Practical 4. Cost-effectiveness analysis with individual level data — SOLUTIONS

Lecture 5 PDF version

This document comments more in details the R script used to perform the whole analysis. Similar results can be obtained using the BUGS interface directly — but as we mentioned in the class, this is usually less efficient and so we focus here on the R script.

The first thing to do is to make sure that you are in the correct folder. You can check the location of your R workspace by typing the following command

```
getwd()
```

which would return something like

```
[1] "/home/gianluca"
```

— the symbol `[1]` here indicates the the answer is a vector and the current rows starts with its first (and, in this case, only) element.

You can move to different directories by using a command like the following

```
setwd("/home/gianluca/Mystuff")
```

which, in this case, would move the workspace in the subfolder `Mystuff`. In `Rstudio` you can also use the menus under `Session` to move across the folders on your computer.

The next step is to load in the R workspace the relevant "packages". In this case, this is done using the following command (that are typeset in red).

```
library(R2OpenBUGS)
```

You can of course load any other package you require at any point in your script.

Question 1

We now proceed to load the data on costs, using the following R command (here the hash symbol "#" indicates a comment)

```
# Loads the data on costs only into R from the txt file (list originally prepared for BUGS)
cost.data=source("cost-data.txt")$value
```

The R command `source` executes into the R workspace the commands included in the file that is used as its argument. In this case, the file `cost-data.txt` contains a *list* of values for a set of different variables and the above comment assigns this list to a new R variable named `cost.data`. Notice here that we also add the suffix `$value` to the `source` command. The reason for this is that in this way, R can strip the values for the elements of the list contained in the `.txt` file from all the "meta-data"" (i.e. external information that is irrelevant to us). For example, compare the following output

```
cost.data=source("cost-data.txt")$value
names(cost.data)
```

```
[1] "N1"    "N2"    "cost1" "cost2"
```

with the one obtained by omitting the `$value` suffix.

```
cost.data=source("cost-data.txt")
names(cost.data)
```

```
[1] "value"   "visible"
```

```
names(cost.data$value)
```

```
[1] "N1"    "N2"    "cost1" "cost2"
```

In the first case, the elements of the object `cost.data` are the variables that we need to use as data for the `BUGS`model. In the second one, these are actually "hidden" inside the object `values` that is stored inside the object `cost.data`.

We are now ready to start setting everything up to run `BUGS`remotely from our `R` session. The first things to do are to define the relevant inputs to the `BUGS` function that will do just that.

```
# Runs the BUGS model from R
model.file="normal-mod.txt"
params <- c("mu","ss","ls","delta.c","dev","total.dev")
n.chains=2
n.burnin=1000
n.iter=2000
n.thin=1
debug=FALSE
```

- The first command here defines a string with the path to the file in which we have saved the model code. This will instruct `BUGS`about where to read the model assumptions.
- The second line defines the parameters to be monitored and stores them (as strings of names) into the vector `params` — of course you can use any naming convention you like; so this vector may be called `parameters`, or `x` instead.
- The third line defines a numeric variable in which we specify the number of Markov chains we want to run — in this case 2. It is usually a good idea to run more than one chain, because by starting them from different initial values, this helps assessing convergence to the target posterior distributions of all the relevant variables in our model.
- The fourth line defines the number of iterations to be used as "burn-in", i.e. to get closer to the core of the target posterior distributions. In this case, we are selecting 1000 — there is no reason why 1000 should *always* be sufficient, so we should assess convergence very carefully (more on this later).
- The fifth line defines the total number of simulations to be run, which will be used to obtain the samples that we will use for the analysis after discarding the burn-in. In this case, we set `n.iter`=2000, which means that, in total, we will run the MCMC for $2 \times (2000) = 4000$ simulations (because we have selected 2 chains). Of these, the first $2 \times 1000 = 2000$ will be discarded as burn-in, which means that the summary statistics will be computed on a sample of 2000 simulations.
- The sixth line defines the "thinning" of the chains. In this case, `n.thin`=1, which means that we are actually storing every single iteration after the burn-in for our analysis. In general, `BUGS`will save 1 every `n.thin` simulations, so using a thinning level above 1 means that some of the simulations will be discarded (and consequently, to have the same overall sample size we will need a longer run of the MCMC). This may help reduce autocorrelation in our results (see the lecture slides and both *BMHE* and *The BUGS Book*, for more details).
- Finally, the seventh line defines a logical variable `debug`. In this case we set it to the value `FALSE`, which means that `BUGS`will be called remotely and we will not see it in action. If we set `debug=TRUE`, then `BUGS`would forcibly take over from `R` and we would see it opening its windows and spitting out its output. This works **under `MS Windows` only**.

At this point we are finally ready to call `BUGS`, which we do using the following command.

```
m=bugs(data=cost.data,inits=NULL,model.file=model.file,parameters.to.save=params,
       n.chains=n.chains,n.iter=n.iter,n.burnin=n.burnin,n.thin=n.thin,DIC=T,debug=debug)
```

The model is run by `BUGS`and while this is happening, we lose access to the `R` session (i.e. you cannot use `R` to make other calculations while `BUGS`is running). When `BUGS`is finished, then `R` takes over again and if everything has worked, the object `m` is stored in our workspace. We can inspect it by using the following command.

```
names(m)
```

```
 [1] "n.chains"        "n.iter"          "n.burnin"        "n.thin"
 [5] "n.keep"          "n.sims"          "sims.array"      "sims.list"
 [9] "sims.matrix"     "summary"         "mean"            "sd"
[13] "median"          "root.short"      "long.short"      "dimension.short"
[17] "indexes.short"   "last.values"     "isDIC"           "DICbyR"
[21] "pD"              "DIC"             "model.file"
```

There are many variables inside the object `m` and we can use the "$" operator in `R` to access them. For example, the command

```
m$n.sims
```

```
[1] 2000
```

returns the total number of simulations used by `BUGS`to compute the posterior distributions.

