JAGS: Just Another Gibbs Sampler

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

November 20, 2018

1 Introduction

In the context of Bayesian inference it is common to find situations where it is not possible to obtain the joint posterior distribution in a closed form. For these cases, Markov Chain Monte Carlo (MCMC) methods, such as Gibbs Sampling [1, 2] and Metropolis-Hastings [3, 4], are largely utilized to indirectly sample from the joint posterior distribution through its conditionals. To do so, it is necessary to obtain the conditional distributions and then implement Metropolis and/or Gibbs Sampling. Depending on problem's complexity, hundreds of lines of code might be required.

Some statistical software, based on MCMC, was introduced in order to offer efficient implementation for Bayesian problems without requiring the user for the specification of the conditional's distributions nor to write code. For instance, the software Bayesian inference Using Gibbs Sampling (BUGS) [5] was introduced in 1989 to provide a platform to perform Bayesian analysis from a script. After some years, BUGS started to be developed as WinBUGS [6], where the latter was initially available just for Microsoft Windows users and it was no longer possible to compile the analysis from a script.

Since BUGS was no longer available, other programs have been proposed. For instance, Just Another Gibbs Sampler (JAGS) [7] was introduced in 2003 as an open source software, written in C++. It runs on Linux, Windows, Mac and other versions of Unix system and was inspired by the BUGS functionalities. Its main motivations were to create a program that could be continuously modified/improved and also to create a new flexible software where new Bayesian methodologies could be implemented, tested and compared. The Gibbs Sampler function utilized by JAGS is the Adaptive Rejection Metropolis Sampler (ARMS), which was proposed by [8]. In short, this sampler constructs an envelope of the log of the target density and combined with a Metropolis step has the ability to sample from both univariate log-concave and non-log-concave densities.

Another open source software that performs general Bayesian analysis based on MCMC is Stan [9]. Differently from JAGS, Stan samples from both univariate and multivariate marginal distributions and utilizes Hamiltonian Monte Carlo, which provides a faster convergence to the target distribution by using the gradient of its log. In addition, to verify the convergence of the generated chains, Stan runs many chains setting different initial values and checks whether all the chains have approximately converged to the same value. Similarly, NIMBLE [10] is also an alternative tool, built in R, that performs statistical analysis

Made with Goodnotes

JAGS: Just Another Gibbs Sampler

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

November 20, 2018

1 Introduction

In the context of Bayesian inference it is common to find situations where it is not possible to obtain the joint posterior distribution in a closed form. For these cases, Markov Chain Monte Carlo (MCMC) methods, such as Gibbs Sampling [1, 2] and Metropolis-Hastings [3, 4], are largely utilized to indirectly sample from the joint posterior distribution through its conditionals. To do so, it is necessary to obtain the conditional distributions and then implement Metropolis and/or Gibbs Sampling. Depending on problem's complexity, hundreds of lines of code might be required.

Some statistical software, based on MCMC, was introduced in order to offer efficient implementation for Bayesian problems without requiring the user for the specification of the conditional's distributions nor to write code. For instance, the software Bayesian inference Using Gibbs Sampling (BUGS) [5] was introduced in 1989 to provide a platform to perform Bayesian analysis from a script. After some years, BUGS started to be developed as WinBUGS [6], where the latter was initially available just for Microsoft Windows users and it was no longer possible to compile the analysis from a script.

Since BUGS was no longer available, other programs have been proposed. For instance, Just Another Gibbs Sampler (JAGS) [7] was introduced in 2003 as an open source software, written in C++. It runs on Linux, Windows, Mac and other versions of Unix system and was inspired by the BUGS functionalities. Its main motivations were to create a program that could be continuously modified/improved and also to create a new flexible software where new Bayesian methodologies could be implemented, tested and compared. The Gibbs Sampler function utilized by JAGS is the Adaptive Rejection Metropolis Sampler (ARMS), which was proposed by [8]. In short, this sampler constructs an envelope of the log of the target density and combined with a Metropolis step has the ability to sample from both univariate log-concave and non-log-concave densities.

Another open source software that performs general Bayesian analysis based on MCMC is Stan [9]. Differently from JAGS, Stan samples from both univariate and multivariate marginal distributions and utilizes Hamiltonian Monte Carlo, which provides a faster convergence to the target distribution by using the gradient of its log. In addition, to verify the convergence of the generated chains, Stan runs many chains setting different initial values and checks whether all the chains have approximately converged to the same value. Similarly, NIMBLE [10] is also an alternative tool, built in R, that performs statistical analysis

Made with Goodnotes

applying computationally-intensive methods. Although it is built in R, it also fits models written in the BUGS language and utilizes C++ for compiling.

The main goal of this document is to introduce the usage of JAGS as a software to perform Bayesian statistics without having to implement many lines of codes. JAGS offers an interface script where all the specification related to the data, model, priors and hyper-parameters are carried out. In addition, in order to exemplify its applicability, we provide a set of examples involving linear regression, generalized linear models, generalized linear mixed models and survival models. All the examples are implemented both in R [11] and Python [12] through interfaces provided by the packages R2jags [13] and pyjags [14], respectively.

This document is organized as follows. In the Chapter 2 we introduce JAGS. In Chapter 3 we present some examples using JAGS in R and Python. Chapter 4 describes new models that can also be implemented in JAGS. Finally, in Chapter 5 we present our final remarks and future works.

