Making Markov Models Shiny

Robert Smith

21 February 2020

Robert Smith[1] & Paul Schneider[1]

[1]ScHARR, University of Sheffield

Abstract

Health economic models have typically been built in spreadsheet software, mostly Microsoft Excel. However high level programming languages are increasingly being used as model complexity increases and additional requirements such as probabilistic sensitivity analysis (PSA) result in long model run-time in excel . There is a particular push towards using R in health economics (Incerti et al., 2019; Jalal et al., 2017) because it is commonly used by statisticians, flexible and transparent, reusable and adaptable. One of R's strengths is its large number of packages (e.g. bcea, heemod, darthpack) which allow R programmers to leverage the work of others within their code. However as a script based approach R's weakness is that it does not have a user interface. The CRAN package 'shiny', which allows programmers to create Web-browser interfaces which interact with underlying R models, resolves this limitation. This tutorial uses a 4 state markov model (Alarid-Escudero et al., 2020) as a case-study to demonstrate a simple method of creating a shiny application from an existing R model. All code and data are provided on the authors' GitHub page.

Introduction

As the complexity and computational requirements of decision models increase there is growing recognition of the advantages of high level programming languages which support statistical analysis (e.g. R, Python, C++, MATLAB, Java). Of these R, an open-source functional programming language, appears to be ahead of the competition for decision modelling (Jalal et al. 2017), in part because it is free, supported by a large community of statisticians and health economists and has extensive resources including tutorials, guidelines and packages which allow users to build upon the work of others, but also partially because it is the only programming environment accepted by NICE (Check this).

However, despite the many strengths of programming approaches to decision modelling, an important limitation until recently has been the lack of an easily interpretable user interface which is simple to create. Even Microsoft Excel provides the user with a tidy-ish front page to a model in which a decision maker can change a parameter in a specific cell. Incerti et al. (2019) identify web applications as being an essential part of any model in R, stating that they "believe that the future of cost-effectiveness modeling lies in web apps, in which graphical interfaces are used to run script-based models" (p. 577).

R-Shiny is one such web application which allows programmers to create a graphical interface which looks like a website, and allows users to interact with underlying R models (Beely, 2013). Baio & Heath (2017) identify "web applications created using R Shiny"(p.e5) to be the "future of applied statistical modelling, particularly for cost-effectiveness analysis" . Using Shiny, it is possible to create flexible user interfaces which allow users to change parameters, run underlying R code and display results. This has multiple benefits:

It makes script based models that are technically transparent, but only to those who have the ability to technical knowledge to understand them, open for sensitivity analysis to those with no programming knowledge (Jansen et al., 2019) .

It is particularly important for models which are flexible enough to cover multiple regions, organisations or perspectives and as such reporting results of sensitivity analysis would be overwhelming.

In some cases the model inputs will be unknown, or quickly outdated, such that constantly recreating the model would be unfeasible and it is far more efficient to allow the user to input updated parameters, and assuming the same structure, run the updated model.

While it is preferable that models constructed in R are made open-access to improve transparency, replicability and collaboration, it is not a requirement of shiny. Code and data can be kept private, and the application shared internally, or using password protection online through a variety of providers.

We provide a simple example using a previously published 4-state markov model, the Sick-Sicker model which has been described in previous publications (Krijkamp et al., 2020; Alarid-Escudero et al., 2020) and in open source teaching by the DARTH workgroup (Decision Analysis in R for Technologies in Health, 2019). The resulting user interface of this tutorial example is simple, allowing the decision-maker to change only the costs and treatment effect. However, once the reader is confident in creating a shiny app from an existing model, iteratively adapting the application to allow for different/additional inputs and outputs, and meet aesthetic requirements is relatively straight forward.

This R Markdown document has been created to create the code chunks for the accomopanying publication. It can be used as a tutorial to create web applications from script based models created in R.

Methods

This paper provides a tutorial in the use of shiny for health economic modelling. It uses a previously published 4-state markov model, the Sick-Sicker model (Krijkamp et al., 2020; Alarid-Escudero et al., 2020) as a case-study, using the DARTH coding framework when creating the shiny app (Alarid-Escudero et al., 2019). We have adapted it such that it has one purpose for this tutorial, to create PSA outputs.