The first thing to do once the model has run is to check the summary statistics for the posterior distributions, together with the convergence diagnostics. We can do this using the `print` function. For example, the command

```
print(m,digits=2)
```

```
Inference for Bugs model at "normal-mod.txt",
Current: 2 chains, each with 2000 iterations (first 1000 discarded)
Cumulative: n.sims = 2000 iterations saved
             mean    sd    2.5%     25%     50%     75%   97.5% Rhat n.eff
mu[1]       22.72  1.18   20.46   21.90   22.71   23.51   25.10    1  2000
mu[2]       24.53  1.28   22.09   23.69   24.52   25.39   27.12    1  2000
ss[1]      486.18 38.60  418.00  459.65  483.50  509.12  568.61    1  2000
ss[2]      550.47 41.47  477.48  520.80  548.55  576.90  634.80    1  2000
ls[1]        3.09  0.04    3.02    3.06    3.09    3.12    3.17    1  2000
ls[2]        3.15  0.04    3.08    3.13    3.15    3.18    3.23    1  2000
delta.c      1.81  1.74   -1.62    0.63    1.83    3.04    5.06    1  2000
dev[1]    2995.63  2.02 2994.00 2994.00 2995.00 2996.00 3001.00    1  1900
dev[2]    3064.17  2.07 3062.00 3063.00 3064.00 3065.00 3070.00    1  2000
total.dev 6059.77  2.81 6056.00 6058.00 6059.00 6061.00 6067.00    1  2000
deviance  6059.77  2.81 6056.00 6058.00 6059.00 6061.00 6067.00    1  2000

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 = 3.90 and DIC = 6064.00
DIC is an estimate of expected predictive error (lower deviance is better).
```

shows the summary table for the nodes (variables) we have chosen to monitor. The optional input `digits=2` instructs `R` to show the results using 2 significant figures. The table reports the mean, standard deviation and selected quantiles of the posterior distributions. We can typically use the 2.5% and the 97.5% quantiles to approximate a 95% *credible* interval. In addition, the table reports the values for $\hat{R}$, the potential scale reduction (or Gelman-Rubin statistic) and the effective sample size (`n.eff`) — see the lecture slides and both *BMHE* and *The BUGS Book*, for more details.

Given this analysis, the mean cost is 22.72 for the control arm and 24.53 for the intervention arm. The mean difference in costs is 1.81 — recall that costs are entered in `BUGS` $\times 1000$. We can also manipulate the simulations to produce further analyses. For example, we can use the code

```
plot(
    m$sims.list$mu[,1],
    m$sims.list$mu[,2],
    xlab="Population average cost in arm 1 (x 1000)",
    ylab="Population average cost in arm 2 (x 1000)",
    pch=20,
    cex=.6,
    main="Joint distribution of mean costs"
)
```

to display (in the graph below) the posterior joint distribution of the population average costs in the two arms (the `R` script provided for the practical provides also some description of the graphical parameters used in this call).

Notice again the use of the "$" operator to access elements of the object `m`. In this case, the element `sims.list` is a list containing `all` the simulated values for all the monitored nodes. You can actually inspect them with the following code.

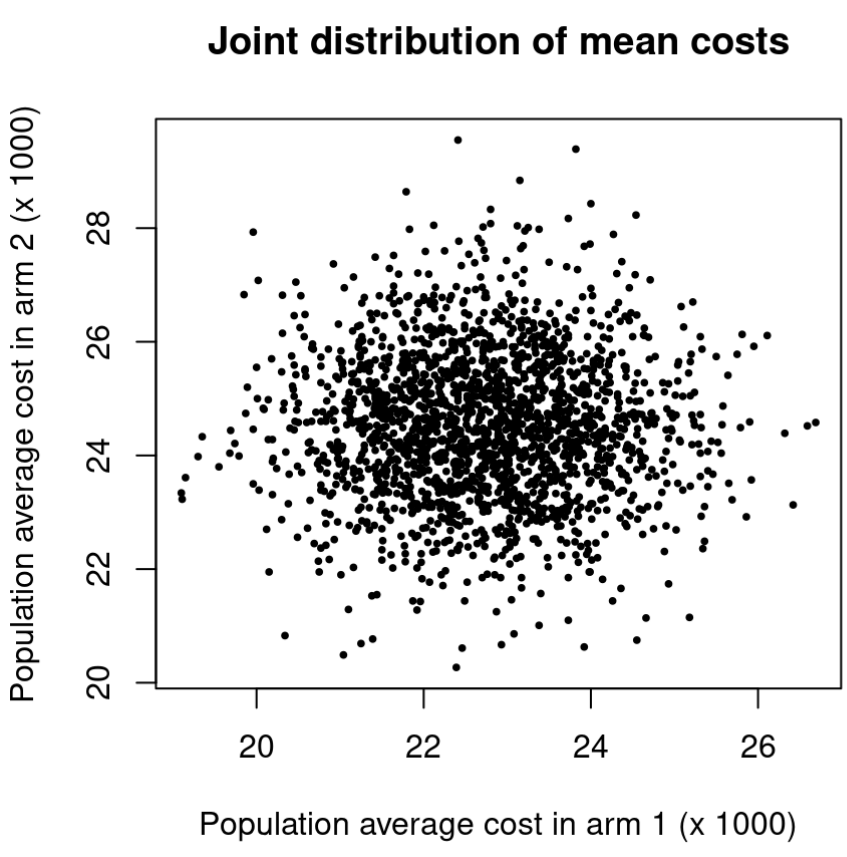
```
names(m$sims.list)
```

```
[1] "mu"        "ss"        "ls"        "delta.c"   "dev"       "total.dev"
[7] "deviance"
```

```
head(m$sims.list$mu[,1])
```

```
[1] 22.72 21.52 23.16 22.62 23.22 23.54
```

The R command `head` shows by default the first 6 values of a variable.



