Words of Concern

Point to be As Allan Bloom has said "Education is the movement from darkness to light". Through this handbook, I have tried to illuminate what might otherwise appear as black boxes to some. In doing so, I have used references from several other authors to synthesize or simplify or elaborate information. This is not possible without omitting details that I deem trivial while dilating the data that I consider relevant to topic. Every effort has been made to avoid errors. In spite of this, some errors might have crept in. Any errors or discrepancies noted maybe brought to my notice which I shall rectify in my next revision.

Thanks,
Deeksha.V.
Interface College of Computer Applications (ICCA)
Davangere

Bibliography

1. Tilman M. Davies, -The book of R: A first course in programming and statistics, San Francisco, 2016.
2. Vishwas R. Pawgi, -Statistical computing using R software, Nirali prakashan publisher, edition, 2022.
3. https://www.youtube.com'watch?y=KlsYCECWEWE
4. https://www.geeksforgeeks.org/r-tutorial/
5. https://www.tutorialspoint.com/r/index.html“

Syllabus:

Course Code: DSC14
Course Title: Statistical Computing & R Programming (Theory)
Credits - 04
Sem - V
Hours- 52
Formative Assessment Marks: 40
Summative Assessment Marks: 60
Duration of SEA/Exam: 02 hrs.

Course Outcomes (COs): After the successful completion of the course, the student will be able to:

C01 Explore fundamentals of statistical analysis in R environment.
C02 Describe key terminologies, concepts and techniques employed in Statistical Analysis
C03 Define Calculate, Implement Probability and Probability Distributions to solve a wide variety of problems
C04 Conduct and interpret a variety of Hypothesis Tests to aid Decision Making
C05 Understand Analyse, and interpret Correlation Probability and Regression to analyse the underlying relationships between different variables.

Contents **52 Hrs**

UNIT 1: - 10Hrs

Introduction of the R language, numeric, arithmetic, assignment, and vectors Matrices and Arrays, Non-numeric Values, Lists and Data Frames, Special Values, Classes, and Coercion, Basic Plotting.

UNIT 2: - 10Hrs

Reading and writing files, Programming, Calling Functions, Conditions and Loops stand-alone statement with illustrations in exercise 10.1, stacking statements, coding loops, Writing Functions, Exceptions, Timings, and Visibility

UNIT 3: - 11Hrs

Statistics And Probability, basic data visualisation, probability, common probability distributions: common probability mass functions, bernoulli, binomial, poisson distributions, common probability density functions, uniform, normal, student's t-distribution

UNIT 4: - 10Hrs

Statistical testing and modelling, sampling distributions, hypothesis testing, components of hypothesis test, testing means, testing proportions, testing categorical variables, errors and power, Analysis of variance.

UNIT 5: - 11Hrs

Simple linear regression, multiple linear regression, linear model selection and diagnostics
Advanced graphics: plot customization, plotting regions and margins, point and click coordinate interaction, customizing traditional R plots, specialized text and label notation.
Defining colours and plotting a higher dimension, representing and using color.
3D scatter plots.

Unit - 1

R programming is a language and environment specifically designed for statistical computing, data analysis, and graphical representation.
It is widely used among statisticians, data scientists, and researchers for various data-related tasks, including data manipulation, statistical modeling, and visualization.
It supports procedural programming, object-oriented programming (OOP), and functional programming.
R is both a programming language and an open-source software environment.

Introduction to language

What is R Programming

- R is a computer language used for statistical computing.
- The R software was initially developed by Ross Ihaka and Robert Gentleman in mid 1990's. Since 1997, the R project has been organized by the R development core team.
- R is an open-source software. It is developed for the Unix, Macintosh and Windows families of operating systems.
- R is an excellent software to use while learning statistics with the help of computer.
- It provides a coherent, flexible system for data analysis which can be extended as per users' requirements.
- R language is similar to S language so that it allows migrating to the commercially supported S-plus software if desired.
- R is an integrated suit of software facilities for data manipulation, calculation and graphical display.
- Many people use R as a statistical package.
- Following are some important features of R language software.
    - An effective data handing and storage facility.
    - A set of large number of operators for calculations on an array in particular matrices.
    - A large collection of intermediate tools for data analysis
    - Excellent graphical facilities for analysis of data and to display the results directly on computer or as a hard copy.
    - R is a simple and effective programming language which includes conditional loops, user defined functions, Input/Output facilities etc.
    - R has excellent in-built help system.
    - R is compatible with S-plus.
    - R has flexibility.
- Thus R is a useful software for interactive data analysis.

Statistical Analysis
Data Analytics
Data visualisation
Machine learning
Web applications

Variables

- In R programming, variables are used to store and manipulate data.
- They are containers that hold values of different types, such as numbers, strings, logical values, or more complex objects like vectors, matrices, data frames, etc.
- R Programming Language is a dynamically typed language, i.e. the R Language Variables are not declared with a data type rather they take the data type of the R-object assigned to them.

Variable Naming Rules:

- Variable names in R are case-sensitive.
- Variable names can consist of letters, digits, periods (.), and underscores (_).
- Variable names must start with a letter or a period (but not a digit).
- Avoid using reserved keywords (e.g., if, else, for) as variable names.

Declaring and Initializing Variables in R Language

- In R programming, there are three operators which we can use to assign the values to the variable.
- We can use leftward, rightward, and equal_to operator for this purpose.
- R Variables Syntax

- Using equal to operators
  variable_name = value

- Using leftward operator
  variable_name <- value

- Using rightward operator
  value -> variable_name

Example:-

```
x = 10
y <- 20
"ICCA" -> z
print(x)
print(y)
print(z)
```

Output

```
> print(x)
[1] 10
> print(y)
[1] 20
> print(z)
[1] "ICCA"
>
```

Multiple Variables

- R allows you to assign the same value to multiple variables in one line:

Example

```
a <- b <- c <- 500
print(a)
print(b)
print(c)
```

Output

```
> a
[1] 500
> b
[1] 500
> c
[1] 500
>
```

Important Methods for R Variables

- R provides some useful methods to perform operations on variables. These methods are used to determine the data type of the variable, finding a variable, deleting a variable, etc.

1. class() function

- This built-in function is used to determine the data type of the variable provided to it. The R variable to be checked is passed to this as an argument and it prints the data type in return.

Syntax

class(variable)

Example

```
a <- "ICCA"
b <- 2023
c <- 20.562
print(a)
print(b)
print(c)
print(class(a))
print(class(b))
print(class(c))
```

Output

```
> print(a)
[1] "ICCA"
> print(b)
[1] 2023
> print(c)
[1] 20.562
> print(class(a))
[1] "character"
> print(class(b))
[1] "numeric"
> print(class(c))
[1] "numeric"
>
```

2. ls() function

- This built-in function is used to know all the present variables in the workspace.
- This is generally helpful when dealing with a large number of variables at once and helps prevents overwriting any of them.

Syntax

ls()

Example

```
a <- "ICCA"
b <- 10
c <- "Davangere"
print(ls())
```

Output

```
[1] "a" "b" "c"
```


3. rm() function

- This is again a built-in function used to delete an unwanted variable within your workspace.
- This helps clear the memory space allocated to certain variables that are not in use thereby creating more space for others.
- The name of the variable to be deleted is passed as an argument to it.

Syntax

rm(variable)

Example

```
a <- "ICCA"
b <- 10
c <- "Davangere"
rm(a)
print(a)
```

Output

```
> print(a)
Error: object 'a' not found
>
```

Data Types

- A variable can store different types of values such as numbers, characters etc. These different types of data that we can use in our code are called data types.
- These are the following data types which are used in R programming.

Data-Type	Example	Description
Numeric	10.5,223,110	Decimal value is called numeric in R, and it is the default computational data type.
Integer	3L, 66L, 2346L	Here, L tells R to store the value as an integer,
Logical (Boolean)	TRUE, FALSE	It is a special data type for data with only two possible values which can be construed as true/false.
Complex	Z=1+2i, t=7+3i	A complex value in R is defined as the pure imaginary value i.
Character(string)	'a', '"good"', "TRUE", '35.4'	In R programming, a character is used to represent string values
Raw		A raw data type is used to holds raw bytes.

Note:- We can use the class() function to check the data type of a variable:

Numeric Data Type

- In R, the numeric data type represents all real numbers with or without decimal values.

Example

```
x <- 10
y <- 20.6
cat("Data type of x variable :- ",class(x),"\n")
cat("Data type of y variable :- ",class(y),"\n")
```

Output

```
Data type of x variable :-  numeric
Data type of y variable :-  numeric
```

Note:- When R stores a number in a variable, it converts the number into a “double” value or a decimal type with at least two decimal places. This means that a value such as “5” here, is stored as 5.00 with a type of double and a class of numeric

Integer Data Type

- The integer data type specifies real values without decimal points.
- We use the suffix L to specify integer data.
- You can create as well as convert a value into an integer type using the as.integer() function

Example

```
a <- 10L
print(class(a))
print(typeof(a))
```

Output

```
Data type of x variable :-  integer
Data type of y variable :-  integer
```

Logical Data Type(Non-numeric Values)

- R has logical data types that take either a value of true or false.
- A logical value is often created via a comparison between variables.
- Boolean values, which have two possible values, are represented by this R data type: FALSE or TRUE

Example

```
x <- 15
b <- 20
result = x > y
print(result)
cat("Data type of result variable",class(result),"\n")
```

Output

```
[1] FALSE
Data type of result variable logical
```

Complex Datatype in R(Non-numeric values)

- R supports complex data types that are set of all the complex numbers.
- The complex data type is to store numbers with an imaginary component.

Example

```
x = 4+3i
y = sqrt(as.complex(-9))
cat("Data type of x variable :- ",class(x),"\n")
cat("Y :-",y,"\n")
cat("Data type of y variable :- ",class(y),"\n")
```

Output

```
Data type of x variable :-  complex
Y :- 0+3i
Data type of y variable :-  complex
```

Character Data Type(Non-numeric values)

- R supports character data types where you have all the alphabets and special characters.
- It stores character values or strings.
- Strings in R can contain alphabets, numbers, and symbols.
- The easiest way to denote that a value is of character type in R data type is to wrap the value inside single or double inverted commas

Example

```
x = 'a'
y="ICCA"
cat("Data type of x variable,",class(x),"\n")
cat("Data type of y variable,",class(y),"\n")
```

Output

```
Data type of x variable, character
Data type of y variable, character
```

Concatenation of strings

- There are two main functions used to concatenate (or glue together) one or more strings: cat and paste.

- The difference between the two lies in how their contents are returned.
- The first function, cat, sends its output directly to the console screen and doesn't formally return anything.
- The paste function concatenates its contents and then returns the final character string as a usable R object.
- This is useful when the result of a string concatenation needs to be passed to another function or used in some secondary way, as opposed to just being displayed.

Example

```
a <- "ICCA"
cat("College Name:- ",a,"\n")
b <- paste("Colege Name",a)
print(b)
```

Output

```
> source("~/.active-rstudio-document")
College Name:-  ICCA
[1] "Colege Name ICCA"
```

Escape Sequence

- The \ is used to invoke an escape sequence.
- An escape sequence lets you enter characters that control the format and spacing of the string, rather than being interpreted as normal text below table describes some of the most common escape sequences.

Escape sequence	Result
\n	Starts a newline
\t	Horizontal tab
\b	Invokes a backspace
\\	Used as a single backslash
\"	Includes a double quote

Example

```
cat("Hi, we are from \"ICCA\" college, Davangere")
```

Output

```
> source("~/.active-rstudio-document")
Hi, we are from "ICCA" college, Davangere
```


Substrings and Matching

- Pattern matching lets you inspect a given string to identify smaller strings within it.

1. substr()

- Extract or replace substrings in a character vector
- The function substr takes a string x and extracts the part of the string between two character positions (inclusive), indicated with numbers passed as start and stop arguments

Syntax:- substr(x, start, stop)

OR

Syntax :- substr(x, start, stop) <- value

Where,

X a character vector.

start integer. The first element to be extracted or replaced.

Stop integer. The last element to be extracted or replaced.

Example:-

```
str1 <- "Interface college of applications"
print(str1)
substr1 <- substr(x=str1,start = 11,stop=17)
print(substr1)
substr2 <- substr(x=str1,start = 1,stop=9)<-"INTERFACE"
print(str1)
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
[1] "Interface college of applications"
[1] "college"
[1] "INTERFACE college of applications"
```

- If the replacement string is longer than the number of characters indicated by start and stop, then replacement still takes place, beginning at start and ending at stop.
- It cuts off any characters that overrun the number of characters you're replacing.
- If the string is shorter than the number of characters you're replacing, then replacement ends when the string is fully inserted, leaving the original characters up to stop untouched.

2. sub() and gsub()

- Substitution is more flexible using the functions sub and gsub.
- The sub function searches a given string x for a smaller string pattern contained within.
- It then replaces the first instance with a new string, given as the argument replacement.
- The gsub function does the same thing, but it replaces every instance of pattern

Syntax:- **sub(pattern, replacement, x, ignore.case = FALSE)**

AND

Syntax:- **gsub(pattern, replacement, x, ignore.case = FALSE)**

Where,

- pattern: The regular expression pattern you want to match and replace.
- replacement: The string you want to replace the matched pattern with.
- x: The character vector (string) in which you want to perform the substitution
- ignore.case : if FALSE, the pattern matching is case sensitive and if TRUE, case is ignored during matching.

Example:-

```
str1 <- "Interface COLLEGE of Computer applications, Interface COLLEGE of
computer applications"
cat("Original String\n",str1)
substr1 <- sub(pattern="COLLEGE",replacement = "college",x=str1)
cat("\nReplacing using sub()\n",substr1)
substr2 <- gsub(pattern = "interface",replacement =
"INTERFACE",x=str1,ignore.case = TRUE)
cat("\nReplacing using gsub()\n",substr2)
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
Original String
 Interface COLLEGE of Computer applications, Interface COLLEGE of computer applications
Replacing using sub()
 Interface college of Computer applications, Interface COLLEGE of computer applications
Replacing using gsub()
 INTERFACE COLLEGE of Computer applications, INTERFACE COLLEGE of computer applications
```

R Operators

- An Operator is a symbol that tells to perform different operations between operands.
- R programming is very rich in built-in operators.
- R has the following data operators:
  - Arithmetic Operators
  - Assignment Operators
  - Logical Operators
  - Relational Operators
  - Miscellaneous Operators

Arithmetic Operators

- These operators perform basic arithmetic operations like addition, subtraction, multiplication, division, exponent, modulus, etc.

Operator	Operation	Example
+	Addition	x+y
-	Subtraction	x-y
*	Multiplication	x*y
/	Division	x/y
%%	Modulus	x%%y
^	Exponent	x^y

Example

```
x <- 20
y <- 2
cat("Addition of two numbers",x+y,"\n")
cat("Subtraction of two numbers",x-y,"\n")
cat("Multiplication of two numbers",x*y,"\n")
cat("Quotient of two numbers",x/y,"\n")
cat("Remainder of two numbers",x%%y,"\n")
cat("Exponent :- ",x^y,"\n")
```

Output

```
Addition of two numbers 22
Subtraction of two numbers 18
Multiplication of two numbers 40
Quotient of two numbers 10
Remainder of two numbers 0
Exponent :-  400
```


Assignment Operators

- Assignment operators are used to assign values to variables:
- There are two kinds of assignments, leftwards assignment, and rightwards assignment.
  x<- 3 or x = 3 (Leftwards Assignment)
  3 -> x or x = 3 (Rightwards Assignment)

Operator	Description	Example
<- or = or <<-	Leftward Assignment	X<- 10, y<<- 20, z = 30
-> Or ->>	Rightward Assignment	20 -> x, 30 ->> y

Note:- <<- and ->> is a global assignment operator.

Example

```
a <- 10
20 -> b
c = 50
cat("a :- ",a,"\n")
cat("b :- ",b,"\n")
cat("c :- ",c,"\n")
```

Output

```
a :-  10
b :-  20
c :-  50
```

Logical Operator

- The logical operators allow a program to make a decision on the basis of multiple conditions

Operator	Description	Example
&	AND (Element wise)	x&y
|	OR (Element wise)	X|Y
!	Not	!x
&&	AND	X&&Y
||	OR	X||Y

Example

```
x <- c(1,0,3,1)
y <- c(0,2,0,3)
cat("x & y :- ", x & y, "\n")
cat("x | y :- ", x | y, "\n")
cat("!x :- ", !x, "\n")
cat("!y :- ", !y, "\n")
a <- 5
b <- 0
cat("a && b :-",a && b,"\n")
cat("a || b :-",a || b,"\n")
```

Output

```
x & y :-  FALSE FALSE FALSE TRUE
x | y :-  TRUE TRUE TRUE TRUE
!x :-  FALSE TRUE FALSE FALSE
!y :-  TRUE FALSE TRUE FALSE
a && b :- FALSE
a || b :- TRUE
```


Relational Operator

- These operators are used to compare two values or variables.
- To find if one is smaller, greater, equal, not equal, and other similar operations these operators are used.
- A relational operator is always a Logical value, that is either TRUE or FALSE.

Operator	Description	Example
>	Greater than	X>Y
<	Lesser than	X<Y
>=	Greater than or equal to	X>=Y
<=	Lesser than or equal to	X<=Y
==	Equal to	X==Y
!=	Not equal to	X!=Y

Example

```
x<<-20
```

```
2->>y
print("Testing relation between two operand")
cat("is x greater than or equall to y?",x>=y,"\n")
cat("is x Lesser than or equall to y?",x<=y,"\n")
cat("is x equall to y?",x==y,"\n")
```

Output

```
[1] "Testing relation between two operand"
is x greater than or equall to y? TRUE
is x Lesser than or equall to y? FALSE
is x equall to y? FALSE
```

Miscellaneous Operators

- These R programming operators are used for special cases and are not for general mathematical or logical computation.

1. colon operator(:) – It is used to generate a series of numbers in a sequence.

Example

```
x <- 1:9
print(x)
```

Output

```
[1] 1 2 3 4 5 6 7 8 9
```

2. %in% operator :- This operator is used to check if an element belongs to a vector or not.

Example

```
x <- 1:9
print(x)
y = 5
print(y %in% x)
```

Output

```
[1] 1 2 3 4 5 6 7 8 9
[1] TRUE
```

3. %* % operator : This operator multiplies a matrix with its transpose.

- Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows.
- The number of columns of the first matrix must be equal to the number of rows of the second matrix.
- Multiplication of the matrix A with its transpose, B, produces a square matrix.

Example

```
a <- c(1,2,3,4,5,6)
m1 <- matrix(a,nrow = 2, ncol = 3)
print(m1)
m2 <- t(m1)
print(m2)
mul <- m1%*%m2
print(mul)
```

Output

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
     [,1] [,2]
[1,]   35   44
[2,]   44   56
```

- Matrix multiplication is a mathematical operation that takes two matrices as its operands and produces a third matrix.
- The two matrices must have the same number of columns as the first matrix has rows.
- The product matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.

Example

```
A = matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
print("Matrix A")
print(A)
B = matrix(c(5, 6, 7, 8), nrow = 2, ncol = 2)
print("Matrix B")
print(B)
C = A %*% B
cat("Multiplication of Matrix A and B\n")
print(C)
```

Output

```
> source("D:/ICCA/R_Program/operator.R")
[1] "Matrix A"
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[1] "Matrix B"
     [,1] [,2]
[1,]    5    7
[2,]    6    8
Multiplication of Matrix A and B
     [,1] [,2]
[1,]   23   31
[2,]   34   46
```

Basic math Operations

- All common arithmetic operations and mathematical functionality are ready to use at the console prompt.
- You can perform addition, subtraction, multiplication, and division with the symbols +, -, *, and /, respectively.
- Along with arithmetic operations following operations can also be performed.

1. Logarithms

- A logarithm is a mathematical function that represents the exponent to which a fixed number, called the base, must be raised to obtain a given number
- There are three main logarithm functions in R:
  - log(): This is the default logarithm function. It calculates the natural logarithm of a number. The natural logarithm is the logarithm to the base e.
  - log10(): This function calculates the base-10 logarithm of a number.
  - log2(): This function calculates the base-2 logarithm of a number.
- The syntax for all three functions is the same:

  log(x, base=e)
- Where,
  - x is the number whose logarithm is to be calculated.
  - base is the base of the logarithm. The default value is e.
- Here are some things to consider:
  - Both x and the base must be positive.
  - The log of any number x when the base is equal to x is 1.
  - The log of x = 1 is always 0, regardless of the base.
- If you do not specify a base, the log() function will calculate the natural logarithm of the number.

- The natural log, which fixes the base at a special mathematical number—Euler's number. This is conventionally written as e and is approximately equal to 2.718.
- You must provide the value of base yourself if you want to use a value other than e

Example

```
cat("log3(243)=",log(x=243,base=3))
cat("\nNatrural log of log(500)=",log(500))
cat("\nlog10(100)=",log10(100))
cat("\nlog2(8)=",log2(8))
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
log3(243)= 5
Natrural log of log(500)= 6.214608
log10(100)= 2
log2(8)= 3
```

2. Exponentials

- Euler's number gives rise to the exponential function, defined as e raised to the power of x, where x can be any number (negative, zero, or positive).
- The exponential function, f (x) = e x , is often written as exp(x) and represents the inverse of the natural log such that exp(loge x) = loge exp(x) = x

Example

```
> exp(x=3)
[1] 20.08554
> log(20.08554)
[1] 3
> exp(5)
[1] 148.4132
> log(148.4132)
[1] 5
```

3. E-Notation

- When R prints large or small numbers beyond a certain threshold of significant figures, the numbers are displayed using the classic scientific e-notation.
- The e-notation is typical to most programming languages and even many desktop calculators to allow easier interpretation of extreme values.
- In e-notation, any number x can be expressed as xey, which represents exactly x * 10y .
- Consider the number 2,342,151,012,900.
- It could, for example, be represented as follows:
  - 2.3421510129e12, which is equivalent to writing $2.3421510129 \times 10^{12}$
  - 234.21510129e10, which is equivalent to writing $234.21510129 \times 10^{10}$
- You could use any value for the power of y, but standard e-notation uses the power that places a decimal just after the first significant digit.
- For a positive power +y, the e-notation can be interpreted as "move the decimal point y positions to the right."
- For a negative power −y, the interpretation is "move the decimal point y positions to the left." This is exactly how R presents e-notation:

```
> 2342151012900
[1] 2.342151e+12
> 0.0000002533
[1] 2.533e-07
```

4.sqrt()

- You can find the square root of any non-negative number with the sqrt function.
- You simply provide the desired number to x as shown here:

Syntax:- **sqrt(x)**

Example

```
> sqrt(4)
[1] 2
> sqrt(16)
[1] 4
```

Input and Output Statements in R

- There are several functions and statements available for input and output operations.

1. Input from Console

- In R, there are multiple ways to read and save input given by the user. here are a few of them:

a. readline( )

- In R, the readline() function is used to read a single line of text or string input from the console.
- It prompts the user to enter a value and waits for the input.
- The entered text is returned as a character string.
- It is helpful for obtaining user input for various purposes, such as capturing names, prompts, or other textual information in your program.

Example:- 1

```
input_read1 <- readline("Enter the value for input variable")
print(input_read1)
print(typeof(input_read1))
input_read2 <-  readline("Enter the value for input variable")
print(input_read2)
print(typeof(input_read2))
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
Enter the value for input variable ICCA
[1] "ICCA"
[1] "character"
Enter the value for input variable 10
[1] "10"
[1] "character"
```

- You can convert the input to numeric type using the appropriate conversion functions, such as as.integer() or as.numeric().

Example:- 2

```
input_integer <- as.integer(readline("Enter value for input variable "))
print(input_integer)
print(typeof(input_integer))
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
Enter value for input variable 10
[1] 10
[1] "integer"
```

b. scan()

- The scan() function in R is used to read data values from the console or a file into a vector.
- It allows you to input a sequence of values interactively or from a file, and it returns a vector containing those values.

Syntax:- **vector <- scan(file = "", what = "", n = -1, sep = "", quiet = FALSE)**

Where,

- file:- Specifies the file name or connection to read data from. If left empty (""), it reads data from the console.
- what:- Specifies the data type to be read. It can be "character", "numeric", "integer", or "logical".
- n:- Specifies the number of elements to be read. By default, n = -1 reads all available elements.
- sep:- Specifies the separator used to distinguish between elements in the input data. By default, it separates elements using whitespace characters.
- quiet:- If quiet = FALSE (default), it displays the number of items read.

Example

```
input_scan <- scan()
print(input_scan)
print("Example for inputting string by using hyphen(-) as a separator")
input_character1 <- scan(what = "character", sep = "-")
print(input_character1)
print("Enter three numeric values")
input_limit <- scan(n=3,quiet = TRUE)
print(input_limit)
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
1: 10 20 30 40 50
6:
Read 5 items
[1] 10 20 30 40 50
[1] "Example for inputting string by using hyphen(-) as a separator"
1: I-C-C-A
5:
Read 4 items
[1] "I" "C" "C" "A"
[1] "Enter three numeric values"
1: 56 23 85
[1] 56 23 85
```

2.Output to Console

- In R, there are several functions and approaches for displaying output statements.

a. print() function

- The print() function is used to display the value of an object or expression on the console or output panel.
- It automatically formats the output based on the type of the object or expression.

Synatx:- **print("any string") or, print(variable)**

Example

```
a <- 10
print(a)
print("WELCOME TO ICCA")
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
[1] 10
[1] "WELCOME TO ICCA"
>
```

Print output using paste() function inside print() function

- R provides a paste() and paste0() methods to print output with string and variable together.

➢ paste() function:
  - The paste() function concatenates multiple strings together, separated by a specified separator (default is a space).
  - It allows you to concatenate multiple strings, variables, or expressions with control over the separator using the sep argument.

- paste0() function:
  - The paste0() function is a shorthand version of paste() with the sep argument set to an empty string ("").
  - It concatenates strings without any separator.

Example

```
x <- "Interface"
y <- "College"
print(paste(x,y))
print(paste(x,y,sep="-"))
print(paste0(x,y))
```

Output

```
> source("D:/ICCA/R_Program/hello.R")
[1] "Interface College"
[1] "Interface-College"
[1] "InterfaceCollege"
```

b. cat() function

- The cat() function in R is used to concatenate and print multiple character strings or objects to the console or output file.
- Unlike print(), cat() does not include any automatic formatting or line breaks, providing more control over the output.

Syntax :- **cat(..., file= " ", sep = " ", fill = FALSE, labels = NULL, append = FALSE)**

Where,

- File :- A connection, or a character string naming the file to print to. If "" (the default), cat prints to the standard output connection to the
- '...' :- Represents one or more character strings or objects to be concatenated and printed. You can pass multiple arguments separated by commas.
- Sep :- Specifies the separator to be used between the concatenated elements. The default separator is a single space " ".
- Fill :- If set to TRUE, the concatenated output will be filled with spaces to align columns.
- Labels :- A character vector to be used as column headers when fill = TRUE.
- Append:- If TRUE output will be appended to file; otherwise, it will overwrite the contents of file.

Example

```
x <- "Interface college of computer applications"
y <- "Anjaneya badavane"
cat("College Name:-",x,"Address:-",y,fill = 59)
```


Output

```
> source("D:/ICCA/R_Program/hello.R")
College Name:- Interface college of computer applications
Address:- Anjaneya badavane
```

R Data Structures

1. Vector

- A vector is the basic data structure in R that stores data of similar types.
- R vectors are the same as the arrays in C language which are used to hold multiple data values of the same type.
- One major key point is that in R the indexing of the vector will start from '1' and not from '0'.

Vectors in R

Creating Vector

There are various other ways to create a vector in R, which are as follows:

1. using c() function
2. Using the colon(:) operator
3. using the seq() function

1. using c() function

- In R, we use c() function to create a vector.
- This function returns a one-dimensional array or simply vector.
- The c() function is a generic function which combines its argument.

Example

```
eg_vector <- c(10,20,30,40,50)
print(eg_vector)
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] 10 20 30 40 50
```

2. using the colon(:) operator

- We can create a vector with the help of the colon operator.
- There is the following syntax to use colon operator:

z<-x:y

Example

```
eg_colon <- 50:60
print(eg_colon)
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
 [1] 50 51 52 53 54 55 56 57 58 59 60
```


3. seq() function

- In R, we can create a vector with the help of the seq() function.
- A sequence function creates a sequence of elements as a vector.
- The seq() function is used in two ways, i.e., by setting step size with “by” parameter or specifying the length of the vector with the “length.out” feature.

Synatx 1:- **seq(from, to, by)**

- The ‘from’ parameter specifies the starting value of the sequence
- The ‘to’ parameter specifies the end value.
- The by parameter specifies the increment or step size between the numbers in the sequence.

Synatx 2:- **seq(from, to, length.out)**

- The ‘from’ parameter specifies the starting value of the sequence
- The ‘to’ parameter specifies the end value
- The length.out parameter specifies the desired length of the sequence.

Example

```
eg_seq1 <- seq(1,10,length.out = 5)
```

```
cat("Printing desired length of the sequence using length.out\n",eg_seq1,"\n")

eg_seq2 <- seq(1,10,by = 2)
cat("Incrementing the steps using by\n",eg_seq2,"\n")

eg_seq <- seq(10,20)
cat("Printing using Default length(default length = 1)\n",eg_seq,"\n")
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
Printing desired length of the sequence using length.out
 1 3.25 5.5 7.75 10
Incrementing the steps using by
 1 3 5 7 9
Printing using Default length(default length = 1)
 10 11 12 13 14 15 16 17 18 19 20
```

Types of R vectors

In R, there are several types of vectors based on the data they can hold.. Following are some of the types of vectors:

1. Numeric vector

- The decimal values are known as numeric data types in R
- A vector which contains numeric elements is known as a numeric vector.

Example

```
numeric_vector = c(1.21,6.35,85.69,41.36)
print(typeof(numeric_vector))
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] "double"
```

2. Integer vector

- A non-fraction numeric value is known as integer data
- There is two way to assign an integer value to a variable, i.e., by using as.integer() function and appending of L to the value.
- A vector which contains integer elements is known as an integer vector.

Example

```
integer_vector = c(10L,26L,96L)
print(typeof(integer_vector))
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] "integer"
```

3. Logical vector

- The logical data types have only two values i.e., True or False.
- These values are based on which condition is satisfied.
- A vector which contains Boolean values is known as the logical vector.

Example

```
logical_vector = c(TRUE,FALSE,FALSE,TRUE)
print(typeof(logical_vector))
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] "logical"
```

4. Character vector

- A character is held as a one-byte integer in memory.

- In R, there are two different ways to create a character data type value, i.e., using as.character() function and by typing string between double quotes("") or single quotes('').
- A vector which contains character elements is known as an integer vector.

Example

```
character_vector = c('Interface',"college","of","Computer","Applications")
print(typeof(character_vector))
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] "character"
```

Accessing vector elements

1. Indexing

- In R, each element in a vector is associated with a number. The number is known as a vector index.
- We can access elements of a vector using the index number (1, 2, 3 …).

Example

```
character_vector = c('Interface',"college","of","Computer","Application")
cat("Accessing 4th element in a vector:- ",character_vector[4],"\n")
cat("Accessing 1st element in a vector:- ",character_vector[1],"\n")
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
Accessing 4th element in a vector:-  Computer
Accessing 1st element in a vector:-  Interface
```

2. Accessing Multiple elements

- You can also access multiple elements by referring to different index positions with the c() function:

Example

```
eg_vector <- c(10,20,30,40,50)
print(eg_vector)
print(eg_vector[c(2,5,3)])
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] 10 20 30 40 50
[1] 20 50 30
```


3. Loop Over a R Vector

- We can also access all elements of the vector by using a for loop.
- In R, we can also loop through each element of the vector using the for loop.

Example

```
eg_colon <- 50:60
#print(eg_colon)
for (i in eg_colon)
{
  cat(i,"\n")
}
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
50
51
52
53
54
55
56
57
58
59
60
```

Operations on vector

1. Modify Vector Element

- To change a vector element, we can simply reassign a new value to the specific index

Example

```
initial_vector = c('Interface',"college of","Computer","Application")
cat("Initial Vector:- ",initial_vector,"\n")
initial_vector[1] = 'I' # changing element at index 1
initial_vector[2] = 'C'
initial_vector[3] = 'C'
initial_vector[4] = 'A'
cat("Updated vector:- ",initial_vector)
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
Initial Vector:-  Interface college of Computer Application
Updated vector:-  I C C A
```

2. Repeating elements of Vectors in R

- The rep() function in R is used to create vectors by replicating elements.
- It allows you to repeat values or sequences of values to generate a larger vector.
- The basic syntax of the rep() function is as follows:

  rep(x, times,each)
- The x parameter specifies the values or sequence that you want to replicate.
- Times parameter specifies the number of times to repeat the values or sequence.

Example 1: Replicate a single value multiple times

```
eg_rep = rep(6,times = 3)
print(eg_rep)
```

Output:-

```
6 6 6
```

Example 2: Replicate a sequence of values

```
eg_rep1 <- rep(c(2,4,6),times = 2)
print(eg_rep1)
```

Output:-

```
2 4 6 2 4 6
```

Example 3: Replicate each value a specific number of times

```
eg_reg2<- rep(c(1,3,5),each = 2)
print(eg_reg2)
```

Output:-

```
1 1 3 3 5 5
```

Example 4: Replicate with different lengths

```
eg_rep3<- rep(c(1,2,3),times = c(2,3,2))
print(eg_rep3)
```

Output:-

```
1 1 2 2 2 3 3
```

3. Vector Length

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

Syntax:- **length(vector name)**

Example

```
eg_reg2 <- rep(c(1,3,5),each = 2)
print(eg_reg2)
cat("Number of elements within a vector:- ",length(eg_reg2) )
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] 1 1 3 3 5 5
Number of elements within a vector:-  6
```

4. Sort a Vector

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

Example

```
eg_sort1 <-c("watermelon","banana","apple","orange")
cat("Character vector before sorting:- ",eg_sort1,"\n")
cat("Sorting in ascending order:- ",sort(eg_sort1),"\n")
cat("Sorting in descending order:-",sort(eg_sort1,decreasing = TRUE),"\n")
eg_sort2 <- c(52,30,96,48,66,10)
cat("Numeric vector befor sorting:- ",eg_sort2,"\n")
cat("Sorting in ascending order:- ",sort(eg_sort2),"\n")
cat("Sorting in descending order:-",sort(eg_sort2,decreasing = TRUE))
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
Character vector before sorting:-  watermelon banana apple orange
Sorting in ascending order:-  apple banana orange watermelon
Sorting in descending order:- watermelon orange banana apple
Numeric vector befor sorting:-  52 30 96 48 66 10
Sorting in ascending order:-  10 30 48 52 66 96
Sorting in descending order:- 96 66 52 48 30 10
```

5.Check if element present in R vector

- The %in% operator checks if each element of one vector is present in another vector, returning a logical vector.
  Syntax:- **vec1%in%vec2**

Example

```
eg_vec1 <- c(15,96,30,48,66,10)
eg_vec2 <- c(45,33,96,15,66,74)
eg_subset <- eg_vec1%in%eg_vec2
print(eg_subset)
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1]  TRUE  TRUE FALSE FALSE  TRUE FALSE
```


Vector – oriented Behaviour

- Vectors are so useful because they allow R to carry out operations on multiple elements simultaneously with speed and efficiency.
- This vector oriented, vectorized, or element-wise behavior is a key feature of the language.
- Vector oriented behavior in R refers to the ability to perform operations on entire vectors of data, rather than on individual elements.
- This can be a significant performance boost for certain types of operations, as it can avoid the overhead of looping over each element in a vector.

1. Provides element-wise operations

a. Element wise operation of same length

- Operations on two vectors of similar length is a straightforward process, we simply perform operations(add,sub,mul,div) on the corresponding elements together.

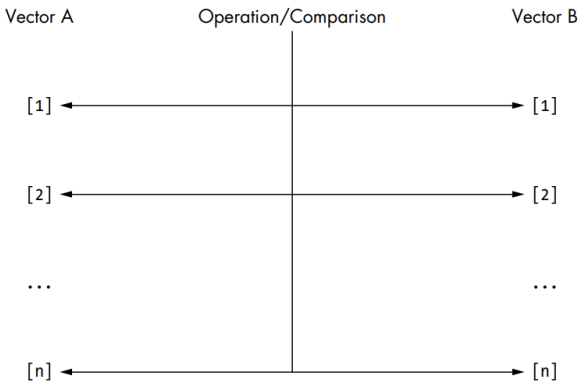


Figure 2-1: A conceptual diagram of the element-wise behavior of a comparison or operation carried out on two vectors of equal length in R. Note that the operation is performed by matching up the element positions.

Example:-

```
vec1 <- 10:15
print(vec1)
vec2 <- 1:6
print(vec2)
print(vec1+vec2)
print(vec1-vec2)
print(vec1*vec2)
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] 10 11 12 13 14 15
[1] 1 2 3 4 5 6
[1] 11 13 15 17 19 21
[1] 9 9 9 9 9 9
[1] 10 22 36 52 70 90
```

b. Element wise operations of different lengths

- The situation is made more complicated when using vectors of different lengths.
- R essentially attempts to replicate, or recycle, the shorter vector by as many times as needed to match the length of the longer vector, before completing the specified operation.

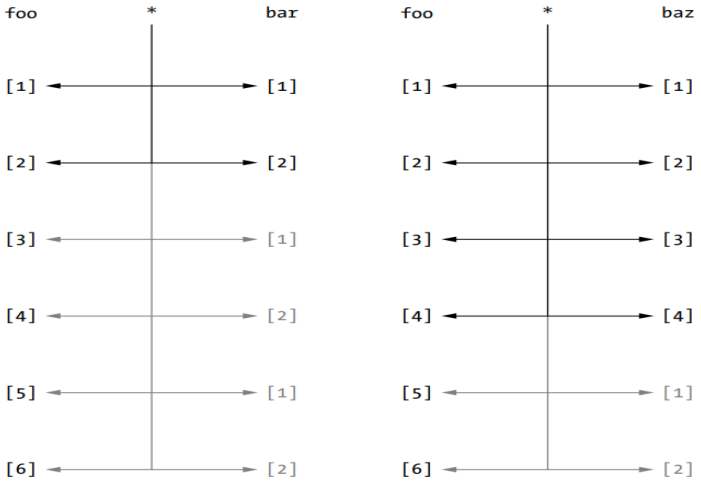


Figure 2-2: An element-wise operation on two vectors of differing lengths. Left: foo multiplied by bar; lengths are evenly divisible. Right: foo multiplied by baz; lengths are not evenly divisible, and a warning is issued.

Example:-

```
vec1 <- 10:15
print(vec1)
vec2 <- 1:3
print(vec2)
print(vec1+vec2)
print(vec1-vec2)
print(vec1*vec2)
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
[1] 10 11 12 13 14 15
[1] 1 2 3
[1] 11 13 15 14 16 18
[1]  9  9  9 12 12 12
[1] 10 22 36 13 28 45
```

c. Scalar- Vector operations

- Scalar vector operations are operations that are performed between a scalar and a vector. Scalars are numbers, while vectors are sequences of numbers.
- The corresponding elements of the scalar and the vector are operated together to produce a new element.

Example

```
a <- 2
sc_vec <- 10:15
print("Scalar vector Operations")
print("Addition")
print(a+sc_vec)
print("Subtraction")
print(a-sc_vec)
print("Division")
print(a/sc_vec)
print("Multiplication")
print(a*sc_vec)
```

d. Vectorized functions

- A vectorized function is a function that operates on an entire vector of data at once, rather than on individual elements of the vector.
- This can be much more efficient than operating on the elements of the vector one by one, especially for large vectors.

a. Sum: The sum() function calculates the sum of all elements in a vector.
b. Mean: The mean() function calculates the average (mean) of the elements in a vector.
c. Median: The median() function calculates the median of the elements in a vector.
d. Minimum and Maximum: The min() and max() functions return the minimum and maximum values, respectively, in a vector.

Example

```
x <- c(1, 2, 3, 4, 5)
sum_x <- sum(x)
cat("Sum of elements:- ",sum_x,"\n")
mean_x <- mean(x)
cat("Mean of elements:-",mean_x,"\n")
median_x <- median(x)
cat("Median of elements:-",median_x,"\n")
min_x <- min(x)
cat("Minimum element:-",min_x,"\n")
max_x <- max(x)
cat("Maxmimum element:-",max_x,"\n")
```

Output

```
> source("D:/ICCA/R_Program/vector.R")
Sum of elements:-  15
Mean of elements:- 3
Median of elements:- 3
Minimum element:- 1
Maxmimum element:- 5
```

R Matrics

- A matrix is a two dimensional data set with columns and rows.
- A column is a vertical representation of data, while a row is a horizontal representation of data.

Create a matrix in R

- A matrix can be created with the matrix() function.

Syntax:- **matrix(data, nrow, ncol, byrow, dimnames)**

Where,

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

Note:- Filling direction :- byrow defines how we want fill elements i.e either row wise or column wise.

Example 1:-

```
row_name = c(1:4)
col_name = c("col1","col2","col3","col4")
a = matrix(c(1:16),nrow = 4,ncol = 4,byrow = TRUE,dimnames = list(row_name,col_name))
print(a)
```

Output

```
  col1 col2 col3 col4
1    1    2    3    4
2    5    6    7    8
3    9   10   11   12
4   13   14   15   16
```

Example 2:-

```
row_name = c(1:4)
col_name = c("col1","col2","col3","col4")
a = matrix(c(1:16),nrow = 4,ncol = 4,dimnames = list(row_name,col_name))
print(a)
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
  col1 col2 col3 col4
1    1    5    9   13
2    2    6   10   14
3    3    7   11   15
4    4    8   12   16
```

Accessing Elements of Matrix (Extracting elements)

- We use the vector index operator [ ] to access specific elements of a matrix in R.

Syntax :- **Matrix_name[row_num,col_num]**

Where,

- Matrix_name :- Name of the matrix
- row_num :- specifies the row position
- col_num:- specifies the column position

Example

```
row_name = c(1:4)
col_name = c("col1","col2","col3","col4")
a = matrix(c(1:16),nrow = 4,ncol = 4,dimnames = list(row_name,col_name))
cat("Displaying complete matrix\n")
print(a)
cat("Accessing element present at 1st row,2nd column",a[1,2],"\n")
cat("Accessing element present at 4th row,3rd column",a[4,3])
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
Displaying complete matrix
  col1 col2 col3 col4
1    1    5    9   13
2    2    6   10   14
3    3    7   11   15
4    4    8   12   16
Accessing element present at 1st row,2nd column 5
Accessing element present at 4th row,3rd column 12
```

Accessing entire row or column

- In R, we can also access the entire row or column based on the value passed inside [].
  - [n,] - returns the entire element of the nth row.
  - [,n] - returns the entire element of the nth column.

Example

```
row_name = c(1:4)
col_name = c("col1","col2","col3","col4")
a = matrix(c(1:16),nrow = 4,ncol = 4,dimnames = list(row_name,col_name))
cat("Displaying complete matrix\n")
print(a)
cat("1st row :- ",a[1,])
cat("\n3rd column:- ",a[,3])
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
Displaying complete matrix
  col1 col2 col3 col4
1    1    5    9   13
2    2    6   10   14
3    3    7   11   15
4    4    8   12   16
1st row :-  1 5 9 13
3rd column:-  9 10 11 12
```


Access More Than One Row or Column

- We can access more than one row or column in R using the c() function.
  - [c(n1,n2),] - returns the entire element of n1 and n2 row.
  - [,c(n1,n2)] - returns the entire element of n1 and n2 column.

Example

```
row_name = c(1:4)
col_name = c("col1","col2","col3","col4")
a = matrix(c(1:16),nrow = 4,ncol = 4,dimnames = list(row_name,col_name))
cat("Displaying complete matrix\n")
print(a)
cat("1st and 3rd  row :- ",a[c(1,3),])
cat("\n3rd and 1st column:- ",a[,c(3,1)])
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
Displaying complete matrix
  col1 col2 col3 col4
1    1    5    9   13
2    2    6   10   14
3    3    7   11   15
4    4    8   12   16
1st and 3rd  row :-  1 3 5 7 9 11 13 15
3rd and 1st column:-  9 10 11 12 1 2 3 4
```

Omitting elements using negative index

- To delete or omit elements from a matrix, you again use square brackets, but this time with negative indexes.
- In R, you can access using negative indices to exclude specific elements from the list.
- When using negative indices, R returns all elements except the ones specified by the negative indices.
- To delete row or column, you can use negative indices.

Example

```
a = matrix(c(1:16),nrow = 4,ncol = 4)
print("original matrix")
print(a)
print("Accessing by omitting index a[3]")
print(a[-3,])
a <- a[-4,-4]
print("Printing after deletion of index a[4,4]")
print(a)
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
[1] "original matrix"
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
[1] "Accessing by omitting index a[3]"
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    4    8   12   16
[1] "Printing after deletion of index a[4,4]"
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
```

Row and Column Bindings

- If you have multiple vectors of equal length, you can quickly build a matrix by binding together these vectors using the built-in R functions, rbind and cbind.
- You can either treat each vector as a row (by using the command rbind) or treat each vector as a column (using the command cbind).

Example

```
v1 <- c(1,2,3)
v2 <- 4:6
print("rbind")
m1 <- rbind(v1,v2)
print(m1)
print("cbind")
m2 <- cbind(v1,v2)
print(m2)
```

Output

```
[1] "rbind"
   [,1] [,2] [,3]
v1    1    2    3
v2    4    5    6
[1] "cbind"
     v1 v2
[1,]  1  4
[2,]  2  5
[3,]  3  6
```

Matrix Dimensions

- Another useful function, dim, provides the dimensions of a matrix stored in your workspace.

Example

```
v1 <- c(1,2,3)
v2 <- 4:6
m1 <- rbind(v1,v2)
print(m1)
print("Dimension of Matrix")
print(dim(m1))
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
   [,1] [,2] [,3]
v1    1    2    3
v2    4    5    6
[1] "Dimension of Matrix"
[1] 2 3
```

Replacing Matrix values

- We can replace values using assignment operation.
- To replace values diagonally we use diag() function.

Example

```
a = matrix(c(1:16),nrow = 4,ncol = 4)
print("Oringinal Matrix")
print(a)
a[4,4] <- 32
print("Replacing Single value")
print(a)
a[2,] <- 101:104
print("Replacing entire row value")
print(a)
print("Replcing Multiple values")
a[c(1,3),c(1,3)] <- c(52,85)
print(a)
print("Replacing diagonal values")
diag(a) <- 0
print(a)
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
[1] "Oringinal Matrix"
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
[1] "Replacing Single value"
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   32
[1] "Replacing entire row value"
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]  101  102  103  104
[3,]    3    7   11   15
[4,]    4    8   12   32
[1] "Replcing Multiple values"
     [,1] [,2] [,3] [,4]
[1,]   52    5   52   13
[2,]  101  102  103  104
[3,]   85    7   85   15
[4,]    4    8   12   32
[1] "Replacing diagonal values"
     [,1] [,2] [,3] [,4]
[1,]    0    5   52   13
[2,]  101    0  103  104
[3,]   85    7    0   15
[4,]    4    8   12    0
```

Matrix Operations and Algebra

1.Matrix Transpose

- A matrix transpose is a matrix that is obtained by exchanging the rows and columns of a given matrix.
- For any m × n matrix A, its transpose, A >, is the n × m matrix obtained by writing either its columns as rows or its rows as columns.
- In R, the transpose of a matrix is found with the function t.

Example

```
a = c(3,4,1,2,5,7)
A <- matrix(a,nrow=3,ncol = 2)
print(A)
print("Transpose of a Matrix A")
print(t(A))
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
     [,1] [,2]
[1,]    3    2
[2,]    4    5
[3,]    1    7
[1] "Transpose of a MAtrix A"
     [,1] [,2] [,3]
[1,]    3    4    1
[2,]    2    5    7
```

2.Identity Matrix

- The identity matrix written as I_m is a particular kind of matrix used in mathematics. It's a square m × m matrix with ones on the diagonal and zeros elsewhere.
- You can create an identity matrix of any dimension using the standard matrix function, but there's a quicker approach using diag() function.

Example

```
a <- diag(x=4)
print(a)
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

3.Scalar Matrix Multiplication

- A scalar value is just a single, univariate value. Multiplication of any matrix A by a scalar value a results in a matrix in which every individual element is multiplied by a.
- R will perform this multiplication in an element-wise manner.
- Scalar multiplication of a matrix is carried out using the standard arithmetic * operator

Example

```
a <- 2
A <- matrix(c(3,4,1,2,5,7),nrow=3,ncol = 2)
print(A)
print("Scalar matrix Multiplication")
print(a*A)
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
     [,1] [,2]
[1,]    3    2
[2,]    4    5
[3,]    1    7
[1] "Scalar matrix Multiplication"
     [,1] [,2]
[1,]    6    4
[2,]    8   10
[3,]    2   14
```

4. Matrix Addition and subtraction

- Addition or subtraction of two matrices of equal size is also performed in an element-wise fashion.
- Corresponding elements are added or subtracted from one another, depending on the operation.

Example

```
A <- matrix(c(3,4,1,2,5,7),nrow=3,ncol = 2)
print("Matrix A")
print(A)
B <- matrix(c(5,2,1,3,4,8),nrow=3,ncol = 2)
print("Matrix B")
print(B)
print("Addition of 2 matrix")
print(A+B)
print("Subtraction of 2 matrix")
print(A-B)
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
[1] "Matrix A"
     [,1] [,2]
[1,]    3    2
[2,]    4    5
[3,]    1    7
[1] "Matrix B"
     [,1] [,2]
[1,]    5    3
[2,]    2    4
[3,]    1    8
[1] "Addition of 2 matrix"
     [,1] [,2]
[1,]    8    5
[2,]    6    9
[3,]    2   15
[1] "Subtraction of 2 matrix"
     [,1] [,2]
[1,]   -2   -1
[2,]    2    1
[3,]    0   -1
```

5.Matrix Multiplication

- In order to multiply two matrices A and B of size m × n and p × q, it must be true that n = p. The resulting matrix A · B will have the size m × q.
- The elements of the product are computed in a row-by-column fashion, where the value at position (AB)I, j is computed by element-wise multiplication of the entries in row I of A by the entries in column j of B, summing the result.
- R provide matrix product operator, written with percent symbols as %*%.

Example

```
A <- matrix(c(3,4,1,2,5,7),nrow=3,ncol = 2)
print("Matrix A")
print(A)
B <- matrix(c(5,2,1,3,4,8),nrow=2,ncol = 3)
print("Matrix B")
print(B)
print("Multiplication of Matrix")
print(A%*%B)
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
[1] "Matrix A"
     [,1] [,2]
[1,]    3    2
[2,]    4    5
[3,]    1    7
[1] "Matrix B"
     [,1] [,2] [,3]
[1,]    5    1    4
[2,]    2    3    8
[1] "Multiplication of Matrix"
     [,1] [,2] [,3]
[1,]   19    9   28
[2,]   30   19   56
[3,]   19   22   60
```

6.Matrix Inversion

- Some square matrices can be inverted. The inverse of a matrix A is denoted A^{-1}. An invertible matrix satisfies the following equation: $AA^{-1} = I_m$
- Matrices that are not invertible are referred to as singular.
- We use solve function to inverse a matrix.

Example

```
A <- matrix(c(3,4,1,2),nrow = 2,ncol = 2)
print("Matrix A")
print(A)
print("Inversion of Matrix A")
print(solve(A))
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
[1] "Matrix A"
     [,1] [,2]
[1,]    3    1
[2,]    4    2
[1] "Inversion of Matrix A"
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
```

- You can also verify that the product of these two matrices (using matrix multiplication rules) results in the 2 × 2 identity matrix

Example

```
A <- matrix(c(3,4,1,2),nrow = 2,ncol = 2)
print("Matrix A")
print(A)
print("Inversion of Matrix A")
print(solve(A))
print("Product of two matrix")
print(A%*%solve(A))
```

Output

```
> source("D:/ICCA/R_Program/eg_matrix.R")
[1] "Matrix A"
     [,1] [,2]
[1,]    3    1
[2,]    4    2
[1] "Inversion of Matrix A"
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
[1] "Product of two matrix"
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```


Array

- R arrays are data objects that store data in more than two dimensions. Arrays are homogeneous in nature.
- This means that they can store values of a single basic data type only. They store the data in the form of layered matrices.

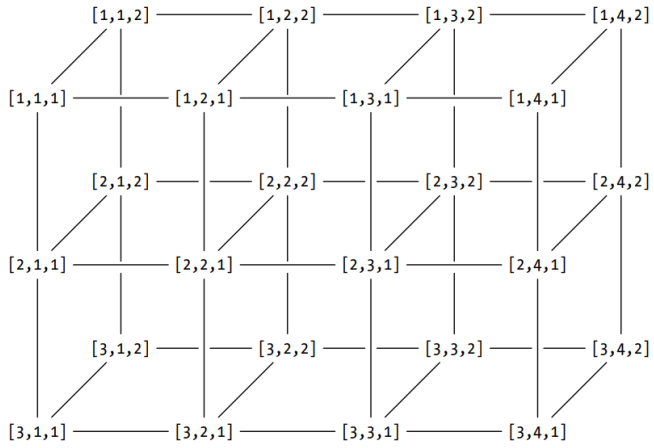


Figure 3-3: A conceptual diagram of a 3 × 4 × 2 array. The index of each element is given at the corresponding position. These indexes are provided in the strict order of `[row,column,layer]`.

Array Definition

- To create these data structures in R, use the array function.

Syntax

Array_name = array(data,dim = c(row_size,column_size,matrices), dimnames = list(row_names,column_names,matrices_names))

Where,

data: - is a vector that provides the values to fill the array,
dim: - is a vector that tells the dimensions of the array,
row_size: - is the number of rows in the array,
column_size: - is the number of columns in the array,
matrices: - denotes the number of matrices in the array,
dimnames: - is a list of names for the dimensions of the array,
row_names: - is a vector with the names for all the rows,
column_names: - is a vector with the names for all the columns,
matrices_names: - is a vector with the names for all the matrices in the array.

Example :- Three dimensional array creation

```
arr = array(1:12,dim=c(2,3,2))
print(arr)
```

Output

```
> source("D:/ICCA/R_Program/eg_array.R")
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Example 2: Four-dimensional array creation

```
arr = array(1:36, dim = c(2,3,3,2))
print(arr)
```

Output

```
> source("D:/ICCA/R_Program/eg_array.R")
, , 1, 1

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

, , 2, 1

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3, 1

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 1, 2

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24

, , 2, 2

     [,1] [,2] [,3]
[1,]   25   27   29
[2,]   26   28   30

, , 3, 2

     [,1] [,2] [,3]
[1,]   31   33   35
[2,]   32   34   36
```

Indexing R array

- We can access the elements of an array by using the square brackets to denote an index.

Example:- Accessing using positive index(3-dimensional array)

```
arr = array(1:12,dim=c(2,3,2))
print("3-dimensional array")
print(arr)
print("Accessing Single element")
print(arr[2,2,1])
print("Accessing multiple columns of same row")
print(arr[2,c(3,1),1])
print("Accessing first row of both matrix")
print(arr[1,,])
print("Accessing multiple row and column")
print(arr[c(1,2),c(2,3),2])
```

Output

```
> source("D:/ICCA/R_Program/eg_array.R")
[1] "3-dimensional array"
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

[1] "Accessing Single element"
[1] 4
[1] "Accessing multiple columns of same row"
[1] 6 2
[1] "Accessing first row of both matrix"
     [,1] [,2]
[1,]    1    7
[2,]    3    9
[3,]    5   11
[1] "Accessing multiple row and column"
     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

Example 2: Accessing using positive index(-dimensional array)

```
arr = array(1:36, dim = c(2,3,3,2))
print(arr)
print(arr[2,2,1,2])
print(arr[2,,2,2])
print(arr[1,,,1])
print(arr[,,2,])
print(arr[1,,,])
```

b. Accessing using negative index

Example

```
arr = array(1:12,dim=c(2,3,2))
print("3-dimensional array")
print(arr)
print("Excluding 2nd column of 1st array")
print(arr[,-2,1])
```


Output

```
> source("D:/ICCA/R_Program/eg_array.R")
[1] "3-dimensional array"
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

[1] "Excluding 2nd column of 1st array"
     [,1] [,2]
[1,]    1    5
[2,]    2    6
```

Modifying R array

- We can use the indexing techniques to access elements or parts of an array. Then we can use re-assignment to change their values.

Example:

```
arr = array(1:12,dim=c(2,3,2))
print("3-dimensional array")
print(arr)
arr[1,2,1] <- 100
print(arr)
```

Output

```
> source("D:/ICCA/R_Program/eg_array.R")
[1] "3-dimensional array"
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 1

     [,1] [,2] [,3]
[1,]    1  100    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

List

- A list in R can contain many different data types inside it.
- A list is a collection of data which is ordered and changeable.
- A single list could contain a numeric matrix, a logical array, a single character string, and a factor object

Creating List

- In R, we use the list() function to create a list.

