darthpack: An R package with DARTH's decision modeling coding framework

Supplementary Material to "A need for change! A coding framework for improving transparency in decision modeling"

DARTH

2019-07-26

Contents

The Sick-Sicker model

In this case-study, we perform a cost-effectiveness analysis (CEA) using a previously published 4-state model called the Sick-Sicker model (Enns et al., 2015). In the Sick-Sicker model, a hypothetical disease affects individuals with an average age of 25 years and results in increased mortality, increased treatment costs and reduced quality of life (QoL). We simulate this hypothetical cohort of 25-year-old individuals over a lifetime (i.e., reaching an age of 100 years old) using 75 annual cycles, represented with `n_t`. The cohort starts in the "Healthy" health state (denoted "H"). Healthy individuals are at risk of developing the illness, at which point they would transition to the first stage of the disease (the "Sick" health state, denoted "S1"). Sick individuals are at risk of further progressing to a more severe stage (the "Sicker" health state, denoted "S2"), which is a constant probability in this case-study. There is a chance that individuals in the Sick state eventually recover and return back to the Healthy state. However, once an individual reaches the Sicker state, they cannot recover; that is, the probability of transitioning to the Sick or Healthy states from the Sicker state is zero. Individuals in the Healthy state face background mortality that is age-specific (i.e., time-dependent). Sick and Sicker individuals face an increased mortality expressed as a hazard rate ratio (HR) of 3 and 10, respectively, on the background mortality rate. Sick and Sicker individuals also experience increased health care costs and reduced QoL compared to healthy individuals. Once simulated individuals die, they transition to the "Dead" state (denoted "D"), where they remain. Figure 1 shows the state-transition diagram of the Sick-Sicker model. The evolution of the cohort is simulated in one-year discrete-time cycles. Both costs and quality-adjusted life years (QALYs) are discounted at an annual rate of 0.03%.

Two alternative strategies exist for this hypothetical disease: a no-treatment and a treatment strategy. Under the treatment strategy, Sick and Sicker individuals receive treatment and continue doing so until they recover or die. The cost of the treatment is additional to the cost of being Sick or Sicker for one year. The treatment improves QoL for those individuals who are Sick but has no effect on the QoL of those who are sicker. To evaluate these two alternative strategies, we perform a CEA.

We assume that most of the parameters of the Sick-Sicker model and their uncertainty have been previously estimated and are known to the analyst. However, while we can identify those who are afflicted with the illness through obvious symptoms, we can not easily distinguish those in the Sick state from the those in the Sicker state. Thus, we can not directly estimate state-specific mortality hazard rate ratios, nor do we know the transition probability of progressing from Sick to Sicker. Therefore, we calibrate the model to different epidemiological data. We internally validated the calibrated model by comparing the predicted outputs from the model evaluated at the calibrated parameters against the calibration targets (Eddy et al., 2012, Goldhaber-Fiebert et al. (2010)).

As part of the CEA, we conducted different deterministic sensitivity analysis (SA), including one-way and

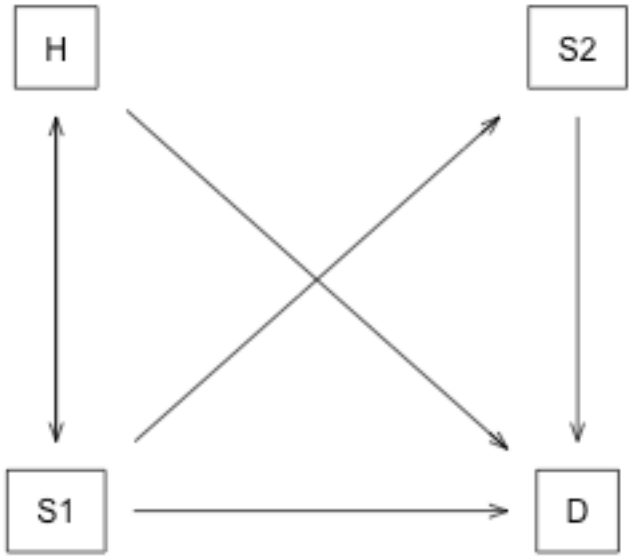

Figure 1: State-transition diagram of the Sick-Sicker model. Healthy individuals can get Sick, die or stay healthy. Sick individuals can recover, transitioning back to healthy, can die, or stay sick. Once individuals are Sicker, they stay Sicker until they die.

two-way SA, and tornado plots. To quantify the effect of parameter uncertainty on decision uncertainty, we conducted a probabilistic sensitivity analysis (PSA) and reported our uncertainty analysis results with a cost-effectiveness acceptability curve (CEAC), cost-effectiveness acceptability frontier (CEAF) and expected loss curves (ELC) (Alarid-Escudero et al., 2019). We also conducted a value of information (VOI) analysis to determine whether potential future research is needed to reduce parameter uncertainty. All steps of the CEA will be described using the different components of the framework.

Set-up

This report is a supplementary material meant to guide you through the `R` code of a fully functional decision model to showcase the framework described by the Decision Analysis in R for Technologies in Health (DARTH) workgroup in the manuscript *A need for change! A coding framework for improving transparency in decision modeling*. The code of this analysis can be downloaded from GitHub (https://github.com/DARTH-git/Decision-Modeling-Framework). We recommend downloading the case-study files as a single .zip file containing all directories. Unzip the folder and save to your desired directory. The framework is divided into different directories, described in Table 1, that could be accessed from the RStudio project *Decision-Modeling-Framework.Rproj*. In this framework, you will find multiple directories as described in Table 1 of the main manuscript. We refer to the directory names of this framework and scripts stored in these directories using *italic* style. This report is created with Markdown and is located in the *reports* directory of the framework. The figures for the case-study can be found in the *figs* directory, data required to conduct some of the analyses of the different components are in the *data* directory and the `R` scripts with functions, are located in the *R* directory. The main `R` scripts that conduct the analyses of the different components of the framework are stored in the `R` directory. In this document we do not show all the `R` code we refer to. Therefore, it is important to follow along while reading this document.

Chapter 1

Define model inputs

As described in the main manuscript, in this first component we declare all model input variables and set their values. The `R` script running the analysis of this component is the *01_model-inputs.R* file in the `analysis` directory.

The input to inform the values is divided in three categories: external, estimated, and calibrated. The majority of the Sick-Sicker model parameters are informed by external data. Only three parameter values need to be estimated using model calibration.

In this component, we start with the general setup of the model, specifying among others the time horizon, name and number of health states, proportion of the cohort in each of the different health states at the start of the simulation and discount rates. The next step is to specify the external parameters. The initial model parameter values and `R` variable names are presented in Table 1.1.

Table 1.1: Description of the initial parameters with their `R` name and value of the Sick-Sicker model.

Parameter	**R name**	**Value**
Time horizon (n_t)	`n_t`	75 years
Names of health states (n)	`v_n`	H, S1, S2, D
Annual discount rate (costs/QALYs)	`d_c`/`d_e`	3%
Annual transition probabilities		
- Disease onset (H to S1)	`p_HS1`	0.15
- Recovery (S1 to H)	`p_S1H`	0.5
- Disease progression (S1 to S2) in the time-homogenous model	`p_S1S2`	0.105
Annual mortality		
- All-cause mortality (H to D)	`p_HD`	age-specific
- Hazard rate ratio of death in S1 vs H	`hr_S1`	3
- Hazard rate ratio of death in S2 vs H	`hr_S2`	3
Annual costs		
- Healthy individuals	`c_H`	$2,000
- Sick individuals in S1	`c_S1`	$4,000

Parameter	R name	Value
- Sick individuals in S2	`c_S2`	$15,000
- Dead individuals	`c_D`	$0
- Additional costs of sick individuals treated in S1 or S2	`c_Trt`	$12,000
Utility weights		
- Healthy individuals	`u_H`	1.00
- Sick individuals in S1	`u_S1`	0.75
- Sick individuals in S2	`u_S2`	0.50
- Dead individuals	`u_D`	0.00
Intervention effect		
- Utility for treated individuals in S1	`u_Trt`	0.95

Age-specific background mortality for healthy individuals is represented by the US population in 2015 and obtained from the Human Mortality database. This information is stored in the *01_all-cause-mortality.csv* file in the *data* directory. Based on this .csv file a vector with mortality rates by age is created using the `load_mort_data` function in the *01_model-inputs_functions.R* script. This function gives us the flexibility to easily import data from other countries or years.

```
print.function(load_mort_data) # print the function
```

```
## function (file = NULL)
## {
##     if (!is.null(file)) {
##         df_r_mort_by_age <- read.csv(file = file)
##     }
##     else {
##         df_r_mort_by_age <- all_cause_mortality
##     }
##     v_r_mort_by_age <- as.matrix(dplyr::select(df_r_mort_by_age,
##         Total))
##     return(v_r_mort_by_age)
## }
## <bytecode: 0x00000000151d5890>
## <environment: namespace:darthpack>
```

Another function in the *01_model-inputs_functions.R* script, is the `load_all_parms` function. This function, which is actually using the `load_mort_data` function, loads all parameters for the decision model from multiple sources and creates a list that contains all parameters and their values.

```
print.function(load_all_params)  # print the function
```

```
## function (file.init = NULL, file.mort = NULL)
```

```
## {
##     if (!is.null(file.init)) {
##         df_params_init <- read.csv(file = file)
##     }
##     else {
##         df_params_init <- df_params_init
##     }
##     v_r_mort_by_age <- load_mort_data(file = file.mort)
##     l_params_all <- with(as.list(df_params_init), {
##         v_names_str <- c("No Treatment", "Treatment")
##         n_str <- length(v_names_str)
##         v_age_names <- n_age_init:(n_age_init + n_t - 1)
##         v_n <- c("H", "S1", "S2", "D")
##         n_states <- length(v_n)
##         v_s_init <- c(H = 1, S1 = 0, S2 = 0, D = 0)
##         l_params_all <- list(v_names_str = v_names_str, n_str = n_str,
##             n_age_init = n_age_init, n_t = n_t, v_age_names = v_age_names,
##             v_n = v_n, n_states = n_states, v_s_init = c(H = 1,
##                 S1 = 0, S2 = 0, D = 0), v_r_mort_by_age = v_r_mort_by_age)
##         return(l_params_all)
##     })
##     l_params_all <- c(l_params_all, df_params_init)
## }
## <bytecode: 0x00000000151d6b38>
## <environment: namespace:darthpack>
```

The `load_all_params` function is informed by the arguments `file.init` and `file.mort`. The `file.init` argument is a string with the location and name of the file with initial set of parameters. The initial parameter values for our case-study are stored in the *01_init-params.csv* file located in the *data* directory. The `load_all_params` function read this .csv file into the function environment as a dataframe called, `df_params_init`.

The `file.mort` argument is a string with the location and name of the file with mortality data. As described before, in our case-study this is the *01_all-cause-mortality.csv* file. Within the `load_all_parms` function, the `load_mort_data` function is used to create a vector with mortality rates from the .csv data.

After loading all the information, the `load_all_params` generates a list called,`l_params_all`, including all parameters for the model including the general setup parameters and the vector of mortality rates. The function also stores the dataframe `df_params_init` with the initial set of parameters in the list. This is all executed in the in the *01_model-inputs.R* script by running the code below.

```
l_params_all <- load_all_params()
```

For the Sick-Sicker model we do not have to estimate parameters, but we do have three parameters that need to be estimated via model calibration. In this stage of the framework, we simply set these parameters

to valid "dummy" values that are compatible with the next phase of the analysis, model implementation, but are ultimately just placeholder values until we conduct the calibration phase. This means that these values will be replaced by the best-fitted calibrated values after we performed the calibration in component 3.

Using a function to create a list of base-case parameters to have all model parameters in a single object is very useful, because this object will have to be updated for the calibration and the different sensitivity analyses in components 3 and 5 of the framework, respectively. Below, we guide you through the components of the function.

Chapter 2

Decision model

In this second component, we build the backbone of the decision analysis: the implementation of the model. This component is performed by the *02_simulation_model.R* script. This file itself is not very large. It simply loads some packages and sources the input from component 01 and in addition it runs the `decision_model` function that is used to caputre the dynamic process of the Sick-Sicker example and stores the output. The output of the model is the traditional cohort trace. The trace describes how the cohort is distributed among the different health states over time and is plotted at the end of this script.