2 JAGS

JAGS is a program to perform inference for Bayesian Hierarchical models, based on MCMC methods, that was proposed by Martyn Plummer in 2003 as an alternative to the BUGS software (which stopped being developed in 1998, as the BUGS project moved onto WinBUGS). Due to this change, some functionalities were lost and WinBUGS was available only for Microsoft Windows users. Plummer's intention was to clone BUGS in order to create a flexible and open source platform where new methodologies could be explored and compared.

The Gibbs Sampler function of JAGS is ARMS [7], which is flexible for dealing with univariate target densities. For log-concave target distribution, ARMS creates an envelope by using secant lines that successively increases the efficiency of the algorithm whenever a sampled value is rejected. That is, through the secant lines the algorithm increases its acceptance rate, thus demanding less computational time. For the cases where the target density is non-log-concave, secant lines are also utilized, but depending on the shape of the target distribution, the secant lines do not form an envelope. Distributions that belong to the exponential family are log-concave, but non-linear functions and some non-exponential-families are non-log-concave.

The main advantages of JAGS, compared to BUGS, is the programming language and its interfaces with other software, such as R and Python. For instance, JAGS was developed in C++, a more known programming language, while BUGS was programmed in Component Pascal. Also, JAGS can be used via command line, script, Python and R. In addition to providing friendly interfaces, Python and R can also be used to manipulate, analyze and visualize the post-processing results (both R and Python have specific libraries for checking the convergence of the chains).

To illustrate how JAGS works, we present a basic code, below, to fit a Bayesian linear regression. First, we carry out a transformation by applying the logarithm function in the response variable, which is then used as the outcome variable in the model block. JAGS allows data transformation either for covariates or response variable, which can sometimes be useful in order to avoid numerical issues or to reduce skewness. There are functions such as log (logarithm), sqrt (square root) and exp (exponential) that can be used to implement

Made with Goodnotes

it. Also, JAGS allows to explicitly declare the dimensions of the objects/quantities that will be considered in the model, although the declaration is not mandatory. For instance, if the dimension is not declared, then it will be inferred based on the index of the loop.

In the model block, we specify the sampling distribution and the linear predictor. In this case, the response variable is assumed to be normally distributed with a certain mean and variance. The linear predictor has 2 components (`alpha` and `beta`) and their prior distributions are Normals with mean zero and large variance (the Normal distribution in JAGS considers the precision rather than variance). For the `precision`, we set an Inverse Gamma with mean 1 and variance 1000.

```
# Data transformation ------------------------------------------------
data {
    n = dim(y)[1];
    for (i in 1:n) {
      y.trans[i] <- log(y[i])
    }
}
# Model specification ------------------------------------------------
model {

# Declaring the dimensions of the objects
y.trans[n], x[n], mu[n], alpha, beta, precision

    for (i in 1:n) {
        y.trans[i] ~ dnorm(mu[i], precision)
        mu[i] <- alpha + beta * x
    }
# Prior distributions
    alpha ~ dnorm(0.0, 1.0E-3)
    beta ~ dnorm(0.0, 1.0E-3)
    aux <- dgamma(0.001, 0.001)
    precision ~ 1.0/ aux
}
```

Since the model is specified, it is necessary to set the data file, number of chains, burn-in period, number of iterations after the burn-in and which parameters must have their results saved. In addition, the burn-in results will not be saved and the inference will be based only on the remainder samples. Usually, these samples are used to create graphics and descriptive measures of the target distribution, such as mean, median, mode and variance. It is important to bear in mind that MCMC are stochastic methods and each time the model is re-run, the generated samples will be different. To avoid it, is necessary to set a seed for each chain. In the Appendix A we present JAGS codes to fit the model introduced above by using the command line.

Made with Goodnotes

3 Running JAGS in R and Python

In this section we present two interfaces to call JAGS from R and Python. For R, the package **R2jags** [13] will be explored and for Python the package **pyjags** [14].

3.1 R2jags

The package **R2jags** provides many functions to perform Bayesian Hierarchical models in JAGS. This package allows the user to monitor the convergence of an MCMC scheme using Rubin and Gelman statistics and also automatically runs an MCMC setting until its convergence. In addition, it has the ability to parallelize processes for multiple chains. Below we present the needed steps to perform a Bayesian analysis by using **R2jags**.

1. Define the model using the BUGS language in a separate file or enter with the model in the R script.

2. Set the seed, the burn-in period and the parameters that must be saved. Optionally, it is possible to set the initial values for each parameter.

3. If necessary, it is possible to update the model using the update method for jags objects. For instance, if the burn-in was not enough to reach convergence, it is not needed to re-run the model.

4. After the model fitting, it is possible to access and save the outputs using the functions *print*, *plot*, *traceplot* and *pdf*. Also, the package **coda** [15] provides additional convergence diagnostics that help to check if all chains converged by using different tests.

The following two examples we provide in order to illustrate how to perform a Bayesian inference in R utilizing **R2jags**. In the first example we introduce a simple random effect model with no covariates. The second one involves the Beta regression model, which can be applied to situations where the response variable is in the unit interval (0, 1).