Example,

```
# list with similar type of data
list1 <- list(24, 29, 32, 34)

# list with different type of data
list2 <- list("ICCA", 38, TRUE, matrix(1:6,nrow=2,ncol=3))
```

Here,

- list1 - list of integers
- list2 - list containing string, integer, Boolean value and a matrix1

Access List Elements in R

- You can retrieve components from a list using indexes, which are entered in double square brackets.

- This action is known as a member reference. When you've retrieved a component this way, you can treat it just like a stand-alone object in the workspace.

Example

```
list_eg <- list(c(10,20,30),matrix(1:6,nrow = 2,ncol = 3),"ICCA",25,26)
print(list_eg[[1]])
print(list_eg[[2]])
```

Output

```
[1] 10 20 30
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Note: In R, the list index always starts with 1. Hence, the first element of a list is present at index 1, second element at index 2 and so on.

Access List Elements in R using negative index

- In R, you can access list items using negative indices to exclude specific elements from the list.
- When using negative indices, R returns all elements except the ones specified by the negative indices.

Example

```
list1 <- list(24, 29, 32, 34)
print(list1[-2])
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[[1]]
[1] 24

[[2]]
[1] 32

[[3]]
[1] 34
```


Operations on List

1. Modify a List Element in R

a. Replacing item with in a list

- To change a list element, we can simply reassign a new value to the specific index.

Example

```
list_eg <- list(c(10,20,30),matrix(1:6,nrow = 2,ncol = 3),"ICCA",25,26)
print("List elements before Modifying")
print(list_eg)
print("List element after Modifying third element ")
list_eg[[3]] <- "Interface college"
print(list_eg)
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[1] "List elements before Modifying"
[[1]]
[1] 10 20 30

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

[[3]]
[1] "ICCA"

[[4]]
[1] 25

[[5]]
[1] 26

[1] "List element after Modifying third element "
[[1]]
[1] 10 20 30

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

[[3]]
[1] "Interface college"

[[4]]
[1] 25

[[5]]
[1] 26
```

b. Concatenating to existing item

- For adding content to existing element we use paste()

Example

```
list_eg <- list(c(10,20,30),matrix(1:6,nrow = 2,ncol = 3),"Interface college",25,26)
print("Before concatenating ")
print(list_eg[[3]])
print("After concatenation")
list_eg[[3]] <- paste(list_eg[[3]],"Of Computer Applications")
print(list_eg[[3]])
```

Output

```
"Before concatenation "
"Interface college"
"After concatenation"
"Interface college Of Computer Applications"
```

2.Add items to R List

a. Using append() function

- We use the append() function to add an item at the end.

Synatx :- **append(x, values, after = length(x))**

Where,

- x :- the vector the values are to be appended to.
- Values :- to be included in the modified vector.
- After :- a subscript, after which the values are to be appended.

Example 1

```
list1 <- list('ICCA','Davangere')
print("List items befor updating")
print(list1)
list1 <- append(list1,'BIET Road')
print("List items after updating")
print(list1)
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[1] "List items befor updating"
[[1]]
[1] "ICCA"

[[2]]
[1] "Davangere"

[1] "List items after updating"
[[1]]
[1] "ICCA"

[[2]]
[1] "Davangere"

[[3]]
[1] "BIET Road"
```

- We can also provide specific index where you want add item using after paratemeter in append. It specifies a subscript, after which the values are to be appended.

Example 2

```
list1 <- list('ICCA','Davangere')
print("List items befor updating")
print(list1)
list1 <- append(list1,'BIET Road', after = 1)
print("List items after updating")
print(list1)
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[1] "List items befor updating"
[[1]]
[1] "ICCA"

[[2]]
[1] "Davangere"

[1] "List items after updating"
[[1]]
[1] "ICCA"

[[2]]
[1] "BIET Road"

[[3]]
[1] "Davangere"
```


b. Adding item using index value

- We can add element at the end of the list using unused index.
- If our last index is 6, at 7th index we can add element using list name
  Synatx:- list_name[unused_index] <- value

Example

```
a <- list(1,2,3)
print("List items")
print(a)
cat("Length of list",length(a),"\n")
a[4] <- 7
```

```
print("After adding item")
print(a)
cat("Length of list after adding new element",length(a),"\n")
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[1] "List items"
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

Length of list 3
[1] "After adding item"
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 7

Length of list after adding new element 4
```

3. Remove Items from a List in R

- R allows us to remove items for a list.
- We first access elements using a list index and add negative sign - to indicate we want to delete the item.
- [-1] - removes 1st item
- [-2] - removes 2nd item and so on.

Example

```
list1 <- list('ICCA','Davangere','BIET Road')
print("List items before deleting")
print(list1)
list1 <- list1[-3]
print("List items after deleting")
print(list1)
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[1] "List items before deleting"
[[1]]
[1] "ICCA"

[[2]]
[1] "Davangere"

[[3]]
[1] "BIET Road"

[1] "List items after deleting"
[[1]]
[1] "ICCA"

[[2]]
[1] "Davangere"
```

Note:-
(1) To access list items, you use negative indices to exclude elements you don't want. For example, list1[-2] will return all elements of list1 except the second element.

(2) To delete list items, you also use negative indices to specify the elements you want to remove. By assigning the modified list back to the original variable, you effectively delete those items from the list. For example, list1<- list1[-2] removes the second item from list1.

b. By assigning NULL value.

Example

```
list1 <- list(24, 29, 32, 34)
print(list1)
list1[2] <- NULL
print("After deletion of ele 29")
print(list1)
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[[1]]
[1] 24

[[2]]
[1] 29

[[3]]
[1] 32

[[4]]
[1] 34

[1] "After deletion of ele 29"
[[1]]
[1] 24

[[2]]
[1] 32

[[3]]
[1] 34
```

4. Length() function

- In R, we can use the length() function to find the number of elements present inside the list.

Example

```
list1 <- list('ICCA','Davangere','BIET Road')
cat("Length of list1:-",length(list1))
```


Output

```
> source("D:/ICCA/R_Program/eg_list.R")
Length of list1:- 3
```

5. Loop over a list

- In R, we can also loop through each element of the list using the for loop

Example

```
list1 <- list('ICCA','BIET Road','Davangere')
for(i in list1)
{
  print(i)
}
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[1] "ICCA"
[1] "BIET Road"
[1] "Davangere"
```

6. Check if Element Exists in R List

- In R, we use the %in% operator to check if the specified element is present in the list or not and returns a boolean value.
  - TRUE - if specified element is present in the list
  - FALSE - if specified element is not present in the list

Example

```
list1 <- list('ICCA','BIET Road','Davangere')
cat('icca'%in% list1,"\n")
cat('Davangere'%in% list1)
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
FALSE
TRUE
```

7. Range of Indexes

- You can specify a range of indexes by specifying where to start and where to end the range, by using the : operator:

Example

```
list1 <- list('Interface','College of ','Computer','Applications','BIET road','Davangere')
print(list1[1:4])
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[[1]]
[1] "Interface"

[[2]]
[1] "College of "

[[3]]
[1] "Computer"

[[4]]
[1] "Applications"
```

Note: The search will start at index 2 (included) and end at index 5 (included).
Remember that the first item has index 1.

8.Join Two Lists

- There are several ways to join, or concatenate, two or more lists in R.
- The most common way is to use the c() function, which combines two elements together:

Example

```
list1 <- list('Interface','College of ','Computer')
list2 <- list('Applications','BIET road','Davangere')
list3 <- c(list1,list2)
print(list3)
```

Output

```
> source("D:/ICCA/R_Program/eg_list.R")
[[1]]
[1] "Interface"

[[2]]
[1] "College of "

[[3]]
[1] "Computer"

[[4]]
[1] "Applications"

[[5]]
[1] "BIET road"

[[6]]
[1] "Davangere"
```

Naming

- You can name list components to make the elements more recognizable and easy to work with.

a. using names()

- Using names() function we can provide names for the list.
- You can now perform member referencing using these names and the dollar operator, rather than the double square brackets.

Example

```
list1 <- list("SEETHA",29,"FEMALE","LECTURE")
names(list1) <- c("NAME","AGE","GENDER","DESIGNATION")
print(list1)
print(list1["NAME"])    #accessing using labels
print(list1$DESIGNATION)
```

Output

b. By assigning label

- To name the components of a list as it is being created, assign a label to each component in the list command.

Example

```
list2 <- list(Name="RAMA",Age=29,Gender="Male")
print(list2)
print(list2[1])
print(list2$Name)
```


Data frame

- A data frame is R's most natural way of presenting a data set with a collection of recorded observations for one or more variables.
- Like lists, data frames have no restriction on the data types of the variables; you can store numeric data, factor data, and so on.
- The R data frame can be thought of as a list with some extra rules attached.
- The most important distinction is that in a data frame (unlike a list), the members must all be vectors of equal length.

Construction

- To create a data frame from scratch, use the data.frame function.

- Supply your data, grouped by variable, as vectors of the same length—the same way you would construct a named list.

Example

```
dataframe1 <- data.frame(person= c("A","B","c","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
print(dataframe1)
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
  person age gender
1      A  25      F
2      B  63      M
3      c  20      M
4      D  45      F
5      E  34      F
```

- The data frames are printed to the console in rows and columns they look more like a matrix than a named list.
- This natural spreadsheet style makes it easy to read and manipulate data sets.
- Each row in a data frame is called a record, and each column is a variable.

Accessing Dataframe.

- To access a column of a data frame, we can use the square bracket [ ] or the double square brackets [[ ]] or the dollar sign $

a. Accessing using index

- Dataframe can be accessed using index, within a square bracket.

Example

```
dataframe1 <- data.frame(person= c("A","B","c","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
print("Accessing specific element")
print(dataframe1[2,2])
print("Accessing first,second,and third elements of first column")
print(dataframe1[1:3,1])
print("Accessing entire first and third row ")
print(dataframe1[c(1,3),])
print("Accessing entire second and third column")
print(dataframe1[,c(1,3)])
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
[1] "Accessing specific element"
[1] 63
[1] "Accessing first,second,and third elements of first column"
[1] "A" "B" "C"
[1] "Accessing entire first and third row "
  person age gender
1      A  25      F
3      C  20      M
[1] "Accessing entire second and third column"
  person gender
1      A      F
2      B      M
3      C      M
4      D      F
5      E      F
```

b. Accessing using names

- You can also use the names of the vectors that were passed to data.frame to access variables even if you don't know their column index positions, which can be useful for large data sets.
- To access using variables we use the same dollar operator.

Example

```
dataframe1 <- data.frame(person= c("A","B","c","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
print("Accessing person data")
print(dataframe1$person)
print("Accessing second element of age column,this is similar to dataframe1[2,2]")
print(dataframe1$age[2])
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
[1] "Accessing person data"
[1] "A" "B" "c" "D" "E"
[1] "Accessing second element of age column,this is similar to dataframe1[2,2]"
[1] 63
```


c. Accessing elements through some conditions(logical record subsets)

- Useful technique with data frames, where you'll often want to examine a subset of entries that meet certain criteria.
- For example, when working with data from a personal details, a researcher might want entries of only female candidate.

Example

```
dataframe1 <- data.frame(person= c("A","B","c","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
#Accessing using matrix like structure
print(dataframe1[dataframe1$gender=="F",])
# excluding column by num
print(dataframe1[dataframe1$gender=="F",-1])
# accessing column by name
print(dataframe1[dataframe1$gender=="F",c("person","gender")])
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
  person age gender
1      A  25      F
4      D  45      F
5      E  34      F
  age gender
1  25      F
4  45      F
5  34      F
  person gender
1      A      F
4      D      F
5      E      F
```

Modifying Data frames

a. Reassigning values

- We can modify a data frame using indexing techniques and reassignment.

Example

```
dataframe1 <- data.frame(person= c("A","B","c","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
print(dataframe1)
# by using index
dataframe1[1,2] <- 38
#by using variable name
dataframe1$age[2]<-100
dataframe1[4,"person"] <- "F"
print(dataframe1)
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
  person age gender
1      A  25      F
2      B  63      M
3      c  20      M
4      D  45      F
5      E  34      F
  person age gender
1      A  38      F
2      B 100      M
3      c  20      M
4      F  45      F
5      E  34      F
```

b. Adding rows

- We can add rows to a data frame by using the rbind() function.

Example

```
dataframe2 <- data.frame(person= c("A","B","C","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
print(dataframe2)
newrecord <- data.frame(person="F",age =20,gender="M")
dataframe2 <- rbind(dataframe2,newrecord)
print(dataframe2)
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
  person age gender
1      A  25      F
2      B  63      M
3      C  20      M
4      D  45      F
5      E  34      F
  person age gender
1      A  25      F
2      B  63      M
3      C  20      M
4      D  45      F
5      E  34      F
6      F  20      M
```

c. Adding columns

- Similarly, we can add a column by using the cbind()function.

Example

```
dataframe2 <- data.frame(person= c("A","B","C","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
print(dataframe2)
phone_num <- c(9732349829,8481789714,7164356909,9483328337,9744365687)
dataframe2 <- cbind(dataframe2,phone_num)
dataframe2$city <- c(rep("DvG",times=5))
print(dataframe2)
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
  person age gender
1      A  25      F
2      B  63      M
3      C  20      M
4      D  45      F
5      E  34      F
  person age gender  phone_num city
1      A  25      F 9732349829  DvG
2      B  63      M 8481789714  DvG
3      C  20      M 7164356909  DvG
4      D  45      F 9483328337  DvG
5      E  34      F 9744365687  DvG
```

Common dataframes functions

- nrow() – The nrow() function returns the number of rows or observations in a data frame.
- ncol() – The ncol() function returns the number of columns or variables in a data frame.
- length() – The length() function returns the length of a data frame which is the same as the ncol property.

Example

```
dataframe2 <- data.frame(person= c("A","B","C","D","E"),
                         age = c(25,63,20,45,34),
                         gender = c("F","M","M","F","F"))
print(dataframe2)
print(nrow(dataframe2))
print(ncol(dataframe2))
print(length(dataframe2))
```

Output

```
> source("D:/ICCA/R_Program/eg_dataframes.R")
  person age gender
1      A  25      F
2      B  63      M
3      C  20      M
4      D  45      F
5      E  34      F
[1] 5
[1] 3
[1] 3
```

Special Values

- There are several special values in R that have unique properties and are used in different ways. Here are some examples:

1. NA: This value represents Not Available or Missing data. It is often used when a value is not known or cannot be determined.

- Example
  - The NA value can be used to represent missing data in a data frame. For example, if you have a data frame with the columns Name, Age, and Gender, and you don't know the age of one of the people in the data frame, you can use the NA value to represent the missing age.

```
> age <- list(10,56,NA,32,25)
> age
[[1]]
[1] 10

[[2]]
[1] 56

[[3]]
[1] NA

[[4]]
[1] 32

[[5]]
[1] 25
```

- In R, the is.na() function is used to check for missing or NA (Not Available) values in an object.

Example:-

```
> vec <- c(1,2,3,NA)
> is.na(vec)
[1] FALSE FALSE FALSE  TRUE
```

2. Inf: This value represents Infinity. It is used to represent numbers that are infinitely large.
-Inf: This value represents Negative Infinity. It is used to represent numbers that are infinitely small.

Example:-

- The Inf value can be used to represent infinity in mathematical expressions. For example, the expression 1/0 evaluates to Inf because it is mathematically undefined to divide by zero.
- The -Inf value can be used to represent negative infinity in mathematical expressions. For example, the expression -1/0 evaluates to -Inf because it is mathematically undefined to divide by zero.

```
> 1/0
[1] Inf
> 5/0
[1] Inf
> -895/0
[1] -Inf
```

- In R, the is.infinite() function is used to check for infinite values in an object.
- The is.infinite() function returns a logical vector indicating which elements of the input are infinite.

Example

```
> vec <- c(1,-Inf,3,Inf)
> is.infinite(vec)
[1] FALSE  TRUE FALSE  TRUE
```

3. NaN: This value represents Not a Number. It is used to represent numbers that are not mathematically defined.

- Example
  - The NaN value can be used to represent numbers that are not mathematically defined.
  - For example, the expression 0/0 evaluates to NaN because it is mathematically undefined to divide by zero.

```
> 0/0
[1] NaN
> sqrt(-9)
[1] NaN
```

- In R, the is.nan() function is used to check for NaN (Not-a-Number) values in an object.
- The is.nan() function returns a logical vector indicating which elements of the input are NaN.

```
> vec <- c(1,2,3,NaN)
> is.nan(vec)
[1] FALSE FALSE FALSE  TRUE
```


4. NULL: This value represents Nothing. It is used to represent a variable that has no value.

- Example
  - The NULL value can be used to represent a variable that has no value.
  - For example, if you create a variable called my_variable, but you don't assign it any value, the value of my_variable will be NULL.

```
> null_eg <- NULL
> print(null_eg)
NULL
```

- Is.null( ) :- n R, the is.null() function is used to check if an object is of the special type NULL.
- The is.null() function returns TRUE if the provided argument is NULL, and FALSE otherwise.

```
> x<- NULL
> is.null(x)
[1] TRUE
```

Coercion

- Coercion in R is the process of converting an object from one data type to another. This can happen implicitly or explicitly.

Implicit coercion

- Implicit coercion occurs when R automatically converts an object to a different type in order to perform an operation.
- For example, if you try to add a numeric value to a character value, R will implicitly convert the character value to a numeric value.

Explicit coercion(As-Dot Coercion Functions)

- Explicit coercion occurs when you explicitly convert an object to a different type using a coercion function.
- For example, the as.numeric() function can be used to convert a character value to a numeric value.
- The following are some of the common coercion functions in R:
  - as.character(): Converts an object to a character value.
  - as.numeric(): Converts an object to a numeric value.
  - as.integer(): Converts an object to an integer value.
  - as.logical(): Converts an object to a logical value.
  - as.complex(): Converts an object to a complex value.

Example

```
num.vec <- c(1,2,3,4,5,6)
print(num.vec)
print(class(num.vec))
num.str<-as.character(num.vec)
print(num.str)
print(class(num.str))
```

Output

```
> source("D:/ICCA/R_Program/example.R")
[1] 1 2 3 4 5 6
[1] "numeric"
[1] "1" "2" "3" "4" "5" "6"
[1] "character"
```


Is-Dot Object-Checking Functions

- Identifying the class of an object is essential for functions that operate on stored objects, especially those that behave differently depending on the class of the object.
- To check whether the object is a specific class or data type, you can use the is-dot functions on the object and it will return a TRUE or FALSE logical value.
- The following are some of the common object checking functions.
  - is.numeric()
  - is.integer()
  - is.matrix()
  - is.vector()
  - is.data.frame()

Example

```
a<- c("row1","row2")
print(is.character(a))
num.vec <- c(1,2,3,4,5,6)
print(is.numeric(num.vec))
print(is.integer(num.vec))
```

Output

```
> source("D:/ICCA/R_Program/example.R")
[1] TRUE
[1] TRUE
[1] FALSE
```

Basic Plotting

- One particularly popular feature of R is its incredibly flexible plotting tools for data and model visualization.
- The easiest way to think about generating plots in R is to treat your screen as a blank, two-dimensional canvas. You can plot points and lines using x and y-coordinates

1. plot() function

- The plot() function in R is a generic function that is used to plot points, lines, and other shapes on a graph.
- It is one of the most commonly used functions in R for data visualization.
- There are a wide range of graphical parameters that can be supplied as arguments to the plot function.
- Some of the most commonly used **graphical parameters** are listed here.
    - type:- what type of plot should be drawn.
    - main:- an overall title for the plot
    - xlab and ylab:- the horizontal axis label, and the vertical axis label.
    - col :- Color (or colors) to use for plotting points and lines.
    - pch :- Stands for point character. This selects which character to use for plotting individual points.
    - cex :- Stands for character expansion. This controls the size of plotted point characters.
    - lty :- Stands for line type. This specifies the type of line to use to connect the points (for example, solid, dotted, or dashed).
    - lwd :- Stands for line width. This controls the thickness of plotted lines.
    - xlim and ylim :- This provides limits for the horizontal range and vertical range (respectively) of the plotting region.

a. Using plot with Coordinate Vectors

- The R function plot, on the other hand, takes in two vectors—one vector of x locations and one vector of y locations.

Example:-

```
x_cordinate <- c(1.1,2,3.5,3.9,4.2)
y_cordinate <- c(2,2.2,-1.3,0,0.2)
plot(x_cordinate,y_cordinate)
```

b. plot type

- By default, the plot function will plot individual points.
- This is the default plot type, but other plot types will have a different appearance. To control the plot type, you can specify a single character valued option for the argument type
    - Possible types are
        - "p" for points,
        - "l" for lines,
        - "b" for both,
        - "c" for the lines part alone of "b",
        - "o" for both 'overplotted',
        - "h" for 'histogram' like (or 'high-density') vertical lines,

- "s" for stair steps,
- "S" for other steps, see 'Details' below,
- "n" for no plotting.

Example:- plot(x,y, type = "b")

c. Title and Axis Labels

- By default, a basic plot won't have a main title, and its axes will be labelled with the names of the vectors being plotted.
- But a main title and more descriptive axis labels often make the plotted data easier to interpret.
- You can add these by supplying text as character strings to main for a title, xlab for the x-axis label, and ylab for the y-axis label.
- By specifying the las (label style) argument, you can change the axes label style. This changes the orientation angle of the labels.
  - 0 :- The default, parallel to the axis
  - 1 :- Always horizontal
  - 2 :- Perpendicular to the axis
  - 3 :- Always vertical

Example:- plot(x,y,main = "Plot Example", xlab = "X Values", ylab="Y Values

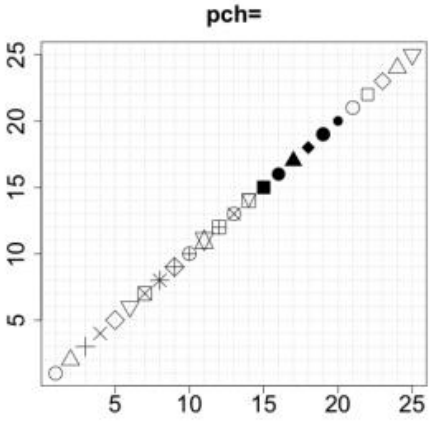
d. Color

- You can set colors with the col parameter in a number of ways.
- The simplest options are to use an integer selector or a character string.
- There are a number of color string values recognized by R, which you can see by entering colors() at the prompt.
- The default color is integer 1 or the character string "black".
- You can also specify colors using RGB (red, green, and blue) levels and by creating your own palettes.

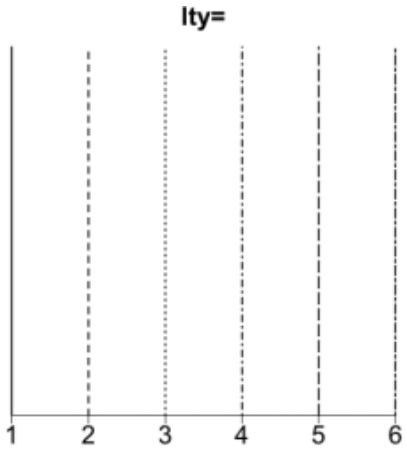
Example:- plot(x,y,pch=17,col="blue")

e. Line and Point Appearances

- To alter the appearance of the plotted points you would use pch, values ranging from 0 to 25.



- To alter the lines you would use lty, values ranging from 1 to 6.



- You can also control the size of plotted points using cex and the thickness of lines using lwd.

Example:- plot(x,y,pch=17,lty=3,cex=2,lwd=2)

f. Plotting Region Limits

- By default R sets the range of each axis by using the range of the supplied x and y values.
- But you might need more space than this to, for example, annotate individual points, add a legend, or plot additional points that fall outside the original ranges.
- You can set custom plotting area limits using xlim and ylim.
- Both parameters require a numeric vector of length 2, provided as c(lower,upper)

Example:- plot(x,y,xlim=c(-2,0.5),ylim=c(1,5))

g. The Box Type

- Specify the bty (box type) argument to change the type of box round the plot area.
  - "o" :- (default) Draws a complete rectangle around the plot.
  - "n" :- Draws nothing around the plot
  - "l", "7","u","]","c" :- Draws a shape around the plot area.

Example :- plot(x,y,bty = "7")

h. Add a Grid

- The plot() function does not automatically draw a grid. However, it is helpful to the viewer for some plots.
- Call the grid() function to draw the grid once you call the plot().

Example:- plot(x,y)
grid()

i. Add a Legend

- You can include a legend to your plot – a little box that decodes the graphic for the viewer.
- Call the legend() function, once you call the plot()
- The position of the legend can be specified using the following keywords : "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" and "center".

Example :- legend("topleft","group1",pch =17,col="blue")

j. Add Points to a Plot

- You can add points to a plot with the points() function.

Example

```
x <- c(1.1,2,3.5,3.9,4.2)
y <- c(2,2.2,-1.3,0,0.2)
plot(x, y, main = "Plot Example",
    xlab = "X Values", ylab = "Y Values", pch =17 , col = "blue",type = "b",
    lty = 3,cex=2,lwd=2,xlim = c(-2,5.0))
```

k. Add Lines to a Plot

- You can add lines to a plot in a very similar way to adding points, using abline().

Example:- abline(h=c(-0.5,1.5),v=1.5,col="red",lty=3,lwd=3)

l. Label Data Points

- Use the text() function to add text labels at any position on the plot.
- The position of the text is specified by the pos argument.
- Values of 1, 2, 3 and 4, respectively places the text below, to the left of, above and to the right of the specified coordinates.

Example:-

```
labeling = c("(1.1,2)","(2,2.2)","(3.5,-1.3)","(3.9,0)","(4.2,4)")
text(x,y,labels = labeling,pos=3,cex=0.8)
```

m. Display Multiple Plots on a Single Page

- There are two main ways to plot multiple plots in a single page in R:
  - Using the par() function.
  - Using the gridExtra package.
- Using the par() function
  - The par() function is a built-in R function that can be used to control the graphical parameters of plots.
  - One of the parameters that can be controlled with the par() function is the mfrow parameter.
  - The mfrow parameter specifies the number of rows and columns of subplots in a plot.
  - To plot two plots in a single page using the par() function, you can use the following steps:
    - Create the first plot.
    - Set the mfrow parameter to c(2, 1). This will create a plot with two rows and one column.
    - Create the second plot.
    - Set the mfrow parameter back to its original value.

Example

```
x<- c(1.1,2,3.5,3.9,4.2)
y <- c(2,2.2,-1.3,0,4)
par(mfrow=c(2,2))
plot(x, y, main = "Plot Example 1", xlab = "X Values", ylab = "Y Values", pch =17 ,
     col = "blue",type = "b", lty = 3,cex=2,lwd=2,xlim = c(-2,5.0),ylim= c(-2,5),las=2,bty = "o")

plot(x, y, main = "Plot Example 2", xlab = "X Values", ylab = "Y Values", pch =17 ,
     col = "blue",type = "b", lty = 3,cex=2,lwd=2,xlim = c(-2,5.0),ylim= c(-2,5),las=2,bty = "o")

plot(x, y, main = "Plot Example 3", xlab = "X Values", ylab = "Y Values", pch =17 ,
     col = "blue",type = "b", lty = 3,cex=2,lwd=2,xlim = c(-2,5.0),ylim= c(-2,5),las=2,bty = "o")

plot(x, y, main = "Plot Example 4", xlab = "X Values", ylab = "Y Values", pch =17 ,
     col = "blue",type = "b", lty = 3,cex=2,lwd=2,xlim = c(-2,5.0),ylim= c(-2,5),las=2,bty = "o")
par(mfrow=c(1,1))
```

Histogram

- hist() function: It is used for drawing histogram for a frequency distribution with or without equal class-width. It can also be used to draw histogram of observations.
- The syntax for the hist() function is:
  hist(x,breaks,freq,labels,density,angle,col,border,main,xlab,ylab,…)

Parameter	Description
x	A vector of values describing the bars which make up the plot
breaks	A number specifying the number of bins for the histogram
freq	If TRUE, hist() gives counts instead of probabilities.
labels	If TRUE, draws labels on top of bars
density	The density of shading lines
angle	The slope of shading lines
col	A vector of colors for the bars
border	The color to be used for the border of the bars
main	An overall title for the plot
xlab	The label for the x axis
ylab	The label for the y axis

Barplot

- barplot() function: It creates a barplot with vertical or horizontal bars.
- The syntax for the barplot() function is:
  barplot(x,y,type,main,xlab,ylab,pch,col,las,bty,bg,cex,…)

Pie chart

- pie() or piechart() function: It is used for drawing a pie chart.
- The syntax for the pie() function is:
  pie(clockwise,init.angle,labels,density,angle,col,border,lty,main,…)

Parameter	Description
clockwise	If True, slices are drawn clockwise ortherwise counter-clockwise
init.angle	The starting angle for the slices
labels	The names for the slices
density	The density of shading lines
angle	The slope of shading lines
col	A vector of colors to be used in filling or shading the slices
border	The color to be used for the border
lty	Type of lines used for plotting pie chart
main	An overall title for the plot

Example

```
survey <- c(apple=40, kiwi=15, grape=30, banana=50, pear=20, orange=35)
par(mfrow=c(2,2))
barplot(survey,col="blue")
plot(survey)
pie(survey,clockwise = TRUE)
hist(survey)
par(mfrow=c(1,1))
```

Unit :- 2

Functions

- Functions are created using the function() directive and are stored as R objects just like anything else.
- In particular, they are R objects of class "function".

Synatx:-

```
Function_name  <- function() {
        ## Block of code
}
```

- Functions in R are "first class objects", which means that they can be treated much like any other R object.
-
- Importantly, Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

1. Adding Arguments
   - Arguments in R are the values that are passed to a function when it is called.
   - Functions can have any number of arguments, and the arguments can be of any type, such as numbers, strings, vectors, matrices, data frames, and other functions.

Example

```
fun1 <- function(a,b){
  print(a)
  print(b)

}
fun1(10,20)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 10
[1] 20
```

2. Returning Results
   - If you want to use the results of a function in future operations (rather than just printing output to the console), you need to return content to the user.

Example

```
fun1 <- function(a,b){
  c <- a+b
  return(c)
}
res <- fun1(10,20)
print(res)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 30
```

3. Argument
   - Arguments are an essential part of most R functions. In this section, you'll consider how R evaluates arguments.

- You'll also see how to write functions that have default argument values, how to make functions handle missing argument values, and how to pass extra arguments into an internal function call with ellipses, etc…

a. Argument Matching

- Another set of rules that determine how R interprets function calls has to do with argument matching.
- Argument matching conditions allow you to provide arguments to functions either with abbreviated names or without names at all.

i. Exact

- Exact argument matching is a type of argument matching in R, where each argument tag is written out in full.
- This is the most exhaustive way to call a function.
- Other benefits of exact matching include the following:
  - Exact matching is less prone to mis-specification of arguments than other matching styles. •
  - The order in which arguments are supplied doesn't matter.
  - Exact matching is useful when a function has many possible arguments but you want to specify only a few.
- The main drawbacks of exact matching are clear:
  - It can be cumbersome for relatively simple operations.
  - Exact matching requires the user to remember or look up the full, case sensitive tags.

Example

```
res <- matrix(nrow=3,dimnames=list(c("A","B","C"),c("D","E","F")),ncol=3,data=1:9)
print(res)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

ii. Partial matching

- Partial matching lets you identify arguments with an abbreviated tag.
- This can shorten your code, and it still lets you provide arguments in any order.
- Partial matching has the following benefits: •
  - It requires less code than exact matching.
  - Argument tags are still visible (which limits the possibility of mis-specification).
  - The order of supplied arguments still doesn't matter
- Drawbacks of partial matching include the following: •
  - The user must be aware of other potential arguments that can be matched by the shortened tag (even if they aren't specified in the call or have a default value assigned).
  - Each tag must have a unique identification, which can be difficult to remember.

Example

```
res <- matrix(nr=3,di=list(c("A","B","C"),c("D","E","F")),nc=3,dat=1:9)
print(res)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

iii. Positional

- The most compact mode of function calling in R is positional matching.
- This is when you supply arguments without tags, and R interprets them based solely on their order.
- Positional matching is usually used for relatively simple functions with only a few arguments, or functions that are very familiar to the user.
- The benefits of positional matching are as follows:
  - Shorter, cleaner code, particularly for routine tasks
  - No need to remember specific argument tags.
- Drawbacks of positional matching:
  - You must look up and exactly match the defined order of arguments.
  - Reading code written by someone else can be more difficult, especially when it includes unfamiliar functions.

Example

```
res <- matrix(1:9,3,3,F,list(c("A","B","C"),c("D","E","F")))
print(res)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

iv. Mixed

- Since each matching style has pros and cons, it's quite common, and perfectly legal, to mix these three styles in a single function call.

Example

```
res <- matrix(1:9,3,3,dim=list(c("A","B","C"),c("D","E","F")))
print(res)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```


b. Default Values

- Default values in R are values that are used for function arguments when the arguments are not specified when the function is called.
- Default values can be specified for any argument, and they can be of any type, such as numbers, strings, vectors, matrices, data frames, and other functions.
- To specify a default value for an argument, you simply add the equal sign (=) and the default value after the argument name.

Example

```
fun1 <- function(a,b=45){
  c <- a+b
  return(c)
}
res <- fun1(10)
print(res)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 55
```

c. Lazy Evaluation

- Arguments to functions are evaluated lazily, so they are evaluated only as needed.

Example

```
f <- function(a, b) {
        a^2
    }
    f(2)
```

- This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

Example 2

```
f <- function(a, b) {
  print(a)
  print(b)
}
f(45)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 45
Error in f(45) : argument "b" is missing, with no default
```

- Notice that “45” got printed first before the error was triggered.
- This is because b did not have to be evaluated until after print(a).
- Once the function tried to evaluate print(b) it had to throw an error

d. Checking for missing arguments

- The missing function checks the arguments of a function to see if all required arguments have been supplied.
- It takes an argument tag and returns a single logical value of TRUE if the specified argument isn’t found.

Example

```
fun1 <- function(a,b){
  if(missing(a) || missing(b)){
    print("Either argument a or b is missing")
  }
  else{
    c <- a+b
    print(c)
  }

}
fun1(10)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] "Either argument a or b is missing"
```

e. Dot-dot-dot(…) or Ellipsis

- The dot-dot-dot (...) in R is a special argument called the ellipsis. It allows a function to take a variable number of arguments.

- For example, the following function takes a variable number of arguments and prints the sum of all the arguments:

Example

```
fun_dot <- function(...){
  x <- as.integer(list(...))
  sum(x)
}
print(fun_dot(10,20,30,40,50))
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 150
```

Note:- Use the list(...) function to convert the ellipsis to a list. This can be useful for accessing the arguments individually.

Specialized Functions

- In this section, you'll look at three kinds of specialized user-defined R function.
  1. Helper function
  2. Disposable function
  3. Recursive function.

1. **Helper Function**

- It is common for R functions to call other functions from within their body code.
- A helper function is a general term used to describe functions written and used specifically to facilitate the computations carried out by another function.
- They're a good way to improve the readability of complicated functions.
- A helper function can be either defined internally (within another function definition) or externally (within the global environment)

a. Externally Defined

- An externally defined helper function is a function that is defined globally or in a separate file or package and is imported or loaded into your R script or session when needed.

Example :-

helper_function.R

```
test <- function(a,b){
  if (missing(a) || missing(b)){
    print("One or both values are misssing")
  }
}
```

eg_function.R

```
source("helper_function.R")
check <- function(x,y){
  test(x,y)
}
check(10)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] "One or both values are misssing"
```

- In the above example a test() function defined within a separate file named helper_function.R and included in the file eg_function.R using source() function.

b. Internally Defined

- An internally defined helper function is a function that is defined within the same script or R environment where it is used.
- These functions are declared directly within the script, function, or code block where they are needed, and they are not stored in external files or packages.
- Internally defined helper functions are often used for small, specific tasks within a particular context.

Example:-

```
check <- function(x,y){
  test <- function(a,b){
   a+b        direct return
  }
  result <- test(x,y)
  return(result)
}
print(check(10,10))
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 20
```

2. **Disposable function**

- In R, a "disposable function" is not a formal or standard term, but it can refer to a function that is designed for a specific, one-time use or a limited scope.
- These functions are often created to perform a single, isolated task, and they may not be reused elsewhere in your code.
- They are sometimes referred to as "throwaway functions" or "anonymous functions."
- You can create disposable functions using anonymous function expressions, which are functions created on the fly without assigning them a name.
- Anonymous functions are defined using the function() construct.

Example

```
(function(x,y){
  z <- x+y
  print(z)
})(3,5)
```


Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 8
```

- Disposable function can be passed to another function as a argument.

Example

```
list_ele <- list(1,2,3,4,5)
res <- lapply(X=list_ele,FUN=(function(x){2 * x}))
print(res)
```

Output

```
test=function(in_fun)
{
  print(in_fun(10))

}

test((function(x){(x+1)}))#returns x+1
```

```
> source("D:/ICCA/R_Program/eg_function.R")
[[1]]
[1] 2

[[2]]
[1] 4

[[3]]
[1] 6

[[4]]
[1] 8

[[5]]
[1] 10
```

- lapply(): This function applies a given function to each element of a list and returns a new list.

3. **Recursive Function**

- A recursive function in R is a function that calls itself during its execution.
- Recursive functions are commonly used to solve problems that can be broken down into smaller, similar sub-problems.
- Recursive functions often consist of two main components:
  - Base Case: A condition that specifies when the recursion should stop. When this condition is met, the function returns a value or performs some final action without making another recursive call.
  - Recursive Case: The part of the function where it calls itself with modified arguments to solve a smaller sub-problem. The function makes one or more recursive calls to itself with the goal of reaching the base case.

Example

```
factorial_recursive <- function(n) {
  if (n == 0) {

    return(1)
  } else {
    return(n * factorial_recursive(n - 1))
  }
}
result <- factorial_recursive(5)
cat("Factorial of 5 is:", result, "\n")
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
Factorial of 5 is: 120
```

Scoping in R

- Scoping in R refers to the visibility of variables and functions in different parts of a program.
- R uses lexical scoping, which means that the visibility of a variable is determined by the environment in which it was defined.
- An environment is a collection of objects, such as variables, functions, and data frames. There are many different environments in R, including:
  - The global environment: This is the environment that is active when you start an R session. It contains all of the objects that you create in the top-level of your R code.
  - Function environments: When you define a function in R, a new environment is created to store the function's arguments and local variables.
  - Package environments: When you load an R package, a new environment is created to store the package's objects

Lexical scoping

- Consider the following function.

```
f <- function(x, y) {
   x^2 + y / z
 }
```

- This function has 2 formal arguments x and y.
- In the body of the function there is another symbol z.
- In this case z is called a free variable.
- The scoping rules of a language determine how values are assigned to free variables.
- Free variables are not formal arguments and are not local variables (assigned inside the function body).

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.
- The search continues down the sequence of parent environments until we hit the top-level environment; this usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the empty environment.
- If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown

Example

```
t=20
p <- function(q){
  s <- 99
  print(q)
  print(s)
  print(t)
  print("Printing the elememt, which is not declared in current program.")
  print(a)
}
p(89)
```

Output

```
> source("D:/ICCA/R_Program/eg_function.R")
[1] 89
[1] 99
[1] 20
[1] "Printing the elememt, which is not declared in current program."
[1] 10
```

Reading Files in R programming

1. **Operations with Built in data set**
   - The data sets that come with R or one of the packages are known as resident data sets or inbuilt data sets.
   - Enter data() at the prompt to bring up a window listing these ready-to use data sets along with a one-line description. These data sets are organized in alphabetical order by name and grouped by package
   - To see a summary of the data sets contained in the package, you can use the library function as follows: library(help="datasets")
   - R-ready data sets have a corresponding help file where you can find important details about the data and how it's organized using ?dataset_name statement at the prompt.

- For example, ?chikWeight to view the detailed documentations about the dataset chickWeight

2. **Operations with contributed data sets**
   - There are many more R-ready data sets that come as part of contributed packages. To access them, first install and load the relevant package
   - First, you have to install the package, which you can do by running the line install.packages("tseries") at the prompt.
   - Then, to access the components of the package, load it using library. library("tseries")
   - To access this object itself, you must explicitly load it using the data function. Eg:- data("ice.river")
   - Then you can work with ice.river in your workspace as usual.

3. **Reading in External Data Files**
   - To analyze your own data, however, you'll often have to import them from some external file.
   - R has a variety of functions for reading characters from stored files and making sense of them.

a. read.table()

- The read.table() function in R is used to read tabular data from a file into a data frame. It is a very versatile function and can be used to read data from a variety of file formats, including .csv, .txt.

Syntax:- **mydatafile <- read.table(file="/Users/tdavies/mydatafile.txt", header=TRUE, sep=" ", na.strings="*", …)**

Where,

File = filename

Header = If a header is present, it's always the first line of the file. This optional feature is used to provide names for each column of data.

Sep = The all-important delimiter is a character used to separate the entries in each line. The delimiter character cannot be used for anything else in the file.

na.string = This is another unique character string used exclusively to denote a missing value. When reading the file, R will turn these entries into the form it recognizes: NA.

Example


```
file <- read.table("D:/ICCA/R_Program/egfile.txt",header = TRUE,sep = " ",na.strings = '*')
print(file)
file1 <- read.table("D:/ICCA/R_Program/Book1.csv",header = TRUE,sep = ",")
print(file1)
```

Note 1: - R can read in files from a website with the same read.table command. All the same rules concerning headers, delimiters, and missing values remain in place; you just have to specify the URL address of the file instead of a local folder location.

Note 2: - You can view textual output of the contents of any folder by using list.files().

b. file.choose()

- The file.choose() function in R is used to interactively select a file from the user's computer. It opens a file explorer window where the user can browse to and select the desired file. The function returns the path to the selected file as a character string.

Example

```
filepath <- file.choose()
print(filepath)
fileopen <- read.table(filepath,header = TRUE,sep = " ",na.strings = "*")
print(fileopen)
```

c. read.csv()

- The read.csv() function in R is used to read comma-separated value (CSV) files into data frames. It is a very versatile function and can be used to read data from a variety of sources.

Syntax:- **read.csv(file, header = TRUE, sep = ",")**

Where,

- file: The path to the CSV file to be read.
- header: A logical value indicating whether the CSV file has a header row. If TRUE, the header row will be used to name the columns of the data frame.
- sep: The character used to separate the columns of the CSV file. By default, sep is ",".

Example

```
fileopen <- read.csv("D:/ICCA/R_Program/Book1.csv",header = TRUE,sep = ",")
print(fileopen)
```

Note: - Spreadsheet Workbooks

- The standard file format for Microsoft Office Excel is .xls or .xlsx. In general, these files are not directly compatible with R.
- There are some contributed package functions that attempt to bridge this gap, , for example, gdata by Warnes et al. (2014) or XLConnect by Mirai Solutions GmbH (2014)—but it's generally preferable to first export the spreadsheet file to a table format, such as CSV.
- Then you can access these files using read.csv() or read.table() functions.

Note: - Web-Based Files

- With an Internet connection, R can read in files from a website with the same read.table command.
- All the same rules concerning headers, delimiters, and missing values remain in place; you just have to specify the URL address of the file instead of a local folder location

Writing Out Data Files and Plots

1. **Write.table()**

- The write.table() function in R is used to write a data frame or matrix to a file. It is a very versatile function and can be used to write data to a variety of file formats, including .csv and .txt.

Syntax:-

write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",eol="\n",na="NA",…)

Where,

- x: The data frame or matrix to be written.
- file: The name of the file to write the data to. If "", the data will be written to the console.
- append: A logical value indicating whether to append the data to the existing file. If FALSE, the existing file will be overwritten.
- sep: The character used to separate the data values. By default, sep is " ".

- eol: The character used to separate the lines of the file. By default, eol is "\n".
- na: The character used to represent missing values. By default, na is "NA".

Example

```
dt <- head(chickwts)
write.table(dt,"D:/ICCA/R_Program/egfile.txt",sep="@",)
file <- read.table("D:/ICCA/R_Program/egfile.txt",sep = "@",na.strings = '*')
print(file)
```

2. **Writing plot and graphic files**

- Plots can also be written directly to a file.
- Instead of displaying the plot immediately on the screen, you can have R follow these steps: open a “file” graphics device, run any plotting commands to create the final plot, and close the device.
- When you’ve finished plotting, you must explicitly close the file device with a call to dev.off()

Syntax:-

jpeg(file = "", width = 480, height = 480, pointsize = 12, quality = 75, bg = "white", res = NA)

Where,

- file: The name of the file to write the image to. If "", the image will be displayed in the R graphics window.
- width: The width of the image in pixels.
- height: The height of the image in pixels.
- pointsize: The default pointsize of plotted text, interpreted at 72 dpi, so one point is approximately one pixel.
- quality: The 'quality' of the JPEG image, as a percentage. Smaller values will give more compression but also more degradation of the image.
- bg: The background color of the image.

Example

```
x <- c(1.1,2,3.5,3.9,4.2)
y <- c(2,2.2,-1.3,0,4)
jpeg("my_chart.jpeg",width = 600,height=600)
plot(x, y, main = "Plot Example",
     xlab = "X Values", ylab = "Y Values", pch =17 , col = "green",type = "b",
     lty = 3,cex=2,lwd=2,xlim = c(-2,5.0),ylim= c(-2,5),las=2,bty = "o")
dev.off()
```


Note:- R supports direct writing to .jpeg, .bmp, .png, .pdf and .tiff files using functions of the same names.

3. **Writing R objects to file**

- If you need to read or write other kinds of R objects, such as lists or arrays, you’ll need the dput and dget commands, which can handle objects in a more ad hoc style
- The dput() and dget() functions in R are used to serialize and deserialize R objects. Serialization is the process of converting an R object into a text representation that can be stored or transmitted.
- Deserialization is the process of converting a serialized text representation of an R object back into an R object.

Example

```
list1 <- list(24, 29, 32, 34)
dput(list1,file="D:/ICCA/R_Program/egfile1.txt")
fileread <- dget("D:/ICCA/R_Program/egfile1.txt")
print(fileread)
```

Note:- The ggsave function can be used to write the most recently plotted ggplot2 graphic to file and performs the device open/close action in one line.

Decision Making Statements or Control statements

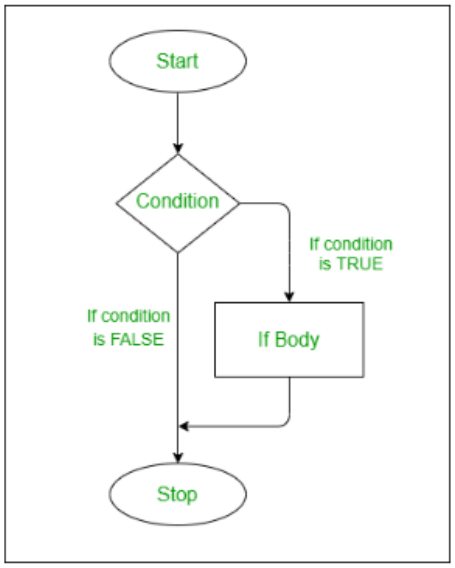
Conditional Statements

1. IF Statement (Stand-Alone Statement)

- It is one of the control statements in R programming that consists of a Boolean expression and a set of statements.
- If the Boolean expression evaluates to TRUE, the set of statements is executed.
- If the Boolean expression evaluates to FALSE, the statements after the end of the If statement are executed.
- The basic syntax for the If statement is given below:

```
if(Boolean_expression)
{
        This block of code will execute if the Boolean expression
        returns TRUE.
}
```

Flow Chart



Example

```
a <- readline(prompt = "Enter the value for a")
a <- as.integer(a)
if( a > 0)
{
  print("The number is positive")
}
```

Output

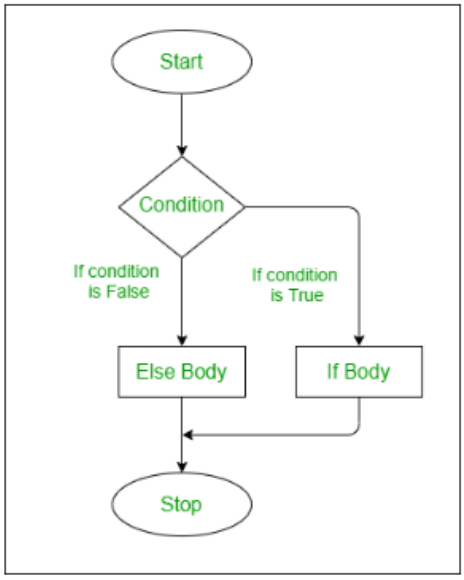
```
Enter the value for a 10
[1] "The number is positive"
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for a -23
[1] "The number is negative"
```

2. If – else statement

- In the If - Else statement, an If statement is followed by an Else statement, which contains a block of code to be executed when the Boolean expression in the If the statement evaluates to FALSE.
- The basic syntax of it is given below:

```
if(Boolean_expression)
{
        This block of code executes if the Boolean expression
        returns TRUE.
} else {
        This block of code executes if the Boolean expression
        returns FALSE.
}
```

Flow Chart



Example

```
a <- readline(prompt = "Enter the value for A")
b <- readline("Enter the value for B")
a <- as.integer(a)
b <- as.integer(b)
if( a > b)
{
  print("A is greater than B ")
}else
{
  print("B is greater than A")
}
```

Output

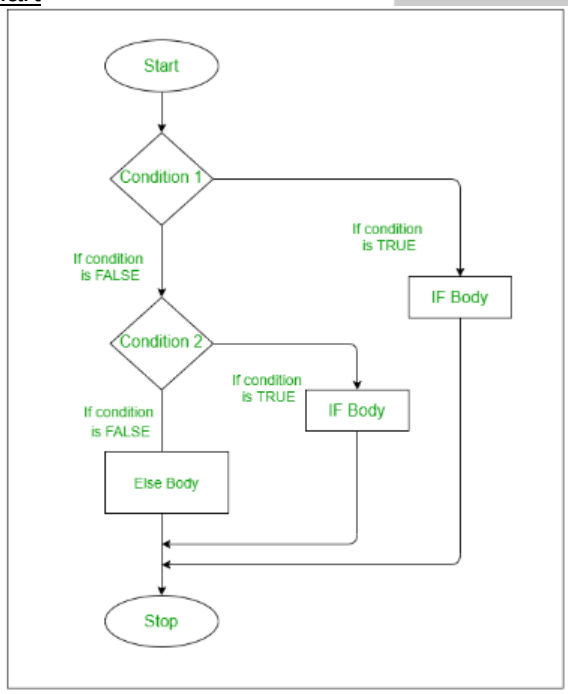
```
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for A 20
Enter the value for B 30
[1] "B is greater than A"
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for A 500
Enter the value for B 300
[1] "A is greater than B "
```

3. if – else – if ladder or else if statement (Stacking statements)

- An Else if statement is included between If and Else statements.
- Multiple Else-If statements can be included after an If statement.
- Once an If a statement or an Else if statement evaluates to TRUE, none of the remaining else if or Else statement will be evaluated.
- The basic syntax of it is given below:

```
if(Boolean_expression1)
{
        This block of code executes if the Boolean expression 1 returns
        TRUE
 } else if(Boolean_expression2)
{
        This block of code executes if the Boolean expression 2 returns
        TRUE
 } else if(Boolean_expression3)
{
        This block of code executes if the Boolean expression returns
        TRUE
} else
{
        This block of code executes if none of the Boolean expression
        returns TRUE
}
```

Flow Chart



Example

```
a <- readline(prompt = "Enter the value for a")
a <- as.integer(a)
if( a > 0)
{
  print("The number is positive")
}else if(a < 0)
{
  print("The number is negative")
}else
{
  print("The number is equal to zero")
}
```

Output

```
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for a 0
[1] "The number is equal to zero"
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for a 10
[1] "The number is positive"
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for a -11
[1] "The number is negative"
```

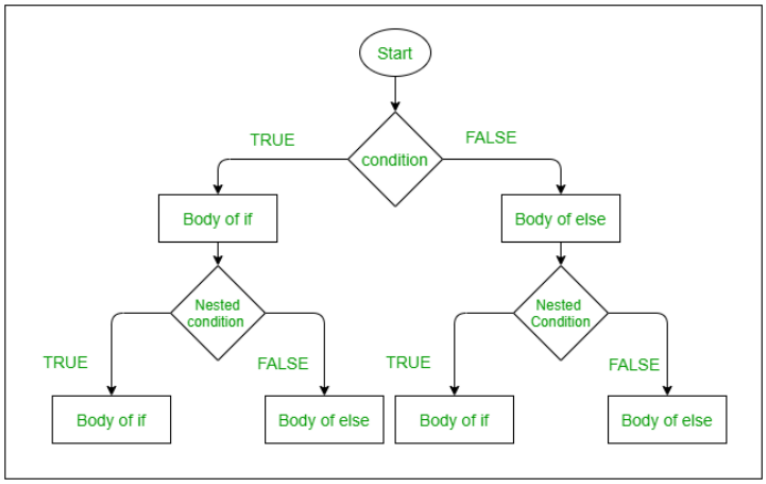
4. Nested if statement (Nesting statements)

- When we have an if-else block as an statement within an if block or optionally within an else block, then it is called as nested if else statement.
- When an if condition is true then following child if condition is validated and if the condition is wrong else statement is executed, this happens within parent if condition.
- If parent if condition is false then else block is executed with also may contain child if else statement.

Syntax:

```
if(parent condition is true)
{
        if( child condition 1 is true)
        {
                execute this statement
        }else
         {
                execute this statement
          }
  } else
{
        if(child condition 2 is true)
        {
                execute this statement
        } else
        {
                execute this statement
        }
 }
```

Flow chart



Example

```
x <- readline("Enter the value for X")
x <- as.integer(x)
if(x>0)
{
  if(x%%2==0)
  {
    cat(x,"is positive even number")
  }
  else
  {
    cat(x,"is a positive odd number")
  }
}else
{
  if(x%%2==0)
  {
    cat(x,"is negative even number")
  }
  else
  {
    cat(x,"is a negative odd number")
  }
}
```


Output

```
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for X 20
20 is positive even number
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for X 25
25 is a positive odd number
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for X -32
-32 is negative even number
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for X -43
-43 is a negative odd number
```

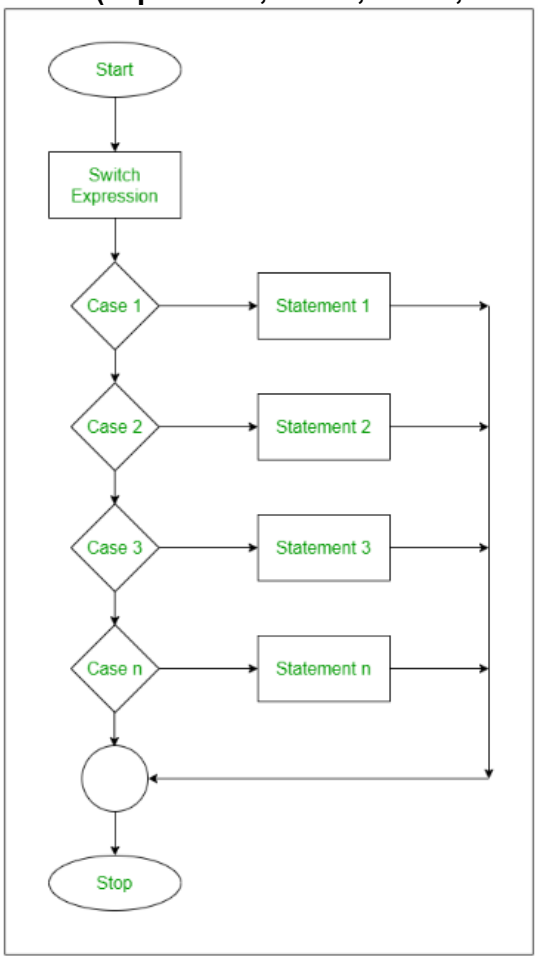
Switch Statement

- In this switch function expression is matched to list of cases.
- If a match is found then it prints that case's value.

- No default case is available here.
- If no case is matched it outputs NULL as shown in example.

Syntax:

switch (expression, case1, case2, case3,…,case n)



number as arguement:
no matter what u do it is always treated as index, u cannot use case names if u take number as arguement,
even if u take number as cases within the " " they wont work.

