IT18015

STATISTICAL ANALYSIS USING R PROGRAMMING

PREPARED BY
DR A KALA
ASSOCIATE PROFESSOR/IT

COURSE OUTCOMES

1. Experiment with the various data structures such as matrices, lists, factors, and data frames
2. Infer knowledge on various file formats and create various graphic displays
3. Formulate and Solve the problems in probability distributions
4. Choose statistical models for analyzing the data
5. Investigate and handle missing data and infer knowledge on advanced graphics

UNIT I - INTRODUCTION

- Introduction to R-Basic Syntax-data Types-variables-Operators-Decision Making-Loops-Functions-Strings-Vectors-Lists-Matrices-Arrays- Factors-Data Frames-Packages-Data Reshaping.

Introduction to R

- **R** is a programming language developed by Ross Ihaka and Robert Gentleman in 1993.
- R possesses an extensive catalog of statistical and graphical methods. It includes machine learning algorithms, linear regression, time series, statistical inference.
- R is a interpreted programming language for data analysis and statistics.
- R is a software environment for statistical analysis ,graphics representation and reporting.
- R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.

What is R used for?

- **Data Analysis**

  - process of cleaning, transforming, and modeling **data** to discover useful information.

- **Statistical Inference**

  - process of analyzing the result and making conclusions from data subject to random variation.

- **Machine learning algorithm**

  - R is the most popular platform for machine learning.

  - It is very powerful because so many machine learning algorithms are provided.

Data analysis with R

Done in a series of steps - programming, transforming, discovering, modeling and communicate the results.

- **Program**: R is a clear and accessible programming tool
- **Transform**: R is made up of a collection of libraries designed specifically for data science
- **Discover**: Investigate the data, refine your hypothesis and analyze them
- **Model**: R provides a wide array of tools to capture the right model for your data
- **Communicate**: Integrate codes, graphs, and outputs to a report with R Markdown or build Shiny apps to share with the world

Features of R

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility.
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

Comments

- Comments are the programmer readable explanation.
- The purpose of adding these comments is to make the source code easier to understand.
- These comments are generally ignored by compilers and interpreters.
- In R programming there is only single-line comment.
- R doesn't support multi-line comment. But if we want to perform multi-line comments, then we can add our code in a false block.
- To comment a single line in RStudio, we can put “#” before each line.
- But when we want to comment multiple lines at one go, we can use following in RStuido:-

Comments - Example

Single line comment

- #My First program in R programming

Multi line comment

```
if(FALSE) {
   "R is an interpreted computer programming language which was created by  Ross Ihaka and Robert Gentleman at the Univ ersity of Auckland, New Zealand "
}
```

Print Statement

- Simply enter the variable name or expression, R will print its value.
- Use the print function for generic printing of any object.
- Use the cat function for producing custom formatted output.

pi

[1] 3.141593

sqrt(2)

[1] 1.414214

print(pi)

[1] 3.141593

print(sqrt(2))

[1] 1.414214

Print statement

Example 1:

print(“Hello, World!”)

Output:

[1] "Hello, World!“

Example 2:

mystring = “Hello, World!”

print(mystring)

Output:

[1] "Hello, World!"

Print statement

- Print format any R value for printing, including structured values such as matrices and lists:

Example 1:

```
print(matrix(c(1,2,3,4), 2, 2))
```

OUTPUT:

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Print statement

Example 2:

```
print(list("a","b","c"))
```

OUTPUT:

```
[[1]]
[1] "a"
[[2]]
[1] "b"
[[3]]
[1] "c"
```

Print statement

- The print function has a significant limitation, however: it prints only one object at a time. Trying to print multiple items gives error message:

Example1 :

print("The zero occurs at", 2*pi, "radians.")

Error in print.default("The zero occurs at", 2 * pi, "radians.") :

Example 2:

print(“hello”,2*pi)

Output:

[1] “hello”

Print statement

To print multiple items is to print them one at a time,

print("The zero occurs at"); print(2*pi); print("radians")

OUTPUT:

[1] "The zero occurs at"

[1] 6.283185

[1] "radians“

Built-in function paste() is used to concatenate strings.

print(paste("The zero occurs at", 2*pi, "radians."))

Output:

“The zero occurs at 6.283185 radians”

cat function

- The cat function lets you combine multiple items into a continuous output:
- An alternative to print function. cat puts a space between each item by default.
- A newline character(\n) must be provided to terminate the line.

cat("The zero occurs at", 2*pi, "radians.", "\n")

OUTPUT:

The zero occurs at 6.283185 radians.

- The cat function can print simple vectors, too:

fib <- c(0,1,1,2,3,5,8,13,21,34)

cat("The first few Fibonacci numbers are:", fib, "...\n")

OUTPUT:

The first few Fibonacci numbers are: 0 1 1 2 3 5 8 13 21 34 ...

Data Types

- In R, the variables are not declared as some data type.
- The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable.

- Numeric - 4.5 is a decimal value called **numerics**.
- Integer - 4 is a natural value called **integers**.
- Complex
- Logical - TRUE or FALSE is a Boolean value called **logical**.
- Character - The value inside " " or ' ' are text (string). They are called **characters**.

Numeric Data Type

- Decimal values are referred to as numeric data types in R.
- This is the default working out data type.
- If you assign a decimal value for any variable x like given below, x will become a numeric type.

Example:

```
a <- 85.4
print(a)
```

OUTPUT:

```
[1]85.4
```

class(a) – returns the type of variable

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

Integer Data Type

- To create an integer variable in R invoke the as.integer() function to define any integer type data.

Example:

```
s <- as.integer(3)  or s <- 3L
print(s)
```

OUTPUT:

```
[1] 3
class(s)
[1] "integer"
```

Complex Data Type

- To define a complex value in R the pure imaginary values 'i‘ is used.

Example:

```
k = 1 + 2i
print(k)
```

OUTPUT:

```
[1] 1+2i
```

Logical Data Type

- A logical value is created for comparison between variables .

Example:

a = 4; b = 6

g = a > b

g

OUTPUT:

[1] FALSE

Character Data Type

- A character object can be used for representing string values in R.
- To convert objects into character values use as.character() function.

Example:

```
g = as.character(62.48)
g
```

OUTPUT:

```
[1] “62.48”
```

Example:

```
s=“hi”
```

OUTPUT:

```
[1] “hi"
```

Raw Data Type

- Raw Data type is used to hold raw bytes.
- **Example:**

V<-charToRaw(“hello”)

print(v)

- **OUTPUT:**

[1] 68 65 6c 6c 6f

Data Structures

- R has a wide variety of objects for holding data, including
  - Scalars
  - Vectors
  - Matrices
  - Data frames
  - Lists
- They differ in terms of the type of data they can hold, how they're created, their structural complexity, and the notation used to identify and access individual elements.

variables

- Variable is a name given to a memory location used to store the information to be manipulated and referenced in the R program.
- The R variable can store an atomic vector, a group of atomic vectors, or a combination of many R objects, and even tables.
- R is a dynamically typed, means it check the type of data when the statement is run.
- We do not have to declare the data type of a variable before we can use it in our program.

Rules for variables

- It should contain letters, numbers, and only dot or underscore characters.
- It should not start with a number
- It should not start with a dot followed by a number
- It should not start with an underscore
- It should not be a reserved keyword.

Variable assignment

- The variables can be assigned values using equal to operator , leftward and rightward.
- There is no need to declare variable first.
- Just assign a value to the name and R will create the variable:

Example:

```
a1  = 10
print(a1)
```

OUTPUT:

```
[1] 10
```

Variable assignment

```
# Assignment using leftward operator.
a2 <- 5
cat (“a2 is ", a2 ,"\n")
```

OUTPUT: a2 is 5

```
# Assignment using rightward operator.
5 -> a4
cat (“a4 is ", a4 ,"\n")
```

OUTPUT:

a4 is 4

Listing Variables

- The ls function displays the names of objects in your workspace:

x <- 10

y <- 50.5

z <- “abc”

ls.str() - Tells about each variable

<u>OUTPUT</u>:

X: num 10

Y: num 50.5

z:chr “abc”

Remove a variable

- To remove unneeded variables or functions from your workspace or to erase its contents completely rm function is used.

Example:

x1 <- 3

X1

OUTPUT:

[1] 3

rm(x1)

x1

Error: object "x1" not found

Operators

- The operators are those symbols which tell the compiler for performing precise mathematical or logical manipulations.
- R programming is loaded with built in operators .
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Assignment Operators
  - Miscellaneous Operators

Arithmetic Operators

Operator	Description	Example	
+	Addition	x <- 5 y <- 16 x+y	a <- c(2,3.3,4) b <- c(11, 5, 3) print(a+b) [1] 13.0 8.3 7
–	Subtraction	x-y	print(a-b) [1] -9.0 -1.7 1.0
*	Multiplication	x*y	print(a*b) [1] 22.0 16.5 12.0
/	Division	y/x	print(a/b) [1]0.18 0.66 1.33
^	Exponent	y^x	print(a^b) [1] 2048.0000 391.3500 64.0000
	Modulus	y%%x	print(a%%b)

Logical Operators

Operator	Description	Example
!	Logical NOT	x <- c(3,0,TRUE,2+2i) !x [1] FALSE TRUE FALSE FALSE
&	Element-wise logical AND	x <- c(3,0,TRUE,2+2i) y <- c(2,4,TRUE,2+3i) x&y [1] TRUE FALSE TRUE TRUE
&&	Logical AND	x&&y [1] TRUE
|	Element-wise logical OR	x|y [1] TRUE TRUE TRUE TRUE
||	Logical OR	x||y [1] TRUE

- Operators & and | perform element-wise operation producing result having length of the longer operand.
- But && and || examines only the first element of the operands resulting into a single length logical vector.
- Zero is considered FALSE and non-zero numbers are taken as TRUE

Relational Operators

Operator	Description	Example
<	Less than	x <- 5 y <- 16 x<y [1] TRUE
>	Greater than	x>y [1] FALSE
<=	Less than or equal to	x<=5 [1] TRUE
>=	Greater than or equal to	y>=20 [1] FALSE
==	Equal to	y == 16 [1] TRUE
!=	Not equal to	x != 5 [1] FALSE

Assignment Operators

Operator	Description	Example
<-, <<-, =	Leftwards assignment	x <- 5 x **output:** [1] 5 x = 9 x **output:** [1] 9 x <<- 5 x **output:** [1] 5
->, ->>	Rightwards assignment	10 -> x x [1] 10 15->>y Y [1] 15

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

The <<- operator is used for assigning to variables in the parent environments (more like global assignments).

The rightward assignments, although available are rarely used.

Miscellaneous Operators

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

Operator	Description	Example
:	Colon operator. It creates the series of numbers in sequence for a vector.	v <- 2:8 print(v) [1] 2 3 4 5 6 7 8
%in%	This operator is used to identify if an element belongs to a vector.	v1 <- 8 v2<-12 t <- 1:10 print(v1%in%t) [1] TRUE print(v2%in%t) [1] FALSE
%*%	This operator is used to multiply a matrix with its transpose.	M = matrix(c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE) t = M %*% t(M) print(t)

Decision Making

- R programming provides three different types of if statements that allows programmers to control their statements within source code.
- if statement
- if….else statement
- If else ladder
- switch statement

If statement

Syntax:

```
if (condition) {
// statements
}
```

Example:

```
num <- 75
if (num > 50) {
print("Number Greater than 50")
}
```

OUTPUT: [1]“Number Greater than 50”

If else statement

Syntax

```
if (condition) {
        // statements
} else {
        // statements
}
```

Or

```
If(condition) statement1 else statement2
```

Example

```
a <- 10
b <- 20
if (a > b) { print("a is greater than b")
} else {     print("b is greater than a")
}
```

OUTPUT: [1] “b is greater than a”

Example

```
x<--5
if(x > 0)
{ print("Non-negative number")
} else {
print("Negative number") }
```

Or

```
if(x > 0) print("Non-negative number") else
   print("Negative number")
```

ifelse

- The ifelse function is a vectorized function of standard R if..else statement.
- The vectorization makes it faster than applying the same function to each element of the vector individually.

Syntax:

ifelse(expression, x, y)

- x:Return values for true elements of expression
- y:Return values for false elements of expression

Ifelse - Example

Example:

```
num <- 10
res  <- ifelse(num %% 2 == 0, "Even","Odd")
res
```

OUTPUT:

```
[1] ”Even”
```

If else ladder

- **Syntax**:

```
if (condition1) {
        // statements
}else if (condition2) {
        // statements
} else if (conditionN) {
        // statements
}else {
        // statements
}
```

If else ladder - example

Example:

```
a <- 10
b <- 10
if (a > b) {
    print("a is greater than b")
} else if(a == b){
   print("a and b are equal")
} else{
    print("b is greater than a")
} OUTPUT: [1] “a and b are equal”
```

Switch statement

Syntax:

switch(expression, case1, case2, case3...., caseN)

- Expression value is tested against multiple case values(case1,case2,case3…..caseN)

Rules for switch statement:

- No default values.
- If there is more than one match the first matching element is returned.

Switch - Example

```
dayOfWeek <- 5
dow <- switch(
   dayOfWeek ,
   "Sunday",
   "Monday",
   "Tuesday",
   "Wednesday",
   "Thursday",
   "Friday",
   "Saturday",
)
print(dow)
```

OUTPUT:[1] “Thursday”

Program - Odd or Even

```
num = as.integer(readline(prompt="Enter a
   number: "))
if((num %% 2) == 0) {
  print(paste(num,"is Even"))
} else {
  print(paste(num,"is Odd"))
}
```

Calculator using switch

```
x <- as.integer(readline("Enter first number"))
y <- as.integer(readline("Enter second number "))
ch <- as.integer(readline("Enter Choice"))
res <- switch(ch,x+y,x-y,x*y,x%/%y)
cat("Result=",res)
```

Output:

Enter first number 10

Enter second number 5

Enter Choice 1

Result= 15

Loops

- Loop statements are used to execute the block of code repeatedly until it meets the specified condition .

Loop Statements:

- for
- while
- repeat

R Loop Control statements:

- break
- next

For loop

Syntax:

```
for (item in items)
{
 // loop body
}
```

- Item holds the current element fetched from the items.
- Items is a vector that allows us to fetch each of the single element.

Example – for loop

Example:

```
employees <- c('John', 'Keith', 'Alex', 'Jason')
for (emp  in employees)
{
print(emp)
}
```

OUTPUT:

```
[1]”john”
[1]”keith”
[1]”Alex”
[1]”Jason”
```

Example – for loop

Example: To count the number of even numbers in a vector.

```
x <- c(2,5,3,9,8,11,6,44,43,47,67,95,33,65,12,45,12)
count <- 0
for (val in x) {
if(val %% 2 == 0)  count = count+1
}
print(count)
```

OUTPUT:

```
[1] 6
```

for loop - Step count by 2

```
for(i in seq(from=1, to=10, by=2))
{
  print(i)
}
```

or

```
for(i in seq(1, 10, 2))
{
    print(i)
}
```

Example – multiplication table

```
num = as.integer(readline(prompt = "Enter a
    number: "))
# use for loop to iterate 10 times
for(i in 1:10) {
  print(paste(num,'x', i, '=', num*i))
}
```

While Loop

Syntax:

```
while(condition)
{
    // loop body
}
```

While loop - example

```
cnt <- 2
while (cnt < 7) {
print(cnt)
cnt = cnt + 1
}
```

Output:

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

Repeat

- Repeat loop executes a block of code repeatedly until it meets a specific condition to break and exit the loop.
- It is mandatory to define an explicit condition with a break statement inside loop body which allows program to exit the loop.

Syntax:

```
repeat {
//statements
}
```

Repeat

Example:

```
ctr <- 1
repeat {
print("Hello, World!")
ctr = ctr+1 if (ctr > 5){ break }
}
```

OUTPUT:[1] “hello,World!”
[1] “hello,World!”
[1] “hello,World!”
[1] “hello,World!”
[1] “hello,World!”
[1] “hello,World!”

Break Statement

Syntax:

```
if (condition) {
break
}
```

Break Statement - Example

```
count <- 0
while (count <= 10) {
   count = count + 1
   if (count == 5) {
      break    }
print(“count")
}
```

OUTPUT:

```
[1] 1
[1] 2
[1] 3
[1] 4
```

Next statement

- Skip over the current iteration of any loop and continues with the next iteration.
- It does not terminate the loop . It just continues with the next iteration.
- When the next statement is encountered in the loop it returns the program execution to the first statement in the loop.

Syntax:

```
if (condition) {
next
}
```

Next statement - example

```
m=10
for (k in 1:m)
{
if (k %% 2 ==0)
        next
print(k)
}
```

Output:

```
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

Home work

- Find the Factorial of a Number
- Check Prime Number
- Check Armstrong Number
- Find sum of natural numbers

Factorial of a number

```
num = as.integer(readline(prompt="Enter a number: "))
factorial = 1
if(num == 0) {
  print("The factorial of 0 is 1")
} else {
  for(i in 1:num) {
    factorial = factorial * i
  }
  print(paste("The factorial of", num ,"is",factorial))
}
```

Prime Number

```
num = as.integer(readline(prompt="Enter a number: "))
flag = 1
for(i in 2:(num-1)) {
    if ((num %% i) == 0) {
      flag = 0
      break
    }
  }
if(num == 2)    flag = 1
if(flag == 1) {
  print(paste(num,"is a prime number"))
} else {
  print(paste(num,"is not a prime number"))
}
```

Armstrong Number

```
num = as.integer(readline(prompt="Enter a number: "))
sum = 0
temp = num
while(temp > 0) {
  digit = temp %% 10
  sum = sum + (digit ^ 3)
  temp =temp %/% 10
}
if(num == sum) {
  print(paste(num, "is an Armstrong number"))
} else {
  print(paste(num, "is not an Armstrong number"))
}
```

Sum of Natural Numbers

```
num = as.integer(readline(prompt = "Enter a number: "))
if(num < 0) {
  print("Enter a positive number")
} else {
  sum = 0

  while(num > 0) {
    sum = sum + num
    num = num - 1
  }
  print(paste("The sum is", sum))
}
```

Functions

- Functions are used to logically break our code into simpler parts which become easy to maintain and understand.

Syntax:

```
func_name <- function (argument)
{
        #statement
        # return statement
}
```

- The reserved word function is used to declare a function in R.
- The statements within the curly braces form the body of the function. These braces are optional if the body contains only a single expression.
- Finally, this function object is given a name by assigning it

Types of Functions in R

- Built – in
- User defined

Built in

- seq(),max(), mean(), sum(x), paste(...) etc.

SEQUENCE:

- This creates a sequence of number by using the predefined function seq().
- print (seq (1,10))

Output: [1] 1 2 3 4 5 6 7 8 9 10

Built in functions

MEAN

- This calculates the mean of all the numbers ranging from 4 to 26
- print (mean (4:26))

OUTPUT: [1] 15

SUM

print(sum(1:10))

Output: [1] 55

MAX / MIN

print(max(2,3))

OUTPUT: [1] 3

Function - Example

```
Pow <- function(a, b) {
result <- a^b
print(result)
}
```

Calling a function:

- Simply call the name with its arguments

```
Pow(5,2)
```

OUTPUT: [1] 25

Named Arguments

- In the above function calls, the argument matching of formal argument to the actual arguments takes place in positional order.
- This means that, in the call pow(8,2), the formal arguments x and y are assigned 8 and 2 respectively.
- We can also call the function using named arguments.
- When calling a function in this way, the order of the actual arguments doesn't matter.

  pow(b = 2, a = 8) **OUTPUT**: [1] 64
- we can use named and unnamed arguments in a single call.

  pow(a=8, 2)

Default Values for Arguments

Example:

```
pow <- function(x, y = 2) {
   result <- x^y
  print(result)
}
pow(8)
```

OUTPUT:[1] 64

```
pow(3,3)
```

OUTPUT: [1] 27

Return expression

Syntax:

return(expression)

Example

```
pow <- function(a, b) {
result <- a^b
return(result)
}
x <- 5
y <- 2
result = pow(x, y)
print(paste(x,"raised to ", y, "is", result))
```

OUTPUT: “ 5 raised to 2 is 25”

Calculator

```
add <- function(x, y)
    {
  return(x + y)
}
subtract <-
    function(x, y) {
  return(x - y)
}
multiply <-
    function(x, y) {
  return(x * y)
}
divide <- function(x,
    y) {
```

```
print("Select operation.")
print("1.Add");print("2.Subtract")
print("3.Multiply");print("4.Divide")
choice = as.integer(readline(prompt="Enter
    choice[1/2/3/4]: "))
num1 = as.integer(readline(prompt="Enter
    first number: "))
num2 = as.integer(readline(prompt="Enter
    second number: "))
result <- switch(choice, add(num1, num2),
    subtract(num1, num2), multiply(num1,
    num2), divide(num1, num2))
print(paste("The result=", result))
```

Recursion

- Recursion is a process in which a function calls by itself.

Example

```
factorial <- function(num) {
    if(num == 1)
        return (1)
     else
        return(num * factorial(num - 1))

}
result <- factorial(5)
print(paste("Factorial of 5 is ",result))
```

OUTPUT: “Factorial of 5 is 120”

Homework

- Fibonacci sequence using recursion

Fibonacci sequence

```
fibonacci <- function(n) {
  if(n <= 1) {
    return(n)
  } else {
    return(fibonacci(n-1) + fibonacci(n-2))
  }
}
nterms = as.integer(readline(prompt="How many terms? "))

print("Fibonacci sequence:")
for(i in 0:(nterms-1)) {
  print(fibonacci(i))
}
```

Strings

- Any value written within a pair of single quote or double quotes in R is treated as a string.
- Internally R stores every string within double quotes, even when you create them with single quote.
- **Rules Applied in String Construction**
  - The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
  - Double quotes can be inserted into a string starting and ending with single quote.
  - Single quote can be inserted into a string starting and ending with double quotes.
  - Double quotes can not be inserted into a string starting and ending with double quotes.

Strings

Example 1:

```
a <- 'String created with single quotes'
print(a)
```

Output:[1] “String created with single quotes”

Example 2:

```
b <- "String created with double quotes"
print(b)
```

Output: [1] "String created with double quotes"

String Manipulation

Concatenating Strings - paste() function

Many strings in R are combined using the **paste()** function. It can take any number of arguments to be combined together.

Syntax

paste(..., sep = " ", collapse = NULL)

Description of the parameters:

... represents any number of arguments to be combined.

sep represents any separator between the arguments. It is optional.

collapse is used to eliminate the space in between two strings. But not the space within two words of one string.

Paste function - Example

```
a <- "Hello"
b <- 'How'
c <- "are you? “
print(paste(a,b,c))
print(paste(a,b,c, sep = "-"))
print(paste(a,b,c, sep = "", collapse = ""))
```

OUTPUT:

```
[1] "Hello How are you? "
[1] "Hello-How-are you? "
[1] "HelloHoware you? "
```

Length of a string – nchar()

nchar() function

This function counts the number of characters including spaces in a string.

Syntax

nchar(x)

Example

result <- nchar(“characters")

print(result)

OUTPUT: [1] 10

s <- c("Moe", "Larry", "Curly")

nchar(s)

OUTPUT: [1] 3 5 5

Length function

- length function returns the length of a vector.
- The length function to a single string in R returns the value 1 because it views that string as a singleton vector—a vector with one element:

EXAMPLE 1:

```
length("Moe")
```

OUTPUT: [1] 1

EXAMPLE 2:

```
length(c("Moe","Larry","Curly"))
```

OUTPUT: [1] 3

toupper() & tolower() functions

Syntax

toupper(x)

tolower(x)

Example

result <- toupper(“welcome")

print(result)

OUTPUT: [1] “WELCOME”

result <- tolower(“HELLO")

print(result)

OUTPUT: [1]”HELLO”

substring() function

- This function extracts parts of a String.
- **Syntax**

  substring(x, first,last)
  - **x** is the character vector input.
  - **first** is the position of the first character to be extracted.
  - **last** is the position of the last character to be extracted.

  Example:

result <- substring("Extract", 5, 7)

Strsplit function

Syntax:

strsplit(string, delimiter)

The delimiter can be either a simple string or a regular expression.

path <- "/home/mike/data/trials.csv"

We can split that path into its components by using strsplit with a delimiter of /:

strsplit(path, "/")

Output:

[[1]]

[1] "" "home" "mike" "data" "trials.csv"

Replacing Substrings – sub & gsub functions

- Use sub to replace the first instance of a substring:

  sub(old, new, string)
- Use gsub to replace all instances of a substring:

  gsub(old, new, string)
- The sub function finds the first instance of the old substring within string and replaces it with the new substring:

  s <- "Curly is the smart one. Curly is funny, too."

  sub("Curly", "Moe", s)

  OUTPUT: [1] "Moe is the smart one. Curly is funny, too."
- gsub does the same thing, but it replaces all instances of the substring (a global replace), not just the first:

  gsub("Curly", "Moe", s)

  OUTPUT:[1] "Moe is the smart one. Moe is funny, too."

Generating All Pairwise Combinations of Strings

Use the outer and paste functions together to generate the matrix of all possible

combinations:

m <- outer(*strings1, strings2, paste, sep=""*)

Example

```
locations <- c("NY", "LA", "CHI", "HOU")
treatments <- c("T1", "T2", "T3")
outer(locations, treatments, paste, sep="-")
```

Output:

```
     [,1]      [,2]      [,3]
[1,] "NY-T1"   "NY-T2"   "NY-T3"
[2,] "LA-T1"   "LA-T2"   "LA-T3"
[3,] "CHI-T1"  "CHI-T2"  "CHI-T3"
[4,] "HOU-T1"  "HOU-T2"  "HOU-T3"
```

toString function

- Used to convert R object to a character string

Syntax:

toString(x, ...)

toString(x, width = NULL, ...)

- It first converts x to character type and then concatenates the elements with”,”.
- width - Suggestion for the maximum field width. Values of NULL or 0 indicate no maximum.

toString function - Example

Example:

```
x <- c("a", "b", "aaaaaaaaaaa")
toString(x)
```

OUTPUT:

```
[1] "a, b, aaaaaaaaaaa"
toString(x, width = 8)
```

OUTPUT:

```
[1] "a, b, ….“
```

Exercise

- Change all letters ‘a’ and ‘t’ to ‘A’ and ‘T’.

  gsub("a","A",gsub("t","T",s5))
- Find the number of characters in a string
- Extract the substring from 5th position to 9th position.

Vectors

- Vector is a basic data structure in R.
- It contains element of the same type. The six data types of atomic vectors are logical, integer, double, character, complex and raw.
- Vectors are classified into two type: atomic vectors and lists.
- In an atomic vector, all the elements are of the same type, but in the list, the elements are of different data types.
- A vector's type can be checked with the typeof() function.
- Another important property of a vector is its length. This is the number of elements in the vector and can be checked with the function length().

Creation of vector

Single element vector:

- When you write just one value in R, it becomes a vector of length 1 and belongs to one of the vector types.

Example:

```
print("abc")
print(12.5)
print(63L)
print(2+3i)
print(charToRaw('hello'))
print(TRUE)
```

Creating Vector – c()function

Vectors are generally created using the c() function.

```
x <- c(1, 5, 4, 9, 0)
x
```

OUTPUT:[1] 1 5 4 9 0

```
typeof(x)
```

OUTPUT:[1] "double"

```
length(x)
```

OUTPUT:[1] 5

- vector of iteger values.

  v <- as.integer(c(1,2,3,4))
- vector of logical values.

  c(TRUE, FALSE, TRUE, FALSE, FALSE)

Output

[1] TRUE FALSE TRUE FALSE FALSE

- A vector can contain character strings.

  c("aa", "bb", "cc", "dd", "ee")

Output

[1] "aa" "bb" "cc" "dd" "ee"

Creating Vector -: operator

Syntax:

c(start:end)

or

x <- start:end

Example 1:

x <- c(1:7)

x

OUTPUT:[1] 1 2 3 4 5 6 7

Example 2:y <- 2:2;

y

OUTPUT:[1] 2 1 0 -1 -2

Creating Vector - seq()

- **Syntax:**

seq(startValue, endValue, by=stepSize)

Example1:

seq(1, 3, by=0.2) # increments by 0.2

OUTPUT:[1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0

Example2:

seq(1, 4, length.out=6) # specify length of the vector

OUTPUT: [1] 1.0 1.6 2.2 2.8 3.4 4.0

Creating Vectors

The non-character values are converted to character type if one of the elements is a character.

Example:

```
s <- c('apple','red',5,TRUE)
s
```

OUTPUT:

```
[1] "apple" "red" "5" "TRUE"
```

Access Elements of a Vector

- Elements of a Vector are accessed using indexing.
- The [ ] brackets are used for indexing.
- Indexing starts with position 1.
- Giving a negative value in the index drops that element from result.

```
x <- c(2,4,6,8,10)
```

OUTPUT:[1] 2 4 6 8 10

```
 x[3]   # access 3rd element
```

OUTPUT:[1] 6

```
x[c(2, 4)]   # access 2nd and 4th element
```

OUTPUT: [1] 4 8

```
 x[-1]   # access all but 1st element
```

OUTPUT:[1] 4 6 8 10

Access Elements of a Vector

- **TRUE, FALSE or 0 and 1 can also be used for indexing.**

```
t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")
v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]
print(v)
```

OUTPUT:

[1] “Sun” “Fri”

```
y <- t[c(1,0,0,0,0,0,0)]
print(y)
```

OUTPUT:

[1] “Sun”

Modify a vector

```
x <-c(-3:2)
x
```

OUTPUT:[1] -3 -2 -1 0 1 2

```
x[2] <- 0  # modify 2nd element
x
```

OUTPUT:

[1] -3 0 -1 0 1 2

Delete a Vector

- We can delete a vector by simply assigning a NULL to it.

```
x <- NULL
x
```

NULL

Vector Manipulation

Vector arithmetic

```
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)
v1+v2
```

OUTPUT: [1] 7 19 4 13 1 13

```
 v1-v2
```

OUTPUT: [1] -1 -3 4 -3 -1 9

```
 v1*v2
```

OUTPUT:[1] 12 88 0 40 0 22

```
 v1/v2
```

OUTPUT:[1] 0.7500000 0.7272727 Inf 0.6250000 0.0000000
5.5000000

Vector Element Recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

```
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11) # V2 becomes c(4,11,4,11,4,11)
v1+v2
```

OUTPUT: [1]7 19 8 16 4 22

```
v1-v2
```

OUTPUT:[1] -1 -3 0 -6 -4 0

Vector Element Sorting

```
v <- c(3,8,4,5,0,11, -9, 304)
sort.result <- sort(v)
# Sort the elements in the reverse order.
revsort.result <- sort(v, decreasing = TRUE)
```

Output

```
[1] -9  0  3  4  5  8 11 304
[1] 304  11  8 5  4  3  0  -9
```

Vector Element Sorting

```
# Sorting character vectors.
v <- c("Red","Blue","yellow","violet")
sort.result <- sort(v)
# Sorting character vectors in reverse order.
revsort.result <- sort(v, decreasing = TRUE)
```

OUTPUT:

```
[1] "Blue" "Red" "violet" "yellow“
 [1] "yellow" "violet" "Red" "Blue
```

Vector Manipulation

- **Combining Vector**

```
n = c(1, 2, 3, 4)
s = c("Hadoop", "Spark", "HIVE", "Flink")
 c(n,s)
```

OUTPUT:[1] "1" "2" "3" "4" "Hadoop" "Spark" "HIVE" "Flink"

Vector Elements Arithmetic

```
x<-c(2,3,4,5)
 x            [1] 2 3 4 5
sum(x)    [1] 14
min(x)    [1] 2
max(x)    [1] 5
mean(x) [1] 3.5
 prod(x)  [1] 120
```

Computing Basic Statistics

- mean(x)
- median(x)
- sd(x)
- var(x)
- cor(x, y)
- cov(x, y)

Computing Basic Statistics

```
x <- c(0,1,1,2,3,5,8,13,21,34)
mean(x)
[1] 8.8
median(x)
[1] 4
sd(x)
[1] 11.03328
```

Computing Basic Statistics

var(x)

[1] 121.7333

x <- c(0,1,1,2,3,5,8,13,21,34)

y <- c(3,4,5,6,7,8,9,10,11,12)

cor(x,y)

[1] 0.8747879

cov(x,y)

[1] 29.22222

Removing “NA”

- Even one NA value in the vector argument causes any of these functions to return NA.

```
x <- c(0,1,1,2,3,NA)
mean(x)
[1] NA
sd(x)
[1] NA
```

- you can override this behavior by setting na.rm=TRUE, which tells R to ignore the NA values:

```
x <- c(0,1,1,2,3,NA)
mean(x, na.rm=TRUE)
[1] 1.4
sd(x, na.rm=TRUE)
[1] 1.140175
```

Comparing Vectors

- The comparison operators (==, !=, <, >, <=, >=) can perform an element-by-element comparison of two vectors.

```
v <- c( 3, pi, 4)
w <- c(pi, pi, pi)
v == w
```

OUTPUT: [1] FALSE TRUE FALSE

```
v!=w
```

OUTPUT:[1] TRUE FALSE TRUE

```
v < w
```

OUTPUT:[1] TRUE FALSE FALSE

- You can also compare a vector's element against a scalar.

```
v == pi
```

OUTPUT:[1] FALSE TRUE FALSE

Comparing vectors – any & all

- After comparing two vectors, you often want to know whether any of the comparisons were true or whether all the comparisons were true.
- The any and all functions handle those tests.

```
v <- c(3, pi, 4)
any(v == pi) # Return TRUE if any element of v equals pi
```

OUTPUT: [1] TRUE

```
 all(v == 0) # Return TRUE if all elements of v are zero
```

OUTPUT: [1] FALSE

Home work

- Write a R program to add, subtract, multiply and divide two vectors of integers type and length 3.
- Write a R program to find Sum, Mean and Product of a Vector, ignore element like NA or NaN.
- Write a R program to find the minimum and the maximum of a Vector.
- Write a R program to sort a Vector in ascending and descending order.
- Write a function to create a vector from 1 to 100. Multiply the elements which are smaller than 5 and larger than 90 with 10 and compute the sum of multiplied values. Multiply other elements with 1.5 and compute the product of multiplied values.

Lists

- List is a data structure having components of mixed data types.
- We can check if it’s a list with typeof() function
- Find its length using length().

list

```
l<- list("Red", "Green", c(21,32,11), TRUE, 51.23,
   119.1)
```

OUTPUT:

```
[[1]] [1] "Red"
[[2]] [1] "Green"
[[3]][1] 21 32 11
[[4]] [1] TRUE
[[5]] [1] 51.23
[[6]] [1] 119.1
```

Naming List Elements – Method1

The list elements can be given names and they can be accessed using these names.

```
l <- list(c("Jan","Feb","Mar"),list("green",12.3))
# Give names to the elements in the list.
names(l) <- c("1st Quarter", "A Inner list")
l
```

Output

```
$`1st Quarter`
[1] "Jan" "Feb" "Mar"
$`A Inner list`
$`A Inner list`
[[1]] [1] "green“
 $`A Inner list`
[[2]] [1] 12.3
```

Naming List Elements – Method2

```
x <- list("a" = 1.5, "b" = TRUE, "c" = c(2,4,6))
```

Output:

```
$a
[1] 1.5
$b
 [1] TRUE
$c [1] 2 4 6
```

Accessing List Elements

```
# Access the first element of the list.
l[1]
$`1st Quarter`
 [1] "Jan" "Feb" "Mar"


# Access the secondelement. As it is also a list, all its elements will be
    printed.
l[2]
$`A Inner list`
$`A Inner list`
[[1]] [1] "green“
 $`A Inner list`
[[2]] [1] 12.3
```

Manipulating List Elements

- We can add, delete and update list elements.

Add element at the end of the list.

l[3]<-12

l[4]<-TRUE

Remove the last element.

l[4] <- NULL

Update the 1st Element.

l[1]<-34.56

Merging Lists

```
# Create two lists.
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")

# Merge the two lists.
mlist <- c(list1,list2)
mlist
```

Output

```
[[1]] [1] 1
[[2]] [1] 2
[[3]] [1] 3
 [[4]] [1] "Sun«
 [[5]] [1] "Mon«
 [[6]] [1] "Tue"
```

Converting List to Vector

```
list1 <- list(1:5)
```

Output:

```
[[1]]
[1] 1 2 3 4 5
```

```
# Convert the list to vector
v1 <- unlist(list1)
```

Output:

```
[1] 1 2 3 4 5
```

Matrices

- Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout.
- They contain elements of the same atomic types.

Syntax

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

- **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 value. If TRUE then the input vector elements are arranged by row.

Example

matrix(1:9, nrow = 3, ncol = 3)

Output:

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

matrix(1:9, 3,3,TRUE) # fill matrix row-wise

Output:

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

```
matrix(list(2,3,4,5), nrow=2)
```

Output:

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

```
matrix(c(2,3,4,5), nrow=2)
matrix(c(2,3,4,5), nrow=2, byrow=TRUE)
```

Output:

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

column and row names.

```
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames =
    list(rownames, colnames))
P
      col1 col2 col3
row1   3    4    5
row2   6    7    8
row3   9   10   11
row4  12   13   14
```

Accessing Elements of a Matrix

Example

P[1,2]

Output:

[1] 4

To access entire row

P[2,]

Output:

col1 col2 col3

6 7 8

To access entire column

P[,3]

Output:

row1 row2 row3 row4

5 8 11 14

Creating matrix - cbind()

- Another way of creating a matrix is by using functions cbind() and rbind() as in column bind and row bind.

Example

cbind(c(1,2,3),c(4,5,6))

Output:

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

Creating matrix - rbind()

Example

```
rbind(c(1,2,3),c(4,5,6))
```

Output:

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

Adding column using cbind

```
m <- matrix(1:4,2,2,TRUE)
```

Output:

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Add column using cbind

```
cbind(m,c(5,6))
```

Output:

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

Adding row using rbind

```
m <- matrix(1:4,2,2,TRUE)
```

Output:

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Add row using rbind

```
rbind(m,c(5,6))
```

Output:

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

Matrix Computations

```
# Create two 2 matrices.
matrix1 <- matrix(c(1,2,3,4), nrow = 2,byrow=TRUE)
matrix2 <- matrix(c(1,2,3,4), nrow = 2, byrow=TRUE)
```

Output:

```
     [,1] [,2]
[1,] 1    2
[2,] 3    4
result <- matrix1 + matrix2
```

Output:

```
     [,1] [,2]
[1,] 2    4
[2,] 6    8
```

Matrix Computations

```
result <- matrix1 - matrix2
```

Output:

```
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

```
result <- matrix1 * matrix2
```

Output:

```
     [,1] [,2]
[1,]    1    4
[2,]    9   16
```

Matrix Computations

```
result <- matrix1 %*% matrix2
```

Output:

```
     [,1] [,2]
[1,]  7   10
 [2,] 15  22
```

```
result <- matrix1 %*% t(matrix2)
```

Output:

```
      [,1] [,2]
 [1,]  5    11
 [2,] 11  25
```

Matrix Computations

```
matrix1 <- matrix(c(1,2,3,4), nrow = 2,byrow=TRUE)
matrix2 <- matrix(c(1,2,3,4,5,6), nrow = 3, byrow=TRUE)
result <- matrix1 %*% matrix2
```

Output:

```
Error in matrix1 %*% matrix2 : non-conformable
    arguments
result <- matrix1 %*% t(matrix2)
```

Output:

```
     [,1] [,2] [,3]
 [1,]  5   11   17
[2,] 11   25   39
```

Matrix Computations

- result <- matrix1 / matrix2

Output:

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

Getting matrix input from user

```
x <- as.integer(readline("Number of rows: "))
y <- as.integer(readline("Number of cols: "))
v1 <- scan()
v1
matrix(c(v1),x,y)
```

Exercise

- Create a 5x5 matrix of integers from 1 to 25, filling one row at a time. Create another 5x5 matrix from vectors v1 and v2. Multiply the created matrices.
- Write a R program to create two 2x3 matrix and add, subtract, multiply and divide the matrices.
- Write a R program to create a matrix taking a given vector of numbers as input and define the column and row names. Display the matrix.

Arrays

- Arrays are the R data objects which can store data in more than two dimensions.
- An array is created using the **array()** function.
- It takes vectors as input and uses the values in the **dim** parameter to create an array.

Syntax

Array_NAME <- **array**(data, dim = (row_Size, column_Size, matrices), dimnames)

- **data –** Data is an input vector that is given to the array.
- **matrices –** Array in R consists of multi-dimensional matrices.
- **row_Size –** row_Size describes the number of row elements that an array can store.
- **column_Size –** Number of column elements that can be stored in an array.
- **dimnames –** Used to change the default names of rows and columns to the user's preference.

Creating Arrays

```
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2))
```

Output:

```
, , 1

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2

     [,1] [,2] [,3]
[1,]    5   10   13
```

Naming Columns and Rows

```
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")
result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames =
    list(row.names,column.names,matrix.names))
print(result)
```

Output:

```
, , Matrix1

     COL1 COL2 COL3
ROW1    5   10   13
ROW2    9   11   14
ROW3    3   12   15
```

Manipulating Array Elements

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
# Take these vectors as input to the array.
array1 <- array(c(vector1,vector2),dim = c(3,3,1))
# Create two vectors of different lengths.
vector3 <- c(9,1,0)
vector4 <- c(6,0,11,3,14,1,2,6,9)
array2 <- array(c(vector3,vector4),dim = c(3,3,1))
# create matrices from these arrays.
matrix1 <- array1[,,1]
matrix2 <- array2[,,1]
# Add the matrices.
result <- matrix1+matrix2
print(result)
```

Accessing Array Elements

```
v1 =  c(1,3,4,5)
v2 =  c(10,11,12,13,14,15)
result = array(c(v1,v2),dim = c(3,3,2))
print("The second row of the second matrix of the
    array:")
print(result[2,,2])
print("The element in the 3rd row and 3rd column
    of the 1st matrix:")
print(result[3,3,1])
```

Calculations across Array Elements

- apply() function is used for calculations in an array .
- **Syntax**

  apply(x, margin, fun)
  - x is an array.
  - a margin is the name of the dataset used.
  - fun is the function to be applied to the elements of the array.

apply()

```
v1 <- c(1,2,3)
v2 <- c(3,4,5,6,7,8)
a1 <- array(c(v1,v2),dim=c(3,3,2))
```

sum of the matrices by row-wise

```
    apply(a1, c(1), sum)
```

Output:

```
[1] 20  26   32
```

sum of the two matrices by column-wise

```
    apply(a1, c(2), sum)
```

Output:

```
[1] 12 24 42
```

convert a matrix to a 1 dimensional array.

```
m=matrix(1:12,3,4)
print(m)
a = as.vector(m)
print("1 dimensional array:")
print(a)
```

Output:

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12
```

Exercise

- Write a R program to create an array of two 3x3 matrices from two given vectors. Print the third row of the second matrix of the array and the element in the1st row and 3rd column of the 1st matrix.
- Write a R program to create a two-dimensional 5x3 array of sequence of even integers greater than 50.

- a <- array(seq(from = 50, length.out = 15, by = 2), c(5, 3))

Factors

- Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data).
- For example: a data field such as marital status may contain only values from single, married, separated, divorced, or widowed.
- In such case, we know the possible values beforehand and these predefined, distinct values are called levels.
- Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers.
- They are useful in the columns which have a limited number of unique values. Like "Male, "Female" and True, False etc. They are useful in data analysis for statistical modeling.
- Factors are created using the **factor()** function by taking a vector as input.

Creating a factor

Example

```
x <- factor(c("single", "married", "married", "single"))
```

Output:

```
 [1] single married married single
 Levels: married single
```

 # Number of levels

```
    nlevels(x)
```

Output:

```
[1] 2
```

```
    class(x)
```

Output:

```
 [1] "factor"
```

Creating a factor

```
data <-
    c("East","West","East","North","North","East","West","Wes
    t","West","East","North")

factor_data <- factor(data)

print(factor_data)
```

Output:

```
[1] East West East North North East West West West East
    North

Levels: East North West
```

- Factors are closely related with vectors. In fact, factors are stored as integer vectors.

```
x <- factor(c("single", "married", "married", "single"));

str(x)

Factor w/ 2 levels "married","single": 2 1 1 2
```

Generating Factor Levels

- We can generate factor levels by using the **gl()** function.
- It takes two integers as input which indicates how many levels and how many times each level.
- **Syntax**

  gl(n, k, labels)

Example

```
v <- gl(3, 4, labels = c("Tampa", "Seattle","Boston"))
print(v)
```

Output:

```
[1] Tampa Tampa Tampa Tampa Seattle Seattle Seattle Seattle
    Boston Boston Boston Boston
 Levels: Tampa Seattle Boston
```

Changing the Order of Levels

```
data <- c("East","West","East","North","North","East","West",
    "West","West","East","North")
factor_data <- factor(data)
print(factor_data)
```

Output:

```
[1] East West East North North East West West West East North
Levels: East North West
```

```
Str(factor_data)
```

Output:

```
Factor w/ 3 levels "East","North",..: 1 3 1 2 2 1 3 3 3 1 ..
```

```
new_order_data <- factor(factor_data,levels =
    c("East","West","North"))
str(new_order_data)
```

Output:

```
Factor w/ 3 levels "East","West",..: 1 2 1 3 3 1 2 2 2 1 ..
```

Nominal Categorical Variable

- A categorical variable has several values but the order does not matter.
- For instance, male or female categorical variable do not have ordering.

Example

```
color_vector <- c('blue', 'red', 'green', 'white', 'black', 'yellow')
# Convert the vector to factor
factor_color <- factor(color_vector)
factor_color
```

Output:

```
[1] blue red green white black yellow
Levels: black blue green red white yellow
```

Ordinal Categorical Variable

- Ordinal categorical variables have a natural ordering.
- We can specify the order, from the lowest to the highest with ordered= TRUE and highest to lowest with ordered = FALSE.

Example

```
mons =
    c("March","April","January","November","January","September","Octo
    ber","September","November","August","January","November","Nove
    mber","February","May","August","July","December","August","Augu
    st","September","November","February","April")
mons = factor(mons)
summary(mons)
```

Output:

```
April August December February January July March May November
    October September
2       4       1       2       3       1       1       1       5
        1       3
```

Ordinal Categorical Variable

```
# creating size vector
size = c("small", "large", "large", "small“,"medium", "large",
    "medium", "medium")

# converting to factor
size_factor <- factor(size)

# ordering the levels
ordered.size <- factor(size, levels = c("small", "medium",
    "large"),  ordered = TRUE)
```

Exercise

- Write R code to find the levels of factor of a vector c(1,3,7,4,5,10,9,8). Convert a given vector to an ordered factor.

Data Frames

- A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.
- Following are the characteristics of a data frame.
  - The column names should be non-empty.
  - The row names should be unique.
  - The data stored in a data frame can be of numeric, factor or character type.
  - Each column should contain same number of data items.

Syntax

```
data.frame(df, stringsAsFactors = TRUE)
```

- **df**: It can be a matrix to convert as a data frame or a collection of variables to join
- **stringsAsFactors**: Convert string to factor by default

Creating a data frame

```
n = c(2, 3, 5)
s = c("aa", "bb", "cc")
b = c(TRUE, FALSE, TRUE)
df = data.frame(n, s, b)
```

Output:

```
  n  s     b
1 2 aa  TRUE
2 3 bb FALSE
3 5 cc  TRUE
```

```
class(df)
```

Output: [1] "data.frame"

```
names(df) <- c("no", "name", "values")
```

Output:

```
    no name values
1 2      aa      TRUE
2 3      bb      FALSE
3 5      cc      TRUE
```

```
nrow(df)
```

Output: [1] 3

```
 ncol(df)
```

Output: [1] 3

Data Frame

```
emp <- data.frame( emp_id = c (1:5),
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
  salary = c(623.3,515.2,611.0,729.0,843.25),
    start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15",
     "2014-05-11“, "2015-03-27"))
)
```

Output:

	emp_id	emp_name	salary	start_date
1	1	Rick	623.30	2012-01-01
2	2	Dan	515.20	2013-09-23
3	3	Michelle	611.00	2014-11-15
4	4	Ryan	729.00	2014-05-11
5	5	Gary	843.25	2015-03-27

Structure of Data frame

```
x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" =
    c("John","Dora"))
 str(x)
```

Output:

```
'data.frame': 2 obs. of 3 variables:
$ SN : int 1 2
$ Age : num 21 15
$ Name: Factor w/ 2 levels "Dora","John": 2 1
```

```
x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" =
    c("John","Dora"), stringsAsFactors = FALSE)
str(x)
```

Output:

```
'data.frame': 2 obs. of 3 variables:
 $ SN : int 1 2
$ Age : num 21 15
$ Name: chr "John" "Dora"
```

Summary of Data in Data Frame

Summary(x)

Output:

SN	Age	Name
Min. :1.00	Min. :15.0	Dora:1
1st Qu.:1.25	1st Qu.:16.5	John:1
Median :1.50	Median :18.0	
Mean :1.50	Mean :18.0	
3rd Qu.:1.75	3rd Qu.:19.5	
Max. :2.00	Max. :21.0	

Extract Data from Data Frame

```
emp$emp_name – # Extract only employee name
result <- data.frame(emp$emp_name,emp$salary)
```

Output:

```
  emp.emp_name emp.salary
1    Rick          623.30
2    Dan           515.20
3    Michelle      611.00
4    Ryan          729.00
5    Gary          843.25
```

Extract the first two rows and then all columns

```
result <- emp[1:2,]
```

Output:

```
  emp_id emp_name salary start_date dept
1      1     Rick  623.3 2012-01-01 IT
2      2      Dan  515.2 2013-09-23 Operations
```

Extract 3rd and 5th row with 2nd and 4th column

```
result <- emp[c(3,5),c(2,4)]
```

Expand Data Frame

A data frame can be expanded by adding columns and rows.

```
x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" =
    c("John", "Dora"), stringsAsFactors = FALSE)
```

Output:

```
    SN Age Name
1   1   21   John
 2  2   15   Dora
```

Adding row

```
    rbind(x,list(1,16,"Paul"))
```

Adding Column

```
    cbind(x,State=c("NY","FL"))
```

or

```
x$State <- c("NY","FL")
```

Deleting Component

- Data frame columns can be deleted by assigning NULL to it.

  x$State <- NULL

Output:

SN Age Name

1 1 21 John

2 2 15 Dora

- Rows can be deleted through reassignments.

  x <- x[-1,]

Output:

SN Age Name

2 2 15 Dora

Displaying data

- head(emp) – displays first 6 rows
- head(emp,n=3) - first 3 rows
- tail(emp) – displays last 6 rows

- **Get the maximum salary**

  sal <- max(emp$salary)

- **Get the details of the person with max salary**

  retval <- subset(emp, salary == max(salary))

Output:

id Name Salary Dept

8 8 Guru 3000 IT

Exercise

- Write R code to create employee data frame with eno, ename and designation. Add a salary column to the employee data frame. Extract the designation of 3rd employee. Extract all employees who are earning more than 10000.

Packages

- R packages are a collection of R functions, complied code and sample data.
- They are stored under a directory called "**library**" in the R environment.
- By default, R installs a set of packages during installation.
- More packages are added later, when they are needed for some specific purpose.
- When we start the R console, only the default packages are available by default.
- Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

Packages

- **Check Available R Packages**

  .libPaths() - used for getting or setting the library trees

Output:

[1] "C:/Users/kala/Documents/R/win-library/3.4"

[2] "C:/Program Files/R/R-3.4.3/library“

- **Get the list of all the packages installed**

  library()

- **Get all packages currently loaded in the R environment**

  search()

Install a New Package

- There are two ways to add new R packages.
- One is installing directly from the CRAN directory and another is downloading the package to your local system and installing it manually.

Install directly from CRAN

- The following command gets the packages directly from CRAN webpage and installs the package in the R environment.

```
install.packages("Package Name")
```

Example: install.packages("XML")

Install package manually

install.packages(file_name_with_path, repos = NULL, type = "source")

Example

Load Package to Library

- Before a package can be used in the code, it must be loaded to the current R environment.
- You also need to load a package that is already installed previously but not available in the current environment.
- A package is loaded using the following command

```
library("package Name”)
```

Example

```
library(“XML”)
```

- To check what packages are installed on your computer, you can use:

  installed.packages()
- Uninstalling a package

  remove.packages(“package name")
- You can check what packages need an update with a call to the function:

  old.packages()
- You can update all packages by using:

  update.packages()

Data Reshaping

- Data Reshaping is about changing the way data is organized into rows and columns.
- Most of the time data processing is done by taking the input data as a data frame.
- It is easy to extract data from the rows and columns of a data frame but there are situations when we need the data frame in a format that is different from format in which we received it.
- R has many functions to split, merge and change the rows to columns and vice-versa in a data frame.

Joining Columns and Rows in a Data Frame

```
# Create data frame1
city <- c("Tampa","Seattle","Hartford","Denver")
state <- c("FL","WA","CT","CO")
 zipcode <- c(33602,98104,06161,80294)
addresses1 <- cbind(city,state,zipcode)
# Create another data frame with similar columns
address2 <- data.frame( city = c("Lowry","Charlotte"),  state =
    c("CO","FL"),   zipcode = c("80230","33949"),   stringsAsFactors =
    FALSE )
# Combine rows from both the data frames.
address <- rbind(address1,address2)
address
```

Merging Data Frames

- We can merge two data frames by using the **merge()** function.
- The data frames must have same column names on which the merging happens.

Example

- we consider the data sets about Diabetes in Pima Indian Women available in the library names "MASS".
- we merge the two data sets based on the values of blood pressure("bp") and body mass index("bmi").
- On choosing these two columns for merging, the records where values of these two variables match in both data sets are combined together to form a single data frame.

•

Merging Data Frames

```
library(MASS)
# Predefined dataframes in R
Pima.te
Pima.tr
# Merge
merged.Pima <- merge(x = Pima.te, y = Pima.tr,
                     by.x = c("bp", "bmi"),
                     by.y = c("bp", "bmi")
)
print(merged.Pima)
nrow(merged.Pima)
```

Transposing a data frame

- change the row to column and vice-versa by using the transpose function.
- It can be simply done by using 't(df_Temp)' .
- address2 <- data.frame(city = c("Lowry","Charlotte"), state = c("CO","FL"), zipcode = c("80230","33949"), stringsAsFactors = FALSE)

Output:

```
    city state zipcode
1 Lowry CO 80230
2 Charlotte FL 33949
```

- res <- t(address2)

Output:

```
              [,1]        [,2]
city  "Lowry" "Charlotte"
state "CO" "FL"
```

Melting and Casting

- Changing the shape of the data in multiple steps to get a desired shape. The functions used to do this are called **melt()** and **cast()**.
- We consider the dataset called ships present in the library called "MASS".

```
library(MASS)
print(ships)
```

Output:

```
   type year period service incidents
1  A    60   60     127     0
2  A    60   75     63      0
3  A    65   60     1095    3
```

Melt

- Now we melt the data to organize it, converting all columns other than type and year into multiple rows.

```
install.packages("reshape")
molten.ships <- melt(ships, id = c("type","year"))
print(molten.ships)
```

Output:

```
    type year variable value
1    A   60    period      60
2    A   60     service     127
3    A   60   incidents      0
```

Cast the Molten Data

- We can cast the molten data into a new form where the aggregate of each type of ship for each year is created.
- It is done using the **cast()** function.

```
recasted.ship <- cast(molten.ships, type+year~variable,sum)
print(recasted.ship)
```

Output:

```
  type year period service incidents
1    A   60    135     190         0
2    A   65    135    2190         7
```

R Programs for Practicing

1. Write a R program to get name from the user. Greet user with the welcome message and name.

Greeting the user with name

```
name = readline("Enter name")
print(paste("welcome",name))
```

R Programs for Practicing

2. write a simple number game in R studio. Pick a random number and see if it matches the user's number input. If it does, “Well Done" will appear and if it does not, "Higher", "Lower" or "You are so close" appears. Define a function that does the above mentioned.

   sample(x, size, replace = FALSE) – # pick a random number

   abs(number)

```
guessnumber <- function (guess) {
  Rs_number <- sample(1:100, 1)
  print(Rs_number)
  if (guess == Rs_number)
    print(‘Well Done')
  else if (abs(guess - Rs_number) < 10)
    print ('You are so close')
  else if (guess < Rs_number)
    print("Lower")
  else print("Higher")
}
guess <- as.integer(readline(prompt="Guess: "))
guessnumber(guess)
```

3. Write a R program to check whether the given year is leap year or not.

Leap year

```
year = as.integer(readline(prompt="Enter a year: "))
if( (year %% 400 == 0) || (year %% 4 == 0) && (year
   %% 100 != 0) )
{
     print(paste(year,"is a leap year"))
} else {
     print(paste(year,"is not a leap year"))
}
```

4. Writ a R program to create a for loop that, given a numeric vector, prints out one number per line, with its square and cube alongside. Get limit from the user.