The function `decision_model` is defined in the *02_simulation_model_functions.R* file in the `R` folder. As described in the paper, constructing a model as a function at this stage facilitates subsequent stages of the model development and analysis. This since these processes will all call the same model function, but pass different parameter values and/or calculate different final outcomes based on the model outputs. In the next part, we will describe the code within the function.

```
print.function(decision_model) # print the code of the function
```

```
## function (l_params_all, err_stop = FALSE, verbose = FALSE)
## {
##     with(as.list(l_params_all), {
##         if ((n_t + n_age_init) > nrow(v_r_mort_by_age)) {
##             stop("Not all the age in the age range have a corresponding mortality rate")
##         }
##         if ((sum(v_s_init) != 1) | !all(v_s_init >= 0)) {
##             stop("vector of initial states (v_s_init) is not valid")
##         }
##         p_HDage <- 1 - exp(-v_r_mort_by_age[(n_age_init + 1) +
##             0:(n_t - 1)])
##         p_S1Dage <- 1 - exp(-v_r_mort_by_age[(n_age_init + 1) +
##             0:(n_t - 1)] * hr_S1)
##         p_S2Dage <- 1 - exp(-v_r_mort_by_age[(n_age_init + 1) +
##             0:(n_t - 1)] * hr_S2)
```

```
##         a_P <- array(0, dim = c(n_states, n_states, n_t), dimnames = list(v_n,
##             v_n, 0:(n_t - 1)))
##         a_P["H", "H", ] <- (1 - p_HDage) * (1 - p_HS1)
##         a_P["H", "S1", ] <- (1 - p_HDage) * p_HS1
##         a_P["H", "D", ] <- p_HDage
##         a_P["S1", "H", ] <- (1 - p_S1Dage) * p_S1H
##         a_P["S1", "S1", ] <- (1 - p_S1Dage) * (1 - (p_S1S2 +
##             p_S1H))
##         a_P["S1", "S2", ] <- (1 - p_S1Dage) * p_S1S2
##         a_P["S1", "D", ] <- p_S1Dage
##         a_P["S2", "S2", ] <- 1 - p_S2Dage
##         a_P["S2", "D", ] <- p_S2Dage
##         a_P["D", "D", ] <- 1
##         check_transition_probability(a_P, err_stop = err_stop,
##             verbose = verbose)
##         check_sum_of_transition_array(a_P, n_states, n_t, err_stop = err_stop,
##             verbose = verbose)
##         m_M <- matrix(0, nrow = (n_t + 1), ncol = n_states, dimnames = list(0:n_t,
##             v_n))
##         m_M[1, ] <- v_s_init
##         for (t in 1:n_t) {
##             m_M[t + 1, ] <- m_M[t, ] %*% a_P[, , t]
##         }
##         return(list(a_P = a_P, m_M = m_M))
##     })
## }
## <bytecode: 0x000000001833d000>
## <environment: namespace:darthpack>
```

The `decision_model` function is informed by the argument `l_params_all`. Via this argument we give the function a list with all parameters of the decision model. For the Sick-Sicker model, these parameters are stored in the list `l_params_all`, which we passed into the function as shown below.

```
l_out_stm <- decision_model(l_params_all = l_params_all) # run the function
```

This function itself has all the mathematical equations of the decision models coded inside. It starts by calculating the age-specific transition probabilities from all non-dead states based on the vector of age-specific mortality rates `v_r_mort_by_age`. These parameters will become vectors of length `n_t`, describing the probability to die for all ages from all non-dead states.

The next part of the function, creates an array that stores age-specific transition probability matrices in each of the third dimension. The transition probability matrix is a core component of a state-transition cohort model (Iskandar, 2018). This matrix contains the probabilities of transitioning from the current health state, indicated by the rows, to the other health states, specified by the columns. Since we have age-specific

transition probabilities, the transition probability matrix is different each cycle. These probabilities are only depending on the age of the cohort, and not on other events; therefore, we can generate all matrices at the start of the model. This results in `n_t` different age-specific matrices that are stored in an array, called `a_P`, of dimensions `n_states` x `n_states` x `n_t`. After initializing the array, it is filled with the transition probability stored in the list. When running the model, we can index the correct transition probability matrix corresponding with the current age of the cohort. We then added some sanity checks to make sure that the transition matrices and the transition probabilities are valid. The first three and last cycles of the transition probability matrices stored in the array `a.P` are shown below.

```
l_out_stm$a_P[, , 1:3] # show the first three time-points of a_P
```

```
## , , 0
##
##            H        S1        S2           D
## H  0.8491385 0.1498480 0.0000000 0.001013486
## S1 0.4984813 0.3938002 0.1046811 0.003037378
## S2 0.0000000 0.0000000 0.9899112 0.010088764
## D  0.0000000 0.0000000 0.0000000 1.000000000
##
## , , 1
##
##            H        S1        S2            D
## H  0.8491513 0.1498502 0.0000000 0.0009985012
## S1 0.4985037 0.3938180 0.1046858 0.0029925135
## S2 0.0000000 0.0000000 0.9900597 0.0099402657
## D  0.0000000 0.0000000 0.0000000 1.0000000000
##
## , , 2
##
##            H        S1        S2           D
## H  0.8490910 0.1498396 0.0000000 0.001069428
## S1 0.4983976 0.3937341 0.1046635 0.003204853
## S2 0.0000000 0.0000000 0.9893570 0.010642959
## D  0.0000000 0.0000000 0.0000000 1.000000000
```

```
l_out_stm$a_P[, , l_params_all$n_t] # show it for the last cycle
```

```
##            H        S1         S2         D
## H  0.6055199 0.1068564 0.00000000 0.2876237
## S1 0.1807584 0.1427991 0.03795926 0.6384833
## S2 0.0000000 0.0000000 0.03365849 0.9663415
## D  0.0000000 0.0000000 0.00000000 1.0000000
```

By comparing these probability matrices, we observe an increase in the probabilities of transitioning to death from all health states.

After the array is filled, the cohort trace matrix, `m_M`, of dimensions `n_t` x `n_states` is initialized. This matrix will store the state occupation at each point in time. The first row of the matrix is informed by the initial state vector `v_s_init`. For the remaining points in time, we iteratively multiply the cohort trace with the age-specific transition probability matrix corresponding to the specific cycle obtained by indexing the array `a_P` appropriately. All the outputs and relevant elements of the decision model are stored in a list, called `l_out_stm`. This list contains the array of the transition probability matrix for all cycles `t` and the cohort trace `m_M`.

```
head(l_out_stm$m_M)    # show the top part of the cohort trace
```

```
##           H        S1         S2           D
## 0 1.0000000 0.0000000 0.00000000 0.000000000
## 1 0.8491385 0.1498480 0.00000000 0.001013486
## 2 0.7957468 0.1862564 0.01568695 0.002309774
## 3 0.7684912 0.1925699 0.03501425 0.003924648
## 4 0.7484793 0.1909659 0.05478971 0.005765035
## 5 0.7306193 0.1873106 0.07413838 0.007931783
```

```
tail(l_out_stm$m_M)    # show the bottom part of the cohort trace
```

```
##              H           S1           S2         D
## 70 0.009928317 0.0022433565 2.035951e-04 0.9876247
## 71 0.007153415 0.0015935925 1.311619e-04 0.9911218
## 72 0.005058845 0.0011174473 8.674540e-05 0.9937370
## 73 0.003460336 0.0007552206 5.436484e-05 0.9957301
## 74 0.002289594 0.0004937081 3.298632e-05 0.9971837
## 75 0.001475636 0.0003151589 1.985106e-05 0.9981894
```

Using the code below, we can graphically show the model dynamics by plotting the cohort trace. Figure 2.1 shows the distribution of the cohort among the different health states at each time point.

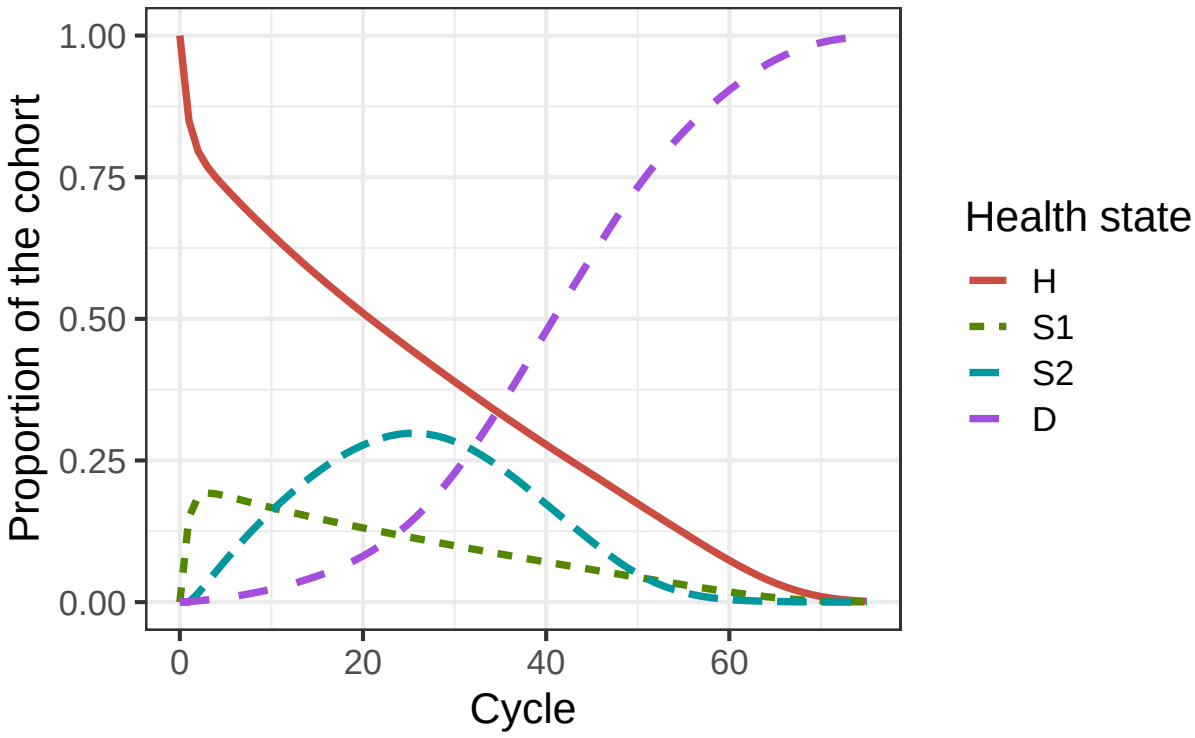

Figure 2.1: Cohort trace of the Sick-Sicker cohort model

Chapter 3

Model calibration

In this third component, we calibrate unknown model parameters by matching model outputs to specified calibration targets. Specifically, we calibrate the Sick-Sicker model to match survival, prevalence and the proportion who are Sicker, among all those afflicted (Sick+Sicker). We used a Bayesian calibration approach using the incremental mixture importance sampling (IMIS) algorithm (Steele et al., 2006), which has been used to calibrate health policy models (Raftery and Bao, 2010, Menzies et al. (2017), Rutter et al. (2018)). Bayesian methods allow us to quantify the uncertainty in the calibrated parameters even in the presence of non-identifiability (Alarid-Escudero et al., 2018). This analysis is coded in the *03_calibration.R* file in the `analysis` folder. The target data is stored in the *03_calibration_targets.RData* file. Similar to component 02 2, in the section *03.1 Load packages*, we start by loading inputs and functions. In addition, we load the calibration targets data into the R workspace. In the next section, *03.2 Visualize targets*, we plot each of the calibration targets with their confidence intervals.

In section *03.3 Run calibration algorithms*, we set the parameters we need to calibrate to fixed values and test if the function `calibration_out` that produces model outputs corresponding to the calibration targets works. This function takes a vector of parameters that need to be calibrated and a list with all parameters of decision model and computes model outputs to be used for calibration routines.

```
print.function(calibration_out) # print the functions
```

```
## function (v_params_calib, l_params_all)
## {
##     l_params_all <- update_param_list(l_params_all = l_params_all,
##         params_updated = v_params_calib)
##     l_out_stm <- decision_model(l_params_all = l_params_all)
##     v_os <- 1 - l_out_stm$m_M[, "D"]
##     v_prev <- rowSums(l_out_stm$m_M[, c("S1", "S2")])/v_os
##     v_prop_S2 <- l_out_stm$m_M[, "S2"]/rowSums(l_out_stm$m_M[,
##         c("S1", "S2")])
##     l_out <- list(Surv = v_os[c(11, 21, 31)], Prev = v_prev[c(11,
##         21, 31)], PropSicker = v_prop_S2[c(11, 21, 31)])
```

Table 3.1: Summary statistics of the posterior distribution

	Mean	2.5%	50%	97.5%	Mode	MAP
p_S1S2	0.1076687	0.0964559	0.1073187	0.1196543	0.1067639	0.107841
hr_S1	2.7261594	1.1010910	2.6790446	4.4020812	2.2857230	2.297053
hr_S2	9.6232710	7.3427361	9.6461532	11.9533402	9.4629634	9.886776

```
##     return(l_out)
## }
## <bytecode: 0x00000000191a62a8>
## <environment: namespace:darthpack>
```

This function is informed by two argument `v_params_calib` and `l_params_all`. The vector `v_params_calib` contains the values of the three parameters of interest. The list `l_params_all` contains all parameters of the decision model. The placeholder values are replaced by `v_params_calib` and with these values the model is evaluated. Model evaluation takes place by running the `decision_model` function, described in component 02. The result in a new list with output of the model corresponding to the parameter values in the `v_params_calib`. With this new decision model output, the overall survival, disease prevalence and the proportion of Sicker in the Sick and Sicker states are calculated. The estimated values for these epidemiological outcomes at different timepoints are combined in a list called `l_out` produced but the `calibration_out`.

Once we make sure this code works, we specify the calibration parameters in section *03.3.1 Specify calibration parameters*. These include setting the seed for the random number generation, specifying the number of random samples to obtain from the calibrated posterior distribution, the name of the input parameters and the range of these parameters that will inform the prior distributions of the calibrated parameters, and the name of the calibration targets: `Surv`, `Prev`, `PropSick`.

In the next section, *03.3.2 Run IMIS algorithm*, we calibrate the Sick-Sicker model with the IMIS algorithm. For this case-study, we assume a normal likelihood and uniform priors. For a more detailed description of IMIS for Bayesian calibration, different likelihood functions and prior distributions, we refer the reader to the tutorial for Bayesian calibration by Menzies et al. (Menzies et al., 2017). We use the `IMIS` function from the `IMIS` package that calls the functions `likelihood`, `sample.prior` and `prior`, to draw samples from the posterior distribution (Raftery and Le Bao, 2012). The functions are specified in the *03_calibration_functions.R* file in the `R` folder. For the `IMIS` function, we specify the incremental sample size at each iteration of IMIS, the desired posterior sample size at the resample stage, the maximum number of iterations in IMIS and the number of optimizers which could be 0. The function returns a list, which we call `l_fit_imis`, with the posterior samples, the diagnostic statistics at each IMIS iteration and the centers of Gaussian components (Raftery and Le Bao, 2012). We store the posterior samples in the matrix `m_calib_post`.

We then explore these posterior distributions in section *03.4 Exploring posterior distribution*. We start by estimating the posterior mean, median and 95% credible interval, the mode and the maximum-a-posteriori (MAP). All for these summary statistics are combined in a dataframe called `df_posterior_summ`. Table 3.1 shows the summary statistics of the posterior distribution.

