SIT718 Real World Analytics

Week 3 Workshop

This week we will start working with whole matrices of data and learn how to implement transformations and visualize the data.

This practical includes the exercises in this documents based upon the data in the CSV files - **volley.txt** and **wontwork.txt** included in the **SIT718.1 Prac 3 - data.zip** archive.

We've also included two additional files in the data archive for you to apply these exercises to:

- **comp_time.txt**
- **wait_service.txt**

Replacing values

We learnt how to assign vectors and values in the previous topic. Sometimes we just want to change one entry. For changing the value of one entry in a vector, we can use the assign command and indicate the index in square brackets, e.g. to change the 5th entry in a vector a we would enter

```
a[5] <- 10
```

We can also replace multiple entries at once

```
a[c(5,7)] <- c(10,3)
a[3:5] <- c(1,0,1)
```

> **R Exercise 1** *Create the vector and change the entries.*
>
> ```
> a <- array(0,20)
> a[5] <- 1
> a[c(3,7,11)] <- c(2,6,1)
> a[17:20] <- c(1,2,1,4)
> ```
>
> *Your expected output when you type* a *and press enter should now be*
>
> ```
> 0 0 2 0 1 0 6 0 0 0 1 0 0 0 0 0 1 2 1 4
> ```

Arrays and matrices

Sometimes we will need to consider whole datasets at a time. For this, rather than just vectors, we will need rows of vectors, or matrices and arrays.

We can build them step by step with the cbind (column bind) and rbind (row bind) functions.

```
a <- cbind(c(2,3,5,1,0), c(6,1,8,2,9))
b <- rbind(c(4,1), c(2,-2),c(5,6))
```

R Exercise 2 *Assign the vectors and perform the* `cbind()` *and* `rbind` *operations.*

```
a <- c(1,2,3,7,9)
a <- cbind(a, c(21,2,1,5,6))
a <- cbind(a, c(2,-1,5,0,-1))
a <- cbind(a,c(1,9,7,2,1),array(6,5))

b <-c(3,6,1,9,2)
b <- rbind(b, c(3,2,1,8,9))
b <- rbind(b, c(4,1,12,1,2))
```

Note that for vectors like `c(3,2,1,8,9)`, R doesn't treat it as either a row or a column when it's by itself and so it is flexible when it comes to combining vectors either as rows or as columns. However once we have a matrix, the `rbind` and `cbind` operations will come up with a warning if the row length is not the same - however it will still merge them (it just fills the remaining space by repeating the sequence of the row).

We can also create large $m \times n$ arrays and then input values later. We will use the array function for this.

Side Note 2.5 *Arrays in* R *differ to matrices in that matrices must have numerical inputs. This can sometimes cause problems when we import our data from somewhere else (see read.table below), however for us the array function will be the most straightforward to use at this time.*

As with the array function previously, the first entry is the default value that will populate the array, however this time we will use a vector `c(rows,columns)` to indicate how many rows and how many columns there needs to be. A matrix/array with 3 rows and 4 columns (prepopulated with 0s) would be

```
A <- array(0,c(3,4))
```

Side Note 2.6 *We also can have higher dimension arrays, however for the moment we will stick to two.*

Once we've created our array with default values, we can then proceed to fill it. We refer to the cells using `A[row,column]` so that the entry in the first row and the second column would be `A[1,2]`. We can also consider entire columns, e.g. the second using `A[,2]` and entire rows using `A[1,]`. Note that this corresponds with our $\mathbf{x}_j$ and $x_{i,j}$ notation.

R Exercise 3 *Create a 3×4 array and then assign values to different entries using the following.*

```
A[3,1] <- 4
A[1,] <- c(1,2,3,4)
A[,2] <- c(6,5,4)
A[2:3,3:4] <- array(-1,c(2,2))
```

Your final matrix should appear as

```
     [,1] [,2] [,3] [,4]
[1,]    1    6    3    4
[2,]    0    5   -1   -1
[3,]    4    4   -1   -1
```

Reading a table

Often we will have data available from some other source, e.g. as an excel spreadsheet. We can import this data using the `read.table()` function. The easiest way is to have the data in a txt or csv file. We need to know whether the entries are separated by commas or spaces and whether they have labels. In the simplest case, we can save the table to an array. If the file is in the same folder as our R working directory[1], we can use the command

```
A <- read.table("thedata.csv")
```

[1] In R studio, the working directory can be set using the Session→Set Working Directory option from the menu. In the standard R application on a Mac, the option to change the working directory is under Misc.

If the data has headers (i.e. the first row is not entries but data labels), or is separated by commas, then we can add extra options.

```
A <- read.table("thedata.csv",header=TRUE,sep=",")
```

It is easiest if our data is already numerical data. In some cases, the data can be coded as text even if it is numbers. If columns in the data are numeric and R has trouble interpreting them as numbers (e.g. it says NA when you try to add 2), they can be extracted from the table using the `as.numeric()` command. The following exercise uses the `write.table()` command as well.

