Theory on R Programming Language

Computer Programming (Mahatma Gandhi University)

Module 1

Introduction to R

What is R?

R is a computer language developed specifically for statistical computing. It is actually more than that, though. R provides a complete environment for interacting with your data. You can directly use the functions that are provided in the environment to process your data without writing a complete program. You also can write your own programs to perform operations that do not have built-in functions, or to repeat the same task multiple times, for instance.

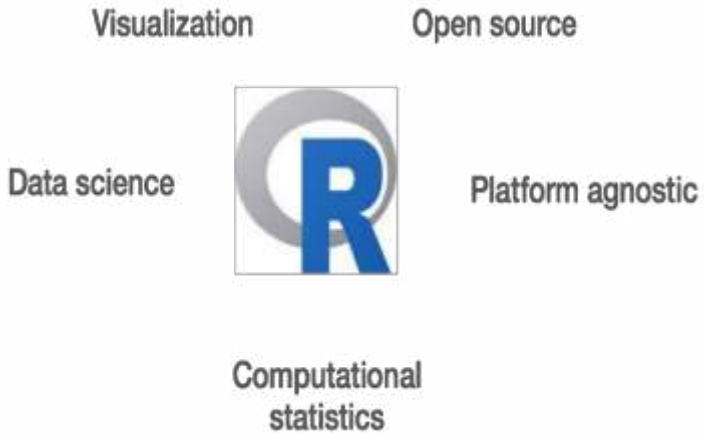
R is an object-oriented language that uses vectors and matrices as its basic operands. This feature makes it quite useful for working on large sets of data using only a few lines of code. The R environment also provides excellent graphical tools for producing complex plots relatively easily. And, perhaps best of all, it is free. It is an open source project developed by many volunteers.

The R environment combines:

- effective handling of big data
- collection of integrated tools
- graphical facilities
- simple and effective programming language

Why use R?

R is a powerful, extensible environment. It has a wide range of statistics and general data analysis and visualization capabilities.



- Data handling, wrangling, and storage
- Wide array of statistical methods and graphical techniques available
- Easy to install on any platform and use (and it's free!)
- Open source with a large and growing community of peers
- R programming is used as a leading tool for machine learning, statistics, and data analysis. Objects, functions, and packages can easily be created by R.
- It's a platform-independent language. This means it can be applied to all operating system.
- It's an open-source free language. That means anyone can install it in any organization without purchasing a license.
- R programming language is not only a statistic package but also allows us to integrate with other languages (C, C++). Thus, you can easily interact with many data sources and statistical packages.
- The R programming language has a vast community of users and it's growing day by day.
- R is currently one of the most requested programming languages in the Data Science job market that makes it the hottest trend nowadays.

Features of R Programming Language

Statistical Features of R:

- Basic Statistics: The most common basic statistics terms are the mean, mode, and median. These are all known as "Measures of Central Tendency." So using the R language we can measure central tendency very easily.
- Static graphics: R is rich with facilities for creating and developing interesting static graphics. R contains functionality for many plot types including graphic maps, mosaic plots, biplots, and the list goes on.
- Probability distributions: Probability distributions play a vital role in statistics and by using R we can easily handle various types of probability distribution such as Binomial Distribution, Normal Distribution, Chi-squared Distribution and many more.

Programming Features of R:

- R Packages: One of the major features of R is it has a wide availability of libraries. R has CRAN(Comprehensive R Archive Network), which is a repository holding more than 10, 0000 packages.
- Distributed Computing: Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency and performance. Two new packages ddR and multidplyr used for distributed programming in R were released in November 2015.

Advantages of R:

- R is the most comprehensive statistical analysis package. As new technology and concepts often appear first in R.
- As R programming language is an open source. Thus, you can run R anywhere and at any time.
- R programming language is suitable for GNU/Linux and Windows operating system.
- R programming is cross-platform which runs on any operating system.
- In R, everyone is welcome to provide new packages, bug fixes, and code enhancements.

Disadvantages of R:

- In the R programming language, the standard of some packages is less than perfect.
- Although, R commands give little pressure to memory management. So R programming language may consume all available memory.
- In R basically, nobody to complain if something doesn't work.

Applications of R:

- We use R for Data Science. It gives us a broad variety of libraries related to statistics. It also provides the environment for statistical computing and design.
- R is used by many quantitative analysts as its programming tool. Thus, it helps in data importing and cleaning.
- R is the most prevalent language. So many data analysts and research programmers use it. Hence, it is used as a fundamental tool for finance.
- Tech giants like Google, Facebook, bing, Accenture, Wipro and many more using R nowadays.

RStudio: An Integrated Development Environment (IDE) for R

RStudio is freely available open-source Integrated Development Environment (IDE). RStudio provides an environment with many features to make using R easier and is a great alternative to working on R in the terminal.

- Graphical user interface, not just a command prompt
- Great learning tool
- Free for academic use
- Platform agnostic
- Open source

Basic Tips for using R

- R is command-line driven. It requires you to type or copy-and-paste commands after a command prompt (>) that appears when you open R. After typing a command in the R console and pressing **Enter** on your keyboard, the command will run. If your command is

not complete, R issues a continuation prompt (signified by a plus sign: +). Alternatively you can write a script in the script window, and select a command, and click the **Run** button.

- R is case sensitive. Make sure your spelling and capitalization are correct.
- Commands in R are also called functions. The basic format of a function in R is: object <- function.name(argument_1 = data, argument_2 = TRUE).
- The up arrow (^) on your keyboard can be used to bring up previous commands that you've typed in the R console.
- The $ symbol is used to select a particular column within the table (e.g., table$column).
- Any text that you do not want R to act on (such as comments, notes, or instructions) needs to be preceded by the # symbol (a.k.a. hash-tag, comment, pound, or number symbol). R ignores the remainder of the script line following #.

RStudio Interface

The RStudio interface has four main panels:

- Console: where you can type commands and see output. The console is all you would see if you ran R in the command line without RStudio.
- Script editor: where you can type out commands and save to file. You can also submit the commands to run in the console.
- Environment/History: environment shows all active objects and history keeps track of all commands run in console
- Files/Plots/Packages/Help

Source

The source pane is where you create and edit R Scripts" – your collections of code. R scripts are just text files with the ".R" extension. When you open RStudio, it will automatically start a new Untitled script. Before you start typing in an untitled R script, you should always save the file under a new file name. That way, if something on your computer crashes while you are working, RStudio will have your code waiting for you when you re-open RStudio, as it has recovered the code that you were editing.

Console

The console is the heart of R. By default, It is present at the bottom left of the window. It is also called a command window. Here is where R actually evaluates code. At the beginning of the console you will see the character. This is a prompt that tells you that R is ready for new code. You can type code directly into the console after the prompt and get an immediate response, just like any REPL. For example, if you type 3+5 into the console and press enter, you will see that R immediately gives an output of 8.

```
3+5
8
```

Environment/History

The Environment tab of this panel shows you the names of all the data objects (like vectors, matrices, and dataframes) that you have defined in your current R session. You can also see information like the number of observations and rows in data objects. As you get more comfortable with R, you might find the Environment / History panel useful. you can also declutter the panes in the screen, or just minimize the window by clicking the minimize button on the top right of the panel.
The history window shows all commands that were executed in the console.

Files/Plots/Packages/Help

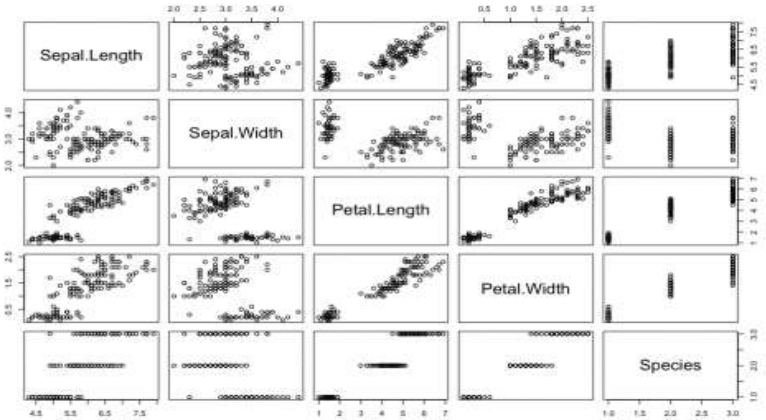
By default, This is located at the bottom right of the window and it shows you lots of helpful information. Here you can open files, view plots, install and load packages, read main pages, and view markdown and other documents in the viewer tab.
Let's go through each tab in detail:

Files
The files panel gives you access to the file directory on your hard drive. One nice feature of the "Files" panel is that you can use it to set your working directory – once you navigate to a folder you want to read and save files to, click "More" and then "Set As Working Directory." We'll talk about working directories in more detail soon.
Plots
The Plots panel shows all your plots. There are buttons for opening the plot in a separate window and exporting the plot as a pdf or jpeg (though you can also do this with code using the `pdf()` or `jpeg()` functions.)

Sample Plot

```
plot(iris)
```

Gives this plot:



Packages

Shows a list of all the R packages installed on your hard drive and indicates whether or not they are currently loaded. Packages that are loaded in the current session are checked while those that are installed but not yet loaded are unchecked. Packages can also be installed using the command (in console), like this one below:

```
install.packages("tidyverse")
```

Help

Help menu for R functions. You can either type the name of a function in the search window or use the code to search for a function with the name.

Advantages of R Studio

One of the main advantages of R is that, If the function you want isn't available you can write your own package and share it with the world.
RStudio includes powerful coding tools designed to develop and publish such packages. It also enhances productivity and supports authoring HTML, PDF, Word Documents, and slideshows too. RStudio also makes it easy to start new or find existing projects.

Variables in R Programming

Variables are used to store the information to be manipulated and referenced in the R program. The R variable can store an atomic vector, a group of atomic vectors, or a combination of many R objects.

Language like C++ is statically typed, but R is a dynamically typed, means it check the type of data type when the statement is run. A valid variable name contains letter, numbers, dot and underlines characters. A variable name should start with a letter or the dot not followed by a number.

Name of variable	Validity	Reason for valid and invalid
_var_name	Invalid	Variable name can't start with an underscore(_).
var_name, var.name	Valid	Variable can start with a dot, but dot should not be followed by a number. In this case, the variable will be invalid.

var_name%	Invalid	In R, we can't use any special character in the variable name except dot and underscore.
2var_name	Invalid	Variable name cant starts with a numeric digit.
.2var_name	Invalid	A variable name cannot start with a dot which is followed by a digit.
var_name2	Valid	The variable contains letter, number and underscore and starts with a letter.

R Operators

R has several operators to perform tasks including arithmetic, logical and bitwise operations.

We have the following types of operators in R programming −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

R Arithmetic Operators

These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

Arithmetic Operators in R

Operator	**Description**
+	Addition
–	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division

An example run

```
> x <- 5
> y <- 16
> x+y
[1] 21
> x-y
[1] -11
> x*y
[1] 80
> y/x
[1] 3.2
> y%/%x
[1] 3
> y%%x
[1] 1
> y^x
[1] 1048576
```

R Relational Operators

Relational operators are used to compare between values. Here is a list of relational operators available in R.

Relational Operators in R

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

An example run

```
> x <- 5
> y <- 16
> x<y
[1] TRUE
> x>y
[1] FALSE
> x<=5
[1] TRUE
> y>=20
[1] FALSE
> y == 16
[1] TRUE
> x != 5
[1] FALSE
```

Operation on Vectors

The above mentioned operators work on vectors. The variables used above were in fact single element vectors.

We can use the function c() (as in concatenate) to make vectors in R.

All operations are carried out in element-wise fashion. Here is an example.

```
> x <- c(2,8,3)
> y <- c(6,4,1)
> x+y
[1]  8 12  4
> x>y
[1] FALSE  TRUE  TRUE
```

When there is a mismatch in length (number of elements) of operand vectors, the elements in shorter one is recycled in a cyclic manner to match the length of the longer one.

R will issue a warning if the length of the longer vector is not an integral multiple of the shorter vector.

```
> x <- c(2,1,8,3)

> y <- c(9,4)

> x+y # Element of y is recycled to 9,4,9,4

[1] 11  5 17  7

> x-1 # Scalar 1 is recycled to 1,1,1,1

[1] 1 0 7 2

> x+c(1,2,3)

[1]  3  3 11  4
```

Warning message:

In x + c(1, 2, 3) :

longer object length is not a multiple of shorter object length

R Logical Operators

Logical operators are used to carry out Boolean operations like AND, OR etc.

Logical Operators in R

Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
|	Element-wise logical OR
||	Logical OR

Operators & and | perform element-wise operation producing result having length of the longer operand.

But && and || examines only the first element of the operands resulting into a single length logical vector.

Zero is considered FALSE and non-zero numbers are taken as TRUE.

An example run.

```
> x <- c(TRUE,FALSE,0,6)
> y <- c(FALSE,TRUE,FALSE,TRUE)
> !x
[1] FALSE  TRUE  TRUE FALSE
> x&y
[1] FALSE FALSE FALSE  TRUE
> x&&y
[1] FALSE
> x|y
[1]  TRUE  TRUE FALSE  TRUE
> x||y
[1] TRUE
```

R Assignment Operators

These operators are used to assign values to variables.

Assignment Operators in R

Operator	**Description**
<-, <<-, =	Leftwards assignment
->, ->>	Rightwards assignment

The operators <- and = can be used, almost interchangeably, to assign to variable in the same environment.

The <<- operator is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used.

Example:

```
> x <- 5
> x
[1] 5
> x = 9
> x
[1] 9
> 10 -> x
> x
[1] 10
```

Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example

:	Colon operator. It creates the series of numbers in sequence for a vector.	`v <- 2:8`
		`print(v)`
		it produces the following result −
		[1] 2 3 4 5 6 7 8

%in%	This operator is used to identify if an element belongs to a vector.	`v1 <- 8`
		`v2 <- 12`
		`t <- 1:10`
		`print(v1 %in% t)`
		`print(v2 %in% t)`
		it produces the following result −
		[1] TRUE
		[1] FALSE

%*%	This operator is used to multiply a matrix with its transpose.	M = matrix(c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE) t = M %*% t(M) print(t) it produces the following result − [,1] [,2] [1,] 65 82 [2,] 82 117

Data Types in R

There are many basic data types in R, which are of frequent occurrence in coding R calculations and programs. Though seemingly in the clear, they can at a halt deliver surprises. Here you will try to understand all of the different forms of data type well by direct testing with the R code.

Here is the list of all the data types provided by R:

- **Numeric**
  Decimal values are referred to as numeric data types in R. This is the default working out data type. If you assign a decimal value for any variable x like given below, x will become a numeric type.
- **Integer**
  If you want to create an integer variable in R, you have to invoke the as.integer() function to define any integer type data. You can be certain that y is definitely an integer by applying the is.integer() function.
- **Complex**
  A complex value for coding in R can be defined using the pure imaginary values 'i'.
- **Logical**
  A logical value is mostly created when a comparison between variables are done.
- **Character**
  A character object can be used for representing string values in R. You have to convert objects into character values using the as.character() function within your code.

Example:

#Logical Data type

variable_logical<- TRUE

cat(variable_logical,"\n")

```
cat("The data type of variable_logical is ",class(variable_logical),"\n\n")

#Numeric Data type
variable_numeric<- 3532
cat(variable_numeric,"\n")
cat("The data type of variable_numeric is ",class(variable_numeric),"\n\n")

#Integer Data type
variable_integer<- 133L
cat(variable_integer,"\n")
cat("The data type of variable_integer is ",class(variable_integer),"\n\n")

#Complex Data type
variable_complex<- 3+2i
cat(variable_complex,"\n")
cat("The data type of variable_complex is ",class(variable_complex),"\n\n")

#Character Data type
variable_char<- "Learning r programming"
cat(variable_char,"\n")
cat("The data type of variable_char is ",class(variable_char),"\n\n")
```

Data Structure in R

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Data structures in R programming are tools for holding multiple values.