In section *03.4.2 Visualization of posterior distribution*, we generate a pairwise scatter plot of the calibrated parameters (Figure 3.2) and a 3D scatter plot of the joint posterior distribution (Figure 3.1). These figures

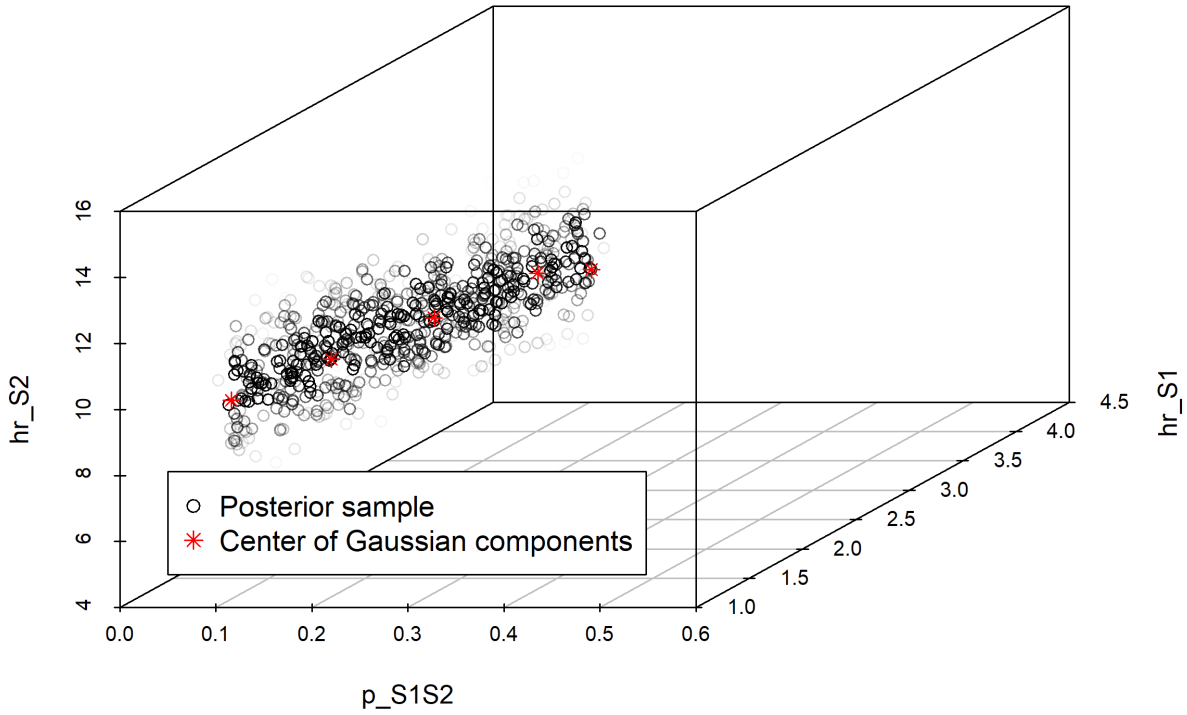

Figure 3.1: Joint posterior distribution

are saved in the *figs* directory.

Finally, the posterior distribution and MAP estimate from the IMIS calibration are stored in the file *03_imis_output.RData*. Storing this data as an .Rdata file allows to import the data in following sections without needing to re-run the calibration component.

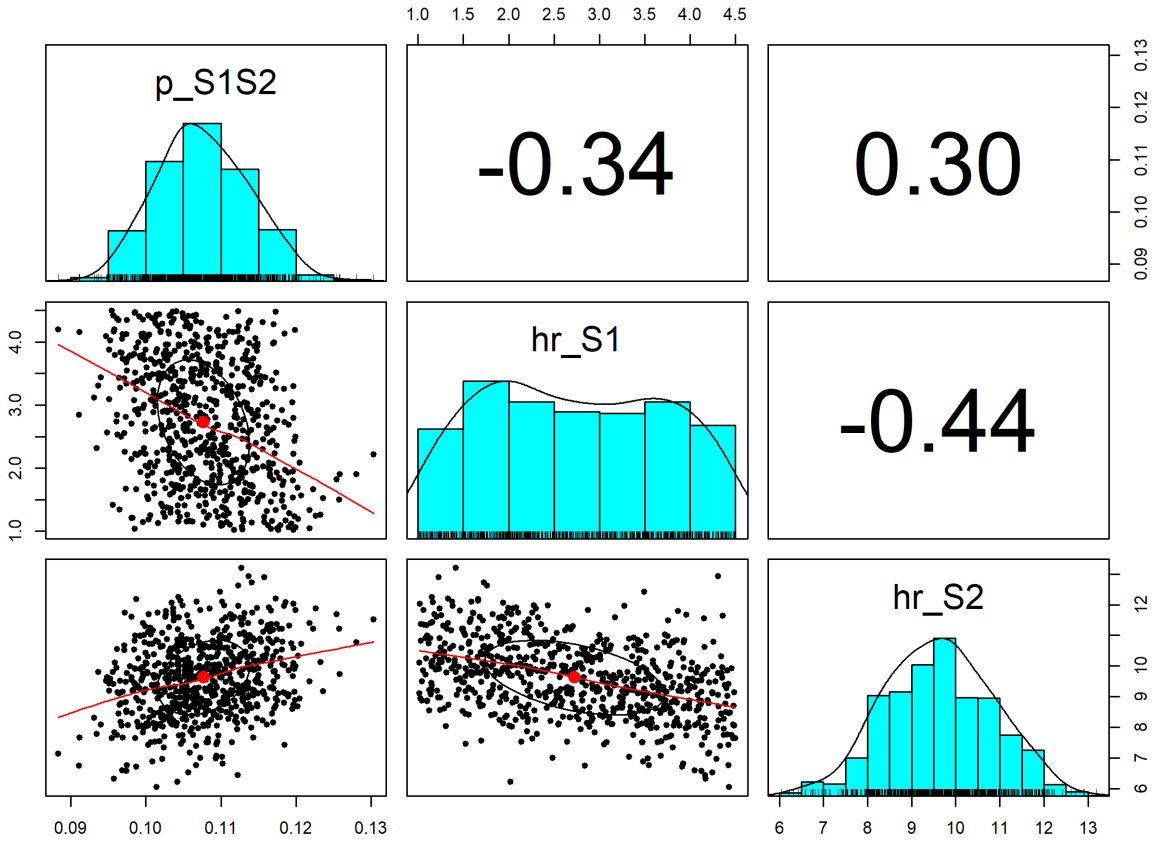

Figure 3.2: Pairwise posterior distribution of calibrated parameters

Chapter 4

Validation

In this forth component, we check the internal validity of our Sick-Sicker model before we move on to the analysis components. To internally validate the Sick-Sicker model, we compare the model-predicted output evaluated at posterior parameters against the calibration targets. This is all done in the *04_validation.R* script in the `analysis` folder.

In section *04.2 Compute model-predicted outputs*, we compute the model-predicted outputs for each sample of posterior distribution as well as for the MAP estimate. We then use the function `data_summary` to summarize the model-predicted posterior outputs into different summary statistics.

```
print.function(data_summary)
```

```
## function (data, varname, groupnames)
## {
##     summary_func <- function(x, col) {
##         c(mean = mean(x[[col]], na.rm = TRUE), median = quantile(x[[col]],
##             probs = 0.5, names = FALSE), sd = sd(x[[col]], na.rm = TRUE),
##             lb = quantile(x[[col]], probs = 0.025, names = FALSE),
##             ub = quantile(x[[col]], probs = 0.975, names = FALSE))
##     }
##     data_sum <- plyr::ddply(data, groupnames, .fun = summary_func,
##         varname)
##     data_sum <- plyr::rename(data_sum, c(mean = varname))
##     return(data_sum)
## }
## <bytecode: 0x0000000018b900f8>
## <environment: namespace:darthpack>
```

This function is informed by three arguments, `data`, `varname` and `groupnames`.

The computation of the model-predicted outputs using the MAP estimate is done by inserting the `v_calib_post_map` data into the previously described `calibration_out` function. This function creates a

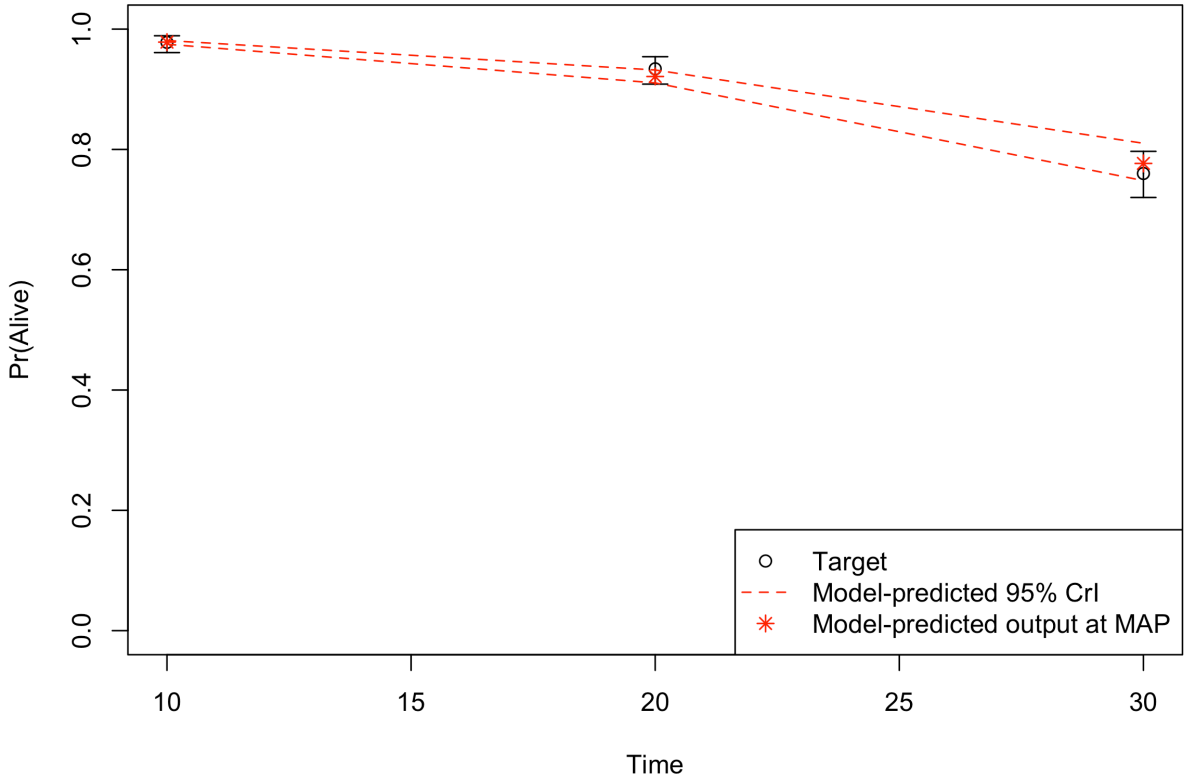

Figure 4.1: Survival data: Model-predicted outputs vs targets.

list including the estimated values for survival, prevalence and the proportion of sicker individuals at cycles 10, 20 and 30.

In sections *04.6 Internal validation: Model-predicted outputs vs. targets*, we check the internal validation by plotting the model-predicted outputs against the calibration targets (Figures 4.1-4.3). The generated plots are saved as .png files in the *fig* folder. These files can be used in reports without the need of re-running the code.

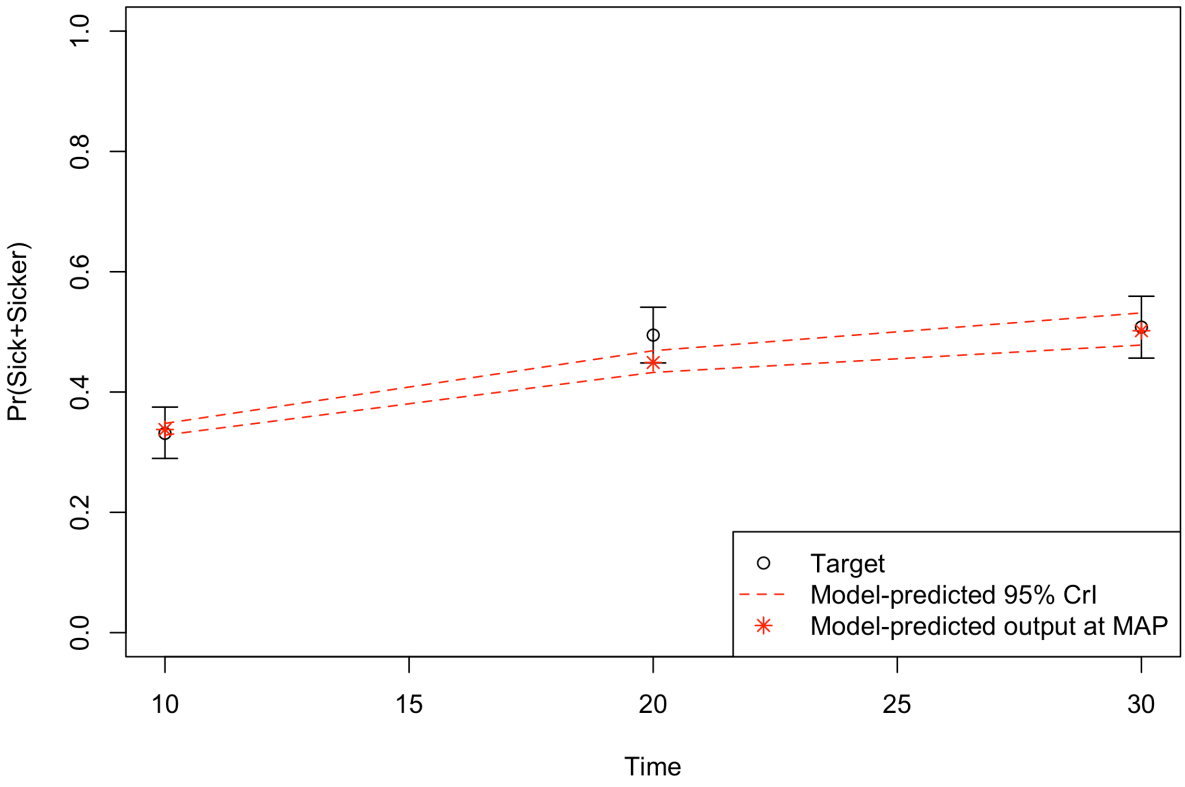

Figure 4.2: Prevalence data of sick individuals: Model-predicted output vs targets.

