R Snippets

Release 0.1

Shailesh

Nov 02, 2017

Contents

This is not an R tutorial or book. This is a handy collection of snippets of R code for achieving various statistical analysis and visualization tasks. If you wish to learn R from beginning, try the books listed on www.bookdown.org. However, if you are familiar with R and want to quickly refresh your memory or look for code for specific tasks, you can refer to these notes.

CHAPTER 1

R Scripting Environment

1.1 Getting Help

help on a function:

```
> help(rnorm)
```

Examples of using a function:

```
> example(rnorm)
```

Starting global help:

```
> help.start()
```

1.2 Workspace

List of objects in workspace:

```
> ls()
```

We can also use `objects` function:

```
> objects()
```

Clear all variables from workspace:

```
> rm(list=ls())
```

Clear a variable from workspace:

```
> rm(a)
```

Clearing the console:

```
> cat("\014")
```

1.3 Arithmetic

Addition:

```
> 2 + 3
[1] 5
```

Subtraction:

```
> 2 - 3
[1] -1
```

Negation:

```
> -8
[1] -8
```

Multiplication:

```
> 2 * 3
[1] 6
```

Division:

```
> 8 / 3
[1] 2.666667
> 8 / -3
[1] -2.666667
```

Integer division:

```
> 8 %/% 3
[1] 2
> -8 %/% 3
[1] -3
> 8 %/% -3
[1] -3
> -8 %/% -3
[1] 2
```

Remainder:

```
> 8 %% 3
[1] 2
> -8 %% 3
[1] 1
> 8 %% -3
[1] -1
```

```
> -8 %% -3
[1] -2
```

Let us combine integer division and remainder:

```
> 2 * 3 + 2
[1] 8
> -3 * 3 + 1
[1] -8
> (-3) * (-3) + (-1)
[1] 8
> (2) * (-3) + (-2)
[1] -8
```

Exponentiation:

```
> 10^1
[1] 10
> 11^2
[1] 121
> 11^3
[1] 1331
```

Some more complicated expressions

```
> 10^2  + 36
```

Compounded interest over a number of years:

```
> 1000 * (1 + 10/100)^5
[1] 1610.51
```

1.4 Variables

Assignment:

```
> a=4
> a<-4
> 3 -> a
> a
[1] 3
```

Display:

```
> a
```

Use:

```
> a*5
> a=a+10
> a<-a+10
```

Assignment through function:

```
> assign("x", c(1.4, 2.3, 4.4))
> x
[1] 1.4 2.3 4.4
```

Assignments in other direction:

```
2 -> x
```

1.5 Data Types

A vector:

```
> x <- c(1, 2, 3)
> x
[1] 1 2 3
> x[1]
[1] 1
> x[1:2]
[1] 1 2
```

A sequence:

```
> x <- 1:4
> x
[1] 1 2 3 4
```

A matrix:

```
> x <- matrix(1:4, nrow=2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

An array:

```
> x <- array(1:16, dim=c(2,2,4))
> x[1,1,1]
[1] 1
> x[1,2,1]
[1] 3
> x[1,2,3]
[1] 11
```

A character vector or string:

```
x <- "hello"
```

A list:

```
> x <- list(a=4, b=2.4, c="hello")
> x$a
[1] 4
> x$b
[1] 2.4
```

```
> x$c
[1] "hello"
```

A data frame:

```
> frm <- data.frame(x=c(1,2,3), y=c(21, 20, 23), z=c("a", "b", "c"))
> frm$x
[1] 1 2 3
> frm$y
[1] 21 20 23
> frm$z
[1] a b c
Levels: a b c
> frm[1,]
  x  y z
1 1 21 a
```

Extended arithmetic with vectors:

```
> 11 ^ c(1,2,3,4)
[1]    11   121  1331 14641
> c(1,2,3,4) ^ 3
[1]  1  8 27 64
> c(1,2,3) * c(2,3,4)
[1]  2  6 12
```

CHAPTER 2

Core Data Types

2.1 Vectors

Creating a vector (via concatenation):

```
> b = c(3,4,5, 8, 10)
```

Sequence of integers:

```
> v = c(-3:4)
> v <- -3:4
```

Concatenating multiple ranges:

```
> v = c(1:4, 8:10)
```

Accessing elements:

```
> b[2]
```

Indexing is 1-based.

Creating a vector over a range with a step size:

```
> v2  = seq(from=0, to=1, by=0.25)
```

This will include both 0 and 1. It will have 5 elements.

Colon has higher precedence:

```
> 2*1:4
[1] 2 4 6 8
```

Backward sequence:

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

Sequence with a specified length from a start:

```
> seq(from=1, by=.5, length=4)
[1] 1.0 1.5 2.0 2.5
```

Sequence with a specified length from an end:

```
> seq(to=1, by=.5, length=4)
[1] -0.5  0.0  0.5  1.0
```

Computing the step size of a sequence automatically:

```
> seq(from=4, to=-4, length=5)
[1]  4  2  0 -2 -4
```

Summing two vectors:

```
> v3 = v1 + v2
```

Summing elements of a vector:

```
> s = sum(v1)
```

Cumulative sum:

```
> x <- c(1,3, 2, -1, 4,-6)
> cumsum(x)
[1] 1 4 6 5 9 3
```

Product of all elements:

```
> x <- c(1,3, 2, -1, 4,-6)
> prod(x)
[1] 144
```

Sorting:

```
> sort(c(3,2,1))
```

Sub-vector:

```
> v = c(1:10)
> v = [1:4]
```

Assigning names to vector entries:

```
> x <- 1:4
> names(x) <- c("a", "b", "c", "d")
> x
a b c d
1 2 3 4
> x["a"]
a
1
```

```
> x["e"]
<NA>
  NA
```

Empty vectors:

```
> e <- numeric()
> e
numeric(0)
> e <- character()
> e
character(0)
> e <- complex()
> e
complex(0)
> e <- logical()
> e
logical(0)
```

Increasing size of a vector:

```
> e <- numeric()
> e[3]
[1] NA
> e[3] <- 10
> e[3]
[1] 10
> e
[1] NA NA 10
```

Truncating a vector:

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x <- x[2:4]
> x
[1] 2 3 4
```

Reversing a vector:

```
> rev(1:3)
[1] 3 2 1
```

First few elements of a vector:

```
> head(1:8, n=4)
[1] 1 2 3 4
```

Last few elements of a vector:

```
> tail(1:8, n=4)
[1] 5 6 7 8
```

Interleaving two vectors:

```
> x <- c(1,2,3)
> y <- c(4,5,6)
```

```
> z <- c(rbind(x,y))
>
> z
[1] 1 4 2 5 3 6
```

Inner product of two vectors:

```
> c(1, 0, 1) %*% c(-1, 0, -1)
     [,1]
[1,]   -2
```

Outer product of two vectors:

```
> v1 <- 1:3
> v2 <- 2:4
> v1 %o% v2
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    4    6    8
[3,]    6    9   12
> outer(v1,v2)
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    4    6    8
[3,]    6    9   12
```

Outer sum of two vectors:

```
> outer(v1,v2, '+')
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    4    5    6
[3,]    5    6    7
```

Outer subtraction of two vectors:

```
> outer(v1,v2, '-')
     [,1] [,2] [,3]
[1,]   -1   -2   -3
[2,]    0   -1   -2
[3,]    1    0   -1
```

Evaluating a 2-variable function f(x,y) over a grid of x and y values:

```
> x <- seq(0, 1, by=0.5)
> x
[1] 0.0 0.5 1.0
> y <- seq(0, 1, by=0.2)
> f <- function(x, y) x*y /(x+y+1)
> outer(x,y, f)
     [,1]       [,2]      [,3]      [,4]      [,5]      [,6]
[1,]    0 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000
[2,]    0 0.05882353 0.1052632 0.1428571 0.1739130 0.2000000
[3,]    0 0.09090909 0.1666667 0.2307692 0.2857143 0.3333333
```

Constructing the multiplication table:

```
> outer(2:11, 1:10)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    2    4    6    8   10   12   14   16   18    20
 [2,]    3    6    9   12   15   18   21   24   27    30
 [3,]    4    8   12   16   20   24   28   32   36    40
 [4,]    5   10   15   20   25   30   35   40   45    50
 [5,]    6   12   18   24   30   36   42   48   54    60
 [6,]    7   14   21   28   35   42   49   56   63    70
 [7,]    8   16   24   32   40   48   56   64   72    80
 [8,]    9   18   27   36   45   54   63   72   81    90
 [9,]   10   20   30   40   50   60   70   80   90   100
[10,]   11   22   33   44   55   66   77   88   99   110
```

By default a vector is neither a row vector or a column vector. It is a line vector.

Coercing a vector into a row vector:

```
> v <- 1:3
> v
[1] 1 2 3
> t(v)
     [,1] [,2] [,3]
[1,]    1    2    3
```

Coercing into a column vector::

> t(t(v)) [,1]

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

Alternative way:

```
> dim(v) <- c(3,1)
> v
     [,1]
[1,]    1
[2,]    2
[3,]    3
> dim(v) <- c(1,3)
> v
     [,1] [,2] [,3]
[1,]    1    2    3
```

Converting a vector into a row vector:

```
> rbind(v)
```

Converting a vector into a column vector:

```
> cbind(v)
```

Repeating a vector:

```
> v <- 1:4
> rep(v, 4)
 [1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
```

Controlling the final length:

```
> rep(v, 4, length.out=10)
 [1] 1 2 3 4 1 2 3 4 1 2
```

Repeating each element few times then repeating the whole sequence:

```
> rep(v, times=3, each=2)
 [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
```

Separate repetition count for each element:

```
> rep(v, c(1,2,3,4))
 [1] 1 2 2 3 3 3 4 4 4 4
```

Membership test:

```
> 1 %in% 1:4
[1] TRUE
> 1 %in% 2:4
[1] FALSE
> 3 %in% 1:4
[1] TRUE
> 3 %in% c(1:2, 4:8)
[1] FALSE
> 'a' %in% c('a', 'b', 'c')
[1] TRUE
> 'aa' %in% c('a', 'aab', 'c')
[1] FALSE
> 'aa' %in% c('a', 'aa', 'c')
[1] TRUE
```

2.1.1 Index Vectors

Logical index vectors:

```
> x
[1]   1   4  NA   5 NaN
> is.na(x)
[1] FALSE FALSE  TRUE FALSE  TRUE
> y <- x[!is.na(x)]
> y
[1] 1 4 5
```

Integral index vectors:

```
> x <- sample(1:10, 10)
> x
 [1]  4  1  3  7  9 10  5  2  8  6
> x[c(1,4,7,10)]
[1] 4 7 5 6
> x[seq(1,10, 2)]
[1] 4 3 9 5 8
> x[c(1:4, 1:4)]
[1] 4 1 3 7 4 1 3 7
> paste(c("x","y")[rep(c(1,2,2,1), times=4)], collapse='')
[1] "xyyxxyyxxyyxxyyx"
```

Excluding some indices:

```
> x
 [1]  8  4  3  7 10  5  9  6  2  1
> x[-c(1,4,8:10)]
[1]  4  3 10  5  9
```

Accessing vector entries by their names:

```
> x <- 1:4
> names(x) <- c("a", "b", "c", "d")
> x[c("c", "b")]
c b
3 2
```

2.2 Matrices

Creating a matrix by specifying rows:

```
> m = matrix(c(1:12), nrow=3)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The entries in matrix are read from the data vector in column major order.

Creating a matrix by specifying columns:

```
> m = matrix(c(1:12), ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Matrix dimensions:

```
> m <- matrix(1:6, nrow=2)
> nrow(m)
[1] 2
> ncol(m)
[1] 3
> dim(m)
[1] 4 3
```

Accessing an element:

```
> m[1,1]
[1] 1
```

Accessing first row:

```
> m[1,]
[1] 1 5 9
```

Accessing first column:

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

Accessing first and second rows:

```
> m[1:2,]
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
```