Output:

[1] "1 1 1"
[1] "2 4 8“
[1] "3 9 27"
[1] "4 16 64"
[1] "5 25 125"

```
no <- as.integer(readline("Enter the limit"))
for(n in 1:no)
{
  print(paste(n,n^2,n^3))
}
```

5. Compute the truth table for logical AND.

Output

```
    [,1] [,2] [,3]
v1 "T" "F" "F"
v2 "F" "T" "F"
v3 "T" "T" "T"
v4 "F" "F" "F“
```

```
v1 <- c("T","F","F")
v2 <- c("F","T","F")
v3 <- c("T","T","T")
v4 <- c("F","F","F")
rbind(v1,v2,v3,v4)
```

6. Consider the vector 1:K, where K is a positive integer. Get vector input from user at runtime.Write a program that determines how many elements in the vector are exactly divisible by 3.

Solution

```
v <- scan()
cnt <-0
for(i in v)
{
    if(i%%3 == 0)
                cnt=cnt+1
 }
 print(paste("No.of Elements divisible by 3 are",cnt))
```

7. Create the data frame ‘employee.df’ with empno, name, age and gender. Use a simple ‘ifelse’ statement to add a new column ‘male.teen’ to the data frame. This is a boolean column, indicating TRUE if the observation is a male younger than 21 years.

Output:

Empno	Name	Age	Gender	Male.teen
1	John	20	M	TRUE
2	Eva	22	F	FALSE
3	Smith	21	M	FALSE

```
employee <-
    data.frame(Empno=1:3,Name=c("John","Eva","Smith"),Age=
    c(20,22,21),Gender=c("M","F","M"))
employee$Male.teen <- ifelse(employee$Gender == "M" &
                             employee$Age < 21, "TRUE", "FALSE")
employee
```

8. Create three vectors x,y,z with integers and each vector has 3 elements. Combine the three vectors to become a 3×3 matrix A where each column represents a vector. Change the row names to a,b,c.

```
v1 <- c(1,2,3)
v2 <- c(4,5,6)
v3 <- c(7,8,9)
rn <- c("a","b","c")
A <- matrix(c(v1,v2,v3),3,3,dimnames=list(rn))
A
```

Output:

```
  [,1] [,2] [,3]
a    1    4    7
b    2    5    8
c    3    6    9
```

9) If x = c(1, 2, 3, 3, 5, 3, 2, 4, NA), what are the levels of factor(x)?

10) Create a list called x with two two vectors of length 1 called a and b whose value is 1 and 2 respectively.

11) Create the dataframes.

buildings <- data.frame(location=c(1, 2, 3), name=c("building1", "building2", "building3"))

data <- data.frame(survey=c(1,1,1,2,2,2), location=c(1,2,3,2,3,1), efficiency=c(51,64,70,71,80,58))

The dataframes, buildings and data have a common key variable called, “location”. Use the merge() function to merge the two dataframes by “location”, into a new dataframe.

11) buildings <- data.frame(location=c(1, 2, 3),
name=c("building1", "building2", "building3"))

data <- data.frame(survey=c(1,1,1,2,2,2),
location=c(1,2,3,2,3,1),
efficiency=c(51,64,70,71,80,58))

merge(x=buildings,y=data,by.x="location",by.y="
location")

UNIT II
DATASET AND GRAPHICS

UNIT-II

- Input and Output-Entering Data from the Keyboard-CSV file-Excel File-Binary File-XML file-JSON file-Web Data-Database-Graphics-Pie Charts-Bar Charts-Box Plots-Dot plots-Histograms-Line Graphs-Scatter plots-Kernel density plots-Writing plot to a file-Changing graphical parameters.

Dataset

- A **data set** is usually a rectangular array of data with rows representing observations and columns representing variables.
- A **data set** is a collection of numbers or values that relate to a particular subject.
- For **example**, the test scores of each student in a particular class is a **data set**.

Patient Dataset

PatientID	AdmDate	Age	Diabetes	Status
1	10/15/2009	25	Type1	Poor
2	11/01/2009	34	Type2	Improved
3	10/21/2009	28	Type1	Excellent
4	10/28/2009	52	Type1	Poor

List of pre-loaded data

- To see the list of pre-loaded data, the function data() is used.

Load a built-in R data set:

Syntax:

- data(“dataset_name”)

Inspect the data set:

Syntax:

- head(dataset_name)

Built in Dataset - Example

Loading a built-in R data

```
data(mtcars)
# Print the first 6 rows
head(mtcars, 6)
```

OUTPUT:

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Input and Output

- All statistical work begins with data, and most data is available inside files and databases.
- Dealing with input is probably the first step of implementing any significant statistical project.
- All statistical work ends with reporting numbers back to a client.
- Formatting and producing output is probably the climax of your project.

Entering Data from the Keyboard

- For very small datasets, enter the data as literals using the c() constructor for vectors:

```
score <- c(61, 66, 90, 88, 100)
score
```

Output:

```
[1] 61 66 90 88 100 101 102
```

- Alternatively, you can create an empty data frame and then invoke an editor to populate it:

```
scores <- data.frame()
scores <- edit(score)
```

Output:

```
[1] 1 2 3 4 5
```

Entering Data from the Keyboard

EXAMPLE 2:

```
mydata <- data.frame(age=numeric(0), gender=character(0), weight=numeric(0))
mydata <- edit(mydata) or fix(mydata)
```

- Assignments like age=numeric(0) create a variable of a specific mode, but without actual data.

OUTPUT:

```
  age gender weight
1   1      M      3
2  10      F     27
```

Entering Data from the Keyboard

- We can add additional variables by clicking on the titles of unused columns.

mydata <- data.frame(age=numeric(0), gender=character(0), weight=numeric(0))

mydata <- edit(mydata) or fix(mydata)

OUTPUT:

	age	gender	weight	born year
1	1	F	4	1999
2	10	M	27	2010

Input - readline()

– readline() is used for inputting a line from the keyboard in the form of a string:

Example

z <-readline("Enter Input") **or**

z <-readline(prompt="Enter Input")

Output:

Enter Input 10

print(z)

Output:

[1] "10“

class(z)

Output:

[1] "character"

Input - scan()

- scan() function is used to type a few numbers into a vector from the keyboard.

  inp <- scan()

Output:

1: 2

2: 3

3: 4

4: press Enter

Read 3 items

inp

Output:

[1] 2 3 4

Printing to the screen

Printing Fewer Digits

- R normally formats floating-point output to have seven digits:

```
pi
```

Output:

```
[1] 3.141593
```

```
print(pi, digits=3)
```

Output:

```
[1] 3.14
```

Printing using cat()

- The cat function does not give direct control over formatting.
- Instead, use the format function to format numbers.

```
cat(pi, "\n")
```

Output:

```
3.141593
```

```
cat(format(pi,digits=3), "\n")
```

Output:

```
3.14
```

Redirecting Output to a File

- The output of the cat function is redirected to the file by using its file argument:

Syntax

cat("The answer is", answer, "\n", file="*filename")*

Example 1

name <-"xyz“

cat(name, “\n”, file="analysisReport.out")

Example 2

- The same file can be appended using

  result<-20

  cat(results, ”\n”, file="analysisRepart.out", append=TRUE)

Method 2

Example 3

```
name<-VsemA
no-57
con <- file("analysisReport1.out", "w")
cat(name, file=con, sep="\n")
cat(no, file=con,sep="\n")
close(con)
```

Output:

```
analysisReport1.out
xyz
57
```

sink()

- The sink function is used to redirect *all* output from both print and cat.
- Call sink with a filename argument to begin redirecting console output to that file.
- When you are done, use sink with no argument to close the file and resume output to the console:

Syntax:

sink("*filename*")

sink()

sink()

- The cat function writes to a file if you supply a file argument, which can be either a filename or a connection.
- The print function cannot redirect its output, but the sink function can force all output to a file.
- sink is to capture the output of an R script:

```
sink("script_output.txt")
source(“hello.R")
sink()
```

OUTPUT:

In script_output.txt [1] "Hello world"

Reading from a file

Example

```
c <- file(“analysisReport.out","r")
 readLines(c)
close(c)
 readLines(c, n=1)  # n specify the maximum number of lines
    to read
```

Output:

```
[1] "xyz " "20 "
```

Listing files

- The list.files function shows the contents of your working directory:

  list.files()

- To see all the files in your subdirectories, too, use

  list.files(recursive=TRUE)

- To see the hidden files:

  list.files(all.files=TRUE)

Reading Fixed-Width Records

- Suppose we want to read an entire file of fixed-width records

Syntax

Syntax:

records <- read.fwf("*filename*", widths=c(w1, w2, ..., w*n*))

Example 1

```
cat("123456", "987654", file="ff",sep="\n")
cat("ABCDEF","WELCOME",file="ff",append=TRUE,sep="\n")
read.fwf("ff", width=c(1,2,3))
```

Output:

	V1	V2	V3
1	1	23	456
2	9	87	654
3	A	BC	D H
4	W	EL	COM

Reading Fixed-Width Records

- **Example 2**

read.fwf(ff,width=c(1,2,3),col.names=c(“one”,”Tewo”,”Three”))

Output:

	One	Two	Three
1	1	23	456
2	9	87	654

Reading Tabular Data Files

- read.table function is used to read file as a table. It returns a data frame

Syntax

dfrm <- read.table("*filename")*

Example

- By default, it assumes the data fields are separated by white space (blanks or tabs).

dfrm <- read.table(" fixed-width.txt “)

Output:

	V1	V2	V3	V4
1	Fisher	R.A.	1890	1962
2	Pearson	Karl	1857	1936
3	Cox	Gertrude	1900	1978
4	Yates	Frank	1902	1994
5	Smith	Kirstine	1878	1939

Reading Tabular Data Files

- To assign column names and row names

  df1 <- read.table("fixed-width",row.names = c("r1","r2","r3","r4","r5"),col.names=c("c1","c2","c3","c4"))

- If your file uses a separator other than white space, specify it using the sep parameter.
- If our file used colon (:) as the field separator, we would read it this way:

  dfrm <- read.table(" fixed-width.txt ", sep=":")

- To prevent read.table from interpreting character strings as factors, set the stringsAsFactors parameter to FALSE:

  dfrm <- read.table("statisticians.txt", stringsAsFactor=FALSE)

- Now we can tell read.table that our file contains a header line, and it will use the column names when it builds the data frame:

```
dfrm <- read.table("statisticians.txt", header=TRUE,
stringsAsFactor=FALSE)
```

Writing table to a file

Syntax

write.table(dataset,”filename”)

Example

write.table(dfrm, “out.txt”)

CSV files

- A CSV is a **comma-separated values** file, which allows data to be saved in a tabular format.
- CSVs look like a spreadsheet but with a **.csv** extension.
- CSV files can be used with most any spreadsheet program, such as Microsoft Excel or Google Spreadsheets.
- They differ from other spreadsheet file types because you can only have a single sheet in a file, they can not save cell, column, or row.
- Also, you cannot save formulas in this format.

Read CSV Files

- The read.csv function is used to read CSV files.
- If your CSV file has a header line

Synatx

tbl <- read.csv("*filename")*

Example

data <- read.csv(“input.csv")

Output:

id Name Salary Dept

1 1 Rick 623.3 IT

2 2 Dan 515.2 Finance

3 3 Michelle 1000.0 HR

nrow and ncol

By default the **read.csv()** function gives the output as a data frame.

ncol(data) – returns number of colums

Output:

[1] 4

nrow(data) - returns number of rows

Output:

[1] 8

- If you don’t want to convert string as factors in reading, then specify stringsAsFactors = FALSE.

  newdata <- read.csv(“input.csv", stringsAsFactors = FALSE)

csv files with no headers

- If your CSV file does not contain a header line, set the header option to FALSE:

```
tbl <- read.csv("filename", header=FALSE)
```

Example

```
data1 <- read.csv(“input1.csv", header=FALSE)
data1
```

Output:

```
   V1    V2        V3       V4
1  1     Rick    623.3      IT
2   2     Dan    515.2   Finance
3   3    Michelle 1000.0  HR
```

Working with csv files

- **Get the maximum salary**

  sal <- max(data$Salary)

Output:

[1] 3000

- **Get the details of the person with max salary**

  retval <- subset(data, Salary == max(Salary))

Output:

id Name Salary Dept

8 8 Guru 3000 IT

Working with csv files

- **Get all the people working in IT department**

  retval <- subset(data, Dept == "IT")

Output:

id Name Salary Dept

1 1 Rick 623.3 IT

4 4 Ryan 1500.0 IT

7 7 Simon 2500.0 IT

8 8 Guru 3000.0 IT

- **Get the persons in IT department whose salary is greater than 1500**

  info <- subset(data, Salary > 1500 & Dept == "IT")

Output:

id Name Salary Dept

7 7 Simon 2500 IT

8 8 Guru 3000 IT

Writing into a CSV File

- The **write.csv()** function is used to create the csv file.

```
write.csv(retval,"output.csv")
newdata <- read.csv("output.csv")
newdata
```

Output:

```
  X id Name Salary Dept
1 1 1   Rick    623.3  IT
2 4 4   Ryan 1500.0 IT
3 7 7   Simon 2500.0 IT
4 8 8   Guru 3000.0 IT
```

- Here the column X is automatically created.
- This can be dropped using additional parameters while writing the file.

```
write.csv(retval,"output.csv“,row.names=FALSE)
```

Excel file

- Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format.

Method1

Installing and loading readxl package

- Install

  install.packages("readxl")
- Load

  library("readxl")

Using readxl package

- The **readxl** package comes with the function **read_excel**() to read xls and xlsx files

Example

my_data<-read_excel(“inputnew.xlsx")

my_data

Output:

id Name Salary Dept

<dbl> <chr> <dbl> <chr>

1 1. Rick 623. IT

2 2. Dan 515. Finance

3 3. Michelle 1000. HR

Using readxl package

- It’s also possible to choose a file interactively using the function **file.choose**().

  my_data <- read_excel(file.choose())

- **Specify sheet with a number or name**

  my_data <- read_excel("my_file.xlsx", sheet = "data")

  my_data <- read_excel("my_file.xlsx", sheet = 2)

Importing Excel files using xlsx package – Method 2

- **Install xlsx Package**
  install.packages("xlsx")
- **Load**
  library("xlsx")

Using xlsx package

read.xlsx(file, sheetIndex, header=TRUE)

read.xlsx2(file, sheetIndex, header=TRUE)

header: a logical value. If TRUE, the first row is used as column names.

Example

```
data <- read.xlsx("input.xlsx", sheetIndex = 1)
```

Write data to an Excel file

```
# Installing the package
install.packages("writexl")

# Loading package
library(writexl)
```

Syntax

```
write_xlsx(data,”filename”)
```

Example

```
write_xlsx(Data, "New_Data1.xlsx")
```

Binary Files

- A binary file is a file that contains information stored only in form of bits and bytes.
- They are not human readable as the bytes in it translate to characters and symbols which contain many other non-printable characters.
- Attempting to read a binary file using any text editor will show characters like Ø and ð.
- The binary file has to be read by specific programs to be useable.

- R provides two different functions for dealing with binary files.

writeBin() - for creating the binary file

Syntax

- writeBin (object, con)

readBin() – reading binary files

Syntax

- readBin (con, what, n)
  - con is the connection object used for reading or writing the binary file
  - object is the data that to be written into file
  - what is the mode, which can be character, integer etc and it represents the readable bytes.
  - n is the amount of bytes for reading from the binary file.

Writing into Binary File

```
# Creating a data frame
df = data.frame( "ID" = c(1, 2, 3, 4),    "Name" = c("Tony", "Thor", "Loki",
    "Hulk"),  "Age" = c(20, 34, 24, 40),  "Pin" = c(756083, 756001, 751003,
    110011) )


  # Creating a connection object
con = file("myfile.dat", "wb")

# Write the column names of the data frame
writeBin(colnames(df), con)

# Write the records in each of the columns to the file
writeBin(c(df$ID, df$Name, df$Age, df$Pin), con)

# Close the connection object
close(con)
```

Reading the Binary File

```
# Creating a connection object
con = file("myfile.dat", "rb")

# Read the column names
colname = readBin(con, character(),  n = 4)

# Read column values
# n = 20 as here 16 values and 4 column names
con = file("myfile.dat", "rb")
bindata = readBin(con, integer(), n = 20)

# Read the ID values ,  as first 1:4 byte for col name ,  then values of ID col
    is within 5 to 8
ID = bindata[5:8]
```

```
Name = bindata[9:12]
 Age = bindata[13:16]
 PinCode = bindata[17:20]
```
 # Combining all the values and make it a data frame
```
finaldata = cbind(ID, Name, Age, PinCode)
colnames(finaldata)= colname
print(finaldata)
```
Output:

	ID	Name	Age	Pin
[1,]	0	0	0	0
[2,]	1072693248	1074266112	1074790400	1073741824
[3,]	0	0	0	0
[4,]	1073741824	1074790400	1074266112	1072693248

XML Files

- XML stands for Extensible Markup Language.
- You can read a xml file in R using the "XML" package.
- **Install**

  install.packages("XML")
- **Load**

  library(“XML”)

employee.xml

```
<RECORDS>
  <EMPLOYEE>
    <ID>1</ID>
    <NAME>Rick</NAME>
    <SALARY>623.3</SALARY>
    <STARTDATE>1/1/2012</STARTDATE>
    <DEPT>IT</DEPT>
  </EMPLOYEE>

  <EMPLOYEE>
    <ID>2</ID>
    <NAME>Dan</NAME>
    <SALARY>515.2</SALARY>
    <STARTDATE>9/23/2013</STARTDATE>
    <DEPT>Operations</DEPT>
  </EMPLOYEE>
</RECORDS>
```

Reading XML File

Example

```
result <- xmlParse(file = “employee.xml")
```

Output:

```
<?xml version="1.0"?>
<RECORDS>
 <EMPLOYEE>
<ID>1</ID>
 <NAME>Rick</NAME>
<SALARY>623.3</SALARY>
 <STARTDATE>1/1/2012</STARTDATE>
<DEPT>IT</DEPT>
</EMPLOYEE>
</RECORDS>
```

xmlRoot()

The xmlRoot() function gets access to the root node and its elements.

Exmple

```
rootnode <- xmlRoot(result)
```

 # Find number of nodes in the root.

```
rootsize <- xmlSize(rootnode)
```

Output:

```
[1] 2
```

Details of the First Node

print(rootnode[1])

Get Different Elements of a Node

 # Get the first element of the first node.

print(rootnode[[1]][[1]])

 # Get the fifth element of the first node.

print(rootnode[[2]][[5]])

Extract XML data

Example

```
data <- xmlSApply(rootnode,function(x) xmlSApply(x,
    xmlValue))
data
```

Output:

	EMPLOYEE	EMPLOYEE
ID	"1"	"2"
NAME	"Rick"	"Dan"
SALARY	"623.3"	"515.2"
STARTDATE	"1/1/2012"	"9/23/2013"
DEPT	"IT"	"Operations"

XML to Data Frame

Example

xmldataframe <- xmlToDataFrame(“employee.xml")

Output:

ID NAME SALARY STARTDATE DEPT

1 1 Rick 623.3 1/1/2012 IT

2 2 Dan 515.2 9/23/2013 Operations

XML to list

l <- xmlToList(“employee.xml")

Output:

```
$EMPLOYEE $EMPLOYEE$ID
 [1] "1“
 $EMPLOYEE$NAME
 [1] "Rick“
 $EMPLOYEE$SALARY
 [1] "623.3"
$EMPLOYEE$STARTDATE
[1] "1/1/2012"
$EMPLOYEE$DEPT [
1] "IT"
$EMPLOYEE $EMPLOYEE$ID
 [1] "2“
 $EMPLOYEE$NAME
 [1] "Dan"
$EMPLOYEE$SALARY
[1] "515.2“
 $EMPLOYEE$STARTDATE
 [1] "9/23/2013"
$EMPLOYEE$DEPT [1] "Operations"
```

JSON FILES

- JSON file stores data as text in human-readable format.
- Json stands for JavaScript Object Notation
- Install rjson Package

  install.packages("rjson")

Input Data (input.json)

{ "ID":["1","2","3","4","5","6","7","8"],
"Name":["Rick","Dan","Michelle","Ryan","Gary","Nina","Simon","Guru"],
"Salary":["623.3","515.2","611","729","843.25","578","632.8","722.5"],
"StartDate":[
"1/1/2012","9/23/2013","11/15/2014","5/11/2014","3/27/2015","5/21/2013", "7/30/2013","6/17/2014"],
"Dept":[
"IT","Operations","IT","HR","Finance","IT","Operations","Finance"] }

Read the JSON File

Example

```
result <- fromJSON(file = "input.json")
```

Output

```
$ID [1] "1" "2" "3" "4" "5" "6" "7" "8"

$Name [1] "Rick" "Dan" "Michelle" "Ryan" "Gary" "Nina"
    "Simon" [8] "Guru“

$Salary [1] "623.3" "515.2" "611" "729" "843.25" "578" "632.8"
    "722.5"

$StartDate [1] "1/1/2012" "9/23/2013" "11/15/2014"
    "5/11/2014" "3/27/2015" "5/21/2013" [7] "7/30/2013"
    "6/17/2014"

$Dept [1] "IT" "Operations" "IT" "HR" "Finance" "IT" [7]
    "Operations" "Finance"
```

Convert JSON to a Data Frame

Example

```
jsondf <- as.data.frame(result)
jsondf
```

Output

```
  ID Name Salary StartDate Dept
1 1 Rick 623.3 1/1/2012 IT
2 2 Dan 515.2 9/23/2013 Operations
 3 3 Michelle 611 11/15/2014 IT
4 4 Ryan 729 5/11/2014 HR
5 5 Gary 843.25 3/27/2015 Finance
6 6 Nina 578 5/21/2013 IT
7 7 Simon 632.8 7/30/2013 Operations
8 8 Guru 722.5 6/17/2014 Finance
```

Writing JSON objects to .Json file

- toJSON() function from can be used to prepare a JSON object
- use write() function for writing the JSON object to a local file.

Example

```
l=list("apple", "banana", "rose")
```

read the above list to JSON

```
jsonData <- toJSON(l)
```

write JSON object to file

```
write(jsonData, "output.json")
```

Output:

Output.json

```
["apple","banana","rose"]
```

Web Data

- Many websites provide data for consumption by its users.
- Using R programs, we can programmatically extract specific data from websites.

Using data.table's fread()

- **Install and Load**

  Install.packages("data.table")

  library(data.table)

Using data.table's fread()

Example

```
mydat <-
fread('http://www.stats.ox.ac.uk/pub/datasets/csb/ch11b.dat')
head(mydat)
```

Output:

```
   V1 V2 V3 V4 V5
1:  1 307 930 36.58 0
2:  2 307 940 36.73 0
3:  3 307 950 36.93 0
4:  4 307 1000 37.15 0
5:  5 307 1010 37.23 0
6:  6 307 1020 37.24 0
```

Using read.csv function

Example

- To download the data from .txt file on Internet into R

file <- “ http://www.ospo.noaa.gov/data/land/bbep2/biomass_burning.txt? filename=LocalFile.txt.gz&dir=C:/R/Data ”
Biomass_Burning_Data <- read.csv(file, header=TRUE)

Webscraping using RCurl

```
library("RCurl")
file_url <-
"https://raw.githubusercontent.com/jennybc/gapminder/master/inst/extdata/gapminder.tsv"
gap_data_url <- getURL(file_url)
gap_data <- read.csv(textConnection(gap_data_url),sep = "\t")
head(gap_data)
```

Web Scraping

- **Web scraping** is a technique for converting the data present in unstructured format (HTML tags) over the web to the structured format which can easily be accessed and used.

Why do we need Web Scraping?

- Scraping movie rating data to create movie recommendation engines.
- Scraping text data from Wikipedia and other sources for making NLP-based systems or training deep learning models for tasks like topic recognition from the given text.
- Scraping labeled image data from websites like Google, Flickr, etc to train image classification models.
- Scraping data from social media sites like Facebook and Twitter for performing tasks Sentiment analysis, opinion mining, etc.
- Scraping user reviews and feedbacks from e-commerce sites like Amazon, Flipkart, etc.

Ways to scrape data

There are several ways of scraping data from the web. Some of the popular ways are:

- **Human Copy-Paste:** This is a slow and efficient way of scraping data from the web. This involves humans themselves analyzing and copying the data to local storage.
- **Text pattern matching:** Another simple yet powerful approach to extract information from the web is by using regular expression matching facilities of programming languages.
- **API Interface:** Many websites like Facebook, Twitter, LinkedIn, etc. provides public and/ or private APIs which can be called using the standard code for retrieving the data in the prescribed format.
- **DOM Parsing:** By using web browsers, programs can retrieve the dynamic content generated by client-side scripts. It is also possible to parse web pages into a DOM tree, based on which programs can retrieve parts of these pages.

Web scraping using DOM Parsing

- install.packages('rvest')
- Library(‘rvest’)
- an open source software named Selector Gadget is used to perform Web scraping.
- Using this you can select the parts of any website and get the relevant tags to get access to that part by simply clicking on that part of the website.

Scraping a webpage using R

- **#Loading the rvest package**

  library('rvest')
- **#Specifying the url for desired website to be scraped**

  url <-
  'http://www.imdb.com/search/title?count=100&release_date=2016,2016&title_type=feature'
- **#Reading the HTML code from the website**

  webpage <- read_html(url)

- **#Using CSS selectors to scrape the rankings** section
  rank_data_html <- html_nodes(webpage,'.text-primary’)
- **#Converting the ranking data to text**

  rank_data <- html_text(rank_data_html)

  head(rank_data)

Output:

  [1] "1." "2." "3." "4." "5." "6.“

- **#Data-Preprocessing: Converting rankings to numerical**
  rank_data<-as.numeric(rank_data)

Output:

  [1] 1 2 3 4 5 6

- **#Using CSS selectors to scrape the title section**
- title_data_html <- html_nodes(webpage,'.lister-item-header a')
- **#Converting the title data to text**

  title_data <- html_text(title_data_html)

  head(title_data)

Output:

[1] "Sing" "Moana" "Moonlight" "Hacksaw Ridge"
[5] "Passengers" "Trolls"

- **#Using CSS selectors to scrape the description section**
  description_data_html <- html_nodes(webpage,'.ratings-bar+ .text-muted')
- **#Converting the description data to text**

  description_data <- html_text(description_data_html)
- **#Data-Preprocessing: removing '\n'**
- description_data<-gsub("\n","",description_data)

Creating a data frame

cbind(rank_data,rating_data,description_data)

Database

- R can connect easily to many relational databases like MySql, Oracle, Sql server etc. and fetch records from them as a data frame.
- Once the data is available in the R environment, it becomes a normal R data set and can be manipulated or analyzed using all the powerful packages and functions.

RMySQL Package

- R has a built-in package named "RMySQL" which provides native connectivity between with MySql database.
- You can install this package in the R environment using the following command.

  install.packages("RMySQL")

Methods of Usage

- dbConnect — connect to a database
- dbDisconnect — close the connection to a database
- dbSendQuery — send a query to the database and use fetch to get results
- dbGetQuery — send a query to the database and get the results
- dbClearResult — returns TRUE or FALSE after clearing results
- dbListTables — shows all the tables in a database
- dbListFields — shows the names of columns in a database
- dbExistsTable — returns TRUE or FALSE
- dbRemoveTable — returns TRUE or FALSE
- dbWriteTable — stores a data frame in a database
- dbReadTable — reads a data from database as dataframe

dbConnect

Create A Connection To A DBMS

dbDisconnect

To Close the Connection

Usage

```
dbConnect(drv, ...)
dbDisconnect(conn, ...)
```

Arguments

drv

an object that inherits from `DBIDriver`, a character string specifying the DBMS driver, e.g., "RMariaDB", "RMySql" or possibly another `dbConnect` object.

conn

a connection object as produced by `dbConnect`.

...

authorization arguments needed by the DBMS instance; these typically include `user`, `password`, `dbname`, `host`, `port`, etc. For details see the appropriate `DBIDriver`.

`dbDisconnect` returns a logical value indicating whether the operation succeeded or not.

To connect to a MySQL database simply install the package and load the library.

```
install.packages("RMySQL")
library(RMySQL)
```

Connecting to MySQL:

Once the RMySQL library is installed create a database connection object.

```
mydb = dbConnect(MySQL(), user='user',
password='password', dbname='database_name', host='host')

dbDisconnect(mydb)
```

RMySql is a database interface and MySql driver for R

dbSendQuery, dbGetQuery, dbClearResult.

Execute A Statement On A Given Database Connection
Submits and executes an arbitrary SQL statement on a specific connection. Also, clears (closes) a result set.

Usage

```
dbSendQuery(conn, statement, ...)
dbGetQuery(conn, statement, ...)
dbClearResult(res, ...)
```

Arguments

conn
a connection object.

statement
a character vector of length 1 with the SQL statement.

res
a result set object (i.e., the value of `dbSendQuery`).

...
database-specific parameters may be specified.

- **dbSendQuery** only executes the SQL statement to the database engine.

- It does *not* extracts any records --- for that you need to use the function **fetch** (make sure you invoke **dbClearResult** when you finish fetching the records you need).

- The function **dbGetQuery** does all these in one operation (submits the statement, fetches all output records, and clears the result set).

- **dbGetQuery** returns a data.frame with the output (if any) of the query.

- **dbClearResult** frees all resources associated with a result set.

- **dbClearResult** returns a logical indicating whether clearing the result set was successful or not.

dbSendQuery and dbGetQuery

```
dbSendQuery(mydb, 'drop table if exists some_table,
some_other_table')
dbGetQuery(mydb, ‘select * from table')
```

Retrieving data from MySQL:
To retrieve data from the database we need to save a results set object.

```
rs = dbSendQuery(mydb, "select * from some_table")
```

To access the results in R we need to use the `fetch` function.

```
data = fetch(rs, n=-1)
```

This saves the results of the query as a data frame object. The `n` in the function specifies the number of records to retrieve, using `n=-1` retrieves all pending records.

dbClearResult(rs)

Clears the fetching process from rs

dbReadTable, dbWriteTable, dbExistsTable, dbRemoveTable

```
dbWriteTable(conn, name, value, row.names = F, ..., overwrite =
F, append = F)
dbReadTable(conn, name, row.names = "row_names", ...)
```

Arguments

conn

a database connection object.

name

a character string specifying a DBMS table name.

value

a data.frame (or coercible to data.frame).

row.names

in the case of `dbReadTable`, this argument can be a string or an index specifying the column in the DBMS table to be used as `row.names` in the output data.frame (a `NULL`, `""`, or 0 specifies that no)

overwrite

a logical specifying whether to overwrite an existing table or not. Its default is `FALSE`.

append

a logical specifying whether to append to an existing table in the DBMS. Its default is `FALSE`.

...

any optional arguments that the underlying database driver supports

Making tables:
We can create tables in the database using R dataframes.

```
dbWriteTable(mydb, name='table_name',
value=data.frame.name)
```

```
dbWriteTable(con, “data", data[, ])
dbReadTable(con, “data")

dbWriteTable(con, " data ", data[6:10, ], append = TRUE)
dbReadTable(con, " data ")

dbWriteTable(con, " data ", data[1:10, ], overwrite =
TRUE)
dbReadTable(con, " data ")

# No row names
dbWriteTable(con, " data ", data[1:10, ], overwrite =
TRUE, row.names = FALSE)
dbReadTable(con, " data ")
```

row.names

•If `FALSE` or `NULL`, row names are ignored.

•If `TRUE`, row names are converted to a column named "row_names”.

•If `NA`, a column named "row_names" is created if the data has custom row names, no extra column is created in the case of natural row names.

The default is `row.names = FALSE`.

```
dbExistsTable(con, “data")

dbRemoveTable(con, “data")
```

dbListTables and dbListFields

- dbListTables(dbCon)
- dbListFields(dbCon, table_name)

- Example:

```
# list the tables in database
dbListTables(mysqlconnection)

# List the Field Names of the Table "data"
dbListFields(mysqlconnection, "data")
```

Exercise

1) Write a R program to get Input for the Employee (No, Name, Salary) data frame from user at runtime and create an excel file.
2) Create an Xml file for students and Write a R program to convert into json file.
3) Write a R program to read data from csv file and create a binary file by using the data.

Graphics - Pie Charts

- R Programming language has numerous libraries to create charts and graphs.
- A pie-chart is a representation of values as slices of a circle with different colors. The slices are labeled and the numbers corresponding to each slice is also represented in the chart.
- In R the pie chart is created using the **pie()** function which takes positive numbers as a vector input.
- The additional parameters are used to control labels, color, title etc.

Pie chart

Syntax

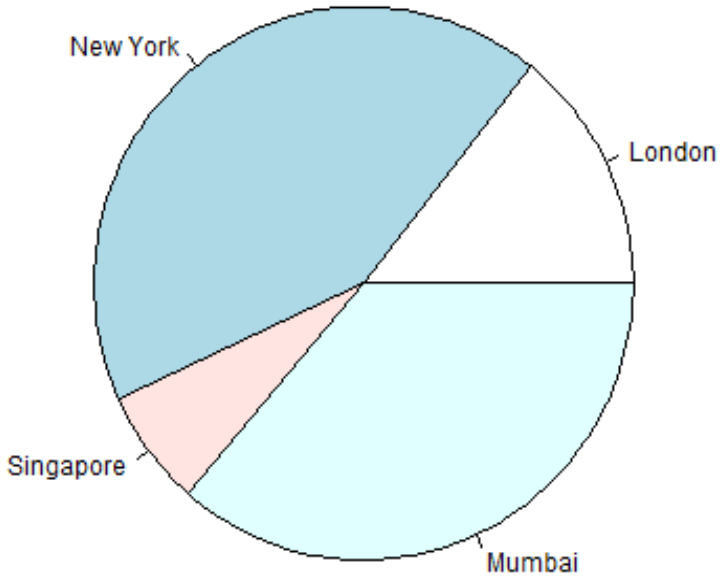
pie(x, labels, radius, main, col, clockwise, density, border)

- **Description of the parameters**
  - **x** is a vector containing the numeric values used in the pie chart.
  - **labels** is used to give description to the slices.
  - **radius** indicates the radius of the circle of the pie chart.(value between −1 and +1).
  - **main** indicates the title of the chart.
  - **col** indicates the color palette.
  - **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.
  - **density** specify the shading lines density (in lines per inch). By default, it is NULL, which means no shading lines.
  - **border** indicates border color.

Pie chart - Example

```
x <- c(21, 62, 10, 53)
country <- c("London", "New York", "Singapore", "Mumbai")
```

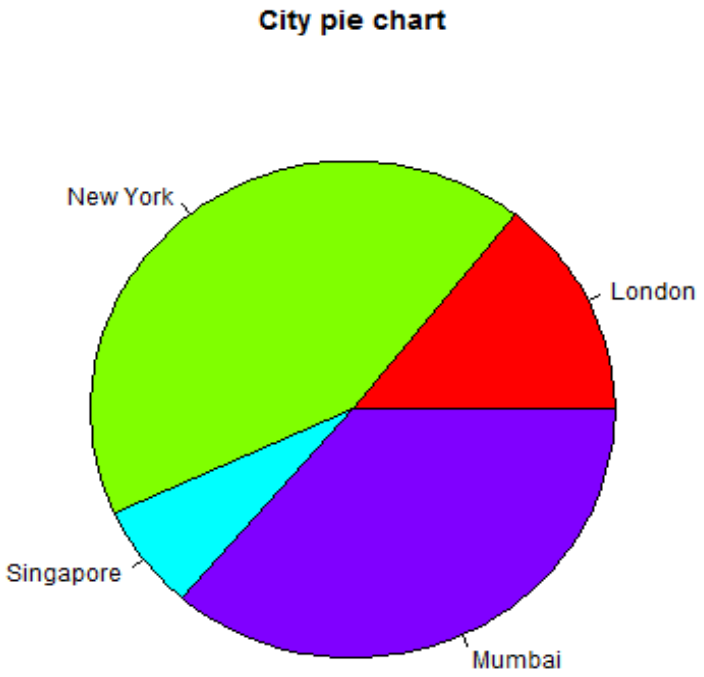
- # Give the chart file a name.
  png(file = "city.png")
- # Plot the chart.
  pie(x,labels=country)
- # Save the file.
  dev.off()



Pie Chart Title and Colors

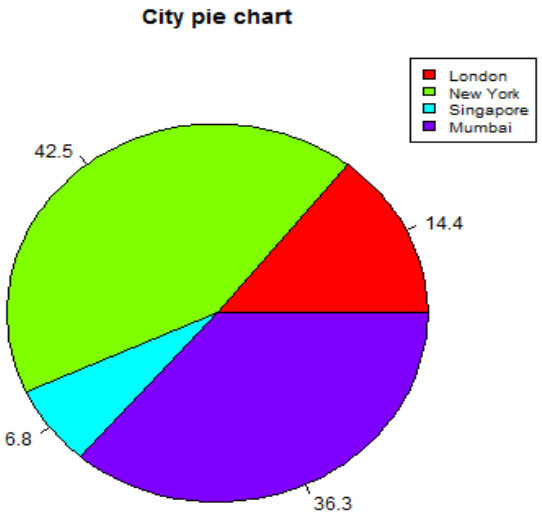
- # Plot the chart with title and rainbow color pallet.

  pie(x, labels, main = "City pie chart", col = rainbow(length(x)))



Slice Percentages and Chart Legend

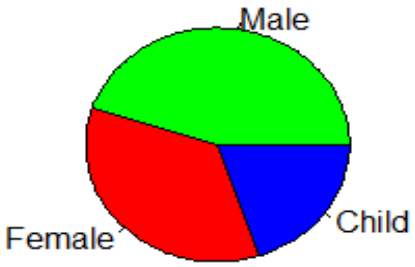
```
piepercent<- round(100*x/sum(x), 1)
pie(x, labels = piepercent, main = "City pie chart",col =
rainbow(length(x)))
legend("topright", c("London","New
York","Singapore","Mumbai"), cex = 0.5,fill =
rainbow(length(x)))
```



Pie chart for data frame Values

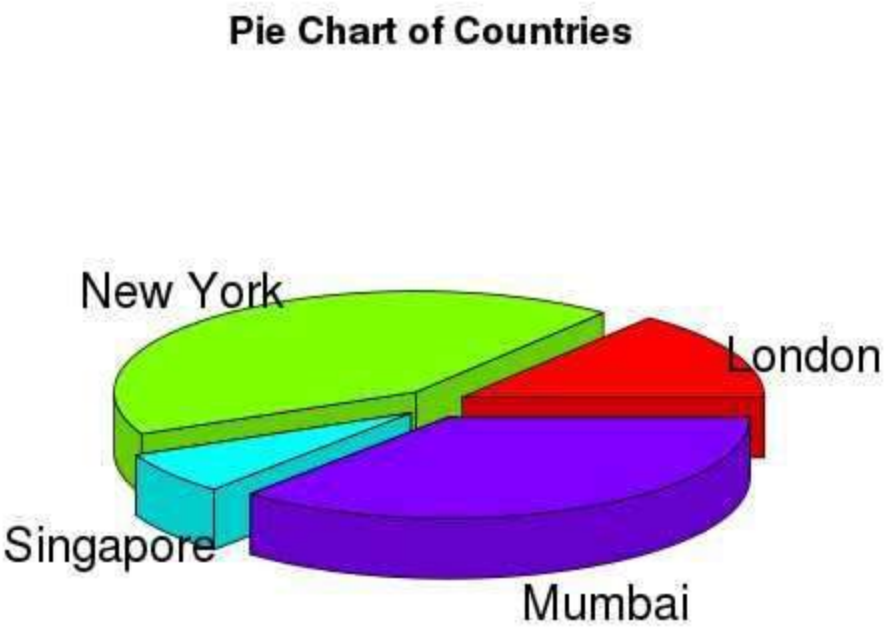
```
df <-
    data.frame(Count=c(45,35,20),Group=c("Male","Female","Ch
    ild"))
pie(df$Count,df$Group,col=c("Green","Red","Blue"))
```

Output:



3D Pie Chart

```
library(plotrix)
x <- c(21, 62, 10,53)
lbl <- c("London","New York","Singapore","Mumbai")
pie3D(x,labels = lbl,explode = 0.1, main = "Pie Chart of Countries ")
```



Bar Chart

- A bar chart represents data in rectangular bars with length of the bar proportional to the value of the variable.
- R uses the function **barplot()** to create bar charts.

Syntax

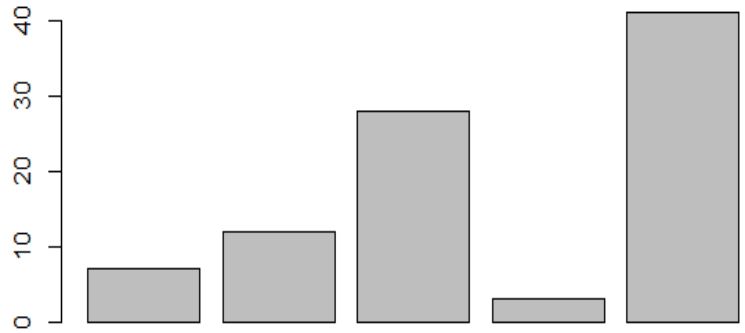
barplot(H,xlab,ylab,main, names.arg,col)

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

Bar Chart - Example

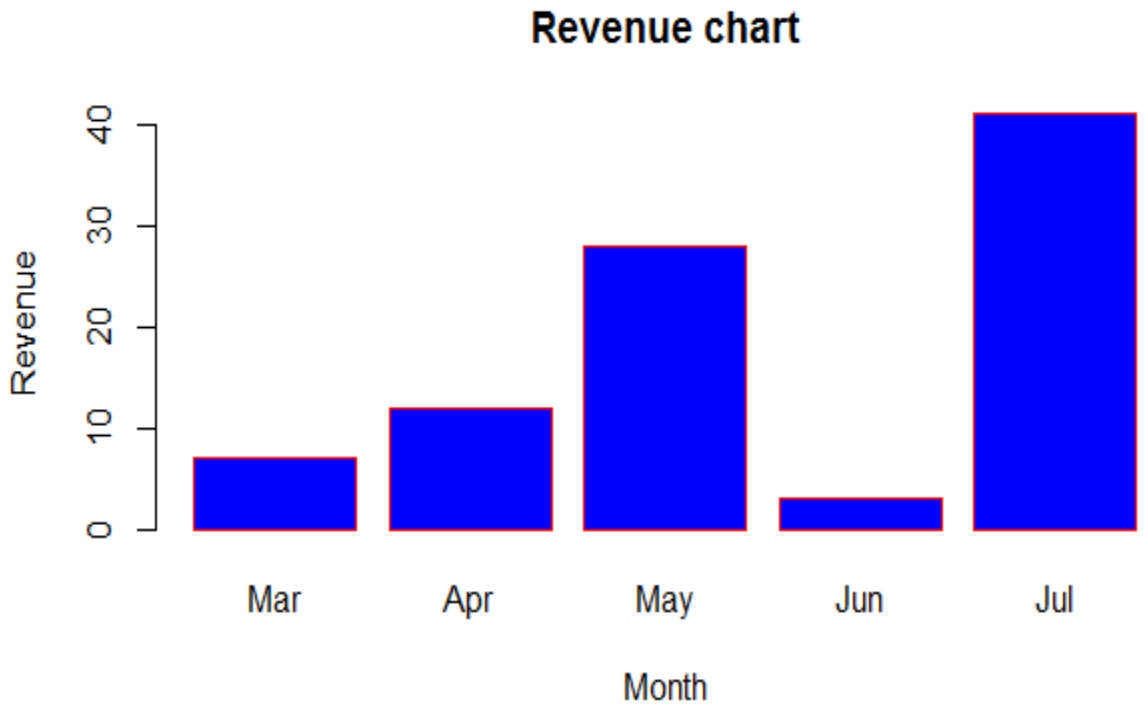
- # Create the data for the chart
  H <- c(7,12,28,3,41)

- # Plot the bar chart
  barplot(H)



Bar Chart Labels, Title and Colors

```
H <- c(7,12,28,3,41)
M <- c("Mar","Apr","May","Jun","Jul")
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="blue",
  main="Revenue chart",border="red")
```



Group Bar Chart and Stacked Bar Chart

- We can create bar chart with groups of bars and stacks in each bar by using a matrix as input values.
- More than two variables are represented as a matrix which is used to create the group bar chart and stacked bar chart.
- **# Create the input vectors.**

  colors = c("green","orange","brown")

  months <- c("Mar","Apr","May","Jun","Jul")

  regions <- c("East","West","North")

- **# Create the matrix of the values.**

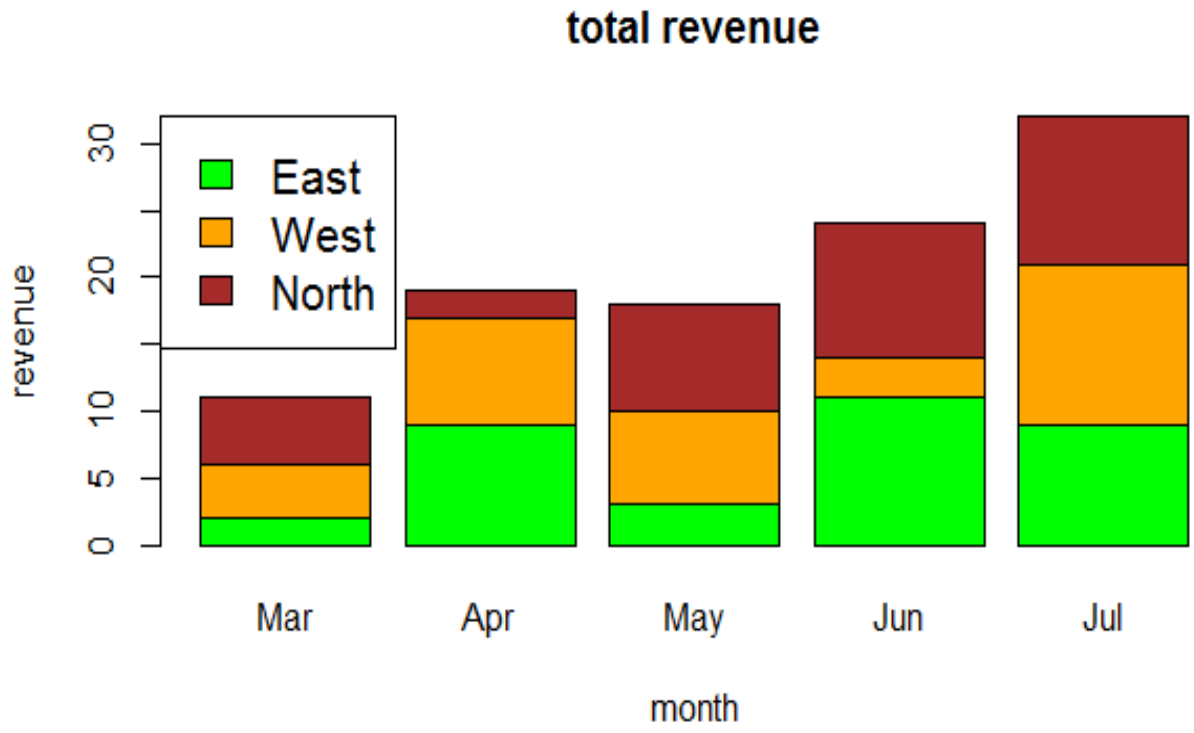
```
Values <- matrix(c(2,9,3,11,9,4,8,7,3,12,5,2,8,10,11), nrow = 3, ncol = 5, byrow = TRUE)
```

- **# Create the bar chart**

```
barplot(Values, main = "total revenue", names.arg = months, xlab = "month", ylab = "revenue", col = colors)
```

- **# Add the legend to the chart**

```
legend("topleft", regions, cex = 1.3, fill = colors)
```

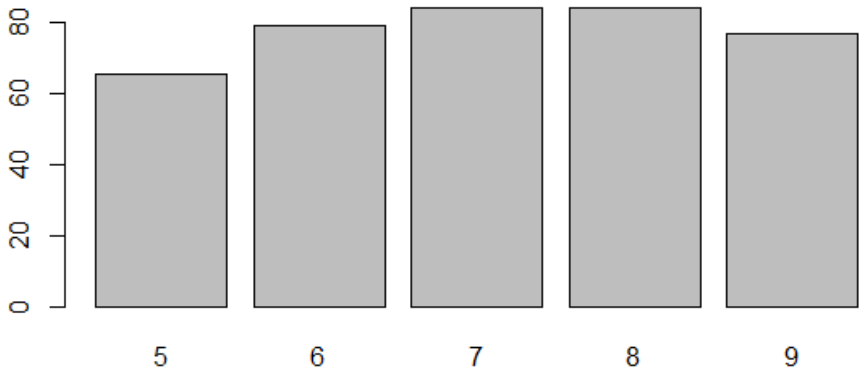

total revenue
East
West
North
revenue
0
5
10
20
30
Mar
Apr
May
Jun
Jul
month

Bar Chart

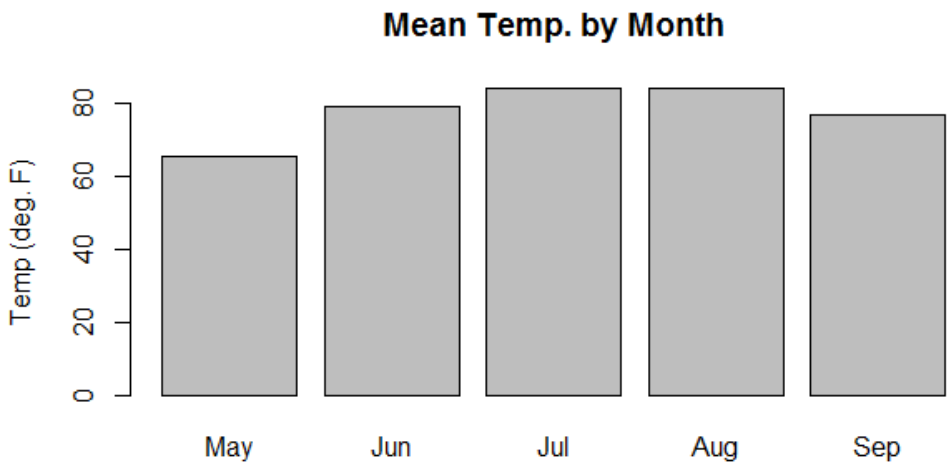
- For example, the airquality dataset contains a numeric Temp column and a Month column.
- We can create a bar chart of the mean temperature by month in two steps. First, we compute the means:

  heights <- tapply(airquality$Temp, airquality$Month, mean)
- That gives the heights of the bars, from which we create the bar chart:

  barplot(heights)



- barplot(heights,main="Mean Temp. by Month“,names.arg=c("May", "Jun", "Jul", "Aug", "Sep"), ylab="Temp (deg. F)")



Stacked Bar Plot

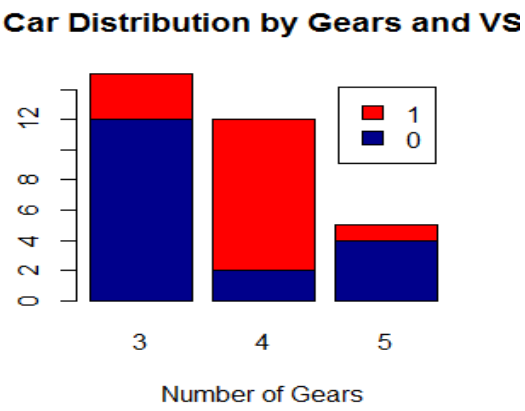
counts <- table(mtcars$vs, mtcars$gear)

Counts

Output:

```
   3   4  5
0 12  2   4
1 3   10  1
```

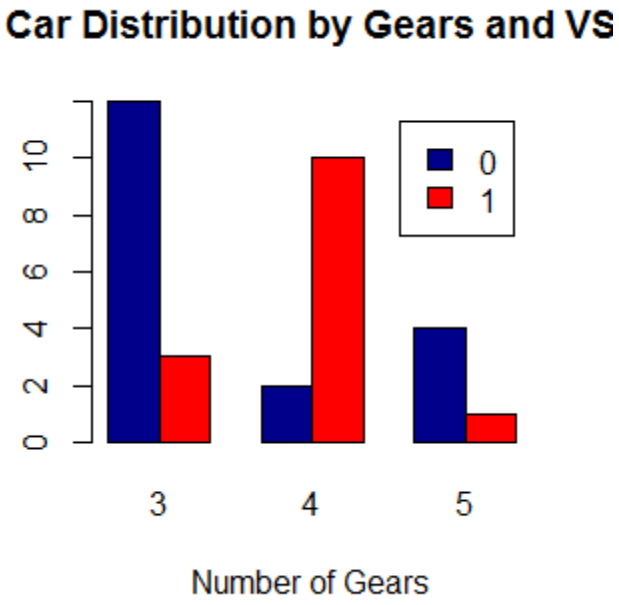
barplot(counts, main="Car Distribution by Gears and VS",xlab="Number of Gears", col=c("darkblue","red"),legend = rownames(counts))



Grouped Bar Plot

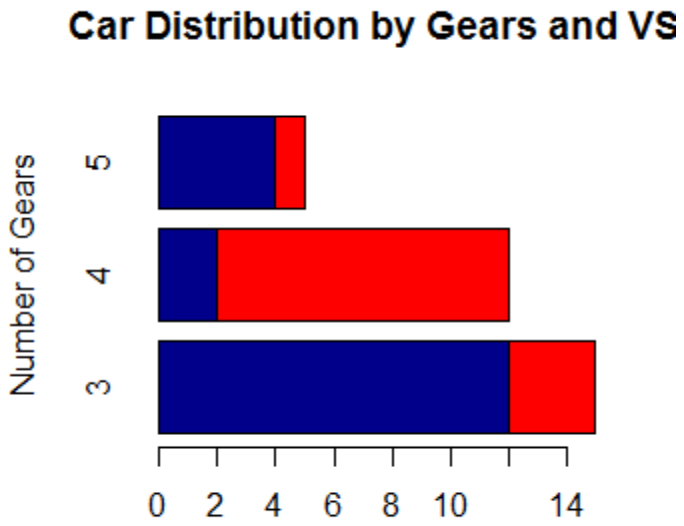
```
barplot(counts, main="Car Distribution by Gears and
VS",xlab="Number of Gears", col=c("darkblue","red"),legend
= rownames(counts), beside=TRUE)
```

Output:



Horizontal Bar Plot

```
barplot(counts, main="Car Distribution by Gears and
VS",ylab="Number of Gears", col=c("darkblue","red"),
horiz=TRUE)
```



Box plot

- Boxplots are a measure of how well distributed is the data in a data set.
- This graph represents the minimum, maximum, median, first quartile and third quartile in the data set.
- It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.
- Boxplots are created in R by using the **boxplot()** function.

Box Plot

- **Syntax**

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

Box Plot

Example

```
input <- mtcars[,c('mpg','cyl')]
print(head(input))
```

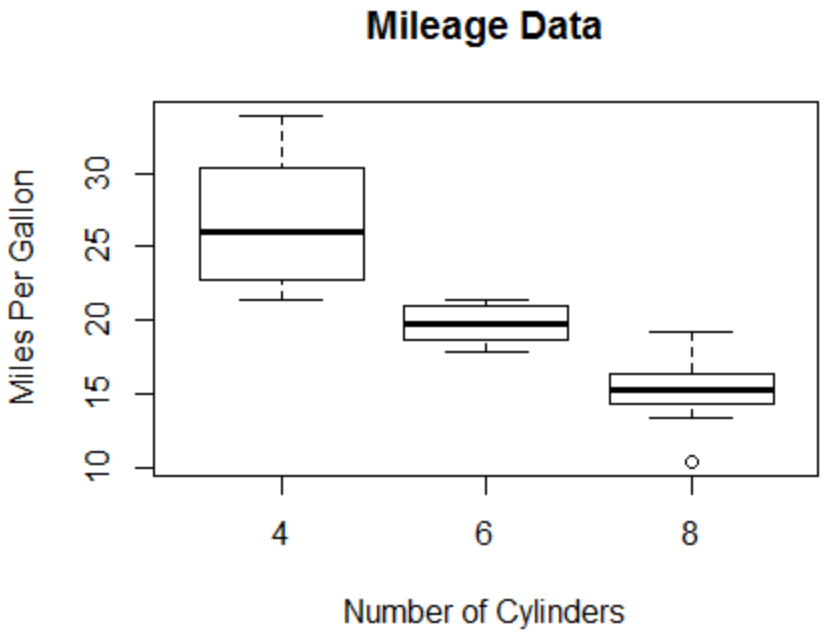
Output:

	mpg	cyl
Mazda RX4	21.0	6
Mazda RX4 Wag	21.0	6
Datsun 710	22.8	4
Hornet 4 Drive	21.4	6
Hornet Sportabout	18.7	8
Valiant	18.1	6

- The function boxplot() can also take in formulas of the form y~x where, y is a numeric vector which is grouped according to the value of x.