* u cannot have default case if u take number input

if u want to take numbers as the case names , first u have to convert it to: as.character(input)

```
x <- 2
result <- switch(as.character(x),
                 "1" = "One",
                 "2" = "Two",
                 "3" = "Three",
                 "Unknown number"  # Default case
)
print(result)  # Outputs: "Two"
```

Example:- Expressions in terms of the integer value

```
num1 <- readline("Enter the value for num1")
num1 <- as.integer(num1)
num2 <- readline("Enter the value for num2 ")
num2 <- as.integer(num2)
cat( "1:ADDITION \n2:SUBTRATION \n3: MULTIPLICATION")
res <- readline("Enter your choice ")
res <- as.integer(res)
switch(res,
       cat("The addition of",num1,"and",num2,"is",num1+num2),
       cat("The subtraction of",num1,"and",num2,"is",num1-num2),
       cat("The product of",num1,"and",num2,"is",num1*num2)
       )
```

Output

```
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for num1 86
Enter the value for num2 24
1:ADDITION
2:SUBTRATION
3: MULTIPLICATION
Enter your choice 1
The addition of 86 and 24 is 110
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for num1 96
Enter the value for num2 42
1:ADDITION
2:SUBTRATION
3: MULTIPLICATION
Enter your choice 2
The subtraction of 96 and 42 is 54
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for num1 40
Enter the value for num2 2
1:ADDITION
2:SUBTRATION
3: MULTIPLICATION
Enter your choice 3
The product of 40 and 2 is 80
```

Example:- Expression in terms of string value

```
num1 <- readline("Enter the value for num1")
num1 <- as.integer(num1)
num2 <- readline("Enter the value for num2 ")
num2 <- as.integer(num2)
cat( "A:ADDITION \nS:SUBTRACTION \nM:MULTIPLICATION")
res <- readline("Enter your choice ")
res <- toupper(res)
switch(res,
    "A" = cat("The addition of",num1,"and",num2,"is",num1+num2),
    "S" = cat("The subtraction of",num1,"and",num2,"is",num1-num2),
    "M" = cat("The product of",num1,"and",num2,"is",num1*num2)
)
```

Output

```
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for num1 100
Enter the value for num2 50
A:ADDITION
S:SUBTRACTION
M:MULTIPLICATION
Enter your choice a
The addition of 100 and 50 is 150
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for num1 100
Enter the value for num2 50
A:ADDITION
S:SUBTRACTION
M:MULTIPLICATION
Enter your choice S
The subtraction of 100 and 50 is 50
> source("D:/ICCA/R_Program/contolstatements.R")
Enter the value for num1 100
Enter the value for num2 2
A:ADDITION
S:SUBTRACTION
M:MULTIPLICATION
Enter your choice m
The product of 100 and 2 is 200
```

Loops

- Loops are used to repeat the execution of a block of code.
- Loops help you to save time, avoid repeatable blocks of code, and write cleaner code.
- In R, there are three types of loops:
  - while loops
  - for loops
  - repeat loops

1. While Loop

- while loops are used when you don't know the exact number of times a block of code is to be repeated.
- The basic syntax of while loop in R is:

```
while (test_expression)
{
        # block of code
}
```

- Here, the test_expression is first evaluated.
- If the result is TRUE, then the block of code inside the while loop gets executed.
- Once the execution is completed, the test_expression is evaluated again and the same process is repeated until the test_expression evaluates to FALSE.
- The while loop will terminate when the boolean expression returns FALSE.

Example:- Calculating the sum of first 10 natural numbers.

```
num <- 1
sum <- 0
while(num <=10 )
{
  sum = sum+num
  num <- num+1
}
cat("Sum of first 10 natural numbers is",num)
```

Output

```
> source("D:/ICCA/R_Program/contolstatements.R")
Sum of first 10 natural numbers is 11
```


2.For Loop

- A for loop is used to iterate over a list, vector or any other object of elements.
- The syntax of for loop is:

```
for (value in sequence)
{
  # block of code
}
```

- Here, sequence is an object of elements and value takes in each of those elements.

Example

```
num <- as.integer(readline("Enter the number"))
for (i in 1:10)
{
  cat(num,"*",i,"=",(num*i),"\n")
}
```

Output

```
Enter the number 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

3. Repeat Loop

- You can use the repeat loop in R to execute a block of code multiple times.
- However, the repeat loop does not have any condition to terminate the loop.
- You need to put an exit condition implicitly with a break statement inside the loop.
- The syntax of repeat loop is:

```
repeat
{
    # statements
    if(stop_condition) {
       break
    }
 }
```

- Here, we have used the repeat keyword to create a repeat loop.
- It is different from the for and while loop because it does not use a predefined condition to exit from the loop.

Example

```
count <- 0
repeat
{
  print("Welcome to Interface college of computer applications")
  count <- count+1
  if (count == 5)
  {
    break
  }
}
```


Output

```
> source("D:/ICCA/R_Program/contolstatements.R")
[1] "Welcome to Interface college of computer applications"
[1] "Welcome to Interface college of computer applications"
[1] "Welcome to Interface college of computer applications"
[1] "Welcome to Interface college of computer applications"
[1] "Welcome to Interface college of computer applications"
```

Jump statements

- We use the R break and next statements to alter the flow of a program. These are also known as jump statements in programming:
  - break - terminate a looping statement
  - next - skips an iteration of the loop

R break Statement

- You can use a break statement inside a loop (for, while, repeat) to terminate the execution of the loop. This will stop any further iterations.
- The syntax of the break statement is:

```
if (test_expression)
{
    break
}
```

- The break statement is often used inside a conditional (if...else) statement in a loop. If the condition inside the test_expression returns True, then the break statement is executed.

Example

```
num <- as.integer(readline("Enter the number"))
for (i in 1:10)
{
  if(i == 8)
    break
  cat(num,"*",i,"=",(num*i),"\n")
 }
```

Output

```
Enter the number 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
```

R next Statement

- In R, the next statement skips the current iteration of the loop and starts the loop from the next iteration.
- The syntax of the next statement is:

```
if (test_condition)
{
    next
}
```

- If the program encounters the next statement, any further execution of code from the current iteration is skipped, and the next iteration begins.

Example

```
num <- as.integer(readline("Enter the number"))
for (i in 1:10)
{
  if(i == 8)
    next
  cat(num,"*",i,"=",(num*i),"\n")
}
```

Output

```
Enter the number 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 9 = 45
5 * 10 = 50
```

Exception handling

- Error handling in R is an important aspect of writing robust and reliable code.
- Errors can occur for various reasons, such as invalid input data, unexpected conditions, or issues with external resources.
- To handle errors effectively, R provides several mechanisms and functions.
    1. Using formal notifications
    2. Using try and trycatch functions.

1. Using formal notifucations
    a. Stop() function
    - The stop() function in R is used to generate an error and terminate the execution of the program.
    - It takes a single argument, which is the error message that you want to display.

Example

```
error_test <- function(x){
  if(x<=0){
    stop("'x' is less than or equal to 0... TERMINATE")
  }
  return(5/x)
}
result <- error_test(0)
print(result)
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
Error in error_test(0) : 'x' is less than or equal to 0... TERMINATE
```

    b. Warning() function
    - The warning() function in R is used to generate a warning message but does not terminate the execution of the program.
    - It takes a single argument, which is the warning message that you want to display.

Example:-

```
warn_test <- function(x){
  if(x<=0){
    warning("'x' is less than or equal to 0 but setting it to 1 and
continuing")
    x <- 1
  }
  return(5/x)
}
result <- warn_test(0)
print(result)
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
[1] 5
Warning message:
In warn_test(0) : 'x' is less than or equal to 0 but setting it to 1 and
continuing
```

2. Using try and tryCatch
    a. try()

- The try() function in R is used to evaluate an expression and trap any errors that occur during the evaluation. If an error occurs, the value of the try() function is an object of class try-error.
- This object contains the error message and the error condition.
- The syntax of the try() function is:

  try(expr, silent = FALSE, ...)

Where,

- expr is the expression that you want to try to evaluate.
- silent is a logical value that determines whether or not the error message is printed to the console.

Example 1:- Without setting silent to true.

```
test <- function(x) {
  res <- log(x)
  print(res)
}
try(test(a))
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
Error in log(x) : non-numeric argument to mathematical function
```

Example 2:- Using silent option to ignore error message

```
test <- function(x) {
  res <- log(x)
  print(res)
}
try(test(a),silent=TRUE)
print("Error ignored")
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
[1] "Error ignored"
```

- The try() function can be used to handle errors in a variety of ways. For example, you can use it to:
  - Print the error message to the console.
  - Log the error message to a file.
  - Take some other action, such as retrying the expression or skipping the rest of the code.

Note:- Setting silent=TRUE only suppresses error messages, not warnings. To supress warning use supressWarning() function.

Example

```
test <- function(x) {
  res <- log(x)
  print(res)
}
suppressWarnings(try(test(-3),silent=TRUE))
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
[1] NaN
```

b. tryCatch()

- The tryCatch() function in R is a powerful tool that can be used to handle errors gracefully. It takes three arguments:
  - expr: The expression that you want to try to evaluate.

- error: A handler function that will be called if an error occurs.
- warning: An optional handler function that will be called if a warning occurs.

- The tryCatch() function will evaluate the expression and return its value if there are no errors or warnings.
- If an error occurs, the error handler function will be called.
- If a warning occurs, the warning handler function will be called (if it is specified).

Example

```
log_eg <- function(x)
{
  tryCatch(
    {
      result <- log(x)
      print(result)
    },
    error = function(err){
      message("An error occured")
      print(err)
    },
    warning = function(w){
      message("A warning occured")
     print(w)
    }
  )
}
log_eg()
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
An error occured
```

Timing and visibility

- Progress and Timing R is often used for lengthy numeric exercises, such as simulation or random variate generation.
- For these complex, time-consuming operations, it's often useful to keep track of progress or see how long a certain task took to complete.

1. Textual Progress Bars:
   - Progress bars are used to show the progress of a long-running operation.
   - This can be useful for a variety of tasks, such as downloading a large file, training a machine learning model, or processing a large data set
   - Progress bars can help users to stay informed about the progress of an operation and to manage their expectations.
   - They can also be used to identify and troubleshoot problems, if any.
   - Progress bars can be implemented in a variety of ways. One common approach is to use a text-based progress bar.
   - This type of progress bar is displayed in the console and is typically updated periodically to show the progress of the operation.

Example

```
prog_test <- function(n){
  result <- 0
  progbar <- txtProgressBar(min=0,max=n,style=3,char="--")
  for(i in 1:n){
    result <- result +i
    Sys.sleep(0.5)
    setTxtProgressBar(progbar,value=i)
  }
  close(progbar)
  return(result)
}
print(prog_test(5))
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
  |-------------------------------------------------------------------------------------------------------------| 100%
[1] 15
```

2. Measuring Completion Time
   - If you want to know how long a computation takes to complete, you can use the Sys.time command.
   - This command outputs an object that details current date and time information based on your system.

   Example

```
a <- Sys.time()
time_calculation <- function(n){
  result=0
  for(i in 1:n){
    result <- result + i
    Sys.sleep(0.1)
  }
    return(result)
}
print(time_calculation(5))
b <- Sys.time()
print(b-a)
```

Output

```
> source("D:/ICCA/R_Program/exception_handling.R")
[1] 15
Time difference of 0.8990359 secs
```


Note:- If you need more detailed timing reports, there are more sophisticated tools. For example, you can use proc.time() to receive not just the total elapsed “wall clock” time but also computer-related CPU timings.

Masking

- If there are two objects with the same name in different environments, the object in the current environment will be used. This is known as masking.
- For example, say you define a function with the same name as a function in an R package that you have already loaded.
- R responds by masking one of the objects that is, one object or function will take precedence over the other and assume the object or function name, while the masked function must be called with an additional command.
- This protects objects from overwriting or blocking one another.

1. **Function and Object Distinction**
   - When two functions or objects in different environments have the same name, the object that comes earlier in the search path will mask the later one.
   - That is, when the object is sought, R will use the object or function it finds first, and you'll need extra code to access the other, masked version.
   - Here's how sum works normally, adding up all the elements in the vector eg:

```
> eg <- c(2,4,6,8)
> sum(eg)
[1] 20
> |
```

   - Example of masking, you'll define a function with the same name as a function in the base package: sum.

```
sum <- function(x){
  result <- 0
  for(i in 1:length(x)){
    result <- result + x[i]^2
   } return(result)
}
```

   - This version of sum takes in a vector x and uses a for loop to square each element before summing them and returning the result.
   - This can be imported into the R console without any problem, but clearly, it doesn't offer the same functionality as the (original) built-in version of sum.

```
> eg <- c(2,4,6,8)
> sum(eg)
[1] 120
```

   - This happens because the user-defined function is stored in the global environment (.GlobalEnv), which always comes first in the search path.
   - R's built-in function is part of the base package, which comes at the end of the search path. In this case, the user-defined function is masking the original.
   - Now, if you want R to run the base version of sum, you have to include the name of its package in the call, with a double colon

```
> eg <- c(2,4,6,8)
> sum(eg)
[1] 120
> base::sum(eg)
[1] 20
```


Note:-

1. The search() function in R is used to search for objects in the R search path. The search path is a list of environments that R will search when you refer to an object
2. The attach() function in R is used to attach an environment to the search path. This means that the objects in the environment will be available to the current environment without having to specify the environment name.
3. The detach() function in R is used to detach an environment from the search path. This means that the objects in the environment will no longer be available to the current environment without having to specify the environment name.
4. The rm() function in R is used to remove objects from the current environment. This can be useful for freeing up memory or for removing objects that are no longer needed.

Unit :- 3
Statistics and Probability

https://chatgpt.com/share/672f4525-c450-8010-a06c-b0db45aabf45

Elementary Statistics

- Statistics is the practice of turning data into information to identify trends and understand features of populations.

Important terms in Statistics

1. **Raw Data**
   - Raw data is the records or observations that make up a sample.
   - Depending on the nature of the intended analysis, these data could be stored in a specialized R object, often a data frame.
2. **Variables**
   - **A** variable is a characteristic of an individual in a population, the value of which can differ between entities within that population.
     Ex:- Name, age, gender, height, weight etc…
   - Variables can take on a number of forms, determined by the nature of the values they may take.
     1. Numerical data
     2. Categorical data

i. Numerical data: - A numeric variable is one whose observations are naturally recorded as number.

Or

Numeric variable represents measurable quantities that can take on numeric values There are two types of numeric variables: continuous and discrete.

a. **Continuous variable**:- Continues variable can take on any value with in a specific range of value. They are often measured with decimal values.
   Example :- Height, weight, temperature etc..
b. **Discrete variable: -** A discrete variable can only take a specific, separate values, often integers.
   Example:- Number of children in a family, number of students in a class, Number of shirts you have etc.…

Continuous values can take any value within a given range. They are usually measurements and can include fractions and decimals.
Continuous data can be infinitely divided into smaller parts

Discrete values are distinct and separate; they can only take specific values and are usually counts.
Discrete data cannot be subdivided into smaller units meaningfully, so fractional values often do not make sense here.

ii. Categorial Variable

- A categorical variable is a variable that can take on a limited number of non-numerical values. These values represent different categories or groups. Categorical variables can be either nominal or ordinal.
  a. **Nominal categorical:** Nominal categorical variables have categories that have no inherent order.
     Example:- "eye color" has the categories "blue," "green," "brown," and "hazel."
     Marital stautus has the categories “Married”, “Single”,and “Divorced”
  b. **Ordinal categorical**: - Ordinal categorical variables have categories that have an inherent order.
     Example, the categorical variable:- "level of education" has the categories "high school diploma," "associate's degree," "bachelor's degree," and "master's degree."
     Shirt size :- S,M,L,XL etc

Nominal values categorize data into distinct groups, but these categories have no inherent order or ranking.Categories are qualitative (non-numerical) and unordered.

Ordinal values also categorize data, but with a clear, meaningful order or ranking among the categories.Categories are qualitative but ordered.

3. Univariate and Multivariate data

- **Univariate data:-** When discussing or analyzing data related to only one dimension, you’re dealing with univariate data.

Univariate data involves only one variable. It's the simplest form of data analysis, focused on understanding the distribution, central tendency, and spread of a single variable.
Since there's only one variable, no relationships or comparisons between variables are analyzed.
Examples:
Temperature in a city over a month (only one variable: temperature).

Multivariate data involves two or more variables, which allows for the analysis of relationships between the variables.
Multivariate data analysis helps to examine how multiple variables interact with each other.
Example
Height and weight of individuals

Example:- The height of a group of people is a univariate data set. Exam scores of a group of students (a single variable representing scores).

- **Multivariate data:-** When it's necessary to consider data with respect to variables that exist in more than one dimension, your data are considered multivariate, Example: -Housing Price Prediction, The dataset might include variables like: House size (in square feet), Number of bedrooms, Number of bathrooms.

4. Population :- In statistics, a population is the entire set of individuals, objects, or events from which a sample is drawn for a study. It can be a group of people, a set of items, or a collection of events.

The entire set of individuals, items, or data that you are interested in studying or analyzing.If you are studying the heights of all adults in a city, the population is all adults in that city.

5. Sample: - Sampling is the process of selecting a subset of the population, known as a sample, to collect data from. The sample should be representative of the population, so that the results of the study can be generalized to the entire population.

A subset of the population selected for analysis, used to make inferences about the entire population.
Eg: If you survey 500 adults from the city to study their heights, those 500 people are the sample.

Example:- Calculating average height of all adults in India
Population: All adults in the India
Sample: A group of 100 randomly selected adults in the India.

- We can use the sample to estimate the population mean height of all adults in the India. To do this, we would calculate the sample mean height of the 100 adults in the sample. The sample mean height would be an estimate of the population mean height of all adults in the India.

6. Statistic:- Statistics is a number that describes a sample of data. It is calculated from a sample, and it is an estimate of the corresponding parameter.

A numerical value calculated from a sample that describes or summarizes some aspect of it.
Example: The average height of the 500 adults surveyed is a statistic.

A numerical value that describes a characteristic of the entire population (often unknown and estimated using a sample statistic).
Example: The average height of all adults in the city is a parameter.

7. Parameter:- Parameter is a number that describes a population of data. It is calculated from the entire population, and it is the true value of the characteristic being measured.

8. Outlier: - Outliers An outlier is an observation that does not appear to "fit" with the rest of the data. It is a noticeably extreme value when compared with the bulk of the data, in other words, an anomaly.

A data point significantly different from other observations in a dataset, which may indicate variability or errors.
Example: If most adults in the city are between 150–190 cm tall, a person with a height of 250 cm is an outlier.

f. Variables
Definition: Characteristics or properties of individuals or items in a dataset that can take on different values.
Types:
Quantitative Variables: Numerical values (e.g., age, height).
Qualitative Variables: Categories or labels (e.g., gender, color).
Example: In a dataset about people, variables could include age, height, weight, and occupation.

Summary statistics

- Now that you've learned the basic terminology, you're ready to calculate some statistics with R. In this section, you'll look at the most common types of statistics used to summarize the different types of variables.

1. **Centrality :- Mean, Median, Mode**

- Measures of centrality are commonly used to explain large collections of data by describing where numeric observations are centered.

a. Mean: -

- The arithmetic mean, or simply the mean, is the most common measure of centrality, and it is used as a central "balance point" of a collection of observations.
- The mean is calculated by adding up the values of all the numbers in a set and dividing by the number of numbers in the set.
- For a set of n numeric measurements labeled x = {x1, x2,. . . , xn }, you find the sample mean $\bar{x}$ as follows:

$$\bar{x} = \frac{(x_1 + x_2 + \ldots + x_n)}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- So, for example, if you observe the data 2,4.4,3,3,2,2.2,2,4, the mean is calculated like this:
  (2 + 4.4 + 3 + 3 + 2 + 2.2 + 2 + 4)/ 8 = 2.825
- To calculate the mean in R, you can use the mean() function. The mean() function takes a vector as input and returns the mean as a numeric value.

Example: -

```
> chickwts$weight
 [1] 179 160 136 227 217 168 108 124 143 140 309 229 181 141 260 203 148 169 213 257 244 271 243 230 248
[26] 327 329 250 193 271 316 267 199 171 158 248 423 340 392 339 341 226 320 295 334 322 297 318 325 257
[51] 303 315 380 153 263 242 206 344 258 368 390 379 260 404 318 352 359 216 222 283 332
> mean(chickwts$weight)
[1] 261.3099
```

b. Median: -

- The median is the middle value in a set of data when the data is sorted from least to greatest.
- If the data set has an odd number of observations, the median is the middle value.
- If the data set has an even number of observations, the median is the average of the two middle values.
- Using the notation for n measurements labeled
  x = {x1, x2,. . . , xn },
- You find the sample median m¯ x as follows
  - Sort the observations from smallest to largest to give the "order statistics" $x_i^{(1)}, x_j^{(2)}, \ldots, x_k^{(n)}$, where $x_i^{(t)}$ denotes the tth smallest observation, regardless of observation number $i, j, k, \ldots$.
  - Then, do the following:

$$\tilde{m}_x = \begin{cases} x_i^{\left(\frac{n+1}{2}\right)}, & \text{if } n \text{ is odd} \\ \left(x_i^{\left(\frac{n}{2}\right)} + x_j^{\left(\frac{n}{2}+1\right)}\right)/2, & \text{if } n \text{ is even} \end{cases}$$

- So, for example, if you observe the data 2,4.4,3,3,2,2.2,2,4, the median is calculated like this:
  - Sorting them from smallest to largest yields 2, 2, 2, 2.2, 3, 3, 4, 4.4. With n = 8 observations, you have n/2 = 4.
  - The median is
    $\left(x_i^{(4)} + x_j^{(5)}\right)/2 = (2.2 + 3)/2 = 2.6$

- To find the median in R, you can use the median() function. The median() function takes a vector as input and returns the median as a numeric value.

Example

```
> head(iris$Sepal.Length,10)
 [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
> median(head(iris$Sepal.Length))
[1] 4.95
```

c. Mode: -

- In statistics, the "mode" is a measure of central tendency that represents the value or values that occur most frequently in a dataset. In a given dataset, the mode is the value that has the highest frequency or count.
- To find the statistical mode (most frequently occurring value) in a dataset in R, you can use the table() function to create a frequency table and then extract the mode from the table.

Example:-

```
vec1 <- c(1,2,3,1,2,8,5,6,4,1,2,3,5,1,1,1)
a <- table(vec1)
print(a)
mode_num <- a[a==max(a)]
print(mode_num)
```

Output: -

```
> source("D:/ICCA/R_Program/stats.R")
vec1
1 2 3 4 5 6 8
6 3 2 1 2 1 1
1
6
```

2. Counts, Percentages, and Proportions

- In this section, you'll look at the summary of data that aren't necessarily numeric.
- It makes little sense, for example, to ask R to compute the mean of a categorical variable, but it is sometimes useful to count the number of observations that fall within each category these counts or frequencies represent the most elementary summary statistic of categorical data.

a. Count:-

- Count is the number of occurrences of a specific value or category in a dataset. It represents the absolute quantity of a particular element.
- You can calculate the count of a specific value in a vector or dataset using functions like table() and then access the count for a specific value.

Example: -

```
read_file <- read.table("D:/ICCA/R_Program/Book1.csv",header = TRUE, sep=",")
print(table(read_file$Gender))
```

Output

```
> source("D:/ICCA/R_Program/stats.R")

Female   Male
     8     14
```

b. Proportions:-

- Proportion represents the relative size of a specific category or value in relation to the total. It is typically expressed as a fraction or decimal.
- To calculate the proportion of a specific value in a dataset, you can divide the count of that value by the total count.

Example: -

```
read_file <- read.table("D:/ICCA/R_Program/Book1.csv",header = TRUE, sep=",")
print(table(read_file$Gender)/nrow(read_file))
print(table(read_file$Gender[read_file$Gender=="Female"])/nrow(read_file))
print(prop.table(table(read_file$Gender)))
```

Output: -

```
> source("D:/ICCA/R_Program/stats.R")

   Female      Male
0.3636364 0.6363636

   Female
0.3636364

   Female      Male
0.3636364 0.6363636
```

c. Percentage

- Percentage is a way to express the proportion as a relative value scaled to 100. It is often used to describe the proportion in terms of a percentage of the total.
- To calculate the percentage of a specific value in a dataset, you can multiply the proportion by 100.

Example: -

```
read_file <- read.table("D:/ICCA/R_Program/Book1.csv",header = TRUE, sep=",")
print(table(read_file$Gender)/nrow(read_file)*100)
print(table(read_file$Gender[read_file$Gender=="Female"])/nrow(read_file)*100)
print(prop.table(table(read_file$Gender))*100)
```

Output: -

```
> source("D:/ICCA/R_Program/stats.R")

  Female     Male
36.36364 63.63636

  Female
36.36364

  Female     Male
36.36364 63.63636
```

3. Quantiles, Percentiles, and the Five-Number Summary

Quantiles are values that divide a dataset into equal-sized intervals, where each interval contains the same proportion of the data points

a. Quantiles

- A quantile is a point in a distribution that divides the distribution into a specified number of equal parts. For example, the quartiles divide the distribution into four equal parts, the deciles divide the distribution into ten equal parts.
- The median is itself a quantile—it gives you a value below which half of the measurements lie—it's the 0.5th quantile.
- Alternatively, quantiles can be expressed as a percentile—this is identical but on a "percent scale" of 0 to 100. In other words, the pth quantile is equivalent to the 100 × pth percentile. The median, therefore, is the 50th percentile.
- There are a number of different algorithms that can be used to compute quantiles and percentiles. They all work by sorting the observations from smallest to largest and using some form of weighted average to find the numeric value that corresponds to p.
- Obtaining quantiles and percentiles in R is done with the quantile function.

Example: -

```
vec1 <- c(1,2,3,1,2,8,5,6,4,1,2,3,5,1,1,1)
print(quantile(vec1,prob=0.5))
```

Output

```
> source("D:/ICCA/R_Program/stats.R")
50%
  2
```

- As you can see, quantile takes the data vector of interest as its first argument, followed by a numeric value supplied to prob, giving the quantile of interest.
- In fact, prob can take a numeric vector of quantile values.
- This is convenient when multiple quantiles are desired.

Example: -

```
vec1 <- c(1,2,3,1,2,8,5,6,4,1,2,3,5,1,1,1)
print(quantile(vec1,prob=c(0,0.25,0.5,0.75,1)))
```

Output

```
> source("D:/ICCA/R_Program/stats.R")
  0%  25%  50%  75% 100%
1.00 1.00 2.00 4.25 8.00
```

The five-number summary is a concise way of describing the distribution of a dataset. It includes five specific statistics:

Minimum: The smallest data point.
First Quartile (Q1): The value below which 25% of the data falls (25th percentile).
Median (Q2): The middle value that divides the dataset into two equal halves (50th percentile).
Third Quartile (Q3): The value below which 75% of the data falls (75th percentile).
Maximum: The largest data point.

b. Five – number summary

- The five-number summary is a statistical method for describing the distribution of a data set. It consists of the following five values:
  - Minimum: The smallest value in the data set.
  - First quartile (Q1): The middle value between the minimum and the median.
  - Median (Q2): The middle value in the data set, when the data is sorted in ascending order.
  - Third quartile (Q3): The middle value between the median and the maximum.
  - Maximum: The largest value in the data set.
- To calculate the five-number summary of a data set, you can use summary()

Example: -

```
vec1 <- c(1,2,3,1,2,8,5,6,4,1,2,3,5,1,1,1)
print(summary(vec1))
```

Output

```
> source("D:/ICCA/R_Program/stats.R")
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.000   2.000   2.875   4.250   8.000
```

4. Spread: Variance, Standard Deviation, and the Interquartile Range

- A measure of how dispersed the values in a data set are. The range of values in a data set and how far apart the values are from each other.

a. Sample Variance: -

Sample variance is a measure of how spread out a set of values is from their mean. In statistics, it quantifies the average of the squared differences between each data point and the mean of the data set. In R, you can calculate sample variance using the var() function.

- The sample variance measures the degree of the spread of numeric observations around their arithmetic mean.
- It is calculated as the average of the squared deviations from the mean.
- A high variance indicates that the data points are spread out over a wider range, while a low variance indicates that the data points are clustered closer to the mean.
- For a set of n numeric measurements labeled x = {x1, x2,. . . , xn }, the sample variance s 2 x is given by the following, where $\bar{x}$ is the sample mean described in Equation

$$s_x^2 = \frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \ldots + (x_n - \bar{x})^2}{n-1} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

  - So, for example, if you observe the data 2,4.4,3,3,2,2.2,2,4, the variance is calculated like this:
    - First, find the mean :- (2 + 4.4 + 3 + 3 + 2 + 2.2 + 2 + 4)/ 8 = 2.825.
    - To calculate the Variance, compute the difference of each from the mean, square it and find then find the average once again.

$$\frac{(2-2.825)^2+(4.4-2.825)^2+\ldots+(4-2.825)^2}{7}$$
$$=\frac{(-0.825)^2+(1.575)^2+\ldots+(1.175)^2}{7}$$
$$=\frac{6.355}{7}=0.908$$

- The direct R command for computing these measures of spread is var(variance)

Example: -

```
vec1 <- c(2,4,6,8,10)
print(var(vec1))
```

Output: -

```
> source("D:/ICCA/R_Program/stats.R")
[1] 10
```

b. Standard deviation

The standard deviation is a measure of the spread or dispersion of a set of values relative to their mean. It tells us, on average, how far each value in the data set is from the mean. Unlike variance, standard deviation is in the same units as the data itself, making it easier to interpret.

- Standard deviation is a measure of how spread out a set of data is.
- It is calculated as the square root of the variance, which is the average of the squared deviations from the mean.
- With the same notation for a sample of n observations, the sample standard deviation s is found by taking the square root of Equation.

$$s_x=\sqrt{s^2}=\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2}$$

- For example, based on the sample variance calculated earlier, the standard deviation of the eight hypothetical observations is as follows (to three decimal places):

$$\sqrt{0.908}=0.953$$

- Thus, a rough way to interpret this is that 0.953 represents the average distance of each observation from the mean.
- The direct R command for computing these measures of spread is sd(Standard deviation)

Example: -

```
vec1 <- c(2,4,6,8,10)
print(var(vec1))
print(sd(vec1))
```


Output: -

```
> source("D:/ICCA/R_Program/stats.R")
[1] 10
[1] 3.162278
```

c. Interquartile range

The interquartile range (IQR) is a measure of statistical dispersion, representing the spread of the middle 50% of a data set. It tells us how spread out the central values are, helping to understand the variability of data around the median without being affected by extreme values (outliers).

- The IQR measures the width of the “middle 50 percent” of the data, that is, the range of values that lie within a 25 percent quartile on either side of the median.
- IQR can be used to identify outliers in a data set.
- The IQR is computed as the difference between the upper and lower quartiles of your data. Formally, where Qx denotes the quantile function, the IQR is given as

$$\mathrm{IQR}_x=Q_x(0.75)-Q_x(0.25)$$

- The direct R commands for computing these measures of spread is IQR(interquartile)

Example: -

```
vec1 <- c(1,2,3,1,2,8,5,6,4,1,2,3,5,1,1,1)
print(quantile(vec1,probs=c(0.25,0.75)))
print(IQR(vec1))
```

Output: -

```
> source("D:/ICCA/R_Program/stats.R")
 25%  75%
1.00 4.25
[1] 3.25
```

Covariance is a statistical measure that indicates the direction of the linear relationship between two variables. It tells us whether two variables tend to increase or decrease together (positive covariance) or move in opposite directions (negative covariance).

Positive Covariance: When one variable increases, the other tends to increase as well.
Negative Covariance: When one variable increases, the other tends to decrease.
Zero Covariance: Indicates no relationship between the variables.

5. Covariance and Correlation

- When analyzing data, it's often useful to be able to investigate the relationship between two numeric variables to assess trends.
- For example, you might expect height and weight observations to have a noticeable positive relationship—taller people tend to weigh more.

a. Covariance: -

- The covariance expresses how much two numeric variables "change together" and the nature of that relationship, whether it is positive or negative.
- Suppose for n individuals you have a sample of observations for two variables, labeled x = {x1, x2,. . . , xn } and y = {y1, y2,. . . , yn }, where xi corresponds to yi for i = 1,. . . ,n.
- The sample covariance rx y is computed with the following, where $\bar{x}$ and $\bar{y}$ represent the respective sample means of both sets of observations:

$$r_{xy} = \frac{1}{n-1}\sum_{i-1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- When you get a positive result for rx y , it shows that there is a positive linear relationship—as x increases, y increases.
- When you get a negative result, it shows a negative linear relationship—as x increases, y decreases, and vice versa.
- When rx y = 0, this indicates that there is no linear relationship between the values of x and y.
- To demonstrate, let's use the original eight illustrative observations, which I'll denote here with x = {2,4.4,3,3,2,2.2,2,4}, and the additional eight observations denoted with y = {1,4.4,1,3,2,2.2,2,7}.
- Remember that both x and y have sample means of 2.825. The sample covariance of these two sets of observations is as follows (rounded to three decimal places)

$$\frac{(2-2.825)\times(1-2.285)+\ldots+(4-2.825)\times(7-2.825)}{7}$$
$$=\frac{(-0.825)(-1.825)+\ldots+(1.175)(4.175)}{7}$$
$$=\frac{10.355}{7}=1.479$$

- The figure is a positive number, so this suggests there is a positive relationship based on the observations in x and y.
- In R covariance is implemented using cov( ) function.

Example: -

```
height <- c(5.3,5.7,6,4.6,5.0)
weight <-c(53,57,60,46,50)
print(cov(height,weight))
```

Output: -

```
> source("D:/ICCA/R_Program/stats.R")
[1] 3.07
```

b. Correlation: -

- Correlation allows you to interpret the covariance further by identifying both the direction and the strength of any association.
- There are different types of correlation coefficients, but the most common of these is Pearson's product-moment correlation coefficient, the default implemented by R.
- Pearson's sample correlation coefficient ρ_{xy} is computed by dividing the sample covariance by the product of the standard deviation of each data set.

Correlation measures the strength and direction of the linear relationship between two variables. Unlike covariance, correlation is standardized, meaning it always falls between
–
1
−1 and
1
1, which makes it easier to interpret and compare across different datasets.

$$\rho_{xy} = \frac{r_{xy}}{s_x s_y},$$

- The correlation coefficient is usually represented using the symbol r, and it ranges from -1 to +1.
- A correlation coefficient quite close to 0, but either positive or negative, implies little or no relationship between the two variables.
- A correlation coefficient, P_{xy} = 1, means a positive relationship between the two variables, with increases in one of the variables being associated with increases in the other variable.
- A correlation coefficient, P_{xy} = -1, indicates a negative relationship between two variables, with an increase in one of the variables being associated with a decrease in the other variable.
- In R correlation is implemented using cor() function

Example: -

```
height <- c(5.3,5.7,6,4.6,5.0)
weight <-c(53,57,60,46,50)
print(cov(height,weight))
print(cor(height,weight))
```


Output: -

```
> source("D:/ICCA/R_Program/stats.R")
[1] 3.07
[1] 1
```

Note: -

- The tapply() helps us to compute statistical measures (mean, median, min, max, etc..) or a self-written function operation for each factor variable in a vector.
- It helps us to create a subset of a vector and then apply some functions to each of the subsets.

Example:-

```
read_file <- read.table("D:/ICCA/R_Program/Book1.csv",header = TRUE, sep=",")
a <- tapply(read_file$Gender,read_file$Gender,function(x)length(x)/nrow(read_file))
print(a)
b<- tapply(read_file$Total,read_file$Gender,mean)
print(b)
```

Output: -

```
> source("D:/ICCA/R_Program/stats.R")
   Female      Male
0.3636364 0.6363636
   Female      Male
106.12500  86.28571
```

Basic Data Visualization

Data visualization in R refers to the process of representing data graphically to make it easier to interpret and understand. It involves creating charts, graphs, and plots to communicate insights, patterns, trends, and relationships within the data.

- Data visualization is an important part of a statistical analysis.

Barplots and Pie Charts

- Barplots and pie charts are commonly used to visualize qualitative data by category frequency.

A bar plot (or bar chart) is a graphical representation of data that uses rectangular bars to display the frequency, count, or other measures of different categories. The length (or height) of each bar is proportional to the value it represents.

1. Building a Barplot
   - In R, the barplot() function is used to create bar plots. A bar plot is a graphical representation of the frequencies or counts of categorical data
   - A barplot draws either vertical or horizontal bars, typically separated by white space, to visualize frequencies according to the relevant categories.

Syntax: - **barplot(height, names.arg = NULL, beside = TRUE, horiz = FALSE, col = NULL, main = NULL, xlab = NULL, ylab = NULL)**

Where,

- height: A numeric vector or matrix of values representing the heights of the bars.
- names.arg: A vector of names or labels for each bar.
- beside: Logical. If TRUE, the bars are plotted beside each other; if FALSE, the bars are stacked.
- horiz: Logical. If TRUE, the bars are drawn horizontally.
- col: A vector of colors for the bars.
- main: A title for the plot.
- xlab: Label for the x-axis.
- ylab: Label for the y-axis.

```
# Data for the bar plot
sales <- c(23, 17, 35, 29, 41)
days <- c("Mon", "Tue", "Wed", "Thu", "Fri")

# Creating a bar plot
barplot(sales,
        names.arg = days,
        col = "skyblue",
        main = "Weekly Sales",
        xlab = "Days of the Week",
        ylab = "Number of Products Sold")
```

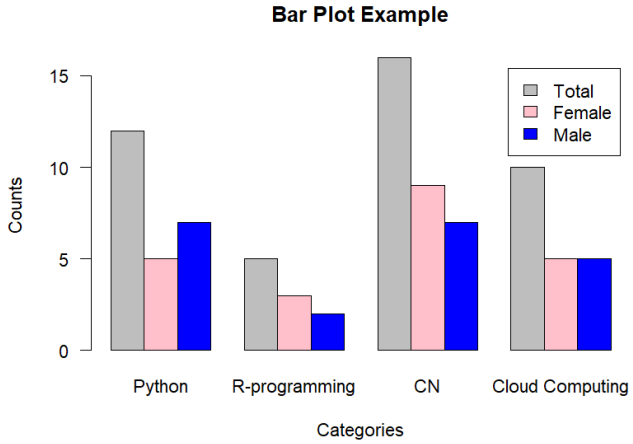
Example:- Vertical Bar plot

```
col_name <- c("Python", "R-programming", "CN", "Cloud Computing")
row_name <- c("Total","Female", "Male")
Total_count <- c(12,5,16,10)
Female_Lecture <- c(5, 3, 9, 5)
male_Lecture <- c(7, 2, 7, 5)

# Create a matrix
mat <- matrix(c(Total_count,Female_Lecture, male_Lecture),
              nrow = 3, ncol = 4, byrow = TRUE,
              dimnames = list(row_name, col_name))

# Print the matrix
print(mat)

# Create a bar plot
barplot(mat, beside = TRUE, names.arg = col_name, col = c("grey","pink", "blue"),
        main = "Bar Plot Example", xlab = "Categories", ylab = "Counts", las = 1,
        legend.text = c("Total","Female","Male"))
```

```
# Sample data
product1 <- c(23, 17, 35, 29, 41)
product2 <- c(19, 24, 32, 28, 37)
sales_data <- rbind(product1, product2)

# Creating a grouped bar plot
barplot(sales_data,
        beside = TRUE, # beside = TRUE makes it grouped
        names.arg = days,
        col = c("blue", "green"),
        main = "Weekly Sales Comparison",
        xlab = "Days",
        ylab = "Number of Products Sold",
        legend = c("Product 1", "Product 2"))
```

Output:-

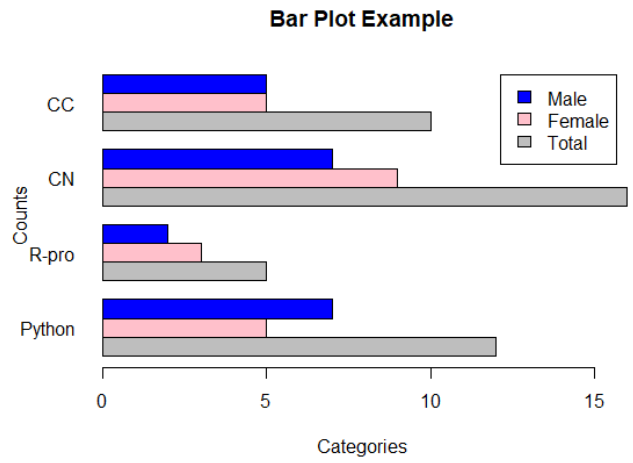
```
> source("D:/ICCA/R_Program/datavisual.R")
       Python R-programming CN Cloud Computing
Total      12             5 16               10
Female      5             3  9                5
Male        7             2  7                5
```



Example:- Horizontal bar plot

```
col_name <- c("Python", "R-pro", "CN", "CC")
row_name <- c("Total","Female", "Male")
Total_count <- c(12,5,16,10)
Female_Lecture <- c(5, 3, 9, 5)
male_Lecture <- c(7, 2, 7, 5)

# Create a matrix
mat <- matrix(c(Total_count,Female_Lecture, male_Lecture),
              nrow = 3, ncol = 4, byrow = TRUE,
              dimnames = list(row_name, col_name))

# Print the matrix
print(mat)

# Create a bar plot
barplot(mat, beside = TRUE, names.arg = col_name, horiz = TRUE,col =
c("grey","pink", "blue"),
        main = "Bar Plot Example", xlab = "Categories", ylab = "Counts", las = 1,
    legend.text = c("Total","Female","Male"))
```


Output: -



Pie Chart

- The pie chart is an alternative option for visualizing frequency based quantities across levels of categorical variables, with appropriately sized “slices” representing the relative counts of each categorical variable.
- In R, the pie() function is used to create a pie chart.

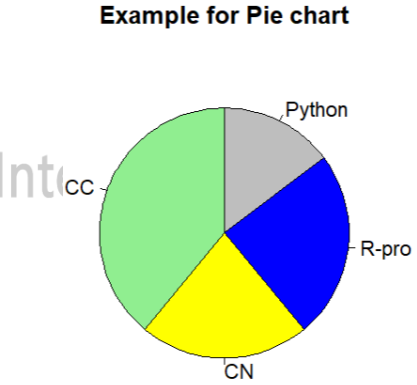
Syntax:- pie(data,clockwise,init.angle,labels,density,angle,col,border,lty,main,…)

Parameter	Description
clockwise	If True, slices are drawn clockwise ortherwise counter-clockwise
init.angle	The starting angle for the slices
labels	The names for the slices
density	The density of shading lines
angle	The slope of shading lines
col	A vector of colors to be used in filling or shading the slices
border	The color to be used for the border
lty	Type of lines used for plotting pie chart
main	An overall title for the plot

Example: -

```
col_name <- c("Python", "R-pro", "CN", "CC")
Total_count <- c(30,50,45,80)
Female_Lecture <- c(5, 3, 9, 5)
pie(Total_count,labels = col_name,clockwise = TRUE,main="Example for Pie chart
",col = c("gray","blue","yellow","lightgreen"))
```

Output




```
# Data for the pie chart
market_share <- c(25, 15, 30, 20, 10)
products <- c("Product A", "Product B", "Product C", "Product D", "Product E")

# Creating the pie chart
pie(market_share,
    labels = products,
    col = c("blue","red","green","violet","yellow"),
    main = "Market Share of Products",clockwise=TRUE)
```

Histogram

- To visualize the distribution of continuous measurements, you can use a histogram.
- A histogram also measures frequencies, but in targeting a numeric-continuous variable, it’s first necessary to “bin” the observed data, meaning to define intervals and then count the number of continuous observations that fall within each one.
- The size of this interval is known as the binwidth.
- Syntax: - **hist(x,breaks,freq,labels,density,angle,col,border,main,xlab,ylab,…)**

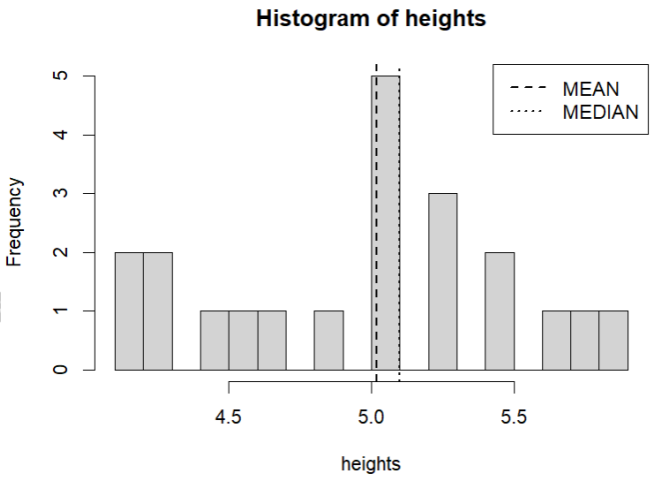
Parameter	Description
x	A vector of values describing the bars which make up the plot
breaks	A number specifying the number of bins for the histogram
freq	If TRUE, hist() gives counts instead of probabilities.
labels	If TRUE, draws labels on top of bars
density	The density of shading lines
angle	The slope of shading lines
col	A vector of colors for the bars
border	The color to be used for the border of the bars
main	An overall title for the plot
xlab	The label for the x axis
ylab	The label for the y axis

```
# Data for the histogram
scores <- c(55, 65, 72, 75, 80, 85, 85, 88, 90, 92, 95, 60, 78,
81, 83, 86, 77, 82, 84, 91,
        93, 94, 69, 73, 74, 66, 58, 59, 62, 68)

# Creating the histogram
hist(scores,
    col = "lightblue", breaks=15,
    main = "Distribution of Test Scores",
    xlab = "Scores",
    ylab = "Frequency")
```

Example

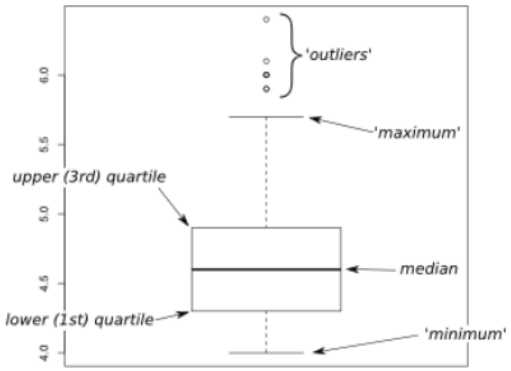
```
heights<- c(5.1,5.3,5.5,5.7,5.1,5.5,5.8,5.1,5.3,5.9,5.3,5.1,5.1,4.1,4.3,4.2,4.5,4.7,4.9,4.6,4.3)
hist(heights,breaks = 15)
abline(v=c(mean(heights),median(heights)),lty=c(2,3),lwd=2)
legend("topright",legend = c("MEAN","MEDIAN"),lty=c(2,3),lwd=2)
```

Output



Box and Whisker Plots

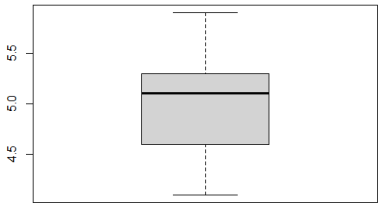
- In R, the boxplot() function is used to create boxplots, also known as box-and-whisker plots.
- Boxplots are useful for visualizing the distribution of a dataset, particularly in terms of central tendency, spread, and the presence of outliers.



Example 1: - Stand-Alone Boxplots

```
heights<- c(5.1,5.3,5.5,5.7,5.1,5.5,5.8,5.1,5.3,5.9,5.3,5.1,5.1,4.1,4.3,4.2,4.5,4.7,4.9,4.6,4.3)

boxplot(heights)
```

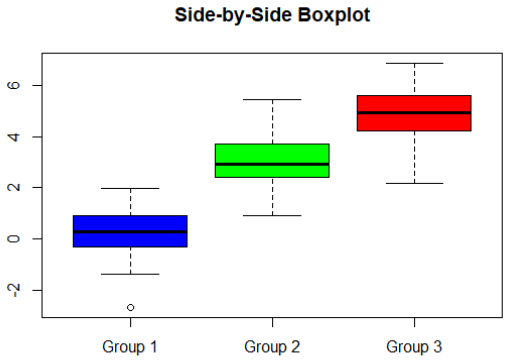
Output: -



Example 2:- Side – By – Side BoxPlot

```
# Sample data
group1 <- rnorm(50, mean = 0, sd = 1)
group2 <- rnorm(50, mean = 3, sd = 1)
group3 <- rnorm(50, mean = 5, sd = 1)
# Create side-by-side boxplot
boxplot(group1,group2,group3, col = c("blue", "green", "red"), names = c("Group 1",
"Group 2", "Group 3"), main = "Side-by-Side Boxplot")
```


Output




```
# Data for three classes
class_A <- c(72, 85, 78, 92, 88, 75, 79, 91, 83)
class_B <- c(87, 95, 67, 85, 89, 73, 81, 84, 90)
class_C <- c(77, 80, 78, 85, 82, 74, 76, 81, 79)

# Creating side-by-side box plots
boxplot(class_A, class_B, class_C,
        names = c("Class A", "Class B", "Class C"),
        main = "Side-by-Side Box Plot of Test Scores by Class",
        ylab = "Scores",
        col = c("lightblue", "lightgreen", "pink"))
```

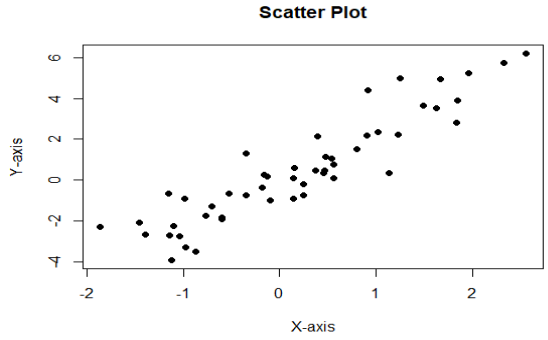
Scatter Plot

- A scatter plot is a graphical representation of two continuous variables, where individual data points are plotted on a two-dimensional graph.
- Each point on the graph represents the values of the two variables.
- One variable is plotted on the horizontal axis (X-axis), and the other variable is plotted on the vertical axis (Y-axis).
- Scatter plots are useful for visualizing the relationship between two variables.
- They can help you identify patterns, trends, clusters, and outliers in the data.
- The position of each point on the scatter plot corresponds to the values of the two variables for a particular observation.
- In R, you can create scatter plots using the plot() function

Example :- Single Plot

```
x <- rnorm(50)
y <- 2 * x + rnorm(50)
plot(x, y, main = "Scatter Plot", xlab = "X-axis", ylab = "Y-axis", pch = 16, col = "black")
```

Output: -




```
# Data for height and weight
height <- c(150, 160, 165, 170, 175, 180, 185, 190)
weight <- c(55, 60, 62, 68, 70, 75, 80, 85)

# Creating the scatter plot
plot(height, weight,
   main = "Scatter Plot of Height vs Weight",
   xlab = "Height (cm)",
   ylab = "Weight (kg)",
   col = "blue",
   pch = 19)  # 19 represents solid circle points
```

Example 2: - Matrix of Plots

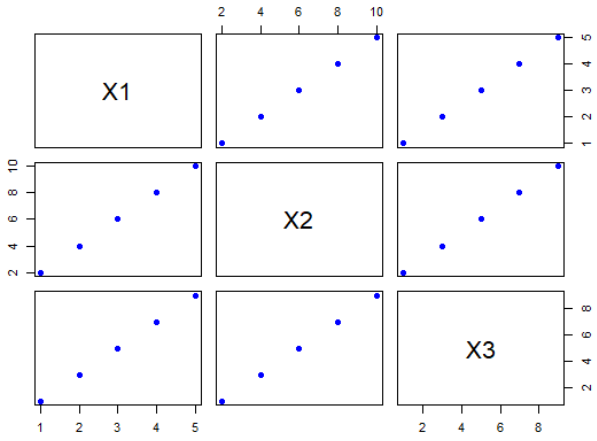
To create a matrix of scatter plots in R, you can use the pairs() function

```
# Sample data
data <- data.frame(
  X1 = c(1,2,3,4,5),
  X2 = c(2,4,6,8,10),
  X3 = c(1,3,5,7,9)
)
# Create a matrix of scatter plots
pairs(data, pch = 16, col = "blue")
```



```
# Create a data frame with scores in three subjects
data <- data.frame(
  Math = c(85, 78, 92, 88, 76, 95, 89, 82, 90, 75),
  Science = c(80, 70, 85, 89, 65, 92, 84, 79, 88, 72),
  English = c(78, 82, 91, 85, 73, 88, 87, 80, 84, 77)
)

# Creating a scatter plot matrix
pairs(data,
      pch = 16,          # Solid circle points
      col = "blue",      # Blue color for all points
      main = "Scatter Plot Matrix of Student Scores")
```

Output



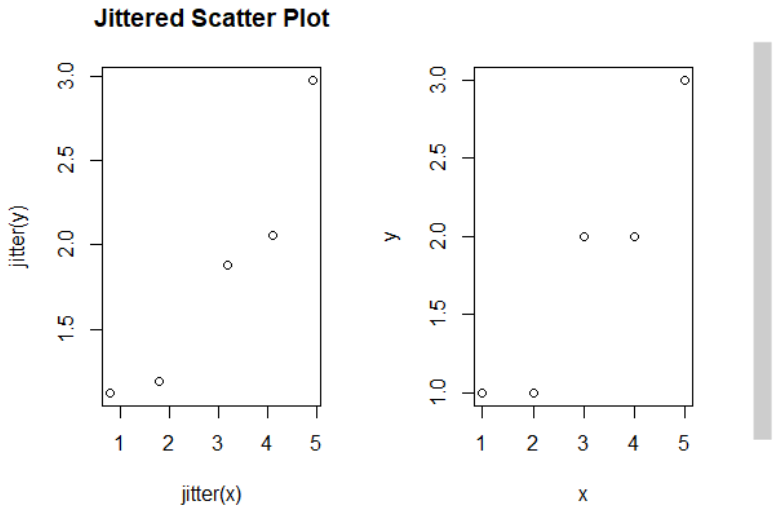
Note:- The **jitter()** function in R is often used to add a small amount of random noise to a numeric vector. This is commonly used in plotting to prevent overplotting, where multiple points might coincide on the plot and make it difficult to discern individual data points.

Example:-

```
x <- c(1, 2, 3, 4, 5)
y <- c(1, 1, 2, 2, 3)
# Create a scatter plot with jittered points
par(mfrow=c(1,2))
print(jitter(x))
print(jitter(y))
plot(jitter(x), jitter(y), main = "Jittered Scatter Plot")
plot(x,y)
par(mfrow=c(1,1))
```

Output:-