3.1.1 Example 1: Random effect model

Consider the random effect model in the following form

$$Y_{ij} = \alpha + z_j + \epsilon_{ij},$$

where $z_j \sim \mathrm{N}(0, \sigma_z^2)$, $\epsilon_{ij} \sim \mathrm{N}(0, \sigma^2)$, Y_{ij} is the random variable associated to the response variable for the observation $i = 1, .., n_j$ in group $j = 1, .., M$, α is the overall mean, z_j is the random effect of the group j and ϵ_{ij} is an error. Hence, we know that $Y_{ij}|\alpha, z_j \sim \mathrm{N}(\alpha+z_j, \sigma^2)$, and since there is no covariate in the model, the aim is to estimate the overall mean α, the random effects z_j, the variance of the random effects σ_z^2 and the overall variance σ^2. To illustrate the applicability of this model, we simulate a dataset of length 1000 with 10 groups, where each group has approximately 100 observations and the same variability. The parameters α and σ^2 were set in 5 and 10, respectively. The random effects were generated from a $\mathrm{N}(0, \sigma_z^2)$, with $\sigma_z^2 = 5$.

Made with Goodnotes

```
# Simulating data ------------------------------------------
set.seed(123)
M = 10 # Number of groups
alpha = 5
sigma = 10
sigma_b = 5
nj = sample(80:110, M, replace = TRUE) # Obs in each group [80, 110]
N = sum(nj)
b = rnorm(M, 0, sigma_b)
group = rep(1:M, times = nj)
y = rnorm(N, mean = alpha + b[group], sd = sigma)
```

The model specification is carried out below. First, we set the sampling distribution and then the priors. For the intercept, a Normal distribution with mean 0 and large variance is considered. For the random effects, we also assume a Normal distribution with mean 0, but for its variance we set a chi-Squared distribution with mean 50 and variance 100. Finally, for the overall variance σ^2 we consider a Weibull distribution with shape and rate parameters equal to 10 and 30, respectively. In the Section 9.2 of the JAGS documentation there is a list of all functions that can be utilized either as sampling distribution or prior.

```
# Jags code to fit the model to the simulated data ------------
model_code = '
model
{
  # Likelihood
  for (i in 1:N) {
    y[i] ~ dnorm(alpha + z[group[i]], sigma^-2)
  }
  # Priors
  alpha ~ dnorm(0, 100^-2)
  for (j in 1:M) {
    z[j] ~ dnorm(0, sigma_z^-2)
  }
  sigma_z ~ dchisqr(50)
  sigma ~ dweib(10, 30)
}
'
```

In order to perform the analysis, it is necessary to set the database as a list and to create a vector with the names of the parameters, which the user has interest in, to save the results. For instance, if the model has a lot of parameters, all of them must be specified as the convergence diagnostics must be carried out for all parameters. After this pre-run setting, the model can be run by calling the function `jags`, which requires the dataset, model specification, number of parallel chains and the number of iterations for the burn-in and post-burn-in periods. Optionally, the number of thinning (`n.thin`) and the starting

Made with Goodnotes

values for each parameter in each chain (`inits`) can be specified. Presented below is the R code with the setting mentioned above.

```
# Setting the seed
set.seed(12345)

# Set up the data
model_data = list(n = N, y = y)

# Choose the parameters to watch
model_pars =  c("alpha", "z", "sigma", "sigma_z")

# Run the model
model_run = jags(data = model_data,
            parameters.to.save = model_pars,       # Parameters to be saved
            model.file=textConnection(model_code), # model specification
            n.chains=4,                            # Number of chains
            n.iter=1000,                           # Iterations Post-burn-in
            n.burnin=200)                          # Iterations for burn-in
```

Running the code above, chains of non-independent samples are generated from the joint posterior distribution through the conditionals for α, z_j, σ^2 and σ_z^2. By using functions from the package **coda**, is possible to visually analyze the convergence of the chains and have descriptive measures of the posterior conditional distributions, such as mean, quantiles and credible intervals. Additionally, there are functions, such as *geweke.diag*, *heidel.diag* and *raftery.diag*, that compute statistical tests in order to automate the convergence diagnostics. These functions can be helpful for problems with many parameters, where the visualization of each trace plot becomes arduous.

3.1.2 Example 2: Beta Regression model

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

$$f(Y_i|p,q) = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} Y_i^p (1-Y_i)^{q-1},$$

where $0 < Y_i < 1$, $p > 0$, $q > 0$ and $\Gamma(\cdot)$ is the gamma function. Usually, in the regression models we desire to model the average of the response variable. In this sense, it is recommended to have a reparameterization in the Beta distribution in order to have a regression structure [16]. Let $\mu = p/(p+q)$ and $q = p+q$. Then, the Beta distribution can be re-written in the form

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

Made with Goodnotes

where $0 < Y_i < 1$, $\mathbb{E}(Y_i) = \mu$, $\mathbb{V}(Y_i) = \mu(1-\mu)/(1+\phi)$, $0 < \mu < 1$ and $\phi > 0$. Thus, it is possible to model $g(\mu) = X_i\boldsymbol{\beta}$, where $g(\cdot)$ is the link function that maps the unit interval into $\mathbb{R}$, which must be strictly monotonic and twice differentiable.

For this example, we also simulated the dataset that was considered in the analysis. The only covariate was generated from the uniform distribution, the sample size was 1000 and the parameters β_0, β_1 and ϕ were set -1, 0.2 and 5, respectively; see Appendix C for the R code. Below we set Beta as the sampling distribution (`dbeta(a,b)`) and the logit as the link function. Then, the linear predictor is written as well as the specification of the prior distributions. We adopt Normal prior distributions for betas and a uniform distribution for ϕ. In JAGS, the Beta regression model is specified as following

```
# Jags code ---------------------------------------------------------
code = '''
model
{
  # Likelihood
  for (t in 1:T) {
  y[t] ~ dbeta(a[t], b[t])
  a[t] <- mu[t] * phi
  b[t] <- (1 - mu[t]) * phi
  logit(mu[t]) <- alpha + beta * x[t]
  }

  # Priors
  alpha ~ dnorm(0, 10^-2)
  beta ~ dnorm(0, 10^-2)
  phi ~ dunif(0, 10)
}
'''
```

After the model specification, further step involves calling the function `jags` in order to set up the model and generate posterior samples. This step we omit here as it is equal to that available in the previous example. Also, it is important to bear in mind that $g(\cdot)$ can be something other than logit. For instance, functions such as probit, which is the cumulative distribution of a standard normal, log-log and complementary log-log can also be used.

