Practical 5. Evidence synthesis and decision models — SOLUTIONS

Lecture 5 PDF version

Decision modelling

Once you have checked the model code in the file `EvSynth.txt`, you can concentrate on the `R` script. The first part is fairly simple.

```
# Influenza example --- source: Cooper et al (2004); Baio (2012)

# Sets up the working directory to the current one
working.dir <- getwd()
setwd(working.dir)

# Defines the data
# Number of interventions (t=0: control; t=1: prophylactic use of Neuramidase Inhibitors (NI)
T <- 2

# Evidence synthesis on effectiveness of NIs prophylaxis vs placebo
r0 <- r1 <- n0 <- n1 <- numeric()   # defines observed cases & sample sizes
r0 <- c(34,40,9,19,6,34)
r1 <- c(11,7,3,3,3,4)
n0 <- c(554,423,144,268,251,462)
n1 <- c(553,414,144,268,252,493)
S <- length(r0)             # number of relevant studies

# Evidence synthesis on incidence of influenza in healthy adults (under t=0)
x <- m <- numeric()         # defines observed values for baseline risk
x <- c(0,6,5,6,25,18,14,3,27)
m <- c(23,241,159,137,519,298,137,24,132)
H <- length(x)

# Data on costs
unit.cost.drug <- 2.4       # unit (daily) cost of NI
length.treat <- 6*7     # 6 weeks course of treatment
c.gp <- 19          # cost of GP visit to administer prophylactic NI
vat <- 1.175            # VAT
c.ni <- unit.cost.drug*length.treat*vat

# Informative prior on cost of influenza
mu.inf <- 16.78         # mean cost of influenza episode
sigma.inf <- 2.34       # sd cost of influenza episode
tau.inf <- 1/sigma.inf^2    # precision cost of influenza episode
```

The First couple of lines set up the working directory. In previous practicals, we have seen how you can do this by setting a path to the folder you want to use. In this case, we first define a variable `working.dir`, which we set equal to the current directory (accessed by the `R` command `getwd()`). The second line is actually not necessary, strictly speaking, because we already are in the current directory. But this shows, again, how we can use `getwd()` and `setwd()` to move across the folders in our computer. Notice that we are using in the script the assign "`->`" sign, while in previous practicals we have used the equal "`=`" sign. For all intents and purposes, `R` considers them as meaning the same thing.

Then we start to define the data. Some of the variables we need to define are simple scalars, e.g. `T <- 2`, the number of interventions. Others are vectors, in which case we need to first define them as `numeric()`. Notice that you can cascade the `->` operator as in `r0 <- r1 <- n0 <- n1 <- numeric()`, which defines several objects as equal to each other and to an empty vector.

The next bit of code defines a R function that we will use to compute the value of the parameters to associate with a logNormal distribution so that the mean and standard deviation *on the natural scale* are (approximately) equal to some input values.

```
# Informative prior on length of influenza episodes
## Compute the value of parameters (mulog,sigmalog) for a logNormal
## distribution to have mean and sd (m,s)
lognPar <- function(m,s) {
  s2 <- s^2
  mulog <- log(m) - 0.5 * log(1+s2/m^2)
  s2log <- log(1+(s2/m^2))
  sigmalog <- sqrt(s2log)
  list(mulog = mulog, sigmalog = sigmalog)
}
```

R functions are defined by the keyword `function` and may or may not have arguments. If you want to specify a function without argument, you can use the following code: `myfn <- function()`. The commands included between the two curly brackets `{` and `}` are those that the function will execute.

In this case, we are specifying that the function called `lognPar` has two inputs (arguments). `m` is the mean that you want your logNormal distribution to have on the natural scale, while `s` indicates its intended standard deviation. The function proceeds to first defining the variance `s2` by squaring the standard deviation; then it creates a new variable `mulog` defined as the mean on the log scale (cfr. Lecture 7) in terms of the mean and standard deviation on the natural scale; then it creates `s2log`, the variance on the log scale, as well as `sigmalog`, the standard deviation on the log scale. Finally, the variables that we want the function to output are included in a (named) list. When this code is "sourced" (or executed), `lognPar` becomes available to your R workspace and for example you can use it using a command like the following.

```
lognPar(3,2)
```

```
$mulog
[1] 0.9147499

$sigmalog
[1] 0.6064031
```

```
x <- rlnorm(10000,lognPar(3,2)$mulog,lognPar(3,2)$sigmalog)
c(mean(x),sd(x),quantile(x,0.025),quantile(x,0.975))
```

```
                      2.5%     97.5%
3.0265977 2.0450362 0.7703335 8.2644729
```

which returns the values you should use to model a logNormal distribution so that its mean and standard deviation match your intended values. You can check that all works the way you want by simulating a variable `x` using the R built-in function `rlnorm` — in this case we can do 10000 simulations using `lognPar(3,2)$mulog` as the mean and `lognPar(3,2)$sigmalog` as the standard deviation. The summary statistics provided above shows that effectively this works perfectly.