```
> source("~/.active-rstudio-document")
[1] 1.104658 1.844843 3.135591 4.125540 5.196565
[1] 0.8193258 1.0987152 1.9677254 2.0328576 3.1322936
```



Probability

- A probability is a number that describes the "magnitude of chance" associated with making a particular observation or statement.

(OR)

- The probability is the measure of the likelihood of an event to happen.
- It's always a number between 0 and 1 (inclusive) and is often expressed as a fraction.

Terminologies in Probability

probability refers to the likelihood of an event occurring, often calculated using mathematical rules and distributions. R provides functions to compute probabilities, simulate random events, and analyze probability distributions such as normal, binomial, and Poisson.

1. Sample Space:- The set of every possible outcome in a probability experiment is known as the sample space.
   Example: - The sample space for tossing a coin is head and tail
   S = {head,tail}
   The sample space for dice thrown
   S={1,2,3,4,5,6}

2. Sample point: - It is one of the possible result.
   Example: - The example of tossing a coin.

S = {head,tail}
- Head is one of the possible outcome. Head is one of the sample point.
- Tail is one of the possible outcome.

3. Experiment: - An experiment is a process or procedure that generates a set of possible outcomes.
   Example:- tossing a coin, Rolling a dice.

4. Event: - An event is a subset of the sample space. It represents a specific outcome or a set of outcomes of the experiment.
   - Pr(E), denotes the probability of an event E.
   - To describe the chance of event A actually occurring, you use a probability, denoted by Pr(A)
   - At the extremes, Pr(A) = 0 suggests A cannot occur, and Pr(A) = 1 suggests that A occurs with complete certainty.

   Example:- The sample space for tossing of three coins simultaneously is given by S={TTT,TTH,THT,HTT,HHH,HHT,HTH,THH}. Suppose, if we want to find the outcomes which have at least two heads, then the set of all possibilities can be given as : Pr(HH) = { HHH,HHT,HTH,THH}.

   Formula for probability

   P(E) = Number of favourable outcomes / Total number of outcomes

   Pr(HH) = 4/8 = ½

5. Approaches of Probability

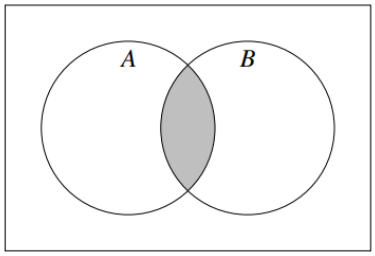
   a. Classical Probability: -
      - Classical probability is a type of probability that is based on the assumption that all possible outcomes of an event are equally likely.
      - This means that classical probability can only be used for events where the number of possible outcomes is known and all of the outcomes are equally likely.
      - For example, if you flip a fair coin, there are two possible outcomes: heads or tails. Since these outcomes are equally likely, the classical probability of getting heads is 1/2.

   b. Frequentist Probability: -
      - Frequentist probability is a type of probability that is based on the idea that probability can be measured by observing the frequency of an event in repeated trials.
      - To calculate the frequentist probability of an event, we need to know the number of times the event has occurred in the past and the total number of trials that have been conducted.
      - For example, if we flip a coin 100 times and get heads 55 times, the frequentist probability of getting heads is 55/100 = 0.55.

   c. Bayseian Proability
      - Bayesian probability is a type of probability that is based on the idea that probability represents a degree of belief. This means that Bayesian probability can be used to update our beliefs about the world in light of new evidence.
      - For example, if we know that the probability of rain tomorrow is 50%, and we see that the forecast is now calling for a 70% chance of rain, then we can update our probability of rain to be 70%.

6. Conditional Probability
   - A conditional probability is the probability of one event occurring after taking into account the occurrence of another event.
   - The quantity Pr(A|B) represents “the probability that A occurs, given that B has already occurred,” and vice versa if you write Pr(B|A).
   - If Pr(A|B) = Pr(A), then the two events are independent; if Pr(A|B) , Pr(A), then the two events are dependent. Generally, you can’t assume that Pr(A|B) is equal to Pr(B|A).

7. Intersection: -
   - Intersection The intersection of two events is written as Pr(A ∩ B) and is read as “the probability that both A and B occur simultaneously.”
   - It is common to represent this as a Venn diagram, as shown here,



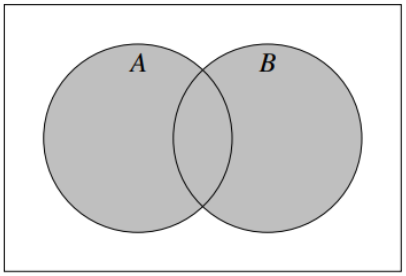
   - Here, the disc labeled A represents the outcome (or outcomes) that satisfies A, and disc B represents the outcomes for B.
   - The shaded area represents the specific outcome (or outcomes) that satisfies both A and B, and the area outside both discs represents the outcome (or outcomes) that satisfies neither A nor B.
   - Theoretically, we have

$$\Pr(A \cap B) = \Pr(A|B) \times \Pr(B) \quad \text{or} \quad \Pr(B|A) \times \Pr(A)$$

   - The equation is also known as the multiplication rule for the probability of the intersection of two events
   - If Pr(A ∩ B) = 0, then you say the two events are mutually exclusive. In other words, they cannot occur simultaneously.
   - When two events A and B are independent, it means that the occurrence of one event does not affect the probability of the other event occurring. In this case, the multiplication rule simplifies to $\Pr(A \cap B) = \Pr(A) \cdot \Pr(B)$.
   - Examples of intersection in probability include finding the probability of drawing a red, face card from a deck of cards (intersection of "red cards" and "face cards")

8. Union: -

   The union of two events is written as Pr(A ∪ B) and is read as “the probability that A or B occurs.” Here is the representation of a union as a Venn diagram:

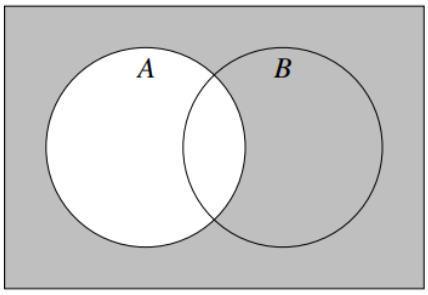


   - Theoretically, we have this,

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$$

- This formula is known as the Inclusion-Exclusion Principle in probability theory.
- The reason you subtract Pr(A∩B) in the formula is to avoid double-counting the probability of events that are in the intersection of A and B. By subtracting Pr(A∩B), you ensure that this shared probability is counted only once in the final probability of the union.
- Examples of union in probability include finding the probability of drawing a red card or a face card from a deck of cards (union of "red cards" and "face cards")

9. Complement: -
    - The probability of the complement of an event is written as Pr(A¯) and is read as “the probability that A does not occur



    - The probability of the complement of an event, denoted as $\Pr(\bar{A}) = 1 - \Pr(A)$
    - Example, If event A is "rolling a 3 on a fair six-sided die," then the complement of A, A', represents all the outcomes that are not a 3 (i.e., 1, 2, 4, 5, and 6).

Random variables and probability distributions **Refer word notes

1. Random variable: -
    - A random variable in statistics is a variable whose value is determined by the outcome of a random experiment.
    - It is a numerical description of the outcome of an experiment. Random variables can be either discrete or continuous.
    - Discrete random variables can take on a finite or infinite number of values. For example, the number of heads in 10 flips of a coin is a discrete random variable, as it can take on the values 0, 1, 2, ..., 10.
    - Continuous random variables can take on any value within a certain range. For example, the weight of a person is a continuous random variable, as it can take on any value between 0 and infinity.
    - When you’ve actually made observations of a random variable, these are referred to as realizations
    - Random variables can be characterized by their probability distribution, which describes the likelihood of the random variable taking on different values.
    - For discrete random variables, this is often represented by a probability mass function (PMF), while for continuous random variables, it is represented by a probability density function (PDF).
    - The cumulative distribution function (CDF) is another important concept associated with random variables, which gives the probability that a random variable is less than or equal to a specific value.

2. Probability Distribution: -

- A probability distribution is a mathematical function that describes the probability of different possible values of a random variable.
- There are two main types of probability distributions:
- Discrete Probability Distribution:
  - This type of distribution is associated with discrete random variables, which can take on a countable set of distinct values.
  - The probability distribution for a discrete random variable is often represented by a probability mass function (PMF), which assigns probabilities to each possible value.
- Continuous Probability Distribution:
  - This type of distribution is associated with continuous random variables, which can take on an uncountable range of values.
  - The probability distribution for a continuous random variable is typically represented by a probability density function (PDF), which describes the relative likelihood of the variable falling within a particular interval.

Discrete Random Variable

- This type of distribution is associated with discrete random variables, which can take on a countable set of distinct values.

Important Concepts in Discrete Random Variable

A Probability Mass Function (PMF) is a function that defines the probability of each possible value for a discrete random variable. It describes the distribution of a discrete random variable by assigning probabilities to each of its specific outcomes.

a. Probability Mass Function

- Probability distributions associated with discrete random variables are indeed represented by probability mass functions (PMFs).
- Discrete random variables are those that can take on a countable set of distinct values, and the PMF assigns a probability to each of these values.
- In mathematical notation, for a discrete random variable X, the PMF is often denoted as **P(X = x),** which represents the probability that X takes on a specific value x.
- The PMF assigns probabilities to each possible outcome of the discrete random variable. It satisfies the following conditions:
  - For any value x, 0 <= P(X = x) <= 1.
  - The sum of probabilities over all possible values of X equals 1: Σ P(X = x) = 1, where the sum is taken over all possible values of X.
- Common examples of discrete probability distributions include the binomial distribution, Poisson distribution, and the discrete uniform distribution

The Cumulative Probability Distribution (CDF) of a discrete random variable is a function that gives the cumulative probability of the variable taking a value less than or equal to a specific value. In other words, the CDF represents the probability that the random variable 'X' is less than or equal to some value 'x'.

b. Cumulative probability distribution of Discrete Random Variable

- The CDF provides the probability that the random variable is less than or equal to a specific value. For a discrete random variable, the CDF is a step function that increases at each possible value of the random variable.
- The CDF for a discrete random variable X is typically denoted as F(x) and is defined as follows:

  F(x) = P(X ≤ x)
- In other words, the CDF at a particular value x gives you the probability that the random variable X takes on a value less than or equal to x.
- To calculate the CDF for a specific value of x for a discrete random variable, you sum the probabilities of all possible values less than or equal to x. In mathematical terms:
- **$F(x) = \Sigma P(X = x_i)=1$,** where the sum is taken over all x_i ,values such that $x_i \leq x$.
- For example, if you have a discrete random variable representing the number of heads obtained when flipping a fair coin three times, the CDF would show how the cumulative probabilities change as you move from 0 to 3 heads:

  F(0) = P(X ≤ 0) = P(X = 0)

F(1) = P(X ≤ 1) = P(X = 0) + P(X = 1)
F(2) = P(X ≤ 2) = P(X = 0) + P(X = 1) + P(X = 2)
F(3) = P(X ≤ 3) = P(X = 0) + P(X = 1) + P(X = 2) + P(X = 3)

c. Mean and Variance of a discrete random variable

- **The mean** of a discrete random variable, often denoted as E(X) or μ (mu), represents the average or expected value of the variable.
- It is calculated as the weighted sum of all possible values of the random variable, each multiplied by its respective probability. The formula for the mean of a discrete random variable X is:

$$\mu_X = \mathbb{E}[X] = x_1 \times \Pr(X = x_1) + \ldots + x_k \times \Pr(X = x_k)$$
$$= \sum_{i=1}^{k} x_i \Pr(X = x_i)$$

- To find the mean, simply multiply the numeric value of each outcome by its corresponding probability and sum the results.
- **Variance** measures the spread or dispersion of the values of a random variable around its mean.
- For a discrete random variable, the variance, often denoted as Var(X) or σ^2 (sigma squared), is calculated as the weighted sum of the squared differences between each value and the mean, each multiplied by its respective probability. The formula for the variance of a discrete random variable X is:

$$\sigma_X^2 = \operatorname{Var}[X] = (x_1 - \mu_X)^2 \times \Pr(X = x_1)$$
$$+ \ldots + (x_k - \mu_X)^2 \times \Pr(X = x_k)$$
$$= \sum_{i=1}^{k} (x_i - \mu_X) \Pr(X = x_i)$$

Continuous Random Variable

- A continuous random variable is a random variable that can take on an infinite number of possible values.
- These values are typically real numbers, but they can also be complex numbers or other types of continuous values.

Important Concepts in Continuous Random Variable

a. Probability Density Function

- It defines the probability distribution of a continuous random variable by specifying the likelihood of the variable falling within a specific range or interval.
- Continuous random variables typically represent measurements or quantities that can assume any real number within their range.
- To find the probability that X falls within the interval [a, b], you need to calculate the area under the PDF curve for the range [a, b]. This is equivalent to finding the integral of the PDF function over that interval.
- The probability that X falls within the interval [a, b] is given by the integral of the PDF from 'a' to 'b':
  P(a ≤ X ≤ b) = ∫[a, b] f(w) dw
  - The integral symbol (∫) represents the area under the curve.
  - 'a' and 'b' are the lower and upper bounds of the interval.
  - 'f(w)' is the Probability Density Function.
- The PDF has the following properties:

- It's non-negative: f(w) ≥ 0 for all x.
- The total area under the PDF curve from negative infinity to positive infinity is equal to 1: **∫[-∞, ∞] f(w) dw = 1.**

- Common examples of continuous probability distributions include the normal (Gaussian) distribution, exponential distribution, and the uniform distribution.

b. Cumulative Probability Distribution of Continuous random variable

- The cumulative probability distribution of a continuous random variable, also known as the cumulative distribution function (CDF), provides the probability that the random variable falls below or equals a specific value. The CDF is often denoted as F(x), where x represents the specific value of the random variable.
- The CDF is defined as follows:
  F(w) = P(X ≤ w)
- To calculate the cumulative probability for a specific value x with a continuous random variable, you can use the CDF. In mathematical notation, you can write it as:
- **F(w) = ∫[-∞, w] f(u) du**, where, f(t) is the Probability Density Function (PDF) of the continuous random variable X. The integral sums up the probabilities from negative infinity to x, representing the cumulative probability up to that point.

c. Mean and Variance of a Continuous Random Variable

- **The mean,** denoted as E(X) or μ, represents the center or average value of the random variable's probability distribution.
- For a continuous random variable X with Probability Density Function (PDF) f(x), the mean is calculated as:
  μ = E(W) = ∫[-∞, ∞] w * f(w) dw
- In words, you multiply each value x by the probability of X taking on that value (given by the PDF) and sum or integrate these products over the entire range of possible values.
- **Variance,** denoted as Var(X) or σ², measures the spread or dispersion of the probability distribution.
- It tells you how much the random variable values deviate from the mean. The variance of a continuous random variable X is calculated as:
  σ² = Var(W) = ∫[-∞, ∞] (w - μ)² * f(w) dw
- In this formula, (x - μ) represents the difference between each value x and the mean μ. You square this difference, multiply it by the probability of X taking on that value (given by the PDF), and integrate over the entire range.

Shape, Skew, Modality

- Shape," "skew," and "modality" are terms used to describe the characteristics and patterns of probability distributions or data sets in statistics.
- These terms provide insights into the nature of the data or distribution. Here's what each term means:

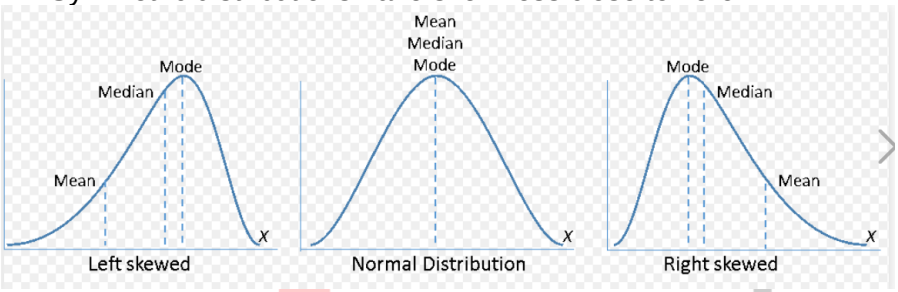
1. **Shape:**
   - "Shape" refers to the overall form or appearance of a probability distribution or data set.
   - It can describe the way the data points are distributed and the general pattern they exhibit.
   - Shapes can vary, and common shapes include bell-shaped, uniform, skewed, bimodal, and multimodal.

Common Shapes:

- Bell-shaped: Data is concentrated around the mean and symmetrically distributed. This shape is often associated with the normal distribution.

- Uniform: Data is evenly spread across the range with no clear peaks or troughs.
- Skewed: Data is asymmetric, with a longer tail on one side. Skewness can be positive (right-skewed) or negative (left-skewed).
- Bimodal: Data has two distinct peaks or modes.
- Multimodal: Data has more than two distinct peaks, indicating multiple modes.

2. **Skewness:**
   - "Skewness" measures the asymmetry of a probability distribution or data set.
   - A distribution or data set can be positively skewed (right-skewed), negatively skewed (left-skewed), or approximately symmetric (no skew).
   - Positive skew means that the tail on the right side is longer or fatter, while negative skew means the left side's tail is longer or fatter.
   - Symmetric distributions have skewness close to zero.



3. **Modality:**
   - "Modality" refers to the number of modes, or peaks, in a probability distribution or data set.
   - A distribution can be unimodal (one peak), bimodal (two peaks), trimodal (three peaks), or multimodal (more than three peaks).
   - Modality provides insights into the number of prominent groups or clusters within the data.

Common probability Mass Function

- Here are some common probability mass functions associated with well-known discrete probability distributions.

1. **Bernoulli Distribution**

- The Bernoulli distribution is the probability distribution of a discrete random variable that has only two possible outcomes, such as success or failure.
- This type of variable can be referred to as binary or dichotomous.
- X be the binary random variable for the success or failure of an event, where X = 0 is failure, X = 1 is success, and p is the known probability of success.
- Below Table shows the probability mass function for X.

Table 16-1: The Bernoulli Probability Mass Function

x	0	1
$Pr(X = x)$	$1 - p$	p

- In mathematical terms, for a discrete random variable X = x, the Bernoulli mass function f is :- **$P(X=x) = p^x (1-p)^{1-x}$; x = {0,1},** where p is a parameter of the distribution.
- The following are the key points to remember:
  - X is dichotomous and can take only the values 1 (“success”) or 0 (“failure”).
  - p should be interpreted as “the probability of success,” and therefore $0 \le p \le 1$
- The mean and variance are defined as follows, respectively:

$$\mu_X = p \qquad \text{and} \qquad \sigma_X^2 = p(1-p)$$

- Lets use the common example of rolling a die, with success defined as getting a 4, and you roll once.
- Therefore a binary random variable X that can be modeled using the Bernoulli distribution, with the probability of success p = 1/6 .
  Pr(Rolling a 4) = P(X=1) = $(1/6)^1 * (5/6)^{1-1}$ = 1/6 *1 = 1/6
  E(X) = 1/6 and V(X) = 1/6*5/6 = 5/36

2. Binomial distribution

- The binomial distribution is a probability distribution that describes the number of successful outcomes in a fixed number of independent Bernoulli trials.
- Each trial has only two possible outcomes, typically labeled as "success" and "failure," with associated probabilities of success (p) and failure (q = 1 - p).
- The probability mass function (PMF) of the binomial distribution, denoted as **B(x; n, p)**, gives the probability of observing exactly x successes in n trials. It is defined as,

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}; \quad x = \{0, 1, \ldots, n\}$$

The Binomial Distribution is a discrete probability distribution that describes the number of successes in a fixed number of independent trials, where each trial has two possible outcomes (success or failure) and a constant probability of success.

In R, functions related to the Binomial Distribution include dbinom, pbinom, qbinom, and rbinom.

where

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

- The binomial distribution is characterized by two parameters:
  - n: The number of trials, which is a fixed and known value.
  - p: The probability of success at each trial.

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

- The above equation is known as binomial coefficient. This coefficient is also referred as combination.
- The mean (μ) and variance (σ^2) of the binomial distribution are given by:
  - Mean :- **μ = n * p**
  - Variance: - **σ^2 = n * p * (1 - p)**
- R supports following functions related to binomial distribution with specified parameters.

dbinom (x,n,p)	It gives individual binomial probability at X=x.
pbinom (x,n,p)	It gives the distribution function P[X<=x].
qbinom ( )	It gives quantile function.
rbinom (m,n,p)	It generates a random sample of size m from binomial distribution.

1. dbinom function

- The dbinom() function in R returns the probability of a specific number of successes in a binomial distribution
- This function calculates the probability mass function (PMF) of the binomial distribution.
- Syntax: - **dbinom(x,size,p)**
  - To the dbinom function, you provide the specific value of interest as x, a vector, argument is possible for x.
  - The total number of trials, n, as size.

- The probability of success at each trial, p, as prob.

Example: -

```
#Probability of getting exactly 3 successes in 5 trials with a success probability of 0.3
    pmf_drv <- dbinom(3,5,0.3)
    print(pmf_drv)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.1323
```

2. pbinom function

- The pbinom() function in R returns the cumulative probability of having at most a specified number of successes in a binomial distribution
- This function calculates the cumulative distribution function (CDF) of the binomial distribution.
- Syntax: - **pbinom(x,size,p)**
  - To the pbinom function, you provide the specific value of interest as x.
  - The total number of trials, n, as size.
  - p, as prob.

Example: -

```
#Cumulative probability of having at most 3 successes in 5 trials with a success probability of 0.3
    cd_drv <- pbinom(3,5,0.3)
    print(cd_drv)
```

Output:-

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.96922
```

3. qbinom function

- The qbinom() function in R returns the quantile of a binomial distribution. It is used to find the number of trials required to achieve a specified cumulative probability or quantile
- Syntax: **- qbinom(p,size,prob)**
- Where,
  - p: The cumulative probability for which you want to find the quantile. It represents the probability of having at most x successes in size trials.
  - size: The number of trials in the binomial experiment.
  - prob: The probability of success on any individual trial.

Example:-

```
#Number of trials required to achieve a cumulative probability of at most 0.8 with a success probability of 0.3
    quantile_bd <- qbinom(0.8,10,0.3)
    print(quantile_bd)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 4
```

4. rbinom function

- The rbinom() function in R returns a vector of random samples generated from a binomial distribution with specified parameters.
- The function simulates random trials, and the output is a collection of random values that follow a binomial distribution.
- Syntax: - **rbinom(n,size,prob)**
- Where,

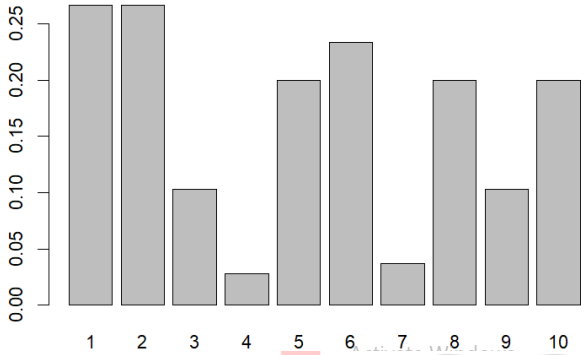
- n: The number of random samples you want to generate.
- size: The number of trials in the binomial experiment.
- prob: The probability of success on any individual trial.

Example: -

```
#Generate 5 random samples from a binomial distribution with 10 trials and a success probability of 0.4
random_sample <- rbinom(10,10,0.4)
print(random_sample)
pmf_bd <- dbinom(random_sample,10,0.3)
print("Probability Distribution")
print(pmf_bd)
barplot(pmf_bd,names.arg = 1:10)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
 [1] 3 3 5 0 4 2 6 4 5 4
[1] "Probability Distribution"
 [1] 0.26682793 0.26682793 0.10291935 0.02824752 0.20012095 0.23347444 0.03675691
 [8] 0.20012095 0.10291935 0.20012095
```



Poisson Distribution

- The Poisson distribution is a probability distribution that describes the number of rare, random events or occurrences within a fixed interval of time or space, when the events are independent and the average rate of occurrence is known.
- The Poisson distribution is characterized by a single parameter, often denoted as λ (lambda), which represents the average rate of occurrence of events.
- The probability mass function (PMF) of the Poisson distribution is given by:
  P(X = x) = (e^(-λ) * λ^x) / x!
- Where,
  - P(X = x) is the probability of observing exactly x events.
  - e is the base of the natural logarithm, approximately equal to 2.71828.
  - λ is the average rate of event occurrence within the given interval.
  - x is the number of events you want to calculate the probability for.
  - x! denotes the factorial of x.
- The mean (μ) and variance (σ^2) of the Poisson distribution are both equal to λ:
  - Mean: - **μ = λ**
  - Variance: - **σ^2 = λ**

R Built-in function for Poisson distribution

- In R, we can work with the Poisson distribution using various built-in functions to calculate probabilities, cumulative probabilities, quantiles, and generate random samples from a Poisson distribution.

1. dpois function

- The dpois() function in R returns the probability of observing exactly a specified number of events in a Poisson distribution with a given average rate of occurrence (λ).
- It calculates the probability mass function (PMF) of the Poisson distribution.
- Syntax: **dpois(x, lambda)**
- Where,
  - x: The specific number of events for which you want to calculate the probability.
  - lambda: The average rate of occurrence (mean) in the Poisson distribution.
- The dpois() function returns a single numeric value representing the probability of observing exactly x events. It tells you how likely it is to see precisely that number of events in the Poisson distribution with the given λ.

Example:-

```
# Probability of observing exactly 3 events in a Poisson distribution with lambda = 5
pmf_pd <- dpois(3,5)
print(pmf_pd)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.1403739
```

2. ppois function

- The ppois() function in R returns the cumulative probability of observing at most a specified number of events in a Poisson distribution with a given average rate of occurrence (λ).
- It calculates the cumulative distribution function (CDF) of the Poisson distribution.
- Syntax: **ppois(x, lambda,)**
- Where,
  - q: The specific number of events for which you want to calculate the cumulative probability.
  - lambda: The average rate of occurrence (mean) in the Poisson distribution.
- The ppois() function returns a single numeric value representing the cumulative probability of observing at most q events in the Poisson distribution with the given λ.

Example: -

```
# Cumulative probability of observing at most 2 events in a Poisson distribution with lambda = 5
cum_pd <- ppois(2,lambda = 5)
print(cum_pd)
```


Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.124652
```

3. qpois function

- The qpois() function in R returns the quantile of a Poisson distribution. It helps you find the number of events (or observations) required to achieve a specified cumulative probability or quantile in a Poisson distribution with a given average rate of occurrence (λ).
- Syntax: **qpois(p, lambda)**
  - p: The cumulative probability for which you want to find the quantile, i.e., the probability of observing at most x events.
  - lambda: The average rate of occurrence (mean) in the Poisson distribution.
- The qpois() function returns a single integer value representing the number of events required to achieve a cumulative probability of at most p in the Poisson distribution with the given λ.

Example: -

```
# Number of events required to achieve a cumulative probability of at most 0.8 in a Poisson distribution with lambda = 5
    quantile_pd <- qpois(0.8, lambda = 5)
    print(quantile_pd)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 7
```

4. rpois function

- The rpois() function in R returns a random sample of values generated from a Poisson distribution with a specified average rate of occurrence (λ). It is used to simulate random events based on a Poisson distribution.
- Syntax: **rpois(n, lambda)**
  - n: The number of random samples you want to generate.
  - lambda: The average rate of occurrence (mean) in the Poisson distribution.
- The rpois() function returns a numeric vector containing n random values, each representing the number of events or occurrences that occurred in independent simulated trials, following a Poisson distribution with the specified λ.

Example: -

```
# Generate 5 random samples from a Poisson distribution with lambda = 5
    random_pd <- rpois(5,lambda = 5)
    print(random_pd)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1]  7 10  6  4  4
```

Common Probability Density Functions

- Probability density functions (PDFs) are mathematical functions that describe the probability distribution of a continuous random variable.
- Here are some of the most common PDFs:

1. **Uniform Distribution:-**
   - In a uniform distribution, all values within a specified range are equally likely to occur. It is often represented as a rectangular shape when visualized, where all values in the range have the same probability density.
   - For a continuous random variable a ≤ X ≤ b, the uniform density function f is

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \le x \le b; \\ 0 & \text{otherwise} \end{cases}$$

   - Where, a and b are parameters of the distribution defining the limits of the possible values X can take.
   - X can take any value in the interval bounded by a and b.
   - a and b can be any values, provided that a < b, and they represent the lower and upper limits, respectively, of the interval of possible values.
   - The mean and variance are as follows:

$$\mu_X = \frac{a+b}{2} \quad \text{and} \quad \sigma_X^2 = \frac{(b-a)^2}{12}$$

   - The cumulative distribution function of a continuous uniform distribution is a linear function that increases uniformly from 0 to 1 within the specified range.
   - The CDF for a continuous uniform distribution on the interval [a, b] is given by:
     F(x) = 0 for x < a

F(x) = (x - a) / (b - a) for a <= x <= b
F(x) = 1 for x > b

R Built- in Functions for uniform distribution

1. **dunif()**

- In R, the dunif() function is used to compute the probability density function (PDF) of a uniform distribution at specific values.
- It allows you to find the probability of a continuous random variable taking on a particular value in a uniform distribution.
- Syntax: - **dunif(x, min = 0, max = 1)**
- Where,
  - x: The value(s) at which you want to compute the PDF.
  - min: The minimum value of the uniform distribution (default is 0).
  - max: The maximum value of the uniform distribution (default is 1).

Example:-

```
#Compute the PDF of a uniform distribution at specific values
pdf_ud <- dunif(c(-2,-0.2,-0.33,1.3,0.8,2),min=-0.4,max=1.5)
print(pdf_ud)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.0000000 0.5263158 0.5263158 0.5263158 0.5263158 0.0000000
```

2. **punif()**

- In R, the punif() function is used to compute the cumulative distribution function (CDF) of a uniform distribution.
- The CDF provides the probability that a random variable following a uniform distribution is less than or equal to a specified value
- Syntax: - **punif(x, min = 0, max = 1)**
- Where,
  - x: The value(s) at which you want to compute the CDF.
  - min: The minimum value of the uniform distribution (default is 0).
  - max: The maximum value of the uniform distribution (default is 1).

Example: -

```
cdf_ud <- punif(c(0.2,0.5,0.7,1.2),min=-1,max=1)
print(cdf_ud)
```


Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.60 0.75 0.85 1.00
```

3. **qunif() function**

- The qunif() function returns the quantile corresponding to a given probability p in a uniform distribution.
- Syntax:- - **punif(p, min = 0, max = 1)**
- Where,
  - p: is the probability for which you want to find the quantile.
  - min: is the minimum value of the distribution (default is 0).
  - max: is the maximum value of the distribution (default is 1).

Example:-

```
q_ud <- qunif(0.1,min=2,max=4)
print(q_ud)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 2.2
```

4. **runif() function**

- The runif() function in R is used to generate random numbers from a uniform distribution.
- Syntax: - **runif(n, min = 0, max = 1)**
- Where,
  - n: The number of random numbers to generate.
  - min: The minimum value of the distribution (default is 0).
  - max: The maximum value of the distribution (default is 1).

Example: -

```
r_ud <- runif(10)
print(r_ud)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
 [1] 0.4677788 0.3639063 0.7803121 0.3519924 0.4865643 0.3732001 0.6779228 0.5118448
 [9] 0.7300008 0.1421339
```

Normal Distribution

- The normal distribution, also known as the Gaussian distribution or bell curve, is a continuous probability distribution that is symmetric and bell-shaped.
- It is characterized by two parameters: the mean (μ) and the standard deviation (σ).
- The probability density function (PDF) of a normal distribution is given by:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}$$

- Where,
  - x is the variable.
  - μ is the mean of the distribution.
  - σ is the standard deviation.
- The cumulative distribution function (CDF) is the integral of the PDF and gives the probability that a random variable from the distribution will be less than or equal to a certain value. For a normal distribution:

  F(x | μ,)=∫[x,−∞] f(t|μ,σ)dt

R built-in Functions for normal distribution

1. **dnorm() function**

- The dnorm() function in R is used to compute the probability density function (PDF) of the normal distribution at specified values.
- Syntax: - **dnorm(x, mean = 0, sd = 1)**
- Where,
  - x: A numeric vector of values at which to evaluate the PDF.
  - mean: The mean of the distribution (default is 0).
  - sd: The standard deviation of the distribution (default is 1).
- The dnorm() function in R returns the value of the probability density function (pdf) of the normal distribution given a certain random variable x

Example:-

```
height <- c(5.5,4.5,5.9,4.3,4.1,5.3,5.5,5.4)
prob_nd <- dnorm(5.5,mean =mean(height),sd = sd(height) )
print(prob_nd)
```

Output: -

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.4839176
```

2. **pnorm()**

- The pnorm() function in R is used to compute the cumulative distribution function (CDF) of a normal distribution.
- The CDF gives the probability that a random variable from the distribution is less than or equal to a specified value.
- Synatx: - **pnorm(q, mean = 0, sd = 1)**
- Where,
    - q: the quantile (the value for which you want to compute the cumulative probability).
    - mean: the mean of the distribution.
    - sd: the standard deviation of the distribution.

Example: -

```
height <- c(5.5,4.5,5.9,4.3,4.1,5.3,5.5,5.4)
cpf <- pnorm(5.9,mean =mean(height),sd = sd(height))
print(cpf)
```

Output:-

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.8966758
```

3. **qnorm()**

- The qnorm() function in R is used to compute quantiles of a normal distribution.
- In other words, it gives you the value of the random variable for a specified probability or percentile.
- Syntax: **- qnorm(p, mean = 0, sd = 1)**
- Where,
    - p: the probability for which you want to find the quantile.
    - mean: the mean of the distribution.
    - sd: the standard deviation of the distribution.

Example

```
height <- c(5.5,4.5,5.9,4.3,4.1,5.3,5.5,5.4)
cpf <- qnorm(0.5,mean =mean(height),sd = sd(height))
print(cpf)
```


Output

```
> source("D:/ICCA/R_Program/prob.R")
[1] 5.0625
```

4. **rnorm()**

- The rnorm() function in R is used to generate random numbers from a normal distribution.
- Syntax:- **rnorm(n, mean = 0, sd = 1)**
    - n: the number of random samples to generate.
    - mean: the mean of the distribution.
    - sd: the standard deviation of the distribution.

Example

```
random_norm <- rnorm(50)
print(random_norm)
```

Output

```
> source("D:/ICCA/R_Program/prob.R")
 [1]  1.22408180  0.35981383  0.40077145  0.11068272 -0.55584113  1.78691314  0.49785048
 [8] -1.96661716  0.70135590 -0.47279141 -1.06782371 -0.21797491 -1.02600445 -0.72889123
[15] -0.62503927 -1.68669331  0.83778704  0.15337312 -1.13813694  1.25381492  0.42646422
[22] -0.29507148  0.89512566  0.87813349  0.82158108  0.68864025  0.55391765 -0.06191171
[29] -0.30596266 -0.38047100 -0.69470698 -0.20791728 -1.26539635  2.16895597  1.20796200
[36] -1.12310858 -0.40288484 -0.46665535  0.77996512 -0.08336907  0.25331851 -0.02854676
[43] -0.04287046  1.36860228 -0.22577099  1.51647060 -1.54875280  0.58461375  0.12385424
[50]  0.21594157
```

Student's t- distribution

ibution is a probability distribution used to estimate population parameters when the sample size is small and/or the population standard deviation is unknown. It is symmetric and bell-shaped, similar to the normal distribution, but has heavier tails,

- The Student's t-distribution is a continuous probability distribution generally used when dealing with statistics estimated from a sample of data.
- Any particular t-distribution looks a lot like the standard normal distribution it's bell-shaped, symmetric, and unimodal, and it's centered on zero.
- The difference is that while a normal distribution is typically used to deal with a population, the t-distribution deals with sample from a population.
- In the context of the t-distribution, degrees of freedom refer to the number of values in the final calculation of a statistic that are free to vary. Specifically, for a t-distribution, degrees of freedom are associated with the sample size.
- If you have a sample of size n, the degrees of freedom for a t-distribution would be n−1. This adjustment accounts for the fact that you've used information from the sample to estimate the population mean.
- For example, if you have 20 data points in your sample, you would use a t-distribution with 19 degrees of freedom.

It is defined by degrees of freedom (df).
As the degrees of freedom increase, the t-distribution approaches the normal distribution.

R built-in Functions for Student's t- distribution

1. dt()

- The dt() function is used to evaluate the probability density function (PDF) of a t-distribution.
- Syntax: - **dt(x, df)**
- Where,
  - x: vector of quantiles.
  - df: degrees of freedom.

Example:-

```
random_t <- rt(20,19)
probability_t <- dt(random_t,19)
print("Random Value genertion")
print(random_t)
print("Probability")
print(probability_t)
```

Output

```
> source("D:/ICCA/R_Program/prob.R")
[1] "Random Value genertion"
 [1] -2.89930212  0.10617930  0.02394968 -2.06397452 -0.47983441 -0.26476778 -0.32954318
 [8] -0.49077634 -0.81654526 -0.86307400 -2.04560334 -1.41567715 -0.63366322 -0.31546909
[15] -0.23312853  0.26418187  0.11140686  0.69434174 -0.46425344 -0.39842027
[1] "Probability"
 [1] 0.01009916 0.39140114 0.39361096 0.05207616 0.34904950 0.37949333 0.37191689 0.34712763
 [9] 0.27887474 0.26803138 0.05379703 0.14443914 0.31942927 0.37368839 0.38264247 0.37955501
[17] 0.39116704 0.30646035 0.35173075 0.36229735
```

2. pt()

- The pt() function in R is used to compute the cumulative distribution function (CDF) of the Student's t-distribution.
- Syntax:- **pt(q, df)**
- Where,

- q: a vector of quantiles.
- df: degrees of freedom.

Example: -

```
cum_pt <- pt(2,10)
print(cum_pt)
```

- This will return the probability that a random variable from the Student's t-distribution with 10 degrees of freedom will be less than or equal to 2.5.

Output:-

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.963306
```

3. qt()

- The qt() function in R is used to calculate the quantile function of the Student's t-distribution.
- The quantile function is the inverse of the cumulative distribution function (CDF).
- Syntax:- **qt(p, df)**
- Where,
  - p is the probability.
  - df is the degrees of freedom.

Example:-

```
cum_pt <- pt(2,10)
print(cum_pt)
quna_qt <- qt(0.963306,10)
print(quna_qt)
```

- This will return the value of the random variable such that there is a 96.33% chance that a random variable from the Student's t-distribution with 10 degrees of freedom will be less than or equal to that value.

Output

```
> source("D:/ICCA/R_Program/prob.R")
[1] 0.963306
[1] 2
```

4. rt()

- The rt() function is used to generate random numbers from a t-distribution with a specified number of degrees of freedom. Here's the basic syntax:
- Syntax: - **rt(n, df)**
- Where,
  - n: the number of random samples to generate.
  - df: degrees of freedom.

Example

```
random_t <- rt(20,19)
print(random_t)
```

Output

```
> source("D:/ICCA/R_Program/prob.R")
 [1] -0.882808660  0.746252470  0.312759644  1.409608846 -0.689448425 -0.281343130
 [7] -0.099523091 -1.789352559  1.314911690  1.745753475  0.206749239  1.130267661
[13] -0.972567319 -0.183071829 -0.277925710  0.501388616 -0.003792052 -0.165530387
[19]  0.561588733 -0.646632936
```

Unit IV :- Statistical Testing and modelling

Sampling Distribution

In statistics, sampling distribution refers to the probability distribution of a statistic (like mean, variance, or proportion) that is obtained from multiple random samples of a specific size taken from a population. It describes how the statistic varies from sample to sample and helps us to understand the spread and behaviour of the estimate around the true population parameter.
Key Concepts
Population: The entire set of data or observations you're interested in.
Sample: A subset of data randomly drawn from the population.
Statistic: A numerical summary of a sample, such as the mean or variance.
Sampling Distribution: The distribution of a statistic (e.g., sample mean) over many samples.
Example:
If you take multiple random samples from a population and calculate the sample mean for each, the distribution of those sample means is the sampling distribution of the mean.

Standard error

The Standard Error (SE) measures the variability (or dispersion) of a sample statistic, such as the sample mean or sample proportion, from the true population parameter. It is essentially the standard deviation of the sampling distribution of a statistic.
The standard error describes variability in the sampling distribution of the statistic

- The standard deviation of a sampling distribution is referred to as a Standard error.
- The standard error, on the other hand, is a measure of the variability of a sample statistic. It quantifies how much a sample statistic (such as the mean) is expected to vary from the true population parameter.
- The standard error is influenced by two main factors:
    - Sample Size (n):
        - As the sample size increases, the standard error decreases. Larger sample sizes provide more reliable estimates of the population parameter.
    - Population Variability (σ):
        - Higher variability in the population leads to higher standard errors. If individual observations in the population are more spread out, the sample mean is expected to vary more from sample to sample.

1. Distribution for a sample mean

- The arithmetic mean is arguably the most common measure of centrality.
- Mathematically, the variability inherent in an estimated sample mean is described as follows:
    - Formally, denote the random variable of interest as $\bar{X}$. This represents the mean of a sample of n observations from the "raw observation" random variable X, as in x1, x2,. . . , xn.
- Those observations are assumed to have a true finite mean $-\infty < \mu_X < \infty$ and a true finite standard deviation $0 < \sigma_X < \infty$.
- The conditions for finding the probability distribution of a sample mean vary depending on whether you know the value of the standard deviation.

Situation 1: Standard Deviation Known

When the true value of the standard deviation σX is known, then the following are true:

1. If X itself is normal, the sampling distribution of $\bar{X}$ is a normal distribution, with mean μ_X and standard error $\sigma_X/\sqrt{n}$.
2. If X is not normal, the sampling distribution of $\bar{X}$ is still approximately normal, with mean μ_X and standard error $\sigma_X/\sqrt{n}$, and this approximation improves arbitrarily as $n \to \infty$. This is known as the central limit theorem (CLT)

Situation 2: Standard Deviation Unknown

- The properties of the sampling distribution of the sample mean ($\bar{X}$) when the standard deviation of the population (σ) is unknown, and the sample standard deviation (s) is used instead.
    - Standardized values of the sampling distribution of $\bar{X}$ follow a t-distribution with v = n − 1 degrees of freedom; standardization is performed using the standard error $s_X/\sqrt{n}$.

- If, additionally, n is small, then it is necessary to assume the distribution of X is normal for the validity of this t-based sampling distribution of $\bar{X}$.

2. Distribution for a Sample Proportion

- Sampling distributions for sample proportions are interpreted in much the same way.
- If n trials of a success/failure event are performed, you can obtain an estimate of the proportion of successes; if another n trials are performed, the new estimate could vary. It's this variability that we're investigating.
- The random variable of interest, P^(P cap), represents the estimated proportions of successes over any n trials, each resulting in some defined binary outcome.
- It is estimated as p^ = x/n , where x is the number of successes in a sample of size n.
- Let the corresponding true proportion of successes (often unknown) simply be denoted with π.

Note:- π symbol is used to represent standard notation to represent true population proportion.

- The sampling distribution of P^ is approximately normal with mean π and standard error $\sqrt{\pi(1-\pi)/n}$. The following are the key things to note:
  - This approximation is valid if n is large and/or π is not too close to either 0 or 1.
  - There are rules of thumb to determine this validity; one such rule is to assume the normal approximation is satisfactory if both nπ and n(1 − π) are greater than 5.
  - When the true π is unknown or is unassumed to be a certain value, it is typically replaced by p^ in all of the previous formulas.

Confidence Intervals

- A confidence interval (CI) is an interval defined by a lower limit l and an upper limit u, used to describe possible values of a corresponding true population parameter in light of observed sample data.
- Interpretation of a confidence interval therefore allows you to state a "level of confidence" that a true parameter of interest falls between this upper and lower limit, often expressed as a percentage.
- The following are the important points to note:
  - The level of confidence is usually expressed as a percentage, such that you'd construct a 100 × (1 − α) percent confidence interval, where 0 < α < 1 is an "amount of tail probability."
  - The three most common intervals are defined with either α = 0.1 (a 90 percent interval), α = 0.05 (a 95 percent interval), or α = 0.01 (a 99 percent interval).
  - Colloquially, you'd state the interpretation of a confidence interval (l,u) as "I am 100 × (1 − α) percent confident that the true parameter value lies somewhere between l and u.
- Confidence intervals may be constructed in different ways, depending on the type of statistic and therefore the shape of the corresponding sampling distribution.
- For symmetrically distributed sample statistics, like means and proportions, a general formula is
  Confidence intervals = statistic ± critical value × standard error
- Where, statistic is the sample statistic under scrutiny, critical value is a value from the standardized version of the sampling distribution that corresponds to α, and standard error is the standard deviation of the sampling distribution.
- The product of the critical value and standard error is referred to as the error component of the interval(Margin of error) ; subtraction of the error component from the value of the statistic provides l, and addition provides u

1. An Interval for a Mean
   - The sampling distribution of a single sample mean depends primarily on whether you know the true standard deviation of the raw measurements, σX.
   - Then, provided the sample size for this sample mean is roughly n ≥ 30, the CLT ensures a symmetric sampling distribution—which will be normal if you know the true value of σX, or t based with v = n − 1 df if you must use the sample standard deviation, s, to estimate σX.
   - To construct an appropriate interval, you must first find the critical value corresponding to α.
   - By definition the CI is symmetric, so this translates to a central probability of (1 − α) around the mean, which is exactly α/2 in the lower tail and the same in the upper tail.
   - As we're interested in the sample mean and its sampling distribution, we must calculate the sample mean $\bar{x}$, the sample standard deviation s, and the appropriate standard error of the sample mean, s/ √ n.

2. An Interval for a proportion
   - Establishing a CI for a sample proportion follows the same rules as for the mean. With knowledge of the sampling distribution, you obtain critical values from the standard normal distribution, and for an estimate of p^ from a sample of size n, the interval itself is constructed with the standard error √ p^(1 – p^)/n.

Hypothesis Testing
Hypothesis testing is a statistical method used to make decisions or inferences about a population parameter based on sample data. It helps determine whether there is enough evidence in the data to support or reject a certain claim (the hypothesis).

Components of Hypothesis

1. **Hypotheses**
   Hypothesis testing is a statistical method that uses data to evaluate two competing hypotheses. The two hypotheses are the
   - Null hypothesis (H_0) :- The null hypothesis is often (but not always) defined as an equality, =, to a null value
   - alternative hypothesis (H_a): - the alternative hypothesis, is often defined in terms of an inequality to the null value.
     - When HA is defined in terms of a less-than statement, with < , it is one – sided; this is also called as lower- tailed test.
     - When HA is defined in terms of a greater-than statement, with >, it is one-sided; this is also called an upper-tailed test.
     - When HA is merely defined in terms of a different-to statement, with it is two-sided; this is also called a two-tailed test.
2. **Test Statistic**
   - Once the hypotheses are formed, sample data are collected, and statistics are calculated according to the parameters detailed in the hypotheses. The test statistic is the statistic that's compared to the appropriate standardized sampling distribution to yield the p-value.

3. **P-value**
   The p-value is the probability value that's used to quantify the amount of evidence, if any, against the null hypothesis.
   The exact nature of calculating a p-value is dictated by the type of statistics being tested and the nature of H_A.

4. **Significance level**
   - For every hypothesis test, a significance level, denoted α, is assumed.
   - This is used to qualify the result of the test.
   - The significance level defines a cutoff point, at which you decide whether there is sufficient evidence to view H_0 as incorrect and favor H_A instead.
     - If the p-value is greater than or equal to α, then you conclude there is insufficient evidence against the null hypothesis, and therefore you retain H_0 when compared to H_A
     - If the p-value is less than α, then the result of the test is statistically significant. This implies there is sufficient evidence against the null hypothesis, and therefore you reject H_0 in favor of H_A.
   - Common or conventional values of α are α = 0.1, α = 0.05, and α = 0.01.

Testing Mean

- In statistics, testing means typically refers to hypothesis testing involving the means of one or more populations. The goal is to assess whether there is enough evidence in a sample to infer something about a population parameter, usually the population mean.

Single mean (One sample t -test)

- Single mean hypothesis testing is a statistical procedure used to assess whether there is enough evidence in a sample to make a claim about a population mean. This type of hypothesis testing is often framed as a comparison between the sample mean and a hypothesized population mean.
- Let's dive straight into an example—a one-sample t-test.
- A manufacturer of a snack was interested in the mean net weight of contents in an advertised 80-gram pack.
- A consumer calls in with a complaint—over time they have bought and precisely weighed the contents of 44 randomly selected 80-gram packs from different stores and recorded the weights as follows:
- snacks <- c(87.7,80.01,77.28,78.76,81.52,74.2,80.71,79.5,77.87,81.94,80.7, 82.32,75.78,80.19,83.91,79.4,77.52,77.62,81.4,74.89,82.95, 73.59,77.92,77.18,79.83,81.23,79.28,78.44,79.01,80.47,76.23, 78.89,77.14,69.94,78.54,79.7,82.45,77.29,75.52,77.21,75.99, 81.94,80.41,77.7)
- The customer claims that they've been shortchanged because their data cannot have arisen from a distribution with mean μ = 80, so the true mean weight must be less than 80.
- To investigate this claim, the manufacturer conducts a hypothesis test using a significance level of α = 0.05.

Steps involved in single Hypothesis testing

1. Formulate Hypothesis.
   H_0 :- μ = 80
   H_a :- μ <80
2. Choose a significance level
   α = 0.05
3. Collect data
   snacks <- c(87.7,80.01,77.28,78.76,81.52,74.2,80.71,79.5,77.87,81.94,80.7, 82.32,75.78,80.19,83.91,79.4,77.52,77.62,81.4,74.89,82.95, 73.59,77.92,77.18,79.83,81.23,79.28,78.44,79.01,80.47,76.23, 78.89,77.14,69.94,78.54,79.7,82.45,77.29,75.52,77.21,75.99, 81.94,80.41,77.7)
   Mean value :- 78.9106.

4. Test Statistic
   - The test statistic T in a hypothesis test for a single mean with respect to a null value of μ0 is given as $T = \bar{x} - \mu_0 / (s/\sqrt{n})$.
   - Where,
     - $\bar{X}$ is the sample mean,
     - μ0 is the hypothesized population mean,
     - s is the sample standard deviation,
     - n is the sample size.
5. Determine the p-value
   - Find the probability of T value
6. Make decision
   a. If the p-value is greater than or equal to α, then you conclude there is insufficient evidence against the null hypothesis, and therefore you retain H_0 when compared to H_A
   b. If the p-value is less than α, then the result of the test is statistically significant. This implies there is sufficient evidence against the null hypothesis, and therefore you reject H_0 in favor of H_A.

R – Function

- R function used for one – sample test is t.test()

Syntax:- t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"), mu = 0)

Where,

- x: a numeric vector of data values for the first group.
- 1y: (optional) a numeric vector of data values for the second group (for two-sample tests).
- alternative: a character string specifying the alternative hypothesis. It can be one of "two.sided" (default), "less", or "greater".
- mu: the hypothesized population mean under the null hypothesis (default is 0).

Example

```
snacks <- c(87.7,80.01,77.28,78.76,81.52,74.2,80.71,79.5,77.87,81.94,80.7,
            82.32,75.78,80.19,83.91,79.4,77.52,77.62,81.4,74.89,82.95,
            73.59,77.92,77.18,79.83,81.23,79.28,78.44,79.01,80.47,76.23,
            78.89,77.14,69.94,78.54,79.7,82.45,77.29,75.52,77.21,75.99,
            81.94,80.41,77.7)
print(t.test(snacks,mu = 80,alternative = "less"))
```


Output

```
        One Sample t-test

data:  snacks
t = -2.3644, df = 43, p-value = 0.01132
alternative hypothesis: true mean is less than 80
95 percent confidence interval:
     -Inf 79.68517
sample estimates:
mean of x
 78.91068
```

- In examining the result for the snack example, with a p-value of around 0.011, With α set at 0.05 for this particular test, H0 is rejected.

Two Means

- A two sample t-test is a way of comparing the means of two different groups of measurements to see if they are significantly different or not.
- The typical null hypothesis is usually defined as μ1 and μ2 being equal
- A two sample t-test can be used for two types of data: independent samples or paired samples.
- Independent samples are when the measurements in one group are not related to the measurements in the other group. For example, you might want to compare the mean weight of two different species of turtles, or the mean test scores of two different classes of students.
- Paired samples are when the measurements in one group are matched or paired with the measurements in the other group. For example, you might want to compare the mean blood pressure of patients before and after a treatment, or the mean satisfaction ratings of customers.

a. Unpaired/ Independent Sample: Unpooled Variance

- Unpaired or independent samples are when the measurements in one group are not related to the measurements in the other group. For example, you might want to compare the mean weight of two different species of turtles, or the mean test scores of two different classes of students.
- Unpooled variance is a way of estimating the variability of the data when the assumption of equal variances is not valid. This means that the two groups of measurements have different amounts of variation or spread. For example, you might want to compare the mean height of men and women, but the height of men might have more variation than the height of women.
- When you want to compare the means of two unpaired or independent samples with unpooled variance, you need to use a slightly modified version of the t-test called the **Welch test.**
- This test uses separate or unpooled variances for each group, and adjusts the degrees of freedom and the critical value accordingly. The formula for the test statistic and the confidence interval for the difference between the two means are

$$T = \frac{\bar{x}_2 - \bar{x}_1 - \mu_0}{\sqrt{s_1^2/n_1 + s_2^2/n_2}},$$

- whose distribution is approximated by a t-distribution with v degrees of freedom, where

$$\nu = \left\lfloor \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{(s_1^2/n_1)^2/(n_1 - 1) + (s_2^2/n_2)^2/(n_2 - 1)} \right\rfloor$$

Example :- After collecting a sample of 44 packs from the original manufacturer A manufacturer, the consumer goes out and collects 31 packs randomly selected 80-gram packs from a B manufacturer. And the claim is μ2 is greater than μ1

1. Formulate Hypothesis
   H_0 = μ2 - μ1 = 0
   H1 = μ2 > μ1
2. Choose a significance level
   α = 0.05
3. Collect data

A <- c(87.7,80.01,77.28,78.76,81.52,74.2,80.71,79.5,77.87,81.94,80.7,
82.32,75.78,80.19,83.91,79.4,77.52,77.62,81.4,74.89,82.95,
73.59,77.92,77.18,79.83,81.23,79.28,78.44,79.01,80.47,76.23,
78.89,77.14,69.94,78.54,79.7,82.45,77.29,75.52,77.21,75.99, 81.94,80.41,77.7)
Mean value :- 78.9106.
snacks_B <- c(80.22,79.73,81.1,78.76,82.03,81.66,80.97,81.32,80.12,78.98,
79.21,81.48,79.86,81.06,77.96,80.73,80.34,80.01,81.82,79.3,
79.08,79.47,78.98,80.87,82.24,77.22,80.03,79.2,80.95,79.17,81)
Mean value:- 80.1571

4. Test statistic
   - The formula for the test statistic and the confidence interval for the difference between the two means are

$$T = \frac{\bar{x}_2 - \bar{x}_1 - \mu_0}{\sqrt{s_1^2/n_1 + s_2^2/n_2}},$$

   - Whose distribution is approximated by a t-distribution with v degrees of freedom, where

$$\nu = \left\lfloor \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{(s_1^2/n_1)^2/(n_1 - 1) + (s_2^2/n_2)^2/(n_2 - 1)} \right\rfloor$$

5. Determine the p-value
   - Find the probability of T value
6. Make decision
   a. If the p-value is greater than or equal to α, then you conclude there is insufficient evidence against the null hypothesis, and therefore you retain H_0 when compared to H_A
   b. If the p-value is less than α, then the result of the test is statistically significant. This implies there is sufficient evidence against the null hypothesis, and therefore you reject H_0 in favor of H_A.

R – Function

- R function used for unpooled version of two – sample test is t.test()

Syntax:- t.test(x, y = NULL, alternative = c("two.sided", "less", "greater",conf.level=0.95), mu = 0)

Where,

- x: a numeric vector of data values for the first group.
- 1y: (optional) a numeric vector of data values for the second group (for two-sample tests).
- alternative: a character string specifying the alternative hypothesis. It can be one of "two.sided" (default), "less", or "greater".
- mu: the hypothesized population mean under the null hypothesis (default is 0).
- conf.level: the confidence level for the interval (default is 0.95).

Example

snacks_A <- c(87.7,80.01,77.28,78.76,81.52,74.2,80.71,79.5,77.87,81.94,80.7,
82.32,75.78,80.19,83.91,79.4,77.52,77.62,81.4,74.89,82.95,
73.59,77.92,77.18,79.83,81.23,79.28,78.44,79.01,80.47,76.23,
78.89,77.14,69.94,78.54,79.7,82.45,77.29,75.52,77.21,75.99,
81.94,80.41,77.7)
snacks_B <- c(80.22,79.73,81.1,78.76,82.03,81.66,80.97,81.32,80.12,78.98,

```
            79.21,81.48,79.86,81.06,77.96,80.73,80.34,80.01,81.82,79.3,
            79.08,79.47,78.98,80.87,82.24,77.22,80.03,79.2,80.95,79.17,81)
print(t.test(snacks_B,snacks_A,alternative = "greater",conf.level = 0.9))
```

Output

```
> source("D:/ICCA/R_Program/Testing_mean.R")

        Welch Two Sample t-test

data:  snacks_B and snacks_A
t = 2.4455, df = 60.091, p-value = 0.008706
alternative hypothesis: true difference in means is greater than 0
90 percent confidence interval:
 0.5859714       Inf
sample estimates:
mean of x mean of y
 80.15710  78.91068
```

- With a small p-value of 0.008706, you'd conclude that there is sufficient evidence to reject H0 in favor of HA (indeed, the p-value is certainly smaller than the stipulated α = 0.1 significance level as implied by conf.level=0.9)

b. Unpaired/ Independent Sample: pooled Variance

- Unpaired or independent samples are when the measurements in one group are not related to the measurements in the other group. For example, you might want to compare the mean weight of two different species of turtles, or the mean test scores of two different classes of students.
- Pooled variance is a way of estimating the variability of the data when the assumption of equal variances is valid. This means that the two groups of measurements have the same amount of variation or spread.
- The formula for Test statistic of pooled variance

$$T = \frac{\bar{x}_2 - \bar{x}_1 - \mu_0}{\sqrt{s_p^2\,(1/n_1 + 1/n_2)}},$$

- Where, S_p^2 is a pooled standard deviation, calculated using the below formula

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}$$

- Whose distribution is a t-distribution with **v = n1 + n2 − 2** degrees of freedom

R function used for pooled version of two – sample test is t.test()

Syntax:- t.test(x, y = NULL, mu = 0, alternative = c("two.sided", "less", "greater" , conf.level = 0.95 ,var.equal = FALSE),

Where,

- x: a numeric vector of data values for the first group.
- 1y: (optional) a numeric vector of data values for the second group (for two-sample tests).
- alternative: a character string specifying the alternative hypothesis. It can be one of "two.sided" (default), "less", or "greater".
- mu: the hypothesized population mean under the null hypothesis (default is 0).
- conf.level: the confidence level for the interval (default is 0.95).
- var.equal: a logical indicating whether to assume equal variances (only applicable for two-sample tests, default is FALSE).

  Hypothesis H_0 = μ2 - μ1 = 0

H1 = μ2 - μ1 != 0

Example:-

```
snacks_A <- c(87.7,80.01,77.28,78.76,81.52,74.2,80.71,79.5,77.87,81.94,80.7,
              82.32,75.78,80.19,83.91,79.4,77.52,77.62,81.4,74.89,82.95,
              73.59,77.92,77.18,79.83,81.23,79.28,78.44,79.01,80.47,76.23,
              78.89,77.14,69.94,78.54,79.7,82.45,77.29,75.52,77.21,75.99,
              81.94,80.41,77.7)
snacks_B <- c(80.22,79.73,81.1,78.76,82.03,81.66,80.97,81.32,80.12,78.98,
              79.21,81.48,79.86,81.06,77.96,80.73,80.34,80.01,81.82,79.3,
              79.08,79.47,78.98,80.87,82.24,77.22,80.03,79.2,80.95,79.17,81)
print(t.test(snacks_B,snacks_A,alternative = "two.sided",conf.level = 0.9,var.equal =
TRUE))
```

Output

```
        Two Sample t-test

data:  snacks_B and snacks_A
t = 2.151, df = 73, p-value = 0.03479
alternative hypothesis: true difference in means is not equal to 0
90 percent confidence interval:
 0.2810388 2.2117911
sample estimates:
mean of x mean of y
 80.15710  78.91068
```

c. Paired/Dependent Samples

- Paired or dependent samples are when the measurements in one group are related to the measurements in the other group. For example, you might want to compare the blood pressure of patients before and after a treatment
- In these cases, each group contains the same subjects or subjects that are matched in some way.
- A common statistical test for paired or dependent samples is the paired t-test.
- This test compares the mean difference between the two groups with a hypothesized value, usually zero.
- The null hypothesis is that there is no difference between the two groups, or that the mean difference is zero.
- The alternative hypothesis is that there is a difference between the two groups, or that the mean difference is not zero.
- The paired two-sample t-test considers the difference between each pair of values.
- Labeling one set of n measurements as x1, . . ., xn and the other set of n observations as y1, . . ., yn, the difference, d, is defined as di = yi − xi ; i = 1, . . ., n.
- Compute the Sample Mean of Differences ($\bar{d}$).

$$\bar{d} = \frac{\sum_{i=1}^{n} d_i}{n}$$

- The test statistic T is given as

$$T = \frac{\bar{d} - \mu_0}{s_d / \sqrt{n}},$$

- Where $\bar{d}$ is the mean of the pairwise differences, sd is the sample standard deviation of the pairwise differences, and μ0 is the null value (usually zero). The statistic T follows a t-distribution with n − 1 df

R function paired sample is t.test()

Syntax:- t.test(x, y = NULL, mu = 0, alternative = c("two.sided", "less", "greater" , conf.level = 0.95 ,paired = TRUE),
Where,

- x: a numeric vector of data values for the first group.
- 1y: (optional) a numeric vector of data values for the second group (for two-sample tests).
- alternative: a character string specifying the alternative hypothesis. It can be one of "two.sided" (default), "less", or "greater".
- mu: the hypothesized population mean under the null hypothesis (default is 0).
- conf.level: the confidence level for the interval (default is 0.95).

Example

```
rate.before <- c(52,66,89,87,89,72,66,65,49,62,70,52,75,63,65,61)
rate.after <- c(51,66,71,73,70,68,60,51,40,57,65,53,64,56,60,59)
print(t.test(rate.after,rate.before,alternative="less",paired = TRUE))
```

Output

```
        Paired t-test

data:  rate.after and rate.before
t = -4.8011, df = 15, p-value = 0.0001167
alternative hypothesis: true mean difference is less than 0
95 percent confidence interval:
      -Inf -4.721833
sample estimates:
mean difference
       -7.4375
```

Testing Proportions

- In statistics, when we talk about means, we're often referring to the average values of numerical data.
- However, there are situations where our data is categorical, like success/failure (1/0) in binary trials. In these cases, we talk about proportions, which is like the average of a series of binary outcomes.

a. Single Proportion (One sample Z-test)

- Hypothesis testing on a single proportion is a statistical analysis used to determine whether a sample proportion significantly differs from a hypothesized population proportion.

Example:- The website of a blogger who believes that the chance of getting an upset stomach shortly after eating that particular food is 20 percent. The individual is curious to determine whether his true rate of stomach upset π is any different from the blogger's quoted value and, over time, visits these fast-food outlets for lunch on n = 29 separate occasions, recording the success (TRUE) or failure (FALSE) of experiencing an upset stomach

Steps involved

1. Formulate Hypotheses:
   - Null Hypothesis (H0): π =0.20.
   - Alternative Hypothesis (H1): π ≠0.20.

2. Choose a significance level
   α = 0.05

3. Collect Data:
   - Record the number of successes (upset stomach) and failures (no upset stomach) over the n=29 occasions.
   - sick <- c(0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1).

4. Z-Test Statistic for Proportions:
   Use the formula:-

$$Z = \frac{\hat{p} - \pi_0}{\sqrt{\frac{\pi_0(1-\pi_0)}{n}}}$$

5. Find the p-value

6. Make a Decision:
   - If the calculated z value falls into the critical region or if the p-value is less than 0.05, reject the null hypothesis.
   - If the calculated z value is not in the critical region or if the p-value is greater than 0.05, fail to reject the null hypothesis.

R – Function

Syntax :- prop.test(x, n, p = NULL, alternative = "two.sided", conf.level = 0.95, correct = TRUE)

Where,
- x: A numeric vector of successes (number of successes).
- n: A numeric vector of the total number of trials.
- p: A vector of the null hypothesis proportions (default is p=0.5p=0.5).
- alternative: A character string specifying the alternative hypothesis. Possible values are "two.sided", "less", or "greater".
- conf.level: The confidence level for the confidence interval.
- correct: Logical, indicating whether to apply continuity correction (default is TRUE).

Example

```
sick <- c(0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1)
print(prop.test(x=sum(sick), n= length(sick),p=0.2,correct = FALSE))
```

Output

```
        1-sample proportions test without continuity correction

data:  sum(sick) out of length(sick), null probability 0.2
X-squared = 1.0431, df = 1, p-value = 0.3071
alternative hypothesis: true p is not equal to 0.2
95 percent confidence interval:
 0.1469876 0.4571713
sample estimates:
        p
0.2758621
```

Two Proportion (Two sample Z-test)
- Testing two proportions involves comparing the proportions of success in two independent groups to determine if there is a significant difference between them.

- This type of hypothesis test is often used in situations where you want to assess whether the proportions of success in two groups are equal or if one is significantly greater or lesser than the other.

Example: - For an example, consider a group of students taking a statistics exam. In this group are n1 = 233 students majoring in psychology, of whom x1 = 180 pass, and n2 = 197 students majoring in geography, of whom 175 pass. Suppose it is claimed that the geography students have a higher pass rate in statistics than the psychology students

Steps involved

1. Formulate Hypotheses
   - H0 : π2 − π1 = 0
   - HA : π2 − π1 > 0

2. Choose a significance level
   - α = 0.05

3. Collect Data:
   - For each group, record the number of successes and the total number of trials.

