Computational Statistics in R Lab

Computer Science PG Study Material
Paper: CS-292 Module - II

Anupam Pattanayak[1]
Assistant Professor,
Department of Computer Science,
Raja N. L. Khan Women's College (Autonomous),
Midnapore, West Bengal

April 11, 2020

[1]anupam.pk@gmail.com

Contents

1

R - Introduction

In class, we have seen usage of some preliminary commands in R. We will revisit those and learn more commands and then do programming using R.

we will refer the book by Cotton[1], and the book by Matloff[2].

1.1 Installing R in Ubuntu

We use the *R* in Ubuntu. To install R, you may use either of the following:

I. Open Terminal. If you are not sure about what terminal means, then just press Ctrl+Alt+T. It will open the terminal. Then type the following command:
sudo apt-get install r-base

```
$ sudo apt-get install r-base
```

Here, $ is the command promt of Linux. It automatically appears in terminal. Do not type it explicitly while entering the above command.

II. Open Ubuntu Software Center. Search for r-base in search window there. It will return a package with title *GNU R statistical computation and graphics system* with subtitle *r-base*. Click on the install button to install it.

Once the installation of R is complete, we can open it by entering the command R in terminal, and it gives us something like below:

[1]Learning R - A Step by Step Functional Guide by Richard Cotton, Orielly
[2]The Art of R Programming- A Tour by Norman Matloff, No Starch Press

```
$ R

R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (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.

>
```

R is a *functional programming language*. Like C is a *procedural programming language*, Java is a *object-oriented programming language*, Prolog is a *logic programming language*. R also supports procedural as well as object-oriented programming. R uses *interpreter* for execution. That is, R does not use compiler.

We will use the symbol > to mean R command prompt. To get help about any command from the R command prompt, we use ?command-name. For example, if we want to get help about *mean*, we can use the following command for it.

```
> ?mean
```

It will produce output something like below.

```
mean                       package:base                     R
    Documentation

Arithmetic Mean

Description:
      Generic function for the (trimmed) arithmetic mean.

Usage:

      mean(x, ...)

      ## Default S3 method:
      mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:

        x: An R object.  Currently there are methods for numeric/
    logical
           vectors and date, date-time and time interval objects.
           Complex vectors are allowed for trim = 0, only.

     trim: the fraction (0 to 0.5) of observations to be trimmed
    from
           each end of x before the mean is computed.  Values of
    trim
           outside that range are taken as the nearest endpoint.

    na.rm: a logical value indicating whether NA values should be
           stripped before the computation proceeds.

      ...: further arguments passed to or from other methods.

Value:

      If 'trim' is zero (the default), the arithmetic mean of the
     values
      in 'x' is computed, as a numeric or complex vector of
    length one.
      If 'x' is not logical (coerced to numeric), numeric (
    including
      integer) or complex, 'NA_real_' is returned, with a warning
    .

      If 'trim' is non-zero, a symmetrically trimmed mean is
    computed
      with a fraction of 'trim' observations deleted from each
    end
      before the mean is computed.
:
```

We can continue seeing the help documentation by pressing *Enter* or ↓ key, or *PgDn* Key. Press *q* key to quit from this help page.

We can also see the help by using the following.

```
> help("mean")
```

To exit from R, give *q()* command.

```
> q()
```

It will ask if we want to save workspace. Press *y* for yes, *n* for no, *c* for cancel.

2

R - Basic Commands

We will see basic commands for integer operations. R treats every variable as a vector object.

2.1 Initialize a variable

If we want to work with a scalar value, then R will treat as a vector of size 1. For example, if we want to initialize a variable a with value 10, then we can do either of the following.

```
> a=10
>
```

```
> a<-10
>
```

Enter a in the command prompt to see the value of a. Observe the output style. The output would be as follows:

```
> a
[1]  10
>
```

2.2 Add, Subtract, Product, Divide, Remainder Operations

Let us now sum the two values as follows.

```
> a=20
> b=30
> a+b
[1] 50
>
```

Similary, we can use - for subtraction, ∗for multiplication, / for division. However, to use remainder operator, we need to use %%.

```
> a+b
[1] 50
> a-b
[1] -10
> a*b
[1] 600
> b/a
[1] 1.5
> b%%a
[1] 10
>
```

For floating-point operations, we can initialize and use the operators +, -, ∗, /, %% as shown above, for the integer values. We show a sample below.

```
> x=10.5
> y=30.25
> x+y
[1] 40.75
> x%%y
[1] 10.5
> y%%x
[1] 9.25
>
```

To compute *powers* like x^3, x^4, we use ôperator, as illustrated below.

```
> x=3
> x^3
[1] 27
```

```
> x^4
[1] 81
>
```

2.3 Logical-OR, Logical-AND operations

We use || for logical-OR, && for logical-AND operation. This is shown below. Observe the outputs.

```
> x=8
> y=0
> x||y
[1] TRUE
> x&&y
[1] FALSE
>
```

2.4 Relational Operators

We use == for equality, != for inequality, > for greater than, <= for greater equal to, < for less than, and <= for less equal to relations. See the usage off these operators.

```
> m=5
> n=9
> m==n
[1] FALSE
> m!=n
[1] TRUE
> m>n
[1] FALSE
> m<n
[1] TRUE
> m<=n
[1] TRUE
>
```

