Kelly Technologies

R Programming

Introduction

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

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.

This programming language was named R based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs language S.

This tutorial is designed for software programmers, statisticians and data miners who are looking forward for developing statistical software using R programming. If you are and trying to understand the R programming language as beginners, this tutorial will give you enough understanding on almost all the concepts of the language from where you can take yourself to higher level of expertise.

Features of R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. There are following important 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.

As a conclusion, R is world's most widely used statistics programming language. It's the # 1 choice of data scientists and supported by a vibrant and talented community of contributors. R is taught in universities and deployed in mission critical business applications. This tutorial will teach you R programming along with suitable examples in simple and easy steps.

Installation

Windows Installation

You can download the Windows installer version of R from R-3.2.2 for Windows (32/64 bit) and save it in a local directory.

As it is a Windows installer (.exe) with a name "R-version-win.exe". You can just double click and run the installer accepting the default settings. If your Windows is 32-bit version, it installs the 32-bit version. But if your windows is 64-bit, then it installs both the 32-bit and 64-bit versions.

After installation you can locate the icon to run the Program in a directory structure "R\R-3.2.2\bin\i386\Rgui.exe" under the Windows Program Files. Clicking this icon brings up the R-GUI which is the R console to do R Programming.

Linux Installation

R is available as a binary for many version of Linux at the location R Binaries.

The instruction to install for various flavors of Linux varies. These steps are mentioned under each type of Linux version in the mentioned link. Still you are in hurry, then you can use **yum** command to install R as follows:

```
$ yum install R
```

Installing R Studio

RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.

Hello World in R

As a convention we will start learning R programming by writing a "Hello, World!" program. Depending on the needs, you can program either at R command prompt or you can use a R script file to write your program. Let's check both one by one.

R Command Prompt

Once you have R environment setup then its easy to start your R command prompt just type the following command at your command prompt:

```
$ R
```

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows:

```
> myString <- "Hello, World!"
> print ( myString)
[1] "Hello, World!"
```

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

R Script File

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called **Rscript**. So let's start with writing following code in a text file called test.R:

```
# My first program in R Programming
myString <- "Hello, World!"
```

```
print ( myString)
```

Save above code in a file test.R and execute it at Linux command prompt as given below. Even if you are using Windows or other system, syntax will remain same.

```
$ Rscript test.R
```

When we run above program, it produces the following result.

```
[1] "Hello, World!"
```

R - Data Types

While doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

In contrast to other Programming languages like C and java 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. There are many types of R-objects. The frequently used ones are

- **Vectors**
- **Lists**
- **Matrices**
- **Arrays**
- **Factors**
- **Data Frames**

The simplest of these objects is the **vector object** and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

R Programming

Data Type	Example	Verify

Logical	TRUE , FALSE	`v <- TRUE`
		`print(class(v))`
		it produces following result:
		`[1] "logical"`

Numeric	12.3, 5, 999	`v <- 23.5`
		`print(class(v))`
		it produces following result:
		`[1] "numeric"`

Integer	2L, 34L, 0L	`v <- 2L`
		`print(class(v))`
		it produces following result:
		`[1] "integer"`

Complex	3 + 2i	`v <- 2+5i`
		`print(class(v))`
		it produces following result:
		`[1] "complex"`

Character	'a', '"good"', "TRUE", '23.4'	`v <- "TRUE"`
		`print(class(v))`
		it produces following result:
		`[1] "character"`

Raw	"Hello" is stored as 48 65 6c 6c 6f	`v <- charToRaw("Hello")`
		`print(class(v))`
		it produces following result:
		`[1] "raw"`

So in R the very basic data types are the R-objects called **vectors** which hold elements of different classes as shown above. Please note in R the number of

classes is not confined to only the above six types. For example we can use many atomic vectors and create an array whose class will become array.

Vectors

When you want to create vector with more than one element, you should use **c** function which means to combine the elements into a vector.

```
# Create a vector.
apple <- c( red','green',"yellow")
print(apple)

# Get the class of the vector.
print(class(apple))
```

When we execute above code, it produces following result:

```
[1] "red"    "green"  "yellow"
[1] "character"
```

Lists

A list is a R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.
list1 <- list(c(2,5,3),21.3,sin)

# Print the list.
print(list1)
```

When we execute above code, it produces following result:

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

[[2]]
[1] 21.3

[[3]]
function (x)  .Primitive("sin")
```

Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.
M = matrix( c('a','a','b','c','b','a'), nrow=2,ncol=3,byrow = TRUE)
print(M)
```

When we execute above code, it produces following result:

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

Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

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

When we execute above code, it produces following result:

```
, , 1

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

, , 2

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

Factors

Factors are the r-objects which are created using a vector. It stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or boolean etc in the input vector. They are useful in statistical modeling.

Factors are created using the **factor** function.The **nlevels** functions gives the count of levels.

```
# Create a vector.
apple_colors <- c('green','green','yellow','red','red','red','green')

# Create a factor object.
factor_apple <- factor(apple_colors)

# Print the factor.
print(factor_apple)
print(nlevels(factor_apple))
```

When we execute above code, it produces following result:

```
[1] green  green  yellow red    red    red    yellow green
Levels: green red yellow
# applying the nlevels function we can know the number of distinct
values
[1] 3
```

Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the **data.frame** function.

```
# Create the data frame.
BMI <-  data.frame(
                    gender = c("Male", "Male","Female"),
                    height = c(152, 171.5, 165),
                    weight = c(81,93, 78),
                    Age =c(42,38,26)
                    )
print(BMI)
```

When we execute above code, it produces following result:

```
  gender height weight Age
1   Male  152.0     81  42
2   Male  171.5     93  38
3 Female  165.0     78  26
```

R- Variables

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R-objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot. and underscore allowed.
2var_name	invalid	Starts with a number
.var_name , var.name	valid	Can start with a dot. but the dot.should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid
_var_name	invalid	Starts with _ which is not valid

Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using **print** or **cat** function. The **cat** function combines multiple items into a continuous print output.

```
# Assignment using equal operator.
var.1 = c(0,1,2,3)

# Assignment using leftward operator.
var.2 <- c("learn","R")

# Assignment using rightward operator.
c(TRUE,1) -> var.3

print(var.1)
cat ("var.1 is ", var.1 ,"\n")
cat ("var.2 is ", var.2 ,"\n")
cat ("var.3 is ", var.3 ,"\n")
```

When we execute above code, it produces following result:

```
[1] 0 1 2 3
var.1 is  0 1 2 3
var.2 is  learn R
var.3 is  1 1
```

Note: The vector cTRUE,1 has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R -object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
var_x <- "Hello"
cat("The class of var_x is ",class(var_x),"\n")

var_x <- 34.5
cat("  Now the class of var_x is ",class(var_x),"\n")

var_x <- 27L
cat("   Next the class of var_x becomes ",class(var_x),"\n")
```

When we execute above code, it produces following result:

```
The class of var_x is  character
  Now the class of var_x is  numeric
    Next the class of var_x becomes  integer
```

Finding Variables

To know all the variables currently available in the workspace we use the **ls** function. Also the lsfunction can use patterns to match the variable names.

```
print(ls())
```

When we execute above code, it produces following result:

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

Note: It is a sample output depending on what variables are declared in your environment.

The ls function can use patterns to match the variable names.

```
# List the variables starting with the pattern "var".
print(ls(pattern="var"))
```

When we execute above code, it produces following result:

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

The variables starting with **dot**. are hidden, they can be listed using "all.names=TRUE" argument to ls function.

```
print(ls(all.name=TRUE))
```

When we execute above code, it produces following result:

```
[1] ".cars"         ".Random.seed" ".var_name"    ".varname"
".varname2"
[6] "my var"        "my_new_var"   "my_var"       "var.1"        "var.2"
[11]"var.3"         "var.name"     "var_name2."   "var_x"
```

Deleting Variables

Variables can be deleted by using the **rm** function. Below we delete the variable var.3. On printing the value of the variable error is thrown.

```
rm(var.3)
print(var.3)
```

When we execute above code, it produces following result:

```
[1] "var.3"
Error in print(var.3) : object 'var.3' not found
```

All the variables can be deleted by using the **rm** and **ls** function together.

```
rm(list=ls())
print(ls())
```

When we execute above code, it produces following result:

```
character(0)
```

R - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

Types of Operator

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

Arithmetic Operators

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

R Programming

Operator	Description	Example

+	Adds two vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v+t)`
		it produces following result:
		`[1] 10.0 8.5 10.0`

–	Subtracts second vector from the first	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v-t)`
		it produces following result:
		`[1] -6.0 2.5 2.0`

*	Multiplies both vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v*t)`
		it produces following result:
		`[1] 16.0 16.5 24.0`

/	Divide the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v/t)`
		When we execute above code, it produces following result:
		`[1] 0.250000 1.833333 1.500000`

%%	Give the remainder of the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v%%t)`
		it produces following result:
		`[1] 2.0 2.5 2.0`

%/%	The result of	`v <- c( 2,5.5,6)`

	division of first vector with second quotient	`t <- c(8, 3, 4)`
		`print(v%/%t)`
		it produces following result:
		`[1] 0 1 1`

^	The first vector raised to the exponent of second vector	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v^t)`
		it produces following result:
		`[1] 256.000 166.375 1296.000`

Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a boolean value.

Operator	Description	Example

>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>t)`
		it produces following result:
		`[1] FALSE TRUE FALSE FALSE`

<	Checks if each element of the first vector is less than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v < t)`
		it produces following result:
		`[1] TRUE FALSE TRUE FALSE`

==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v==t)`

it produces following result:

```
[1] FALSE FALSE FALSE  TRUE
```

<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v<=t)`
		it produces following result:
		`[1]  TRUE FALSE  TRUE  TRUE`

>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>=t)`
		it produces following result:
		`[1] FALSE  TRUE FALSE  TRUE`

!=	Checks if each element of the first vector is unequal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v!=r)`
		it produces following result:
		`[1]  TRUE  TRUE  TRUE FALSE`

Logical Operators

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a boolean value.

Operator	Description	Example

&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding	`v <- c(3,1,TRUE,2+3i)`
		`t <- c(4,1,FALSE,2+3i)`
		`print(v&t)`

	element of the second vector and gives a output TRUE if both the elements are TRUE.	it produces following result: `[1] TRUE TRUE FALSE TRUE`

|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	`v <- c(3,0,TRUE,2+2i)`
		`t <- c(4,0,FALSE,2+3i)`
		`print(v|t)`
		it produces following result:
		`[1] TRUE FALSE TRUE TRUE`

!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	`v <- c(3,0,TRUE,2+2i)`
		`print(!v)`
		it produces following result:
		`[1] FALSE TRUE FALSE FALSE`

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

Operator	Description	Example

&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	`v <- c(3,0,TRUE,2+2i)`
		`t <- c(1,3,TRUE,2+3i)`
		`print(v&&t)`
		it produces following result:
		`[1] TRUE`

||	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	`v <- c(0,0,TRUE,2+2i)`
		`t <- c(0,3,TRUE,2+3i)`
		`print(v||t)`
		it produces following result:
		`[1] FALSE`

Assignment Operators

These operators are used to assign values to vectors.

Operator	Description	Example

<-
or
=
or
<<-
	Called Left Assignment	`v1 <- c(3,1,TRUE,2+3i)`
		`v2 <<- c(3,1,TRUE,2+3i)`
		`v3 = c(3,1,TRUE,2+3i)`
		`print(v1)`
		`print(v2)`
		`print(v3)`
		it produces following result:
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`

->
or
->>
	Called Right Assignment	`c(3,1,TRUE,2+3i) -> v1`
		`c(3,1,TRUE,2+3i) ->> v2`
		`print(v1)`
		`print(v2)`
		it produces following result:
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`

Miscellaneous Operators

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

Operator	Description	Example

:	Colon operator. It creates the	`v <- 2:8`
		`print(v)`
		it produces following result:

	series of numbers in sequence for a vector.	`[1] 2 3 4 5 6 7 8`

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

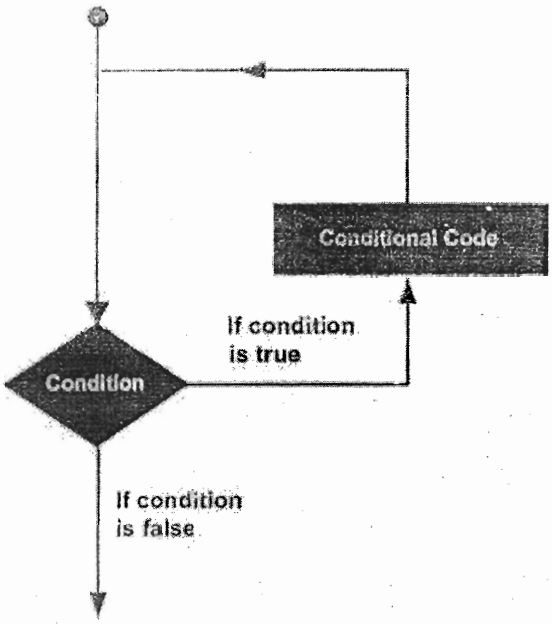
%*%	This operator is used to multiply a matrix with its transpose.	`M = matrix( c(2,6,5,1,10,4), nrow=2,ncol=3,byrow = TRUE)`
		`t = M %*% t(M)`
		`print(t)`
		it produces following result:
		`[,1] [,2]`
		`[1,] 65 82`
		`[2,] 82 117`

R-Loops

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



R programming language provides following kinds of loop to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
repeat loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Like a while statement, except that it tests the condition at the end of the loop body.

Repeat Loop

The **Repeat loop** executes the same code again and again until a stop condition is met.

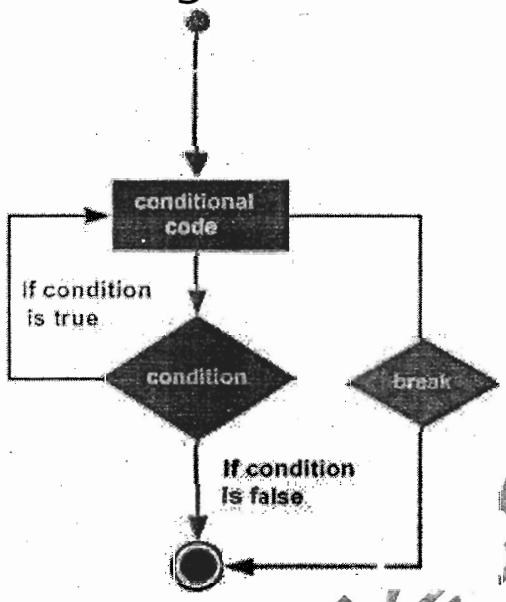
Syntax

The basic syntax for creating a repeat loop in R is:

```
repeat {
```

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

Flow Diagram



Example

```
v <- c("Hello","loop")
cnt <- 2
repeat{
   print(v)
   cnt <- cnt+1
   if(cnt > 5){
      break
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

```
[1] "Hello" "loop"
```

While Loop

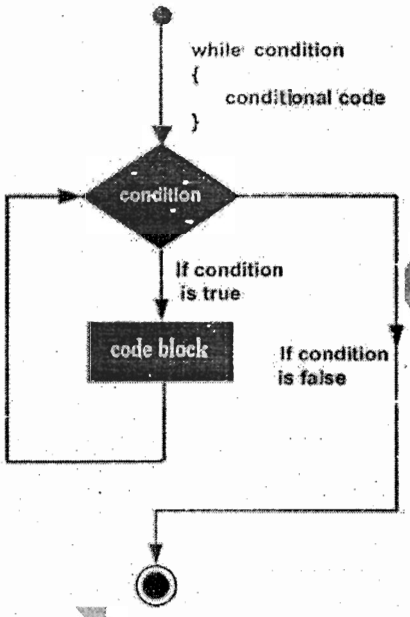
The While loop executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a while loop in R is :

```
while (test_expression) {
    statement
}
```

Flow Diagram



Here key point of the **while** loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
v <- c("Hello","while loop")

cnt <- 2

while (cnt < 7){
```

```
   print(v)

   cnt = cnt + 1

}
```

When the above code is compiled and executed, it produces the following result :

```
[1] " Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
```

For Loop

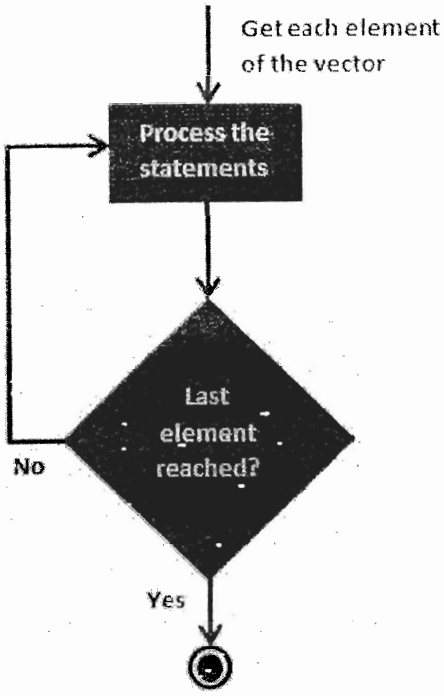
A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The basic syntax for creating a for loop statement in R is:

```
for (value in vector) {
   statements
}
```

Flow Diagram



R's for loops are particularly flexible in that they are not limited to integers, or even numbers in the input. We can pass character vectors, logical vectors, lists or expressions.

Example

```
v <- LETTERS[1:4]
for ( i in v) {
   print(i)
}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "A"
[1] "B"
[1] "C"
[1] "D"
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

R supports the following control statements. Click the following links to check their detail.

Control Statement	Description
break statement	Terminates the **loop** statement and transfers execution to the statement immediately following the loop.
Next statement	The next statement simulates the behavior of R switch.

R- break Statement

The break statement in R programming language has the following two usages:

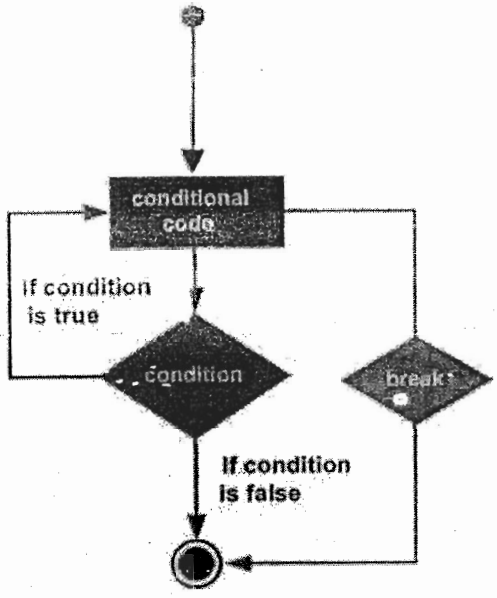
- When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the *loop*.
- It can be used to terminate a case in the switch statement (covered in the next chapter).

Syntax

The basic syntax for creating a break statement in R is:

```
break
```

Flow Diagram



Example

```
v <- c("Hello","loop")
cnt <- 2
repeat{
   print(v)
   cnt <- cnt+1
   if(cnt > 5){
      break
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

Next Statement:

The **next** statement in R programming language is useful when we want to skip the current iteration of a loop without terminating it. On

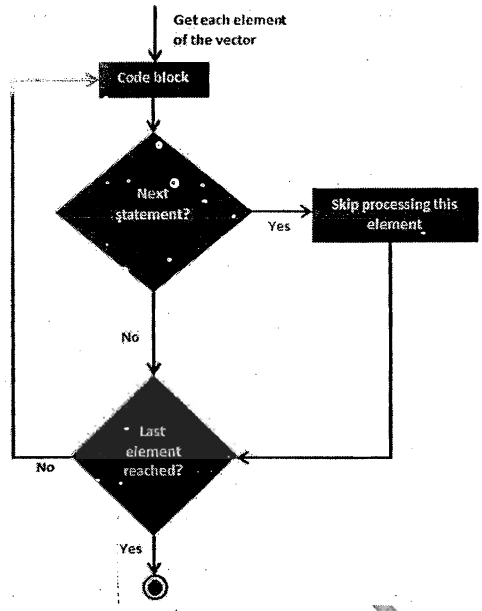
encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax

The basic syntax for creating a next statement in R is:

```
next
```

Flow Diagram



Example

```
v <- LETTERS[1:6]
for ( i in v){
   if (i == "D"){
      next
   }
   print(i)
}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "A"
[1] "B"
[1] "C"
```

```
[1] "E"
[1] "F"
```

R-Functions

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

Function Components

The different parts of a function are:

- **Function Name:** This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments:** An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body:** The function body contains a collection of statements that defines what the function does.
- **Return value:** The return value of a function is the last expression in the function body to be evaluated.

Example

R Programming

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

Built-in Function

Simple examples of in-built functions
are **seq, mean, max, sum**x and **paste**... etc. They are directly called by user written programs. You can refer most widely used R functions.

```
# Create a sequence of numbers from 32 to 44.
print(seq(32,44))

# Find mean of numbers from 25 to 82.
print(mean(25:82))

# Find sum of numbers frm 41 to 68.
print(sum(41:68))
```

When we execute above code, it produces following result:

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

User Defined Function

We can create user defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
                b <- i^2
                print(b)
                }
                        }
```

Calling a Function

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
```

```
for(i in 1:a) {
              b <- i^2
              print(b)
              }
                        }

# Call the function new.function supplying 6 as an argument.
new.function(6)
```

When we execute above code, it produces following result:

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
# Create a function without an argument.
new.function <- function() {
              for(i in 1:5) {
              print(i^2)
              }
                        }

# Call the function without supplying an argument.
new.function()
```

When we execute above code, it produces following result:

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Calling a Function with Argument Values bypositionandbyname

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.
```

```
new.function <- function(a,b,c) {
                result <- a*b+c
                print(result)
                                }

# Call the function by position of arguments.
new.function(5,3,11)

# Call the function by names of the arguments.
new.function(a=11,b=5,c=3)
```

When we execute above code, it produces following result:

```
[1] 26
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.
new.function <- function(a = 3,b =6) {
              result <- a*b
              print(result)
                                }

# Call the function without giving any argument.
new.function()

# Call the function with giving new values of the argument.
new.function(9,5)
```

When we execute above code, it produces following result:

```
[1] 18
[1] 45
```

Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
new.function <- function(a, b) {
          print(a^2)
          print(a)
          print(b)
                              }

# Evaluate the function without supplying one of the arguments.
new.function(6)
```

When we execute above code, it produces following result:

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```

String Manipulations

Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string with in 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.
- Single quote can not be inserted into a string starting and ending with single quote.

Example of Valid Strings

Following examples clarify the rules about creating a string in R.

```
a <- 'Start and end with single quote'
print(a)

b <- "Start and end with double quotes"
print(b)

c <- "single quote ' in between double quotes"
print(c)

d <- 'Double quotes " in between single quote'
print(d)
```

When the above code is run we get the following output:

```
[1] "Start and end with single quote"
[1] "Start and end with double quotes"
[1] "single quote ' in between double quote"
[1] "Double quote \" in between single quote"
```

Example of Invalid Strings

```
e <- 'Mixed quotes"
print(e)

f <- 'Single quote ' inside single quote'
print(f)

g <- "Double quotes " inside double quotes"
print(g)
```

When we run the script it fails giving below results.

```
...: unexpected INCOMPLETE_STRING

.... unexpected symbol
1: f <- 'Single quote ' inside

unexpected symbol
1: g <- "Double quotes " inside
```

R Programming

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

The basic syntax for paste function is :

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

Following is the description of the parameters used:

- **...** 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.

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 = ""))
```

When we execute above code, it produces following result:

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

Formatting numbers & strings - format function

Numbers and strings can be formatted to a specific style using **format** function.

syntax

The basic syntax for format function is :

```
format(x, digits, nsmall,scientific,width,justify = c("left", "right",
"centre", "none"))
```

Following is the description of the parame ers used:

- **x** is the vector input.
- **digits** is the total number of digits displayed.
- **nsmall** is the minimum number of digits to the right of the decimal point.
- **scientific** is set to TRUE to display scientific notation.
- **width** indicates the minimum width to be displayed by padding blanks in the beginning.
- **justify** is the display of the string to left, right or center.

Example

```
# Total number of digits displayed. Last digit rounded off.
result <- format(23.123456789, digits = 9)
print(result)

# Display numbers in scientific notation.
result <- format(c(6, 13.14521), scientific = TRUE)
print(result)

# The minimum number of digits to the right of the decimal point.
result <- format(23.47, nsmall = 5)
print(result)

# Format treats everything as a string.
result <- format(6)
print(result)

# Numbers are padded with blank in the beginning for width.
result <- format(13.7, width = 6)
print(result)
```

```
# Left justify strings.
result <- format("Hello",width = 8, justify = "l")
print(result)

# Justfy string with center.
result <- format("Hello",width = 8, justify = "c")
print(result)
```

When we execute above code, it produces following result:

```
[1] "23.1234568"
[1] "6.000000e+00" "1.314521e+01"
[1] "23.47000"
[1] "6"
[1] "  13.7"
[1] "Hello   "
[1] " Hello  "
```

Counting number of characters in a string - nchar function

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

Syntax

The basic syntax for nchar function is :

```
nchar(x)
```

Following is the description of the parameters used:

- **x** is the vector input.

Example

```
result <- nchar("Count the number of characters")
print(result)
```

When we execute above code, it produces following result:

```
[1] 30
```

Changing the case - toupper & tolower functions

These functions change the case of characters of a string.

Syntax

The basic syntax for toupper & tolower function is :

```
toupper(x)
tolower(x)
```

Following is the description of the parameters used:

- **x** is the vector input.

Example

```
# Changing to Upper case.
result <- toupper("Changing To Upper")
print(result)

# Changing to lower case.
result <- tolower("Changing To Lower")
print(result)
```

When we execute above code, it produces following result:

```
[1] "CHANGING TO UPPER"
[1] "changing to lower"
```

Extracting parts of a string - substring function

This function extracts parts of a String.

Syntax

The basic syntax for substring function is :

```
substring(x,first,last)
```

Following is the description of the parameters used:

- 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

```
# Extract characters from 5th to 7th position.
```

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

When we execute above code, it produces following result:

```
[1] "act"
```

Vectors

Vectors are the most basic R data objects and there are six types of atomic vectors. They are - logical, integer, double, complex, character and raw.

Vector Creation

Single Element Vector

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

```
# Atomic vector of type character.
print("abc");

# Atomic vector of type double.
print(12.5)

# Atomic vector of type integer.
print(63L)

# Atomic vector of type logical.
print(TRUE)

# Atomic vector of type complex.
print(2+3i)

# Atomic vector of type raw.
print(charToRaw('hello'))
```

When we execute above code, it produces following result:

```
[1] "abc"
[1] 12.5
[1] 63
[1] TRUE
[1] 2+3i
[1] 68 65 6c 6c 6f
```

Multiple elements vector

Using colon operator with numeric data

```
# Creating a sequence from 5 to 13.
v <- 5:13
print(v)

# Creating a sequence from 6.6 to 12.6.
v <- 6.6:12.6
print(v)

# If the final element specified does not belong to the sequence then it is discarded.
v <- 3.8:11.1
print(v)
```

When we execute above code, it produces following result:

```
[1]  5  6  7  8  9 10 11 12 13
[1]  6.6  7.6  8.6  9.6 10.6 11.6 12.6
[1]  3.8  4.8  5.8  6.8  7.8  8.8  9.8 10.8
```

Using sequence (Seq.) operator

```
# Create vector with elements from 5 to 9 incrementing by 0.4.
print(seq(5, 9, by=0.4))
```

When we execute above code, it produces following result:

```
 [1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
```

Using the c() function

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

```
# The logical and numeric values are converted to characters.
s <- c('apple','red',5,TRUE)
print(s)
```

When we execute above code, it produces following result:

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

Accessing Vector Elements

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

```
# Accessing vector elements using position.
t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")
u <- t[c(2,3,6)]
print(u)

# Accessing vector elements using logical indexing.
v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]
print(v)

# Accessing vector elements using negative indexing.
x <- t[c(-2,-5)]
print(x)

# Accessing vector elements using 0/1 indexing.
y <- t[c(0,0,0,0,0,0,1)]
print(y)
```

When we execute above code, it produces following result:

```
[1] "Mon" "Tue" "Fri"
[1] "Sun" "Fri"
[1] "Sun" "Tue" "Wed" "Fri" "Sat"
[1] "Sun"
```

Vector Manipulation

Vector arithmetic

Two vectors of same length can be added, substarcted, multiplied or divided giving the result as a vector output.