In fact, we can use the newly created function to complete the definition of the main data for our model, as shown below.

```
m.l <- 8.2                          # original value in the paper: 8.2
s.l <- sqrt(2)                      # original value in the paper: sqrt(2)
mu.l <- lognPar(m.l,s.l)$mulog      # mean time to recovery (log scale)
sigma.l <- lognPar(m.l,s.l)$sigmalog    # sd time to recovery (log scale)
tau.l <- 1/sigma.l^2                # precision time to recovery (log scale)

# Parameters of unstructured effects
mean.alpha <- 0
sd.alpha <- sqrt(10)
prec.alpha <- 1/sd.alpha^2
mean.mu.delta <- 0
sd.mu.delta <- sqrt(10)
prec.mu.delta <- 1/sd.mu.delta^2
mean.mu.gamma <- 0
sd.mu.gamma <- 1000
prec.mu.gamma <- 1/sd.mu.gamma^2
```

All these are fairly simple. The only thing that is perhaps worth noticing is that the `BUGS` model will use some Normal and logNormal distributions and so we will need to use precisions. For this reason, we create for example the quantity `tau.l`, which is 1 divided by a variance (i.e. a precision), which we can directly use.

We can now call `BUGS` and run the model.

```
# Prepares to launch OpenBUGS
library(R2OpenBUGS)

# Creates the data list
data <- list(S=S,H=H,r0=r0,r1=r1,n0=n0,n1=n1,x=x,m=m,mu.inf=mu.inf,tau.inf=tau.inf,
             mu.l=mu.l,tau.l=tau.l,mean.alpha=mean.alpha,prec.alpha=prec.alpha,
             mean.mu.delta=mean.mu.delta,prec.mu.delta=prec.mu.delta,
             mean.mu.gamma=mean.mu.gamma,prec.mu.gamma=prec.mu.gamma)

# Points to the txt file where the OpenBUGS model is saved
filein <- "EvSynth.txt"

# Defines the parameters list
params <- c("p1","p2","rho","l","c.inf","alpha","delta","gamma")

# Creates a function to draw random initial values
inits <- function(){
    list(alpha=rnorm(S,0,1),delta=rnorm(S,0,1),mu.delta=rnorm(1),
       sigma.delta=runif(1),gamma=rnorm(H,0,1),mu.gamma=rnorm(1),
       sigma.gamma=runif(1),c.inf=rnorm(1))
}

# Sets the number of iterations, burnin and thinning
n.iter <- 10000
n.burnin <- 9500
n.thin <- 20

# Finally calls OpenBUGS to do the MCMC run and saves results to the object "es"
es <- bugs(data=data,inits=inits,parameters.to.save=params,model.file=filein,
    n.chains=2, n.iter, n.burnin, n.thin, DIC=TRUE, working.directory=working.dir)
```

Most of these commands should be fairly familiar by now. We first load `R2OpenBUGS`, then include all the relevant data in a list and define the path to the file where the model is saved (here we assume that the file is in the working directory) and then define the parameters to monitor.

When it comes to defining the inital values, we use this time a bespoke function that we create to simulate suitable values for the variables we want to initialise. For example, the BUGS model defines

```
...
# Evidence synthesis for efffectiveness of NIs
    for (s in 1:S) {
        r0[s] ~ dbin(pi0[s],n0[s])
      ...
        delta[s] ~ dnorm(mu.delta,tau.delta)
        alpha[s] ~ dnorm(mean.alpha,prec.alpha)
    }
...
```

which implies that the node `alpha` is a vector with length `S`. Thus, when we initialise it, we need to provide R and BUGS with `S` values. We do this in our `inits` function by defining `alpha=rnorm(S,0,1)` — this creates `S` random draws from a Uniform(0,1) distribution.

