Practical 3. Introduction to R and cost-effectiveness analysis using BCEA - SOLUTIONS

Lecture 3 PDF version

Introduction to R and cost-effectiveness analysis using BCEA — SOLUTIONS

MCMC in R/BUGS

The model file is fairly simple in this case and it is coded as in the following

```
# Model for Laplace's analysis of birth's data
model {
    y ~ dbin(theta,n)
    theta ~ dunif(0,1)
}
```

which you can save on a file `ModelLaplace.txt`. Of course, the naming convention is completely up to you and you can use any name or file extension, for that matter, e.g. something like `model.bug` is perfectly acceptable to BUGS, as long as all is consistent in your call from R

The next step consists into inputting the data in the R workspace. Assuming you have opened R, you can point it in the "working" directory, i.e. the folder in which you have stored the relevant files, say something like `C:\gianluca` if you are under `MS Windows`. R uses `Unix` notation and thus paths to folder are characterised by a slash (not a forward slash, as in `MS Windows`). Thus you can point R to the relevant directory using the `setwd` command, for example as in the following.

An alternative way of specifying a path in R is to use "double forward slash", as in the following

To R , both commands have the same meaning.
Now we can start inputting the data in the workspace. Because the example is so simple, we can simply write down the values we want to associated with each of the relevant variables, for example as in the following commands.

```
y=241945
n=493527
data=list(y=y,n=n)
```

The first two commands define new variables `y` and `n` and assign them the relevant values. The third line creates a new R object — that is a *list* in which we store `y` and `n`. Lists are very helpful because they act as boxes in which you can put variables of different nature (scalars, matrices, strings, numbers, etc.). The way R defines variables and lists in particular is rather straightforward: notice however that you can assign names to elements of a list — we do this here by using the option `y=y,n=n`.
You can inspect the list by using various R commands, for example as in the following.

```
names(data)
```

```
[1] "y" "n"
```

```
data$y
```

```
[1] 241945
```

The command `names` returns the names of the objects included inside the data list, while the "$" operator allows us to access objects contained inside other objects. So the command `data$y` looks inside the object `data` and returns the value of the object `y`.

The next step we need to follow before we can run our model using BUGSis to set up the environment and main variables for the MCMC. We do this using the R commands below.

```
filein="ModelLaplace.txt"
params="theta"
inits_det=list(list(theta=.1),list(theta=.9))
inits_ran=function(){list(theta=runif(1))}
```

The first one creates a string variable with the path to the model file. In this case, we assume that the `.txt` file is stored in the working directory and so all that is necessary is a pointer to the file name. If the `.txt` were stored somewhere, e.g. in the folder `C:\gianluca\myfiles`, then we would need to specify the full path to the file, e.g. `filein=C:/gianluca/myfiles/ModelLaplace.txt`. Of course, there is nothing special about the name `filein` — and you can use any name you like, provided you are then consistent in the call to `BUGS`.

The second command defines the parameters to be monitored. In this case, the model is so simple that it only has one parameter, so we simply define a string variable `params`. If we had more than one parameter, we would need to create a vector of names — in `R` we can do this by using the "collection" operator, for example as in the following.

```
params=c("theta1","theta2","theta3")
```

The third and fourth commands show two different ways of creating initial values to pass to `BUGS`. The first one uses deterministic initial values, in the sense that you are passing specific values to `R` , which it in turn will send to `BUGS`. If you want to do this, you need to create a "list of lists". In other words, first you define a variable (in this case `inits_det`) as a list. Then, you create other lists inside it — each of this list should contain a specific value for each of the variables you need to initialise. The number of lists inside the upper-level list is defined by the number of Markov chain you want to run (see below and the lecture slides). You can explore the list you have just created by simply calling its name.

```
inits_det
```

```
[[1]]
[[1]]$theta
[1] 0.1


[[2]]
[[2]]$theta
[1] 0.9
```

This may be a bit daunting at first glance, but once you get your head around it, the way in which `R` manages its objects is actually very informative and very much linked with its object-oriented programming nature. The output is characterised by a series of "tags". For example, the tag `[[1]]` indicates the first element (i.e. the first list included) in the main object (i.e. the variable `inits_det`). So for example, you can access elements inside an object using the "'double square bracket" and/or the "$" notation, for example as in the following.

```
inits_det[[1]]
```

```
$theta
[1] 0.1
```

```
inits_det[[1]]$theta
```

```
[1] 0.1
```

The lower level tag `[1]` is used to indicate the variables stored inside the upper-level element of the object (e.g. the value `0.1` that we have assigned to `theta` in the first list inside `inits_det`).

