10.2 Model Comparison in BUGS

Zongyi Liu

2023-06-22

```
# install.packages("BRugs")
```

10.2 Model Comparison in BUGS

The example in 10.1 showed a case in which two models were formed by two discrete values of a continuous hyperparameter. More generally different models can be thought of as dependent on a categorical hyperparameter, which merely indexes the models.

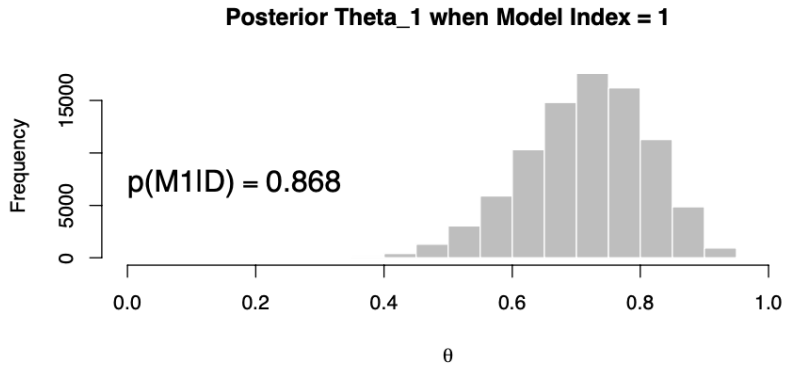
When the indexical hyperparameter has value j, then the j^{th} model is used to estimate the data.

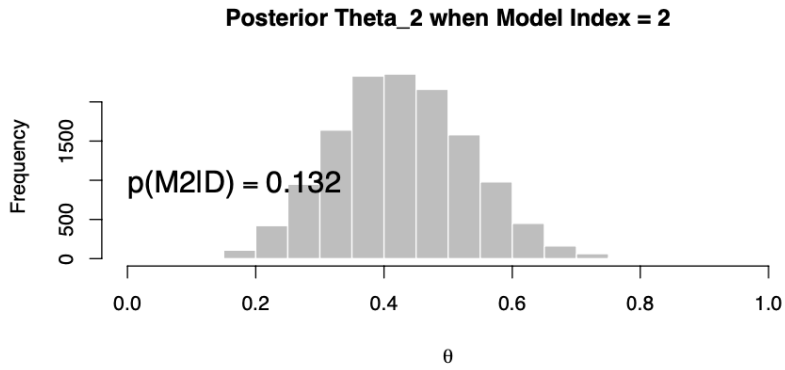
10.2.1 A Simple Example

We can continue with the example in 10.1 to illustrate the basic idea in BUGS.

```
model{
    # Likelihood:
    for ( i in 1:nflips ) {
        y[i] ~ dbern( theta )  # y[i] distributed as Bernoulli
    }
    # Prior distribution:
    theta ~ dbeta( aTheta , bTheta ) # theta distributed as beta density
    aTheta <- muTheta * kappaTheta
    bTheta <- (1-muTheta) * kappaTheta
    # Hyperprior:
    muTheta <- muThetaModel[ modelIndex ]
    muThetaModel[1] <- .75
    muThetaModel[2] <- .25
    kappaTheta <- 12
    # Hyperhyperprior:
    modelIndex ~ dcat( modelProb[] )
    modelProb[1] <- .5
    modelProb[2] <- .5
}
# ... end BUGS model specification
```

The output plot can be shown as below:





10.2.2 A Realistic Example with "Pseudopriors"

If we reconsider the case introduced in 9.3.1. There were four different category structures, a.k.a. conditions, for learning. In the initial BRugs code for that example, each condition was allowed its own mean (μ) and certainty (k) hyperparameter values for the beta distribution from which individual participant biases were generated.

Alternatively, we also considered a model that assumed the same certainty value for all four conditions. This model was diagrammed in the left side of Figure 9.17, p. 183. This assumption says that all participants are equally dependent on their condition's μ parameter, even though the μ parameter may differ between groups.

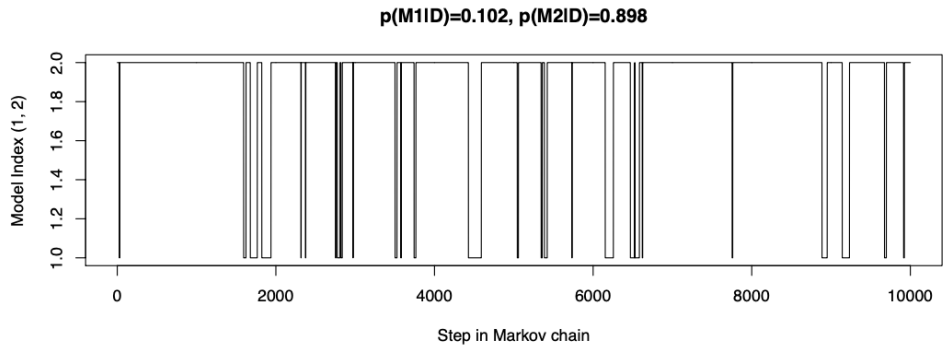
```
model {
   for ( i in 1:nSubj ) {
      # Likelihood:
      nCorrOfSubj[i] ~ dbin( theta[i] , nTrlOfSubj[i] )
      # Prior on theta: Notice nested indexing.
      theta[i] ~ dbeta( aBeta[ CondOfSubj[i] ] ,
                        bBeta[ CondOfSubj[i] ] )I(0.0001,0.9999)
   }
   # Hyperprior on mu and kappa:
   kappa0 ~ dgamma( shapeGamma , rateGamma )
   for ( j in 1:nCond ) {
      mu[j] ~ dbeta( aHyperbeta , bHyperbeta )
      kappa[j] ~ dgamma( shapeGamma , rateGamma )
   }
   for ( j in 1:nCond ) {
```