Notice here the interesting fact that BUGSeffectively adds a dimension to each node. So the node `mu` is defined in the model code as a vector with 2 values (one for arm 1 and the other for arm 2). But because when processed by BUGSwe record for each of these two elements `n.sims` simulations, then the resulting object turns into a matrix with `n.sims` = 2000 rows and 2 columns. So, in reference to the code used to generate the plot of the joint distribution of the mean costs, `m$sims.list$mu[,1]` indicates all the rows and the first column and `m$sims.list$mu[,2]` indicates all the rows and the second column of the matrix `m$mu`.

Question 2

We have already included in the vector `params` the nodes related to the deviance — these are `dev` and `total.dev`, which are coded in the BUGSmodel to compute manually the deviance associated with the underlying Normal model. In practice you do not need to do this and BUGSwill calculate (and monitor) the deviance automatically — assuming that there is at least one observed variable (you can think of why this is!).

Going back to the summary statistics table, we can see that the overall model deviance is, on average, 6059.77. This is the same value, whether computed manually (in the node `total.dev`) or automatically (in the node `deviance`).

Question 3

The first thing to do now is to load the new dataset, which includes data on costs as well as utilities. Then we can setup our call to BUGSpointing to the correct model file. Finally, because this new model (based on Gamma-Gamma structure — see the lecture slides) is more complex, we need to provide BUGSwith a set of suitable initial values (identified using a trial-and-error procedure). We do this using the following R commands.

```
# Loads the data on costs only into R from the txt file (list originally prepared for BUGS)
cost.utility=source("cost-util-data.txt")$value

# Specifies the new model file
model.file="cgeg-mod.txt"

# Loads the 3 sets of initial values from the .txt files
inits1=source("cgeg-inits1.txt")$value
inits2=source("cgeg-inits2.txt")$value
inits3=source("cgeg-inits3.txt")$value
# And combines them into a single list
inits=list(inits1,inits2,inits3)

# Re-defines the list of parameters to be monitored
params=c("mu.c","mu.e","delta.c","delta.e","shape.c","shape.e","beta","INB","CEAC")

# Re-defines the burn-in, number of simulations and number of chains
n.burnin=1000
n.iter=4000          # NB: this adds 3000 simulations to the 1000 of burnin
n.chains=3
```

Notice that we need to be consistent in the definition of the number of chains and the set up of the `inits`. So if we select 3 chains, then `inits` needs to be a list comprising of 3 lists (as in this case) — failure to do this will result in `R` complaining and stopping with an error message.

After that, we are ready to call `BUGS`and run the new model, which we do using the following command.

```
m2=bugs(data=cost.utility,inits=inits,model.file=model.file,parameters.to.save=params,
    n.chains=n.chains,n.iter=n.iter,n.burnin=n.burnin,n.thin=n.thin,DIC=T,debug=debug)
```

The results can be again printed in the form of a summary table.

```
print(m2,digits=2)
```

