Probabilistic Sensitivity Analysis: Generation

Fernando Alarid-Escudero, Greg Knowlton, Eva Enns, and the DARTH Team

2021-05-30

Overview

This vignette explains how to use `dampack` to generate your own PSA using only a decision analytic model and information about the distributions that define your model parameters of interest. If both costs and effects are calculated within the decision model, the resulting `psa` object will be compatible with all of `dampack`'s PSA analysis functions, which are explained at length in the `psa_analysis` vignette (type `vignette("psa_generation", package = "dampack")` in the console to view this vignette).

Decision Model Format

In order to generate a PSA in `dampack`, the user must input the code for the decision analytic model in a standardized format that is compatible with the `run_psa` function. This is the same format required for the `FUN` argument of `run_owsa_det` and `run_twsa_det`.

The user-defined model function must accept a single input containing a list of the parameters from the `params_basecase` argument. In the example model shown below, this list is named `l_params`, and the variables contained in this list are the only variables that are allowed to be varied through the `params_range` argument in the DSA. Optionally, additional function inputs for `FUN` can be supplied through the `...` argument of `run_owsa_det`/`run_twsa_det`, but these inputs are not allowed to vary in the sensitivity analysis. These additional inputs must be arguments of `FUN`, like `n_wtp` in the example of `calculate_ce_out()` below. `FUN` and its component functions are also able to incorporate variables stored in the global environment, such as `n_age_init` or `n_age_max` in the example.

