

Data Mininf Using R- All commands

#####

Utility Commands

getwd()

setwd("E:/DataScience/Workshop_DM_with_R/Scripts")

getwd()

dir() # lists the contents of current working directory.

ls() # lists names of objects in R environment

help.start() # provides general help

?foo # Get help on function/package 'foo'

data() # lists all example datasets in currently loaded packages.

library() # lists all available packages

install.packages("name_of_package") // installing packages

install.packages("dplyr") # installs package dplyr.

library("name_of_package") // to load a package

library("dplyr")

library(rpart) # load package "rpart" in R.

data(mtcars) # loads dataset "mtcars" in R.

rm(mtcars) # removes one or more objects from R workspace.

history(2) # lists last # commands. default 25.

help(package="package-name") # provides brief description of package, an index of functions and datasets in package.

x=3

print(x) # print value of obejct x on terminal.

sessionInfo() # returns information about the current R session

BVIMITBVIMITBVIMIT

`packageVersion("rpart")` # returns the version description of package rpart.

`packageInformation()`

`attach(object)` # attaches given object to R search path

`detach(object)` # detaches given object from R search path

`q()` # Quit current R session

data() to see all datasets in R

#####

Data Types

- Data types
- checking type of variable
- printing variable and objects
- R Data Structures– (Vector, Matrix, List, Factor, Data frame, Table)

Five basic types in R are - Character, Numeric, Integer, Complex, Logical (true/false).

Common data objects are - Vector, Matrix, List, Factor, Data frame, Table.

Basic data types in R can be divided into the following types:

Data type	Example
● numeric	(10.5, 55, 787)
● integer	(1L, 55L, 100L, where the letter "L" declares this as an integer)
● complex	(9 + 3i, where "i" is the imaginary part)
● character	("k", "R is exciting", "FALSE", "11.5")
● logical	(TRUE or FALSE)

Creating and assigning to a variable:

`x <- 10`

`x`

`y = 5`

`y`

BVIMITBVIMITBVIMIT

Checking the type of variable:

class(x) # to check the data type of a variable

example

```
x=3
```

```
class(x)
```

```
[1] "numeric"
```

```
y='hello'
```

```
class(y)
```

```
[1] "character"
```

Printing Variable

```
x
```

```
# explicit printing
```

```
print(x)
```

is., as. Functions - R has is.* and as.* family of functions that can be used to check whether a variable is of given type and convert the variable to a specific type.

```
x <- 'c'
```

```
x
```

#check if x is of type character

```
is.character(x)
```

```
as.integer(x) #convert to integer
```

Type Conversion

You can convert from one type to another with the following functions:

- `as.numeric()`
- `as.integer()`
- `as.complex()`

```
y <- '2'
```

```
y
```

```
class(y)
```

```
y <- as.integer(y) #convert to integer
```

```
y # print value of y
```

```
class(y)
```

EXAMPLE

```
v <- TRUE  
print(class(v))
```

```
[1] "logical"
```

```
v <- 23.5
```

```
print(class(v))
```

it produces the following result –

```
[1] "numeric"
```

```
v <- 2L  
print(class(v))
```

it produces the following result –

```
[1] "integer"
```

```
v <- 2+5i  
print(class(v))
```

it produces the following result –

```
[1] "complex"
```

```
v <- "TRUE"  
print(class(v))
```

it produces the following result –

```
[1] "character"
```

String

```
s <- "Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."
```

```
s # print the value of str
```

```
[1] "Lorem ipsum dolor sit amet,\nconsectetur adipiscing elit,\n sed do eiusmod tempor incididunt\nut labore et dolore magna aliqua."
```

However, note that R will add a "**\n**" at the end of each line break. This is called an escape character, and the **n** character indicates a **new line**.

If you want the line breaks to be inserted at the same position as in the code, use the `cat()` function:

```
> cat(ss)  
Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.
```

String Length

There are many useful string functions in R.

BVIMITBVIMITBVIMIT

For example, to find the number of characters in a string, use the `nchar()` function:

Example

```
str <- "Hello World!"
```

```
nchar(str)
```

1. Vector

A vector is simply a list of items that are of the same type.

To combine the list of items to a vector, use the `c()` function and separate the items by a comma.

