Using JAGS in R with the rjags package

Andrew O. Finley

March 6, 2013

1 The ordinary linear regression model

We make use of several libraries in the following example session, including:

- `library(coda)`
- `library(rjags)`

Let's begin by considering the ordinary linear regression model. Here, we are interested in explaining the variability in *outcome* or *response* variable y given a set of *predictors* or *covariates* that were observed together at n sample units. For the i-th observation we can write the model as

$$y_i = \boldsymbol{x}_i'\boldsymbol{\beta} + \epsilon_i, \tag{1}$$

where the $\boldsymbol{x}_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,p})'$ is a $p \times 1$ vector (with x_{i0} set to one, i.e., the intercept), $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)'$ is a $p \times 1$ vector of regression coefficients, and error term ϵ_i is assumed to follow a Normal distribution with mean zero and variance σ^2, expressed as $\epsilon_i \overset{\text{i.i.d.}}{\sim} N(0, \sigma^2)$. The data model, i.e., collecting over the n observations, is expressed as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{2}$$

where $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)'$ is a $n \times 1$ vector, $\boldsymbol{X}$ is the $n \times p$ matrix with $\boldsymbol{x}_i'$ set as the i-th row, and $\boldsymbol{\epsilon}$ is the $n \times 1$ error vector distributed $N(\boldsymbol{0}, \sigma^2 I)$ with zero vector $\boldsymbol{0}$ and $n \times n$ idenity matrix I.

It is often helpful, especially when we're formulating BUGS or JAGS model statements, to write this model as

$$y_i \sim N(\boldsymbol{x}_i'\boldsymbol{\beta}, \sigma^2), \text{ for } i = 1, \ldots, n, \tag{3}$$

or collected over the n observations the multivariate representation is

$$\boldsymbol{y} \sim N(\boldsymbol{X}\boldsymbol{\beta}, \sigma^2 I). \tag{4}$$

2 JAGS in a nutshell

Paraphrased from the users manual, JAGS (Just Another Gibbs Sampler) is a program for the analysis of Bayesian models using Markov chain Monte Carlo (MCMC) which is not wholly unlike OpenBUGS (`http://www.openbugs.info`). JAGS was written with three aims in mind: 1) to have an engine for the BUGS language that runs on Unix; 2) to be extensible, allowing users to write their own functions, distributions, and samplers, and; 3) to be a platform for experimentation with ideas in Bayesian modeling.

JAGS is designed to work closely with the R language and environment for statistical computation and graphics (`http://www.r-project.org`). You will find it useful to install the R coda package analyze the output. You can also use the rjags package to work directly with JAGS from within R as illustrated below.

2.1 Running a JAGS model from R

Running a JAGS model refers to generating samples from the posterior distribution of the model parameters. This takes place in five steps:

1. Definition of the model
2. Compilation
3. Initialization
4. Adaptation and burn-in
5. Monitoring

The subsequent stages of analysis are done outside of JAGS: convergence diagnostics, model assessment, and summarizing the samples must be done using other packages more suited to this task. There are several R packages designed for analyzing MCMC output, and JAGS can be used from within R using the rjags package.

3 Intercept only model

Let's begin by simulating some data from model (1).

```
> set.seed(1)
> n <- 100
> beta.0 <- 0
> sigma.sq <- 5
> y <- rnorm(n, beta.0, sqrt(sigma.sq))
```

We will want to sample from the posterior distributions of `beta.0` and `sigma.sq`, which correspond to β_0 and σ^2 in model (1). To complete the Bayesian specification, we need to choose prior distributions for each parameter. A customary prior for β's is Normal and we write $\beta_0 \sim N(\mu_{\beta_0}, \sigma^2_{\beta_0})$ (for this model, one might also use a *flat* prior). Assuming we don't have much prior information about β_0, reasonable *vague* or *non-informative* hyper-parameters would be $\mu_{\beta_0} = 0$ and $\sigma^2_{\beta_0} = 100000$. **Warning**—distribution definitions in JAGS and BUGS languages differ from those used in R, e.g., R's `dnorm` takes a standard deviation for the dispersion parameter, whereas, JAGS and BUGS take the precision which is the inverse of the variance parameter. On this side of the pond, most of us are used to working with a variance or standard deviation, not the precision. With this in mind, here is the content of the JAGS model file *ex-1.jag*.

```
model{
  for (i in 1:n){
    y[i] ~ dnorm(beta.0, tau.sq)
  }

  beta.0 ~ dnorm(0, 0.0001)
  tau.sq <- 1/sigma.sq
  sigma.sq ~ dunif(0, 100)
}
```