Creating the Boxplot

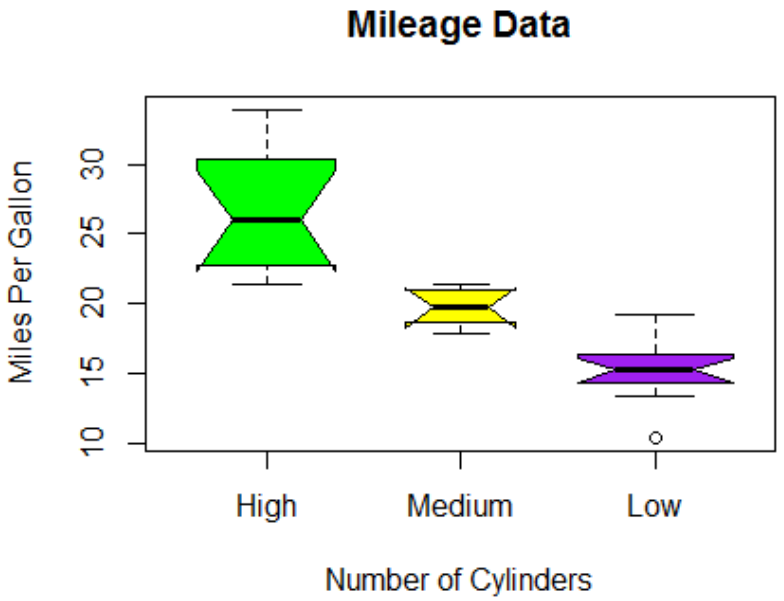
- boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders", ylab = "Miles Per Gallon", main = "Mileage Data")



Boxplot with Notch

- We can draw boxplot with notch to find out how the medians of different data groups match with each other.

```
boxplot(mpg ~ cyl, data = mtcars,
        xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon",
        main = "Mileage Data",
        notch = TRUE,
        varwidth = TRUE,
        col = c("green","yellow","purple"),
        names = c("High","Medium","Low")
)
```



Dot plot

- Dot plot in R also known as dot chart is an alternative to bar charts, where the bars are replaced by dots.
- A simple Dot plot in R can be created using dotchart function.
- **Syntax**

  dotchart (NumericVector, cex = 1, col = “black”, labels = NULL, main = NULL, pch = 1, sub = NULL, xlab = NULL)

- **Description of the parameters**

cex - plot scaling factor(size) . More the value of cex, more the plot size will be

col - colour of the dot

labels - A vector containing the label names for each plotted value.

main - Title of the dot chart

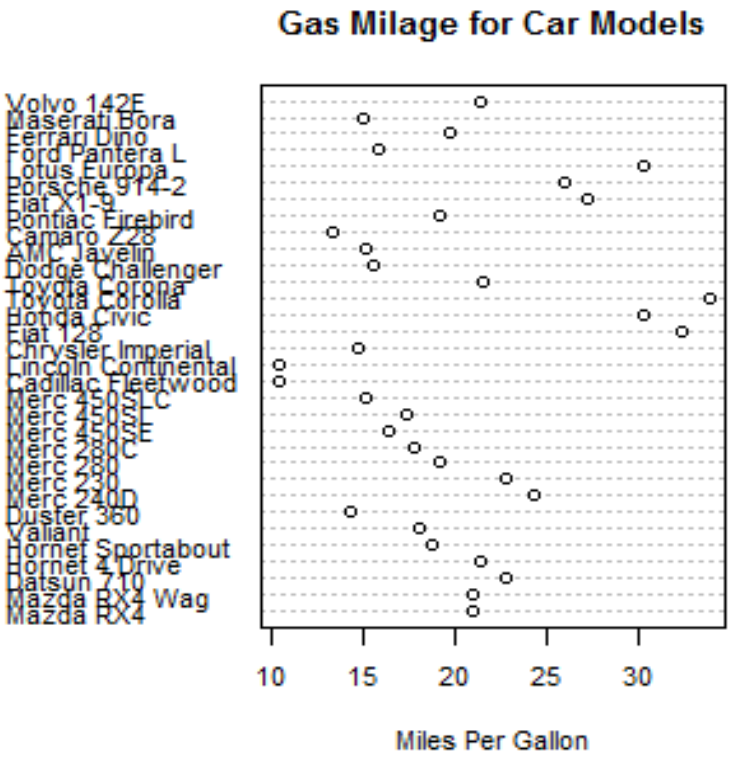
pch - numeric value which decides the type of plot … if pch=1 then dot, pch=2 then triangle, pch=3 then ‘+’

Sub - subtitle of the dot chart

Xlab - x axis label

Example

- dotchart(mtcars$mpg,labels=row.names(mtcars), main="Gas Milage for Car Models", xlab="Miles Per Gallon", cex=0.7)

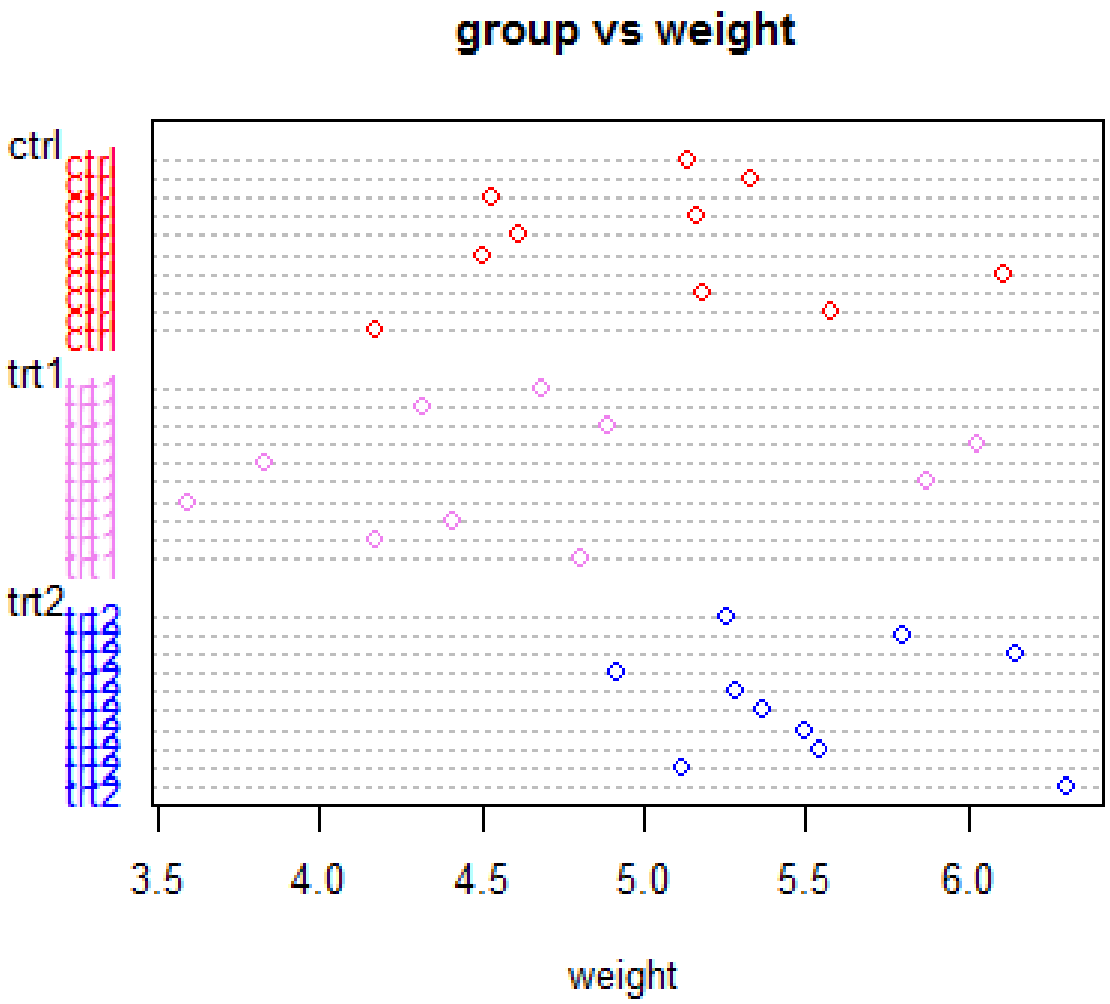


Dot plot in R for groups

- Suppose if we want to create the different dot plots for different group of the same data set.
- PlantGrowth data set have 3 groups ctrl, trt1 and trt2.

```
pg <- PlantGrowth
pg$color[pg$group=="ctrl"] <- "red"
pg$color[pg$group=="trt1"] <- "Violet"
pg$color[pg$group=="trt2"] <- "blue“
dotchart(PlantGrowth$weight,
    labels=PlantGrowth$group,cex=0.8,groups= PlantGrowth$group,
        main="group vs weight",
        xlab="weight", gcolor="black", color=pg$color)
```

Dot plot - Output



Histogram

- A histogram represents the frequencies of values of a variable bucketed into ranges.
- Histogram is similar to bar chat but the difference is it groups the values into continuous ranges.
- Each bar in histogram represents the height of the number of values present in that range.
- R creates histogram using **hist()** function.

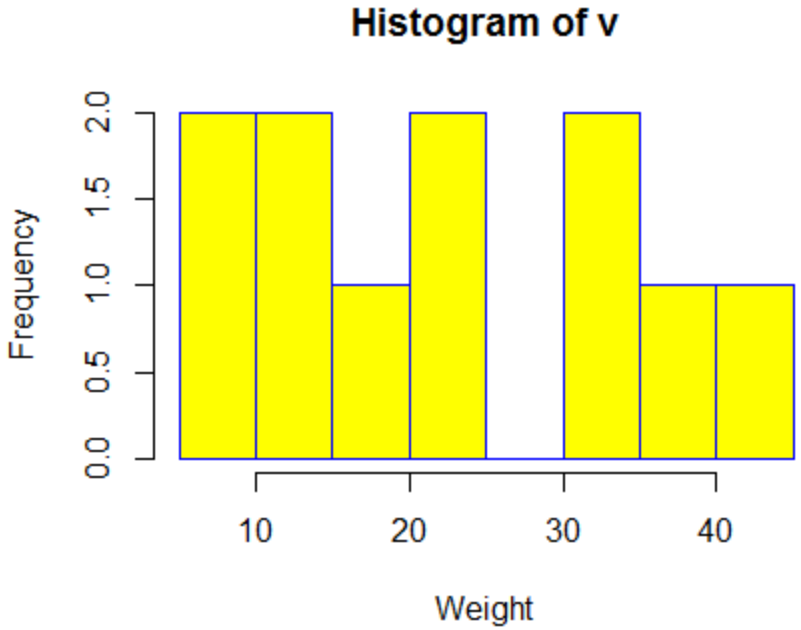
Histogram

- **Syntax**

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

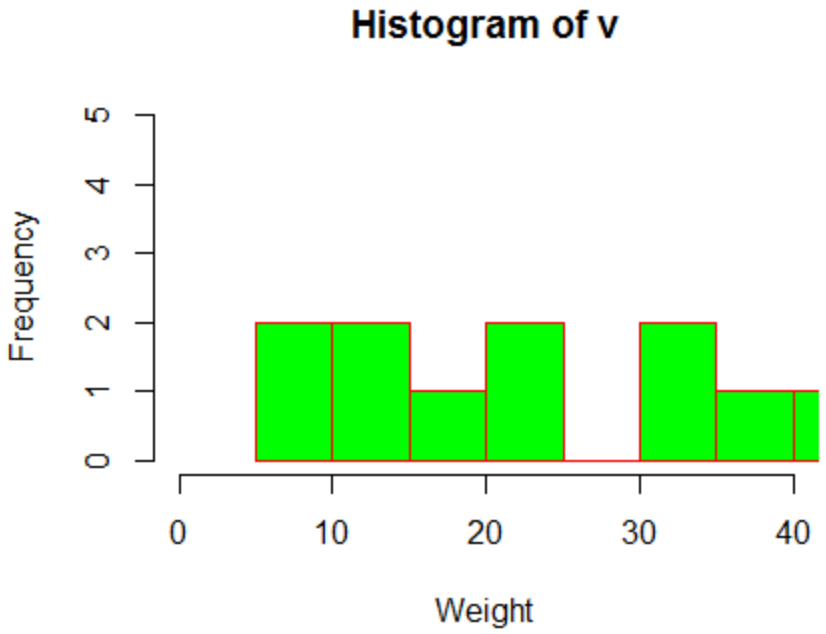
Example

```
v <-  c(9,13,21,8,36,22,12,41,31,33,19)
hist(v,xlab = "Weight",col = "yellow",border = "blue")
```



Range of X and Y values

```
hist(v,xlab = "Weight",col = "green",border = "red", xlim =
    c(0,40), ylim = c(0,5),breaks = 5)
```



Line Graphs

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

Line Graphs

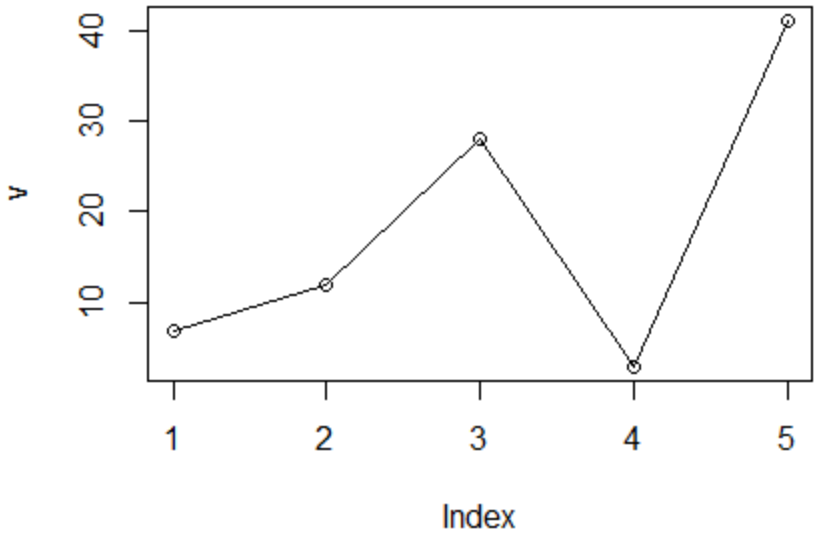
- **Syntax**

  plot(v,type,col,xlab,ylab,lty)
- **Description of the parameters**
  - **v** is a vector containing the numeric values.
  - **xlab** is the label for x axis.
  - **ylab** is the label for y axis.
  - **main** is the Title of the chart.
  - **col** is used to give colors to both the points and lines.

- **type**: character indicating the type of plotting. Allowed values are:
  - “p” for points
  - “l” for lines
  - “b” for both points and lines
  - “c” for empty points joined by lines
  - “o” for overplotted points and lines
  - “s” and “S” for stair steps
  - “n” does not produce any points or lines
- **lty**: line types. Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings “blank”, “solid”, “dashed”, “dotted”, “dotdash”, “longdash”, or “twodash”, where “blank” uses ‘invisible lines’ (i.e., does not draw them).

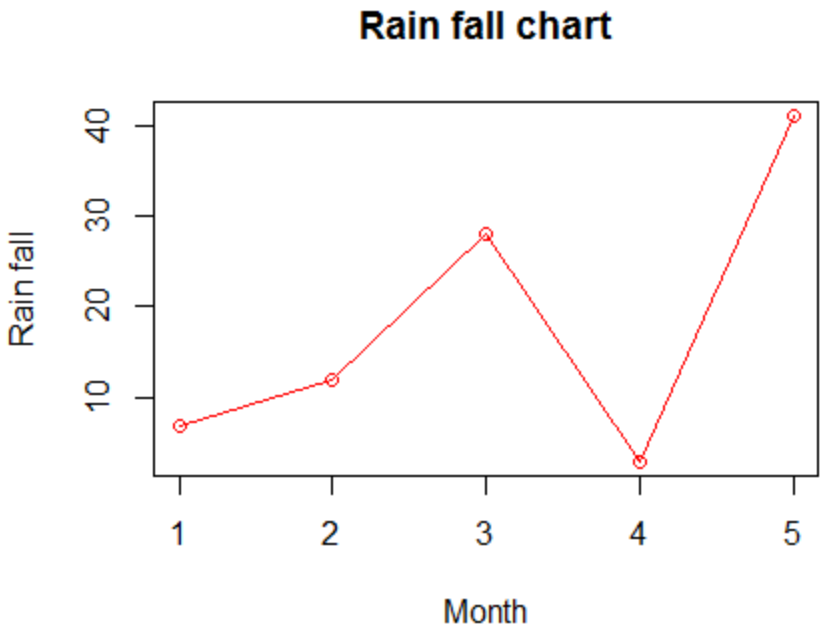
Example

```
v <- c(7,12,28,3,41)
plot(v,type = "o")
```



Line Chart Title, Color and Labels

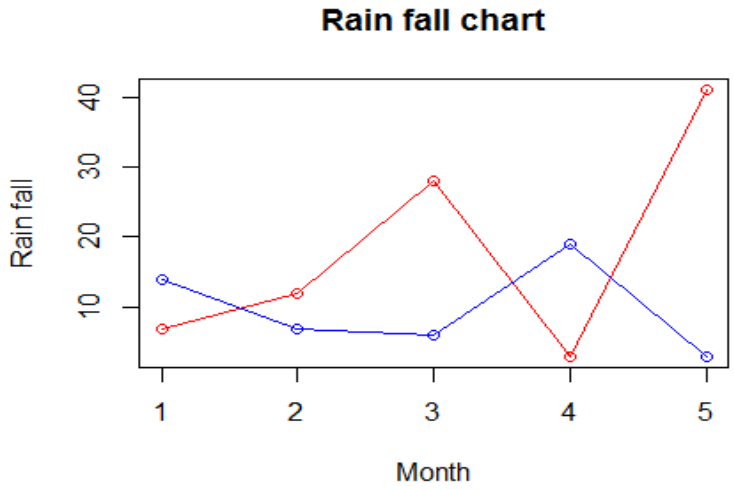
```
v <- c(7,12,28,3,41)
plot(v,type = "o", col = "red", xlab = "Month", ylab = "Rain
    fall",main = "Rain fall chart")
```



Multiple Lines in a Line Chart

- More than one line can be drawn on the same chart by using the **lines()**function.
- After the first line is plotted, the lines() function can use an additional vector as input to draw the second line in the chart,

```
v <- c(7,12,28,3,41)
t <- c(14,7,6,19,3)
plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall",
    main = "Rain fall chart")
lines(t, type = "o", col = "blue")
```



Scatterplots

- Scatterplots show many points plotted in the Cartesian plane.
- Each point represents the values of two variables.
- One variable is chosen in the horizontal axis and another in the vertical axis.
- The simple scatterplot is created using the **plot()** function.

Scatterplots

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

Example

- We use the data set "**mtcars**" available in the R environment to create a basic scatterplot.
- use the columns "wt" and "mpg" in mtcars.

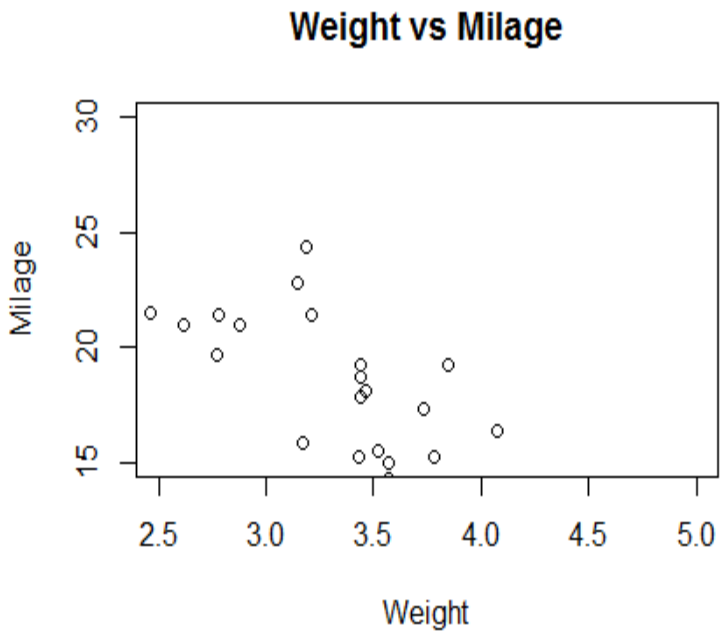
```
input <- mtcars[,c('wt','mpg')]
head(input)
```

Output:

	wt	mpg
Mazda RX4	2.620	21.0
Mazda RX4 Wag	2.875	21.0
Datsun 710	2.320	22.8
Hornet 4 Drive	3.215	21.4
Hornet Sportabout	3.440	18.7
Valiant	3.460	18.1

Creating the Scatterplot

```
input <- mtcars[,c('wt','mpg')]
plot(x = input$wt,y = input$mpg,
     xlab = "Weight",
     ylab = "Milage",
     xlim = c(2.5,5),
     ylim = c(15,30),
     main = "Weight vs Milage"
)
```



Scatterplot Matrices

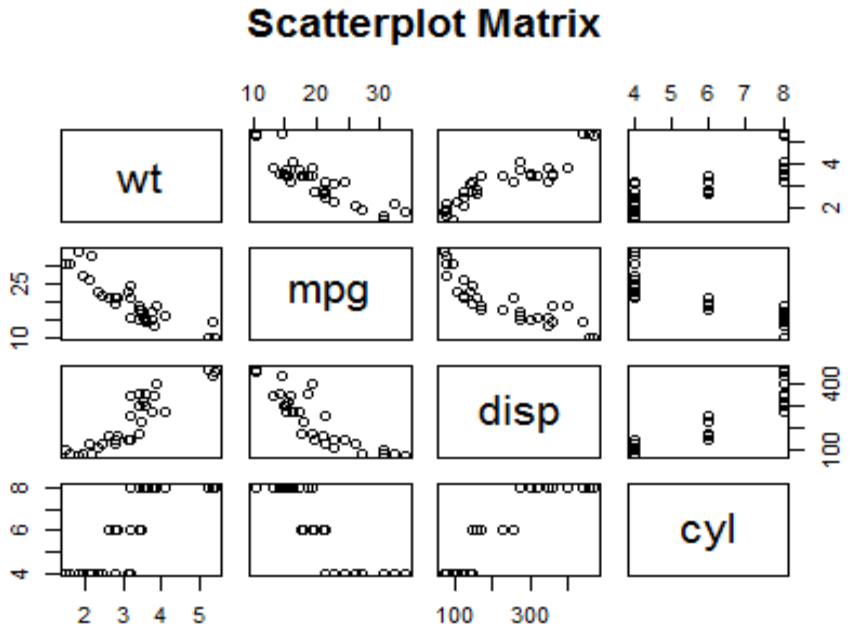
- When we have more than two variables and we want to find the correlation between one variable versus the remaining ones we use scatterplot matrix.
- We use **pairs()** function to create matrices of scatterplots.
- **Syntax**

  pairs(formula, data)
- **Description of the parameters**
  - **formula** represents the series of variables used in pairs.
  - **data** represents the data set from which the variables will be taken.

Example

- # Plot the matrices between 4 variables giving 12 plots.
- # One variable with 3 others and total 4 variables.

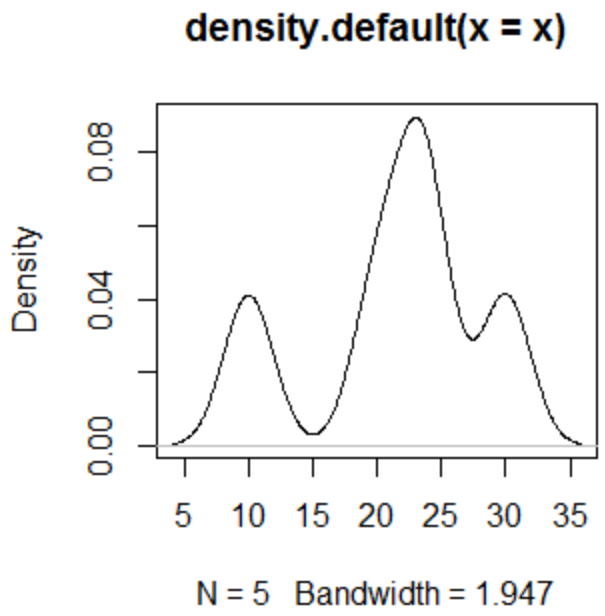
  pairs(~wt+mpg+disp+cyl,data = mtcars,main = "Scatterplot Matrix")



Kernel Density Plot

- Density plots can be thought of as plots of smoothed histograms.
- The smoothness is controlled by a bandwidth parameter.
- Kernel density plots are usually a much more effective way to view the distribution of a variable.

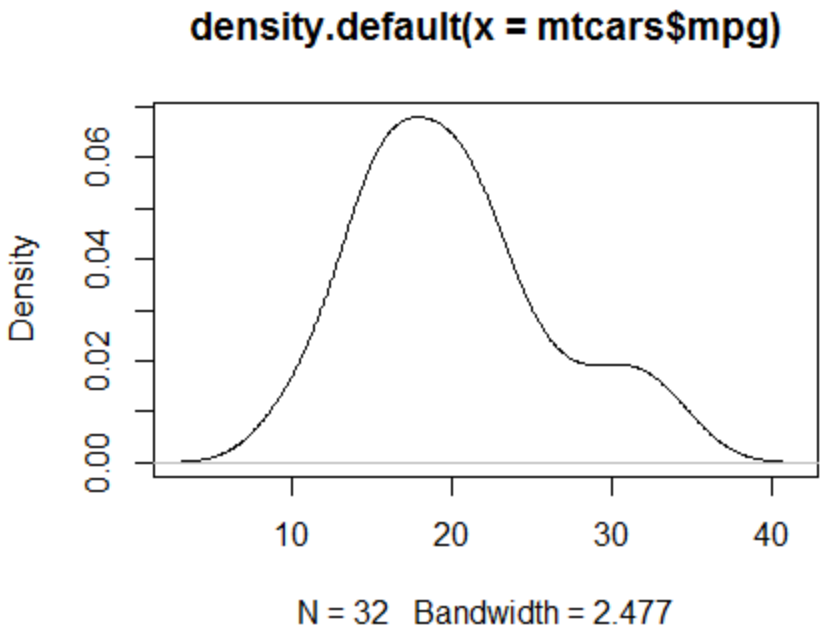
```
x <- c(10,20,24,23,30)
plot(density(x))
```



Kernel Density Plots

Example

```
d <- density(mtcars$mpg)
plot(d)
```

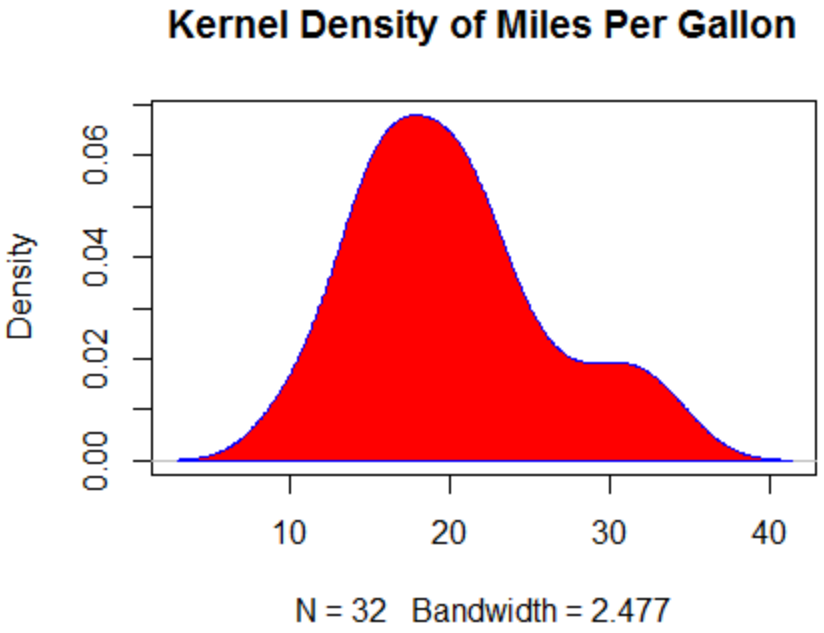


Example

```
d <- density(mtcars$mpg)
plot(d, main="Kernel Density of Miles Per Gallon")
```

Filled density plot

```
polygon(d, col="red", border="blue")
```



Writing plot to a file

- The first step in deciding how to save plots is to decide on the output format that you want to use.

Format	Driver	Notes
JPG	jpeg	be used anywhere, but doesn't resize
PNG	png	Can be used anywhere, but doesn't resize
WMF	win.metafile	Windows only; best choice with Word; easily resizable
PDF	pdf	Best choice with pdflatex; easily resizable
Postscript	postscript	Best choice with latex and Open Office; easily resizable

Methods

A General Method

```
jpeg('rplot.jpg')
plot(x,y)
dev.off()
```

Another Approach

```
dev.copy(png,'myplot.png')
dev.off()
```

Another Approach

Plots panel –> Export –> Save as Image or Save as PDF

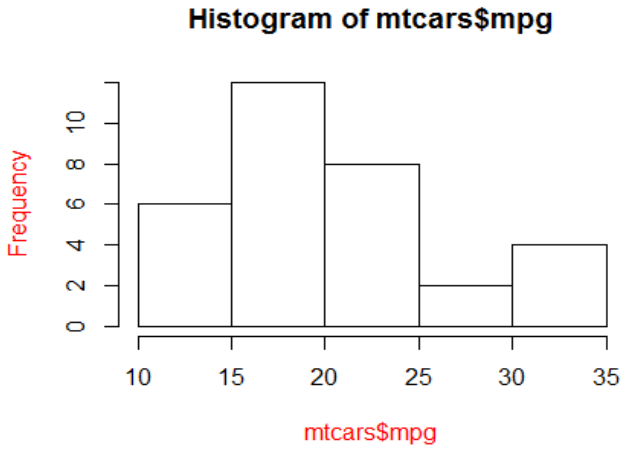
Changing graphical parameters

- You can customize many features of your graphs (fonts, colors, axes, titles) through graphic options.
- **One way** is to specify these options in the **par( )** function.
- If you set parameter values here, the changes will be in effect for the rest of the session or until you change them again.
- **Syntax**

  par(optionname=value, optionname=value, ...)

Set a graphical parameter using par()

```
par()                  # view current settings
opar <- par()          # make a copy of current settings
par(col.lab="red")     # red x and y labels
hist(mtcars$mpg)       # create a plot with these new settings
```




```
par(opar)              # restore original settings
```

Method 2

- specify graphical parameters by providing the **optionname**=value pairs directly to a high level plotting function.
- In this case, the options are only in effect for that specific graph.

Example

- hist(mtcars$mpg, col.lab="red")

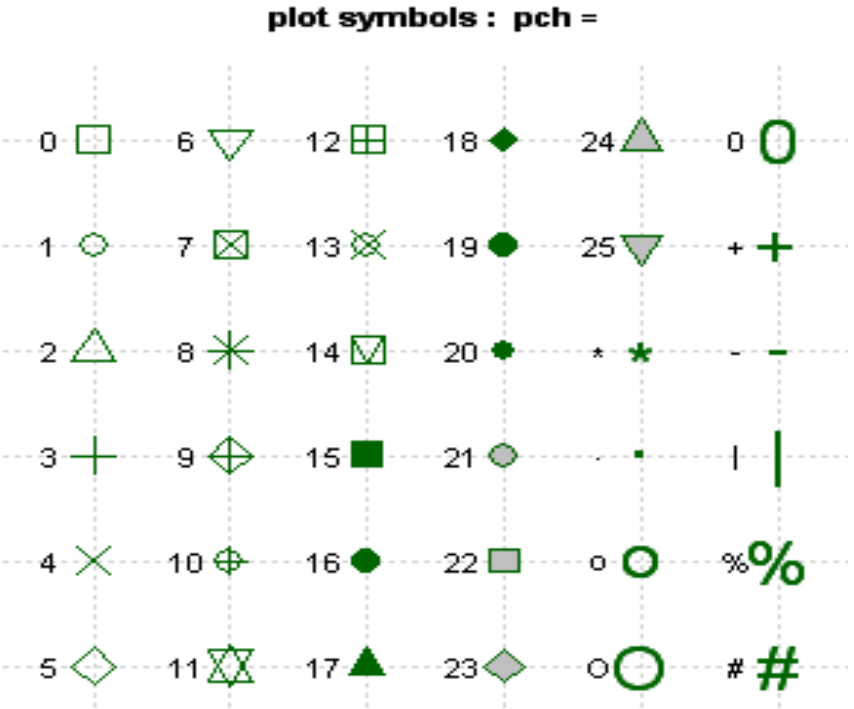
Text and Symbol Size

- The following options can be used to control text and symbol size in graphs.

Option	Description
cex	number indicating the amount by which plotting text and symbols should be scaled relative to the default. 1=default, 1.5 is 50% larger, 0.5 is 50% smaller, etc.
cex.axis	magnification of axis annotation relative to cex
cex.lab	magnification of x and y labels relative to cex
cex.main	magnification of titles relative to cex
cex.sub	magnification of subtitles relative to cex

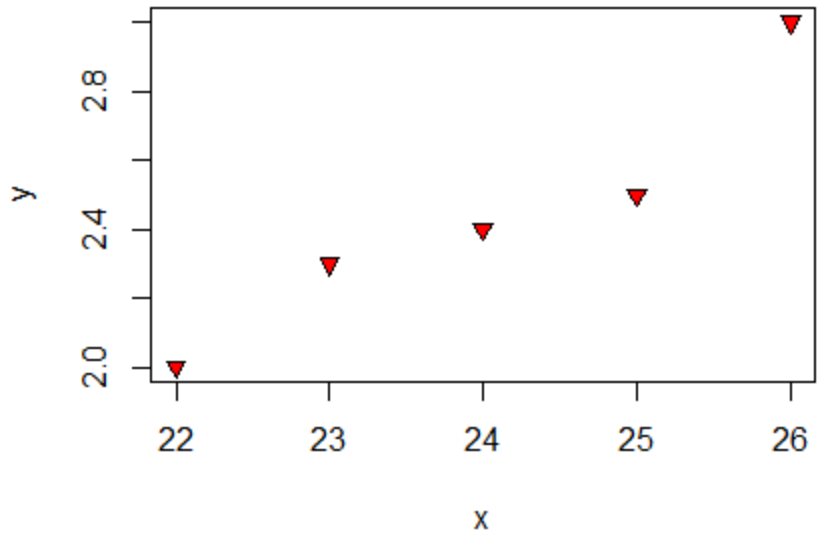
Plotting Symbols

- Use the **pch=** option to specify symbols to use when plotting points.
- For symbols 21 through 25, specify border color (col=) and fill color (bg=).



Example

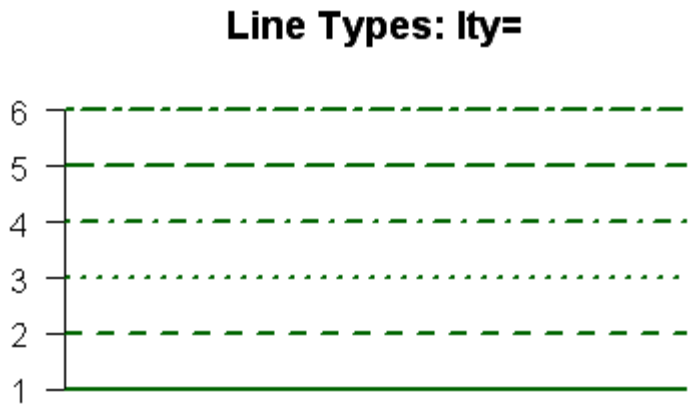
```
x <- c(22,23,24,25,26)
y <- c(2,2.3,2.4,2.5,3)
plot(x, y, pch = 25, col = "black", cex = 1,bg="red")
```



Lines

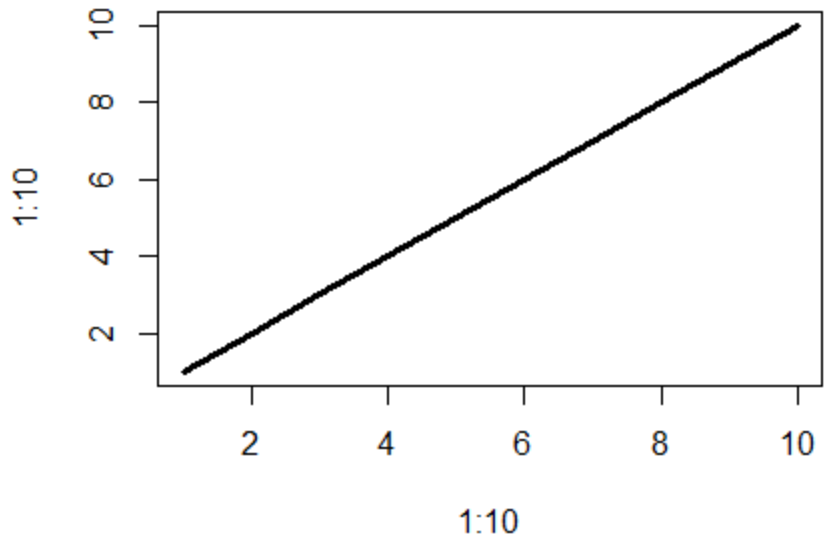
- You can change lines using the following options.
- This is particularly useful for reference lines, axes, and fit lines.

option	description
lty	line type. see the chart below.
lwd	line width relative to the default (default=1). 2 is twice as wide.



Lines - Example

```
plot(1:10, 1:10,type="l",lty=1,lwd=3)
```



Colors

- Options that specify colors include the following.

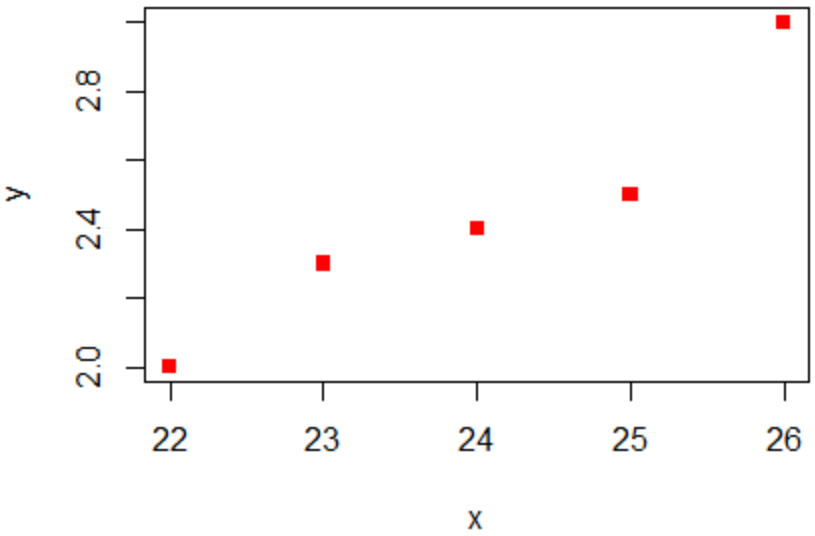
option	description
col	Default plotting color. Some functions (e.g. lines) accept a vector of values that are recycled.
col.axis	color for axis annotation
col.lab	color for x and y labels
col.main	color for titles
col.sub	color for subtitles
fg	plot foreground color (axes, boxes - also sets col= to same)
bg	plot background color

colors

- You can specify colors in R by index, name, hexadecimal, or RGB.
  For example **col=1**, **col="white"**, and **col="#FFFFFF"** are equivalent.

Example

```
x <- c(22,23,24,25,26)
y <- c(2,2.3,2.4,2.5,3)
plot(x, y, pch = 15, col = "red")
```



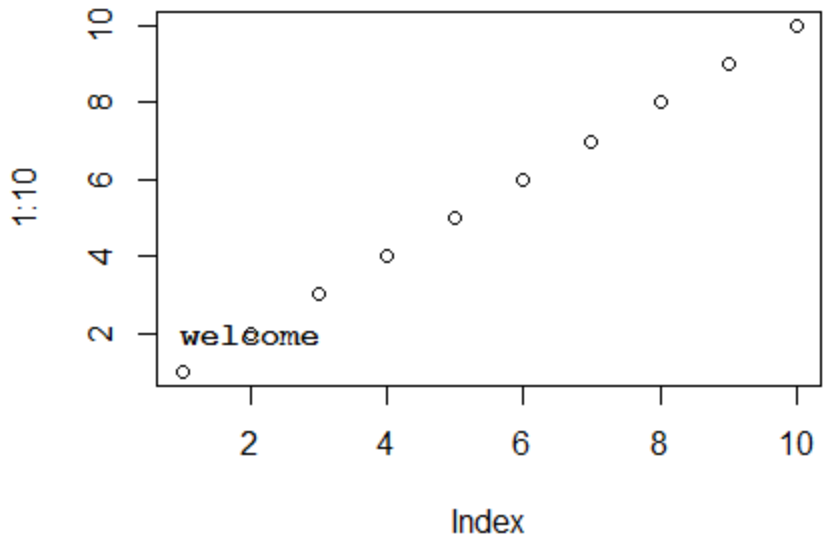
Fonts

- You can easily set font size and style and font family.

option	description
font	Integer specifying font to use for text. 1=plain, 2=bold, 3=italic, 4=bold italic, 5=symbol
font.axis	font for axis annotation
font.lab	font for x and y labels
font.main	font for titles
font.sub	font for subtitles
ps	font point size (roughly 1/72 inch) text size=ps*cex
family	font family for drawing text. Standard values are "serif", "sans", "mono", "symbol". Mapping is device dependent.

Fonts - Example

```
plot(1:10)
text(2, 2, "welcome", family = "mono", font=2)
```



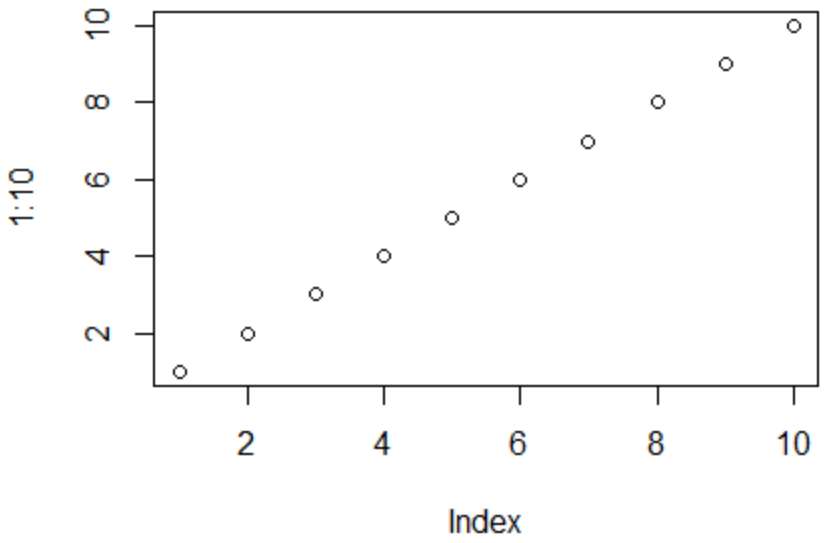
Margins and Graph Size

- You can control the margin size using the following parameters.

option	description
mar	numerical vector indicating margin size c(bottom, left, top, right) in lines. default = c(5, 4, 4, 2) + 0.1
mai	numerical vector indicating margin size c(bottom, left, top, right) in inches
pin	plot dimensions (width, height) in inches

Example

- plot(1:10, mar = c(2, 2, 2, 2))



UNIT III - PROBABILITY

Introduction-Sample Space -Events-Counting Methods-Conditional probability -Independent Events-Bayes Rule-Random Variables-Probability distribution-Discrete and continuous Distribution-Multivariate Distribution.

Introduction

- Probabilities are associated with experiments where the outcome is not known in advance or cannot be predicted.
- For example, if you toss a coin, will you obtain a head or tail? If you roll a die will obtain 1, 2, 3, 4, 5 or 6?
- The value of a probability is a number between 0 and 1 inclusive.
- An event that cannot occur has a probability equal to 0 and the probability of an event that is certain to occur has a probability equal to 1.

Deterministic and Random

- A **deterministic experiment** is one whose outcome may be predicted with certainty beforehand.

  Example - adding two numbers such as 2 + 3.

- A **random experiment** is one whose outcome is determined by chance. We posit that the outcome of a random experiment may not be predicted with certainty beforehand, even in principle.

  Example - tossing a coin, rolling a die

Sample space

- For a random experiment E, the set of all possible outcomes of E is called the sample space and is denoted by the letter S .
- **Example 1:** For the coin-toss experiment, S would be the results “Head” and “Tail”, which we may represented by

  S = {H, T}.
- **Example 2:** If a die is rolled, the sample space S is given by

  S = {1,2,3,4,5,6}
- **Example 3:** If two coins are tossed, the sample space S is given by

  S = {HH,HT,TH,TT}

How to do it with R

- A sample space is (usually) represented by a data frame, that is, a rectangular collection of variables.
- Each row of the data frame corresponds to an outcome of the experiment.
- **Example -**Consider the random experiment of dropping a Styrofoam cup onto the floor from a height of four feet.
- The cup hits the ground and eventually comes to rest. It could land upside down, right side up, or it could land on its side. We represent these possible outcomes of the random experiment by the following.

```
S <- data.frame(lands = c("down", "up", "side"))
```

Example

- Consider the random experiment of tossing a coin. The outcomes are H and T.
- We can set up the sample space quickly with the tosscoin function:

```
install.packages("prob")
library("prob")
tosscoin(1)
```

Output:

```
  Toss1
1     H
2     T
```

- The number 1 tells that we only want to toss the coin once.

Example

- **roll a fair die**

  rolldie(1)

Output:

  X1

1 1

2 2

3 3

4 4

5 5

6 6

- The rolldie function defaults to a 6-sided die, but we can specify others with the nsides argument.

- The command rolldie(1, nsides = 4) would be used to roll a 4-sided die three times.

  rolldie(1, nsides = 4)

Output:

```
  X1
1  1
2  2
3  3
4  4
```

Example

- we would like to draw one card from a standard set of playing cards (it is a long data frame):

  cards() - This generates a data frame sample space of a standard deck of 52 playing cards.

  cards(jokers = FALSE, makespace = FALSE)

Output:

rank suit

1 2 Club

2 3 Club

3 4 Club

4 5 Club

Sampling from Urns

- This is the most fundamental type of random experiment.
- We have an urn that contains a bunch of distinguishable objects (balls) inside.
- We shake up the urn, reach inside, grab a ball, and take a look.
- But there are all sorts of variations on this theme. Maybe we would like to grab more than one ball – say, two balls. What are all of the possible outcomes of the experiment now?
- **Sampling with replacement** - We could select a ball, take a look, put it back, and sample again.
- **Sampling without replacement** - select a ball, take a look – but do not put it back – and sample again

- Suppose we do not actually keep track of which ball came first.
- All we observe are the two balls, and we have no idea about the order in which they were selected.
- We call this **unordered sampling** (in contrast to ordered) because the order of the selections does not matter with respect to what we observe.
- Tossing a coin twice is equivalent to selecting two balls labeled H and T from an urn, with replacement.
- The die-roll experiment is equivalent to selecting a ball from an urn with six elements, labeled 1 through 6.

How to do it with R

- The prob package accomplishes sampling from urns with the **urnsamples** function.

Syntax

urnsamples(x, size, replace = FALSE, ordered = FALSE)

- **x** - represents the urn from which sampling is to be done.
- **size -** tells how large the sample will be.
- **ordered and replace -** arguments are logical and specify how sampling will be performed.

Example

- Let our urn simply contain three balls, labeled 1, 2, and 3, respectively. We are going to take a sample of size 2 from the urn.

Ordered, With Replacement

- If sampling is with replacement, then we can get any outcome 1, 2, or 3 on any draw.
- Further, by “ordered” we mean that we shall keep track of the order of the draws that we observe.

  urnsamples(1:3, size = 2, replace = TRUE, ordered = TRUE)

  A={1,2,3} and k=2

Output:

	X1	X2
1	1	1
2	2	1
3	3	1
4	1	2
5	2	2
6	3	2
7	1	3
8	2	3
9	3	3

Ordered, Without Replacement

- Here sampling is without replacement, so we may not observe the same number twice in any row. Order is still important.

  urnsamples(1:3, size = 2, replace = FALSE, ordered = TRUE)

  A={1,2,3} and k=2

Output:

	X1	X2
1	1	2
2	2	1
3	1	3
4	3	1
5	2	3
6	3	2

Unordered, Without Replacement

- we may not observe the same outcome twice, but in this case, we will only retain those outcomes which (when jumbled) would not duplicate earlier ones.

  urnsamples(1:3, size = 2, replace = FALSE, ordered = FALSE)

Output:

	X1	X2
1	1	2
2	1	3
3	2	3

Unordered, With Replacement

- We replace the balls after every draw, but we do not remember the order in which the draws came.

  urnsamples(1:3, size = 2, replace = TRUE, ordered = FALSE)

Output:

	X1	X2
1	1	1
2	1	2
3	1	3
4	2	2
5	2	3
6	3	3

Sampling from cards

```
S <- cards()
urnsamples(S, size = 2)
```

Output:

```
[[1000]]
      rank suit
26    A  Diamond
51    K   Spade
```

Example

- Consider an urn which has 5 red ball, 3 green balls and 8 blue balls. We randomly pick one ball from the urn.

  First, we need to carefully define the urn from which the sample is drawn. This is done by using rep function.

  rep(x) function replicates the values x

```
urn=rep(c("red","green","blue"),times=c(5,3,8))
urnsamples(urn,1)
```

Output:

```
  out
1 red 2 red 3 red 4 red 5 red 6 green 7 green 8 green 9 blue 10
  blue 11 blue 12 blue 13 blue 14 blue 15 blue 16 blue
```

Events

- An event A is a collection of outcomes, or in other words, a subset of the sample space.
- After the performance of a random experiment E we say that the event A occurred if the experiment's outcome belongs to A.
- For instance, in the coin-toss experiment the events A = {Heads} and B = {Tails} would be mutually exclusive.

Events

- When we say "Event“, it mean one (or more) outcomes.

Example Events:

- Getting a Tail when tossing a coin is an event
- Rolling a "5" is an event.

An event can include several outcomes:

- Choosing a "King" from a deck of cards (any of the 4 Kings) **is also** an event
- Rolling an "even number" (2, 4 or 6) is an event

Type of Events

Events can be:

- **Independent** - each event is **not** affected by other events
- **Dependent** - also called "Conditional", where an event **is** affected by other events
- **Mutually Exclusive** - events can't happen at the same time

Independent Events

- Events can be "Independent", meaning each event is **not affected** by any other events.

Example:

- You toss a coin three times and it comes up "Heads" each time ... what is the chance that the next toss will also be a "Head"?
- The chance is simply 1/2, or 50%, just like ANY OTHER toss of the coin.
- What it did in the past will not affect the current toss!

Dependent Events

- means they **can be affected by previous events**.

Example: Drawing 2 Cards from a Deck

- After taking one card from the deck there are **less cards** available, so the probabilities change!

look at the chances of getting a King.

- For the 1st card the chance of drawing a King is 4 out of 52
- But for the 2nd card: If the 1st card was a King, then the 2nd card is **less** likely to be a King, as only 3 of the 51 cards left are Kings.
- If the 1st card was **not** a King, then the 2nd card is slightly **more** likely to be a King, as 4 of the 51 cards left are King.

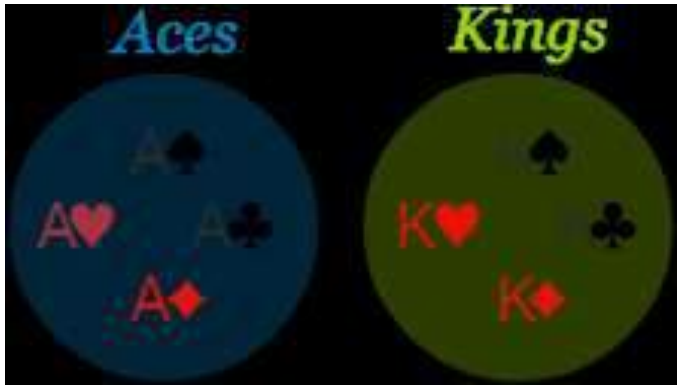
- **Replacement:** When we put each card **back** after drawing it the chances don't change, as the events are **independent**.
- **Without Replacement**: The chances will change, and the events are **dependent**.

Mutually Exclusive

- we can't get both events at the same time.

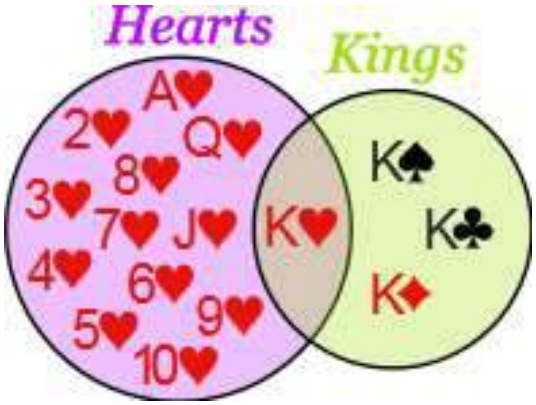
Examples:

- Turning left or right are Mutually Exclusive (you can't do both at the same time)
- Heads and Tails are Mutually Exclusive
- Kings and Aces are Mutually Exclusive



What isn't Mutually Exclusive

- Kings and Hearts are **not** Mutually Exclusive, because we can have a King of Hearts!



How to do it with R

Tossing two coins

```
S <- tosscoin(2, makespace = TRUE)
```

Output:

```
  toss1 toss2 probs
1 H      H     0.25
2 T      H     0.25
3 H      T      0.25
4 T      T      0.2
```

Given a data frame sample/probability space S, we may extract rows using the [] operator:

```
S[1:3, ]
```

Output:

```
  toss1 toss2 probs
1 H      H      0.25
2 T      H      0.25
3 H      T      0.25
```

```
S[c(2, 4), ]
```

Output:

```
  toss1 toss2 probs
2 T         H      0.25
4 T         T      0.25
```

We may also extract rows that satisfy a logical expression using the subset function, for instance

```
S <- cards()
subset(S, suit == "Heart")
```

Output:

```
      rank suit
27 2 Heart
28 3 Heart
29 4 Heart
30 5 Heart
 31 6 Heart
32 7 Heart
33 8 Heart
34 9 Heart
35 10 Heart
36 J Heart
37 Q Heart
38 K Heart
 39 A Heart
```

```
subset(S, rank %in% 7:9)
```

Output:

```
  rank suit
6 7 Club
7 8 Club
8 9 Club
19 7 Diamond
20 8 Diamond
21 9 Diamond
32 7 Heart
33 8 Heart
34 9 Heart
45 7 Spade
46 8 Spade
47 9 Spade
```

```
subset(rolldie(3), X1 + X2 + X3 > 16)
```

Output:

```
     X1 X2 X3
180  6   6    5
210  6   5    6
215  5   6    6
216  6   6    6
```

Functions for Finding Subsets

The %in% function

- The function %in% helps to learn whether each value of one vector lies somewhere inside another vector.

```
x <- 1:10
y <- 8:12
y %in% x
```

Output:

```
[1] TRUE TRUE TRUE FALSE FALSE
```

The isin function

- To check whether the whole vector y is in x. We can do this with the isin function.

```
 x <- 1:10
y <- c(3, 3, 7)
    all(y %in% x)
```

Output:

[1] TRUE

```
     isin(x, y)
```

Output:

[1] FALSE

- argument ordered which tests whether the elements of y appear in x in the order in which they are appear in y.

```
isin(x, c(3, 4, 5), ordered = TRUE)
```

Output:

[1] TRUE

```
isin(x,c(3,5,4),ordered=T)
```

Output:

[1] FALSE

- The connection to probability is that have a data frame sample space and we would like to find a subset of that space.
- A data.frame method was written for isin that simply applies the function to each row of the data frame.

```
S <- rolldie(4)
subset(S, isin(S, c(2, 2, 6), ordered = TRUE))
```

Output:

```
     X1 X2 X3 X4
188   2  2  6  1
404   2  2  6  2
620   2  2  6  3
836   2  2  6  4
1052  2  2  6  5
1088  2  2  1  6
```

Set Union, Intersection, and Difference

- Given subsets A and B, it is often useful to manipulate them in an algebraic fashion.
- we have three set operations - union, intersection, and difference.

Name	Denoted	Defined by elements	Code
Union	A U B	in A or B or both	union(A,B)
Intersection	A ∩B	in both A and B	intersect(A,B)
Difference	A\B	in A but not in B	setdiff(A,B)

Union

```
S = cards()
A = subset(S, suit == "Heart")
B = subset(S, rank %in% 7:9)
union(A,B)
```

Output:

	rank	suit
6	7	Club
7	8	Club
8	9	Club
19	7	Diamond
20	8	Diamond
21	9	Diamond
27	2	Heart
28	3	Heart

Intersect and Set

intersect(A, B)

Output:

rank suit

32 7 Heart

33 8 Heart

34 9 Heart

setdiff(A, B)

Output:

rank suit

27 2 Heart

28 3 Heart

29 4 Heart

30 5 Heart

31 6 Heart

35 10 Heart

Counting Methods

- The equally-likely model is a convenient and popular way to analyze random experiments.
- **Equally likely events** are those events which have an equal probability of occuring.
- **For example**: When we toss an unbiased coin, the probability of getting a heads is 1/2 and the probability of getting a tails is 1/2. So, it is an equally likely event.
- when the equally likely model applies, finding the probability of an event A is equal to counting the number of outcomes that A contains (together with the number of events in S).