The user-defined model function must return a data.frame where the first column contains a character vector of the strategy names, and the subsequent columns contain numeric vectors of all relevant model outcomes. Each row of the data.frame will consist of a strategy name followed by the corresponding outcome values for that strategy. These model outcomes must be calculated internally within `FUN`. The model outcomes are not limited to typical outcomes like cost or effectiveness and can be any numerical outcome that the user chooses to model.

```r
library(dampack)
run_sick_sicker_model <- function(l_params, verbose = FALSE) {
  with(as.list(l_params), {
    # l_params must include:
    # -- disease progression parameters (annual): r_HD, p_S1S2, hr_S1D, hr_S2D,
    # -- initial cohort distribution: v_s_init
    # -- vector of annual state utilities: v_state_utility = c(u_H, u_S1, u_S2, u_D)
    # -- vector of annual state costs: v_state_cost = c(c_H, c_S1, c_S2, c_D)
    # -- time horizon (in annual cycles): n_cyles
    # -- annual discount rate: r_disc

    ####### SET INTERNAL PARAMETERS #########################################
```

```r
# state names
v_names_states <- c("H", "S1", "S2", "D")
n_states <- length(v_names_states)

# vector of discount weights
v_dw  <- 1 / ((1 + r_disc) ^ (0:n_cycles))

# state rewards
v_state_cost <- c("H" = c_H, "S1" = c_S1, "S2" = c_S2, "D" = c_D)
v_state_utility <- c("H" = u_H, "S1" = u_S1, "S2" = u_S2, "D" = u_D)

# transition probability values
r_S1D <- hr_S1D * r_HD   # rate of death in sick state
r_S2D <- hr_S2D * r_HD   # rate of death in sicker state
p_S1D <- 1 - exp(-r_S1D) # probability of dying when sick
p_S2D <- 1 - exp(-r_S2D) # probability of dying when sicker
p_HD  <- 1 - exp(-r_HD)   # probability of dying when healthy

## Initialize transition probability matrix
# all transitions to a non-death state are assumed to be conditional on survival
m_P <- matrix(0,
              nrow = n_states, ncol = n_states,
              dimnames = list(v_names_states, v_names_states)) # define row and column names
## Fill in matrix
# From H
m_P["H", "H"]   <- (1 - p_HD) * (1 - p_HS1)
m_P["H", "S1"]  <- (1 - p_HD) * p_HS1
m_P["H", "D"]   <- p_HD
# From S1
m_P["S1", "H"]  <- (1 - p_S1D) * p_S1H
m_P["S1", "S1"] <- (1 - p_S1D) * (1 - (p_S1H + p_S1S2))
m_P["S1", "S2"] <- (1 - p_S1D) * p_S1S2
m_P["S1", "D"]  <- p_S1D
# From S2
m_P["S2", "S2"] <- 1 - p_S2D
m_P["S2", "D"]  <- p_S2D
# From D
m_P["D", "D"]   <- 1

# check that all transition matrix entries are between 0 and 1
if(!all(m_P <= 1 & m_P >= 0)){
  stop("This is not a valid transition matrix (entries are not between 0 and 1")
} else
  # check transition matrix rows add up to 1
  if (!all.equal(as.numeric(rowSums(m_P)),rep(1,n_states))){
    stop("This is not a valid transition matrix (rows do not sum to 1)")
  }

####### INITIALIZATION ##########################################
# create the cohort trace
m_Trace <- matrix(NA, nrow = n_cycles + 1 ,
              ncol = n_states,
              dimnames = list(0:n_cycles, v_names_states)) # create Markov trace

# create vectors of costs and QALYs
```

```r
    v_C <- v_Q <- numeric(length = n_cycles + 1)

    ############# PROCESS ###########################################

    m_Trace[1, ] <- v_s_init # initialize Markov trace
    v_C[1] <- 0 # no upfront costs
    v_Q[1] <- 0 # no upfront QALYs

    for (t in 1:n_cycles){ # throughout the number of cycles
      m_Trace[t + 1, ] <- m_Trace[t, ] %*% m_P # calculate trace for cycle (t + 1) based on cycle
         t

      v_C[t + 1] <- m_Trace[t + 1, ] %*% v_state_cost

      v_Q[t + 1] <- m_Trace[t + 1, ] %*% v_state_utility

    }

    #############  PRIMARY ECONOMIC OUTPUTS  #########################

    # Total discounted costs
    n_tot_cost <- t(v_C) %*% v_dw

    # Total discounted QALYs
    n_tot_qaly <- t(v_Q) %*% v_dw

    #############  OTHER OUTPUTS   ###################################

    # Total discounted life-years (sometimes used instead of QALYs)
    n_tot_ly <- t(m_Trace %*% c(1, 1, 1, 0)) %*% v_dw

    ####### RETURN OUTPUT  ###########################################
    out <- list(m_Trace = m_Trace,
                m_P = m_P,
                l_params,
                n_tot_cost = n_tot_cost,
                n_tot_qaly = n_tot_qaly,
                n_tot_ly = n_tot_ly)

    return(out)
  }
  )
}


simulate_strategies <- function(l_params, wtp = 100000){
    # l_params must include:
    # -- *** Model parameters ***
    # -- disease progression parameters (annual): r_HD, p_S1S2, hr_S1D, hr_S2D,
    # -- initial cohort distribution: v_s_init
    # -- vector of annual state utilities: v_state_utility = c(u_H, u_S1, u_S2, u_D)
    # -- vector of annual state costs: v_state_cost = c(c_H, c_S1, c_S2, c_D)
    # -- time horizon (in annual cycles): n_cyles
    # -- annual discount rate: r_disc
    # -- *** Strategy specific parameters ***
  # -- treartment costs (applied to Sick and Sicker states): c_trtA, c_trtB
  # -- utility with Treatment_A (for Sick state only): u_trtA
  # -- hazard ratio of progression with Treatment_B: hr_S1S1_trtB

with(as.list(l_params), {

  ####### SET INTERNAL PARAMETERS #########################################
  # Strategy names
  v_names_strat <- c("No_Treatment", "Treatment_A", "Treatment_B")
  # Number of strategies
  n_strat <- length(v_names_strat)

  ## Treatment_A
  # utility impacts
  u_S1_trtA <- u_trtA
  # include treatment costs
  c_S1_trtA <- c_S1 + c_trtA
  c_S2_trtA <- c_S2 + c_trtA

  ## Treatment_B
  # progression impacts
  r_S1S2_trtB <- -log(1 - p_S1S2) * hr_S1S2_trtB
  p_S1S2_trtB <- 1 - exp(-r_S1S2_trtB)
  # include treatment costs
  c_S1_trtB <- c_S1 + c_trtB
  c_S2_trtB <- c_S2 + c_trtB



  ####### INITIALIZATION ##########################################
  # Create cost-effectiveness results data frame
  df_ce <- data.frame(Strategy = v_names_strat,
                      Cost = numeric(n_strat),
                      QALY = numeric(n_strat),
                      LY = numeric(n_strat),
                      stringsAsFactors = FALSE)


  ######### PROCESS ##############################################
  for (i in 1:n_strat){
    l_params_markov <- list(n_cycles = n_cycles, r_disc = r_disc, v_s_init = v_s_init,
                            c_H =  c_H, c_S1 = c_S2, c_S2 = c_S1, c_D = c_D,
                            u_H =  u_H, u_S1 = u_S2, u_S2 = u_S1, u_D = u_D,
                            r_HD = r_HD, hr_S1D = hr_S1D, hr_S2D = hr_S2D,
                            p_HS1 = p_HS1, p_S1H = p_S1H, p_S1S2 = p_S1S2)

    if (v_names_strat[i] == "Treatment_A"){
      l_params_markov$u_S1 <- u_S1_trtA
      l_params_markov$c_S1 <- c_S1_trtA
      l_params_markov$c_S2 <- c_S2_trtA

    } else if(v_names_strat[i] == "Treatment_B"){
      l_params_markov$p_S1S2 <- p_S1S2_trtB
      l_params_markov$c_S1   <- c_S1_trtB
      l_params_markov$c_S2   <- c_S2_trtB
      }

      l_result <- run_sick_sicker_model(l_params_markov)

      df_ce[i, c("Cost", "QALY", "LY")] <- c(l_result$n_tot_cost,
                                             l_result$n_tot_qaly,
                                             l_result$n_tot_ly)
      df_ce[i, "NMB"] <- l_result$n_tot_qaly * wtp - l_result$n_tot_cost
    }

    return(df_ce)
  })
}
```