R's base data structures are often organized by their dimensionality (1D, 2D, or nD) and whether they're homogeneous (all elements must be of the identical type) or heterogeneous (the elements are often of various types). This gives rise to the six data types which are most frequently utilized in data analysis.

The most essential data structures used in R include:

- Vectors
- Lists
- Dataframes
- Matrices
- Arrays
- Factors

Vectors

Vector is one of the basic data structures in R programming. It is homogenous in nature, which means that it only contains elements of the same data type. Data types can be numeric, integer, character, complex or logical.

The vector in R programming is created using the c() function. Coercion takes place in a vector from lower to top, if the elements passed are of different data types from Logical to Integer to Double to Character.

The typeof() function is used to check the data type of the vector, and class() function is used to check the class of a vector.

For example:

```
Vec1 <- c(44, 25, 64, 96, 30)

Vec2 <- c(1, FALSE, 9.8, "hello world")

typeof(Vec1)

typeof(Vec2)
```

Output:

```
[1] "double"

[1] "character"
```

To delete a vector, we simply do the following

```
Vec1 <- NULL

Vec2 <- NULL
```

List in R Programming

A list in R programming is a non-homogenous data structure, which implies that it can contain elements of different data types. A list is a generic object consisting of an ordered collection of objects. Lists are heterogeneous data structures. These are also one-dimensional data structures. It accepts numbers, characters, lists, and even matrices and functions inside it. It is created using the list() function.

For example:

```
list1<- list("Sam", "Green", c(8,2,67), TRUE, 51.99, 11.78,FALSE)

print(list1)
```

Matrix in R Programming

The matrix in R programming is a 2-dimensional data structure that is homogenous in nature, which means that it only accepts elements of the same data type. Coercion takes place if elements of different data types are passed. It is created using the matrix() function.

The basic syntax to create a matrix is given below:

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

where,

data = the input element of a matrix given as a vector.

nrow = the number of rows to be created.

ncol = the number of columns to be created.

byrow = the row-wise arrangement of the elements instead of column-wise

dimnames = the names of columns/rows to be created.

For example:

```
M1 <- matrix(c(1:9), nrow = 3, ncol =3, byrow= TRUE)

print(M1)
```

Output:

```
[,1] [,2] [,3]

[1,]    1    2    3

[2,]    4    5    6

[3,]    7    8    9
```

Arrays

There is another type of data objects which can store data in more than two dimensions known as arrays. "An array is a collection of a similar data type with contiguous memory allocation." Suppose, if we create an array of dimension (2, 3, 4) then it creates four rectangular matrices of two rows and three columns. In R, an array is created with the help of array() function. This function takes a vector as an input and uses the value in the dim parameter to create an array.

Example:

```
a <- array(c('green','yellow'),dim = c(3,3,2))
print(a)
```

When we execute the above code, it produces the following result −

```
, , 1

     [,1]     [,2]     [,3]
[1,] "green"  "yellow" "green"
[2,] "yellow" "green"  "yellow"
[3,] "green"  "yellow" "green"

, , 2

     [,1]     [,2]     [,3]
[1,] "yellow" "green"  "yellow"
[2,] "green"  "yellow" "green"
[3,] "yellow" "green"  "yellow"
```

Factor

Factors in R programming are used in data analysis for statistical modeling. They are used to categorize unique values in columns, like “Male, “Female”, “TRUE”, “FALSE” etc., and store them as levels. They can store both strings and integers. They are useful in columns that have a limited number of unique values.

Factors can be created using the factor() function and they take vectors as inputs.

Example:

```
fac = factor(c("Male", "Female", "Male", "Male", "Female", "Male", "Female"))

print(fac)
```

Output:

```
[1] Male   Female Male   Male   Female Male   Female

Levels: Female Male
```

Dataframes

Dataframes are generic data objects of R which are used to store the tabular data. Dataframes are the foremost popular data objects in R programming because we are comfortable in seeing the data within the tabular form. They are two-dimensional, heterogeneous data structures. These are lists of vectors of equal lengths.

Data frames have the following constraints placed upon them:

- A data-frame must have column names and every row should have a unique name.
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.

To create a data frame we use the data.frame() function.

Some DataFrame Functions

- head() - shows first 6 rows
- tail() - shows last 6 rows
- dim() - returns the dimensions of data frame (i.e. number of rows and number of columns)
- nrow() - number of rows
- ncol() - number of columns
- str() - structure of data frame - name, type and preview of data in each column
- names() or colnames() - both show the names attribute for a data frame
- sapply(dataframe, class) - shows the class of each column in the data frame

Example:

```
Name = c("Amiya", "Raj", "Asish")

Language = c("R", "Python", "Java")

Age = c(22, 25, 45)

df = data.frame(Name, Language, Age)

print(df)
```

Output:

```
   Name   Language    Age
1  Amiya       R        22
2   Raj      Python     25
3  Asish     Java       45
```

Which() in R

The which() function in R returns the position or the index of the value which satisfies the given condition. The Which() function in R gives you the position of the value in a logical vector. The position can be of anything like rows, columns and even vector as well.

The Syntax: which(x, arr.ind = FALSE)

- x can be any logical test vector
- arr.ind is an optional parameter primarily uses for working with matrices (multiple dimension arrays); it indicates you want the array indices (both x and y data points, for example, vs a single index for one dimensional arrays).
- You can use the function which in R to scan a data structure and identify the elements within that data structure which meet a specific condition. The which function returns the indices of the matching items in the data object.

str() in R

str() function in R Language is used for compactly displaying the internal structure of a R object. It can display even the internal structure of large lists which are nested. It provides one liner output for the basic R objects letting the user know about the object and its constituents. It can be used as an alternative to summary() but str() is more compact than summary(). It gives information about the rows(observations) and columns(variables) along with additional information like the names of the columns, class of each columns followed by few of the initial observations of each of the columns.

Vector	List
It has contiguous memory.	While it has non-contiguous memory.
It is synchronized.	While it is not synchronized.
Vector may have a default size.	List does not have default size.
In vector, each element only requires the space for itself only.	In list, each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list.

Vector	List
Insertion at the end requires constant time but insertion elsewhere is costly.	Insertion is cheap no matter where in the list it occurs.
Vector is thread safe.	List is not thread safe.
Deletion at the end of the vector needs constant time but for the rest it is O(n).	Deletion is cheap no matter where in the list it occurs.
Random access of elements is possible.	Random access of elements is not possible.
Iterators become invalid if elements are added to or removed from the vector.	Iterators are valid if elements are added to or removed from the list.

R length Function

length() function gets or sets the length of a vector (list) or other objects.

Get vector length:

```
> x <- c(1,2,5,4,6,1,22,1)
> length(x)
[1] 8
```

length() function can be used for all R objects. For an environment it returns the object number in it. NULL returns 0. Most other objects return length 1.

View() Function in R

The View() function in R invokes a spreadsheet-style data viewer on a matrix-like R object. To view all the contents of a defined object, use the View() function.

Syntax

View(ObjectName, title)

Parameters

ObjectName: It is an R object coerced to a data frame with non-zero numbers of rows and columns.

title: It is a title for a viewer window. Defaults to the ObjectName prefixed by Data.

Return Value

It returns an Invisible NULL. The functions put up a window and return immediately: the window can be closed via its controls or menus.

Getting Information on a Dataset

There are several functions for listing the contents of an object or dataset.

- To list the objects in the working environment, use the ls() function.
- To list the variables in your data, use the names() function.
- To list the structure of your data, use the str() method.
- To list the levels of factors in your data, use the levels() function.
- To get the dimension of the object, use the dim() method.
- To get the class of the object, use the class() method.
- To get the first n number of rows and columns of the dataset, use the head() function.
- To get the last n number of rows and columns of the dataset, use the tail() function.

Level()

levels() function in R Language is used to get or set the levels of a factor.

Syntax: levels(x)

Parameters:

x: Factor Object

Example 1:

```
# R program to get levels of a factor

# Creating a factor

gender <- factor(c("female", "male", "male", "female"));

gender
```

```
# Calling levels() function
# to get the levels
levels(gender)
```

Output:

```
[1] female male   male   female
Levels: female male
[1] "female" "male"
```

Files in R Programming

So the two most common operations that can be performed on a file are:

- Importing/Reading Files in R
- Exporting/Writing Files in R

Reading Files in R

When a program is terminated, the entire data is lost. Storing in a file will preserve our data even if the program terminates. If we have to enter a large number of data, it will take a lot of time to enter them all. However, if we have a file containing all the data, we can easily access the contents of the file using a few commands in R. You can easily move your data from one computer to another without any changes. So those files can be stored in various formats. It may be stored in .txt(tab-separated value) file, or in a tabular format i.e .csv(comma-separated value) file or it may be on internet or cloud. R provides very easier methods to read those files.

file.choose(): In R it's also possible to choose a file interactively using the function file.choose(), and if you're a beginner in R programming then this method is very useful for you.

Example:

```
# R program reading a text file using file.choose()
myFile = read.delim(file.choose(), header = FALSE)
# If you use the code above in RStudio
# you will be asked to choose a file
print(myFile)
```

read.csv(): read.csv() is used for reading “comma separated value” files (“.csv”). In this also the data will be imported as a data frame.

Example:

```
# R program to read a file in table format

# Using read.csv()

myData = read.csv("basic.csv")

print(myData)
```

file.choose(): You can also use file.choose() with read.csv() just like before.

Example:

```
# R program to read a file in table format

# Using file.choose() inside read.csv()

myData = read.csv(file.choose())

print(myData)
```

The header Argument

The default for read.csv(...) is to set the header argument to TRUE. This means that the first row of values in the .csv is set as header information (column names). If your data set does not have a header, set the header argument to FALSE.

The stringsAsFactors Argument

In older versions of R (prior to 4.0) this was perhaps the most important argument in read.csv(), particularly if you were working with categorical data. This is because the default behavior of R was to convert character strings into factors, which may make it difficult to do such things as replace values. It is important to be aware of this behaviour, which we will demonstrate.

Writing Data to CSV files

CSV stands for Comma Separated Values. These files are used to handle a large amount of statistical data. Following is the syntax to write to a CSV file:

Syntax:

```
write.csv(my_data, file = "my_data.csv")
```

write.csv2(my_data, file = "my_data.csv")

Writing Data to Excel files

To write data to excel we need to install the package known as “xlsx package”, it is basically a java based solution for reading, writing, and committing changes to excel files. It can be installed as follows:

install.packages("xlsx")

and can be loaded as:

library("xlsx")

General syntax of using it is:

write.xlsx(my_data, file = “result.xlsx”, sheetName = “my_data”, append = FALSE).

rbind() Function

rbind() function in R Language is used to combine specified Vector, Matrix or Data Frame by rows.

Syntax: rbind(x1, x2, …, deparse.level = 1)

Parameters:

x1, x2: vector, matrix, data frames

deparse.level: This value determines how the column names generated. The default value of deparse.level is 1.

Example 1:

Initializing two vectors

x <- 2:7

y <- c(2, 5)

Calling rbind() function

rbind(x, y)

Output:

[, 1] [, 2] [, 3] [, 4] [, 5] [, 6]

x 2 3 4 5 6 7

y 2 5 2 5 2 5

cbind() Function

cbind() function in R Language is used to combine specified Vector, Matrix or Data Frame by columns.

Syntax: cbind(x1, x2, …, deparse.level = 1)

Parameters:

x1, x2: vector, matrix, data frames

deparse.level: This value determines how the column names generated. The default value of deparse.level is 1.

Example 1:

```
# Initializing two vectors
x <- 2:7
y <- c(2, 5)
# Calling cbind() function
cbind(x, y)
```

Output:

```
     x y
[1, ] 2 2
[2, ] 3 5
[3, ] 4 2
[4, ] 5 5
[5, ] 6 2
[6, ] 7 5
```

summary() Function in R

The summary() is an inbuilt generic function in R used to produce result summaries of various model fitting functions. The summary() method entreats specific methods that depend on the class of the first argument.

Syntax: summary(object, maxsum)

Parameters:

object: R object

maxsum: integer value which indicates how many levels should be shown for factors

Example 1:

```
vec <- 1:5

vec

cat("The summary() of vector is", "\n")

summary(vec)
```

Output

```
[1] 1 2 3 4 5

The summary() of vector is

Min. 1st Quantile  Median  Mean  3rd Quantile   Max.

1       2           3     3        4        5
```

1. Measure of Central Tendency

Measures of central tendency is one of the most popular techniques used for data summarization of a series. It tells about how the group of data is clustered around the centre value of the distribution. When you have a large amount of data, then in order to manage them, we use the method called averages. The purpose of its computation is to identify the most representative value among the data items. Therefore, we would deal only with single representation of data rather than having a very large series of observations. This is helpful for comparison purpose and also to understand the characteristics of the series. Averages are also considered under data exploration stages before building statistical models for deriving solutions to your problem. As you know, the application of measures of central tendency is meaningful if it is only applied to a specific form of data. Initially we need to study the data limitation before choosing an

appropriate measures of central tendency. The following are the most popular measures of central tendency.

Central tendency performs the following measures:

• Arithmetic Mean

• Geometric Mean

• Harmonic Mean

• Median

• Mode

1. Arithmetic Mean

The arithmetic mean is simply called the average of the numbers which represents the central value of the data distribution. It is calculated by adding all the values and then dividing by the total number of observations. The computation of arithmetic mean depends on each observation of the series. Hence it is influenced by extreme observation if the series has an outlier. Since arithmetic mean acts as a representation of series of data so it should be only considered for homogeneous observation. For any larger variation in the series, arithmetic mean may not be a good measure of central tendency. Using R software one could easily obtain the value of the mean using summary function. In R language, arithmetic mean can be calculated by mean() function.

Syntax: mean(x, trim, na.rm = FALSE)

***Parameters*:**

x: Represents object

trim: Specifies number of values to be removed from each side of object before calculating the

mean. The value is between 0 to 0.5

na.rm: If TRUE then removes the NA value from x

Example:

```
x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23) # Defining vector

print(mean(x)) # Print mean
```

Output:

```
[1] 21.5
```

Application: It is used in report card to find the final grade of students in a class based on multiple subject marks. It is also used to find the average age of cabin crew in flights.

2. Geometric Mean

The geometric mean is a type of mean that is computed by multiplying all the data values and thus, shows the central tendency for given data distribution. prod() and length() function helps in finding the geometric mean for given set of numbers as there is no direct function for geometric mean. Geometric mean is the only average that is recommended for finding average growth (decline) rates. It is defined as the nth root of the product of n terms. Since it is defined in product terms so the observation shouldn't be having zero or negative values. Geometric mean is not easy to understand. The presence of few extreme values has no considerable effect on geometric mean. It is also popularly used in banking and insurance sector for finding rates of interest and rates of depreciation, etc. We don't have a built-in function in R for its computation but one could find it by using its formula directly in R platform.

Syntax: prod(x)^(1/length(x))

where,

prod() function returns the product of all values present in vector x

length() function returns the length of vector x

Example:

```
# Defining vector

x <- c(1, 5, 9, 19, 25)

# Print Geometric Mean

print(prod(x)^(1 / length(x)))
```

Output:

```
[1] 7.344821
```

Applications: It is used to calculate annual returns on portfolio of stocks. It is also used in calculations of index numbers measuring changes in variables.

3. Harmonic Mean

Harmonic mean is another type of mean used as another measure of central tendency. It is computed as reciprocal of the arithmetic mean of reciprocals of the given set of values. Harmonic Mean is based on mathematic computation like Arithmetic Mean and Geometric Mean. It is used only with quantitative data. It is defined as the number of observations over the sum of reciprocals of given values. It is however complex to understand and is not a popular measure of central tendency. It is capable of further mathematical treatment as it depends upon every observation in the series. Harmonic Mean is popularly used for finding average distances of a moving body that changes its position from place to place. We don't have R function for finding harmonic mean, therefore we need to use its formula directly to find its mean.

