Decision modeling with R: lessons learned from the development of hesim

Devin Incerti | Genentech
July 1, 2021 | R for HTA Showcase

Background

- Decision models are complex mathematical models that often contain large numbers of parameters and complex relationships between inputs and outputs
- Complexity often result in models that are not transparent, efficient, or flexible
- While it has argued that this is caused, in part, by choice of software (e.g., Excel vs R), software is not the only culprit
- Good code is just as important---if not more so--- than the choice of software
- Following software engineering best practices can improve the quality of models and enhance their credibility

Attributes of good code*

Transparency 	*Transparent code is easily understood and reproducible*
Reliability 	*Reliable code produces expected output without errors*
Efficiency 	*Efficient code runs fast enough and is programmed quickly enough to achieve its intended purpose*
Extensibility 	*Extensible code is easily adapted to allow for new capabilities or functionality*

*Attributes listed here are non-exhaustive

Example

Decision problem

- 5 competing treatment strategies
- 10 representative patients (vary by age and sex)
- Decision framework: CEA

Decision model

- Markov cohort model
- Probabilistic sensitivity analysis
- 3 health states (sick, sicker, death)
- Yearly model cycles
- 20 year time horizon

Complement = 1 – Pr(sick) – Pr(death)

Treatment effects for treatment strategies are (log) odds ratios

Additional covariates Z (age, sex)

Multinomial logistic regression

	Sick	Sicker	Death
Sick	C	$\frac{e^{A\gamma_2+Z\delta_2}}{1+\sum_{k=2}^{3} e^{\gamma_3 A+Z\delta_k}}$	$\frac{e^{A\gamma_3+Z\delta_3}}{1+\sum_{k=2}^{3} e^{\gamma_k A+Z\delta_k}}$
Sicker	0	C	$1-\exp[-rt]$
Death	0	0	1

Rate modeled as function of age and sex

Lesson 1: plan ahead

Start by thinking about the output you will need/want to answer your decision problem

State probabilities

Sample	Strategy	Patient	Cycle	Pr(sick)	Pr(sicker)	Pr(death)
1	1	1	0	.9	.08	.01
1	1	1	1	.85	.13	.02
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Expected values (costs and QALYs)

Sample	Strategy	Patient	Costs	QALYs
1	1	1	50,000	7
1	1	1	45,000	6.5
⋮	⋮	⋮	⋮	⋮

We can then think about the steps/functions needed to produce that output

Markov modeling steps

1. Set treatment strategies and patients (input data)
2. Sample parameters for PSA
3. Simulate state probabilities
4. Simulate costs & QALYs
5. Perform CEA
   a. ICER
   b. PSA

Modeling pipeline

```
input_data <- make_input_data()
params <- get_params()
stprobs <- sim_stateprobs(input_data, params)
ev <- sim_ev(stprobs, input_data, params)
icer_out <- icer(ev)
psa_out <- psa(ev)
```

Lesson 1 cont'd: convenient model inputs

A convenient way to model multiple treatment strategies (n = 5) and patients (n = 10) is to create a single data frame (the "input data")

	strategy_id	patient_id	age	female	strategy2	strategy3	strategy4	strategy5
1	1	1	65.18746	1	0	0	0	0
2	1	2	63.15747	1	0	0	0	0
49	5	9	48.73327	1	0	0	0	1
50	5	10	62.43522	0	0	0	0	1

In a PSA, the parameters are drawn from suitable probability distributions:

Multinomial logit coefficients for sick → sicker transition (same for sick -> death)

strategy2	strategy3	strategy4	strategy5	intercept	age	female
-0.1987610309	-0.5188277554	-0.3727531176	-0.0015692633	-1.6293260952	0.0007976859	-0.1043042118

Coefficients for rate parameter of sicker → death transition

intercept	age	female
-1.159677128	0.004135311	-0.100374964

Recall that we need simulation results for every strategy, patient, and PSA sample. An important consideration is whether to draw the parameters once at each iteration (as shown here) or to draw all at once (i.e., to vectorize)

Lesson 2: choose a style guide and stick to it

Let's start with some "bad" code:

A "dot" is confusing because its unclear if the function is an S3 method

Usually better for functions to be verbs

*Inconsistent naming conventions for **objects***

*Inconsistent naming conventions for **functions***

But don't need a verb if function computes well known noun

```
state_probs <- sim.StateProbs()

expectedValues <- expected_values_sim()

ICER <- icer()

psa_out <- psa()
```

Some better code:

```
stateprobs <- sim_stateprobs()

expected_values <- sim_expected_values()

icer_out <- icer()

psa_out <- psa()
```