Generating Parameter Samples for a PSA

The `gen_psa_samp` function creates a `data.frame` of parameter value samples based on the underlying distributions specified by the user. Each row of the returned `data.frame` is an independently sampled set of the parameters varied in the PSA. To produce a `psa` object, the `run_psa` function will take each row of this `data.frame` and calculate the outcomes for each strategy in the user-defined model. The `data.frame` returned by `gen_psa_samp` matches the format required by the `parameters` argument of the `make_psa_obj` function.

`gen_psa_samp` has five arguments: `params` is a vector containing the names of each parameter to be varied in the PSA; `dists` is a vector of the same length indicating which type of distribution this parameter will be drawn from; `parameterization_types` is a vector indicating the format of how these parameter distributions are defined; `dists_params` is a list of vectors, where each element of the list contains the values necessary to define the distribution for a parameter based upon its corresponding element of `dists` and `parameterization_types`; and finally, `nsamp` is a numeric value indicating the number of PSA samples to be generated.

Details about the allowable distributions, their parameterization types and the corresponding formats for `dists_params` can be found in the help documentation by typing `?gen_psa_samp` in the console. Within the example below, the first parameter in the PSA, `"p_HS1"`, follows a `"beta"` distribution, which has an `"a, b"` parameterization type (which stands for alpha, beta), and the two values for alpha and beta are `30` and `170`, respectively. If the user does not possess estimates for the alpha and beta parameters for the beta distribution but does have estimates for the mean and standard deviation of `"p_HS1"`, they also could choose to parameterize this distribution using `parameterization_types = "mean, sd"`. In this case, the first element of the dists_params list would need to be a numeric vector of length 2 containing the estimated mean and standard deviation for `"p_HS1"`. `dampack` would then use a method-of-moments estimator to calculate an alpha and beta parameter for this distribution from which the PSA sample values are drawn.

```
my_params <- c(#Transition probabilities
               "p_HS1",
               "p_S1H",
               "p_S1S2",
               #Hazard ratios
               "hr_S1",
               "hr_S2",
               "hr_S1S2_trtB",
               #Costs
               "c_H",
               "c_S1",
               "c_S2",
```

```r
                 "c_trtA",
                 "c_trtB",
                 #Utilities
                 "u_H",
                 "u_S1",
                 "u_S2",
                 "u_TrtA")

my_dists <- c(#Transition probabilities
              "beta",
              "beta",
              "beta",
              #Hazard ratios
              "log-normal",
              "log-normal",
              "log-normal",
              #Costs
              "gamma",
              "gamma",
              "gamma",
              "gamma",
              "gamma",
              #Utilities
              "truncated-normal",
              "truncated-normal",
              "truncated-normal",
              "truncated-normal")

my_parameterization_types <- c(#Transition Probabilities
                               "a, b",
                               "a, b",
                               "a, b",
                               #Hazard ratios
                               "mean, sd",
                               "mean, sd",
                               "mean, sd",
                               #Costs
                               "shape, scale",
                               "shape, scale",
                               "shape, scale",
                               "shape, scale",
                               "shape, scale",
                               #Utilities
                               "mean, sd, ll, ul",
                               "mean, sd, ll, ul",
                               "mean, sd, ll, ul",
                               "mean, sd, ll, ul")

my_dists_params <- list(#Transition Probabilities
                        c(7.5, 42.5),
                        c(12, 12),
                        c(15, 133),
                        #Hazard ratios
                        c(3, 0.5),
                        c(10, 0.5),
                                    c(0.6, .01),
                                    #Costs
                                    c(44.5, 45),
                                    c(178, 22.5),
                                    c(900, 16.67),
                                    c(576, 21),
                                    c(676, 19),
                                    #Utilities
                                    c(1, 0.01, NA, 1),
                                    c(0.75, 0.02, NA, 1),
                                    c(0.5, 0.03, NA, 1),
                                    c(0.95, 0.02, NA, 1))

my_psa_params <- gen_psa_samp(params = my_params,
                              dists = my_dists,
                              parameterization_types = my_parameterization_types,
                              dists_params = my_dists_params,
                              n = 100)
```