Example:

```
x <- c(1, 5, 8, 10) # Defining vector

print(1 / mean(1 / x)) # Print Harmonic Mean
```

Output:

```
[1] 2.807018
```

Applications: Harmonic Mean is popularly used for finding average distances of a moving body that changes its position from place to place. It is also used in measuring time, speed, price, etc.

4. Median

Median in statistics is another measure of central tendency which represents the middlemost value of a given set of values. In R language, median can be calculated by median() function. Median is referred to as positional average. It is a value that divides the distribution of data into two equal halves. Therefore, one would find a 50% of data above the median value and another 50% of data below the median value. As being a positional average, the computation of Median does not depend on extreme observations. Hence median is not influenced by outliers. We could find median value using summary function in R. The random Forest library can be used to impute the missing values using Median for numeric variables.

Syntax: median(x, na.rm = FALSE)

Parameters:

x: It is the data vector

na.rm: If TRUE then removes the NA value from x

Example:

```
x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23) # Defining vector

median(x) # Print Median
```

Output:

```
[1] 21.5
```

Applications: It is a better measure of accessing salary, net worth, etc. It is useful for situations where calculations on data is not possible but data can be ordered like IQ, happiness, etc.

5. Mode

The mode of a given set of values is the value that is repeated most in the set. There can exist multiple mode values in case if there are two or more values with matching maximum frequency. Mode is the value which occurs frequently in a data series. It is easy to compute and is not

influenced by extreme observation. The main advantage of Mode is when the variable are categorical, where model value could be determined unlike arithmetic mean and median value. Mode is used for missing value imputation for categorical variables using randomForest library in R. Model can be easily located graphically. You shouldn't be surprised that the R's mode function (mode ()) does not provide a model value. It shows the datatype of the particular variable which does not comply with our standard expectation. So how one would find mode using R software? We need to use table function for finding mode. As you know the table function in R provides frequency distribution of the variable. Thus the value with highest frequency is a modal value.

Example 1: Single-mode value

In R language, there is no function to calculate mode. So, modifying the code to find out the mode for a given set of values.

```
x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23, 29, 56, 37, 45, 1, 25, 8)

#Defining vector

y <- table(x) # Generate frequency table

print(y) # Print frequency table

m <- names(y)[which(y == max(y))] # Mode of x

print(m) # Print mode
```

Output:

```
x

1 3 5 7 8 12 13 14 20 23 25 29 37 39 40 45 56

1 1 1 1 1 1 1 1 1 4 1 1 1 1 1 1 2

[1] "23"
```

Example 2: Multiple Mode values

```
x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40,23, 14, 12, 56, 23, 29, 56, 37, 45, 1, 25, 8, 56, 56)

# Defining vector

y <- table(x) # Generate frequency table

print(y) # Print frequency table

m <- names(y)[which(y == max(y))] # Mode of x

print(m) # Print mode
```

Output:

x

1 3 5 7 8 12 13 14 20 23 25 29 37 39 40 45 56

1 1 1 1 1 1 1 1 4 1 1 1 1 1 1 4

[1] "23" "56"

Applications: It is used in situations like voting. It is also used in Business forecasting of orders for multitype products like shirts of different sizes.

B. Variability in R Programming

Variability (also known as Statistical Dispersion) is another feature of descriptive statistics. Measures of central tendency and variability together comprise of descriptive statistics. Variability shows the spread of a data set around a point.

Example: Suppose, there exist 2 data sets with the same mean value:

A = 4, 4, 5, 6, 6

Mean(A) = 5

B = 1, 1, 5, 9, 9

Mean(B) = 5

So, to differentiate among the two data sets, R offers various measures of variability.

Measures of Variability/Dispersion

Following are some of the measures of variability that R offers to differentiate between data sets:

• Variance

• Standard Deviation

• Range

• Mean Deviation

• Interquartile Range

1. Variance

Variance is a measure that shows how far is each value from a particular point, preferably mean value. Mathematically, it is defined as the average of squared differences from the mean value.

In the R language, there is a standard built-in function to calculate the variance of a data set.

Syntax: var(x)

Parameter:

x: It is data vector

Example:

```
x <- c(5, 5, 8, 12, 15, 16) # Defining vector

print(var(x)) # Print variance of x
```

Output:

```
[1] 23.76667
```

Application: it is a measure of how far a set of numbers is spread out from their average value. Variance is an important tool in the sciences, where statistical analysis of data is common.

2. **Standard Deviation**

Standard deviation in statistics measures the spreadness of data values with respect to mean and mathematically, is calculated as square root of variance. In R language, there is no standard builtin function to calculate the standard deviation of a data set. So, modifying the code to find the standard deviation of data set.

Example:

```
x <- c(5, 5, 8, 12, 15, 16) # Defining vector

d <- sqrt(var(x)) # Standard deviation

print(d) # Print standard deviation of x
```

Output:

```
[1] 4.875107
```

Applications: In physical science, for example, the reported standard deviation of a group of repeated measurements gives the precision of those measurements. Standard deviation is often used as a measure of the risk associated with price-fluctuations of a given asset, or the risk of a portfolio of assets.

3. Range

Range is the difference between maximum and minimum value of a data set. In R language, max() and min() is used to find the same, unlike range() function that returns the minimum and maximum value of data set.

Example:

```
x <- c(5, 5, 8, 12, 15, 16) # Defining vector

print(range(x)) # range() function output

# Using max() and min() function to calculate the range of data set

print(max(x)-min(x))
```

Output:

```
[1] 5 16

[1] 11
```

Applications: It can be used to find the difference between the low and high prices of security over a certain period of time. It can also be used as a measure of the volatility of a security. It is also used in weather forecasts, temperature range, quality control, fluctuation in share prices, etc.

4. Mean Deviation

Mean deviation is a measure calculated by taking an average of the arithmetic mean of the absolute difference of each value from the central value. Central value can be mean, median, or mode. In R language, there is no standard built-in function to calculate mean deviation. So, modifying the code to find mean deviation of the data set.

Example:

```
x <- c(5, 5, 8, 12, 15, 16) # Defining vector

md <- sum(abs(x-mean(x)))/length(x) # Mean deviation

print(md) # Print mean deviation
```

Output:

```
[1] 4.166667
```

Applications: It helps to see the difference between each of the scores and the beginning average scores. It helps the teacher to see if the test was too hard, too easy, or just right based upon the mathematical outcomes. Biologists use this to compare the differences in animal weight to decide what a healthy weight might be.

5. Interquartile Range

Interquartile Range is based on splitting a data set into parts called as quartiles. There are 3 quartile values (Q1, Q2, Q3) that divide the whole data set into 4 equal parts. Q2 specifies the median of the whole data set. Mathematically, the interquartile range is depicted as:

IQR = Q3 – Q1

where,

Q3 specifies the median of n largest values

Q1 specifies the median of n smallest values

In R language, there is built-in function to calculate the interquartile range of data set.

***Syntax*:** IQR(x)

Parameter:

x: It specifies the data set

Example:

x <- c(5, 5, 8, 12, 15, 16) # Defining vector

print(IQR(x)) # Print Interquartile range

Output:

[1] 8.5

***Applications*:** Some companies use quartiles to benchmark the companies. It is used in businesses as a marker for their income rates. They are also used to identify outliers in the dataset

C. Skewness and Kurtosis

In statistics, skewness and kurtosis are the measures which tell about the shape of the data distribution or simply, both are numerical methods to analyze the shape of data set unlike, plotting graphs and histograms which are graphical methods. These are normality tests to check the irregularity and asymmetry of the distribution. To calculate skewness and kurtosis in R language, moments package is required.

Skewness

Skewness is a statistical numerical method to measure the asymmetry of the distribution or data set. It tells about the position of the majority of data values in the distribution around the mean value.

Formula:

$$\gamma_1 = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{3/2}}$$

Applications: It can be used for economical analysis in finance and investing. Skewness is a descriptive statistic that can be used in conjunction with the histogram and the normal quantile plot to characterize the data or distribution.

Generally, we have three types of skewness.

▪ ***Symmetrical***: When the skewness is close to 0 and the mean is almost the same as the median

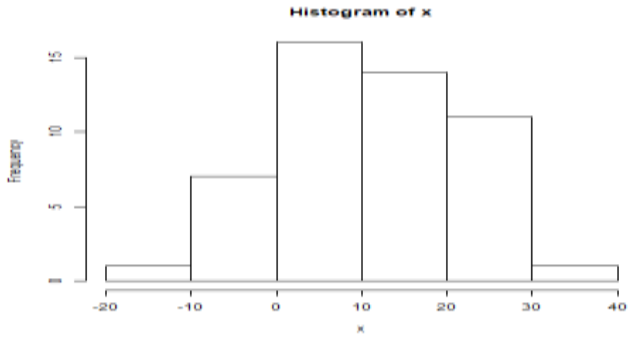
Example:

```
Library(moments) #required for skewness() function
x <- rnorm(50,10,10) #defininf normally distributed data vector
png(file=”zeroskewness.png”) #output to be present as PNG file
print(skewnwss(x)) #print the skewness of distribution
hist(x) # histogram of distribution
dev.off() # saving the file
```

Output:

[1] -0.02991511

Graphical Representation:



▪ ***Negative skew***: When the left tail of the histogram of the distribution is longer and the majority of the observations are concentrated on the right tail. In this case, we can use also the term “left-skewed” or “left-tailed”. and the median is greater than the mean.

Example:

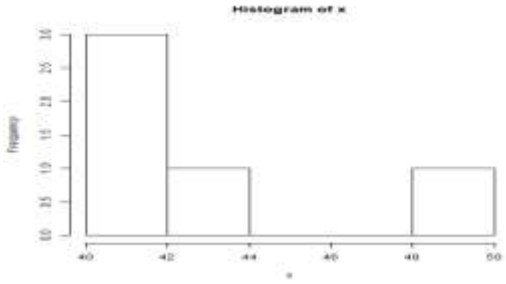
```
library(moments)        # Required for skewness() function
x <- c(10, 11, 21, 22, 23, 25)          # Defining data vector
png(file = "negativeskew.png")          # output to be present as PNG file
print(skewness(x))      # Print skewness of distribution
hist(x)          # Histogram of distribution
dev.off()        # Saving the file
```

▪ ***Positive skew***: When the right tail of the histogram of the distribution is longer and the

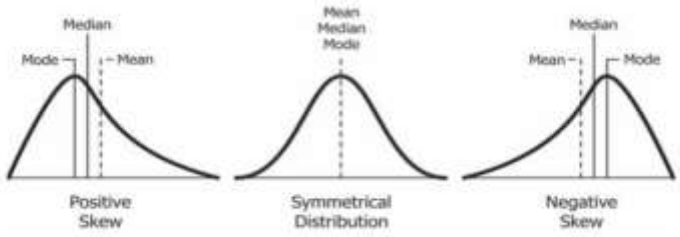
majority of the observations are concentrated on the left tail. In this case, we can use also the term “right-skewed” or “right-tailed”. and the median is less than the mean.

Example:

```
library(moments)        # Required for skewness() function
x <- c(40, 41, 42, 43, 50)        # Defining data vector
png(file = "positiveskew.png")          # output to be present as PNG file
print(skewness(x))      # Print skewness of distribution
hist(x)           # Histogram of distribution
dev.off()         # Saving the file
```



The graph below describes the three cases of skewness. Focus on the Mean and Median.



Kurtosis

Kurtosis is a numerical method in statistics that measures the sharpness of the peak in the data distribution.

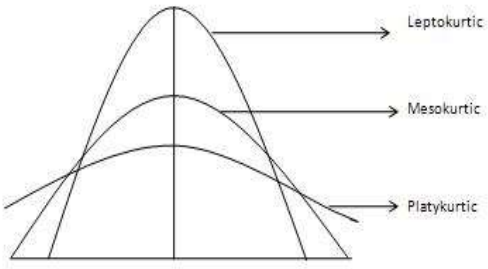
Formula:

$$\gamma_2 = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^4}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^2}$$

In statistics, we use the kurtosis measure to describe the “tailedness” of the distribution as it describes the shape of it. It is also a measure of the “peakedness” of the distribution. A high

kurtosis distribution has a sharper peak and longer fatter tails, while a low kurtosis distribution has a more rounded pean and shorter thinner tails.

Applications: kurtosis is a useful measure of whether there is a problem with outliers in a data set. kurtosis describes the shape of a probability distribution.



The main three types of kurtosis.

▪ ***Mesokurtic***: This is the normal distribution with a value =3.

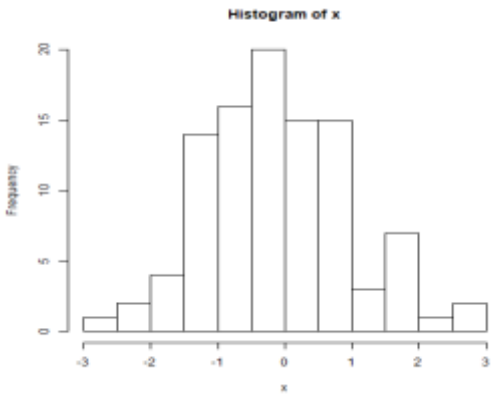
Example:

```
library(moments)        # Required for kurtosis() function
x <- rnorm(100)         # Defining data vector
png(file = "mesokurtic.png")          # output to be present as PNG file
print(kurtosis(x))      # Print skewness of distribution
hist(x)          # Histogram of distribution
dev.off()        # Saving the file
```

Output:

```
[1] 2.963836
```

Graphical Representation



▪ ***Leptokurtic:*** This distribution has fatter tails and a sharper peak. The kurtosis is "positive" with a value greater than 3

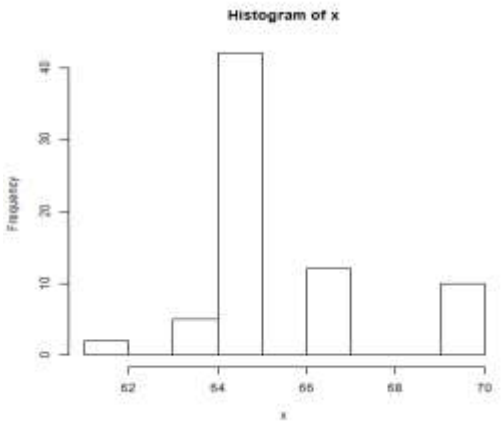
Example:

```
library(moments) # Required for kurtosis() function
x <- c(rep(61, each = 2), rep(64, each = 5),rep(65, each = 42), rep(67, each = 12),
rep(70, each = 10)) # Defining data vector
png(file = "leptokurtic.png") # output to be present as PNG file
print(kurtosis(x)) # Print skewness of distribution
hist(x) # Histogram of distribution
dev.off() # Saving the file
```

Output:

```
[1] 3.696788
```

Graphical Representation



▪ ***Platykurtic:*** The distribution has a lower and wider peak and thinner tails. The kurtosis is "negative" with a value less than 3

Example:

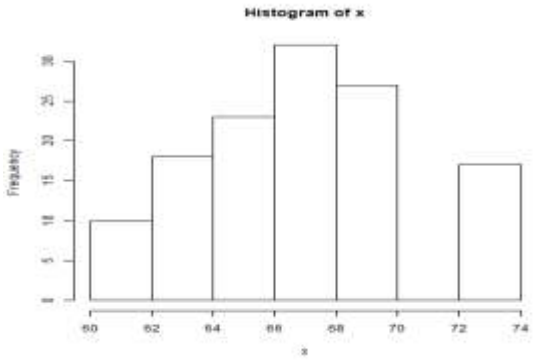
```
library(moments) # Required for kurtosis() function
x <- c(rep(61, each = 10), rep(64, each = 18), rep(65, each = 23), rep(67, each = 32),
rep(70, each = 27),rep(73, each = 17)) # Defining data vector
```

```
png(file = "platykurtic.png") # output to be present as PNG file
print(kurtosis(x)) # Print skewness of distribution
hist(x) # Histogram of distribution
dev.off() # Saving the file
```

Output:

[1] 2.258318

Graphical Representation



MODULE 2

How to change Row Names of DataFrame in R ?