Accessing a sub-matrix (1st 2 rows, last 2 columns):

```
> m[1:2, 2:3]
     [,1] [,2]
[1,]    5    9
[2,]    6   10
```

Computing the sum of all elements:

```
> sum(m)
[1] 78
```

Sum over each row:

```
> rowSums(m)
[1] 15 18 21 24
```

Sum over each column:

```
> colSums(m)
[1] 10 26 42
```

Computing the mean of all elements:

```
> mean(m)
[1] 6.5
```

Mean over each row:

```
> rowMeans(m)
[1] 5 6 7 8
```

Mean over each column:

```
> colMeans(m)
[1]  2.5  6.5 10.5
```

Subtracting the mean from each column of a matrix:

```
> A <- matrix(c(3, 2, -1, 2, -2, .5, -1, 4, -1), nrow=3)
> colMeans(A)
[1] 1.3333333 0.1666667 0.6666667
```

```
> B <- scale(A, scale=F)
> round(colMeans(B), digits=2)
[1] 0 0 0
> round(B, digits=2)
      [,1]  [,2]  [,3]
[1,]  1.67  1.83 -1.67
[2,]  0.67 -2.17  3.33
[3,] -2.33  0.33 -1.67
attr(,"scaled:center")
[1] 1.3333333 0.1666667 0.6666667
```

Binding columns:

```
> cbind(1:4, 2:5, 3:6, 4:7)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    3    4    5
[3,]    3    4    5    6
[4,]    4    5    6    7
```

Binding rows:

```
> rbind(1:4, 2:5, 3:6, 4:7)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    3    4    5
[3,]    3    4    5    6
[4,]    4    5    6    7
```

Series of row and column binds:

```
> m <- cbind(1:4, 2:5)
> m <- cbind(m, 3:6)
> m <- rbind(m, 9:11)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    3    4
[3,]    3    4    5
[4,]    4    5    6
[5,]    9   10   11
```

An all zeros matrix:

```
> matrix(0, 2,3)
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
```

An all ones matrix:

```
> matrix(1, 2,3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
```

An identity matrix:

```
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Diagonal matrix:

```
> diag(1:3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
> diag(c(3, 10, 11))
     [,1] [,2] [,3]
[1,]    3    0    0
[2,]    0   10    0
[3,]    0    0   11
```

Diagonal matrix with additional columns:

```
> diag(c(3, 10, 11), ncol=5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    0    0    0    0
[2,]    0   10    0    0    0
[3,]    0    0   11    0    0
```

Diagonal elements get repeated on additional rows:

```
> diag(c(3, 10, 11), nrow=5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    0    0    0    0
[2,]    0   10    0    0    0
[3,]    0    0   11    0    0
[4,]    0    0    0    3    0
[5,]    0    0    0    0   10
```

Extracting the diagonal elements of a matrix:

```
> m <- matrix(1:6, nrow=2)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> diag(m)
[1] 1 4
```

Transpose of a matrix:

```
> matrix(1:6, nrow=2)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> t(matrix(1:6, nrow=2))
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Addition and subtraction

```
> A <- matrix(c(1:12), nrow=3)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> A + A
     [,1] [,2] [,3] [,4]
[1,]    2    8   14   20
[2,]    4   10   16   22
[3,]    6   12   18   24
> A - A
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
```

Element wise multiplication and division:

```
> A * A
     [,1] [,2] [,3] [,4]
[1,]    1   16   49  100
[2,]    4   25   64  121
[3,]    9   36   81  144
> A / A
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    1    1    1
[3,]    1    1    1    1
```

Element wise power operation:

```
> A^3
     [,1] [,2] [,3] [,4]
[1,]    1   64  343 1000
[2,]    8  125  512 1331
[3,]   27  216  729 1728
> A^(0.5)
         [,1]     [,2]     [,3]     [,4]
[1,] 1.000000 2.000000 2.645751 3.162278
[2,] 1.414214 2.236068 2.828427 3.316625
[3,] 1.732051 2.449490 3.000000 3.464102
```

Row wise addition, subtraction, multiplication, division:

```
> v <- c(2,1,4)
> A + v
     [,1] [,2] [,3] [,4]
[1,]    3    6    9   12
[2,]    3    6    9   12
[3,]    7   10   13   16
> A - v
     [,1] [,2] [,3] [,4]
[1,]   -1    2    5    8
[2,]    1    4    7   10
[3,]   -1    2    5    8
> A * v
```

```
     [,1] [,2] [,3] [,4]
[1,]    2    8   14   20
[2,]    2    5    8   11
[3,]   12   24   36   48
> A / v
     [,1] [,2] [,3] [,4]
[1,] 0.50  2.0 3.50    5
[2,] 2.00  5.0 8.00   11
[3,] 0.75  1.5 2.25    3
```

Column wise addition, subtraction, multiplication, division:

```
> v <- c(2, 3, 1, 4)
> t(t(A) + v)
     [,1] [,2] [,3] [,4]
[1,]    3    7    8   14
[2,]    4    8    9   15
[3,]    5    9   10   16
> t(t(A) - v)
     [,1] [,2] [,3] [,4]
[1,]   -1    1    6    6
[2,]    0    2    7    7
[3,]    1    3    8    8
> t(t(A) * v)
     [,1] [,2] [,3] [,4]
[1,]    2   12    7   40
[2,]    4   15    8   44
[3,]    6   18    9   48
> t(t(A) / v)
     [,1]     [,2] [,3] [,4]
[1,]  0.5 1.333333    7 2.50
[2,]  1.0 1.666667    8 2.75
[3,]  1.5 2.000000    9 3.00
```

Another way:

```
> A + rep(v, each=3)
     [,1] [,2] [,3] [,4]
[1,]    3    7    8   14
[2,]    4    8    9   15
[3,]    5    9   10   16
> A - rep(v, each=3)
     [,1] [,2] [,3] [,4]
[1,]   -1    1    6    6
[2,]    0    2    7    7
[3,]    1    3    8    8
> A * rep(v, each=3)
     [,1] [,2] [,3] [,4]
[1,]    2   12    7   40
[2,]    4   15    8   44
[3,]    6   18    9   48
> A / rep(v, each=3)
     [,1]     [,2] [,3] [,4]
[1,]  0.5 1.333333    7 2.50
[2,]  1.0 1.666667    8 2.75
[3,]  1.5 2.000000    9 3.00
```