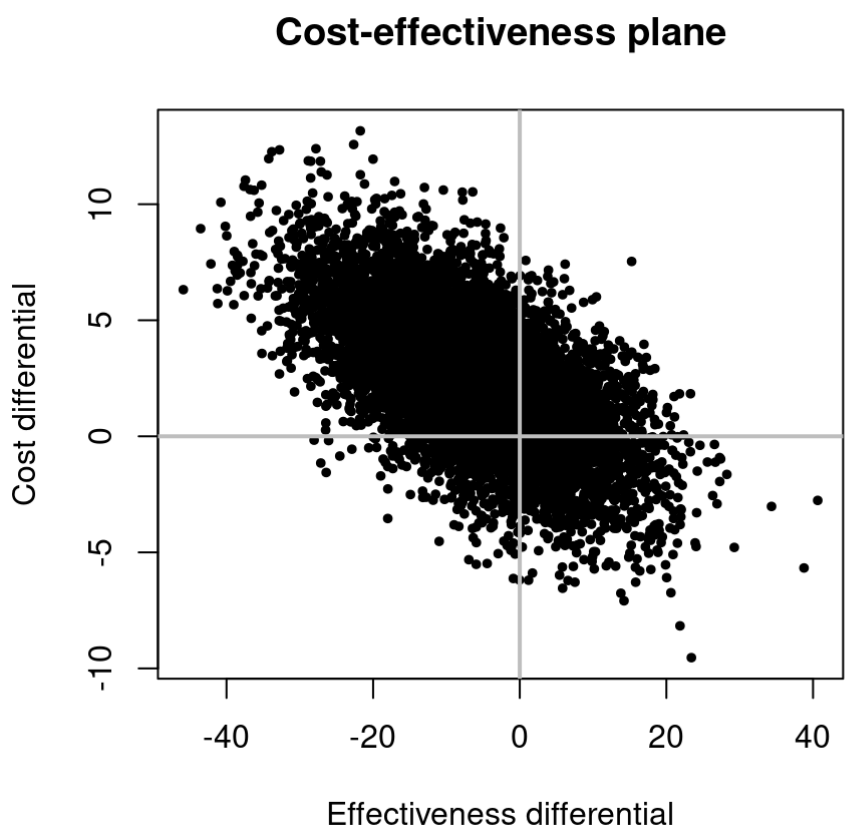
```
Inference for Bugs model at "cgeg-mod.txt",
Current: 3 chains, each with 4000 iterations (first 1000 discarded)
Cumulative: n.sims = 9000 iterations saved
             mean    sd    2.5%     25%     50%     75%   97.5% Rhat n.eff
mu.c[1]     23.47  1.99   20.02   22.09   23.31   24.69   27.83    1   700
mu.c[2]     25.99  2.02   22.47   24.57   25.84   27.24   30.38    1  3000
mu.e[1]     74.70  7.13   61.77   69.81   74.28   79.17   90.15    1  1300
mu.e[2]     81.05  8.14   66.62   75.26   80.63   86.45   97.78    1  6100
delta.c      2.52  2.80   -3.02    0.66    2.55    4.38    7.94    1   660
delta.e     -6.34 10.74  -27.39  -13.84   -6.27    1.02   14.81    1  1800
shape.c[1]   1.18  0.09    1.01    1.12    1.18    1.24    1.36    1  3300
shape.c[2]   1.67  0.13    1.43    1.58    1.67    1.76    1.95    1  9000
shape.e[1]   0.41  0.03    0.35    0.39    0.41    0.43    0.46    1  9000
shape.e[2]   0.37  0.03    0.32    0.36    0.37    0.39    0.43    1  9000
beta[1]      0.16  0.02    0.12    0.14    0.16    0.18    0.21    1  1100
beta[2]      0.17  0.02    0.13    0.15    0.17    0.18    0.21    1  3100
INB[1]      -2.52  2.80   -7.94   -4.38   -2.55   -0.66    3.02    1   660
INB[2]      -3.15  3.57  -10.17   -5.55   -3.18   -0.74    3.81    1   730
INB[3]      -3.79  4.46  -12.59   -6.81   -3.82   -0.74    4.87    1   810
INB[4]      -4.42  5.43  -14.98   -8.15   -4.41   -0.68    6.01    1   900
INB[5]      -5.06  6.42  -17.57   -9.50   -5.05   -0.62    7.23    1   970
INB[6]      -5.69  7.44  -20.19  -10.84   -5.68   -0.61    8.70    1  1000
INB[7]      -6.33  8.47  -23.01  -12.18   -6.36   -0.54   10.07    1  1100
INB[8]      -6.96  9.51  -25.67  -13.52   -6.93   -0.46   11.42    1  1100
INB[9]      -7.59 10.56  -28.38  -14.90   -7.60   -0.39   12.83    1  1200
INB[10]     -8.23 11.61  -31.06  -16.23   -8.20   -0.27   14.26    1  1200
INB[11]     -8.86 12.67  -33.72  -17.60   -8.83   -0.18   15.60    1  1300
CEAC[1]      0.18  0.39    0.00    0.00    0.00    0.00    1.00    1   680
CEAC[2]      0.19  0.39    0.00    0.00    0.00    0.00    1.00    1   810
CEAC[3]      0.20  0.40    0.00    0.00    0.00    0.00    1.00    1   910
CEAC[4]      0.21  0.41    0.00    0.00    0.00    0.00    1.00    1  1200
CEAC[5]      0.22  0.41    0.00    0.00    0.00    0.00    1.00    1  1300
CEAC[6]      0.23  0.42    0.00    0.00    0.00    0.00    1.00    1  2200
CEAC[7]      0.23  0.42    0.00    0.00    0.00    0.00    1.00    1  2800
CEAC[8]      0.23  0.42    0.00    0.00    0.00    0.00    1.00    1  2600
CEAC[9]      0.24  0.43    0.00    0.00    0.00    0.00    1.00    1  2000
CEAC[10]     0.24  0.43    0.00    0.00    0.00    0.00    1.00    1  2200
CEAC[11]     0.24  0.43    0.00    0.00    0.00    0.00    1.00    1  2300
deviance  9756.35  4.44 9750.00 9753.00 9756.00 9759.00 9766.00    1  2600

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 = 9.93 and DIC = 9766.00
DIC is an estimate of expected predictive error (lower deviance is better).
```

As is possible to see, all values for $\hat{R}$ are essentially 1, indicating that the model has converged. Autocorrelation is somewhat large, as confirmed by the fact that the effective sample size is relatively small in comparison to the actual sample size (9000 simulations) for many if not all the nodes. In real-world analyses, this would grant further analysis — for example by inspecting the traceplots of the nodes and possibly running the model for longer or using thinning (see *BMHE*, chapter 4 for `R` code to create traceplots).

The next part consists in manipulating the object `m2` in order to obtain more advanced analyses. Notice that in fact, we can use `BCEA` to do all these, but for the sake of practice, we use our own `R` code, in this case.

The first thing to do is a cost-effectiveness plot. Recall that this is obtained by plotting the joint distribution of Δ_e, Δ_c. Thus, it is sufficient to access the nodes `delta.e` and `delta.c` inside the object `m2` and the `R` function `plot`, as shown below.

```
plot(m2$sims.list$delta.e,m2$sims.list$delta.c,pch=20,cex=.8,xlab="Effectiveness differential",
    ylab="Cost differential",main="Cost-effectiveness plane")
abline(v=0,lwd=2,col="gray")
abline(h=0,lwd=2,col="gray")
```



The command `abline` can be used to add a line to the graph. The input `v=0` indicates a vertical line at 0, while the input `h=0` specifies a horizontal line at 0. The extra parameters `lwd` and `col` specify the width of the line and the color used, respectively (you can use `help(plot)` and `help(colours)` in your R terminal to get more information).

