Introduction to R for cost-effectiveness analysis

Eva Enns & Zoe Kao

2020-04-06

1 R Basics

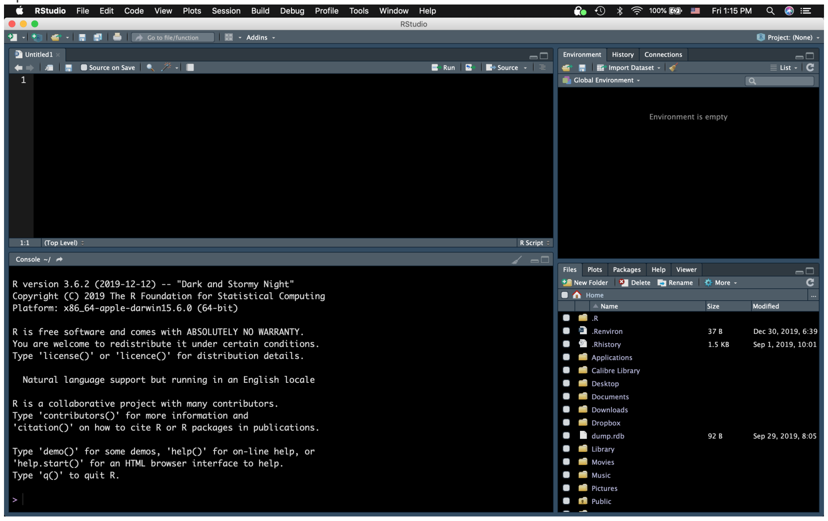
This lecture is to introduce the elements in R programming language that are relevant to decision model and decision analysis.

1.1 Agenda

- Install R and RStudio
- Data type
- Path to your working directory
- Data input and output
- For loop
- If statement
- R function
- Plots
- Useful packages in the class

1.2 Install R and RStudio

- You could install R here.
- Be aware of your operating system
- The original R interface is not user-friendly. We recommend using the RStudio IDE here
    - Just download the free version!
- Open RStudio



```
R version 3.6.2 (2019-12-12) -- "Dark and Stormy Night"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Name	Size	Modified
.R		
.Renviron	37 B	Dec 30, 2019, 6:39
.Rhistory	1.5 KB	Sep 1, 2019, 10:01
Applications		
Calibre Library		
Desktop		
Documents		
Downloads		
Dropbox		
dump.rdb	92 B	Sep 29, 2019, 8:05
Library		
Movies		
Music		
Pictures		
Public		

1.3 Data type

- In this class, we will mainly use `vector`, `matrix`, `array`, `list`, and `data.frame`.
- These types of data are used intensively in constructing Markov model and in the package of decision analyses

1.3.1 Vector

```
x1 <- c(3, 1, 4)
print(x1)
```

```
## [1] 3 1 4
```

```
x2 <- c(2, 7, 1)
print(x1 + x2)
```

```
## [1] 5 8 5
```

```
future_human <- c("earther", "martian", "belter")
print(future_human)
```

```
## [1] "earther" "martian" "belter"
```

- You could combine different type of vectors
    - This operation coerces the data type into text

```
x1_and_future_human <- c(x1, future_human)
print(x1_and_future_human)
```

```
## [1] "3"       "1"       "4"       "earther" "martian" "belter"
```

- We often create the initial status in the Markov model using vector

    - The distribution of healthy, sick, and dead in a population
    - You can name the elements in the vector

```
v_init <- c(0.9, 0.09, 0.01)
names(v_init) <- c("healthy", "sick", "dead")
print(v_init)
```

```
## healthy    sick    dead
##    0.90    0.09    0.01
```

```
print(sum(v_init))
```

```
## [1] 1
```

- Access an element(s) in a vector

```
print(v_init[2])
```

```
## sick
## 0.09
```

```
print(v_init["sick"])
```

```
## sick
## 0.09
```

```
print(v_init[2:3])
```

```
## sick dead
## 0.09 0.01
```

```
print(v_init[c("sick", "dead")])
```

```
## sick dead
## 0.09 0.01
```

1.3.2 Matrix & Array

- Matrix is defined with two dimension using the numbers of columns and rows

```
x <- matrix(c(1, 0, 0,
              0.8, 0.2, 0,
              0, 0, 1),
            byrow = T, nrow = 3)
print(x)
```

```
##      [,1] [,2] [,3]
## [1,]  1.0  0.0    0
## [2,]  0.8  0.2    0
## [3,]  0.0  0.0    1
```

- We often use this as the basis of transition matrix in Markov model or use this to trace the outcomes over time.
- You could also provide the names to the columns and rows

```
rownames(x) <- colnames(x) <- c("healthy", "sick", "dead")
print(x)
```

```
##         healthy sick dead
## healthy     1.0  0.0    0
## sick        0.8  0.2    0
## dead        0.0  0.0    1
```

- We can access the value of a specific cell

```
print(x[2, 2])
```

```
## [1] 0.2
```

```
print(x["sick", "sick"])
```

```
## [1] 0.2
```

- Matrix multiplication is often used in Markov models. For example, we can multiply the initial state with the transition matrix
    - `v_init` is not a matrix. Thus, we use `t()` to convert `v_init` into a 1×3 matrix.
    - `%*%` is the symbol for matrix multiplication in `R`

```
print(t(v_init) %*% x)
```

```
##      healthy  sick dead
## [1,]   0.972 0.018 0.01
```

- If transition probabilities in the transition matrix vary with time, `array()` is more useful.
- `array()` can be seen as a data type that stores multiple matrices all at once.
- Let's create two transition matrices and combine them into an array.

```
tr1 <- x
tr2 <- matrix(c(0.9, 0.1, 0,
                0.7, 0.2, 0.1,
                0, 0, 1),
              byrow = T, nrow = 3,
              dimnames = list(c("healthy", "sick", "dead"),
                              c("healthy", "sick", "dead")))
```

- Create an array with dimension $3 \times 3 \times 2$

```
tr_array <- array(dim = c(3, 3, 2),
                  data = cbind(tr1, tr2),
                  dimnames = list(c("healthy", "sick", "dead"),
                                  c("healthy", "sick", "dead"),
                                  c(1, 2)))
print(tr_array)
```

```
## , , 1
##
##         healthy sick dead
## healthy     1.0  0.0    0
## sick        0.8  0.2    0
## dead        0.0  0.0    1
##
## , , 2
##
##         healthy sick dead
## healthy     0.9  0.1  0.0
## sick        0.7  0.2  0.1
## dead        0.0  0.0  1.0
```

- We can access each transition matrix by proper indexing

```
print(tr_array[ , ,  1])
```

```
##         healthy sick dead
## healthy     1.0  0.0    0
## sick        0.8  0.2    0
## dead        0.0  0.0    1
```

```
print(tr_array[ , ,  2])
```

```
##         healthy sick dead
## healthy     0.9  0.1  0.0
## sick        0.7  0.2  0.1
## dead        0.0  0.0  1.0
```

- We can also access a specific value of the transition array

```
print(tr_array[1, 2, 1])
```

```
## [1] 0
```

- Similarly, we can perform calculation using a slice of array

```
v_time2 <- t(v_init) %*% tr_array[ , , 1]
print(v_time2)
```

```
##      healthy  sick dead
## [1,]   0.972 0.018 0.01
```

```
v_time3 <- v_time2 %*% tr_array[ , , 2]
print(v_time3)
```

```
##      healthy   sick   dead
## [1,]  0.8874 0.1008 0.0118
```

1.3.3 List

- List can combine different types of data without affecting the data type.