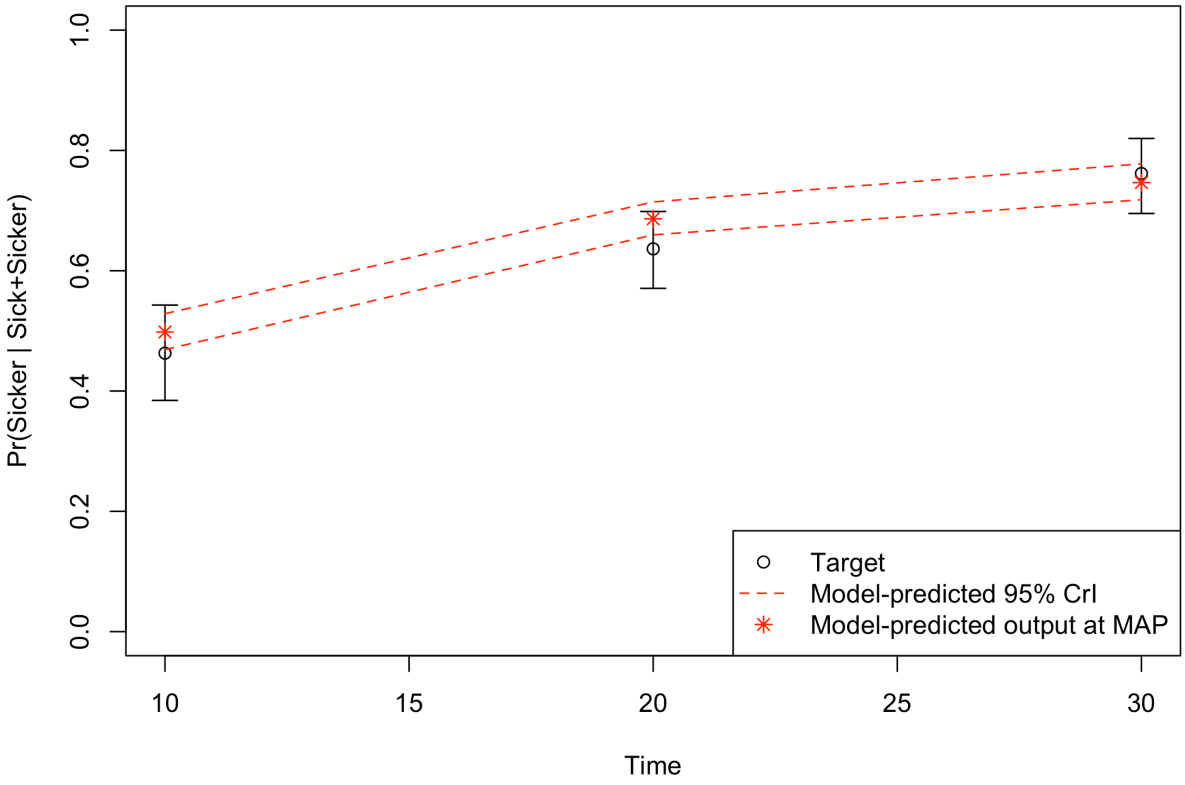

Figure 4.3: Proportion who are Sicker, among all those afflicted (Sick + Sicker): Model-predicted output.

Chapter 5

Analysis

The analysis component is where the elements in components 1-4 are combined to answer the question(s) of interest given current information and to quantify the value of potential further research. Our framework separates the analysis in three subcomponents: *05a Deterministic analysis*, *05b Uncertainty analysis* and *05c Value of information analysis*. For the Sick-Sicker case-study, we use all three subcomponents to conduct the CEA and to quantify the uncertainty of our decision. For procedures in the CEA, we rely on the `R` package `dampack`, which is available here: https://github.com/DARTH-git/dampack. Instructions for installing `dampack` are described in Appendix 0 provided in the *app0_packages_setup.R* script of the *analysis* folder.

5.1 05a Deterministic analysis

In this subcomponent, we perform a deterministic CEA, followed by some deterministic sensitivity analysis, including one-way, two-way and tornado sensitivity analyses. The function script of this subcomponent, *05a_deterministic_analysis_function.R*, contains the function `calculate_ce_out`. This function calculates costs and effects for a given vector of parameters using a simulation model. We need to run our simulation model using the calibrated parameter values, but the list we created in component 01 1 still contain the placeholder values for the calibrated parameters. This means we need to update these values by the calibrated values stored in the vector `v_calib_post_map`. The function `update_param_list` updates the list of parameters with new values for some specific parameters.

```
print.function(update_param_list)
```

```
## function (l_params_all, params_updated)
## {
##     if (typeof(params_updated) != "list") {
##         params_updated <- split(unname(params_updated), names(params_updated))
##     }
##     l_params_all <- modifyList(l_params_all, params_updated)
##     return(l_params_all)
```

```
## }
## <bytecode: 0x000000001e401730>
## <environment: namespace:darthpack>
```

The first argument of the function, called `l_params_all`, is a list with all the parameters of decision model. The second argument, `params_updated`, is an object with parameters for which values need to be updated. The function returns the list `l_params_all` with updated values.

In the *05a_deterministic_analysis.R* script we execute the `update_param_list` function for our case-study, resulting in the list `l_params_basecase` where the placeholder values for `p_S1S2`, `hr_S1` and `hr_S2` are replaced by the calibration estimates.

```
l_params_basecase <- update_param_list(l_params_all, v_calib_post_map)
```

We use this new list as an argument in the `calculate_ce_out` function. In addition, we specify the willingness-to-pay (WTP) threshold value using the `n_wtp` argument of this function. This WTP value is used to compute a net monetary benefit (NMB) value. If the user does not specify the WTP, a default value of $100,000/QALY will be used by the function.

```
df_out_ce <- calculate_ce_out(l_params_all = l_params_basecase,
                              n_wtp = 150000)
print.function(calculate_ce_out) # print the function
```

```
## function (l_params_all = load_all_params(), n_wtp = 1e+05)
## {
##     with(as.list(l_params_all), {
##         v_dwc <- 1/((1 + d_e)^(0:(n_t)))
##         v_dwe <- 1/((1 + d_c)^(0:(n_t)))
##         l_model_out_no_trt <- decision_model(l_params_all = l_params_all)
##         l_model_out_trt <- decision_model(l_params_all = l_params_all)
##         m_M_no_trt <- l_model_out_no_trt$m_M
##         m_M_trt <- l_model_out_trt$m_M
##         v_u_no_trt <- c(u_H, u_S1, u_S2, u_D)
##         v_u_trt <- c(u_H, u_Trt, u_S2, u_D)
##         v_c_no_trt <- c(c_H, c_S1, c_S2, c_D)
##         v_c_trt <- c(c_H, c_S1 + c_Trt, c_S2 + c_Trt, c_D)
##         v_tu_no_trt <- m_M_no_trt %*% v_u_no_trt
##         v_tu_trt <- m_M_trt %*% v_u_trt
##         v_tc_no_trt <- m_M_no_trt %*% v_c_no_trt
##         v_tc_trt <- m_M_trt %*% v_c_trt
##         tu_d_no_trt <- t(v_tu_no_trt) %*% v_dwe
##         tu_d_trt <- t(v_tu_trt) %*% v_dwe
##         tc_d_no_trt <- t(v_tc_no_trt) %*% v_dwc
##         tc_d_trt <- t(v_tc_trt) %*% v_dwc
```

```
##         v_tc_d <- c(tc_d_no_trt, tc_d_trt)
##         v_tu_d <- c(tu_d_no_trt, tu_d_trt)
##         v_nmb_d <- v_tu_d * n_wtp - v_tc_d
##         df_ce <- data.frame(Strategy = v_names_str, Cost = v_tc_d,
##             Effect = v_tu_d, NMB = v_nmb_d)
##         return(df_ce)
##     })
## }
## <bytecode: 0x000000001eb23838>
## <environment: namespace:darthpack>
```

After calculating the discount weights, this function runs the simulation model using the previously described function `decision_model` in the *02_simulatiomn_model_function.R* script. Inside the function `calculate_ce_out`, the simulation model is run for both the treatment, `l_model_out_trt`, and no treatment, `l_model_out_no_trt`, strategies of the Sick-Sicker model. Running it for both treatment strategies is done for illustration purposes. In this case-study, the resulting cohort traces are identical and we could have executed it only once.

In the second part of the function we create multiple vectors for both the cost and effects of both strategies. These vectors multiply the cohort trace to compute the cycle-specific rewards. This results in vectors of total costs (`v_tc`) and total effects (`v_tu`) per cycle. By multiplying these vectors with the vectors with the discount weights for costs (`v_dwc`) and effects (`v_dwe`) we get the total discounted mean costs (`tc_d_no_trt` and `tc_d_trt`) and QALYs (`tu_d_no_trt` and `tu_d_trt`) for both strategies. These values are used in the calculation of the NMB. Finally, the total discounted costs, effectiveness and NMB are combined in the dataframe `df_ce`. The results for our case-study are shown below.

```
df_out_ce # print the dataframe
```

```
##       Strategy     Cost  Effect     NMB
## 1 No Treatment 115244.8 20.0233 2888250
## 2    Treatment 214165.8 20.7226 2894224
```

This dataframe of CE results can be used as an argument in the `calculate_icers` function from the `dampack` package to calculate the incremental cost-effectiveness ratios (ICERs) and noting which strategies are weakly and strongly dominated. Table 5.1 shows the result of the deterministic CEA.

```
df_cea_det <- calculate_icers(cost = df_out_ce$Cost,
                              effect = df_out_ce$Effect,
                              strategies = l_params_basecase$v_names_str)
```

Finally, Figure 5.1 shows the cost-effectiveness frontier of the CEA.

We then conduct a series of deterministic sensitivity analysis. First, we conduct a one-way sensitivity analysis (OWSA) on the variables `c_Trt`, `p_HS1`, `u_S1` and `u_Trt` and a two-way sensitivity analysis (TWSA) using the owsa_det and twsa_det functions. We use the output of these functions to produce different SA plots, such as OWSA tornado, one-way optimal strategy and TWSA plots (Figures 5.2 - 5.5).

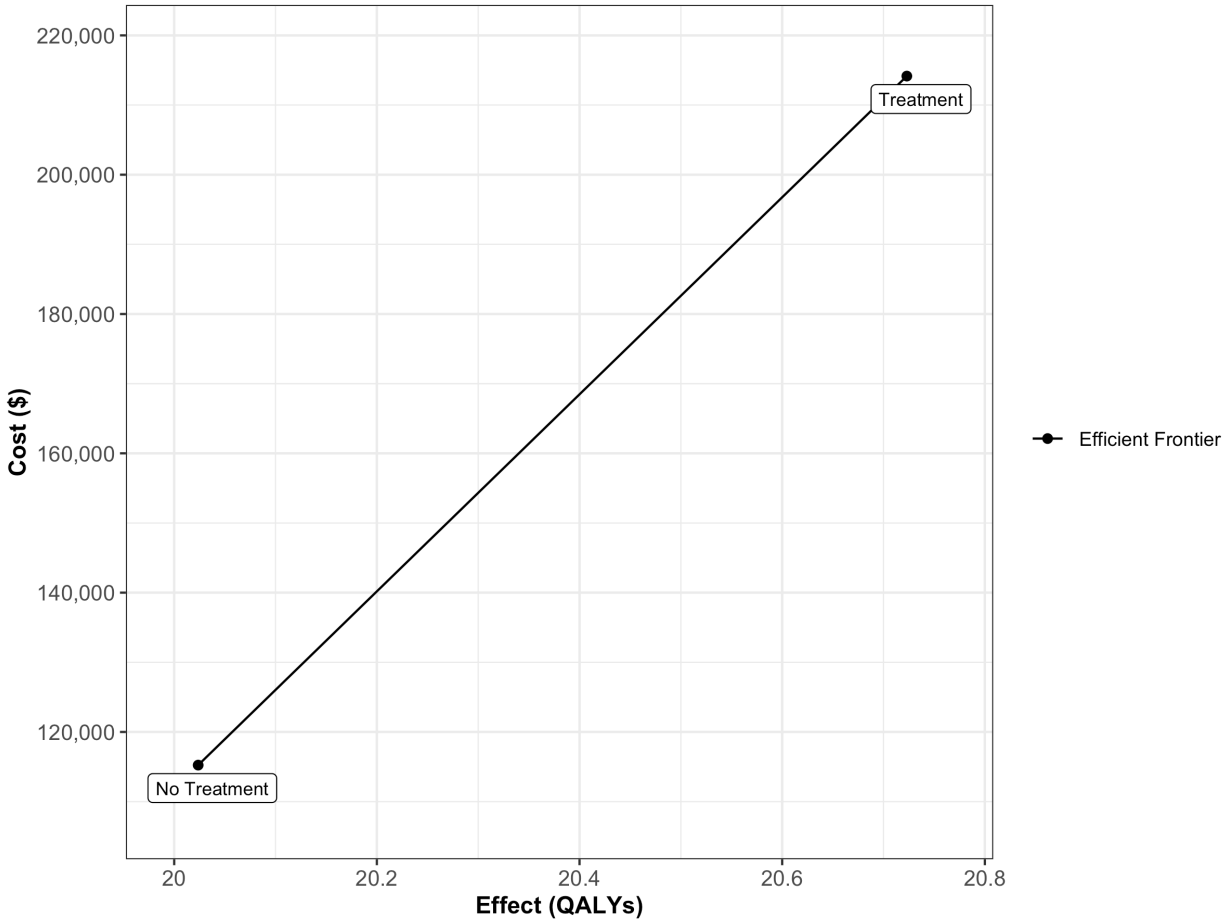

Figure 5.1: Cost-effectiveness frontier.