The Multiplication Principle

- Suppose that an experiment is composed of two successive steps.
- The first step may be performed in n1 distinct ways while the second step may be performed in n2 distinct ways. Then the experiment may be performed in n1n2 distinct ways.
- More generally, if the experiment is composed of k successive steps which may be performed in n1, n2, . . . , nk distinct ways, respectively, then the experiment may be performed in $n_1 n_2 \cdot \cdot \cdot n_k$ distinct ways.

Sample problem

- A fast-food restaurant has a meal special: $5 for a drink, sandwich, side item and dessert.
- **The choices are:**
  - Sandwich: Grilled chicken, All Beef Patty, Vegeburger and Fish Filet.
  - Side: Regular fries, Cheese Fries, Potato Wedges.
  - Dessert: Chocolate Chip Cookie or Apple Pie.
  - Drink: Fanta, Dr. Pepper, Coke, Diet Coke and Sprite.
- **Q. How many meal combos are possible?**
- There are 4 stages:
  - Choose a sandwich.
  - Choose a side.
  - Choose a dessert.
  - Choose a drink.
- There are 4 different types of sandwich, 3 different types of side, 2 different types of desserts and five different types of drink.
- The number of meal combos possible is 4 * 3 * 2 * 5 = 120.

- You take a survey for five questions with "yes" or "no" answers. How many different ways could you complete the survey?
- There are 5 stages: Question 1, question 2, question 3, question 4, and question 5.
- There are 2 choices for each question (Yes or No). So the total number of possible ways to answer is: $2 * 2 * 2 * 2 * 2 = 2^5 = 32$.

Q: A company puts a code on each different product they sell. The code is made up of 3 numbers and 2 letters. How many different codes are possible?

There are 5 stages (number 1, number 2, number 3, letter 1 and letter 2).
There are 10 possible numbers: 0 – 9.
There are 26 possible letters: A – Z.
So we have:
10 * 10 * 10 * 26 * 26 = 676000 possible codes.

Ordered Samples

- Imagine a bag with n distinguishable balls inside. Now shake up the bag and select k balls at random. How many possible sequences might we observe?
- **Proposition-** The number of ways in which one may select an ordered sample of k subjects from a population that has n distinguishable members is
- n^k if sampling is done with replacement,
- $n(n-1)(n-2)\cdots(n-k+1)$ if sampling is done without replacement.

Examples

- Take a coin and flip it 7 times. How many sequences of Heads and Tails are possible?

- In a class of 20 students, we randomly select a class president, a class vicepresident, and a treasurer. How many ways can this be done?

- We rent five movies to watch over the span of two nights. We wish to watch 3 movies on the first day. How many distinct sequences of 3 movies could we possibly watch?

- **Answer: $2^7 = 128$.**
- **Answer: $20 \cdot 19 \cdot 18 = 6840$.**
- **Answer: $5 \cdot 4 \cdot 3 = 60$.**

Unordered Samples

- The number of ways in which one may select an unordered sample of k subjects from a population that has n distinguishable members is
- (n − 1 + k)!/[(n − 1)!k!] if sampling is done with replacement,
- n!/[k!(n − k)!] if sampling is done without replacement.
- The quantity n!/[k!(n − k)!] is called a binomial coefficient and plays a special role in mathematics; it is denoted

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{k \cdot (k-1) \cdot (k-2) \cdots 2 \cdot 1}$$

and is read “n choose k”.

	ordered = TRUE	ordered = FALSE
• replace = TRUE	n^k	$(n-1+k)!/(n-1)!k!$
• replace = FALSE	$n!/(n-k)!$	$\binom{n}{k}$

- You rent five movies to watch over the span of two nights, but only wish to watch 3 movies the first day. Your friend, Fred, wishes to borrow some movies to watch at his house on the first day. You allow him to select 2 movies from the set of 5. How many choices does Fred have?
- **Answer:**

$$\binom{5}{2} = 10$$

How to do it with R

- The factorial n! is computed with the command factorial(n)
- The binomial coefficient $\binom{n}{k}$ with the command choose(n,k).
- We will compute the number of outcomes for each of the four urnsamples
- examples . Recall that we took a sample of size two from an urn with three distinguishable elements.

```
    nsamp(n = 3, k = 2, replace = TRUE, ordered = TRUE)
[1] 9
    nsamp(n = 3, k = 2, replace = FALSE, ordered = TRUE)
[1] 6
    nsamp(n = 3, k = 2, replace = FALSE, ordered = FALSE)
[1] 3
    nsamp(n = 3, k = 2, replace = TRUE, ordered = FALSE)
[1] 6
```

The Multiplication Principle

- A benefit of nsamp is that it is vectorized so that entering vectors instead of numbers for n, k, replace, and ordered results in a vector of corresponding answers.
- This becomes particularly convenient for combinatorics problems.

Example

Question

- There are 11 artists who each submit a portfolio containing 7 paintings for competition in an art exhibition. Unfortunately, the gallery director only has space in the winners' section to accommodate 12 paintings in a row equally spread over three consecutive walls. The director decides to give the first, second, and third place winners each a wall to display the work of their choice. The walls boast 31 separate lighting options apiece. How many displays are possible?

Answer

- The judges will pick 3 (ranked) winners out of 11 (with rep = FALSE, ord =TRUE).
- Each artist will select 4 of his/her paintings from 7 for display in a row (rep = FALSE, ord = TRUE), and lastly, each of the 3 walls has 31 lighting possibilities (rep = TRUE, ord= TRUE). These three numbers can be calculated quickly with

```
n <- c(11, 7, 31)
k <- c(3, 4, 3)
r <- c(FALSE, FALSE, TRUE)
x <- nsamp(n, k, rep = r, ord = TRUE)
```

Output: [1] 990 840 29791

- By the Multiplication Principle, the number of ways to complete the experiment is the product of the entries of x:

  prod(x)

Output: [1] 24774195600

- Compare this with the some other ways to compute the same thing:

  (11 * 10 * 9) * (7 * 6 * 5 * 4) * 31^3

  [1] 24774195600

  or alternatively

  prod(9:11) * prod(4:7) * 31^3

  [1] 24774195600

Conditional Probability

- Consider a full deck of 52 standard playing cards. Now select two cards from the deck, in succession.
- Let A = {first card drawn is an Ace} and B = {second card drawn is an Ace}.
- Since there are four Aces in the deck, it is natural to assign P(A) = 4/52.
- Suppose we look at the first card. What now is the probability of B? Of course, the answer depends on the value of the first card.
- If the first card is an Ace, then the probability that the second also is an Ace should be 3/51.

- if the first card is not an Ace, then the probability that the second is an Ace should be 4/51.
- As notation for these two situations we write

$$P(B|A) = 3/51,\ P(B|A^c) = 4/51.$$

The **conditional probability** of B given A, denoted P(B|A), is defined by

$$\mathbb{P}(B|A) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(A)}, \quad \text{if } \mathbb{P}(A) > 0.$$

Example

- Toss a coin twice. The sample space is given by

  $$S = \{HH, HT, TH, TT\}.$$

- Let A = {a head occurs} and B = {a head and tail occur}.
- It should be clear that $P(A) = 3/4$, $P(B) = 2/4$, and $P(A \cap B) = 2/4$.
- What now are the probabilities $P(A|B)$ and $P(B|A)$?

  $$P(A|B) = IP(A \cap B) / P(B) = 2/4 / 2/4 = 1,$$

- in other words, once we know that a Head and Tail occur, we may be certain that a Head occurs. Next

  $$P(B|A) = P(A \cap B) \ / P(A) \ = 2/4 / 3/4 \ = 2 / 3$$

How to do in R

```
s <- tosscoin(2,makespace=T)
s
```

Output:

```
  toss1 toss2 probs
1 H     H     0.25
2 T     H     0.25
3 H     T     0.25
4 T     T     0.25
```

- Next we define the events

```
A <- s[1:3,]
B <- s[2:3,]
```

- To do conditional probability, we use the given argument of the prob function:

```
Prob(A,given=B)
```

Output:

```
[1] 1
```

```
Prob(B,given=A)
```

Output:

```
[1] 0.6666667
```

- Roll a six-sided die twice. The sample space consists of all ordered pairs (i, j) of the numbers 1, 2, . . . , 6, that is, S = {(1, 1), (1, 2), . . . , (6, 6)}. #S= 6^2 = 36.
- Let A = {outcomes match} and B = {sum of outcomes at least 8}.
- The sample space may be represented by a matrix:

	1	2	3	4	5	6
1	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)
2	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)
3	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)
4	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)
5	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)
6	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)

- Now it is clear that

  P(A) = 6/36, P(B) = 15/36, and P(A ∩B) = 3/36.

Finally,

$$\mathbb{P}(A|B) = \frac{3/36}{15/36} = \frac{1}{5}, \quad \mathbb{P}(B|A) = \frac{3/36}{6/36} = \frac{1}{2}.$$

How to do it with R

```
S <- rolldie(2, makespace = TRUE)
head(S)
```

Output:

```
  X1 X2 probs
1  1  1 0.02777778
2  2  1 0.02777778
3  3  1 0.02777778
4  4  1 0.02777778
5  5  1 0.02777778
6  6  1 0.02777778
```

- Next we define the events

  A <- subset(S, X1 == X2)

  B <- subset(S, X1 + X2 >= 8)
- To do conditional probability, we use the given argument of the prob function:

  prob(A, given = B)

Output:

[1] 0.2

prob(B, given = A)

Output:

[1] 0.5

- we do not actually need to define the events A and B separately as long as we reference the original probability space S as the first argument of the prob calculation:

  Prob(S, X1==X2, given = (X1 + X2 >= 8))

Output:

[1] 0.2

Prob(S, X1+X2 >= 8, given = (X1==X2))

Output:

[1] 0.5

Properties and Rules

- The following theorem establishes that conditional probabilities behave just like regular probabilities when the conditioned event is fixed.

For any fixed event A with P(A) > 0,

1. P(B|A) ≥ 0, for all events B ⊂ S ,
2. P(S |A) = 1, and
3. If B1, B2, B3,. . . are disjoint events, then

$$\mathbb{P}\left(\bigcup_{k=1}^{\infty} B_k \,\middle|\, A\right) = \sum_{k=1}^{\infty} \mathbb{P}(B_k|A).$$

For any events A, B, and C with P(A) > 0,

1. $\mathbb{P}(B^c|A) = 1 - \mathbb{P}(B|A)$.

2. *If* $B \subset C$ *then* $\mathbb{P}(B|A) \leq \mathbb{P}(C|A)$.

3. $\mathbb{P}[(B \cup C)|A] = \mathbb{P}(B|A) + \mathbb{P}(C|A) - \mathbb{P}[(B \cap C|A)]$.

4. ***The Multiplication Rule.*** *For any two events A and B,*

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\,\mathbb{P}(B|A).$$

And more generally, for events $A_1, A_2, A_3, \ldots, A_n$,

$$\mathbb{P}(A_1 \cap A_2 \cap \cdots \cap A_n) = \mathbb{P}(A_1)\,\mathbb{P}(A_2|A_1) \cdots \mathbb{P}(A_n|A_1 \cap A_2 \cap \cdots \cap A_{n-1}).$$

- The Multiplication Rule is very important because it allows us to find probabilities in random experiments that have a sequential structure.
- **Example -** At the beginning of the section we drew two cards from a standard playing deck. Now we may answer our original question, what is P(both Aces)?

$$\mathbb{P}(\text{both Aces}) = \mathbb{P}(A \cap B) = \mathbb{P}(A)\,\mathbb{P}(B|A) = \frac{4}{52} \cdot \frac{3}{51} \approx 0.00452.$$

How to do it with R

- First we employ the cards function to get a data frame L with two columns: rank and suit. Both columns are stored internally as factors with 13 and 4 levels, respectively.

  L <- cards()

- Next we sample two cards randomly from the L data frame by way of the urnsamples function. It returns a list M which contains all possible pairs of rows from L (there are choose(52,2) of them). The sample space for this experiment is exactly the list M.

  M <- urnsamples(L, size = 2)

- we associate a probability model with the sample space.

  N <- probspace(M)

- Now that we have the probability space N we are ready to do some probability.
- We use the prob function. The only trick is to specify the event of interest correctly, and recall that we were interested in P(both Aces).
- But if the cards are both Aces then the rank of both cards should be "A", which sounds like a job for the all function:

```
Prob(N, all(rank=="A"))
```

Output:

```
[1] 0.004524887
```

- Consider an urn with 10 balls inside, 7 of which are red and 3 of which are green. Select 3 balls successively from the urn. Let A = 1st ball is red, B = 2nd ball is red, and C =3rd ball is red. Then

$$\mathbb{P}(\text{all 3 balls are red}) = \mathbb{P}(A \cap B \cap C) = \frac{7}{10} \cdot \frac{6}{9} \cdot \frac{5}{8} \approx 0.2917.$$

How to do it with R

- We need to set up an urn (vector L) to hold the balls,
- we sample from L to get the sample space (data frame M), and
- we associate a probability vector (column probs) with the outcomes (rows of M) of the sample space. The final result is a probability space (an ordinary data frame N).

```
L <- rep(c("red", "green"), times = c(7, 3))

M <- urnsamples(L, size = 3, replace = FALSE, ordered = TRUE)

N <- probspace(M)
```

- Now let us think about how to set up the event {all 3 balls are red}. Rows of N that satisfy this condition have X1=="red"& X2=="red"& X3=="red“.

  Prob(N,X1=="red" & X2=="red" & X3=="red")

Output:

  [1] 0.2916667

- The isrep function (short for “is repeated”) in the prob package was written for this purpose.
- The command isrep(N,"red",3) will test each row of N to see whether the value "red" appears 3 times. The result is exactly what we need to define an event with the prob function.

  Prob(N, isrep(N, "red", 3))

Output:

  [1] 0.2916667

- What is the probability of getting two "red"s?

  prob(N, isrep(N, "red", 2))

Output: [1] 0.525

- What is the probability of observing "red", then "green", then "red"?

  prob(N, isin(N, c("red", "green", "red"), ordered = TRUE))

Output: [1] 0.175

- What is the probability of observing "red", "green", and "red", in no particular order?

  prob(N, isin(N, c("red", "green", "red")))

Output: [1] 0.525

Independent Events

- Toss a coin twice. The sample space is S = {HH, HT, TH, TT}. We know that P(1st toss is H) = 2/4, P(2nd toss is H) = 2/4, and P(both H) = 1/4. Then

$$\begin{aligned}\mathbb{P}(2^{\text{nd}}\text{ toss is } H \mid 1^{\text{st}}\text{ toss is } H) &= \frac{\mathbb{P}(\text{both } H)}{\mathbb{P}(1^{\text{st}}\text{ toss is } H)},\\ &= \frac{1/4}{2/4},\\ &= \mathbb{P}(2^{\text{nd}}\text{ toss is } H).\end{aligned}$$

- Events A and B are said to be independent if

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\,\mathbb{P}(B).$$

- Otherwise, the events are said to be dependent.
- The connection with the above example stems from the following. We know that when IP(B) > 0 we may write

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

- In the case that A and B are independent, the numerator of the fraction factors so that IP(B) cancels with the result:

$$\mathbb{P}(A|B) = \mathbb{P}(A) \quad \text{when } A, B \text{ are independent.}$$

- The events A, B, and C are mutually independent if the following four conditions are met:

$$\begin{aligned}\mathbb{P}(A \cap B) &= \mathbb{P}(A)\,\mathbb{P}(B),\\ \mathbb{P}(A \cap C) &= \mathbb{P}(A)\,\mathbb{P}(C),\\ \mathbb{P}(B \cap C) &= \mathbb{P}(B)\,\mathbb{P}(C),\end{aligned}$$

and

$$\mathbb{P}(A \cap B \cap C) = \mathbb{P}(A)\,\mathbb{P}(B)\,\mathbb{P}(C).$$

- If only the first three conditions hold then A, B, and C are said to be independent pairwise.
- Note that pairwise independence is not the same as mutual independence when the number of events is larger than two.

- Toss ten coins. What is the probability of observing at least one Head?
- Let Ai ={the ith coin shows H}, i = 1, 2, . . . , 10.
- Supposing that we toss the coins in such a way that they do not interfere with each other, this is one of the situations where all of the Ai may be considered mutually independent due to the nature of the tossing. Of course, the only way that there will not be at least one Head showing is if all tosses are Tails. Therefore,

$$\begin{aligned}
\mathbb{P}(\text{at least one } H) &= 1 - \mathbb{P}(\text{all } T), \\
&= 1 - \mathbb{P}(A_1^c \cap A_2^c \cap \cdots \cap A_{10}^c), \\
&= 1 - \mathbb{P}(A_1^c)\,\mathbb{P}(A_2^c) \cdots \mathbb{P}(A_{10}^c), \\
&= 1 - \left(\frac{1}{2}\right)^{10},
\end{aligned}$$

- which is approximately 0.9990234.

How to do it with R

```
S <- tosscoin(10, makespace = TRUE)
A <- subset(S, isrep(S, vals = "T", nrep = 10))
1 - Prob(A)
```

Output:

```
[1] 0.9990234
```

Independent, Repeated Experiments

- It is common to repeat a certain experiment multiple times under identical conditions and in an independent manner. We have seen many examples of this already: tossing a coin repeatedly, rolling a die or dice, etc.
- The **iidspace** function was designed specifically for this situation. It has three arguments:
  - x, which is a vector of outcomes,
  - ntrials, which is an integer telling how many times to repeat the experiment,
  - probs to specify the probabilities of the outcomes of x in a single trial.

- We may represent tossing our unbalanced coin three times with the following:

  iidspace(c("H","T"), ntrials = 3, probs = c(0.7, 0.3))

Output:

	X1	X2	X3	probs
1	H	H	H	0.343
2	T	H	H	0.147
3	H	T	H	0.147
4	T	T	H	0.063
5	H	H	T	0.147
6	T	H	T	0.063
7	H	T	T	0.063
8	T	T	T	0.027

Bayes' Rule

- Let B1, B2, . . . , Bn be mutually exclusive and exhaustive and let A be an event with P(A) > 0. Then

$$\mathbb{P}(B_k|A) = \frac{\mathbb{P}(B_k)\,\mathbb{P}(A|B_k)}{\sum_{i=1}^{n}\mathbb{P}(B_i)\,\mathbb{P}(A|B_i)}, \quad k = 1, 2, \ldots, n.$$

- Proof. The proof follows from looking at $P(B_k \cap A)$ in two different ways. For simplicity, suppose that $P(B_k) > 0$ for all k. Then

$$\mathbb{P}(A)\,\mathbb{P}(B_k|A) = \mathbb{P}(B_k \cap A) = \mathbb{P}(B_k)\,\mathbb{P}(A|B_k).$$

- Since P(A) > 0 we may divide through to obtain

$$\mathbb{P}(B_k|A) = \frac{\mathbb{P}(B_k)\,\mathbb{P}(A|B_k)}{\mathbb{P}(A)}.$$

- Now remembering that $\{B_k\}$ is a partition, the Theorem of Total Probability gives the denominator of the last expression to be

$$\mathbb{P}(A) = \sum_{k=1}^{n} \mathbb{P}(B_k \cap A) = \sum_{k=1}^{n} \mathbb{P}(B_k)\,\mathbb{P}(A|B_k).$$

Misfiling Assistants.

- In this problem, there are three assistants working at a company: Moe, Larry, and Curly. Their primary job duty is to file paperwork in the filing cabinet when papers become available. The three assistants have different work schedules:

	Moe	Larry	Curly
Workload	60%	30%	10%

- That is, Moe works 60% of the time, Larry works 30% of the time, and Curly does the remaining 10%, and they file documents at approximately the same speed.
- Suppose a person were to select one of the documents from the cabinet at random.

- Let M be the event

M = {Moe filed the document}

and let L and C be the events that Larry and Curly, respectively, filed the document.

What are these events' respective probabilities?

In the absence of additional information, reasonable prior probabilities would just be

	Moe	Larry	Curly
Prior Probability	$\mathbb{P}(M) = 0.60$	$\mathbb{P}(L) = 0.30$	$\mathbb{P}(C) = 0.10$

- Now, the boss comes in one day, opens up the file cabinet, and selects a file at random. The boss discovers that the file has been misplaced. The boss is so angry at the mistake that (s)he threatens to fire the one who erred. The question is: who misplaced the file?
- The boss has information about Moe, Larry, and Curly's filing accuracy in the past (due to historical performance evaluations). The performance information may be represented by the following table:

	Moe	Larry	Curly
Misfile Rate	0.003	0.007	0.010

- Moe misfiles 0.3% of the documents he is supposed to file.
- Notice that Moe was correct: he is the most accurate filer, followed by Larry, and lastly Curly.

- The boss would like to use this updated information to update the probabilities for the three assistants, that is, (s)he wants to use the additional likelihood that the document was misfiled to update his/her beliefs about the likely culprit.
- Let A be the event that a document is misfiled.
- What the boss would like to know are the three probabilities

$$\mathbb{P}(M|A),\ \mathbb{P}(L|A),\ \text{and}\ \mathbb{P}(C|A).$$

- We use Bayes' Rule in the form

$$\mathbb{P}(M|A) = \frac{\mathbb{P}(M \cap A)}{\mathbb{P}(A)}.$$

- Let's try to find IP(M ∩ A), which is just P(M) ·P(A|M) by the Multiplication Rule.
- We already know IP(M) = 0.6 and P(A|M) is nothing more than Moe's misfile rate, given above to be IP(A|M) = 0.003. Thus, we compute

  $$P(M \cap A) = (0.6)(0.003) = 0.0018.$$

- Using the same procedure we may calculate

  $$P(L \cap A) = 0.0021 \text{ and } P(C \cap A) = 0.0010.$$

- Further, these possibilities are mutually exclusive. We may use the Theorem of Total Probability to write

  $$P(A) = P(M \cap A) + P(L \cap A) + P(C \cap A)$$

- Thus

  $$P(A) = 0.0018 + 0.0021 + 0.0010 = 0.0049.$$

- Therefore, Bayes' Rule yields

$$\mathbb{P}(M|A) = \frac{0.0018}{0.0049} \approx 0.37.$$

- This last quantity is called the posterior probability that Moe misfiled the document, since it incorporates the observed data that a randomly selected file was misplaced (which is governed by the misfile rate). We can use the same argument to calculate

	Moe	Larry	Curly
Posterior Probability	$\mathbb{P}(M\|A) \approx 0.37$	$\mathbb{P}(L\|A) \approx 0.43$	$\mathbb{P}(C\|A) \approx 0.20$

- The conclusion: **Larry gets the axe.**

How to do it with R

- **Misfiling assistants**
- We store the prior probabilities and the likelihoods in vectors.

```
prior <- c(0.6, 0.3, 0.1)
like <- c(0.003, 0.007, 0.01)
post <- prior * like
post/sum(post)
```

Output:

```
[1] 0.3673469 0.4285714 0.2040816
```

Bayes' Theorem

- Bayes' Theorem is a way of finding a probability when we know certain other probabilities.
- The formula is:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where:

- P(A|B) – the probability of event A occurring, given event B has occurred
- P(B|A) – the probability of event B occurring, given event A has occurred
- P(A) – the probability of event A
- P(B) – the probability of event B

Example: Picnic Day

You are planning a picnic today, but the morning is cloudy

- Oh no! 50% of all rainy days start off cloudy!
- But cloudy mornings are common (about 40% of days start cloudy)
- And this is usually a dry month (only 3 of 30 days tend to be rainy, or 10%)

- **What is the chance of rain during the day?**

- We will use Rain to mean rain during the day, and Cloud to mean cloudy morning.
- The chance of Rain given Cloud is written P(Rain|Cloud)
- So let's put that in the formula:

P(Rain|Cloud) = *P(Rain) P(Cloud|Rain) /* P(Cloud)

P(Rain) is Probability of Rain = 10%

P(Cloud|Rain) is Probability of Cloud, given that Rain happens = 50%

P(Cloud) is Probability of Cloud = 40%

P(Rain|Cloud) = *0.1 X 0.5 /* 0.4 = .125

Or a 12.5% chance of rain.

- Not too bad, let's have a picnic!

How to do in R

```
probCloudgivenrain <-0.5
probcloud <-0.4
probrain <- 0.1
probraingivencloud <- (probrain *
    probCloudgivenrain)/probcloud
Probrainivencloud
```

Output:

```
[1] 0.125
```

- Imagine 100 people at a party, and you tally how many wear pink or not, and if a man or not, and get these numbers:

	Pink	Not Pink
Man	5	35
Not Man	20	40

Can you discover **P(Man|Pink)** ?

P(Man|Pink) = *P(Man) P(Pink|Man) /* P(Pink)

	Pink	Not Pink	
Man	5	35	40
Not Man	20	40	60
	25	75	100

- the probability of being a man is P(Man) = 40/ **100** = 0.4
- the probability of wearing pink is P(Pink) = 25/**100** = 0.25
- the probability that a man wears pink is P(Pink|Man) = 5/**40** = 0.125
- P(Man|Pink) = P(Man) P(Pink|Man) / P(Pink)
- P(Man|Pink) = 0.4 × 0.125 / **0.25** = 0.2

Random Variables

- A Random Variable is a set of **possible values** from a random experiment.
- Example: Tossing a coin: we could get Heads or Tails.
- Let's give them the values **Heads=0** and **Tails=1** and we have a Random Variable "X":

$$X = \{0, 1\}$$

- We have an **experiment** (such as tossing a coin)
- We give **values** to each event
- The **set of values** is a **Random Variable**

- Let E be the experiment of flipping a coin twice. We have seen that the sample space is S = {HH, HT, TH, TT}.
- Now define the random variable X = the number of heads. That is, for example, X(HH) = 2, while X(HT) = 1.
- We may make a table of the possibilities:

$\omega \in S$	HH	HT	TH	TT
$X(\omega) = x$	2	1	1	0

- Taking a look at the second row of the table, we see that the support of X – the set of all numbers that X assumes – would be $S_X = \{0, 1, 2\}$.

- Let E be the experiment of flipping a coin repeatedly until observing a Head.
- The sample space would be S = {H, TH, TTH, TTTH, . . .}. Now define the random variable Y = the number of Tails before the first head. Then the support of Y would be S_Y = {0, 1, 2, . . .}.

How to do it with R

- The primary function is the addrv function. There are two ways to use it.

Supply a Defining Formula

- Write a formula defining the random variable inside the function, and it will be added as a column to the data frame. As an example, let us roll a 4-sided die three times, and let us define the random variable $U = X1 - X2 + X3$

```
S <- rolldie(3, nsides = 4, makespace = TRUE)
S <- addrv(S, U = X1 - X2 + X3)
```

head(S)

Output:

	X1	X2	X3	U	probs
1	1	1	1	1	0.015625
2	2	1	1	2	0.015625
3	3	1	1	3	0.015625
4	4	1	1	4	0.015625
5	1	2	1	0	0.015625
6	2	2	1	1	0.015625

We see from the U column it is operating just like it should. We can now answer questions like

prob(S, U > 6)

Output: [1] 0.015625

Supply a Function

- The addrv function has an argument FUN. Its value should be a legitimate function from R, such as sum, mean, median, *etc.*

Or define own function

V = max(X1, X2, X3) and

W = X1 + X2 + X3.

```
S <- addrv(S, FUN = max, invars = c("X1", "X2", "X3"), name = "V")
S <- addrv(S, FUN = sum, invars = c("X1", "X2", "X3"), name = "W")
head(S)
  X1 X2 X3 U V W probs
1 1 1 1 1 1 3 0.015625
2 2 1 1 2 2 4 0.015625
3 3 1 1 3 3 5 0.015625
4 4 1 1 4 4 6 0.015625
5 1 2 1 0 2 4 0.015625
6 2 2 1 1 2 5 0.015625
```

- **Marginal Distributions**
- We can use the marginal function to aggregate the rows of the sample space by values of V, all the while accumulating the probability associated with V's distinct values.

  marginal(S, vars = "V")

Output:

```
  V probs
1 1 0.015625
2 2 0.109375
3 3 0.296875
4 4 0.578125
```

suppose we would like to examine the joint distribution of V and W.

marginal(S, vars = c("V", "W"))

Output:

	V	W	probs
1	1	3	0.015625
2	2	4	0.046875
3	2	5	0.046875
4	3	5	0.046875
5	2	6	0.015625

Discrete Distribution

- A **discrete distribution** describes the probability of occurrence of each value of a discrete random variable.
- A **discrete random variable** has a countable number of possible values.
- It take values in a finite or countable infinite support set.
- The probability of each value of a discrete random variable is between 0 and 1, and the sum of all the probabilities is equal to 1.

Discrete Random Variables

Probability Mass Functions

- Discrete random variables are characterized by their supports which take the form

$$S_X = \{u_1, u_2, \ldots, u_k\} \text{ or } S_X = \{u_1, u_2, u_3 \ldots\}.$$

- Every discrete random variable X has associated with it a probability mass function (PMF) $f_X : S_X \to [0, 1]$ defined by

$$f_X(x) = \mathbb{P}(X = x), \quad x \in S_X.$$

- All PMFs satisfy

1. $f_X(x) > 0$ for $x \in S$,
2. $\sum_{x \in S} f_X(x) = 1$, and
3. $\mathbb{P}(X \in A) = \sum_{x \in A} f_X(x)$, for any event $A \subset S$.

Example

- **Toss a coin 3 times**. The sample space would be

  S = {HHH, HTH, THH, TTH, HHT, HTT, THT, TTT}
- Now let X be the number of Heads observed. Then X has support

  $S_X = \{0, 1, 2, 3\}$.
- Assuming that the coin is fair and was tossed in exactly the same way each time, it is not unreasonable to suppose that the outcomes in the sample space are all equally likely.

What is the PMF of X?

- Notice that X is zero exactly when the outcome TTT occurs, and this event has probability 1/8. Therefore, $f_X(0) = 1/8$, and the same reasoning shows that $f_X(3) = 1/8$.
- Exactly three outcomes result in X = 1, thus, $f_X(1) = 3/8$ and $f_X(2)$ holds the remaining 3/8 probability (the total is 1).

- We can represent the PMF with a table:

$x \in S_X$	0	1	2	3	Total
$f_X(x) = \mathbb{P}(X = x)$	1/8	3/8	3/8	1/8	1

Mean, Variance, and Standard Deviation

- There are numbers associated with PMFs.
- One important example is the **mean μ**, also known as E X:

$$\mu = \mathbb{E}\, X = \sum_{x \in S} x f_X(x),$$

provided the (potentially infinite) series $\sum |x| f_X(x)$ is convergent.

- Another important number is the **variance**:

$$\sigma^2 = \mathbb{E}(X - \mu)^2 = \sum_{x \in S} (x - \mu)^2 f_X(x),$$

- which can be computed with the alternate formula

$$\sigma^2 = \mathbb{E}\, X^2 - (\mathbb{E}\, X)^2.$$

- Directly defined from the variance is the **standard deviation.**

$$\sigma = \sqrt{\sigma^2}.$$

Example

- Toss a coin 3 times. We can represent the PMF with a table

$x \in S_X$	0	1	2	3	Total
$f_X(x) = \mathbb{P}(X = x)$	1/8	3/8	3/8	1/8	1

- We will calculate the mean of X

$$\mu = \sum_{x=0}^{3} x f_X(x) = 0 \cdot \frac{1}{8} + 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} =$$

- $\mu = 1.5$

Related to the probability mass function $f_X(x) = \mathbb{P}(X = x)$ is another important function called the cumulative distribution function (CDF), F_X. It is defined by the formula

$$F_X(t) = \mathbb{P}(X \leq t), \quad -\infty < t < \infty. \tag{5.1.5}$$

We know that all PMFs satisfy certain properties, and a similar statement may be made for CDFs. In particular, any CDF F_X satisfies

- F_X is nondecreasing ($t_1 \leq t_2$ implies $F_X(t_1) \leq F_X(t_2)$).
- F_X is right-continuous ($\lim_{t \to a^+} F_X(t) = F_X(a)$ for all $a \in \mathbb{R}$).
- $\lim_{t \to -\infty} F_X(t) = 0$ and $\lim_{t \to \infty} F_X(t) = 1$.

We say that X has the distribution F_X and we write $X \sim F_X$. In an abuse of notation we will also write $X \sim f_X$ and for the named distributions the PMF or CDF will be identified by the family name instead of the defining formula.

- $F(x) = P(X \leq x) = \Sigma_{x_i \leq x} f(x_i)$

$P(X<=1) = P(X=0)+P(X=1)$

How to do it with R

- The mean and variance of a discrete random variable is easy to compute.
- We will start by defining a vector x containing the support of X.
- A vector f to contain the values of f_X at the respective outcomes in x:

```
x <- c(0,1,2,3)
f <- c(1/8, 3/8, 3/8, 1/8)
mu <- sum(x * f)
mu
```

Output:

- [1] 1.5

- To compute the variance σ^2, we subtract the value of mu from each entry in x, square the answers, multiply by f, and sum.

```
sigma2 <- sum((x-mu)^2 * f)
sigma2
```

Output:

```
[1] 0.75
```

The standard deviation σ is simply the square root σ^2.

```
sigma <- sqrt(sigma2)
sigma
```

Output:

```
[1] 0.8660254
```

- Finally, we may find the values of the CDF F_X on the support by accumulating the probabilities in f_X with the cumsum function.

```
F = cumsum(f)
F
```

Output:

```
[1] 0.125 0.500 0.875 1.000
```

- We can also do this with the distrEx package .
- We define a random variable X as an object, then compute things from the object such as mean, variance, and standard deviation with the functions E, var, and sd:

```
library(distrEx)
X <- DiscreteDistribution(supp = 0:3, prob = c(1,3,3,1)/8)
E(X)
[1] 1.5
var(X)
[1] 0.75
sd(X)
[1] 0.8660254
```

Continuous Distributions

- A continuous distribution describes the probabilities of the possible values of a continuous random variable.
- A continuous random variable is a random variable with a set of possible values (known as the range) that is infinite and uncountable.

Continuous Random Variables

- Continuous random variables have supports that look like

$$S_X = [a, b] \text{ or } (a, b)$$

  or unions of intervals of the above form.

- **Examples** of random variables that are often taken to

be continuous are:

- the height or weight of an individual
- other physical measurements such as the length or size of an object
- durations of time

- Every continuous random variable X has a probability density function (PDF) denoted f_X associated with it that satisfies three basic properties:

1. $f_X(x) > 0$ for $x \in S_X$,
2. $\int_{x \in S_X} f_X(x)\,dx = 1$, and
3. $\mathbb{P}(X \in A) = \int_{x \in A} f_X(x)\,dx$, for an event $A \subset S_X$.

- CDF has a relatively convenient form:

$$F_X(t) = \mathbb{P}(X \le t) = \int_{-\infty}^{t} f_X(x)\,dx, \quad -\infty < t < \infty.$$

- For any continuous CDF F_X the following are true.

  - F_X is nondecreasing , that is, $t_1 \le t_2$ implies $F_X(t_1) \le F_X(t_2)$.
  - F_X is continuous (see Appendix E.2). Note the distinction from the discrete case: CDFs of discrete random variables are not continuous, they are only right continuous.
  - $\lim_{t\to-\infty} F_X(t) = 0$ and $\lim_{t\to\infty} F_X(t) = 1$.

- There is a handy relationship between the CDF and PDF in the continuous case. Consider the derivative of F_X:

$$F'_X(t) = \frac{\mathrm{d}}{\mathrm{d}t} F_X(t) = \frac{\mathrm{d}}{\mathrm{d}t} \int_{-\infty}^{t} f_X(x)\,\mathrm{d}x = f_X(t),$$

Expectation of Continuous Random Variables

For a continuous random variable X the expected value of $g(X)$ is

$$\mathbb{E}\, g(X) = \int_{x \in S} g(x) f_X(x)\, \mathrm{d}x,$$

provided the (potentially improper) integral $\int_S |g(x)|\, f(x)\mathrm{d}x$ is convergent. One important example is the mean μ, also known as $\mathbb{E}\, X$:

$$\mu = \mathbb{E}\, X = \int_{x \in S} x f_X(x)\, \mathrm{d}x,$$

provided $\int_S |x| f(x)\mathrm{d}x$ is finite. Also there is the variance

$$\sigma^2 = \mathbb{E}(X - \mu)^2 = \int_{x \in S} (x - \mu)^2 f_X(x)\, \mathrm{d}x,$$

which can be computed with the alternate formula $\sigma^2 = \mathbb{E}\, X^2 - (\mathbb{E}\, X)^2$. In addition, there is the standard deviation $\sigma = \sqrt{\sigma^2}$. The moment generating function is given by

$$M_X(t) = \mathbb{E}\ \mathrm{e}^{tX} = \int_{-\infty}^{\infty} \mathrm{e}^{tx} f_X(x)\, \mathrm{d}x,$$

provided the integral exists (is finite) for all t in a neighborhood of $t = 0$.

Example

- Let the continuous random variable X have PDF

$$f_X(x) = 3x^2, \quad 0 \le x \le 1.$$

We will see later that f_X belongs to the *Beta* family of distributions. It is easy to see that $\int_{-\infty}^{\infty} f(x)\mathrm{d}x = 1$.

$$\begin{aligned}\int_{-\infty}^{\infty} f_X(x)\mathrm{d}x &= \int_0^1 3x^2\,\mathrm{d}x \\ &= x^3\big|_{x=0}^{1} \\ &= 1^3 - 0^3 \\ &= 1.\end{aligned}$$

This being said, we may find $\mathbb{P}(0.14 \le X < 0.71)$.

$$\begin{aligned}\mathbb{P}(0.14 \le X < 0.71) &= \int_{0.14}^{0.71} 3x^2\mathrm{d}x, \\ &= x^3\big|_{x=0.14}^{0.71} \\ &= 0.71^3 - 0.14^3 \\ &\approx 0.355167.\end{aligned}$$

We can find the mean and variance in an identical manner.

$$\begin{aligned}\mu = \int_{-\infty}^{\infty} x f_X(x)\mathrm{d}x &= \int_0^1 x \cdot 3x^2 \,\mathrm{d}x, \\ &= \frac{3}{4}x^4\Big|_{x=0}^1, \\ &= \frac{3}{4}.\end{aligned}$$

It would perhaps be best to calculate the variance with the shortcut formula $\sigma^2 = \mathbb{E}\,X^2 - \mu^2$:

$$\begin{aligned}\mathbb{E}\,X^2 = \int_{-\infty}^{\infty} x^2 f_X(x)\mathrm{d}x &= \int_0^1 x^2 \cdot 3x^2 \,\mathrm{d}x \\ &= \frac{3}{5}x^5\Big|_{x=0}^1 \\ &= 3/5.\end{aligned}$$

which gives $\sigma^2 = 3/5 - (3/4)^2 = 3/80$.

How to do in R

Let X have PDF $f(x) = 3x^2, 0 < x < 1$ and find $\mathbb{P}(0.14 \leq X \leq 0.71)$.

```
f <- function(x) 3 * x^2
integrate(f, lower = 0.14, upper = 0.71)
```

Output:

0.355167 with absolute error < 3.9e-15

- We could integrate the function x f (x) = 3*x^3 from zero to one to get the mean

```
f <- function(x) 3 * x^3
integrate(f, lower = 0, upper = 1)
```

Output:

0.75 with absolute error < 8.3e-15

- Let us redo Example with the distr package. We define an absolutely continuous random variable:

```
library(distr)
f <- function(x) 3 * x^2
X <- AbscontDistribution(d = f, low1 = 0, up1 = 1)
p(X)(0.71) - p(X)(0.14)
```

Output:

```
[1] 0.355167
```

```
    library(distrEx)
    E(X)
[1] 0.7496337
    var(X)
[1] 0.03768305
    3/80
[1] 0.0375
```

Example

- We will try one with unbounded support to brush up on improper integration.
- Let the random variable *X have PDF*

$$f_X(x) = \frac{3}{x^4}, \quad x > 1.$$

We can show that $\int_{-\infty}^{\infty} f(x)\mathrm{d}x = 1$:

$$\begin{aligned}
\int_{-\infty}^{\infty} f_X(x)\mathrm{d}x &= \int_1^{\infty} \frac{3}{x^4}\,\mathrm{d}x \\
&= \lim_{t\to\infty} \int_1^t \frac{3}{x^4}\,\mathrm{d}x \\
&= \lim_{t\to\infty} \left. 3\frac{1}{-3}x^{-3}\right|_{x=1}^{t} \\
&= -\left(\lim_{t\to\infty} \frac{1}{t^3} - 1\right) \\
&= 1.
\end{aligned}$$

We calculate $\mathbb{P}(3.4 \leq X < 7.1)$:

$$\begin{aligned}
\mathbb{P}(3.4 \leq X < 7.1) &= \int_{3.4}^{7.1} 3x^{-4}\mathrm{d}x \\
&= 3\,\frac{1}{-3}x^{-3}\Big|_{x=3.4}^{7.1} \\
&= -1(7.1^{-3} - 3.4^{-3}) \\
&\approx 0.0226487123.
\end{aligned}$$

We locate the mean and variance just like before.

$$\begin{aligned}
\mu = \int_{-\infty}^{\infty} x f_X(x)\mathrm{d}x &= \int_1^{\infty} x \cdot \frac{3}{x^4}\,\mathrm{d}x \\
&= 3\,\frac{1}{-2}x^{-2}\Big|_{x=1}^{\infty} \\
&= -\frac{3}{2}\left(\lim_{t\to\infty}\frac{1}{t^2} - 1\right) \\
&= \frac{3}{2}.
\end{aligned}$$

Again we use the shortcut $\sigma^2 = \mathbb{E}X^2 - \mu^2$:

$$\begin{aligned}
\mathbb{E}X^2 = \int_{-\infty}^{\infty} x^2 f_X(x)\mathrm{d}x &= \int_1^{\infty} x^2 \cdot \frac{3}{x^4}\,\mathrm{d}x \\
&= 3\,\frac{1}{-1}x^{-1}\Big|_{x=1}^{\infty} \\
&= -3\left(\lim_{t\to\infty}\frac{1}{t^2} - 1\right) \\
&= 3,
\end{aligned}$$

which closes the example with $\sigma^2 = 3 - (3/2)^2 = 3/4$.

Let X have PDF $f(x) = 3/x^4$, $x > 1$. We may integrate the function $xf(x) =$ 3/x^3 from zero to infinity to get the mean of X.

```
g <- function(x) 3/x^3
integrate(g, lower = 1, upper = Inf)
```

Output:

1.5 with absolute error < 1.7e-14

Multivariate Distributions

- We have seen distributions, discrete and continuous. They were all univariate, however, meaning that we only considered one random variable at a time.
- We can imagine many random variables associated with a single person:
- their height, their weight, their wrist circumference (all continuous), or their eye/hair color, shoe size, whether they are right handed, left handed, or ambidextrous (all categorical), and we can even surmise reasonable probability distributions to associate with each of these variables.
- But there is a difference: for a single person, these variables are related. For instance, a person's height betrays a lot of information about that person's weight. The concept we are hinting at is the notion of ***dependence between random variables***.

Joint and Marginal Probability Distributions

- Consider two discrete random variables X and Y with PMFs f_X and f_Y that are supported on the sample spaces S_X and S_Y, respectively.
- Let $S_{X,Y}$ denote the set of all possible observed pairs (x, y), called the joint support set of X and Y. Then the joint probability mass function of X and Y is the function $f_{X,Y}$ defined by

$$f_{X,Y}(x, y) = \mathbb{P}(X = x, Y = y), \quad \text{for } (x, y) \in S_{X,Y}.$$

Every joint PMF satisfies

$$f_{X,Y}(x, y) > 0 \text{ for all } (x, y) \in S_{X,Y},$$

and

$$\sum_{(x,y)\in S_{X,Y}} f_{X,Y}(x, y) = 1.$$

- PMFs f_X and f_Y are called the marginal PMFs of X and Y, respectively.
- If we are given only the joint PMF then we may recover each of the marginal PMFs by using the Theorem of Total Probability

$$\begin{aligned} f_X(x) &= \mathbb{P}(X = x), \\ &= \sum_{y \in S_Y} \mathbb{P}(X = x,\ Y = y), \\ &= \sum_{y \in S_Y} f_{X,Y}(x, y). \end{aligned}$$

$$f_Y(y) = \sum_{x \in S_Y} f_{X,Y}(x, y).$$

- Associated with the joint PMF is the joint cumulative distribution function $F_{X,Y}$ defined by

$$F_{X,Y}(x, y) = \mathbb{P}(X \leq x,\ Y \leq y), \quad \text{for } (x, y) \in \mathbb{R}^2.$$

Example

- Roll a fair die twice. Let X be the face shown on the first roll, and let Y be the face shown on the second roll.
- For this example, it suffices to define

$$f_{X,Y}(x,y) = \frac{1}{36}, \quad x = 1,\ldots,6,\ y = 1,\ldots,6.$$

The marginal PMFs are given by $f_X(x) = 1/6$, $x = 1, 2, \ldots, 6$, and $f_Y(y) = 1/6$, $y = 1, 2, \ldots, 6$, since

$$f_X(x) = \sum_{y=1}^{6} \frac{1}{36} = \frac{1}{6}, \quad x = 1,\ldots,6,$$

and the same computation with the letters switched works for Y.

Example

- Let the random experiment again be to roll a fair die twice, except now let us define the random variables U and V by

  U = the maximum of the two rolls, and

  V = the sum of the two rolls.
- We see that the support of U is $S_U = \{1, 2, \ldots, 6\}$ and the support of V is $S_V = \{2, 3, \ldots, 12\}$.
- We may represent the sample space with a matrix, and for each entry in the matrix we mav calculate the value that U assumes.

U	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	2	3	4	5	6
3	3	3	3	4	5	6
4	4	4	4	4	5	6
5	5	5	5	5	5	6
6	6	6	6	6	6	6

(a) $U = \max(X, Y)$

V	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

(b) $V = X + Y$

- We can use the table to calculate the marginal PMF of U, we know that each entry in the matrix has probability 1/36 associated with it.
- For instance, there is only one outcome in the matrix with U = 1, namely, the top left corner. This single entry has probability 1/36, therefore, it must be that $f_U(1) = IP(U = 1) = 1/36$.
- Similarly we see that there are three entries in the matrix with U = 2, thus $f_U(2) = 3/36$. Continuing in this manner

  we will find the marginal distribution of U may be written

$$f_U(u) = \frac{2u-1}{36}, \quad u = 1, 2, \ldots, 6.$$

- We may do a similar thing for V.
- Collecting all of the probability we will find that the marginal PMF of V is

$$f_V(v) = \frac{6 - |v - 7|}{36}, \quad v = 2, 3, \ldots, 12.$$

- We may collapse the two matrices into one, big matrix of pairs of values (*u*, *v)*.

(U, V)	1	2	3	4	5	6
1	(1,2)	(2,3)	(3,4)	(4,5)	(5,6)	(6,7)
2	(2,3)	(2,4)	(3,5)	(4,6)	(5,7)	(6,8)
3	(3,4)	(3,5)	(3,6)	(4,7)	(5,8)	(6,9)
4	(4,5)	(4,6)	(4,7)	(4,8)	(5,9)	(6,10)
5	(5,6)	(5,7)	(5,8)	(5,9)	(5,10)	(6,11)
6	(6,7)	(6,8)	(6,9)	(6,10)	(6,11)	(6,12)

Table 7.2: Joint values of $U = \max(X, Y)$ and $V = X + Y$

- Again, each of these pairs has probability 1/36 associated with it and we are looking at the joint PDF of (U, V).
- Many of the pairs are repeated, but some of them are not: (1, 2) appears twice, but (2, 3) appears only once.

	2	3	4	5	6	7	8	9	10	11	12	Total
1	1/36											1/36
2		2/36	1/36									3/36
3			2/36	2/36	1/36							5/36
4				2/36	2/36	2/36	1/36					7/36
5					2/36	2/36	2/36	2/36	1/36			9/36
6						2/36	2/36	2/36	2/36	2/36	1/36	11/36
Total	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36	1

Table 7.3: The joint PMF of (U, V)

How to do it with R

- First we set up the sample space with the rolldie function.
- Next, we add random variables U and V with the addrv function. We take a look at the very top of the data frame (probability space) to make sure that everything is operating according to plan.

```
S <- rolldie(2, makespace = TRUE)
S <- addrv(S, FUN = max, invars = c("X1", "X2"), name = "U")
S <- addrv(S, FUN = sum, invars = c("X1", "X2"), name = "V")
head(S)
```

Output:

	X1	X2	U	V	probs
1	1	1	1	2	0.02777778
2	2	1	2	3	0.02777778
3	3	1	3	4	0.02777778
4	4	1	4	5	0.02777778
5	5	1	5	6	0.02777778
6	6	1	6	7	0.02777778

- The U and V columns have been added to the data frame and have been computed correctly. This result would be fine as it is, but the data frame has too many rows: there are repeated pairs (u, v) which show up as repeated rows in the data frame.
- The goal is to aggregate the rows of S such that the result has exactly one row for each unique pair (u, v) with positive probability. This sort of thing is exactly the task for which the marginal function was designed.

```
UV <- marginal(S, vars = c("U", "V"))
head(UV)
```

```
  U V      probs
1 1 2 0.02777778
2 2 3 0.05555556
3 2 4 0.02777778
4 3 4 0.05555556
5 3 5 0.05555556
6 4 5 0.05555556
```

- The data frame is difficult to understand. It would be better to have a tabular display .We can do that with the xtabs function.

  xtabs(round(probs, 3) ~ U + V, data = UV)

U	V 2	3	4	5	6	7	8	9	10	11	12
1	0.028	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	0.000	0.056	0.028	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
3	0.000	0.000	0.056	0.056	0.028	0.000	0.000	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.056	0.056	0.056	0.028	0.000	0.000	0.000	0.000
5	0.000	0.000	0.000	0.000	0.056	0.056	0.056	0.056	0.028	0.000	0.000
6	0.000	0.000	0.000	0.000	0.000	0.056	0.056	0.056	0.056	0.056	0.028

- We can repeat the process with marginal to get the univariate marginal distributions of U and V separately.
- marginal(UV, vars = "U")

	U	probs
1	1	0.02777778
2	2	0.08333333
3	3	0.13888889
4	4	0.19444444
5	5	0.25000000
6	6	0.30555556

- head(marginal(UV, vars = "V"))

	V	probs
1	2	0.02777778
2	3	0.05555556
3	4	0.08333333
4	5	0.11111111
5	6	0.13888889
6	7	0.16666667

Continuing the reasoning for the discrete case, given two continuous random variables X and Y there similarly exists[2] a function $f_{X,Y}(x, y)$ associated with X and Y called the *joint probability density function* of X and Y. Every joint PDF satisfies

$$f_{X,Y}(x, y) \geq 0 \text{ for all } (x, y) \in S_{X,Y},$$

and

$$\iint_{S_{X,Y}} f_{X,Y}(x, y)\, \mathrm{d}x\, \mathrm{d}y = 1.$$

In the continuous case there is not such a simple interpretation for the joint PDF; however, we do have one for the joint CDF, namely,

$$F_{X,Y}(x, y) = \mathbb{P}(X \leq x,\ Y \leq y) = \int_{-\infty}^{x} \int_{-\infty}^{y} f_{X,Y}(u, v)\, \mathrm{d}v\, \mathrm{d}u,$$

for $(x, y) \in \mathbb{R}^2$. If X and Y have the joint PDF $f_{X,Y}$, then the marginal density of X may be recovered by

$$f_X(x) = \int_{S_Y} f_{X,Y}(x, y)\, \mathrm{d}y, \quad x \in S_X$$

and the marginal PDF of Y may be found with

$$f_Y(y) = \int_{S_X} f_{X,Y}(x, y)\, \mathrm{d}x, \quad y \in S_Y.$$

Example

- Let the joint PDF of (X, Y) be given by

$$f_{X,Y}(x,y) = \frac{6}{5}\left(x+y^2\right), \quad 0<x<1,\ 0<y<1.$$

The marginal PDF of X is

$$\begin{aligned} f_X(x) &= \int_0^1 \frac{6}{5}\left(x+y^2\right)\,dy, \\ &= \frac{6}{5}\left(xy+\frac{y^3}{3}\right)\Big|_{y=0}^{1}, \\ &= \frac{6}{5}\left(x+\frac{1}{3}\right), \end{aligned}$$

for $0<x<1$, and the marginal PDF of Y is

$$\begin{aligned} f_Y(y) &= \int_0^1 \frac{6}{5}\left(x+y^2\right)\,dx, \\ &= \frac{6}{5}\left(\frac{x^2}{2}+xy^2\right)\Big|_{x=0}^{1}, \\ &= \frac{6}{5}\left(\frac{1}{2}+y^2\right), \end{aligned}$$

UNIT-IV

STATISTICS

UNIT-IV

- Regression-Linear-Multiple-Logistic-Poisson-Analysis of Covariance-Time Series Analysis-Nonlinear Least Square-Decision Tree-Random Forest-Survival Analysis-t-Test-Chi Square Test,ANOVA.