3.2 pyjags

The package **pyjags** provides to Python 2 and 3 users an interface to JAGS through a high-level API. The package was created in 2016 by Tomasz Miskol, and since then it has been maintained on his Github page[1]. Besides JAGS, the package requires the C++11 compiler and the **numpy** package. Basically, the R and Python codes are identical with small differences due to the programming language.

[1] `https://github.com/tmiasko/pyjags`. Accessed on 8th November of 2018.

Made with Goodnotes

In this Section we also provide two examples with simulated data in order to show how to perform a Bayesian analysis in Python utilizing **pyjags**. The first example involves the Bayesian linear model, which is usually applied when the response variable is continuous. The second one considers the Logistic regression, often used for binary classification problems.

3.2.1 Example 1: Bayesian linear model

We recall the Bayesian linear model presented in Chapter 2. We simulated the data that was utilized to fit the model. The sample size was 100 and we generated the only variable from the uniform distribution and the parameters were set 2, 3 and 1 for α, β and σ^2, respectively; see the Python code in the Appendix B. In the Bayesian linear model the sampling distribution is a Normal, which was also assumed as prior for α and β. For $\tau = 1/\sigma^2$, it was set an Inverse Gamma with mean 1 and variance 1000.

```
# Jags code to fit the model to the simulated data ---------------
code = '''
model {
    for (i in 1:N) {
        y[i] ~ dnorm(alpha + beta * x[i], tau)
    }
    # Priors
    alpha ~ dnorm(0, 1e-2)
    beta ~ dnorm(0, 1e-2)
    tau <- 1 / sigma^2
    sigma ~ dgamma(0.001, 0.001)
}
'''
```

After the model specification, we set up the model by using the function `pyjags.Model`. Note that it is necessary to specify the model (`code`), dataset, number of chains and the number of samples after the burn-in period. After setting the model, the function `sample` runs the model providing 500 samples as burn-in, 1000 as post-burn-in and saving the results for the parameters α, β and σ^2. As in R, burn-in samples are discarded and the inference is based just on the remainder ones. Besides these arguments, the package **pyjags** also offers options for processing chains in parallel, reading the model from a file, progress bar and running adaptation steps to maximize samplers efficiency [14].

```
# Set up the model
model = pyjags.Model(code, data=dict(x=x, y=y, N=N), chains=4)

# Burn-in and thinning
model.sample(500, vars=[], thin = 2)

# Post-burn-in and saving results
samples = model.sample(1000, vars=['alpha', 'beta', 'sigma'])
```


Made with Goodnotes

3.2.2 Example 1: Logistic regression model

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 $Y_i \sim \text{Binomial}(p_i)$, in the form of

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

where $Y_i \in \{0,1\}$, $\log(\frac{p_i}{1-p_i}) = X_i\boldsymbol{\beta}$, $X_i = (1, x_{i,1}, ..., x_{i,1})$ is the line vector of covariates associated to the individual i and $\boldsymbol{\beta}$ is the vector of unknown parameters. The function $\log(\frac{p_i}{1-p_i})$ is called logit and provides the link between the random variable and its deterministic components (covariates). Below we show how the simulated dataset was built. The sample size was 100 and we generated two variables from the uniform distribution and the parameters were set 1, 0.2 and -0.5 for β_0, β_1 and β_2, respectively.

```
# Simulated data ------------------------------------------------
np.random.seed(123)
np.set_printoptions(precision=2) # digits of precision for floating point
n = 100;
x_1 = np.random.uniform(0, 10, n)
x_2 = np.random.uniform(0, 10, n)
beta_0 = 1
beta_1 = 0.2
beta_2 = -0.5
logit_p = beta_0 + beta_1 * x_1 + beta_2 * x_2
p = np.exp(logit_p)/(1+np.exp(logit_p)) # inverse logit
y = np.random.binomial(1,p,n)
```

Since the simulated dataset is generated, we can write the logistic model in the JAGS language. First, we set Bernoulli as the sampling distribution (`dbin(p,1)`), write the linear predictor applying the logit function and specify the prior distributions for each one of the betas. Note that prior distributions were set with small values of precision, which implies large variance.

```
# Jags code to fit the model to the simulated data ----------------
code = """
model
{
  # Likelihood
  for (t in 1:n) {
  y[t] ~ dbin(p[t], 1)
  logit(p[t]) <- beta_0 + beta_1 * x_1[t] + beta_2 * x_2[t]
  }

  # Priors
```


Made with Goodnotes

```
  beta_0 ~ dnorm(0.0,0.01)
  beta_1 ~ dnorm(0.0,0.01)
  beta_2 ~ dnorm(0.0,0.01)
}
"""
```

As in the previous example, to generate the posterior samples is necessary to call the function `jags` providing the dataset, the parameters of interest, the model as well as the number of iterations for the burn-in and post-burn-in. It is important to remember that each one of the 10 random effects will be estimated and the convergence of their chains must be verified. For instance, if at least one of the chains has not converged, it can indicate that the number of iterations of the burn-in must be increased.