Matrix multiplication:

```
> m <- matrix(1:4, nrow=2)
> m %*% m
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```

Quadratic form:

```
> v = c(1:2)
> v %*% m %*% v
     [,1]
[1,]   27
```

Note that the vector v is being treated as both row vector and column vector.

Cross product of two matrices:

```
> A <- matrix(c(1,1,1,3,0,2), nrow=3)
> B <- matrix(c(0,7,2,0,5,1), nrow=3)
> A
     [,1] [,2]
[1,]    1    3
[2,]    1    0
[3,]    1    2
> B
     [,1] [,2]
[1,]    0    0
[2,]    7    5
[3,]    2    1
> t(A) %*% B
     [,1] [,2]
[1,]    9    6
[2,]    4    2
> crossprod(A, B)
     [,1] [,2]
[1,]    9    6
[2,]    4    2
> A %*% t(B)
     [,1] [,2] [,3]
[1,]    0   22    5
[2,]    0    7    2
[3,]    0   17    4
> tcrossprod(A, B)
     [,1] [,2] [,3]
[1,]    0   22    5
[2,]    0    7    2
[3,]    0   17    4
```

Computing the Gram matrix for a given matrix $A^T A$:

```
> A <- matrix(c(1,1,1,3,0,2), nrow=3)
> t(A) %*% A
     [,1] [,2]
[1,]    3    5
[2,]    5   13
> crossprod(A)
     [,1] [,2]
[1,]    3    5
```

```
[2,]    5   13
```

Computing the frame AA^T:

```
> A <- matrix(c(1,1,1,3,0,2), nrow=3)
> A %*% t(A)
     [,1] [,2] [,3]
[1,]   10    1    7
[2,]    1    1    1
[3,]    7    1    5
```

Outer product of two matrices:

```
> m1 <- matrix(1:4, nrow=2)
> m2 <- matrix(c(1,3,5,7), nrow=2)
> outer(m1, m2)
, , 1, 1

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

, , 2, 1

     [,1] [,2]
[1,]    3    9
[2,]    6   12

, , 1, 2

     [,1] [,2]
[1,]    5   15
[2,]   10   20

, , 2, 2

     [,1] [,2]
[1,]    7   21
[2,]   14   28
```

Assigning names to rows and columns:

```
> m <- matrix(c(1:4), nrow=2)
> colnames(m) <- c("x", "y")
> rownames(m) <- c("a", "b")
> m
  x y
a 1 3
b 2 4
```

2.3 Arrays

Creating an array:

```
> a <- array(1:10, dim=c(4,4,4))
> a
, , 1

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

, , 2

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

, , 3

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

, , 4

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

Checking its dimensions:

```
> dim(a)
[1] 4 4 4
```

Accessing its elements:

```
> a[1,1,1]
[1] 1
> a[1,2, 1:4]
[1] 5 1 7 3
>
```

Creating an array from a vector:

```
> x <- 1:18
> dim(x) <- c(2,3,3)
> x
, , 1

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

```
, , 2

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

, , 3

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

Recycling of vector elements while constructing of an array:

```
> a <- array(1:4, dim=c(2,3,3))
> a
, , 1

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

, , 2

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

, , 3

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

Generalized transpose of an array:

```
> a <- array(1:4, dim=c(2,3,4))
> b <- aperm(a, perm=c(3,2, 1))
> dim(b)
[1] 4 3 2
```

The usual transpose of a matrix is a special case

2.3.1 Index Matrices

Using an index matrix to pick out elements from an array:

```
> data <- array(1:20, dim=c(5,4))
> data
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
> indices <- cbind(c(1,2,3), c(1,3,2))
> indices
```

```
     [,1] [,2]
[1,]    1    1
[2,]    2    3
[3,]    3    2
> data[indices]
[1]  1 12  8
```

Each row in the index matrix identifies one element in the data array to be picked. The number of columns in the index matrix must be same as the dimension of the data array.

Updating array elements using the index matrix:

```
> data[indices] <- 0
> data
     [,1] [,2] [,3] [,4]
[1,]    0    6   11   16
[2,]    2    7    0   17
[3,]    3    0   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

Indices with NA and 0:

```
> indices <- cbind(c(1,2,3, NA, 2), c(2,3,4, 2, 0))
> data[indices]
[1]  6  0 18 NA
```

Rows containing NA return NA. Rows containing 0 are ignored.

Extracting the elements of the anti-diagonal from a matrix:

```
> m <- matrix(1:9, nrow=3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> indices = cbind(1:3, rev(1:3))
> indices
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
> m[indices]
[1] 7 5 3
```

A matrix with 0 everywhere and 1 in the anti-diagonal:

```
> m <- matrix(0, 3,3)
> m[indices] = 1
> m
     [,1] [,2] [,3]
[1,]    0    0    1
[2,]    0    1    0
[3,]    1    0    0
```

This is also known as anti-diagonal matrix.

2.3.2 The recycling rule

- The expression is scanned from left to right.
- Any short vector operands are extended by recycling their values until they match the size of any other operands.
- As long as short vectors and arrays only are encountered, the arrays must all have the same dim attribute or an error results.
- Any vector operand longer than a matrix or array operand generates an error.
- If array structures are present and no error or coercion to vector has been precipitated, the result is an array structure with the common dim attribute of its array operands.