In every model specification file, you have to start out by telling JAGS that you're specifying a model. Then you set up the model for every single data point using a `for` loop. Here, we say that `y[i]` is distributed normally (hence the `y~dnorm` call) with mean `beta.0` and precision `tau.sq`. Then we specify our priors for `beta.0` and `tau.sq`, which are meant to be constant across the loop. We tell JAGS that `beta.0` is distributed normally with mean 0 and standard deviation 1000. Then we specify `tau.sq` in a round-about way. We say that `tau.sq` is a deterministic function (hence the deterministic `<-` instead of the distributional ~) of

sigma.sq. In this way we can put our prior on sigma.sq. We could of course skip all of this and just put the prior directly on tau.sq, but I generally have a better feel for the distribution of a variance as opposed to a precision parameter, especially when we consider priors other than uniform, e.g., Gamma, log-Normal, inverse-Gamma, chi-squared (see, e.g., Gelman 2006, for prior suggestions for variance parameters).

Once the JAGS model is complete, the rest of the work happens in R. In the code below, we define those data elements that are referenced in the JAGS model and initial values for the parameters. Recall, JAGS will take a look at the model and determine an appropriate MCMC sampler, e.g., Gibbs, Metropolis-Hastings, slice, or other specialized sampling algorithm. All of these samplers require some initial values for the parameters. In some cases, JAGS can guess starting values on its own, but it is typically best to provide reasonable values—especially for complex models. The JAGS model file, data objects, and initial values are passed to the jags.model function, which constructs a jags model object. Here too, we specify the number of MCMC chains via the n.chains argument. The final argument in the function call below, n.adapt, specifies the number of initial samples during which the samplers adapt behavior to maximize efficiency (e.g., this would be a step size adjustments if a Metropolis-Hastings random walk algorithm is used). The sequence of samples generated during this adaptive phase is not a Markov chain, and therefore should not be included in the resulting posterior samples.

```
> data <- list(y = y, n = n)
> inits <- list(beta.0 = 0, sigma.sq = 1)
> jags.m <- jags.model(file = "ex-1.jag", data = data,
+     inits = inits, n.chains = 3, n.adapt = 100)

Compiling model graph
   Resolving undeclared variables
   Allocating nodes
   Graph Size: 108

Initializing model
```

Upon calling the jags.model function, the underlying JAGS program compiles the model, checks for any undefined variables and parameters, and gets the model ready to collect posterior samples. These checks are written to the terminal as seen in the call above.

We often specify models with many parameters, e.g., in random effects models we might have thousands. It can be computationally expensive to explicitly collect posterior samples from all of these parameters—referred to as *monitoring* in JAGS and BUGS documentation. Therefore, we explicitly specify which parameters should be monitored. Then, given the jags.model object, the desired number of samples are collected via a call to the coda.samples function and, as shown below, stored in a coda object.

```
> params <- c("beta.0", "sigma.sq")
> samps <- coda.samples(jags.m, params, n.iter = 2000)
> plot(samps)
```

Figure 1 illustrates the posterior samples from 2000 MCMC iterations. Summaries of these samples, after suitable *burn-in*, are then used for parameter inference. As show in the code and output below, the call to summary produces commonly assessed summary statistics. I generally look at the median and lower/upper 95% credible intervals. Importantly, given the posterior samples, we can look at any function of the posterior distribution. This is often handy when the model is fit using some transformation of a given variable, but our interest is in the parameter value associated with the non-transformed variable. For example, say a covariate was log transformed to improve convergence or to meet some model assumptions. Then, inference about the impact of the non-transformed covariate is obtained by simply exponentiating each posterior sample prior to summarization (life is not so simple for frequentists).

```
> burn.in <- 1000
> summary(window(samps, start = burn.in))
```