During this project 3 Python scripts[2] were created to perform Bayesian inference by using **pyjags**. The models involve: i) a simple Bayesian linear model; ii) a logistic regression and; iii) Beta regression model. For all three we provide examples by using simulated data as well as convergence diagnosis and some visualizations for conditional densities and trace plots.

4 New models

In this Section we present new models that we implemented in JAGS. First, we introduce the Poisson model, which is frequently utilized for count data where mean and variance are approximately equal. Second, we present an exponential survival model, which can be used in the survival and reliability contexts. Third, we deal with a Gaussian mixture model, which can be useful for cases where the distribution of the response variable is multimodal.

4.1 Poisson Model

A sequence of random variables $\{Y_i\}_{i=1}^n$ is said to have a Poisson distribution with parameter λ if it takes integers values with the probability mass function given by

$$P(Y_i = y_i) = \frac{\exp\{-\lambda\}\lambda^{y_i}}{y_i!},$$

where $\lambda > 0$. The mean of the distribution can be obtained as

$$\begin{aligned} E(Y_i) &= \sum_{i=1}^{\infty} \frac{\exp\{-\lambda\}\lambda^{Y_i}}{Y_i!} Y_i \\ &= \exp\{-\lambda\} \sum_{i=1}^{\infty} \frac{\lambda^{Y_i}}{Y_i!} Y_i \\ &= \lambda \exp\{-\lambda\} \sum_{i=1}^{\infty} \frac{\lambda^{Y_i-1}}{Y_i - 1!} \\ &= \lambda \exp\{-\lambda\} \exp\{\lambda\} = \lambda. \end{aligned}$$

[2]The scripts are available in `https://github.com/andrewcparnell/jags_examples/tree/master/jags_scripts`

Made with Goodnotes

This mean can be modelled via a link function passed in a systematic component. The systematic component has the same form as the one for linear regression, that is

$$\lambda_i = \beta_0 + \beta_1\phi(x_{i1}) + \cdots + \beta_d\phi(x_{id}),$$

where $\phi(\cdot)$ is an one-to-one function that can be applied to avoid numerical issues or to reduce skewness. However, the equation presented above may produce values between $-\infty$ and ∞, possibly outside the boundaries of λ. In order to produce only appropriate values for λ, we commonly use a link function, which links the regression equation to a function of λ. This function needs to be unlimited in order to satisfy the regression equation. For the Poisson regression case, the most widely used link function is the logarithm, resulting in a equation that has the form

$$\begin{aligned}\log(\hat{\lambda}_i) &= \beta_0 + \beta_1\phi(x_{i1}) + \cdots + \beta_d\phi(x_{id}),\\ \hat{\lambda}_i &= \exp(\beta_0 + \beta_1\phi(x_{i1}) + \cdots + \beta_d\phi(x_{id})).\end{aligned}$$

To extend this model for the Bayesian approach, it is only necessary to attribute prior distributions for the regression parameters $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_d)^\top$. Usually, the prior is taken to be a Multivariate Normal Distribution centered at 0, or

$$\boldsymbol{\beta} \sim N_d(0, \boldsymbol{\Sigma}_d).$$

The posterior distribution of the parameters, given the data $\{\mathbf{X}, \mathbf{y}\}$, has the form

$$P(\boldsymbol{\beta}|\mathbf{X}, \mathbf{y}) = \frac{P(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta})P(\boldsymbol{\beta})}{\int P(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta})P(\boldsymbol{\beta})d\boldsymbol{\beta}},$$

where $\mathbf{X} = (\mathbf{x_1}, ..., \mathbf{x_n})^\top$ is the design matrix, $\mathbf{x_i} = (1, x_{i1}, x_{i2}, , x_{id})$ is a line vector and $\mathbf{y} = (y_1, ..., y_n)^\top$ is a column vector. As there is a lack of conjugacy between the Poisson likelihood and the Normal prior, there is no closed-form for the expression of this posterior distribution. In this situation, an approximate posterior can be used. First, we need to approximate the likelihood of Y_i as

$$\begin{aligned}P(Y_i = y_i|\mathbf{X}, \boldsymbol{\beta}) &= \prod_{i=1}^{n} \frac{1}{y_i} N(\mu(\mathbf{x_i})|\log(Y_i), y_i^{-1}),\\ &= \frac{|\Sigma_y|^{\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} \exp\Big\{ -\frac{1}{2}||\mathbf{X}^\top\boldsymbol{\beta} - \mathbf{t}||^2_{\Sigma_y}\Big\},\end{aligned}$$

where $\Sigma_\mathbf{y} = diag\Big(\Big[\frac{1}{y_1}, \ldots, \frac{1}{y_n}\Big]\Big)$, $\mathbf{t} = \log(\mathbf{y})$ is the element-wise log of y_i and $\mu(\mathbf{x_i}) = \beta_0 + \beta_1\phi(x_{i1}) + \cdots + \beta_d\phi(x_{id})$. By using Bayes' theorem, we can then find that

$$\begin{aligned}\log(\boldsymbol{\beta}|\mathbf{y}, \mathbf{X}) &\propto \log(p(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta})) + \log(p(\boldsymbol{\beta})),\\ &\approx -\frac{1}{2}||\mathbf{X}^\top\boldsymbol{\beta} - \mathbf{t}||^2_{\Sigma_y}|| + -\frac{1}{2}||\boldsymbol{\beta}||^2_{\Sigma_p}.\end{aligned}$$