```
temp_ls <- list(future_human = future_human, v_init = v_init, tr_array = tr_array)
print(temp_ls)
```

```
## $future_human
## [1] "earther" "martian" "belter"
##
## $v_init
## healthy    sick    dead
##    0.90    0.09    0.01
##
## $tr_array
## , , 1
##
##         healthy sick dead
## healthy     1.0  0.0    0
## sick        0.8  0.2    0
## dead        0.0  0.0    1
##
## , , 2
```

```
##
##         healthy sick dead
## healthy     0.9  0.1  0.0
## sick        0.7  0.2  0.1
## dead        0.0  0.0  1.0
```

- There are two approaches to access an element in the list

```
print(temp_ls[[1]])
```

```
## [1] "earther" "martian" "belter"
```

```
print(temp_ls$v_init)
```

```
## healthy    sick    dead
##    0.90    0.09    0.01
```

- You can access a specific element in an element of a list

```
print(temp_ls$tr_array[, , 1])
```

```
##         healthy sick dead
## healthy     1.0  0.0    0
## sick        0.8  0.2    0
## dead        0.0  0.0    1
```

1.3.4 `data.frame`

- If your Stats or Biostats classes use `R`, you probably have encountered `data.frame()` very frequently.
- Let's create a `data.frame()`

```
profile <- data.frame(name = c("Amos", "Bobbie", "Naomi"),
                      human_type = c("earther", "martian", "belter"),
                      height = c(1.8, 2.1, 1.78))
print(profile)
```

```
##     name human_type height
## 1   Amos    earther   1.80
## 2 Bobbie    martian   2.10
## 3  Naomi     belter   1.78
```

- `data.frame()` is essentially a named list of vectors with the same length.

```
typeof(profile)
```

```
## [1] "list"
```

```
length(profile$name)
```

```
## [1] 3
```

```
length(profile$human_type)
```

```
## [1] 3
```

```
length(profile$height)
```

```
## [1] 3
```

- Access elements in `data.frame()`

```
profile[profile$name == "Bobbie", ]
```

```
##     name human_type height
## 2 Bobbie    martian    2.1
```

```
profile[profile$name == "Bobbie", "height"]
```

```
## [1] 2.1
```

1.4 Path to your working directory

- It is recommended to create a folder/directory for each of your project.
- You would store all your R script, data, and outputs in the directory.
- Whenever you are working on a project, set up your working directory first.

```
setwd("path-to-your-project-folder")
setwd("zoeslaptop/Documents/CEA/introR/")
```

- The `setwd()` function sets up your working directory and allows you to access the files and folders in the working directory much easier.
- For example, I want to run the R script `hello.R` in the `Rscript/` folder in my working directory.

- Without setting up the working directory first, I have to type `source("zoeslaptop/Documents/CEA/introR/Rscript/hello.R")`
- If setting up the working directory first, I can run the script by this command: `source("Rscript/hello.R")`

- The format of path could be slightly different between windows, unix, and linux
- RStudio provides a very nice feature `project`. A project sets up the working environment automatically. Whenever you open a project, you are in the working directory to a specific project already!
- You could find more information about creating an R project and other useful tip here.

1.5 Data input and output

- You could load/read or save/write your own datasets.
- R data format family:
- `.RData`, `.rda`, and `.rds`.
- You can retrieve the information in these data objects using functions `load()` and `readRDS()`
- You can save the data using `save()` and `saveRDS()`.
- It is common that you might encounter other data types (e.g., `.csv` and `.txt`) or even data format for other software (e.g., Stata and SAS).
- To read or save other types of data, you could find the information here and here.

1.6 Loops

- Loops help repeat routine or calculations.
- Let's watch a video from Codecademy.

- Components included in loops
  - A well-defined routine/process
  - When to start
  - When to stop
- The most used loop is `for` loop in many decision models. The structure follows:

```
for (i in c(start : end)) {
  # Routine / Process
}
```

Example 1: Fibonacci sequence is the sum of the two preceding numbers. We start the sequence from 0 and 1. What are the first 10 Fibonacci numbers? (0 and 1 are the 1st and 2nd Fibonacci number, respectively.)

- Note that we want to get all the 10 Fibonacci numbers. We need to create a vector to store all the 10 numbers.
  - We already know the first two values. Thus, we replace the first two values in the `fib_vec`.

```
fib_vec <- rep(0, 10)
fib_vec[c(1:2)] <- c(0, 1)
print(fib_vec)
```

```
##  [1] 0 1 0 0 0 0 0 0 0 0
```

- The start point and end point are 3 and 10 because we already knew the first two Fibonacci numbers.

```
for (i in c(3 : 10)) {
  fib_vec[i] <- fib_vec[i - 1] + fib_vec[i - 2]
}

print(fib_vec)
```

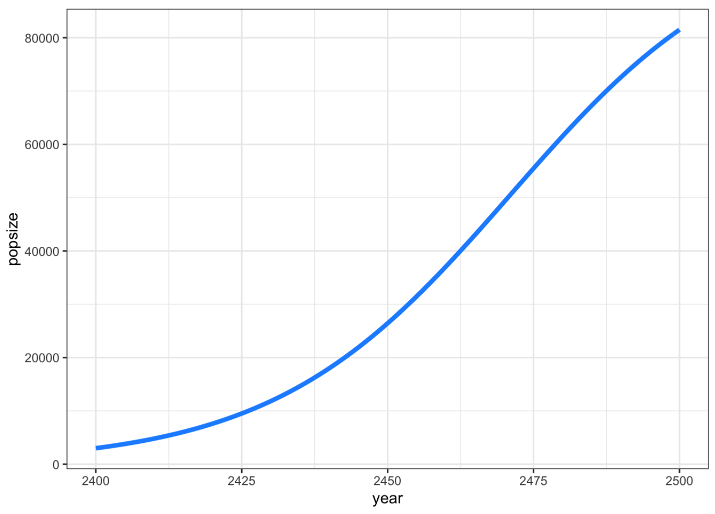
```
##  [1]  0  1  1  2  3  5  8 13 21 34
```

Example 2: In year 2400, there are 3000 martians in Mars colony. The growth rate of the martian population follows this formula $0.05P(t)(1-\frac{P(t)}{10000})$ What is the population size in year 2500?

- Note that the growth rate depends on the population size at each year.
- You can use for loop to calculate the population size at each year.

```
# initialize a vector of population size over the next 100 years
popsize <- rep(3000, 100)

# Calculate
for (t in c(1 : 100)) {
  popsize[t + 1] <- popsize[t] + 0.05 * popsize[t] * (1 - popsize[t] / 100000)
}

popdf <- data.frame(year = c(2400:2500),
                    popsize = popsize)
print(popdf[popdf$year == 2500, ])
```

```
##     year  popsize
## 101 2500 81499.86
```



1.7 If statement

- If statement is used when you just want to execute the chunk of the code if a certain condition is met.
- The structure follows

```
if (condition1) {
  # Execute some code
} else if (condition2) {
  # Execute some code
} else {
  # Execute some code
}
```

- `else if` and `else` are not always needed.
- Before we jump into examples of if statement, let's take a look at how to write the condition in the `if()`

```
"yoda" == "windu"
```

```
## [1] FALSE
```

```
Jedi <- c("yoda", "windu", "kenobi")
"yoda" %in% Jedi
```

```
## [1] TRUE
```

- If statement only execute the code when the condition in `if()` is true.

```
Mandalorian <- c("satine", "sabine", "jango")

x <- "yoda"

if (x %in% Jedi) {
  print("May the force be with you!")
} else if (x %in% Mandalorian) {
  print("This is the way!")
} else {
  print("Hello, world!")
}
```

```
## [1] "May the force be with you!"
```

- Note that R is case sensitive. If you type `"Yoda"`, you get `"Hello, world!"`

1.8 R function

- As your code grows, copying the entire decision model many times is annoying.
- Consider programming some routines / processes in a function if you will reuse the routines / processes many times.
- There are many example functions in `R`, e.g., `lm()`, `sum()`, `print()`, etc.
- The primary components of an `R` function are the `body()`, `formals()`, and `environment()`. We often want to return the results from the function.
  - `body()`: the code inside the function.
  - `formals()`: the input arguments.
  - `environment()`: where the variables in the function are located.
  - `return()`: return the relevant output

```
speak <- function(x) {
  Jedi <- c("yoda", "windu", "kenobi")
  Mandalorian <- c("satine", "sabine", "jango")

  if (x %in% Jedi) {
    say <- "May the force be with you!"
    membership <- "Jedi"
  } else if (x %in% Mandalorian) {
    say <- "This is the way!"
    membership <- "Mandalorian"
  } else {
    say <- "Hello, world!"
    membership <- "Not Jedi or Mandalorian"
  }
  return(list(say = say, membership = membership))
}

formals(speak)
```

```
## $x
```

```
body(speak)
```

```
## {
##     Jedi <- c("yoda", "windu", "kenobi")
##     Mandalorian <- c("satine", "sabine", "jango")
##     if (x %in% Jedi) {
##         say <- "May the force be with you!"
##         membership <- "Jedi"
##     }
##     else if (x %in% Mandalorian) {
##         say <- "This is the way!"
##         membership <- "Mandalorian"
##     }
##     else {
##         say <- "Hello, world!"
##         membership <- "Not Jedi or Mandalorian"
##     }
##     return(list(say = say, membership = membership))
## }
```

```
environment(speak)
```

```
## <environment: R_GlobalEnv>
```

- What is the result return from the function?