Regression

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

Linear Regression

- In Linear Regression these two variables are related through an equation, where exponent (power) of both these variables is 1.
- Mathematically a linear relationship represents a straight line when plotted as a graph.
- The general mathematical equation for a linear regression is

$$y = ax + b$$

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

The Linear Regression Equation

- Linear regression is a way to model the relationship between two variables.
- You might also recognize the equation as the **slope formula**.
- The equation has the form Y= a + bX, where Y is the dependent variable (that's the variable that goes on the Y axis), X is the independent variable (i.e. it is plotted on the X axis), b is the slope of the line and a is the y-intercept.

$$a = \frac{(\sum y)(\sum x^2) - (\sum x)(\sum xy)}{n(\sum x^2) - (\sum x)^2}$$

$$b = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

- Step 1

SUBJECT	AGE X	GLUCOSE LEVEL Y	XY	X^2	Y^2
1	43	99	4257	1849	9801
2	21	65	1365	441	4225
3	25	79	1975	625	6241
4	42	75	3150	1764	5625
5	57	87	4959	3249	7569
6	59	81	4779	3481	6561
Σ	247	486	20485	11409	40022

- **Step 2:** Use the following equations to find a and b

$$a = \frac{(\Sigma y)(\Sigma x^2) - (\Sigma x)(\Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2}$$

$$b = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2}$$

a = **65.1416**

b = **.385225**

- **Step 3:** Insert the values into the equation.
  y = a + bx
  y = 65.14 + .385225x
- **Step 4 :** Predict the value of y when new x value is given

  If x=20 then y= 72.84607

Steps to Establish a Regression in R

The **steps to create the relationship** is

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

lm() Function

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

Syntax

lm(formula,data)

Description of the parameters

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

Create Relationship Model & get the Coefficients

```
x <- c(43,21,25,42,57,59)
y <- c(99,65,79,75,87,81)
relation <- lm(y~x)
relation
```

Output:

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
    65.1416       0.3852
```

Get the Summary of the Relationship

summary(relation)

```
Call:
lm(formula = y ~ x)

Residuals:
      1       2       3       4       5       6
17.2938 -8.2313  4.2278 -6.3210 -0.0994 -6.8698

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  65.1416    13.4453   4.845  0.00837 **
x             0.3852     0.3083   1.249  0.27964
```

predict() Function

Syntax

predict(object, newdata)

Description of the parameters

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

predict() Function

```
a<-data.frame(x=20)
result<-predict(relation,a)
result
```

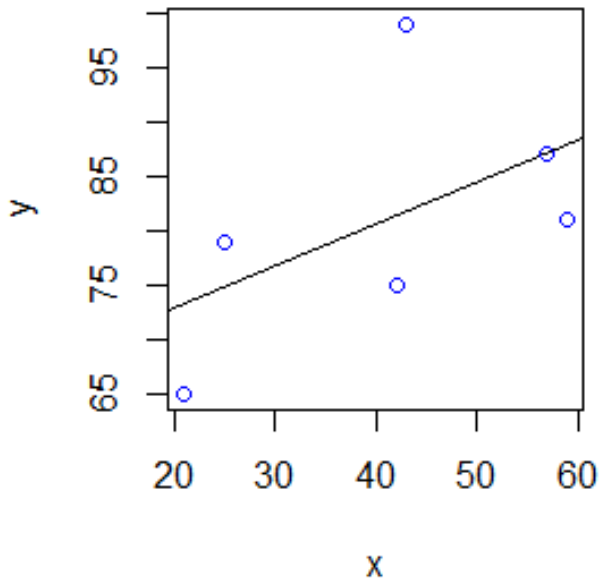
Output:

```
       1
72.84607
```

Visualize the Regression Graphically

```
x <- c(43,21,25,42,57,59)
y <- c(99,65,79,75,87,81)
plot(x,y,abline(lm(y~x)),col="blue")
```

Output:



Example 2

- Below is the sample data representing the observations

- # Values of height

151, 174, 138, 186, 128, 136, 179, 163, 152, 131

- # Values of weight.

63, 81, 56, 91, 47, 57, 76, 72, 62, 48

Create Relationship Model & get the Coefficients

```
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.
relation <- lm(y~x)
print(relation)
```

Output:

```
Call:
lm(formula = y ~ x)
Coefficients:
(Intercept)     x
-38.4551    0.6746
```

Get the Summary of the Relationship

summary(relation)

Output:

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-6.3002 -1.6629  0.0412  1.8944  3.9775

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -38.45509    8.04901  -4.778  0.00139 **
x             0.67461    0.05191  12.997 1.16e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.253 on 8 degrees of freedom
Multiple R-squared:  0.9548,    Adjusted R-squared:  0.9491
F-statistic: 168.9 on 1 and 8 DF,  p-value: 1.164e-06
```

Predict the weight of new persons

```
# Find weight of a person with height 170.
a <- data.frame(x = 170)
result <-  predict(relation,a)
Result
```

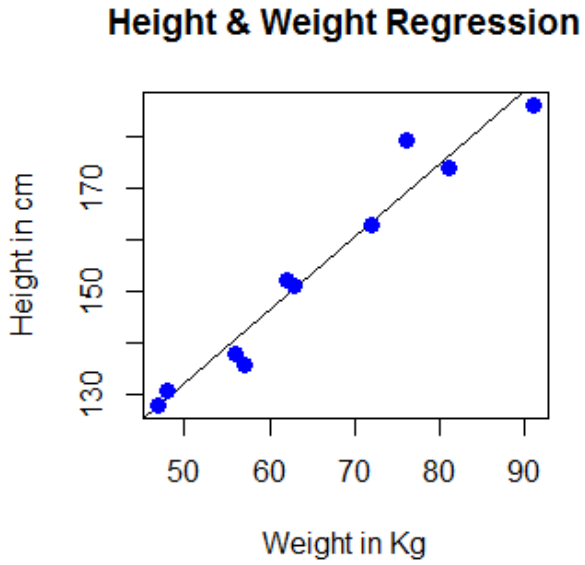
Output:

```
       1
76.22869
```

Visualize the Regression Graphically

```
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
relation <- lm(y~x)
plot(y,x,col = "blue",main = "Height & Weight Regression",
    abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in
    Kg",ylab = "Height in cm")
```

Output:



Multiple Regression

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

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

Description of the parameters used

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

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

lm() Function

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

Syntax

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

Description of the parameters

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

Input Data

- Consider the data set "mtcars" available in the R environment.
- It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.
- The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables.

```
input <- mtcars[,c("mpg","disp","hp","wt")]
head(input)
```

Output:

```
                   mpg disp  hp    wt
Mazda RX4         21.0  160 110 2.620
Mazda RX4 Wag     21.0  160 110 2.875
Datsun 710        22.8  108  93 2.320
Hornet 4 Drive    21.4  258 110 3.215
Hornet Sportabout 18.7  360 175 3.440
Valiant           18.1  225 105 3.460
```

Create Relationship Model & get the Coefficients

```
input <- mtcars[,c("mpg","disp","hp","wt")]
model <- lm(mpg~disp+hp+wt, data = input)
model

a <- coef(model)[1]
Xdisp <- coef(model)[2]
Xhp <- coef(model)[3]
Xwt <- coef(model)[4]
```

Output

```
Call:
lm(formula = mpg ~ disp + hp + wt, data = input)

Coefficients:
(Intercept)         disp           hp           wt
  37.105505    -0.000937    -0.031157    -3.800891

> a <- coef(model)[1]
> a
(Intercept)
   37.10551
> Xdisp <- coef(model)[2]
> Xhp <- coef(model)[3]
> Xwt <- coef(model)[4]
>
> Xdisp
        disp
-0.0009370091
> Xhp
         hp
-0.03115655
> Xwt
       wt
-3.800891
```

Create Equation for Regression Model

- Based on the above intercept and coefficient values, we create the mathematical equation

Y = a+Xdisp.x1+Xhp.x2+Xwt.x3

Y = 37.15+(-0.000937)*x1+(-0.0311)*x2+(-3.8008)*x3.

Predicting New Values

- We can use the regression equation created above to predict the mileage when a new set of values for displacement, horse power and weight is provided.
- For a car with disp = 221, hp = 102 and wt = 2.91 the predicted mileage is −

  Y = 37.15+(-0.000937)*221+(-0.0311)*102+(-3.8008)*2.91

  Y = 22.7104

R Code

```
p<-data.frame(disp = 221, hp = 102, wt = 2.91 )
predict(model,p)
```

Output:

```
       1
22.65987
```

The Regression Equation in Matrix Form

With multiple regression, there is one dependent variable and *k* dependent variables. The regression equation is:

$$\hat{y} = b_0 + b_1x_1 + b_2x_2 + \ldots + b_{k-1}x_{k-1} + b_kx_k$$

where $\hat{y}$ is the predicted value of the dependent variable, b_k are regression coefficients, and x_k is the value of independent variable *k*.

To express the regression equation in matrix form, we need to define three matrices: **Y**, **b**, and **X**.

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ . \\ . \\ . \\ b_k \end{bmatrix}$$

$$\mathbf{X} = \begin{bmatrix} 1 & X_{1,1} & X_{1,2} & . & . & . & X_{1,k} \\ 1 & X_{2,1} & X_{2,2} & . & . & . & X_{2,k} \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ 1 & X_{n,1} & X_{n,2} & . & . & . & X_{n,k} \end{bmatrix}$$

- Given these matrices, the multiple regression equation can be expressed concisely as:

  $$\mathbf{Y} = \mathbf{Xb}$$

- This simple expression describes the regression equation for 1, 2, 3, or *any* number of independent variables.

Normal Equations in Matrix Form

- Just as the regression equation can be expressed compactly in matrix form, so can the normal equations. The least squares normal equations can be expressed as:

  $$\mathbf{X'Y = X'Xb}$$

- Here, matrix **X'** is the transpose of matrix **X**.
- To solve for regression coefficients, simply pre-multiply by the inverse of **X'X**:

  $$\mathbf{(X'X)^{-1}X'Xb = (X'X)^{-1}X'Y}$$

  $$\mathbf{b = (X'X)^{-1}X'Y}$$

- where $\mathbf{(X'X)^{-1}X'X = I}$, the identity matrix.

- Consider the table below. It shows three performance measures for five students.

Student	Test score	IQ	Study hours
1	100	110	40
2	90	120	30
3	80	100	20
4	70	90	0
5	60	80	10

- Using multiple regression, develop a regression equation to predict test score, based on (1) IQ and (2) the number of hours that the student studied.

- For this problem, we have some raw data; and we want to use this raw data to define a regression equation:

$$y = b_0 + b_1x_1 + b_2x_2$$

- where y is the predicted test score; b_0, b_1, and b_2 are regression coefficients; x_1 is an IQ score; and x_2 is the number of hours that the student studied.
- On the right side of the equation, the only unknowns are the regression coefficients. To define the regression coefficients, we use the following equation:

$$\mathbf{b = (X'X)^{-1}X'Y}$$

- To solve this equation, we need to complete the following steps:
- Define **X**.
- Define **X'**.
- Compute **X'X**.
- Find the inverse of **X'X**.
- Define **Y**.

Let's begin with matrix **X**. Matrix **X** has a column of 1's plus two columns of values for each independent variable. So, this is matrix **X** and its transpose **X'**:

$$\mathbf{X} = \begin{bmatrix} 1 & 110 & 40 \\ 1 & 120 & 30 \\ 1 & 100 & 20 \\ 1 & 90 & 0 \\ 1 & 80 & 10 \end{bmatrix} \qquad \mathbf{X'} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 110 & 120 & 100 & 90 & 80 \\ 40 & 30 & 20 & 0 & 10 \end{bmatrix}$$

Given **X'** and **X**, it is a simple matter to compute **X'X**.

$$\mathbf{X'X} = \begin{bmatrix} 5 & 500 & 100 \\ 500 & 51{,}000 & 10{,}800 \\ 100 & 10{,}800 & 3{,}000 \end{bmatrix}$$

$$(\mathbf{X'X})^{-1} = \begin{bmatrix} 101/5 & -7/30 & 1/6 \\ -7/30 & 1/360 & -1/450 \\ 1/6 & -1/450 & 1/360 \end{bmatrix}$$

Next, we define **Y**, the vector of dependent variable scores. For this problem, it is the vector of test scores.

$$\mathbf{Y} = \begin{bmatrix} 100 \\ 90 \\ 80 \\ 70 \\ 60 \end{bmatrix}$$

With all of the essential matrices defined, we are ready to compute the least squares regression coefficients.

$$\mathbf{b} = (\mathbf{X'X})^{-1}\mathbf{X'Y}$$

$$\mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 20 \\ 0.5 \\ 0.5 \end{bmatrix}$$

TO do in R

```
dat <-
    data.frame(student=c(1,2,3,4,5)),test=c(100,90,80,70,60),IQ=c
    (110,120,100,90,80),hr=c(40,30,20,0,10))
dat
re <- lm(test~IQ+hr,data=dat)
re
```

Example 2

	youtube	facebook	newspaper	sales
1	276.1	45.4	83.0	26.5
2	53.4	47.2	54.1	12.5
3	20.6	55.1	83.2	11.2
4	181.8	49.6	70.2	22.2

Predict for the value

youtube=250.1,facebook=35.4, newspaper=81.0

```
dat <-
    data.frame(youtube=c(276.1,53.4,20.6,181.8),facebook=c(4
    5.4,47.2,55.1,49.6),
    newspaper=c(83.0,54.1,83.2,70.2),
    sales=c(26.5,12.5,11.2,22.2))
dat
re <- lm(sales~youtube+facebook+newspaper,data=dat)
re
summary(re)
p<-data.frame(youtube=c(250.1),facebook=c(35.4),
              newspaper=c(81.0),sales=c(26.5))
predict(re,p)
```

Logistic Regression

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

$$y = 1/(1+e^{-(a+b1x1+b2x2+b3x3+...)})$$

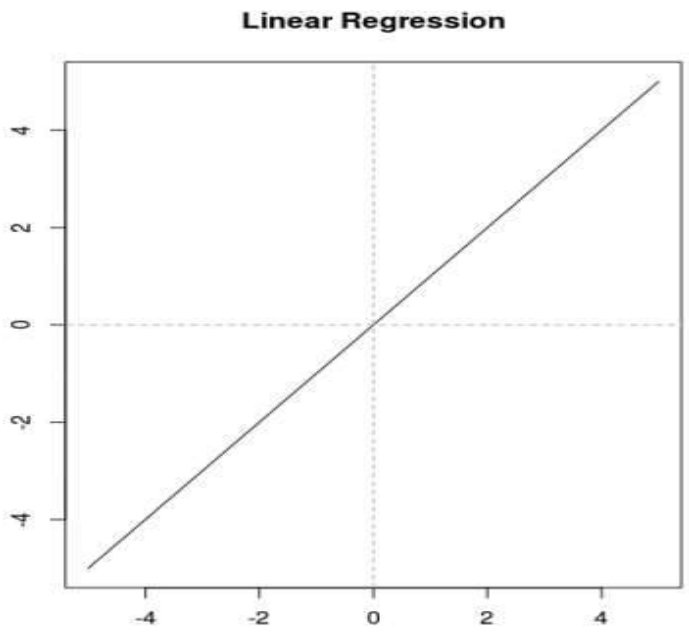
Description of the parameters

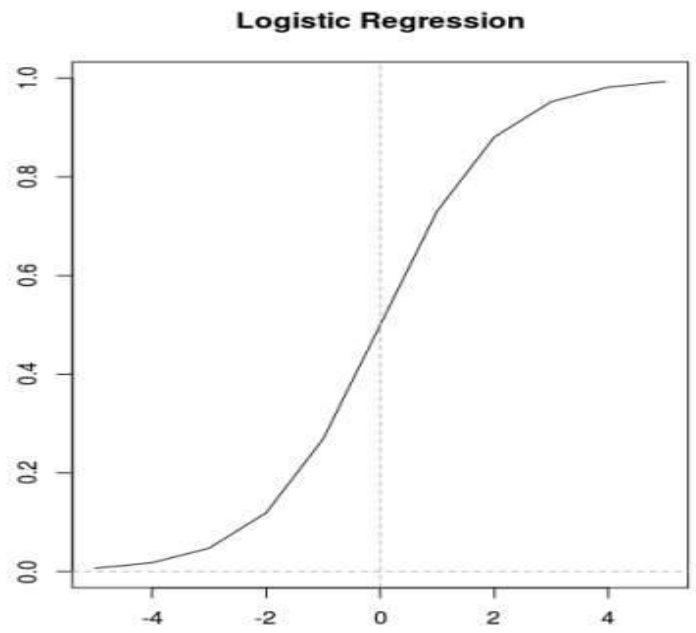
- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are the coefficients which are numeric constants.

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

If linear regression serves to predict continuous Y variables, logistic regression is used for binary classification.

Linear regression is not capable of predicting probability. If you use linear regression to model a binary response variable, for example, the resulting model may not restrict the predicted Y values within 0 and 1. Here's where logistic regression comes into play, where you get a probability score that reflects the probability of the occurrence at the event.





glm function

- **Syntax**

  glm(formula,data,family)

Following is the description of the parameters used −

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

Example

- The in-built data set "mtcars" describes different models of a car with their various engine specifications.
- In "mtcars" data set, the transmission mode (automatic or manual) is described by the column am which is a binary value (0 or 1).
- We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.

```
input <- mtcars[,c("am","cyl","hp","wt")]
head(input)
```

Output:

```
                  am cyl  hp    wt
Mazda RX4          1   6 110 2.620
Mazda RX4 Wag      1   6 110 2.875
Datsun 710         1   4  93 2.320
Hornet 4 Drive     0   6 110 3.215
Hornet Sportabout  0   8 175 3.440
Valiant            0   6 105 3.460
```

Create Regression Model

- We use the **glm()** function to create the regression model and get its summary for analysis.

  am.data = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)

```
Call:  glm(formula = am ~ cyl + hp + wt, family = binomial, data
= input)

Coefficients:
(Intercept)          cyl           hp           wt
   19.70288      0.48760      0.03259     -9.14947

Degrees of Freedom: 31 Total (i.e. Null);  28 Residual
Null Deviance:      43.23
Residual Deviance: 9.841        AIC: 17.84
```

summary(am.data)

```
Call:
glm(formula = am ~ cyl + hp + wt, family = binomial, data = input)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.17272  -0.14907  -0.01464   0.14116   1.27641

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 19.70288    8.11637   2.428   0.0152 *
cyl          0.48760    1.07162   0.455   0.6491
hp           0.03259    0.01886   1.728   0.0840 .
wt          -9.14947    4.15332  -2.203   0.0276 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Predict

```
p <- data.frame(cyl=4,hp=93,wt=2.315)
a <- predict(am.data,p,type="response")
result <- ifelse(a>=0.5,1,0)
result
```

Output

1

Example 2

```
install.packages("caTools")
library(caTools)

# Splitting dataset
split <- sample.split(mtcars, SplitRatio = 0.8)
train_reg <- subset(mtcars, split == "TRUE")
test_reg <- subset(mtcars, split == "FALSE")

# Training model
logistic_model <- glm(vs ~ wt + disp,  data = train_reg,  family =
    "binomial")
logistic_model

summary(logistic_model)
```

```
# Predict test data based on model
predict_reg <- predict(logistic_model,  test_reg, type = "response")
predict_reg

# Changing probabilities
predict_reg <- ifelse(predict_reg >0.5, 1, 0)

# Evaluating model accuracy
# using confusion matrix
table(test_reg$vs, predict_reg)

missing_classerr <- mean(predict_reg != test_reg$vs)
print(paste('Accuracy =', 1 - missing_classerr))
```

Poisson Regression

- Poisson Regression involves regression models in which the response variable is in the form of counts and not fractional numbers.
- For example, the count of number of births or number of wins in a football match series.
- Also the values of the response variables follow a Poisson distribution.
- The general mathematical equation for Poisson regression is −

$$\log(y) = a + b1x1 + b2x2 + bnxn$$

Description of the parameters

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

glm() function

- **Syntax**

  glm(formula,data,family)

Description of the parameters

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

Example

- We have the in-built data set "warpbreaks" which describes the effect of wool type (A or B) and tension (low, medium or high) on the number of warp breaks per loom. Let's consider "breaks" as the response variable which is a count of number of breaks. The wool "type" and "tension" are taken as predictor variables.

input <- warpbreaks

	breaks	wool	tension
1	26	A	L
2	30	A	L
3	54	A	L
4	25	A	L
5	70	A	L
6	52	A	L
7	51	A	L
8	26	A	L
9	67	A	L
10	18	A	M
11	21	A	M
12	29	A	M
13	17	A	M
14	12	A	M

Create Regression Model

```
output <-glm(formula = breaks ~ wool+tension, data = input,
             family = poisson)
summary(output)
```

```
Call:
glm(formula = breaks ~ wool + tension, family = poisson, data = input
)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.6871  -1.6503  -0.4269   1.1902   4.2616

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.69196    0.04541  81.302  < 2e-16 ***
woolB       -0.20599    0.05157  -3.994 6.49e-05 ***
tensionM    -0.32132    0.06027  -5.332 9.73e-08 ***
tensionH    -0.51849    0.06396  -8.107 5.21e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 297.37  on 53  degrees of freedom
Residual deviance: 210.39  on 50  degrees of freedom
AIC: 493.06

Number of Fisher Scoring iterations: 4
```

- In the summary we look for the p-value in the last column to be less than 0.05 to consider an impact of the predictor variable on the response variable.
- As seen the wooltype B having tension type M and H have impact on the count of breaks

- **Predict**

  newdata = data.frame(wool = "B", tension = "M")

  predict(output, newdata, type = "response")

Output:

1

23.68056

Analysis of Covariance (ANCOVA)

- We use Regression analysis to create models which describe the effect of variation in predictor variables on the response variable.
- Sometimes, if we have a categorical variable with values like Yes/No or Male/Female etc.
- The simple regression analysis gives multiple results for each value of the categorical variable.
- In such scenario, we can study the effect of the categorical variable by using it along with the predictor variable and comparing the regression lines for each level of the categorical variable.
- Such an analysis is termed as **Analysis of Covariance** also called as **ANCOVA**.

Analysis of covariance is used to test the main and interaction effects of categorical variables on a continuous dependent variable, controlling for the effects of selected other continuous variables, which co-vary with the dependent.

Example

```
input <- mtcars[,c("am","mpg","hp")]
head(input)
```

Output:

```
                  am  mpg  hp
Mazda RX4          1 21.0 110
Mazda RX4 Wag      1 21.0 110
Datsun 710         1 22.8  93
Hornet 4 Drive     0 21.4 110
Hornet Sportabout  0 18.7 175
Valiant            0 18.1 105
```

Model1: Interaction between hp and am

```
fit1 <- aov(mpg~hp*am,data = input)
summary(fit1)
```

```
             Df Sum Sq Mean Sq F value   Pr(>F)
hp            1  678.4   678.4  77.391 1.50e-09 ***
am            1  202.2   202.2  23.072 4.75e-05 ***
hp:am         1    0.0     0.0   0.001    0.981
Residuals    28  245.4     8.8
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

- Both horse power and transmission type has significant effect on miles per gallon as the p value in both cases is less than 0.05.
- Interaction between these two variables is not significant as the p-value is more than 0.05.

- **Model2:** No interaction between hp and am

```
fit2 <- aov(mpg~hp+am,data = input)
summary(fit2)
```

```
            Df Sum Sq Mean Sq F value   Pr(>F)
hp           1  678.4   678.4   80.15 7.63e-10 ***
am           1  202.2   202.2   23.89 3.46e-05 ***
Residuals   29  245.4     8.5
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

- both horse power and transmission type has significant effect on miles per gallon as the p value in both cases is less than 0.05.

Comparing Two Models

Objective: to conclude if the interaction of the variables is truly in-significant

anova(fit1,fit2)

Output:

```
Analysis of Variance Table

Model 1: mpg ~ hp * am
Model 2: mpg ~ hp + am
  Res.Df    RSS Df  Sum of Sq     F Pr(>F)
1     28 245.43
2     29 245.44 -1 -0.0052515 6e-04 0.9806
```

- As the p-value is greater than 0.05 we conclude that the interaction between hp and am is not significant.

- So mpg will depend in a similar manner on the horse power of the car in both auto and manual transmission mode.

Time Series Analysis

- Time series is a series of data points in which each data point is associated with a timestamp.
- A simple example is the price of a stock in the stock market at different points of time on a given day.
- Another example is the amount of rainfall in a region at different months of the year.
- R language uses many functions to create, manipulate and plot the time series data. The data for the time series is stored in an R object called **time-series object**. It is also a R data object like a vector or data frame.

The time series object is created by using the **ts()** function.

Syntax

timeseries.object.name <- ts(data, start, end, frequency)

Description of the parameters

data is a vector or matrix containing the values used in the time series.

start specifies the start time for the first observation in time series.

end specifies the end time for the last observation in time series.

frequency specifies the number of observations per unit time.

Example

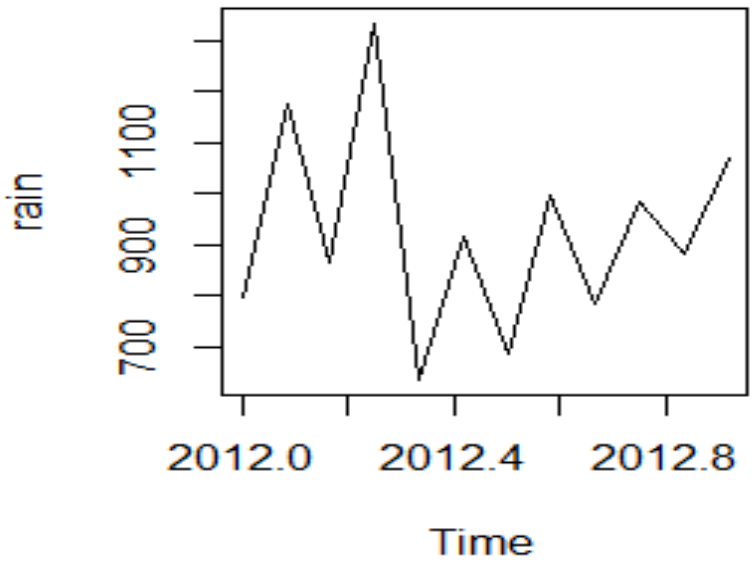
- Consider the annual rainfall details at a place starting from January 2012.
- We create an R time series object for a period of 12 months and plot it.

```
rainfall <-
    c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,
    882.8,1071)
rain <- ts(rainfall,start = c(2012,1),frequency = 12)
rain
```

```
        Jan    Feb    Mar    Apr    May    Jun    Jul    Aug
2012  799.0 1174.8  865.1 1334.6  635.4  918.5  685.5  998.6
        Sep    Oct    Nov    Dec
2012  784.2  985.0  882.8 1071.0
```

Plotting time series

plot(rain)



Representing Time Series Data

zoo and xts package

They define a data structure for time series, and they contain many useful functions for working with time series data. Create a zoo object this way, where x is a vector or data frame and dt is a vector of corresponding dates or datetimes:

```
library(zoo)
ts <- zoo(x, dt)
```

Create an xts object in this way:

```
library(xts)
ts <- xts(x, dt)
```

Convert between representations of the time series data by using as.zoo and as.xts:

as.zoo(ts)

Converts ts to a zoo object

as.xts(ts)

Converts ts to an xts object

Example

creates a zoo object that contains the price of IBM stock for the first five days of 2010; it uses Date objects for the index:

```
prices <- c(132.45, 130.85, 130.00, 129.55, 130.85)
dates <- as.Date(c("2010-01-04", "2010-01-05", "2010-01-06","2010-01-07","2010-01-08"))
ibm.daily <- zoo(prices, dates)
ibm.daily
```

Output:

```
2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
    132.45     130.85     130.00     129.55     130.85
```

- This example captures the price of IBM stock at one-second intervals.
- It represents time by the number of hours past midnight starting at 9:30 a.m. (1 second ≈ 0.00027778 hours):

```
prices <- c(131.18, 131.20, 131.17, 131.15, 131.17)
seconds <- c(9.5, 9.500278, 9.500556, 9.500833, 9.501111)
ibm.sec <- zoo(prices, seconds)
ibm.sec
```

Output:

```
    9.5 9.5003 9.5006 9.5008 9.5011
131.18 131.20 131.17 131.15 131.17
```

- Previous two examples used a single time series, where the data came from a vector.
- Both zoo and xts can also handle multiple, parallel time series. For this, capture the several time series in a data frame and then create a multivariate time series by calling the zoo (or xts) function:

  ts <- zoo(dfrm, dt) # OR: ts <- xts(dfrm, dt)

- The second argument is a vector of dates (or datetimes) for each observation.

- Once the data is captured inside a zoo or xts object, you can extract the pure data via coredata, which returns a simple vector (or matrix):

  coredata(ibm.daily)

```
 [1] 132.45 130.85 130.00 129.55 130.85
```

- You can extract the date or time portion via index:

  index(ibm.daily)

```
[1] "2010-01-04" "2010-01-05" "2010-01-06" "2010-01-07"
    "2010-01-08"
```

Subsetting a Time Series

You can index a zoo or xts object by position. Use one or two subscripts, depending

upon whether the object contains one time series or multiple time series:

ts[i]

- Selects the ith observation from a single time series

ts[j,i]

- Selects the ith observation of the jth time series of multiple time series

- You can index the time series by a date object. Use the same type of object as the index of your time series. This example assumes that the index contains Date objects:

  ts[as.Date("*yyyy-mm-dd*")]

- You can index it by a sequence of dates:

  dates <- seq(*startdate, enddate, increment*)

  ts[dates]

- The window function can select a range by start and end date:

  window(ts, start=*startdate, end=enddate*)

```
    ibm.daily[2]
2010-01-05
130.85
    ibm.daily[2:4]
2010-01-05 2010-01-06 2010-01-07
130.85          130.00       129.55
    ibm.daily[as.Date('2010-01-05')]
2010-01-05
130.85
```

We can select by a vector of Date objects:

dates <- seq(as.Date('2010-01-04'), as.Date('2010-01-08'), by=2)

ibm.daily[dates]

2010-01-04 2010-01-06 2010-01-08

132.45 130.00 130.85

The window function is easier for selecting a range of consecutive dates:

window(ibm.daily, start=as.Date('2010-01-05'), end=as.Date('2010-01-07'))

2010-01-05 2010-01-06 2010-01-07

130.85 130.00 129.55

Merging Several Time Series

- Use the zoo object to represent the time series; then use the merge function to combine them:

  merge(ts1, ts2)

- Merging two time series is an incredible headache when the two series have differing timestamps.

  merge(ibm.daily,ibm.sec)

	ibm.daily	ibm.sec
1970-01-10	NA	131.18
1970-01-10	NA	131.20
1970-01-10	NA	131.17
1970-01-10	NA	131.15
1970-01-10	NA	131.17
2010-01-04	132.45	NA
2010-01-05	130.85	NA
2010-01-06	130.00	NA
2010-01-07	129.55	NA
2010-01-08	130.85	NA

Lagging a Time Series

- You want to shift a time series in time, either forward or backward.
- Use the lag function. The second argument, k, is the number of periods to shift the data:

  lag(ts, k)

  ibm.daily

2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08

132.45 130.85 130.00 129.55 130.85

- To shift the data forward one day, we use k=+1:

  lag(ibm.daily, k=+1, na.pad=TRUE)

2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08

130.85 130.00 129.55 130.85 NA

- We also set na.pad=TRUE to fill the trailing dates with NA. Otherwise, they would simply be dropped, resulting in a shortened time series.

To shift the data backward one day, we use k=-1. Again we use na.pad=TRUE to pad the beginning with NAs:

lag(ibm.daily, k=-1, na.pad=TRUE)

2010-01-04	2010-01-05	2010-01-06	2010-01-07	2010-01-08
NA	132.45	130.85	130.00	129.55

Computing Successive Differences

- Given a time series, x, you want to compute the difference between successive observations: (x2 − x1), (x3 − x2), (x4 − x3),
- Use the diff function:

diff(x)

diff(ibm.daily)

2010-01-05	2010-01-06	2010-01-07	2010-01-08
-1.60	-0.85	-0.45	1.30

Performing Calculations on Time Series

Fortunately, when we divide one series by the other, R aligns the series for us and returns a zoo object:

diff(ibm.daily) / ibm.daily

2010-01-05 2010-01-06 2010-01-07 2010-01-08

-0.012227742 -0.006538462 -0.003473562 0.009935040

- We can scale the result by 100 to compute the percentage change, and the result is another a zoo object:

100 * (diff(ibm.daily) / ibm.daily)

2010-01-05 2010-01-06 2010-01-07 2010-01-08

-1.2227742 -0.6538462 -0.3473562 0.9935040

log(ibm.daily)

2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08

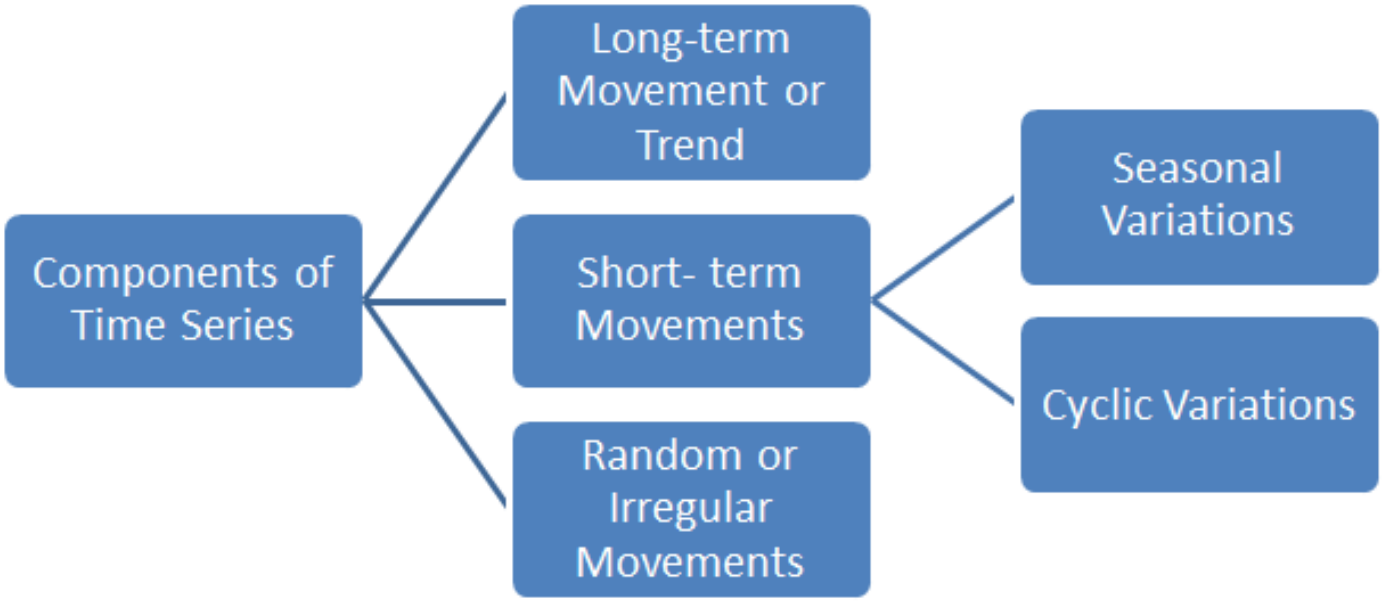
4.886205 4.874052 4.867534 4.864067 4.874052

Uses of Time Series

- It helps us to predict the future behavior of the variable based on past experience
- It is helpful for business planning as it helps in comparing the actual current performance with the expected one
- We can compare the changes in the values of different variables at different times or places, etc.

Components for Time Series Analysis

- The various reasons or the forces which affect the values of an observation in a time series are the components of a time series.
- The four categories of the components of time series are
  - Trend
  - Seasonal Variations
  - Cyclic Variations
  - Random or Irregular movements


Components of Time Series
Long-term Movement or Trend
Short- term Movements
Random or Irregular Movements
Seasonal Variations
Cyclic Variations

Trend

- The trend shows the general tendency of the data to increase or decrease during a long period of time.
- A trend is a smooth, general, long-term, average tendency.
- It is observable that the tendencies may increase, decrease or are stable in different sections of time.
- But the overall trend must be upward, downward or stable.

Linear and Non-Linear Trend

- If we plot the time series values on a graph in accordance with time t.
- The pattern of the data clustering shows the type of trend.
- If the set of data cluster more or less around a straight line, then the trend is linear otherwise it is non-linear.

Seasonal Variations

- These are the rhythmic forces which operate in a regular and periodic manner over a span of less than a year.
- The repeating short-term cycle in the series.
- They have the same or almost the same pattern during a period of 12 months.
- This variation will be present in a time series if the data are recorded hourly, daily, weekly, quarterly, or monthly.

Random or Irregular Movements

- They are not regular variations and are purely random or irregular.
- These fluctuations are unforeseen, uncontrollable, unpredictable, and are erratic.
- These forces are earthquakes, wars, flood and any other disasters.

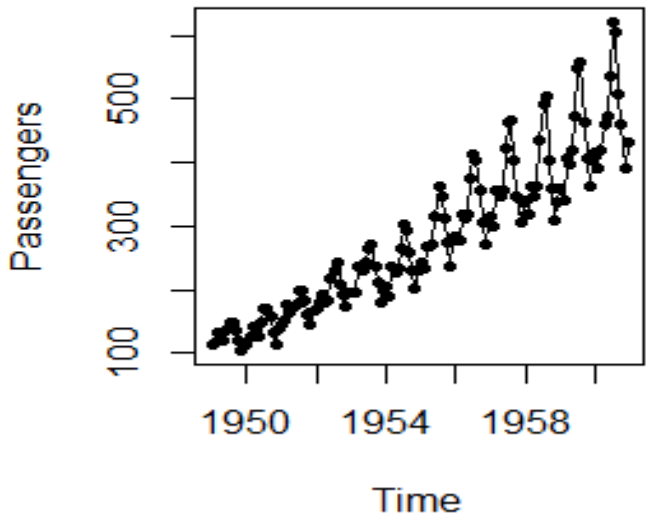
Example

- analyze the R dataset of monthly totals of international airline passengers between 1949 to 1960 and apply a simple model to forecast 3-point estimates for 1961's monthly totals.

```
data(AirPassengers)
class(AirPassengers)
 [1] "ts"
 start(AirPassengers)
[1] 1949 1
end(AirPassengers)
[1] 1960 12
frequency(AirPassengers)
[1] 12
```

Load Data & Plot

```
AP <- AirPassengers
plot(AP, ylab="Passengers",type="o", pch =20)
```



Interpretation

$$Y(t)=T(t)*S(t)*e(t))$$

where,

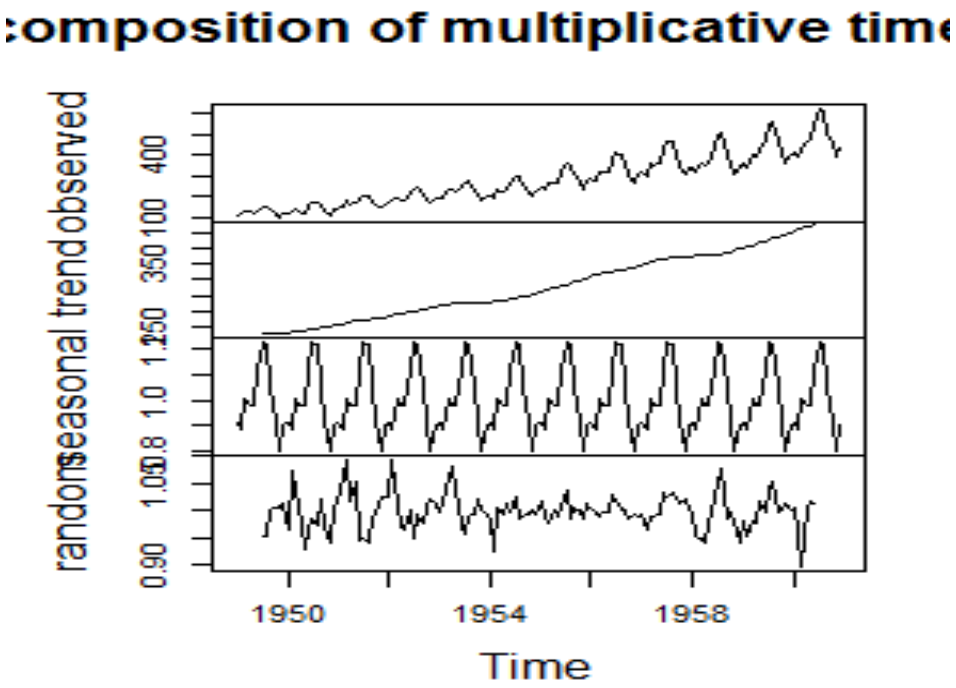
* Y(t) is the number of passengers at time t,
* T(t) is the trend component at time t,
* S(t) is the seasonal component at time t,
* and e(t) is the random error component at time t.

Decomposing the Data

- Decomposing the data into its trend, seasonal, and random error components will give some idea how these components relate to the observed dataset.

  AP.decompM <- decompose(AP, type = "multiplicative")

  plot(AP.decompM)



Model Fitting

Trend Component

- Inspecting the trend component in the decomposition plot suggests that the relationship is linear, thus fitting a linear model:

```
t <- seq(1, 144, 1)
modelTrend <- lm(formula = AP.decompM$trend ~ t)
```

summary(modelTrend)

```
Call:
lm(formula = AP.decompM$trend ~ t)

Residuals:
     Min       1Q   Median       3Q      Max
-22.9162  -6.0845   0.6094   5.8658  23.4748

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 84.64827    2.05100   41.27   <2e-16 ***
t            2.66694    0.02504  106.50   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 10.96 on 130 degrees of freedom
  (12 observations deleted due to missingness)
Multiple R-squared:  0.9887,    Adjusted R-squared:  0.9886
F-statistic: 1.134e+04 on 1 and 130 DF,  p-value: < 2.2e-16
```

- Therefore, the relationship between trend and time can be expressed as:
  T(t)=2.667t+84.648
- And so for 1961 (time 145 to 156 inc.), the trend component (T) is:

  Data1961 <- data.frame("T" = 2.667*seq(145, 156, 1) + 84.648, S=rep(0,12), e=rep(0,12), row.names = c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"))

  Data1961

- Output:

	T	S	e
Jan	471.363	0	0
Feb	474.030	0	0
Mar	476.697	0	0
Apr	479.364	0	0
May	482.031	0	0
Jun	484.698	0	0
Jul	487.365	0	0
Aug	490.032	0	0
Sep	492.699	0	0
Oct	495.366	0	0
Nov	498.033	0	0
Dec	500.700	0	0

Seasonal Component

- Inspecting the seasonal (S) component of the decomposition reveals:

  AP.decompM$seasonal

- Thus the seasonal (S) component to the new 1961 dataset is:

  Data1961$S <- unique(AP.decompM$seasonal)

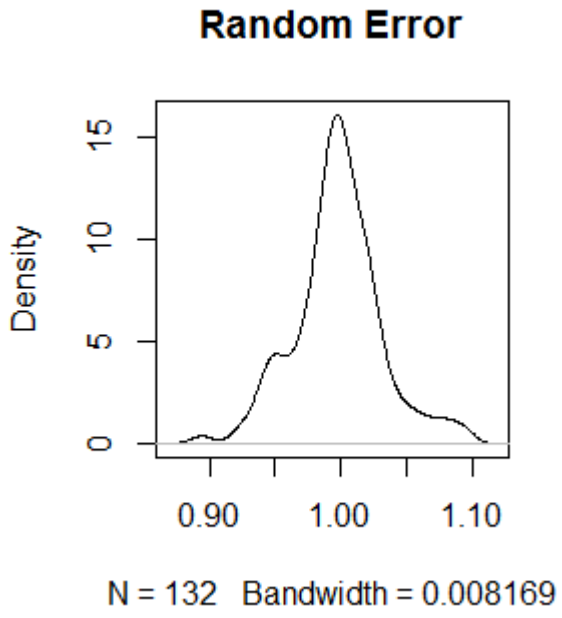
  Data1961

	T	S	e
Jan	471.363	0.9102304	0
Feb	474.030	0.8836253	0
Mar	476.697	1.0073663	0
Apr	479.364	0.9759060	0
May	482.031	0.9813780	0
Jun	484.698	1.1127758	0
Jul	487.365	1.2265555	0
Aug	490.032	1.2199110	0
Sep	492.699	1.0604919	0
Oct	495.366	0.9217572	0
Nov	498.033	0.8011781	0
Dec	500.700	0.8988244	0

Random Error Component

- Ploting the density estimation of the random error (e) component of the decomposition shows an approximate normal distribution:

  plot(density(AP.decompM$random[7:138]), main="Random Error")



- Bootstrapping the mean statistic of the random error would produce an accurate approximation of the population mean of the random error.

  mean(AP.decompM$random[7:138])

Output: [1] 0.9982357

  which is 1.

- Thus the decomposed dataset for 1961 is:

  Data1961$e <- 1

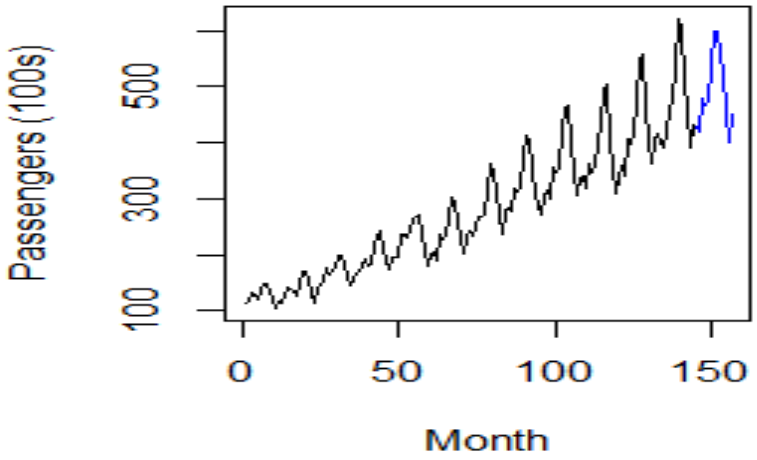
Predictions (1961)

```
Data1961$R <- Data1961$T * Data1961$S * Data1961$e
Data1961$R
```

Output:

```
[1] 429.0489 418.8649 480.2085 467.8142 473.0546 539.3602
    597.7802 597.7954 522.5033  456.6072 399.0131 450.0414
```

```
xr = c(1,156)
plot(AP.decompM$x, xlim=xr, ylab = "Passengers (100s)", xlab =
    "Month")
lines(data.frame(AP.decompM$x))
lines(Data1961$R, x=seq(145,156,1), col="blue")
```



ARIMA model

- A popular and widely used statistical method for time series forecasting is the ARIMA model.
- ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average.
- It is a class of model that captures a suite of different standard temporal structures in time series data.
- We will be following an ARIMA modeling procedure of the AirPassengers dataset as follows:
  1. Perform exploratory data analysis
  2. Decomposition of data
  3. Test the stationarity
  4. Fit a model used an automated algorithm
  5. Calculate forecasts

Library(ggfortify**)**

library(tseries)

library(forecast)

LOAD DATA

data(AirPassengers)

AP <- AirPassengers

PERFORM EXPLORATORY DATA ANALYSIS

class(AP)

summary(AP)

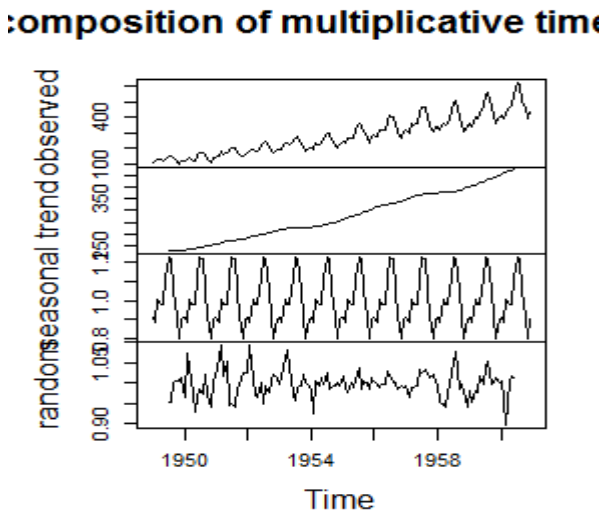
frequency(AP)

Plot(AP)

TIME SERIES DECOMPOSITION

- We will decompose the time series for estimates of trend, seasonal, and random components using moving average method.
- With this model, we will use the decompose function in R.

```
decomposeAP <- decompose(AP,"multiplicative")
plot(decomposeAP)
```



TEST STATIONARITY OF THE TIME SERIES

Test stationarity of the time series (ADF)

- In order to test the stationarity of the time series, let's run the Augmented Dickey-Fuller Test using the adf.test function from the tseries R package.

  First set the hypothesis test:

- The null hypothesis H0 : that the time series is non stationary The alternative hypothesis HA : that the time series is stationary

```
adf.test(AP)
```

Output:

```
Augmented Dickey-Fuller Test
data: AP
Dickey-Fuller = -7.3186, Lag order = 5, p-value = 0.01
 alternative hypothesis: stationary
```

As per the test results above, the p-value is 0.01 which is <0.05 therefore we reject the null in favour of the alternative hypothesis that the time series is stationary.

FIT A TIME SERIES MODEL

ARIMA Model

- Use the auto.arima function from the forecast R package to fit the best model and coefficients, given the default parameters including seasonality as TRUE.

  arimaAP <- auto.arima(AP)

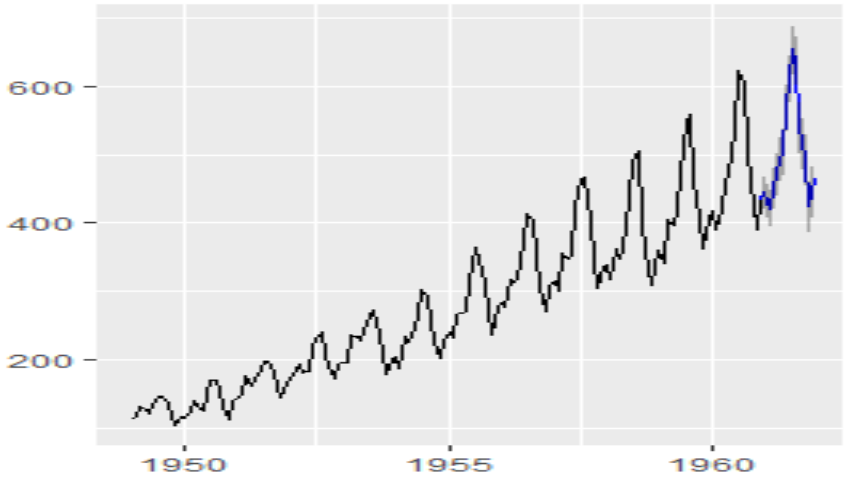
  arimaAP

```
Series: AP
ARIMA(2,1,1)(0,1,0)[12]

Coefficients:
          ar1     ar2      ma1
      0.5960  0.2143  -0.9819
s.e.  0.0888  0.0880   0.0292

sigma^2 estimated as 132.3:  log likelihood=-504.92
AIC=1017.85   AICc=1018.17   BIC=1029.35
```

CALCULATE FORECASTS

- Finally we can plot a forecast of the time series using the forecast function, with a 95% confidence interval where h is the forecast horizon periods in months.

  forecastAP <- forecast(arimaAP, level = c(95), h = 12)
  autoplot(forecastAP)



Non Linear Least Square

- Linear regression is a basic tool. It works on the assumption that there exists a linear relationship between the dependent and independent variable.
- However, not all problems have such a linear relationship. In fact, many of the problems we see today are nonlinear in nature.
- A very basic example is our own decision making process which involves deciding an outcome based on various questions. For example, when we decide to have dinner, our thought process is not linear. It is based a combination of our tastes, our budget, our past experiences with a restaurant, alternatives available, weather conditions etc.

- This is how non-linear regression came into practice – a powerful alternative to linear regression for nonlinear situations.
- Similar to linear regression, nonlinear regression draws a line through the set of available data points in such a way that the line fits to the data with the only difference that the line is not a straight line or in other words, not linear.
- The goal of both linear and non-linear regression is to adjust the values of the model's parameters to find the line or curve that comes closest to your data.
- To perform this, Non-Linear Least Square approach is used to minimize the total sum of squares of residual values or error values i.e., the difference between vertical points on the graph from regression line and will fit the non-linear function accordingly.

Syntax

nls(formula, data, start)

Description of the parameters

- **formula** is a nonlinear model formula including variables and parameters.
- **data** is a data frame used to evaluate the variables in the formula.
- **start** is a named list or named numeric vector of starting estimates.

Example

- We will consider a nonlinear model with assumption of initial values of its coefficients.
- So let's consider the below equation for this purpose −

  a = b1*x^2+b2

- Let's assume the initial coefficients to be 1 and 3 and fit these values into nls() function.