In this case there are two functions witin the model, the first gen_psa creates a set of psa inputs, the second runs the model for a specific set of PSA inputs.

Functions

Creating PSA inputs.

```r
f_gen_psa <- function(n_sim = 1000, SI_c_Trt){

  df_psa <- data.frame(

    # Transition probabilities (per cycle)
    p_HS1   = rbeta(n_sim, 30, 170),        # probability to become sick when healthy
    p_S1H   = rbeta(n_sim, 60, 60) ,        # probability to become healthy when sick
    p_S1S2  = rbeta(n_sim, 84, 716),        # probability to become sicker when sick

    p_HD    = rbeta(n_sim, 10, 1990)      ,  # probability to die when healthy
    hr_S1   = rlnorm(n_sim, log(3),  0.01),  # rate ratio of death in S1 vs healthy
    hr_S2   = rlnorm(n_sim, log(10), 0.02),  # rate ratio of death in S2 vs healthy

    # Cost vectors with length n_sim
    c_H   = rgamma(n_sim, shape = 100, scale = 20)    , # cost of remaining one cycle in state H
    c_S1  = rgamma(n_sim, shape = 177.8, scale = 22.5), # cost of remaining one cycle in state S1
    c_S2  = rgamma(n_sim, shape = 225, scale = 66.7)  , # cost of remaining one cycle in state S2
    c_D   = 0                                         , # cost of being in the death state
    c_Trt = SI_c_Trt,                                      # cost of treatment (per cycle)
    # Utility vectors with length n_sim
    u_H   = rtruncnorm(n_sim, mean =    1, sd = 0.01, b = 1), # utility when healthy
    u_S1  = rtruncnorm(n_sim, mean = 0.75, sd = 0.02, b = 1), # utility when sick
    u_S2  = rtruncnorm(n_sim, mean = 0.50, sd = 0.03, b = 1), # utility when sicker
    u_D   = 0                                               , # utility when dead
    u_Trt = rtruncnorm(n_sim, mean = 0.95, sd = 0.02, b = 1)  # utility when being treated
  )

  return(df_psa)
}
```

Running the model for a specific set of PSA inputs

```r
f_MM_sicksicker <- function(params) {
  with(as.list(params), {

    # compute internal paramters as a function of external parameter
    r_HD    = - log(1 - p_HD) # rate of death in healthy
    r_S1D   = hr_S1 * r_HD    # rate of death in sick
    r_S2D   = hr_S2 * r_HD      # rate of death in sicker
    p_S1D   = 1 - exp(-r_S1D) # probability to die in sick
    p_S2D   = 1 - exp(-r_S2D) # probability to die in sicker

    v_dwe <- v_dwc <- 1 / (1 + d_r) ^ (0:n_t) # calculate discount weight for each cycle based on disco

    # create transition probability matrix for NO treatment
    m_P <- matrix(0,
                  nrow = n_states, ncol = n_states,
                  dimnames = list(v_n, v_n))
    # fill in the transition probability array
    ### From Healthy
    m_P["H", "H"]  <- 1 - (p_HS1 + p_HD)
    m_P["H", "S1"] <- p_HS1
    m_P["H", "D"]  <- p_HD
    ### From Sick
    m_P["S1", "H"]  <- p_S1H
    m_P["S1", "S1"] <- 1 - (p_S1H + p_S1S2 + p_S1D)
    m_P["S1", "S2"] <- p_S1S2
    m_P["S1", "D"]  <- p_S1D
    ### From Sicker
    m_P["S2", "S2"] <- 1 - p_S2D
    m_P["S2", "D"]  <- p_S2D
    ### From Dead
    m_P["D", "D"] <- 1

    m_TR <- matrix(NA, nrow = n_t + 1 , ncol = n_states,
                   dimnames = list(0:n_t, v_n))     # create Markov trace (n_t + 1 because R doesn't un

    m_TR[1, ] <- c(1, 0, 0, 0)                      # initialize Markov trace

    ############# PROCESS ###########################################
    for (t in 1:n_t){                                 # throughout the number of cycles
      m_TR[t + 1, ] <- m_TR[t, ] %*% m_P             # estimate the Markov trace for cycle the next cycle
    }

    ############ OUTPUT  ###########################################
    # create vectors of utility and costs for each state
    v_u_trt    <- c(u_H, u_Trt, u_S2, u_D)
    v_u_no_trt <- c(u_H, u_S1, u_S2, u_D)

    v_c_trt    <- c(c_H, c_S1 + c_Trt, c_S2 + c_Trt, c_D)
    v_c_no_trt <- c(c_H, c_S1, c_S2, c_D)

    # estimate mean QALys and costs
    v_E_no_trt <- m_TR %*% v_u_no_trt
    v_E_trt    <- m_TR %*% v_u_trt

    v_C_no_trt <- m_TR %*% v_c_no_trt
    v_C_trt    <- m_TR %*% v_c_trt

    ### discount costs and QALYs
    te_no_trt <- t(v_E_no_trt) %*% v_dwe  # 1x31 %*% 31x1 -> 1x1
    te_trt    <- t(v_E_trt) %*% v_dwe

    tc_no_trt <- t(v_C_no_trt) %*% v_dwc
    tc_trt    <- t(v_C_trt)    %*% v_dwc

    results <- c("Cost_NoTrt" = tc_no_trt,
                 "Cost_Trt"   = tc_trt,
                 "QALY_NoTrt" = te_no_trt,
                 "QALY_Trt"   = te_trt,
                 "ICER"       = (tc_trt - tc_no_trt)/(te_trt - te_no_trt))

    return(results)
  }
  )
}
```