2.4 Lists

Creating a list:

```
> l = list(a=c(1,2,3), b=c(1:10), c=3)
> l
$a
[1] 1 2 3

$b
 [1]  1  2  3  4  5  6  7  8  9 10

$c
[1] 3

> l$a
[1] 1 2 3
> l$b
 [1]  1  2  3  4  5  6  7  8  9 10
> l$c
[1] 3
```

Names in the list:

```
> names(l)
[1] "a" "b" "c"
```

Accessing list elements:

```
> l[[1]]
[1] 1 2 3
> l[[2]]
 [1]  1  2  3  4  5  6  7  8  9 10
> l[[3]]
[1] 3
> l$a
[1] 1 2 3
> l$c
[1] 3
> l$c + 2
[1] 5
> l$b + 3
 [1]  4  5  6  7  8  9 10 11 12 13
```

```
> l$a * l$a
[1] 1 4 9
> l[['a']]
[1] 1 2 3
> l[['b']]
 [1]  1  2  3  4  5  6  7  8  9 10
> l[['c']]
[1] 3
```

`[]` returns a sublist while `[[]]` returns a list element:

```
> l[1]
$a
[1] 1 2 3

> l[c(1,2)]
$a
[1] 1 2 3

$b
 [1]  1  2  3  4  5  6  7  8  9 10
```

Iterating over list elements:

```
> for (name in names(l)){print(l[[name]])}
[1] 1 2 3
 [1]  1  2  3  4  5  6  7  8  9 10
[1] 3
```

Appending elements in list:

```
> for (name in names(l)){print(c(name,":", l[[name]]), quote=FALSE)}
[1] a : 1 2 3
 [1] b  :  1  2  3  4  5  6  7  8  9  10
[1] c : 3
[1] d : 4
[1] e : 5
```

Removing the last element:

```
> l[length(l)] <- NULL
> length(l)
[1] 4
> for (name in names(l)){print(c(name,":", l[[name]]), quote=FALSE)}
[1] a : 1 2 3
 [1] b  :  1  2  3  4  5  6  7  8  9  10
[1] c : 3
[1] d : 4
```

Removing an intermediate element from list:

```
> l[['c']] <- NULL
> names(l)
[1] "a" "b" "d"
> for (name in names(l)){print(c(name,":", l[[name]]), quote=FALSE)}
[1] a : 1 2 3
 [1] b  :  1  2  3  4  5  6  7  8  9  10
[1] d : 4
```

```
> length(l)
[1] 3
```

Creating lists without names:

```
> l2 <- list(1,2,"hello")
> l2
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] "hello"

> names(l2) <- c("x", "y", "z")
> l2
$x
[1] 1

$y
[1] 2

$z
[1] "hello"
```

Concatenating two lists:

```
> c(l, l2)
$a
[1] 1 2 3

$b
 [1]  1  2  3  4  5  6  7  8  9 10

$c
[1] 3

$x
[1] 1

$y
[1] 2

$z
[1] "hello"
```

From list to vector:

```
> l <- list (a=1, b=2, c=4)
> unlist(l)
a b c
1 2 4
> names(unlist(l))
[1] "a" "b" "c"
```

List of Expressions

We can also prepare a list of expressions:

```
> list(a=2+3, b=4*3)
$a
[1] 5

$b
[1] 12

> alist(a=2+3, b=4*3)
$a
2 + 3

$b
4 * 3

> l <- alist(a=2+3, b=4*3)
> l$a
2 + 3
> eval(l$a)
[1] 5
> eval(l$b)
[1] 12
```

While the `list` function evaluates its arguments, `alist` doesn't. Finally, we use `eval` for evaluating the expressions stored in the list.

2.5 Factors

Factoring a vector of numeric values:

```
>  v <- c(1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4)
> vf <- factor(v)
> levels(vf)
[1] "1" "2" "3" "4"
> vf
 [1] 1 1 2 2 2 3 3 3 3 4 4
Levels: 1 2 3 4
```

Constructing ordered factors:

```
> vf <- factor(v, levels=c(1,2,3,4), ordered=TRUE)
> vf
 [1] 1 1 2 2 2 3 3 3 3 4 4
Levels: 1 < 2 < 3 < 4
```

Converting the factors back to numeric values to compute the mean:

```
> mean(as.numeric(levels(vf)[vf]))
[1] 2.545455
```

Factoring a vector of strings:

```
> colors <- sample(c("red", "green", "blue"), 10, replace = TRUE)
> colors <- factor(colors)
> colors
 [1] blue  green green blue  green blue  red   red   blue  red
Levels: blue green red
> levels(colors)
[1] "blue"  "green" "red"
```

Using factors for grouping to compute the mean:

```
> colors <- c('r', 'r', 'g', 'b', 'r', 'g', 'g', 'b', 'b', 'r')
> length(colors)
[1] 10
> lengths <-c(1, 1, 2, 2, 1, 1, 1, 2, 2, 3)
> length(lengths)
[1] 10
> colorsf <- factor(colors)
> mean(lengths)
[1] 1.6
> tapply(lengths, colorsf, mean)
       b        g        r
2.000000 1.333333 1.500000
```

Generating a sequence of factors:

```
> gl(2,8)
 [1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
Levels: 1 2
> as.integer(gl(2,8))
 [1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
```

Generating factors with labels:

```
> gl(2,8, labels=c("x", "y"))
 [1] x x x x x x x x y y y y y y y y
Levels: x y
> as.integer(gl(2,8, labels=c("x", "y")))
 [1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
```

By default the generated factors are unordered.

Generating ordered factors:

```
> gl(2,8, labels=c("c", "b"), ordered=TRUE)
 [1] c c c c c c c c b b b b b b b b
Levels: c < b
```

We can use the length argument to repeat the sequence:

```
> gl(2,1,10)
 [1] 1 2 1 2 1 2 1 2 1 2
Levels: 1 2
> gl(2,2,10)
 [1] 1 1 2 2 1 1 2 2 1 1
Levels: 1 2
> gl(2,2,12)
 [1] 1 1 2 2 1 1 2 2 1 1 2 2
Levels: 1 2
```

```
> gl(2,3,12)
 [1] 1 1 1 2 2 2 1 1 1 2 2 2
Levels: 1 2
```

2.6 Data Frames

Creating a data frame:

```
> df <- data.frame(x=c(11,12,13), y=c(21,22,23), z=c(7,20, 10))
   x  y  z
1 11 21  7
2 12 22 20
3 13 23 10
```

Accessing first row:

```
> df[1,]
   x  y z
1 11 21 7
```

Column names:

```
> colnames(df)
[1] "x" "y" "z"
```

Row names:

```
> rownames(df)
[1] "1" "2" "3"
```

By default, the row names are the auto-generated row numbers.