The fourth command defines "random" initial values. To do so, instead of you passing a specific value, you can create a `R` function that simulates them from a suitable probability distribution. In this case, we know that `theta` is defined in our model code to be a probability and so it has to range between 0 and 1. As we have seen in the classes, a Uniform distribution may be a good candidate and thus in this case we set up a little function that creates a list and puts inside it a number generated randomly from a Uniform distribution. In `R`, built-in functions such as `runif` have default values. So a call

```
runif(1)
```

```
[1] 0.06749364
```

randomly samples 1 value from a Uniform distribution defined in the default range $[0;1]$. You can get more information on `R` functions typing the command `help(name_of_the_function)` to your `R` terminal.

At this point we are finally ready to call `BUGS`in background and do the MCMC simulation for us. We can do so using the following code.

```
library(R2OpenBUGS)
model= bugs(data=data,inits=inits_det,parameters.to.save=params,model.file=filein,
              n.chains=2,n.iter=10000,n.burnin=4500,n.thin=1,DIC=TRUE)
```

Firstly we load the package `R2OpenBUGS`, which allows us to link `R` to `BUGS`. Then we define a new object `model`, which will collect the output of the call to the function `bugs`. This takes several arguments. In this case, we are using the deterministic initial values and considering 2 chains. We ask `BUGS`to do a burn-in (cfr. slides as well as *BMHE* and *The BUGS Book*) of 4500 iterations and then simulate for another 10000 iterations after that. We do not require thinning (or, equivalently, a thinning of 1, which simply means that all the 10000 simulations after burn-in are stored for our analysis). Finally, we instruct `BUGS`to compute the DIC (more on this later — do not worry about it for now).

The commands

```
names(model)
```

```
 [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"     
```

```
model$n.iter
```

```
[1] 10000
```

will return the list of names for all the elements included in the `model` object; and access the value of the element `n.iter` included inside the object `model`, respectively.

The `R2OpenBUGS` has a “print” method — meaning that we can display the results of our model in a tabular form, simply using the following command (the option `digits=3` defines the number of significant figures to be displayed).

```
print(model,digits=3)
```

```
Inference for Bugs model at "ModelLaplace.txt", 
Current: 2 chains, each with 10000 iterations (first 4500 discarded)
Cumulative: n.sims = 11000 iterations saved
           mean    sd   2.5%   25%   50%    75%  97.5%  Rhat n.eff
theta     0.490 0.001  0.489  0.49  0.49  0.491  0.492 1.001 11000
deviance 14.562 1.427 13.560 13.66 14.02 14.870 18.740 1.002  3200

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 = 1.002 and DIC = 15.560
DIC is an estimate of expected predictive error (lower deviance is better).
```

This returns a table with summary statistics from the posterior distribution of the nodes we have instructed R and BUGSto monitor. Notice that BUGSwill automatically monitor a node deviance — see Lecture 5. For now, let us not worry about it!

In this case, we can easily assess that the estimate for the probability of a girl being born is on average 0.490 and we can approximate a 95% *credible* interval by considering as lower and upper limits the 2.5% and 97% quantiles of the posterior distribution. In this case, the interval estimate is 0.489 to 0.492. Because the entire interval is below the threshold of 0.5, Laplace concluded in his analysis that it was "morally certain" that a boy birth was more likely than a girl (see *BMHE*, chapter 2).