Method 1 : using rownames()

A data frame's rows can be accessed using rownames() method in the R programming language. We can specify the new row names using a vector of numerical or strings and assign it back to the rownames() method. The data frame is then modified reflecting the new row names. The number of items in the vector should be equivalent to the number of rows in the data frame.

Syntax:

```
rownames(dataframe)
```

Example:

```
df <- data.frame(First = c(1,2,3,4) ,
                 Second = c("a","ab","cv","dsd"),
                 Third=c(7:10))                    # declaring a data frame
print ("Original DataFrame : ")                    # print original data frame
print (df)
rownames <- rownames(df)                           # printing original rownames of data frame
print ("Original row names ")
print (rownames)
rownames(df) <- c("Row1","Row2","Row3","Row4")     # changing row names of data frame
print ("Modified DataFrame : ")                    # print changed data frame
print (df)
```

Output:

```
[1] "Original DataFrame : "
      First Second Third
1     1      a     7
2     2     ab     8
```

3 3 cv 9

4 4 dsd 10

[1] "Original column names "

[1] "1" "2" "3" "4"

[1] "Modified DataFrame : "

First Second Third

Row1 1 a 7

Row2 2 ab 8

Row3 3 cv 9

Row4 4 dsd 10

- Column names are addressed by unique names.

Method 1: using colnames() method

colnames() method in R is used to rename and replace the column names of the data frame in R.

The columns of the data frame can be renamed by specifying the new column names as a vector. The new name replaces the corresponding old name of the column in the data frame. The length of new column vector should be equivalent to the number of columns originally. Changes are made to the original data frame.

Syntax:

colnames(df) <- c(new_col1_name,new_col2_name,new_col3_name)

Example:

```
df = data.frame(
col1 = c('A', 'B', 'C', 'J', 'E', NA,'M'),
col2 = c(12.5, 9, 16.5, NA, 9, 20, 14.5),
col3 = c(NA, 3, 2, NA, 1, NA, 0))          # declaring the columns of data frame
print("Original data frame : ")            # printing original data frame
print(df)
print("Renaming columns names ")
```

```
colnames(df) <- c('C1','C2','C3')        # assigning new names to the columns of the data frame
print("New data frame : ")               # printing new data frame
print(df)
```

Output:

```
[1] “Original data frame : “
    col1 col2 col3
1   A 12.5   NA
2   B  9.0    3
3   C 16.5    2
```

Data Extraction in R

In data extraction, the initial step is data pre-processing or data cleaning. In data cleaning, the task is to transform the dataset into a basic form that makes it easy to work with. One characteristic of a clean/tidy dataset is that it has one observation per row and one variable per column.

The next step in data extraction is data manipulation. In data manipulation, the task is to modify the data to make it easier to read and more organized. We manipulate the data for data analysis and data visualization. Data manipulation is also used with the term ‘data exploration’ which involves organizing data using the available sets of variables.

At times, the data collection process done by machines involves lots of errors and inaccuracies in reading. Data manipulation is also used to remove these inaccuracies and make data more accurate and precise.

Exploring Raw Data

Raw data is data collected from a source, which has not yet been processed for usage. Typically, the readily available data is not in a state in which it can be used efficiently for data extraction. Such data is difficult to manipulate and often needs to be processed in some way, before it can be used for data analysis and data extraction in general, and is referred to as raw data or source data.

Data Manipulation in R

What are dplyr and tidyr?

The dplyr package contains various functions that are specifically designed for data extraction and data manipulation. These functions are preferred over the base R functions because the former process data at a faster rate and are known as the best for data extraction, exploration, and transformation.

Some of the important functions for data manipulation in R are as follows:

- select(): to select columns (variables)
- filter(): to filter (subset) rows
- mutate(): to create new columns
- summarise(): to summarize (or aggregate) data
- group_by(): to group data
- arrange(): to sort data
- join(): to join data frames

The package **tidyr** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups (e.g., a time period, an experimental unit like a plot or a batch number). Moving back and forth between these formats is non-trivial, and tidyr gives you tools for this and more sophisticated data manipulation.

The sole purpose of the tidyr package is to simplify the process of creating tidy data. Tidy data describes a standard way of storing data that is used wherever possible throughout the tidyverse. If you once make sure that your data is tidy, you'll spend less time punching with the tools and more time working on your analysis.

tidyr and dplyr packages provide fundamental functions for Cleaning, Processing, & Manipulating Data

tidyr:

- gather()
- spread()
- separate()
- unite()

Gather()

The gather() function from the tidyr package can be used to "gather" a key-value pair across multiple columns. The gather() function will take multiple columns and collapse them into key-value pairs, duplicating all other columns as needed.

This function uses the following basic ***syntax***:

gather(data, key = "key", value = "value", …, na.rm = FALSE, convert = FALSE, factor_key = FALSE)

where:

data : Name of the data frame

key : Name of the key column to create

value : Name of the value column to create

……. : names of columns to gather (or not gather)

na.rm : option to remove observations with missing values (represented by NAs)

convert : if TRUE will automatically convert values to logical, integer, numeric, complex or factor as appropriate

Example:

```
#create data frame
df <- data.frame(player=c('A', 'B', 'C', 'D'),
                 year1=c(12, 15, 19, 19),
                 year2=c(22, 29, 18, 12))


#view data frame
df
  player year1 year2
1      A    12    22
2      B    15    29
3      C    19    18
4      D    19    12
```

We can use the ***gather()*** function to create two new columns called “year” and “points” as follows:

```
library(tidyr)
#gather data from columns 2 and 3
gather(df, key="year", value="points", 2:3)
  player  year points
1      A year1     12
2      B year1     15
3      C year1     19
4      D year1     19
```

5 A year2 22

6 B year2 29

7 C year2 18

8 D year2 12

Spread()

spread() function: It helps in reshaping a longer format to a wider format. The spread() function spreads a key-value pair across multiple columns.

Syntax:

spread(data, key, value, fill = NA, convert = FALSE)

where,

data: A data frame.

Key: Column names or positions.

Value: Column names or positions.

Fill: If set, missing values will be replaced with this value.

Convert: If TRUE, type.convert() with asis=TRUE will be run on each of the new columns.

Example:

```
#create data frame
df <- data.frame(player=rep(c('A', 'B'), each=4),
                 year=rep(c(1, 1, 2, 2), times=2),
                 stat=rep(c('points', 'assists'), times=4),
                 amount=c(14, 6, 18, 7, 22, 9, 38, 4))
#view data frame
df
  player year    stat amount
1      A    1  points     14
2      A    1 assists      6
3      A    2  points     18
```

4 A 2 assists 7

5 B 1 points 22

6 B 1 assists 9

7 B 2 points 38

8 B 2 assists 4

We can use the spread() function to turn the values in the stat column into their own columns:

```
library(tidyr)

#spread stat column across multiple columns

spread(df, key=stat, value=amount)
```

player year assists points

1 A 1 6 14

2 A 2 7 18

3 B 1 9 22

4 B 2 4 38

Separate()

separate() function: It converts longer data to a wider format. The separate() function turns a single character column into multiple columns.

Syntax:

```
separate(data, col, into, sep = " ", remove = TRUE, convert = FALSE)
```

where,

data: A data frame.

Col: Column name or position.

Into: Names of new variables to create as character vector. Use NA to omit the variable in the output.

Sep: The separator between the columns.

Remove: If set TRUE, it will remove input column from the output data frame.

Convert: If TRUE, will run type.convert() with as.is = TRUE on new columns.

Example:

```
#create data frame
df <- data.frame(player=c('A', 'A', 'B', 'B', 'C', 'C'),
                 year=c(1, 2, 1, 2, 1, 2),
                 stats=c('22-2', '29-3', '18-6', '11-8', '12-5', '19-2'))
#view data frame
df
  player year stats
1      A    1  22-2
2      A    2  29-3
3      B    1  18-6
4      B    2  11-8
5      C    1  12-5
6      C    2  19-2
```

We can use the separate() function to separate the stats column into two new columns called "points" and "assists" as follows:

```
library(tidyr)
#separate stats column into points and assists columns
separate(df, col=stats, into=c('points', 'assists'), sep='-')
  player year points assists
1      A    1     22       2
2      A    2     29       3
3      B    1     18       6
4      B    2     11       8
5      C    1     12       5
6      C    2     19       2
```

unite() function

It merges two columns into one column. The unite() function is a convenience function to paste together multiple variable values into one. In essence, it combines two variables of a single observation into one variable.

Syntax:

unite(data, col, …, sep = "_", remove = TRUE)

data: A data frame.

Col: The name of the new column.

….: A selection of desired columns. If empty, all variables are selected.

sep A separator to use between values.

remove: :If TRUE, remove input columns from output data frame.

Example:

```
#create data frame
df <- data.frame(player=c('A', 'A', 'B', 'B', 'C', 'C'),
                 year=c(1, 2, 1, 2, 1, 2),
                 points=c(22, 29, 18, 11, 12, 19),
                 assists=c(2, 3, 6, 8, 5, 2))
#view data frame
df
```

	player	year	points	assists
1	A	1	22	2
2	A	2	29	3
3	B	1	18	6
4	B	2	11	8
5	C	1	12	5
6	C	2	19	2

We can use the unite() function to unite the "points" and "assists" columns into a single column:

```
library(tidyr)
#unite points and assists columns into single column
```

unite(df, col='points-assists', c('points', 'assists'), sep='-')

```
  player year points-assists
1      A    1         22-2
2      A    2         29-3
3      B    1         18-6
4      B    2         11-8
5      C    1         12-5
6      C    2         19-2
```

Create Subset of a Dataframe in R

subset() function in R Language is used to create subsets of a Data frame. This can also be used to drop columns from a data frame.

Syntax: subset(df, expr)

Parameters:

df: Data frame used

expr: Condition for subset

Example:

```
df<-data.frame(row1 = 0:2, row2 = 3:5, row3 = 6:8)          # Creating a Data Frame
print ("Original Data Frame")
print (df)
df1<-subset(df, select = row2)          # Creating a Subset
print("Modified Data Frame")
print(df1)
```

Output:

```
[1] "Original Data Frame"
  row1 row2 row3
1    0    3    6
2    1    4    7
```

3 2 5 8

[1] "Modified Data Frame"

row2

1 3

2 4

3 5

Missing Data

In R, missing values are represented by the symbol NA (not available). Impossible values (e.g., dividing by zero) are represented by the symbol NaN (not a number). Unlike SAS, R uses the same symbol for character and numeric data.

Testing for Missing Values: Example

is.na(x) # returns TRUE of x is missing

y <- c(1,2,3,NA)

is.na(y) # returns a vector (F F F T)

Excluding Missing Values from Analyses: Example

Arithmetic functions on missing values yield missing values.

x <- c(1,2,NA,3)

mean(x) # returns NA

mean(x, na.rm=TRUE) # returns 2

The function **complete.cases()** returns a logical vector indicating which cases are complete.

list rows of data that have missing values

mydata[!complete.cases(mydata),]

The function **na.omit()** returns the object with listwise deletion of missing values.

create new dataset without missing data

newdata <- na.omit(mydata)

Data Visualization in R

Data visualization is the technique used to deliver insights in data using visual cues such as graphs, charts, maps, and many others. This is useful as it helps in intuitive and easy understanding of the large quantities of data and thereby make better decisions regarding it.

The popular data visualization tools that are available are Tableau, Plotly, R, Google Charts, Infogram, and Kibana. The various data visualization platforms have different capabilities, functionality and use cases. They also require a different skill set. This article discusses the use of R for data visualization.

R is a language that is designed for statistical computing, graphical data analysis, and scientific research. It is usually preferred for data visualization as it offers flexibility and minimum required coding through its packages.

Types of Data Visualizations

Some of the various types of visualizations offered by R are:

Bar Plot

There are two types of bar plots- horizontal and vertical which represent data points as horizontal or vertical bars of certain lengths proportional to value of the data item. They are generally used for continuous and categorical variable plotting. By setting the horiz parameter to true and false, we can get horizontal and vertical bar plots respectively.

Syntax

barplot(H,xlab,ylab,main, names.arg,col)

Following is the description of the parameters used –

- **H** is a vector or matrix containing numeric values used in bar chart.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the title of the bar chart.
- **names.arg** is a vector of names appearing under each bar.
- **col** is used to give colors to the bars in the graph.

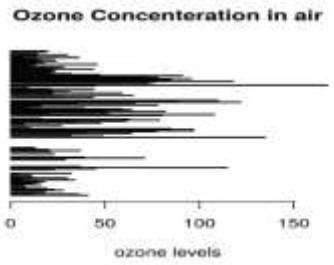
Example 1:

Horizontal Bar Plot for

Ozone concentration in air

barplot(airquality$Ozone, main = 'Ozone Concenteration in air',

xlab = 'ozone levels', horiz = TRUE)



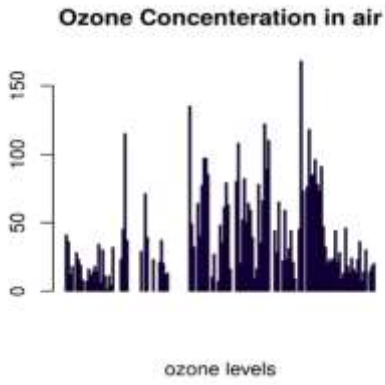
Example 2:

Vertical Bar Plot for

Ozone concentration in air

barplot(airquality$Ozone, main = 'Ozone Concenteration in air',

xlab = 'ozone levels', col ='blue', horiz = FALSE)



Bar plots are used for the following scenarios:

To perform a comparative study between the various data categories in the data set.

To analyze the change of a variable over time in months or years.

Histogram

A histogram is like a bar chart as it uses bars of varying height to represent data distribution. However, in a histogram values are grouped into consecutive intervals called bins. In a Histogram, continuous values are grouped and displayed in these bins whose size can be varied.

Syntax

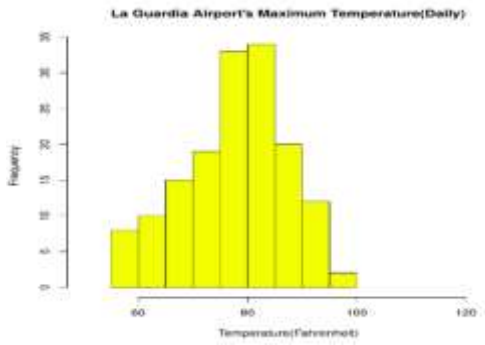
hist(v,main,xlab,xlim,ylim,breaks,col,border)

Following is the description of the parameters used −

- **v** is a vector containing numeric values used in histogram.
- **main** indicates title of the chart.
- **col** is used to set color of the bars.
- **border** is used to set border color of each bar.
- **xlab** is used to give description of x-axis.
- **xlim** is used to specify the range of values on the x-axis.
- **ylim** is used to specify the range of values on the y-axis.
- **breaks** is used to mention the width of each bar.

Example:

```
# Histogram for Maximum Daily Temperature

data(airquality)

hist(airquality$Temp, main ="La Guardia Airport's Maximum Temperature(Daily)",

   xlab ="Temperature(Fahrenheit)",

   xlim = c(50, 125), col ="yellow",

   freq = TRUE)
```



For a histogram, the parameter xlim can be used to specify the interval within which all values are to be displayed.

Another parameter freq when set to TRUE denotes the frequency of the various values in the histogram and when set to FALSE, the probability densities are represented on the y-axis such that ther are of the histogram adds up to one.

Histograms are used in the following scenarios:

To verify an equal and symmetric distribution of the data.

To identify deviations from expected values.