In fact, we are *not* initialising all the unobserved nodes associated with a probability distribution: if you look at the BUGS model code, you will notice that the nodes `phi` and `l` are also of this kind and so, technically, they need initialisation. If we do not do anything, BUGS will take care of it itself. But we can simply add them by simply modifying the `inits` function as following.

```
# Creates a function to draw random initial values
inits <- function(){
    list(alpha=rnorm(S,0,1),delta=rnorm(S,0,1),mu.delta=rnorm(1),
       sigma.delta=runif(1),gamma=rnorm(H,0,1),mu.gamma=rnorm(1),
       sigma.gamma=runif(1),c.inf=rnorm(1)
         # Now add the new variables
         , # Make sure you include a 'comma' between variables!
         l=runif(1),phi=rnorm(1)
         )
}
```

Notice that we need to separate the variables included in the list using commas. You can see what this function does by simply calling it, e.g. `inits()`.

We instruct R and BUGS to run this model for 2 chains — notice we hard-code this in the call to the function `bugs`. You can always do this, although it is *not* best practice (and you are probably better off by creating suitable variables and then referring to them as inputs to functions). We select a burn-in of 9500 iterations and then do a further 10000 iterations, which we thin by 20. This means we run the model for a total of $2 \times (9500 + 10000) = 39000$ iterations and then because we throw away the first 2×9500 and we only store 1 in 20 of the remaining, the final analysis is based on 1000 iterations.

Once BUGS has finished, we regain control of the R session and we can print the output.

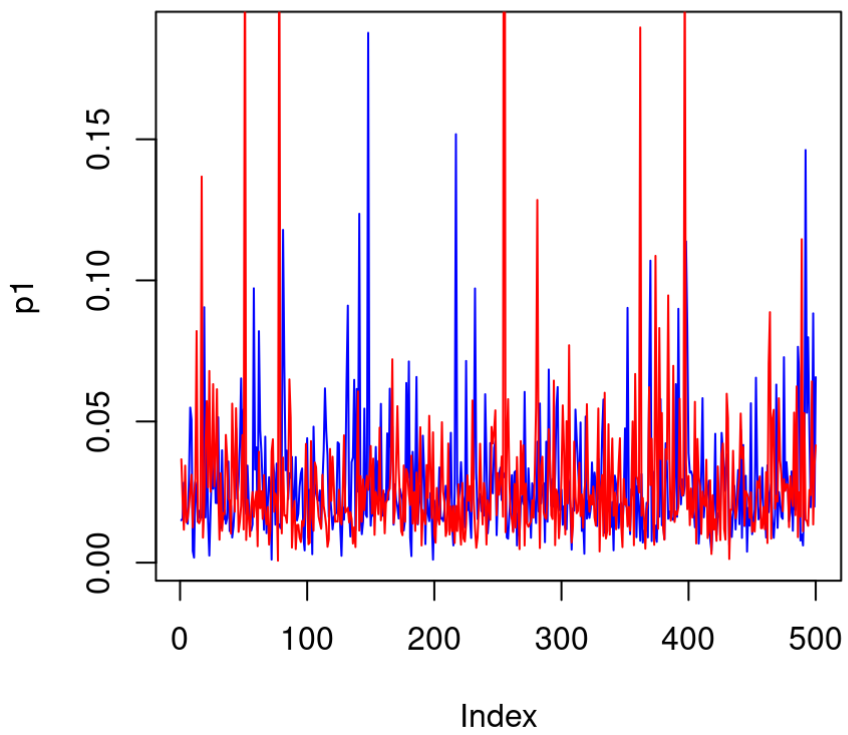
```
# Displays the summary statistics
print(es,digits=2)
```

```
Inference for Bugs model at "EvSynth.txt",
Current: 2 chains, each with 10000 iterations (first 9500 discarded), n.thin = 20
Cumulative: n.sims = 1000 iterations saved
          mean   sd  2.5%   25%   50%   75%  97.5% Rhat n.eff
p1        0.03 0.03  0.01  0.01  0.02  0.03   0.08 1.00  1000
p2        0.01 0.01  0.00  0.00  0.00  0.01   0.02 1.00  1000
rho       0.22 0.09  0.10  0.16  0.20  0.24   0.42 1.01  1000
l         8.16 1.36  5.88  7.20  8.03  8.99  10.97 1.00  1000
c.inf    16.74 2.41 12.04 15.16 16.72 18.19  21.74 1.01   190
alpha[1] -2.68 0.17 -3.03 -2.79 -2.68 -2.57  -2.37 1.00  1000
alpha[2] -2.29 0.16 -2.63 -2.39 -2.28 -2.18  -1.99 1.00  1000
alpha[3] -2.65 0.30 -3.28 -2.84 -2.63 -2.44  -2.10 1.00  1000
alpha[4] -2.62 0.23 -3.08 -2.76 -2.61 -2.46  -2.20 1.00  1000
alpha[5] -3.58 0.35 -4.30 -3.81 -3.56 -3.33  -2.95 1.00  1000
alpha[6] -2.59 0.18 -2.95 -2.70 -2.59 -2.48  -2.25 1.00  1000
delta[1] -1.39 0.32 -1.92 -1.61 -1.40 -1.18  -0.73 1.00  1000
delta[2] -1.72 0.31 -2.37 -1.91 -1.71 -1.52  -1.19 1.01  1000
delta[3] -1.51 0.45 -2.35 -1.80 -1.53 -1.26  -0.56 1.00  1000
delta[4] -1.73 0.40 -2.76 -1.95 -1.69 -1.47  -1.03 1.00  1000
delta[5] -1.37 0.51 -2.31 -1.70 -1.44 -1.10  -0.11 1.00   980
delta[6] -1.91 0.42 -2.90 -2.16 -1.86 -1.62  -1.27 1.02   150
gamma[1] -0.48 0.43 -1.33 -0.76 -0.47 -0.18   0.33 1.01  1000
gamma[2] -5.28 0.86 -7.25 -5.82 -5.19 -4.62  -3.91 1.00  1000
gamma[3] -4.60 0.72 -6.18 -4.99 -4.53 -4.11  -3.37 1.00  1000
gamma[4] -3.47 0.48 -4.47 -3.80 -3.43 -3.13  -2.65 1.00  1000
gamma[5] -4.70 0.46 -5.74 -4.97 -4.66 -4.38  -3.90 1.00   570
gamma[6] -5.37 0.78 -7.08 -5.81 -5.30 -4.84  -4.06 1.00  1000
gamma[7] -3.87 0.56 -5.02 -4.25 -3.85 -3.45  -2.90 1.00   640
gamma[8] -3.34 0.97 -5.45 -3.89 -3.29 -2.66  -1.71 1.01   170
gamma[9] -3.07 0.41 -3.92 -3.34 -3.05 -2.78  -2.34 1.00  1000
deviance 94.40 5.71 84.48 90.46 93.82 97.84 107.20 1.00  1000

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 info (using the rule, pD = Dbar-Dhat)
pD = 16.93 and DIC = 111.30
DIC is an estimate of expected predictive error (lower deviance is better).
```