In any MCMC analysis, convergence is a crucial point. Because the Uniform distribution (that we have used as prior for theta) can be considered as a special case of the Beta distribution, we know from Lecture 3 that actually ours is a case of Binomial-Beta conjugate model. Thus, we do not really have issues with convergence and in fact the BUGSmodel is not even really a MCMC exercise (the simulations are still obtained but all the calculations are done in close-form because of conjugacy). This is reflected in the value of the potential scale reduction factor (or Gelman-Rubin statistic $\hat{R}$), described in the column headed Rhat, which show values well below the threshold of 1.1. Similarly, autocorrelation is not an issue as the effective sample size (described in the column headed neff) is exactly identical with the actual sample size. In particular, notice that: we have 2 chains, we run the model for 10000 iterations per chain after a burn-in of 4500 iterations per chain that are discarded. This makes the total number of simulations used for our analysis $2 \times (10000 - 4500) = 11000$. This number is stored also in the object model$n.sims.

A handy workaround that allows us to avoid using the "$" or "double square bracket" notation is to "attach" an object to the workspace. This means that we effectively bypass the upper-level object and make its elements available directly to R and we can do this by using the command

```
attach.bugs(model)
```

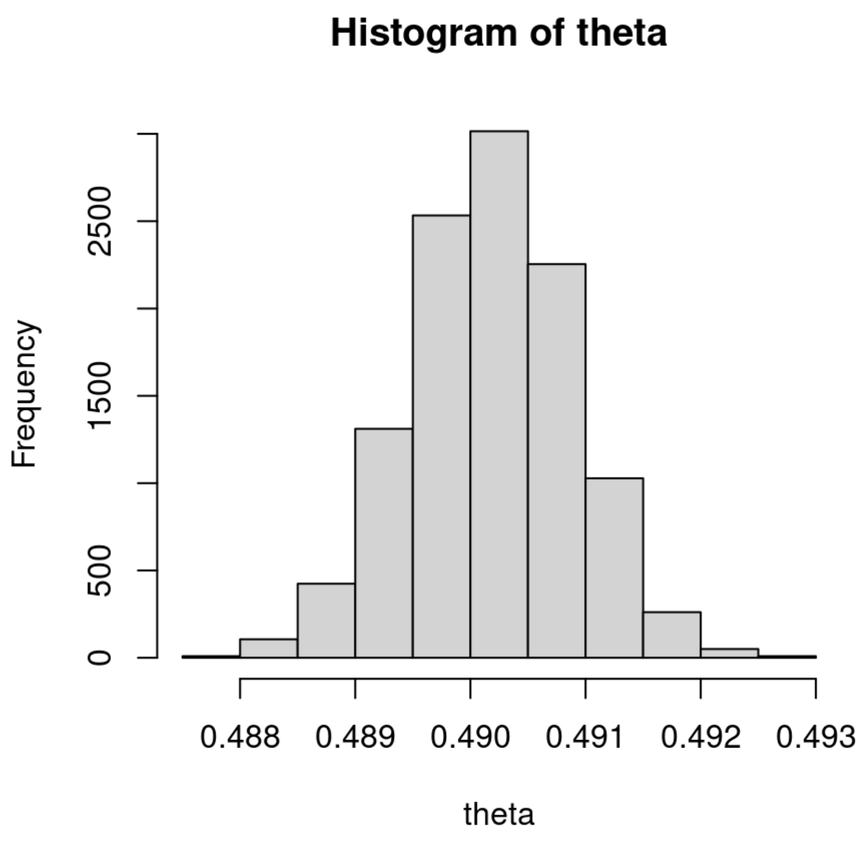
NB: It is possible that this command does not produce the required effect (in newer versions of R2OpenBUGS). An alternative that should work is

```
attach.all(model)
```

(i.e. use the function attach.all, still from the package R2OpenBUGS, where the argument in brackets is the name of the BUGS model for which you want to make all the output directly available on your R workspace).