Box Plot

Statistical summary of the given data is presented graphically using a boxplot. A boxplot depicts information like the minimum and maximum data point, the median value, first and third quartile and interquartile range.

Syntax

The basic syntax to create a boxplot in R is −

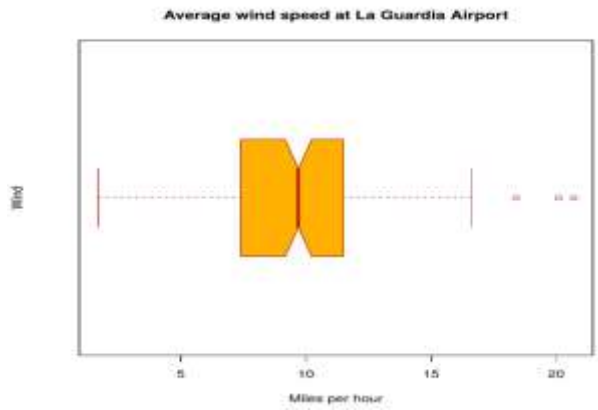
boxplot(x, data, notch, varwidth, names, main)

Following is the description of the parameters used −

- **x** is a vector or a formula.
- **data** is the data frame.
- **notch** is a logical value. Set as TRUE to draw a notch.
- **varwidth** is a logical value. Set as true to draw width of the box proportionate to the sample size.
- **names** are the group labels which will be printed under each boxplot.
- **main** is used to give a title to the graph.

Example:

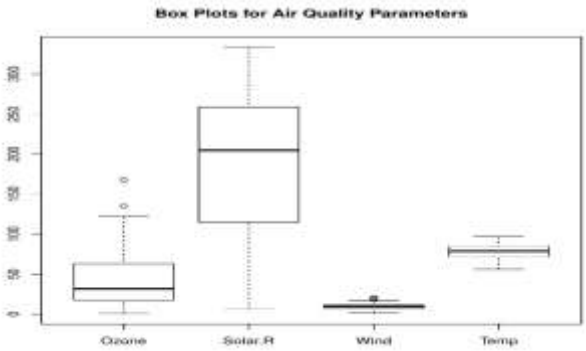
```
# Box plot for average wind speed

data(airquality)

boxplot(airquality$Wind, main = "Average wind speed at La Guardia Airport",

    xlab = "Miles per hour", ylab = "Wind",

    col = "orange", border = "brown",

    horizontal = TRUE, notch = TRUE)
```



Multiple box plots can also be generated at once through the following code:

Example:

```
# Multiple Box plots, each representing
# an Air Quality Parameter
boxplot(airquality[, 0:4],
        main ='Box Plots for Air Quality Parameters')
```



Box Plots are used for:

To give a comprehensive statistical description of the data through a visual cue.

To identify the outlier points that do not lie in the inter-quartile range of data.

Scatter Plot

A scatter plot is composed of many points on a Cartesian plane. Each point denotes the value taken by two parameters and helps us easily identify the relationship between them.

Syntax

The basic syntax for creating scatterplot in R is −

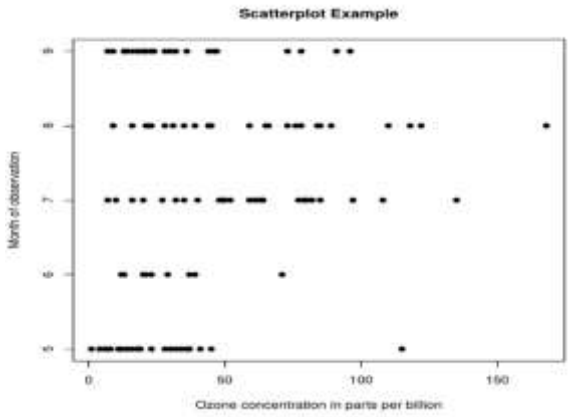
plot(x, y, main, xlab, ylab, xlim, ylim, axes)

Following is the description of the parameters used −

- **x** is the data set whose values are the horizontal coordinates.
- **y** is the data set whose values are the vertical coordinates.
- **main** is the tile of the graph.
- **xlab** is the label in the horizontal axis.
- **ylab** is the label in the vertical axis.
- **xlim** is the limits of the values of x used for plotting.
- **ylim** is the limits of the values of y used for plotting.
- **axes** indicates whether both axes should be drawn on the plot.

Example:

```
# Scatter plot for Ozone Concentration per month
data(airquality)
plot(airquality$Ozone, airquality$Month, main ="Scatterplot Example",
   xlab ="Ozone Concentration in parts per billion",
   ylab =" Month of observation ", pch = 19)
```



Scatter Plots are used in the following scenarios:

To show whether an association exists between bivariate data.

To measure the strength and direction of such a relationship.

Advantages of Data Visualization in R: R has the following advantages over other tools for data visualization:

- R offers a broad collection of visualization libraries along with extensive online guidance on their usage.
- R also offers data visualization in the form of 3D models and multipanel charts.
- Through R, we can easily customize our data visualization by changing axes, fonts, legends, annotations, and labels.

Disadvantages of Data Visualization in R:R also has the following disadvantages:

- R is only preferred for data visualization when done on an individual standalone server.
- Data visualization using R is slow for large amounts of data as compared to other counterparts.

Pie Chart

pie-chart is a representation of values as slices of a circle with different colors. The slices are labeled and the numbers corresponding to each slice is also represented in the chart.

In R the pie chart is created using the **pie()** function which takes positive numbers as a vector input. The additional parameters are used to control labels, color, title etc.

Syntax

pie(x, labels, radius, main, col, clockwise)

Following is the description of the parameters used −

- **x** is a vector containing the numeric values used in the pie chart.
- **labels** is used to give description to the slices.
- **radius** indicates the radius of the circle of the pie chart.(value between −1 and +1).
- **main** indicates the title of the chart.
- **col** indicates the color palette.
- **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

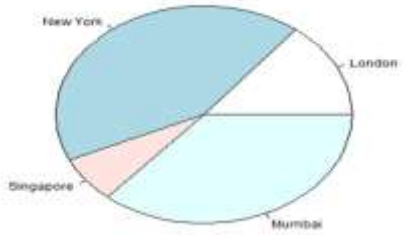
Example

A very simple pie-chart is created using just the input vector and labels. The below script will create and save the pie chart in the current R working directory.

```
# Create data for the graph.
x <- c(21, 62, 10, 53)
labels <- c("London", "New York", "Singapore", "Mumbai")

# Give the chart file a name.
```

```
png(file = "city.png")

# Plot the chart.
pie(x,labels)

# Save the file.
dev.off()
```

When we execute the above code, it produces the following result −



Line Chart

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The **plot()** function in R is used to create the line graph.

Syntax

plot(v,type,col,xlab,ylab)

Following is the description of the parameters used −

- **v** is a vector containing the numeric values.
- **type** takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the Title of the chart.
- **col** is used to give colors to both the points and lines.

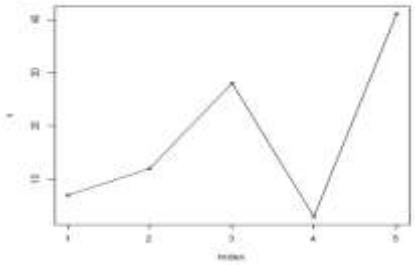
Example

A simple line chart is created using the input vector and the type parameter as "O". The below script will create and save a line chart in the current R working directory.

```
# Create the data for the chart.
v <- c(7,12,28,3,41)

# Give the chart file a name.
png(file = "line_chart.jpg")
```

```
# Plot the bar chart.
plot(v,type = "o")

# Save the file.
dev.off()
```

When we execute the above code, it produces the following result −



Application Areas:

- Presenting analytical conclusions of the data to the non-analyst departments of your company.
- Health monitoring devices use data visualization to track any anomaly in blood pressure, cholesterol and others.
- To discover repeating patterns and trends in consumer and marketing data.
- Meteorologists use data visualization for assessing prevalent weather changes throughout the world.
- Real-time maps and geo-positioning systems use visualization for traffic monitoring and estimating travel time.

ggplot2 Library

ggplot2 is a R package dedicated to data visualization. It can greatly improve the quality and aesthetics of your graphics, and will make you much more efficient in creating them.ggplot2 allows to build almost any type of chart. The ggplot2 package is extremely flexible and repeating plots for groups is quite easy.

We start by loading the required packages. **ggplot2** is included in the **tidyverse** package.

```
library(tidyverse)
```

ggplot Library

To understand ggplot, you need to ask yourself, what are the fundamental parts of every data graph? They are:

- Aesthetics – these are the roles that the variables play in each graph. A variable may control where points appear, the color or shape of a point, the height of a bar and so on.
- Geoms – these are the geometric objects. Do you need bars, points, lines?
- Statistics – these are the functions like linear regression you might need to draw a line.
- Scales – these are legends that show things like circular symbols represent females while circles represent males.
- Facets – these are the groups in your data. Faceting by gender would cause the graph to repeat for the two genders.

A graph starts with the function ggplot(), which takes two arguments. The first argument is the source of the data. The second argument maps the data components of interest into components of the graph. That argument is a function called aes(), which stands for aesthetic mapping. Each argument to aes() is called an aesthetic.

For example, if you're creating a histogram of Temp in the airquality data frame, you want Type on the x-axis. The code looks like this:

ggplot(airquality, aes(x=Temp))

These geom functions come in a variety of types: ggplot2 supplies one for almost every graphing need, and provides the flexibility to work with special cases. For a histogram, the geom function is geom_histogram(). For a bar plot, it's geom_bar(). For a point, it's geom_point().

To add a geom to ggplot, you use a plus sign:

```
ggplot(airquality, aes(x=Temp)) +
  geom_histogram()
```

That's just about it, except for any finishing touches to the graph's appearance. To modify the appearance of the geom, you add arguments to the geom() function. To modify the background color scheme, you can add one or more theme() functions. To add labels to the axes and a title to the graph, you add the function labs().

So, the overall structure for a ggplot graph is

```
ggplot(data_source, aes(map data components to graph components)) +
  geom_xxx(arguments to modify the appearance of the geom) +
  theme_xxx(arguments to change the overall appearance) +
```

```
labs(add axis-labels and a title)
```

It's like building a house: The ggplot() function is the foundation, the geom() function is the house, theme() is the landscaping, and labs() puts the address on the door. Additional functions are available for modifying the graph.

The ggplot2 package offers two main functions: quickplot() and ggplot(). The quickplot() function – also known as qplot() – mimics R's traditional plot() function in many ways. It is particularly easy to use for simple plots. Below is an example of the default plots that qplot() makes. The command that created each plot is shown in the title of each graph. Most of them are useful except for middle one in the left column of qplot(workshop, gender).

Plotting with ggplot2

ggplot2 is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

ggplot2 plots work best with data in the 'long' format, i.e., a column for every dimension, and a row for every observation. Well-structured data will save you lots of time when making figures with **ggplot2**

ggplot graphics are built layer by layer by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

To build a ggplot, we will use the following basic template that can be used for different types of plots:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) +  <GEOM_FUNCTION>()
```

- use the ggplot() function and bind the plot to a specific data frame using the data argument

```
ggplot(data = surveys_complete)
```

- define an aesthetic mapping (using the aesthetic (aes) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g., as x/y positions or characteristics such as size, shape, color, etc.

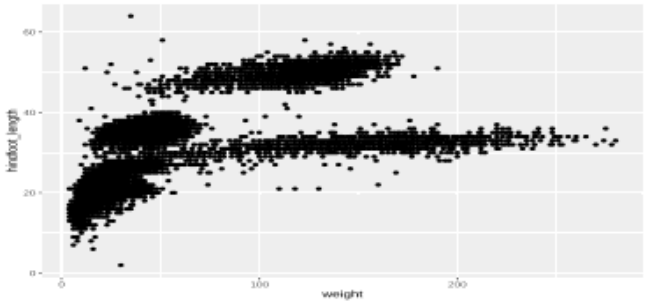
```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length))
```

- add 'geoms' – graphical representations of the data in the plot (points, lines, bars). **ggplot2** offers many different geoms; we will use some common ones today, including:

- geom_point() for scatter plots, dot plots, etc.
- geom_boxplot() for, well, boxplots!
- geom_line() for trend lines, time series, etc.

To add a geom to the plot use + operator. Because we have two continuous variables, let's use geom_point() first:

```
ggplot(data = surveys_complete, aes(x = weight, y = hindfoot_length)) +
  geom_point()
```



The + in the **ggplot2** package is particularly useful because it allows you to modify existing ggplot objects. This means you can easily set up plot "templates" and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```
# Assign plot to a variable
surveys_plot <- ggplot(data = surveys_complete,
                       mapping = aes(x = weight, y = hindfoot_length))

# Draw the plot
surveys_plot +
    geom_point()
```

Notes

- Anything you put in the ggplot() function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x- and y-axis you set up in aes().
- You can also specify aesthetics for a given geom independently of the aesthetics defined globally in the ggplot() function.
- The + sign used to add layers must be placed at the end of each line containing a layer. If, instead, the + sign is added in the line before the other layer, **ggplot2** will not add the new layer and will return an error message.
- You may notice that we sometimes reference 'ggplot2' and sometimes 'ggplot'. To clarify, 'ggplot2' is the name of the most recent version of the package. However, any time we call the function itself, it's just called 'ggplot'.

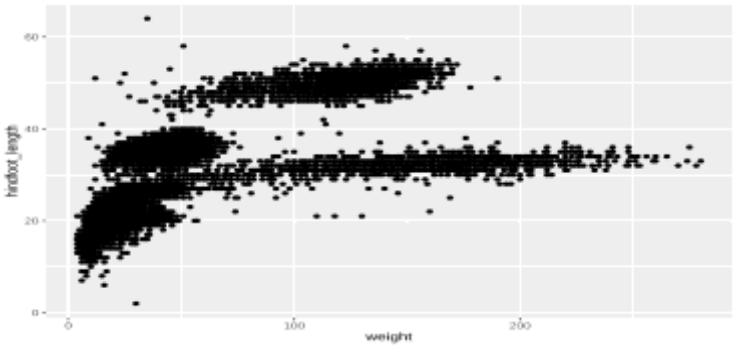
```
# This is the correct syntax for adding layers
```

```
surveys_plot +
  geom_point()

# This will not add the new layer and will return an error message
surveys_plot
  + geom_point()
```

Building your plots iteratively

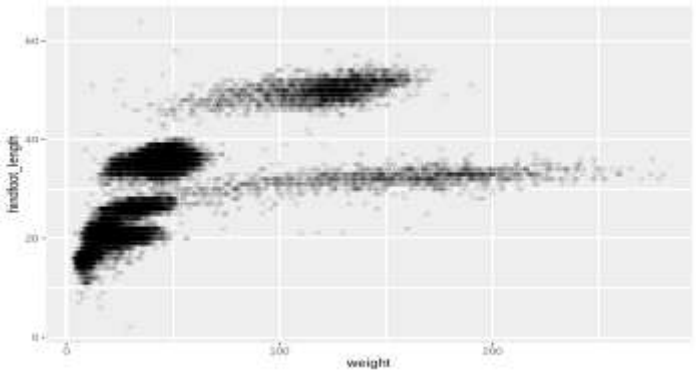
Building plots with **ggplot2** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```
ggplot(data = surveys_complete, aes(x = weight, y = hindfoot_length)) +
    geom_point()
```



Then, we start modifying this plot to extract more information from it. For instance, we can add transparency (alpha) to avoid overplotting:

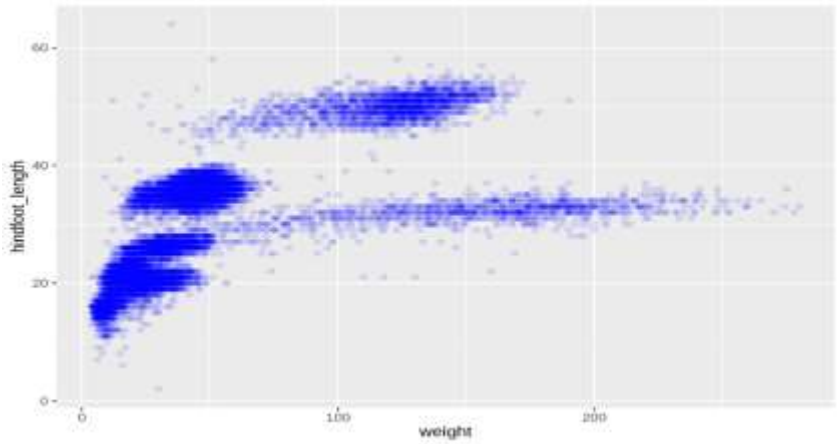
```
ggplot(data = surveys_complete, aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1)
```



We can also add colors for all the points:

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
```

```
  geom_point(alpha = 0.1, color = "blue")
```

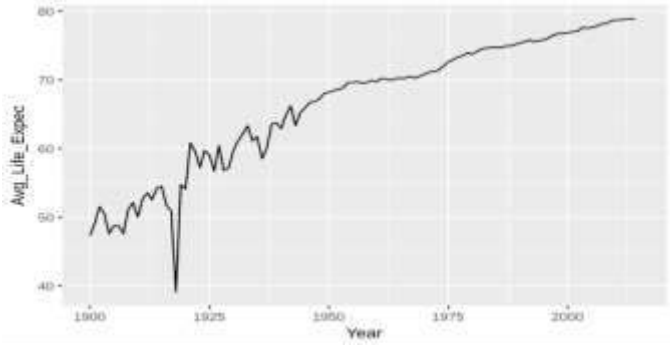


Line Graph

To create a line graph with ggplot(), we use the geom_line() function. A geom is the name for the specific shape that we want to use to visualize the data. All of the functions that are used to draw these shapes have geom in front of them. geom_line() creates a line graph, geom_point() creates a scatter plot, and so on.

```
life_expec %>%

  ggplot(aes(x = Year, y = Avg_Life_Expec)) +


  geom_line()
```

Notice how after the use of the ggplot() function, we start to add more layers to it using a + sign. This is important to note because we use %>% to tell ggplot() what data to function. *After* using ggplot(), we use + to add more layers to the plot.