Creating Vector: contains objects of same class.

using `c()` function

```
x <- c(11.3, 27.5, 33.8)
```

```
x
```

To create a vector with numerical values in a sequence, use **the `:` operator**:

```
numbers <- 1:10
```

```
numbers
```

```
x <- c(5:25)
```

```
x
```

using `vector()` function

```
y <- vector("logical", length=10)
```

```
y
```

Vector Length

To find out how many items a vector has, use the **`length()`** function:

```
f <- c(1,2,3,4,5)
```

```
length(f)
```

Sort a Vector

To sort items in a vector alphabetically or numerically, use the `sort()` function:

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
numbers <- c(13, 3, 5, 7, 20, 2)
```

```
sort(fruits) # Sort a string
```

```
sort(numbers) # Sort numbers
```

```
BVIMITBVIMITBVIMIT
```

Vector operations: Various arithmetic operations can be performed member-wise. Like:

Multiplication by a scalar.

Addition of two vectors.

Multiplication of two vectors, etc.

```
y <- c(4,5,6)
```

multiplication by a scalar

```
5*y
```

addition of two vectors

```
x+y
```

multiplication of two vectors

```
x*y
```

```
x
```

```
y
```

x to the power y

```
x^y
```

2. MATRIX

The basic syntax for creating a matrix in R is –

```
matrix(data, nrow, ncol, byrow, dimnames)
```

Following is the description of the parameters used –

- **data** is the input vector which becomes the data elements of the matrix.
- **nrow** is the number of rows to be created.
- **ncol** is the number of columns to be created.
- **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
- **dimname** is the names assigned to the rows and columns.

Creating Matrix: Two-dimensional array having elements of same class.

using matrix() function.

```
m <- matrix(c(11,12,13,55,60,65,66,72,78),nrow=3,ncol=3)
```

```
m
```

#dimensions of matrix m

```
dim(m)
```

#attributes of matrix m

```
attributes(m)
```

By default, elements in matrix are filled by column.

BVIMITBVIMITBVIMIT

#attribute of matrix can be used to fill elements **by row**.

```
m <- matrix(c(11,12,13,55,60,65,66,72,78),nrow=3,ncol=3,byrow = TRUE)  
m
```

```
# Access the element at 3rd column and 1st row.  
print(m[1,3])
```

cbind-ing and rbind-ing: By using cbind() and rbind() functions

cbind() function **combines vector, matrix or data frame by columns**. The row number of the two datasets must be equal. If two vectors do not have the same length, the elements of the short one will be repeated.

```
x<-c(1,2,3)  
y<-c(11,12,13)  
cbind(x,y)  
      [1,] 1  11  
      [2,] 2  12  
      [3,] 3  13  
rbind(x,y)  
      [1,] [2,] [3,]  
x      1    2    3  
y     11   12   13
```

Matrix operations/functions:

1. Multiplication by a scalar.
2. Addition, subtraction and multiplication of two matrices.
3. Transpose, determinant of a matrix, etc.

1. multiplication by a scalar

```
m  
p<-3*m  
p
```

2. addition of two matrices

```
n <- matrix(c(4,5,6,14,15,16,24,25,26),nrow=3,ncol=3)  
n  
m  
q <- m+n  
q
```

matrix multiplication by using %*%

```
o<-matrix(c(4,5,6,14,15,16),nrow=3,ncol=2)  
r <- m %*% o  
r
```

3. transpose of matrix

```
m
```

```
mdash<-t(m)
```

```
mdash
```

filling a matrix by row

```
s <- matrix(c(4,5,6,14,15,16,24,25,26), nrow=3,ncol=3,byrow=TRUE)
```

```
s
```

#sum of row elements

```
m
```

```
m_row_sum<-rowSums(m)
```

```
m_row_sum
```

#sum of column elements

```
m
```

```
m_col_sum <- colSums(m)
```

```
m_col_sum
```

1. Extract Values from a Matrix

```
m <- matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, ncol = 3, byrow = T)
```

```
m
```

```
m[2, 3]
```

```
m[2:3, ] # get the second and third row
```

```
m[, 2:3]
```

```
m[2, ]
```

```
m[, 3]
```

```
m[, c(1, 3)]
```

```
m[c(1, 3), ]
```

3. LIST

- A list in R can contain **many different data types** inside it. A list is a collection of data which is ordered and changeable.

- To create a list, use the **list()** function:

```
# List: A special type of vector containing elements of different classes.
```

```
# Elements of list can be accessed by giving element index or name in [[]].
```

```
x <- list(1,"p",TRUE,2+4i)
```

```
x
```

```
x <- list(y,"p",TRUE,2+4i) # y= y<-c(11,12,13)
```

```
x
```

```
# Create a list containing a vector, a matrix and a list.
```

```
list_data <- list(c("Jan","Feb","Mar"), matrix(c(1,2,3,4,5,6), nrow = 2),list("red",12))
```



```
# Give names to the elements in the list.

names(list_data) <- c("1st Quarter", "A_Matrix", "An Inner list")

# Access the first element of the list.

print(list_data[1])

# Access the third element. As it is also a list, all its elements
will be printed.

print(list_data[3])

# Access the list element using the name of the element.

print(list_data$A_Matrix)
```

4. Factor

Factor: Represents categorical data. Can be ordered or unordered.

Factors are the data objects which are used to categorize the data and store it as levels. Factors are created using the **factor ()** function by taking a vector as input.