Table 5.1: Deterministic cost-effectiveness analysis results of the Sick-Sicker model comparing no treatment with treatment.

Strategy	Cost	Effect	Inc_Cost	Inc_Effect	ICER
No Treatment	115244.8	20.0233	NA	NA	NA
Treatment	214165.8	20.7226	98921.01	0.6992988	141457.4

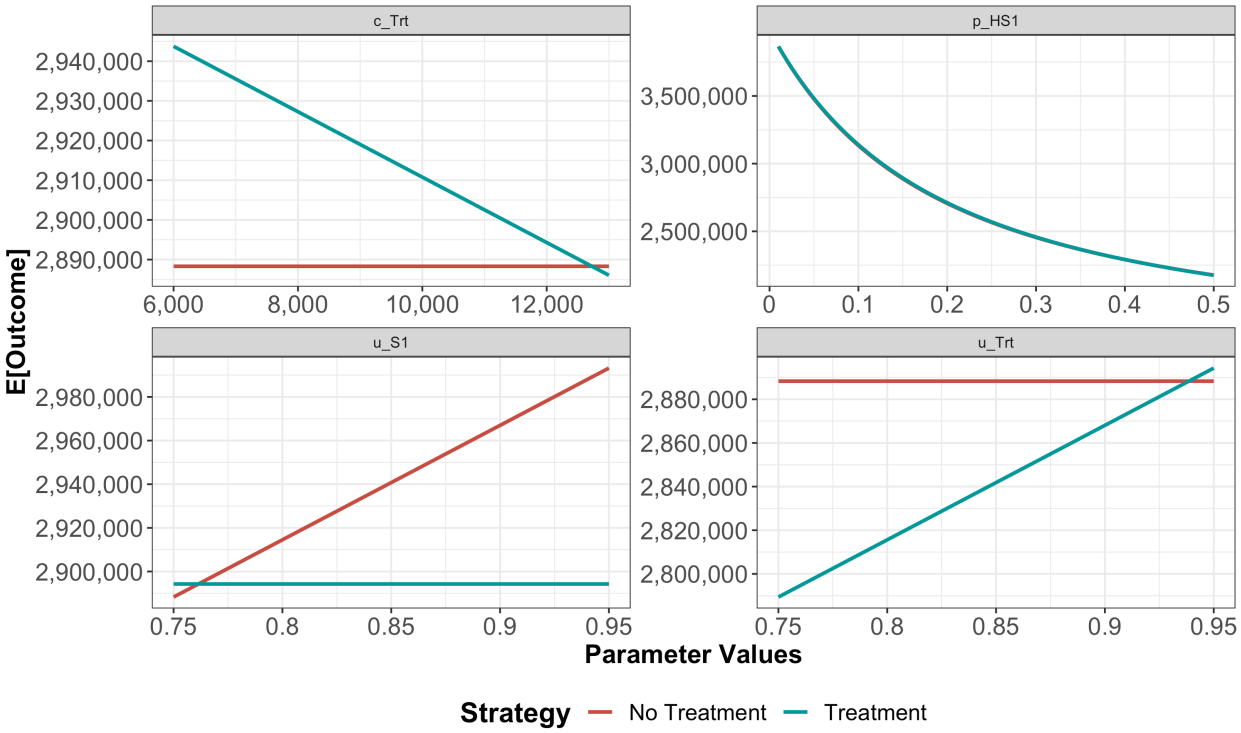

Figure 5.2: One-way sensitivity analysis results

5.2 05b Probabilistic analysis

In this subcomponent, we evaluate decision uncertainty by propagating the uncertainty through the CEA using probabilistic sensitivity analysis (PSA). Until now we used the parameter values as described in Table 1.1. However, we are uncertain about these values. Most of these input parameters are defined by probability distribution as described in Table 5.2.

Table 5.2: Description of parameters with their R name and distribution.

Parameter	**R name**	**Distribution**
Annual transition probabilities		
- Disease onset (H to S1)	`p_HS1`	`beta(30, 170)`
- Recovery (S1 to H)	`p_S1H`	`beta(60, 60)`
Annual costs		
- Healthy individuals	`c_H`	`gamma(shape = 100, scale = 20)`

Parameter	R name	Distribution
- Sick individuals in S1	c_S1	gamma(shape = 177.8, scale = 22.5)
- Sick individuals in S2	c_S2	gamma(shape = 225, scale = 66.7)
- Additional costs of sick individuals treated in S1 or S2	c.Trt	gamma(shape = 73.5, scale = 163.3)
Utility weights		
- Healthy individuals	u_H	truncnorm(mean = 1, sd = 0.01, b = 1)
- Sick individuals in S1	u_S1	truncnorm(mean = 0.75, sd = 0.02, b = 1)
- Sick individuals in S2	u_S2	truncnorm(mean = 0.50, sd = 0.03, b = 1)
Intervention effect		
- Utility for treated individuals in S1	u_Trt	truncnorm(mean = 0.95, sd = 0.02, b = 1)

In a PSA we sample the input parameter values from these distributions and we then run the model at each sample. In the file *05b_uncertainty_analysis_functions.R* we created a single function, called `generate_psa_params`. This function generates a PSA dataset for all the CEA input parameters. We specify the number of PSA samples via the `n_sim` argument. The function also accepts specifying a seed to allow reproducibility of the results.

```
print.function(generate_psa_params) # print the function
```

```
## function (n_sim = 1000, seed = 20190220)
## {
##     data("m_calib_post")
##     n_sim <- nrow(m_calib_post)
##     set_seed <- seed
##     df_psa_params <- data.frame(m_calib_post, p_HS1 = rbeta(n_sim,
##         30, 170), p_S1H = rbeta(n_sim, 60, 60), c_H = rgamma(n_sim,
##         shape = 100, scale = 20), c_S1 = rgamma(n_sim, shape = 177.8,
##         scale = 22.5), c_S2 = rgamma(n_sim, shape = 225, scale = 66.7),
##         c_Trt = rgamma(n_sim, shape = 73.5, scale = 163.3), c_D = 0,
##         u_H = truncnorm::rtruncnorm(n_sim, mean = 1, sd = 0.01,
##             b = 1), u_S1 = truncnorm::rtruncnorm(n_sim, mean = 0.75,
```