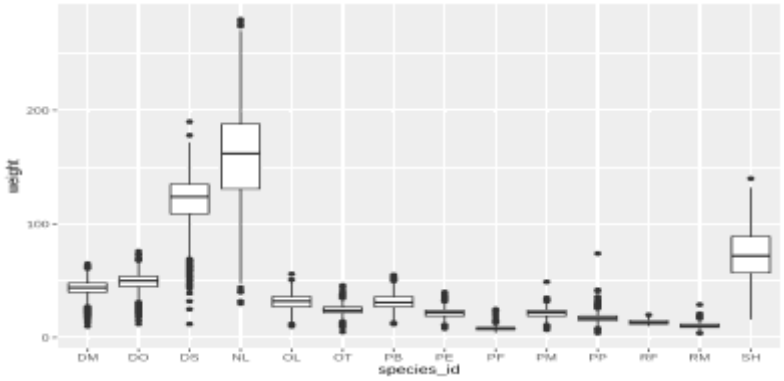


Boxplot

The graph has only one type of geometric object: bars. The ggplot() function itself only needs to specify the data set to use. Note the unusual use of the plus sign "+" to add the effect of of geom_bar() to ggplot(). Only one variable plays an "aesthetic" role: workshop. The aes()

function sets that role.We can use boxplots to visualize the distribution of weight within each species:

```
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
  geom_boxplot()
```



Customizing ggplot2 Graphs

Unlike base R graphs, the ggplot2 graphs are not effected by many of the options set in the par( ) function. They can be modified using the theme() function, and by adding graphic parameters within the qplot() function. For greater control, use ggplot() and other functions provided by the package. Note that ggplot2 functions can be chained with "+" signs to generate the final plot.

Pie Chart

Pie Chart, also known as circle chart, is the graphical representation of the relative size or frequency of the data in a circular format. Basically, it helps in visualizing the relative size or frequency of a particular group of data as a part of the whole.

Function used:

pie() function as the name suggests is used for visualizing a pie chart.

Syntax: pie(x, labels, radius, main, col, clockwise)

Parameters:

- x: This parameter is the vector containing the value of the pie chart.
- labels: This parameter is the vector containing the labels of all the slices in Pie Chart.
- radius: This parameter is the value of the radius of the pie chart. This value is between -1 to 1.

- main: This parameter is the title of the chart.
- col: This parameter is the color used in the pie chart.
- clockwise: This parameter is the logical value which is used to draw the slices in clockwise or anti-clockwise direction.

For building a Pie Chart in R, we can use ggplot2 package, but it does not have a direct method to do so. Instead, we plot a bar graph and then convert it into Pie Chart using coord_polar() function.

Approach:

- Import library
- Create data
- Create dataframe
- Plot a bar graph
- Convert bar graph into Pie chart
- Remove numerical values and grid

Program 2: Pie Chart using ggplot2

```
library(ggplot2)
df = data.frame(x <- c(3,3,2,1,1),
                labels <- c('ADA','CN','PDS','CPDP','PE'))
ggplot(df, aes(x="", y=x, fill=labels)) +geom_bar(width = 1, stat = "identity") +
  coord_polar("y", start=0) +theme_void()
```

Output:

qplot()

The **qplot()** function can be used to create the most common graph types. While it does not expose **ggplot**'s full power, it can create a very wide range of useful plots. The format is:

qplot(*x*, *y*, data=, color=, shape=, size=, alpha=, geom=, method=, formula=, facets=, xlim=, ylim= xlab=, ylab=, main=, sub=)

where the options are:

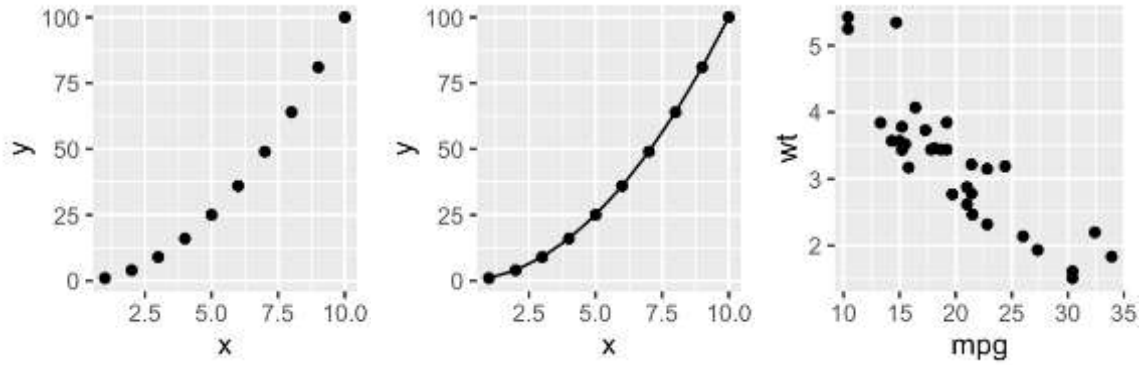
option	Description
alpha	Alpha transparency for overlapping elements expressed as a fraction between 0 (complete transparency) and 1 (complete opacity)
color, shape, size, fill	Associates the levels of variable with symbol color, shape, or size. For line plots, color associates levels of a variable with line color. For density and box plots, fill associates fill colors with a variable. Legends are drawn automatically.
data	Specifies a data frame
facets	Creates a trellis graph by specifying conditioning variables. Its value is expressed as *rowvar ~ colvar*. To create trellis graphs based on a single conditioning variable, use *rowvar~*. or *.~colvar*)
geom	Specifies the geometric objects that define the graph type. The geom option is expressed as a character vector with one or more entries. geom values include "point", "smooth", "boxplot", "line", "histogram", "density", "bar", and "jitter".
main, sub	Character vectors specifying the title and subtitle
method, formula	If geom="smooth", a loess fit line and confidence limits are added by default. When the number of observations is greater than 1,000, a more efficient smoothing algorithm is employed. Methods include "lm" for regression, "gam" for generalized additive models, and "rlm" for robust regression. The formula parameter gives the form of the fit. For example, to add simple linear regression lines, you'd specify geom="smooth", method="lm", formula=y~x. Changing the formula to y~poly(x,2) would produce a quadratic fit. Note that the formula uses the letters x and y, not the names of the variables. For method="gam", be sure to load the mgcv package. For method="rml", load the MASS package.
x, *y*	Specifies the variables placed on the horizontal and vertical axis. For univariate plots (for example, histograms), omit *y*
xlab, ylab	Character vectors specifying horizontal and vertical axis labels
xlim,ylim	Two-element numeric vectors giving the minimum and maximum values for the horizontal and vertical axes, respectively

While qplot() is easy to use for simple graphs, it does not use the powerful grammar of graphics.

Basic scatter plots

The plot can be created using data from either numeric vectors or a data frame:

```
# Use data from numeric vectors
x <- 1:10; y = x*x
# Basic plot
qplot(x,y)
# Add line
qplot(x, y, geom=c("point", "line"))
# Use data from a data frame
qplot(mpg, wt, data=mtcars)
```



Box plot, dot plot and violin plot

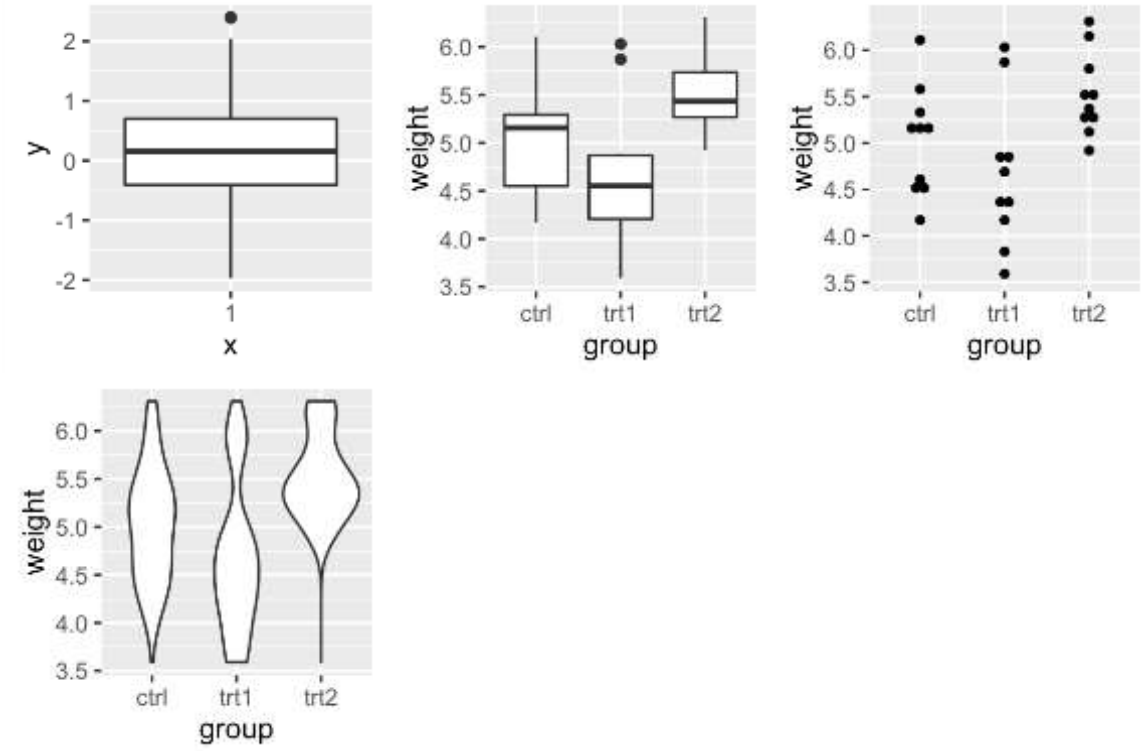
PlantGrowth data set is used in the following example :

```
head(PlantGrowth)
##   weight group
## 1   4.17  ctrl
## 2   5.58  ctrl
## 3   5.18  ctrl
## 4   6.11  ctrl
## 5   4.50  ctrl
## 6   4.61  ctrl
```

- **geom = "boxplot"**: draws a box plot
- **geom = "dotplot"**: draws a dot plot. The supplementary arguments *stackdir = "center"* and *binaxis = "y"* are required.
- **geom = "violin"**: draws a violin plot. The argument **trim** is set to FALSE

```
# Basic box plot from a numeric vector
x <- "1"
y <- rnorm(100)
qplot(x, y, geom="boxplot")
```

```
# Basic box plot from data frame
qplot(group, weight, data = PlantGrowth,
      geom=c("boxplot"))
# Dot plot
qplot(group, weight, data = PlantGrowth,
      geom=c("dotplot"),
      stackdir = "center", binaxis = "y")
# Violin plot
qplot(group, weight, data = PlantGrowth,
      geom=c("violin"), trim = FALSE)
```



Histogram

The **histogram** and **density** plots are used to display the distribution of data.

Generate some data

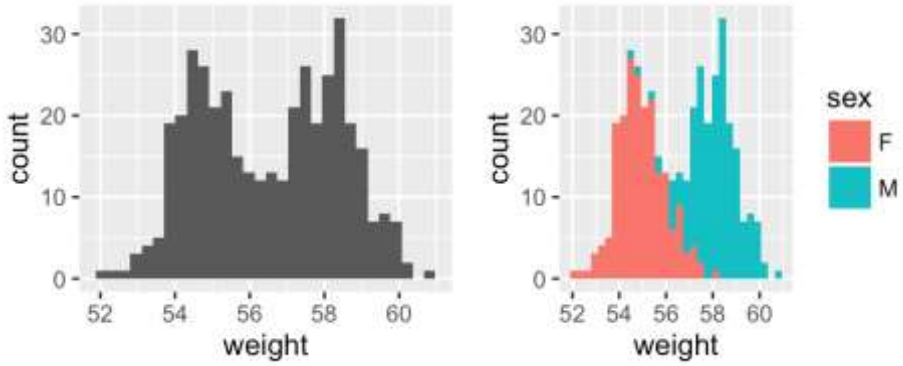
The R code below generates some data containing the weights by sex (M for male; F for female):

```
set.seed(1234)
mydata = data.frame(
      sex = factor(rep(c("F", "M"), each=200)),
      weight = c(rnorm(200, 55), rnorm(200, 58)))
head(mydata)
```

```
##   sex   weight
## 1   F 53.79293
## 2   F 55.27743
## 3   F 56.08444
## 4   F 52.65430
## 5   F 55.42912
## 6   F 55.50606
```

Histogram plot

```
# Basic histogram
qplot(weight, data = mydata, geom = "histogram")
# Change histogram fill color by group (sex)
qplot(weight, data = mydata, geom = "histogram",
   fill = sex)
```



Pipe:

The pipe operator is a special operational function available under the magrittr and dplyr package (basically developed under magrittr), which allows us to pass the result of one function/argument to the other one in sequence and express a sequence of multiple operations. It is generally denoted by symbol %>% in R Programming. Usage of this operator increases, readability, efficiency, and simplicity of your code when you have nested functions in your code loop.

They can greatly simplify your code and make your operations more intuitive. However they are not the only way to write your code and combine multiple operations.

Important tips for piping

- Remember though that you don't assign anything within the pipes - that is, you should not use <- inside the piped operation. Only use this at the beginning if you want to save the output
- Remember to add the pipe %>% at the end of each line involved in the piped operation. A good rule of thumb: RStudio will automatically indent lines of code that are part of a

piped operation. If the line isn't indented, it probably hasn't been added to the pipe. If you have an error in a piped operation, always check to make sure the pipe is connected as you expect.