```
status <- c("low","high","medium","high","low")
# using factor() function
x <- factor(status, ordered=TRUE,levels=c("low","medium","high"))
x
levels(x)
nlevels(x)
s <- factor(c("male", "female", "female", "male"))
# 'levels' argument is used to set the order of levels.
# First level forms the baseline level.
# Without any order, levels are called nominal. Ex. - Type1, Type2, .
# With order, levels are called ordinal. Ex. - low, medium, .
```

5. Data frame

Used to store tabular data. Can contain different classes.

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Example 1

```
student_id<-c(1,2,3)
BVIMITBVIMITBVIMIT
```

```
student_names<-c("Priya","Shyam","Seeta")
position<-c("First","Second","Third")
#using data.frame() function
data<-data.frame(student_id, student_names, position)
data
```

	student_id	student_names	position
1	1	Ram	First
2	2	Shyam	Second
3	3	Laxman	Third

```
> |
```

```
data[, c("student_id","student_names")] # display only student_id, student_names columns
```

#column names of data. for a dataframe, colnames() can also be used.

```
names(data) OR colnames(data)
```

Example 2

```
emp_data <- data.frame(
  emp_id = c (1:4),
  emp_name = c("Reeta","Priya","Seeta","Satyam"),
  salary = c(456,24,322,255))
print(emp_data)
class(emp_data)
# Name the data frame
names(emp_data) <- c('EID', 'ENAME', 'SALARY')
```

#accessing a particular column

```
data$student_id
```

#no. of rows in data

```
nrow(data)
```

#no. of columns in data

```
ncol(data)
```

Attach, and detaching data.

attach(object) # attaches given object to R search path

detach(object) # detaches given object from R search path

```
data1=data.frame(x1=c(1,2,3,4), x2=c(2,4,6,8),x3=c(5,10,15,20))
```

```
attach(data1)
```

```
BVIMITBVIMITBVIMIT
```

```

> data1=data.frame(x1=c(1,2,3,4), x2=c(2,4,6,8),x3=c(5,10,15,20))
> data1
  x1 x2 x3
1  1  2  5
2  2  4 10
3  3  6 15
4  4  8 20
> x1
Error: object 'x1' not found
> attach(data1)
> x1
[1] 1 2 3 4
> detach(data1)
> x1
Error: object 'x1' not found

```

#####

Operators

Arithmetic operators: Regular operators like +, - etc. ^ (exponentiation),

%% (modulus), %/% (integer division).

Relational and Logical operators: <, <=, >, >=, ==, !=, ! (Not), | (Or), & (And)

Assignment operators: <-, =, <<- (search through parent env. also).

Also exist -> and ->>. <<-, ->> mainly used in function definitions.

Special operators: :(to create sequence), %in% (to find in a range or vector),

%*% (mat. multiplication).

Moreover specific packages can also provide special operators.

For ex.: %>% of magrittr.

#####

Control structures

R provides all types of control structures: **if-else, for, while, repeat, break,**

next, return.

Mainly used within functions/scripts.

```
x<-5
```

```
#if-else structure
```

```
if(x > 7)
```

```
  y <- TRUE else
```

```
  y <- FALSE
```

```
y
```

```
#for loop
```

```
BVIMITBVIMITBVIMIT
```

```

for(i in 1:10)
  print(i)

#while loop
count<-0
while(count < 10)
  count<-count+1

count

```

Looping functions

```

# These functions can be used to loop over various type of objects.
# lapply - Loop over a list and evaluate a function on each element.
# sapply - Same as lapply but try to simplify the result.
# apply - Apply a function over the margins of an array.
## Traverses the array (matrix) either row-wise or column-wise to apply the function.
set.seed(789)
x<-list(a=1:5,b=rnorm(20))
x
#lapply returns a list
a <- lapply(x,sum)
a
class(a)