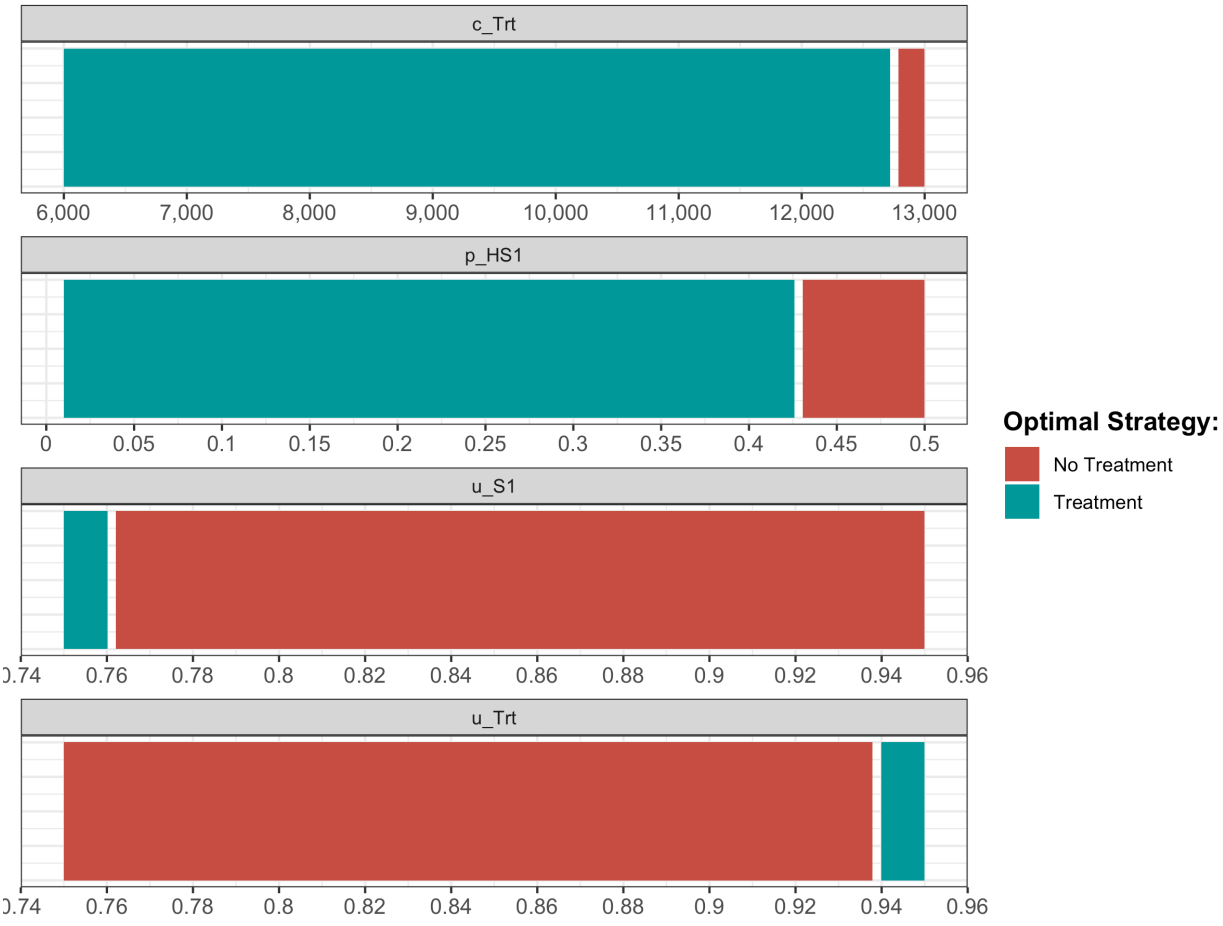

Figure 5.3: The optimal strategy with OWSA

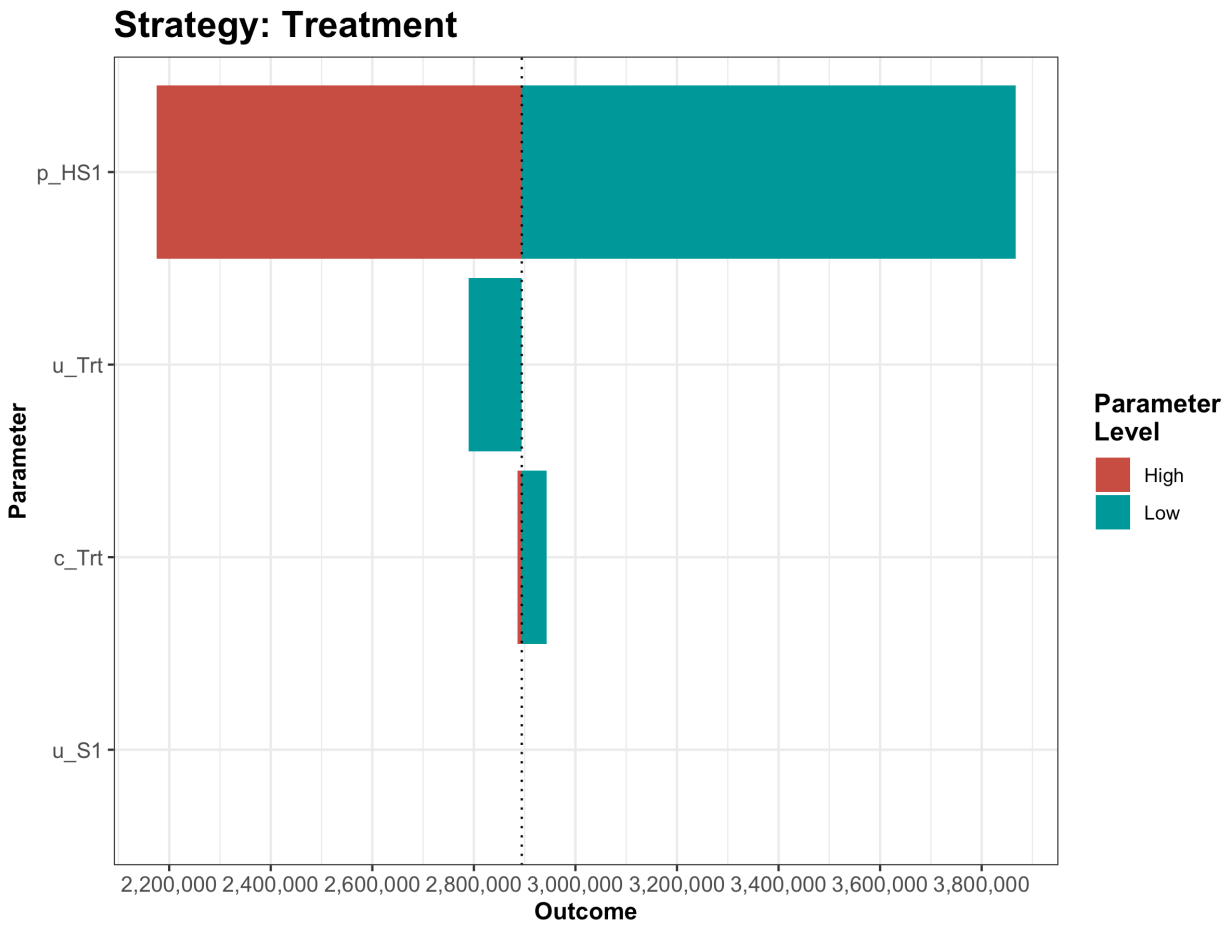

Figure 5.4: The tornado plot

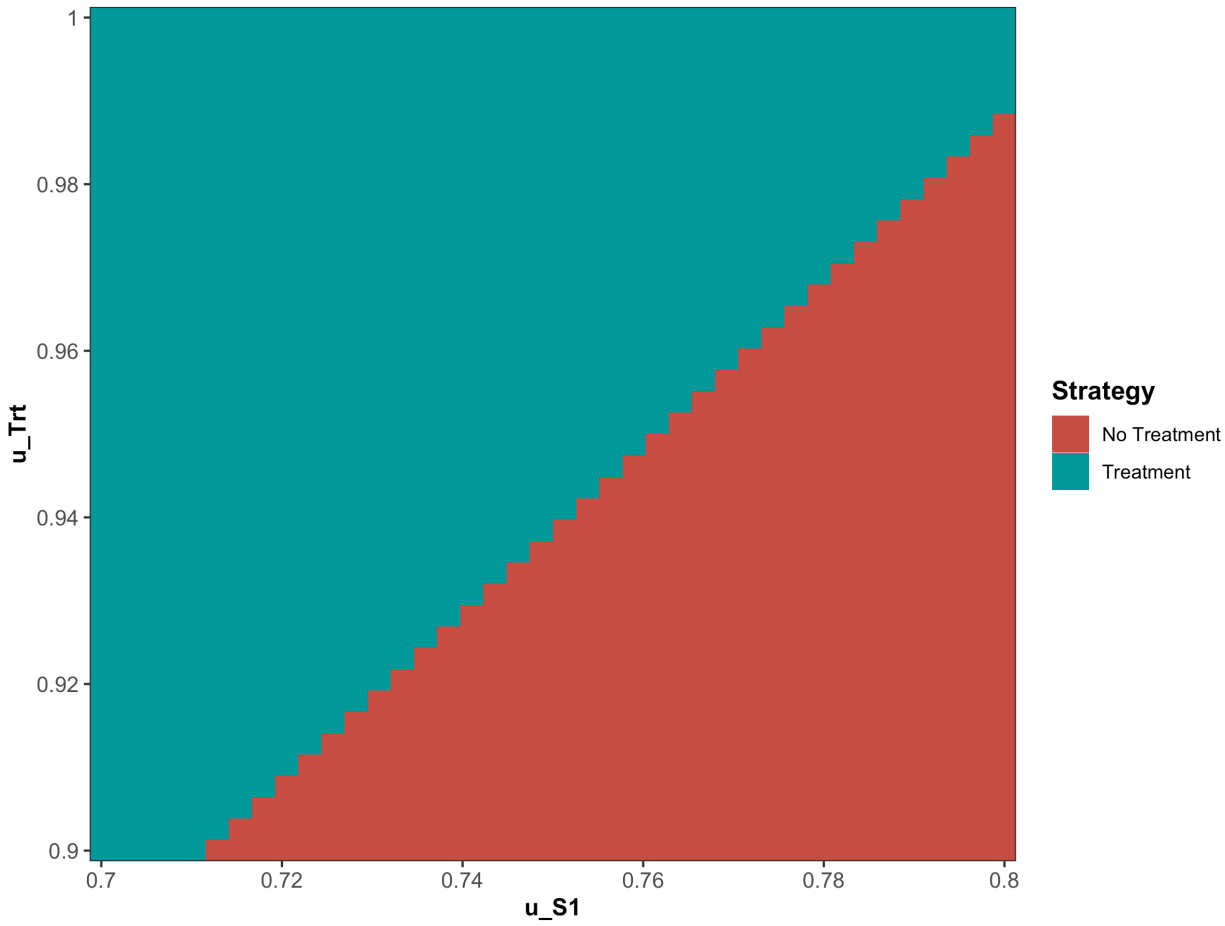

Figure 5.5: Two-way sensitivity results.

Table 5.3: Probabilistic cost-effectiveness analysis results of the Sick-Sicker model comparing no treatment with treatment

Strategy	Cost	Effect	Inc_Cost	Inc_Effect	ICER
No.Treatment	115538.9	19.95305	NA	NA	NA
Treatment	213755.8	20.63891	98216.84	0.6858646	143201.5

```
##              sd = 0.02, b = 1), u_S2 = truncnorm::rtruncnorm(n_sim,
##              mean = 0.5, sd = 0.03, b = 1), u_D = 0, u_Trt = truncnorm::rtruncnorm(n_sim,
##              mean = 0.95, sd = 0.02, b = 1))
##     return(df_psa_params)
## }
## <bytecode: 0x000000001e3aa920>
## <environment: namespace:darthpack>
```

The function returns the `df_psa_input` dataframe with a PSA dataset of the input parameters. With this dataframe we can run the PSA to produce distributions of costs, effectiveness and NMB. The PSA is performed by the *05b_probabilistic_analysis.R* script. As shown in the code below, the `df_psa_input` dataframe is used by the `update_param_list` function to generate the corresponding list of parameters for the PSA. For each simulation, we perfrom three steps. First, the list of parameters is updated by the `update_param_list` function. Second, the model is executed by the `calculate_ce_out` function using the updated parameter list and third, the dataframes `df_c` and `df_e` store the estimated cost and effects, respectively. The final part of this loop is to satisfy the modeler when waiting on the results, by displaying the simulation progress.

```
for(i in 1:n_sim){
  l_psa_input <- update_param_list(l_params_all, df_psa_input[i, ])
  df_out_temp <- calculate_ce_out(l_psa_input)
  df_c[i, ] <- df_out_temp$Cost
  df_e[i, ] <- df_out_temp$Effect
  # Display simulation progress
  if(i/(n_sim/10) == round(i/(n_sim/10), 0)) {
    cat('\r', paste(i/n_sim * 100, "% done", sep = " "))
  }
}
```

We can plot the results using the `plot` function from `dampack`. Figure 5.6 shows the CE scatter plot with the joint distribution of costs and effects for each strategy and their corresponding 95% confidence ellipse.

Next, we perform a CEA using the previously used `calculate_icers` functions from `dampack`. Table 5.3 shows the results of the probabilistic CEA. In addition, we plot a cost-effectiveness plane with the frontier, the cost-effectiveness acceptability curves (CEACs) and frontier (CEAF), expected Loss curves (ELCs) (Figure @ref(fig:05b_cea_frontier_psa) - 5.9) (Alarid-Escudero et al., 2019). Followed by creating linear regression metamodeling sensitivity analysis graphs (Figure 5.10 - @ref(fig:05b-twsa-lrm-uS1-uTrt_nmb))(Jalal et al., 2013). All generated figures are shown below and stored to the *figs* folder .

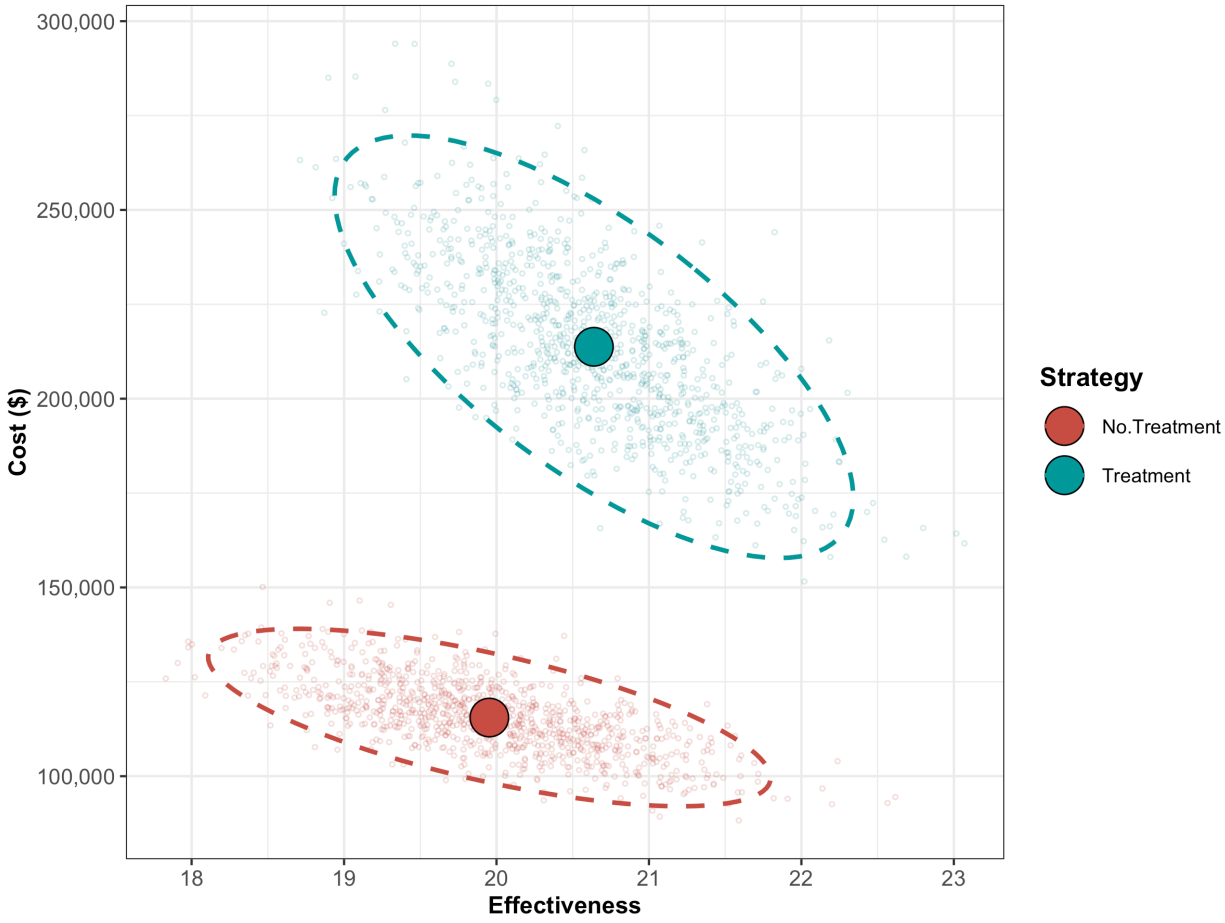

Figure 5.6: The cost-effectiveness plane graph showing the results of the probabilistic sensitivity analysis for the Sick-Sicker case-study.