```
x <- c(1.6,2.1,2,2.23,3.71,3.25,3.4,3.86,1.19,2.21)
y <- c(5.19,7.43,6.94,8.11,18.75,14.88,16.06,19.12,3.21,7.58)

model <- nls(y ~ b1*x^2+b2,start = list(b1 = 1,b2 = 3))
summary(model)
```

```
Formula: y ~ b1 * x^2 + b2

Parameters:
   Estimate Std. Error t value Pr(>|t|)
b1  1.19542    0.02503  47.764 4.08e-11 ***
b2  1.99692    0.21663   9.218 1.55e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3678 on 8 degrees of freedom

Number of iterations to convergence: 1
Achieved convergence tolerance: 1.082e-08
```

```
a <- data.frame(x=1.6)
predict(model,a)
```

Output:

```
[1] 5.057202
```

```
sum(resid(model)^2)
```

Output:

```
[1] 1.081935
```

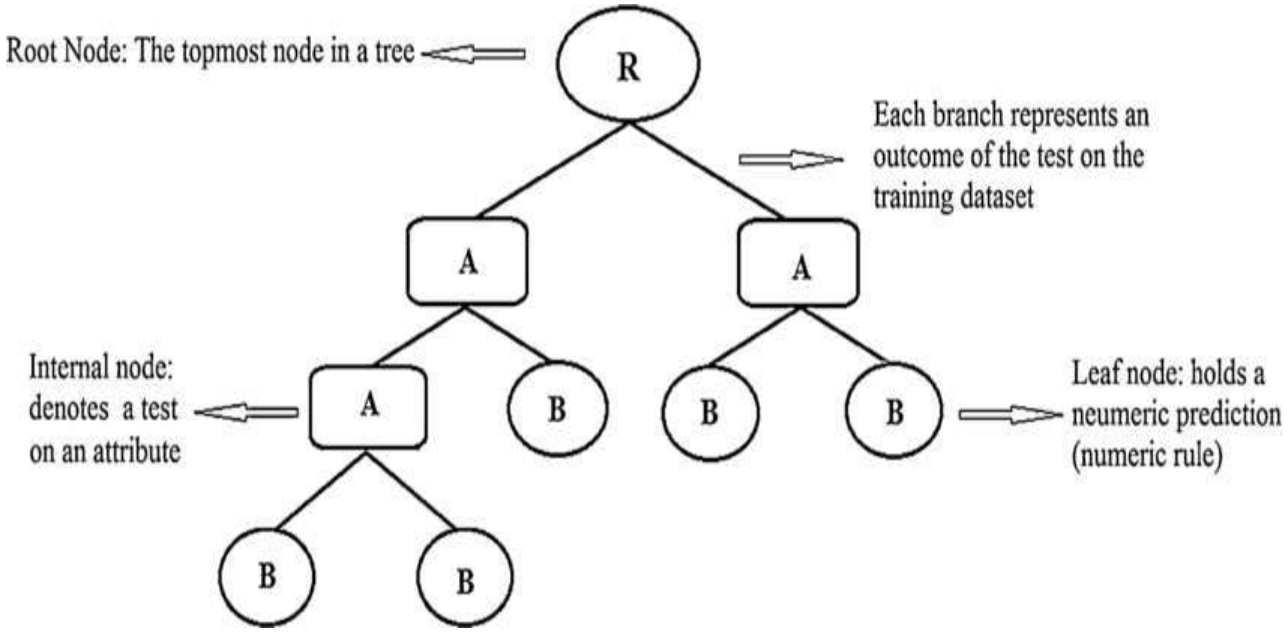
Decision Tree

- Decision tree builds classification or regression models in the form of a tree structure.
- It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed.
- The final result is a tree with **decision nodes** and **leaf nodes**.
- A decision node has two or more branches .
- Leaf node represents a classification or decision.
- The topmost decision node in a tree which corresponds to the best predictor called **root node**.
- Decision trees can handle both categorical and numerical data.

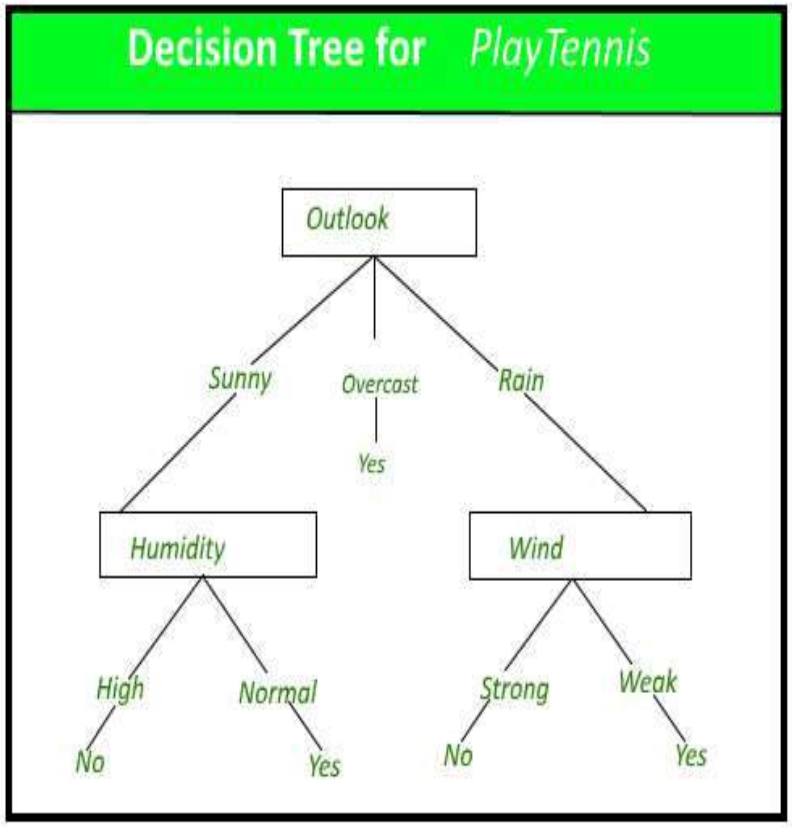
Decision Tree

- A decision tree is a flowchart-like structure in which each internal node represents a “test” on an attribute , each branch represents the outcome of the test, and each leaf node represents a class label .
- The paths from root to leaf represent classification rules.
- Decision trees are referred to as CART(Classification and Regression trees).
- Tree based methods empower predictive models with high accuracy, stability and ease of interpretation.
- Unlike linear models, they map non-linear relationships quite well.

Example – Decision Tree



Example - Decision Tree



Terms – Decision Tree

- **Root Node:** It represents entire population or sample and this further gets divided into two or more homogeneous sets.
- **Splitting:** It is a process of dividing a node into two or more sub-nodes.
- **Decision Node:** When a sub-node splits into further sub-nodes, then it is called decision node.
- **Leaf/ Terminal Node:** Nodes do not split is called Leaf or Terminal node.
- **Pruning:** When we remove sub-nodes of a decision node, this process is called pruning. You can say opposite process of splitting.

Terms – Decision Tree

- **Branch / Sub-Tree:** A sub section of entire tree is called branch or sub-tree.
- **Parent and Child Node:** A node, which is divided into sub-nodes is called parent node of sub-nodes whereas sub-nodes are the child of parent node.

Decision Tree Algorithm Pseudocode

- The decision tree algorithm tries to solve the problem, by using tree representation.
- Each internal node of the tree corresponds to an attribute, and each leaf node corresponds to a class label.
- Place the best attribute of the dataset at the root of the tree.
- Split the training set into subsets. Subsets should be made in such a way that each subset contains data with the same value for an attribute.
- Repeat step 1 and step 2 on each subset until you find leaf nodes in all the branches of the tree.

Types of Decision Trees

- **ID3** → (extension of D3)

  - Iterative Dichotomiser 3

  C4.5 → (successor of ID3)

  CART → (Classification And Regression Tree)

Steps in ID3 algorithm:

- It begins with the original set S as the root node.
- On each iteration of the algorithm, it iterates through the very unused attribute of the set S and calculates **Entropy(H)** and **Information gain(IG)** of this attribute.
- It then selects the attribute which has the smallest Entropy or Largest Information gain.
- The set S is then split by the selected attribute to produce a subset of the data.
- The algorithm continues to recur on each subset, considering only attributes never selected before.

Attribute Selection Measures

- Entropy,
  Information gain,
  Gini index,
  Gain Ratio,
  Reduction in Variance
  Chi-Square

Entropy

- Entropy is a measure of the randomness in the information being processed.
- The higher the entropy, the harder it is to draw any conclusions from that information.
- ID3 follows the rule — A branch with an entropy of zero is a leaf node and A branch with entropy more than zero needs further splitt

$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

Play Golf	
Yes	No
9	5

Entropy(PlayGolf) = Entropy (5,9)
= Entropy (0.36, 0.64)
= - (0.36 $\log_2$ 0.36) - (0.64 $\log_2$ 0.64)
= 0.94

Entropy in binary classification

- Entropy measures the *impurity* of a collection of examples. It depends from the distribution of the random variable *p*.
  - *S* is a collection of training examples
  - p_+ the proportion of positive examples in *S*
  - p_- the proportion of negative examples in *S*

$$Entropy\ (S) \equiv -p_+ \log_2 p_+ - p_- \log_2 p_-$$

$$Entropy\ ([14+, 0-]) = -14/14 \log_2 (14/14) - 0 \log_2 (0) = 0$$

$$Entropy\ ([9+, 5-]) = -9/14 \log_2 (9/14) - 5/14 \log_2 (5/14) = 0.94$$

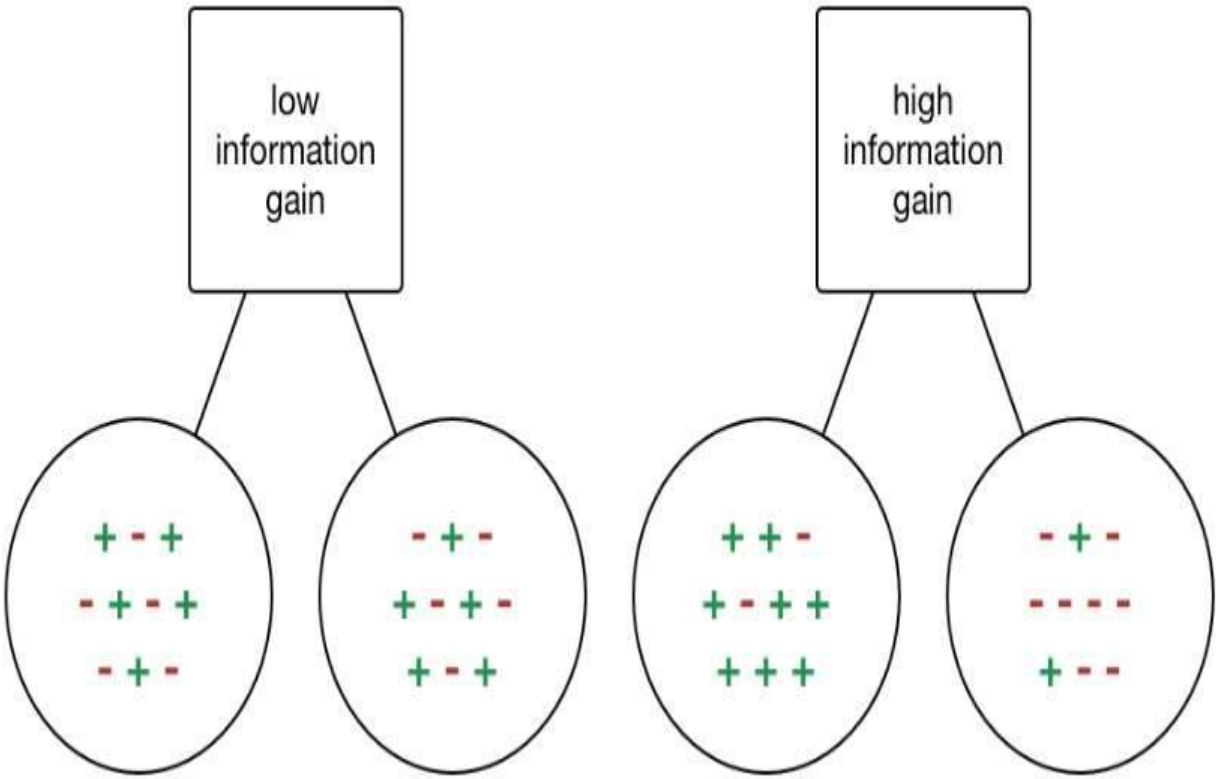
$$Entropy\ ([7+, 7-]) = -7/14 \log_2 (7/14) - 7/14 \log_2 (7/14) =$$

$$= 1/2 + 1/2 = 1$$

Information Gain

- Information gain or **IG** is a statistical property that measures how well a given attribute separates the training examples according to their target classification.
- Constructing a decision tree is all about finding an attribute that returns the highest information gain and the smallest entropy.

Information Gain



Information Gain

Information Gain(T,X) = Entropy(T) - Entropy(T, X)

IG(PlayGolf, Outlook) = E(PlayGolf) - E(PlayGolf, Outlook)
= 0.940 - 0.693
= 0.247

Classification using ID3 algorithm

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Classification Using ID3

- There are four independent variable to determine the dependent variable.
- The independent variables are Outlook, Temperature, Humidity, and Wind.
- The dependent variable is whether to play football or not.
- As the first step, we have to find the parent node for our decision tree. For that follow the steps:

Find the entropy of class variable.

$E(S) = -[(9/14)\log(9/14) + (5/14)\log(5/14)] = 0.94$

Data for outlook

		play		
		yes	no	total
	sunny	3	2	5
Outlook	overcast	4	0	4
	rainy	2	3	5
				14

$$E(S, \text{outlook}) = (5/14)*E(3,2) + (4/14)*E(4,0) + (5/14)*E(2,3)$$
$$= (5/14)(-(3/5)\log(3/5)-(2/5)\log(2/5))+ (4/14)(0)$$
$$+ (5/14)((2/5)\log(2/5)-(3/5)\log(3/5)) = 0.693$$

***calculate average weighted entropy*.**

- we have found the total of weights of each feature multiplied by probabilities.
- ***Next step is to find the information gain***. It is difference between parent entropy and average weighted entropy we found above.
- IG(S, outlook) = 0.94 - 0.693 = 0.247
- Similarly find Information gain for Temperature, Humidity and Windy.
- IG(S, Temperature) = 0.940 - 0.911 = 0.029

- IG(S, Humidity) = 0.940 - 0.788 = 0.152
- IG(S, Windy) = 0.940 - 0.8932 = 0.048
- ***Now select the feature having largest information gain***.
- Here it is Outlook. So it forms first node(root node) of our decision tree.

Data after root node is decided

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Overcast	Hot	High	Weak	Yes
Overcast	Cool	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Rain	Mild	Normal	Weak	Yes
Rain	Mild	High	Strong	No

- Since overcast contains only examples of class ‘Yes’ we can set it as yes. That means If outlook is overcast football will be played.

- Next step is to find the next node in our decision tree.
- Now we will find one under sunny.
- We have to determine which of the following Temperature ,Humidity or Wind has higher information gain.

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes

- Calculate parent entropy E(sunny)
- E(sunny) = (-(3/5)log(3/5)-(2/5)log(2/5)) = 0.971.
- Now Calculate information gain of Temperature. IG(sunny, Temperature)

		play		
		yes	no	total
	hot	0	2	2
Temperature	cool	1	1	2
	mild	1	0	1
				5

E(sunny, Temperature) = (2/5)*E(0,2) + (2/5)*E(1,1) + (1/5)*E(1,0)=2/5=0.4

Now calculate information gain.

IG(sunny, Temperature) = 0.971–0.4 =0.571

Similarly we get

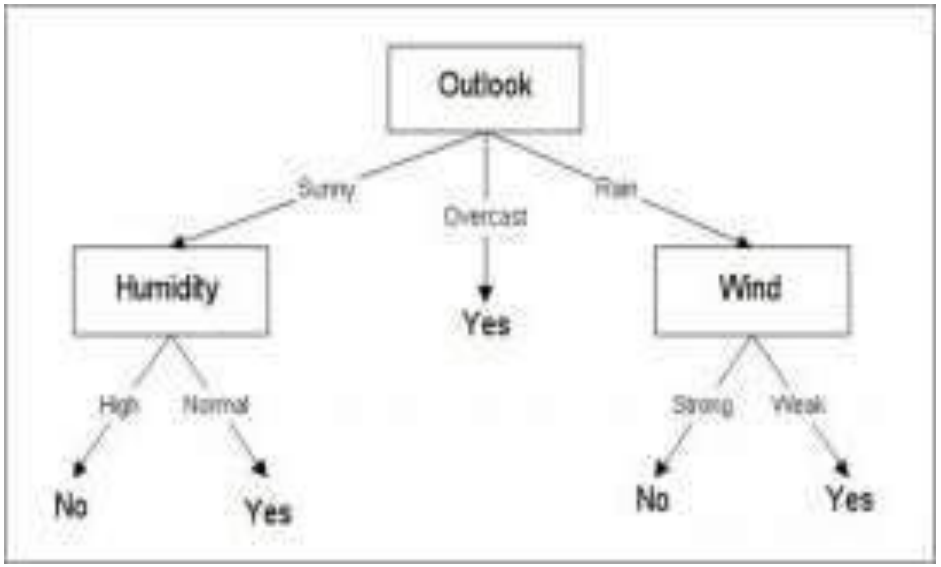
IG(sunny, Humidity) = 0.971

IG(sunny, Windy) = 0.020

- Here IG(sunny, Humidity) is the largest value. So Humidity is the node which comes under sunny.
- For humidity from the above table, we can say that play will occur if humidity is normal and will not occur if it is high. Similarly, find the nodes under rainy.

	play	
Humidity	yes	no
high	0	3
normal	2	0

Decision Tree to Decision Rules



- IF outlook=overcast then play=Yes
- IF Outlook=sunny and Humidity=High then play=No

Disadvantages:

- More likely to overfit noisy data.
- The probability of overfitting on noise increases as a tree gets deeper.
- A solution for it is **pruning**.

Advantages:

- It requires fewer data preprocessing from the user, for example, there is no need to normalize columns.
- Decision trees are easy to interpret and visualize.
- It can be used for feature engineering such as predicting missing values, suitable for variable selection.

```
library(rpart)
library(rpart.plot)

Outlook=c('Sunny','Sunny','Overcast','Rain','Rain','Rain','Overcast','Sunny','Sunny','Rain','Sunny','Overcast','Overcast','Rain')
Temp=c('Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild','Mild','Mild','Hot','Mild')
Humidity=c('High','High','High','High','Normal','Normal','Normal','High','Normal','Normal','Normal','High','Normal','High')
Windy=c('Week','Strong','Weak','Weak','Weak','Strong','Strong','Weak','Weak','Weak','Strong','Strong','Weak','Strong')
Play=c('No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes','Yes','No')

dat=data.frame(Outlook,Temp,Humidity,Windy,Play)
```

```
model <- rpart( Play ~ Outlook + Temp + Humidity + Windy,
    data = dat, control = rpart.control(minsplit = 2))
rpart.plot(model)
```

Minsplit - the minimum number of observations that must exist in a node in order for a split to be attempted.

```
a <-
    data.frame(Outlook="Rain",Humidity="Normal",Temp="Hot",
    Windy="Strong")
predict(model,a)
```

The R package Party

- install.packages("party")
- The package "party" has the function ctree() which is used to create and analyze decison tree.

Syntax

ctree(formula, data)

- formula is a formula describing the predictor and response variables.
- data is the name of the data set used.

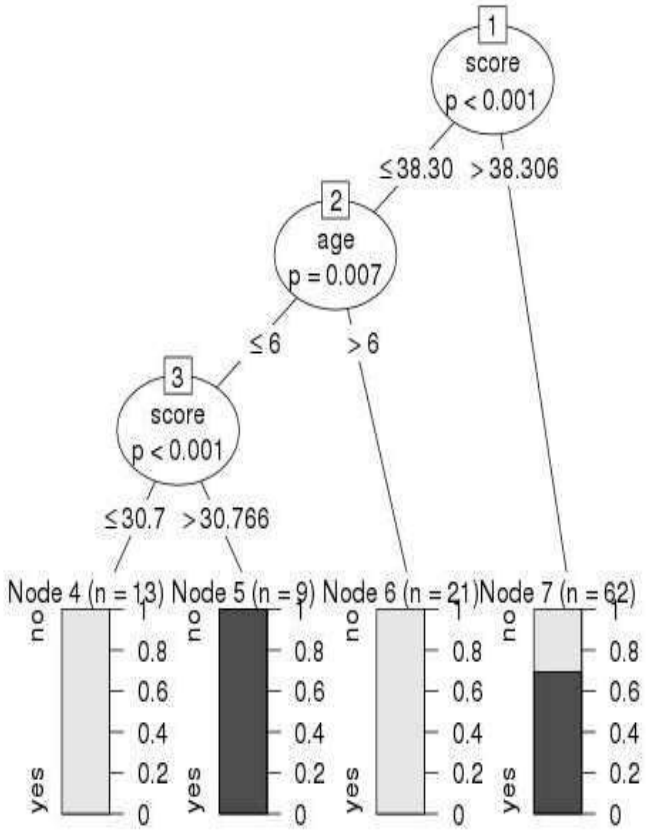
Input Data

- The R in-built data set named **readingSkills** to create a decision tree.
- It describes the score of someone's readingSkills if we know the variables "age","shoesize","score" and whether the person is a native speaker or not.

```
library(party)
head(readingSkills)
  nativeSpeaker   age   shoeSize      score
1           yes     5   24.83189   32.29385
2           yes     6   25.95238   36.63105
3            no    11   30.42170   49.60593
4           yes     7   28.66450   40.28456
5           yes    11   31.88207   55.46085
6           yes    10   30.07843   52.83124
```

- We will use the **ctree()** function to create the decision tree and see its graph.

```
library(party)
input.dat <- readingSkills[c(1:105),]
output.tree <- ctree(  nativeSpeaker ~ age + shoeSize + score,
  data = input.dat)
plot(output.tree)
```


1
score
p < 0.001
≤ 38.30
> 38.306
2
age
p = 0.007
≤ 6
> 6
3
score
p < 0.001
≤ 30.7
> 30.766
Node 4 (n = 13)
Node 5 (n = 9)
Node 6 (n = 21)
Node 7 (n = 62)
no
yes
1
0.8
0.6
0.4
0.2
0

- From the decision tree shown above we can conclude that anyone whose readingSkills score is less than 38.3 and age is more than 6 is not a native Speaker.

```
library(datasets)
library(caTools)
library(party)
library(dplyr)
library(magrittr)
```

Step 2:

```
head(readingSkills)
```

Step 3: Splitting dataset into 4:1 ratio for train and test data

```
sample_data = sample.split(readingSkills, SplitRatio = 0.8)
train_data <- subset(readingSkills, sample_data == TRUE)
test_data <- subset(readingSkills, sample_data == FALSE)
```

Step 4: Create the decision tree model using ctree and plot the model

```
model<- ctree(nativeSpeaker ~ ., train_data)
plot(model)
```

Step 5: Making a prediction

```
predict_model<-predict(model, test_data)
m_at <- table(test_data$nativeSpeaker, predict_model)
m_at
```

Step 6: Determining the accuracy of the model developed

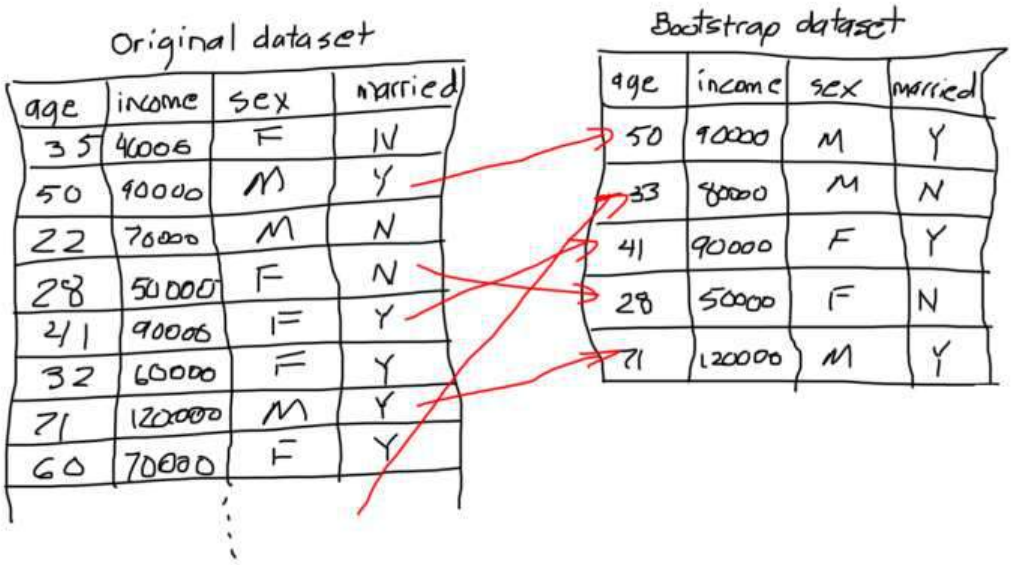
```
ac_Test <- sum(diag(m_at)) / sum(m_at)
print(paste('Accuracy for test', ac_Test))
```

Random Forest

- In the **random forest** approach, a large number of **decision** trees are created. Every observation is fed into every **decision** tree.
- The most common outcome for each observation is used as the final output.
- A new observation is fed into all the trees and taking a majority vote for each classification model.
- The **R** package "**randomForest**" is used to create **random forests**.

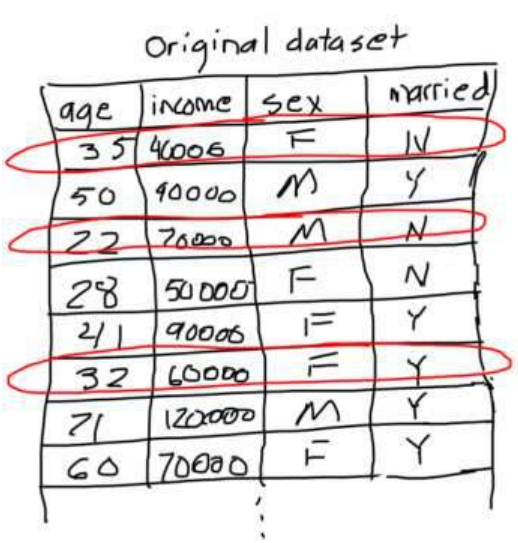
Random Forest

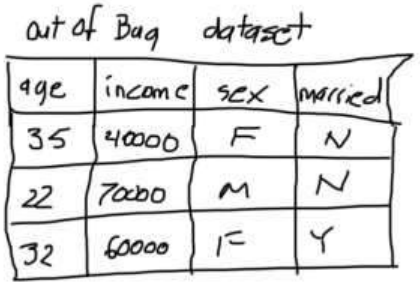
- The random forest algorithm works by aggregating the predictions made by multiple decision trees of varying depth.
- Every decision tree in the forest is trained on a subset of the dataset called the bootstrapped dataset.



Out of Bag

- The portion of samples that were left out during the construction of each decision tree in the forest are referred to as the Out-Of-Bag (OOB) dataset.
- The model will automatically evaluate its own performance by running each of the samples in the OOB dataset through the forest.



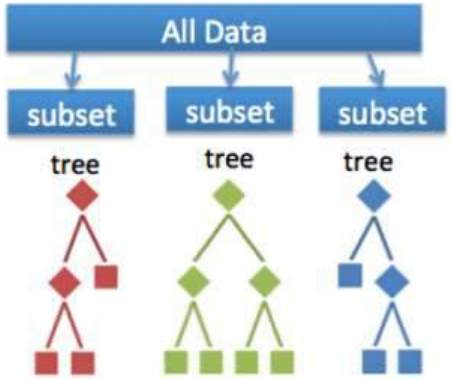


Random Forest

- When deciding on the criteria with which to split a decision tree, the impurity produced by each feature is measured using the Gini index or entropy.
- In random forest, however, we randomly select a predefined number of feature as candidates.
- The latter will result in a larger variance between the trees which would otherwise contain the same features (i.e those which are highly correlated with the target label).

Random Forest

- When the random forest is used for classification and is presented with a new sample, the final prediction is made by taking the majority of the predictions made by each individual decision tree in the forest.
- The final prediction is made by taking the average of the predictions made by each individual decision tree in the forest.
- Generally, the more trees in the forest the more robust the forest looks like.
- Similarly, in the random forest classifier, the higher the number of trees in the forest, greater is the accuracy of the results.



Random Forest

- Random forest is an ensemble of decision trees, it randomly selects a set of parameters and creates a decision tree for each set of chosen parameters.

Example- Random Forest

- consider the below sample data set. In this data set we have four predictor variables, namely:
- Weight
- Blood flow
- Blocked Arteries
- Chest Pain

Blood Flow	Blocked Arteries	Chest Pain	Weight	Heart Disease
Abnormal	No	No	130	No
Normal	Yes	Yes	195	Yes
Normal	No	Yes	218	No
Abnormal	Yes	Yes	180	Yes

Steps – Creating Random Forest

- This data set is used to create a Random Forest that predicts if a person has heart disease or not.

Creating A Random Forest

Step 1: Create a Bootstrapped Data Set

- Bootstrapping is an estimation method used to make predictions on a data set by re-sampling it.
- To create a bootstrapped data set, randomly select samples from the original data set.
- can select the same sample more than once.

Step 2: Creating Decision Trees

- Next task is to build a Decision Tree by using the bootstrapped data set created in the previous step.

Step 3: Go back to Step 1 and Repeat

- Each Decision Tree predicts the output class based on the respective predictor variables used in that tree.
- Finally, the outcome of all the Decision Trees in a Random Forest is recorded and the class with the majority votes is computed as the output class.
- Thus, now create more decision trees by considering a subset of random predictor variables at each step.
- To do this, go back to step 1, create a new bootstrapped data set and then build a Decision Tree by considering only a subset of variables at each step.

- **Step 4: Predicting the outcome of a new data point** Now that we've created a random forest, let's see how it can be used to predict whether a new patient has heart disease or not.
- **Step 5: Evaluate the Model**
- Our final step is to evaluate the Random Forest model. Earlier while we created the bootstrapped data set, we left out one entry/sample since we duplicated another sample. In a real-world problem, about 1/3rd of the original data set is not included in the bootstrapped data set.

How to do it in R

install.packages("randomForest")

Syntax

randomForest(formula, data)

- formula is a formula describing the predictor and response variables.
- data is the name of the data set used.

- R in-built data set named readingSkills is used to create a decision tree.
- It describes the score of someone's readingSkills if we know the variables “age","shoesize","score" and whether the person is a native speaker.

```
library(party)
head(readingSkills)
```

	nativeSpeaker	age	shoeSize	score
1	yes	5	24.83189	32.29385
2	yes	6	25.95238	36.63105
3	no	11	30.42170	49.60593
4	yes	7	28.66450	40.28456
5	yes	11	31.88207	55.46085
6	yes	10	30.07843	52.83124

- Use the **randomForest()** function to create the decision tree and see it's graph.

Create the forest.

output.forest <- randomForest(nativeSpeaker ~ age + shoeSize + score, data = readingSkills)

output.forest

```
Call:
 randomForest(formula = nativeSpeaker ~ age + shoeSize + score,
      data = readingSkills)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 1

        OOB estimate of  error rate: 1.5%
Confusion matrix:
    no yes class.error
no  99   1        0.01
yes  2  98        0.02
```

Prediction

```
a<-data.frame(age=10,shoeSize=30.265,score=51.678)
predict(model,a)
```

Output:

1

yes

Example 2

```
Library(caTools)
sample_data = sample.split(readingSkills, SplitRatio = 0.8)
train_data <- subset(readingSkills, sample_data == TRUE)
test_data <- subset(readingSkills, sample_data == FALSE)
model<- randomForest(nativeSpeaker ~ ., train_data)
model
predict_model<-predict(model, test_data)
m_at <- table(test_data$nativeSpeaker, predict_model)
m_at
ac_Test <- sum(diag(m_at)) / sum(m_at)
print(paste('Accuracy for test', ac_Test))
```

Survival Analysis

- Survival analysis deals with predicting the time when a specific event is going to occur.
- It is also known as failure time analysis or analysis of time to death.
- For example predicting the number of days a person with cancer will survive or predicting the time when a mechanical system is going to fail.
- The R package named **survival** is used to carry out survival analysis.
- This package contains the function **Surv()** which takes the input data as a R formula and creates a survival object among the chosen variables for analysis.
- Then we use the function **survfit()** to create a plot for the analysis.

install.packages("survival")

Syntax

Surv(time,event)

survfit(formula)

Description of the parameters

time is the follow up time until the event occurs.

event indicates the status of occurrence of the expected event.

formula is the relationship between the predictor variables.

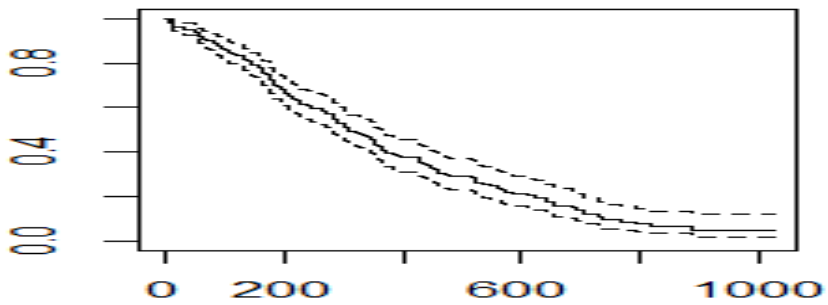
Example

- We will use the Survival package for the analysis.
- using **Lung** dataset preloaded in survival package which contains data of 228 patients with advanced lung cancer from North Central cancer treatment group based on 10 features.
- Here, we are interested in "**time**" and "**status**" as they play an important role in analysis.
- Time represents the survival time of patients. Since patients survive, we will consider their status as dead or non-dead.

```
install.packages("survival")
library(survival)
# Dataset information
?lung
head(lung)
# Fitting the survival model
Survival_Function = survfit(Surv(lung$time, lung$status ==
    2)~1)
Survival_Function
```

- The **Surv()** function takes time and status as input and creates an object which serves as the input of **survfit()** function.
- We pass ~1 in **survfit()** function to ensure that we are telling the function to fit the model on basis of survival object and have an interrupt.
- **survfit()** creates survival curves and prints number of values, number of events(people suffering from cancer), the median time and 95% confidence interval.
-

plot(Survival_Function)



- Here, the x-axis specifies "**Number of days**" and the y-axis specifies the "**probability of survival**". The dashed lines are upper confidence interval and lower confidence interval.
- We also have the confidence interval which shows the margin of error expected i.e In days of surviving 200 days, upper confidence interval reaches 0.76 or 76% and then goes down to **0.60 or 60%**.

Hypothesis Testing

- A hypothesis is made by the researchers about the data collected for any experiment or data set.
- A hypothesis is an assumption made by the researchers that are not mandatory true.
- In simple words, a hypothesis is a decision taken by the researchers based on the data of the population collected.
- **Hypothesis Testing** in R Programming is a process of testing the hypothesis made by the researcher or to validate the hypothesis.
- To perform hypothesis testing, a random sample of data from the population is taken and testing is performed. Based on the results of testing, the hypothesis is either selected or rejected. This concept is known as **Statistical Inference**.

Four Step Process of Hypothesis Testing

- **State the hypothesis-** This step is started by stating null and alternative hypothesis which is presumed as true.
- **Formulate an analysis plan and set the criteria for decision-** In this step, significance level of test is set. The significance level is the probability of a false rejection in a hypothesis test.
- **Analyze sample data-** In this, a test statistic is used to formulate the statistical comparison between the sample mean and the mean of the population or standard deviation of the sample and standard deviation of the population.
- **Interpret decision-** The value of the test statistic is used to make the decision based on the significance level. For example, if the significance level is set to 0.1 probability, then the sample mean less than 10% will be rejected. Otherwise, the hypothesis is retained to be true.

t-test

- The basic idea behind a t-test is to use statistic to evaluate two contrary hypotheses.
- H0: NULL hypothesis: The average is the same as the sample used
- H3: True hypothesis: The average is different from the sample used
- The t-test is commonly used with small sample sizes. To perform a t-test, you need to assume normality of the data.
- The basic syntax for t.test() is:

  t.test(x, y = NULL, mu = 0, var.equal = FALSE)

arguments: - x : A vector to compute the one-sample t-test

- y: A second vector to compute the two sample t-test –
- mu: Mean of the population
- var.equal: Specify if the variance of the two vectors are equal. By default, set to `FALSE`

One-sample t-test

The t-test, compares the mean of a vector against a theoretical mean. The formula used to compute the t-test is:

$$t = \frac{m - \mu}{\frac{s}{\sqrt{n}}}$$

Here

m refers to the mean

μ to the theoretical mean

s is the standard deviation

n the number of observations.

- To evaluate the statistical significance of the t-test, you need to compute the **p-value**.
- The **p-value** ranges from 0 to 1, and is interpreted as follow:
- A p-value lower than 0.05 means you are strongly confident to reject the null hypothesis, thus H3 is accepted.
- A p-value higher than 0.05 indicates that you don't have enough evidences to reject the null hypothesis.

Example

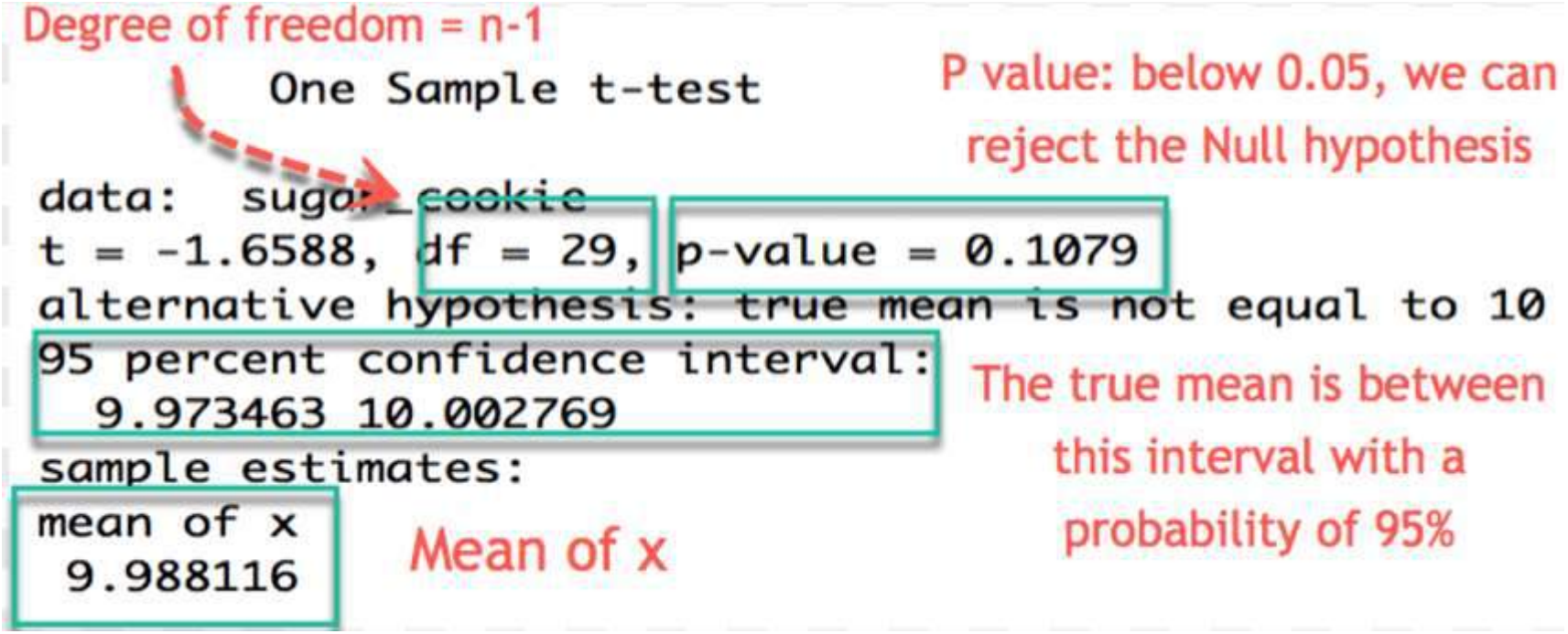
- Suppose you are a company producing cookies.
- Each cookie is supposed to contain 10 grams of sugar. The cookies are produced by a machine that adds the sugar in a bowl before mixing everything.
- You believe that the machine does not add 10 grams of sugar for each cookie. If your assumption is true, the machine needs to be fixed. You stored the level of sugar of thirty cookies.

- You can create a distribution with 30 observations with a mean of 9.99 and a standard deviation of 0.04.

  set.seed(123)

  sugar_cookie <- rnorm(30, mean = 9.99, sd = 0.04) head(sugar_cookie)
- You can use a one-sample t-test to check whether the level of sugar is different than the recipe. You can draw a hypothesis test:
- H0: The average level of sugar is equal to 10
- H3: The average level of sugar is different than 10
- You use a significance level of 0.05.

- # H0 : mu = 10
- t.test(sugar_cookie, mu = 10)



- The p-value of the one sample t-test is 0.1079 and above 0.05.
- You can be confident at 95% that the amount of sugar added by the machine is between 9.973 and 10.002 grams.
- You cannot reject the null (H0) hypothesis. There is not enough evidence that amount of sugar added by the machine does not follow the recipe.

Two sample T-test

It is used to help us to understand that the difference between the two means is real or simply by chance.

```
shopOne <- rnorm(50, mean = 140, sd = 4.5)
shopTwo <- rnorm(50, mean = 150, sd = 4)
t.test(shopOne, shopTwo, var.equal = TRUE)
```

```
        Two Sample t-test

data:  shopOne and shopTwo
t = -13.094, df = 98, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal t
o 0
95 percent confidence interval:
 -11.136861  -8.205398
sample estimates:
mean of x mean of y
 140.2657  149.9368
```

You obtained a p-value is lower than the threshold of 0.05.

You conclude the averages of the two groups are significantly different.

Paired Sample T-test

- This is a statistical procedure that is used to determine whether the mean difference between two sets of observations is zero.
- In a paired sample t-test, each subject is measured two times, resulting in pairs of observations.
- sweetOne <- c(rnorm(100, mean = 14, sd = 0.3))
- sweetTwo <- c(rnorm(100, mean = 13, sd = 0.2))
- t.test(sweetOne, sweetTwo, paired = TRUE)

```
        Paired t-test

data:  sweetOne and sweetTwo
t = 27.192, df = 99, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal t
o 0
95 percent confidence interval:
 0.8995374 1.0411525
sample estimates:
mean of the differences
              0.9703449
```

Chi-Square test

- **Chi-Square test** is a statistical method to determine if two categorical variables have a significant correlation between them.
- Both those variables should be from same population and they should be categorical like − Yes/No, Male/Female, Red/Green etc.
- For example, we can build a data set with observations on people's ice-cream buying pattern and try to correlate the gender of a person with the flavor of the ice-cream they prefer. If a correlation is found we can plan for appropriate stock of flavors by knowing the number of gender of people visiting.

Syntax

chisq.test(data)

- **data** is the data in form of a table containing the count value of the variables in the observation.

- The null and alternative hypotheses are:
- H0 : the variables are independent, there is **no** relationship between the two categorical variables. Knowing the value of one variable does not help to predict the value of the other variable
- H1 : the variables are dependent, there is a relationship between the two categorical variables. Knowing the value of one variable helps to predict the value of the other variable

Example

- In the built-in data set survey, the **Smoke** column records the students smoking habit, while the **Exer** column records their exercise level.
- The allowed values in Smoke are "Heavy", "Regul" (regularly), "Occas" (occasionally) and "Never". As for Exer, they are "Freq" (frequently), "Some" and "None".
- We can tally the students smoking habit against the exercise level with the table function in R. The result is called the **contingency table** of the two variables.

```
library(MASS)
head(survey)
tbl = table(survey$Smoke, survey$Exer)
tbl
```

	Freq	None	Some
Heavy	7	1	3
Never	87	18	84
Occas	12	3	4
Regul	9	1	7

chisq.test(tbl)

```
        Pearson's Chi-squared test

data:  tbl
X-squared = 5.4885, df = 6, p-value = 0.4828

Warning message:
In chisq.test(tbl) : Chi-squared approximation may be incorrect
```

As the p-value 0.4828 is greater than the .05 significance level, we do not reject the null hypothesis that the smoking habit is independent of the exercise level of the students.

- The warning message found in the solution above is due to the small cell values in the contingency table.
- To avoid such warning, we combine the second and third columns of tbl, and save it in a new table named ctbl. Then we apply the chisq.test function against ctbl instead.

```
ctbl = cbind(tbl[,"Freq"], tbl[,"None"] + tbl[,"Some"])
ctbl
```

```
      [,1] [,2]
Heavy    7    4
Never   87  102
Occas   12    7
Regul    9    8
```

chisq.test(ctbl)

```
        Pearson's Chi-squared test

data:  ctbl
X-squared = 3.2328, df = 3, p-value = 0.3571
```

ANOVA

- **Analysis of Variance (**ANOVA) is a statistical test for estimating how a quantitative dependent variable changes according to the levels of one or more categorical independent variables.
- ANOVA tests whether there is a difference in means of the groups at each level of the independent variable.
- ANOVA test involves setting up:

  Null Hypothesis: All population mean are equal.

  Alternate Hypothesis: Atleast one population mean is different from other.
- ANOVA test are of two types:

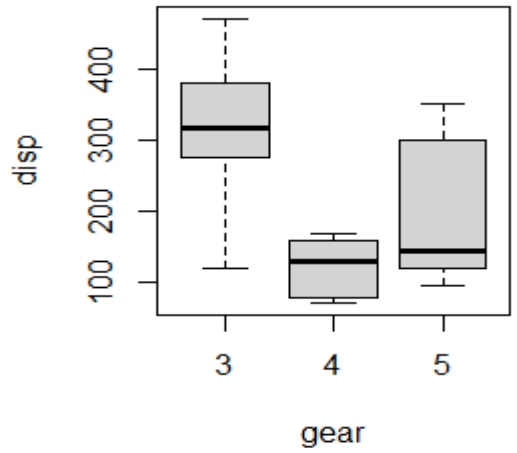
  One way ANOVA: It takes one categorical group into consideration.

  Two way ANOVA: It takes two categorical group into consideration.

Performing One Way ANOVA test

- One way ANOVA test is performed using mtcars dataset between disp attribute, a continuous attribute and gear attribute, a categorical attribute.

```
head(mtcars)
boxplot(mtcars$disp~factor(mtcars$gear),xlab = "gear", ylab = "disp")
```



```
# Step 1: Setup Null Hypothesis and Alternate Hypothesis
# H0 = mu = mu01 = mu02(There is no difference
# between average displacement for different gear)
# H1 = Not all means are equal
# Step 2: Calculate test statistics using aov function
mtcars_aov <- aov(mtcars$disp~factor(mtcars$gear))
summary(mtcars_aov)
```

```
                    Df Sum Sq Mean Sq F value   Pr(>F)
factor(mtcars$gear)  2 280221  140110   20.73 2.56e-06 ***
Residuals           29 195964    6757
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

- The summary shows that gear attribute is very significant to displacement(Three stars denoting it).
- Also, P value less than 0.05, so it proves that gear is significant to displacement i.e related to each other and we reject the Null Hypothesis.

Performing Two Way ANOVA test

- Two way ANOVA test is performed using mtcars dataset between disp attribute, a continuous attribute and gear attribute, a categorical attribute, am attribute, a categorical attribute.

```
mtcars_aov2 <- aov(mtcars$disp~factor(mtcars$gear)* factor(mtcars$am))
summary(mtcars_aov2)
```

```
                     Df Sum Sq Mean Sq F value   Pr(>F)
factor(mtcars$gear)   2 280221  140110  20.695 3.03e-06 ***
factor(mtcars$am)     1   6399    6399   0.945    0.339
Residuals            28 189565    6770
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

- The summary shows that gear attribute is very significant to displacement(Three stars denoting it) and am attribute is not much significant to displacement.
- P-value of gear is less than 0.05, so it proves that gear is significant to displacement i.e related to each other.
- P-value of am is greater than 0.05, am is not significant to displacement i.e not related to each other.

UNIT V
ADVANCED METHODS

- Advanced methods for missing data-Steps in dealing with missing data-Identifying missing values-Exploring missing value patterns-Understanding the sources and impact of missing data-rational Approaches for dealing with incomplete data-Lit wise deletion-Multiple Imputation-Advanced Graphics-Lattice Package-ggPlot2 Package-Interactive graphs.

- Data can be missing for many reasons.
- Survey participants may forget to answer one or more questions, refuse to answer sensitive questions, or grow fatigued and fail to complete a long questionnaire.
- Recording equipment may fail, internet connections may be lost, and data may be miscoded.
- Most statistical methods assume that you're working with complete matrices, vectors, and data frames. In most cases, you have to eliminate missing data before you address the substantive questions that led you to collect the data.
- You can eliminate missing data by

(1) removing cases with missing data, or

(2) replacing missing data with reasonable substitute values.

- In either case, the end result is a dataset without missing values.

Steps in dealing with missing data

1. Identify the missing data.
2. Examine the causes of the missing data.
3. Delete the cases containing missing data or replace (impute) the missing values with reasonable alternative data values.

A classification system for missing data

Missing completely at random

- If the presence of missing data on a variable is unrelated to any other observed or unobserved variable, then the data are missing completely at random (MCAR).

Missing at random

•If the presence of missing data on a variable is related to other observed variables but not to its own unobserved value, the data is missing at random (MAR).

Not missing at random

- If the missing data for a variable is neither MCAR nor MAR, it is not missing at random (NMAR).

Methods for handling Incomplete Data

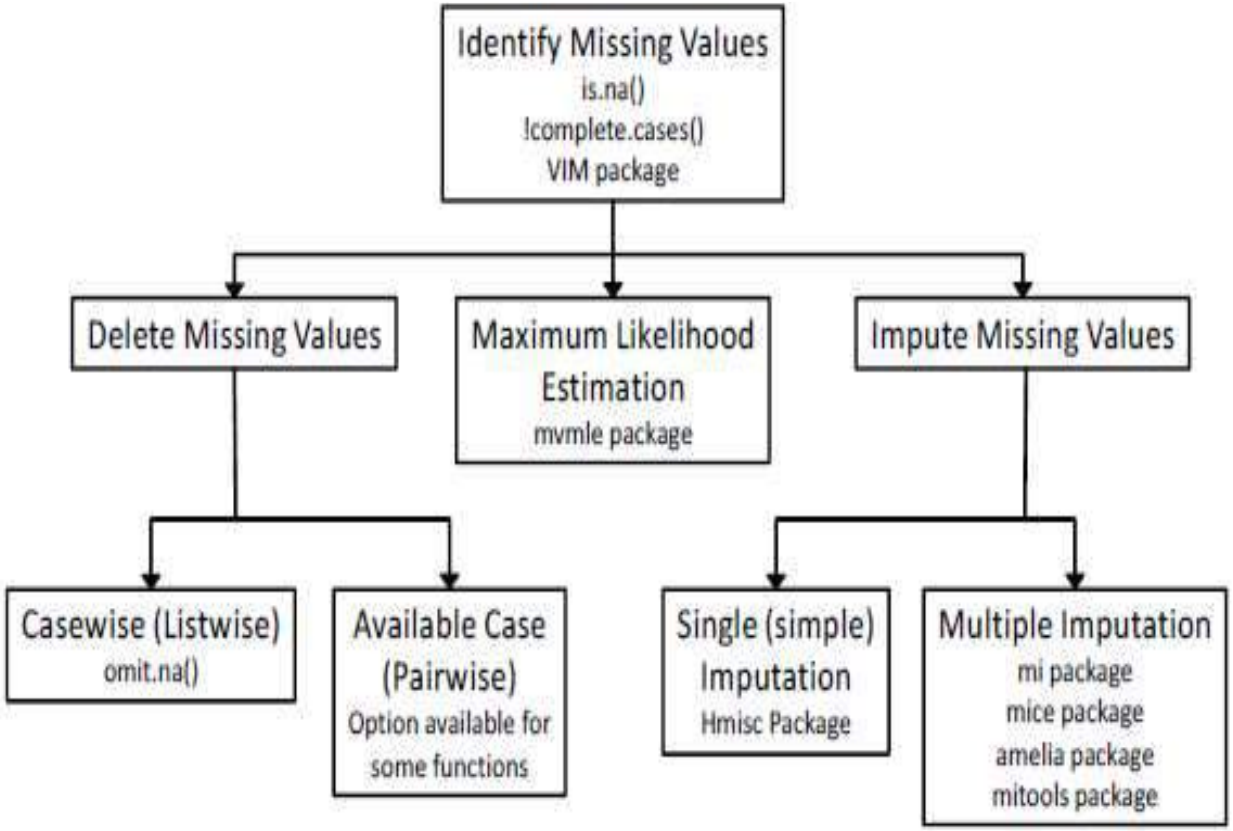


Figure 15.1 **Methods for handling incomplete data, along with the R packages that support them**

Identifying missing values

- R represents missing values using the symbol NA (not available) and impossible values by the symbol NaN (not a number).
- In addition, the symbols Inf and -Inf represent positive infinity and negative infinity, respectively.
- The functions is.na(), is.nan(), and is.infinite() can be used to identify missing, impossible, and infinite values respectively.
- Each returns either TRUE or FALSE.

x	is.na(x)	is.nan(x)	is.infinite(x)
x <- NA	TRUE	FALSE	FALSE
x <- 0 / 0	TRUE	TRUE	FALSE
x <- 1 / 0	FALSE	FALSE	TRUE

Missing Data - Example

- let y <- c(1, 2, 3, NA).
- Then is.na(y) will return the vector

OUTPUT:

c(FALSE, FALSE, FALSE, TRUE).

- The function return an object that's the same size as its argument, with each element replaced by TRUE if the element is of the type being tested, and FALSE otherwise.

Missing Data - Example

- mammal sleep dataset (sleep) provided in the VIM package
- The data come from a study by Allison and Chichetti (1976) that examined the relationship between sleep, ecological, and constitutional variables for 62 mammal species.
- Sleep variables included length of dreaming sleep (Dream), nondreaming sleep (NonD), and their sum (Sleep).
- The constitutional variables included body weight in kilograms (BodyWgt), brain weight in grams (BrainWgt), life span in years (Span),and gestation time in days (Gest).
- The ecological variables included degree to which species were preyed upon (Pred), degree of their exposure while sleeping (Exp), and overall danger (Danger) faced.
- The ecological variables were measured on 5-point rating scales that ranged from 1 (low) to 5 (high).

- The function complete.cases() can be used to identify the rows in a matrix or data frame that don’t contain missing data.
- It returns a logical vector with TRUE for every row that contains complete cases and FALSE for every row that has one or more missing values.

```
Library(VIM)
data(sleep,package="VIM")
head(sleep)
# list the rows that do not have missing values
sleep[complete.cases(sleep),]
# list the rows that have one or more missing values
sleep[!complete.cases(sleep),]
```

- Examining the output reveals that 42 cases have complete data and 20 cases have one or more missing values.
- Because the logical values TRUE and FALSE are equivalent to the numeric values 1 and 0, the sum() and mean() functions can be used to obtain useful information about missing data.
- Consider the following:

```
 sum(is.na(sleep$Dream))
[1] 12
 mean(is.na(sleep$Dream))
[1] 0.19
 mean(!complete.cases(sleep))
[1] 0.32
```

- The results indicate that there are 12 missing values for the variable Dream. Nineteen percent of the cases have a missing value on this variable.
- In addition, 32 percent of the cases in the dataset contain one or more missing values.

Exploring missing values patterns

- To determine which variables have missing values, in what amounts, and in what combinations, tabular, graphical, and correlational methods are used.

Tabulating missing values

- The md.pattern() function in the mice package will produce a tabulation of the missing data patterns in a matrix or data frame.
- Apply this function to the sleep dataset:

```
library(mice)
data(sleep, package="VIM")
md.pattern(sleep)
```

Tabulating missing values

	BodyWgt	BrainWgt	Pred	Exp	Danger	Sleep	Span	Gest	Dream	NonD
42	1	1	1	1	1	1	1	1	1	1	0
2	1	1	1	1	1	1	0	1	1	1	1
3	1	1	1	1	1	1	1	0	1	1	1
9	1	1	1	1	1	1	1	1	0	0	2
2	1	1	1	1	1	0	1	1	1	0	2
1	1	1	1	1	1	1	0	0	1	1	2
2	1	1	1	1	1	0	1	1	0	0	3
1	1	1	1	1	1	1	0	1	0	0	3
0	0	0	0	0	4	4	4	12	14	3	8

Tabulating missing values

- The 1's and 0's in the body of the table indicate the missing values patterns, with a 0 indicating a missing value for a given column variable and a 1 indicating a non missing value.
- The first row describes the pattern of "no missing values" (all elements are 1).
- The second row describes the pattern "no missing values except for Span."
- The first column indicates the number of cases in each missing data pattern, and the last column indicates the number of variables with missing values present in each pattern.

Tabulating missing values

- There are 42 cases without missing data and 2 cases that are missing Span alone.
- Nine cases are missing both NonD and Dream values.
- The dataset contains a total of (42 x 0) + (2 x 1) + … + (1 x 3) = 38 missing values.
- The last row gives the total number of missing values present on each variable.

Exploring missing data visually

- The VIM package provides numerous functions for visualizing missing values patterns in datasets.
- The aggr() function plots the number of missing values for each variable alone and for each combination of variables.

```
library("VIM")
aggr(sleep, prop=FALSE, numbers=TRUE)
```

Exploring missing data visually - aggr

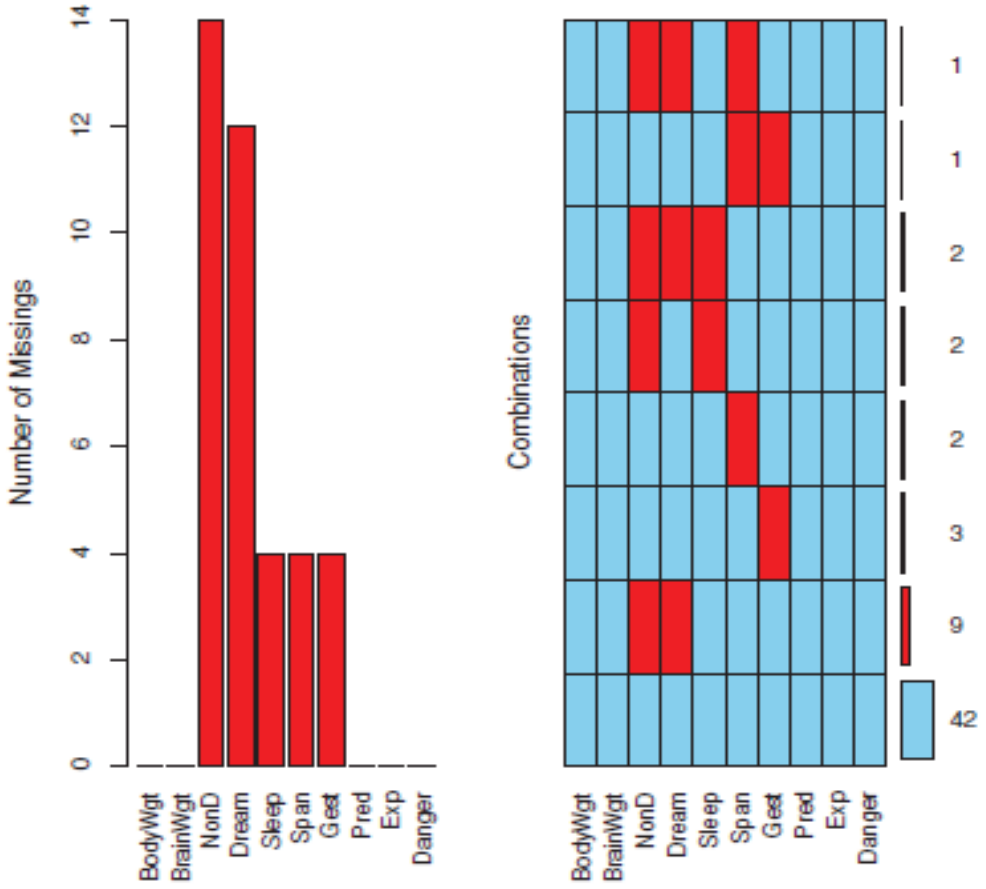


Figure 15.2 `aggr()` **produced plot of missing values patterns for the sleep dataset.**

Exploring missing data visually

- From the above graph the variable NonD has the largest number of missing values (14), and that 2 mammals are missing NonD, Dream, and Sleep scores.
- Forty-two mammals have no missing data.

Exploring missing data visually

- The statement aggr(sleep, prop=TRUE, numbers=TRUE) produces the same plot, but proportions are displayed instead of counts.
- The option numbers=FALSE (the default) suppresses the numeric labels.

matrixplot() function

- It produces a plot displaying the data for each case.
- Here, the numeric data is rescaled to the interval [0, 1] and represented by grayscale colors, with lighter colors representing lower values and darker colors representing larger values.
- By default, missing values are represented in red.
- A matrix plot allows you to see if the presence of missing values on one or more variables is related to the actual values of other variables.
- Here, you can see that there are no missing values on sleep variables (Dream, NonD, Sleep) for low values of body or brain weight (BodyWgt, BrainWgt).

matrixplot() function

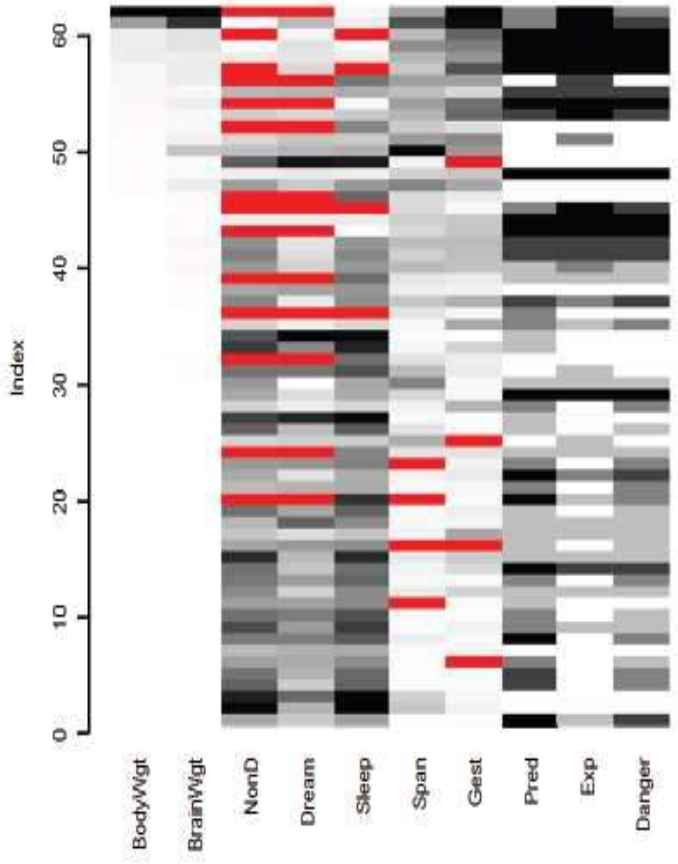


Figure 15.3 Matrix plot of actual and missing values by case (row) for the sleep dataset. The matrix is sorted by `BodyWgt`.

marginplot() function

- The marginplot() function produces a scatter plot between two variables with information about missing values shown in the plot’s margins.
- Consider the relationship between amount of dream sleep and the length of a mammal’s gestation.