```
speak("jango")
```

```
## $say
## [1] "This is the way!"
##
## $membership
## [1] "Mandalorian"
```

- In this class, we might ask you to change some part of the code in a markov model function.

Question: How would you program the Matian population growth in a function? What are the input arguments? What are the results return from the function?

1.9 Plots

1.9.0.1 Histogram

- Showing the distribution of a variable

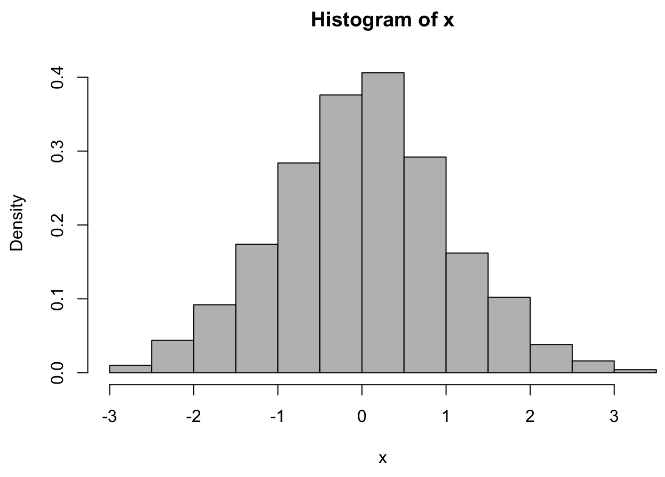
```
x <- rnorm(1000, 0, 1) # draw 1000 samples from a normal distribution
print(mean(x))
```

```
## [1] 0.02082678
```

```
print(sd(x))
```

```
## [1] 1.018553
```

```
hist(x, freq = F, col = "gray")
```



1.9.0.2 Scatter plots

- Inspect the correlation between two variables

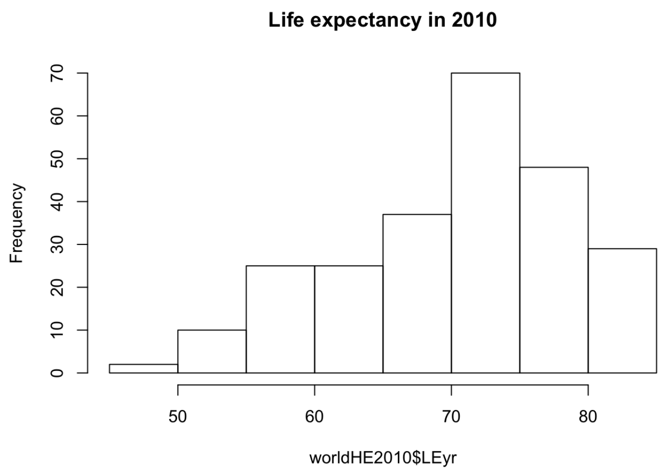
```
library(CEAutil)
data(worldHE)

print(head(worldHE))
```

```
##        Entity Code Year LEyr HealthExp     Pop
## 1 Afghanistan  AFG 1800   NA        NA 3280000
## 2 Afghanistan  AFG 1801   NA        NA 3280000
## 3 Afghanistan  AFG 1802   NA        NA 3280000
## 4 Afghanistan  AFG 1803   NA        NA 3280000
## 5 Afghanistan  AFG 1804   NA        NA 3280000
## 6 Afghanistan  AFG 1805   NA        NA 3280000
```

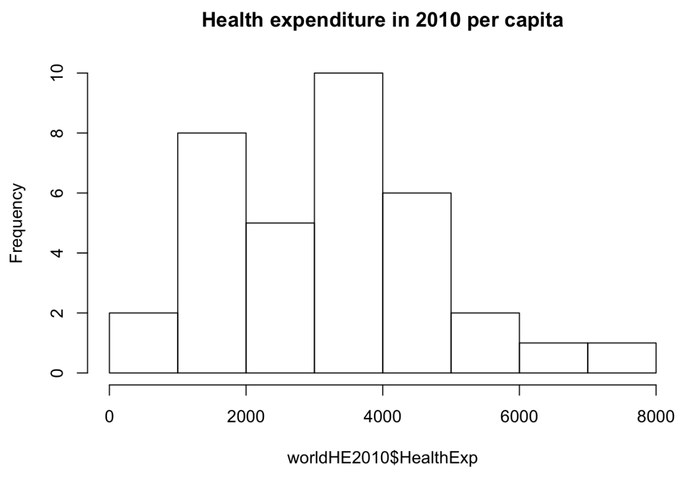
- Only focus on the data in 2010

```
worldHE2010 <- worldHE[worldHE$Year == 2010, ]
```

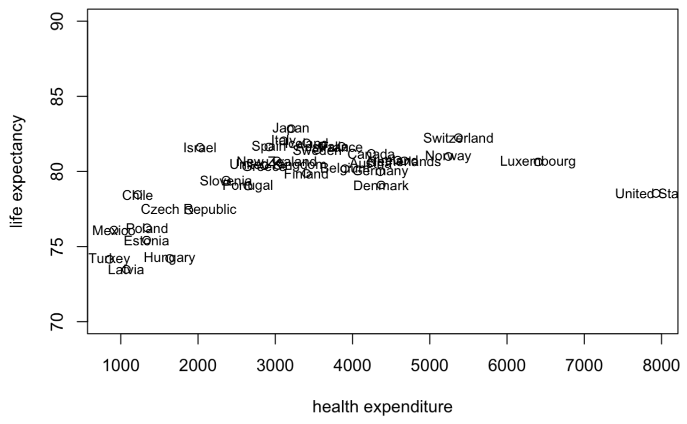
```
hist(worldHE2010$LEyr, main = "Life expectancy in 2010")
```



```
hist(worldHE2010$HealthExp, main = "Health expenditure in 2010 per capita")
```



```
plot(worldHE2010$HealthExp, worldHE2010$LEyr, ylim = c(70, 90),
     xlab = "health expenditure", ylab = "life expectancy")
text(worldHE2010$HealthExp, worldHE2010$LEyr, labels = worldHE2010$Entity, cex = 0.8)
```

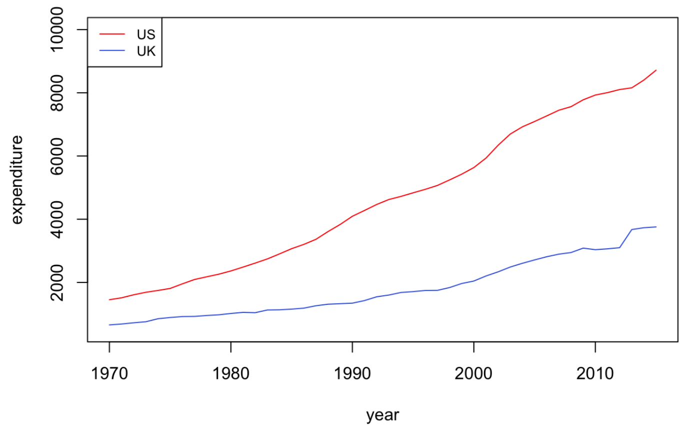


source

1.9.0.3 Line plots

- What is the trend of health expenditure in the US 1970-2015?

```
worldHE_US <- worldHE[worldHE$Entity == "United States" & worldHE$Year >= 1970 & worldHE$Year <= 201
5, ]
# adding lines: UK's expenditure at the same time period
worldHE_UK <- worldHE[worldHE$Entity == "United Kingdom" & worldHE$Year >= 1970 & worldHE$Year <= 20
15, ]

plot(worldHE_US$Year, worldHE_US$HealthExp, type = "l", col = "red",
     ylab = "expenditure", xlab = "year", ylim = c(500, 10000))
lines(worldHE_UK$Year, worldHE_UK$HealthExp, col = "royalblue")
legend("topleft", legend=c("US", "UK"),
       col=c("red", "royalblue"), lty = 1, cex = 0.8)
```



- There is a great package to help you make better plots! `ggplot2`

1.10 Useful packages in the class

- The packages that we will use in the class include: `ggplot2`, `dampack`, and `CEAutil`.
- Here are the code to install these packages.