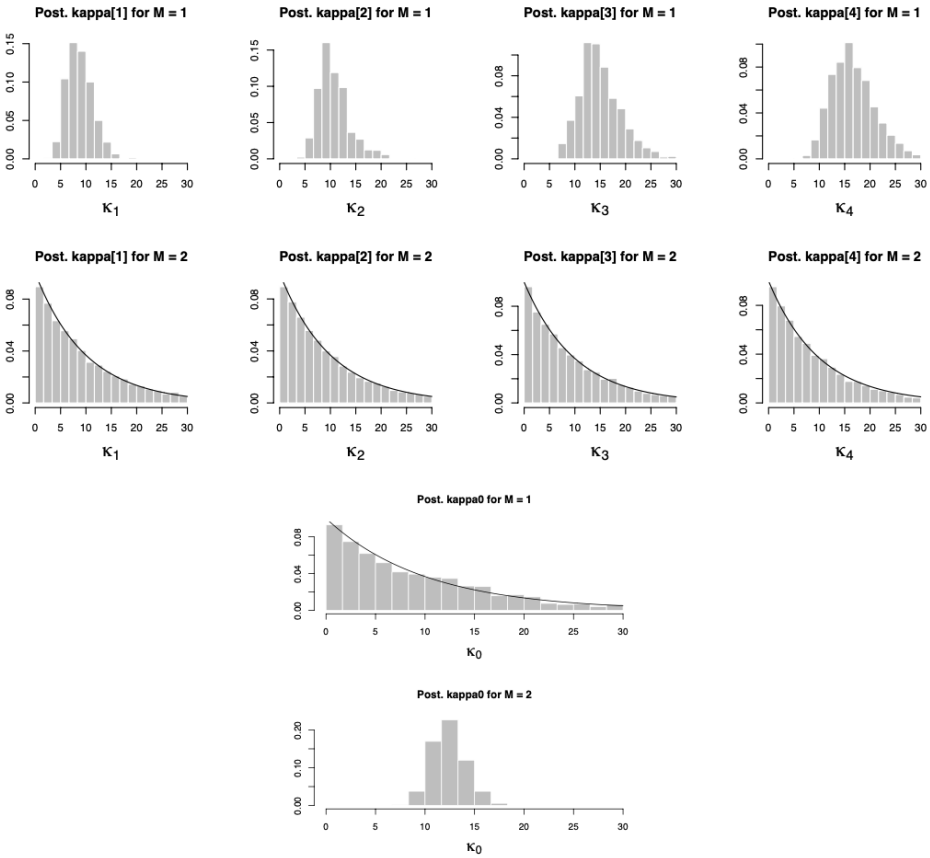
```
    aBeta[j] <- mu[j]     * ((kappa[j]*(2-mdlIdx))+(kappa0*(mdlIdx-1)))
    bBeta[j] <- (1-mu[j]) * ((kappa[j]*(2-mdlIdx))+(kappa0*(mdlIdx-1)))
    # BUGS equals(,) won't work here, for no apparent reason.
    # Took me hours to isolate this problem (argh!). So, DO NOT use:
    # aBeta[j] <- mu[j]     * (kappa[j]*equals(mdlIdx,1)+kappa0*equals(mdlIdx,2))
    # bBeta[j] <- (1-mu[j]) * (kappa[j]*equals(mdlIdx,1)+kappa0*equals(mdlIdx,2))
  }
  # Constants for hyperprior:
  aHyperbeta <- 1
  bHyperbeta <- 1
  shapeGamma <- 1.0
  rateGamma <- 0.1
  # Hyperprior on model index:
  mdlIdx ~ dcat( modelProb[] )
  modelProb[1] <- .5
  modelProb[2] <- .5
}
```

The model specification begins with the binomial likelihood for each individual's data. Then each individual `theta[i]` value is distributed as a beta distribution with parameters `aBeta` and `bBeta`. On lines 25–26 something new happens. The `aBeta` and `bBeta` values de- pend on the value of the model index, `mdlIdx`. For example, when `mdlIdx` is 1, then `aBeta[j] <- mu[j] * kappa[j]`, but when `mdlIdx` is 2, then `aBeta[j] <- mu[j] * kappa0`. The program accomplishes this conditional assignment by a contorted algebraic manipulation because the BUGS.

