: Posteriors for the parameters of the Beta regression: comparison between MCMC by hand done in python and JAGS

As for the Beta regression case, Figure 13 shows us that for β and the intercept, `JAGS` results are closer to the true values. The precision parameter ϕ is very well achieved by both of the methods. In Figure 14 we have the posteriors for these parameters. We can clearly see how `JAGS` is varying less for the first two parameters considered, β and the intercept.

4 Final Remarks

Markov Chain Monte Carlo methods are very well established in the Bayesian literature. Their usefullness is clear when we talk about approximating intractable posteriors. In this project our main contributions were to provide the explanation of some MCMC methods, along with the implementation for three models. The models considered were: linear regression, logistic regression and Beta regression. Gibbs sampling and the metropolis algortithm was used for the first model, while for the last two, only metropolis was implemented.

We were able to work with a models that comprise a wide range of problems, in the sense that they can be applied in a big variety if cases. We deal with both continuous and unlimited data, continuous and limited data and discrete data models. For all of the models, we provided the Bayesian explanation and the code for Gibbs sampling using `R` and `python`. In total we created 6 scripts, 3 for each programming language, being that simulated data was generated to conduct the analysis.

At the end we also worked on a comparison of the results of our implementation and the software `JAGS`. Overall, the software `JAGS` produced similar results to the implementation by hand for the logistic regression case. Except for the intercept, the other regressions coefficients had very close resulting posterior distributions for the two approaches. As for the Beta regression, we observed that `JAGS` had some small advantages, especially for the regression parameters, for which the posteriors differed a little in a favour of `JAGS`. The precision parameters ϕ had good results for both methods.

The implementation of MCMC by hand is both interesting and challenging regarding our ability to understand and code our own models from the beginning to the evaluation of the results. Simulating the correct data has a crucial role here, as well as plugging it in properly to the models. Some models present complicated conditional distributions that usually need some re-arrangements/simplifications in order to avoid a time-consuming implementation. Details such as forgetting a constant or the mispecification of the prior distribution can make a significative difference in the algorithm's convergence depending on the size of the dataset and how precise is the information provided by the prior. Also, the proposal distribution in the Metropolis-Hastings and the pre- and post-burn-in periods must be well specified. For instance, if the acceptance rate is high, strong auto-correlation may be observed. On the other hand, if the acceptance rate is low and the number of iterations for the post-burn-in is not satisfactory, the convergence cannot be reached. Aside from these points, the implementation by hand helped us to have better understanding about both mathematical and computational aspects involved in the Bayesian inference process.

Extensions can easily be made for future models which will require calculating the posterior distributions and manipulating current code. As for possible models, we can mention all the ones already implemented in the `JAGS` examples repository in GitHub. Once we know the code can be wrote in `JAGS`, we also know it can be implemented by hand, once the program is using Markov Chain Monte Carlo methods that were the object of study of this project.

References

Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. I. (2003). An introduction to MCMC for machine learning. *Machine Learning*, 50(1–2):5–43.

Brooks, S., Gelman, A., Jones, G., and Meng, X.-L. (2011). *Handbook of Markov Chain Monte Carlo*. CRC press.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). Stan : A Probabilistic Programming Language. *Journal of Statistical Software*.

Ferrari, S. L. P. and Cribari-Neto, F. (2004). Beta regression for modelling rates and proportions. *Journal of Applied Statistics*.

Gasparini, M., Gilks, W. R., Richardson, S., and Spiegelhalter, D. J. (1997). Markov Chain Monte Carlo in Practice. *Technometrics*.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2014). *Bayesian Data Analysis*. Chapman and Hall/CRC, 3nd ed. edition.

Ntzoufras, I. (2008). *Bayesian Modeling Using WinBUGS*.

Plummer, M. (2003). JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *DSC 2003 Working Papers*.

Thomas, A., Best, N., and Way, R. (2003). WinBUGS User Manual. *Components*.