Lesson 3: write modular code

```
sim_stateprobs <- function(input_data, params,
                           n_samples, n_cycles, x0) {
  for (s in 1:n_samples) {
    for (i in 1:nrow(input_data)) {

      # A bunch of code to sample the parameters here
      params_sample <- list()
      for  (j in 1:n_params) {
        k <- ncol(params[[j]]$mean[j])
        params_sample[[j]] <- rnorm(k, params[[j]]$mean,
                                    params[[j]]$sd)}
       }

    # A lot of code to get transition probability matrix
    # Predict probabilities with multinomial logit
    # Predict probabilities with exponential model

    # More code to simulate the Markov chain
     for (t in 1:n_cycles) {
      x[t + 1, ] <- x[t, ] %*% p
     }
  } # End loop of input data
} # End loop over treatment strategies
```

VS

```
sim_stateprobs1 <- function(input_data, params,
                            n_cycles, x0) {
  params_sample <- sample_params(n = 1, params)
  tpmat <- tpmatrix(input_data, params_sample)
  sim_markov_chain(x0, p = tpmat, n_cycles
}
```

```
sim_stateprobs <- function(input_data, params,
                           n_samples, n_cycles, x0) {
  for (s in 1:n_samples) {
    for (i in 1:nrow(input_data)) {
     sim_stateprobs1(input_data, params, n_cycles, x0)
   }
  }
}
```

- Why?
  - Can read the code almost like reading the English language (*transparency*)
  - Easier to test functions and identify source of bugs (*reliability*)
  - Easier to refactor when adding new features (*extensibility*)

Lesson 3 cont'd: modularizing the construction of the transition probability matrices

```
tpmatrix <- function(input_data, params)
rbind(
   tp_sick(input_data, params),
   tp_sicker(input_data, params),
   c(0, 0, 1) # Death is an absorbing state
 )
)
```

```
            sick    sicker      death
[1,] 0.7553365 0.1585003 0.08616317
[2,] 0.0000000 0.7441817 0.25581834
[3,] 0.0000000 0.0000000 1.00000000
```

Transition probabilities from sick state

```
tp_sick <- function(input_data, params) {
  beta <- params[c("sick_sicker", "sick_death")]
  x <- make_x(input_data, beta[[1]])
  mlogit_probs(x, beta)
}
```

Transition probabilities from sicker state

```
tp_sicker <- function(input_data, params) {
  beta <- params[["sicker_death"]]
  x <- make_x(input_data, beta)
  rate <- exp(x %*% t(beta))
  prob_death <- 1 - exp(-rate)
  c(sick = 0, sicker = 1 - prob_death, death = prob_death)
}
```

Transition probabilities from sick state predicted from multinomial logit model

```
mlogit_probs <- function(x, beta) {

  # General code to predict probabilities given
  # coefficients from a multinomial regression
}
```

Create "input matrix" by selecting columns of input data corresponding to parameters

```
make_x <- function(input_data, params) {
  input_data[, "intercept"] <- 1 # Add intercept
  as.matrix(input_data[, colnames(params)])
}
```

Lesson 4: vectorize R code when feasible

- In our toy model with 5 treatment strategies, 10 patients, 20 (annual) model cycles, and 1,000 PSA samples, there are 5 * 10 * 20 * 1,000 = 1,000,000 iterations
- Looping many times in pure R can be slow; the prior code runs in ~45 seconds
- We can speed up the code considerably (~2 seconds) by vectorizing (looping with compiled code) with `hesim`

Using `hesim::tpmatrix()` to precompute transition probabilities for every combination of ID variables

	sick.sick	sick.sicker	sick.death	sicker.sick	sicker.sicker	sicker.death	death.sick	death.sicker	death.death
1:	0.7648088	0.1412419	0.09394930	0	0.6996350	0.3003650	0	0	1
2:	0.7408574	0.1577365	0.10978398	0	0.6651783	0.3348217	0	0	1
3:	0.7368429	0.1585800	0.11456486	0	0.6576189	0.3423811	0	0	1
4:	0.7701127	0.1401260	0.08817804	0	0.7097757	0.2902243	0	0	1
5:	0.7429434	0.1572912	0.10733075	0	0.6691363	0.3308637	0	0	1

	sample	strategy_id	patient_id
1:	1	1	1
2:	1	1	2
3:	1	1	3
4:	1	1	4
5:	1	1	5

The GitHub repo for this presentation shows how the non-vectorized code can be slightly tweaked to create transition matrices in a vectorized manner. `hesim` will then implement a Markov chain (written in C++) for each row in the table above

Concluding thoughts

- R is a good software tool for decision modeling but its important to following good software practices
- If done carefully, it can result in very transparent modeling; however, writing transparent code is a lot of extra effort and neither academia, industry, or consulting provide strong incentives for doing this
- There is a lot of potential to leverage open-source packages to make work more efficient and less error prone; we should work together more
- We aren’t software engineers, but we should try to write code a little more like them