```
# Create two vectors.
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)

# Vector addition.
add.result <- v1+v2
print(add.result)

# Vector substraction.
sub.result <- v1-v2
print(sub.result)

# Vector multiplication.
multi.result <- v1*v2
print(multi.result)

# Vector division.
divi.result <- v1/v2
print(divi.result)
```

When we execute above code, it produces following result:

```
[1]  7 19  4 13  1 13
[1] -1 -3  4 -3 -1  9
[1] 12 88  0 40  0 22
[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)


add.result <- v1+v2

print(add.result)


sub.result <- v1-v2

print(sub.result)
```

When we execute above code, it produces following result:

```
[1]  7 19  8 16  4 22
[1] -1 -3  0 -6 -4  0
```

Vector element sorting

Elements in a vector can be sorted using **sort()** function.

```
v <- c(3,8,4,5,0,11, -9, 304)


# Sort the elements of the vector.

sort.result <- sort(v)

print(sort.result)


# Sort the elements in the reverse order.

revsort.result <- sort(v, decreasing = TRUE)

print(revsort.result)


# Sorting character vectors.

v <- c("Red","Blue","yellow","violet")

sort.result <- sort(v)

print(sort.result)


# Sorting character vectors in reverse order.

revsort.result <- sort(v, decreasing = TRUE)

print(revsort.result)
```

When we execute above code, it produces following result:

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

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

Lists

Lists are the R objects which contain elements of different types like - numbers, strings, vectors and another list inside it. A list can also contain a matrix or a functior as its elements. List is created using **list()** function.

Creating a List

Following is an example to create a list containing strings, numbers, vectors and a logical values

```
# Create a list containing strings, numbers, vectors and a logical values.
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

When we execute above code, it produces following result:

```
[[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

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

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow=2),
list("green",12.3))

# Gi e names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Show the list.
print(list_data)
```

When we execute above code, it produces following result :

```
$`1st_Quarter`
[1] "Jan" "Feb" "Mar"

$A_Matrix
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8

$A_Inner_list
$A_Inner_list[[1]]
[1] "green"

$A_Inner_list[[2]]
[1] 12.3
```

Accessing List Elements

Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

We continue to use the list in the above example

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow=2),
list("green",12.3))

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

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


# Access the first element of the list.
print(list_data[1])


# Access the thrid element. As it is also a list, all its elements will be printed.
print(list_data[3])


# Access the list element using the name of the element.
print(list_data$A_Matrix)
```

When we execute above code, it produces following result :

```
$`1st_Quarter`
[1] "Jan" "Feb" "Mar"

$A_Inner_list
$A_Inner_list[[1]]
[1] "green"

$A_Inner_list[[2]]
[1] 12.3

     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8
```

Manipulating List Elements

We can add, delete and update list elements as shown below. We can add and delete elements only at the end of a list. But we can update any element.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow=2),
list("green",12.3))


# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")


# Add element at the end of the list.
```

```
list_data[4] <- "New element"
print(list_data[4])

# Remove the last element.
list_data[4] <- NULL

# Print the 4th Element.
print(list_data[4])

# Update the 3rd Element.
list_data[3] <- "updated element"
print(list_data[3])
```

When we execute above code, it produces following result :

```
[[1]]
[1] "New element"

$
NULL

$`A Inner list`
[1] "updated element"
```

Merging Lists

You can merge many lists into one list by placing all the lists inside one list() function.

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

# Merge the two lists.
merged.list <- c(list1,list2)

# Print the merged list.
print(merged.list)
```

When we execute above code, it produces following result :

```
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] "Sun"

[[5]]
[1] "Mon"

[[6]]
[1] "Tue"
```

Converting List to Vector

A list can be converted to a vector so that the elements of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. To do this conversion we use the **unlist()** function. It takes the list as input and produces a vector.

```
# Create lists.
list1 <- list(1:5)
print(list1)

list2 <- list(10:14)
print(list2)

# Convert the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)

print(v1)
print(v2)
```

```
# Now add the vectors
result <- v1+v2
print(result)
```

When we execute above code, it produces following result :

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

[[1]]
[1] 10 11 12 13 14

[1] 1 2 3 4 5
[1] 10 11 12 13 14
[1] 11 13 15 17 19
```

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. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations.

A Matrix is created using the **matrix** function.

Syntax

The basic syntax for creating a matrix in R is:

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

Following is the description of the parameters used:

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

Example

Create a matrix taking a vector of numbers as input

```
# Elements are arranged sequentially by row.
M <- matrix(c(3:14), nrow=4, byrow=TRUE)
print(M)

# Elements are arranged sequentially by column.
N <- matrix(c(3:14), nrow=4, byrow=FALSE)
print(N)

# Define the 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))
print(P)
```

When we execute above code, it produces following resul :

```
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
[4,]   12   13   14
     [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
     col1 col2 col3
row1    3    4    5
row2    6    7    8
row3    9   10   11
row4   12   13   14
```

Accessing Elements of a Matrix

Elements of a matrix can be accessed by using the column and row index of the element. We consider the matrix P above to find the specific elements below.

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")
```

```
# Create the matrix.
P <- matrix(c(3:14), nrow=4, byrow=TRUE, dimnames=list(rownames,
colnames))

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

# Access the element at 2nd column and 4th row.
print(P[4,2])

# Access only the  2nd row.
print(P[2,])

# Access only the 3rd column.
print(P[,3])
```

When we execute above code, it produces following result:

```
[1] 5
[1] 13
col1 col2 col3
   6    7    8
row1 row2 row3 row4
   5    8   11   14
```

Matrix Computations

Various mathematical operations are performed on the matrices using the R operators. The result of the operation is also a matrix.

The dimensions numberofrowsandcolumns should be same for the matrices involved in the operation.

Matrix Addition & Subtraction

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow=2)
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow=2)
print(matrix2)
```

```
# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)

# Subtract the matrices
result <- matrix1 - matrix2
cat("Result of subtraction","\n")
print(result)
```

When we execute above code, it produces following result:

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of addition
     [,1] [,2] [,3]
[1,]    8   -1    5
[2,]   11   13   10
Result of subtraction
     [,1] [,2] [,3]
[1,]   -2   -1   -1
[2,]    7   -5    2
```

Matrix Multiplication & Division

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow=2)
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow=2)
print(matrix2)

# Multiply the matrices.
result <- matrix1 * matrix2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- matrix1 / matrix2
cat("Result of division","\n")
```

```
print(result)
```

When we execute above code, it produces following result:

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of multiplication
     [,1] [,2] [,3]
[1,]   15    0    6
[2,]   18   36   24
Result of division
     [,1]      [,2]      [,3]
[1,]  0.6      -Inf 0.6666667
[2,]  4.5 0.4444444 1.5000000
```

Factors:

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.

Example

```
# Create a vector as input.

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

print(data)

print(is.factor(data))


# Apply the factor function.

factor_data <- factor(data)

print(factor_data)

print(is.factor(factor_data))
```

When we execute above code, it produces following result:

```
 [1] "East"  "West"  "East"  "North" "North" "East"  "West"  "West"  "West"  "East"
"North"
[1] FALSE
 [1] East  West  East  North North East  West  West  West  East  North
Levels: East North West
[1] TRUE
```

Factors in Data Frame

On creating any data frame with a column of text data, R treats the text column as categorical data and creates factors on it.

```
# Create the vectors for data frame.
height <- c(132,151,162,139,166,147,122)
weight <- c(48,49,66,53,67,52,40)
gender <- c("male","male","female","female","male","female","male")

# Create the data frame.
input_data <- data.frame(height,weight,gender)
print(input_data)

# Test if the gender column is a factor.
print(is.factor(input_data$gender))

# Print the gender column so see the levels.
print(input_data$gender)
```

When we execute above code, it produces following result:

```
  height weight gender
1    132     48   male
2    151     49   male
3    162     66 female
4    139     53 female
5    166     67   male
6    147     52 female
7    122     40   male
[1] TRUE
[1] male   male   female female male   female male
Levels: female male
```

Changing the order of Levels

The order of the levels in a factor can be changed by applying the factor function again with new order of the levels.

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

# Create the factors

factor_data <- factor(data)

print(factor_data)


# Apply the factor function with required order of the level.

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

print(new_order_data)
```

When we execute above code, it produces following result:

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

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)
```

Following is the description of the parameters used:

- **n** is a integer giving the number of levels.
- **k** is a integer giving the number of replications.
- **labels** is a vector of labels for the resulting factor levels.

Example

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

print(v)
```

When we execute above code, it produces following result:

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

Dataframes:

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.

Create Data Frame

```
# Create the data frame.
emp.data <- 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")),
         stringsAsFactors=FALSE
                    )
# Print the data frame.
print(emp.data)
```

When we execute above code, it produces following result:

```
  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
```

Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

```
# Create the data frame.
emp.data <- 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")),
         stringsAsFactors=FALSE
                              )
# Get the structure of the data frame.
str(emp.data)
```

When we execute above code, it produces following result:

```
'data.frame':   5 obs. of  4 variables:
 $ emp_id    : int  1 2 3 4 5
 $ emp_name  : chr  "Rick" "Dan" "Michelle" "Ryan" ...
 $ salary    : num  623 515 611 729 843
 $ start_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-11" ...
```

Summary of Data in data frame

The statistical summary and nature of the data can be obtained by applying**summary()** function.

```
# Create the data frame.
emp.data <- 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")),
         stringsAsFactors=FALSE
                              )
# Print the summary.
print(summary(emp.data))
```

When we execute above code, it produces following result:

```
     emp_id     emp_name             salary        start_date
 Min.   :1   Length:5           Min.   :515.2   Min.   :2012-01-01
```

```
 1st Qu.:2   Class :character   1st Qu.:611.0   1st Qu.:2013-09-23
 Median :3   Mode  :character   Median :623.3   Median :2014-05-11
 Mean   :3                      Mean   :664.4   Mean   :2014-01-14
 3rd Qu.:4                      3rd Qu.:729.0   3rd Qu.:2014-11-15
 Max.   :5                      Max.   :843.2   Max.   :2015-03-27
```

Extract Data from data frame

Extract specific column from a data frame using column name.

```
# Create the data frame.
emp.data <- 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")),
         stringsAsFactors=FALSE
                              )
# Extract Specific columns.
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

When we execute above code, it produces following result:

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

Extract first two rows and all columns

```
# Create the data frame.
emp.data <- 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")),
         stringsAsFactors=FALSE
```

```
                          )
# Extract first two rows.
result <- emp.data[1:2,]
print(result)
```

When we execute above code, it produces following result:

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

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

```
# Create the data frame.
emp.data <- 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")),
         stringsAsFactors=FALSE
                          )

# Extract 3rd and 5th row with 2nd and 4th column.
result <- emp.data[c(3,5),c(2,4)]
print(result)
```

When we execute above code, it produces following result:

```
  emp_name start_date
3 Michelle 2014-11-15
5     Gary 2015-03-27
```

Expand data frame

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

Add column

Just add the column vector using a new column name.