Notice that this operation comes with some risks — every variable that is currently present in the R workspace with the same name as a variable included in the object model will be now overwritten. Nevertheless, we can now use all the elements of model directly; for example we can produce a histogram of the posterior distribution for theta simply using the following command.

```
hist(theta)
```



which confirms that, effectively, there is no chance that theta is greated than 0.5.

Health economic evaluation in R using BCEA

First we load a dataset that has been previously saved. There are several format to which you can save R data or R workspaces — in this case we use the format `.RData`. We can load a `.RData` file simply using the built-in R command `load`.

```
load("Vaccine.RData")
names(he)
```

```
 [1] "n_sim"         "n_comparators" "n_comparisons" "delta_e"
 [5] "delta_c"       "Kmax"          "k"             "ib"
 [9] "eib"           "kstar"         "best"          "ref"
[13] "comp"          "step"          "interventions" "delta.e"
[17] "delta.c"
```

The file `Vaccine.RData` contains the object `he`, which in turns has several elements in it.

We can now load the package BCEA (assuming you have already installed it) and use it to post-process `he` and produce relevant summaries/analyses.

The practical uses the **current** version of BCEA, which is available from CRAN. However, the development version is in the "beta-testing" version and will soon replace the current stable version. As this will be a **major** update, some of the commands below *may* break in the future — differences will be minor and changes will be easy to address.

```
library(BCEA)
```

```

Attaching package: 'BCEA'
```

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

    contour
```

```
ICER=mean(he$delta.c)/mean(he$delta.e)
ICER
```

```
[1] 20097.59
```

The first thing we need to do is to compute the ICER. If you check with your slides for Lecture 3, you will see that

$$\text{ICER} = \frac{\text{E}[\Delta_c]}{\text{E}[\Delta_e]}$$

and so we create a new variable `ICER` to which we assign as value the ratio of the mean of the element `he$delta.c` to the mean of the element `he$delta.e`. Its value is returned to be 20097.59. The only difficulty of this part is to realise how to translate the correct formula for the ICER into R commands (i.e. you need to take the ratio of the means, not the mean of the ratio!) and to access the elements `delta.e` and `delta.c` from inside the main object `he`.

Secondly, we are asked to compute the Expected Incremental Benefit (EIB) for a value of the willingness to pay threshold $k = 30000$. Now, from the Lecture we know that

$$\text{EIB} = k\text{E}[\Delta_e] - \text{E}[\Delta_c]$$

and so we can easily compute this using the following R commmands.

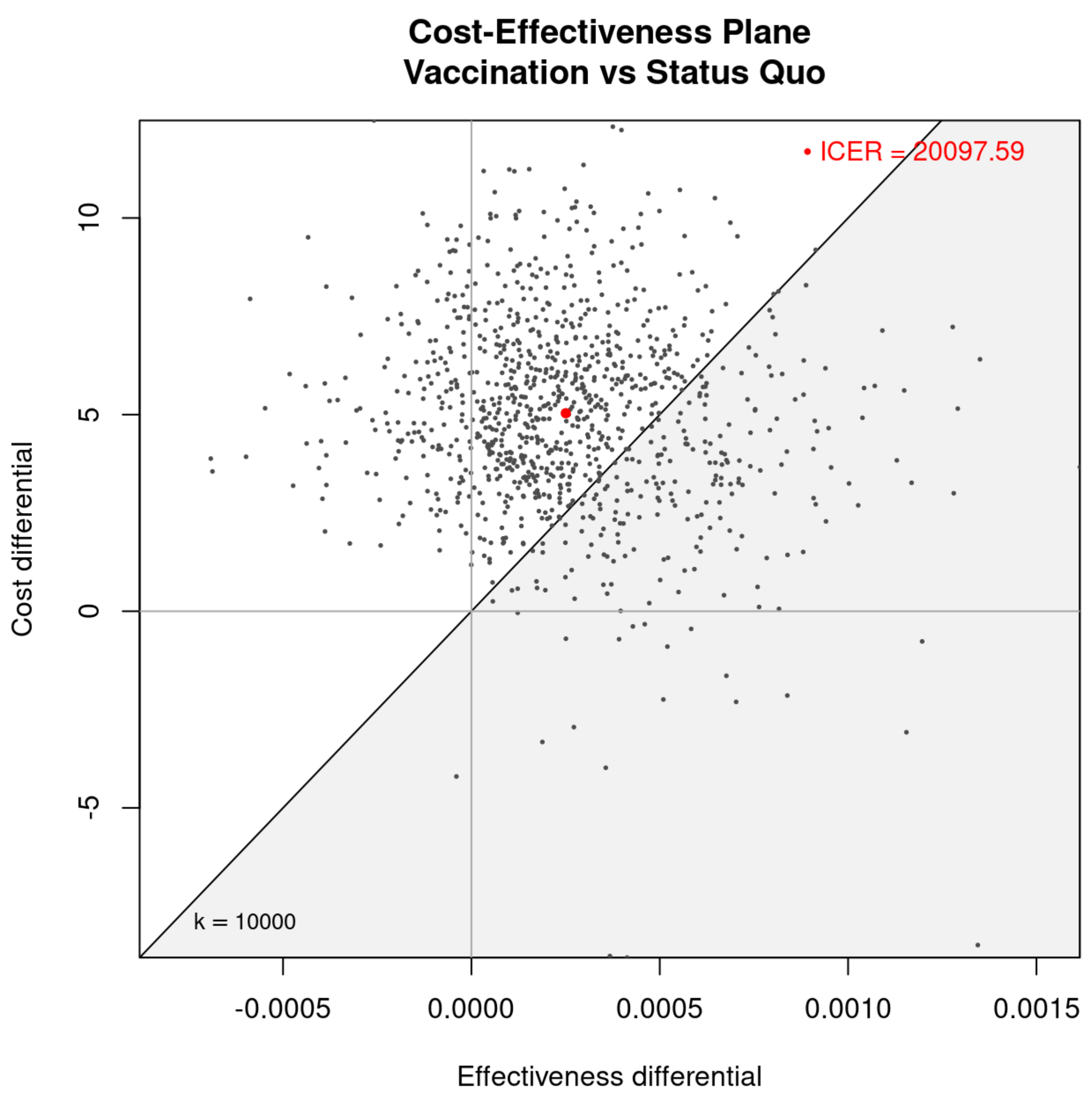
```
k=30000
EIB=k*mean(he$delta.e)-mean(he$delta.c)
EIB
```