We can use this graph to assess the uncertainty around the overall cost-effectiveness analysis; for example, we can visually assess the proportion of points lying in each of the four quadrants — for instance, the vast majority seems to be in the NW quadrant, where the new intervention is more expensive and less effective.

Next we turn to the analysis of the Incremental Net Benefit (INB). We are asked to assess its value for a willingness to pay of $k =$£500. So, using the code provided in the model file, we can use the following R commands to identify the corresponding index.

```
K=numeric()
K.space=0.1
for (j in 1:11) {
    K[j]=(j-1)*K.space
}
idx=which(K==0.5)
```

Firstly, we recreate in the R workspace the vector of possible willingness to pay thresholds, `K`. Unlike BUGS, R requires us to define any non-scalar quantity before we can use it, e.g. inside a loop. We can do this using the command `K=numeric()`, which effectively creates a new variable `K` and tells R to expect a numeric vector (which can also be length 1, i.e. be a scalar). Next, we set the step of 0.1 (=£100) in the variable `K.space` and use it to fill in the vector `K`. Notice that in R we create loops using the `for` function, which takes as arguments

- the index (in this case `j`);
- the starting point (in this case 1);
- the ending point (in this case 11).

The R notation is fairly straightforward as basically instructs the computer to move `j` "in 1 to 11" — this means that R will repeat the commands specified inside the loop (delimited by the two curly brackets "{" and "}") upon varying the index `j` from 1 to 2, 3, …, 11.

The resulting vector is

```
K
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

and the variable `idx` is computed as the element of `K` that is exactly equal to 0.5, as defined in the function `which` above. Notice that in logical functions (e.g. `if`, `while`, `which`), `R` requires that the equality condition is represented by "`==`" (notice the double sign of equality).

At this point, we can use the `R` built-in functions `mean` and `quantile` to estimate the average and 95% interval for the INB at $k = \pounds 500$. We can do this using the following commands.

```
mean(m2$sims.list$INB[,idx])
```

```
[1] -5.691401
```

```
quantile(m2$sims.list$INB[,idx],.025)
```

```
    2.5%
-20.19075
```

```
quantile(m2$sims.list$INB[,idx],.975)
```

```
  97.5%
8.70215
```

Notice again that `INB` is defined as a vector in the `BUGS`model (with one value for each value of `K`). This means that resulting object from the `BUGS`simulations is turned into a matrix (increased by one dimension) and what we need is to access all the rows of the `idx`-th column (which is associated with $k = \pounds 500$).

We can manipulate the simulation further to obtain an estimate of the probability that the INB is positive at this threshold. This can be easily obtained by computing the proportion of simulations for which this is true, which we do using the following command.

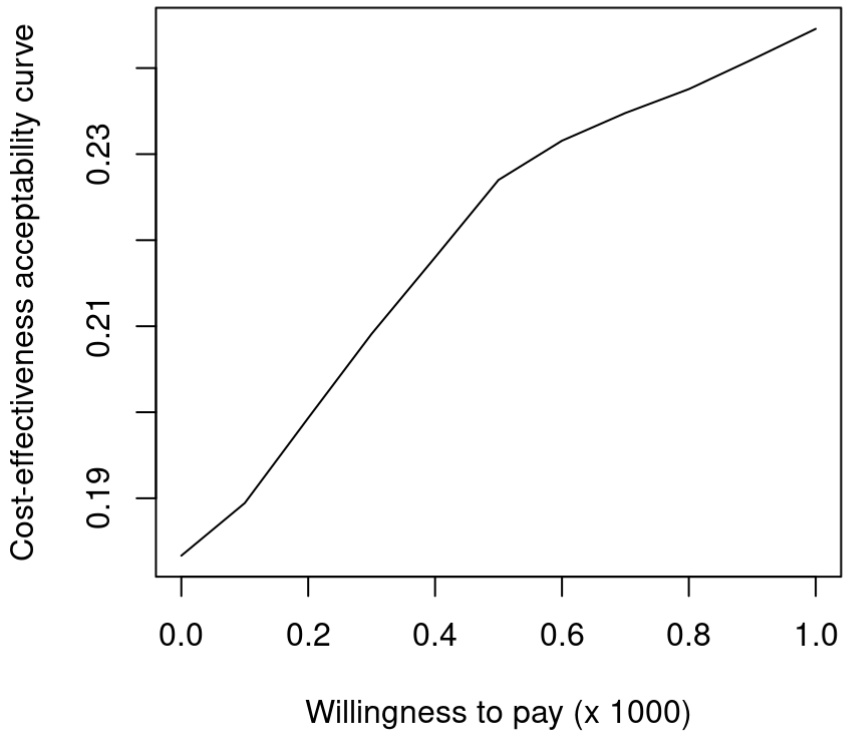
```
sum(m2$sims.list$INB[,idx]>0)/m2$n.sims
```