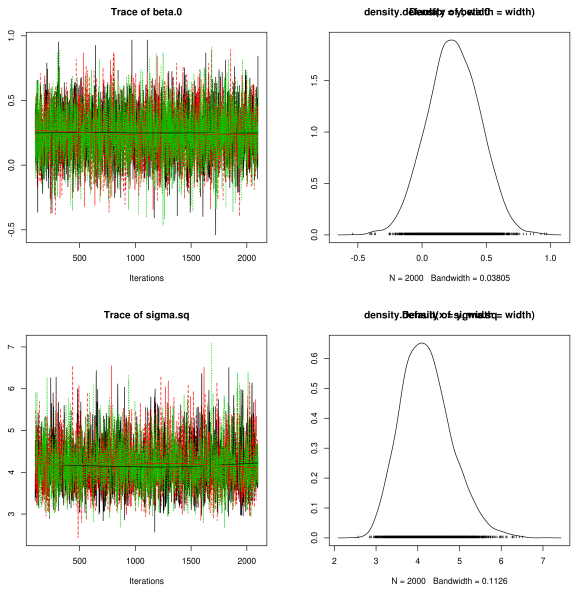

Figure 1: Posterior sample chain and density plots for β_0 and σ^2.

```
Iterations = 1000:2100
Thinning interval = 1
Number of chains = 3
Sample size per chain = 1101

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

           Mean     SD Naive SE Time-series SE
beta.0   0.2433 0.2044 0.003557       0.003403
sigma.sq 4.2047 0.6161 0.010720       0.015649

2. Quantiles for each variable:

            2.5%    25%    50%    75% 97.5%
beta.0   -0.1501 0.1057 0.2433 0.3818 0.642
sigma.sq  3.1579 3.7624 4.1383 4.5740 5.573

> round(summary(window(samps, start = burn.in))$quantiles[,
+     c(3, 1, 5)], 2)

          50%  2.5% 97.5%
beta.0   0.24 -0.15  0.64
sigma.sq 4.14  3.16  5.57
```

Parameter inference from posterior samples is only valid when the MCMC chains, associated with the given parameter of interest, have converged. Gelman (2004) provides a detailed discussion on convergence diagnostics. At minimum we should check that the chains are mixing (assuming dispersed starting values were used to initialize the chains) using the trace plots, e.g., Figure 1, and consider a convergence a rule such

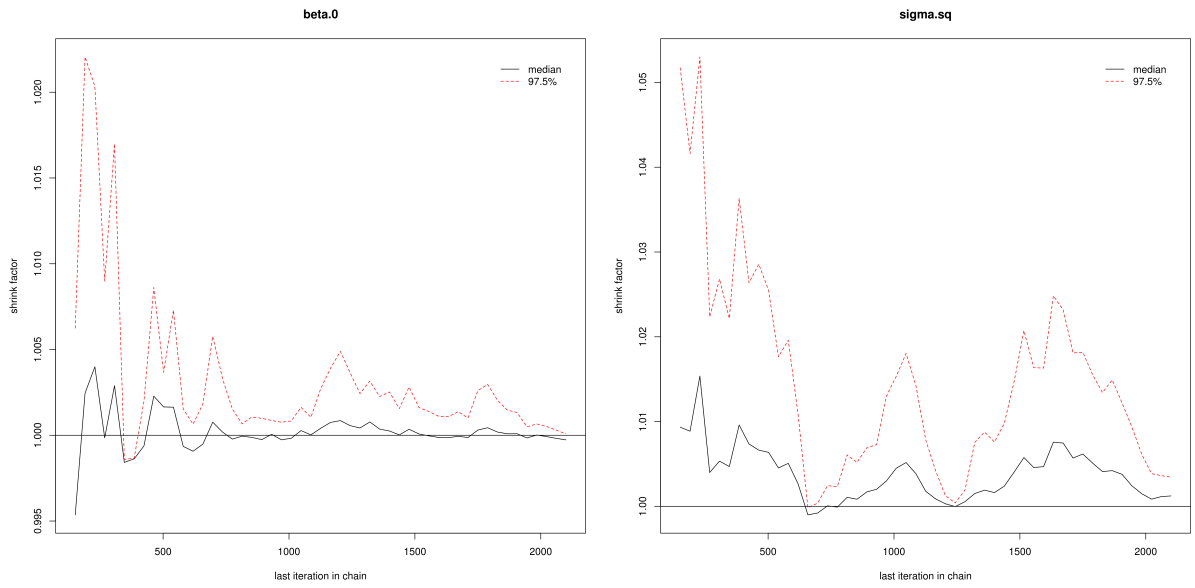

Figure 2: Gelman-Rubin diagnostic plots for β_0 and σ^2.