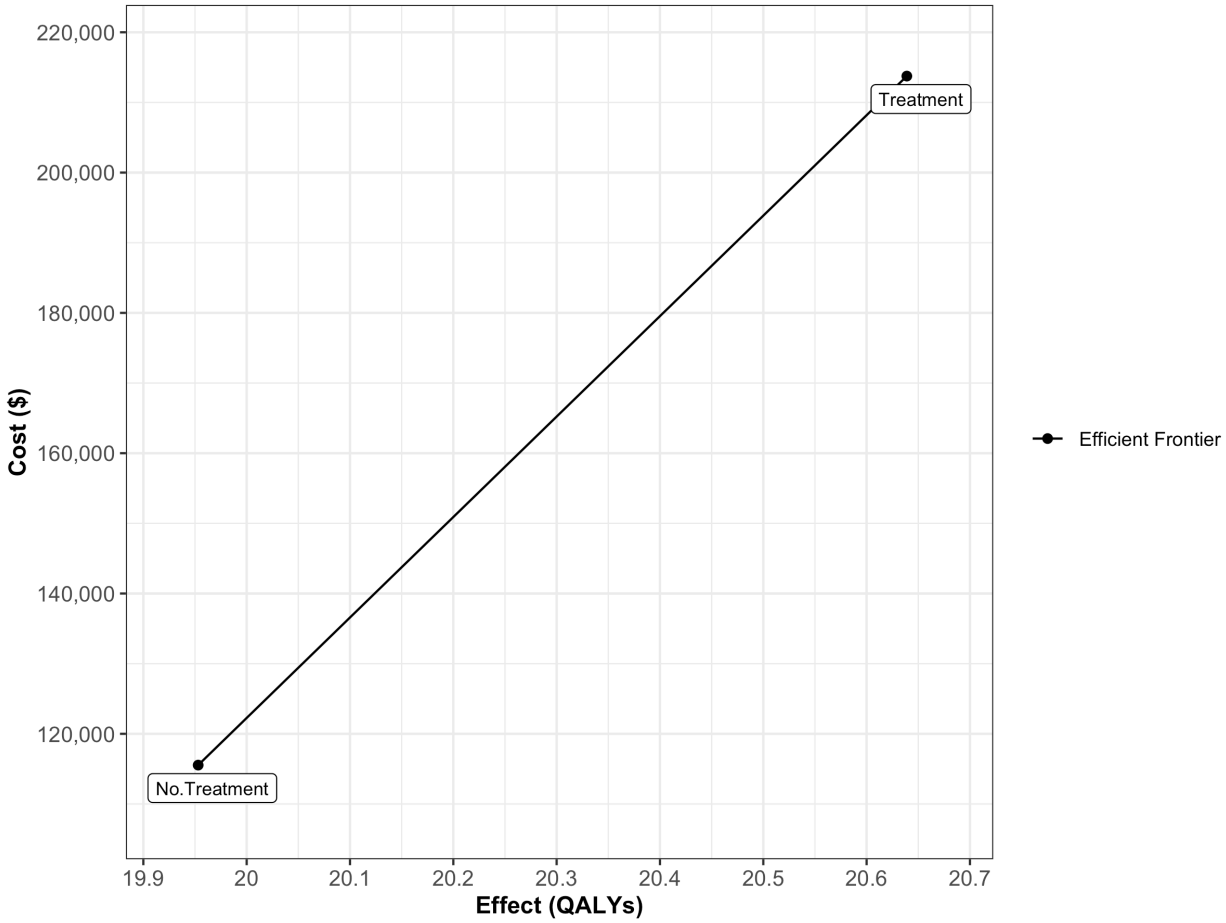

Figure 5.7: Cost-effectiveness frontier

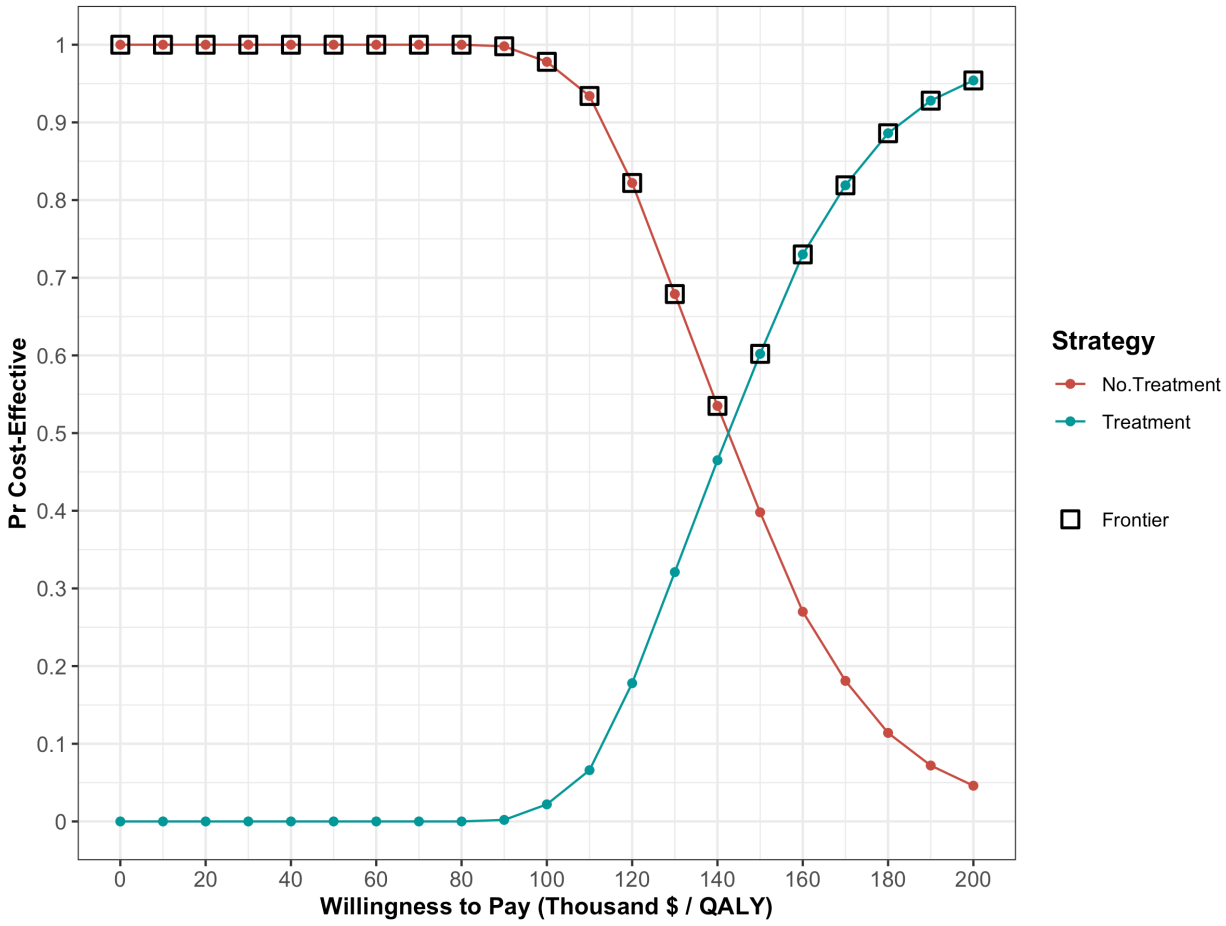

Figure 5.8: Cost-effectiveness acceptability curves (CEACs) and frontier (CEAF).

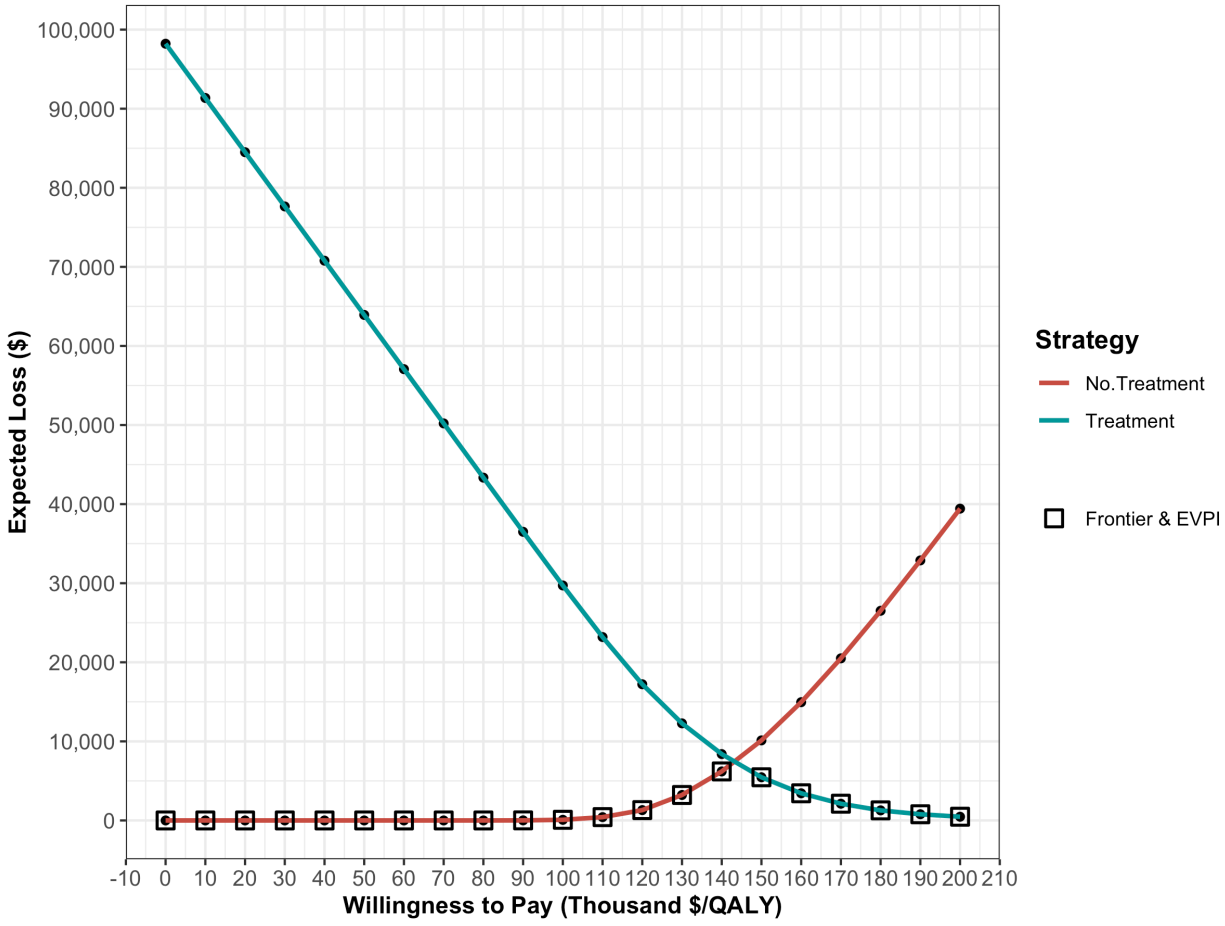

Figure 5.9: Expected Loss Curves.

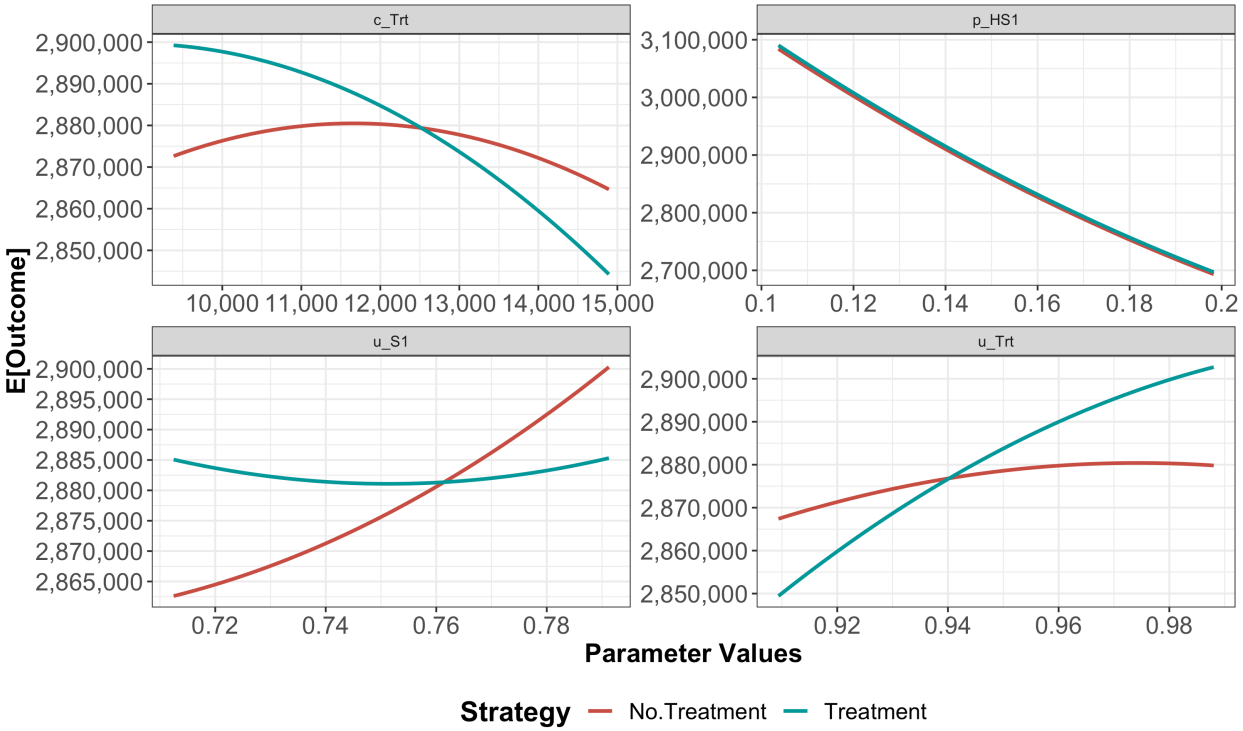

Figure 5.10: One-way sensitivity analysis (OWSA).

5.3 05c Value of information

In the VOI component, the results from the PSA generated in the probabilistic analysis subcomponent are used to determine whether further potential research is needed. We use the `calc_evpi` function from the `dampack` package to calculate the expected value of perfect information (EVPI). Figure 5.14 shows the EVPI for the different WTP values.

```
evpi <- calc_evpi(wtp = v_wtp, psa = l_psa)
```