Creating a Wrapper

When using a web application it is likely that the user will want to be able to change parameter inputs and re-run the model. In order to make this simple, we recommend wrapping the entire PSA process into a function. We call this function *wrapper* and use *f_* to denote that this is a function.

The wrapper function has as its inputs all the things which we may wish to vary using shiny. We set the default values to those of the base model in any report/publication. The moedl then generates PSA inputs using the *f_gen_psa* function, creates a table of results, and finally loops through the PSA, running the model with each set of PSA inputs (a row from df_psa) in turn. The function then returns the results (the costs and qalys for treatment and no treatment for each PSA run).

```
f_wrapper <- function(

  #================================================================
  #                         Shiny inputs
```

```r
#==================================================================
n_age_init = 25,    # age at baseline default is 25
n_age_max  = 110,   # maximum age of follow up default is 110
d_r        = 0.035,  # discount rate for costs & QALYS (NICE 3.5%)
n_sim      = 1000,   # number of simulations default 1000
c_Trt      = 50 # cost of treatment deault 50

){



#==================================================================
#                          Non-shiny inputs
#==================================================================
n_t <- n_age_max - n_age_init # time horizon, number of cycles
v_n <- c("H", "S1", "S2", "D") # the 4 health states of the model:
n_states <- length(v_n) # number of health states


#==================================================================
#                          Create PSA Inputs
#==================================================================

df_psa <- f_gen_psa(n_sim = n_sim, c_Trt)


#==================================================================
#                          RUN PSA
#==================================================================

# Initialize matrix of results outcomes
df_out <- matrix(NaN,
               nrow = n_sim,
               ncol = 5,
               dimnames = list(1:n_sim,c("Cost_NoTrt", "Cost_Trt",
                                          "QALY_NoTrt", "QALY_Trt",
                                          "ICER")))
# loop through psa inputs running the model for each.
for(i in 1:n_sim){
  df_out[i,] <- f_MM_sicksicker(df_psa[i, ])
  cat('\r', paste(round(i/n_sim * 100), "% done", sep = " "))       # display the progress of the sim
}

df_out <- as.data.frame(df_out) # convert matrix to dataframe

return(df_out) # output the dataframe from the function

}
```

Integrating into Shiny

Set-up

The set-up is relatively simple, load the shiny package from the library so that we can use the shiny functions, the source the file to load the wrapper function (which includes all model functionality).

```
library(shiny)             # we need the function shiny installed, this loads it from the library.
source("../R/wrapper.R")   # sources the wrapper function, within wrapper need to be all functions nece
```

User Interface

The user interface is extremely flexible, we show the code for a very simple structure (fluidpage) with a sidebar containing inputs and a main panel containing outputs. We have done very little formatting in order to minimize the quantity of code while maintaining all functionality. In order to get an aesthetically pleasing application we would have much more sophisticated formatting, relying on css, HTML and javascript.