```
n_psychology <- 233
x_statistic_psychology <- 180
n_geography <- 197
x_statistic_geography <- 175
```

4. Calculate Test statistic
   - Use the formula

$$Z = \frac{\hat{p}_2 - \hat{p}_1 - \pi_0}{\sqrt{p^*(1-p^*)\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}}$$

   - let pˆ1 = x1/n1 be the sample proportion for x1 successes in n1 trials corresponding to π1, and the same quantities as pˆ2 = x2/n2 for π2.
   - $p*$, present in the denominator. This is **a pooled proportion**, given as follows:

$$p^* = \frac{x_1 + x_2}{n_1 + n_2}$$

5. Calculate p-value
   - Based on the null hypothesis, determine the critical region or calculate the p-value associated with the test statistic.

6. Make a Decision:
   - If the test statistic falls into the critical region or if the p-value is less than your chosen significance level (α), reject the null hypothesis.
   - If the test statistic is not in the critical region or if the p-value is greater than α, fail to reject the null hypothesis.

R – Function

Syntax :- prop.test(x, n, p = NULL, alternative = "two.sided", conf.level = 0.95, correct = TRUE)

Where,

- x: A numeric vector of successes (number of successes).
- n: A numeric vector of the total number of trials.
- p: A vector of the null hypothesis proportions (default is p=0.5p=0.5).
- alternative: A character string specifying the alternative hypothesis. Possible values are "two.sided", "less", or "greater".
- conf.level: The confidence level for the confidence interval.
- correct: Logical, indicating whether to apply continuity correction (default is TRUE).

Example

```
n_psychology <- 233
x_statistic_psychology <- 180
n_geography <- 197
x_statistic_geography <- 175
print(prop.test(x = c(x_statistic_geography,x_statistic_psychology),
                n=c(n_geography,n_psychology),alternative = "greater",correct = FALSE))
```

Output

```
> source("D:/ICCA/R_Program/Testing_mean.R")

        2-sample test for equality of proportions without continuity correction

data:  c(x_statistic_geography, x_statistic_psychology) out of c(n_geography, n_psychology)
X-squared = 9.9395, df = 1, p-value = 0.0008089
alternative hypothesis: greater
95 percent confidence interval:
 0.05745804 1.00000000
sample estimates:
   prop 1    prop 2
0.8883249 0.7725322
```

Testing Categorial Values

Single Categorial Variable

Chi- Square Test for Goodness-of-fit

- Hypothesis testing on a single categorical variable typically involves examining whether the distribution of observations across different categories is consistent with a particular expectation or if there is evidence of a significant difference from what is expected.
- The chi-square goodness-of-fit test is a common method used for hypothesis testing with a single categorical variable.

Hypotheses:

- Null Hypothesis (H0): The observed distribution is not significantly different from the expected distribution.
- Alternative Hypothesis (H1 or Ha): There is a significant difference between the observed and expected distributions of the categorical variable.
- The test statistic for the chi-square test for Goodness-of-fit is calculated as follows

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i},$$

- Where
  - Oi is the observed count
  - Ei is the expected count in the ith category; i = 1, , k.
- Excepted value calculation

- Equal Distribution: If you expect the categories to be equally likely, you can distribute the total observations equally among the categories. For k categories, the expected frequency for each category (Ei) would
  E_i =Total Observations/k

- Based on Proportions: If you have information on the expected proportions of each category, you can multiply these proportions by the total number of observations to get the expected frequencies.
  E_i =Proportion of Category i ×Total Observations.

Example: - Let us consider the following table given number of breakdown in a factory in various days of week. Check whether breakdowns are uniformly distributed.

Days	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Number of breakdowns	14	22	16	18	12	19	11

Step 1: - Formulate Hypothesis

H_0 : - Breakdowns are uniformly distributed

H_a: - Breakdowns are not uniformly distributed.

Step 2: - Choose a significance

α = 0.05

Step 3:- Calculate test statistic

Calculate using the formula

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i},$$

- With n-1 degree of freedom

Step 4: - Check Critical Value or P-value:

- Compare the calculated chi-square statistic with the critical value from the chi-square distribution table or use a statistical software to find the p-value.

Step 5: - Make a Decision:

- If the p-value is less than the chosen significance level (e.g., 0.05), reject the null hypothesis.
- If the chi-square statistic is greater than the critical value, reject the null hypothesis.

R - Function

Synatx: - chisq.test(x)

Where,

x: A vector or matrix of observed frequencies or a table-like object

Example:

```
breakdown <- c(14,22,16,18,12,19,11)
print(chisq.test(breakdown))
```

Output

```
> source("D:/ICCA/R_Program/Testing_mean.R")

        Chi-squared test for given probabilities

data:  breakdown
X-squared = 5.875, df = 6, p-value = 0.4373
```

Two Categorical values

- Hypothesis testing for two categorical variables typically involves analyzing the association or independence between the two variables.
- The most common test for this purpose is test for independence, a particular variant of chi-square test for two categorical values.
- Hypotheses:
  - H0 : Variables A and B are independent. (or, There is no relationship between A and B.)
  - HA : Variables A and B are not independent. (or, There is a relationship between A and B.)
- The test statistic for the chi-square test for independence is calculated as follows

$$\chi^2 = \sum_{i=1}^{k_r} \sum_{j=1}^{k_c} \frac{(O_{[i,j]} - E_{[i,j]})^2}{E_{[i,j]}},$$

- Where,
  - O[i, j] is the observed count and
  - E[i, j] is the expected count at row position i and column position j.
  - Each E[i, j] is found as the sum total of row i multiplied by the sum total of column j, all divided by N.

$$E_{[i,j]} = \frac{\left(\sum_{u=1}^{k_r} O_{[u,j]}\right) \times \left(\sum_{v=1}^{k_c} O_{[i,v]}\right)}{N}$$

Example: - Suppose you collect data from 200 individuals and classify them based on their smoking habits and lung disease status.

	Lung disease(yes)	Lung disease(no)	Total
Smoker	30	50	80
Non-Smoker	20	100	120
Total	50	150	200

Steps:

1. Define Hypotheses:
   - H0: Smoking habits and lung disease status are independent.
   - H1: Smoking habits and lung disease status are not independent.
2. Choose a significance level
   α = 0.05
3. Set up the Observed Frequencies:
   - Use the provided data to fill in the observed frequencies in the contingency table.
4. Compute Expected Frequencies:
   - Calculate the expected frequencies (Eij) for each cell assuming independence:

$$E_{[i,j]} = \frac{\left(\sum_{u=1}^{k_r} O_{[u,j]}\right) \times \left(\sum_{v=1}^{k_c} O_{[i,v]}\right)}{N}$$

5. Calculate the Chi-Square Statistic:
   - Use the chi-square formula:

$$\chi^2 = \sum_{i=1}^{k_r} \sum_{j=1}^{k_c} \frac{(O_{[i,j]} - E_{[i,j]})^2}{E_{[i,j]}},$$

   - With degree of freedom v =(kr – 1) * (kc – 1) .

6. Check Critical Value or P-value:
   - Compare the calculated chi-square statistic with the critical value from the chi-square distribution table or use a statistical software to find the p-value.
7. Make a Decision:
   - If the p-value is less than the chosen significance level (e.g., 0.05), reject the null hypothesis.
   - If the chi-square statistic is greater than the critical value, reject the null hypothesis.

R Function

Synatx: - chisq.test(x)

Where,

x: A vector or matrix of observed frequencies or a table-like object .

Example: -

```
row_name <- c("Smoker","Non-Smoker","Total")
col_name <- c("Lung disease(Yes)","Lung disease(No)","Total")
a <- matrix(c(30,50,80,20,100,120,50,150,200),nrow =3,ncol=3,
        dimnames = list(row_name,col_name),byrow = TRUE)
print(chisq.test(a))
```

Output: -

```
        Pearson's Chi-squared test

data:  a
X-squared = 11.111, df = 4, p-value = 0.02534
```

	Lung disease(Yes)	Lung disease(No)	Total
Smoker	30	50	80
Non-Smoker	20	100	120
Total	50	150	200

Error and Power

Hypothesis test Error

- Hypothesis test errors are mistakes that occur when you draw the wrong conclusion from a statistical test.
- There are two types of hypothesis test errors: Type I and Type II.
- A Type I error occurs when you incorrectly reject a true H0. In any given hypothesis test, the probability of a Type I error is equivalent to the significance level α.
- A Type II error occurs when you incorrectly retain a false H0 (in other words, fail to accept a true HA). Since this depends upon what the true HA actually is, the probability of committing such an error, labeled, is not usually known in practice.

Type I Error

- If your p-value is less than , you reject the null statement. If the null is really true, though, the directly defines the probability that you incorrectly reject it. This is referred to as a Type I error.
- Below figure provides a conceptual illustration of a Type I error probabil ity for a supposed hypothesis test of a sample mean, where the hypotheses are set up as
  - H0 : = μ =μ_0 and HA : μ > μ_0

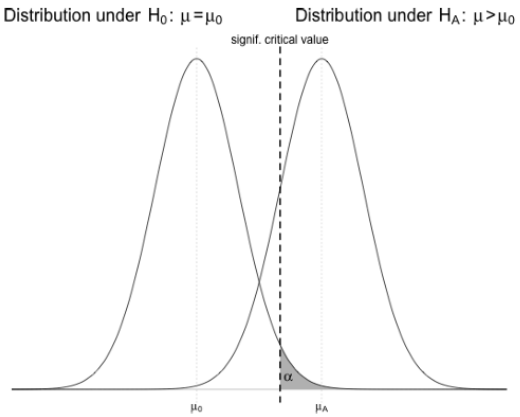


Figure 18-2: A conceptual diagram of the Type I error probability α

- The null hypothesis distribution is centered on the null value the alternative μ_0 ; the alternative hypothesis distribution is centered to its right at some mean μ_A in the above figure.
- If the null hypothesis is really true, then the probability it is incorrectly rejected for this test will be equal to the significance level , located in the upper tail of the null distribution.

Type II Error

- The issues with Type I errors might suggest that it's desirable to perform a hypothesis test with a smaller value.
- Reducing the significance level for any given test leads directly to an increase in the chance of committing a Type II error.
- A Type II error refers to incorrect retention of the null hypothesis—in other words, obtaining a p-value greater than the significance level when it's the alternative hypothesis that's actually true.
- Below figure illustrates the probability of a Type II error, shaded and denoted β.

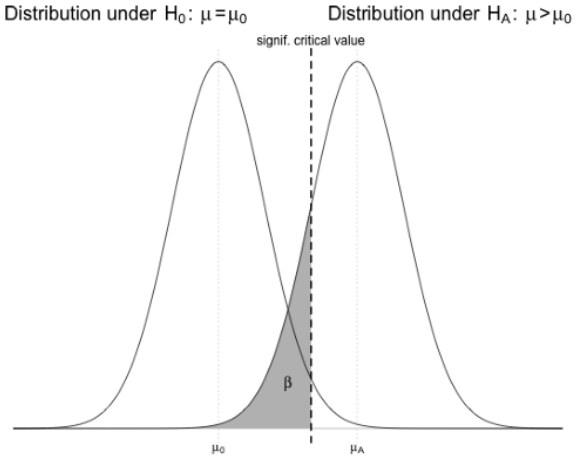


Figure 18-3: A conceptual diagram of the Type II error probability β

Statistical Power

- Power is the probability of correctly rejecting a null hypothesis that is untrue.
- For a test that has a Type II error rate of β, the statistical power is found simply with 1- β.
- It's desirable for a test to have a power that's as high as possible. The simple relationship with the Type II error probability means that all factors impacting the value of also directly affect power.
- For the same one-sided H0 : = $\mu = \mu_0$ and HA : $\mu > \mu_0$ example, below figure shades the power of the test—the complement to the Type II error rate. By convention, a hypothesis test that has a power greater than 0.8 is considered statistically powerful.

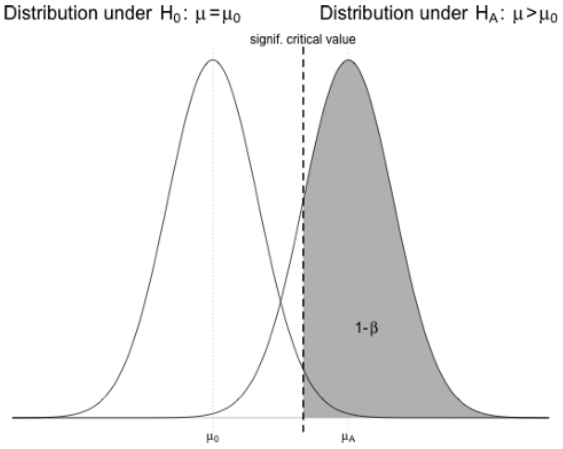


Figure 18-4: A conceptual diagram of statistical power 1 – β

Analysis of Variance

- Analysis of Variance (ANOVA) is a statistical method used to compare means among more than two groups. It assesses whether there are any statistically significant differences between the means of three or more independent (unrelated) groups.

One-Way ANOVA

- The simplest version of ANOVA is referred to as one-way or one-factor analysis.
- The one-way ANOVA is used to test two or more means for equality
- Those means are split by a categorical group or factor variable.
- ANOVA is often used to analyze experimental data to assess the impact of an intervention.
- For example, comparing the mean weights of the chicks in the built-in chickwts data set, split according to the different food types they were fed.

a) Hypotheses and Diagnostic Checking

- A categorical-nominal variable that splits a total of N numeric observations into k distinct groups, where k >2.
- Statistically compare the k groups' means,μ_1,……., μ_k , to see whether they can be claimed to be equal.
- The standard hypotheses are as follows:
  - H0 : $\mu_1 = \mu_2 =$……$= \mu_k$
  - HA : : μ_1, μ_2 ,……, μ_k are not all equal (alternatively, at least one mean differs).
- In fact, when k = 2, the two-sample t-test is equivalent to ANOVA; for that reason, ANOVA is most frequently employed when k > 2.
- The following assumptions need to be satisfied in order for the results of the basic one-way ANOVA test to be considered reliable:

- **Independence** The samples making up the k groups must be independent of one another, and the observations in each group must be independent and identically distributed (iid).
- **Normality** The observations in each group should be normally dis tributed, or at least approximately so.
- **Equality of variances** The variance of the observations in each group should be equal, or at least approximately so.

b) <u>One-Way ANOVA Table Construction</u>

- Construct a one-way ANOVA table step by step with formulas. Suppose we have k groups and n_i observations in each group.
  - <u>Definitions:</u>
    - N: Total number of observations ($N = \sum_{i=1}^{k} n_i$)
    - X_{ij}: Observation j in group i
    - $\bar{X}_i$: Mean of group i
    - $\bar{X}$: Overall mean
  - <u>Formulas</u>
    i. Between-Groups Sum of Squares (SSB):
    $$SSB = \sum_{i=1}^{k} n_i(\bar{X}_i - \bar{X})^2$$
    ii. Within-Groups Sum of Squares (SSW):
    $$SSW = \sum_{i=1}^{k} \sum_{j=1}^{n_i} (X_{ij} - \bar{X}_i)^2$$
    iii. Degrees of Freedom:
    - $df_{\text{Between}} = k - 1$ (Between-Groups df)
    - $df_{\text{Within}} = N - k$ (Within-Groups df)

    iv. Mean Squares (MS):
    - $MS_{\text{Between}} = \frac{SSB}{df_{\text{Between}}}$
    - $MS_{\text{Within}} = \frac{SSW}{df_{\text{Within}}}$

    v. F-Statistic
    $$F = \frac{MS_{\text{Between}}}{MS_{\text{Within}}}$$

  - <u>ANOVA Table</u>

Source	Sum of Squares (SS)	Degrees of Freedom (df)	Mean Squares (MS)	F-Statistic	p-Value
Between-Groups	SSB	df_{Between}	MS_{Between}	F	p-value
Within-Groups	SSW	df_{Within}	MS_{Within}	-	-
Total	$SSB + SSW$	$N - 1$	-	-	-

    - <u>Group row/Factor row</u> <u>(Between- Groups):</u> - This relates to the data in the individual groups of interest, thereby accounting for the <u>between-group variability</u>.

- Error row/Residual row (Within Groups): - This accounts for the random deviation from the estimated means of each group, thereby accounting for the within group variability.
- TOTAL row: - This represents the raw data, based on the previous three ingredients. It is used to find the Error SS by differencing

c) Building ANOVA Tables with the aov Function

- R allows you to easily construct an ANOVA table for the chick weight test using the built-in aov function as follows:

Example:

```
chick.anova <- aov(weight~feed,data=chickwts)
print(summary(chick.anova))
```

Output

```
> source("D:/ICCA/R_Program/Testing_mean.R")
            Df Sum Sq Mean Sq F value   Pr(>F)
feed         5 231129   46226   15.37 5.94e-10 ***
Residuals   65 195556    3009
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

d) Interpretation

- If the p-value associated with the F-statistic in the "Between-Groups" row is less than your chosen significance level, you reject the null hypothesis.
- A rejection suggests that there is evidence that at least one group mean is different from the others.

Two-Way ANOVA

- Two-way ANOVA is an extension of the one-way ANOVA that allows for the simultaneous comparison of the effect of two categorical independent variables (factors) on a continuous dependent variable.
- It explores how the two factors interact and affect the dependent variable.

a) Components of Two-Way ANOVA:

1. Factors: Independent variables
   These are the categorical variables that define the groups in the study. In two-way ANOVA, you have two factors.
2. Interaction Effect:
   This refers to the combined effect of the two factors on the dependent variable. It assesses whether the effect of one factor on the dependent variable is consistent across all levels of the other factor.
3. Main Effects:
   Main Effects: These are the individual effects of each factor on the dependent variable, ignoring the other factor.

b) A suite of Hypothesis

The standard hypothesis are as follows

a. Denote numeric outcome variable with O and grouping variable as G1 and G2.
b. H_0 : G1 has no main (marginal) effect on the mean of O
   G2 has no main (marginal) effect on the mean of O
   There is no interactive effect of G1 and G2 on the mean of O

   H_A : Each statement in H_0 is incorrect.

c) **aov function**

Example

```
two_way_ANOVA <- aov(mpg~cyl+gear,data=mtcars)
print(two_way_ANOVA)
```

Output

```
> source("D:/ICCA/R_Program/Testing_mean.R")
Call:
   aov(formula = mpg ~ cyl + gear, data = mtcars)

Terms:
                    cyl     gear Residuals
Sum of Squares  817.7130   5.4313  302.9029
Deg. of Freedom        1        1        29

Residual standard error: 3.231862
Estimated effects may be unbalanced
```

Kruskal-Wallis

- The Kruskal-Wallis test is a non-parametric statistical test used to determine whether there are statistically significant differences between the medians of three or more independent (unrelated) groups.
- The standard hypothesis are as follows:
  - H_0 : Group medians are all equal.
  - H_A : Group medians are not all equal (alternatively, at least one group median differs)

R Function

- The R function for the Kruskal-Wallis test is kruskal.test.
- It is used to test for differences in the medians of three or more independent groups.

Syntax

kruskal.test(formula, data)

- Where,
  - formula: A formula of the form response ~ group. This specifies the dependent variable and the grouping variable. The dependent variable should be numeric, and the grouping variable should be a factor.
  - data: An optional data frame containing the variables in the formula.

Example

```
kruskal_wallis <- kruskal.test(mpg~cyl,data=mtcars)
print(kruskal_wallis)
```


Output

```
> source("D:/ICCA/R_Program/Testing_mean.R")

        Kruskal-Wallis rank sum test

data:  mpg by cyl
Kruskal-Wallis chi-squared = 25.746, df = 2, p-value = 2.566e-06
```

Unit V

Simple Linear Regression

- Linear regression models: a suite of methods used to evaluate precisely how variables relate to each other.

1. An Example of a Linear Relationship

```
library(MASS)
cor(MASS::survey$Wr.Hnd, MASS::survey$Height, use = "complete.obs")
sum(is.na(MASS::survey$Wr.Hnd) | is.na(MASS::survey$Height))
plot(MASS::survey$Height ~ MASS::survey$Wr.Hnd, xlab = "Writing handspan (cm)",
ylab = "Height (cm)")
```

- The provided code demonstrates a bivariate relationship between student heights and writing handspans, which is a simple example of a linear relationship.
- The code snippet demonstrates the exploration of a linear relationship between two variables, which is the essence of simple linear regression. In simple linear regression, you model the relationship between a dependent variable (in this case, height) and an independent variable (writing handspan) using a linear equation.
- The plot function creates a scatter plot, with student heights (survey$Height) on the y-axis and writing handspans (survey$Wr.Hnd) on the x-axis. This plot allows you to visually inspect the relationship between the two variables.
- The cor function calculates the correlation coefficient between writing handspans and heights. In this case, the correlation coefficient of approximately 0.601 indicates a moderate positive linear association.
- The last line checks for missing values and counts the number of missing observation pairs in the dataset.

2. General concepts in Linear Regression

- The purpose of a linear regression model is to come up with a function that estimates the mean of one variable given a particular value of another variable.
- These variables are known as the response variable (the “outcome” variable whose mean you are attempting to find) and the explanatory variable (the “predictor” variable whose value you already have).

a. Definition of the Model

- Assume you’re looking to determine the value of response variable Y given the value of an explanatory variable X.
- The simple linear regression model states that the value of a response is expressed as the following equation:

$$Y|X = \beta_0 + \beta_1 X + \epsilon$$

  - The notation YX reads as “the value of Y conditional upon the value of X.

➢ Residual Assumptions

- The validity of the conclusions you can draw based on the model in is critically dependent on the assumptions made about , which are defined as follows:
  - The value of ϵ is assumed to be normally distributed such that $\epsilon \sim N(0, \sigma)$.
  - That ϵ is centered (that is, has a mean of) zero.
  - The variance of ϵ, σ^2, is constant.

The ϵ term represents random error.

- Parameters
  - The value denoted by β_0 is called the intercept, and that of β_1 is called the slope. Together, they are also referred to as the regression coefficients and are interpreted as follows:
    - The intercept, β_0, is interpreted as the expected value of the response variable when the predictor is zero.
    - Generally, the slope, β_1, is the focus of interest. This is interpreted as the change in the mean response for each one-unit increase in the predictor. When the slope is positive, the regression line increases from left to right (the mean response is higher when the predictor is higher); when the slope is negative, the line decreases from left to right (the mean response is lower when the predictor is higher). When the slope is zero, this implies that the predictor has no effect on the value of the response. The more extreme the value of 1 (that is, away from zero), the steeper the increasing or decreasing line becomes.

b. Estimating the Intercept and Slope Parameters

- The goal is to use data to estimate the regression parameters, yielding the estimates $\hat{\beta_0}$ and $\hat{\beta_1}$; this is referred to as fitting the linear model.
- In this case, the data comprise n pairs of observations for each individual.
- The fit ted model of interest concerns the mean response value, denoted ˆy, for a specific value of the predictor, x, and is written as follows:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

- Alternative notations such as E[Y] or E[Y|X=x] may be used on the left side of the model equation to emphasize that the model gives the mean (expected value) of the response.
- For compactness, many simply use something like ˆy, as shown here.
- Let your n observed data pairs be denoted x_i and y_i for the predictor and response variables, respectively; i = 1, ..., n.
- Then, the parameter estimates for the simple linear regression function are.

$$\hat{\beta}_1 = \rho_{xy}\frac{s_y}{s_x} \quad \text{and} \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x}$$

- Where
  - $\bar{x}$ and $\bar{y}$ are the sample means of the x_is and y_is.
  - s_x and s_y are the sample standard deviations of the x_is and y_is.
  - p_{xy} is the estimate of correlation between X and Y based on the data
- Estimating the model parameters in this way is referred to as least-squares regression

c. Fitting Linear Models with lm

- The lm function in R is used to perform linear regression.
  - survfit <- lm(Height ~ Wr.Hnd, data = survey)
- The above line creates a fitted linear model object of the mean student height by handspan and stores it in survfit variable.
- The result of the linear regression analysis is stored in the object survfit, which is of class "lm". In R, an object of class "lm" is essentially a list containing several components that describe the linear regression model.
  - print(survfit)

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = Height ~ Wr.Hnd, data = MASS::survey)

Coefficients:
(Intercept)       Wr.Hnd
    113.954        3.117
```

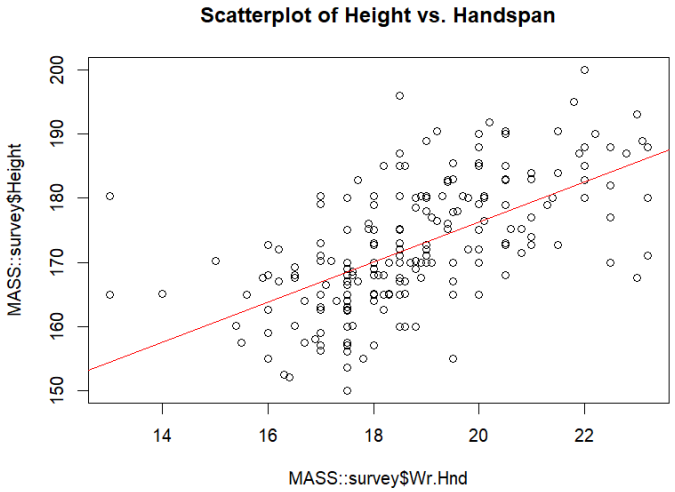
- This reveals that the linear model for this example is estimated as follows:
  ˆ y =113.954+3.117x.
- Intercept (β^0): The expected value of the response variable when the predictor is zero.
- Slope (β^1): The change in the mean response for each one-unit increase in the predictor. In this example, for every 1 cm increase in handspan, a student's height is estimated to increase by 3.117 cm.
- If evaluate the mathematical function for ˆy, at a range of different values for x, we end up with a straight line plotting the results.
- Considering the definition of intercept, as the expected value of the response variable when the predictor is zero, in the current example, this would imply that the mean height of a student with a handspan of 0 cm is 113.954 cm
- The slope, the change in the mean response for each one-unit increase in the predictor, is 3.117.
- This states that, on average, for every 1 cm increase in handspan, a student's height is estimated to increase by 3.117 cm
- To visualize the fitted regression line along with the raw data, the abline function is used.
- When passed an object of class "lm" (like survfit), abline adds the fitted regression line to an existing plot.

Example

```
library(MASS)
survfit <- lm(Height ~ Wr.Hnd, data = MASS::survey)
# print(survfit)
plot(MASS::survey$Wr.Hnd, MASS::survey$Height, main = "Scatterplot of Height vs. Handspan")
abline(survfit, col = "red")  # Adds the fitted regression line in red
```

Output

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = Height ~ Wr.Hnd, data = MASS::survey)

Coefficients:
(Intercept)       Wr.Hnd
    113.954        3.117
```



d. Illustrating Residuals

- When the parameters are estimated the fitted line is referred to as an implementation of least-squares regression because it's the line that minimizes the average squared difference between the observed data and itself.
- This concept is easier to understand by drawing the distances between the observations and the fitted line, formally called residuals.
- First, let's extract two specific records from the Wr.Hnd and Height data vectors and call the resulting vectors obsA and obsB.

```
library(MASS)
survfit <- lm(Height ~ Wr.Hnd, data = MASS::survey)
print(survfit)
obsA <- c(MASS::survey$Wr.Hnd[197],MASS::survey$Height[197])
print(obsA)
obsB <- c(MASS::survey$Wr.Hnd[154],MASS::survey$Height[154])
print(obsB)
```

```
> source("D:/ICCA/R_Program/datavisual.R")
[1]  15.00 170.18
[1]  21.50 172.72
```

- Next, briefly inspect the names of the members of the survfit object.

```
> names(survfit)
 [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "na.action"     "xlevels"       "call"          "terms"
[13] "model"
```

- These members are the components that automatically make up a fitted model object of class "lm".
- The component called "coefficients", contains a numeric vector of the estimates of the intercept and slope.
- We can extract this component in the same way we would perform a member reference on a named list: by entering survfit$coefficients at the prompt.
- Where possible, though, it's technically preferable for programming purposes to extract such components using a "direct-access" function.
- For the coefficients component of an "lm" object, the function you use is coef.

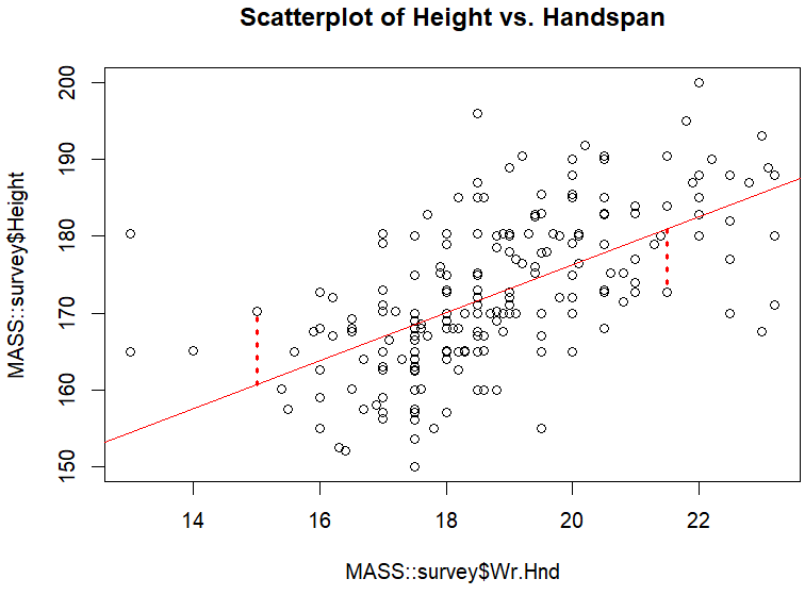
```
mycof <- coef(survfit)
print(mycof)
beta0.hat <- mycof[1]
beta1.hat <- mycof[2]
```

- The coef function is then used to extract the coefficients, and they are assigned to the objects beta0.hat and beta1.hat

```
> source("D:/ICCA/R_Program/datavisual.R")
(Intercept)       Wr.Hnd
 113.953623     3.116617
```

- Finally, used segments to draw the vertical dashed lines

```
plot(MASS::survey$Wr.Hnd, MASS::survey$Height, main = "Scatterplot of
Height vs. Handspan")
abline(survfit, col = "red")
segments(x0=c(obsA[1],obsB[1]),y0=beta0.hat+beta1.hat*c(obsA[1],obsB[1]),
x1=c(obsA[1],obsB[1]),y1=c(obsA[2],obsB[2]),lty=3,col="red",lwd = 3)
```



- Note that the dashed lines meet the fitted line at the vertical axis locations passed to y_0, which, with the use of the regression coefficients beta0.hat and beta1.hat

Example

```
library(MASS)
survfit <- lm(Height ~ Wr.Hnd, data = MASS::survey)
print(survfit)
obsA <- c(MASS::survey$Wr.Hnd[197],MASS::survey$Height[197])
print(obsA)
mycof <- coef(survfit)
print(mycof)
beta0.hat <- mycof[1]
beta1.hat <- mycof[2]
plot(MASS::survey$Wr.Hnd, MASS::survey$Height, main = "Scatterplot of Height vs.
Handspan")
abline(survfit, col = "red")
segments(x0=c(obsA[1],obsB[1]),y0=beta0.hat+beta1.hat*c(obsA[1],obsB[1]),
        x1=c(obsA[1],obsB[1]),y1=c(obsA[2],obsB[2]),lty=3,col="red",lwd = 3)
```


3. Statistical Inference

- The process of estimating a regression equation is described as relatively straightforward. This involves finding the best-fitting line that represents the relationship between the predictor and response variables.

- Estimation is just the beginning, and the focus shifts to what can be inferred from the results. In other words, the goal is to interpret the statistical significance of the estimated relationship.
- The central question in simple linear regression is whether there is statistical evidence to support the presence of a relationship between the predictor and the response. This involves exploring whether changes in the explanatory variable affect the mean outcome.

a. Summarizing the Fitted Model

- In R, model-based inference is automatically conducted when dealing with linear model (lm) objects.
- Using the summary function on an lm object provides more detailed output compared to simply printing the object to the console.
- The focus is on two aspects of the information presented in the summary:
    - Significance Tests for Regression Coefficients:
        - The summary function provides significance tests associated with the regression coefficients. These tests help assess the statistical significance of each coefficient, indicating whether they significantly differ from zero.
    - Coefficient of Determination (R-squared):
        - The summary output includes the interpretation of the coefficient of determination, labeled as R-squared. This statistic is a measure of how well the regression model explains the variability in the response variable. A higher R-squared value suggests a better fit of the model to the data.

Example

```
library(MASS)
survfit <- lm(Height ~ Wr.Hnd, data = MASS::survey)
print(summary(survfit))
```

Output

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = Height ~ Wr.Hnd, data = MASS::survey)

Residuals:
     Min       1Q   Median       3Q      Max
-19.7276  -5.0706  -0.8269   4.9473  25.8704

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 113.9536     5.4416   20.94   <2e-16 ***
Wr.Hnd        3.1166     0.2888   10.79   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 7.909 on 206 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3612,    Adjusted R-squared:  0.3581
F-statistic: 116.5 on 1 and 206 DF,  p-value: < 2.2e-16
```

b. Regression Coefficient Significance Tests

- The coefficients table in the summary output contains point estimates of the intercept and slope. The intercept is labeled explicitly, and the slope is labeled after the name of the predictor variable.

- Standard errors of these estimates are also provided in the table.
- Simple linear regression coefficients, estimated using least-squares, follow a t-distribution with degrees of freedom equal to n−2, where n is the number of observations in the model fit.
- Standardized t-values and p-values are reported for each parameter.
- These represent the results of a two-tailed hypothesis test formally defined as
  H0 : $\beta_j = 0$
  HA : $\beta_j \neq 0$
- The null hypothesis (H0) is defined as j=0 for the intercept (j=0) and j=1 for the slope (j=1).
- The alternative hypothesis (HA) is two-sided, indicating interest in whether there is any effect of the covariate, not its specific direction.
- A small p-value suggests strong evidence against H0.
- In the given example, a p-value < 2×10^{-16} is reported.
- For the predictor variable (in this case, handspan), a small p-value suggests strong evidence against the claim that the predictor has no effect on the mean level of the response.
- The test for the slope parameter is typically more interesting, as it assesses whether an increase in the predictor variable is associated with a change in the response variable.
- The results suggest evidence that an increase in handspan is associated with an increase in height among the population being studied.
- The average increase in height for each additional centimeter of handspan is approximately 3.12 cm.
- Confidence intervals for the estimates can be obtained using the confint function in R.

```
> confint(survfit,level=0.95)
                2.5 %     97.5 %
(Intercept) 103.225178 124.682069
Wr.Hnd        2.547273   3.685961
```

- The example shows a 95% confidence interval for the slope parameter (Wr.Hnd) with lower and upper bounds.

c. Coefficient of Determination

- The output of summary also provides the values of Multiple R-squared and Adjusted R-squared, which are particularly interesting.
- Both of these are referred to as the coefficient of determination; they describe the proportion of the variation in the response that can be attributed to the predictor.
- For simple linear regression, the first (unadjusted) measure is simply obtained as the square of the estimated correlation coefficient.
- For the student height example, first store the estimated correlation between Wr.Hnd and Height as rho.xy, and then square it:

  Example

```
library(MASS)
survfit <- lm(Height ~ Wr.Hnd, data = MASS::survey)
rho.xy <- cor(MASS::survey$Wr.Hnd,MASS::survey$Height,use="complete.obs")
print(rho.xy^2)
```

  Output

```
> source("D:/ICCA/R_Program/datavisual.R")
[1] 0.3611901
```

- The result is same as the Multiple R-squared value

- This tells you that about 36.1 percent of the variation in the student heights can be attributed to handspan.

d. Other summary output

- The summary of the model object provides, even more useful information.
- The "residual standard error" is the estimated standard error of the term;
- Below residual standard error it also reports any missing values.
- The output also provides a five-number summary for the residual distances.
- At last, provided with a certain hypothesis test performed using the F-distribution. This is a global test of the impact of your predictor(s) on the response.
- We can access all the output provided by summary directly, as individual R objects, rather than having to read them off the screen from the entire printed summary.

4. Prediction

- The next step in concluding the initial discussions on linear regression involves utilizing the fitted model for predictive purposes.
- Fitting a statistical model helps understand and quantify relationships in data (e.g., a 3.1166 cm height increase per 1 cm handspan increase).
- Fitted models enable predicting outcome values even for unobserved explanatory variable values.
- Any point estimates or predictions from the model need to be accompanied by a measure of spread.

a. Confidence Interval or Prediction Interval?

- With a fitted simple linear model, you can calculate a point estimate for the mean response value based on a given explanatory variable value by plugging it into the fitted model equation.
- Due to inherent variation, similar to sample statistics, a confidence interval for the mean response (CI) is used to quantify this uncertainty.
- The 100(1−α) percent confidence interval for the mean response at a specific x is calculated using the formula:

$$\hat{y} \pm t_{(1-\alpha/2,n-2)} s_\epsilon \sqrt{\frac{1}{n} + \frac{(x-\bar{x})^2}{(n-1)s_x^2}}$$

- Where y^ is the fitted value, $t_{(1-2\alpha,n-2)}$ is the critical value from a t-distribution, s is the estimated residual standard error, $\bar{x}$ is the sample mean, and s_x^2 is the variance of the predictor observations.
- A prediction interval (PI) for an observed response, distinct from a confidence interval, provides the potential range of values for an individual realization of the response variable at a given x.
- The 100(1−α) percent prediction interval for an individual response at a specific x is calculated using a formula similar to the confidence interval but with an additional 1+1/n in the square root.
- Notably, a prediction interval is wider than a confidence interval, emphasizing the distinction between describing the variability of the mean response (CI) and providing a possible range for an individual observation (PI).

b. Interpreting Intervals

- To determine the mean height for students with a handspan of 14.5 cm and for students with a handspan of 24 cm.
- The point estimates themselves are easy—just plug the desired x values into the regression equation

```
library(MASS)
survfit <- lm(Height ~ Wr.Hnd, data = MASS::survey)
mycof <- coef(survfit)
beta0.hat <- mycof[1]
beta1.hat <- mycof[2]
print(as.numeric(beta0.hat+beta1.hat*14.5))
print(as.numeric(beta0.hat+beta1.hat*24))
```

```
> source("D:/ICCA/R_Program/datavisual.R")
[1] 159.1446
[1] 188.7524
```

- According to the model, you can expect mean heights to be around 159.14 and 188.75 cm for handspans of 14.5 and 24 cm, respectively.
- The as.numeric coercion function is used simply to strip the result of the annotative names that are otherwise present from the beta0.hat and beta1.hat objects.

i. Confidence Intervals for Mean Heights

- To find confidence intervals for these estimates, you could calculate them using built-in predict command.
- To use predict, you first need to store your x values in a particular way: as a column in a new data frame. The name of the column must match the predictor used in the original call to create the fitted model object.

```
xvals <- data.frame(Wr.Hnd=c(14.5,24))
print(xvals)
```

```
> source("D:/ICCA/R_Program/datavisual.R")
  Wr.Hnd
1   14.5
2   24.0
```

- Now, when predict is called, the first argument must be the fitted model object of interest, survfit for this example.

- Next, in the argument new data, you pass the specially constructed data frame containing the specified predictor values.
- To the interval argument you must specify "confidence" as a character string value. The confidence level, here set for 95 percent, is passed (on the scale of a probability) to level.

```
mypred.ci <- predict(survfit,newdata=xvals,interval="confidence",level=0.95)
print(mypred.ci)
```

```
> source("D:/ICCA/R_Program/datavisual.R")
       fit      lwr      upr
1 159.1446 156.4956 161.7936
2 188.7524 185.5726 191.9323
```

- This call will return a matrix with three columns, whose number (and order) of rows correspond to the predictor values you supplied in the newdata data frame.

- The first column, with a heading of fit, is the point estimate on the regression line; you can see that these numbers match the values you worked out earlier.
- The other columns provide the lower and upper CI lim its as the lwr and upr columns, respectively.

Prediction Intervals for Individual Observations

- The predict function will also provide your prediction intervals.
- To find the prediction interval for possible individual observations with a certain probability, you simply need to change the interval argument to "prediction"

```
mypred.pi <- predict(survfit,newdata=xvals,interval="prediction",level=0.95)
print(mypred.pi)
```

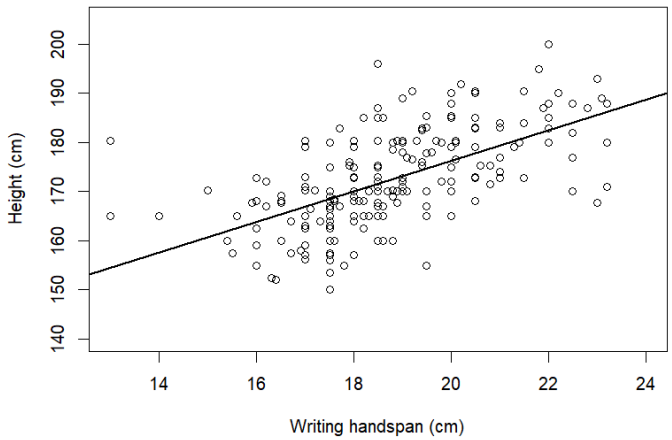
```
> source("D:/ICCA/R_Program/datavisual.R")
       fit      lwr      upr
1 159.1446 143.3286 174.9605
2 188.7524 172.8390 204.6659
```

- The fitted values remain the same.
- The widths of the PIs, however, are significantly larger than those of the corresponding CIs this is because raw observations themselves, at a specific x value, will naturally be more variable than their mean. Interpretation changes accordingly.
- The intervals describe where raw student heights are predicted to lie “95 percent of the time.
- ” For a handspan of 14.5 cm, the model predicts individual observations to lie somewhere between 143.3 cm and 175.0 cm with a probability of 0.95; for a handspan of 24 cm, the same PI is estimated at 172.8 cm and 204.7 cm (when rounded to 1 d.p.).

c. Plotting Intervals

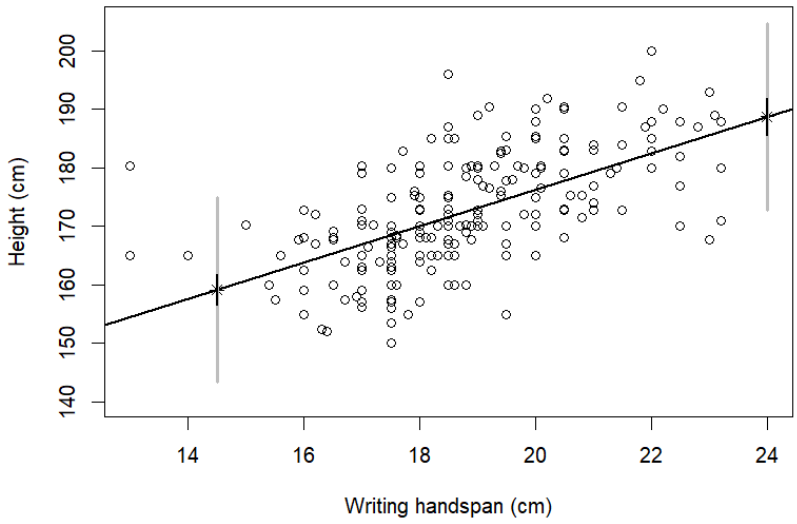
- Both CIs and PIs are well suited to visualization for simple linear regression models.

```
plot(MASS::survey$Height~MASS::survey$Wr.Hnd,xlim=c(13,24),ylim=c(
140,205),
 xlab="Writing handspan (cm)",ylab="Height (cm)")
abline(survfit,lwd=2)
```

- To this add the locations of the fitted values for x = 145 and x = 24, as well as two sets of vertical lines showing the CIs and PIs.

```
points(xvals[,1],mypred.ci[,1],pch=8)
segments(x0=c(14.5,24),y0=c(mypred.pi[1,2],mypred.pi[2,2]),
        x1=c(14.5,24),y1=c(mypred.pi[1,3],mypred.pi[2,3]),col="gray",lwd=3)
segments(x0=c(14.5,24),y0=c(mypred.ci[1,2],mypred.ci[2,2]),
        x1=c(14.5,24),y1=c(mypred.ci[1,3],mypred.ci[2,3]),lwd=2)
```



- The call to points marks the fitted values for these two particular values of x. The first call to segments lays down the PIs as thickened vertical gray lines, and the second lays down the CIs as the shorter vertical black lines. The coordinates for these plotted line segments are taken directly from the mypred.pi and mypred.ci objects, respectively.

d. Interpolation vs. Extrapolation

- Interpolation refers to making predictions when the specified x value falls within the range of observed data.
- Extrapolation occurs when the x value of interest lies outside the range of observed data.

- In the context of point predictions, x=145 is an example of interpolation, as it falls within the observed data range.
- On the other hand, x=24 is an example of extrapolation, as it lies outside the range of observed data.
- In general, interpolation is preferred over extrapolation. It is more reasonable to use a fitted model for predictions in the vicinity of observed data.
- While interpolation is generally preferable, extrapolation may still be considered reliable if it is not too far outside the observed data range. The reliability depends on the scale and context.
- The case of predicting student height at x=24 in the student height example is an extrapolation. While it is outside the range of observed data, it is not significantly so in terms of scale. The estimated intervals for the expected value visually appear reasonable given the distribution of other observations.
- Extreme extrapolation, such as predicting student height at a handspan of 50 cm, is cautioned against. Using the fitted model for predictions far beyond the observed data range may lead to unreliable results.

5. Understanding Categorical Predictors

- It's also possible to use a discrete or categorical explanatory variable, made up of k distinct groups or levels, to model the mean response.

a. Binary Variables: k = 2

- In the below equation, where the regression model is specified as

$$Y|X = \beta_0 + \beta_1 X + \epsilon$$

for a response variable Y and predictor X, and

$$\epsilon \sim N(0, \sigma^2)$$

- In the case of a binary categorical predictor with two levels (coded as 0 or 1), the interpretation of the model parameters β0 and β1 is different from the conventional intercept and slope interpretation.
- β0: This represents the baseline or reference value of the response variable (Y) when the predictor variable (X) is 0. In other words, if X=0, then Y=β0.
- β1: This represents the additive effect on the mean response if X=1. In other words, when the predictor variable is 1, the mean response becomes β0+β1.
- So, the interpretation is more like having two intercepts. If the predictor variable is 0, the response is at the baseline β0, and if the predictor variable is 1, the response is at β0+β1.
- In terms of estimation, you would estimate the parameters β^0 and β^1 from your data. The predicted mean response (y^) would be given by:
  y^=β^0+β^1x
- This represents the estimated mean response for a given value of the binary predictor x.

Linear Regression Model of Binary Variables

- To assess whether there is statistical evidence of a difference in height between males and females, you can use a simple linear regression model.
- In this context, the response variable is the height of students, and the predictor variable is the categorical variable "Sex," which has two levels: Female and Male.
- Here's how you can fit a linear regression model using the lm function in R:

```
library(MASS)
survfit2 <- lm(Height~Sex,data=MASS::survey)
print(summary(survfit2))
```



```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = Height ~ Sex, data = MASS::survey)

Residuals:
    Min      1Q  Median      3Q     Max
-23.886  -5.667   1.174   4.358  21.174

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  165.687      0.730  226.98   <2e-16 ***
SexMale       13.139      1.022   12.85   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.372 on 206 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.4449,    Adjusted R-squared:  0.4422
F-statistic: 165.1 on 1 and 206 DF,  p-value: < 2.2e-16
```

- In the context of a linear regression model with a factor predictor like "Sex," the reporting of coefficients is slightly different.
- The estimate of (intercept) is labeled as (Intercept), representing the estimated mean height if a student is female.
- The estimate of β1 for the "Sex Male" coefficient corresponds to the estimated difference in mean height for a male student.
- The regression equation is then expressed as:

  y^=β^0+β^1x
- In this case, x is defined as "the individual is male," taking the value 0 for female and 1 for male.
- The model assumes the level "female" as the reference, and the effect of "being male" on mean height is explicitly estimated.
- The hypothesis tests for β0 and β1 are defined as follows:

  H0:βj=0

  HA:βj≠0
- The test for β1 is usually of more interest, as it determines whether there is statistical evidence that the mean response variable (height) is affected by the explanatory variable (sex). In other words, it assesses whether β1 is significantly different from zero, indicating a significant impact of being male on the mean height.

Predictions from a Binary Categorical Variable

- In the context of a binary categorical predictor like "Sex," making predictions is straightforward because there are only two possible values (0 or 1).
- When evaluating the regression equation, the decision is simply whether to use the estimated coefficient β^1 (if an individual is male) or not (if an individual is female).
- To demonstrate this, you can create extra observations with the same level names as the original data and then use the predict function to find the mean heights for these new values of the predictor:

```
extra.obs <- factor(c("Female","Male","Male","Male","Female"))
print(predict(survfit2,newdata=data.frame(Sex=extra.obs),interval="confidenc
e",
      level=0.9))
```

```
> source("D:/ICCA/R_Program/datavisual.R")
       fit      lwr      upr
1 165.6867 164.4806 166.8928
2 178.8260 177.6429 180.0092
3 178.8260 177.6429 180.0092
4 178.8260 177.6429 180.0092
5 165.6867 164.4806 166.8928
```

- The output shows that predictions are different only between the two sets of values. The point estimates for instances of "Female" are identical, representing β^0 with 90 percent confidence intervals.
- Similarly, the point estimates and confidence intervals for instances of "Male" are also the same as each other, based on a point estimate of β^0+β^1.

b. Multilevel Variables: k > 2

- It's common to work with data where the categorical predictor variables have more than two levels so that (k > 2).
- These can also be referred to as multilevel categorical variables.
- To deal with this more complicated situation while retaining interpretability of your parameters, we must first dummy code the predictor into k 1 binary variables.

Dummy Coding Multilevel Variables

- For a categorical variable X with k>2 levels, dummy coding involves creating k−1 binary variables, denoted as X(1),X(2),…,X(k−1).
- Each X(i) represents a binary variable for the i-th level of the original categorical variable.
- If an individual falls into the i-th level of the original variable, X(i) takes the value 1, and all other X(j) variables take the value 0.
- If X can take values 1, 2, 3, and 4, the dummy coding results in X(1),X(2),X(3),X(4) binary variables.
- The table provided (Table 20-1) illustrates the dummy coding for six observations of a categorical variable with k=4 groups.

Table 20-1: Illustrative Example of Dummy Coding for Six Observations of a Categorical Variable with $k = 4$ Groups

X	$X_{(1)}$	$X_{(2)}$	$X_{(3)}$	$X_{(4)}$
1	1	0	0	0
2	0	1	0	0
2	0	1	0	0
1	1	0	0	0
4	0	0	0	1
3	0	0	1	0

- In the subsequent regression model, usually, only k−1 dummy variables are used. One of these variables acts as the reference or baseline level and is incorporated into the overall intercept of the model.
- The model is represented as y^=β^0+β^1X(2)+β^2X(3)+…+β^k−1X(k−1), assuming the first level is the reference level.
- The estimated model includes β^0 as the overall intercept and β^1,β^2,…,β^k−1 as coefficients associated with the dummy variables.
- The choice of the reference level affects the interpretation of the coefficients. The coefficients modify the baseline coefficient β^0 depending on the original category of the observation.
- The predicted mean value of the response is a combination of the overall intercept and the specific coefficients associated with the dummy variables.
- Dummy coding is necessary because categorical variables with more than two levels cannot be treated in the same numeric sense as continuous variables in regression.
- It avoids inappropriate assumptions about numeric relationships between different categories.
- The specific values of the estimated coefficients may change with the choice of the reference level, but overall interpretations remain consistent.

Linear Regression Model of Multilevel Variables

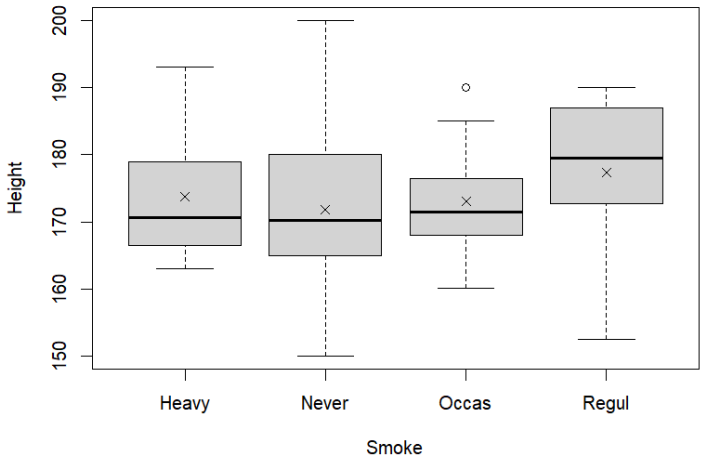
- There are two things you should check before fitting your model
  1. Thecategorical variable of interest should be stored as a (formally unordered) factor.
  2. Youshould check that you're happy with the category assigned as the reference level.
- You can check if a variable is a factor using the is.factor() function.