2.5 Logarithm and Exponentiation

we use *log( )* for natural logarithm (base e), *log2( )* for logarithm to the base 2, and *log10( )* for logarithm to the base 10. See the following example to see these commands. hown below. Observe the outputs.

```
> x=8
> log(x)
[1] 2.079442
> log2(x)
[1] 3
> log10(10)
[1] 1
>
```

We use *exp(x)* to compute e^x. See the usage of *exp(x)* in the following.

```
> exp(3)
[1] 20.08554
> exp(1)
[1] 2.718282
>
```

2.6 Basic Trigonometric Functions

we use $sin(\theta)$ for computing Sine function. Similarly, we use $cos(\theta)$ for computing Cosine function. See the following example.

```
> sin(0)
[1] 0
> cos(0)
[1] 1
>
```

we use $asin(\theta)$ for computing $Sin^{-1}(\theta)$. Similarly, we use $acos(\theta)$ for computing $Cos^{-1}(\theta)$. Now, look at the following example.

```
> acos(cos(0))
[1] 0
> asin(sin(1))
[1] 1
>
```

2.7 Some Mathematical Utility Functions

Here, we show the usage of some utility functions such as *sqrt*() for computing square root, *abs*() for computing absolute value, $floor(x)$ to compute largest integer less than x, *ceiling*(y) to compute smallest integer greater than y, and *round*(z) to round off z.

```
> sqrt(90)
[1] 9.486833
> sqrt(64)
[1] 8
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> abs(-1)
[1] 1
> abs(-1.5)
[1] 1.5
> floor(1.5)
[1] 1
> ceiling(1.5)
[1] 2
> round(1.67)
[1] 2
>
```

2.8 Some Special Values

Did you notice the output of *sqrt(-1)* in the previous section? It was *NaN*. It stands for *Not a Number*. Some operations does not yield a number, NaN used to represent that scenario.

Inf and -*Inf* are used to denote ∞ and $-\infty$.

pi is used to denote π. That is, it has the value $\frac{22}{7}$.

Sometimes, *NULL* is used to represent no value.

Now, look at the at the following output instances to get these values.

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> 1/0
[1] Inf
> 0-Inf
```

```
[1] -Inf
> Inf-Inf
[1] NaN
> pi
[1] 3.141593
>
```

2.9 Complex Numbers

You might wonder by now, if we can work with complex numbers. Yes, we can work with complex numbers. See the following commands.

```
> x=5+2i
> x
[1] 5+2i
> print(x)
[1] 5+2i
> y=5-2i
> x*y
[1] 29+0i
> > class(x)
[1] "complex"
>
```

2.10 Working with Hexa-Decimal Numbers

We can do some basic operations with hexa-decimal numbers. Prefix *0X* or *0x* is used before the hexa-decimal number. Following example shows the use of this hexa-decimal notaton.

```
> 0Xf
[1] 15
> 0x10
[1] 16
> 0x10+0xf
[1] 31
```

3

R - Input and Output

Did you notice the use of *print(x)* to print value of a complex number x in the last chapter?

Standard input device means keyboard, and *standard output device* means monitor. Remember, in C proggramming we used to write the statement *#include<stdio.h>*, which provides access to various input and output related library functions associated with standard input device and output device.

3.0.1 Display into Standard Output Device

Well, *print( )* is the function to display values or a string. For example, we use print( ) to display value of an object and a string in the following example.

```
> print("Raja N. L. Khan Women's College")
[1] "Raja N. L. Khan Women's College"
> m=2020
> print(m)
[1] 2020
> print(m)
[1] 2020
>
```

We can also used *write( )* to display into *standard output device*, that is monitor. Although, more appropriate use of *write( )* is in writting output to file. To write output in standard output device, use "" as the second argument of *write( )*. Look at the following example.

```
> write("hello","")
hello
> x=2020
> write(x,"")
2020
>
```

To display multiple objects (or, single object), we can use *cat( )* command. *cat* stands for concatenation. This is best used to concatenate two objects and then to write to file. If file is not mentioned, by default cpncatenated output is redirected to standar output device. We can use separator to separate out objects. We will see working with files later on. Now, look at the following example.

```
> cat(x)
2020>
> cat(x,"")
2020 > 5
[1] 5
> x
[1] 2020
> y=2030
> cat(x,y)
2020 2030>
>
```

3.0.2 Read from Standard Input Device

We can use *scan( )*, or *readline()*, or *readLines()* for reading from *standard output device* or some file. By default, whatever read in this way, is treated as string. See the usage of these functions in the following example.

```
> msg<-readline("Enter a number: ")
Enter a number: 5
> msg
[1] "5"
> ?readline()
> ?scan()
> n<-scan()
1: 5
2: 55
3: 555
4:
Read 3 items
> n
[1]   5  55 555
> n<-scan(,n=1)
```

```
1: 5
Read 1 item
> n
[1] 5
> n<-scan("stdin",n=1)
5
Read 1 item
> m<-readLines("stdin",n=1)
50
> m
[1] "50"
>
```

Observe the outputs closely. We use n=1 to indicate that only one value is to be read from keyboard. When we did not give n=1 in the statement *n<-scan()*, then it did not stop after reading one value, it actually continued to read for vector. However, we can force to read just one value by pressing *Enter* key again instead of giving another input.