All seems reasonable — the value for $\hat{R}$ is below the 1.1 threshold for all the monitored nodes and the effective sample size is reasonably large (and close to the nominal value `n.sims`=1000). You can play around with reducing the level of thinning (to increase the sample size) or increasing the total number of iterations to see how the results are affected, but in general terms, the model seems to have reached convergence.

One graphical way of confirming this is to create “traceplots” of the simulations, which you can do using the following code.

```
# Convergence check through traceplots (example for node p1)
plot(es$sims.list$p1[1:500],t="l",col="blue",ylab="p1")
points(es$sims.list$p1[501:1000],t="l",col="red")
```



Here we first plot the first half of the simulations (in this case for the node `p1`). `BUGS` stores the simulations for all the different chains by stacking one chain after the other and thus we do this by accessing the first 500 values of the simulations for `p1`, which are stored inside the `BUGS` object as `es$sims.list$p1`. The `R` notation `[1:500]` instructs `R` to access the positions 1 to 500 of a vector. The options used in the `plot` function specify that we want to plot lines (`t=l`) in blue (`col=blue`) and use a $y-$axis label "p1". Then we add to the existing plot using the `R` built-in function `points`. This has a syntax very similar to `plot` but does not overwrite an existing graph. In this case, we superimpose the elements from position 501 to position 1000 of the vector `es$sims.list$p1` (the simulations for the second chain). We use the options `t=l` and `col=red` to instruct `R` to plot red lines.

