Fitting Bayesian linear regression models in JAGS

We will use the union density data to demonstrate how a Bayesian linear model can be fitted in JAGS.

Union density is defined as the percentage of the work force who belongs to a trade union. There are a number of competing theories on what explains cross-national variation in union density.

To explore the different theories we use data from $n = 20$ countries with a continuous history of democracy since World War II. We have the following variables: Union density (Uden), (log) labour force size (LabF), industrial concentration (IndC), left wing government (LeftG), measured in late 1970s.

We are looking to fit the following linear regression model:

$$\begin{aligned} \text{Uden}_i &\sim N(\mu_i, \sigma^2) \\ \mu_i &= b_0 + b_1(\text{LeftG}_i - \overline{\text{LeftG}}) + b_2(\text{LabF}_i - \overline{\text{LabF}}) + b_3(\text{IndC}_i - \overline{\text{IndC}}) \end{aligned}$$

with vague priors

$$\begin{aligned} 1/\sigma^2 &\sim Gamma(0.001, 0.001) \\ b_0 &\sim N(0, 100000) \\ b_1 &\sim N(0, 100000) \\ b_2 &\sim N(0, 100000) \\ b_3 &\sim N(0, 100000) \end{aligned}$$

The aim is to get parameter estimates for b_0 and $\boldsymbol{b} = (b_1, b_2, b_3)$.

In order to get coefficient estimates we have to

1. Load the necessary `R` packages and load the data into `R`.
2. Define the above Bayesian linear regression model in JAGS.
3. Put the data into a list, initialise the stochastic nodes, and decide which parameters we want to monitor.
4. Fit the model with at least two chains and check convergence (note that two chains need two sets of initial values). If there's no convergence we can try updating the fitted model by adding more iterations.
5. Finally we can extract the posterior estimates from the fitted model, look at traceplots, density plots.

1. Load necessary packages and the data.

First we load the packages necessary to fit a Bayesian linear regression model in JAGS.

```r
library(R2jags)
library(coda)
library(lattice)
```

And we also manipulated the data so that we have a separate data vector for `Uden`, `LabF`, `LeftG` and `IndC`. Usually the data is stored in a separate csv file, but here, since the data set is quite small, we decided to just list the observations.

```r
# data
dat <- c(82.4,8.28,111.84,1.55,80.0,6.90,73.17,1.71,
74.2,4.39,17.25,2.06,73.3,7.62,59.33,1.56,
71.9,8.12,43.25,1.52,69.8,7.71,90.24,1.52,
68.1,6.79,0.00,1.75,65.6,7.81,48.67,1.53,
59.4,6.96,60.00,1.64,58.9,7.41,83.08,1.58,
51.4,8.60,33.74,1.37,50.6,9.67,0.00,0.86,
48.0,10.16,43.67,1.13,39.6,10.04,35.33,0.92,
37.7,8.41,31.50,1.25,35.4,7.81,11.87,1.68,
31.2,9.26,0.00,1.35,31.0,10.59,1.92,1.11,
28.2,9.84,8.67,0.95,24.5,11.44,0.00,1.00)

dat <-matrix(dat,nrow=20,ncol=4,byrow=TRUE)

Uden <- dat[,1]
LabF <- dat[,2]
LeftG <- dat[,3]
IndC <- dat[,4]
```

2. Define the model

Next we define the model using the BUGS language. Since we have observations from 20 countries, we will use a for loop to code the model up.

Note that for logical nodes (that is when the variable can be defined using `<-`) we can take advantage of the vectorised way `R` does computations (that is a for loop might not be necessary), but in case of stochastic dependence (when we are using `~`) we always have to use a for loop.

To define the model, we essentially translate the above model definition to the BUGS language line by line.

Note also that JAGS uses the precision instead of the variance to parametrise the normal distribution.

```r
jags.mod <- function(){
  for(i in 1:20){
    Uden[i]~dnorm(mu[i],tau) # normal likelihood

    # linear predictor with centered covariates
    mu[i] <- b0+b[1]*(LeftG[i]-mean(LeftG[]))+
      b[2]*(LabF[i]-mean(LabF[]))+b[3]*(IndC[i]-mean(IndC[]))
  }

  # prior on residual error variance
  tau ~ dgamma(0.001,0.001)
  sigma2 <- 1/tau # residual error variance

  # vagues priors on regression coefficients

  b0 ~ dnorm(0,0.00001)
  for(k in 1:3){
    b[k] ~ dnorm(0,0.00001)
  }
}
```

3. Data, Initial values, Parameters to monitor

JAGS needs the data to be passed on as a list, thus we first create a list from the previously defined data vectors, and call this list `jags.data`.

```r
jags.data <- list("Uden","LabF","LeftG","IndC")
```