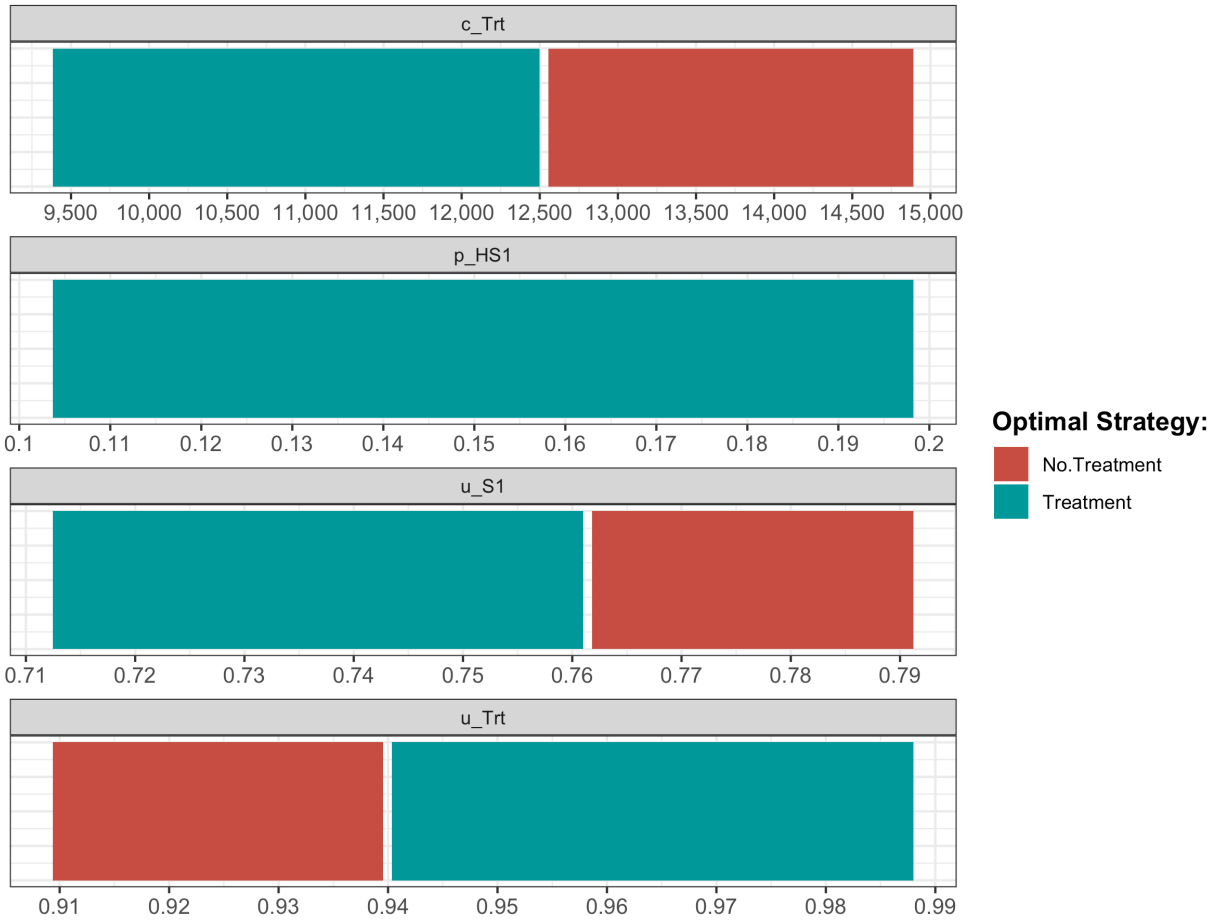

Figure 5.11: Optimal strategy with OWSA

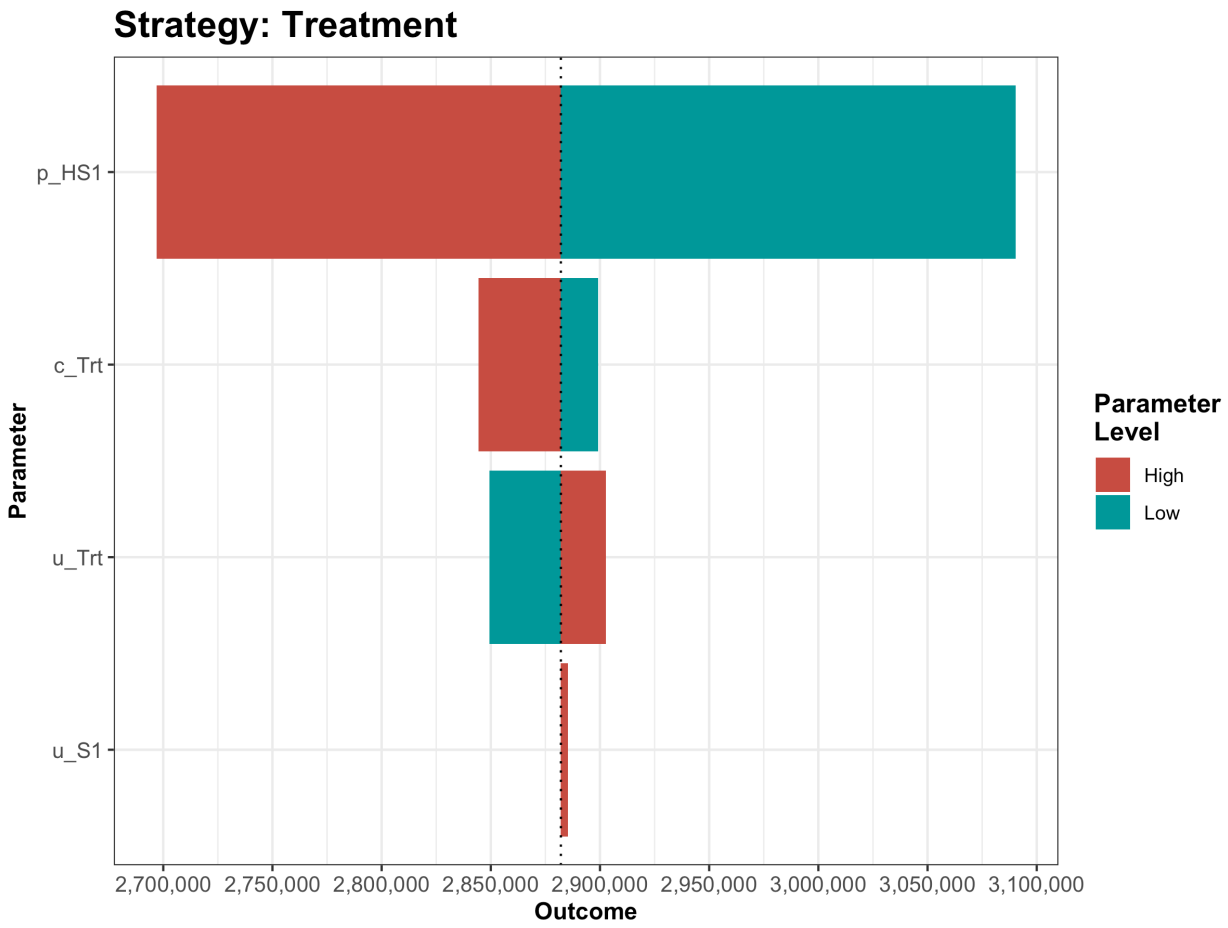

Figure 5.12: Tornado plot

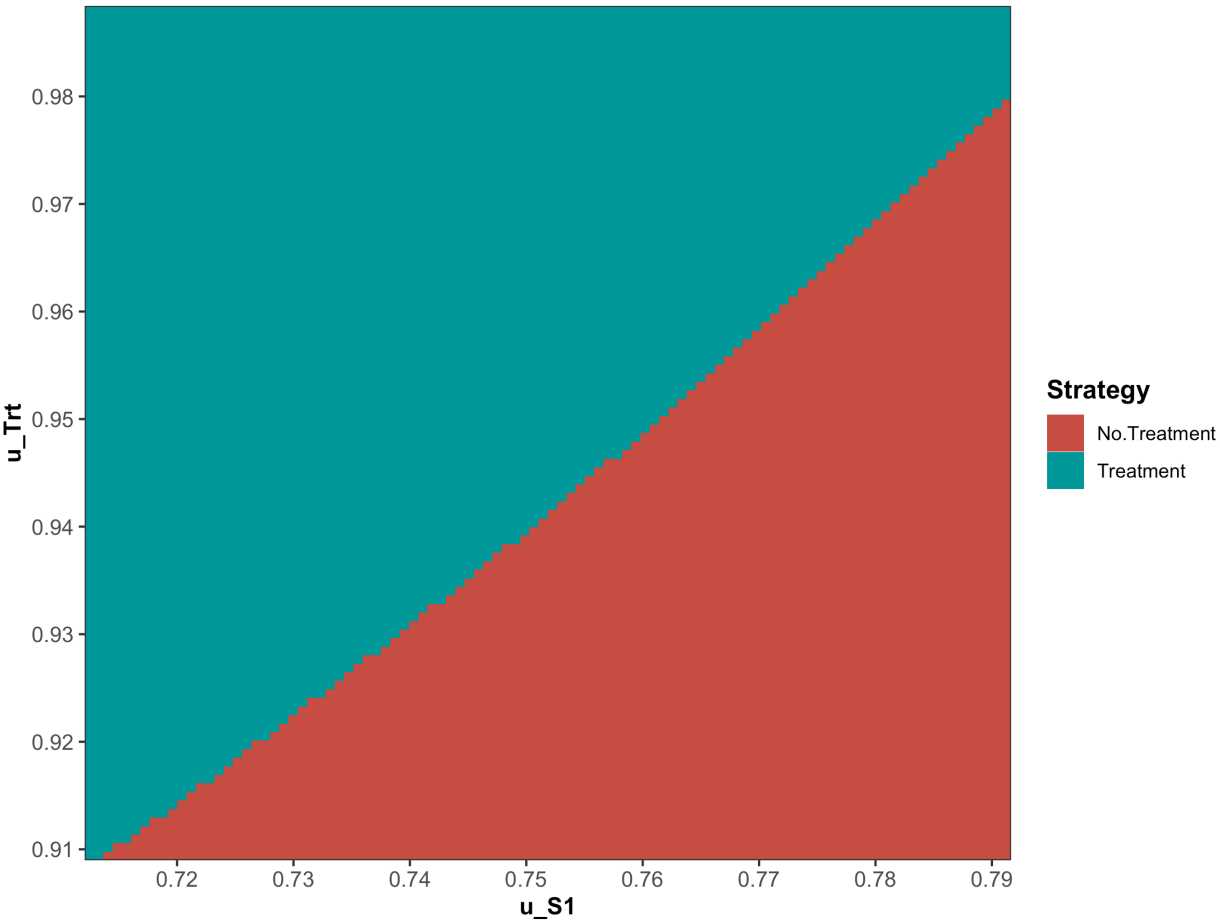

Figure 5.13: Two-way sensitivity analysis (TWSA).

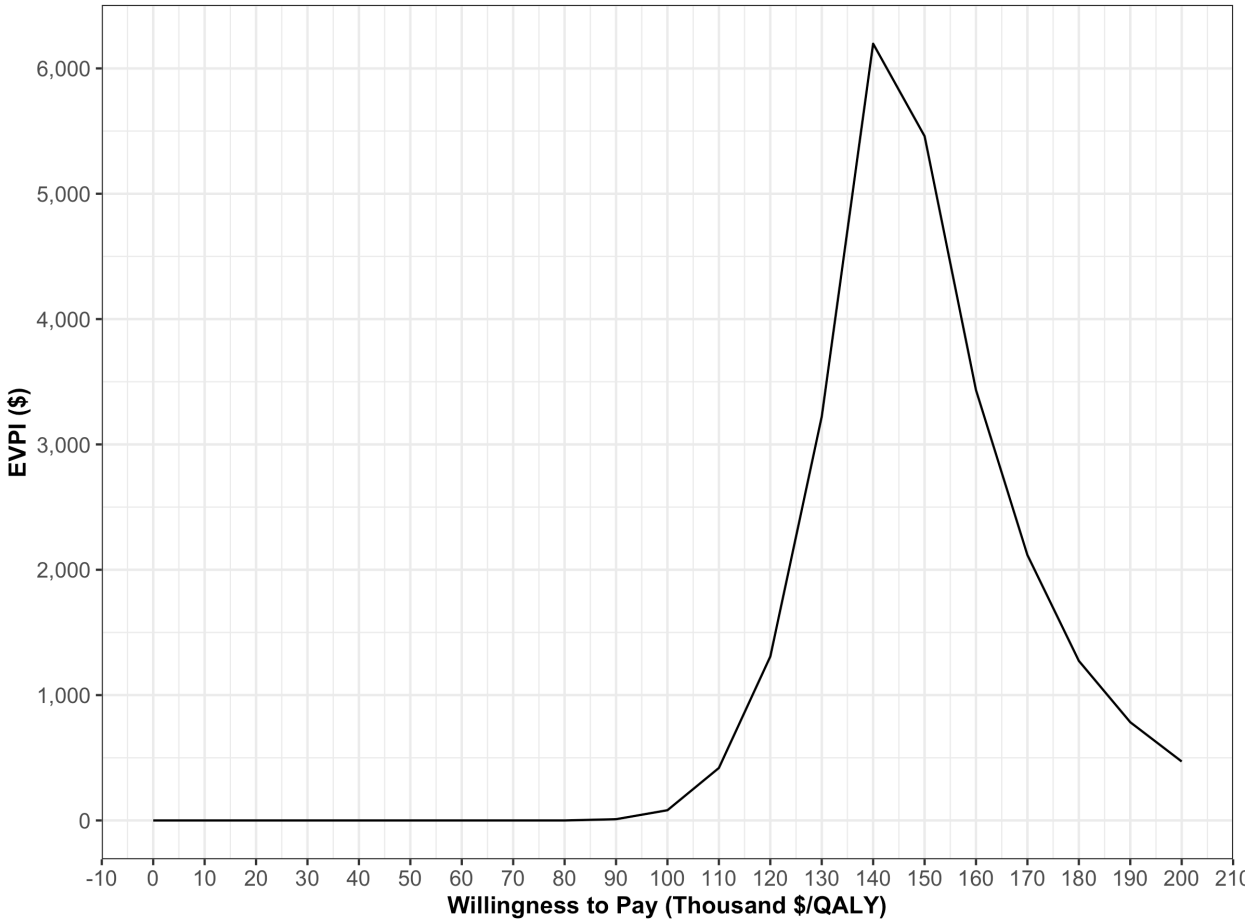

Figure 5.14: Expected value of perfect information

Bibliography

Alarid-Escudero, F., Enns, E., Kuntz, K., Michaud, T., and Jalal, H. (2019). "Time Traveling Is Just Too Dangerous" But Some Methods Are Worth Revisiting: The Advantages of Expected Loss Curves Over Cost-Effectiveness Acceptability Curves and Frontier. *Value in Health*, 22(5):611–618.

Alarid-Escudero, F., MacLehose, R., Peralta, Y., Kuntz, K., and EA, E. (2018). Nonidentifiability in model calibration and implications for medical decision making. *Medical Decision Making*, 38(7):810–821. PMID: 30248276.

Eddy, D. M., Hollingworth, W., Caro, J. J., Tsevat, J., McDonald, K. M., and Wong, J. B. (2012). Model transparency and validation: A report of the ISPOR-SMDM modeling good research practices task force-7. *Medical Decision Making*, 32(5):733–743.

Enns, E. A., Cipriano, L. E., Simons, C. T., and Kong, C. Y. (2015). Identifying Best-Fitting Inputs in Health-Economic Model Calibration: A Pareto Frontier Approach. *Medical Decision Making*, 35(2):170–182.

Goldhaber-Fiebert, J., Stout, N., and Goldie, S. (2010). Empirically evaluating decision-analytic models. *Value in Health*, 13(5):667–674.

Iskandar, R. (2018). A theoretical foundation for state-transition cohort models in health decision analysis. *PloS one*, 13(12):e0205543–e0205543.

Jalal, H., Dowd, B., Sainfort, F., and Kuntz, K. M. (2013). Linear regression metamodeling as a tool to summarize and present simulation model results. *Medical Decision Making*, 33(7):880–90.

Menzies, N. A., Soeteman, D. I., Pandya, A., and Kim, J. J. (2017). Bayesian Methods for Calibrating Health Policy Models: A Tutorial. *PharmacoEconomics*, 35(6):613–624.

Raftery, A. and Bao, L. (2010). Estimating and Projecting Trends in HIV/AIDS Generalized Epidemics Using Incremental Mixture Importance Sampling. *Biometrics*, 66(4):1162–1173.

Raftery, A. and Le Bao (2012). *IMIS: Increamental Mixture Importance Sampling*. R package version 0.1.

Rutter, C., Ozik, J., DeYoreo, M., and N, C. (2018). Microsimulation Model Calibration using Incremental Mixture Approximate Bayesian Computation. *arXiv*, (april):1–20.

Steele, R., Raftery, A., and Emond, M. (2006). Computing Normalizing Constants for Finite Mixture Models via Incremental Mixture Importance Sampling (IMIS). *Journal ofComputational and Graphical Statistics*, 15(3):712–734.