Points to Note:

- The pipe operator is used when we have nested functions to use in R Programming. Where the result of one function becomes the argument for the next function.
- The pipe functions improve the efficiency as well as readability of code.
- Basic use of the pipe function is to create a pipeline or a chain of functional arguments that work exactly the same as the nested functions.
- We can also use the assignment pipe (%<>%) also known as a compound assignment infix-operator to assign a value to the right-hand side towards the object to the left-hand side without using the traditional assignment operator (<-).
- The pipe operator can also be used to create a chain of functional arguments while working with plotting functions
- There are additional pipes such as tee %T>% pipe. This pipe allows us to continue the chaining under R Programming that gets terminated due to adding pipeline after the plotting functions.

What is the Basic Use of the Pipe Operator?

As it is evident, the pipe operator allows us to assign an argument to a given function and is used in most of the nested functional arguments where the result of one function is an argument for the other function, see the example below for nested functions:

```
summarize(
  group_by(
    filter(mtcars, mtcars$carb >1),
    cyl
  ),
  mpg_mean = mean(mpg)
)
```

Here in this example, if you see, multiple functions are used to filter, group, and summarize the data from the mtcars dataset. We filtered the data first, then grouped it by mean value, and finally summarized it. This code though looks difficult to read and the user might get confused while reading it.

Assignment Using the Pipe (%<>%) Operator

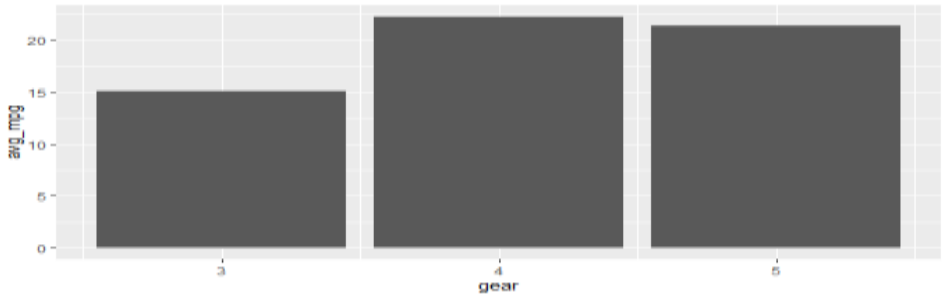
There is this interesting operator called Compound Assignment Infix-Operator (%<>%) under the magrittr package. This function both do the piping value and assigning the same towards an object.

How to Use pipe Operator With ggplot2

We can also use the pipe operator with graphical functions that are a part of the well known ggplot2 function. This will allows us a pipeline that allows us to do an Exploratory Data Analytics (EDA). This method increases the speed of doing aggregations (as you don't need to create those manually) and at the same time, you can have the luxury to avoid those aggregations which you don't want to have in EDA. Let's see the code below which allows us to work with the ggplot2 in combination with pipe operator.

```
library(ggplot2)
mtcars %>%
  filter(carb > 1) %>%
  group_by(gear) %>%
  summarize(avg_mpg = mean(mpg)) %>%
  ggplot(aes(x = gear, y = avg_mpg)) +
  geom_bar(stat = "identity")
```

Here, the bar plot will be created against the avg-mpg values with respect to the no. of gears. We also have used the data values which has carb > 1. Here, the pipeline operator is used to create a pipeline of filtered carb values which are grouped by gear values and summarized by avg_mpg (average value of mpg) and then plotted with gear on x-axis and avg_mpg on the y-axis. See the output graph of this code as below:



limitations:

- Because it chains functions in a linear order, the pipe is less applicable to problems that include multidirectional relationships.
- The pipe can only transport one object at a time, meaning it's not so suited to functions that need multiple inputs or produce multiple outputs.
- It doesn't work with functions that use the current environment, nor functions that use lazy evaluation

Module 3

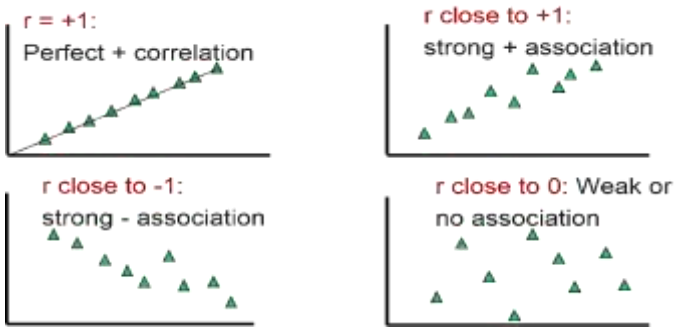
R language is basically developed by statisticians to help other statisticians and developers faster and efficiently with the data. As by now, we know that machine learning is basically working with a large amount of data and statistics as a part of data science the use of R language is always recommended. Therefore, the R language is mostly becoming handy for those working with machine learning making tasks easier, faster, and innovative. Here are some top advantages of R language to implement a machine learning algorithm in R programming.

Correlation

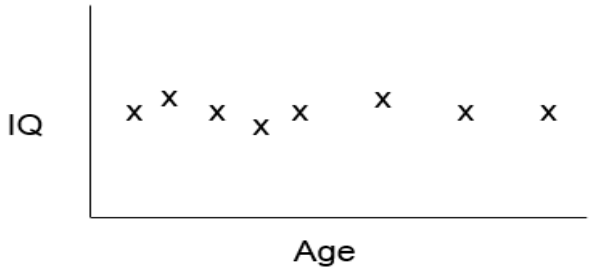
It is a statistical measure that defines the relationship between two variables that is how the two variables are linked with each other. It describes the effect of change in one variable on another variable.
If the two variables are increasing or decreasing in parallel then they have a positive correlation between them and if one of the variables is increasing and another one is decreasing then they have a negative correlation with each other. If the change of one variable has no effect on another variable then they have a zero correlation between them.

The Correlation Coefficient (r)

The sample correlation coefficient (r) is a measure of the closeness of association of the points in a scatter plot to a linear regression line based on those points, as in the example above for accumulated saving over time. Possible values of the correlation coefficient range from -1 to +1, with -1 indicating a perfectly linear negative, i.e., inverse, correlation (sloping downward) and +1 indicating a perfectly linear positive correlation (sloping upward).



A correlation coefficient close to 0 suggests little, if any, correlation. The scatter plot suggests that measurement of IQ do not change with increasing age, i.e., there is no evidence that IQ is associated with age.



Calculation of the Correlation Coefficient

The equations below show the calculations sed to compute "r". However, you do not need to remember these equations. We will use R to do these calculations for us. Nevertheless, the equations give a sense of how "r" is computed.

$$r = \frac{Cov(X,Y)}{\sqrt{{s_x}^2 {s_y}^2}}$$

where Cov(X,Y) is the covariance, i.e., how far each observed (X,Y) pair is from the mean of X and the mean of Y, simultaneously, and and s_x^2 and s_y^2 are the sample variances for X and Y.

. Cov (X,Y) is computed as:

$$Cov\left(X, Y\right) = \frac{\Sigma(X-\bar{X})(Y-\bar{Y})}{n-1}$$

cor() computes the **correlation coefficient**
cor.test() test for association/correlation between paired samples. It returns both the **correlation coefficient** and the **significance level**(or p-value) of the correlation .

Pearson Correlation

The most commonly used type of correlation is Pearson correlation, named after Karl Pearson, introduced this statistic around the turn of the 20th century. **Pearson's *r*** measures the *linear* relationship between two variables, say *X* and *Y*. A correlation of 1 indicates the data points perfectly lie on a line for which ***Y*** increases as ***X*** increases. A value of -1 also implies the data points lie on a line; however, *Y* decreases as *X* increases. The formula for ***r*** is

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

(in the same way that we distinguish between $\bar{Y}$ and μ, similarly we distinguish *r* from ρ)

The Pearson correlation has two assumptions:

1. The two variables are normally distributed. We can test this assumption using

a. A statistical test (Shapiro-Wilk)
b. A histogram
c. A QQ plot

2. The relationship between the two variables is linear. If this relationship is found to be curved, etc. we need to use another correlation test. We can test this assumption by examining the scatterplot between the two variables.

To calculate Pearson correlation, we can use the **cor() function**. The default method for cor() is the Pearson correlation. Getting a correlation is generally only half the story, and you may want to know if the relationship is statistically significantly different from 0.

- H_0: There is no correlation between the two variables: $\rho = 0$
- H_a: There is a nonzero correlation between the two variables: $\rho \neq 0$

Spearman's rank correlation

Spearman's rank correlation is a nonparametric measure of the correlation that uses the rank of observations in its calculation, rather than the original numeric values. It measures the ***monotonic*** relationship between two variables *X* and *Y*. That is, if *Y* tends to increase as *X* increases, the Spearman correlation coefficient is positive. If *Y* tends to decrease as *X* increases, the Spearman correlation coefficient is negative. A value of zero indicates that there is no tendency for *Y* to either increase or decrease when *X* increases. The Spearman correlation measurement makes **no** assumptions about the distribution of the data.

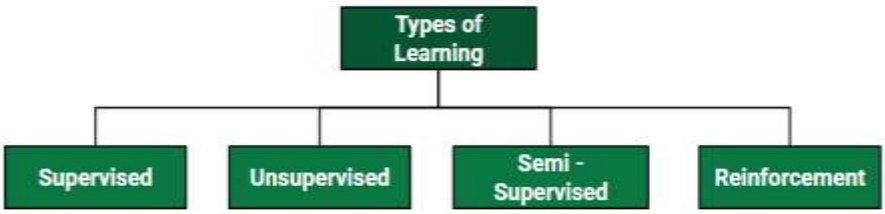
The formula for Spearman's correlation ρ_s is

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}.$$

where d_i is the difference in the *ranked* observations from each group, $(x_i - y_i)$, and n is the sample size.

Advantages to Implement Machine Learning Using R Language

- It provides good explanatory code. For example, if you are at the early stage of working with a machine learning project and you need to explain the work you do, it becomes easy to work with R language comparison to python language as it provides the proper statistical method to work with data with fewer lines of code.
- R language is perfect for data visualization. R language provides the best prototype to work with machine learning models.
- R language has the best tools and library packages to work with machine learning projects. Developers can use these packages to create the best pre-model, model, and post-model of the machine learning projects. Also, the packages for R are more advanced and extensive than python language which makes it the first choice to work with machine learning projects.



Supervised Learning

Supervised learning as the name indicates the presence of a supervisor as a teacher. Basically supervised learning is a learning in which we teach or train the machine using data that is well labeled which means some data is already tagged with the correct answer. After that, the machine is provided with a new set of examples(data) so that the supervised learning algorithm analyses the training data(set of training examples) and produces a correct outcome from labeled data. Supervised learning classified into two categories of algorithms:

- Classification: A classification problem is when the output variable is a category, such as “Red” or “blue” or “disease” and “no disease”.
- Regression: A regression problem is when the output variable is a real value, such as “dollars” or “weight”.

Unsupervised Learning

Unsupervised learning is the training of machine using information that is neither classified nor labeled and allowing the algorithm to act on that information without guidance. Here the task of the machine is to group unsorted information according to similarities, patterns, and differences without any prior training of data. Unlike supervised learning, no teacher is provided that means no training will be given to the machine. Therefore the machine is restricted to find the hidden structure in unlabeled data by our-self. Unsupervised learning classified into two categories of algorithms:

- Clustering: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.
- Association: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y.

Regression

Regression analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variable is called predictor variable whose value is gathered through experiments. The other variable is called response variable whose value is derived from the predictor variable.

Regression analysis is a group of statistical processes used in R programming and statistics to determine the relationship between dataset variables. Generally, regression analysis is used to determine the relationship between the dependent and independent variables of the dataset. Regression analysis helps to understand how dependent variables changes when one of the independent variable is changes and other independent variables are kept constant. This helps in building a regression model and further, helps in forecasting the values with respect to a change in one of the independent variables. On the basis of types of dependent variables, number of independent variables and the shape of the regression line, there are 4 types of regression analysis techniques i.e., Linear Regression, Logistic Regression, Multinomial Logistic Regression and Ordinal Logistic Regression.

Types of Regression Analysis

Linear Regression

Linear Regression is one of the most widely used regression techniques to model the relationship between two variables. It uses a linear relationship to model the regression line. There are 2 variables used in the linear relationship equation i.e., predictor variable and response variable.

The regression line created using this technique is a straight line. The response variable is derived from predictor variables. Predictor variables are estimated using some statistical experiments. Linear regression is widely used but these techniques is not capable of predicting the probability.

In Linear Regression these two variables are related through an equation, where exponent (power) of both these variables is 1. Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.

The general mathematical equation for a linear regression is −

$y = ax + b$

Following is the description of the parameters used −

- y is the response variable.
- x is the predictor variable.
- a and b are constants which are called the coefficients.

Steps to Establish a Regression

A simple example of regression is predicting weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Carry out the experiment of gathering a sample of observed values of height and corresponding weight.
- Create a relationship model using the **lm()** functions in R.
- Find the coefficients from the model created and create the mathematical equation using these
- Get a summary of the relationship model to know the average error in prediction. Also called **residuals**.
- To predict the weight of new persons, use the **predict()** function in R.

lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in linear regression is −

lm(formula,data)

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

predict() Function

Syntax

The basic syntax for predict() in linear regression is −

predict(object, newdata)

Following is the description of the parameters used −

- **object** is the formula which is already created using the lm() function.
- **newdata** is the vector containing the new value for predictor variable.

P-value

The p-Values are very important. Because, we can consider a linear model to be statistically significant only when both these p-Values are less than the pre-determined statistical significance level of 0.05. This can visually interpreted by the significance stars at the end of the row against

each X variable. The more the stars beside the variables p-Value, the more significant the variable.

Whenever there is a p-value, there is always a Null and Alternate Hypothesis associated. A standard way to test if the predictors are not meaningful is looking if the p-values smaller than 0.05. When p Value is less than significance level (< 0.05), you can safely *reject the null hypothesis* that the co-efficient of the predictor is zero.

Residuals

A good way to test the quality of the fit of the model is to look at the residuals or the differences between the real values and the predicted values.

How to test if your linear model has a good fit?

One measure very used to test how good is your model is the coefficient of determination or R^2. This measure is defined by the proportion of the total variability explained by the regression model.

R2=Explained Variation of the model / Total variation of the model

This can seem a little bit complicated, but in general, for models that fit the data well, R^2 is near 1. Models that poorly fit the data have R^2 near 0. However, it's essential to keep in mind that sometimes a high R^2 is not necessarily good every single time (see below residual plots) and a low R^2 is not necessarily always bad. In real life, events don't fit in a perfectly straight line all the time.

One problem with this R^2 is that it cannot decrease as you add more independent variables to your model, it will continue increasing as you make the model more complex, even if these variables don't add anything to your predictions (like the example of the number of siblings). For this reason, the adjusted R^2 is probably better to look at if you are adding more than one variable to the model, since it only increases if it reduces the overall error of the predictions.

R Squared Computation

Remember, the total information in a variable is the amount of variation it contains.

$$R^2 = 1 - \frac{RSS}{TSS}$$

where, RSS is the Residual Sum of Squares given by

$$RSS = \sum_i^n (y_i - \hat{y}_i)^2$$

and the Sum of Squared Total is given by

$$TSS = \sum_i^n (y_i - \bar{y}_i)^2$$

Here, y-hat is the fitted value for observation i and y-bar is the mean of Y.

We don't necessarily discard a model based on a low R-Squared value.

To compare the efficacy of two different regression models, it's a good practice to use the validation sample to compare the AIC of the two models.

Besides AIC, other evaluation metrics like mean absolute percentage error (MAPE), Mean Squared Error (MSE) and Mean Absolute Error (MAE) can also be used.

What is adjusted R-Squared?

As you add more X variables to your model, the R-Squared value of the new bigger model will always be greater than that of the smaller subset.