```
library(MASS)
print(is.factor(MASS::survey$Smoke))
print(table(MASS::survey$Smoke))
```

```
[1] TRUE

Heavy Never Occas Regul
   11   189    19    17
```

- The result from is.factor(survey$Smoke) indicates indeed have a factor vector at hand, the call to table yields the number of students in each of the four categories.
- The objective is to investigate whether there is statistical evidence to support a difference in mean student height depending on smoking frequency.
- Create a set of boxplots of these data with the following two lines

```
library(MASS)
boxplot(Height~Smoke,data=MASS::survey)
points(1:4,tapply(MASS::survey$Height,MASS::survey$Smoke,mean,na.rm=
TRUE),pch=4)
```



- By default, when a factor is created, the levels are arranged in alphabetical order unless explicitly specified otherwise.
- When using a factor as a predictor in a linear model (e.g., with the lm function), R automatically designates the first level of the factor as the reference level.
- In the example, the factor is named "Smoke," and the levels are presumably "Heavy," "Never," "Occasional," and "Regular." If not explicitly defined, R would order these alphabetically, and "Heavy" becomes the first level.
- The linear model is fitted using the lm function, incorporating the factor "Smoke" as a predictor.
- The output from a subsequent call to summary provides information about the fitted model, including details about the predictor variables.
- When examining the summary output, it is noted that the first level of "Smoke," which is "Heavy," has been used as the reference level in the linear model.

```
library(MASS)
survfit3 <- lm(Height~Smoke,data=MASS::survey)
print(summary(survfit3))
```

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = Height ~ Smoke, data = MASS::survey)

Residuals:
   Min     1Q Median     3Q    Max
-25.02  -6.82  -1.64   8.18  28.18

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 173.7720     3.1028  56.005   <2e-16 ***
SmokeNever   -1.9520     3.1933  -0.611    0.542
SmokeOccas   -0.7433     3.9553  -0.188    0.851
SmokeRegul    3.6451     4.0625   0.897    0.371
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.812 on 205 degrees of freedom
  (28 observations deleted due to missingness)
Multiple R-squared:  0.02153,   Adjusted R-squared:  0.007214
F-statistic: 1.504 on 3 and 205 DF,  p-value: 0.2147
```

Predictions from a Multilevel Categorical Variable

- Point estimates are obtained through prediction using the predict function in R.

```
library(MASS)
survfit3 <- lm(Height~Smoke,data=MASS::survey)
one.of.each <- factor(levels(MASS::survey$Smoke))
print( predict(survfit3,newdata=data.frame(Smoke=one.of.each),
        interval="confidence",level=0.95))
```

```
> source("D:/ICCA/R_Program/datavisual.R")
       fit      lwr      upr
1 173.7720 167.6545 179.8895
2 171.8200 170.3319 173.3081
3 173.0287 168.1924 177.8651
4 177.4171 172.2469 182.5874
```


c. Changing the Reference Level

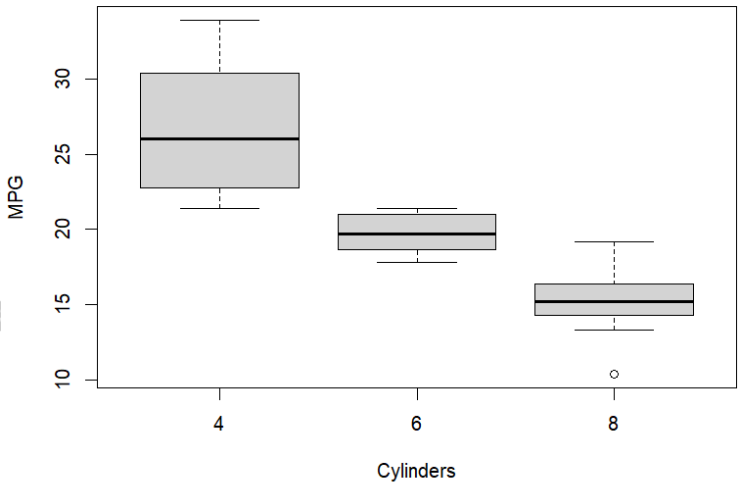
- The default reference level is automatically chosen in R based on alphabetical order, but you might decide to change it for interpretative reasons.
- Changing the reference level doesn't affect the overall significance of the factor but results in the estimation of different coefficients.
- The main reason for changing the reference level is to provide a more intuitively natural baseline for interpretation. This is particularly relevant when there's a specific category that serves as a natural reference point
- The relevel function in R allows for quick and easy reordering of factor levels to redefine the reference level.
- Example: If you want to set "Never" as the reference level in the "Smoke" variable, you can use SmokeReordered <- relevel(survey$Smoke, ref="Never").
- The levels() function can then be used to confirm the new order.
- The relevel function in R allows for quick and easy reordering of factor levels to redefine the reference level.
- Example: If you want to set "Never" as the reference level in the "Smoke" variable, you can use SmokeReordered <- relevel(survey$Smoke, ref="Never").

- The levels() function can then be used to confirm the new order.

d. Treating Categorical Variables as Numeric

- The way in which lm decides to define the parameters of the fitted model depends primarily on the kind of data you pass to the function.
- The lm imposes dummy coding only if the explanatory variable is an unordered factor vector.
- Sometimes the categorical data wanted to analyze haven't been stored as a factor.
- If the categorical variable is a character vector, lm will implicitly coerce it into a factor.
- If, however, the intended categorical variable is numeric, then lm performs linear regression exactly as if it were a continuous numeric predictor; it estimates a single regression coefficient, which is interpreted as a "per-one-unit-change" in the mean response.
- This may seem inappropriate if the original explanatory variable is supposed to be made up of distinct groups. In some settings, however, especially when the variable can be naturally treated as numeric-discrete, this treatment is not only valid statistically but also helps with interpretation.
- For example consider the mtcars dataset with variables mpg (continuous) and cyl (discrete; 4, 6, or 8 cylinders).
- The text suggests treating cyl as a categorical variable, and mpg as the response variable.
- box plots are well suited to reflect the grouped nature of cyl as a predictor

```
boxplot(mtcars$mpg~mtcars$cyl,xlab="Cylinders",ylab="MPG")
```



- When fitting the associated regression model, must be aware of what you're instructing R to do.
- Since the cyl column of mtcars is numeric, and not a factor vector per se, lm will treat it as continuous if you just directly access the data frame.

```
carfit <- lm(mpg~cyl,data=mtcars)
print(summary(carfit))
```

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = mpg ~ cyl, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-4.9814 -2.1185  0.2217  1.0717  7.5186

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  37.8846     2.0738   18.27  < 2e-16 ***
cyl          -2.8758     0.3224   -8.92 6.11e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.206 on 30 degrees of freedom
Multiple R-squared:  0.7262,    Adjusted R-squared:  0.7171
F-statistic: 79.56 on 1 and 30 DF,  p-value: 6.113e-10
```

e. Equivalence with One-Way ANOVA

- Regression models with a single nominal categorical predictor aim to describe the mean response value for different groups.
- This task is similar to one-way ANOVA, where the goal is to compare means across multiple groups and determine if at least one mean is different from the others.
- Simple linear regression with a single categorical predictor, implemented using least-squares estimation, is considered another way to perform one-way ANOVA.
- Alternatively, ANOVA is described as a special case of least-squares regression
- The p-value reported at the end of the summary of an lm object for a single categorical predictor is equivalent to the p-value obtained from a one-way ANOVA test.
- This p-value is referred to as the "overall" or "global" significance test, indicating whether there is evidence against the null hypothesis that group means are equal.
- An example is provided using the aov function in R to perform one-way ANOVA on the student height modeled by smoking status.
- The output from aov includes an F-statistic and associated p-value, similar to the output from the lm summary.
- The global test provided by lm, known as the omnibus F-test, serves as a generalization of ANOVA.
- While equivalent to one-way ANOVA in the context of a single categorical predictor, the omnibus F-test is a useful overall test assessing the combined statistical contribution of several predictors to the outcome.
- The residual standard error given in the lm summary is equivalent to the square root of the Mean Square Error (MSE) obtained from the ANOVA output.
- Least-squares regression models offer more than just coefficient-specific tests; they provide a broader understanding of the statistical contribution of predictors to the outcome.
- The omnibus F-test is not only a confirmation of ANOVA results but also serves as a stand-alone test for the statistical contribution of predictors

Multiple Linear Regression

1. Terminology

- A lurking variable influences the response, another predictor, or both, but goes unmeasured (or is not included) in a predictive model. For example, say a researcher establishes a link between the volume of trash thrown out by a household and whether the household owns a trampoline. The potential lurking variable here would

be the number of children in the household—this variable is more likely to be positively associated with an increase in trash and chances of owning a trampoline. An interpretation that suggests owning a trampoline is a cause of increased waste would be erroneous.

- The presence of a lurking variable can lead to spurious conclusions about causal relationships between the response and the other predictors, or it can mask a true cause-and-effect association; this kind of error is referred to as confounding.
- A nuisance or extraneous variable is a predictor of secondary or no interest that has the potential to confound relationships between other variables and so affect your estimates of the other regression coefficients. Extraneous variables are included in the modeling as a matter of necessity, but the specific nature of their influence on the response is not the primary interest of the analysis.

2. Theory

- Here, we'll look at how the models work in a mathematical sense and get a glimpse of the calculations that happen "behind the scenes" when estimating the model parameters in R

a. Extending the Simple Model to a MultipleModel

- The general framework of multiple linear regression, where the goal is to predict the value of a continuous response variable Y based on the values of multiple independent explanatory variables $X_1,X_2,...,X_p$.
- The multiple linear regression model is defined as:

  $$Y=\beta_0+\beta_1X_1+\beta_2X_2+...+\beta_pX_p+\epsilon$$

- Here:
  - Y is the response variable.
  - $\beta_0,\beta_1,...,\beta_p$ are the regression coefficients.
  - $X_1,X_2,...,X_p$ are the independent explanatory variables.
  - ϵ represents the error term, assumed to be normally distributed with mean 0.
- The goal is to estimate the regression coefficients $\beta_0,\beta_1,...,\beta_p$) such that the predicted values (Y^) are as close as possible to the actual observed values.
- In the context of least-squares estimation for multiple linear regression, the text mentions finding the values of $\beta_0,\beta_1,...,\beta_p$ that minimize the sum of squared differences between the observed response values (yi) and the predicted values ($y^_i$):

  $$\sum_{i=1}^{n}(y_i-(\beta_0+\beta_1x_{1i}+\beta_2x_{2i}+...+\beta_px_{pi}))^2$$

- Where:
  - n is the number of data records.
  - y_i is the observed response value for the i-th record.
  - x_{ji} is the observed value of the j-th explanatory variable for the i-th record.
- The goal is to find the values of $\beta_0,\beta_1,...,\beta_p$ that minimize this sum, which is achieved through various optimization techniques. Once these values are estimated, they can be used to make predictions for new data based on the multiple linear regression model.

b. Estimating in Matrix Form

- The computations involved in minimizing the squared distance are made much easier by a matrix representation of the data. When dealing with n multivariate observations, write Equation as follows,

  $$Y=X\beta+\epsilon$$

- Where Yand ϵ denote nx1 column matrices such that

$$\boldsymbol{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \text{and} \quad \boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

- Here,y_i and ϵ_i refer to the response observation and random error term for the i^{th} individual. The quantity β is a (p+1)*1 column matrix of the regression coefficients, and then the observed predictor data for all individuals and explanatory variables are stored in a n * (p+1) matrix X, called the design matrix:

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} \quad \text{and} \quad \boldsymbol{X} = \begin{bmatrix} 1 & x_{1,1} & \dots & x_{p,1} \\ 1 & x_{1,2} & \dots & x_{p,2} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{1,n} & \dots & x_{p,n} \end{bmatrix}$$

- The minimization of above matrix equations providing the estimated regression coefficient values is then found with the following calculation:

$$\hat{\boldsymbol{\beta}} = \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_p \end{bmatrix} = (\boldsymbol{X}^\top \cdot \boldsymbol{X})^{-1} \cdot \boldsymbol{X}^\top \cdot \boldsymbol{Y}$$

c. A Basic Example

- We can manually estimate the β j (j=0,1,….,p)in R using the functions %*%(matrix multiplication), t(matrix transposition), and solve(matrix inversion). As a quick demonstration, let's consider two predictor variables: X1as continuous and X2 as binary. target regression equation is therefore $y\hat{} = \beta\hat{}_0 + \beta\hat{}_{1x1} + \beta\hat{}_{2x2}$.

- Suppose we collect the following data, where the response data, data for X1,and data for X2,for n=8 individuals, are given in the columns y, x1,and x2,respectively.

```
demo.data<-data.frame(y=c(1.55,0.42,1.29,0.73,0.76,-1.09,1.41,-0.32),
x1=c(1.13,-0.73,0.12,0.52,-0.54,-1.15,0.20,-1.09),
x2=c(1,0,1,1,0,1,0,1))
print(demo.data)
```

```
> source("D:/ICCA/R_Program/datavisual.R")
     y    x1 x2
1  1.55  1.13  1
2  0.42 -0.73  0
3  1.29  0.12  1
4  0.73  0.52  1
5  0.76 -0.54  0
6 -1.09 -1.15  1
7  1.41  0.20  0
8 -0.32 -1.09  1
```

- To get your point estimates in = [β_0,β_1,β_2] for the linear model, you first have to construct X and Y as required by the following equation

$$\hat{\boldsymbol{\beta}} = \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_p \end{bmatrix} = (\boldsymbol{X}^\top \cdot \boldsymbol{X})^{-1} \cdot \boldsymbol{X}^\top \cdot \boldsymbol{Y}$$

```
Y<-matrix(demo.data$y)
n<-nrow(demo.data)
X<-matrix(c(rep(1,n),demo.data$x1,demo.data$x2),nrow=n,ncol=3)
print(X)
```

```
> source("D:/ICCA/R_Program/datavisual.R")
     [,1]  [,2] [,3]
[1,]    1  1.13    1
[2,]    1 -0.73    0
[3,]    1  0.12    1
[4,]    1  0.52    1
[5,]    1 -0.54    0
[6,]    1 -1.15    1
[7,]    1  0.20    0
[8,]    1 -1.09    1
```

- Now execute the below line corresponding to above equation

```
BETA.HAT<-solve(t(X)
print(BETA.HAT)
```

```
            [,1]
[1,]  1.2254572
[2,]  1.0153004
[3,] -0.6980189
```

- We've used least-squares to fit model based on the observed data in demo.data, which results in the estimates $\hat{\beta}_0$=1.225, $\hat{\beta}_1$=1.015, and $\hat{\beta}_2$ = -0.698.

3. Implementing in R and Interpreting

- When it comes to output and interpretation, working with multiple explanatory variables follows the same rules as you've seen in Chapter 20. Any numeric-continuous variables (or a categorical variable being treated as such) have a slope coefficient that provides a "per-unit-change" quantity. Any k-group categorical variables (factors, formally unordered) are dummy coded and provide k 1 intercepts.

a. Additional Predictors

- Multiple linear regression is a technique that can be used to analyze the relationship between one response variable and two or more predictor variables.
- With the response variable on the left as usual, you specify the multiple predictors on the right side of the ~ symbol; altogether this represents the formula argument.
- To fit a model with several main effects, use + to separate any variables you want to include

- For example consider the survey data set in the MASS package contains data on student height, handspan, and sex, among other variables.

Example 1

```
library(MASS)
survmult <- lm(Height~Wr.Hnd+Sex,data=MASS::survey)
print(summary(survmult))
```

Output

```
Call:
lm(formula = Height ~ Wr.Hnd + Sex, data = MASS::survey)

Residuals:
     Min       1Q   Median       3Q      Max
-17.7479  -4.1830   0.7749   4.6665  21.9253

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 137.6870     5.7131  24.100  < 2e-16 ***
Wr.Hnd        1.5944     0.3229   4.937 1.64e-06 ***
SexMale       9.4898     1.2287   7.724 5.00e-13 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 6.987 on 204 degrees of freedom
  (30 observations deleted due to missingness)
Multiple R-squared:  0.5062,    Adjusted R-squared:  0.5014
F-statistic: 104.6 on 2 and 204 DF,  p-value: < 2.2e-16
```

- Simple linear regression models were fitted using height as the response variable and handspan or sex as the predictor variable.
- The results showed that both handspan and sex had a significant effect on height, with positive coefficients indicating that higher handspan and being male were associated with higher height.
- The parameter estimates of a multiple linear regression model can be interpreted as the expected change in the response variable for a one-unit change in the predictor variable, holding all other predictors constant.
- The mean height when compared to the mean for females (the category used as the reference level)

Note:- You can continue to add explanatory variables in the same way if you need to do so.

b. Interpreting Marginal Effects

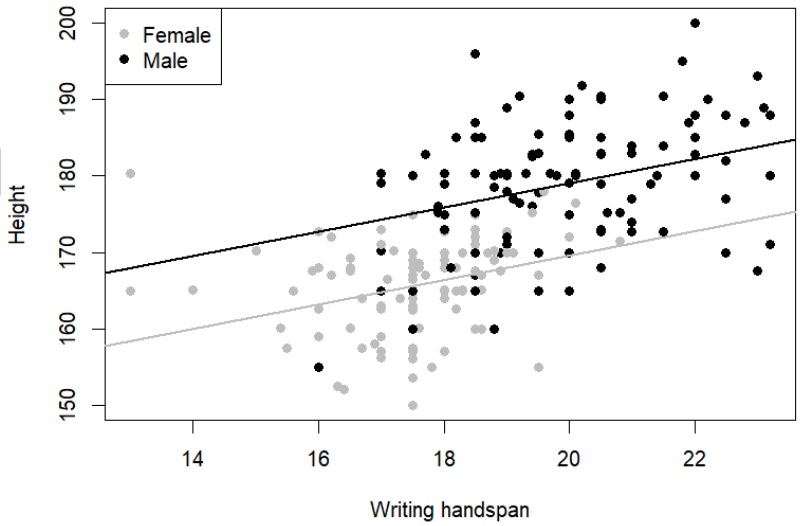
- In multiple regression, each predictor's coefficient is estimated while considering the effects of all other predictors in the model.
- The coefficient for a specific predictor (e.g., Z) is interpreted as the change in the mean response for a one-unit increase in Z, while holding all other predictors constant.
- Consider the example of the model (survmult) that includes only the explanatory variables of sex and handspan.
- For students of the same sex:
  - A 1 cm increase in handspan leads to an estimated increase of 15.944 cm in mean height.
- For students of similar handspan:
  - Males, on average, will be 94.898 cm taller than females.
- Highlights the difference in the values of the estimated predictor coefficients in survmult compared to their respective simple linear model fits.

- Indicates that both coefficients in survmult still show evidence against the null hypothesis of being zero, suggesting the presence of confounding.
- Emphasizes the general usefulness of multiple regression in addressing confounding effects.
- Indicates that using only single predictor models could lead to misleading interpretations, and the determination of the "true" impact of each explanatory variable is clearer in a multivariate context.
- Note that the coefficient of determination for the survmult model is noticeably higher than in the single-variate models.
- Indicates that using multiple regression accounts for more variation in the response.
- Presents the fitted model equation, expressing mean height as a function of handspan and sex.
- Example equation: Mean height = 137.687 + 15.94 * "handspan" + 94.9 * "sex."

c. Visualizing the Multiple Linear Model

- Visualizing the observed data and fitted multiple linear model of student height modeled by handspan and sex.

```
library(MASS)
survmult <- lm(Height~Wr.Hnd+Sex,data=MASS::survey)
print(summary(survmult))
survcoefs <- coef(survmult)
plot(MASS::survey$Height~MASS::survey$Wr.Hnd,
    col=c("gray","black")[as.numeric(MASS::survey$Sex)],
    pch=16,xlab="Writing handspan",ylab="Height")
abline(a=survcoefs[1],b=survcoefs[2],col="gray",lwd=2)
abline(a=survcoefs[1]+survcoefs[3],b=survcoefs[2],col="black",lwd=2)
legend("topleft",legend=levels(MASS::survey$Sex),col=c("gray","black"),pch=
16)
```



d. Finding Confidence Intervals

- We can easily find confidence intervals for any of the regression parameters in multiple regression models with confint.

```
library(MASS)
survmult <- lm(Height~Wr.Hnd+Sex,data=MASS::survey)
print(confint(survmult))
```

```
> source("D:/ICCA/R_Program/datavisual.R")
                 2.5 %     97.5 %
(Intercept) 126.4226293 148.951272
Wr.Hnd        0.9577186   2.231173
SexMale       7.0673172  11.912311
```

e. Omnibus F-Test

- The omnibus F-test more generally for multiple regression models as a test with the following hypothese.

$$H_0 : \beta_1 = \beta_2 = \ldots = \beta_p = 0$$

$$H_A : \text{At least one of the } \beta_j \neq 0 \text{ (for } j = 1, \ldots, p)$$

- The F-test helps boil all that down, allowing you to conclude either of the following:
  1. Evidence against H0 if the associated p-value is smaller than your chosen significance level ,which suggests that your regression—your combi nation of the explanatory variables—does a significantly better job of predicting the response than if you removed all those predictors.
  2. No evidence against H0 if the associated p-value is larger than, which suggests that using the predictors has no tangible benefit over having an intercept alone.
- The F-test statistic is computed using the coefficient of determination (R-squared), the number of regression parameters (p), and the number of observations (n).
- The formula is: F= **R2·(n−p−1) /(1−R2)·p**
- The F-test statistic follows an F distribution with degrees of freedom df1 = p and df2 = n - p - 1.
- A small p-value indicates that the full model explains significantly more variation in the response variable than the null model, and thus the predictors are useful.
- A large p-value indicates that the full model does not explain much more variation in the response variable than the null model, and thus the predictors are not useful.
- The test does not tell which predictors are significant or what their coefficients are, so further analysis is needed to interpret the effects of each predictor.

f. Predicting from a Multiple Linear Model

- Prediction (or forecasting) for multiple regression follows the same rules as for simple regression.
- It's important to remember that point predictions found for a particular covariate profile the collection of predictor values for a given individual—are associated with the mean (or expected value) of the response; that confidence intervals provide measures for mean responses; and that prediction intervals provide measures for raw observations
- Have to consider the issue of interpolation (predictions based on x values that fall within the range of the originally observed covariate data) versus extrapolation (prediction from x values that fall outside the range of said data).
- The R syntax for predict is identical to that used in Simple Linear Regression

Example

```
library(MASS)
survmult <- lm(Height~Wr.Hnd+Sex,data=MASS::survey)
print(predict(survmult,newdata=data.frame(Wr.Hnd=16.5,Sex="Male"),
      interval="confidence",level=0.95))
```

Output

```
> source("D:/ICCA/R_Program/datavisual.R")
       fit      lwr      upr
1 173.4851 170.9419 176.0283
```

4. Transforming Numeric Variables

- Numeric transformation refers to the application of a mathematical func tion to your numeric observations in order to rescale them. Finding the square root of a number and converting a temperature from Fahrenheit to Celsius are both examples of a numeric transformation.
- The two most common approaches: polynomial and logarithmic transformations.

a. Polynomial

let's observe a curved relationship in our data such that a straight line isn't a sensible choice for modeling it.

In an effort to fit our data more closely, a polynomial or power transformation can be applied to a specific predictor variable in your regression model.

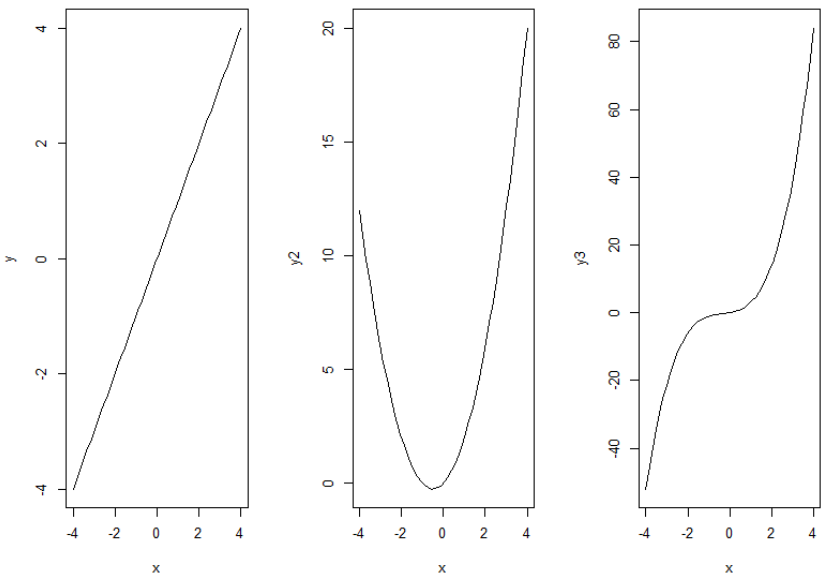
This is a straightforward technique that, by allowing polynomial curvature in the relationships, allows changes in that predictor to influence the response in more complex ways than otherwise possible. We can achieve this by including additional terms in the model definition that represent the impact of progressively higher powers of the variable of interest on the response.

Let us consider the example the following sequence between 4 and 4, as well as the simple vectors computed from it:

Example

```
# Define the sequence of x values
x <- seq(-4, 4, length = 50)
# Generate a vector y (linear relationship)
y <- x
# Generate a vector y2 (quadratic relationship)
y2 <- x + x^2
# Generate a vector y3 (cubic relationship)
y3 <- x + x^2 + x^3
# Plot the three vectors separately
par(mfrow=c(1,3))
plot(x, y, type = "l")   # Linear plot
plot(x, y2, type = "l")  # Quadratic plot
plot(x, y3, type = "l")  # Cubic plot
par(mfrow=c(1,1))
```

Output



- The sequence of x values is created using seq() from -4 to 4, with 50 points.
- The vector y is assigned as a linear relationship with x.
- The vector y2 is assigned to have a quadratic relationship with x by adding the squared value of x.
- The vector y3 is assigned to have a cubic relationship with x by adding the cubed value of x.
- The subsequent three plot() functions create separate line plots for each vector, illustrating the different polynomial relationships:
- The first plot (plot(x, y, type = "l")) shows a straight line, representing a linear relationship (polynomial of order 1).
- The second plot (plot(x, y2, type = "l")) depicts a curve, indicating a quadratic relationship (polynomial of order 2).
- The third plot (plot(x, y3, type = "l")) shows a more pronounced curve, representing a cubic relationship (polynomial of order 3).
- These plots visually demonstrate how the inclusion of higher-order terms in a regression model allows for more flexible modeling of the relationships between predictor variables and the response. Polynomial curvature provides a way to capture non-linear patterns in the data that cannot be adequately represented by a simple straight line.

Fitting a Polynomial Transformation

- The below provided R code demonstrates the fitting of multiple regression models to the relationship between engine displacement (disp) and miles per gallon (mpg) in the mtcars dataset.

1. **Simple Linear Model:**

Example

```
car.order1 <- lm(mpg ~ disp, data = mtcars)
print(summary(car.order1))
```

Output

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = mpg ~ disp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-4.8922 -2.2022 -0.9631  1.6272  7.2305

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 29.599855   1.229720  24.070  < 2e-16 ***
disp        -0.041215   0.004712  -8.747 9.38e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.251 on 30 degrees of freedom
Multiple R-squared:  0.7183,    Adjusted R-squared:  0.709
F-statistic: 76.51 on 1 and 30 DF,  p-value: 9.38e-10
```

- The simple linear model (car.order1) suggests a negative linear impact of displacement on mileage.
- The summary output provides information about the coefficients, including the interpretation that, for each additional cubic inch of displacement, the mean response decreases by about 0.041 miles per gallon.

2. **Quadratic Model:**

Example

```
car.order2 <- lm(mpg ~ disp + I(disp^2), data = mtcars)
```

```
print(summary(car.order2))
```

Output

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = mpg ~ disp + I(disp^2), data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9112 -1.5269 -0.3124  1.3489  5.3946

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.583e+01  2.209e+00  16.221 4.39e-16 ***
disp        -1.053e-01  2.028e-02  -5.192 1.49e-05 ***
I(disp^2)    1.255e-04  3.891e-05   3.226   0.0031 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.837 on 29 degrees of freedom
Multiple R-squared:  0.7927,    Adjusted R-squared:  0.7784
F-statistic: 55.46 on 2 and 29 DF,  p-value: 1.229e-10
```

- Use of the I function around a given term in the formula is necessary when said term requires an arithmetic calculation.
- The quadratic model (car.order2) includes a quadratic term (disp^2) to capture the apparent curve in the data.
- The output indicates that the quadratic component is statistically significant (p-value of 0.0031), implying that the model with a quadratic component is a better-fitting model.
- The coefficient of determination (R-squared) is noticeably higher than the simple linear model.

3. **Cubic Model:**

Example

```
car.order3 <- lm(mpg ~ disp + I(disp^2) + I(disp^3), data = mtcars)
print(summary(car.order3))
```

Output

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = mpg ~ disp + I(disp^2) + I(disp^3), data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.0896 -1.5653 -0.3619  1.4368  4.7617

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.070e+01  3.809e+00  13.310 1.25e-13 ***
disp        -3.372e-01  5.526e-02  -6.102 1.39e-06 ***
I(disp^2)    1.109e-03  2.265e-04   4.897 3.68e-05 ***
I(disp^3)   -1.217e-06  2.776e-07  -4.382  0.00015 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.224 on 28 degrees of freedom
Multiple R-squared:  0.8771,    Adjusted R-squared:  0.8639
F-statistic: 66.58 on 3 and 28 DF,  p-value: 7.347e-13
```

- The cubic model (car.order3) adds a cubic term (disp^3) to the model. The output shows that the cubic component also offers a statistically significant contribution. However, adding higher-order terms beyond cubic does not improve the fit.

❖ The interpretation of the fitted multiple regression model (car.order3) is presented in terms of the coefficients for each term, providing a mathematical expression for the relationship between engine displacement and miles per gallon.
❖ The process of fitting models with increasing polynomial orders allows for capturing more complex relationships in the data. However, it's essential to strike a balance between model complexity and goodness of fit, as adding too many higher-order terms can lead to overfitting and may not generalize well to new data.

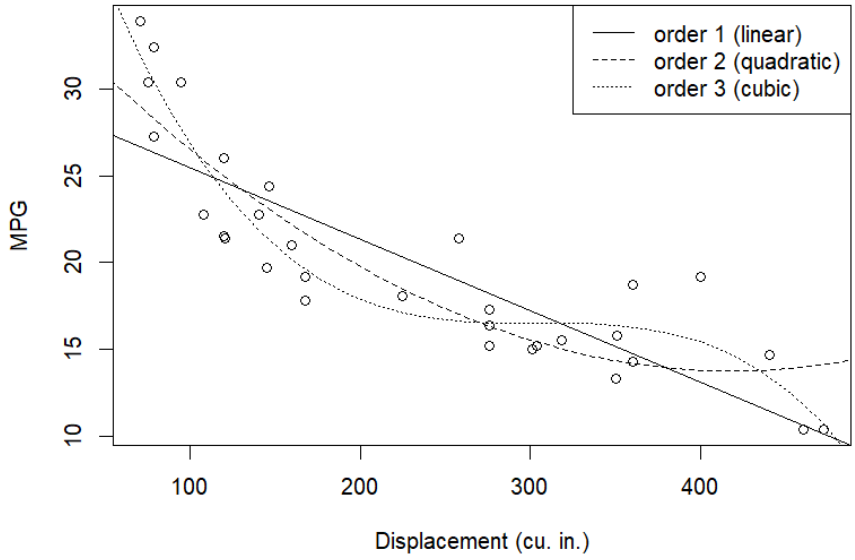
Plotting the Polynomial Fit

- To visualize the different models use the plot function

Example

```
plot(mtcars$disp,mtcars$mpg,xlab="Displacement (cu. in.)",ylab="MPG")
abline(car.order1)
disp.seq <- seq(min(mtcars$disp)-50,max(mtcars$disp)+50,length=30)
car.order2.pred <- predict(car.order2,newdata=data.frame(disp=disp.seq))
lines(disp.seq,car.order2.pred,lty=2)
car.order3.pred <- predict(car.order3,newdata=data.frame(disp=disp.seq))
lines(disp.seq,car.order3.pred,lty=3)
legend("topright",lty=1:3,
        legend=c("order 1 (linear)","order 2 (quadratic)","order 3 (cubic)"))
```

Output



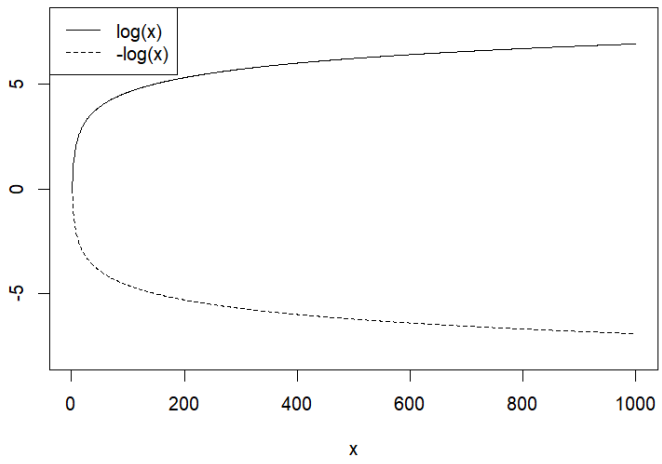
Pitfalls of Polynomials

- Pitfall of using polynomial terms in linear regression models is the instability of the fitted trend, especially when performing extrapolation.
- Extrapolation involves making predictions outside the range of the observed data, and polynomial models, particularly those of higher orders, can lead to unreliable predictions in such scenarios.

b. Logarithmic

- In statistical modeling situations where you have positive numeric observations, it's common to perform a log transformation of the data.
- Log transformations are employed to dramatically reduce the overall range of the data. This is beneficial for visualizing patterns, identifying relationships, and improving the interpretability of the data.
- The transformation brings extreme observations closer to a measure of centrality. This is crucial for handling outliers and extreme values, making the data more amenable to statistical analysis.
- Log transformations are particularly effective in reducing the severity of heavily skewed data. Skewed data distributions can negatively impact the assumptions of many statistical models, and log transformations help mitigate this issue.
- In the context of regression modeling, log transformations can be used to capture trends in the data. This is especially relevant when dealing with relationships that exhibit apparent curves that "flatten off" at higher values.
- Unlike some higher-degree polynomials, log transformations provide a way to capture curvature without introducing the same kind of instability outside the range of the observed data. This helps in creating more stable regression models.
- To briefly illustrate the typical behavior of the log transformation consider the below graph

```
plot(1:1000, log(1:1000), type="l", xlab="x", ylab="", ylim=c(-8,8))
lines(1:1000, -log(1:1000), lty=2)
legend("topleft", legend=c("log(x)", "-log(x)"), lty=c(1,2))
```

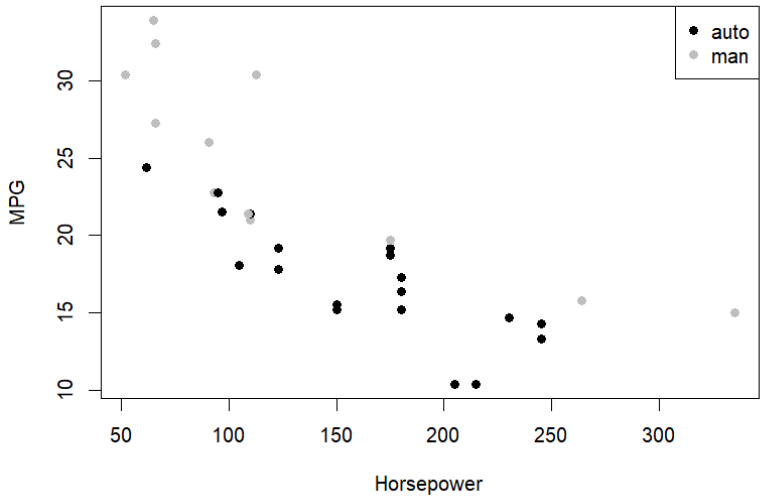


- This plots the log of the integers 1 to 1000 against the raw values, as well as plotting the negative log.
- We can see the way in which the log transformed values taper off and flatten out as the raw values increase.

Fitting the Log Transformation

- The below R code demonstrates the use of a log transformation in multiple linear regression to capture curved relationships between predictor variables and the response variable (mileage, represented by mpg), while also considering the effect of another predictor variable (am, representing transmission type).

```
plot(mtcars$hp, mtcars$mpg, pch=19, col=c("black",
"gray")[factor(mtcars$am)],
    xlab="Horsepower", ylab="MPG")
legend("topright", legend=c("auto", "man"), col=c("black", "gray"), pch=19)
```



- Above code creates a scatterplot of mileage (mpg) against horsepower (hp). Different colors are used to distinguish between automatic (auto) and manual (man) transmission types.
- The scatterplot serves as an exploratory visualization to observe the relationship between horsepower and mileage, considering the transmission type.
- Let's fit a linear model with a log transformation of horsepower and the transmission type as predictor variables:

```
# Fit linear model with log transformation of horsepower and transmission type
car.log <- lm(mpg ~ log(hp) + am, data=mtcars)
# Display summary of the linear model
print(summary(car.log))
```

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = mpg ~ log(hp) + am, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9084 -1.7692 -0.1432  1.4032  6.3865

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  63.4842     5.2697  12.047 8.24e-13 ***
log(hp)      -9.2383     1.0439  -8.850 9.78e-10 ***
am            4.2025     0.9942   4.227 0.000215 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.592 on 29 degrees of freedom
Multiple R-squared:  0.827,     Adjusted R-squared:  0.8151
F-statistic: 69.31 on 2 and 29 DF,  p-value: 8.949e-12
```

- This code uses the lm function to fit a linear model (car.log) with mpg as the response variable and both the natural log of hp and am as predictor variables.
- The summary function provides detailed information about the fitted model.
- The log transformation of horsepower (log(hp)) is used to allow for curved relationships in situations where a straight line might not be appropriate.

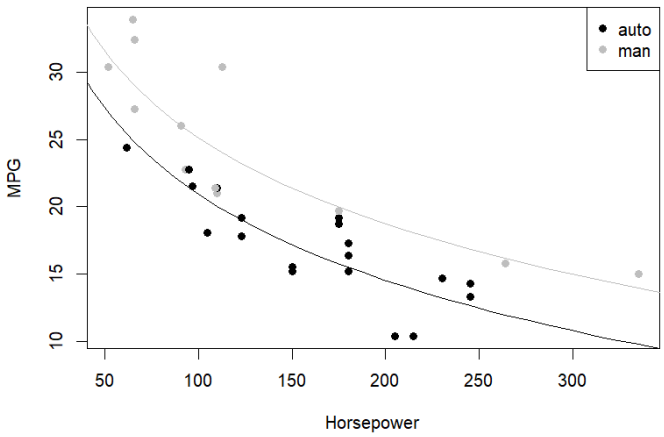
- Interpreting the output of summary(car.log) would give insights into the coefficients, significance levels, and other statistics associated with the model, helping to assess the relationships between the predictors and the response variable.

Plotting the Log Transformation Fit

- To visualize the fitted model, you first need to calculate the fitted values for all desired predictor values.
- The following code creates a sequence of horse power values (minus and plus 20 horsepower) and performs the required prediction for both transmission types.

```
hp.seq <- seq(min(mtcars$hp) - 20, max(mtcars$hp) + 20, length=30)
n <- length(hp.seq)
# Predict values for both transmission types using the fitted model
car.log.pred <- predict(car.log, newdata=data.frame(hp=rep(hp.seq, 2),
am=rep(c(0,1), each=n)))
# Add lines to the scatterplot
lines(hp.seq, car.log.pred[1:n])
lines(hp.seq, car.log.pred[(n+1):(2*n)], col="gray")
```

- hp.seq <- seq(min(mtcars$hp) - 20, max(mtcars$hp) + 20, length=30):
  - This line creates a sequence of 30 horsepower values ranging from 20 horsepower below the minimum observed value to 20 horsepower above the maximum observed value.
- car.log.pred <- predict(car.log, newdata=data.frame(hp=rep(hp.seq, 2), am=rep(c(0,1), each=n))):
  - This line uses the predict function to calculate predicted values for the specified horsepower values and both transmission types.
  - The newdata argument is constructed using a data frame with replicated hp.seq values for both transmission types (0 for automatic, 1 for manual).
- lines(hp.seq, car.log.pred[1:n]):
  - Adds a line to the scatterplot for the predicted values corresponding to automatic transmission.
- lines(hp.seq, car.log.pred[(n+1):(2*n)], col="gray"):
  - Adds a gray line for the predicted values corresponding to manual transmission.
- This code adds lines to the scatterplot created earlier, illustrating the fitted model's predictions for both automatic and manual transmission types. The color distinction helps visualize the impact of transmission type on the relationship between horsepower and mileage.



5. Interactive Terms

- So far, we've looked only at the joint main effects of how predictors affect the outcome variable (and one-to-one transformations thereof).
- Now we'll look at interactions between covariates.
- An interactive effect between predictors is an additional change to the response that occurs at particular combi nations of the predictors.

a. Concept and Motivation

- Diagrams below are often used to help explain the concept of interactive effects. These diagrams show your mean response value, ˆy, on the vertical axis, as usual, and a predictor value for the variable x1 on the horizontal axis. They also show a binary categorical variable x2, which can be either zero or one. These hypothetical variables are labeled as such in the images.

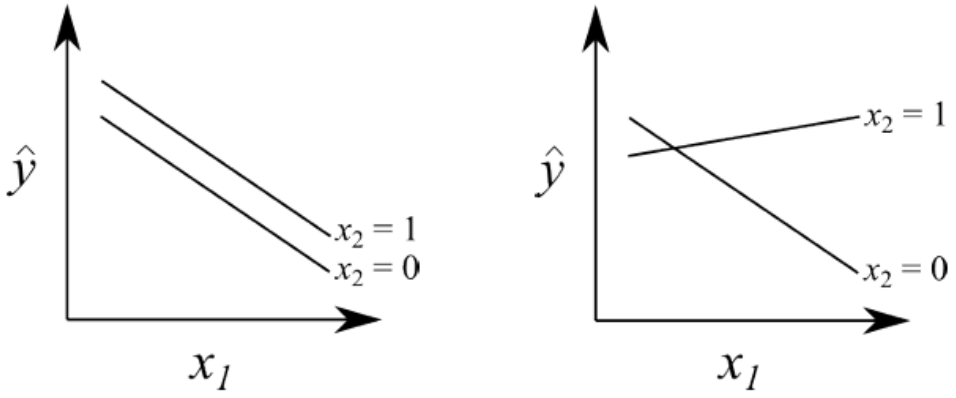


Figure 21-7: Concept of an interactive effect between two predictors x_1 and x_2, on the mean response value $\hat{y}$. Left: Only main effects of x_1 and x_2 influence $\hat{y}$. Right: An interaction between x_1 and x_2 is needed in addition to their main effects in order to model $\hat{y}$.

b. One Categorical, One Continuous

- Generally, a two-way interaction between a categorical and a continuous predictor should be understood as effecting a change in the slope of the continuous predictor with respect to the nonreference levels of the categorical predictor.
- In the presence of a term for the continuous variable, a categorical variable with k levels will have k-1 main effect terms, so there will be a further k-1 interactive terms between all the alternative levels of the categorical variable and the continuous variable.
- The different slopes for x1 by category of x2 for ˆy can be seen clearly on the above right figure.
- In such a situation, in addition to the main effects for x1 and x2, there would be one interactive term in the fitted model corresponding to x2 = 1.
- This defines the additive term needed to change the slope in x1 for x2 = 0 to the new slope in x1 for x2 = 1.
- Let's proceed with fitting a multiple linear regression model on the diabetes dataset from the faraway package.
- In this example, we'll model the total cholesterol level (chol) by considering the age of the individual (age) and the body frame type (frame).
- Additionally, we'll include a two-way interaction between age and body frame to explore whether the effect of age on cholesterol varies based on the body frame.

Example

```
library(faraway)
dia.fit <- lm(chol~age+frame+age:frame,data=diabetes)
print(summary(dia.fit))
```

Output

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = chol ~ age + frame + age:frame, data = diabetes)

Residuals:
    Min      1Q  Median      3Q     Max
-131.90  -26.24   -5.33   22.17  226.11

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     155.9636    12.0697  12.922  < 2e-16 ***
age               0.9852     0.2687   3.667  0.00028 ***
framemedium      28.6051    15.5503   1.840  0.06661 .
framelarge       44.9474    18.9842   2.368  0.01840 *
age:framemedium  -0.3514     0.3370  -1.043  0.29768
age:framelarge   -0.8511     0.3779  -2.252  0.02490 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 42.34 on 384 degrees of freedom
  (13 observations deleted due to missingness)
Multiple R-squared:  0.07891,   Adjusted R-squared:  0.06692
F-statistic:  6.58 on 5 and 384 DF,  p-value: 6.849e-06
```

- Inspecting the estimated model parameters in the output, you can see a main effect coefficient for age, main effect coefficients for the two levels of frame (that aren’t the reference level), and two further terms for the interactive effect of age with those same nonreference levels.

c. Two Categorical

- A linear regression model is another statistical method that allows you to describe the relationship between a continuous response variable and one or more explanatory variables, which can be either continuous or categorical. You can also include interaction terms in a linear regression model to capture the interaction effect.
- The effect of one variable on the response variable depends on the value of the other variable. For example, the effect of wool type on the number of warp breaks may differ depending on the tension level.
- The following R code implements a linear regression model to analyze the interactive effect of two categorical explanatory variables, namely wool and tension, on the response variable breaks (mean number of warp breaks in lengths of yarn).

```
# Fit a linear regression model with interaction terms
warp.fit <- lm(breaks ~ wool * tension, data = warpbreaks)
# Display summary of the linear regression model
print(summary(warp.fit))
```

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = breaks ~ wool * tension, data = warpbreaks)

Residuals:
     Min       1Q   Median       3Q      Max
-19.5556  -6.8889  -0.6667   7.1944  25.4444

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   44.556      3.647  12.218 2.43e-16 ***
woolB        -16.333      5.157  -3.167 0.002677 **
tensionM     -20.556      5.157  -3.986 0.000228 ***
tensionH     -20.000      5.157  -3.878 0.000320 ***
woolB:tensionM  21.111      7.294   2.895 0.005698 **
woolB:tensionH  10.556      7.294   1.447 0.154327
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.94 on 48 degrees of freedom
Multiple R-squared:  0.3778,    Adjusted R-squared:  0.3129
F-statistic: 5.828 on 5 and 48 DF,  p-value: 0.0002772
```

- Here the cross-factor symbol * is used, rather than wool + tension + wool:tension.
- When both predictors in a two-way interaction are categorical, there will be a term for each nonreference level of the first predictor combined with all nonreference levels of the second predictor.
- In this example, wool is binary with only k = 2 levels and tension has k = 3; therefore, the only interaction terms present are the "medium" (M) and "high" (H) tension levels ("low", L, is the reference level) with wool type B (A is the reference level).
- Therefore, altogether in the fitted model, there are terms for B, M, H, B:M, and B:H.
- These results provide the same conclusion as the ANOVA analysis— there is indeed statistical evidence of an interactive effect between wool type and tension on mean breaks, on top of the contributing main effects of those predictors.

d. Two Continuous

- In the case of two continuous predictors, an interaction term allows for a modification of the slope associated with one variable based on the value of the other continuous variable.
- The interaction term captures the joint influence of the two continuous predictors and allows for a more nuanced modeling of their relationship.
- Unlike an interaction between a continuous and a categorical predictor, where the modification is based on distinct categories, an interaction between two continuous variables allows for a continuous modification. This means that the effect of one continuous predictor on the response variable can vary continuously based on the values of the other continuous predictor.
- Let's consider an example using the mtcars data frame, considering MPG as a function of horsepower and weight.
- The fitted model includes an interaction term in addition to the main effects of the two continuous predictors (horsepower and weight).

Example

```
car.fit <- lm(mpg~hp*wt,data=mtcars)
print(summary(car.fit))
```

```
> source("D:/ICCA/R_Program/datavisual.R")

Call:
lm(formula = mpg ~ hp * wt, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.0632 -1.6491 -0.7362  1.4211  4.5513

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 49.80842    3.60516  13.816 5.01e-14 ***
hp          -0.12010    0.02470  -4.863 4.04e-05 ***
wt          -8.21662    1.26971  -6.471 5.20e-07 ***
hp:wt        0.02785    0.00742   3.753 0.000811 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.153 on 28 degrees of freedom
Multiple R-squared:  0.8848,    Adjusted R-squared:  0.8724
F-statistic: 71.66 on 3 and 28 DF,  p-value: 2.981e-13
```

- The key observation is that there is a single estimated interactive term in the model, and it is deemed significantly different from zero.
- When the interaction term is statistically significant, it indicates that the relationship between the response variable (MPG) and one of the continuous predictors is influenced by the value of the other continuous predictor.

e. Higher-Order Interactions

- Two-way interactions are the most common kind of interactions we’ll encounter in applications of regression methods.
- This is because for three-way or higher-order terms, need a lot more data for a reliable estimation of interactive effects, and there are a number of interpretative complexities to overcome.
- Three-way interactions are far rarer than two-way effects, and four-way and above are rarer still.
- The following R code fits a multiple linear regression model to the nuclear power plant construction cost data.
- The model includes main effects for the continuous variable cap (plant capacity) and binary variables cum.n, ne, and ct (number of similar constructions, northeastern location, and cooling tower presence, respectively).
- Additionally, it includes all two-way interactions and the three-way interaction among cum.n, ne, and ct.