```
# Create the data frame.
emp.data <- 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")),
        stringsAsFactors=FALSE
                        )

# Add the "dept" coulmn.
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
v <- emp.data
print(v)
```

When we execute above code, it produces following result:

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

Add row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the**rbind()** function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

```
# Create the first data frame.
emp.data <- 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")),
        dept=c("IT","Operations","IT","HR","Finance"),
        stringsAsFactors=FALSE
                        )
```

```
# Create the second data frame
emp.newdata <-     data.frame(
          emp_id = c (6:8),
          emp_name = c("Rasmi","Pranab","Tusar"),
          salary = c(578.0,722.5,632.8),
          start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),
          dept = c("IT","Operations","Fianance"),
          stringsAsFactors=FALSE
                                    )

# Bind the two data frames.
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

When we execute above code, it produces following result:

```
  emp_id emp_name salary start_date       dept
1      1     Rick 623.30 2012-01-01         IT
2      2      Dan 515.20 2013-09-23 Operations
3      3 Michelle 611.00 2014-11-15         IT
4      4     Ryan 729.00 2014-05-11         HR
5      5     Gary 843.25 2015-03-27    Finance
6      6    Rasmi 578.00 2013-05-21         IT
7      7   Pranab 722.50 2013-07-30 Operations
8      8    Tusar 632.80 2014-06-17   Fianance
```


Reshapng Data:

Data Reshaping in R is about changing the way data is organized into rows and columns. Most of the time data processing in R 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 the from the 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

We can join multiple vectors to create a data frame using the **cbind()** function. Also we can merge two data frames using **rbind()** function.

```
# Create vector objects.
city <- c("Tampa","Seattle","Hartford","Denver")
state <- c("FL","WA","CT","CO")
zipcode <- c(33602,98104,06161,80294)

# Combine above three vectors into one data frame.
addresses <- cbind(city,state,zipcode)

# Print a header.
cat("# # # # The First data frame\n")

# Print the data frame.
print(addresses)

# Create another data frame with similar columns
new.address <- data.frame(
   city = c("Lowry","Charlotte"),
   state = c("CO","FL"),
   zipcode = c("80230","33949"),
   stringsAsFactors=FALSE
)

# Print a header.
cat("# # # The Second data frame\n")

# Print the data frame.
print(new.address)

# Combine rows form both the data frames.
all.addresses <- rbind(addresses,new.address)
```

```
# Print a header.

cat("# # # The combined data frame\n")


# Print the result.

print(all.addresses)
```

When we execute above code, it produces following result:

```
# # # # The First data frame
     city       state zipcode
[1,] "Tampa"    "FL"  "33602"
[2,] "Seattle"  "WA"  "98104"
[3,] "Hartford" "CT"  "6161"
[4,] "Denver"   "CO"  "80294"
# # # The Second data frame
       city state zipcode
1     Lowry    CO   80230
2 Charlotte    FL   33949
# # # The combined data frame
       city state zipcode
1     Tampa    FL   33602
2   Seattle    WA   98104
3  Hartford    CT    6161
4    Denver    CO   80294
5     Lowry    CO   80230
6 Charlotte    FL   33949
```

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.

In the example below 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.

```
library(MASS)

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)
```

When we execute above code, it produces following result:

```
   bp  bmi npreg.x glu.x skin.x ped.x age.x type.x npreg.y glu.y skin.y ped.y
1  60 33.8       1   117     23 0.466    27     No       2   125     20 0.088
2  64 29.7       2    75     24 0.370    33     No       2   100     23 0.368
3  64 31.2       5   189     33 0.583    29    Yes       3   158     13 0.295
4  64 33.2       4   117     27 0.230    24     No       1    96     27 0.289
5  66 38.1       3   115     39 0.150    28     No       1   114     36 0.289
6  68 38.5       2   100     25 0.324    26     No       7   129     49 0.439
7  70 27.4       1   116     28 0.204    21     No       0   124     20 0.254
8  70 33.1       4    91     32 0.446    22     No       9   123     44 0.374
9  70 35.4       9   124     33 0.282    34     No       6   134     23 0.542
10 72 25.6       1   157     21 0.123    24     No       4    99     17 0.294
11 72 37.7       5    95     33 0.370    27     No       6   103     32 0.324
12 74 25.9       9   134     33 0.460    81     No       8   126     38 0.162
13 74 25.9       1    95     21 0.673    36     No       8   126     38 0.162
14 78 27.6       5    88     30 0.258    37     No       6   125     31 0.565
15 78 27.6      10   122     31 0.512    45     No       6   125     31 0.565
16 78 39.4       2   112     50 0.175    24     No       4   112     40 0.236
17 88 34.5       1   117     24 0.403    40    Yes       4   127     11 0.598
   age.y type.y
1     31     No
2     21     No
3     24     No
4     21     No
5     21     No
6     43    Yes
7     36    Yes
8     40     No
9     29    Yes
10    28     No
11    55     No
12    39     No
13    39     No
14    49    Yes
15    49    Yes
```

```
16    38     No
17    28     No
[1] 17
```

Melting and Casting

One of the most interesting aspects of R programming is about 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)
```

When we execute above code, it produces following result.

```
   type year period service incidents
1     A   60     60     127         0
2     A   60     75      63         0
3     A   65     60    1095         3
4     A   65     75    1095         4
5     A   70     60    1512         6
.............
.............
8     A   75     75    2244        11
9     B   60     60   44882        39
10    B   60     75   17176        29
11    B   65     60   28609        58
.............
.............
17    C   60     60    1179         1
18    C   60     75     552         1
19    C   65     60     781         0
.............
.............
```

Melt the data

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

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

```
print(molten.ships)
```

When we execute above code, it produces following result:

```
    type year  variable value
1      A   60    period    60
2      A   60    period    75
3      A   65    period    60
4      A   65    period    75
.............
.............
9      B   60    period    60
10     B   60    period    75
11     B   65    period    60
12     B   65    period    75
13     B   70    period    60
.............
.............
41     A   60   service   127
42     A   60   service    63
43     A   65   service  1095
.............
.............
70     D   70   service  1203
71     D   75   service     )
72     D   75   service  2051
73     E   60   service    45
74     E   60   service     0
75     E   65   service   789
.............
.............
101    C   70 incidents     6
102    C   70 incidents     2
103    C   75 incidents     0
104    C   75 incidents     1
105    D   60 incidents     0
106    D   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)
```

When we execute above code, it produces following result:

```
   type year period service incidents
1     A   60    135     190         0
2     A   65    135    2190         7
3     A   70    135    4865        24
4     A   75    135    2244        11
5     B   60    135   62058        68
6     B   65    135   48979       111
7     B   70    135   20163        56
8     B   75    135    7117        18
9     C   60    135    1731         2
10    C   65    135    1457         1
11    C   70    135    2731         8
12    C   75    135     274         1
13    D   60    135     356         0
14    D   65    135     480         0
15    D   70    135    1557        13
16    D   75    135    2051         4
17    E   60    135      45         0
18    E   65    135    1226        14
19    E   70    135    3318        17
20    E   75    135     542         1
```

Kelly Technologies

WhyLearnStatistics?

Youareprobablyaskingyourselfthequestion when andwherel willuse statistics. Ifyoureadanynewspaperorwatchtelevision or usetheInternet you willseestatisticalinformation.Therearestatisticsaboutcrime sports education politics and realestate.Typically when youreadanewspaperarticleorwatchanewsprogramontelevision you aregivensampleinformation.

Withthisinformation you maymakeadecisionaboutthecorrectnessofastatement claim or fact".Statisticalmethodscanhelpyoumakethebesteducatedguess".

Sinceyouwillundoubtedlybegivenstatisticalinformationatsomep ointinyourlife you needtoknowsometechniquestoanalyzetheinformationthoughtfully.Thi nkaboutbuyingahouseormanagingabudget.Thinkaboutyourchosenpr ofession.Thefieldsofeconomics business, psychology, education, biology, law, computer science, police science and earlychildhooddevelopmentrequireatleastonecourseinstatistics.

DescriptiveStatistics

1. MeasuresofCentralTendency
   a. Mean
   b. Median
   c. PercentilesorQuartiles

a. ArithmeticMean

Themeancanbecalculatedbyaveragingthedatapoints
Considerthedata:11122344444
Inthefirstwayofcalculatingthemeanweget:

$$\mu=\frac{1+1+1+2+2+3+4+4+4+4+4}{11}$$

=2:7
MeaninR
x=c(45666777777888910)
>mean(x)
[1]7

b. Median

Denotingorrelatingtoavalueorquantitylyingatthemidpointofafrequencydistributionofobservedvaluesorquantitiessuchthatthereisanequalnumberofpointsaboveorbelowit.

Num:1346791115
Median:(6+7)/2=3.5
MedianinR
>x=c(1346791115)
>median(x)
[1]6.5

c. Quartiles

Quartilesare25thand75thpercentilesrepresentedasQ1andQ3
The25thpointinthefrequencydistributioniscalledquartileone(Q1)
andthe75thpointiscalledQ3

SummaryfunctioninR

>x=c(1, 3 , 6 ,7, 9, 11, 15)
>median(x)
[1]6.5
>summary(x)
Min.1stQu.MedianMean3rdQu.Max.
1.003.756.50 7.009.5015.00

2. MeasuresofSpread
   a. Variance
   b. Standarddeviation
   c. Interquartilerange(Q3–Q1)

Variance

Thevarianceisinprincipletheaverageofthesquaresofthedeviations.
Thedeviationismeasuredasthedifferenceofeachobservationfromitsmean
StandardDeviation
Thesquarerootofvarianceisconsideredandstandarddeviation

$$\text{variance} = \sigma^2 = \frac{\sum (x_r - \mu)^2}{n}$$

$$\text{standard deviation} \quad \sigma = \sqrt{\frac{\sum (x_r - \mu)^2}{n}}$$

μ = mean

```
>x=c(1,3,4,6,7,9,11,15)
>var(x)          #Varianceofvectorx
[1]20.85714
>sd(x)           #StandarddeviationofvectorX
[1]4.566962
```

DisplayingData

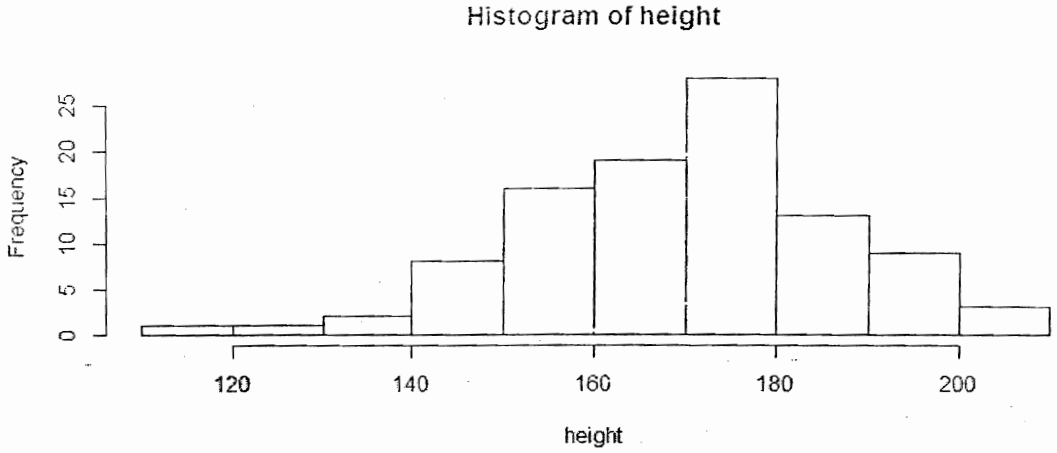
Astatisticalgraphisatoolthathelpsyoulearnabouttheshape ofthedistributionofasample.Thegraphcanbeamoreeffectivewayofpres entingdatathanamassofnumbersbec ausewecanseewheredatacluster sandwherethereareonlyafewdatavalues.Newspapersandthelnternet usegraphstoshow rendsandtoenablereaderstocomparefactsandfigur esquickly.

Histograms

Thehistogramisafrequentlyusedmethodfordisplayingthedistribu tionofcontinuousnumericaldata.Anadvantageofahistogramisthatitcan readilydisplaylargedatasets.Aruleofthumbistouseahistogramwhenthe datasetconsistsof100valuesormore.

The height of students in a class is measured in centimeters

