R computing for Business Data Analytics

Instructor: Dr. Howard Hao-Chun Chuang

Assistant Professor, Department of Management Information Systems

National Chengchi University, Taipei, Taiwan 116

chuang@nccu.edu.tw

Last Revised: August 2014

1.1 Getting started with *R*

- Go to http://cran.r-project.org/
- Choose an operating system, e.g., Windows, Mac OS.
- Choose the “base” program and download the file
- Install the setup program on your laptop/desktop
- *R* documentation: http://cran.r-project.org/manuals.html
- Springer’s Use *R*! series http://www.springer.com/series/6991
- Google for *R* http://www.rseek.org/
- The Nabble-*R* help forum http://r.789695.n4.nabble.com/
- Quick-*R* http://www.statmethods.net/

R is a numerical analysis system and a programming language, which means you type code and compile it (unlike many other software packages). *R* contains many powerful packages that facilitate data analysis, visualization, and reporting. Packages make up the backbone of the *R* community and experience.

Go to http://cran.r-project.org/web/views/ for an overview.

So, how can we install and load packages in *R*? e.g., *coefplot*

R is updated on an irregular basis!

1.2 Basics of *R*

- Calculation

The basic order/priority of operations in *R*: Parenthesis, Exponents, Multiplication, Division, Addition, and Subtraction.

```
> 1+2+3
> 3*7*2
> 6+7*3/2
> (4*6)+5
> 4*(6+5)
> 17/5
> 17%%5
> 17%/%5

> round(pi)
> round(pi, 1)
> ceiling(pi)
> floor(pi)
```

- Variable assignment

Variable names can contain any combination of alphanumeric characters, periods (.) and underscores (_). Note that variable names CANNOT start with a number or an underscore. The valid variable assignment operators are left arrow (< −) and equal sign (=). I personally use the second operator (=) all the time.

```
> x < − 2
> x
```

The arrow operator can also go in the other direction

```
> 3 − > z
> z
> y = 5
> y
```

A more labor-intensive way to assign variables is to use the *assign* function.

```
> assign (“j”, 4)
> j
```

We can also remove the variable using a function.

```
> rm(j)
> j
> rm(list = ls())
```

Each function has its arguments. To find out what the arguments are, let's get help.

```
> help(rm)
```

1.3 Numerical vectors

A vector is a collection of elements, all of the same type. A vector cannot be of mixed type, e.g., a mix of numbers and characters. Also, vectors in *R* are NOT like mathematical vectors that have row or column orientation. Column or row vectors can be represented as one-dimensional matrices, which will be discussed later.

- Create a vector

```
> x=c(1, 2, 3, 4)
> x
> x=1:4
> x
> x=seq(1, 4, by=1)
> x
```

- Identify non-repetitive elements in a vector

```
> unique(x)
```

- How many elements does a vector have?

```
> length(x)
```

- Sort a vector

```
> sort(x, decreasing=FALSE)
> sort(x, decreasing=TRUE)
```

- Logical comparisons for a vector

```
> x<=2
> x > 2
> any (x < 2)
> all (x > 2)
> which(x==2)
> which(x!=2)
```

- Assess individual elements of a vector

```
> x[1]
> x[1:2]
> x[c(1,4)]
```

- Filter a vector

```
> x[which(x>=2 & x<=3)]
> x[which(x>=2 | x<=3)]
```

- Name elements of a vector

```
> c(One=1, Two=2, Three=3, Four=4)
> x=1:4
> names(x)=c("a", "b", "c", "d")
> x
```

- Vector operations

```
> x=rep(0, 4)
> x
> x=x+(1:4)
> x
> x*3
> x+2
> x-3
> x/4
> x**2
> x^2
> sqrt(x)
```

Note that *R* is case sensitive.

```
> X=x^2
> X
> X*x
```

1.4 Character vectors

- Nominal factors

```
> educ=c("High School", "College", "Master", "Doctorate")
> educ #Turn educ into Nominal factors
> educ.factorN=as.factor(educ)
> educ.factorN
> as.numeric(educ.factorN)
```

- Ordinal factors

```
> educ=c("High School", "College", "Master", "Doctorate")
> educ.factorO=factor(x=educ, levels= c("High School", "College", "Master",
"Doctorate", ordered=TRUE)
> educ.factorO
```

Can you see the differences now?