Accessing named columns:

```
> df$x
[1] 11 12 13
> df$y
[1] 21 22 23
```

Accessing columns by number:

```
> df[,1]
[1] 11 12 13
```

Another example:

```
> hw = data.frame(hello=c(1,2,3), world=c(4,5,6))
> hw
  hello world
1     1     4
2     2     5
3     3     6
```

Summing each column:

```
> colSums(df)
 x  y  z
36 66 37
```

Summing each row:

```
> rowSums(df)
[1] 39 54 46
```

Data frame from a list:

```
> l <- list(x=c(1,2,3), y=c(3,2,1))
> df <- as.data.frame((l))
> df
  x y
1 1 3
2 2 2
3 3 1
```

Seeing first few rows of a large data frame:

```
> df <- data.frame(x=1:40, y=(1:40)^2, z=(1:40)^3)
> head(df)
  x  y   z
1 1  1   1
2 2  4   8
3 3  9  27
4 4 16  64
5 5 25 125
6 6 36 216
> head(df, n=4)
  x  y  z
1 1  1  1
2 2  4  8
3 3  9 27
4 4 16 64
```

Seeing last few rows of a large data frame:

```
> tail(df)
    x    y     z
35 35 1225 42875
36 36 1296 46656
37 37 1369 50653
38 38 1444 54872
39 39 1521 59319
40 40 1600 64000
> tail(df, n=3)
    x    y     z
38 38 1444 54872
39 39 1521 59319
40 40 1600 64000
```

Seeing some random rows from a data frame:

```
> dplyr::sample_n(df, 4)
    x   y     z
13 13 169  2197
```

```
12 12 144  1728
30 30 900 27000
27 27 729 19683
```

It is also possible to sample with replacement:

```
> df <- data.frame(x=1:4, y=(1:4)^2, z=(1:4)^3)
> dplyr::sample_n(df, 6, replace=T)
    x  y  z
2   2  4  8
2.1 2  4  8
4   4 16 64
2.2 2  4  8
2.3 2  4  8
3   3  9 27
```

Attaching workspace to columns of data frame

Let's create a data frame:

```
> df <- data.frame(x=c(1,2,3), y=c(3,2,1))
```

At this moment, x and y are not available in the search path:

```
> x
Error: object 'x' not found
```

Let's now attach the variables in data frame on the search path:

```
> attach(df)
```

We can now access x and y directly:

```
> x
[1] 1 2 3
> y
[1] 3 2 1
```

Let's detach the variables in data frame from the search path:

```
> detach(df)
```

x and y are no more accessible directly:

```
> x
Error: object 'x' not found
```

If we update the data frame variables while they are attached in search path, this will not be reflected correctly.

Let's attach our data frame:

```
> attach(df)
```

Let's update the x variable:

```
> df$x <- x +  y
> df
  x y
1 4 3
2 4 2
3 4 1
```

This doesn't reflect in the attached variables:

```
> x
[1] 1 2 3
> y
[1] 3 2 1
```

Let's detach and re-attach the data frame:

```
> detach()
> attach(df)
```

We can now see the updated variables:

```
> x
[1] 4 4 4
> y
[1] 3 2 1
```

rbind and cbind

They work pretty much like the matrices.

Let's create a data frame:

```
> df <- data.frame(x=c(1,2,3), y=c('a', 'b', 'c'))
> df
  x y
1 1 a
2 2 b
3 3 c
```

Let's bind a new row:

```
> rbind(df, c(4, 'c'))
  x y
1 1 a
2 2 b
3 3 c
4 4 c
```

Let's bind a new column:

```
> cbind(df, z=c(T, F, F))
  x y     z
1 1 a  TRUE
2 2 b FALSE
3 3 c FALSE
```

There are some caveats though. Note that a variable with character values gets factored by default during the creation of a data frame. Thus, we can only insert a factor level which is already there in the table in that column later. See for example:

```
> rbind(df, c(4, 'd'))
  x    y
1 1    a
2 2    b
3 3    c
4 4 <NA>
Warning message:
In `[<-.factor`(`*tmp*`, ri, value = "d") :
  invalid factor level, NA generated
```

Viewing a data frame in RSutio:

```
> View(df)
```

Describing a data set:

```
> df <- data.frame(x=c(1,2,3), y=c('a', 'b', 'c'))
> str(df)
'data.frame':   3 obs. of  2 variables:
 $ x: num  1 2 3
 $ y: Factor w/ 3 levels "a","b","c": 1 2 3
> df <- data.frame(x=1:40, y=(1:40)^2, z=(1:40)^3)
> str(df)
'data.frame':   40 obs. of  3 variables:
 $ x: int  1 2 3 4 5 6 7 8 9 10 ...
 $ y: num  1 4 9 16 25 36 49 64 81 100 ...
 $ z: num  1 8 27 64 125 216 343 512 729 1000 ...
```

Identifying columns with missing data

Let us prepare a data frame with some columns having missing data:

```
> df <- data.frame(x=1:10, y=11:20, z=21:30, a=31:40, b=41:50)
> df$x[5] <- NA
> df$a[7] <- NA
> df
    x  y  z  a  b
1   1 11 21 31 41
2   2 12 22 32 42
3   3 13 23 33 43
4   4 14 24 34 44
5  NA 15 25 35 45
6   6 16 26 36 46
7   7 17 27 NA 47
8   8 18 28 38 48
9   9 19 29 39 49
10 10 20 30 40 50
```

Compute the number of non-zero entries in each column:

```
> colSums(is.na(df))
x y z a b
1 0 0 1 0
```

Identify columns with non-zero entries in above:

```
> colSums(is.na(df)) > 0
    x     y     z     a     b
 TRUE FALSE FALSE  TRUE FALSE
```