```
marginplot(sleep[c("Gest","Dream")], pch=c(20),
col=c("darkgray", "red", "blue"))
```

marginplot() function

- The body of the graph displays the scatter plot between Gest and Dream (based on complete cases for the two variables).

In the left margin, box plots display the distribution of Dream for mammals with (dark gray) and without (red) Gest values.

- In grayscale, red is the darker shade.
- Four red dots represent the values of Dream for mammals missing Gest scores.
- In the bottom margin, the roles of Gest and Dream are reversed.
- A negative relationship exists between length of gestation and dream sleep and that dream sleep tends to be higher for mammals that are missing a gestation score.
- The number of observations with missing values on both variables at the same time is printed in blue at the intersection of both margins (bottom left).

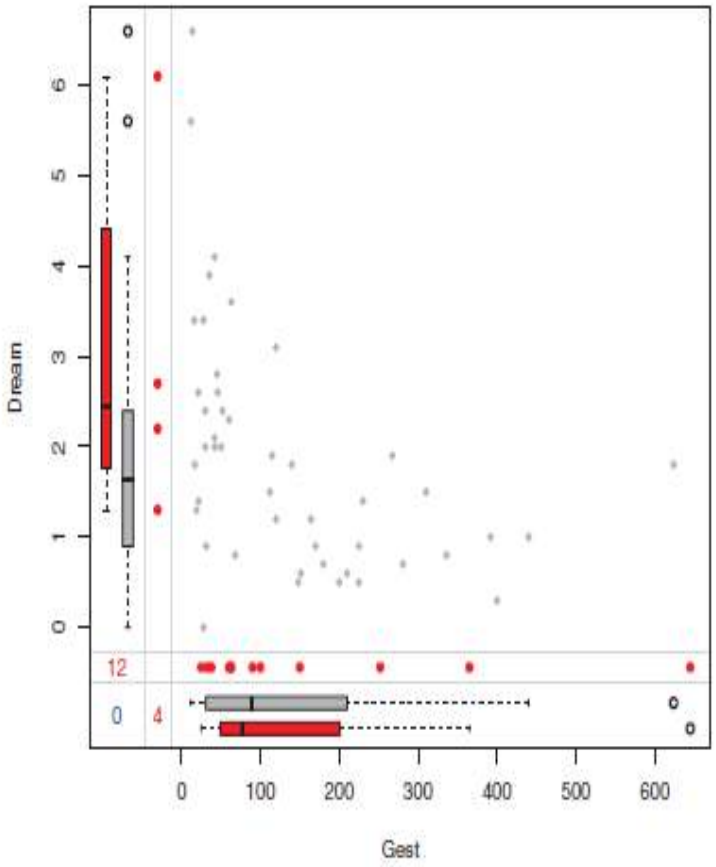


Figure 15.4 Scatter plot between amount of dream sleep and length of gestation, with information about missing data in the margins

Using correlations to explore missing values

- can replace the data in a dataset with indicator variables, coded 1 for missing and 0 for present.
- The resulting matrix is sometimes called a *shadow matrix*.
- Correlating these indicator variables with each other and with the original (observed) variables can help you to see which variables tend to be missing together, as well as relationships between a variable's "missingness" and the values of the other variables.

Using correlations to explore missing values

x <- as.data.frame(abs(is.na(sleep)))

The elements of data frame x are 1 if the corresponding element of sleep is missing and 0 otherwise.

head(sleep, n=5)

	BodyWgt	BrainWgt	NonD	Dream	Sleep	Span	Gest	Pred	Exp	Danger
1	6654.000	5712.0	NA	NA	3.3	38.6	645	3	5	3
2	1.000	6.6	6.3	2.0	8.3	4.5	42	3	1	3
3	3.385	44.5	NA	NA	12.5	14.0	60	1	1	1
4	0.920	5.7	NA	NA	16.5	NA	25	5	2	3
5	2547.000	4603.0	2.1	1.8	3.9	69.0	624	3	5	4

```
head(x, n=5)
```

	BodyWgt	BrainWgt	NonD	Dream	Sleep	Span	Gest	Pred	Exp	Danger
1	0	0	1	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	1	1	0	0	0	0	0	0
4	0	0	1	1	0	1	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0

Using correlations to explore missing values

The statement

y <- x[which(sd(x) > 0)]

extracts the variables that have some (but not all) missing values, and

cor(y)

gives you the correlations among these indicator variables:

	NonD	Dream	Sleep	Span	Gest
NonD	1.000	0.907	0.486	0.015	-0.142
Dream	0.907	1.000	0.204	0.038	-0.129
Sleep	0.486	0.204	1.000	-0.069	-0.069
Span	0.015	0.038	-0.069	1.000	0.198
Gest	-0.142	-0.129	-0.069	0.198	1.000

Using correlations to explore missing values

- can see that Dream and NonD tend to be missing together (r=0.91).
- To a lesser extent, Sleep and NonD tend to be missing together (r=0.49) and Sleep and Dream tend to be missing together (r=0.20).
- The relationship between the presence of missing values in a variable and the observed values on other variables:

```
cor(sleep, y, use="pairwise.complete.obs")
```

Using correlations to explore missing values

```
            NonD  Dream   Sleep   Span   Gest
BodyWgt    0.227  0.223  0.0017 -0.058 -0.054
BrainWgt   0.179  0.163  0.0079 -0.079 -0.073
NonD          NA     NA      NA -0.043 -0.046
Dream      -0.189     NA -0.1890  0.117  0.228
Sleep      -0.080 -0.080      NA  0.096  0.040
Span        0.083  0.060  0.0052     NA -0.065
Gest        0.202  0.051  0.1597 -0.175     NA
Pred        0.048 -0.068  0.2025  0.023 -0.201
Exp         0.245  0.127  0.2608 -0.193 -0.193
Danger      0.065 -0.067  0.2089 -0.067 -0.204
Warning message:
In cor(sleep, y, use = "pairwise.complete.obs") :
  the standard deviation is zero
```

Using correlations to explore missing values

- In this correlation matrix, the rows are observed variables, and the columns are indicator variables representing missingness.
- can ignore the warning message and NA values in the correlation matrix.
- From the first column of the correlation matrix, can see that non dreaming sleep scores are more likely to be missing for mammals with higher body weight (r=0.227),gestation period (r=0.202), and sleeping exposure (0.245).

Understanding the sources and impact of missing data

- Identify the amount, distribution, and pattern of missing data in order to evaluate

(1) the potential mechanisms producing the missing data

(2) the impact of the missing data on ability to answer substantive questions.

The following questions:

■ What percentage of the data is missing?

■ Is it concentrated in a few variables, or widely distributed?

■ Does it appear to be random?

■ Does the covariation of missing data with each other or with observed data suggest a possible mechanism that's producing the missing values?

Understanding the sources and impact of missing data

- If the missing data are concentrated in a few relatively unimportant variables, you may be able to delete these variables and continue your analyses normally.
- If there's a small amount of data (say less than 10 percent) that's randomly distributed throughout the dataset (MCAR), you may be able to limit your analyses to cases with complete data and still get reliable and valid results.
- If you can assume that the data are either MCAR or MAR, you may be able to apply multiple imputation methods to arrive at valid conclusions.
- If the data are NMAR, you can turn to specialized methods, collect new data, or go into an easier and more rewarding profession.

Rational approaches for dealing with incomplete data

- In a rational approach, use mathematical or logical relationships among variables to attempt to fill in or recover the missing values.

Examples :

- In the sleep dataset, the variable Sleep is the sum of the Dream and NonD variables.
- If you know a mammal's scores on any two, you can derive the third.
- Thus, if there were some observations that were missing only one of the three variables, you could recover the missing information through addition or subtraction.

Logical relationships - Example

- logical relationships to recover missing data comes from a set of leadership studies in which participants were asked if they were a manager (yes/no) and the number of their direct reports (integer).
- If they left the manager question blank but indicated that they had one or more direct reports, it would be reasonable to infer that they were a manager.

Complete-case analysis (listwise deletion)

- In complete-case analysis, only observations containing valid data values on every variable are retained for further analysis.
- This involves deleting any row containing one or more missing values, and is also known as listwise, or case-wise, deletion.
- Most popular statistical packages employ listwise deletion as the default approach for handling missing data.
- it’s so common in carrying out analyses like regression or ANOVA where there’s a “missing values problem” to be dealt with.

Complete-case analysis (listwise deletion)

- The function complete.cases() can be used to save the cases (rows) of a matrix or data frame without missing data:

  newdata <- mydata[complete.cases(mydata),]

- The same result can be accomplished with the na.omit function:

  newdata <- na.omit(mydata)

Complete-case analysis (listwise deletion)

- In both statements, any rows containing missing data are deleted from mydata before the results are saved to newdata.
- To find the correlations among the variables in the sleep study.
- Applying listwise deletion, delete all mammals with missing data prior to calculating the correlations:

```
cor(na.omit(sleep))
```

Complete-case analysis (listwise deletion)

	BodyWgt	BrainWgt	NonD	Dream	Sleep	Span	Gest	Pred	Exp	Danger
BodyWgt	1.00	0.96	-0.4	-0.07	-0.3	0.47	0.71	0.10	0.4	0.26
BrainWgt	0.96	1.00	-0.4	-0.07	-0.3	0.63	0.73	-0.02	0.3	0.15
NonD	-0.39	-0.39	1.0	0.52	1.0	-0.37	-0.61	-0.35	-0.6	-0.53
Dream	-0.07	-0.07	0.5	1.00	0.7	-0.27	-0.41	-0.40	-0.5	-0.57
Sleep	-0.34	-0.34	1.0	0.72	1.0	-0.38	-0.61	-0.40	-0.6	-0.60
Span	0.47	0.63	-0.4	-0.27	-0.4	1.00	0.65	-0.17	0.3	0.01
Gest	0.71	0.73	-0.6	-0.41	-0.6	0.65	1.00	0.09	0.6	0.31
Pred	0.10	-0.02	-0.4	-0.40	-0.4	-0.17	0.09	1.00	0.6	0.93
Exp	0.41	0.32	-0.6	-0.50	-0.6	0.32	0.57	0.63	1.0	0.79
Danger	0.26	0.15	-0.5	-0.57	-0.6	0.01	0.31	0.93	0.8	1.00

Complete-case analysis (listwise deletion)

- The correlations in this table are based solely on the 42 mammals that have complete data on all variables.

```
cor(sleep, use="complete.obs")
```

- To study the impact of life span and length of gestation on the amount of dream sleep, employ linear regression with listwise deletion:

```
fit <- lm(Dream ~ Span + Gest, data=na.omit(sleep))
summary(fit)
```

OUTPUT

Call:

lm(formula = Dream ~ Span + Gest, data = na.omit(sleep))

Residuals:

Min 1Q Median 3Q Max

-2.333 -0.915 -0.221 0.382 4.183

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 2.480122 0.298476 8.31 3.7e-10 ***

Span -0.000472 0.013130 -0.04 0.971

Gest -0.004394 0.002081 -2.11 0.041 *

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ‘ 1

Residual standard error: 1 on 39 degrees of freedom

Multiple R-squared: 0.167, Adjusted R-squared: 0.125

F-statistic: 3.92 on 2 and 39 DF, p-value: 0.0282

- mammals with shorter gestation periods have more dream sleep
- (controlling for life span) and that life span is unrelated to dream sleep when controlling for gestation period. The analysis is based on 42 cases with complete data.

Complete-case analysis (listwise deletion)

- Listwise deletion assumes that the data are MCAR (that is, the complete observations are a random subsample of the full dataset).
- In the current example, we've assumed that the 42 mammals used are a random subsample of the 62 mammals collected.
- To the degree that the MCAR assumption is violated, the resulting regression parameters will be biased. Deleting all observations with missing data can also reduce statistical power by reducing the available sample size.
- In the example above, listwise deletion reduced the sample size by 32 percent.

Multiple imputation

- Multiple imputation (MI) provides an approach to missing values that's based on repeated simulations.
- MI is frequently the method of choice for complex missing values problems.
- In MI, a set of complete datasets (typically 3 to 10) is generated from an existing dataset containing missing values.
- Monte Carlo methods are used to fill in the missing data in each of the simulated datasets.
- Standard statistical methods are applied to each of the simulated datasets, and the outcomes are combined to provide estimated results and confidence intervals that take into account the uncertainty introduced by the missing values.
- Good implementations are available in R through the Amelia, mice, and mi packages.

Multiple imputation

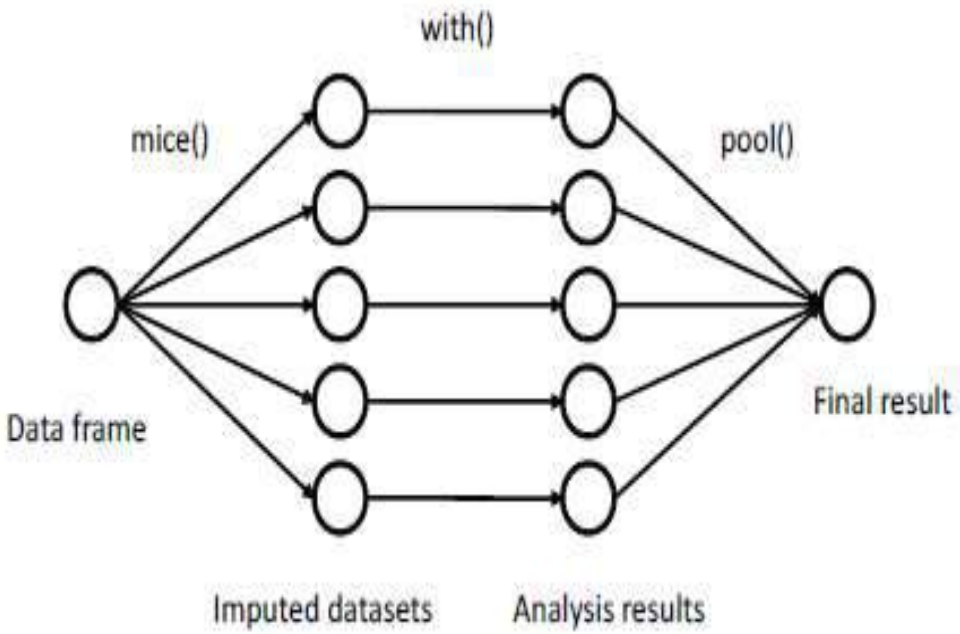


Figure 15.5 **Steps in applying multiple imputation to missing data via the `mice` approach.**

Multiple imputation

- The function mice() starts with a data frame containing missing data and returns an object containing several complete datasets (the default is 5).
- Each complete dataset is created by imputing values for the missing data in the original data frame.
- There's a random component to the imputations, so each complete dataset is slightly different.
- The with() function is then used to apply a statistical model (for example, linear or generalized linear model) to each complete dataset in turn.
- Finally, the pool() function combines the results of these separate analyses into a single set of results.
- The standard errors and p-values in this final model correctly reflect the uncertainty produced by both the missing values and the multiple imputations.

Multiple imputation

- An analysis based on the mice package will typically conform to the following structure:

```
library(mice)
imp <- mice(mydata, m)
fit <- with(imp, analysis)
pooled <- pool(fit)
summary(pooled)
```

Multiple imputation

- mydata is a matrix or data frame containing missing values.
- imp is a list object containing the *m* imputed datasets, along with information on how the imputations were accomplished. By default, *m*=5.
- *analysis* is a formula object specifying the statistical analysis to be applied to each of the *m* imputed datasets. Examples include lm() for linear regression models, glm() for generalized linear models, gam() for generalized additive models, and nbrm() for negative binomial models. Formulas within the parentheses give the response variables on the left of the ~ and the predictor variables (separated by + signs) on the right.
- fit is a list object containing the results of the *m* separate statistical analyses.
- pooled is a list object containing the averaged results of these *m* statistical analyses.

Multiple imputation

- apply multiple imputation to our sleep dataset.
- use all 62 mammals.

```
 library(mice)
data(sleep, package="VIM")
imp <- mice(sleep, seed=1234)
fit <- with(imp, lm(Dream ~ Span + Gest))
pooled <- pool(fit)
summary(pooled)
```

Multiple imputation

	est	se	t	df	Pr(>|t|)	lo 95
(Intercept)	2.58858	0.27552	9.395	52.1	8.34e-13	2.03576
Span	-0.00276	0.01295	-0.213	52.9	8.32e-01	-0.02874
Gest	-0.00421	0.00157	-2.671	55.6	9.91e-03	-0.00736

	hi 95	nmis	fmi
(Intercept)	3.14141	NA	0.0870
Span	0.02322	4	0.0806
Gest	-0.00105	4	0.0537

Multiple imputation

- can access more information about the imputation by examining the objects created in the analysis.
- example, view a summary of the imp object:

imp

OUTPUT

Multiply imputed data set
Call:
mice(data = sleep, seed = 1234)
Number of multiple imputations: 5
Missing cells per column:
BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred
0 0 14 12 4 4 4 0
Exp Danger
0 0
Imputation methods:
BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred
"" "" "pmm" "pmm" "pmm" "pmm" "pmm" ""
Exp Danger
"" ""
VisitSequence:
NonD Dream Sleep Span Gest
3 4 5 6

```
PredictorMatrix:
         BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger
BodyWgt        0        0    0     0     0    0    0    0   0      0
BrainWgt       0        0    0     0     0    0    0    0   0      0
NonD           1        1    0     1     1    1    1    1   1      1
Dream          1        1    1     0     1    1    1    1   1      1
Sleep          1        1    1     1     0    1    1    1   1      1
Span           1        1    1     1     1    0    1    1   1      1
Gest           1        1    1     1     1    1    0    1   1      1
Pred           0        0    0     0     0    0    0    0   0      0
Exp            0        0    0     0     0    0    0    0   0      0
Danger         0        0    0     0     0    0    0    0   0      0
Random generator seed value:  1234
```

Multiple imputation

- From the resulting output, you can see that five synthetic datasets were created, and that the predictive mean matching (pmm) method was used for each variable with missing data.
- No imputation ("") was needed for BodyWgt, BrainWgt, Pred, Exp, or Danger, because they had no missing values.
- The Visit Sequence tells you that variables were imputed from right to left, starting with NonD and ending with Gest.
- Finally, the Predictor Matrix indicates that each variable with missing data was imputed using all the other variables in the dataset.

Multiple imputation

- imp$Dream
- displays the five imputed values for each of the 12 mammals with missing data on the Dream variable.

	1	2	3	4	5
1	0.5	0.5	0.5	0.5	0.0
3	2.3	2.4	1.9	1.5	2.4
4	1.2	1.3	5.6	2.3	1.3
14	0.6	1.0	0.0	0.3	0.5
24	1.2	1.0	5.6	1.0	6.6
26	1.9	6.6	0.9	2.2	2.0
30	1.0	1.2	2.6	2.3	1.4
31	5.6	0.5	1.2	0.5	1.4
47	0.7	0.6	1.4	1.8	3.6
53	0.7	0.5	0.7	0.5	0.5
55	0.5	2.4	0.7	2.6	2.6
62	1.9	1.4	3.6	5.6	6.6

Multiple imputation

- can view each of the *m* imputed datasets via the complete() function. The format is

  complete(imp, action=#)

- where # specifies one of the *m* synthetically complete datasets.

```
dataset3 <- complete(imp, action=3)
dataset3
```

	BodyWgt	BrainWgt	NonD	Dream	Sleep	Span	Gest	Pred	Exp	Danger
1	6654.00	5712.0	2.1	0.5	3.3	38.6	645	3	5	3
2	1.00	6.6	6.3	2.0	8.3	4.5	42	3	1	3
3	3.38	44.5	10.6	1.9	12.5	14.0	60	1	1	1
4	0.92	5.7	11.0	5.6	16.5	4.7	25	5	2	3
5	2547.00	4603.0	2.1	1.8	3.9	69.0	624	3	5	4
6	10.55	179.5	9.1	0.7	9.8	27.0	180	4	4	4

[…output deleted to save space…]

Other approaches to missing data

- R supports several other approaches for dealing with missing data.
- There are two methods for dealing with missing data

(1) pairwise deletion

(2) simple imputation.

Table 15.2 Specialized methods for dealing with missing data

Package	Description
`Hmisc`	Contains numerous functions supporting simple imputation, multiple imputation, and imputation using canonical variates
`mvnmle`	Maximum likelihood estimation for multivariate normal data with missing values
`cat`	Multiple imputation of multivariate categorical data under log-linear models
`arrayImpute`, `arrayMissPattern`, `SeqKnn`	Useful functions for dealing with missing microarray data
`longitudinalData`	Contains utility functions, including interpolation routines for imputing missing time series values
`kmi`	Kaplan–Meier multiple imputation for survival analysis with missing data
`mix`	Multiple imputation for mixed categorical and continuous data under the general location model
`pan`	Multiple imputation for multivariate panel or clustered data

Pairwise deletion

- Pairwise deletion is often considered an alternative to listwise deletion when working with datasets containing missing values.
- In pairwise deletion, observations are only deleted if they're missing data for the variables involved in a specific analysis.
- Consider the following code:

```
cor(sleep, use="pairwise.complete.obs")
```

	BodyWgt	BrainWgt	NonD	Dream	Sleep	Span	Gest	Pred	Exp	Danger
BodyWgt	1.00	0.93	-0.4	-0.1	-0.3	0.30	0.7	0.06	0.3	0.13
BrainWgt	0.93	1.00	-0.4	-0.1	-0.4	0.51	0.7	0.03	0.4	0.15
NonD	-0.38	-0.37	1.0	0.5	1.0	-0.38	-0.6	-0.32	-0.5	-0.48
Dream	-0.11	-0.11	0.5	1.0	0.7	-0.30	-0.5	-0.45	-0.5	-0.58
Sleep	-0.31	-0.36	1.0	0.7	1.0	-0.41	-0.6	-0.40	-0.6	-0.59
Span	0.30	0.51	-0.4	-0.3	-0.4	1.00	0.6	-0.10	0.4	0.06
Gest	0.65	0.75	-0.6	-0.5	-0.6	0.61	1.0	0.20	0.6	0.38
Pred	0.06	0.03	-0.3	-0.4	-0.4	-0.10	0.2	1.00	0.6	0.92
Exp	0.34	0.37	-0.5	-0.5	-0.6	0.36	0.6	0.62	1.0	0.79
Danger	0.13	0.15	-0.5	-0.6	-0.6	0.06	0.4	0.92	0.8	1.00

Simple (nonstochastic) imputation

- In simple imputation, the missing values in a variable are replaced with a single value (for example, mean, median, or mode).
- Using *mean substitution* you could replace missing values on Dream with the value 1.97 and missing values on NonD with the value 8.67 (the means on Dream and NonD, respectively).
- The substitution is nonstochastic, meaning that random error isn't introduced (unlike multiple imputation).
- An advantage to simple imputation is that it solves the "missing values problem" without reducing the sample size available for the analyses.

Simple imputation is simple, but it produces biased results for data that aren't MCAR.

- If there are moderate to large amounts of missing data, simple imputation is likely to underestimate standard errors, distort correlations among variables, and produce incorrect p-values in statistical tests.

Advanced Graphics

The four graphic systems in R

- The **base graphic system** in R, written by Ross Ihaka, is included in every R installation. Most of the graphs produced in previous chapters rely on base graphics functions.
- The **grid graphics system**, written by Paul Murrell (2006), is implemented through the grid package. Grid graphics offer a lower-level alternative to the standard graphics system. The user can create arbitrary rectangular regions on graphics devices, define coordinate systems for each region, and use a rich set of drawing primitives to control the arrangement and appearance of graphic elements.

- The **lattice package** , written by Deepayan Sarkar (2008), implements trellis graphics as outlined by Cleveland (1985, 1993). Built using the grid package, the lattice package has grown beyond Cleveland's original approach to visualizing multivariate data, and now provides a comprehensive alternative system for creating statistical graphics in R.
- The **ggplot2 package** , written by Hadley Wickham (2009a), provides a system for creating graphs based on the grammar of graphics described by Wilkinson (2005) and expanded by Wickham (2009b). The intention of the ggplot2 package is to provide a comprehensive, grammar-based system for generating graphs in a unified and coherent manner, allowing users to create new and innovative data visualizations.

- Base graphic functions are automatically available.
- To access grid and lattice functions, you must load the package explicitly (for example, library(lattice)) .
- To access ggplot2 functions, install the package (install.packages("ggplot2")).

Access to Graphic system

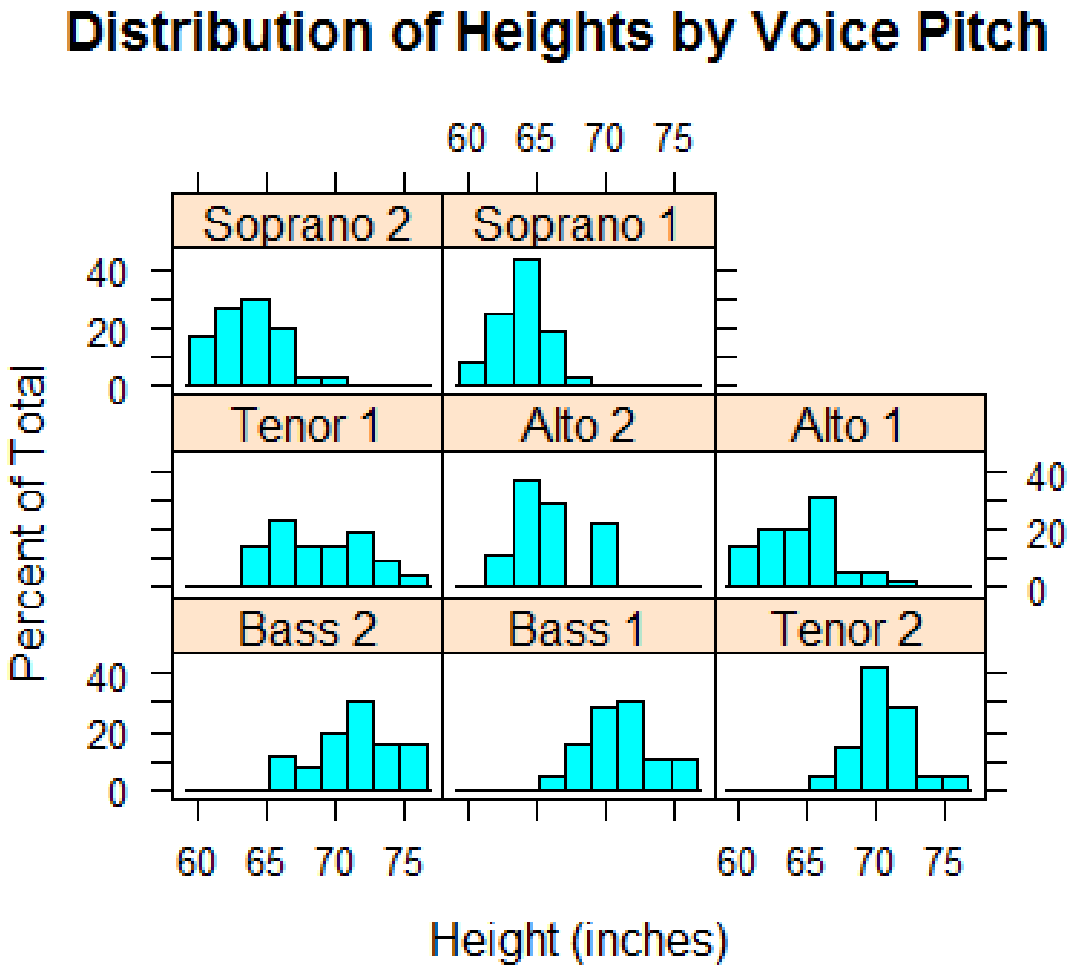
System	Included in base installation?	Must be explicitly loaded?
base	Yes	No
grid	Yes	Yes
lattice	Yes	Yes
ggplot2	No	Yes

The lattice package

- The lattice package provides a comprehensive graphical system for visualizing univariate and multivariate data.
- In particular, many users turn to the lattice package because of its ability to easily generate trellis graphs.
- A trellis graph displays the distribution of a variable or the relationship between variables, separately for each level of one or more other variables.
- Consider the following question: *How do the heights of singers in the New York Choral Society vary by their vocal parts?*
- Data on the heights and voice parts of choral members is provided in the singer dataset contained in the lattice package.

```
library(lattice)
histogram(~height| voice.part, data = singer,          main="Distribution
    of Heights by Voice Pitch",  xlab="Height (inches)")
```

- height is the dependent variable, voice.part is called the *conditioning variable*, *and* a histogram is created for each of the eight voice parts. It appears that tenors and basses tend to be taller than altos and sopranos.


Distribution of Heights by Voice Pitch
Soprano 2
Soprano 1
Tenor 1
Alto 2
Alto 1
Bass 2
Bass 1
Tenor 2
Percent of Total
Height (inches)
0
20
40
60
65
70
75

- In trellis graphs, a separate panel is created for each level of the conditioning variable.
- If more than one conditioning variable is specified, a panel is created for each combination of factor levels. The panels are arranged into an array to facilitate comparisons. A label is provided for each panel in an area called the strip.
- The user has control over the graph displayed in each panel, the format and placement of the strip, the arrangement of the panels, the placement and content of legends, and many other graphic features.
- The lattice package provides a wide variety of functions for producing univariate (dot plots, kernel density plots, histograms, bar charts, box plots), bivariate (scatter plots, strip plots, parallel box plots), and multivariate (3D plots, scatter plot matrices) graphs.

Each high-level graphing function follows the format

graph_function(formula, data=, options)

where:

- graph_function is one of the functions listed in the second column of table.
- formula specifies the variable(s) to display and any conditioning variables.
- data specifies a data frame.
- options are comma-separated parameters used to modify the content, arrangement, and annotation of the graph.

Graph types and corresponding functions in the lattice package

Graph type	Function	Formula examples
3D contour plot	`contourplot()`	`z~x*y`
3D level plot	`levelplot()`	`z~y*x`
3D scatter plot	`cloud()`	`z~x*y|A`
3D wireframe graph	`wireframe()`	`z~y*x`
Bar chart	`barchart()`	`x~A` or `A~x`
Box plot	`bwplot()`	`x~A` or `A~x`
Dot plot	`dotplot()`	`~x|A`
Histogram	`histogram()`	`~x`
Kernel density plot	`densityplot()`	`~x|A*B`
Parallel coordinates plot	`parallel()`	`dataframe`
Scatter plot	`xyplot()`	`y~x|A`
Scatter plot matrix	`splom()`	`dataframe`
Strip plots	`stripplot()`	`A~x` or `x~A`

- Let lowercase letters represent numeric variables and uppercase letters represent categorical variables (factors). The formula in a high-level graphing function typically takes the form

  y ~ x | A * B

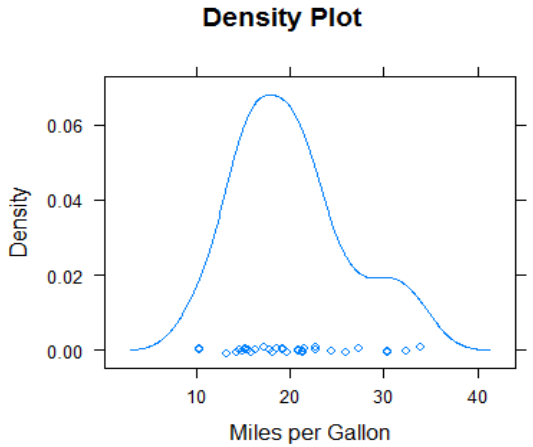
- where variables on the left side of the vertical bar are called the primary variables and variables on the right are the conditioning variables.
- Primary variables map variables to the axes in each panel. Here, y~x describes the variables to place on the vertical and horizontal axes, respectively. For single-variable plots, replace y~x with ~x.
- For 3D plots, replace y~x with z~x*y.
- For multivariate plots (scatter plot matrix or parallel coordinates plot) replace y~x with a data frame.
- Note that conditioning variables are always optional.

- Following this logic, ~x|A displays numeric variable x for each level of factor A.
- y~x|A*B displays the relationship between numeric variables y and x separately for every combination of factor A and B levels.
- A~x displays categorical variable A on the vertical axis and numeric variable x on the horizontal axis.
- ~x displays numeric variable x alone.

Example

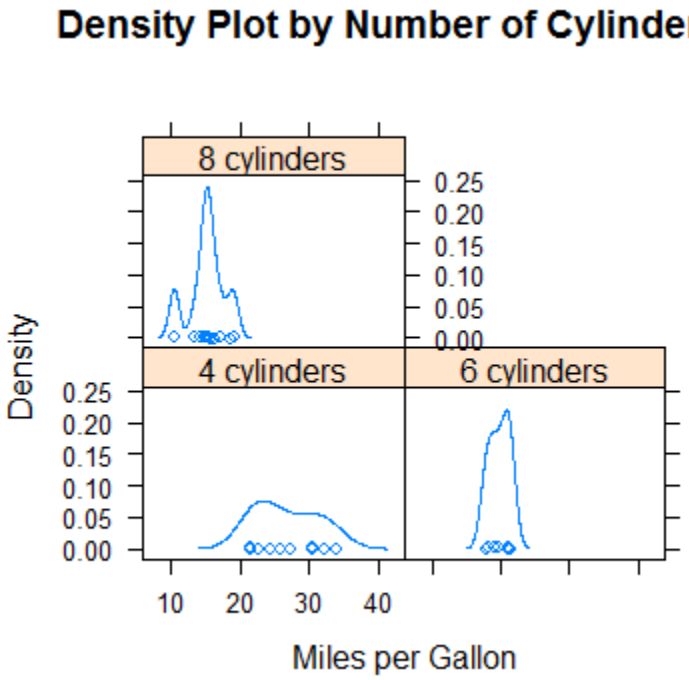
- The graphs are based on the automotive data (mileage, weight, number of gears, number of cylinders, and so on) included in the mtcars data frame .

```
attach(mtcars)
gear <- factor(gear, levels=c(3, 4, 5), labels=c("3 gears", "4 gears", "5 gears"))
cyl <- factor(cyl, levels=c(4, 6, 8), labels=c("4 cylinders", "6 cylinders", "8 cylinders"))
densityplot(~mpg, main="Density Plot“,xlab="Miles per Gallon")
```



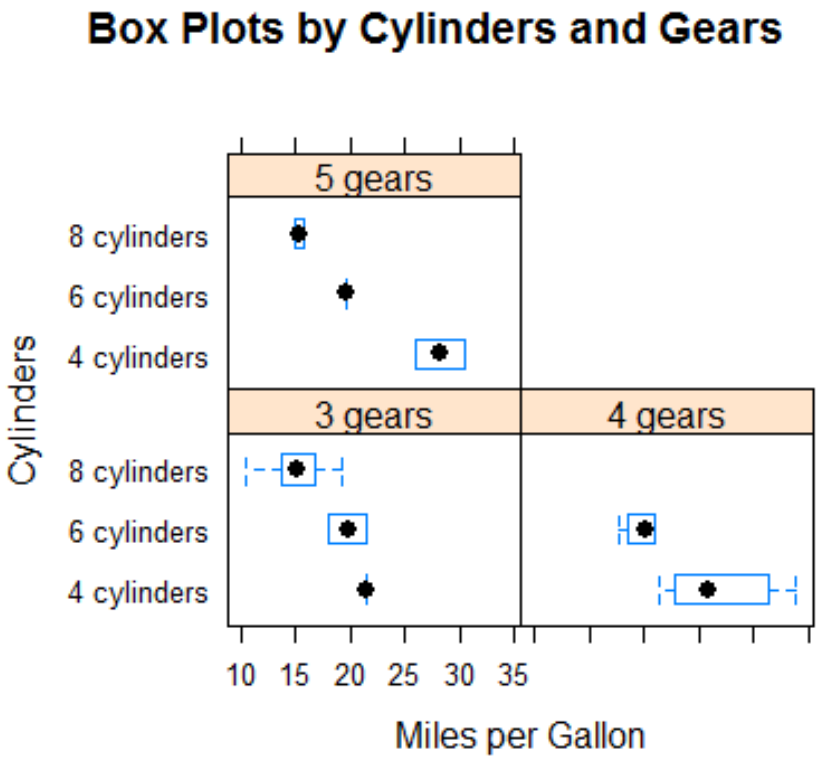
Kernel Density Plot

densityplot(~mpg | cyl, main="Density Plot by Number of Cylinders", xlab="Miles per Gallon")



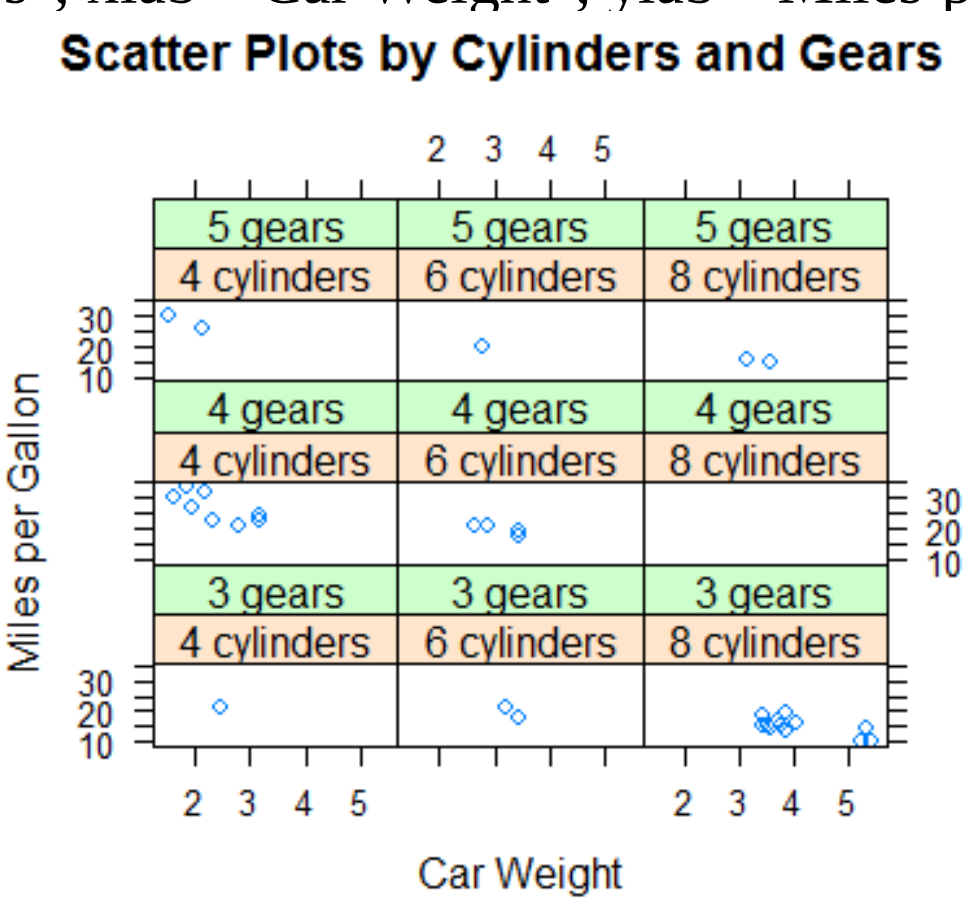
Box Plot

bwplot(cyl ~ mpg | gear,main="Box Plots by Cylinders and Gears", xlab="Miles per Gallon", ylab="Cylinders")



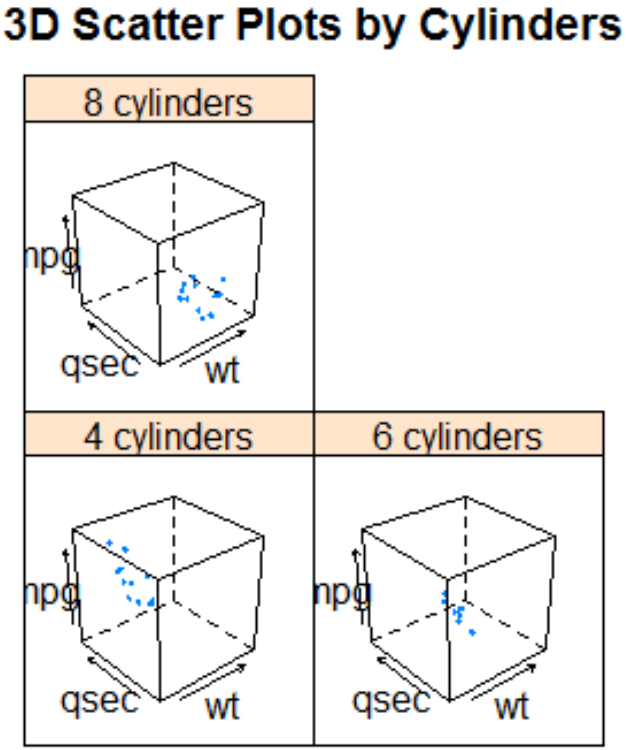
Scatter Plot

xyplot(mpg ~ wt | cyl * gear, main="Scatter Plots by Cylinders and Gears", xlab="Car Weight", ylab="Miles per Gallon")



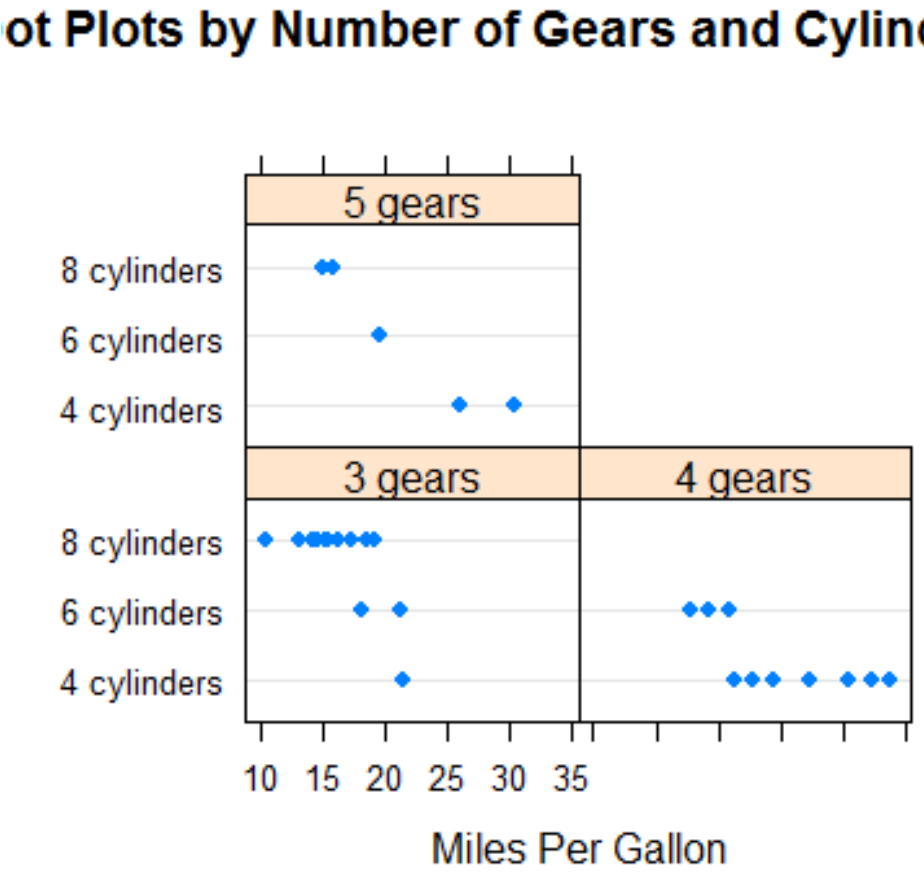
3D Scatter Plot

```
cloud(mpg ~ wt * qsec | cyl, main="3D Scatter Plots by
    Cylinders")
```



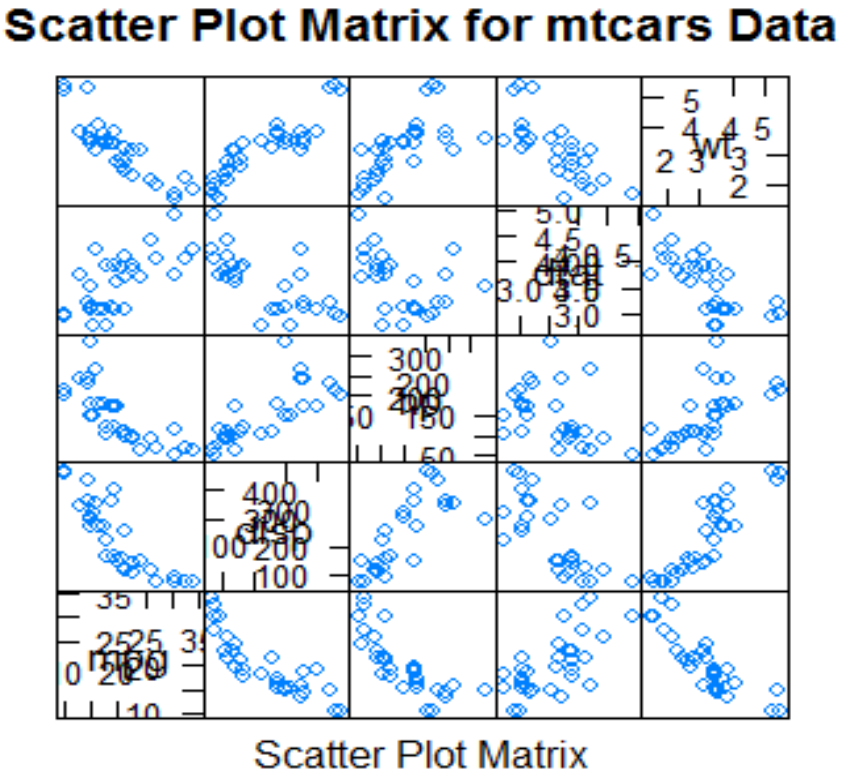
Dot Plot

dotplot(cyl ~ mpg | gear, main="Dot Plots by Number of Gears and Cylinders", xlab="Miles Per Gallon")



Scatter Plot matrix

- splom(mtcars[c(1, 3, 4, 5, 6)], main="Scatter Plot Matrix for mtcars Data")
- detach(mtcars)



- High-level plotting functions in the lattice package produce graphic objects that can be saved and manipulated.
- For example,

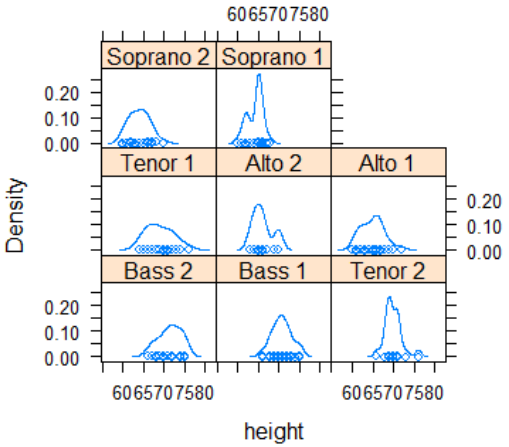
  mygraph <- densityplot(~height|voice.part, data=singer)
- creates a trellis density plot and saves it as object mygraph. But no graph is displayed.
- Issuing the statement plot(mygraph) (or simply mygraph) will display the graph.

- It’s easy to modify lattice graphs through the use of options.

Options	Description
`aspect`	A number specifying the aspect ratio (height/width) for the graph in each panel.
`col, pch, lty, lwd`	Vectors specifying the colors, symbols, line types, and line widths to be used in plotting, respectively.
`groups`	Grouping variable (factor).
`index.cond`	List specifying the display order of the panels.
`key` (or `auto.key`)	Function used to supply legend(s) for grouping variable(s).
`layout`	Two-element numeric vector specifying the arrangement of the panels (number of columns, number of rows). If desired, a third element can be added to indicate the number of pages.
`main, sub`	Character vectors specifying the main title and subtitle.
`panel`	Function used to generate the graph in each panel.

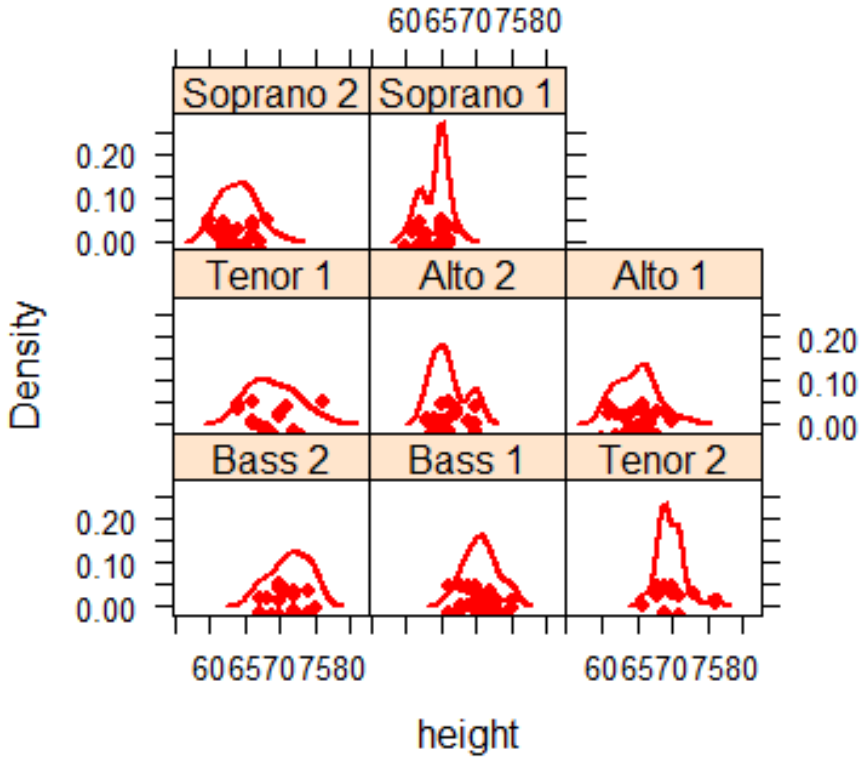
Options	Description
`scales`	List providing axis annotation information.
`strip`	Function used to customize panel strips.
`split, position`	Numeric vectors used to place more than one graph on a page.
`type`	Character vector specifying one or more plotting options for scatter plots (p=points, l=lines, r=regression line, smooth=loess fit, g=grid, and so on).
`xlab, ylab`	Character vectors specifying horizontal and vertical axis labels.
`xlim, ylim`	Two-element numeric vectors giving the minimum and maximum values for the horizontal and vertical axes, respectively.

```
mygraph <- densityplot(~height|voice.part, data=singer)
plot(mygraph)
```



- You can also use the update() function to modify a lattice graphic object. Continuing the singer example, the following would redraw the graph using red curves and symbols (color="red"), filled dots (pch=16), smaller (cex=.8) and more highly jittered points (jitter=.05), and curves of double thickness (lwd=2).

- update(mygraph, col="red", pch=16, cex=.8, jitter=.05, lwd=2)



Conditioning variables

- one of the most powerful features of lattice graphs is the ability to add conditioning variables.
- If one conditioning variable is present, a separate panel is created for each level.
- If two conditioning variables are present, a separate panel is created for each combination of levels for the two variables.
- Typically, conditioning variables are factors. But what if you want to condition on a continuous variable? One approach would be to transform the continuous variable into a discrete variable using R's cut() function .
- Alternatively, the lattice package provides functions for transforming a continuous variable into a data structure called a shingle. Specifically, the continuous variable is divided up into a series of (possibly) overlapping ranges.

- For example, the function

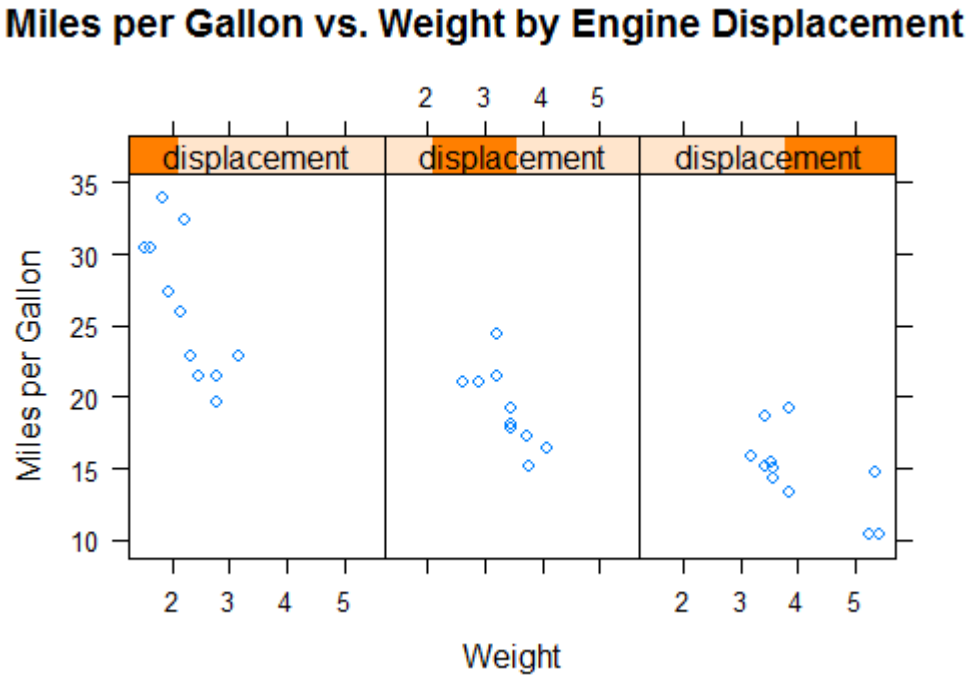
  myshingle <- equal.count(x, number=#, overlap=proportion)
- will take continuous variable x and divide it up into # intervals, with proportion overlap, and equal numbers of observations in each range, and return it as the variable myshingle (of class shingle).
- Printing or plotting this object (for example, plot(myshingle)) will display the shingle's intervals.

Example

- Once a continuous variable has been converted to a shingle, you can use it as a conditioning variable.
- For example, let's use the mtcars dataset to explore the relationship between miles per gallon and car weight conditioned on engine displacement.
- Because engine displacement is a continuous variable, first let's convert it to a shingle variable with three levels:

  displacement <- equal.count(mtcars$disp, number=3, overlap=0)
- Next, use this variable in the xyplot() function:

  xyplot(mpg~wt|displacement, data=mtcars, main = "Miles per Gallon vs. Weight by Engine Displacement", xlab = "Weight", ylab = "Miles per Gallon", layout=c(3, 1), aspect=1.5)



- We have also used options to modify the layout of the panels (three columns and one row) and the aspect ratio (height/width) in order to make comparisons among the three groups easier.

Panel functions

- Each of the high-level plotting functions employs a default function to draw the panels. These default functions follow the naming convention panel.
- graph_function, where graph_function is the high-level function. For example,

  xyplot(mpg~wt|displacement, data=mtcars)

could have also be written as

  xyplot(mpg~wt|displacement,data=mtcars,panel=panel.xyplot)

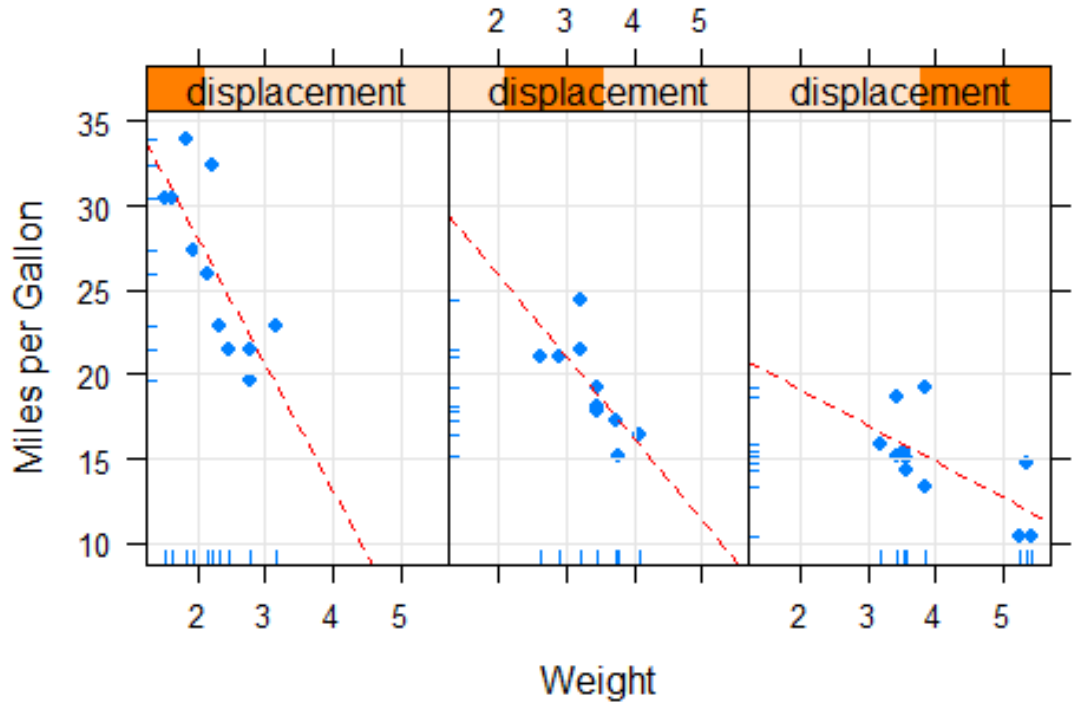
- This is a powerful feature because it allows you to replace the default panel function with a customized function of your own design.

xyplot with custom panel function

- if you wanted to include regression lines, rug plots, and grid lines. You can do this by creating your own panel function

```
displacement <- equal.count(mtcars$disp, number=3, overlap=0)
mypanel <- function(x, y) {
panel.xyplot(x, y, pch=19)
panel.rug(x, y)
panel.grid(h=-1, v=-1)
panel.lmline(x, y, col="red", lwd=1, lty=2)
}
xyplot(mpg~wt|displacement, data=mtcars, layout=c(3, 1),aspect=1.5,
main = "Miles per Gallon vs. Weight by Engine Displacement",
xlab = "Weight", ylab = "Miles per Gallon", panel = mypanel)
```

Miles per Gallon vs. Weight by Engine Displacement

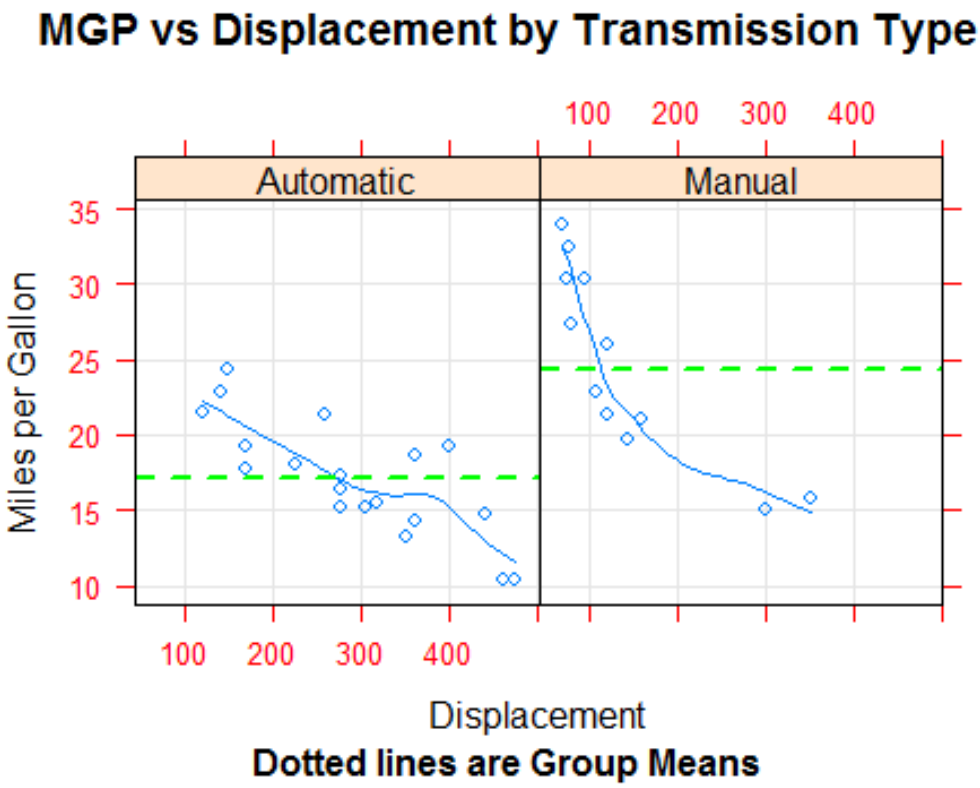


- we’ll graph the relationship between gas mileage and engine displacement(considered as a continuous variable), conditioned on type of automobile transmission.
- In addition to creating separate panels for automatic and manual transmission engines, we’ll add smoothed fit lines and horizontal mean lines.