This example user interface below is made up of two components, a titlepanel and a sidebar layout. The sidebarLayout function has within it a sidebar and a main panel. These are all contained within the *fluidpage* function which creates the ui.

The title panel contains the title "Sick Sicker Model in Shiny", the sidebar panel contains two numeric inputs and a slider input ("Treatment Cost","PSA runs","initial age") and an Action Button ("Run / update model").

The values of the inputs have ids which are used by the server function, we denote these with an SI to indicate they are Shiny Inputs ("SI_c_Trt","SI_n_sim","SI_n_age_init").

The action button also has an id, this is not an input into the model wrapper (f_wrapper) so we leave out the SI and call it "run_model".

The main panel contains two objects which have been output from the server: tableOutput("icer_table") is a table of results, and plotOutput("CE_plane") is a cost-effectiveness plane plot. It is important that the format (e.g. tableOutput) matches the format of the object from the server "icer_table". The two h3() functions are simply headings which appear as "Results Table" and "Cost-effectiveness Plane".

```
ui <- fluidPage(

  titlePanel("Sick Sicker Model in Shiny"),

  # SIDEBAR
  sidebarLayout(

    sidebarPanel(

    numericInput(inputId = "SI_c_Trt",
                 label = "Treatment Cost",
                 value = 200,
                 min = 0,
                 max = 400),

    numericInput(inputId = "SI_n_sim",
                 label = "PSA runs",
                 value = 1000,
                 min = 0,
                 max = 400),

    sliderInput(inputId = "SI_n_age_init",
                label = "initial age",
                min = 10,
                max = 80,
                value = 25),


    actionButton("run_model","Run / update model")

                ),  # close sidebarPanel

  mainPanel(

    h3("Results Table"),

    tableOutput("icer_table"),

    h3("Cost-effectiveness Plane"),

    plotOutput("CE_plane")

            ) # close mainpanel

      ) # close sidebarlayout

  ) # close UI fluidpage
```

Server

The server is marginally more complicated than the user interface. It is created by a function with inputs and outputs.

The observe event indicates that when the action button (run_model) is pressed the code within the curly brackets is run.

The first thing that happens is that the model wrapper function *f_wrapper* is used, with the inputs from the numeric inputs and sliders in the user interface ("SI_c_Trt","SI_n_age_init","SI_n_sim"). The *input$* prefix indicates that the objects have come from the user interface. The results of the model are stored as the dataframe object *df_model_res*.

The ICER table is then created and output (note the prefix *output$*) in the object *icer_table*. See previous section on the user interface and note that the tableOutput was reliant on "icer_table". The function renderTable rerenders the table continuously so that table always reflects the values from the data-frame of results created above. In this simple example we have created a table of results using code within the *renderTable* function. In reality we would generally use a function which creates a publication quality table which is aesthetically pleasing. There are numerous packages which provide this functionality.

The Cost-effectiveness plane is created in a similar process, using the *renderPlot* function to continuously update a plot which is created using baseR plot function using icers calcualted from the results dataframe *df_model_res*. For aesthetic purposes we recommend this is replaced by a ggplot plot which has much improved functionality.

```
server <- function(input, output){

 observeEvent(input$run_model,
              ignoreNULL = F, {


  df_model_res = f_wrapper(c_Trt = input$SI_c_Trt,
                           n_age_init = input$SI_n_age_init,
                           n_sim = input$SI_n_sim)


    # ICER TABLE
    output$icer_table <- renderTable({

      df_res_table <- data.frame(

        Option =  c("Treatment","No Treatment"),

        QALYs  =  c(mean(df_model_res$QALY_Trt),mean(df_model_res$QALY_NoTrt)),

        Costs  =  c(mean(df_model_res$Cost_Trt),mean(df_model_res$Cost_NoTrt)),

        Inc.QALYs = c(mean(df_model_res$QALY_Trt) - mean(df_model_res$QALY_NoTrt),NA),

        Inc.Costs = c(mean(df_model_res$Cost_Trt) - mean(df_model_res$Cost_NoTrt),NA),

        ICER = c(mean(df_model_res$ICER),NA)
      )

      df_res_table[,2:6] <- round(df_res_table[,2:6],digits = 2)

      df_res_table

      }) # table plot end.


   #  CE PLANE
    output$CE_plane <- renderPlot({

      #model_res = model_res() # calling reactive function
      df_model_res$inc_C <- df_model_res$Cost_Trt - df_model_res$Cost_NoTrt
      df_model_res$inc_Q <- df_model_res$QALY_Trt - df_model_res$QALY_NoTrt

      # create cost effectiveness plane plot
      plot(x = df_model_res$inc_Q,
           y = df_model_res$inc_C,
           xlab = "Incremental QALYs",
           ylab = "Incremental Costs",
           xlim = c(min(df_model_res$inc_Q,df_model_res$inc_Q*-1),
                    max(df_model_res$inc_Q,df_model_res$inc_Q*-1)),
           ylim = c(min(df_model_res$inc_C,df_model_res$inc_C*-1),
                    max(df_model_res$inc_C,df_model_res$inc_C*-1)),
           abline(h = 0,v=0)
```

```
          ) # plot end
    }) # renderplot end

 }) # Observe Event End


  } # Server end
```

Running the app

The app can be run within the R file using the function *shinyApp* which depends on the *ui* and *server* which have been created and described above. Running this creates a shiny application in the local environment (e.g. your desktop). In order to deploy the application onto the web the app needs to be *published* using the publish button in the top right corner of the R-file in RStudio (next to run-app).

```
## ----- run app------

shinyApp(ui, server)
```

Discussion

With the movement to make economic models more transparent and reproducible gaining traction, the shift to the use of script based models written in programming languages seems inevitable. This move will be gradual, but will nevertheless require upskilling of health economists used to working in excel through short courses, and the introduction of new courses at universities. It is our opinion that these new courses should include some instruction on the creation of user interfaces and web applications for script based economic models. Since the most predominant script based programming environment in health economics is currently R, we recommend including a tutorial in R-Shiny within these courses.

As demonstrated in this tutorial, creating a web application for an economic model created in R programming environment is relatively straightforward. The authors' experience of creating these web apps has led us to the conclusion that the most efficient method is to work iteratively, first ensuring that the model is working as intended before making small incremental changes to the UI and server one item at a time. While experienced programmers can make substantial time savings by combining multiple steps we have found that the time taken to correct mistakes far outweighs the time savings associated with combining steps.

From our experience in working with stakeholders from a variety of sectors, there is still a concern about the process of deploying code and data to an external server. While providers such as ShinyIO provide assurances of SSR encryption and user authentication clients with particularly sensitive data may still have concerns. This problem can be avoided in two ways: firstly if clients have their own server and the ability to deploy applications they can maintain control of all data and code, and secondly the application could simply not be deployed, and instead simply created during a meeting using code and data shared in a zip file.

The movement towards script based health economic models with web based user interfaces is particularly useful in situations where a general model structure has been created with a variety of stakeholders in mind, each of which may have different input parameters and wish to conduct sensitivity analysis specific to their decision. For example the World Health Organisation Department of Sexual and Reproductive Health and Research recently embedded a shiny application into their website. The application runs a heemod model in R in an external server, and allows users to select their country and change country specific input parameters, run the model and display results. The process of engagement, the ability to 'play' with the model and test the extremes of the decision makers' assumptions gives stakeholders more control over models, making them feel less like black boxes, and provides some engagement with the process. While there is a danger that a mis-informed stakeholder may make a mistake in their choice of parameter, we should remember that the role of the model is to inform decision-makers not instruct them … and besides: it is simple to limit the range that parameter inputs can take.

Conclusion

The aim of this tutorial was to provide a useful reference for those hoping to create a user interface for a health economic model created in R. It is our hope that more health economic models will be created open source, and open access so that other economists can critique, learn from and adapt these models. The creation of user interfaces for these apps should improve transparency further, allowing stakeholders and third parties to conduct their own sensitivity analysis. The future is bright, maybe even shiny.