#sapply returns:-
#1. vector if every element in result is of length 1.
#2. matrix if every element in result is a vector of length > 1.
#3. list otherwise
b <-sapply(x, sum)
b
class(b)

```

```

x<-matrix(c(1,2,3,11,12,13), nrow=2, ncol=3,byrow=TRUE)
x
# MARGIN=1 for rows, MARGIN=2 for columns
apply(x,MARGIN=1,FUN=sum)
apply(x,MARGIN=2,FUN=sum)

```

```
#####
```

Functions

```

# As in other programming languages, represent a block of instructions performing a given task.
# Created using the function() directive.

```

```
BVIMITBVIMITBVIMIT
```

Can be passed as arguments to other functions. Can be nested.
Return value is the last expression to be evaluated inside function body.
Have named arguments with default values.
Some arguments can be missing during function calls.
One special argument: R has a special argument notation called ellipses or three-dot ("."). This can be used to show that function can accept variable number and type of arguments. It can also be passed to other functions.

```
add<-function(a=1,b=2,c=3) {  
  s = a+b+c  
  print(s)  
}
```

```
add()  
add(10,11,12)  
add(10)
```

R Source files
Can be used to store functions, commands required to be executed sequentially etc.
Should be saved/created with .R extension.
Extremely important to save all the steps of your data analysis task at a common place.
source() function used to load such R scripts into R workspace.
source("./test.R")
mult()

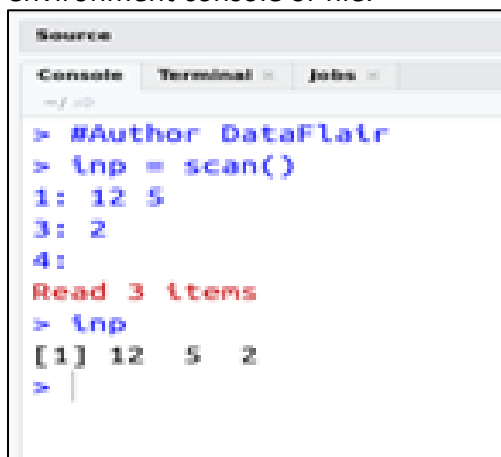
```
source("./test1.R", echo=T) #prints each expression in script before evaluation.  
#####
```

Read and Write Data

Reading from the keyboard:

- **scan()**

Read Data Values: This is used for reading data into the input vector or an input list from the environment console or file.



```
Source  
Console Terminal Jobs  
~/ >  
> #Author DataFlair  
> lnp = scan()  
1: 12 5  
3: 2  
4:  
Read 3 items  
> lnp  
[1] 12 5 2  
>
```

- **readline()**

With readline(), we read multiple lines from a connection.

We can use readline() for inputting a line from the keyboard in the form of a string:

For example:

```
> str = readline()
> str
```

Loading Data from different Data Source

How to Import an Excel File into R

install the readxl package:

```
install.packages("readxl")
library("readxl")
read_excel("Path where your Excel file is stored\\File Name.xlsx")
```

```
read_excel("F:\\RProjects\\Student.xlsx")
```

```
read_excel("Student.xlsx") // if the file in current working directory
```

we can also use

read_xls("Student.xlsx") for .xlsx file

read_xls("Student.xls") for .xls file ETC...

// Saving data into a variable from a sheet and printing

```
dd <- read_xlsx("Student.xlsx", sheet=2)
```

```
print(dd)
```

The R **read.table** function is very useful to import the data from text files from the file system & URLs and store the data in a Data Frame.

Example

```
Data1 <- read.table("student1.csv", header = TRUE, sep = ",")
```

Data1

- **file:** You have to specify the file name or Full path along with file name.

- **header:** If the text file contains Columns names as the First Row, specify TRUE otherwise, FALSE.
- **sep:** It is a short form of the separator. You have to specify the character that is separating the fields. ”, “ means data separated by a comma.

```
dataT <- read.csv("student1.csv", header = TRUE, sep = ",")
```

```
dataT
class(dataT)
dim(dataT)
head(dataT, n = 3)
tail(dataT, n = 4)
str(dataT)      # Structure
```

NOTE : the `read.csv()` function is almost identical to the `read.table()` function, with the difference that it has the `header` and `fill` arguments set as `TRUE` by default.

Write data to disk in a file

```
write.csv(dataT, file = "Mydata.csv")
```

```
#####
```

Data Pre-processing

- Naming and Renaming variables
- adding a new variable.
- Dealing with missing data.
- Dealing with categorical data.
- Data reduction using subsetting

Naming and renaming variables