Next we specify initial values. To check convergence we are planning to run 2 chains, thus we need two sets of initial values for the random parameters (ideally we should initialise all the stochastic nodes). These again should be passed on as a list. Note that Gelman-Rubin convergence diagnostic is not possible with a single chain.

```r
inits1 <-  list("tau" = 100, "b0" = 20, "b"=c(10,-5,-25))
inits2 <- list("tau"=100000,"b0"=-100,"b"=c(-100,100,500))
jags.inits <- list(inits1, inits2)



# Another way of creating the list:
# tau = 100
# b0 = 20
# b = c(10,-5,-25)
# inits <-  list("tau", "b0", "b")



# Yet another way of creating intial values
#
# jags.inits <- function(){
#   list("tau" = rnorm(1), "b0" = rnorm(1), "b" = rnorm(3))
# }
#
# where we can change the used distributions as appropriate.

# The advantage of this method is that we can easily change the number of chains
#  without having to worry about defining more or less initial values -
#  since jags will just call the above initial value function
#  when in need of a new set of initial values.

# The disadvantage is that we have less control over the chosen initial values,
#  and that this version can be harder to set up when the parameters,
#  (including missing values) have a slighly more complex structure
```

Finally set the parameters we want to monitor. At this stage these are the regression coefficients. Here we only need a vector of the parameter names. These don't need to be in a list.

```r
jags.param <- c("b0","b")
```

4. Fit the model and check for convergence

When fitting hte model we set the number of chains to two (`n.chains = 2`), draw 6000 samples for each chain (`n.iter = 6000`), but discard the first 1000 samples as burn-in (`n.burnin = 1000`). At this stage we don't want to thin the samples, so we set `n.thin` to 1 (later on, once we know more about autocorrelation, we can check acf plots and if these reveal correlation between samples, we can run the model again, only this time with thinning - but we don't have to worry about this for now).

Apart from the previously mentioned arguments, we also have to pass on the data, the list of initial values, the model defintition and also the vector of parameters we wish to monitor.

```
# Fit the JAGS model
jags.mod.fit <- jags(data = jags.data, inits = jags.inits,
                     parameters.to.save = jags.param, n.chains = 2, n.iter = 6000,
                     n.burnin = 1000,n.thin=1,model.file = jags.mod)
```

Now that the model is fitted we should check for convergence. For we convert the fitted model object into an MCMC object (this can be done using the `as.mcmc` function) and use the `gelman.diag()` function to extract the Gelman-Rubin test statistic values.

```
jagsfit.mcmc <- as.mcmc(jags.mod.fit)
gelman.diag(jagsfit.mcmc)
```

We can even store these in a dataframe, in case we have a large number of parameters.

```
rhat <- as.data.frame(gelman.diag(jagsfit.mcmc)$psrf)
rhat
```

If the chain has converged these values should be close to 1 (anything above 1.1 usually indicates lack of convergence). If the `gelman.diag` command doesn't run try setting the `multivariate` argument to `FALSE`. *Here we have perfect convergence.*

If there's no convergence we can draw new samples using the `update` function:

```
jags.mod.fit.upd <- update(jags.mod.fit, n.iter=10000)

# We can also try the following function, which will keep updating the model
# until it converges (assuming that convergence with the given conditions is possible)
# jags.mod.fit.upd <- autojags(jags.mod.fit)
```

5. Summary statistics, traceplots, density plots

Note that we could also look at traceplots in the previous step to assess the mixing of the chains, which can help to assess convergence.

First however we look at some posterior summary statistics for the variables we monitored (point, interval estimates). Since we specified the burn-in at the start these samples are not included in the summary statistics.

```
print(jags.mod.fit.upd)
```

To get the traceplots corresponding to the monitored nodes we can either use the `traceplot` function, or convert the fitted model into an MCMC object, which allows us to look at more diagnostics.

```r
traceplot(jags.mod.fit.upd)


jagsfit.mcmc <- as.mcmc(jags.mod.fit.upd)
# symmary statistics
summary(jagsfit.mcmc)

# plot both traceplots and posterior density functions
# (for all the parameters)
plot(jagsfit.mcmc)
# Or separately
#   traceplots
xyplot(jagsfit.mcmc)
#   density plots
densityplot(jagsfit.mcmc)
```

If there's a specific parameter we are interested in, we can use the `MCMCtrace` function of the `MCMCvis` package to look at the traceplot and densityplot for this specific parameter separately. For example, we might be interested in the coefficient of the labour force size (b_2). For this we would use

```r
library(MCMCvis)

# traceplot
MCMCtrace(jagsfit.mcmc,
          params=c('b\\[2\\]'),
          type = 'trace',
          ind = TRUE,
          ISB = FALSE,
          pdf = FALSE)

# posterior density
MCMCtrace(jagsfit.mcmc,
          params=c('b\\[2\\]'),
          type = 'density',
          ind = TRUE,
          ISB = FALSE,
          pdf = FALSE)
```