as the Gelman-Rubin diagnostic (Gelman and Rubin 1992). Basically, Gelman-Rubin measures whether there is a significant difference between the variance within several chains and the variance between several chains by the *potential scale reduction factors*. Convergence is diagnosed when the chains have "forgotten" their initial values, and the output from all chains is indistinguishable. This diagnostic is available in coda and illustrated in the code below.

```
> gelman.diag(samps)

Potential scale reduction factors:

         Point est. Upper C.I.
beta.0            1          1
sigma.sq          1          1

Multivariate psrf

1

> gelman.plot(samps)
```

The `gelman.diag` gives you the scale reduction factors for each parameter. A factor of 1 means that between chain variance and within chain variance are equal, larger values mean that there is still a notable difference between chains. A rule of thumb is that values of 1.1 and less suggests adequate convergence. The `gelman.plot`, Figure 2, shows you the development of the scale-reduction over the chain iterations, which is useful to see whether a low chain reduction is also stable (sometimes, the factors go down and then up again). This plot is also useful for determining a burn-in, which is the iteration after which the lines stay below $\sim$1.1.

4 Prediction

Say you have some sample units for which you want to predict y. In more realistic settings we typically want predictions for units where we have observed $\boldsymbol{x}$ but not y; however, since we are dealing with an intercept only model in this illustration then the intercept β_0 is the only information to inform prediction. Regardless, making prediction in BUGS or JAGS is the same in either setting—simply set those units in the response vector to `NA`. The resulting samples at the `NA` locations in the response vector are samples from the posterior predictive distribution and can be summarized just like model parameters. This is illustrated in the code below where 25 observations are withheld for subsequent prediction.

```
> n.0 <- 25
> y[1:n.0] <- NA
> data <- list(y = y, n = n)
> inits <- list(beta.0 = 0, sigma.sq = 1)
> jags.m <- jags.model(file = "ex-1.jag", data = data,
+     inits = inits, n.chains = 3, n.adapt = 100)

Compiling model graph
   Resolving undeclared variables
   Allocating nodes
   Graph Size: 108

Initializing model

> params <- c("beta.0", "sigma.sq", "y")
> samps <- coda.samples(jags.m, params, n.iter = 2000)
> round(summary(window(samps[, paste("y[", 1:25, "]",
+     sep = "")], start = burn.in))$quantiles[, c(3,
+     1, 5)], 2)

       50%  2.5% 97.5%
y[1]  0.16 -3.86  4.24
y[2]  0.15 -3.89  4.22
y[3]  0.22 -3.79  4.05
y[4]  0.13 -3.94  4.27
y[5]  0.19 -3.98  4.17
y[6]  0.21 -3.86  4.33
y[7]  0.22 -3.64  4.03
y[8]  0.20 -3.64  4.11
y[9]  0.21 -3.70  4.13
y[10] 0.19 -3.83  4.10
y[11] 0.23 -3.74  4.19
y[12] 0.13 -3.72  4.24
y[13] 0.17 -3.92  4.31
y[14] 0.16 -3.74  4.13
y[15] 0.15 -3.79  4.13
y[16] 0.21 -3.83  4.31
y[17] 0.19 -3.86  4.04
y[18] 0.16 -3.88  4.11
y[19] 0.24 -3.89  4.30
y[20] 0.18 -3.92  4.30
y[21] 0.26 -3.82  4.12
y[22] 0.15 -3.95  4.28
```

```
y[23] 0.15 -3.83  4.44
y[24] 0.25 -3.76  4.23
y[25] 0.21 -3.76  4.17
```

5 References

Gelman, A. and Rubin, D.B. (1992) Inference from iterative simulation using multiple sequences. *Statistical Science*, 7:457–511.

Gelman, A. (2006) Prior distributions for variance parameters in hierarchical models. *Bayesian Analysis*, 1:1–19.