1.5 Missing data

- NA

```
> z=c(1, 2, NA, 3, NA, 4)
> z
> is.na(z)
```

This can be applied to character vectors too.

```
> mean(z)
> mean(z, na.rm=TRUE)

> z>2
> z[z>2]
> subset(z, z>2)

> z[is.na(z)==FALSE]
> z[is.na(z)=TRUE]
```

- NULL

NULL is the absence of anything. Unlike NA that is missingness, NULL is nothingness.

```
> z=c(1, 2, NULL, 3, 4)
> z
```

NULL simply did not get stored in the vector z

```
> d = NULL
> is.null(d)
> is.null(7)
```

is.null is among the few NON-vectorized functions (since NULL cannot be part of a vector).

1.6 Importing & exporting data

- Importing data from a file

```
> results=read.table("?:/…/results.txt", header=T)
> results=read.table("?:\\...\\results.txt", header=T)
```

read.table assumes that the data in the text file are separated by spaces.

read.csv, used when the data points are separated by commas.

read.csv2, used when the data points are separated by semicolons.

Sometimes people use *scan* or *source* instead.

- Exporting data to a file

Use help( ) for *write, write.table*, *cat*, and *dump*.

1.7 Data structures

- *data.frame*

data.frame is just like an Excel spreadsheet that has has columns and rows.

```
> results=read.table("?:/…/results.txt", header=T)
> results$arch1[5] #My preferred way to retrieve the value
```

or

```
> attach(results)
> names(results)
> arch1[5]
```

- What is *attach* for?

```
> data1=data.frame(x=1:10, y=21:30, z=5:14)
> data2=data.frame(x=1:5, y=21:25, z=5:1)
> attach(data1)
> x[ ]
> detach(data1)
> attach(data2)
>x[ ]
> data2$x[ ]
```

- More on *data.frame*

Let's create a *data.frame* using some vectors.

```
> x=10:1
> y=-4:5
> q=c("Hockey", "Football", "Baseball", "Curling", "Rugby", "Lacrosse", "Basketball",
"Tennis", "Cricket", "Soccer")
> theDF=data.frame(x, y, q)
> theDF
```

We could further name those vectors.

```
> theDF=data.frame(First=x, Second=y, Sport=q)
> theDF
```

Below are some useful commands.

```
> nrow(theDF)
> ncol(theDF)
> dim(theDF)
> colnames(theDF)
> rownames(theDF)
```

> theDF$Sport

> head(theDF, n=?)

> tail(theDF, n=?)

Locate row 3, columns 2 through 3

> theDF[3, 2:3]

How to get rows 3 and 5, columns 1 and 3?

>

We can also access multiple columns by name

> theDF[, c("First", "Sport")]

Now, we want to access the "Sport" column only.

> theDF[, "Sport"]

> class(theDF[, "Sport"])

or

> theDF[, "Sport", drop=FALSE]

> class(theDF[, "Sport", drop=FALSE])

Do you see the delicate differences now?

- List

A *list* is a container that holds arbitrary objects of either the same type or varying types. A *list* can contain all numeric or characters or a mix of the two or *data.frame*s, or, other lists.

> list(1, 2, 3)

> list(c(1, 2, 3))

Now create a more complicated list

> list5=list(theDF, 1:10, list(c(1, 2, 3), 3:7))

> names(list5)=c("data.frame", "vector", "list")

> list5

Assess an individual element of a list using double square brackets.

> list5[[1]]

Get a specific column.

```
> list5[[1]][ , 2]
> list5[[1]][ , 2, drop=FALSE]
```

Do the differences look familiar to you?

- Array

An *array* is essentially a multidimensional vector (while a *matrix* is restricted to two dimensions). An *array* has to be all of the SAME type. The first element is the row index, the second is the column index, and the remaining elements are for outer dimensions.

```
> theArrray=array(1:12, dim=c(2, 3, 2))
> theArray
> theArray[1, , ]
> theArray[1, ,1]
> theArray[ , , 1]
```

The three operations above should allow you to see how an array works. I personally do NOT recommend using array.

- Matrix

A *matrix* is a very fundamental mathematical structure that is essential to statistics. I cannot overemphasize the importance of matrix to computational statistics. To be proficient in data analytics, you must excel in manipulating matrices in *R*.

The *nrow*, *ncol*, and *dim* functions work just fine here as they do for *data.frame*.