```r
if(!require(ggplot2)) install.packages("ggplot2")
if(!require(devtools)) install.packages("devtools")
if(!require(remotes)) install.packages("remotes")
if(!require(dampack)) remotes::install_github("DARTH-git/dampack", dependencies = TRUE)
if(!require(CEAutil)) remotes::install_github("syzoekao/CEAutil", dependencies = TRUE)
```

2 Decision tree

2.1 Agenda

- Introduction of Amua
- Creating a decision tree using Amua
- Example
- Exporting Amua decision tree to R script
- Prepare the R decision tree from Amua for further analyses

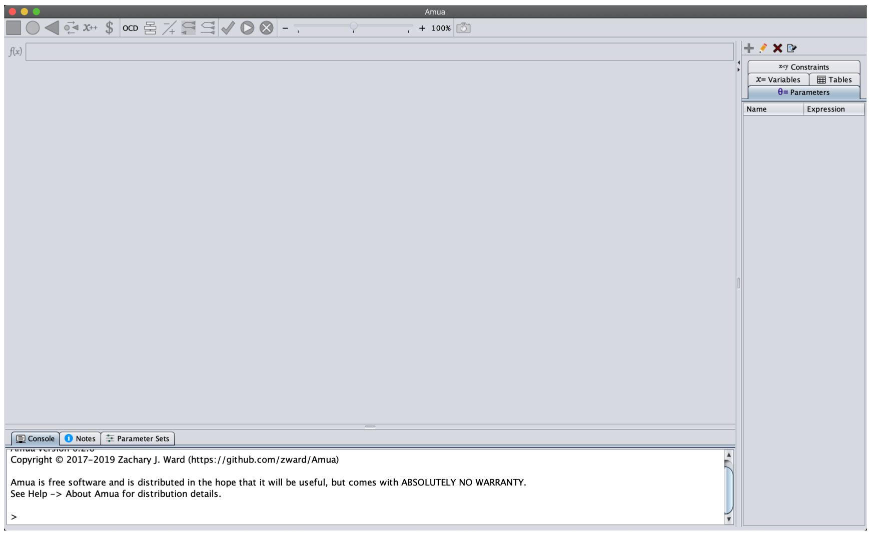
For a simple decision tree, we can draw the tree easily. As the decision tree grows, there is software helping you draw the tree such as TreeAge and Amua. In this class, we use Amua because it is free. We will show how to use Amua to build a decision tree and export the tree to R script for CEA analysis.

2.2 Install Amua

Follow the instruction here to install Amua. Be aware of the difference between Mac and Windows users! After you install Amua, remember where you store the software.

2.3 Create a decision tree

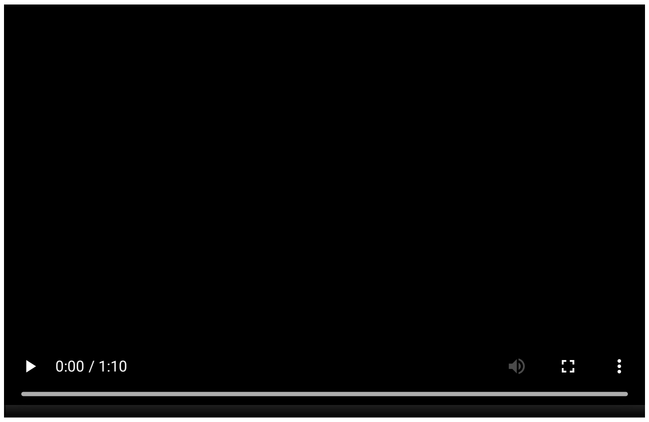
2.3.1 Open the `Amua.jar`.



```
Copyright © 2017–2019 Zachary J. Ward (https://github.com/zward/Amua)

Amua is free software and is distributed in the hope that it will be useful, but comes with ABSOLUTELY NO WARRANTY.
See Help -> About Amua for distribution details.