Identify corresponding column names:

```
> colnames(df)[colSums(is.na(df)) > 0]
[1] "x" "a"
```

2.7 Time Series

Creating a time series from an observation vector:

```
> observations <- sample(1:10, 24, replace=T)
> observations
 [1]  2  7  2  6  2  5  5  8  8  6  4  9  8  6  3  2  5  1  2  5  4  8  5 10
> time_series <- ts(observations, start=c(2016,1), frequency=12)
> time_series
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2016   2   7   2   6   2   5   5   8   8   6   4   9
2017   8   6   3   2   5   1   2   5   4   8   5  10
```

Some properties of time series:

```
> class(time_series)
[1] "ts"
> mode(time_series)
[1] "numeric"
> typeof(time_series)
[1] "integer"
```

Extracting a window from the time series:

```
> window(time_series, start=c(2016, 7), end=c(2016, 12))
     Jul Aug Sep Oct Nov Dec
2016   5   8   8   6   4   9
```

CHAPTER 3

Language Facilities

3.1 Operator precedence rules

- Function calls and grouping expressions: (), { }
- **Index and lookup operators:**
  - Indexing: [], [[]]
  - Namespace access: ::, ::
  - Component, slot extraction: $, @
- **Arithmetic:**
  - Exponentiation: ^ (right to left)
  - Unary plus, minus: +, -
  - Sequence operator: :
  - Special operators: %any%, %%, %/%, %*%, %o%
  - Multiply, Divide: *, /
  - add, subtract : +, -
- Comparison: <, > , <=, >=, ==, !=
- Negation : !
- And: &, &&
- Or: | , ||
- Formulas: ~
- Right wise Assignment: ->, ->>
- Assignment =
- Assignment (right to left) <-, <<-

- Help: ?

3.2 Expressions

One expression per line:

```
> x <-1
> y <- x+2
> z <- y + x
> x
[1] 1
> y
[1] 3
> z
[1] 4
```

Multiple expressions in single line:

```
> x <- 1; y <- x+2; z <- y + x
> x; y; z
[1] 1
[1] 3
[1] 4
```

Series of expressions followed by a value:

```
> {x <- 1; y <- x+2; z <- y + x; z}
[1] 4
```

3.3 Flow Control

Help about control flow:

```
?Control
```

if, else, ifelse

if

```
> if (T) c(1,2)
[1] 1 2
> if (F) c(1,2)
```

if else:

```
> if (T)  c(1,2,3) else matrix(c(1,2,3, 4), nrow=2)
[1] 1 2 3
> if (F)  c(1,2,3) else matrix(c(1,2,3, 4), nrow=2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

vectorized ifelse:

```
> v1 <- c(1,2,3,4)
> v2 <- c(5,6,7,8)
> cond <- c(T,F,F,T)
> ifelse(cond, v1, v2)
[1] 1 6 7 4
```

Logical operations:

```
> T && F
[1] FALSE
> T || F
[1] TRUE
```

Element wise logical operations:

```
> v1 <- c(T,T,F,F)
> v2 <- c(T, F, T, F)
> v1 | v2
[1]  TRUE  TRUE  TRUE FALSE
> v1 & v2
[1]  TRUE FALSE FALSE FALSE
```

repeat

A simple repeat loop

```
> x <- 10
> repeat { if (x == 0) break ; x = x - 1; print(x)}
[1] 9
[1] 8
[1] 7
[1] 6
[1] 5
[1] 4
[1] 3
[1] 2
[1] 1
[1] 0
```

If your repeat loop is stuck in an infinite loop, press ESC key.

for

Simple for loops:

```
> for (i in seq(1,10)) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
```

```
[1] 8
[1] 9
[1] 10
> for (i in seq(1,10, by=2)) print(i)
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

Results are not printed inside a loop without using the `print` function as above:

```
> for (i in seq(1,10)) i
>
```

For loop for computing sum of squares:

```
ul <- rnorm(30)
usq <- 0
for (i in 1:10){
        usq <- ul[i] * ul[i]
}
```

Off course a better solution is `sum(ul^2)`.

Nested for loops:

```
nrow <- 10
ncol <- 10
m <- matrix(nrow=nrow, ncol=ncol)

for (i in 1:nrow){
        for (j in 1:ncol){
                m[i, j] <- i + j
        }
}
```

while

A simple while loop:

```
> i <- 10; while ( i < 20 ) {i <- i +1; print(i)}
[1] 11
[1] 12
[1] 13
[1] 14
[1] 15
[1] 16
[1] 17
[1] 18
[1] 19
[1] 20
```

While loop with next and break

```
> i <- 10; while (T) {i <- i +1; if (i == 20) break; if ( i %% 2 == 0) next; print(i);
↪}
[1] 11
[1] 13
[1] 15
[1] 17
[1] 19
```

..rubric:: iterators

Installing the package:

```
> install.packages('iterators')
```

Loading the package:

```
> library(iterators)
```

Creating an iterator:

```
> ii <- iter(1:4)
```

Using the iterator:

```
> nextElem(ii)
[1] 1
> nextElem(ii)
[1] 2
> nextElem(ii)
[1] 3
> nextElem(ii)
[1] 4
> nextElem(ii)
Error: StopIteration
```

An iterator recycling the elements:

```
> ii <- iter(1:4, recycle = T)
> for (i in 1:10) print(nextElem(ii))
[1] 1
[1] 2
[1] 3
[1] 4
[1] 1
[1] 2
[1] 3
[1] 4
[1] 1
[1] 2
```

foreach

Installing the package:

```
> install.packages('foreach')
```

Loading the library:

```
> library(foreach)
```

Checking the variation on growth of income with compounded interest rate:

```
> unlist(foreach(i=1:10) %do% {100 * (1 + i/100)^5})
 [1] 105.1010 110.4081 115.9274 121.6653 127.6282 133.8226 140.2552 146.9328 153.8624␣
↪161.0510
```