```
[1] 0.227
```

The logical expression `m2$sims.list$INB[,idx]>0` checks whether each of the `m2$n.sims` simulations in the `idx`-th column of the matrix `m2$sims.list$INB` is positive. If so, it returns a 1, while if not, it will return a 0. Thus to sum over all these 1s and 0s and then divide by the total number of simulations, will provide an estimate of the probability that INB is positive.

Finally, we can compute and plot the Cost-Effectiveness Acceptability Curve (CEAC), using the following code.

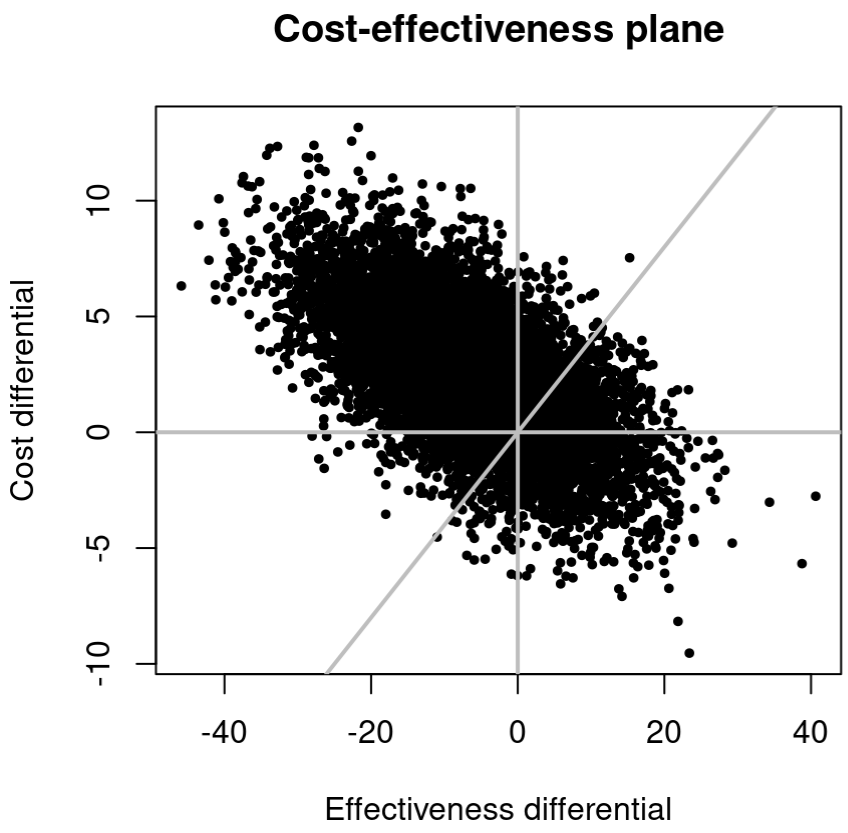
```
CEAC=numeric()
for (i in 1:length(K)) {
    CEAC[i]=mean(m2$sims.list$CEAC[,i])
}
plot(K,CEAC,xlab="Willingness to pay (x 1000)",ylab="Cost-effectiveness acceptability curve",
     main="",t="l")
```



Firstly, recall that the CEAC is in fact the probability that INB is positive, for each selected value of the willingness to pay. So we define a vector `CEAC`, in which we want to store the best estimate (i.e. the mean of the posterior distribution) from our model. To do this, we create a `for` loop. Notice that this time, we are being a bit clever and instead of specifying the ending point of the loop, we let `R` compute it itself by defining it equal to `length(K)` — that is, obviously, the length of the vector `K`.

Once the vector `CEAC` has been filled in, we can simply plot it agains the values of the willingness to pay using the `plot` function. Notice that, by default, `plot` uses open dots to display the selected values. To overwrite this behaviour, we need to specify the option `t=l`, which instructs `R` to use a line (curve) instead.

We can link the CEAC with the Cost-Effectiveness plane by noticing that the former is essentially the proportion of "futures" (i.e. simulated points in the latter) that lie below the line $\Delta_e = k\Delta_c$, for a given willingness to pay threshold, k.



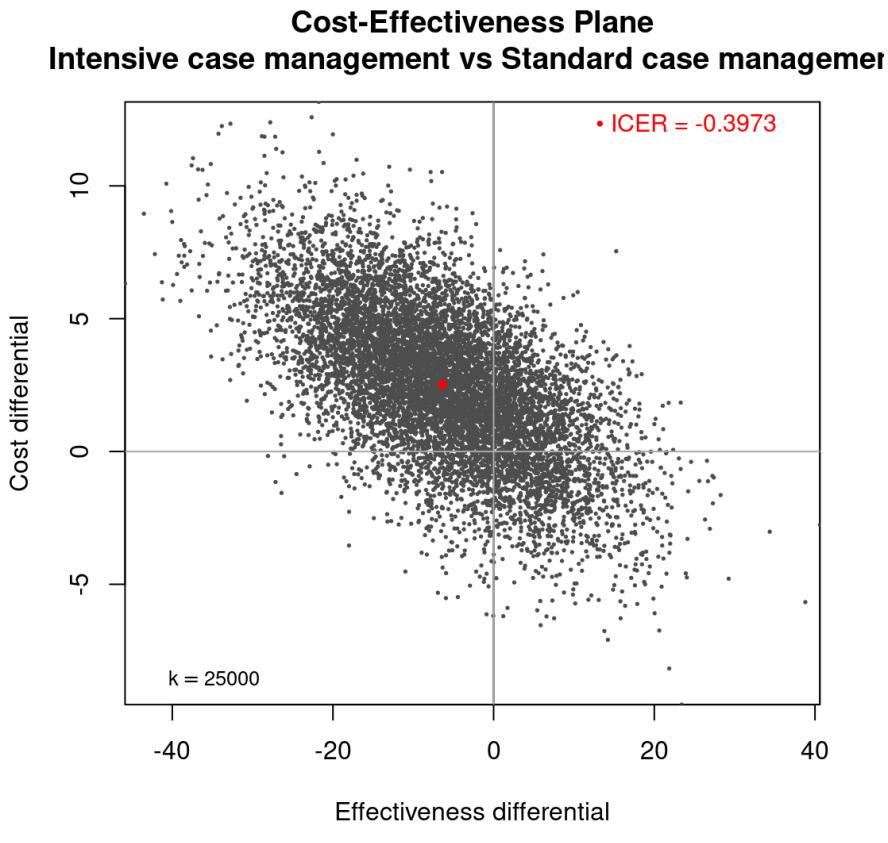
For example, the graph above plots the line $\Delta_e = 0.4\Delta_c$, which implies we're considering $k = 0.4$ (recall that the costs are scaled by £1000, so in fact, we mean k =£400!). As is possible to see, most of the points lie **above** the willingness to pay. In other words, it is much more likely that the intervention $t = 1$ is *not* cost-effective. The proportion of points below the line (BMHE refers to this as the "sustainability area") is below 0.5 and it indicates, for that given willingness to pay, the CEAC.

In reality, we do not really need to perform the economic evaluation "by hand", ie programming the code above to compute the various health economic summaries and graphical representations. This is the point of `BCEA`!… In this case, the output of the `BUGS` model does contain the population average measures of costs and effectiveness — these are the parameters `mu.c` and `mu.e`. We can simply pass these as "input" to `bcea` and obtain all the necessary and relevant economic summaries and plots. For instance, we can use the following code to create two matrices `e` and `c`, which we can then feed to `BCEA` as input.