>
```

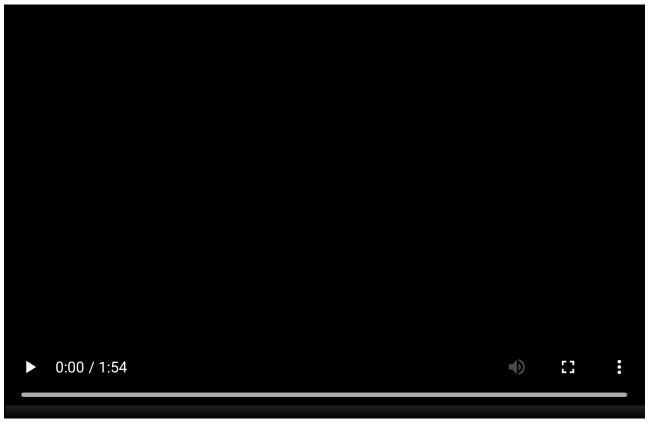
2.3.2 Decision, chance, terminal nodes, and decision tree

- Amua includes three type of nodes: decision, chance and terminal nodes in a decision tree.



2.3.3 Parameterization

- While you can hard code all the parameter values in each branch or terminal node, you could parameterize the input parameters (especially the parameters that might vary).



2.3.4 Adding more outcomes

- You could add multiple outcomes of interest at the terminal nodes.
- Sometimes we might be interested not only in cost but also in epidemiological outcomes

2.3.5 Change analysis

- Change your analysis type to cost-effectiveness analysis.
- Remember to select the cost and effectiveness outcomes.
- Also select the baseline strategy.

2.3.6 Save the model and export the tree to R code

- We then save the model and export the tree structure to R code.
- Notice that Amua created a folder, called `temp_Export/` .
- In this folder, there are two `.R` files: `main.R` and `functions.R` .
- The body of the decision tree model is in `main.R` .

2.4 Example

Dracula is preparing for a spring break party tonight. But there's one final decision that Dracula is struggling with. Being a vampire, he intends to bite and suck the blood of some of his guests at the party (about 25% of them, by his best estimate). While being bitten by a vampire won't turn his guests into vampires or zombies, like some horror movies might suggest, there is a 50% chance that a vampire bite results in a rather severe bacterial infection, of which 66% of cases require hospitalization for an average of 1 night.

Being a gracious host, Dracula is considering different ways of administering antibiotic prophylaxis to his guests to reduce their risk of infection. One option is to administer the antibiotics to his victim-guests just before he bites them – this would reduce the risk of a vampire bite infection by 20%. However, the antibiotics can be even more effective, reducing the risk of infection by 90%, if administered at least 30 minutes before being bitten. To achieve this, Dracula is considering putting the antibiotics into the drinks served at the party to ensure that his guests are all properly dosed before he bites his victims. But this means that all his guests would be exposed to the antibiotics (not just those he intends to bite), and he knows that about 5% of people are severely allergic to these antibiotics and would require immediate hospitalization if exposed. Dracula is therefore also considering not administering antibiotic prophylaxis at all to avoid this harm.

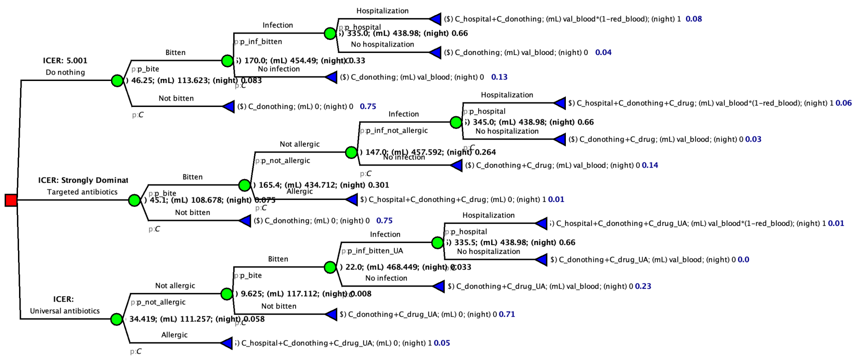
However, all the healthy blood doesn't come without cost. This party will cost Dracula $1000 for a total of 200 guests (an average of $5 per guest). In addition, Dracula expects the cost of antibiotic to be $10 for each guest he bites. If Dracula decides to administer the antibiotics in all the drinks served at the party, the total cost of antibioitics is expected to be $100 (Dracula gets discount for buying a large batch of antibiotics!). Also, Dracula is willing to pay for the cost of hospitalization for any guest who experience bacteiral infection due to his bites because he feels responsible. The cost of hospitalization per person per night is $500. Dracula expects to get an average of 470 mL healthy blood by biting a guest. However, if the guest ends up hospitalized due to bacterial infection, the healthy blood that Dracula can get is reduced by 10%. Dracula hopes that you can help him determine which strategy he should choose.

Think about the following quesitons:

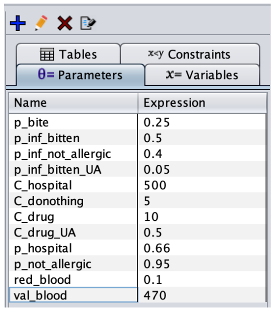
1. List out Dracula's strategies.
2. What are the potential outcomes for Dracula's party?
3. Draw the decision tree for Dracula's party.

You can build a decision tree via Amua.

- The tree structure

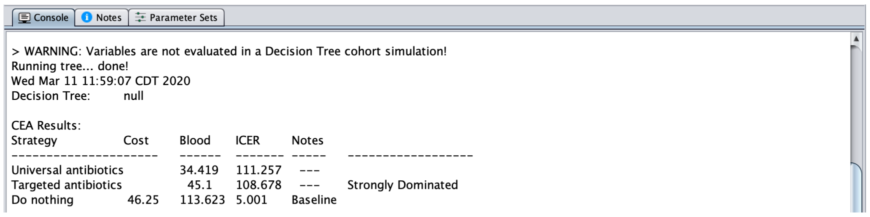

ICER: 5.001
Do nothing
Bitten
p:p_bite
) 46.25; (mL) 113.623; (night) 0.083
Infection
p:p_inf_bitten
) 170.0; (mL) 454.49; (night) 0.33
Hospitalization
p:p_hospital
) 335.0; (mL) 438.98; (night) 0.66
($) C_hospital+C_donothing; (mL) val_blood*(1-red_blood); (night) 1 0.08
No hospitalization
($) C_donothing; (mL) val_blood; (night) 0 0.04
No infection
($) C_donothing; (mL) val_blood; (night) 0 0.13
Not bitten
p:C
($) C_donothing; (mL) 0; (night) 0 0.75
ICER: Strongly Dominat
Targeted antibiotics
p:p_bite
) 45.1; (mL) 108.678; (night) 0.075
Not allergic
p:p_not_allergic
) 165.4; (mL) 434.712; (night) 0.301
p:p_inf_not_allergic
) 147.0; (mL) 457.592; (night) 0.264
) 345.0; (mL) 438.98; (night) 0.66
$) C_hospital+C_donothing+C_drug; (mL) val_blood*(1-red_blood); (night) 1 0.06
($) C_donothing+C_drug; (mL) val_blood; (night) 0 0.03
($) C_donothing+C_drug; (mL) val_blood; (night) 0 0.14
Allergic
$) C_hospital+C_donothing+C_drug; (mL) 0; (night) 1 0.01
($) C_donothing; (mL) 0; (night) 0 0.75
ICER:
Universal antibiotics
p:p_not_allergic
34.419; (mL) 111.257; (night) 0.058
) 9.625; (mL) 117.112; (night) 0.008
p:p_inf_bitten_UA
) 22.0; (mL) 468.449; (night) 0.033
) 335.5; (mL) 438.98; (night) 0.66
) C_hospital+C_donothing+C_drug_UA; (mL) val_blood*(1-red_blood); (night) 1 0.01
$) C_donothing+C_drug_UA; (mL) val_blood; (night) 0 0.0
$) C_donothing+C_drug_UA; (mL) val_blood; (night) 0 0.23
$) C_donothing+C_drug_UA; (mL) 0; (night) 0 0.71
) C_hospital+C_donothing+C_drug_UA; (mL) 0; (night) 1 0.05

- The parameters



Name	Expression
p_bite	0.25
p_inf_bitten	0.5
p_inf_not_allergic	0.4
p_inf_bitten_UA	0.05
C_hospital	500
C_donothing	5
C_drug	10
C_drug_UA	0.5
p_hospital	0.66
p_not_allergic	0.95
red_blood	0.1
val_blood	470

- The CEA outcomes



```
> WARNING: Variables are not evaluated in a Decision Tree cohort simulation!
Running tree... done!
Wed Mar 11 11:59:07 CDT 2020
Decision Tree:      null

CEA Results:
Strategy              Cost      Blood     ICER      Notes
---------------------           ------    -------   -----     ------------------
Universal antibiotics           34.419    111.257   ---
Targeted antibiotics            45.1      108.678   ---       Strongly Dominated
Do nothing            46.25     113.623   5.001     Baseline
```

2.5 Wrapper function for decision tree exported from Amua

- The decision tree created in Amua can be exported to R.
- However, the R script created by Amua is not easy to use for advance CEA analysis (e.g., PSA, one/two-way sensitivity analysis, EVPI, etc.).
- We created some wrapper functions to reshape the R script from Amua. This transformation of code allows us to do further analysis using other packages in R.
- Install the package using the following code.

```
if(!require("remotes")) install.packages("remotes")
if(!require("dplyr")) install.packages("dplyr")
if(!require("CEAutil")) remotes::install_github("syzoekao/CEAutil", dependencies = TRUE)
if(!require(dampack)) remotes::install_github("DARTH-git/dampack", dependencies = TRUE)

library(CEAutil)
```

- We will use two functions from the package to convert the R script exported from Amua.

1. `parse_amua_tree()` :

This function only takes an input argument, the path to the `main.R` file of the Amua decision model. The function returns a list of outputs. Output 1 `param_ls` is a list of input parameters with basecase values used in Amua. Output 2 `treefunc` is the R code of the Amua decision model formatted as text.

```
treetxt <- parse_amua_tree("AmuaExample/DraculaParty_Export/main.R")

print(treetxt$param_ls)
```

```
## $p_bite
## [1] 0.25
##
## $p_inf_bitten
## [1] 0.5
##
## $p_inf_not_allergic
## [1] 0.4
##
## $p_inf_bitten_UA
## [1] 0.05
##
## $C_hospital
## [1] 500
##
## $C_donothing
## [1] 5
##
## $C_drug
## [1] 10
##
```

```
## $C_drug_UA
## [1] 0.5
##
## $p_hospital
## [1] 0.66
##
## $p_not_allergic
## [1] 0.95
##
## $red_blood
## [1] 0.1
##
## $val_blood
## [1] 470
```

2. `dectree_wrapper()` :

We use the two outputs from the `parse_amua_tree()` as the input arguments in the `dectree_wrapper()`. The input arguments in the wrapper function include `params_basecase`, `treefunc`, and `popsize`. `params_basecase` takes a list of named input parameters. `treefunc` takes the text file organized by the `parse_amua_tree()`. `popsize` is defaulted as 1 but you could change your population size.