With this result, expanding the norm term and completing the square, the posterior distribution is approximately Gaussian, with the form

Made with Goodnotes

$$P(\boldsymbol{\beta}|\mathbf{X}, \mathbf{y}) \approx N(\boldsymbol{\beta}|\hat{\boldsymbol{\lambda}}_{\boldsymbol{\beta}}, \hat{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}}),$$

where

$$\hat{\boldsymbol{\lambda}}_{\boldsymbol{\beta}} = (\mathbf{X}\boldsymbol{\Sigma}_y^{-1}\mathbf{X}^\top + \boldsymbol{\Sigma}_n^{-1})^{-1}\mathbf{X}\boldsymbol{\Sigma}_n^{-1}t,$$
$$\hat{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}} = (\mathbf{X}\boldsymbol{\Sigma}_y^{-1}\mathbf{X}^\top + \boldsymbol{\Sigma}_n^{-1})^{-1}.$$

```
# Jags code to fit the model to the simulated data
model_code = '
model
{
  # Likelihood
  for (i in 1:T) {
    y[i] ~ dpois(p[i])
    log(p[i]) <- alpha + beta_1 * x_1[i] + beta_2 * x_2[i]
  }
  # Priors
  alpha ~ dnorm(0.0, 0.01)
  beta_1 ~ dnorm(0.0, 0.01)
  beta_2 ~ dnorm(0.0, 0.01)
}
'
```

4.2 Survival Model

In survival analysis we are usually interested in modelling the time until a certain event occurs - i.e., the "failure". To define it, let T be a random variable representing the survival times of individuals in some population. The probability density function of T can be written as

$$F(t) = P(T \leq t) = \int_0^t f(u)du.$$

The function that gives the probability of an individual surviving, until the time t, is the survivor function, and it can be described as

$$S(t) = 1 - F(t) = P(T > t).$$

Survival data is also often censored, in the sense that the survival times are only known for a portion of the individuals. The other portion is only known to be higher than some values, or, in other words, that the actual time of failure is not known for some reason (e.g. end of the study). In this case, the likelihood is written as

$$L(t) = \prod_{i=1}^{n} [f(t_i)]^{\delta_i} [S(t_i)]^{1-\delta_i},$$

where δ_i is the indicator variable that takes 1 for the failures and 0 for censored observations. The likelihood is described this way because for the non-censored data, the distribution

Made with Goodnotes

remains the same, but when it gets censored, the interest is only in the probability of t_i being smaller than T, or the cumulative probability until the time t_i.

Some models can be proposed for the failure time. The Exponential regression model can be generalized to obtain a regression model by allowing the failure time to be a function of the $\boldsymbol{\beta}\phi(\mathbf{x_i})$. First, let us consider that the hazard, at time t, for an individual with covariates $\mathbf{x_i}$ can be written as

$$\begin{aligned}\lambda_i(t;\mathbf{x_i}) &= \lambda(Z(\mathbf{x_i})),\\ &= \lambda_0 \exp\{\boldsymbol{\beta}\phi(\mathbf{x_i})\},\end{aligned}$$

where λ_0 is the baseline risk for the individuals assumed to be constant. As for the Exponential distribution, its form is

$$f(t) = \frac{1}{\alpha}\exp\Big\{-\frac{t}{\alpha}\Big\}, \quad \alpha > 0.$$

Using the hazard function as $\frac{1}{\alpha}$, we easily obtain the conditional density function of T given $\mathbf{X}$ as

$$f(t|\mathbf{X}) = \lambda_0 \exp\{\boldsymbol{\beta}\phi(\mathbf{X})\}\exp\{-t\lambda_0 \exp\{\boldsymbol{\beta}\phi(\mathbf{X})\}\}.$$

This means that we are assuming $\lambda(t;\mathbf{x_i})$ as the rate for the occurrence of failure times and we are interested in estimating it. To extend this model for the Bayesian case, we can attribute priors for the vector of parameters $\boldsymbol{\beta}$, usually as

$$\boldsymbol{\beta} \sim N_d(0, \boldsymbol{\Sigma}_d).$$

```
# Jags code to fit the model to the simulated data
model_code = '
model
{
  # Likelihood
  for (i in 1:T) {
    mu[i] = exp(beta_1 * x_1[i] + beta_2 * x_2[i])
    t[i] ~ dexp(mu[i] * lambda_0)

  }
  # Priors
  lambda_0 ~ dgamma(1, 1)
  beta_1 ~ dnorm(0.0, 0.01)
  beta_2 ~ dnorm(0.0, 0.01)
}
'
```

4.3 Gaussian Mixtures

Mixture distributions are used when the measurements of a random variable are taken under two different conditions. The famous example is about human height: if the data is collected

Made with Goodnotes

without differentiating between men and women, the resulting observations will have a bimodal distribution. The bimodal distribution indicates that we are actually dealing with a mixture that is a sum of two weighted distributions.