```
[1] 2.48131
```

Because for this willingness to pay threshold $\text{EIB} > 0$, then the new treatment $t = 1$ is more cost-effective than the comparator $t = 0$.

Next, we are required to display the cost-effectiveness plane, using the BCEA built-in command `ceplane.plot`. This gives the following result.

```
ceplane.plot(he,wtp=10000)
```

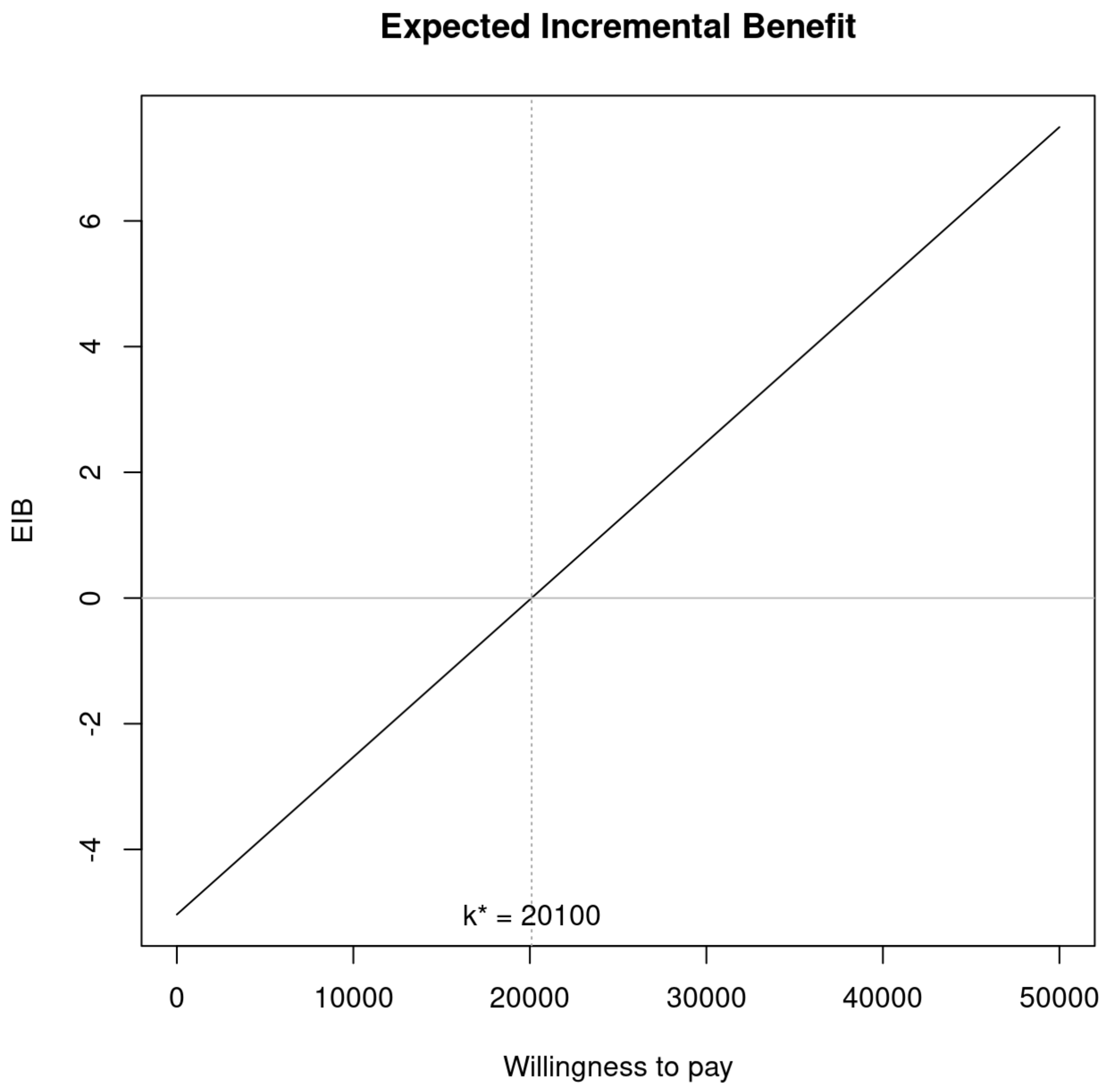


The reason why BCEA says "ICER=NULL" in the top-right corner of the graph is because the object `he` has been modified from the standard BCEA output (effectively, the variable `ICER` is computed within a BCEA object. But because you were required to compute the ICER yourselves, this has been removed and so `ceplane.plot` does not know what value to use. If you are bothered by this, you can simply define in R `he$ICER=ICER` and re-run the `ceplane.plot` command. Now BCEA will print on the graph the value of the ICER.

