Computation V

Statistics 220

Spring 2005

WinBUGS

While `WinBUGS`is a stand alone program, the easiest way to run it is from `R`, using Andy Gelman's `bugs.R`setup.

If you want to run it from your own Windows machine, `WinBUGS` is available from

http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml

The current version is 1.4.1.

There is a newer version, known as `OpenBUGS`, which has just been updated to version 2.1.0

http://mathstat.helsinki.fi/openbugs/

This version will run under Windows and Intel based linux. Up until recently `WinBUGS` appeared to be more stable than `OpenBUGS`, though that may have changed with the 2.1.0 update.

While `WinBUGS`is a stand alone program, the easiest way to run on a Windows machine is from `R`, using Andy Gelman's `bugs.R`setup. It is available at

http://www.stat.columbia.edu/∼gelman/bugsR/

It is compatible with both `WinBUGS` and `OpenBUGS`, at least up to version 2.0.1 (beta).

Note that there another `R` front end to `OpenBUGS`, `BRugs`, which is compatible with `OpenBUGS`, but not `WinBUGS` (I think).

WinBUGS **Modelling Considerations**

- Proper priors
- Distributions available
  - Discrete univariate:
    Bernoulli, Binomial, Categorical, Negative Binomial, Poisson
  - Continuous univariate:
    Beta, Chi-square, Double Exponential (Laplace), Exponential, Gamma, Log-normal, Logistic, Normal, Pareto, t, Uniform, Weibull
  - Discrete multivariate:
    Multinomial
  - Continuous multivariate:
    Dirichlet, Multivariate Normal, Multivariate t, Wishart
- Note that more complicated situations are possible using programming tricks in `WinBUGS`.

Components of a `WinBUGS` Run

- Model
- Data
- Initial values
- Values to be returned

  Parameters of the model, new values (e.g. y^{rep})

`WinBUGS` **Model**

Components of a model are known as nodes in the `WinBUGS` documentation. These are either stochastic (random variables) or logical (deterministic functions of other nodes).

The `WinBUGS` language has a `S-Plus`/`R` feel to it. However it isn't a perfect match

Model:

$$\begin{aligned} y_{ij} &\overset{ind}{\sim} N(\theta_i, \sigma^2) \\ \theta_i &\overset{iid}{\sim} N(\mu, \tau^2) \\ \mu &\sim N(0, 10^6) \\ \tau^2 &\sim U(0, 1000) \\ \sigma^2 &\sim \text{Inv}-\text{Gamma}(0.001, 0.001) \end{aligned}$$

WinBUGS Model:

```
model {
  for(i in 1:nbeer) {
    for(j in 1:nobs) {
      sodium[i,j] ~ dnorm(theta[i],sigma2inv)
    }
    theta[i] ~ dnorm(mu,tau2inv)
  }
  mu ~ dnorm(0,1.0e-6)              # mean of random effect
  tau2 ~ dunif(0,1000)              # variance of random effect
  tau2inv <- pow(tau,-2)            # precision of random effect
  tau <- sqrt(tau2)                 # sd of random effect
  sigma2inv ~ dgamma(0.001, 0.001) # measurement error precision
  sigma <- 1 / sqrt(sigma2inv)     # measurement error sd
}
```

When defining a model, a stochastic node is defined with $\sim$ and a logical node with $<-$ (like `R`).

The name for all density functions in `WinBUGS` start with d. Often the names are the same as the `R` equivalent, but not always (e.g. `dbinom(n,p)` in `R` and `dbin(p,n)` in `WinBUGS`). Also the order of arguments may be different

Also for location-scale families, a precision parameter is used instead of a scale parameter. For example, for the normal, the density is parameterized as

$$f(x|\mu,\tau) = \sqrt{\frac{\tau}{2\pi}} \exp\left(\frac{\tau}{2}(x-\mu)^2\right)$$

instead of the more usual

$$f(x|\mu,\sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

Another example is the logistic distribution, which is parameterized as

$$f(x|\mu,\tau) = \frac{\tau \exp(\tau(x-\mu))}{(1+\exp(\tau(x-\mu)))^2}$$

instead of

$$f(x|\mu,\sigma) = \frac{\exp((x-\mu)/\sigma)}{\sigma(1+\exp((x-\mu)/\sigma))^2}$$

Nodes can only occur on the left side of a definition once. So don't do

```
y[i] ~ dgamma(alpha, beta)
y[i] ~ dnorm(mu, sigma2inv)
```

in the same model file.

Also, functions can only be used on the right side of logical nodes, except for link functions in generalized linear models. For example a probit regresson model can be defined either by

```
model {
  for(i in 1:nobs) {
    y[i] ~ dbin(p[i], n[i])
    probit(p[i]) <- alpha + beta*x[i]
  }
  alpha ~ dnorm(0, 1.0E-6)
  beta ~ dnorm(0, 1.0$-6)
}
```

or

```
model {
  for(i in 1:nobs) {
    y[i] ~ dbin(p[i], n[i])
    p[i] <- phi(alpha + beta*x[i])
  }
  alpha ~ dnorm(0, 1.0E-6)
  beta ~ dnorm(0, 1.0$-6)
}
```

Also stochastic nodes to not need to be tied directly to data. For example, simulating y^{rep}s in the above model can be done with

```
model {
  for(i in 1:nobs) {
    y[i] ~ dbin(p[i], n[i])
    yrep[i] ~ dbin(p[i],n[i])
    p[i] <- phi(alpha + beta*x[i])
  }
  alpha ~ dnorm(0, 1.0E-6)
  beta ~ dnorm(0, 1.0$-6)
}
```

Data

- Popular format is `S-Plus` list format (displayed as text). She the examples
- For arrays, data can also be input in rectangular arrays
- Variable names must match the names in the model file

Starting Values

- Can be given for stochastic nodes
- Must **not** be given for logical nodes. You will be a cryptic error message (Incompatible copy)
- You do not need to give for all nodes. `WinBUGS` has an algorithm for choosing starting values for nodes without user specified values.
- The `WinBUGS` algorithm for starting values they refer to as forward sampling.

  1. Start with any hyperparameters that aren't given starting values and sample from there prior.
  2. Go to the next level in the hierarchy. Sample starting values for any nodes at this level from the model, given values for nodes lower in the hierarchy

Example:

Suppose that the model

```
model {
  for(i in 1:nobs) {
    y[i] ~ dbin(p[i], n[i])
    yrep[i] ~ dbin(p[i],n[i])
    p[i] <- phi(alpha + beta*x[i])
  }
  alpha ~ dnorm(0, 1.0E-6)
  beta ~ dnorm(0, 1.0E-6)
}
```

is to be run. Suppose that starting values are given for `alpha` and `y[i]`, but not `beta` and `yrep[i]`.

The forward sampling algorithm would

1. Sample `beta` from `beta` $\sim$ `dnorm(0, 1.0E-6)`
2. Then for each i, sample `yrep[i]` from `yrep[i]` $\sim$ `dbin(p[i],n[i])` based on the `alpha` and `beta` available.

- Usually better to define them yourself.

- You particularly want to do this if the prior is vague, as α and β are in this case.

- If you are running multiple chains, use multiple starting values, particularly ones you are worried about converging properly. This will allow for better convergence checks with $\hat{R}$.

Output - Values Returned

When running a chain, you do not need to store all the values from the chain.

- Burnin - usually not stored
- Thinning - only every k values are stored after the burnin period
- Some variables may not be needed for the analysis, and do not need to be stored (e.g. low level hyperparameters)

Running `WinBUGS`

- Can run `WinBUGS` directly
- The necessary inputs can be entered in multiple files or in a single "Compound" document
- Can run from `R` with `bugs.R` front end

`bugs.R`

To use `bugs.R`, there are four main steps.

1. Defining your data

2. Setting the starting values

3. Stating what is to be returned

4. Running `WinBUGS`

This is best exhibited with an example

```
beers <- read.table("beer_data.txt", header=T)
sodium <- matrix(beers$Sodium,ncol=8,byrow=T)
sodium.mean <- apply(sodium,1,mean)
sodium.var <- apply(sodium,1,var)
nbeer <- dim(sodium)[1]
nobs <- dim(sodium)[2]

data <- list ("nbeer", "nobs", "sodium")

inits <- function() {
     list(theta=rnorm(6, sodium.mean, sqrt(sodium.var/nobs)),
     mu=rnorm(1,mean(sodium.mean), sqrt(var(sodium.mean))),
     tau2=var(sodium.mean),
     sigma2inv=mean(sodium.var))}

parameters <- c("theta", "mu", "tau", "sigma")

beer.sim <- bugs(data, inits, parameters, "beer.bug",
                  n.chains=5, n.iter=500, n.thin=5)
```

1. Defining your data:

   This must be a list of character strings with the names of the variables

   ```
   data <- list ("nbeer", "nobs", "sodium")
   ```

2. Setting the starting values

   This must be a function, which returns a list of the starting values

   ```
   inits <- function() {
        list(theta=rnorm(6, sodium.mean, sqrt(sodium.var/nobs)),
        mu=rnorm(1,mean(sodium.mean), sqrt(var(sodium.mean))),
        tau2=var(sodium.mean),
        sigma2inv=mean(sodium.var))}
   ```

   The values in the list must be stochastic nodes of the model.

3. Stating what is to be returned

   A vector of strings containing the variables to be returned

```
parameters <- c("theta", "mu", "tau", "sigma")
```

4. Running `WinBUGS`

   Need to specify what the data, initial value, parameters to be returned, and the model file are. In addition you can specify the number of chains (≥ 2), number of iterations, thinning rate, and the burnin are (these all have defaults)

```
beer.sim <- bugs(data, inits, parameters, "beer.bug",
                 n.chains=5, n.iter=500, n.burnin = 250,
                 n.thin=5)
```

   The model file must have the extension `.txt` or `.bug` to run correctly.

When `WinBUGS` finishes it will return all the information from the simulation to the list `beer.sim`. In addition it attaches two lists to the search path.

In position 2 of the search path, it attaches the list `bugs.sim` which contains all the desired variables output by `WinBUGS`. Also in this list is the deviance for each simulated data set. This allows for the calculation of DIC and p_D for model comparison. If you store the output of `bugs`, this information will be there in a couple of different formats.

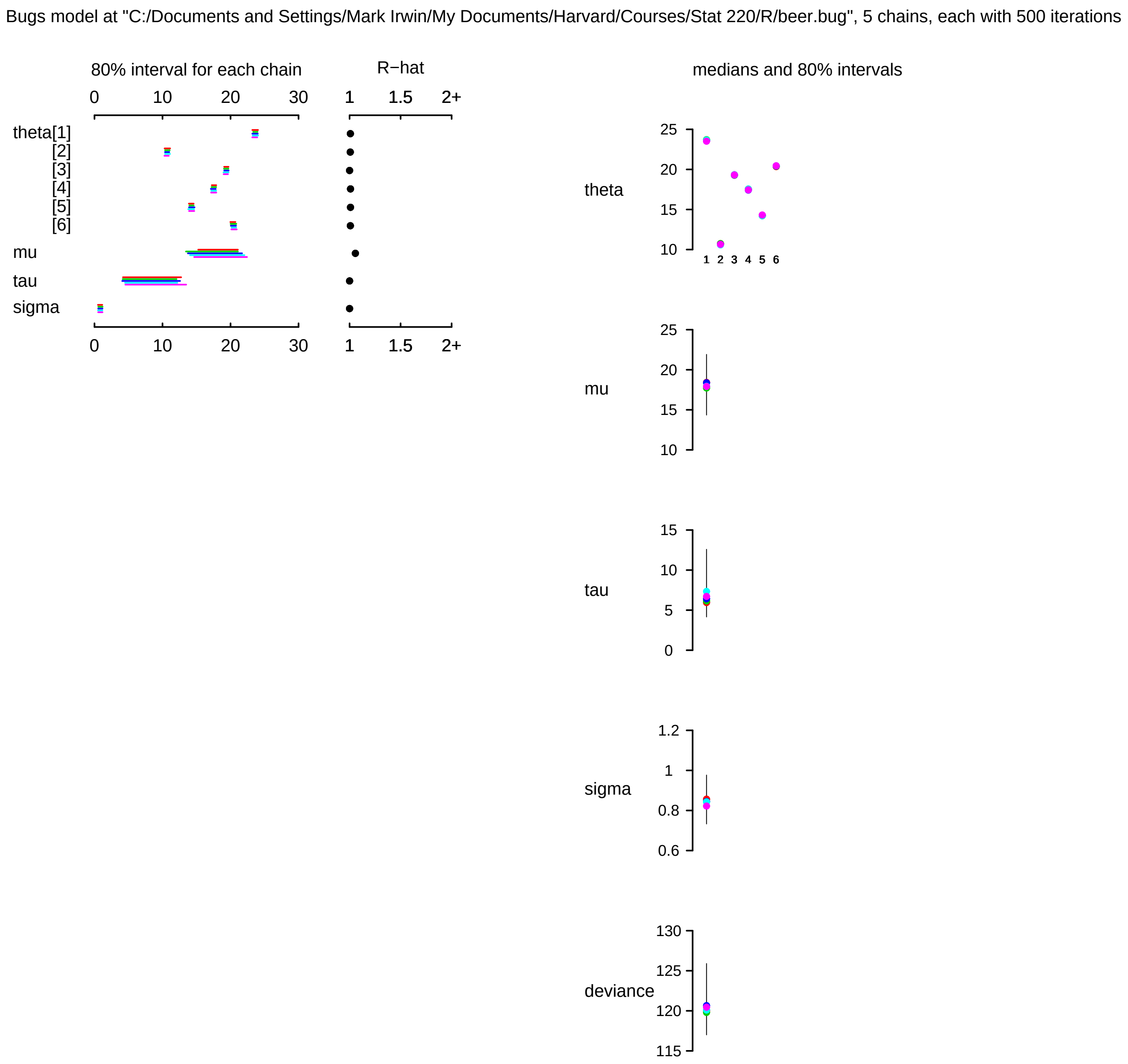
In position 3, it attaches the list `bugs.all`, which is a copy of the output of bugs.

Note that these two lists will die when you quit `R`, so you will need to store the information from the run if you want to use it later.

In addition, you get two other pieces of output. The first is a summary table of the run. In addition to be printed to the screen, it is also stored in the output in list component `summary`. Access, for example, by `beer.sim$summary`. The second is a graphical summary, based on information, at least partly, in the information contained in the printed summary.

```
Inference for Bugs model at "C:/Documents and Settings/Mark
Irwin/My Documents/Harvard/Courses/Stat 220/R/beer.bug"
 5 chains, each with 500 iterations (first 250 discarded), n.thin = 5
 n.sims = 250 iterations saved
Time difference of 6 secs
          mean  sd  2.5%   25%   50%   75% 97.5% Rhat n.eff
theta[1]  23.6 0.3  23.1  23.4  23.6  23.8  24.2  1.0   200
theta[2]  10.7 0.3  10.1  10.5  10.7  10.9  11.2  1.0   250
theta[3]  19.3 0.3  18.9  19.2  19.3  19.5  19.9  1.0   250
theta[4]  17.5 0.3  16.9  17.3  17.5  17.7  18.1  1.0   170
theta[5]  14.3 0.3  13.7  14.1  14.3  14.4  14.8  1.0   250
theta[6]  20.4 0.3  19.8  20.2  20.4  20.6  21.0  1.0   240
mu        18.1 3.6  11.4  16.4  18.1  19.9  25.6  1.1   250
tau        7.6 3.8   3.4   5.1   6.6   9.3  16.9  1.0   250
sigma      0.8 0.1   0.7   0.8   0.8   0.9   1.1  1.0   250
deviance 121.0 3.8 115.6 118.1 120.2 122.9 131.1  1.0   250
 pD = 7.4 and DIC = 128.4 (using the rule, pD = var(deviance)/2)

 For each parameter, n.eff is a crude measure of effective sample size,
 and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
 DIC is an estimate of expected predictive error (lower deviance is better).
```


Bugs model at "C:/Documents and Settings/Mark Irwin/My Documents/Harvard/Courses/Stat 220/R/beer.bug", 5 chains, each with 500 iterations
80% interval for each chain
R−hat
medians and 80% intervals
theta[1]
[2]
[3]
[4]
[5]
[6]
mu
tau
sigma
theta
mu
tau
sigma
deviance
0 10 20 30
1 1.5 2+