```
data = read.table(file="PrimeMinisters.csv", sep = ",")
data[1:2,]
names(data)
```

```
data = read.table(file="PrimeMinisters.csv", sep = ",", header = T)
data[1:2,]
names(data)
```

Adding headers

```
data = read.csv(file="PrimeMinisters-noheader.csv",
               col.names=c("PMNAME","DateOfBirth", "DateOfDeath"))
data[1:2, ]
names(data)
```

Renaming Headers

```
names(data) <- c("PMNAME_renamed","DateOfBirth_renamed", "DateOfDeath")
names(data)
data[1:2, ]
```

Add new column to data frame

```
data <- read.csv("student1.csv", header = T)
data
data$TOTMKS <- data$MATHS + data$Science
data$MEANMKS <- (data$MATHS + data$Science)/2
str(data)
```

derive columns to/from new/existing columns

```
data$RES [data$MATHS < 50 | data$Science < 50] <- "FAIL"
data$RES [data$MATHS >= 50 & data$Science >= 50] <- "PASS"
data
```

```
data[which.min(data$MATHS), ]
data[which.max(data$Science), ]
```

```
mean(data$MATHS)
median(data$MATHS)
mean(data$Science)
median(data$Science)
```

#####

Sort

```
sort(data$MATHS)
sort(data$MATHS, decreasing = T)
sort(data$RES)
```

```
order(data$MATHS, decreasing = TRUE) // shows row numbers of the sorted data
data[order(data$MATHS, decreasing = TRUE), ]
```

#####

Date Conversion

use `as.Date()` to convert strings to dates
`mydates <- as.Date(c("2007-06-22", "2004-02-13"))`

Sys.Date() returns today's date.

date() returns the current date and time.

The default format is **yyyy-mm-dd**

The following symbols can be used with the **format()** function to print dates.

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month (00-12)	00-12

%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007

```
today <- Sys.Date()
format(today, format="%B %d %Y")
```

```
# Syntax for the function is : as.Date (InputCharacterString, FormatUsedInInput)
# Specify format used in input string using following symbols : "14 November 1889" is
# using "%d %B %Y"
```

as.Date() function to convert character data to dates

```
data = read.csv(file="PrimeMinisters.csv", header = T)
data
str(data)
```

An example of date conversion - YYYY-MM-DD

```
as.Date("14 November 1889", "%d %B %Y")
```

Converting a complete column

```
data$DOBnew <- as.Date(data$Date.of.birth, "%d %B %Y")
data
str(data)
```

```
#####
```

Detecting and Handling Missing values

```
# NA - Not Available - i.e. missing value
NA + 4
```

```
# Create a vector V with 1 NA value
```

```
V <- c(1,2,NA,3)
```

```
V
```

```
class(V)
```

Sum with and without NA (remove NA)

```
sum(V)
```

```
sum(V, na.rm = T) // sum after removing NA
```

```
is.na(V)
```

```
FALSE FALSE TRUE FALSE
```

```
naVals <- is.na(V)
```

Get values that are not NA

```
!naVals
```

```
V[!naVals]
```

```
complete.cases(V)
```

```
BVIMITBVIMITBVIMIT
```

```
V[complete.cases(V)]
```

```
dataC <- read.csv(file = "PrimeMinisters.csv", na.strings = "") # blank cells are replaced with NA
dataC
dim(dataC)
```

```
#####
```

Data Imputation

In R, there are a lot of packages available for imputing missing values - the popular ones being **Hmisc**, **missForest**, **Amelia** and **mice**.

Imputation is a method to fill in the missing values with estimated ones. **Mean / Mode / Median** imputation is one of the most frequently used methods. It consists of replacing the missing data for a given attribute by the mean or median (quantitative attribute) or mode (qualitative attribute) of all known values of that variable.

```
library(Hmisc)
```

```
## create a vector
```

```
x = c(1,2,3,NA,4,4,NA)
```

```
# mean imputation - from package, mention name of function to be used
```

```
x <- impute(x, fun = mean)
```

```
x
```

```
x = c(1,2,3,NA,4,4,NA)
```

```
# median imputation - from package, mention name of function to be used
```

```
x <- impute(x, fun = median)
```

```
x
```

```
#####
```

Data Merging

```
d1 = read.csv("student1.csv", header = T)
```

```
d1
```

```
# d2 data frame
```

```
d2 = read.csv("StudentEng.csv", header = T)
```

```
d2
```

```
# Merge using ID
```

```
m=merge(d1,d2,by="ID")
```

```
m
```