```
# Defines the population average "effectiveness measures" (NB: days in hospital are bad so take -e!
e = -m2$sims.list$mu.e
# Defines the population average "cost measures"
c = m2$sims.list$mu.c
# Calls BCEA
library(BCEA)
# Runs the function `bcea` to obtain the health economics summary
he=bcea(
  # Defines the inputs
  e,c,
  # Labels for the two intervention groups
  interventions=c("Standard case management","Intensive case management"),
  # Defines the "reference" interventions (the one that is evaluated)
  ref=2,
  # Selects the maximum value for the willingness to pay threshold
  Kmax=1000
)
# Now can summarise the decision problem
summary(he)
```

```
NB: k (wtp) is defined in the interval [0 - 1000]

Cost-effectiveness analysis summary

Reference intervention:  Intensive case management
Comparator intervention: Standard case management

Standard case management dominates for all k in [0 - 1000]


Analysis for willingness to pay parameter k = 1000

                         Expected utility
Standard case management            -74728
Intensive case management           -81073

                                                      EIB    CEAC    ICER
Intensive case management vs Standard case management -6345.2 0.28222 -0.3973


Optimal intervention (max expected utility) for k = 1000: Standard case management

EVPI 5007.4
```

```
# Or plot the results
ceplane.plot(he)
```



Running the model using R2jags

In general, there aren't many differences in running a model using `JAGS` or `BUGS`. In this particular case, however, a "vanilla" run of the code in the file `IPD_analysis.R` may give some interesting inconsistency.

> ⚠ You do **NOT** need to run the model using `R2jags` — this is just so you are aware of the potential issues. Or, of course, in case you *are* running `JAGS` and have encountered this first hand!

The script basically is identical — the only differences are that

- You load the package `R2jags` instead of `R2OpenBUGS`;
- You call the function `jags` instead of the function `bugs`. Specifically, the call to `jags` should **not** include the option `debug`, which is specific to `R2OpenBUGS`.

However, if you run the script and call

```
model.file="normal-mod.txt"
params <- c("mu","ss","ls","delta.c","dev","total.dev")
n.chains=2
n.burnin=1000
n.iter=2000
n.thin=1
m=jags(data=cost.data,inits=NULL,model.file=model.file,parameters.to.save=params,
       n.chains=n.chains,n.iter=n.iter,n.burnin=n.burnin,n.thin=n.thin,DIC=T)
```

interestingly the summary table looks like this.

```
print(m,digits=2)
```

```
Inference for Bugs model at "normal-mod.txt", fit using jags,
 2 chains, each with 2000 iterations (first 1000 discarded)
 n.sims = 2000 iterations saved
           mu.vect sd.vect     2.5%      25%      50%      75%    97.5% Rhat n.eff
delta.c       1.57    9.44   -17.92    -4.38     1.55     7.24    20.43 1.10  1500
dev[1]     3855.90   82.27  3691.47  3780.61  3909.93  3927.10  3936.55 4.07     2
dev[2]     3583.62  403.59  3062.69  3079.54  3831.10  3962.03  3973.30 6.47     2
ls[1]         4.87    0.13     4.61     4.76     4.96     4.98     5.00 4.04     2
ls[2]         4.27    0.78     3.13     3.31     4.78     4.98     5.00 4.95     2
mu[1]        23.03    7.37     9.02    18.15    23.03    27.71    37.95 1.05  2000
mu[2]        24.60    5.90    11.22    22.26    24.71    27.12    37.77 1.25    96
ss[1]     17582.19 4099.19 10189.24 13518.36 20224.03 21270.14 21888.23 4.10     2
ss[2]     11626.39 9770.87   525.79   756.32 15145.23 21154.70 21903.90 5.50     2
total.dev  7439.52  483.20  6755.59  6859.67  7742.10  7888.03  7909.25 6.12     2
deviance   7439.52  483.20  6755.59  6859.67  7742.10  7888.03  7909.25 6.12     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 info (using the rule, pD = var(deviance)/2)
pD = 15432.5 and DIC = 22872.0
DIC is an estimate of expected predictive error (lower deviance is better).
```

The exact same model, with the exact same data, shows different results with evidence of **very** poor mixing in the chains (high values of `Rhat` and very low values for `n.eff`)!

The reason for this is in the different way in which `OpenBUGS` and `JAGS` handle the generation of initial values: the former uses a random draw from the prior distribution, while the latter uses values that are restricted to be in the proximity of the mean of the prior distribution[1].

In this case, `R2OpenBUGS` does a better job at selecting initial values that are closer to the main mass in the posterior distribution, which means that the process converges much more easily and quickly. The process can be "saved" either by running the chains for longer, or by selecting "better" initial values. For instance, running the code