R Exercise 4 *Use the following to create and write a table to the working directory*

```
write.table(cbind(c("a",1,2),c(3,2,5)),"wontwork.txt")
```

Now read the table from the file and assign it to `my.table` *using the* `read.table()` *command.*

```
my.table <- read.table("wontwork.txt")
```

See what happens when you input the following.

Input	*Expected Output*

`my.table`	`  V1 V2`
	`1  a  3`
	`2  1  2`
	`3  2  5`

`my.table+2`	`   V1 V2`
	`1  NA  5`
	`2  NA  4`
	`3  NA  7`

`my.table[2,]+2`	`   V1 V2` `2  NA  4`
`as.numeric(my.table[2,])+2`	`3  4`

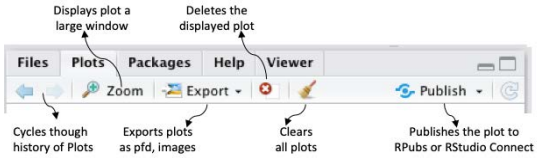
* *You will get an error message if you use R version 4.0.2.*

Data Visualization

In this section, we will learn to build simple plots and about the different aspects of formating the plots in R Studio. We will be using basic built-in graphics system in R, known as the **base** graphics system.

Plotting using R Studio

R studio displays the plots in the build-in window.



Plot Types and Plot fuctions

`plot()` - creates scatter plots
`barplot()` - creates bar plots
`hist()` - creates histograms & frequency diagrams
`boxplot()` - creates box plots
`pie()` - creates pie charts
`curve()` - creates curves of mathematical equations

The plot() function creates different types of plots depending upon the input variable types. For example, in the case of `plot(X, Y)` if both variables are numeric, a scatter plot is produced. But if X is a factor and Y is numeric, a boxplot is produced. If in case only one argument is given and it is a data frame contains only numeric columns, all possible scatter plots are produced. If both X and Y contains factor variables a mosaic plot is created.

R Exercise 5 *Enter in a set of normally distributed random numbers & plot.*

```
x <- rnorm(20,sd=4,mean=10)
y <- 4.5*x - 2.0 + rnorm(20,sd=5,mean=0)
cor(x,y)
plot(x,y,xlab="X-Axis",ylab="Y-Axis",main="Plot Title")
x1 <- runif(6,12,24)
y1 <- 2.5*x1 - 1.0 + runif(6,-5,5)
points(x1,y1,col=2)
```

Plotting in two variables

If we just want to view a variable to get an idea of the distribution, we can input the vector containing that variable's data. So to plot our original Sprint data.

```
plot(original[,1])
```

This just plots the value in sequence, so along the x-axis is the index of the datum and the y-axis contains its value. It can be easier to see the distribution by first sorting the data. So we can enter

```
plot(sort(original[,1]))
```

Of course, usually histograms are used to plot distributions. So we can also use the following to get a snapshot of our data.

```
hist(original[,1])
```

If we want to plot a single variate function, plotting is reasonably straightforward, however we usually need to create our 'x' values and 'y' values beforehand. To create equispaced x values, the easiest way is to create a vector with our colon option and then scale it to our interval.

So if we want to create a vector of 100 points over the unit interval, we can write `(1:100)/100`. If we want to start at 0 (which would give us 101 points), then we can write `(0:100)/100`.

To plot the function $f(t) = t^2$ for these values, we can either use

```
x <- (1:100)/100
plot(x^2)
```

or if we want the x labels to correspond with the data in that variable, we would include these as the first argument and the transformed values as the second argument.

```
x <- (1:100)/100
plot(x,x^2)
```

However, if we want to plot a function like our piecewise function, we need to create the y values separately first. To create the vector of y values, we will also now need to use a repeating operation `for()`.

The `for()` function can perform an operation for every entry of a set. This is very useful when we can index our numbers.

The following sequence of operations first creates a vector of y-values (pre-filled with zeros). It uses `length(x)` so that it will be the same length as our `x` vector, but we could also just write 100. It then says for all the numbers in 1 to 100 (using the `1:100` vector), each entry in the `y` vector will be replaced by the piecewise function of the *corresponding* entry in the `x` vector. The # can be used in R for notes. Anything on the line that comes after the # is "commented out".

```
y <- array(0,length(x))      # 1. create a vector of zeros
for(i in 1:length(x))        # 2. perform this operation
{y[i] <- x[i]^2}             #    changing 'i' for the numbers
                             #    between 1 to 100.
```

We now should be able to plot this piecewise function.

```
plot(x,y)
```