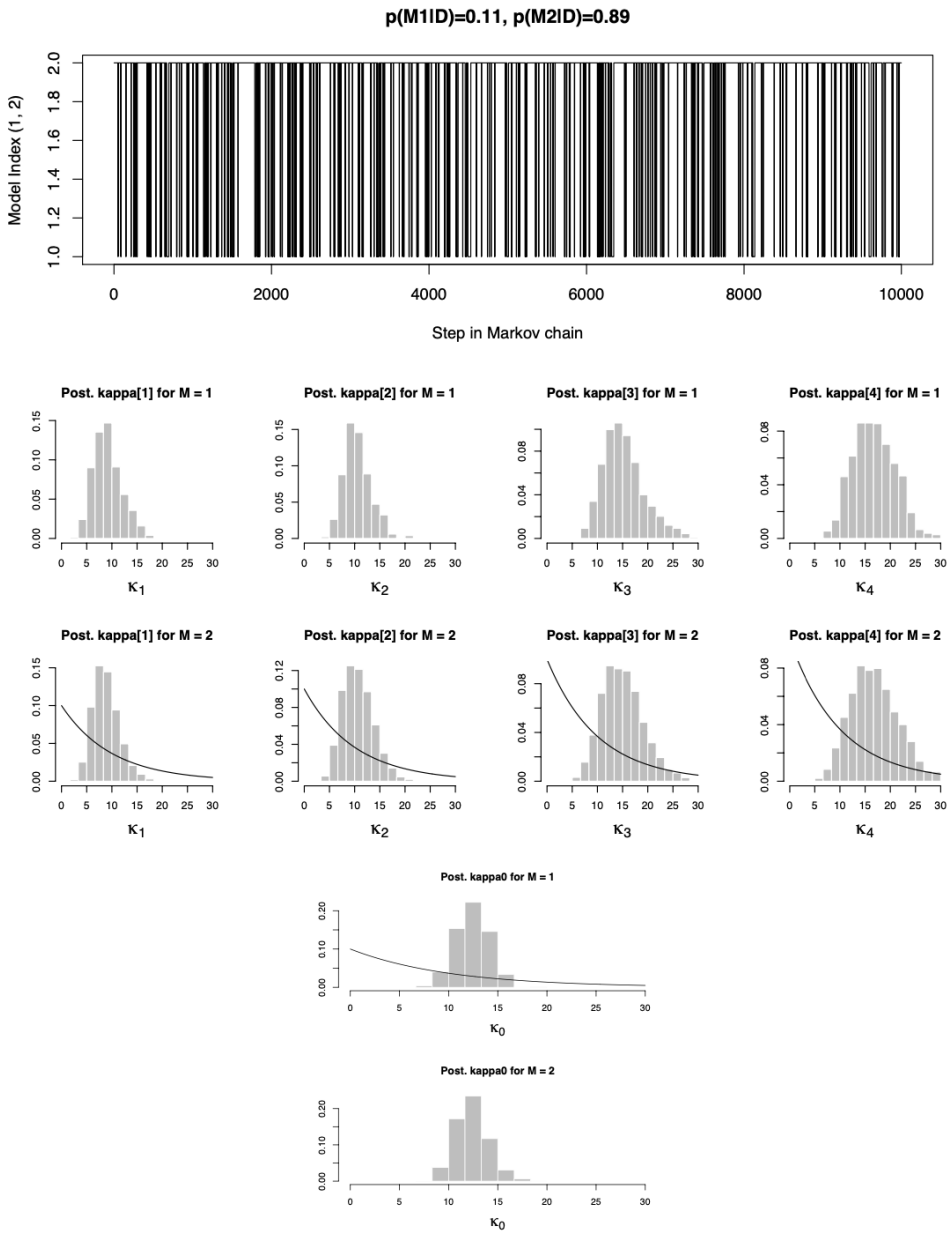
Figure 10.3 illustrates the output. The top panel shows the MCMC chain for the model index. The most obvious feature is that there is a much higher probability for model 2 than model 1.

Below is the plot without pseudopriors





Below this the plot with pseudopriors



10.2.3 Some Advice when Using Transdimensional MCMC with Pseudopriors

When doing the first run without pseudopriors, we should not use a separate model that uses only the real priors. All we need is the program that can accommodate the pseudopriors, which we run iteratively with different specifications for the pseudopriors. The first time you run it, the pseudopriors are specified as the real priors.

When one model is much less believable than another, the unbelievable model rarely gets visited in the chain, and so there are few points in the sample to represent its parameter values. To compensate for this imbalance, we can arbitrarily set the the model-index prior probabilities to compensating values.

WE can then re-run the sampler with the model priors set to `modelProb[1] <- .98` and `modelProb[2] <- .02`, so that model 1 gets visited a lot more frequently. This will give you a better sample of the posteriors for

creating reasonable pseudopriors. When we fiddle with the model priors, beware that the relative posterior probabilities of the models is the Bayes factor times the ratio of the prior probabilities.