```
> height = c( 182, 168, 172, 154, 174, 176, 193, 156, 157, 186, 143
182, 194, 187, 171,  178, 157, 156, 172, 157,
171,164,142,140,202,176,165,176,175,170,
169,153,169,158,208,185,157,147,160,173,164,182,175,165,194,1
78,178,186,165,180,174,169,173,199,163,160,172,177,165205,
193,158,180,167,165,183,171,191,191,152,148,176,155,156,177,1
80,186,167,174,171,148,153,136,199,161,150,181,166,147,168,
188,170,189,117,174,187,141,195,129,172)
```

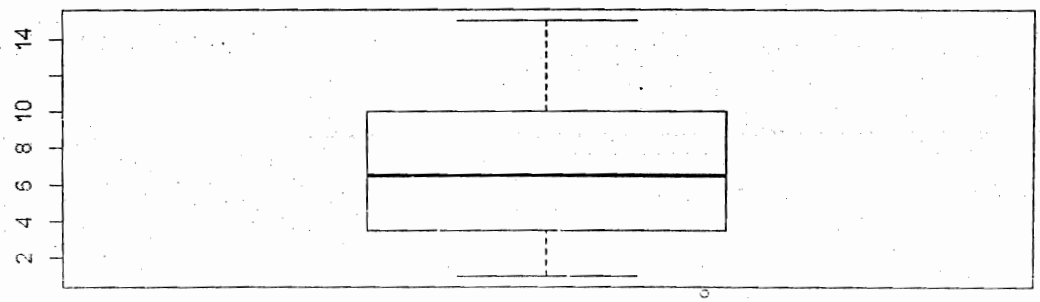


A histogram will help in understanding the shape or distribution of the data

Box Plots

The box plot or box-whisker plot gives a good graphical overall impression ofthe concentration of the data. It also shows how far from most of the data theextreme values are. In principle the box plot is constructed from five values: thesmallest value the first quartile the median the third quartile and the largestvalue.

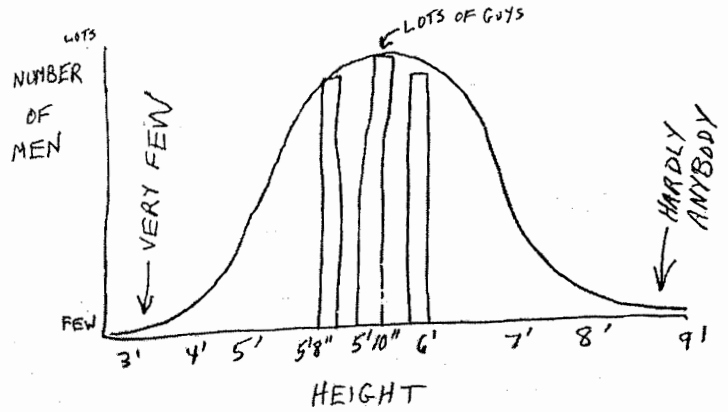
X = 1 1 2 2 4 6 6.8 7.2 8 8.3 9 10 10 11.5



The Normal Random Variable

The Normal Distribution

The distribution of height of students in a class (refer to the histogram above)



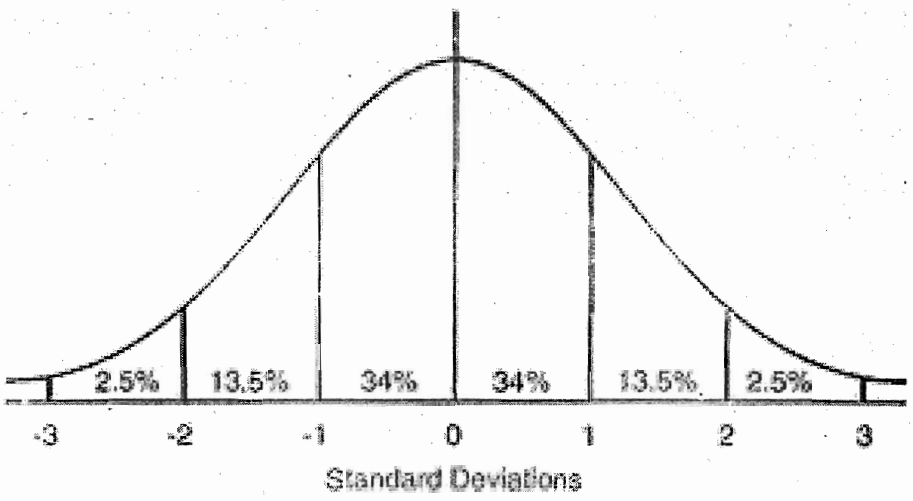
The properties of normal distribution

- 68% of data points within one standard deviation
- 95% of data pointswithin two standard deviations
- 99.7% of data pointswithin three standard deviations

The Standard Normal Distribution

The standard normal distribution is a normal distribution of standardized valueswhich are called z-scores with mean 0 and a standard deviation as 1.

$Z = (X - \mu)/\sigma$



Kelly Technologies

Application of z scores

Comparing scores of data points across different ranges

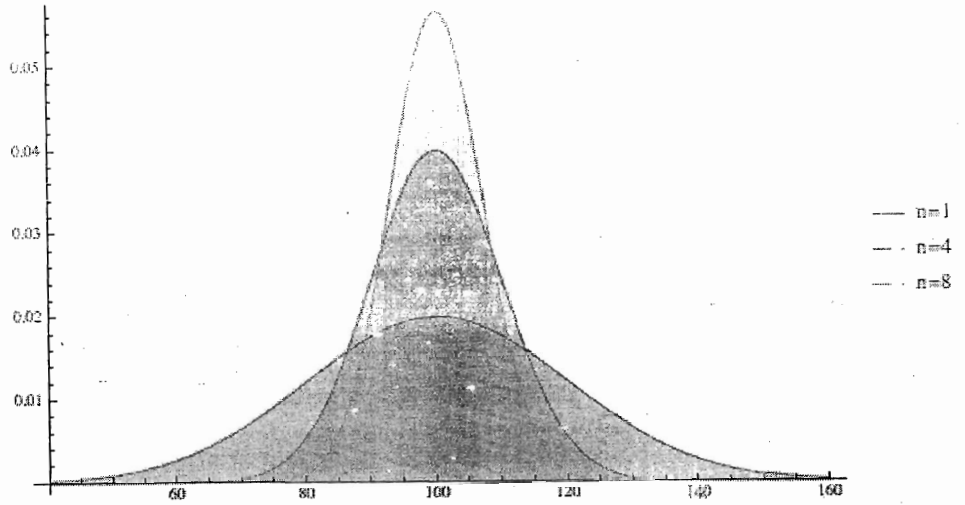
Inferential Statistics

Sampling methods

a. Simple random sampling – randomly choosing a few data points from a population without replacement
b. Systematic sampling – selecting a few observations from a population using any simple logic (every odd or even observation)
c. Stratified sampling - Selecting the sample in the same proportion of a particular group represented in a population (e.g. Males 60% and females 40%)

Sampling distribution

The distribution of a statistic (e.g. avg) of all possible samples of a population follows a normal distribution. This distribution of sample means of a population is called sampling distribution



The sampling distribution varies with the size of the sample
As the sample size increases the spread of the sampling distribution will decrease and exhibit leaner shape.

Estimation

Point estimation – an estimate of a quantity by analyzing the sample and inferencing it for the population
e.g. estimating the mean of a population by calculating the mean of a sample

mean of x x^ = sum(x) / length(x) -1
The inferential statistic mean represented as x bar is the point estimate of the mean of the population.
Note : the mean is calculated using length(x) – 1 due to degrees of freedom. The sample mean should be calculated using n-1whereas population mean can be calculated using N

Confidence Interval

A point estimator of the expectation of a measurement is the sample average ofthe variable that is associated with the measurement. A confidence interval isan interval of numbers that is likely to contain the parameter value.

Confidence Intervals for the Mean

Point estimate may not be enough to understand the characteristics of population as the mean of the sample may vary for every different sample. By computing a confidence interval we can find out the range of the means for different samples.

Hypothesis Testing

Hypothesis testing refers to the process of choosing between competing hypotheses (Null and alternate hypothesis) about a probability distribution based on observed data from the distribution.

Hypothesis: A claim or proposition set as an explanation of an occurrence or phenomenon based on an observation from a sample

E.g. Girls score better than boys

Representation - H: $Score_{Girls} > Score_{Boys}$

Aclaim or proposal is always considered as an alternatehypothesis and an opposite of the claim is called null hypothesis and always considered true unless proved otherwise

This process of validating a claim is called Hypothesis Testing

Practical Applications:

1. Comparison of means
   a. Single sample t-test (only one group)
   b. Two sample t-test (between two groups)
      i. Independent samples (one tailed or two tailed)
      ii. Paired samples
   c. Analysis of variance (more than two groups)
      i. F-test

Testing a Hypothesis:

Null hypothesis significance testing (NHST) was the creation of R. A. Fisher who also developed the analysisof variance (see the discussion below regarding the t distribution). In NHST there are

two competing hypotheses:the null hypothesis and the alternative hypothesis. The null hypothesis is presumptively true. A hypothesis is astatement about population parameters. The null hypothesis is always a statement of no difference no effect nochange or no relationship.

In the case of a t test the null hypothesis is that there is no difference between two means. For example ac ompany changed its customer service practices and wants to compare customer satisfaction after the changeto the customer satisfaction before the change. The null hypothesis is that there is no difference in customersatisfaction before and after the change. The alternative hypothesis is always a statement that there is adifference effect change or relationship.

In our current example the alternative hypothesis is either one-tailed (the customer satisfaction ratings arehigher after the change) or two-tailed (the customer satisfaction is different after the change—it could have gonedown). The null and alternative hypotheses are mutually exclusive (only one can be true) and exhaustive (one ofthem must be true).

The problem is that we rarely know the populationso we make inferences from samples. Our onlydecision is to reject or not to reject the null hypothesis. Based on our sample evidencewe conclude the null
Hypothesis should be rejected or should not be rejected. Of courseour decision could be incorrectso we hedgeour bets a little. We could make a Type I error (rejecting a true null hypothesis) or a Type II error (failing to rejecta false null hypothesis). We call the probability of Type I error alpha and the probability of Type II error beta.Examine the following figure to understand the relationships among these errorsconfidenceand statisticalpower.

	Population State of Affairs	
	H_0 is True	H_0 is False
Reject H_0	Incorrect Decision Type I Error α	Correct Decision Power(1 - β)
Do Not Reject H_0	Correct Decision Confidence(1 - α)	Incorrect Decision Type II Error β

Relationship of Type I and Type II errors to power and confidence

See that we can only make a Type I error when the null hypothesis is true and we can only make a Type IIerror when the null hypothesis is false. We control the probability of a Type I error by selecting our alpha level
(Traditionally .05) in advance of the hypothesis test. A commonly accepted level of statistical power is .95. Thismeans we are willing to make a Type II error up to 5% of the time. We cannot control a Type II error as directly
As we can control a Type I errorbut all things being equal the greater the sample size and the larger the effect size in thepopulation the more powerful our test will be.

The One-sample *t* Test

In the one-sample t testwe are comparing a sample mean to a known or hypothesized population mean. Wediscussed this test

briefly along with the topic of confidence intervals for the mean. As you will soon learnthe
paired-samplest test can also be seen as a special case of the one-sample t test.The formula for t as you will remember from our previous discussion is

$$t = \frac{\bar{x} - \mu}{\frac{s}{\sqrt{n}}}$$

The null hypothesis is that the expected mean difference between the sample mean and the populationmean is zero or in other words that the expected value of the sample mean is equal to the population mean.We compare this value to the t distribution with n – 1 degrees of freedom to determine a right-tailed left-tailedor two-tailed probability. As you will recallthet.test function also reports a confidence interval for the mean
Difference.

As an examplelet us generate a simple random sample of 100 observations from a normal distribution with amean of 50 and a standard deviation of 10.

```
> rnorm1 = rnorm(100 50 10)
>head(rnorm1)
[1] 45.89727 62.34311 59.36383 36.38846 54.87434
56.45252

>mean(rnorm1)
[1] 50.5038
```

The mean of the rnorm1 is 50.50 which slightly is higher than the expected mean of 50. Is this difference significant?

The use of single sample t-test is to test the hypothesis that the sample came from a population with a mean of 50

Alternate hypothesis - H_A: mean(x) != 50

Kelly
Technologies

Null hypothesis - H_0 : mean(x) = 50

```
>t.test(rnorm1 mu = 50)

        One Sample t-test

data:  rnorm1
t = 0.503df = 99 p-value = 0.6161
alternative hypothesis: true mean is not equal to
50
95 percent confidence interval:
 48.51649 52.49111
sample estimates:
  mean of x
  50.5038
```

Interpretation of the result:

1. The p-value is 0.61 which is greater than the
2. 95% confidence interval of the mean is 48.51 to 52.49

Conclusion: Since the p-value is higher than the alpha (0.05) we reject the alternate hypothesis and accept the null hypothesis that "the sample indeed came from a population with a mean of 50".

Kelly Technologies

Exploratory Data Analysis and Visualization

Reading Data into R

Table File

A data table can resides in a text file. The cells inside the table are separated by blank characters. Here is an example of a table with 4 rows and 3 columns.

```
100   a1   b1
200   a2   b2
300   a3   b3
400   a4   b4
```

Now copy and paste the table above in a file named "mydata.txt" with a text editor. Then load the data into the workspace with the function read.table.

```
> mydata = read.table("mydata.txt")  # read text file
> mydata                             # print data frame
  V1 V2 V3
1 100 a1 b1
2 200 a2 b2
3 300 a3 b3
4 400 a4 b4
```

For further detail of the function read.table, please consult the R documentation.

```
> help(read.table)
```

CSV File

The sample data can also be in **comma separated values** (CSV) format. Each cell inside such data file is separated by a special character, which usually is a comma, although other characters can be used as well.

The first row of the data file should contain the column names instead of the actual data. Here is a sample of the expected format.

```
Col1,Col2,Col3
100,a1,b1
200,a2,b2
300,a3,b3
```

After we copy and paste the data above in a file named "mydata.csv" with a text editor, we can read the data with the function read.csv.

```
> mydata = read.csv("mydata.csv")  # read csv file
> mydata
  Col1 Col2 Col3
1  100   a1   b1
2  200   a2   b2
3  300   a3   b3
```

In various European locales, as the comma character serves as the decimal point, the functionread.csv2 should be used instead. For further detail of the read.csv and read.csv2 functions, please consult the R documentation.

```
> help(read.csv)
```

Excel File

Quite frequently, the sample data is in **Excel** format, a.id needs to be imported into R prior to use. For this, we can use the function read.xls from the gdata package. It reads from an Excel spreadsheet and returns a data frame. The following shows how to load an Excel spreadsheet named "mydata.xls". This method requires Perl runtime to be present in the system.

```
> library(gdata)                  # load gdata package
> help(read.xls)                  # documentation
> mydata = read.xls("mydata.xls")  # read from first sheet
```

Alternatively, we can use the function loadWorkbook from the XLConnect package to read the entire workbook, and then load the worksheets with rea Worksheet. The XLConnect package requires Java to be pre-installed.