```
param_ls <- treetxt[["param_ls"]]
treefunc <- treetxt[["treefunc"]]
```

- Expected cost per person

```
tree_output <- dectree_wrapper(params_basecase = param_ls, treefunc = treefunc, popsize = 1)
print(tree_output)
```

```
##                strategy expectedBlood expectedHospitalization     Cost
## 1             Donothing      113.6225               0.0825000 46.25000
## 2  Targetedantibiotics      108.6781               0.0752000 45.10000
## 3 Universalantibiotics      111.2566               0.0578375 34.41875
```

- Expected cost of 200 guests

```
tree_output <- dectree_wrapper(params_basecase = param_ls, treefunc = treefunc, popsize = 200)
print(tree_output)
```

```
##                strategy expectedBlood expectedHospitalization    Cost
## 1             Donothing      22724.50                 16.5000 9250.00
## 2  Targetedantibiotics      21735.62                 15.0400 9020.00
## 3 Universalantibiotics      22251.33                 11.5675 6883.75
```

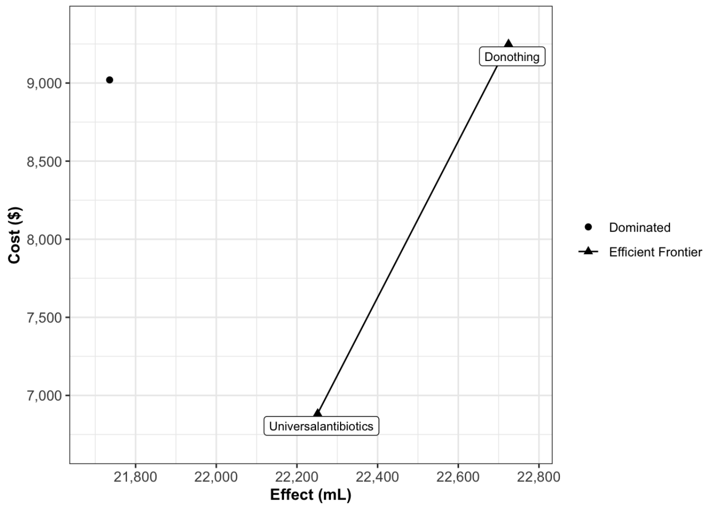
2.6 Calculate ICERs

```
library(dampack)
dracula_icer <- calculate_icers(tree_output$Cost,
                                tree_output$expectedBlood,
                                tree_output$strategy)
print(dracula_icer)
```

```
##                Strategy    Cost   Effect Inc_Cost Inc_Effect      ICER Status
## 1 Universalantibiotics 6883.75 22251.33       NA         NA        NA     ND
## 2             Donothing 9250.00 22724.50  2366.25   473.1725 5.000819     ND
## 3  Targetedantibiotics 9020.00 21735.62       NA         NA        NA      D
```

- Create an ICER plot

```
plot(dracula_icer, effect_units = "mL")
```



2.7 Change parameter values

- Many of the parameters are organized as a variable in the decision tree. We can easily see how the cost and effectiveness vary with changes of parameters of interest.

Example 1. Dracula has been starved over the long cruel winter in Minnesota. The Spring break is the first time that his guests are willing to come to his party in several months. Therefore, Dracula is going to seize the chance to bite as many guests as possible. The probability that he bites a guest is now increased to 50%. What are the cost and effectiveness of each strategy?

```
param_ls$p_bite <- 0.5

tree_output <- dectree_wrapper(params_basecase = param_ls, treefunc = treefunc, popsize = 200)

print(tree_output)
```

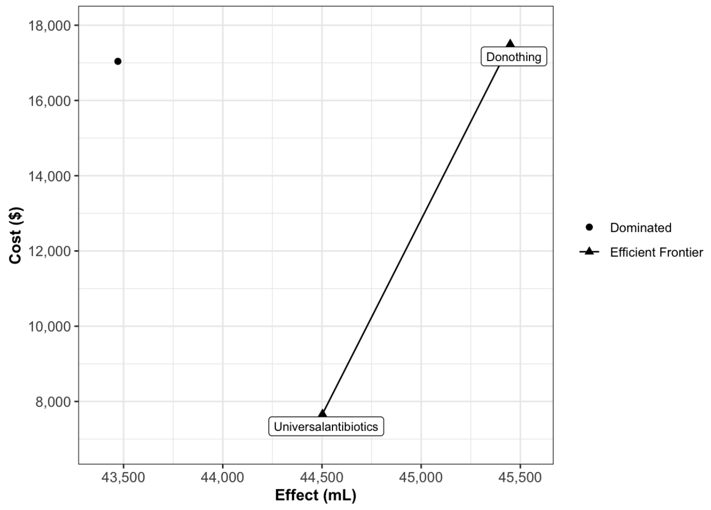
```
##               strategy expectedBlood expectedHospitalization    Cost
## 1            Donothing      45449.00                  33.000 17500.0
## 2  Targetedantibiotics      43471.24                  30.080 17040.0
## 3 Universalantibiotics      44502.65                  13.135  7667.5
```

- ICER & ICER plot

```
dracula_icer <- calculate_icers(tree_output$Cost,
                                tree_output$expectedBlood,
                                tree_output$strategy)
print(dracula_icer)
```

```
##               Strategy    Cost   Effect Inc_Cost Inc_Effect     ICER Status
## 1 Universalantibiotics  7667.5 44502.65       NA         NA       NA     ND
## 2            Donothing 17500.0 45449.00   9832.5    946.345 10.38997     ND
## 3  Targetedantibiotics 17040.0 43471.24       NA         NA       NA      D
```

```
plot(dracula_icer, effect_units = "mL")
```



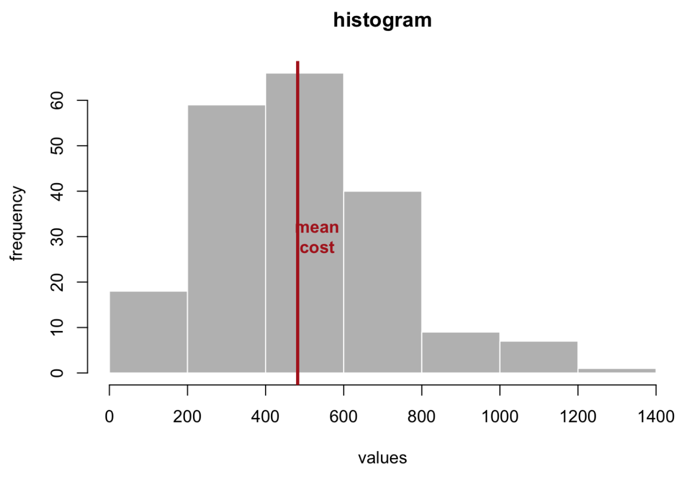
Example 2. The cost of hospitalization due to bacterial infection varies from guest to guest depending on the healthcare that a guest has. Overall, the cost of hospitalization has a mean of $500 with a standard deviation $300 following a gamma distribution. What are the mean cost and effectiveness of the party across all 200 guests?

- Generate 200 samples of the cost of hospitalization from gamma distribution

```
require(dampack)

C_hospital <- gen_psa_samp(params = c("C_hospital"),
                           dists = c("gamma"),
                           parameterization_types = c("mean, sd"),
                           dists_params = list(c(500, 250)),
                           nsamp = 200)

cat("mean and sd are", c(mean(C_hospital$C_hospital), sd(C_hospital$C_hospital)), ", respectively.")
```

```
## mean and sd are 482.1101 232.8722 , respectively.
```

```
hist(C_hospital$C_hospital, main = "histogram", xlab = "values", ylab = "frequency", col = "gray", b
order = F)
abline(v = mean(C_hospital$C_hospital), col = "firebrick", lwd = 3)
text(mean(C_hospital$C_hospital) + 50, 30, "mean\ncost", col = "firebrick", font = 2)
```



- Calculate the expected cost and effectiveness for each sample of cost

```
tree_vary <- dectree_wrapper(params_basecase = param_ls,
                             treefunc = treefunc,
                             vary_param_samp = C_hospital)
```

```
## Warning in dectree_wrapper(params_basecase = param_ls, treefunc = treefunc, :
## nsamp are not the parameters included in params_basecase. The function will
## ignore these parameters.
```

- Let’s take a look at some elements in the tree_vary output

```
print(names(tree_vary))
```

```
## [1] "expectedBlood"           "expectedHospitalization"
## [3] "Cost"                    "param_samp"
```

```
print(head(tree_vary$param_samp))
```

```
##   nsamp C_hospital
## 1     1   569.1042
## 2     2   163.4579
## 3     3   749.0782
## 4     4   367.1748
## 5     5   952.7950
## 6     6   598.1560
```

```
print(head(tree_vary$expectedBlood))
```

```
##   Donothing Targetedantibiotics Universalantibiotics
## 1   227.245            217.3562             222.5133
## 2   227.245            217.3562             222.5133
## 3   227.245            217.3562             222.5133
## 4   227.245            217.3562             222.5133
## 5   227.245            217.3562             222.5133
## 6   227.245            217.3562             222.5133
```