To set up and compute a mixture model, we introduce the vector of unobserved indicators $\mathbf{z}$, which are random variables specifying the mixture component from which each particular observation is drawn. The mixture model is viewed hierarchically: the observations y are modeled conditionally on the vector $\mathbf{z}$, having $\mathbf{z}$ itself a probabilistic specification. The representation of this configuration can be done as

$$p(Y_i = y_i|\boldsymbol{\theta}, \boldsymbol{\lambda}) = \lambda_1 f_1(y_i|\theta_i) + \cdots + \lambda_H f_H(y_i|\theta_H),$$

where $\boldsymbol{\theta} = (\theta_1, ..., \theta_H)$ the vector $\boldsymbol{\lambda} = (\lambda_1, ..., \lambda_H)$ represents the proportions of the population taken as being drawn from each $f_H(y_i|\theta_H)$ distribution, for $h = 1, \ldots, H$, also that $\sum_{h=1}^{H} \lambda_h = 1$. Usually, the mixture components are assumed to be part of the same parametric family, such as the Gaussian, but with different parameter vectors. The unobserved variables are written as

$$z_{ih} = \begin{cases} 1, & \text{of the } i\text{th unite is drawn from the } h\text{th mixture component} \\ 0, & \text{otherwise} \end{cases}$$

In this way, given $\boldsymbol{\lambda}$, the distribution of each vector $\mathbf{z_i} = (z_{i1}, \ldots, z_{iH})$ is Multinomial$(1; \lambda_1, \ldots, \lambda_H)$. The mixture parameters $\boldsymbol{\lambda}$ are thought as the hyperparameters determining the distribution fo $\mathbf{z}$. The joint distribution of $\mathbf{y}$ and $\mathbf{z}$, conditioned on the parameters, is

$$p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta}, \boldsymbol{\lambda}) = p(\mathbf{y}|\mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z}|\boldsymbol{\lambda}) = \prod_{i=1}^{n} \prod_{h=1}^{H} (\lambda_h f(y_i|\theta_h))^{z_{ih}},$$

where $\mathbf{z} = (\mathbf{z_1}, ..., \mathbf{z_n})$ with just one z_{ih} equaling 1 for each i. The number of mixture components is assumed to be known and fixed. For the Gaussian mixtures, the $f(y_i|\theta_h)$ are assumed to be Gaussian with mean and dispersion being $\boldsymbol{\theta_i} = (\mu_i, \sigma_i^2)$. We wish to make inference about the parameters $\boldsymbol{\lambda}$ and $\boldsymbol{\theta} = (\mu, \sigma^2)$ by attributing their probability distributions. The natural conjugate prior for $\boldsymbol{\lambda}$ is the Dirichlet$(\alpha_1, \ldots, \alpha_H)$, where α_i is the concentration parameter with $\alpha_i > 0$. As for $\boldsymbol{\theta_i} = (\mu_i, \sigma_i^2)$, we attribute as prior a Gamma$(a/2, b/2)$ for σ_i^2 and a Gaussian(μ_i, σ_i) for $\mu_i|\sigma_i^2$.

```
model_code = '
model {
  # Likelihood:
  for(i in 1:N) {
    y[i] ~ dnorm(mu[i] , 1/sigma_inv)
    mu[i] <- mu_clust[clust[i]]
    clust[i] ~ dcat(lambda_clust[1:Nclust])
  }
  # Prior:
  sigma_inv ~ dgamma( 0.01 ,0.01)
  mu_clust[1] ~ dnorm(0, 10)
```


Made with Goodnotes

```
  mu_clust[2] ~ dnorm(5, 10)

  lambda_clust[1:Nclust] ~ ddirch(ones)
}
'
```

5 Final remarks

The software Just Another Gibbs Sampler (JAGS) is a flexible open source software that performs Bayesian inference, based on MCMC methods, without requiring the user advanced knowledge in programming language. However, there are a few online examples that provide both the code and the mathematical details about the model that is being used. This project aimed to provide a set of R and Python scripts to perform general Bayesian analysis through JAGS, considering both simulated and real datasets.

To carry out a Bayesian analysis by using JAGS, it is basically necessary to specify the sampling distribution and the prior for each parameter. It is important to highlight that the conditional distributions do not need to be specified. After the model specification, the burn-in and post-burn-in must be set and then the results will be generated. For the post-processing results, packages available for R and Python help to check whether the chains have converged as well as to generate descriptive measure that summarize the posterior conditional densities. The packages **R2jags** and **pyjags** were introduced to provide interfaces for dealing with JAGS in R and Python, respectively.

In this project our main contributions were to add mathematical details and provide real dataset examples for five R scripts. The following scripts were considered: the random effect model, Multivariate Normal model, Beta regression, time series Beta Auto-Regressive model of order 1 and Mixture model. For Python, we created 3 scripts considering the simple linear regression, logistic regression and Beta regression. In all of them were included both simulated and real examples.

This project was an interesting exercise in learning how to code up models using JAGS and then applying the models to real datasets. Formatting the data correctly was very important and errors, such as "dimensions mismatch" and "nodes are inconsistent with parameters", were common in our short experience of using JAGS in R. Also, finding equivalent **pyjags** functions for those used in the **R2jags** was not simple. For example, the **R2jags** package has a utility for displaying trace plots of the BUGS object, which displays a plot of iterations versus sampled values for each variable in the chain, with a separate plot per variable. This can be achieved in one line of code using **R2jags**. However, the **pyjags** library does not contain the same functionality. To achieve the same result, using **pyjags**, another library must be called. In this case, the **Pandas** library was used. However, the code required to display the plots is far more substantial than using **R2jags**. Aside from this issue, the basic formatting of the code is very similar to that in **R2jags** and, as such, users relocating from R to Python should feel familiar with the code.