```
mtcars$transmission <- factor(mtcars$am, levels=c(0,1),
labels=c("Automatic", "Manual"))
panel.smoother <- function(x, y) {
panel.grid(h=-1, v=-1)
panel.xyplot(x, y)
panel.loess(x, y)
panel.abline(h=mean(y), lwd=2, lty=2, col="green")
}
xyplot(mpg~disp|transmission,data=mtcars, scales=list(cex=.8,
    col="red"), panel=panel.smoother, xlab="Displacement",
    ylab="Miles per Gallon", main="MGP vs Displacement by
    Transmission Type", sub = "Dotted lines are Group Means",
    aspect=1)
```

- The panel.xyplot() function plots the individual points, and the panel.loess() function plots nonparametric fit lines in each panel.
- The panel.abline() function adds horizontal reference lines at the mean mpg value for each level of the conditioning variable.
- The scales= option renders scale annotations in red and at 80 percent of their default size.

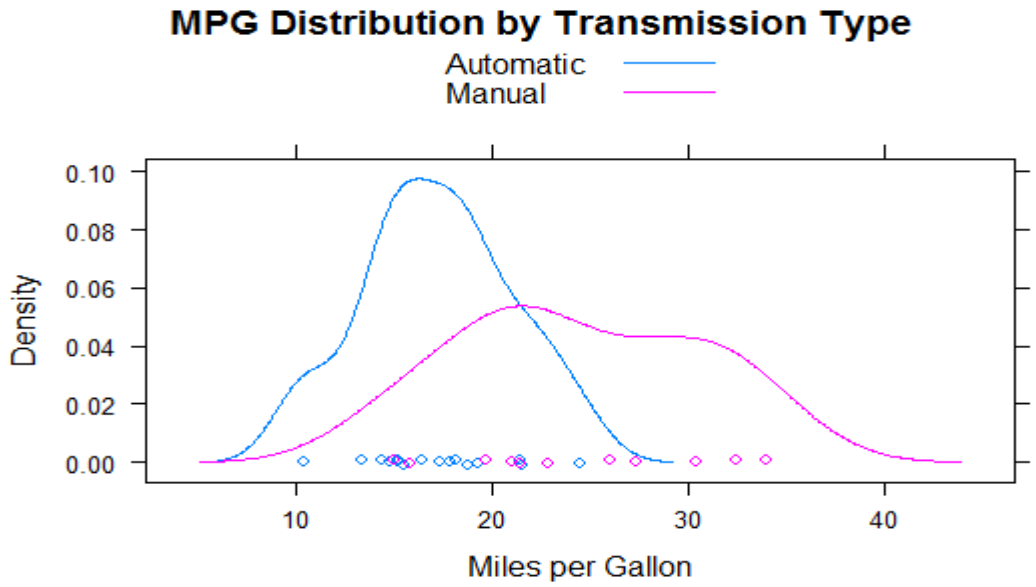
Trellis graph of mpg versus engine displacement conditioned on transmission type. Smoothed lines (loess), grids, and group mean levels have been added.



Grouping variables

- When you include a conditioning variable in a lattice graph formula, a separate panel is produced for each level of that variable.
- If you want to superimpose the results for each level instead, you can specify the variable as a group variable.
- Let’s say that you want to display the distribution of gas mileage for cars with manual and automatic transmissions using kernel density plots.
- You can superimpose these plots using this code:

```
mtcars$transmission <- factor(mtcars$am, levels=c(0, 1),
labels=c("Automatic", "Manual"))
densityplot(~mpg, data=mtcars,group=transmission,
main="MPG Distribution by Transmission Type",
xlab="Miles per Gallon“,auto.key=TRUE)
```

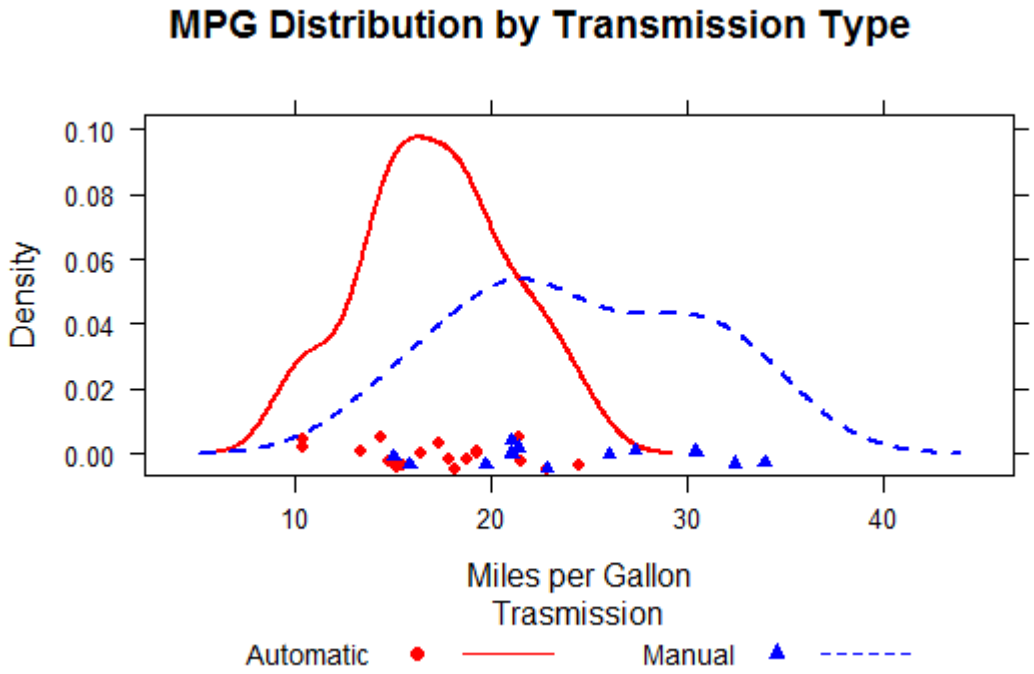


- The option auto.key=TRUE will create a rudimentary legend and place it above the graph.
- You can make limited changes to this automated key by specifying options in a list. For example,

  auto.key=list(space=”right”, columns=1, title=”Transmission”)
- would move the legend to the right of the graph, present the key values in a single column, and add a legend title.

Kernel density plot with a group variable and customized legend

```
mtcars$transmission <- factor(mtcars$am, levels=c(0, 1),
labels=c("Automatic", "Manual"))
colors = c("red", "blue")
lines = c(1,2)
points = c(16,17)
key.trans <- list(title="Trasmission", space="bottom", columns=2,
    text=list(levels(mtcars$transmission)), points=list(pch=points,
    col=colors), lines=list(col=colors, lty=lines), cex.title=1, cex=.9)
densityplot(~mpg, data=mtcars, group=transmission,
main="MPG Distribution by Transmission Type", xlab="Miles per
    Gallon", pch=points, lty=lines, col=colors, lwd=2, jitter=.005,
    key=key.trans)
```

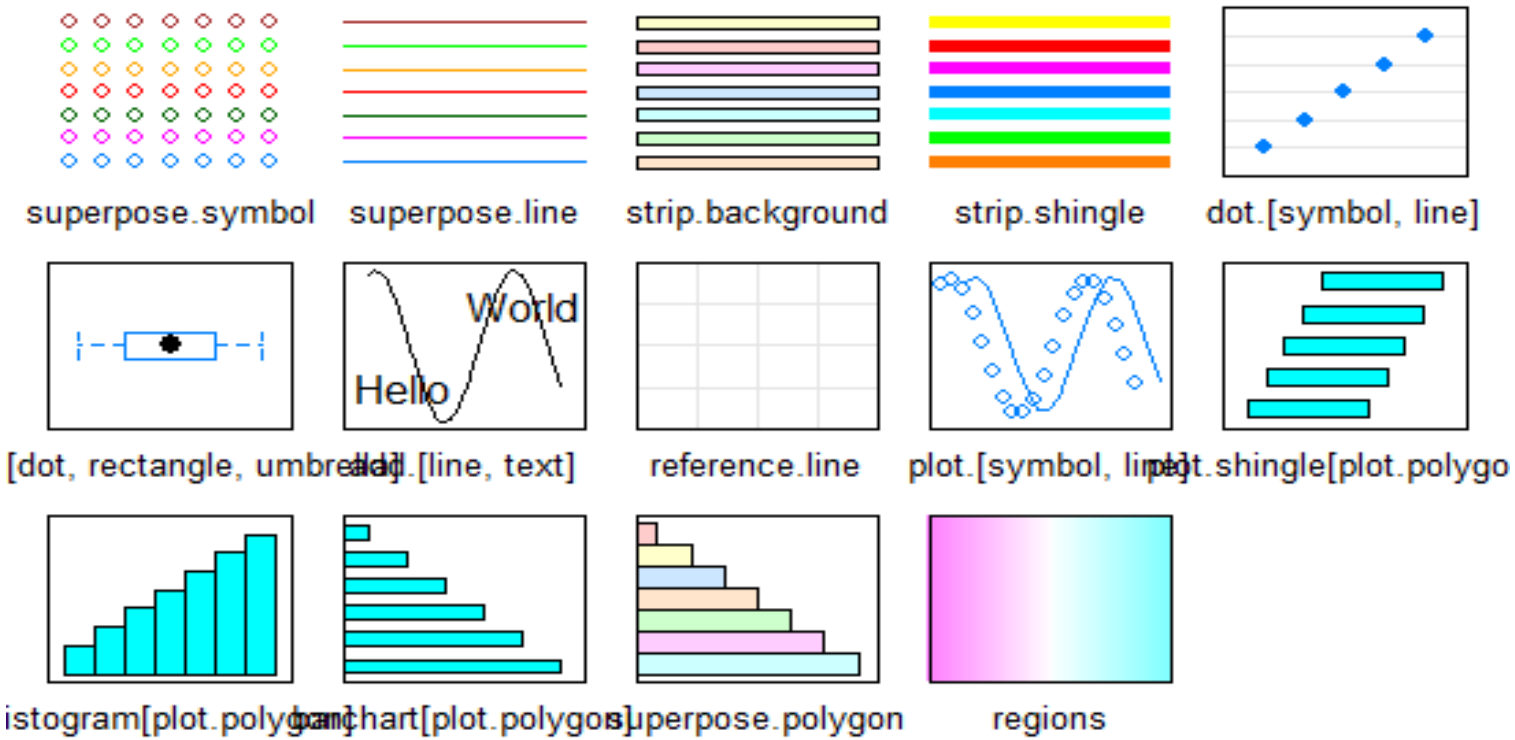

MPG Distribution by Transmission Type
Density
0.10
0.08
0.06
0.04
0.02
0.00
10
20
30
40
Miles per Gallon
Trasmission
Automatic
Manual

Graphic parameters

- you learnt how to view and set default graphics parameters using the par() function .
- Instead, the graphic defaults used by lattice functions are contained in a large list object that can be accessed with the
- trellis.par.get() function and modified through the trellis.par.set() function.
- The show.settings() function can be used to display the current graphic settings visually.

show.settings()

mysettings <- trellis.par.get()



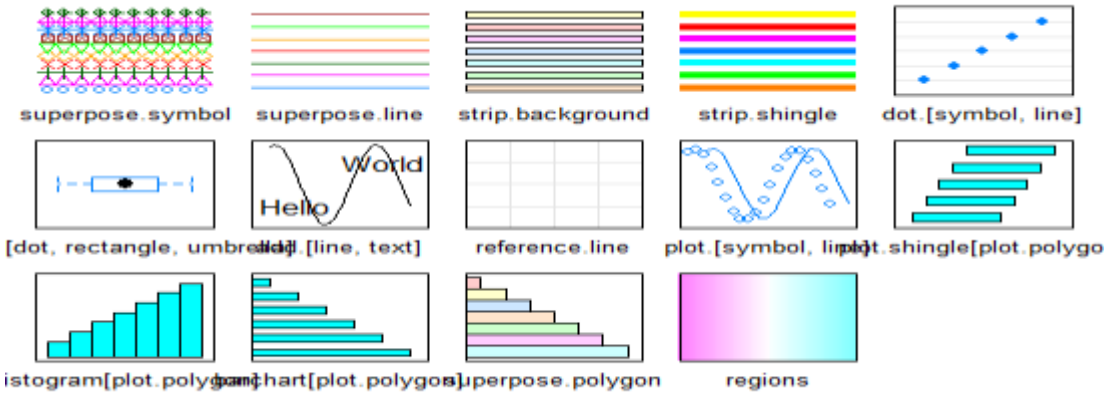
Next, look at the defaults that are specific to superimposed symbols:
mysettings$superpose.symbol

Output

```
$alpha
[1] 1 1 1 1 1 1 1
$cex
[1] 0.8 0.8 0.8 0.8 0.8 0.8 0.8
$col
[1] "#0080ff" "#ff00ff" "darkgreen" "#ff0000" "orange" "#00ff00"
[7] "brown"
$fill
[1] "#CCFFFF" "#FFCCFF" "#CCFFCC" "#FFE5CC" "#CCE6FF"
    "#FFFFCC" "#FFCCCC"
$font
[1] 1 1 1 1 1 1 1
$pch
[1] 1 1 1 1 1 1 1
```

- Here you see that the symbol used for each level of a group variable is an open circle (pch=1). Seven levels are defined, after which symbols recycle.
- Finally, issue the following statements:

```
mysettings$superpose.symbol$pch <- c(1:10)
trellis.par.set(mysettings)
show.settings()
```

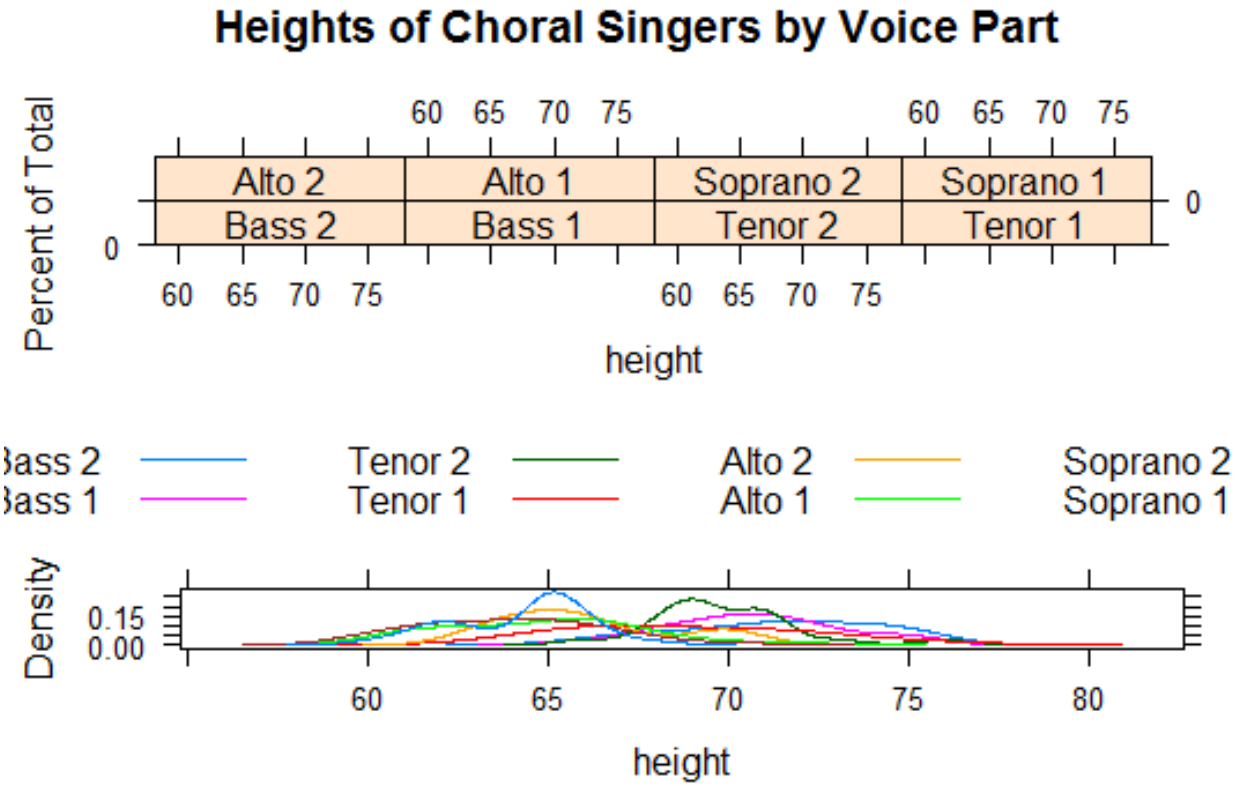


Page arrangement

- The easiest method involves saving your lattice graphs as objects, and using the plot() function with either the split= or position= option specified.
- The split option divides a page up into a specified number of rows and columns and places graphs into designated cells of the resulting matrix.
- The format for the split option is

split=c(placement row, placement column, total number of rows, total number of columns)

```
graph1 <- histogram(~height|voice.part, data=singer,
main="Heights of Choral Singers by Voice Part")
graph2 <- densityplot(~height, data=singer, group=voice.part,
plot.points=FALSE, auto.key=list(columns=4))
plot(graph1, split=c(1, 1, 1, 2))
plot(graph2, split=c(1, 2, 1, 2), newpage=FALSE)
```

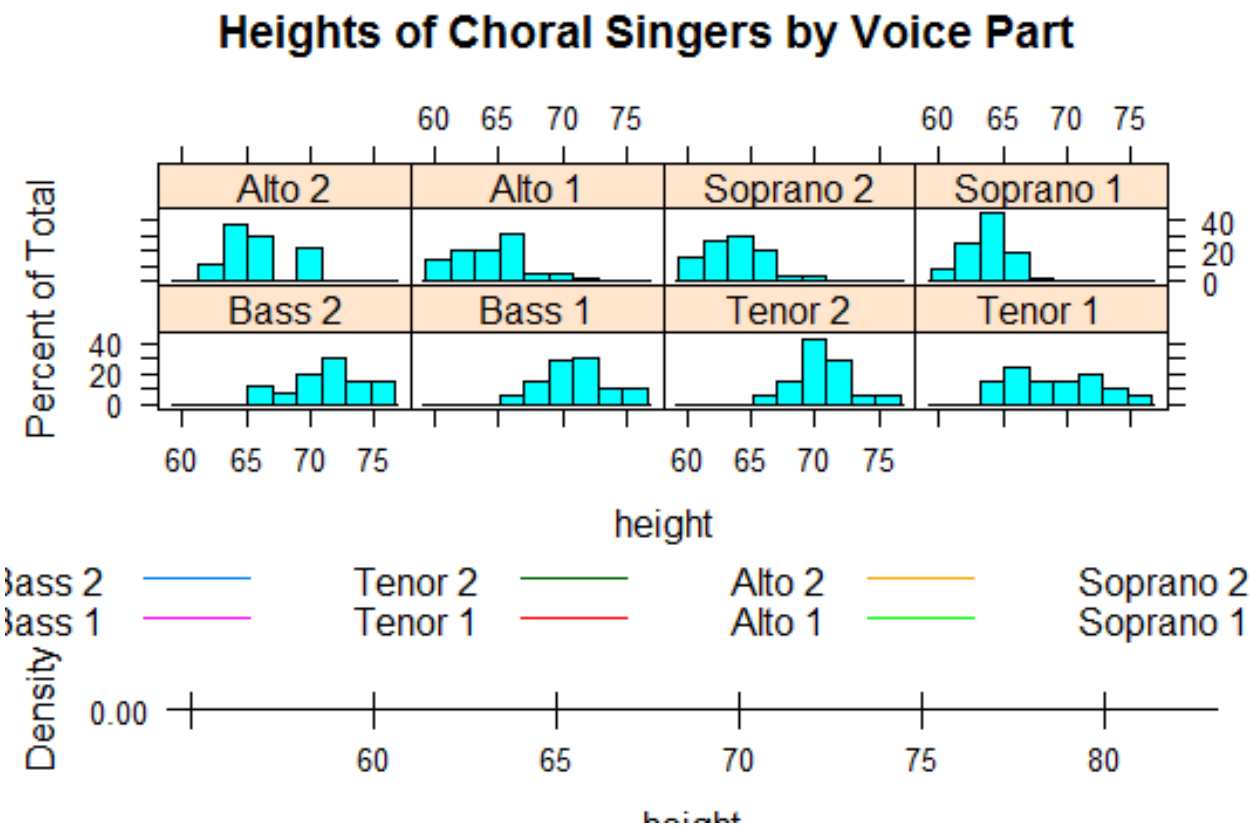

Heights of Choral Singers by Voice Part
Percent of Total
0
60 65 70 75
Alto 2
Alto 1
Soprano 2
Soprano 1
Bass 2
Bass 1
Tenor 2
Tenor 1
height
Bass 2
Bass 1
Tenor 2
Tenor 1
Alto 2
Alto 1
Soprano 2
Soprano 1
Density
0.15
0.00
60 65 70 75 80
height

- places the first graph directly above the second graph.
- Specifically, the first plot() statement divides the page up into one column and two rows and places the graph in the first column and first row (counting top-down and left-right).
- The second plot() statement divides the page up in the same way, but places the graph in the first column and second row.

- You can gain more control of sizing and placement by using the position= option.

```
graph1 <- histogram(~height|voice.part, data=singer,
main="Heights of Choral Singers by Voice Part")
graph2 <- densityplot(~height, data=singer, group=voice.part,
plot.points=FALSE, auto.key=list(columns=4))
plot(graph1, position=c(0, .3, 1, 1))
plot(graph2, position=c(0, 0, 1, .3), newpage=FALSE)
```

- Here, position=c(xmin, ymin, xmax, ymax), where the x-y coordinate system for the page is a rectangle with dimensions ranging from 0 to 1 on both the x and y axes, and the origin (0,0) at the bottom left.


Heights of Choral Singers by Voice Part
Alto 2
Alto 1
Soprano 2
Soprano 1
Bass 2
Bass 1
Tenor 2
Tenor 1
Percent of Total
height
Density
Bass 2
Bass 1
Tenor 2
Tenor 1
Alto 2
Alto 1
Soprano 2
Soprano 1

The ggplot2 package

- The ggplot2 package implements a system for creating graphics in R based on a comprehensive and coherent grammar.
- This provides a consistency to graph creation often lacking in R, and allows the user to create graph types that are innovative and novel.
- The simplest approach for creating graphs in ggplot2 is through the qplot() or quick plot function. The format is

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

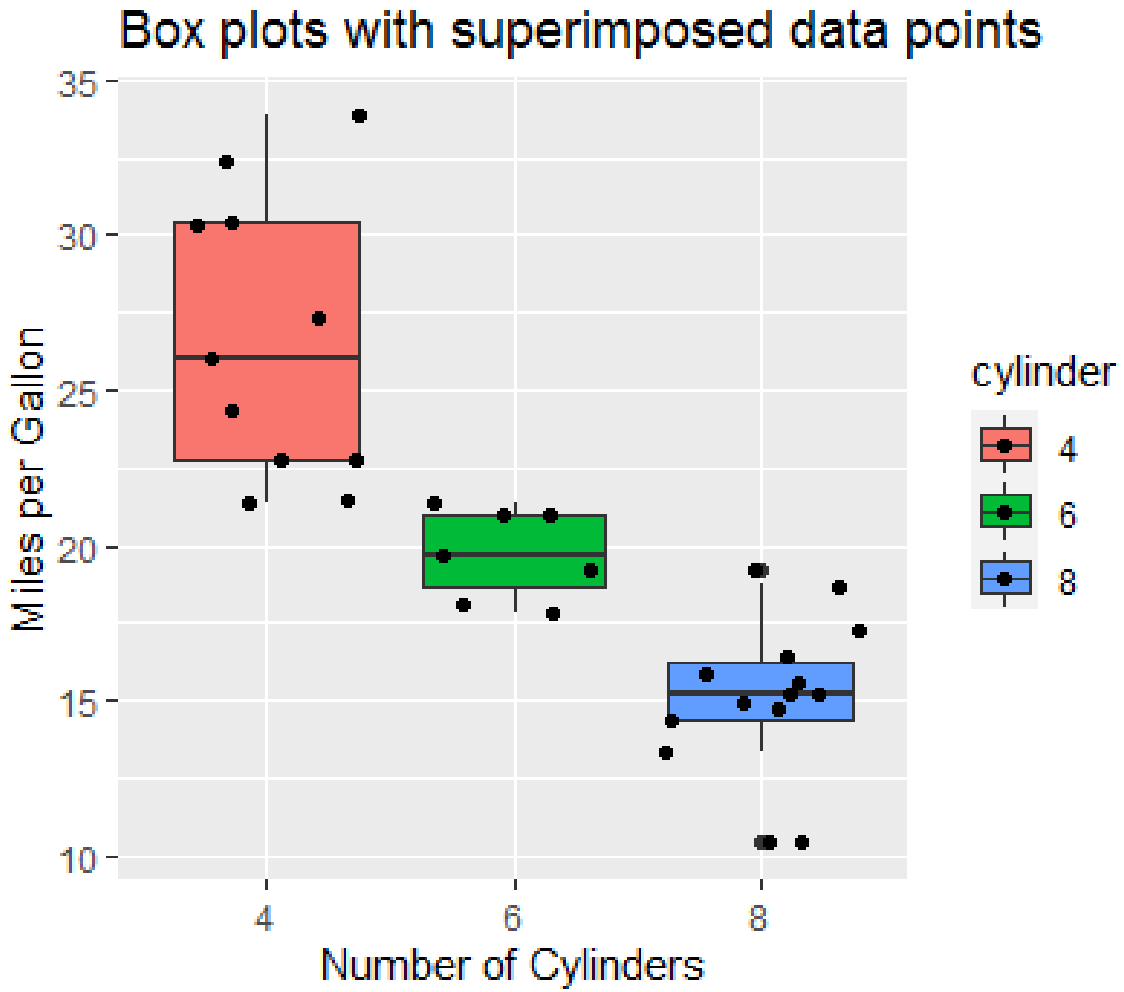
Option	Description
`alpha`	Alpha transparency for overlapping elements expressed as a fraction between 0 (complete transparency) and 1 (complete opacity).
`color`, `shape`, `size`, `fill`	Associates the levels of variable with symbol color, shape, or size. For line plots, `color` associates levels of a variable with line color. For density and box plots, `fill` associates fill colors with a variable. Legends are drawn automatically.
`data`	Specifies a data frame.
`facets`	Creates a trellis graph by specifying conditioning variables. Its value is expressed as `rowvar ~ colvar` (see the example in figure 16.10). To create trellis graphs based on a single conditioning variable, use `rowvar~.` or `.~colvar`.
`geom`	Specifies the geometric objects that define the graph type. The `geom` option is expressed as a character vector with one or more entries. `geom` values include `"point"`, `"smooth"`, `"boxplot"`, `"line"`, `"histogram"`, `"density"`, `"bar"`, and `"jitter"`.
`main`, `sub`	Character vectors specifying the title and subtitle.

`method`, `formula`	If `geom="smooth"`, a loess fit line and confidence limits are added by default. When the number of observations is greater than 1,000, a more efficient smoothing algorithm is employed. Methods include `"lm"` for regression, `"gam"` for generalized additive models, and `"rlm"` for robust regression. The `formula` parameter gives the form of the fit. For example, to add simple linear regression lines, you'd specify `geom="smooth", method="lm", formula=y~x`. Changing the formula to `y~poly(x, 2)` would produce a quadratic fit. Note that the formula uses the letters x and y, not the names of the variables. For `method="gam"`, be sure to load the `mgcv` package. For `method="rml"`, load the `MASS` package.
`x, y`	Specifies the variables placed on the horizontal and vertical axis. For univariate plots (for example, histograms), omit y.
`xlab, ylab`	Character vectors specifying horizontal and vertical axis labels.
`xlim, ylim`	Two-element numeric vectors giving the minimum and maximum values for the horizontal and vertical axes, respectively.

Example

The following code creates box plots of gas mileage by number of cylinders. The actual data points are superimposed (and jittered to reduce overlap). Box plot colors vary by number of cylinders.

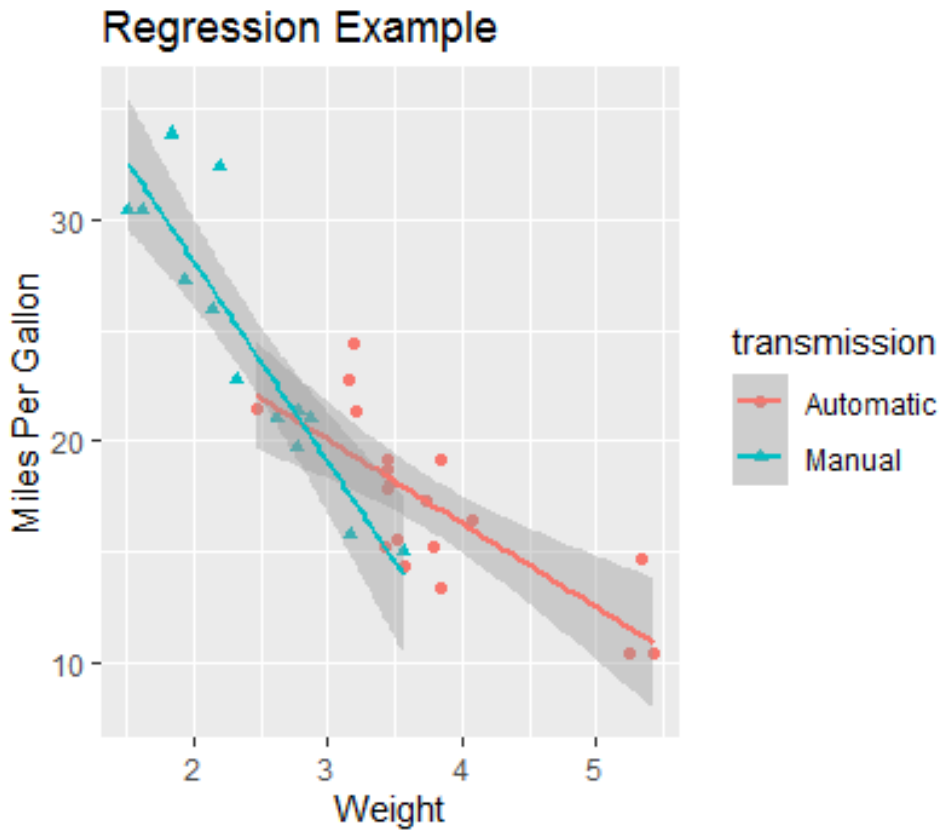
```
library(ggplot2)

mtcars$cylinder <- as.factor(mtcars$cyl)

qplot(cylinder, mpg, data=mtcars, geom=c("boxplot", "jitter"),

fill=cylinder, main="Box plots with superimposed data points", xlab= "Number of Cylinders", ylab="Miles per Gallon")
```


Box plots with superimposed data points
35
30
25
20
15
10
Miles per Gallon
4
6
8
Number of Cylinders
cylinder
4
6
8

Example

- create a scatter plot matrix of gas mileage by car weight and use color and symbol shape to differentiate cars with automatic transmissions from those with manual transmissions.
- Additionally, we'll add separate regression lines and confidence bands for each transmission type.

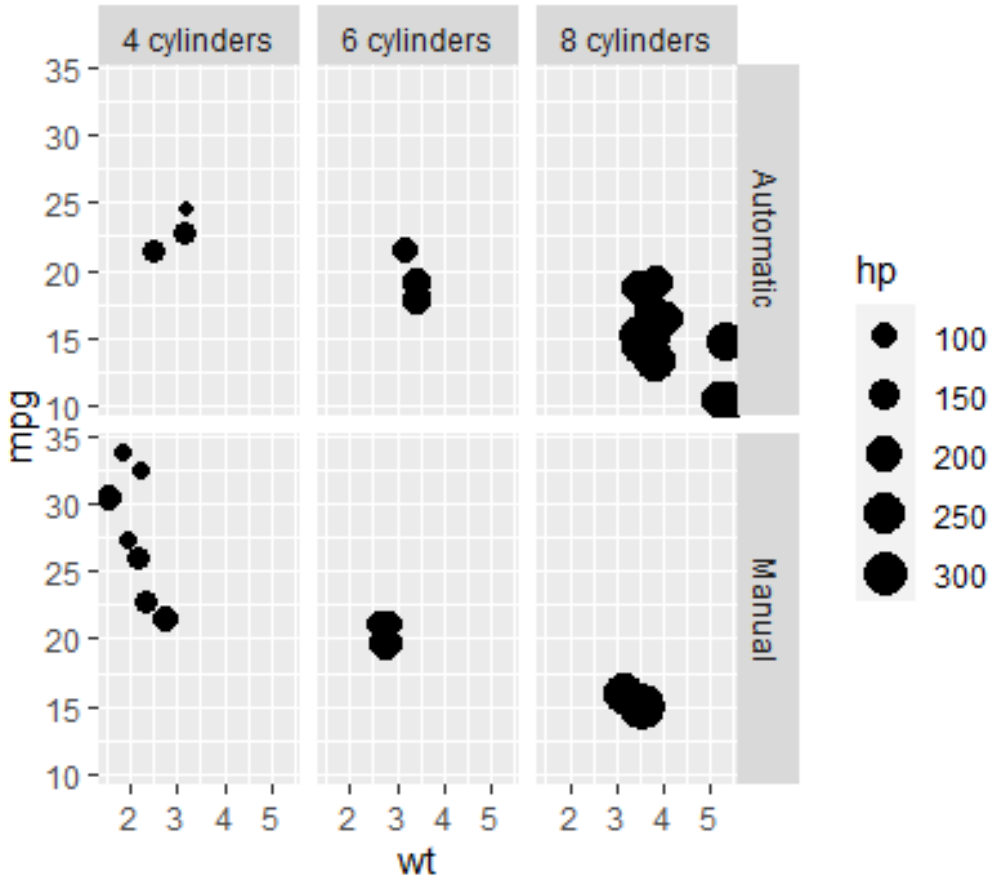
```
transmission <- factor(mtcars$am, levels=c(0, 1),
labels=c("Automatic", "Manual"))
qplot(wt,mpg, data=mtcars, color=transmission,
    shape=transmission,
geom=c("point", "smooth"), method="lm", formula=y~x,
xlab="Weight", ylab="Miles Per Gallon", main="Regression
    Example")
```


Regression Example
Miles Per Gallon
30
20
10
2
3
4
5
Weight
transmission
Automatic
Manual

Example

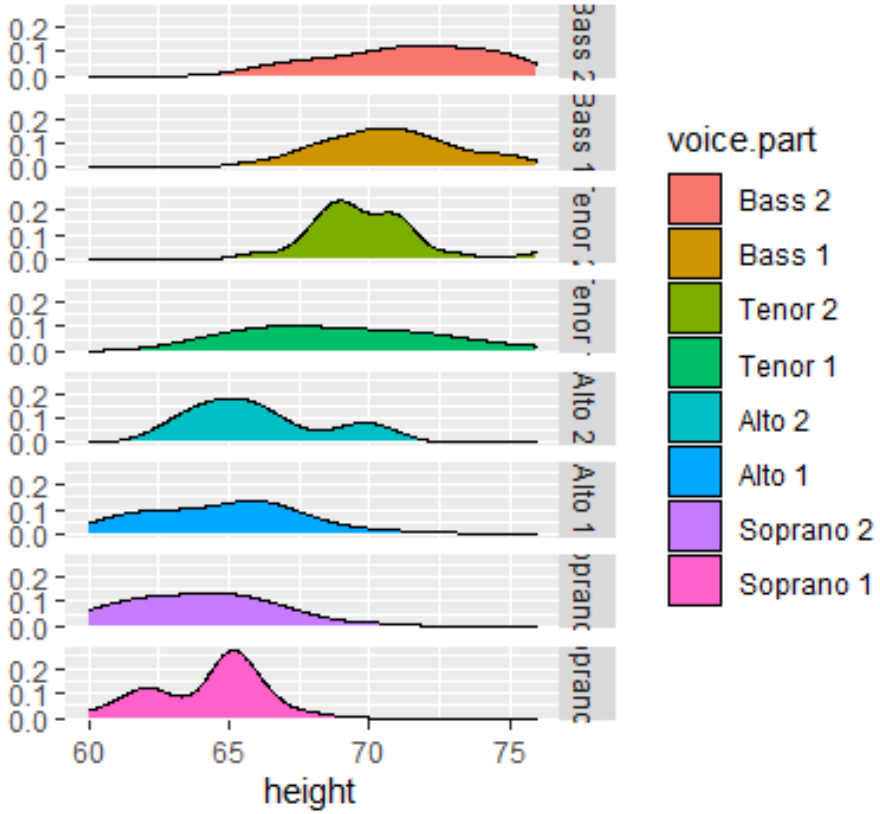
- create a faceted (trellis) graph. Each facet (panel) displays the scatter plot between gas mileage and car weight.
- Row facets are defined by the transmission type, whereas column facets are defined by the number of cylinders present.
- The size of each data point represents the car’s horsepower rating.

```
mtcars$cyl <- factor(mtcars$cyl, levels=c(4, 6, 8),
labels=c("4 cylinders", "6 cylinders", "8 cylinders"))
mtcars$am <- factor(mtcars$am, levels=c(0, 1),
labels=c("Automatic", "Manual"))
qplot(wt,mpg, data=mtcars, facets=am~cyl, size=hp)
```


4 cylinders
6 cylinders
8 cylinders
Automatic
Manual
mpg
wt
hp
100
150
200
250
300

Example

```
data(singer, package="lattice")
qplot(height, data=singer, geom=c("density"),
    facets=voice.part~., fill=voice.part)
```



Interactive graphs

Interacting with graphs: identifying points

- Using the identify() function , you can label selected points in a scatter plot with their row number or row name using your mouse.
- Identification continues until you select Stop or right-click on the graph. For example, after issuing the following statements

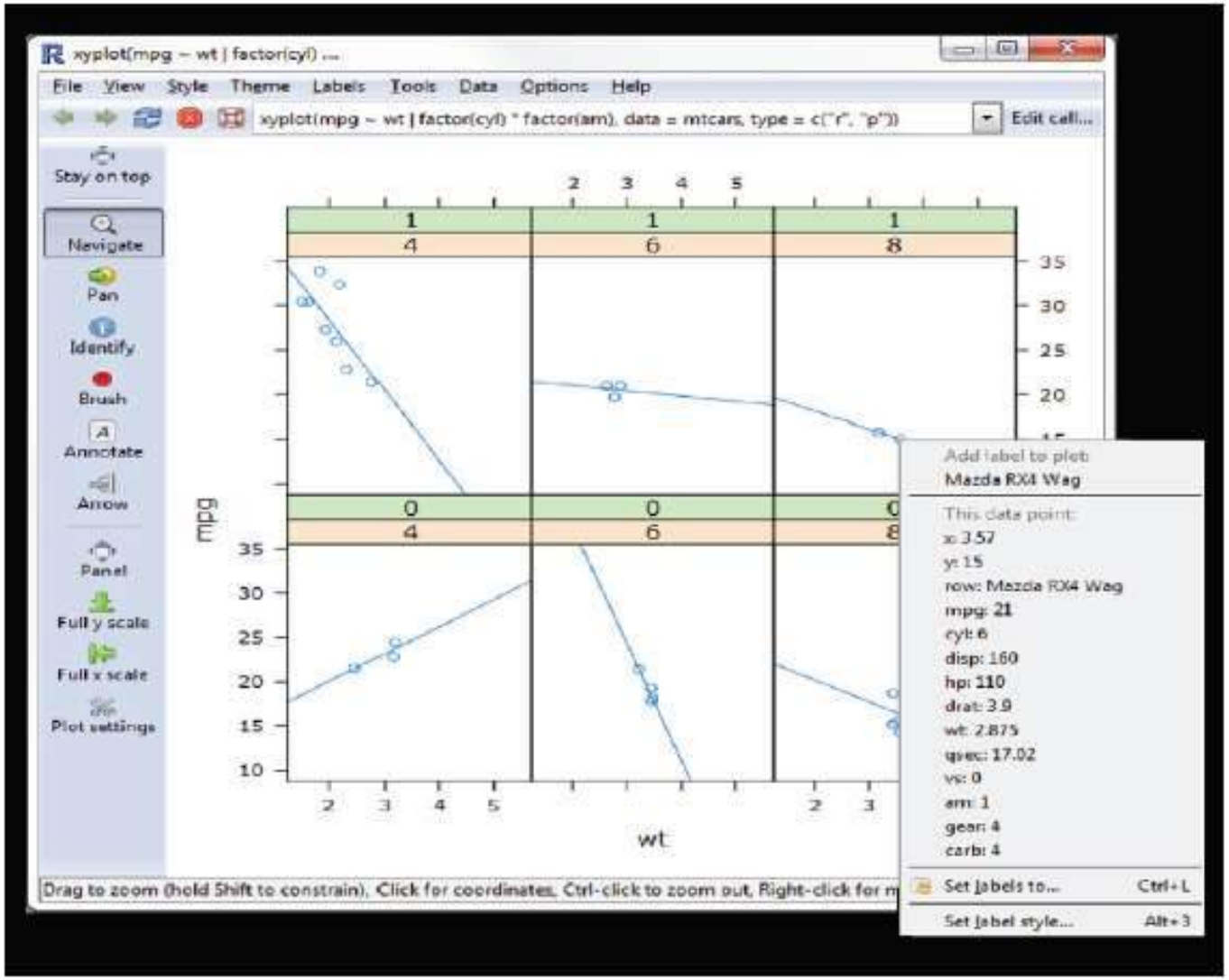
```
plot(mtcars$wt, mtcars$mpg)
identify(mtcars$wt, mtcars$mpg, labels=row.names(mtcars))
```

- the cursor will change from a pointer to a crosshair.
- Clicking on scatter plot points will label them until you select Stop from the Graphics Device menu or right-click on the graph and select Stop from the context menu.

playwith

- The playwith package provides a GTK+ graphical user interface that allows users to edit and interact with R plots.
- You can install the playwith package on any platform using install.packages("playwith", depend=TRUE).
- The playwith()function allows users to identify and label points, view all variable values for an observation, zoom and pan, add annotations (text, arrows, lines,rectangles, titles, labels), change visual elements (colors, text sizes, and so on), apply previously saved styles, and output the resulting graph in a variety of formats.

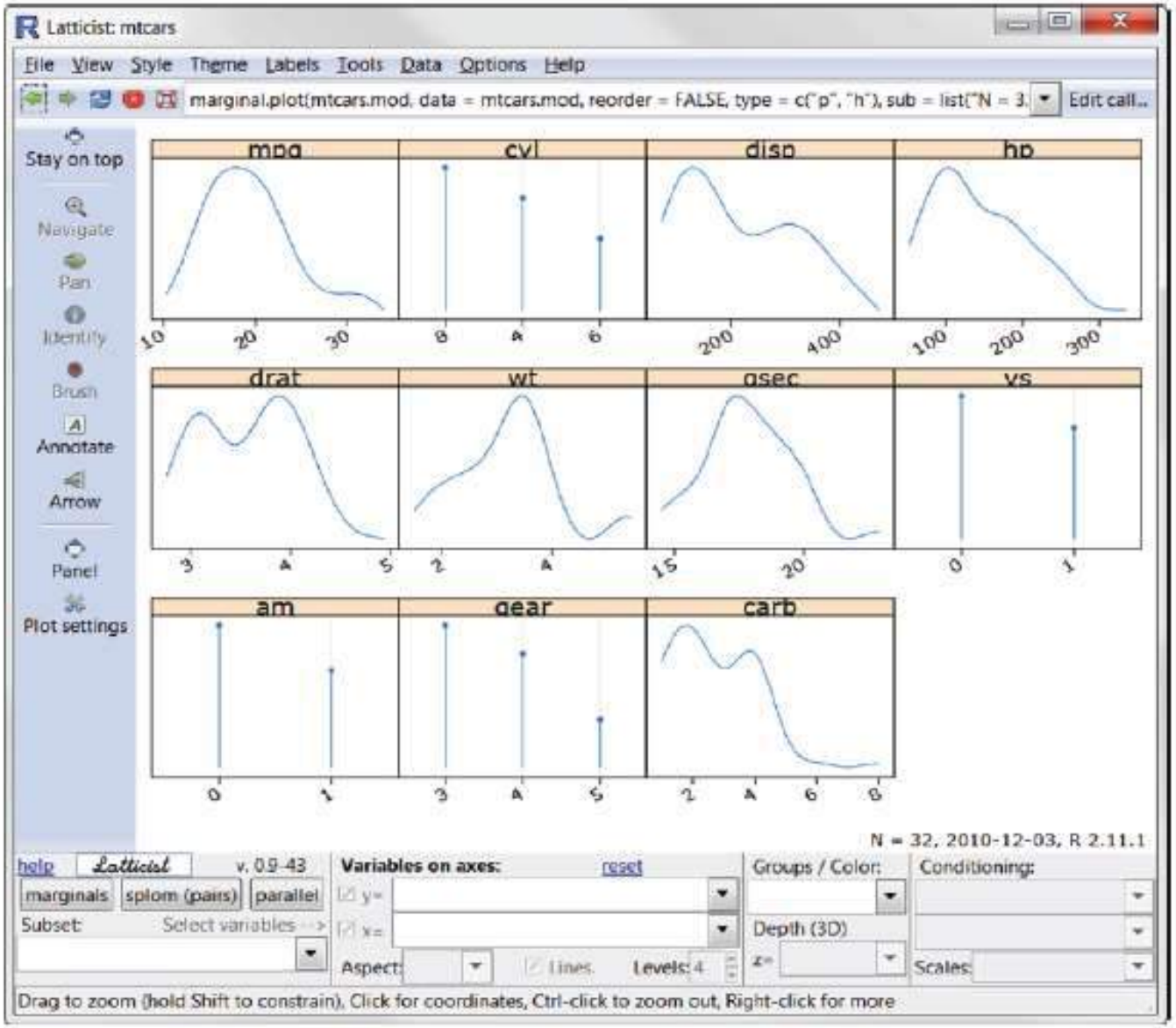
```
library(playwith)
library(lattice)
playwith(xyplot(mpg~wt|factor(cyl)*factor(am),data=mtcars,
    subscripts=TRUE,type=c("r", "p")))
```


xyplot(mpg ~ wt | factor(cyl) ...
File View Style Theme Labels Tools Data Options Help
xyplot(mpg ~ wt | factor(cyl) * factor(am), data = mtcars, type = c("r", "p"))
Edit call...
Stay on top
Navigate
Pan
Identify
Brush
Annotate
Arrow
Panel
Full y scale
Full x scale
Plot settings
mpg
wt
Add label to plot:
Mazda RX4 Wag
This data point:
x: 3.57
y: 15
row: Mazda RX4 Wag
mpg: 21
cyl: 6
disp: 160
hp: 110
drat: 3.9
wt: 2.875
qsec: 17.02
vs: 0
am: 1
gear: 4
carb: 4
Set labels to... Ctrl+L
Set label style... Alt+3
Drag to zoom (hold Shift to constrain). Click for coordinates. Ctrl-click to zoom out. Right-click for m

latticist

- The latticist package lets you explore a data set using lattice displays.
- It provides a graphic user interface to the graphs, but it can also be used to create displays from the vcd package.
- If desired, latticist can also be integrated with playwith. For example, executing the following code

```
library(latticist)
mtcars$cyl <- factor(mtcars$cyl)
mtcars$gear <- factor(mtcars$gear)
latticist(mtcars, use.playwith=TRUE)
```


Latticist: mtcars
File View Style Theme Labels Tools Data Options Help
marginal.plot(mtcars.mod, data = mtcars.mod, reorder = FALSE, type = c("p", "h"), sub = list("N = 3.
Edit call...
Stay on top
Navigate
Pan
Identify
Brush
Annotate
Arrow
Panel
Plot settings
mpg
cyl
disp
hp
drat
wt
qsec
vs
am
gear
carb
N = 32, 2010-12-03, R 2.11.1
help
Latticist
v. 0.9-43
marginals
splom (pairs)
parallel
Subset:
Select variables -->
Variables on axes:
reset
y=
x=
Aspect:
Lines
Levels: 4
Groups / Color:
Depth (3D)
z=
Conditioning:
Scales:
Drag to zoom (hold Shift to constrain). Click for coordinates, Ctrl-click to zoom out, Right-click for more

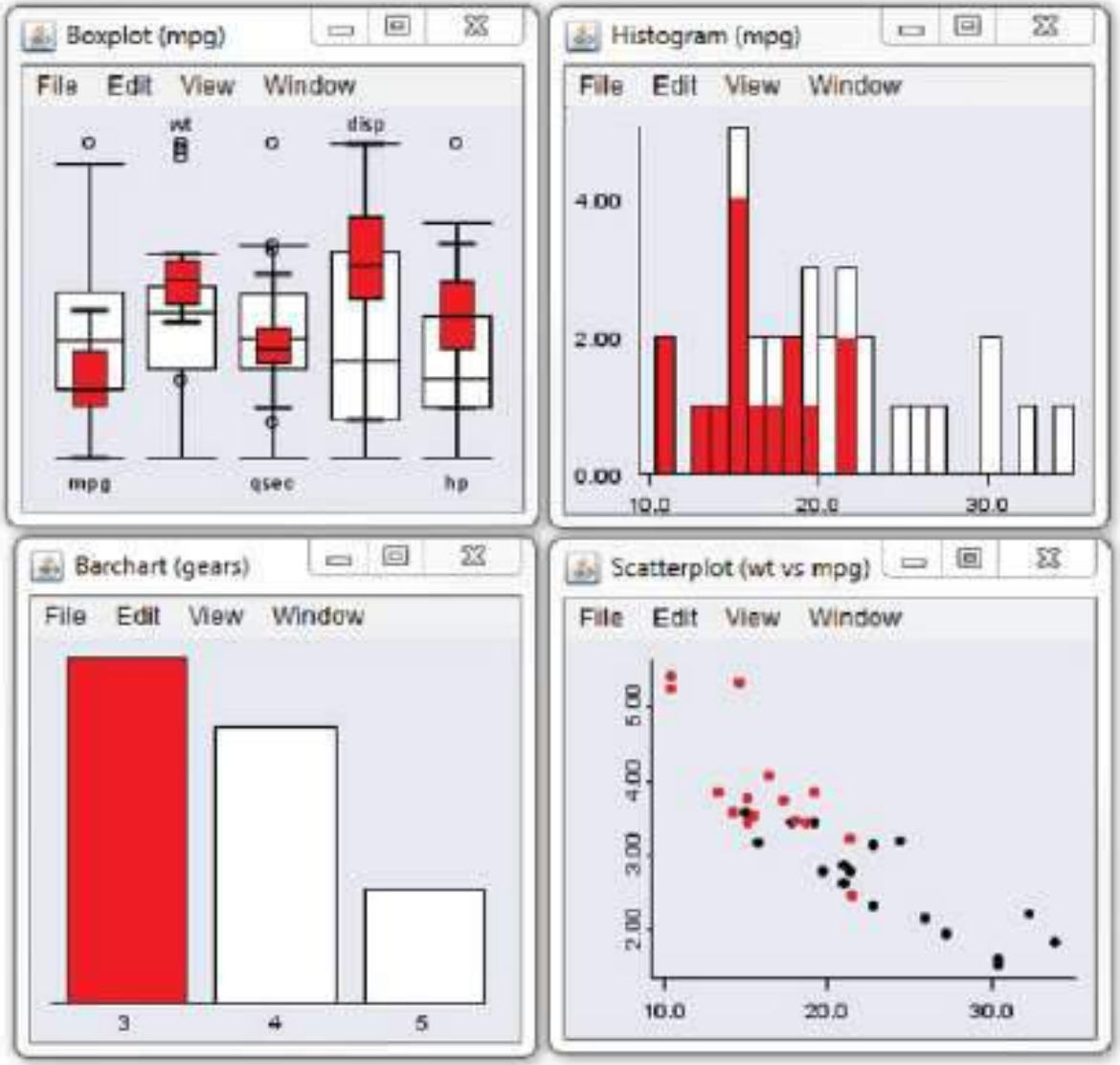
Interactive graphics with the iplots package

- Whereas playwith and latticist allow you to interact with a single graph, the iplots package takes interaction in a different direction.
- This package provides interactive mosaic plots, bar plots, box plots, parallel plots, scatter plots, and histograms that can be linked together and color brushed.
- This means that you can select and identify observations using the mouse, and highlighting observations in one graph will automatically highlight the same observations in all other open graphs.
- You can also use the mouse to obtain information about graphic objects such as points, bars, lines, and box plots.

iplots functions

Function	Description
`ibar()`	Interactive bar chart
`ibox()`	Interactive box plot
`ihist()`	Interactive histogram
`imap()`	Interactive map
`imosaic()`	Interactive mosaic plot
`ipcp()`	Interactive parallel coordinates plot
`iplot()`	Interactive scatter plot

```
library(iplots)
attach(mtcars)
cylinders <- factor(cyl)
gears <- factor(gear)
transmission <- factor(am)
ihist(mpg)
ibar(gears)
iplot(mpg, wt)
ibox(mtcars[c("mpg", "wt", "qsec", "disp", "hp")])
ipcp(mtcars[c("mpg", "wt", "qsec", "disp", "hp")])
imosaic(transmission, cylinders)
detach(mtcars)
```


Boxplot (mpg)
File Edit View Window
wt
disp
mpg
qsec
hp
Histogram (mpg)
File Edit View Window
4.00
2.00
0.00
10.0
20.0
30.0
Barchart (gears)
File Edit View Window
3
4
5
Scatterplot (wt vs mpg)
File Edit View Window
5.00
4.00
3.00
2.00
10.0
20.0
30.0

rggobi

- GGobi is a comprehensive program for the visual and dynamic exploration of high-dimensional data and is freely available for Windows, Mac OS X, and Linux platforms.
- It offers a number of attractive features, including linked interactive scatter plots, bar charts, parallel coordinate plots, time series plots, scatter plot matrices, and 3D rotation; brushing and identification; multivariate transformation methods; and sophisticated exploratory support, including guided and manual 1D and 2D tours.
- The rggobi package provides a seamless interface between GGobi and R.
- The first step in using GGobi is to download and install the appropriate software for your platform (www.ggobi.org/downloads/). Then install the rggobi package within R using install.packages("rggobi", depend=TRUE).

- Once you've installed both, you can use the ggobi() function to run GGobi from within R. This gives you sophisticated interactive graphics access to all of your R data.

```
library(rggobi)
g <- ggobi(mtcars)
```