Create two 5x2 matrices

```
> A=matrix(1:10, nrow=5)
> B=matrix(21:30, nrow=5)
> A
> B
> A+B
```

Let's multiply A and B

```
> A*B
> A%*% t(B)
```

Can you see the differences? Which one is the correct matrix multiplication?

Similar to *data.frame*, we can further assign row and column names. Below are two commands that greatly facilitate matrix manipulation.

```
> cbind(A, B)
> rbind(A, B)
```

Another function that is useful when working with matrices is *apply*, which applies another function to either all of the rows (1) or all of the columns (2) of a matrix.

```
> row.means=apply(B, 2, mean)
> col.means=apply(A, 2, mean)
```

Below lists several useful functions for matrix computation and linear algebra.

det computers the determinant of the matrix.

diag extracts main diagonal of the matrix.

t finds the transpose of the matrix.

solve finds the inverse of the matrix.

eigen returns eigenvalues and eigenvectors of the matrix.

1.8 Summary statistics

- Oft-used metrics

We first generate a random sample of 100 numbers between 1 and 100.

```
> x=sample(x=1:100, size=100, replace=TRUE)
```

Keep this *sample* function in mind. *Sample* is critical as it enables bootstrapping and brute-force Monte-Carlo simulation. We will come back to *sampl*e later.

```
> x
> mean(x) #sample mean
> y=x
> y[sample(x=1:100, size=20, replace=FALSE)]=NA
```

What are we doing here?

```
> y
> mean(y)
```

Remember how we resolve the issue earlier?

Let's generate some random weights.

```
> weights=sample(1:100, 100, replace=FALSE)/sum(1:100)
```

> weighted.mean(x, w=weights)

> var(x) #sample variance

or

> sum((x-mean(x))^2)/(length(x)-1) #follow the formula

> sd(x) #standard deviation

Can you think of an alternative way to calculate the standard deviation?

> median(x)

> min(x)

> max(x)

What would be the range of x?

> summary(x)

> fivenum(x) #Tukey's five number summary

1.9 Graphical displays

R is extremely powerful in generating graphs. Below I simply provide a few representative examples. A huge amount of resources is available on-line and left for you to explore.

- Pie chart

> groupsizes=c(18, 30, 32, 10, 10)

> labels=c("A", "B", "C", "D", "E")

> pie(groupsizes, labels, col=c('grey', 'yellow', 'blue', 'green', 'red'))

- Scatter plot

> x=sample(1:100, 20, replace=TRUE)

> y=sample(1:100, 20, replace=TRUE)

> plot(x, y)

> lines(sort(x), sort(y), type='l')

The following lists some functions that exist to add components onto existing graphs – *points*, *lines*, *text*, *abline*, *polygon*, *segments*, *arrows*, *symbols*, *legend*, etc.

> demo(plotmath) #*expression* is extremely useful for technical/mathematical symbols.

- Boxplot (Box and Whiskers Plot)

Boxplot is a graphical summary based on the median, quartiles, and extreme values. Box represents the 50% interquartile. Whiskers are lines that extend from the box to the highest and lowest values. Line across the box refers to the median. Extreme values are cases more than 1.5 box lengths from the upper or lower end of the box.

Now, first read the data values from *results.txt.*

```
> results=read.table("?:/…/results.txt", header=T)
> attach(results)
> names(results)
> boxplot(prog1, xlab="Programming Semester 1")
```

Generate multiple boxplots

```
> boxplot(arch1~gender, xlab="gender", ylab="Marks(%)", main="Architecture Semester 1")
```

- Stem and leaf plot

```
Stem(prog1)
```

- Pair-wise scatter plots

```
> pairs(results[ , 2:5])
```

- Histograms

```
> bins=c(0, 40, 60, 80, 100)
> hist(arch1, breaks=bins, xlab="Marks(%)", ylab="Number of Students",
main="Architecture Semester 1")
```

Generate multiple histograms within one graph.

```
> dev.new( )
> par(mar=c(3, 4, 1, 1))
> par(mfrow=c(2, 2))
> hist(arch1, xlab="Architecture", main="Semester 1", ylim=c(0, 35))
> hist(arch2, xlab="Architecture", main="Semester 2", ylim=c(0, 35))
> hist(prog1, xlab="Programming", main=" ", ylim=c(0, 35))
> hist(prog2, xlab="Programming", main=" ", ylim=c(0, 35))
```