In any case, because we have selected a different threshold ($k = 10\,000$ instead of $k = 30,000$ as before), this time the ICER (the red dot in the graph) is not included in the "sustainability area" (the grey area in the plane). For this reason, we can conclude that *at this new threshold*, $t = 0$ is more cost-effective.

If we now execute the following command

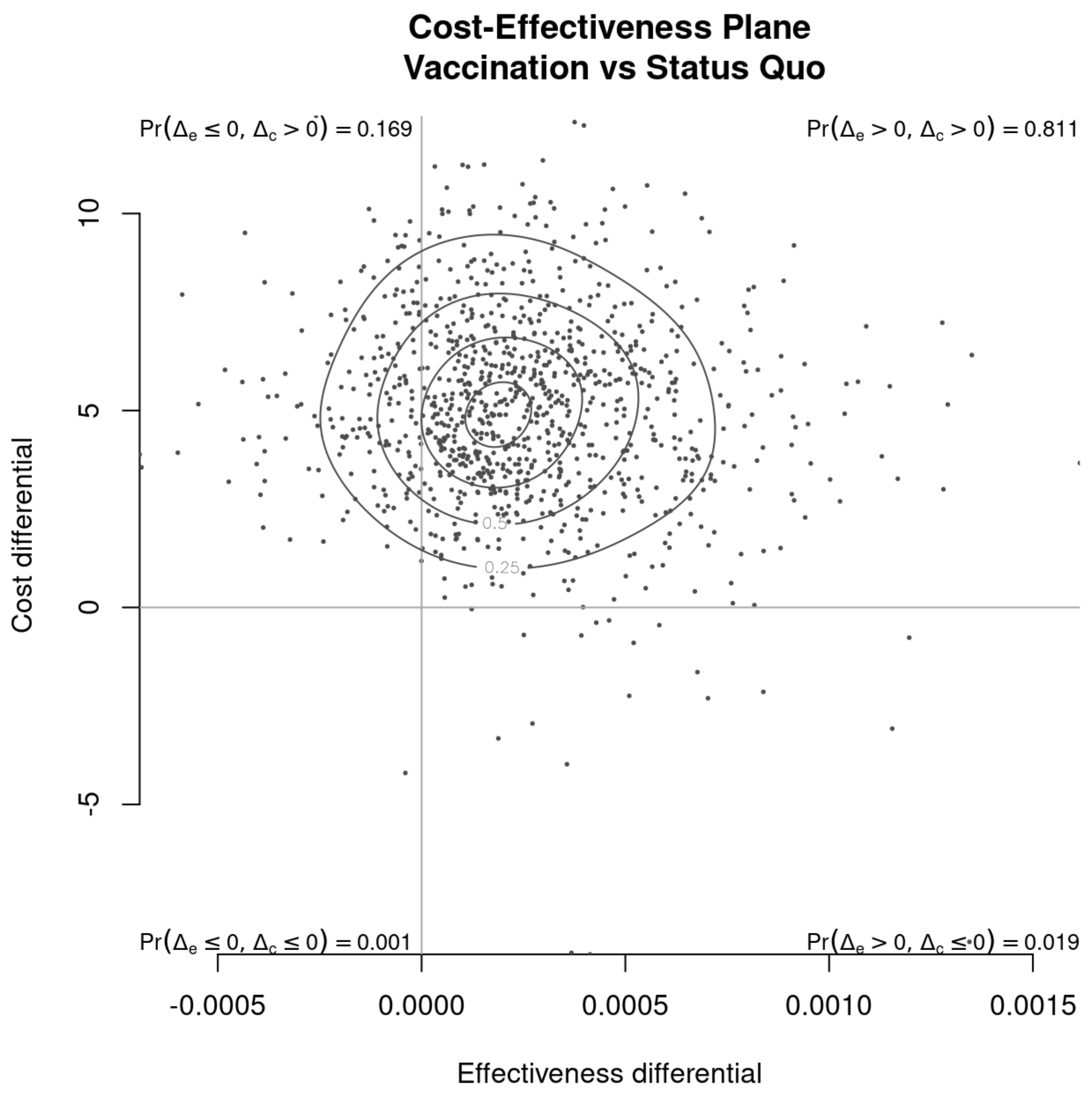
```
eib.plot(he, plot.cri=FALSE)
```



we obtain a plot of the EIB for different values of the willingness to pay. The interpretation of this graph is that for willingness to pay up to about 20100, then $t = 0$ is the most cost-effective treatment (because EIB< 0). After that, $t = 1$ becomes more cost-effective and EIB> 0 indicating that the new treatment has a greater utility than the comparator.

Finally, we can use `BCEA` to show a contour plot of the economic results, by using the following command.

```
contour(he)
```



The probability that the new intervention (vaccination) is dominated by the reference (status quo) is estimated by the proportion if points in the cost-effectiveness plane that lie in the NW quadrant. This is reported by the graph as 0.169.