This is because, since all the variables in the original model is also present, their contribution to explain the dependent variable will be present in the super-set as well. Therefore, whatever new variable you add can only add (if not significantly) to the variation that was already explained.

Adjusted R-Squared is formulated such that it penalises the number of terms (read predictors) in your model. So unlike R-sq, as the number of predictors in the model increases, the adj-R-sq may not always increase. Therefore when comparing nested models, it is a good practice to compare using adj-R-squared rather than just R-squared.

$$R^2_{adj} = 1 - \frac{MSE}{MST}$$

where, *MSE* is the mean squared error given by

$$MSE = \frac{RSS}{(n-q)}$$

where, n is the number of observations and q is the number of coefficients in the model. Therefore, by moving around the numerators and denominators, the relationship between R2 and Radj2 becomes:

$$R^2_{adj} = 1 - \left(\frac{\left(1 - R^2\right)(n-1)}{n-q} \right)$$

Standard Error and F-Statistic

Both standard errors and F-statistic are measures of goodness of fit.

$$Std.\,Error = \sqrt{MSE} = \sqrt{\frac{SSE}{n-q}}$$

$$F - statistic = \frac{MSR}{MSE}$$

where, n is the number of observations, q is the number of coefficients and MSR is the mean square regression, calculated as,

$$MSR = \frac{\sum_i^n \left(\hat{y_i} - \bar{y}\right)}{q-1} = \frac{SST - SSE}{q-1}$$

The higher the F-Statistic the better it is.

What is k- Fold Cross validation and its Purpose?

Suppose, the model predicts satisfactorily on the 20% split (test data), is that enough to believe that your model will perform equally well all the time?

It is possible that few of the data points are not representative of the population on which the model was built.

If the model was trained in such a setup, you cannot expect the model to predict the dist in test dataset with equal accuracy. Simply because, it has not learned the relationship between speed and dist in such a setting.

So, it is important to rigorously test the models performance as much as possible. One way to do this rigorous testing, is to check if the model equation performs equally well, when trained and tested on different distinct chunks of data.

This is exactly what k-Fold cross validation does. Here is how it works:

1. Split your data into k mutually exclusive random sample portions.
2. Then iteratively build k models, keeping one of k-subsets as test data each time.

In each iteration, We build the model on the remaining (k-1 portion) data and calculate the mean squared error of the predictions on the k'th subset.

1. Finally, the average of these mean squared errors (for k portions) is computed.

You can use this metric to compare different linear models. By doing this, you need to check two things from the k-fold predictions:

1. If the each of the k-fold models prediction accuracy isn't varying too much for any one particular sample, and
2. If the lines of best fit from the k-folds don't vary too much with respect the the slope and level.

In other words, they should be parallel and as close to each other as possible. This is what the k-fold cross validation plot (below) reveals.

ModelMetrices

ModelMetrics is a much faster and reliable package for evaluating models. ModelMetrics is written in using Rcpp making it faster than the other packages used for model metrics.

You can install this package from CRAN:
install.packages("ModelMetrics")

Accuracy and Kappa

These are the default metrics used to evaluate algorithms on binary and multi-class classification datasets in caret.

Accuracy is the percentage of correctly classifies instances out of all instances. It is more useful on a binary classification than multi-class classification problems because it can be less clear exactly how the accuracy breaks down across those classes.

Kappa or Cohen's Kappa is like classification accuracy, except that it is normalized at the baseline of random chance on your dataset. It is a more useful measure to use on problems that have an imbalance in the classes (e.g. 70-30 split for classes 0 and 1 and you can achieve 70% accuracy by predicting all instances are for class 0).

RMSE and R^2

These are the default metrics used to evaluate algorithms on regression datasets in caret.

RMSE or Root Mean Squared Error is the average deviation of the predictions from the observations. It is useful to get a gross idea of how well (or not) an algorithm is doing, in the units of the output variable.
R^2 spoken as R Squared or also called the coefficient of determination provides a "goodness of fit" measure for the predictions to the observations. This is a value between 0 and 1 for no-fit and perfect fit respectively.

Area Under ROC Curve

ROC metrics are only suitable for binary classification problems (e.g. two classes). To calculate ROC information, you must change the summary Function in your train Control to be two Class Summary. This will calculate the Area Under ROC Curve (AUROC) also called just Area Under curve (AUC), sensitivity and specificity.

ROC is actually the area under the ROC curve or AUC. The AUC represents a models ability to discriminate between positive and negative classes. An area of 1.0 represents a model that made all predicts perfectly. An area of 0.5 represents a model as good as random.
ROC can be broken down into sensitivity and specificity. A binary classification problem is really a trade-off between sensitivity and specificity.

Sensitivity is the true positive rate also called the recall. It is the number instances from the positive (first) class that actually predicted correctly.
Specificity is also called the true negative rate. Is the number of instances from the negative class (second) class that were actually predicted correctly.

Logarithmic Loss

Logarithmic Loss or LogLoss is used to evaluate binary classification but it is more common for multi-class classification algorithms. Specifically, it evaluates the probabilities estimated by the algorithms

Package nortest

It is used for testing the composite hypothesis of normality.

Anderson-Darling Normality Test

The Anderson-Darling test for normality is one of three general normality tests designed to detect all departures from normality. While it is sometimes touted as the most powerful test, no one test is best against all alternatives and the other 2 tests are of comparable power. The p-values given by Distribution Analyzer for this test may differ slightly from those given in other software packages as they have been corrected to be accurate to 3 significant digits.

The test rejects the hypothesis of normality when the p-value is less than or equal to 0.05. Failing the normality test allows you to state with 95% confidence the data does not fit the normal distribution. Passing the normality test only allows you to state no significant departure from normality was found.

The Anderson-Darling test, while having excellent theoretical properties, has a serious flaw when applied to real world data. The Anderson-Darling test is severely affected by ties in the data due to poor precision. When a significant number of ties exist, the Anderson-Darling will frequently reject the data as non-normal, regardless of how well the data fits the normal distribution. Below is an example of data generated from the normal distribution but rounded to the nearest 0.5 to create ties. A tie is when identical values occurs more than once in the data set:

The AD test is really a hypothesis test.
The null hypothesis (Ho) is that your data is not different from normal.
Your alternate or alternative hypothesis (Ha) is that your data is different from normal.
You will make your decision about whether to reject or not reject the null based on your p-value.
The syntax is: ad.test()

The test statistic for the AD test is:

$$AD = -n - \frac{1}{n}\sum_{i=1}^{n}(2i-1)[\ln F(X_i) + \ln(1 - F(X_{n-i+1}))]$$

Benefits of AD

1. ***Confirms your data distribution***
The AD test will help you determine if your data is not normal rather than tell you whether it is normal. Since the normal distribution is a hypothetical distribution, you can't prove that the data is normal. The AD test will tell you if it is not normal or if it is not different from normal, but it cannot tell you if the data is normal.

2. ***Helps guide your decision***
The p-value, which is based on the value of the AD statistic, will provide you guidance on whether to reject or not reject your null hypothesis.

3. ***Can be simple***
In many cases, the computer software you use will provide you a graphical representation of the data along with the AD value and p-value. This will give you some visual and logical confirmation about your data.

Multiple Regression

Multiple regression is an extension of linear regression into relationship between more than two variables. In simple linear relation we have one predictor and one response variable, but in multiple regression we have more than one predictor variable and one response variable.

The general mathematical equation for multiple regression is −

y = a + b1x1 + b2x2 +...bnxn

Following is the description of the parameters used −

- **y** is the response variable.
- **a, b1, b2...bn** are the coefficients.
- **x1, x2, ...xn** are the predictor variables.

We create the regression model using the **lm()** function in R. The model determines the value of the coefficients using the input data. Next we can predict the value of the response variable for a given set of predictor variables using these coefficients.

lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in multiple regression is −

lm(y ~ x1+x2+x3...,data)

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between the response variable and predictor variables.
- **data** is the vector on which the formula will be applied.

How to Calculate Variance Inflation Factor (VIF) in R

Multicollinearity in regression analysis occurs when two or more predictor variables are highly correlated to each other, such that they do not provide unique or independent information in the regression model.

If the degree of correlation is high enough between variables, it can cause problems when fitting and interpreting the regression model.

The most common way to detect multicollinearity is by using the ***variance inflation factor (VIF),*** which measures the correlation and strength of correlation between the predictor variables in a regression model.

The value for VIF starts at 1 and has no upper limit. A general rule of thumb for interpreting VIFs is as follows:

- A value of 1 indicates there is no correlation between a given predictor variable and any other predictor variables in the model.
- A value between 1 and 5 indicates moderate correlation between a given predictor variable and other predictor variables in the model, but this is often not severe enough to require attention.
- A value greater than 5 indicates potentially severe correlation between a given predictor variable and other predictor variables in the model. In this case, the coefficient estimates and p-values in the regression output are likely unreliable.

Logistic Regression

The Logistic Regression is a regression model in which the response variable (dependent variable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is −

$$y=e^{(a+bx)}/(1+e^{(a+bx)})$$

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are the coefficients which are numeric constants.

The function used to create the regression model is the **glm()** function.

Syntax

The basic syntax for **glm()** function in logistic regression is −

glm(formula,data,family)

Following is the description of the parameters used −

- **formula** is the symbol presenting the relationship between the variables.
- **data** is the data set giving the values of these variables.
- **family** is R object to specify the details of the model. It's value is binomial for logistic regression.

Set.seed()

The set.seed() function sets the starting number used to generate a sequence of random numbers – it ensures that you get the same result if you start with that same seed each time you run the same process. The set.seed helps to create the replicate of the random generation. If the name of the object changes that does not mean the replication will be changed but if we change the position then it will.

Distribution functions in R

Every distribution has four associated functions whose prefix indicates the type of function and the suffix indicates the distribution. To exemplify the use of these functions, I will limit myself to the normal (Gaussian) distribution. The four normal distribution functions are:

- **dnorm**: density function of the normal distribution given a certain random variable x, a population mean μ and population standard deviation σ.
  Syntax*:* dnorm(x, mean, std)
- **pnorm**: cumulative density function of the normal distribution given a certain random variable q, a population mean μ and population standard deviation σ.
  Syntax*:* pnorm(q,mean,std)
- **qnorm**: quantile function of the normal distribution given a certain random variable q, a population mean μ and population standard deviation σ.
  Syntax*:* qnorm(p,mean,std)
- **rnorm**: random sampling from the normal distribution given a certain random variable q, a population mean μ and population standard deviation σ.
  Syntax: rnorm(n,mean,std)

Introduction to Operational Analytics

Operational analytics is a very specific term for a type of analytics which focuses on improving existing operations. This type of analytics, like others, involves the use of various data mining and data aggregation tools to get more transparent information for business planning. The main characteristic that distinguishes operational analytics from other types of analytics is that it is “analytics on the fly," which means that signals emanating from the various parts of a business are processed in real-time to feed back into instant decision making for the business. Some people refer to this as "continuous analytics," which is another way to emphasize the continuous digital feedback loop that can exist from one part of the business to others.

Operational analytics allows you to process various types of information from different sources and then decide what to do next: what action to take, whom to talk to, what immediate plans to make. This form of analytics has become popular with the digitization trend in almost all industry verticals, because it is digitization that furnishes the data needed for operational decision-making.

Software game developers

Let's say that you are a software game developer and you want your game to automatically upsell a certain feature of your game depending on the gamer’s playing habits and the current state of all the players in the current game. This is an operational analytics query because it allows the game developer to make instant decisions based on analysis of current events.

Product managers

Back in the day, product managers used to do a lot manual work, talking to customers, asking them how they use the product, what features in the product slow them down, etc. In the age of operational analytics, a product manager can gather all these answers by querying data that records usage patterns from the product’s user base; and he or she can immediately feed that information back to make the product better.

Marketing managers

Similarly, in the case of marketing analytics, a marketing manager would use to organize a few focus groups, try out a few experiments based on their own creativity and then implement them. Depending on the results of experimentation, they would then decide what to do next. An experiment may take weeks or months. We are now seeing the rise of the "marketing engineer," a person who is well-versed in using data systems.

These marketing engineers can run multiple experiments at once, gather results from experiments in the form of data, terminate the ineffective experiments and nurture the ones that work, all through the use of data-based software systems. The more experiments they can run and the quicker the turnaround times of results, the better their effectiveness in marketing their product. This is another form of operational analytics.

How is Operational Analytics Used in Sales?

A common example of Operational Analytics is often found within SaaS companies leveraging a freemium model. Users can typically sign up and use the product up to a certain limit, at which point they then have to pay. Analytics is then done in the form of a BI dashboard to track the number of signups, the percentage of users who convert to a paid account, and the effectiveness of sales reps in converting those customers. Often, sales reps who spend more time personalizing their outreach to the specific use case tend to over-perform.

However, salespeople typically have to track down this information across a variety of systems like Slack, Salesforce, etc., in order to get the full scoop before sending out a personalized and relevant email. With Operational Analytics, the same data that is feeding into a BI

dashboard is automatically synced into a CRM like Salesforce. This means that contact and account records are enriched to show whether or not the user is fully onboarded, their last login date, and the user's integrations. This leaves the sales team more time to help existing customers and crack into new accounts.

How is Operational Analytics Used in Marketing?

One of the biggest challenges in marketing across every single industry is data accessibility. Nearly every organization today uses data systems like a data warehouse to transform and consolidate the data into a single location for reporting purposes. The data is available to the data teams, but it is difficult for marketers to access without a deep knowledge of SQL.

Since most marketers don't know SQL, this is challenging. As a workaround to this problem, data analysts often deliver various datasets to marketing teams in the form of a CSV. These are manually uploaded into the various systems, whether it be a customer relationship management (CRM) platform like Salesforce or a product analytics platform like Amplitude. Even worse, engineering typically has a backlog of other more important priorities to address before the request from marketing, so it can take a substantial amount of time meaning the data is stale and unusable for marketing purposes. By the

time it is usable, the customer or prospect is at a different point in their journey.

With Operational Analytics, marketing teams can improve customer experiences by sending lifecycle marketing campaigns to customers across any channel as soon as they invite a friend or abandon a shopping cart. They can also increase ROAS by retargeting customers who visited a pricing page and exclude customers who already purchased. Additionally, they can identify high-value customers by creating lookalike audiences and also send conversion events to different ad networks to optimize targeting and customer acquisition costs. Best of all, they don't have to wait around for engineering to give them access to the data they need to run their campaigns. Instead, they can rapidly experiment, iterate, and ask different questions.

How is Operational Analytics Used in Product Analytics?

Businesses are also leveraging Operational Analytics in Product Analytics platforms like Amplitude, and Mixpanel to help companies derive better insights into how their customers are using their products. A common use case revolves around getting information like user id, service area, or product usage information in these product analytic tools to generate more insights. This enables more complex

and deeper analysis on a more granular level and also ensures that different teams have the same view of the customer.

Zeplin was actually able to achieve this using Hightouch. Mixpanel, Zeplin's product analytics tool, had events and reporting for how certain customers were using the product. But it didn't have context into Zeplin specific concepts, such as how many Projects that a given user had. They used Hightouch to enrich group profiles in Mixpanel so that the team could answer questions like "How often do organizations with 5 collaborators login using SSO?". This enriched data helped Zeplin segment their customer base and decide their pricing strategy.

How is Operational Analytics Used in Automation?

Automation is extremely crucial to organizations for multiple reasons. It eliminates human error, speeds up processes, and also gives teams greater visibility into different aspects of their data. One of the best use cases for automation is around messaging and notifications in tools like Slack and Mattermost. These tools have extremely fast response times.

Many companies today connect these communication tools with their various data sources to alert customer success teams when accounts go dormant, share high-intent leads and transactions with the sales team, send product usage characteristics to the product team, and

provide insight into various campaigns for marketers. Best of all, this information is all shared in real-time. Automation doesn't just stop at notifications and messaging either, there is an unlimited amount of use cases that could be addressed within each specific tool. We've written two posts on this exact topic: