BUGS: Language, engines, and interfaces

Patrick Breheny

January 17

The BUGS framework

- The BUGS project (Bayesian inference using Gibbs Sampling) began in 1989
- Its basic philosophy was to separate the language used to describe a model from the actual programs used to carry out the computations (the *engine*)
- This approach has two attractive advantages:
  - One can easily specify complex models without requiring extensive knowledge of Bayesian computational tools (Gibbs sampling, Markov Chains, etc.)
  - The language has remained stable and consistent over time, even as the underlying programs which actually do the sampling are constantly changing

BUGS syntax

The syntax of the BUGS language is relatively straightforward: every line does one of the following things:

- Defines a stochastic node: `sigma ~ dunif(0, 100)`
- Defines a logical (deterministic) node: `tau <- pow(sigma, -2)`
- Provides a comment: `## Prior`
- Defines a loop: `for (i in 1:nx)`

For more details on the distributions and functions available in BUGS, check the appendix of our text (there is an online manual as well)

Engines

- Historically, the most widely used BUGS engine has been a program called WinBUGS
- Further developments by the developers of WinBUGS, however, have focused on a somewhat different program called OpenBUGS; the features and appearance of the programs are similar (at least for now), but OpenBUGS is more compatible with different operating systems
- A separate project, JAGS (Just Another Gibbs Sampler), can be thought of as an engine for running BUGS models, although strictly speaking, the language syntax for the two programs is not always identical
- Engine development is ongoing; a rather promising one, STAN, was just released in December 2012

What the engines do

All engines carry out the following steps for fitting BUGS models:

1. Checking model syntax
2. Reading in data
3. Compiling the model
4. Initializing the simulation
5. Sampling
6. Report results

Interfaces

- One can run BUGS/JAGS directly – you don't need `R` installed on your machine to do so – but it's both easier and more useful to use `R` as an interface (at least, in my opinion)
- Not only are there a variety of programs capable of interfacing with BUGS (SAS, Stata, Matlab, etc.), but there are a variety of `R` packages that accomplish this task
- Nevertheless, we'll focus on `R2jags`, for several reasons: (a) it works well, (b) it's easy to use, (c) has a consistent interface for both BUGS and JAGS (`R2OpenBUGS`), and (d) Gelman and Hill use it in their book

R2jags

- R2jags and R2OpenBUGS are similar in terms of model specification, passing data, structure of output, etc.:

```
Data <- list(y=31, n=39)
model <- function() {
  theta ~ dunif(0,1)
  y ~ dbin(theta, n)
}
fit <- jags(Data, model=model, ...) ## R2jags
fit <- bugs(Data, model=model, ...) ## R2OpenBUGS
```

- Summaries can then be obtained via

```
print(fit)
plot(fit)
```

coda

- `R2jags` interfaces nicely with another `R` package called `coda`, which has a large number of useful tools for analyzing the results of MCMC sampling
- We'll get to most of these tools later, but for now, let's note that

```
mcfit <- as.mcmc(fit) ## Converts to coda format
summary(mcfit)
```

Mean	SD	MCSE_n	MCSE_{ts}
0.7809	0.0639	0.0006	0.0008

OpenBUGS

- For the most part, BUGS/R2OpenBUGS and JAGS/R2jags commands are interchangeable, although occasionally we will encounter minor differences
- The big difference is that JAGS was designed to be run over the command line, while OpenBUGS was designed to be run through a graphical user interface
- Generally speaking, this makes running JAGS through `R` more convenient than running BUGS; however, there are some interesting interactive features you can get through the BUGS GUI that you can't access over the command line

Initial setup

- The first thing you need to do in order to run OpenBUGS through its GUI is to write out the model and data
- You can do this manually, although it's sometimes more convenient to have `R2OpenBUGS` do this by running a small simulation (say, with `n.iter=5`
- You can look at what `R2OpenBUGS` writes to get a sense of this syntax, or consult section 12.4 of our text for the full details

Model specification

Then we need to specify the model; from within OpenBUGS (note that messages are printed at the bottom left describing whether or not these operations are working):

1. Open the model specification tool (Model $\rightarrow$ Specification)
2. Open the model file and click "Check Model"
3. Open the data file and click "Load data"
4. (Optional) set the number of chains
5. Click "compile"
6. Initialize the model, either by loading initial values from a file or by clicking "gen inits"

Sampling

Now we are ready to start sampling:

1. Open the update tool (Model $\rightarrow$ Update) and the sample monitor tool (Inference $\rightarrow$ samples)
2. In the monitor tool, specify the nodes you wish to monitor and click "set" after each one
3. In the update tool, specify the number of updates you want and click "update"
4. In the monitor tool, specify a nodes you wish to look at and click "stats" or "density" (in the future, we will also be interested in diagnostics)
5. From here, you can go back and forth between the sample and monitor tools, updating and inspecting as you see fit

Summary

To recap, the basic steps are:

1. Specify the model
   a. Load the model
   b. Load the data
   c. Load/generate initial values
2. Run the sampler
   a. Monitor nodes
   b. Update model
   c. Repeat as necessary

Binomial data and MC methods

- In this particular example (the premature birth data), we didn't really need to use Monte Carlo methods; we could solve the problem analytically
- Two important points are worth noting, however:
  - MC methods provide correct results (at least in this case), even without deriving the conjugate relationship between the beta and binomial distributions
  - If we alter the problem slightly, we lose conjugacy and an analytic solution becomes much harder (if not impossible) to come by, but we can still use a Monte Carlo approach

A hypothetical crossover study

- For example, suppose we conduct a crossover or matched-pair experiment involving 17 pairs who are given either a treatment or a placebo, and we find that treatment outperformed placebo in 13 out of the 17 pairs
- We are interested in θ, the probability that an individual will do better on treatment than placebo
- However, a beta distribution may not represent our prior beliefs in this case; we may think that the value $\theta = \frac{1}{2}$ is particularly likely, since θ will take on exactly this value if the treatment does nothing

A mixture prior

- Thus, let's consider this more complicated prior:

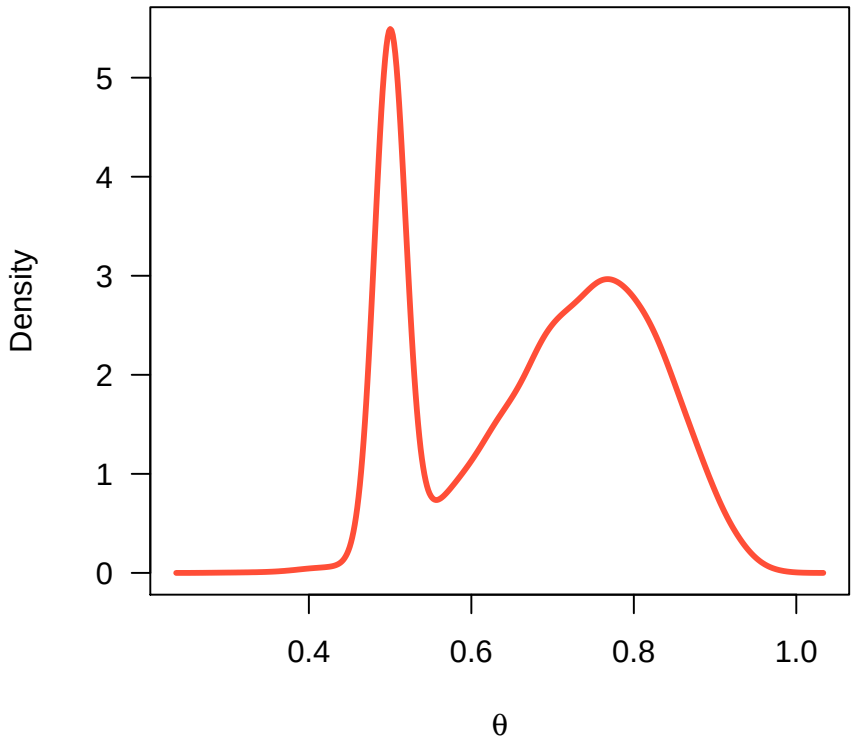
$$H \sim \text{Bern}\left(\frac{1}{2}\right)$$

$$\theta \sim \begin{cases} \text{Unif}(0,1) & \text{if } H = 1 \\ \frac{1}{2} & \text{if } H = 0 \end{cases}$$

- This mixture distribution is not directly available in the BUGS syntax, but is easily implemented through a latent variable approach:

```
theta1 ~ dunif(0,1)
theta <- H*theta1 + (1-H)*0.5
```

Results



$$\Pr(H = 0) = 25.1\%$$

Technically, of course, there is a spike of infinite density at $\theta = 0.5$, but this gets the main idea across

Remarks

- In addition to illustrating the power and flexibility of BUGS, this also illustrates an interesting comparison between Bayesian and frequentist statistics, as the frequentist p-value is 0.049
- The typical frequentist interpretation is that this experiment provides fairly strong ("significant") evidence against the null hypothesis, while the Bayesian approach judges the null hypothesis quite plausible (25% of being true)
- This is fairly typical: although Bayesian and frequentist methods often agree with respect to estimation and confidence intervals, what they imply about hypotheses, model comparisons, and decisions are often very different