```
# Load the boot package
library(boot)
# Fit the multiple linear regression model
nuc.fit <- lm(cost ~ cap + cum.n * ne * ct, data = nuclear)
# Display summary of the linear regression model
summary(nuc.fit)
```

```
Call:
lm(formula = cost ~ cap + cum.n * ne * ct, data = nuclear)

Residuals:
     Min       1Q   Median       3Q      Max
-162.475  -50.368   -8.833   43.370  213.131

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  138.0336    99.9599   1.381 0.180585
cap            0.5085     0.1127   4.513 0.000157 ***
cum.n        -24.2433     6.7874  -3.572 0.001618 **
ne          -260.1036   164.7650  -1.579 0.128076
ct          -187.4904    76.6316  -2.447 0.022480 *
cum.n:ne      44.0196    12.2880   3.582 0.001577 **
cum.n:ct      35.1687     8.0660   4.360 0.000229 ***
ne:ct        524.1194   200.9567   2.608 0.015721 *
cum.n:ne:ct  -64.4444    18.0213  -3.576 0.001601 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 107.3 on 23 degrees of freedom
Multiple R-squared:  0.705,        Adjusted R-squared:  0.6024
F-statistic: 6.872 on 8 and 23 DF,  p-value: 0.0001264
```

- The boot package is loaded to access the nuclear dataset.
- The lm function is used to fit a linear regression model (nuc.fit).
- The formula cost ~ cap + cum.n * ne * ct specifies the response variable (cost) as a function of the main effect of cap and the main effects, two-way interactions, and three-way interaction among cum.n, ne, and ct.
- The summary(nuc.fit) command provides detailed information about the fitted linear regression model. This includes coefficients, standard errors, t-values, and p-values for each term in the model.

Linear Model selection and diagnostics

1. Goodness-of-Fit vs. Complexity

- The purpose of fitting a statistical model is to capture the patterns in the data and explain how the variables are related.
- Fitting a statistical model involves finding a balance between two factors: goodness-of-fit and complexity.
- Goodness-of-fit measures how well the model matches the data. A good fit means the model can accurately describe and predict the data. A bad fit means the model is not suitable for the data.
- Complexity measures how complicated the model is. A complex model has more terms and parameters that need to be estimated. A simple model has fewer terms and parameters.
- A trade-off exists between goodness-of-fit and complexity. A more complex model may have a better fit, but it may also overfit the data and lose generalizability. A simpler model may have a worse fit, but it may also be more parsimonious and robust.

a. Principle of Parsimony

- The principle of parsimony is a guideline that says you should use the simplest model that fits the data well. A parsimonious fit is a model that has few parameters but still explains the data well.
- The best model is not necessarily the one that has the highest goodness-of-fit, but the one that balances goodness-of-fit and simplicity. A model that is too complex may overfit the data and perform poorly on new data. A model that is too simple may underfit the data and miss important patterns.
- To find the best model, you need to compare different models and test how well they fit the data. You also need to consider the statistical significance of the predictors, which tells you how likely they are to have a real effect on the response variable.

b. General Guidelines

- Model selection is the process of choosing the best regression model for a given purpose, such as estimating the effect of some predictors or predicting the outcome for new data.
- Model selection involves deciding which predictor variables to include or exclude from the regression equation, and whether to add any transformations or interactions.
- There are some general guidelines for model selection, such as:

1. Treatment of Categorical Predictors:

- When dealing with categorical predictors, the decision to include or exclude them from the model should be based on the overall significance of the categorical variable rather than individual levels.
- If any nonreference levels of a categorical predictor are statistically significant, it suggests that the categorical variable, as a whole, contributes significantly to the mean response. Removal should be considered only if all nonreference coefficients lack evidence against being zero.
- This principle extends to interaction terms involving categorical predictors, where the entire interaction should be retained if present.

2. Inclusion of Lower-Order Interactions and Main Effects:

- When an interaction term is included in a model, the guideline advises that all lower-order interactions and main effects of the relevant predictors must also be retained in the model.
- This ensures that the interpretation of interactive effects as augmentations of lower-order effects remains valid. Removing the main effect of a predictor should only be considered if there are no interaction terms present in the model involving that predictor.

3. Maintenance of Lower-Order Polynomial Terms:

- In models using polynomial transformations of explanatory variables, the guideline suggests keeping all lower-order polynomial terms in the model if the highest-order term is deemed significant.
- For example, if a model includes an order 3 polynomial transformation for a predictor, it should also include the order 1 and order 2 transformations of that variable.
- This approach is essential due to the mathematical behavior of polynomial functions, where distinct terms for linear, quadratic, cubic, and so on, effects are needed to avoid confounding these effects with each other.

2. Model Selection Algorithms

- Model selection algorithms can be controversial.
- There are several different methods, and no single approach is universally appropriate for every regression model.
- Different selection algorithms can result in different final models

a. Nested Comparisons: The Partial F-Test

- The partial F-test is a statistical method used for model comparison, particularly in the context of nested linear regression models.
- Nested models are those where a smaller, less complex model is a reduced version of a larger, more complex model. The test aims to determine whether the additional complexity in the larger model, represented by extra predictor terms, provides a statistically significant improvement in goodness-of-fit compared to the smaller model.
- For example, the partial F-test is applied to compare two linear regression models:

$$\hat{y}_{\text{redu}} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \ldots + \hat{\beta}_p x_p$$

$$\hat{y}_{\text{full}} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \ldots + \hat{\beta}_p x_p + \ldots + \hat{\beta}_q x_q$$

- Here,
  - p is the number of predictors in the reduced model,
  - q is the number of predictors in the full model.
  - The null hypothesis (H_0) assumes that the coefficients of the additional predictors ($\beta_{p+1}, \beta_{p+2}, \ldots, \beta_q$) are all equal to zero, indicating that adding these predictors does not improve the fit significantly.
- The test statistic (F) is calculated using the coefficient of determination (R^2) for both the reduced (R^2_{redu}) and full (R^2_{full}) models, along with the sample size (n), the number of additional predictors (q−p), and the degrees of freedom (df1=q−p and df2 =n−q).
- The formula for the test statistic is:

$$\mathcal{F} = \frac{(R^2_{\text{full}} - R^2_{\text{redu}})(n - q - 1)}{(1 - R^2_{\text{full}})(q - p)}$$

- The test statistic follows an F-distribution with degrees of freedom df1 and df2 under the assumption of H0.
- The p-value associated with this test statistic is used to assess the significance of the improvement in goodness-of-fit.

Example

```
library(MASS)
# Fitting the models
survmult <- lm(Height ~ Wr.Hnd + Sex, data = MASS::survey)
survmult2 <- lm(Height ~ Wr.Hnd + Sex + Smoke, data = MASS::survey)
# Performing the partial F-test
partial_F_test_result <- anova(survmult, survmult2)
# Displaying the results
print(partial_F_test_result)
```


Output

```
> source("D:/ICCA/R_Program/datavisual.R")
Analysis of Variance Table

Model 1: Height ~ Wr.Hnd + Sex
Model 2: Height ~ Wr.Hnd + Sex + Smoke
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1    204 9959.2
2    201 9914.3  3    44.876 0.3033  0.823
```

Output Interpretation:

- Model 1: Height~Wr.Hnd+Sex (Reduced Model)
- Model 2: Height~Wr.Hnd+Sex+Smoke (Full Model)
- The output includes an analysis of variance table with columns like Res.Df (degrees of freedom), RSS (residual sum of squares), F-statistic, and p-value.
- In this specific example:
  - df1=3 (degrees of freedom for the numerator)
  - df2=201 (degrees of freedom for the denominator)
- The p-value associated with the F-statistic is 0.823, suggesting that there is no significant evidence against the null hypothesis (H_0).
- Therefore, adding the Smoke predictor to the model does not provide a statistically significant improvement in goodness-of-fit.

b. Forward Selection

- Forward selection is a stepwise model selection method used in statistical modeling, particularly in the context of multiple regression analysis. The goal of forward selection is to build a predictive model by iteratively adding the most informative predictor variables to the model based on their statistical significance.
- A step-by-step breakdown of the forward selection process using the nuclear data frame:

1. Fit Initial Model:

   ```
   nuc.0 <- lm(cost ~ 1, data = nuclear)
   summary(nuc.0)
   ```

   - Update the model by adding the selected predictor (date).

2. Perform Forward Selection:

   ```
   add1(nuc.0, scope = . ~ . + date + t1 + t2 + cap + pr + ne + ct + bw + cum.n + pt, test = "F")
   ```

   - Use the add1 function to perform a series of independent partial F-tests.
   - The scope argument defines the fullest model considered, and here it includes main effects of all predictors in the nuclear data frame.
   - The test is set to "F" for partial F-tests.
3. Interpret Output:
   - The output provides information on the improvement in goodness-of-fit for each predictor.
   - Choose the predictor that offers the largest and most significant improvement. In this example, it's date.
4. Update Model:

   ```
   nuc.1 <- update(nuc.0, formula = . ~ . + date)
   summary(nuc.1)
   ```

   - Update the model by adding the selected predictor (date).
5. Repeat Steps 2-4:
   - Continue the process by calling add1 on the updated model (nuc.1) and adding the most significant predictors one at a time.
   - Update the model after each addition.
6. Final Model:
   - Continue until there are no more terms that significantly improve the fit..
7. Summary of the Final Model:
   - Review the summary of the final model, which includes the selected predictors and their coefficients.
8. Subjectivity and Considerations:
   - Acknowledge the subjective nature of model selection; different choices in the order of predictor additions may lead to different final models.

The forward selection method is considered a good way to stay involved in the selection process and carefully consider each addition. However, it requires some subjectivity, and the chosen final model may vary based on different decisions in the selection process

c. Backward Selection

- Backward selection is a model-building technique in statistics, specifically in the context of regression analysis.
- It is a stepwise approach to model selection where the process starts with the most complex model, including all available predictor variables, and iteratively removes the least significant variables until a satisfactory model is achieved.
- The goal of backward selection is to simplify the model while retaining its predictive power.
- Following example illustrates the process of backward elimination for model selection using the **drop1** and **update** functions in R.
- Steps involved

1. Fit the Fullest Model (nuc.0):

- A linear regression model is fitted with all available predictor variables (date, t1, t2, cap, pr, ne, ct, bw, cum.n, pt).

```
nuc.0 <- lm(cost ~ date + t1 + t2 + cap + pr + ne + ct + bw + cum.n + pt, data
= nuclear)
```

2. Display the Summary of the Fullest Model (summary(nuc.0)):

- The summary function provides information about the coefficients, statistical measures, and significance tests for each predictor.

```
summary(nuc.0)
```

3. Perform Partial F-Tests to Identify Insignificant Predictors (drop1):

- The drop1 function is used to perform partial F-tests by dropping each predictor variable one at a time.
- The output shows the impact on goodness-of-fit when each predictor is dropped.

```
drop1(nuc.0, test = "F")
```

4. Update the Model by Removing the Least Significant Predictor (update):

- The least significant predictor is identified based on the p-values from the partial F-tests.
- The update function is used to create a new model by removing the identified predictor.

```
nuc.1 <- update(nuc.0, . ~ . - bw)
```


5. Repeat the Process (drop1 and update):

- The process is repeated by performing partial F-tests on the updated model (nuc.1) and removing the least significant predictor.
- This process continues until the remaining predictors are deemed significant.

```
nuc.2 <- update(nuc.1, . ~ . - pt)
nuc.3 <- update(nuc.2, . ~ . - t1)
nuc.4 <- update(nuc.3, . ~ . - ct)
```

6. Final Model Summary (summary(nuc.4)):

- The summary of the final model (nuc.4) is displayed, showing the estimated regression parameters and post-fit statistics.

```
summary(nuc.4)
```

- It's important to note that the final model obtained through backward elimination may differ from the one obtained through forward selection.
- The order and direction of the selection algorithm can lead to different final models due to the complex relationships between predictor variables.

- The process involves iteratively dropping the least significant predictors based on the p-values of partial F-tests until a satisfactory model is achieved.
- The researcher's judgment plays a crucial role in deciding which predictors to retain or drop.

d. Stepwise AIC Selection

- Stepwise AIC (Akaike's Information Criterion) selection is a model selection technique used in statistical modeling and regression analysis.
- The goal of stepwise AIC selection is to systematically build or refine a regression model by iteratively adding or removing predictor variables to achieve the optimal balance between model complexity and goodness-of-fit, as measured by the AIC.
- Here is a stepwise AIC selection procedure with the example using the **mtcars** dataset in R:

1. AIC Formula:
   - AIC is calculated using the formula AIC = 2L + 2(p + 2), where L is the log-likelihood (a measure of goodness-of-fit), and p is the number of regression parameters in the model (excluding the overall intercept).
2. Interpretation of AIC:
   - A lower AIC value indicates a more parsimonious model. AIC balances goodness-of-fit with a penalty for complexity, promoting models that fit well with fewer parameters.
3. Model Selection based on AIC:
   - The AIC of a model is compared to that of other models, and the model with the lowest AIC is considered the best-fitting and most parsimonious.
4. Stepwise Model Selection with AIC:
   - Stepwise model selection involves adding or deleting terms based on the one move that yields the largest reduction in AIC.
   - The process is iterative, and it helps explore candidate models on the way to the final model fit.
5. Implementation in R:
   - The **step** function in R is used for stepwise model selection with AIC. It provides a comprehensive report at each stage of selection, showing the current model fit, AIC values, and potential moves (addition, deletion, or no change).
6. Final Model:
   - The final selected model is stored as an object (**car.step**), and its summary is presented, showing the coefficients, standard errors, t-values, and p-values for each predictor.
7. Considerations:
   - The presence of non-significant predictors (e.g., qsec) in the final model highlights that AIC-based model selection isn't solely based on significance but also considers the overall model fit and parsimony.
8. Criticism and Adjustment:
   - AIC is criticized for potentially favouring more complex models. Researchers can adjust the penalizing effect of extra predictors by modifying the multiplicative contribution of the (p + 2) term in the AIC formula.

Example

1. Starting Model:

```
car.null <- lm(mpg ~ 1, data = mtcars)
```

- The starting model (**car.null**) is an intercept-only model, represented by **mpg ~ 1**

2. Scope Definition:
   car.step<-step(car.null,scope=.~.+wt*hp*factor(cyl)*disp+am
   +factor(gear)+drat+vs+qsec+carb)
   - The scope defines the fullest model to be considered, including main effects and potential interactions among predictors.
3. Stepwise AIC Selection:
   car.step <- step(car.null, scope = scope)
   - The step function performs the stepwise AIC selection, considering additions or deletions of predictors to minimize AIC.
4. Iterative Process Output: The output from the iterative process shows each step, the current model, the AIC value, and potential moves (addition, deletion, or none).
   Output of car.step statement

```
Start:  AIC=115.94
mpg ~ 1

               Df Sum of Sq     RSS     AIC
+ wt            1    847.73  278.32  73.217
+ disp          1    808.89  317.16  77.397
+ factor(cyl)   2    824.78  301.26  77.752
+ hp            1    678.37  447.67  88.427
+ drat          1    522.48  603.57  97.988
+ vs            1    496.53  629.52  99.335
+ factor(gear)  2    483.24  642.80 102.003
+ am            1    405.15  720.90 103.672
+ carb          1    341.78  784.27 106.369
+ qsec          1    197.39  928.66 111.776
<none>                      1126.05 115.943

Step:  AIC=73.22
mpg ~ wt
```



```
               Df Sum of Sq     RSS     AIC
+ factor(cyl)   2     95.26  183.06  63.810
+ hp            1     83.27  195.05  63.840
+ qsec          1     82.86  195.46  63.908
+ vs            1     54.23  224.09  68.283
+ carb          1     44.60  233.72  69.628
                                          ...
```

5. Final Model: The final selected model is stored as car.step, and its summary is shown:
   R

```
summary(car.step)
```

```
Call:
lm(formula = mpg ~ wt + hp + qsec + wt:hp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.8243 -1.3980  0.0303  1.1582  4.3650

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 40.310410   7.677887   5.250 1.56e-05 ***

wt          -8.681516   1.292525  -6.717 3.28e-07 ***
hp          -0.106181   0.026263  -4.043 0.000395 ***
qsec         0.503163   0.360768   1.395 0.174476
wt:hp        0.027791   0.007298   3.808 0.000733 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.117 on 27 degrees of freedom
Multiple R-squared:  0.8925,        Adjusted R-squared:  0.8766
F-statistic: 56.05 on 4 and 27 DF,  p-value: 1.094e-12
```

3. Residual Diagnostics

- Model diagnostics are essential for ensuring the validity of the regression model and verifying that it accurately represents the relationships within the data.
- The focus is on the theoretical assumptions underpinning multiple linear regression, as mentioned in an earlier section.
- When fitting these models, remember to keep these four things in mind:

1. Errors:
   - The error term in the regression model is assumed to be normally distributed with a mean of zero and a constant variance denoted as $\sigma 2$.
   - Independence: The error associated with one observation is assumed to be independent of the error of any other observation.
   - If the model suggests a violation of these assumptions, further investigation is required, often involving the refitting of a variation of the model.

2. Linearity:
   - It is crucial to assume that the mean response as a function is linear in terms of the regression parameters ($\beta 0, \beta 1, \ldots, \beta p$).
   - Transformations of individual variables and the inclusion of interactions can relax the linearity assumption to some extent.
   - Any diagnostic indication that the relationship is nonlinear should be investigated.

3. Extreme or Unusual Observations:
   - Inspection of extreme data points or those strongly influencing the fitted model is necessary.
   - Incorrectly recorded points or outliers should be identified and, if necessary, removed from the analysis.

4. Collinearity:

- Highly correlated predictors can adversely affect the entire model, leading to misinterpretation of predictor effects.
- Collinearity issues should be addressed to ensure the reliability of the regression results.

a. Inspecting and Interpreting Residuals

- The model assumes that deviations of raw observations from the fitted line are due to normally distributed errors.
- In practice, true error values are unknown, so diagnostic tools are used.
- Diagnostic plots often involve assessing residuals (observed minus fitted values).
- Standardized residuals can be used to ensure consistent variance for comparison.

Diagnostic Tools:

1. Residuals versus Fitted Plot:
   - Random scattering around zero indicates validity of assumptions.
   - Systematic patterns suggest issues like nonlinearity or dependent observations.
   - Detects heteroscedasticity (non-constant variance) as "fanning out."

   Example Diagnostic Plots:
   - Three impressions in below figure: random, systematic, heteroscedastic.
   - Model assumptions affect regression coefficient estimates and their reliability.

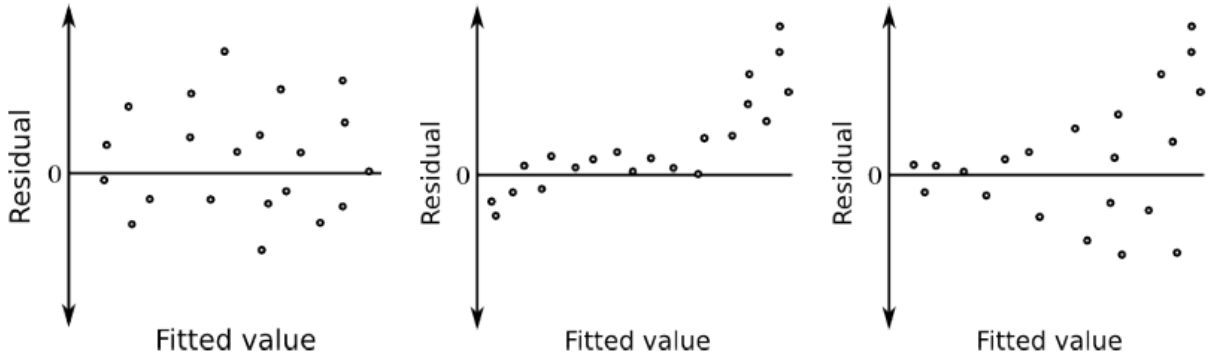


Figure 22-1: Three impressions of a hypothetical residuals versus fitted diagnostic plot from a linear regression: random (left), systematic (middle), and heteroscedastic (right)

2. Scale-Location Plot:
   - Similar to residuals versus fitted plot but uses standardized residuals.
   - Reveals trends in the size of departure of data points from fitted values.
   - Useful for detecting issues like heteroscedasticity.

Example 1

- In this example, the author is using the car.step object, which was created using stepwise AIC (Akaike Information Criterion) selection to choose a model for MPG (miles per gallon) for the mtcars dataset.
- The focus is on diagnosing the fit of the selected model and checking for potential issues with the model assumptions.

Steps and Observations:

1. Applying the plot Function:
   - The plot function is applied to the car.step object, which is an lm (linear model) object resulting from the stepwise AIC selection.
   - By default, the plot function generates six types of diagnostic plots in succession. The user is instructed to hit <Return> to progress through them.
   - However, the author prefers to select each plot individually using the which argument.
2. Residuals versus Fitted Plot (which=1):
   - The first plot generated is the residuals versus fitted plot (which=1).

- R adds a smoothed line to help interpret any trend. The three most extreme points from zero are annotated.
- The model formula is specified below the horizontal axis label.

3. Observations on Residuals versus Fitted Plot:
   - The residuals versus fitted plot for car.step shows little cause for concern.
   - There isn't a discernible trend, and the errors (ei) appear homoscedastic in their distribution (consistent variance).
4. Scale-Location Plot (which=3):

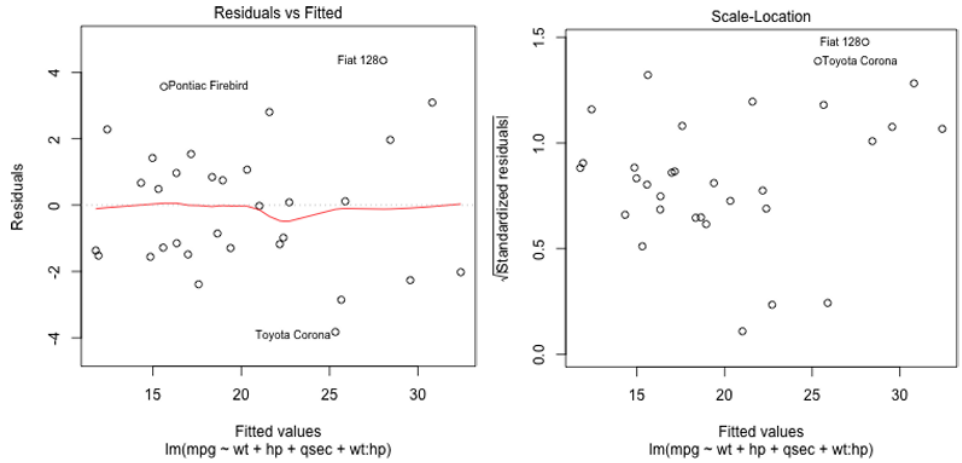


Figure 22-2: Residuals versus fitted and scale-location diagnostic plots for the `car.step` *model*

- This plot is similar to the residuals versus fitted plot but uses the square root of the absolute value of the standardized residuals.

$$|e_i/(\hat{\sigma}\{1-h_{ii}\}^{0.5})|^{0.5}$$

- The plot reveals trends in the size of the departure of each data point from its fitted value as the fitted values increase.

5. Observations on Scale-Location Plot:

- Similar to the residuals versus fitted plot, the scale-location plot for car.step doesn't raise concerns.
- It can be more useful in detecting issues like heteroscedasticity, and a plot with no discernible pattern is desirable.

6. Customizing Scale-Location Plot:

- The plot function allows customization. In this case, the smoothed trend line is removed (add.smooth=FALSE), and the number of extreme points labeled is controlled using id.n=2.

Example 2

- In this example, the Galileo's ball-rolling data is used to illustrate the importance of residual diagnostic plots in assessing the adequacy of regression models.
- Two models are fitted to the data: a simple linear model (gal.mod1) and a quadratic model (gal.mod2). The response variable is "distance traveled" (d), and the explanatory variable is "height" (h).
- Steps and Observations:

1. Data Definition and Model Fitting:
   - A data frame gal is created with seven observations of "distance traveled" and "height."
   - Two regression models are fitted: a simple linear model (gal.mod1) and a quadratic model (gal.mod2).

```
gal <- data.frame(d=c(573,534,495,451,395,337,253),
                  h=c(1,0.8,0.6,0.45,0.3,0.2,0.1))
gal.mod1 <- lm(d~h, data=gal)
gal.mod2 <- lm(d~h+I(h^2), data=gal)
```

2. Visualization and Residual Diagnostic Plots:
   - A plot of the raw data with a simple linear trend is created, and the simple linear model is superimposed:

```
plot(gal$d ~ gal$h, xlab="Height", ylab="Distance")
abline(gal.mod1)
plot(gal.mod1, which=1, id.n=0)
plot(gal.mod2, which=1, id.n=0)
```

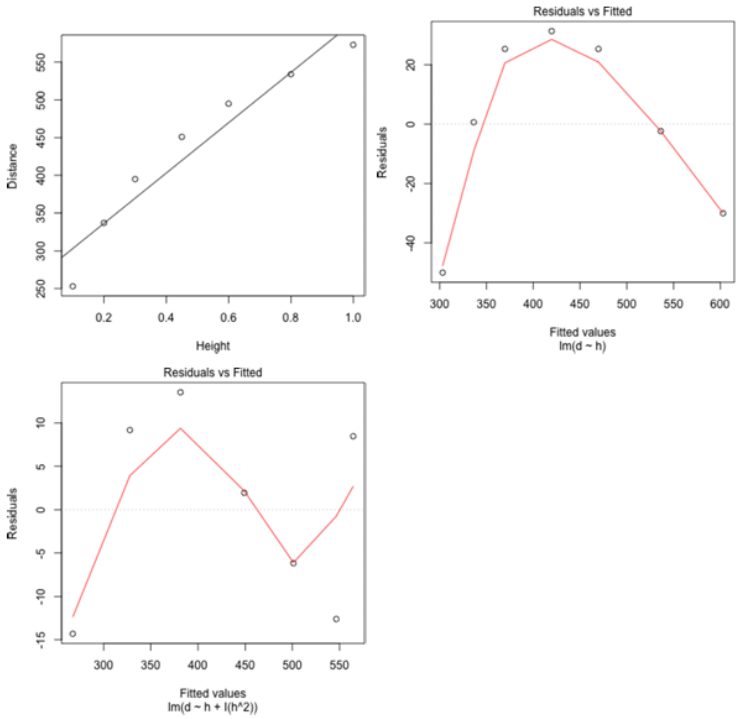


Figure 22-3: Demonstrating residual diagnostics for Galileo's ball-rolling data. Top left: The raw data with a simple linear trend corresponding to `gal.mod1` *superimposed. Top right: Residuals versus fitted for the linear-trend-only model. Bottom: Residuals versus fitted for the quadratic model* `gal.mod2`.

- The top-left plot in the above figure, shows the data with the linear trend.
- Residuals versus fitted plot for the linear-trend-only model (**gal.mod1**) is created:
- The top-right plot in the above figure, shows a systematic pattern in the residuals, suggesting inadequacy of the linear-trend-only model.
- Residuals versus fitted plot for the quadratic model (**gal.mod2**) is created:
- The bottom plot in the above figure, shows that including a quadratic term in "height" removes the prominent curve in the residuals. However, some systematic behaviour in a wavelike form remains.

b. Assessing Normality

- To assess the assumption that the error is normally distributed, two methods are presented: normal QQ plot and performing the Shapiro-Wilk hypothesis test.

Normal QQ Plot:

- To assess normality, a normal QQ plot of the standardized residuals is created.
- The plot function is called on the lm object (car.step) with which=2 to produce the QQ plot.

```
plot(car.step, which=2)
```

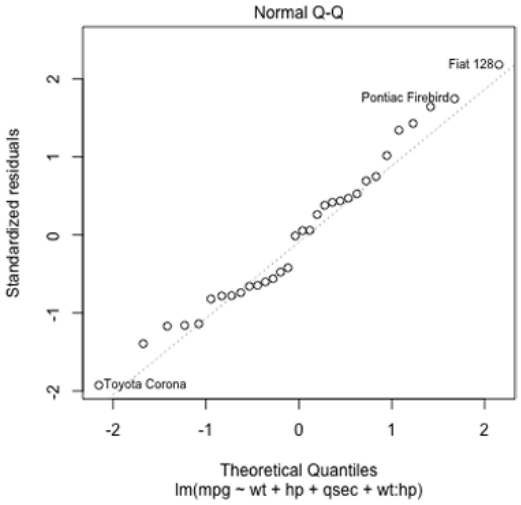


Figure 22-4: Normal QQ plot of the residuals from the car.step model

- The gray diagonal line represents the expected quantiles for a normal distribution.
- The plotted points are the corresponding quantiles of the estimated regression errors.
- Normally distributed data points should closely follow the straight line.
- In above figure, the points generally follow the path of the theoretical normal quantiles.
- Some deviation is expected, but there's no apparent major departure from normality.

Shapiro-Wilk Test:

- Another method to test for normality is the Shapiro-Wilk hypothesis test, performed using the shapiro.test function.

```
shapiro.test(rstandard(car.step))
```

```
        Shapiro-Wilk normality test

data:  rstandard(car.step)
W = 0.97105, p-value = 0.5288
```

- The null hypothesis of the Shapiro-Wilk test is that the data are normally distributed.
- A small p-value would suggest non-normality.
- The reported p-value in the output is large (in this case, 0.5288), indicating no strong evidence against the null hypothesis.
- In other words, there is no strong evidence, according to this test, that the residuals of car.step are not normal.

c. Illustrating Outliers, Leverage, and Influence

- In this section, the importance of investigating individual observations that appear unusual or extreme in the context of linear regression models. The terms "outlier," "leverage," and "influence" are defined and illustrated through hypothetical examples.

Important Key Terms:

1. **Outlier:**

- A general term for an unusual observation in the context of the data.

- In linear regression, an outlier usually has a large residual but is identified as an outlier only if it doesn't conform to the trend of the fitted model.
- An outlier can significantly alter trends described by the fitted model.

2. Leverage:

- Refers to the extremity of the values of the present predictors.
- A high-leverage point is an observation with predictor values extreme enough to potentially significantly affect the slopes or trends in the fitted model.
- An outlier can have high or low leverage.

3. Influence:

- An observation with high leverage that does affect the estimated trends is deemed influential.
- Influence is judged when the response value is taken into account alongside the corresponding predictor values.

Hypothetical Examples:

1. Data Creation:

   Two vectors of ten supposed responses (y) and explanatory (x) values are created.

   ```
   x <- c(1.1,1.3,2.3,1.6,1.2,0.1,1.8,1.9,0.2,0.75)
   y <- c(6.7,7.9,9.8,9.3,8.2,2.9,6.6,11.1,4.7,3)
   ```

2. Additional Observations:

   Six objects (p1x to p3y) store predictor and response values for three additional observation points.

   ```
   p1x <- 1.2; p1y <- 14
   p2x <- 5; p2y <- 19
   p3x <- 5; p3y <- 5
   ```

3. Linear Model Fits:

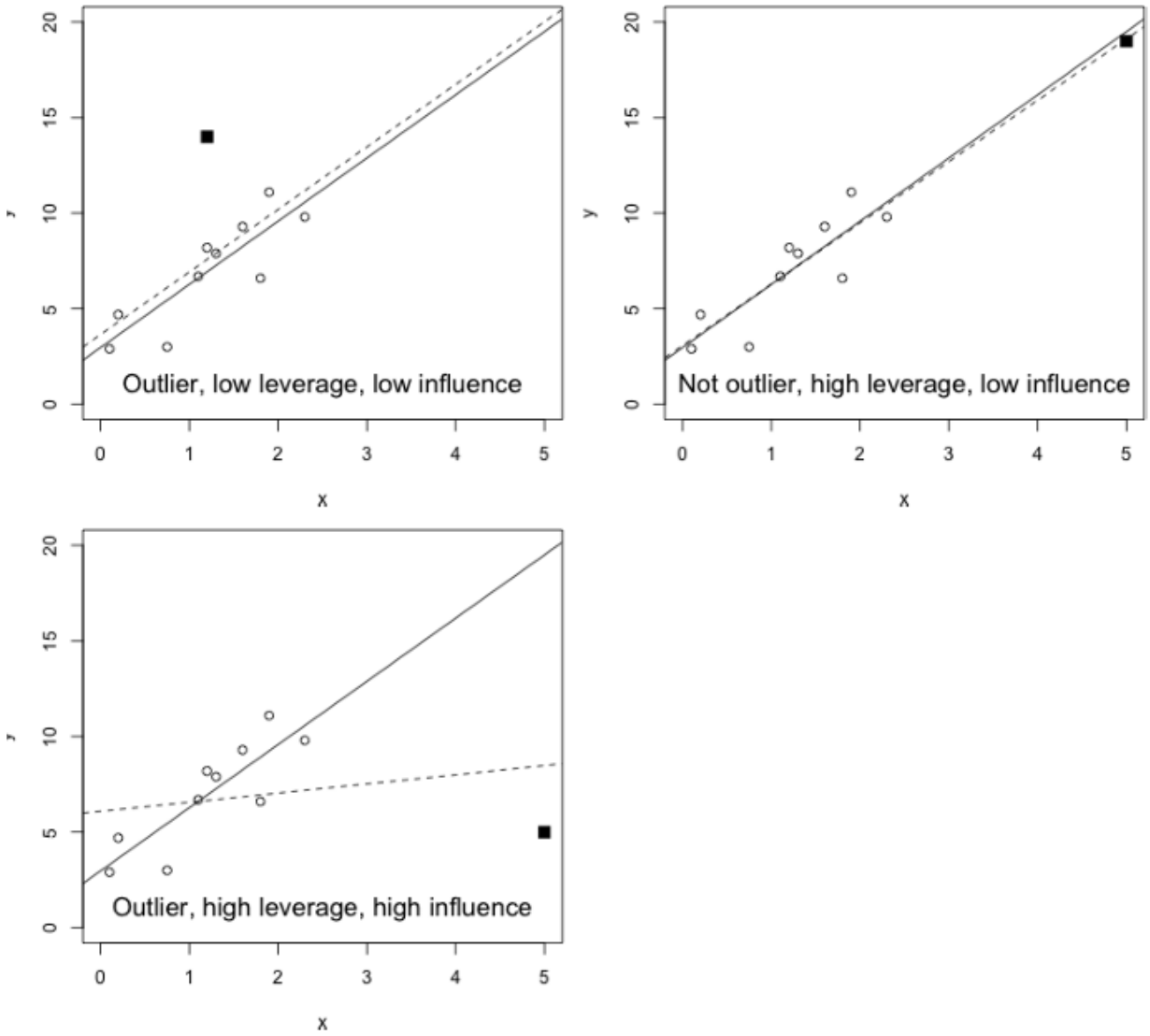
   Four simple linear models (mod.0 to mod.3) are fitted using the original and additional observations.

   ```
   mod.0 <- lm(y~x)
   mod.1 <- lm(c(y,p1y)~c(x,p1x))
   mod.2 <- lm(c(y,p2y)~c(x,p2x))
   mod.3 <- lm(c(y,p3y)~c(x,p3x))
   ```

4. Visualization:

   Scatterplots are created to visually illustrate the definitions of outlier, leverage, and influence.

   ```
   plot(x, y, xlim=c(0,5), ylim=c(0,20))
   # Code for top-left plot
   points(p1x, p1y, pch=15, cex=1.5)
   abline(mod.0)
   abline(mod.1, lty=2)
   text(2, 1, labels="Outlier, low leverage, low influence", cex=1.4)
   # Similar code for other plots
   ```



Top-Left Plot:

- Example of an outlier with low leverage and low influence.
- The additional point is away from the bulk of the data but has low leverage and minimal influence on the fitted model.

Top-Right Plot:

- Example of a high-leverage point with low influence.
- Point 2 is considered a high-leverage point as it sits at an extreme predictor value but has low influence as it barely affects the model.

Bottom Plot:

- Example of an outlier with high leverage and high influence.
- The additional point is away from the original trend, has extreme predictor values, high leverage, and significantly influences the model.

d. Calculating Leverage

- Leverage is calculated using the design matrix structure **X**, and it is denoted as **hii** for the **i**-th observation out of **n** total observations. The leverage is calculated using the formula:

$$H = X(X^\top X)^{-1}X^\top$$

- X is achieved in a straight forward fashion using knowledge of cbind.

- H is subsequently calculated using the corresponding functions for matrix multiplication(%*%),matrixtransposition(t),matrixinversion(solve), and diagonal element extraction(diag).
- Then we can plot the values hii against the values of x themselves. The following code produces the below figure

```
# Constructing the design matrix X
X <- cbind(rep(1, 10), x)
# Calculating leverage manually
hii <- diag(X %*% solve(t(X) %*% X) %*% t(X))
# Plotting leverage against predictor values
plot(hii ~ x, ylab = "Leverage", main = "", pch = 4)
```

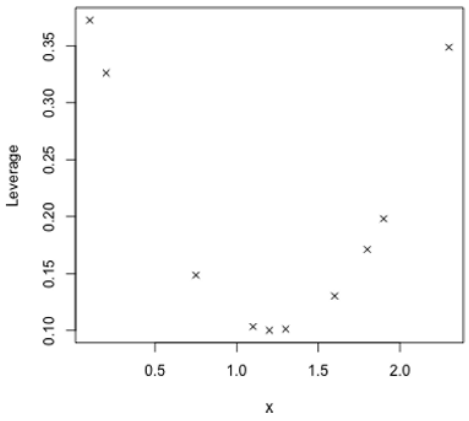


Figure 22-6: Plotting the leverage of the 10 illustrative predictor observations in x

- The plot illustrates the leverage of the 10 illustrative predictor observations against their corresponding predictor values.
- Leverage tends to increase as observations move away from the mean of the predictor data.
- The pattern observed in the plot aligns with the expected behavior of leverage in linear regression.
- Alternatively, the built-in R function **hat values** can be used to obtain leverage values directly from the fitted model object. This function is named after the style of the matrix algebra in the leverage formula.

```
# Obtain leverage values using the hatvalues function
hatvalues(mod.0)
```


e. Cook's Distance

- The Cook's distance (Di) is a measure of the influence of individual observations on a fitted regression model.
- It evaluates the effect of deleting a specific observation from the model. The Cook's distance for observation i is given by the formula:

$$D_i = \sum_{j=1}^{n} \frac{(\hat{y}_j - \hat{y}_j^{(-i)})^2}{(p+1)\hat{\sigma}^2}; \qquad i, j = 1, \ldots, n$$

- Where:
  - $\hat{y}_j$ is the predicted mean response of observation j for the model fitted with all n observations.

- $\hat{y}_{(i)j}$ represents the predicted mean response of observation j for the model fitted without the ith observation.
- p is the number of predictor regression parameters (excluding the intercept).
- σ^ is the estimate of the residual standard error.

- In simple terms, Cook's distance measures how much the model predictions change if a particular observation is excluded.
- Larger values of Di indicate higher influence of the observation on the model.
- There are no strict rules for determining when an observation is influential based on Cook's distance. However, some common guidelines include:
  - If Di>1, the point is considered influential.
  - A more sensitive rule suggests Di>4/n, where n is the number of observations.

Example

1. Dataset Initialization:

```
x <- c(1.1, 1.3, 2.3, 1.6, 1.2, 0.1, 1.8, 1.9, 0.2, 0.75)
y <- c(6.7, 7.9, 9.8, 9.3, 8.2, 2.9, 6.6, 11.1, 4.7, 3)
```

- These vectors **x** and **y** represent ten pairs of predictor and response values.

2. Create Additional Observations***:***

```
p1x <- 1.2
p1y <- 14
p2x <- 5
p2y <- 19
p3x <- 5
p3y <- 5
```

- Three additional observation points are defined: **p1**, **p2**, and **p3**.

3. Linear Model Fitting:

```
mod.0 <- lm(y ~ x)
mod.1 <- lm(c(y, p1y) ~ c(x, p1x))
mod.2 <- lm(c(y, p2y) ~ c(x, p2x))
mod.3 <- lm(c(y, p3y) ~ c(x, p3x))
```

- Four linear regression models are fitted: one with the original data (**mod.0**) and three including each additional point separately.

4. Cook's Distance Calculation:

```
cooks <- rep(NA, length(y))
for (i in 1:length(y)) {
   temp.y <- y[-i]
   temp.x <- x[-i]
   temp.model <- lm(temp.y ~ temp.x)
   temp.fitted <- predict(temp.model, newdata = data.frame(temp.x = x))
   cooks[i] <- sum((fitted(mod.1) - temp.fitted)^2) / (2 *
summary(mod.1)$sigma^2)
}
```

- Cook's distances are calculated manually for each observation in the **mod.1** model.

5. Comparison with Cook's Distance Function***:***

```
cooks.distance(mod.1)
```

- The cooks.distance function is used to calculate Cook's distances for the mod.1 model.

6. Diagnostic Plot:

```
plot(mod.1, which = 4)
plot(mod.2,which=4)
plot(mod.3,which=4)
abline(h=c(1,4/n),lty=2)
```

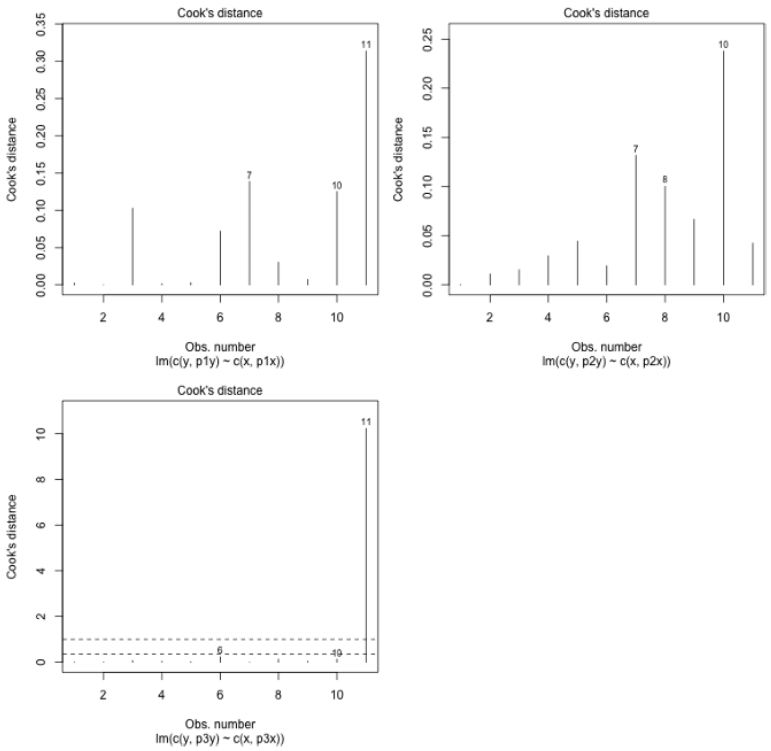


Figure 22-7: Three illustrative examples of the Cook's distance plots produced in R, based on `mod.1` *(top left),* `mod.2` *(top right), and* `mod.3` *(bottom)*

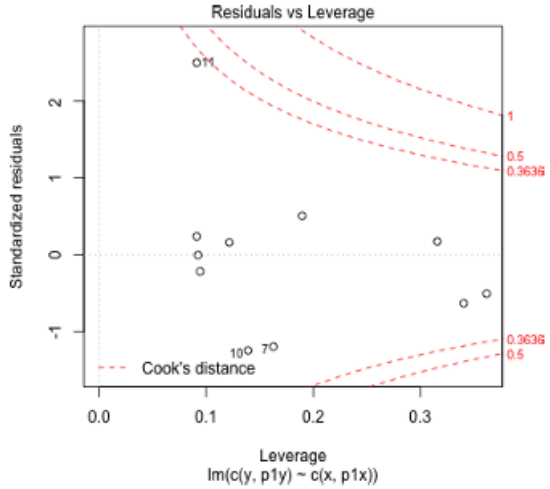
f. Graphically Combining Residuals, Leverage, and Cook's Distance

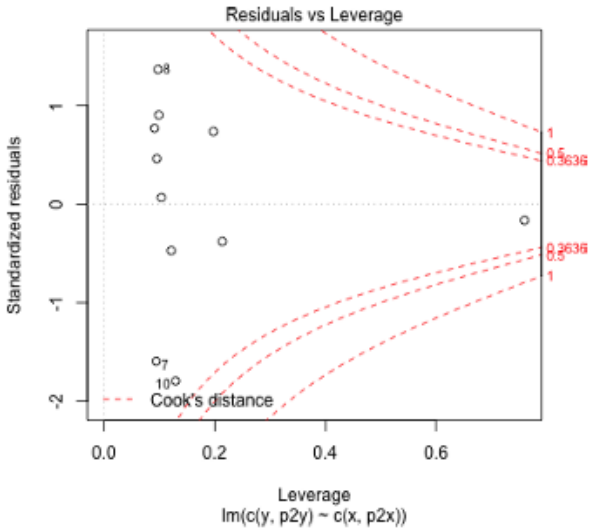
- An excerpt from a statistical analysis or data visualization guide that discusses combination diagnostic plots for linear regression models.
- These plots are used to assess the influence of individual observations on the regression model

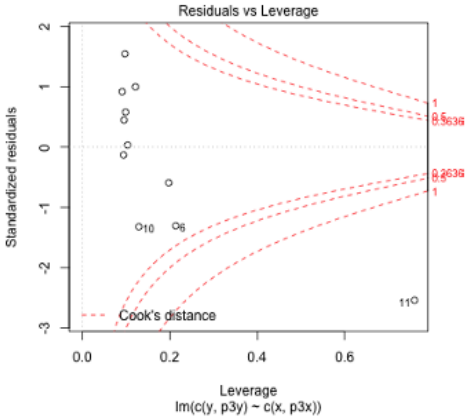
1. Combination Diagnostic Plot (which=5):

- Horizontal axis: Leverage
- Vertical axis: Standardized residuals
- Contours represent Cook's distances
- Helps identify whether high influence is due to high leverage, large residuals, or both
- Contour levels (cook.levels) are specified (4/11, 0.5, 1) for the rule-of-thumb values
- Using the data models mod.1, mod.2, and mod.3 (from the cook's distance example) enter the following code with which=5 to produce the below plots

```
plot(mod.1, which=5, add.smooth=FALSE, cook.levels=c(4/11, 0.5, 1))
plot(mod.2, which=5, add.smooth=FALSE, cook.levels=c(4/11, 0.5, 1))
plot(mod.3, which=5, add.smooth=FALSE, cook.levels=c(4/11, 0.5, 1))
```



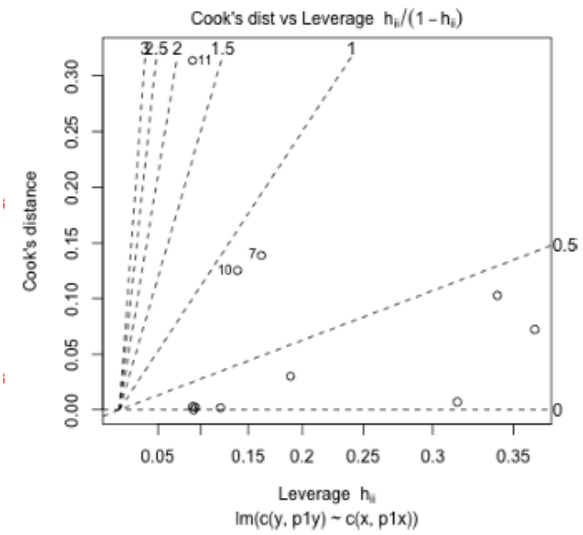


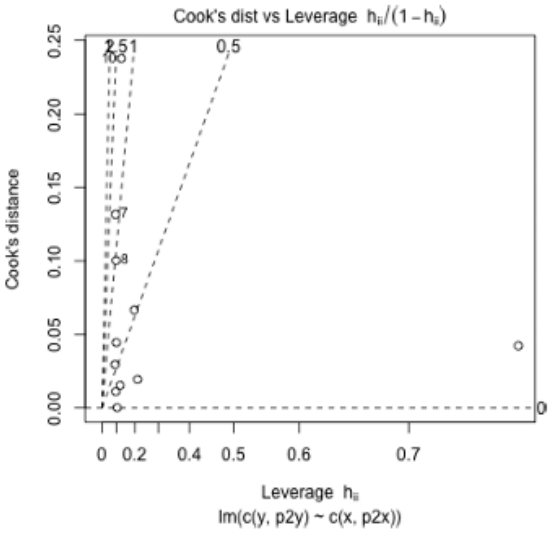


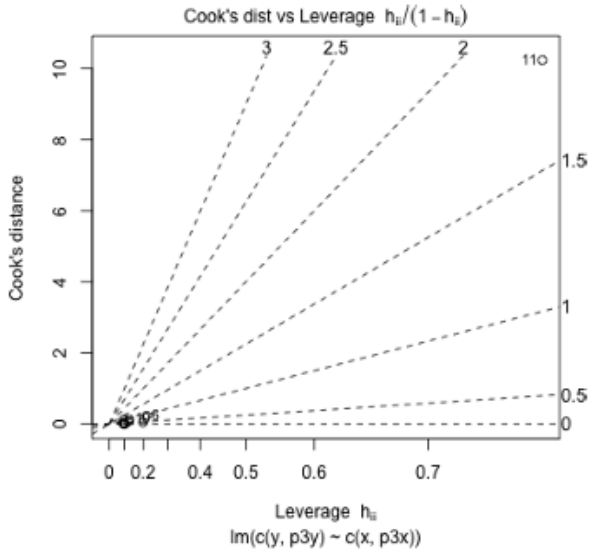
2. Combination Diagnostic Plot (which=6):

- Horizontal axis: Transformed leverage (hii (1-hii))
- Vertical axis: Cook's distances
- Transformation amplifies larger leverage points
- Contours define standardized residuals as a function of scaled leverage and Cook's distance
- Example code for three models (mod.1, mod.2, mod.3):

```
plot(mod.1, which=6, add.smooth=FALSE)
plot(mod.2, which=6, add.smooth=FALSE)
plot(mod.3, which=6, add.smooth=FALSE)
```







3. Application to Real Data (car.step model):

- Application of the combination diagnostic plots to a real data example (car.step model)
- Identifies influential observations based on Cook's distances and leverage
- Example code

```
plot(car.step, which=5, cook.levels=c(4/nrow(mtcars), 0.5, 1))
plot(car.step, which=6, cook.levels=c(4/nrow(mtcars), 0.5, 1))
```

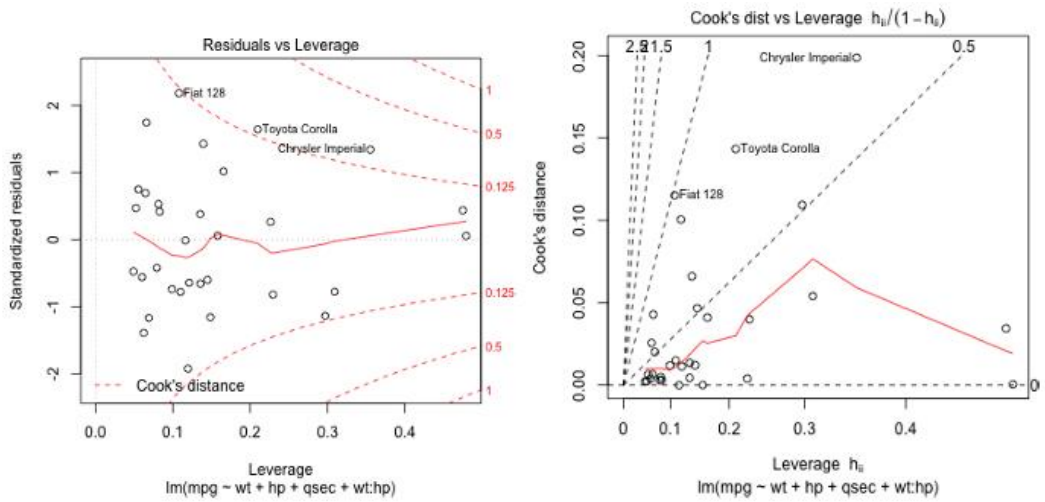


Figure 22-10: Combination diagnostic plot of standardized residuals against leverage (left) and Cook's distance against leverage (right) for the `car.step` *model*

4. Collinearity

- Collinearity (also referred to as multicollinearity) is when two or more of the explanatory variables are highly correlated with each other.

a. **Potential Warning Signs**

- High correlation between two predictors implies there will be some level of redundancy in terms of the information they contain when it comes to the response variable.
- It's a problem since it can destabilize the ability to reliably f it a model and, as noted earlier, therefore be detrimental to any subsequent model-based inference.
- The following items serve as potential warnings of collinearity when you're inspecting a model summary:
    - The omni bus F-test result is statistically significant, but none of the individual t-test results for the regression parameters are significant.
    - The sign of a given coefficient estimate contradicts what you would reasonably expect to see, for example, drinking more wine resulting in a lower blood alcohol level.
    - Parameter estimates are associated with unusually high standard errors or vary wildly when the model is fitted to different random record sub sets of the data.

b. Correlated Predictors: A Quick Example

- Consider the survey data of the statistics students again, located in the MASS package, specifically focusing on predicting student height using variables such as handspan of the writing hand (Wr.Hnd) and nonwriting hand (NW.Hnd).
- The correlation between Wr.Hnd and NW.Hnd is high (0.948), indicating a strong positive linear association between the two variables.
- Perform simple linear regression models:

1. Using Wr.Hnd:

```
summary(lm(Height ~ Wr.Hnd, data = survey))
```

```
Call:
lm(formula = Height ~ Wr.Hnd, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-19.7276  -5.0706  -0.8269   4.9473  25.8704

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 113.9536     5.4416   20.94   <2e-16 ***
Wr.Hnd        3.1166     0.2888   10.79   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.909 on 206 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3612,        Adjusted R-squared:  0.3581
F-statistic: 116.5 on 1 and 206 DF,  p-value: < 2.2e-16
```

- Results showed a significant and positive impact of Wr.Hnd on predicting mean student height.

1. Using NW.Hnd:
   summary(lm(Height ~ NW.Hnd, data = survey))

```
Call:
lm(formula = Height ~ NW.Hnd, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-21.8285  -5.1397  -0.2867   4.5611  25.5750
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 118.0324     5.2912   22.31   <2e-16 ***
NW.Hnd        2.9107     0.2818   10.33   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.032 on 206 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3412,        Adjusted R-squared:  0.338
F-statistic: 106.7 on 1 and 206 DF,  p-value: < 2.2e-16
```

- Results also indicated a significant and positive impact of NW.Hnd on predicting student height.
- However, when you tried to model height based on both predictors simultaneously:
  summary(lm(Height ~ Wr.Hnd + NW.Hnd, data = survey))

```
Call:
lm(formula = Height ~ Wr.Hnd + NW.Hnd, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-20.0144  -5.0533  -0.8558   4.7486  25.8380

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 113.9962     5.4545  20.900   <2e-16 ***
Wr.Hnd        2.7451     1.0728   2.559   0.0112 *
NW.Hnd        0.3707     1.0309   0.360   0.7195
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.926 on 205 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3616,        Adjusted R-squared:  0.3554
F-statistic: 58.06 on 2 and 205 DF,  p-value: < 2.2e-16
```

- The results show that including both Wr.Hnd and NW.Hnd at the same time masks individual contributions, resulting in less statistical significance for each predictor.
- The coefficients for both predictors become less significant, and their p-values increase.
- This suggests collinearity issues between Wr.Hnd and NW.Hnd, where the effects of these variables are intermingled.
- The statistical significance of each predictor decreases when included together in the model, indicating potential multicollinearity.
- The omnibus F-test remains significant, but the individual contributions are not as clear.
- The warning sign here is that when predictors are highly correlated, including them together in a model may lead to difficulties in interpreting their individual effects. It's essential to consider collinearity issues and decide whether it's more appropriate to use one predictor over the other or explore methods to address multicollinearity.

Advanced graphics

Advanced plot customization

1. Handling the Graphic Devices

- It's possible to have multiple graphics devices open, but only one will be deemed active at any given time.
- This is useful when you're working on several plots at once or want to view or alter one plot without closing any others.

a) Manually Opening a New Device

- The typical base R commands, such as plot, hist, boxplot, and so on will automatically open a device for plotting and draw the desired plot, if nothing is currently open.
- We can also open new device windows using **dev.new;** this newest window will immediately become active, and any subsequent plotting commands will affect that particular device.

Example

```
plot(iris$Sepal.Length)
dev.new()
```

```
hist(chickwts$weight)
```

b) Switching Between Devices

- To change something in one window without closing another window, use dev.set followed by the device number , you want to make active.

Example

```
plot(iris$Sepal.Length)
dev.new()
hist(chickwts$weight)
dev.new()
barplot(chickwts$weight)
dev.set(4)
abline(v=mean(chickwts$weight),lty=2)
```

c) Close a device

- To close a graphics device, either click the X with your mouse as you would to close any window or use the dev.off function.
- Calling dev.off() with no arguments simply closes the currently active device. Otherwise, you can specify the device number.

Example

```
plot(iris$Sepal.Length)
dev.new()
hist(chickwts$weight)
dev.new()
barplot(chickwts$weight)
dev.set(4)
abline(v=mean(chickwts$weight),lty=2)
dev.off()
dev.off(5)
```

d) Multiple Plots in One Device

- The practice of displaying more than one graph or chart within a single plotting area or visual output.
- There are mainly two ways to display multiple plots in one device

i. Setting the mfrow parameter

- In R, the par() function is used to set or query graphical parameters, and the mfrow argument within the par() function is specifically used to specify the layout of multiple plots in one device.
- The mfrow argument takes a vector of length 2, where the first element represents the number of rows and the second element represents the number of columns in the layout.
- Syntax: - **par(mfrow = c(rows, columns))**

Example

```
plot(iris$Sepal.Length)
dev.new(width= 8,height=4)
par(mfrow=c(1,2))
plot(chickwts$weight)
hist(chickwts$weight)
```

ii. Defining a Particular Layout

- In R, you can define a particular layout for multiple plots using the layout() function.

- The layout() function allows you to specify the structure of the plotting region by defining a matrix that represents the arrangement of plots.
- Each element of the matrix corresponds to a subplot.

Example

```
plot(iris$Sepal.Length)
dev.new()
# Define a 2x2 layout
my_layout <- matrix(c(3,1, 2, 4), nrow = 2, ncol = 2, byrow = TRUE)
# Set up the layout
layout(my_layout)
# Plot on the second subplot
plot(chickwts$weight)
# Plot on the third subplot
hist(chickwts$weight)
# Plot on the first subplot
barplot(chickwts$weight)
# Plot on the fourth subplot
boxplot(chickwts$weight)
# Reset the layout to default
layout(1)
```

2. Plotting Regions and Margins

- For any single plot created using base R graphics, there are three regions that make up the image.
  - The plot region: - This is where your actual plot appears and where you'll usually be drawing your points, lines, text, and so on. The plot region uses the user coordinate system, which reflects the value and scale of the horizontal and vertical axes.
  - The figure region: - This is the area that contains the space for your axes, their labels, and any titles. These spaces are also referred to as the figure margins.
  - The outer region: - It is also referred to as the outer margins, is additional space around the figure region that is not included by default but can be specified if it's needed.

Important terms

- Measuring Margin Space:
  - Margin space is typically measured in terms of lines of text that can fit on top of one another parallel to each edge of the plotting area.
  - The number of lines is specified as a vector of length 4, representing each of the four sides: c(bottom, left, top, right).
  - These values determine how much space should be left for margins at the bottom, left, top, and right sides of the plot.
- Graphical Parameters oma and mar:
  - oma (outer margin) and mar (figure margin) are graphical parameters in R used to control the size of margins in a plot.
  - oma sets the outer margins, which include space for titles, axis labels, and other annotations outside the plotting area.
  - mar sets the figure margins, controlling the space between the plotting area and the edges of the device where the plot is displayed.
- Initialization with par:
  - Both oma and mar need to be initialized through a call to the par function before you start drawing any new plots.

- By calling par with the appropriate parameters, you set the initial values for these margins, influencing the layout of subsequent plots

a) Default Spacing

- We can find your default figure margin settings with a call to par in R.
- To see the default margins space setting we can use par()$oma and par()$mar
- par()$oma. This expression extracts the values of the outer margin (oma) from the current graphical parameters.
- The par()$mar expression in R extracts the values of the figure margin

Example