```
print(head(tree_vary$Cost))
```

```
##   Donothing Targetedantibiotics Universalantibiotics
## 1  98.90220            95.59328             42.87592
## 2  31.97056            34.58407             16.23510
## 3 128.59791           122.66137             54.69571
## 4  65.58384            65.22309             29.61421
## 5 162.21118           153.30037             68.07481
## 6 103.69575            99.96267             44.78390
```

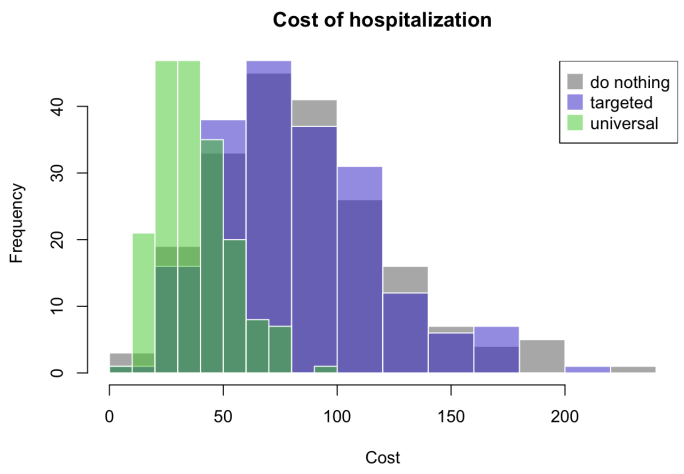
- The mean cost

```
print(summary(tree_vary$Cost))
```

```
##    Donothing       Targetedantibiotics Universalantibiotics
##  Min.   : 15.87   Min.   : 19.91      Min.   : 9.828     
##  1st Qu.: 57.42   1st Qu.: 57.78      1st Qu.:26.364     
##  Median : 79.94   Median : 78.31      Median :35.327     
##  Mean   : 84.55   Mean   : 82.51      Mean   :37.163     
##  3rd Qu.:106.84   3rd Qu.:102.83      3rd Qu.:46.036     
##  Max.   :227.49   Max.   :212.80      Max.   :94.056     
```

- How the expected cost differs between strategy?

```
hist(tree_vary$Cost$Donothing, col = rgb(0.5, 0.5, 0.5, 0.6), border = F,
     main = c("Cost of hospitalization"),
     xlab = "Cost")
hist(tree_vary$Cost$Targetedantibiotics, col = rgb(0.2, 0.2, 0.8, 0.5), border = F, add = T)
hist(tree_vary$Cost$Universalantibiotics, col = rgb(0.3, 0.8, 0.2, 0.5), border = F, add = T)
legend("topright", c("do nothing", "targeted", "universal"),
       col = c(rgb(0.5, 0.5, 0.5, 0.6), rgb(0.2, 0.2, 0.8, 0.5), rgb(0.3, 0.8, 0.2, 0.5)),
       pch = 15, pt.cex = 2)
```



3 Markov model

3.1 Agenda

- General Markov model coding structure
- Follow the coding steps using an example
- The template of Markov model function
- Consideration of strategies
- Prepare your Markov model for further analyses

3.2 General Markov model coding structure

- We will show how to code a Markov model step by step.
- In general, we follow this structure:

```
markov_model <- function(l_param_all,
                         strategy = NULL) {
  #### 1. Read in, define, or transform parameters if needed

  #### 2. Create the transition probability matrices using array

  #### 3. Create the trace matrix to track the changes in the population distribution
  #### through time. You could also create other matrix to track different outcomes,
  #### e.g., costs, incidence, etc.

  #### 4. Get outputs

  #### 5. Return the relevant results
}
```

- Then we write this process into a function `markov_model()`

3.3 Example

The Canadian province of Ontario is considering a province-wide ban on indoor tanning as a means of preventing skin cancer, with a focus on young women. The Ontario Ministry of Health has just finished a large observational study on tanning behaviors and skin cancer incidence among women in Ontario to inform their decision.

In their surveillance study, they found that skin cancer risks differ substantially for individuals who visit tanning salons regularly ("regular tanners") and those who do not ("non-tanners"). The annual risk of skin cancer was 4% for regular tanners vs. 0.5% for non-tanners. (Assume that skin cancer risk depends only on current tanning behavior and not on tanning history).

The Ministry of Health also studied tanning behaviors. They estimated the annual probability of a non-tanner becoming a regular tanner, by age, among women. This probability begins increasing around age 12, peaks at age 24 and then begins to decrease. They also estimated the rate of a regular tanner becoming a non-tanner. This probability is low until age 30, after which it increases with age. These data are summarized in the dataset `ONtan` in the `CEAutil` package.

Skin cancer resolves within one year of diagnosis, with 7% of cases resulting in death. Those who recover following a skin cancer diagnosis nearly all quit tanning, though they re-initiate indoor tanning at the same rate as their peers who have not experienced skin cancer.

The Ontario Ministry of Health is considering an indoor tanning ban for those 18 years of age and younger (reducing the rate of tanning initiation to zero among this age group). They are also considering a full indoor tanning ban, which would reduce the rate of tanning initiation to zero for everyone in Ontario.

However, indoor tanning ban would result in reduced demand for indoor tanning and decrease the number of employees hired in these facilities (there are currently about 1000 employees in the industry). Indoor tanning ban among women younger than 18 year-old would reduce the employmenet by 10% in the industry. A full indoor tanning ban would reduce the employmenet by 80%. We assume that the average annual salary of an employee is 28,000 Canadian dollars. The Ontario Ministry of Health would like to conduct cost-effectiveness analysis for the health benefit over the lifetime of a cohort of 10-year-old girls by considering the cost. Because the outcome is life expectancy, the Ministry of Health does not wish to discount outcomes. (However, if the outcome is QALY, we would discount the outcome at 5% per year for Canadians!)

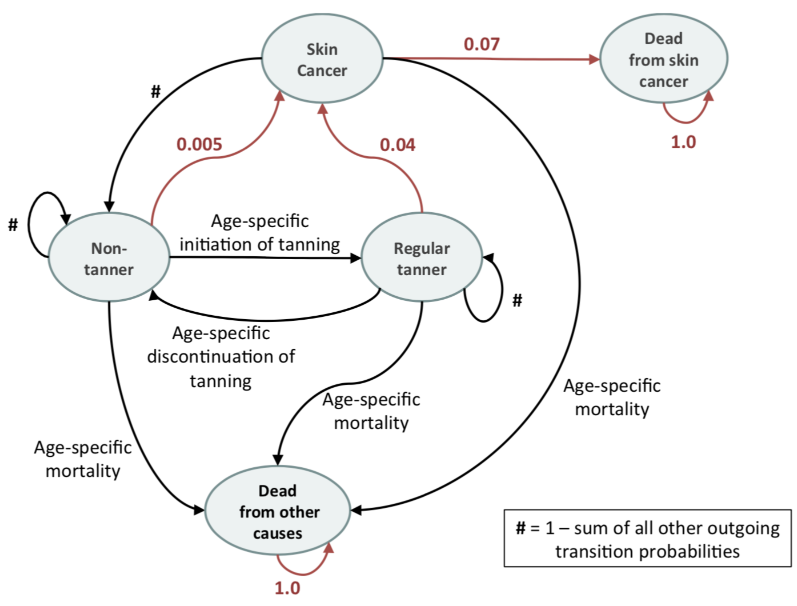
1. Draw a Markov model diagram of tanning behavior and skin cancer that the Ontario Ministry of Health could use to evaluate their tanning policies in terms of the desired outcomes. Use a yearly time step. Include the following states: “Non-tanners”, “Regular tanners”, “Skin cancer”, “Dead from skin cancer”, and “Dead from other causes”. Label all transition probabilities and indicate which are changing over time.
2. Calcuate the remaining life-expectancy of a 10-year-old girl under each strategy.

3.3.1 Step 0. Model setup

Before doing any actual coding:

1. Draw your Markov model diagram

Constant probabilities are shown in red – all others change over time as the cohort ages.



2. Figure out your parameters
   - Which are fixed?
   - Which are time varying?
   - Do you have parameters that require calibration?
     - This is not relevant to our example here but it might happen in your other projects.
3. What is the time horizon?
   - 100 years
4. What is the cycle length?
   - annual time step

3.3.2 Step 1. Set up parameters, data, and variables required

- We start with setting up parameter values, function, or transformation.
- Provide your parameter table:

parameter code	definition	data type
p_init_tan	probability of initiating tanning	table
p_halt_tan	probability of discontinuing tanning	table
p_nontan_to_cancer	probability of having skin cancer among non-tanners	constant
p_regtan_to_cancer	probability of having skin cancer among regular tanners	constant
p_cancer_to_dead	probability of cancer-specific death	constant
p_mort	probability of natural death	table
n_worker	number of workers	constant
salary	average salary of tanning workers	constant
target_red	reduction of workers under targeted ban	constant
universal_red	reduction of workers under universal ban	constant

- You could find the tanning behavior and the life table in the `CEAutil` package.