Generating Outcomes for the PSA

The `run_psa` function is used to calculate outcomes for each strategy for every PSA sample through the user-defined decision model, `FUN`. In this example, the `data.frame` of PSA parameters generated by `gen_psa_samp` should be used as the input for the `psa_samp` argument. The combination of the parameters in the `psa_samp` and the `params_basecase` argument must define every parameter that `FUN` expects within its `l_params` input argument. Other parameters for `FUN` that are not contained within `l_params` list, like the `n_wtp` argument of `calculate_ce_out` can be passed through `...` as an additional argument in `run_psa`. If the decision model in `FUN` is computationally slow and/or the number of PSA samples is extremely large, `run_psa` could take a long time to run. Under these circumstances, it is recommended that you set the `progress` argument to `TRUE` in order to print a progress bar in the console while the function is running to monitor its progress.

```
my_params_basecase <- list(p_HS1 = 0.15,
                           p_S1H = 0.5,
                           p_S1S2 = 0.105,
                           r_HD = 0.002,
                           hr_S1D = 3,
                           hr_S2D = 10,
                           hr_S1S2_trtB = 0.6,
                           c_H = 2000,
                           c_S1 = 4000,
                           c_S2 = 15000,
                           c_D = 0,
                           c_trtA = 12000,
                           c_trtB = 13000,
                           u_H = 1,
                           u_S1 = 0.75,
                           u_S2 = 0.5,
                           u_D = 0,
                           u_trtA = 0.95,
                           n_cycles = 75,
                           v_s_init = c(1, 0, 0, 0),
                           r_disc = 0.03)
```

```
psa_output <- run_psa(psa_samp = my_psa_params,
                      params_basecase = my_params_basecase,
                      FUN = simulate_strategies,
                      outcomes = c("Cost", "QALY", "LY", "NMB"),
                      strategies = c("No_Treatment", "Treatment_A", "Treatment_B"),
                      progress = FALSE)
```

Creating a Fully-functional PSA Object

`run_psa` will return a named list containing a `psa` object for each outcome specified in the `outcomes` argument. Each `psa` object in the list is compatible with `owsa()`, `twsa()`, and their associated downstream functions described in the `psa_analysis` vignette. However, most PSA analysis functions in `dampack` rely on the clear designation of both a cost and effectiveness outcome. To create a PSA object that is compatible with these functions related to cost-effectiveness you must input the results of `run_psa` into the `make_psa_obj` function in the following manner. `make_psa_obj()` requires data.frames for `cost`, `effect`, and `parameters`, and a character vector for `strategies`. The `data.frame`s containing each outcome in the list returned by `run_psa` are stored within `other_outcome`. In this example, the outcome associated with `effect` is named `"Effect"`, and so `psa_output$Effect$other_outcome` is supplied to the corresponding argument of `make_psa_obj`.

```
cea_psa <- make_psa_obj(cost = psa_output$Cost$other_outcome,
                        effect = psa_output$QALY$other_outcome,
                        parameters = psa_output$Cost$parameters,
                        strategies = psa_output$Cost$strategies,
                        currency = "$")
```