The project was challenging in the sense that it demanded efforts not only related to theory but also about the software, JAGS, which was new for the whole group. Even so, the group managed to achieve the goals of the project, in time and with quality. The main

Made with Goodnotes

achievements were the familiarization with JAGS in both, R and Python, and getting more experience about how Bayesian inference can be performed for different models.

New models to be written include new models from the exponential family (GLMs), multivariate regression models (e.g. T-Student and Normal), models for Geostatistical and Spatial data, more complex models for survival analysis and other mixed effects models.

Made with Goodnotes

Appendices

A JAGS on command line

To run on command line the example involving Bayesian linear regression presented in Section 2, a script file must invoke in the following form.

```
jags <script file>
```

The script must contain the following information:

```
model       in "BayLinRegr.txt"          # model specification
data        in "Simulation.txt"          # Database
compile, nchains(4)                      # Number of chains
parameter   in "SetParamet.txt"          # Setting the seed for each chain
initialize                               # Initializing the model
update 1000                              # Burn-in period
monitor alpha                            # Save results for alpha
monitor beta1                            # Save results for beta1
monitor beta2                            # Save results for beta2
monitor precision                        # Save results for precision
update 10000                             # Post-burn-in period
```

The file `BayLinRegr.txt` must contain the model specification in the form presented in the Chapter 2. The `Simulation.txt` is the dataset file containing the response variable and covariates. Finally, the file `SetParamet.txt` contains the initial parameters values and/or the number of seeds for each chain.

B Simulated data for Bayesian linear model

```
np.random.seed(123)
np.set_printoptions(precision=1) # Precision for floating point output
N = 100
alpha = 2
beta = 3
sigma = 1
x = np.random.uniform(0, 10, size=N)
y = np.random.normal(alpha + x*beta, sigma, size=N)
```

C Simulated data for Beta regression model

```
set.seed(123)
N = 100
alpha = -1
```


Made with Goodnotes

```
beta = 0.2
phi = 5
logit_mu = alpha + beta * x
mu = inv.logit(logit_mu)
a = mu * phi
b = (1 - mu) * phi
y = rbeta(N, a, b)
```


Made with Goodnotes

References

[1] S. Geman, D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. IEEE Transactions Pattern Analysis and Machine Intelligence. 1984; 6:721–741.

[2] A. Gelfand, A. Smith. Sampling based approaches to calculating marginal densities. Journal of American Statistical Association. 1990; 85:398–409 Springer, New York.

[3] N. Metropolis, A. Rosenbluth, M. Teller, E. Teller. Equations of state calculations by fast computing machines. Journal of Chemistry and Physics. 1953;1087:1091–21.

[4] W. Hastings. Monte Carlo sampling using Markov chains and their applications. Biometrika. 1970; 57: 97-109.

[5] D. Spiegelhalter, A. Thomas, N. Best, D Lunn. WinBUGS 1.4 manual, 2002b

[6] D. Lunn, A. Thomas, N. Best, D. Spiegelhalter. WinBUGS a Bayesian modelling framework: concepts, structure, and extensibility. Statistics and Computing. 2000. 10:325337.

[7] Martyn Plummer. JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling. Proceedings of the 3rd International Workshop on Distributed Statistical Computing. 2003. March 202. Vienna, Austria. ISSN 1609-395X.

[8] W. Gilks, N. Best, K. Tan. Adaptive rejection Metropolis sampling. Statistics and Computing. 1995. Applied Statistics, 44, 455-472.

[9] Stan Development Team. Python: A dynamic, open source programming language. Stan Modeling Language Users Guide and Reference Manual, Version 2.18.0. http://mc-stan.org

[10] NIMBLE Development Team. NIMBLE User Manual. R package manual version 0.6-12. 2018. doi: 10.5281/zenodo.1322114. https://r-nimble.org.

[11] R Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria. 2018. url: http://www.R-project.org/

[12] Python Core Team. Python: A dynamic, open source programming language. Python Software Foundation. 2018. URL https://www.python.org/.

[13] S. Yu-Sung, Y. Masanao. R2jags: Using R to Run "JAGS". R package version 0.5-7. 2015. URL https://CRAN.R-project.org/package=R2jags.

[14] T. Miasko. pyjags Documentation. Reference Manual, Version 1.2.2. https://media.readthedocs.org/pdf/pyjags/latest/pyjags.pdf

[15] M. Plummer, N. Best, K. Cowles, K. Vines. CODA: convergence diagnosis and output analysis for MCMC. R news. 2006;6(1):7-11.

Made with Goodnotes

[16] S. Ferrari, F. Cribari-Neto. Beta regression for modelling rates and proportions. Journal of Applied Statistics. 2004;31(7):799-815.

[17] D. Denison, C. Christopher, B. Mallick, A. Smith. Bayesian Methods for Nonlinear Classification and Regression. 2002. Wiley.

[18] A. Gelman, J. Carlin, H. Stern, D. Rubin. Bayesian Data Analysis. 2004. 2nd ed. Chapman; Hall/CRC.

[19] J. Kalbfleisch and R. Prentice. The Statistical Analysis of Failure Time Data. 2002. 2nd ed. John Wiley and Sons.

[20] I. Ntzoufras. Bayesian Modeling Using WinBUGS. 2008. doi:10.1002/9780470434567.

[21] A. Royle, R. Dorazio. Hierarchical Modeling and Inference in Ecology: The Analysis of Data from Populations, Metapopulations and Communities. 2008. Elsevier.

Made with Goodnotes