Intro to R for Decision Modeling

Introduction

SickKids and DARTH

11/2/2020

Change `eval` to `TRUE` if you want to knit this document.This worksheet provides an introduction to some key concepts in `R`. This session is split into the following sections:

1. Using `R` as a calculator
2. Assigning values in `R`
3. Creating vectors and matrices
4. Manipulating Matrices in `R`
5. Actions in `R`
6. Customized tools in `R`

Throughout the course, we will demonstrate code and leave some empty *code chunks* for you to fill in. We will also provide solutions after the session.

Feel free to modify this document with your own comments and clarifications.

1. Using `R` as a calculator

Although `R` has lots of capabilities, we will start by using it as a calculator.

EXERCISE 1 Type `2 + 3` in the *code chunk* below and run. Remember that the code chunk can be run using the green arrow at the right hand corner of the chunk or using the keyboard shortcut CTRL + SHIFT + ENTER.

```
# Your turn
2 + 3
```

You should now see that the calculation `2 + 3` has been completed as the output is shown in two places. In the console below and in the R Markdown document directly above.

EXERCISE 2 Perform a few more calculations in the *code chunk* below. You can use the following commands:

- `-`: subtract
- `*`: multiply
- `/`: divide
- `^2`: squared

```
# Your turn
5^5
12/4
(12/4)^2
(51-4)*9
```

2. Assigning values in R

R performs data analysis by applying actions (known as *functions*) to *objects*. One example of an *object* is a dataset that we load into R. Any function you wish to apply (e.g. finding the mean for each variable in your dataset) will be an action on a particular object (the dataset in this case).

We define an object in R using the symbol <-. So for example,

```
# Defining a
a <- 5
```

means that we are defining an object called a and it is defined as being equal to 5. R is case-sensitive so a and A are different objects in R, if both defined.

EXERCISE 3 Define two objects alpha equal to 7 and beta equal to 2 in the *code chunk* below and run. Note that each object needs to be defined on a separate line within the *code chunk*.

```
# Your turn
alpha <- 7
beta <- 2
```

For the *code chunk* above, notice that there was no output in the R Markdown document. This is because we were only defining alpha and beta and not asking R to do any calculations. If you look in the console, you will see the two lines of code you just run are in the console so we know that R has defined these objects.

As R now knows what we mean by alpha and beta, we can use R to apply functions to these objects. The calculations we use in Section 1 are examples of functions, e.g. run the following *code chunk*

```
# Addition
alpha + beta
```

EXERCISE 4 Use the commands from the first section to undertake some additional calculations with alpha and beta in the following *code chunk* and run.

```
# Your turn
alpha - beta
alpha * beta
alpha / beta
alpha ^ 2
```

3. Types of data in R

An object doesn't have to be a number, common R objects include numbers (as we defined above), a character string, a dataset, a vector, a matrix, an array and a plot.

Vectors are a set of numbers with a specific set of properties. To define a vector, we pass a list of numbers to the `c()` function in `R`. For example, the following *code chunk* defines a vector that contains the ages for three different people:

```
# Defining a vector
ages <- c(12, 63, 27)
```

EXERCISE 5 In the *code chunk* below, define `vector1` that contains the numbers 3, 5 and 1 and `vector2` that contains the numbers -1, 8, and -3. Run this *code chunk*.

```
# Your turn
vector1 <- c(3, 5, 1)
vector2 <- c(-1, 8, -3)
```

Character strings contain words, rather than numbers. To ensure that `R` knows you are using a character string, you need to use `""`, for example,

```
# Defining a character string
fruit <- "apple"
```

You can also combine character strings into a vector.

EXERCISE 6 In the *code chunk* below, define a vector `fruits` that contains the strings "apple", "banana" and "orange". Run this *code chunk*.

```
# Your turn
fruits <- c("apple", "banana", "orange")
```

Another important object is a matrix, which is a set of numbers or characters with a specific shape. For example, the following *code chunk* creates a matrix called `matrixA` that contains the numbers 1 to 9 in a 3 by 3 grid:

```
# Creating a matrix
matrixA <- matrix(data = c(1, 2, 3, 4, 5, 6, 7, 8, 9),
                  nrow = 3,
                  ncol = 3)
matrixA
```

In this *code chunk*, we can see that the matrix `matrixA` is printed in this R Markdown document above. This is because we added the extra command `matrixA` to our *code chunk*, this allows us to see what the object `matrixA` is.

EXERCISE 7 Use the following *code chunk* to create a matrix `matrixB` with 2 rows and 4 columns and a matrix `matrixC` with 4 rows and 2 columns:

```
# Your turn

matrixB <- matrix(data = c(1, 2, 3, 4, 5, 6, 7, 8),
                  nrow = 2,
                  ncol = 4)
matrixB

matrixC <- matrix(data = c(1, 2, 3, 4, 5, 6, 7, 8),
                  nrow = 4,
                  ncol = 2)
matrixC
```

We can also use functions on vectors and matrices, for example, if we use `vector1` and `vector2` that we defined above:

```
# Functions on vectors
vector1 + vector2
```

EXERCISE 8 Explore what happens when you use the commands from section 1 on vectors in the *code chunk* below:

```
# Your turn
```

What happens if we change the definition of `vector1`?