As is possible to see, the traceplot "looks good" — the two chains are well mixed and on top of each other, confirming convergence (which ties up with the analysis of `Rhat` and `n.eff` for this particular node). You can try and replicate this analysis for other nodes.

Finally, we need to perform the full economic analysis. We could programme all the commands we need ourselves, but we can use `BCEA` to do most of the work for us.

```
# Attaches the es object to the R workspace (to use the posteriors for the economic analysis)
attach.bugs(es)

# Runs economic analysis
# cost of treatment
c <- e <- matrix(NA,n.sims,T)
c[,1] <- (1-p1)*(c.gp) + p1*(c.gp+c.inf)
c[,2] <- (1-p2)*(c.gp+c.ni) + p2*(c.gp+c.ni+c.inf)
e[,1] <- -l*p1
e[,2] <- -l*p2

library(BCEA)
```

```

Attaching package: 'BCEA'
```

```
The following object is masked from 'package:graphics':

    contour
```

```
treats <- c("status quo","prophylaxis with NIs")
m <- bcea(e,c,ref=2,treats,Kmax=10000)
```

Notice that in this case we "attach" the `BUGS` object `es` to the `R` workspace. This will allow us to access all the relevant quantites stored inside of it directly by calling their name, i.e. we will not need to use the clunky notation `es$sims.list$p1` to access the simulated values for the parameter p_1, but we will just need to call `p1`. This is handy, but, again, we need to be careful because attaching an object will overwrite any other object with the same name that already exists in the `R` workspace.

Once we have done this, we define the suitable economic summaries. We can think of this step as moving from the "Statistical model" to the "Economic model" box (as in here) — cfr. also the slides for Lecture 7. This step is fairly simple – we first create two matrices `e` and `c`, which we will then fill with the simulations for the effectiveness and cost variables for the $T = 2$ treatments considered. Notice that we define `e` and `c` as matrices filled with `NA` (`R` notation for "null" or missing values), with `n.sims`=1000 rows and `T`=2 columns. The notation `c[,1]` should be clear by now. With this, we mean to talk all the rows and only the first column of the matrix `c`. We fill this with suitable values obtained by combining the probabilities of infection and the relevant costs.

At this point, we are ready to load `BCEA` and then call the function `bcea`, which performs the basic economic analysis for us. Before we do so, we define for convenience a vector of names, to associate with the interventions we are considering. In this case, the index 1 is associated with the status quo (so the first column of `c` and `e` contains the simulations for this treatment), while the second is associated with the active intervention.

Finally, we create an object `m` in which we store the output of the call to `bcea`. There are some mandatory and some optional inputs to this function — check `help(bcea)` as well as *BMHE* and *Bayesian Cost-Effectiveness Analysis with the R package BCEA* to see more details. At the very minimum, `bcea` expects that you pass as arguments the matrices including the simulations for effects and costs, in this order. Thus, unless you specify a name for the arguments, `R` will assume you are following the default. For example, the function `bcea` uses the following arguments in exactly this order.

- `e` = a numeric matrix with simulations for the effectiveness variable, for all the treatments considered (must have more than 1 column);
- `c` = a numeric matrix with simulations for the cost variable, for all the treatments considered (must have more than 1 column);
- `ref` = a number to identify the treatment to be considered as the "reference", i.e. the one we are comparing against the other(s). This is an optional argument and unless specified differently, the default is `1`, in which case `BCEA` will assume that the intervention of interest is in the first column of `e` and `c`. In this case, however, we specify `ref=2`;
- `interventions` = a vector of strings of length equal to the number of columns in `e`, giving names to the interventions. This is also an optional argument and unless otherwise specified, `BCEA` will create labels in the form `Intervention1`, `Intervention2`, …;
- `Kmax` = a number specifying the maximum value for the willingness to pay to be considered. The default value is `k=50000`. The willingness to pay is then approximated on a discrete grid in the interval `[0,Kmax]`. The grid is equal to the argument `wtp` — see below — if that parameter is provided, or simply composed of 501 elements if `wtp=NULL` (the default);
- `wtp` = an optional vector containing specific values for the willingness to pay grid. If not specified then `BCEA` will construct a grid of 501 values from 0 to `Kmax` (see point above). This option is useful when performing intensive computations (e.g. for the EVPPI);
- `plot` = a logical value (i.e. `TRUE` or `FALSE`), indicating whether the function should produce the summary plot or not. The default is set to `FALSE`.