```
> library(XLConnect)              # load XLConnect package
> wk = loadWorkbook("mydata.xls")
> df = readWorksheet(wk, sheet="Sheet1")
Or you could also make use of the following xlsx package to import excel files
into R
# read in the first worksheet from the workbook myexcel.xlsx
# first row contains variable names
library(xlsx)
mydata <- read.xlsx("c:/myexcel.xlsx", 1)

# read in the worksheet named mysheet
mydata <- read.xlsx("c:/myexcel.xlsx", sheetName = "mysheet")
```

How to Connect to Database in R

This page will show you how to connect to database in R and return data. This requires the package RODBC.

Notes:

All of the examples on this page show how to connect to an Oracle database using SQL. The SQL is within the double quotes and the R code is outside of the double quotes.

You do not need to establish the connection to the database each time you query it. You can connect once and then continue to use that connection every time you want to query the database. I repeat creating the connection in each example below so that the examples stand on their own.

I've found that believeNRows=FALSE is necessary when using Windows 7 but not Windows XP. I haven't been able to test using other operating systems. You may not need this. Without it I am able to connect but get the following error when querying the database.

```
Error in .Call(C_RODBCFetchRows, attr(channel, "handle_ptr"), max, buffsize, :
 negative length vectors are not allowed
```

First Example: Return All Data from One Table

The first example shows how to connect to database in R and queries the database DATABASE and returns all of the data (this is specified using the * in SQL) from the table DATATABLE. The table is preceded by the database schema SCHEMA and separated by a period. Each of the words in all caps needs within the query needs to be replaced so that the query applies to your database.

If you are using a 64 bit Windows OS you may need to add

```
# Load RODBC package
library(RODBC)

# Create a connection to the database called "channel"
channel <- odbcConnect("DATABASE", uid="USERNAME", pwd="PASSWORD",
believeNRows=FALSE)

# Check that connection is working (Optional)
odbcGetInfo(channel)

# Find out what tables are available (Optional)
Tables <- sqlTables(channel, schema="SCHEMA")

# Query the database and put the results into the data frame "dataframe"
 dataframe <- sqlQuery(channel, "
 SELECT *
 FROM
 SCHEMA.DATATABLE")
```

Second Example: Return Only Specific Fields

The second example shows how to connect to database in R and query the database DATABASE and pull only the specified fields from the table DATATABLE.

Note that loading the RODBC package and creating a connection does not have to be repeated if they were done in the first example. I include them in later examples so that each example is complete.

```
# Load RODBC package
library(RODBC)

# Create a connection to the database called "channel"
channel <- odbcConnect("DATABASE", uid="USERNAME", pwd="PASSWORD",
believeNRows=FALSE)

# Find out what fields are available in the table (Optional)
# as.data.frame coerces the data into a data frame for easy viewing
Columns <- as.data.frame(colnames(sqlFetch(channel, "SCHEMA.DATATABLE")))

# Query the database and put the results into the data frame "dataframe"
 dataframe <- sqlQuery(channel, "
 SELECT SCHOOL,
 STUDENT_NAME,
 FROM
```

```
SCHEMA.DATATABLE")
```

Third Example: Return Only Specific Fields and Records

The third example shows how to connect to database in R and query the database DATABASE and pull only the specified fields from the table DATATABLE, excluding records that don't meet the criteria specified (SCHOOL_YEAR='2011-12').

Note that the field that is being used to filter records does not have to be included in the returned results.

```
# Load RODBC package
library(RODBC)

# Create a connection to the database called "channel"
channel <- odbcConnect("DATABASE", uid="USERNAME", pwd="PASSWORD",
believeNRows=FALSE)

# Query the database and put the results into the data frame "dataframe"
dataframe <- sqlQuery(channel, "
 SELECT SCHOOL,
 STUDENT_NAME
 FROM
 SCHEMA.DATATABLE
 WHERE
 SCHOOL_YEAR='2011-12'")
```

Fourth Example: Joining Two Tables and Returning Only Specific Fields and Records

The fourth example queries the database DATABASE and pulls only the specified fields from the tables DATATABLE and DATATABLE_TWO, excluding records that don't meet the criteria specified (SCHOOL_YEAR='2011-12' and PARENT_AGENCY_NAME='Pine Tree District').

Note that this example also renames DATATABLE to DT and DATATABLE_TWO to DTTWO. Renaming isn't necessary but is often used in SQL queries.

Note also that this example demonstrates renaming of fields within the SQL query using "as".

```
# Load RODBC package
library(RODBC)

# Create a connection to the database called "channel"
channel <- odbcConnect("DATABASE", uid="USERNAME", pwd="PASSWORD",
believeNRows=FALSE)

# Query the database and put the results into the data frame "dataframe"
 dataframe <- sqlQuery(channel, "
```

```
SELECT
DT.SCHOOL_YEAR,
DTTWO.DISTRICT_NAME AS DISTRICT,
DTTWO.SCHOOL_NAME AS SCHOOL,
DT.GRADE_LEVEL AS GRADE,
DT.ACTL_ATT_DAYS AS ACTUAL_DAYS,
DT.POSS_ATT_DAYS AS POSSIBLE_DAYS
FROM
(SCHEMA.DATATABLE DT INNER JOIN SCHEMA.DATATABLE_TWO DTTWO
ON (DT.SCHOOL_YEAR = DTTWO.SCHOOL_YEAR AND
DT.SCHOOL_NUMBER = DTTWO.SCHOOL_CODE))
WHERE
DT.SCHOOL_YEAR = '2011-12' AND
DTTWO.SCHOOL_NAME = 'Pine Tree Elementary School'")
```

Fifth Example: Using a Parameter Set in R to Return Only Specific Records

The fifth example queries the database DATABASE and pulls only the specified fields from the tables DATATABLE and DATATABLE_TWO, excluding records that don't meet the criteria specified using the parameter CYR).

Note that the below must specify that sep="" or it will return nothing. That is because the default separator for paste is a space, meaning that your SQL query would be looking for records where SCHOOL_YEAR=" 2010-11 ".

```
# Load RODBC package
library(RODBC)

# Create a connection to the database called "channel"
channel <- odbcConnect("DATABASE", uid="USERNAME", pwd="PASSWORD",
believeNRows=FALSE)

# Parameter
CYR <- "2010-11"

# Query the database and put the results into the data frame "dataframe"
 dataframe <- sqlQuery(channel, paste("
 SELECT
 YEAR,
 SCHOOL_YEAR,
 DISTRICT_CODE,
 GRADE_LEVEL
 FROM SCHEMA.DATAFRAME
 WHERE
 ((SCHEMA.DATAFRAME.SCHOOL_YEAR='", CYR, "'))
 ", sep=""
 ))
```

Datatypes in R

There are several basic R data types that are of frequent occurrence in routine R calculations. Though seemingly innocent, they can still deliver surprises. Instead of chewing through the language specification, we will try to understand them better by direct experimentation with the R code.

Numeric

Decimal values are called **numerics** in R. It is the default computational data type. If we assign a decimal value to a variable x as follows, x will be of numeric type.

```
> x = 10.5        # assign a decimal value
> x               # print the value of x
[1] 10.5
> class(x)        # print the class name of x
[1] "numeric"
```

Furthermore, even if we assign an integer to a variable k, it is still being saved as a numeric value.

```
> k = 1
> k               # print the value of k
[1] 1
> class(k)        # print the class name of k
[1] "numeric"
```

The fact that k is *not* an integer can be confirmed with the is.integer function.

```
> is.integer(k)  # is k an integer?
[1] FALSE
```

Integer

In order to create an **integer** variable in R, we invoke the as.integer function. We can be assured that y is indeed an integer by applying the is.integer function.

```
> y = as.integer(3)
> y               # print the value of y
[1] 3
> class(y)        # print the class name of y
[1] "integer"
> is.integer(y)  # is y an integer?
[1] TRUE
```

Incidentally, we can coerce a numeric value into an integer with the same as.integer function.

```
> as.integer(3.14)     # coerce a numeric value
[1] 3
```

And we can parse a string for decimal values in much the same way.

```
> as.integer("5.27")  # coerce a decimal string
[1] 5
```

On the other hand, it is erroneous trying to parse a non-decimal string.

```
> as.integer("Joe")    # coerce an non-decimal string
[1] NA
Warning message:
NAs introduced by coercion
```

Often, it is useful to perform arithmetic on logical values. Like the C language, TRUE has the value 1, while FALSE has value 0.

```
> as.integer(TRUE)    # the numeric value of TRUE
[1] 1
> as.integer(FALSE)   # the numeric value of FALSE
[1] 0
```

Complex

A **complex** value in R is defined via the pure imaginary value *i*.

```
> z = 1 + 2i      # create a complex number
> z               # print the value of z
[1] 1+2i
> class(z)        # print the class name of z
[1] "complex"
```

The following gives an error as −1 is not a complex value.

```
> sqrt(-1)        # square root of -1
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

Instead, we have to use the complex value −1 + 0*i*.

```
> sqrt(-1+0i)     # square root of -1+0i
[1] 0+1i
```

An alternative is to coerce −1 into a complex value.

```
> sqrt(as.complex(-1))
[1] 0+1i
```

Logical:

A **logical** value is often created via comparison between variables.

```
> x = 1; y = 2    # sample values
> z = x > y       # is x larger than y?
> z               # print the logical value
[1] FALSE
> class(z)        # print the class name of z
[1] "logical"
```

Standard logical operations are "&" (and), "|" (or), and "!" (negation).

```
> u = TRUE; v = FALSE
> u & v           # u AND v
[1] FALSE
> u | v           # u OR v
[1] TRUE
> !u              # negation of u
[1] FALSE
```

Further details and related logical operations can be found in the R documentation.

```
> help("&")
```

Character

A **character** object is used to represent string values in R. We convert objects into character values with the as.character() function:

```
> x = as.character(3.14)
> x             # print the character string
[1] "3.14"
> class(x)      # print the class name of x
[1] "character"
```

Two character values can be concatenated with the paste function.

```
> fname = "Joe"; lname ="Smith"
> paste(fname, lname)
[1] "Joe Smith"
```

However, it is often more convenient to create a readable string with the sprintf function, which has a C language syntax.

```
> sprintf("%s has %d dollars", "Sam", 100)
[1] "Sam has 100 dollars"
```

To extract a substring, we apply the substr function. Here is an example showing how to extract the substring between the third and twelfth positions in a string.

```
> substr("Mary has a little lamb.", start=3, stop=12)
[1] "ry has a l"
```

And to replace the first occurrence of the word "little" by another word "big" in the string, we apply the sub function.

```
> sub("little", "big", "Mary has a little lamb.")
[1] "Mary has a big lamb."
```

Handling Missing Values

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

Testing for Missing Values

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

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

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

Recoding Values to Missing

```
# recode 99 to missing for variable v1

# select rows where v1 is 99 and recode column v1

mydata$v1[mydata$v1==99] <- NA
```

Excluding Missing Values from Analyses

Arithmetic functions on missing values yield missing values.

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

mean(x) # returns NA

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

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

```
# list rows of data that have missing values

mydata[!complete.cases(mydata),]
```

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

```
# create new dataset without missing data

newdata <- na.omit(mydata)
```

Imputing the missing values can give better results than discarding the samples containing any missing value. Imputing does not always improve the predictions, so please check via cross-validation. Sometimes dropping rows or using marker values is more effective.

Missing values can be replaced by the mean, the median or the most frequent value using the `strategy` hyper-parameter. The median is a more robust estimator for data with high magnitude variables which could dominate results (otherwise known as a 'long tail').

Kelly Technologies

Linear Regression

Linear regression – LR is a superivised learning technique to predict a quantitative response variable.

Simple linear regression:

Simple linear regression lives up to its name: it is a very straight forward approach for predicting a quantitative response Y on the basis of a single- regressionpredictor variable X. It assumes that there is approximately a linearrelationship between X and Y. Mathematically, we can write this linearrelationship as

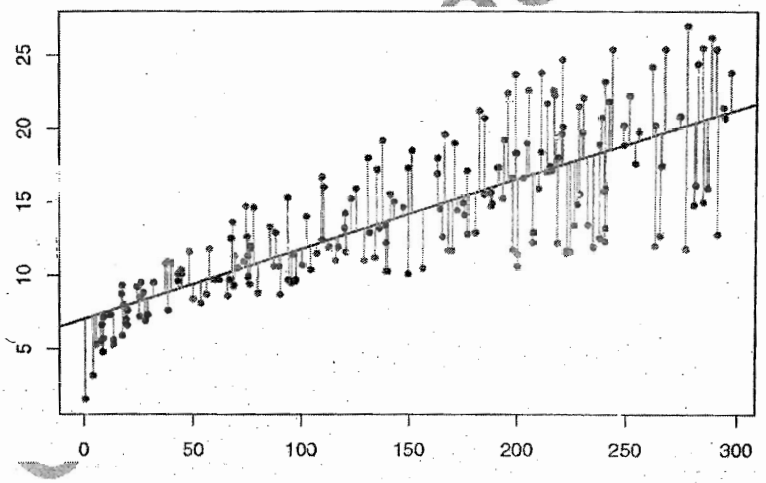
$$Y \approx \beta_0 + \beta_1 X.$$

The task of the regression exercise is to find out the coefficients $\beta_0$andβ_1
The coefficient: are unknowns and we use the data points to estimate the coefficients

$$(x1, y1), (x2, y2), \ldots, (xn, yn)$$

Ourgoal is to obtain coefficient estimates ˆ β0 and ˆ β1 such that the linear model that fits the available data well—that is, so that yi≈ ˆβ0 + ˆ β1xi for i =1, . . . , n. In other words, we want to find an intercept ˆ β0 and a slope ˆ β1 suchthat the resulting line is as close as possible to the n (number of observations) data points.

There are a number of ways of measuring closeness. However, by far themost common approach involves minimizing the *least squares* criterion referred to as **Sum of least squares**



Accuracy of the Coefficient Estimates:

Recall that we assume that the true relationship between X andY takes the form Y = f(X) +ϵ for some unknown function f, where ϵis a mean-zero random error term. If f is to be approximated by a linearfunction, then we can write this relationship as

$$Y = \beta_0 + \beta_1 X + \epsilon.$$

Linear Regression

Here β0 is the intercept term—that is, the expected value of Y when X = 0,and β1 is the slope—the average increase in Y associated with a one-unitincrease in X.The error term is a catch-all for what we miss with thissimple model: the true relationship is probably not linear, there may beother variables that cause variation in Y, and there may be measurementerror. We typically assume that the error term is independent of X.

Hypothesis tests to measure the significance of the estimates

H0: There is no relationship between X and Y

Versus the alternative hypothesis

Ha: There is some relationship between X and Y

Mathematically, this corresponds to testing

$$H_0 : \beta_1 = 0$$

Versus

$$H_a : \beta_1 != 0,$$

A t test is performed on the coefficient of the variable to check if the coefficient is significantly far from zero (which means there is a relationship between the target variable and the independent variable)

	Coefficient	Std. error	t-statistic	p-value
Intercept	7.0325	0.4578	15.36	< 0.0001
TV	0.0475	0.0027	17.67	< 0.0001

In the above example the result from the t test is provided in the last column (p-value) and the value is lesser than 0.05(significance at a 95% confidence level), which means that the variable coefficient is significantly far from zero and a relationship exists between the independent and target variable

Accuracy of the Model:

Need to quantify the extent to which the model fits the data

The quality of a linear regression fit is typically assessedusing two related quantities: the *residual standard error* (RSE) and the R^2statistic.

Residual (ϵ): a residual is the deviation (difference) of the predicted value and the actual value for each data point and represented by ϵ

Kelly Technologies

Linear Regression

Residual Standard Error: a residual standard error is standard deviation of the residuals, in a simple terms the standard deviation of error while predicting.

Note: the n-2 is due to the

$$\mathrm{RSE} = \sqrt{\frac{1}{n-2}\mathrm{RSS}} = \sqrt{\frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

degrees of freedom

The RSE should be as small as possible, the prediction error at the each data point is as minimum as possible for a good predictive model

$R^2$2 Statistic: $R^2$is the proportion of variance in the target variable explained by the model. The value of an R^2lies between 0 and 1. The model with R^2value closer to 1 is the best model for a given set of data and variables.

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

TSS – total um of squares
RSS – residual sum of squares

Multiple Linear Regression

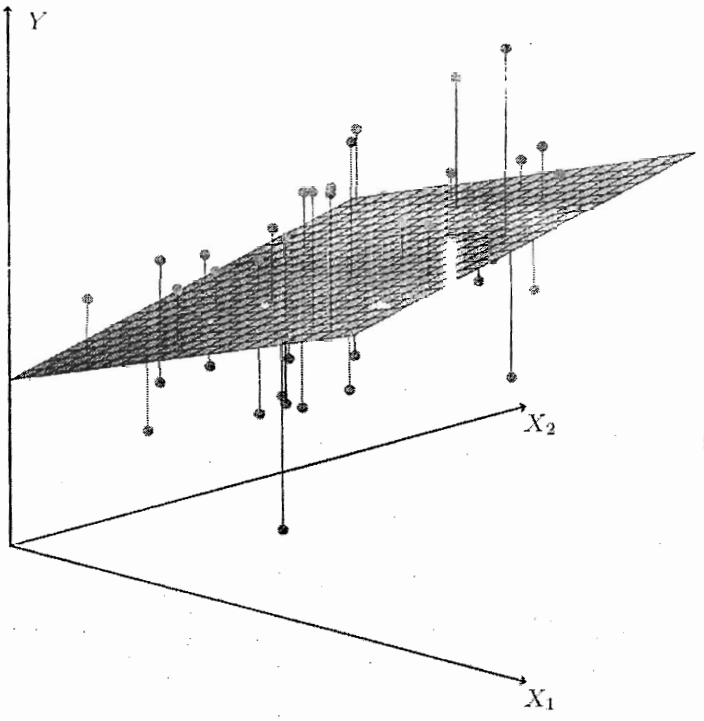
Simple linear regression is a useful approach for predicting a response on the, However, in practice we often have morethan one predictor.

How can we extend our analysis of the advertising data inorderto accommodate these twoadditional predictors on the basisof a single predictor variable.
We can dothis by giving each predictor a separate slope coefficient in a single model.In general, suppose that we have p distinct predictors. Then the multiplelinear regression model takes the form

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon$$

Estimating the coefficients takes a slightly different approach for a multiple linear regression, the effect of each variable on a target variable is arrived at by holding the rest of the independent variables as constant and a partial derivative is applied at each variable to find out the coefficients.
Assume all the points in the form of a n dimensional space, now the regression line is nothing but the hyper plane that minimizes the distance from each point (observations) to the plane.

Linear Regression



	Coefficient	Std. error	t-statistic	p-value
Intercept	2.939	0.3119	9.42	< 0.0001
TV	0.046	0.0014	32.81	< 0.0001
radio	0.189	0.0086	21.89	< 0.0001
newspaper	-0.001	0.0059	-0.18	0.8599

The coefficients of each variable is obtained and a t test is preformed on each variable individually, like in the simple linear regression the variable relationship is measured with the help of the coefficient and the individual p-values to get the significance of each variable.
In the above example it is evident that the variables TV, radio are significant since the p-values are less than 0.05 and the variable is newspaper is not (p-value is 0.8599 >> 0.05). So this variable can be excluded from the model.

Prediction with qualitative variables:
So far we have only considered quantitative (continuous or ordinal) as independent variables in our regression, how to include qualitative(categorical or dichotomous) variables in a regression?

Dummy variable: a dummy variables are new variables created from a categorical variable with n-1 levels in the variable
e.g. if a categorical product variable has three distinct values('a','b','c') then two dummay variables will be created i.e. Product_b, product_c and the level "a" is considered as the reference level.

Linear Regression

Product_b or product_c will be having either 0 or 1 based on the value of product variable, if both product_b and product_c are 0 then by default the product_c value will be 1. Which means product_a is dependent on product_b and product_c values. When both the values are known the third variable value can be found out. That is the reason only two dummy variables are created for a variable with three levels.

Note: n-1 dummy levels will be created for a categorical variable where n is the number of distinct levels in the variable.

Performing multiple linear regression in R

We use the insurance data set for this exercise. The objective is to predict the insurance charges for a given customer. The snap shot of the data is given below.

```
> insurance <- read.csv("insurance.csv", stringsAsFactors = TRUE)
> head(insurance)
  age    sex    bmi children smoker    region   charges
1  19 female 27.900        0    yes southwest 16884.924
2  18   male 33.770        1     no southeast  1725.552
3  28   male 33.000        3     no southeast  4449.462
4  33   male 22.705        0     no northwest 21984.471
5  32   male 28.880        0     no northwest  3866.855
6  31 female 25.740        0     no southeast  3756.622
```

The target variable is charges column and the rest of the variables are independent variables.

The linear regression function lm() is part of the base R package

First linear regression model in r

```
> model1 = lm(charges ~ ., data=insurance)
>summary(model1)

Call:
lm(formula = charges ~ ., data = insurance)

Residuals:
     Min       1Q   Median       3Q      Max
-11304.9  -2848.1   -982.1   1393.9  29992.8

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     -11938.5      987.8 -12.086  < 2e-16 ***
age                256.9       11.9  21.587  < 2e-16 ***
sexmale           -131.3      332.9  -0.394 0.693348
bmi                339.2       28.6  11.860  < 2e-16 ***
children           475.5      137.8   3.451 0.000577 ***
smokeryes        23848.5      413.1  57.723  < 2e-16 ***
regionnorthwest   -353.0      476.3  -0.741 0.458769
regionsoutheast  -1035.0      478.7  -2.162 0.030782 *
regionsouthwest   -960.0      477.9  -2.009 0.044765 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
1

Residual standard error: 6062 on 1329 degrees of freedom
```

```
ciple R-squared:  0.7509,    Adjusted R-squared:  0.7494
statistic: 500.8 on 8 and 1329 DF,  p-value: < 2.2e-16
```

The first argument for the lm function is the formulae of the regression model, target variable followed by "~" and a "." , the . here means that R should include all the other variables in the model.

The other way of passing the variables to the formulae is to include one by one with a "+" sign as mentioned below.

```
> model2 = lm(charges ~ age+sex, data=insurance )
>summary(model2)

Call:
lm(formula = charges ~ age + sex, data = insurance)

Residuals:
   Min     1Q Median     3Q    Max
 -8821  -6947  -5511   5443  48203

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2343.62     994.35   2.357   0.0186 *
age           258.87      22.47  11.523   <2e-16 ***
sexmale      1538.83     631.08   2.438   0.0149 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
1

Residual standard error: 11540 on 1335 degrees of freedom
Multiple R-squared:  0.09344,   Adjusted R-squared:  0.09209
F-statistic:  68.8 on 2 and 1335 DF,  p-value: < 2.2e-16
```

Now only age and sex are included in the model building (training), the model can be interpreted by printing the summary of the model using a summary function.

Validating the variable coefficients:

As a first step the coefficients along with the p-values need to be checked, recollect the significance of the variables discussed in the previous session, all the variables with p-values less than 0.05 are significant and can be considered in the model.

For easy interpretation R also prints the significance of the variables with "*" at the end of summary, the more the number of starts "***" are highly significant and "*" is marginally significant. The variables with a "." Are just below the level of significance 0.05

The variables with p-values greater than 0.05 can be dropped from the model in the next iteration.

Note: some variables with p-values greater than 0.05 can also be included in the model if the variables are more meaningful in terms of business.

Validating the importance of the model.

Let us look at the R2 of model1 and model2. The R2 for model 1 is 0.75 whereas the model2 R2 is 0.09. The model with higher R2 is always better. So model1 will be preferred to model2.
In the model1 the variable sex doesn't seems to be significant the p-value is almost 0.7, higher than the 0.05 limit. So this variable can be excluded from the model.

Model assumptions and validation:

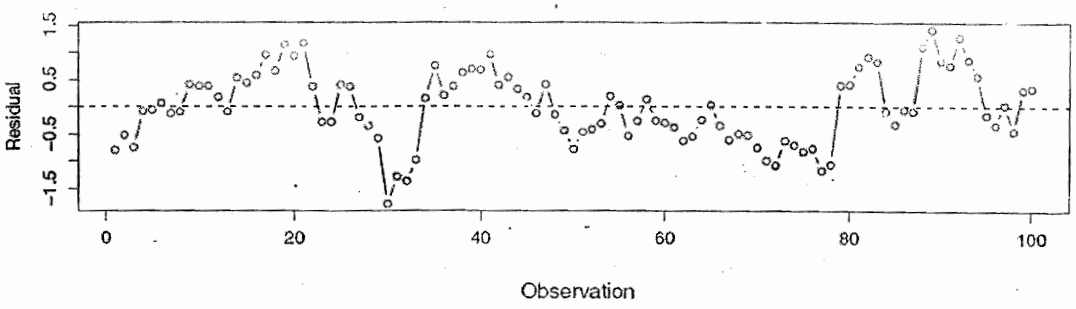
Assumptions of a linear regression model.

1. Linearity and additive relationship
2. Independence of errors (no correlation of error terms, or no auto correlation)
3. Equal variance or homoscedasticity (non-constant variance or no heteroscedasticity)
4. Normality of errors (errors are normally distributed with a mean zero)
5. No multi collinearity (no relationship between independent variables)
6. Outliers and leverage points

Check has to be made to make sure that the above mentioned assumptions are true for a model

Most of the above mentioned points has to be checked with the help of plots or some statistics.

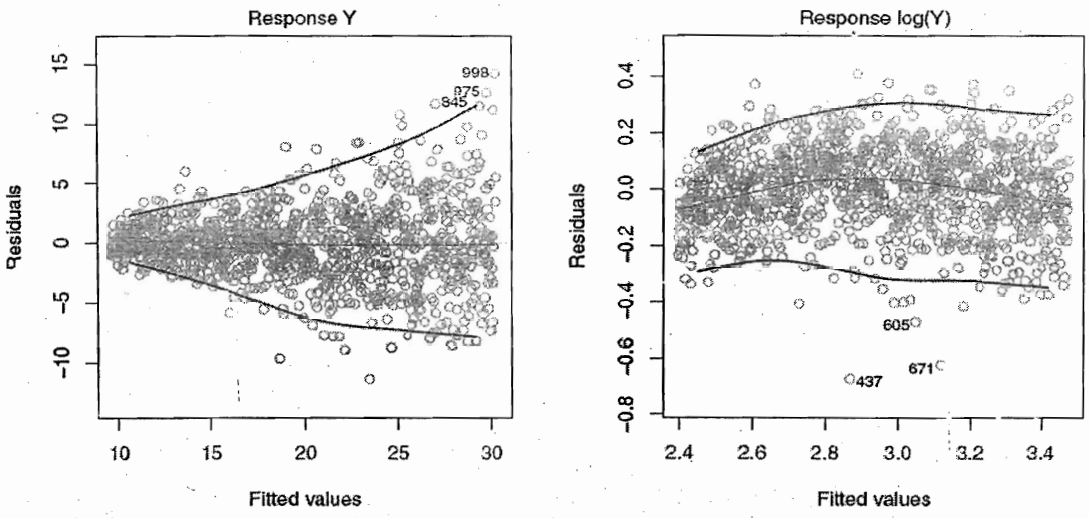
1. Linearity: the relationship between the target variable and independent variable has to be a linear relationship. This can be checked by plotting a scatter plot of all independent variables and target variable or the scatter plot of residuals and fitted values can also help.
2. Independence of errors: if the error term of an observation n depends on the or related to the error of observation n-1 then this phenomenon is called auto correlation. This phenomenon generally occurs in a time series data. The error terms will stay positive for a few observations and slowly tend to zero and move to negative like a sine wave. This should not happen because the error of the current observation is correlated with the previous one.



In the above graph the error terms has negative sign for first few observations and slowly approaches zero and moves positive. This phenomenon is called autocorrelation. The plot of residuals vs observation numbers should reflect a random occurrences of positive negative and neutral errors.

3. Equal variance: the errors or residuals should be normally distributed with a mean zero and constant variance. Which means the magnitude of the errors should remain same across different fitted values.

While predicting the car sale value of a resale car if the errors or residuals are small when the car predicted prices are low and residuals magnitude gradually increase as the price of the car increases, which means the variance of errors are non-constant. This phenomenon is called heteroscedasticity. This can also be verified with a residual vs. fitted values scatter plot.



The graph on the left exhibits heteroscedasticity as one can see the errors are small when the fitted values are small and gradually increases for higher values of fitted values. The shape of the errors replicate a funnel as highlighted by the boundary lines at top and bottom. On the right the magnitude of errors are almost constant thru the fitted values and no funnel like shape of residuals is visible.

4. Normality of errors:

   The errors should be normally distributed with a mean. This can be checked using a QQ plot

5. Multicollinearity: no two independent variables have a correlation between them. Correlation between all the predictor variables can be computed and verified for any strong correlation.
6. Outliers: an outlier is an observation which is far from the rest in terms of target variable. Generally any observation beyond three

standard deviations is considered an outlier. The impact of the outlier can be assessed by excluding that observation from the training samples.

7. High Leverage: high leverage points are the ones that are far from the rest of the observations in terms of independent variables. It is measured using cooks distance. Cooks distance can be computed using the function cooks.distance(). Any observation with a cooks distance of 4/(n-k-1) where n is the number of training observations and k is the number of independent variables in the model, needs special attention.

   Note: there could be few observations which can exhibit both an outlier and leverage, these are extreme observations and can have significant impact on the model. These observations need to be dropped from the training sample.

Kelly Technologies

Logistic Regression

Logistic regression enables us to predict a dichotomous (0, 1) outcome while overcoming the problems withlinear regression. Examples of binary outcomes include the presence and absence ofdefects, attendance and absenteeism, and student retention versus dropout.

There is a problem with using correlation and regression to predict a (0, 1) outcome.Typically, we will have predicted values less than 0 and greater than 1, and of course these are theoreticallyimpossible. The logistic curve, on the other hand, has a very nice property of being asymptotic to 0 and 1 andalways lying between 0 and 1. Thus, logistic regression avoids the problems of linear regression fo predicting binary outcomes.

We can convert a probability to odds, and we use odds in logistic regression. If p is the probability of success,the odds in favor of success, o, are:

$$\text{odds} = \frac{p}{q} = \frac{p}{1-p}$$

Odds can also be converted to probabilities quite easily. If the odds in favor of an event are 5:1, then the
Probability of the event is 5/6. Note that odds, unlike probabilities, can exceed 1. For example, if the probability ofrain is .25, the odds in favor of rain are .25 / .75 = .33, but the odds against rain are .75 / .25 = 3.00.In logistic regression we work with a term called the logit, which is the natural logarithm of the odds. Wedevelop a linear combination of predictors and an intercept term to predict the logit:

$$ln(odas) = b_0 + b_1x_1 + b_2x_2 + \ldots + b_kx_k$$

The *logistic regression model* links the predictor variables to probabilitie through the equation

$$p = f(\alpha + \beta_1x_1 + \beta_2x_2 + \cdots + \beta_kx_k) = \frac{\exp(\alpha + \beta_1x_1 + \beta_2x_2 + \cdots + \beta_kx_k)}{1 + \exp(\alpha + \beta_1x_1 + \beta_2x_2 + \cdots + \beta_kx_k)}$$

Here is how to do the logistic regression in R. We use the glm procedure, and identify the model as a binarylogistic one.

The flight delay dataset is used to demonstrate logistic regression in R.

```
> model1 <- glm(delay ~ . -dayweek , data=file1,
family="binomial")
```

The summary statement is used to print the summary of the model

```
summary(model1)
```

```
Call:
glm(formula = delay ~ . - dayweek, family = "binomial", data = file1)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
```

Logistic Regression

```
-1.2649  -0.4891  -0.3917  -0.2785   8.4904

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) 300.877691 145.821502   2.063  0.03908 *
schedtime    -0.035328   0.002753 -12.832  < 2e-16 ***
carrierDH    -1.818262   0.727360  -2.500  0.01243 *
carrierDL    -2.819720   1.034474  -2.726  0.00642 **
carrierMQ    -1.670131   1.022816  -1.633  0.10250
carrierOH    -2.357087   1.210823  -1.947  0.05157 .
carrierRU    -0.636841   0.552063  -1.154  0.24868
carrierUA    -2.124645   1.106178  -1.921  0.05477 .
carrierUS    -1.842966   1.040781  -1.771  0.07660 .
deptime       0.035737   0.002748  13.007  < 2e-16 ***
destJFK      26.659044  12.863097   2.073  0.03822 *
destLGA      28.659870  13.770194   2.081  0.03741 *
distance     -1.793195   0.862054  -2.080  0.03751 *
originDCA    53.783054  25.735968   2.090  0.03664 *
originIAD    79.905543  38.084516   2.098  0.03590 *
weather1     16.403204 433.378198   0.038  0.96981
daymonth      0.025602   0.010201   2.510  0.01208 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1301.42  on 1319  degrees of freedom
Residual deviance:  792.46  on 1303  degrees of freedom
AIC: 826.46

Number of Fisher Scoring iterations: 15
```

The interpretation is similar to a linear regression model, the significance of the individual variables can be found out from the p-values

The AIC Akike information criterion is used to pick the best model, for the same number of training examples, the model which gives the least AIC is preferred as the best possible model.

Unlike linear regression model there is no R2 in case of a logistic regression model. The model performance can be assessed by few parameters.

Predicting probabilities on future cases using R :

```
test$pred<- predict(model1, newdata=test, type="response")
    delay       pred
4       0 0.07562183
7       0 0.07018478
8       0 0.07169213
10      0 0.11824747
12      0 0.06169601
14      0 0.05813846
```

A matrix of predicted vs actual classes can provide the accuracy of the model, this matrix is called as confusion matrix

1. Accuracy of the model (total correct predictions/total number of observations)
2. Recall
3. Precision

Kelly Technologies

Logistic Regression

Confusion Matrix:

A confusion matrixcontains information about actual and predicted classifications done by a classification system. Performance of such systems is commonly evaluated using the data in the matrix. The following table shows the confusion matrix for a two class classifier. The entries in the confusion matrix have the following meaning in the context of our study:

- a is the number of correct predictions that an instance s negative,
- b is the number of incorrect predictions that an instance is positive,
- c is the number of incorrect of predictions that an instance negative and
- d is the number of correct predictions that an instance is positive.

		Predicted	
		Negative	Positive
Actual	Negative	a	b
	Positive	c	d

The accuracy (AC) is the proportion of the total number of predictions that were correct. It is determined using the equation

$$AC = \frac{a+d}{a+b+c+d}$$

The recall or true positive rate (TP) is the proportion of positive cases that were correctly identified, as calculated using the equation:

$$TP = \frac{d}{c+d}$$

Precision (P) is the proportion of the predicted positive cases that were correct, as calculated using the equation:

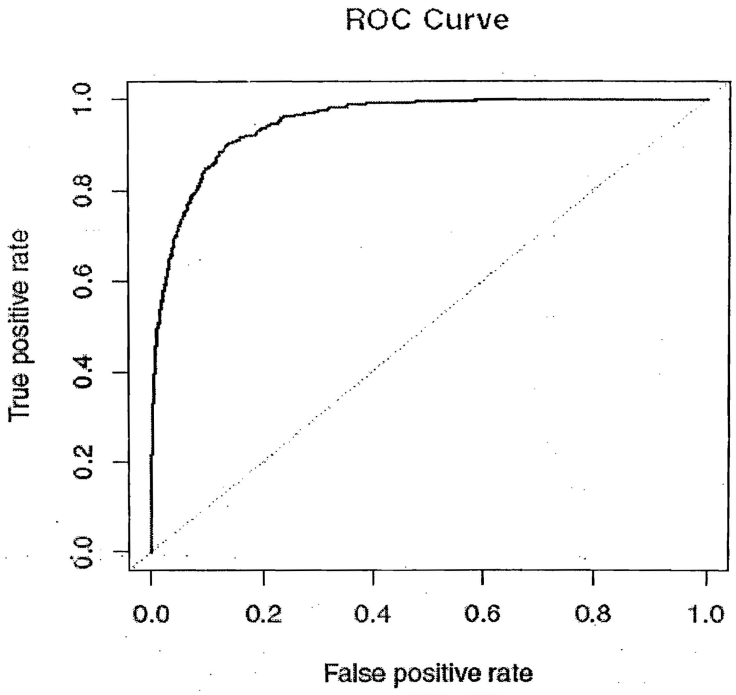
$$P = \frac{d}{b+d}$$

ROC curve

Description : In a Receiver Operating Characteristic (ROC) curve the true positive rate (Sensitivity) is plotted in function of the false positive rate (1-Specificity) for different cut-off points. Each point on the ROC plot represents a sensitivity/specificity pair corresponding to a particular decision threshold. A test with perfect discrimination (no overlap in the two distributions) has a ROC plot that passes through the upper left corner (100% sensitivity, 100% specificity). Therefore the closer the ROC plot is to the upper left corner, the higher the overall accuracy of the test.

Kelly Technologies

Logistic Regression



An ROC curve demonstrates several things:

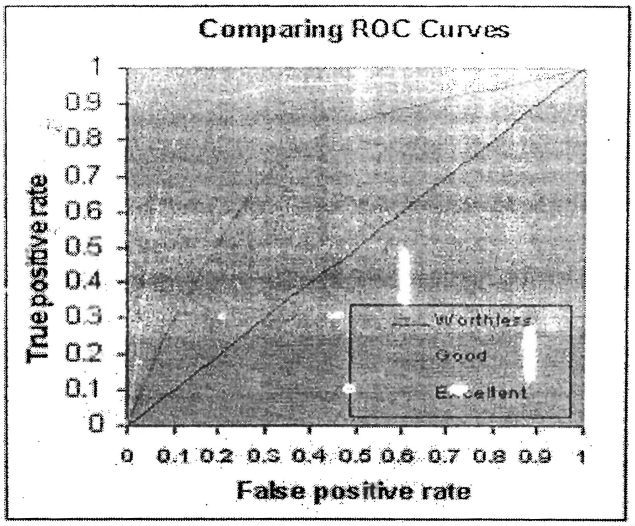
1. It shows the trade off between sensitivity and specificity (any increase in sensitivity will be accompanied by a decrease in specificity).
2. The closer the curve follows the upper-left border of the ROC space, the more accurate the test.
3. The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test.
4. The slope of the tangent line at a cut point gives the likelihood ratio (LR) for that value of the test. 5. The area under the curve is a measure of accuracy.

Judging a ROC curve: The graph below shows three ROC curves representing excellent, good, and worthless tests plotted on the same graph. The accuracy of the test depends on how well the test separates the group being tested into those with and without the disease in question. Accuracy is measured by the area under the ROC curve. An area of 1 represents a perfect test; an area of .5 represents a worthless test. A rough guide for classifying the accuracy of a diagnostic test is the traditional academic point system:

- .90-1 = excellent (A)
- .80-.90 = good (B)
- .70-.80 = fair (C)
- .60-.70 = poor (D)
- .50-.60 = fail (F)

Kelly Technologies

Logistic Regression



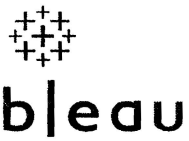
Plot ROC curve in R:

Need to install the package ROCR before executing the below steps

```
library(ROCR)

pred<- prediction(predict(model1, newdata=test,
type="response"), test$delay)
perf<- performance(pred,"tpr","fpr")
plot(perf)
```

OFFERED OTHER COURSES



Microsoft ERP

Microsoft Dynamics AX (2012)

Technical • Finance • Trade & Logistics

Microsoft Dynamic CRM

Kelly Technologies

Flat No: 212, 2nd Floor, Annapurna Block, Aditya Enclave, Ameerpet, Hyderabad, AP.
Ph No: 040 6462 6789, 0998 570 6789 info@kellytechno.com, www.kellytechno.com.