EXERCISE 9 Run the following *code chunk* to change the definition of `vector1`:

```
# Your turn
vector1 <- c(3, 5, 2)
```

The top right hand window will show the defintion of `vector1`. Note that if you overwrite the definition of an object, you cannot retrieve the previous defintion of the object unless you rerun the initial code.

You can check the definition of an object by typing the name of an object into a *code chunk* and running that line of code. This prints the object in the console and in the R Markdown so you can check it is defined correctly. You can also check the defintion in the top right-hand window that displays you *workspace*.

4. Manipulating Matrices in `R`

In `R`, there are three key operations that we can do with matrices, addition (`+`), multiplication (`*`) and matrix multiplication (`%*%`). Each of these operations can only be used if the matrices are compatible.

EXERCISE 10 Run the following operations to determine which matrices or vectors are compatible for matrix addition. Note that some of them will give you errors, make a note of these errors and why they occur to help you understand issues in you coding during the course.

```
# Matrix added to a number
matrixA + 5

# Matrix added to a vector
matrixA + vector1
matrixB + vector1

# Matrix addition
matrixA + matrixA
matrixB + matrixC
```

Notice that when you are adding a vector to a matrix, the addition is done column-wise where the vector is added to each column of the matrix. When adding matrices, the addition is performed element-wise, so the two matrix entries in the same position are added together. Scalar multiplcation `*` is similar to addition.

EXERCISE 11 Copying the code above, or writing your own, explore how scalar multiplication works with matrices and vectors:

```r
# Matrix multipled by a number
matrixA * 5

# Matrix multipled by a vector
matrixA * vector1
matrixB * vector1

# Matrix with scalar multiplication
matrixA * matrixA
matrixB * matrixC
```

Matrix multiplication **%*%**, on the other hand, follows a different set of rules, outlined in the pre-course material.

EXERCISE 12 Run the following code to explore how matrix multiplication works in R:

```r
# Matrix Multiplication
matrixA %*% vector1
matrixA %*% matrixA
matrixB %*% matrixC
matrixC %*% matrixB
```

Notice that the order of the matrices matters for matrix multiplication as the last two operations give different results. It is also important to remember that matrix multiplication can only be undertaken if the number of columns in the first matrix is equal to the number of rows in the second matrix. The resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.

5. Actions in R

Functions in R are used to perform actions on the different objects.

We have already seen some simple actions; +, *, - and / — these are calculator or arithmetic actions.

More complex functions are given in two parts.

- The name of the function, e.g. `c`
- The object to be acted on in brackets, e.g. `(alpha)`

Within the brackets function specific options (formally called "arguments") can also be given after commas, e.g. `(alpha, beta)`

Some examples of function in R include:

- `log()`
- `sum()`
- `prod()`
- `exp()`
- `sort()`

EXERCISE 13 Use `vector1`, `vector2` and `matrixA` to investigate how these different functions are used for vectors and matrices. You can use the *code chunk* below:

```
# Your turn
vector1
log(vector1)
prod(vector1)
vector2
sort(vector2)
sum(vector2)
matrixA
exp(matrixA)
```

Remember, you can run a single line in a *code chunk* by highlighting that line using CTRL + ENTER.

- To access help for a function you can type `?function` or `help(function)` into the `R` console, where `function` is the name of the function you need help for.
- Some *functions* can only be used for certain *objects* and in the future sessions we will see some more complex *functions* that can be used to manipulate our data.
- Many *functions* can take multiple arguments; simply separate them by `,` in your function call.

EXERCISE 14 Open up and read the help document for the function `sum()`.

There are some key functions that are very helpful when working with matrices.

- `t()` creates the matrix transpose, i.e., the matrix where the element in the i-th row and j-th column of the original matrix is set to the j-th row and i-th column.
- `diag(n)` creates a matrix of size `n` by `n` with 1s at each diagonal element and 0s everywhere else, i.e., the identity matrix.

EXERCISE 15 Explore the transponse function using the code provided below:

```
matrixD <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)

# Determine the difference between matrixC and the transpose of matrixC
matrixC
t(matrixC)

# Transpose can allow for matrix multiplication when they may not have been
# compatible before
matrixD %*% t(matrixC)
matrixC %*% matrixD
t(matrixB) %*% matrixD
matrixD %*% matrixB
```

5. Customized tools in `R`

Packages give `R` extra capabilities as they contain additional *functions*. There are over 8000 additional packages in `R` that can perform a huge range of additional *functions*.

Packages can be thought as customized tools that anyone can use that have been developed by people who use `R`. This is one of the many things that make `R` so great!

Before we can use all the *functions* in a package, we need to load those functions in `R` using the command `library`. For example:

```
library(dplyr)
```

allows you to use all the *functions* saved in the `dplyr` package — these functions are very useful for data manipulation.

The first time you use a package it may be necessary to download the package to your RStudio. This is done using the command:

```
install.packages("dplyr")
```

Note the quotation marks `"` that surround the name of the package.

EXERCISE 16 Install `R` package *dplyr*. Usually, the command to install a package is run from the console as running it from *code chunks* can cause problems.