```
m=jags(data=cost.data,inits=NULL,model.file=model.file,parameters.to.save=params,
       n.chains=n.chains,n.iter=10000,n.burnin=9000,n.thin=n.thin,DIC=T)
```

(which creates 10000 simulations, discards the first 9000 and thus saves 1000 per chain — or 2000 in total). The summary statistics look *much* better now:

```
print(m,digits=2)
```

```
Inference for Bugs model at "normal-mod.txt", fit using jags,
 2 chains, each with 10000 iterations (first 9000 discarded)
 n.sims = 2000 iterations saved
          mu.vect sd.vect    2.5%     25%     50%     75%   97.5% Rhat n.eff
delta.c      1.84    1.74   -1.47    0.65    1.85    3.02    5.29 1.00  2000
dev[1]    2995.78    2.21 2993.68 2994.23 2995.08 2996.58 3001.70 1.01  2000
dev[2]    3064.15    1.90 3062.28 3062.78 3063.62 3064.85 3069.33 1.00   510
ls[1]        3.09    0.04    3.01    3.07    3.09    3.12    3.17 1.00  1200
ls[2]        3.16    0.04    3.09    3.13    3.15    3.18    3.24 1.00   590
mu[1]       22.68    1.27   20.27   21.82   22.66   23.51   25.23 1.00   710
mu[2]       24.52    1.23   22.17   23.65   24.53   25.37   26.93 1.00  2000
ss[1]      487.60   39.09  414.84  461.18  485.30  512.50  568.95 1.00  1200
ss[2]      552.24   42.80  478.19  523.20  549.34  578.39  645.84 1.00   590
total.dev 6059.94    2.94 6056.35 6057.80 6059.19 6061.37 6067.45 1.00  2000
deviance  6059.94    2.94 6056.35 6057.80 6059.19 6061.37 6067.45 1.00  2000

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 = var(deviance)/2)
pD = 4.3 and DIC = 6064.3
DIC is an estimate of expected predictive error (lower deviance is better).
```

All the `Rhat` statistics are below 1.1 and the `n.eff` values are much closer, in general, to the "nominal" sample size `n.sims=2000`. Even better, we can pass "reasonable" initial values and ensure even better and quicker convergence. For instance, we could use the files `normal-initis1.txt` and `normal-inits2.txt`, which can be loaded into the workspace and stored in a `list` using the following code.

```
inits=list(source("normal-inits1.txt")$value,source("normal-inits2.txt")$value)
inits
```

```
## [[1]]
## [[1]]$mu
## [1] 0.316036 1.282370
##
## [[1]]$ls
## [1] 0.437099 0.493518
##
##
## [[2]]
## [[2]]$mu
## [1] 1.73574 1.31659
##
## [[2]]$ls
## [1] -1.96202 -1.06980
```

We can run the original model by specifying these

```
m=jags(data=cost.data,inits=inits,model.file=model.file,parameters.to.save=params,
       n.chains=n.chains,n.iter=n.iter,n.burnin=n.burnin,n.thin=n.thin,DIC=T)
```

and see that convergence is *not* an issue, even with simply `n.iter=` 2000 total iterations and `n.burnin=` 1000 discarded as burn-in.

```
print(m,digits=2)
```

```
Inference for Bugs model at "normal-mod.txt", fit using jags,
 2 chains, each with 2000 iterations (first 1000 discarded)
 n.sims = 2000 iterations saved
          mu.vect sd.vect    2.5%     25%     50%     75%   97.5% Rhat n.eff
delta.c      1.85    1.81   -1.83    0.70    1.85    3.08    5.26 1.00   350
dev[1]    2995.70    2.05 2993.69 2994.23 2995.07 2996.50 3001.27 1.01   300
dev[2]    3064.35    2.07 3062.29 3062.87 3063.71 3065.18 3069.96 1.00  2000
ls[1]        3.09    0.04    3.02    3.07    3.09    3.12    3.17 1.00  2000
ls[2]        3.15    0.04    3.08    3.13    3.15    3.18    3.23 1.00   700
mu[1]       22.70    1.23   20.33   21.87   22.70   23.53   25.10 1.00   530
mu[2]       24.55    1.35   21.84   23.63   24.57   25.45   27.15 1.00   950
ss[1]      487.09   38.66  416.90  459.81  484.81  512.02  571.41 1.00  2000
ss[2]      550.44   43.00  475.19  519.92  547.79  578.13  642.69 1.00   690
total.dev 6060.04    2.91 6056.41 6057.88 6059.37 6061.52 6067.23 1.00   740
deviance  6060.04    2.91 6056.41 6057.88 6059.37 6061.52 6067.23 1.00   740

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 = var(deviance)/2)
pD = 4.2 and DIC = 6064.3
DIC is an estimate of expected predictive error (lower deviance is better).
```

---

1. The JAGS manual states on page 16 that "*initial values are not supplied by the user, then each parameter chooses its own initial value based on the values of its parents. The initial value is chosen to be a 'typical value' from the prior distribution. The exact meaning of 'typical value' depends on the distribution of the stochastic node, but is usually the mean, median, or mode*" ↩

PREVIOUS
Linear regression Tutorial