```
> source("D:/ICCA/R_Program/advance_graphics.R")
> par()$oma
[1] 0 0 0 0
> par()$mar
[1] 5.1 4.1 4.1 2.1
```

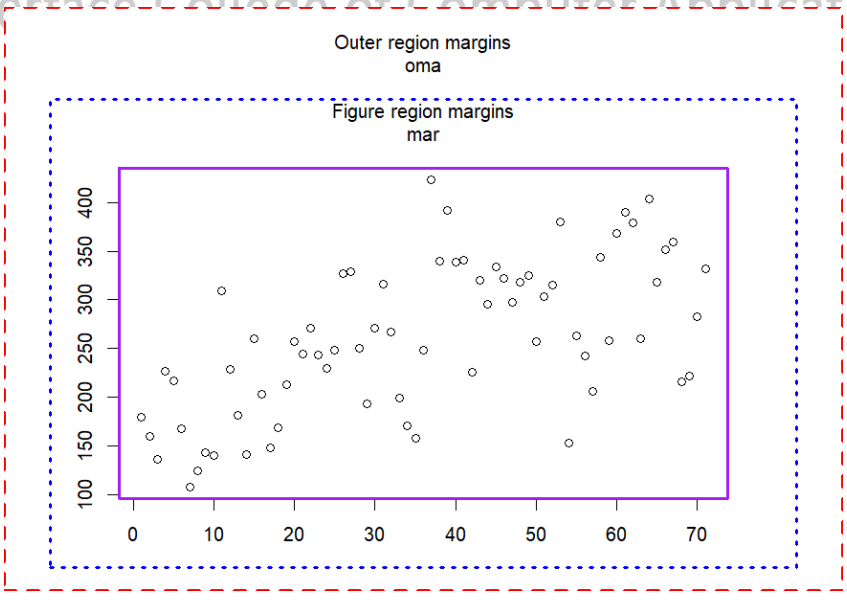
b) Custom Spacing

- To set custom outer margins (oma) and figure margins (mar) in R, we can use the par() function.
- The oma parameter controls the outer margins, which include space for titles, axis labels, and other annotations outside the plotting area.
- The mar parameter controls the size of the margins around the plotting region.

Example

```
par(oma=c(1,2,4,2),mar=c(3,3,3,3) )
plot(chickwts$weight)
box(which = "figure",lty=3, col ="blue",lwd=3)
box(which= "outer",lty=2, col="red",lwd=3)
box(which = "plot",col="purple",lwd = 3)
mtext("Figure region margins\nmar",line=1)
mtext("Outer region margins\noma",line=1,outer=TRUE)
```

Output



Note 1:- The box() function is used to add a box around the current plot.

Syntax: - **box(which = "plot", lty = 1, col = "black")**

Where,

- which: This parameter is optional and determines which parts of the box are drawn. It can take a combination of the following values:
  - "plot": The main box around the plotting region.
  - "figure": The outer box around the entire figure, including labels and margins.
  - "outer": It draws a box around the entire figure, including the plotting region, labels, and margins.
- lty: This parameter is optional and sets the line type of the box. It can take integer values or character strings representing line types. The default is 1 (solid line).
- col: This parameter is optional and sets the color of the box lines. The default is "black.

Note 2: - The mtext() function is used to add text to the margins of a plot in R.

Syntax: - mtext(text, side = 3, line = 0, outer = FALSE…)

Where,

- text: The text you want to display.
- side: An integer specifying which margin to place the text. Possible values are 1 (bottom), 2 (left), 3 (top), 4 (right).
- line: A numeric value specifying the line at which to place the text. The default is 0, meaning just inside the margin.
- outer: A logical value indicating whether to place the text in the outer margin. If outer is TRUE, the side parameter refers to the outer margins (1=bottom, 2=left, 3=top, 4=right).

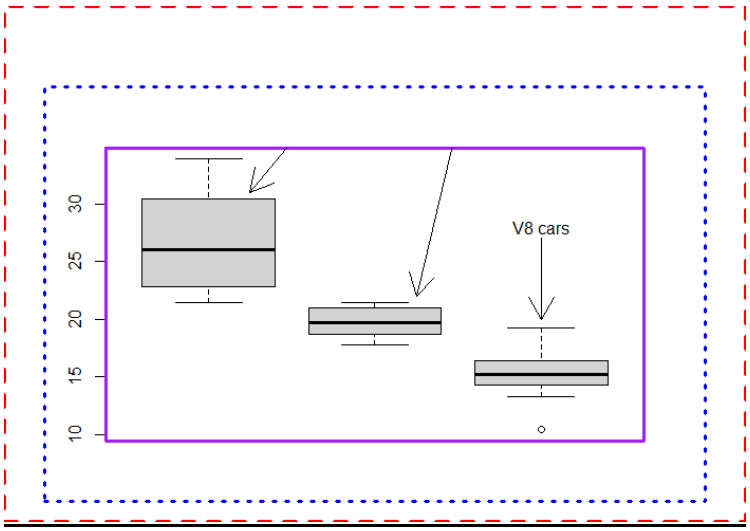
c) Clipping

- Controlling clipping allows you to draw in or add elements to the margin regions with reference to the user coordinates of the plot itself.
- For example, you might want to place a legend outside the plotting area, or you might want to draw an arrow that extends beyond the plot region to embellish a particular observation.
- The graphical parameter xpd controls clipping in base R graphics.
- By default, xpd is set to FALSE, so all drawing is clipped to the available plot region only (with the exception of special margin-addition functions such as mtext).
- Setting xpd to TRUE allows you to draw things outside the formally defined plot region into the figure margins but not into any outer margins.
- Setting xpd to NA will permit drawing in all three areas—plot region, figure margins, and the outer margins.

Example 1

```
dev.new()
par(oma=c(1,2,4,2),mar=c(3,3,3,3) )
boxplot(mtcars$mpg~mtcars$cyl,xaxt="n",ylab="MPG")
box(which = "figure",lty=3, col ="blue",lwd=3)
box(which= "outer",lty=2, col="red",lwd=3)
box(which = "plot",col="purple",lwd = 3)
arrows(x0=c(2,2.5,3),y0=c(44,37,27),x1=c(1.25,2.25,3),y1=c(31,22,20), xpd=FALSE)
text(x=c(2,2.5,3),y=c(45,38,28),c("V4 cars","V6 cars","V8 cars"), xpd=FALSE)
```

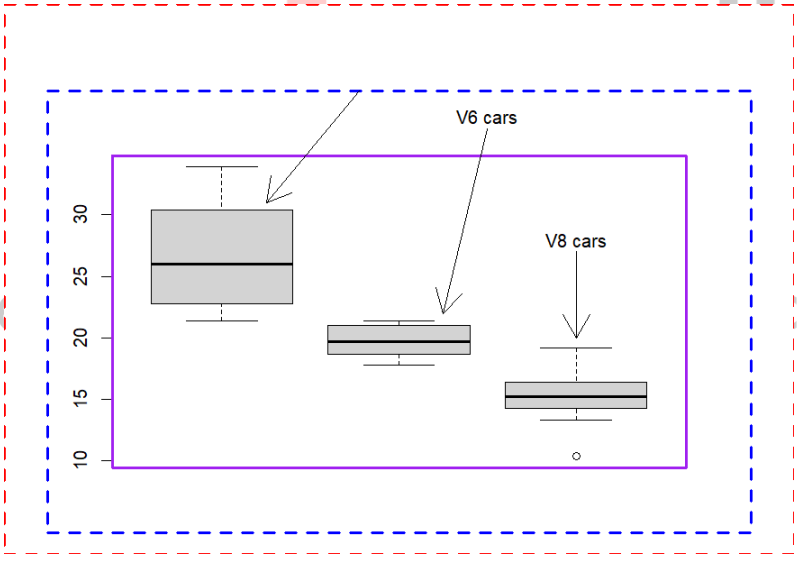
Output



Example 2

```
dev.new()
par(oma=c(1,2,4,2),mar=c(3,3,3,3) )
boxplot(mtcars$mpg~mtcars$cyl,xaxt="n",ylab="MPG")
box(which = "figure",lty=2, col ="blue",lwd=3)
box(which= "outer",lty=2, col="red",lwd=3)
box(which = "plot",col="purple",lwd = 3)
arrows(x0=c(2,2.5,3),y0=c(44,37,27),x1=c(1.25,2.25,3),y1=c(31,22,20), xpd=TRUE)
text(x=c(2,2.5,3),y=c(45,38,28),c("V4 cars","V6 cars","V8 cars"),  xpd=TRUE)
```

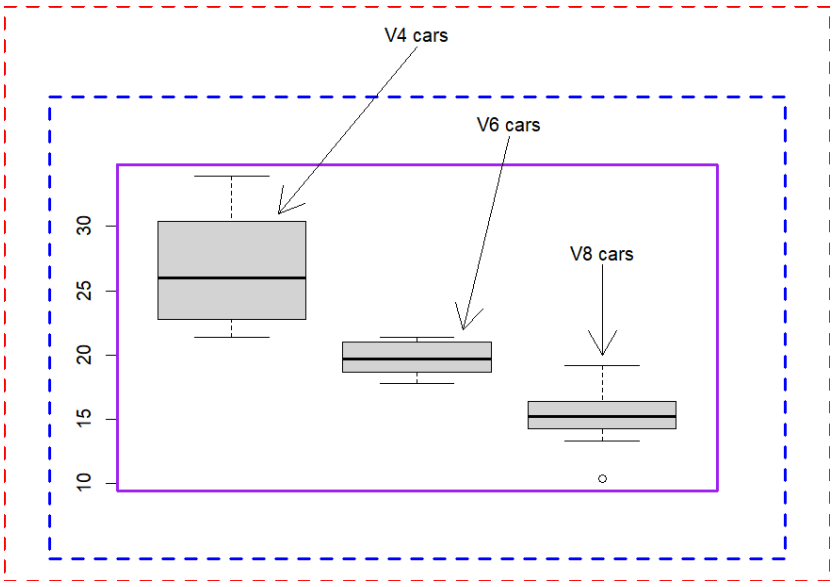
Output



Example 3

```
par(oma=c(1,2,4,2),mar=c(3,3,3,3) )
boxplot(mtcars$mpg~mtcars$cyl,xaxt="n",ylab="MPG")
box(which = "figure",lty=2, col ="blue",lwd=3)
box(which= "outer",lty=2, col="red",lwd=3)
box(which = "plot",col="purple",lwd = 3)
arrows(x0=c(2,2.5,3),y0=c(44,37,27),x1=c(1.25,2.25,3),y1=c(31,22,20), xpd=NA)
text(x=c(2,2.5,3),y=c(45,38,28),c("V4 cars","V6 cars","V8 cars"), xpd=NA)
```

Output



3. Point-and-Click Coordinate Interaction

- Your dealings with the graphics device don't need to be solely command based. Under typical circumstances, R can read mouse clicks you make inside the device.

a. Retrieving Coordinates Silently

- The locator() function is used to interactively locate points on a plot. It allows users to click on a plot with the mouse, and the function returns the coordinates (x, y) of the point clicked.
- This can be useful for identifying specific data points or for interactive data exploration.

Steps for retrieving coordinates using locator

1. Create Simple plot
   - Generate a simple plot with a single point (or any plot of interest).
2. Use locator function
   - Execute the locator() function with no arguments to enable default behavior. This will "hang" the console, and your mouse cursor will change to a plus symbol (+).
3. Click Left Mouse Button
   - Move the cursor to the desired location on the plot.
   - Click the left mouse button to record the coordinates.
   - R silently stores the coordinates for each click.
4. Stop Locator Function:
   - Press the "Esc" key on your keyboard to stop the locator() function.
   - This terminates the interactive coordinate recording

Example

```
plot(chickwts$weight)
point <- locator()
cat("X axis",point$x,"\n")
cat("Y axis",point$y)
```

Output

```
> source("D:/ICCA/R_Program/advance_graphics.R")
X axis 6.360066 13.6273 6.671519 18.09146 27.22741
Y axis 351.1619 179.8381 377.3849 382.6295 336.3022
```

b. Visualizing Selected Coordinates

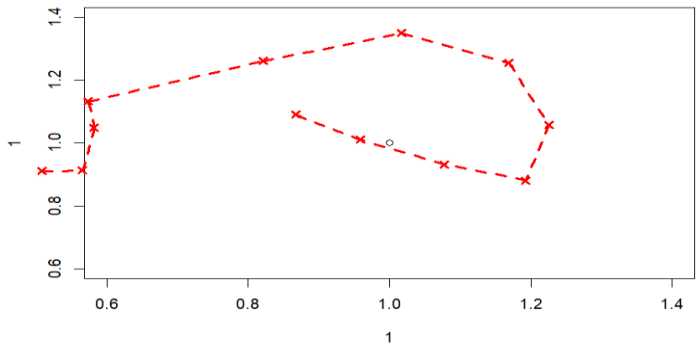
- You can also use locator to plot the points you select as either individual points or as lines.

Example

```
plot(1,1)
Rtist <- locator(type="o",pch=4,lty=2,lwd=3,col="red",xpd=TRUE)
print(Rtist)
```

Output

```
> source("D:/ICCA/R_Program/advance_graphics.R")
$x
 [1] 0.5082010 0.5651524 0.5817633 0.5734578 0.8214337 1.0172041 1.1690744 1.2260258 1.1928042
[10] 1.0777149 0.9590662 0.8677067

$y
 [1] 0.9112025 0.9134224 1.0488386 1.1331963 1.2619527 1.3507503 1.2552929 1.0577184 0.8801233
[10] 0.9311819 1.0110997 1.0910175
```



- Selecting type="o" (as opposed to the silent default, type="n") is what produces the overplotted points and lines.

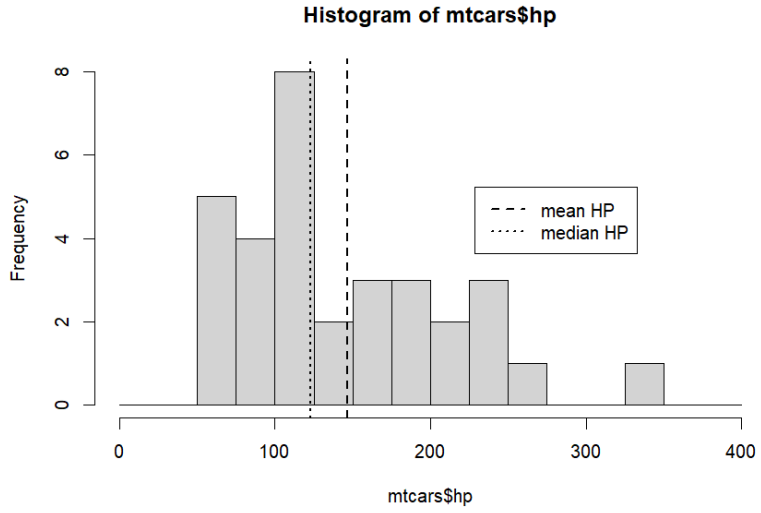
c. Ad Hoc Annotation

- The locator() function in R not only allows you to interactively capture user coordinates but also provides a convenient way to position ad hoc annotations such as legends, labels, or any other graphical elements on your plot.

Example

```
hist(mtcars$hp,breaks = seq(0,400,25))
abline(v=c(mean(mtcars$hp),median(mtcars$hp)),lty = c(2,3),lwd= 2 )
legend(locator(n=1),legend=c("mean HP","median HP"),lty=c(2,3),lwd=2)
```

Output



- An optional argument to locator, n, takes a positive integer for an upper limit on how many points you want to select; it defaults to 512.
- If you specify n=1, locator will automatically terminate after you left-click once in the device, so you don't need to manually exit the function with a right-click.
- When the code is executed, the + cursor will appear on the graphics device, and you simply need to click once for the desired location of the legend.

4. Customizing Traditional R plots

a. Graphical Parameters for Style and Suppression

i. Style parameters

- There are two axis "styles," controlled by the graphical parameters xaxs and yaxs.
- The xaxs parameter in the plot() function in R is used to control the extent of the x-axis data limits. It specifies the style of axis interval calculation when plotting points or lines.
- The yaxs parameter in the plot() function in R is used to control the extent of the y-axis data limits. It specifies the style of axis interval calculation when plotting points or lines.
- The xaxs parameter accepts the following values
  - "r": (default) This setting extends the data range by a small fraction on each side of the data range. It ensures that the data points are not plotted directly on the axis but leaves a small margin. This is the default behavior.
  - "i": This setting makes the x-axis extend exactly to the minimum and maximum values of the data. It does not add any margin.

Example

```
par(mfrow=c(1,2))
plot(chickwts$weight,main = "Default plot")
plot(chickwts$weight,xaxs="i",yaxs="i",main="Custom x and y axis")
par(mfrow=c(1,1))
```

Output

ii. Suppressing parameters

- There are two alternative ways of suppressing default elements in a plot:

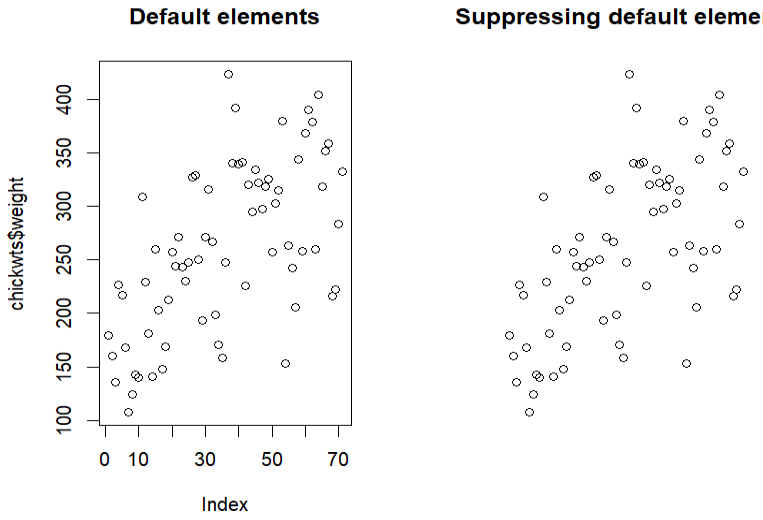
Method 1: Using xaxt, yaxt, and bty Parameters:

- This method involves setting specific parameters to "n" to suppress various elements.
  - xaxt="n": Suppresses the x-axis.
  - yaxt="n": Suppresses the y-axis.
  - bty="n": Suppresses the box around the plot.
  - xlab="" and ylab="": Empty strings for x-axis and y-axis labels.

Example

```
par(mfrow=c(1,2))
plot(chickwts$weight,main="Default elements" )
plot(chickwts$weight,bty="n",xaxt = "n",yaxt = "n",xlab="",ylab="",main="Suppressing default elements")
par(mfrow=c(1,1))
```

Output



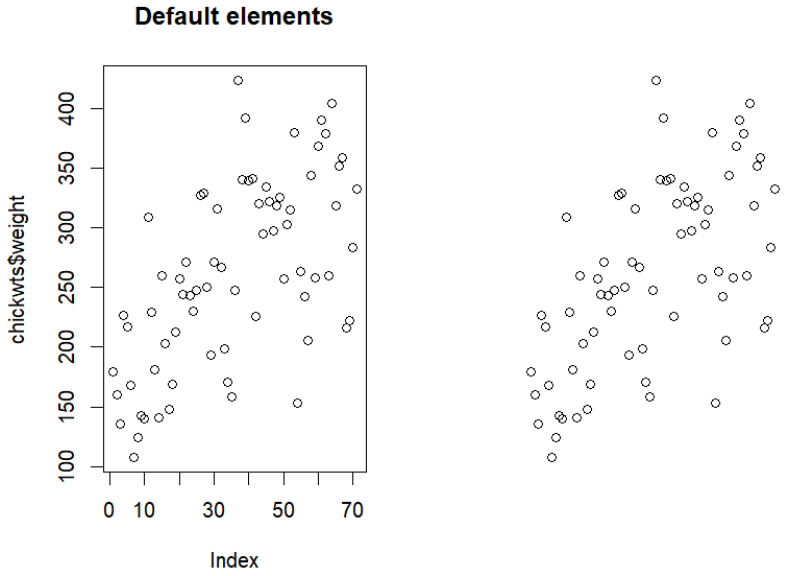
Method 2: Using axes=FALSE and ann=FALSE Parameters:

- This method involves setting axes=FALSE to suppress axes and ann=FALSE to suppress annotations (including the box).
- The ann parameter in the plot() function in R controls whether annotations such as axis labels, title, and other decorations are added to the plot. It stands for "annotations." If ann is set to TRUE (the default), these annotations are added; if set to FALSE, they are suppressed, allowing you to start with a blank canvas.
- The axes parameter in the plot() function in R is used to control the plotting of axes in a plot. Specifically, it determines whether or not axes are drawn in the plot.

Example

```
par(mfrow=c(1,2))
plot(chickwts$weight,main="Default elements" )
plot(chickwts$weight,axes=FALSE,ann = FALSE)
par(mfrow=c(1,1))
```

Output



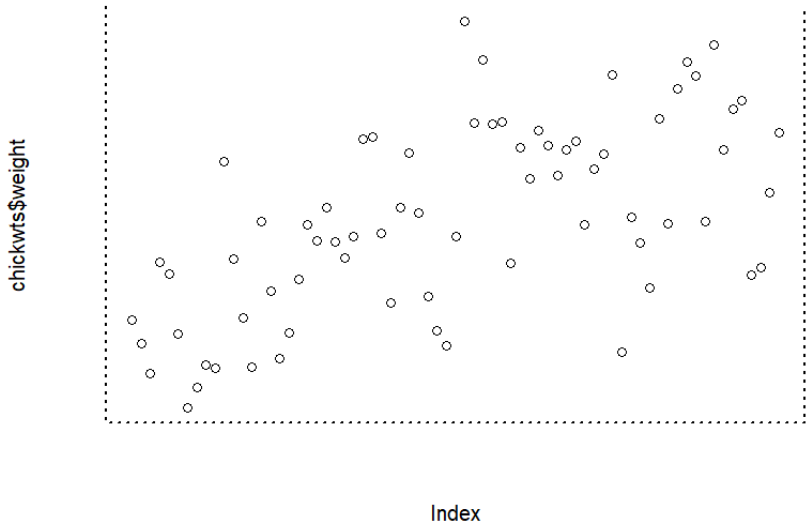
b. Customizing Box

- When you create a plot in R and choose to start with a suppressed-box or suppressed-axis plot—where the default drawing of either the box around the plot or the axes is omitted—you can add a custom box to the current plot region using the box() function.
- The bty parameter within the box() function allows you to specify the type of box you want to draw, giving you the flexibility to customize the appearance of the box.

Example

```
plot(chickwts$weight,axes=FALSE)
box(bty="u",lty=3,lwd=2)
```

Output



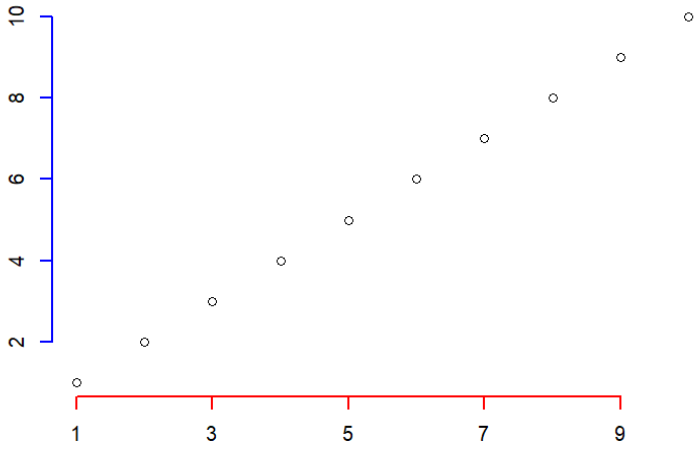
c. Customizing Axes

- The axis function allows you to control the addition and appearance of an axis on any of the four sides of the plot region in greater detail.
- Syntax: - **axis(side, at = NULL, ...)**
- Where,
  - side: Specifies which side of the plot the axis should be drawn on (1 for bottom, 2 for left, 3 for top, 4 for right).
  - at: The positions of tick marks on the axis.

Example

```
plot(1:10,axes=FALSE,ann=FALSE)
axis(side= 1,at=c(1,3,5,7,9),lwd=2,col="red")
axis(side= 2,at=c(2,4,6,8,10),lwd=2,col="blue")
```

Output



5. Specialized Text and Label Notation

- Let's explore immediately accessible tools for controlling fonts and displaying special notation, such as Greek symbols and mathematical expressions

a. Font

- The displayed font is controlled by two graphical parameters: family for the specific font family and font.

➢ Font Family (family):
  - The family parameter controls the specific font family used for text in the plot.
  - The three generic font families are:
    - "sans" (sans-serif, usually the default),
    - "serif" (serif, with small decorative flourishes at the ends of characters),
    - "mono" (monospaced or fixed-width).
  - You can set the font family using the family parameter in graphical functions like text() or universally for a device using the par() function.

➢ Font Style (font):
  - The font parameter is an integer selector that controls the style of the font.
  - Possible values for font include:
    1: Normal text (default).
    2: Bold.
    3: Italic.
    4: Bold and italic.
  - You can set the font style using the font parameter in graphical functions like text() or universally for a device using the par() function.

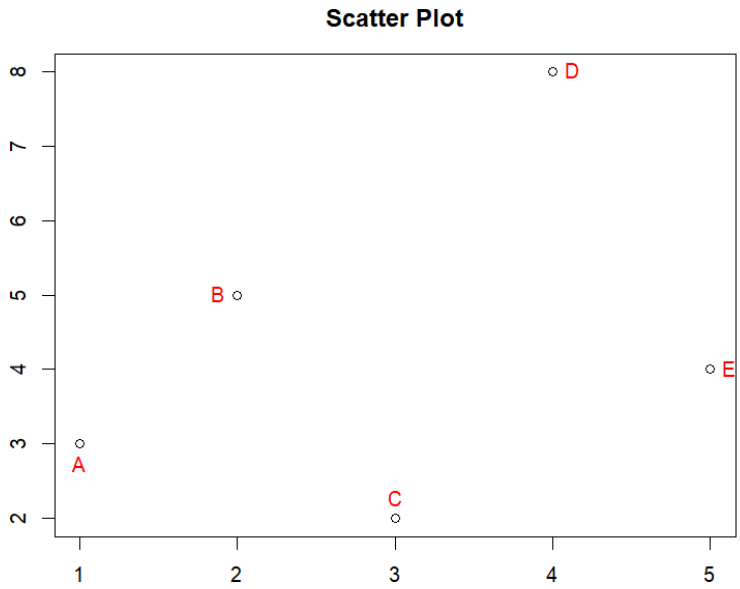
1. **text()**: - text() function is used to add text to a plot. It is commonly used in conjunction with plotting functions like plot() to annotate specific points or regions on a graph with text.

- Syntax: - **text(x, y, labels, pos = NULL, offset = 0.5, vfont = NULL, cex = 1, col = NULL, ...)**
- Where,
  - x: The x-coordinates of the text labels.
  - y: The y-coordinates of the text labels.
  - labels: A vector of character strings containing the text labels.
  - pos: The position of the text relative to the specified coordinates. It can take values 1, 2, 3, or 4 (see details below).
  - offset: A numeric value specifying the offset of the text from the specified coordinates. Default is 0.5.
  - vfont: A vector specifying the font properties (family, face, and size) of the text.
  - cex: A numerical value indicating the amount by which the text should be scaled relative to the default.
  - col: The color of the text.
  - The pos parameter can take the following values:
    - pos = 1: below
    - pos = 2: to the right
    - pos = 3: above
    - pos = 4: to the right

Example

```
x <- 1:5
y <- c(3, 5, 2, 8, 4)
plot(x, y, main = "Scatter Plot", xlab = "X-axis", ylab = "Y-axis")
# Add text to specific points on the plot
text(x, y, labels = c("A", "B", "C", "D", "E"), pos = c(1,2,3,4,4), col = "red")
```

Output



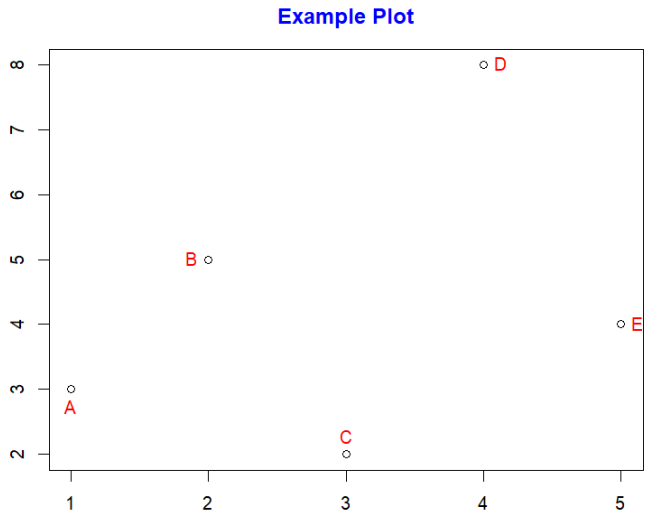
2. mtext: -

- The mtext() function in R is used to add text to the margins of a plot. It is commonly used to add titles, axis labels, or other annotations to the margins of a graph.
- Syntax: - **mtext(text, side = 3, line = 0, outer = FALSE, at = NA, adj = NA, padj = NA, cex = 1, col = NA, font = NA, ...)**
- Where,
    - text: The text to be displayed.
    - side: An integer specifying which margin to place the text on. It can take values 1, 2, 3, or 4 for bottom, left, top, or right margins, respectively.
    - line: The line number on the specified margin where the text should be placed.
    - outer: Logical. If TRUE, the text is placed in the outer margin; if FALSE, it is placed in the inner margin.
    - at: The numeric position where the text should be placed along the specified margin. If NA, the default position is used.
    - adj: Adjustment of text relative to the specified position. Values closer to 0 position the text to the left, while values closer to 1 position it to the right.
    - padj: Vertical adjustment of text. Values closer to 0 position the text lower, while values closer to 1 position it higher.
    - cex: A numerical value indicating the amount by which the text should be scaled relative to the default.
    - col: The color of the text.
    - font: An integer specifying the font style (e.g., 1 for normal, 2 for bold).

Example

```
x <- 1:5
y <- c(3, 5, 2, 8, 4)
# Create a scatter plot
plot(x, y, ann = FALSE)
# Add text to specific points on the plot
text(x, y, labels = c("A", "B", "C", "D", "E"), pos = c(1,2,3,4,4), col = "red")
# Add a title to the top margin
mtext("Example Plot", side = 3, line = 1, cex = 1.2, col = "blue", font = 2)
```

Output



b. Greek Symbols

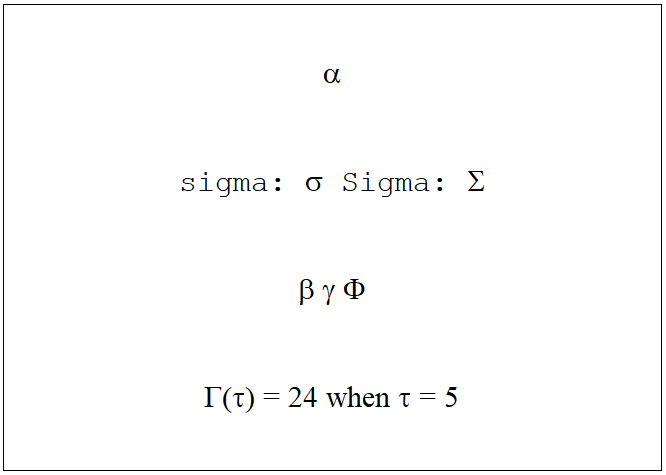
- In R, the expression() function is used for creating mathematical annotations and expressions in plots.
- This is particularly useful for adding Greek symbols, mathematical formulas, or special characters to your plots.
- The expression() function returns an object of class "expression," and it can be passed to various plotting functions that accept character strings for annotation.

Example

```
par(mar=c(3,3,3,3))
plot(1,1,type="n",xlim=c(-1,1),ylim=c(0.5,4.5),xaxt="n",yaxt="n",
      ann=FALSE)
text(0,4,label=expression(alpha),cex=1.5)
text(0,3,label=expression(paste("sigma: ",sigma,"
 Sigma: ",Sigma)),
      family="mono",cex=1.5)
text(0,2,label=expression(paste(beta," ",gamma," ",Phi)),cex=1.5)
text(0,1,label=expression(paste(Gamma,"(",tau,") = 24 when ",tau," = 5")),
      family="serif", cex=1.5)
title(main=expression(paste("Gr",epsilon,epsilon,"k")),cex.main=2)
```


Output

Greek



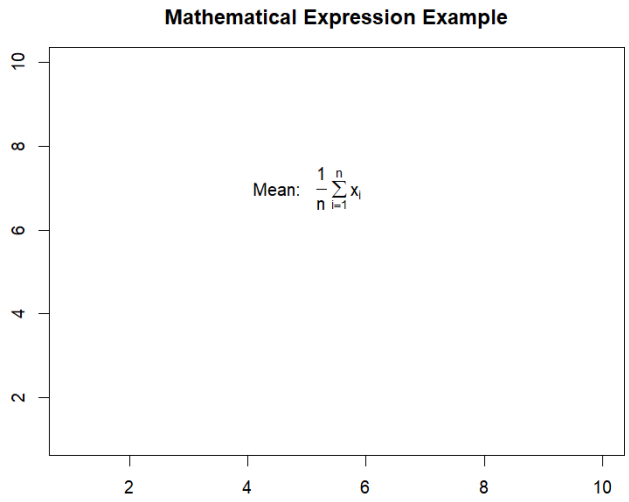
c. Mathematical Expressions

- Formatting entire mathematical expressions in R plots, especially for complex equations or expressions, can be more involved and often resembles the syntax used in markup languages like LaTeX. The expression() function in R supports a subset of LaTeX-like syntax for mathematical expressions.
- Here's a brief overview of some common elements in R plotmath expressions:
- Superscripts and Subscripts:
  - Use ^ for superscripts and _ for subscripts.
  - For example, expression(x^2) represents x squared.
- Greek Symbols:
  - Use the corresponding Greek letters, e.g., alpha, beta, gamma, etc.
  - For example, expression(alpha + beta) represents the sum of alpha and beta.
- Mathematical Operators:
  - Use standard mathematical operators such as +, -, *, /, ==, etc.
  - For example, expression(a * b == c) represents the equation "a times b equals c."
- Roots:
  - Use sqrt() for square roots.
  - For example, expression(sqrt(x)) represents the square root of x.
- Fractions:
  - Use frac() for fractions.
  - For example, expression(frac(a, b)) represents the fraction "a over b."
- Grouping:
  - Use {} for grouping expressions.
  - For example, expression(2 * (x + y)) represents "2 times the quantity x plus y."

Example

```
plot(1:10, 1:10, type = "n", main = "Mathematical Expression Example")
text(5,7,label= expression(paste("Mean:   ",frac(1,n)," ",sum(x[i],i==1,n))))
```

Output



Defining Colors and Plotting in Higher Dimensions

- Now that we've mastered some fundamental visualization skills, you can go beyond the standard x- and y-axes by, for example, coloring points according to some additional value or variable or adding a z-axis for constructing a 3D plot.
- Higher-dimensional plots like this allow us to visually explore your data or models using more variables than would be possible otherwise.

1. Representing and Using Color

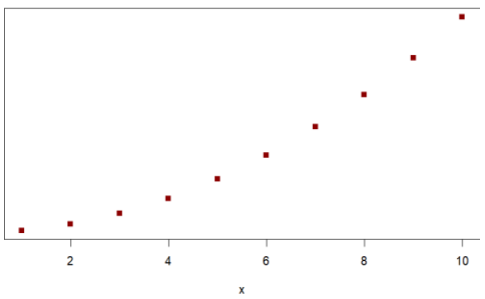
a. Red-Green-Blue Hexadecimal Color Codes

- The color specification in R, particularly using the RGB (Red, Green, Blue) system.
- RGB Color Specification:
  - RGB is a color model where colors are formed by combining different intensities of red, green, and blue components.
  - Each component is represented by an integer ranging from 0 to 255, where 0 is the minimum intensity (absence of color), and 255 is the maximum intensity (full color).
  - Example RGB Triplets: (0, 0, 0): Pure black, (255, 255, 255): Pure white, (0, 255, 0): Full green
  - When using plotting functions like plot(), points(), lines(), etc., you can use the col argument to specify the color of points, lines, or shapes.
  - The rgb() function is often used to create custom colors for these elements.

Example

```
x <- 1:10
y <- x^2
plot(x, y, col = rgb(139, 0, 0, maxColorValue = 255),pch=15)  # Darkred
```

Output

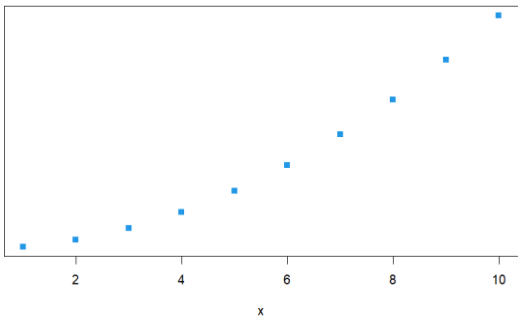


- Color Representation in R:
  - In R, colors can be specified using color names, integer values from 1 to 8, or RGB triplets.
  - The palette() function in R returns the default color palette, which consists of eight colors: "black," "red," "green," "blue," "cyan," "magenta," "yellow," and "gray."
  - Users can use the col argument with integer values from 1 to 8 to select one of these default colors

Example

```
x <- 1:10
y <- x^2
plot(x,y,col=4,pch=15)
```

Output



RGB Values for Named Colors:

- Users can find the RGB values for named colors using the col2rgb function.

Example

```
> col2rgb(c("black","green","white","pink","darkred"))
      [,1] [,2] [,3] [,4] [,5]
red      0    0  255  255  139
green    0  255  255  192    0
blue     0    0  255  203    0
```

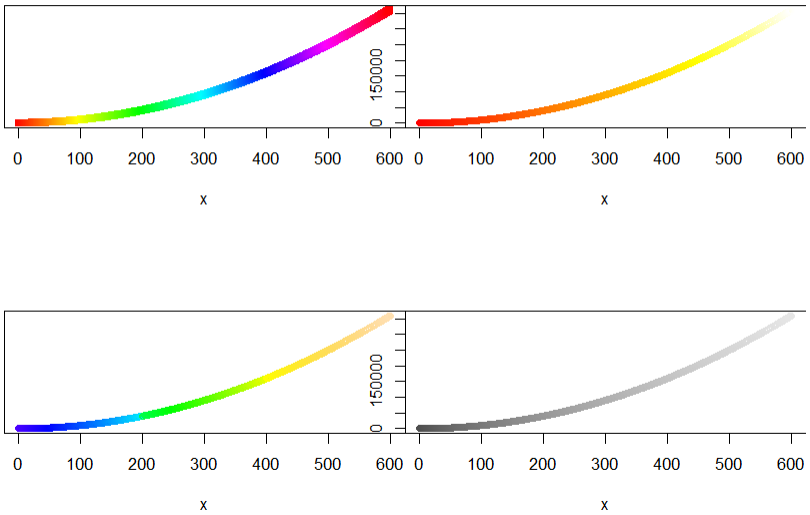

b. Built-in palettes

- There are a number of color palettes built into the base R installation
- Color Palette Functions:
  - The base R installation provides several color palette functions, including:
    - rainbow()
    - heat.colors()
    - terrain.colors()
    - topo.colors()
    - cm.colors()
    - gray.colors()
    - gray()
  - These functions allow you to generate a sequence of colors that are evenly spaced across the entire color range of the palette.
- Generating Color Palettes:
  - You can use these functions to generate a sequence of colors by specifying the number of colors (N) you want in the palette.
  - The resulting colors are returned as a character vector of hexadecimal codes.

Example

```
N <- 600
rbow <- rainbow(N)
heat <- heat.colors(N)
terr <- terrain.colors(N)
topo <- topo.colors(N)
cm <- cm.colors(N)
gry1 <- gray.colors(N)
gry2 <- gray(level = seq(0, 1, length = N))
par(mfrow=c(2,2))
x <- 1:N
y <- x^2
plot(x,y,col=rbow,pch=15)
plot(x,y,col=heat)
plot(x,y,col=topo)
plot(x,y,col=gry1)
par(mfrow=c(1,1))
```

Output



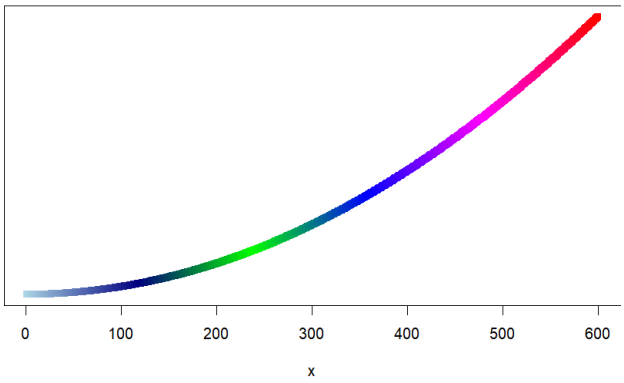
c. Custom palettes

- The colorRampPalette function in R allows you to create custom color palettes by specifying two or more key colors.
- The resulting palette is a function that transitions smoothly between the specified colors.

Example

```
custom_cols<- colorRampPalette(colors =
                                c("lightblue","navyblue","green","blue","magenta","red"))
N=600
cus_palette <- custom_cols(N)
x <- 1:N
y <- x^2
plot(x,y,col=cus_palette,pch=15)
```

Output



d. Using Color Palettes to Index a Continuum

- Using color palettes to index a continuum involves assigning colors to values on a continuous scale.
- This process is useful for visualizing and interpreting data with a continuous variable. There are two methods
  - Through categorization of your continuous values
  - Through normalization of your continuous values

i. Via Categorization

- This method involves dividing the continuous variable into distinct categories or bins.
- Each category is assigned a different color, and observations falling within a specific category are represented with the corresponding color.
- This approach simplifies the representation of a continuous variable, making it easier to interpret, especially when the emphasis is on identifying patterns or trends within specific ranges.
- Here's a step-by-step explanation:

1. Create the Data frame
   - surv <- na.omit(survey[, c("Wr.Hnd", "NW.Hnd", "Height")])
     - This line creates a new data frame called surv with only the columns "Wr.Hnd," "NW.Hnd," and "Height." The na.omit function removes any rows with missing values

2. Choose a Color Palette:
   - NW.pal <- colorRampPalette(colors = c("red4", "yellow2"))
     - This line creates a color palette (NW.pal) using the colorRampPalette function, going from dark red ("red4") to faded yellow ("yellow2").
3. Specify the Number of Bins (k):
   - k <- 5
4. Generate Colors from the Palette:
   - ryc <- NW.pal(k)
     - This line generates a sequence of colors (ryc) based on the color palette and the specified number of bins.
5. Define Breakpoints for Bins:
   - NW.breaks <- seq(min(surv$NW.Hnd), max(surv$NW.Hnd), length = k + 1)
     - This line creates breakpoints (NW.breaks) for the bins using the seq function, spanning the range of nonwriting handspans.
6. Categorize Continuous Values into Bins:
   - NW.fac <- cut(surv$NW.Hnd, breaks = NW.breaks, include.lowest = TRUE)

- The cut function categorizes the nonwriting handspans into bins based on the specified breakpoints.

7. Map Colors to Categories:

- NW.cols <- ryc[as.numeric(NW.fac)]
  - This line assigns colors to each observation based on the categorized nonwriting handspans.

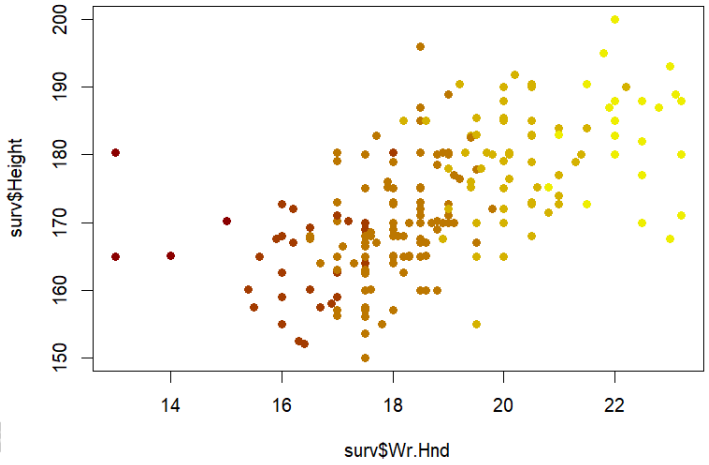
8. Plotting the Data:

- plot(surv$Wr.Hnd,surv$Height,col=NW.cols,pch=19)

Example

```
library(MASS)
surv <- na.omit(MASS::survey[, c("Wr.Hnd", "NW.Hnd", "Height")])
NW.pal <- colorRampPalette(colors = c("red4", "yellow2"))
k <- 5
ryc <- NW.pal(k)
NW.breaks <- seq(min(surv$NW.Hnd), max(surv$NW.Hnd), length = k + 1)
NW.fac <- cut(surv$NW.Hnd, breaks = NW.breaks, include.lowest = TRUE)
NW.cols <- ryc[as.numeric(NW.fac)]
plot(surv$Wr.Hnd,surv$Height,col=NW.cols,pch=19)
```

Output



ii Via Normalization

- Normalization involves scaling the continuous variable to a standard range, often between 0 and 1.
- Colors are then assigned based on the normalized values, with low values corresponding to one color and high values to another.
- This method is useful when you want to emphasize relative differences between values rather than absolute values.
- Common normalization techniques include Min-Max scaling or Z-score normalization.
- Here's a step-by-step explanation:

1. Normalization Function:

```
normalize <- function(datavec){
  lo <- min(datavec, na.rm=TRUE)
  up <- max(datavec, na.rm=TRUE)
  datanorm <- (datavec - lo) / (up - lo)
```

```
  return(datanorm)
}
```

- The normalize function takes a vector of data (datavec) as its argument, calculates the minimum (lo) and maximum (up) values, and then normalizes the data using the formula (datavec - lo) / (up - lo) to bring the values to a standardized range between 0 and 1.

2. Applying Normalization to Nonwriting Handspan Data:

```
surv$NW.Hnd
normalize(surv$NW.Hnd)
```

- The original nonwriting handspan values are displayed, and the normalize function is applied to these values to obtain normalized values between 0 and 1.

3. Creating a Color Palette using colorRamp:

```
NW.pal2 <- colorRamp(colors = c("red4", "yellow2"))
```

- The colorRamp function is used to create a color palette (NW.pal2) that goes from a dark red to a faded yellow, similar to the gray palette.

3. Generating Colors based on Normalized Data:

```
ryc2 <- NW.pal2(normalize(surv$NW.Hnd))
```

- The normalized nonwriting handspan data is used to generate colors using the color palette created with colorRamp.

4. Converting RGB Triplets to Hex Codes:

```
NW.cols2 <- rgb(ryc2, maxColorValue = 255)
```

- The RGB triplets obtained from the colorRamp function are converted to hex codes using the rgb function.

5. Plotting the Data:

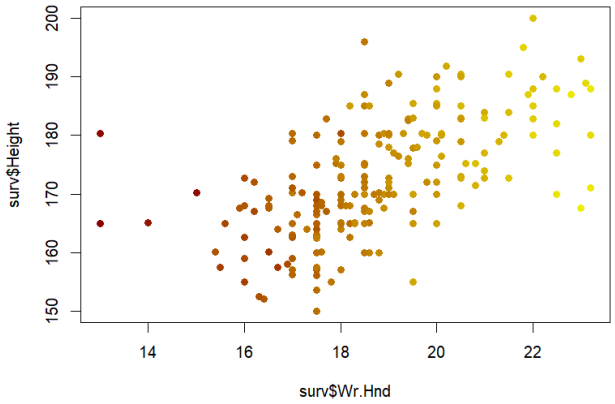
```
plot(surv$Wr.Hnd, surv$Height, col = NW.cols2, pch = 19)
```

- The data is plotted with colors assigned based on the normalized nonwriting handspan values, using the hex codes obtained.

Example

```
library(MASS)
normalize <- function(datavec){
  lo <- min(datavec, na.rm=TRUE)
  up <- max(datavec, na.rm=TRUE)
  datanorm <- (datavec - lo) / (up - lo)
  return(datanorm)
}
surv$NW.Hnd
normalize(surv$NW.Hnd)
NW.pal2 <- colorRamp(colors = c("red4", "yellow2"))
ryc2 <- NW.pal2(normalize(surv$NW.Hnd))
NW.cols2 <- rgb(ryc2, maxColorValue = 255)
plot(surv$Wr.Hnd, surv$Height, col = NW.cols2, pch = 19)
```

Output



e. Including a color legend

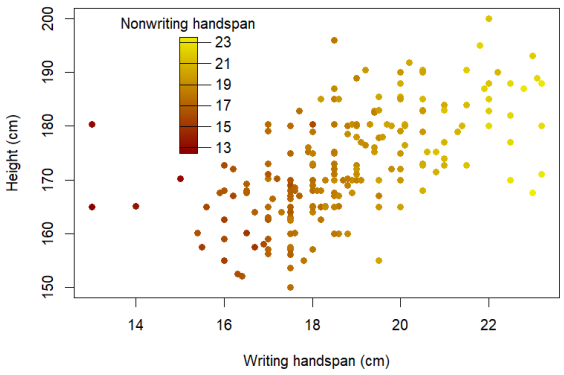
Legends play a crucial role in helping interpret color scales in plots. While we can create legends using base R tools, using contributed functionality from R packages often provides a simpler and more flexible approach.

One useful function for this is the colorlegend command. This is found in the shape package so first download and install shape from CRAN.

Example 1

```
library(MASS)
library(shape)
normalize <- function(datavec){
  lo <- min(datavec, na.rm=TRUE)
  up <- max(datavec, na.rm=TRUE)
  datanorm <- (datavec - lo) / (up - lo)
  return(datanorm)
}
surv$NW.Hnd
normalize(surv$NW.Hnd)
NW.pal2 <- colorRamp(colors = c("red4", "yellow2"))
ryc2 <- NW.pal2(normalize(surv$NW.Hnd))
NW.cols2 <- rgb(ryc2, maxColorValue = 255)
plot(surv$Wr.Hnd,surv$Height,col=NW.cols2,pch=19,
    xlab="Writing handspan (cm)",ylab="Height (cm)")
colorlegend(NW.pal(200),zlim=range(surv$NW.Hnd),zval=seq(13,23,by=2),
          posx=c(0.3,0.33),posy=c(0.5,0.9),main="Nonwriting handspan")
```

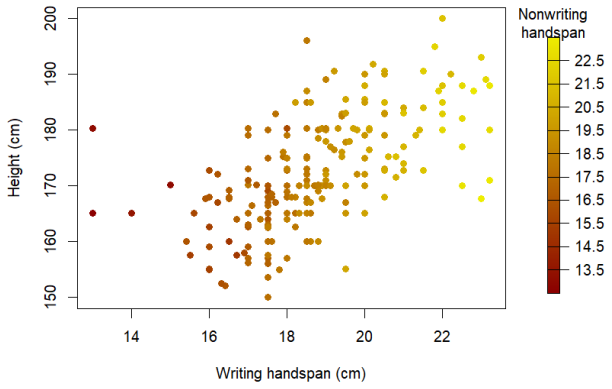
Output



Example 2

```
library(MASS)
library(shape)
normalize <- function(datavec){
  lo <- min(datavec, na.rm=TRUE)
  up <- max(datavec, na.rm=TRUE)
  datanorm <- (datavec - lo) / (up - lo)
  return(datanorm)
}
surv$NW.Hnd
normalize(surv$NW.Hnd)
NW.pal2 <- colorRamp(colors = c("red4", "yellow2"))
ryc2 <- NW.pal2(normalize(surv$NW.Hnd))
NW.cols2 <- rgb(ryc2, maxColorValue = 255)
par(mar=c(5,4,4,6))
plot(surv$Wr.Hnd,surv$Height,col=NW.cols2,pch=19,
     xlab="Writing handspan (cm)",ylab="Height (cm)")
colorlegend(NW.pal(200),zlim=range(surv$NW.Hnd),zval=13.5:22.5,digit=1,
          posx=c(0.89,0.91),main="Nonwriting\nhandspan")
```

Output



f. Opacity

- The ability to specify the opacity (alpha channel) of colors is indeed a useful skill in creating visually appealing and informative plots. In R, this can be achieved using the alpha parameter in various functions that provide color information

1. Opacity in Hex Codes:

- The alpha parameter, when used in functions providing hex codes, allows you to set the opacity of the colors.
- By default, R assumes full opacity when creating colors.
- When you explicitly set opacity using the alpha parameter, the format of hex codes changes slightly.
- Instead of the usual six characters after the #, eight characters will appear, with the last two containing the additional opacity information.

Example

```
> source("D:/ICCA/R_Program/datavisual.R")
> rgb(cbind(255,0,0),maxColorValue=255)
[1] "#FF0000"
> rgb(cbind(255,0,0),maxColorValue=255,alpha=0)
[1] "#FF000000"
> rgb(cbind(255,0,0),maxColorValue=255,alpha=0)
[1] "#FF000000"
> rgb(cbind(255,0,0),maxColorValue=255,alpha=255)
[1] "#FF0000FF"
```

- Note that the first and last colors are identical; it's just that the last hex code explicitly specifies full opacity.

2 Opacity in adjustcolor function

- We can always adjust the opacity of any color with the alpha.f argument (which takes values in the range 0 through 1) of the ready-to-use adjustcolor function.

Example

```
> adjustcolor(rgb(cbind(255,0,0),maxColorValue=255))
[1] "#FF0000FF"
> adjustcolor(rgb(cbind(255,0,0),maxColorValue=255),alpha.f=0.4)
[1] "#FF000066"
> adjustcolor(rgb(cbind(255,0,0),maxColorValue=255),alpha.f=0.8)
[1] "#FF0000CC"
```

g. RGB Alternatives and Further Functionality.

- R provides various ways to represent colors, and besides RGB triplets, two other common color representations are Hue-Saturation-Value (HSV) and Hue-Chroma-Luminance (HCL).
- These alternative representations can be useful, especially when you want to work with different color models or have specific requirements for your visualizations. Here's a brief overview of these color representations:
- HSV (Hue-Saturation-Value):
  - The hsv() function in R allows you to specify colors using the HSV color model.
  - Hue represents the type of color (e.g., red, green, blue), Saturation controls the intensity or vividness of the color, and Value controls the brightness.
- HCL (Hue-Chroma-Luminance):
  - The hcl() function in R allows you to specify colors using the HCL color model.
  - Hue represents the type of color, Chroma controls the intensity or saturation, and Luminance controls the brightness.
- Colorspace Package:
  - The colorspace package provides functionality for translating between different color formats, including RGB, HSV, and HCL. It allows you to convert colors between these representations.
- RColorBrewer Package:
  - The RColorBrewer package is based on color schemes designed by Cynthia Brewer and provides more options for creating color palettes.

2. 3D Scatterplots

- Creating 3D scatterplots in R allows you to visualize relationships among three continuous variables simultaneously.
- One common and popular package for creating 3D scatterplots is the scatterplot3d package, developed by Ligges and Mächler.
- Before using the scatterplot3d package, you need to install and load it. You can install it from CRAN using the following: install.packages("scatterplot3d")

- After installation, load the package: library(scatterplot3d)

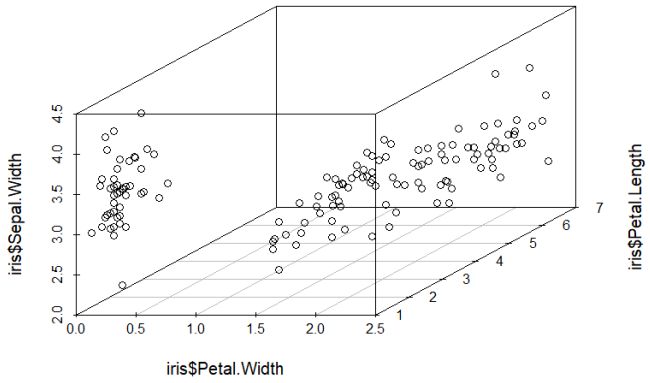
a. Basic Syntax

- The syntax of the scatterplot3d function is similar to the default plot function.
- In the latter, you supply a vector of x- and y-axis coordinates; in the former, you merely supply an additional third vector of values providing the z-axis coordinates.
- With that additional dimension, we can think of these three axes in terms of the x-axis increasing from left to right, the y-axis increasing from foreground to background, and the z-axis increasing from bottom to top.

Example

```
library(scatterplot3d)
scatterplot3d(iris$Petal.Width,iris$Petal.Length,iris$Sepal.Width)
```

Output



b. Visual Enhancement

- The scatterplot3d function in R provides options to enhance the perception of depth in a 3D scatterplot.

1. Coloring Points:
   - Coloring the points based on their position in the z-axis helps to distinguish between foreground and background.
   - The highlight.3d parameter is set to TRUE to enable coloring.
2. Drawing Lines:
   - Setting the type parameter to "h" allows you to draw lines perpendicular to the x-y plane, providing additional visual cues.
   - The lty.hplot parameter controls the line type of the lines perpendicular to the x-y plane.
   - The lty.hide parameter controls the line type for hidden lines.

Example

```
library(scatterplot3d)
scatterplot3d(iris$Petal.Width,iris$Petal.Length,iris$Sepal.Width, highlight.3d =
TRUE, type = "h", lty.hplot = 2, lty.hide = 3, xlab = "Petal width", ylab = "Petal length",
zlab = "Sepal width", main = "Iris Flower Measurements")
```

Output