```
library(CEAutil)

data(ONtan)
ltable <- ONtan$lifetable
behavior <- ONtan$behavior

print(head(ltable))
```

```
##   age      qx
## 1   0 0.00468
## 2   1 0.00017
## 3   2 0.00014
## 4   3 0.00012
## 5   4 0.00011
## 6   5 0.00009
```

```
print(head(behavior))
```

```
##   age p_init_tanning p_halt_tanning
## 1  10           0.05           0.01
## 2  11           0.05           0.01
## 3  12           0.10           0.01
## 4  13           0.10           0.01
## 5  14           0.11           0.01
## 6  15           0.12           0.01
```

- For the constant parameters:

```
p_nontan_to_cancer <- 0.005
p_regtan_to_cancer <- 0.04
p_cancer_to_dead <- 0.07
n_worker <- 1000
salary <- 28000
target_red <- 0.1
universal_red <- 0.8
```

- In addition, we have other required variables
  - timehorizon (`n_t`)
  - names of each health state (`state_names`)
  - initial status (`v_init`)

```
n_t <- 100
state_names <- c("nontan", "regtan", "cancer", "deadnature", "deadcancer")
v_init <- c(1, 0, 0, 0, 0)
```

- These input parameters are often defined **outside** the `markov_model()` function. In this model framework, we use `list()` to combine all the parameters, data, and variables defined beforehand. We provide the name for each the element in the list.

```
l_param_all <- list(p_nontan_to_cancer = 0.005,
                    p_regtan_to_cancer = 0.04,
                    p_cancer_to_dead = 0.07,
                    n_worker = 1000,
                    salary = 28000,
                    target_red = 0.1,
                    universal_red = 0.8,
                    ltable = ltable,
                    behavior = behavior,
                    state_names = c("nontan", "regtan", "cancer", "deadnature", "deadcancer"),
                    n_t = 100,
                    v_init = c(1, 0, 0, 0, 0))
```

- In the function, we start with some basic calculation and transformation of the input parameters, data, and variables that will be used in the Markov model.

```
#### 1. Read in, set, or transform parameters if needed
# We need the age index for matching values of the lifetable and behavior data.
ages <- c(10 : (10 + n_t - 1))

# We extract the probability of natural death age 10-110 from the lifetable.
p_mort <- ltable$qx[match(ages, ltable$age)]

# We modify the values of tanning behavior based on strategy of interest.
if (strategy == "targeted_ban") {
  behavior$p_init_tanning[behavior$age <= 18] <- 0
}
if (strategy == "universal_ban") {
  behavior$p_init_tanning <- 0
}

# We extract the tanning behavior at age 10-110 from the behavior data
p_init_tan <- behavior$p_init_tanning[match(ages, behavior$age)]
p_halt_tan <- behavior$p_halt_tanning[match(ages, behavior$age)]

# We get the number of health states based on the length of the string vector, state_names
n_states <- length(state_names)
```

3.3.3 Step 2. Create the transition probability array

- A transition probability matrix contains the transition probability from one health state to another state.

```
##        state1 state2 state3
## state1    0.1    0.2    0.7
## state2    0.5    0.1    0.4
## state3    0.7    0.0    0.3
```

- In many of our applications, individuals transit from the row state to the column state. Each row should sum up to 1.

```
print(rowSums(transit_matrix))
```

```
## state1 state2 state3
##      1      1      1
```

- The transition probability array can contain more than one transition matrix. Because the behavior and mortality vary over time. The transition matrix is different from year to year. Thus, we create a transition array with the following dimension $n\_state \times n\_state \times n\_$

```
tr_mat <- array(0, dim = c(n_states, n_states, n_t),
                dimnames = list(state_names, state_names, ages))
```

- We initiate the transition array with 0 because most transition probabilities are zeros.
- Then we modify the transition probabilities for each origin state.

```
# 1. Fill out the transition probabilities from non-tanner to other states
tr_mat["nontan", "regtan", ] <- p_init_tan
tr_mat["nontan", "cancer", ] <- p_nontan_to_cancer
tr_mat["nontan", "deadnature", ] <- p_mort
tr_mat["nontan", "nontan", ] <- 1 - p_init_tan - p_nontan_to_cancer - p_mort

# 2. Fill out the transition probabilities from regular tanner to other states
tr_mat["regtan", "nontan", ] <- p_halt_tan
tr_mat["regtan", "cancer", ] <- p_regtan_to_cancer
tr_mat["regtan", "deadnature", ] <- p_mort
tr_mat["regtan", "regtan", ] <- 1 - p_halt_tan - p_regtan_to_cancer - p_mort

# 3. Fill out the transition probabilities from skin cancer to other states.
#    Be careful that this is a tunnel state. Therefore, there is no self loop.
tr_mat["cancer", "deadcancer", ] <- p_cancer_to_dead
tr_mat["cancer", "deadnature", ] <- p_mort
tr_mat["cancer", "nontan", ] <- 1 - p_cancer_to_dead - p_mort

# 4. Fill out the transition probabilities for cancer specific death (this is an absorbing state!!)
tr_mat["deadcancer", "deadcancer", ] <- 1

# 5. Fill out the transition probabilities for natural death (this is an absorbing state!!)
tr_mat["deadnature", "deadnature", ] <- 1
```

- Let's see two slice of the transition array

```
print(tr_mat[, , "20"])
```

```
##              nontan  regtan cancer deadnature deadcancer
## nontan     0.82477 0.17000  0.005    0.00023       0.00
## regtan     0.01000 0.94977  0.040    0.00023       0.00
## cancer     0.92977 0.00000  0.000    0.00023       0.07
## deadnature 0.00000 0.00000  0.000    1.00000       0.00
## deadcancer 0.00000 0.00000  0.000    0.00000       1.00
```

```
print(tr_mat[, , "50"])
```

```
##              nontan  regtan cancer deadnature deadcancer
## nontan     0.98303 0.01000  0.005    0.00197       0.00
## regtan     0.50000 0.45803  0.040    0.00197       0.00
## cancer     0.92803 0.00000  0.000    0.00197       0.07
## deadnature 0.00000 0.00000  0.000    1.00000       0.00
## deadcancer 0.00000 0.00000  0.000    0.00000       1.00
```

- Check whether the transition array meets the criteria of transition matrix

```
# Check whether the transition matrices have any negative values or values > 1!!!
if (any(tr_mat > 1 | tr_mat < 0)) stop("there are invalid transition probabilities")

# Check whether each row of a transition matrix sum up to 1!!!
if (any(round(apply(tr_mat, 3, rowSums), 5) != 1)) stop("transition probabilities do not sum up to o
ne")
```

3.3.4 Step 3. Create the trace matrix and outcome matrix

- Initialize the trace matrix and replace the first row with the initial state (`v_init`)

```
#### 3. Create the trace matrix to track the changes in the population distribution through time
#### You could also create other matrix to track different outcomes,
#### e.g., costs, incidence, etc.
trace_mat <- matrix(0, ncol = n_states, nrow = n_t + 1,
                    dimnames = list(c(10 : (10 + n_t)), state_names))
# Modify the first row of the trace_mat using the v_init
trace_mat[1, ] <- v_init
```

- If you are interested in tracking other outcomes, initialize an empty matrix here.

```
# Suppose that we want to track the cost of having skin cancer for a year
trace_cost <- rep(0, n_t)
```

3.3.5 Step 4. Compute the Markov model over time

- We use `for()` loop to iterate the calculation through time.

```
#### 4. Compute the Markov model over time by iterating through time steps
for(t in 1 : n_t){
  trace_mat[t + 1, ] <- trace_mat[t, ] %*% tr_mat[, , t]
}
```

3.3.6 Step 5. Organize outputs

- The outcome in this example is “expected life years” (`LE`) and total cost due to laying off workers in the industry.
    - Calculate the cost associated with each strategy