It works with iterators too:

```
> unlist(foreach(i=iter(1:10)) %do% {100 * (1 + i/100)^5})
 [1] 105.1010 110.4081 115.9274 121.6653 127.6282 133.8226 140.2552 146.9328 153.8624␣
↪161.0510
```

3.4 Functions

Calling an function:

```
> b = c(2,3,5)
> m = mean(x=b)
> s = sum(c(4,5,8,11))
```

Computing variance by combining multiple functions:

```
> x <- c(rnorm(10000))
> sum((x-mean(x))^2)/(length(x)-1)
[1] 0.992163
```

Defining a function:

```
function_name <- function (arglist){
        body
}
```

Defining our own mean function:

```
my_mean <- function(x){
   s <- sum(x)
   n <- length(x)
   s / n
}
```

Using the function:

```
> my_mean(rivers)
[1] 591.1844
```

Verifying against built-in implementation of mean:

```
> mean(rivers)
[1] 591.1844
```

A log-sum-exp function:

```
log_sum_exp <- function(x){
  xx <- exp(x)
  xxx <- sum(xx)
  log(xxx)
}
```

Let us store its definition into a file named `my_functions.R`.

Loading the function definition:

```
> source('my_functions.R')
```

Calling the function:

```
> log_sum_exp(10)
[1] 10
> log_sum_exp(c(10, 12))
[1] 12.12693
> log_sum_exp(sample(1:100, 100, replace=T))
[1] 100.4429
```

Recursive Functions

Let us solve the Tower of Hanoi problem in R:

```
hanoi <- function(num_disks, from, to, via, disk_num=num_disks){
        if (num_disks == 1){
                cat("move disk", disk_num,  "from ", from, "to", to, "\n")
        }else{
        hanoi(num_disks-1, from, via, to)
                hanoi(1, from, to, via, disk_num)
                hanoi(num_disks-1, via, to, from)
        }
}
```

Let's see this in action:

```
> hanoi(1,'a', 'b', 'c')
move disk 1 from  a to b
> hanoi(2,'a', 'b', 'c')
move disk 1 from  a to c
move disk 2 from  a to b
move disk 1 from  c to b
> hanoi(3,'a', 'b', 'c')
move disk 1 from  a to b
move disk 2 from  a to c
move disk 1 from  b to c
move disk 3 from  a to b
move disk 1 from  c to a
move disk 2 from  c to b
move disk 1 from  a to b
```

3.4.1 Closure in Lexical Scope

Accessing variable in the lexical scope:

```
fourth_power <- function(n){
  sq <- function() n* n
  sq() * sq()
}
```

Let's see this function in action:

```
> fourth_power(2)
[1] 16
> fourth_power(3)
[1] 81
```

Let's create a counter generator function:

```
counter <- function(n){
  list(
    increase = function(){
      n <<- n+1
    },
    decrease = function(){
      n <<- n-1
    },
    value = function(){
      n
    }
  )
}
```

The value `n` is the initial value of the counter. This gets stored in the closure for the function. The function returns a list whose members are functions which manipulate the value of `n` sitting in the closure.

The operator `<<-` is used to update a variable in lexical scope.

Let's now construct a counter object:

```
> v <- counter(10)
> v$value()
[1] 10
```

Let's increase and decrease counter values:

```
> v$increase()
> v$increase()
> v$value()
[1] 12
> v$decrease()
> v$decrease()
> v$value()
[1] 10
```

3.5 Packages

A library is a collection of packages. Libraries are local to an R installation. Typically, there is a global library with the R installation and a user specific library.

List of library paths:

```
> .libPaths()
[1] "C:/Users/Shailesh/R/win-library/3.4" "C:/Program Files/R/R-3.4.2/library"
```

List of installed packages in all libraries:

```
> library()
```

Installing a package:

```
> install.packages("geometry")
```

Loading a package:

```
> library("geometry")
```

Installing a package if it is not installed:

```
> if(!require(psych)){install.packages("psych")}
```

List of currently installed packages:

```
> search()
 [1] ".GlobalEnv"        "package:foreach"   "package:iterators" "package:MASS"
 [5] "package:ggplot2"   "package:e1071"     "tools:rstudio"     "package:stats"
 [9] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"
[13] "package:methods"   "Autoloads"         "package:base"
```

This may vary in your setup.

List of loaded namespaces:

```
> loadedNamespaces()
 [1] "Rcpp"       "codetools"  "grDevices"  "class"      "foreach"    "MASS"
 [7] "grid"       "plyr"       "gtable"     "e1071"      "datasets"   "scales"
[13] "ggplot2"    "rlang"      "utils"      "lazyeval"   "graphics"   "base"
[19] "labeling"   "iterators"  "tools"      "munsell"    "compiler"   "stats"
[25] "colorspace" "methods"    "tibble"
```

3.6 R Scripts

Extension is ".R".

Running a script:

```
> source("foo.R")
```

3.7 Logical Tests

Checking for missing values:

```
> x <- c(1, 4, NA, 5, 0/0)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE  TRUE
```

Checking for not a number values:

```
> is.nan(x)
[1] FALSE FALSE FALSE FALSE  TRUE
```

Checking for vectors:

```
> is.vector(1:3)
[1] TRUE
> is.vector("133")
[1] TRUE
> is.vector(matrix(1:4, nrow=2))
[1] FALSE
```

Checking for matrices:

```
> is.matrix(1:3)
[1] FALSE
> is.matrix(matrix(1:4, nrow=2))
[1] TRUE
```

3.8 Introspection

The mode of an object is the basic type of its fundamental constituents:

```
> x <- 1:10
> mode(x)
[1] "numeric"
```

Class of an object:: > class(x) [1] “integer”

Type of an object:: > typeof(x) [1] “integer”

Length of an object:: > length(x) [1] 10

Mode of a list:

```
> l <- list(1, '2', 3.4, TRUE)
> mode(l)
[1] "list"
```

Mode of a sublist is also list:

```
> mode(l[1])
[1] "list"
```

But individual elements in the list have different modes:

```
> mode(l[[1]])
[1] "numeric"
> mode(l[[2]])
[1] "character"
```

List of attributes

```
> l <- list("1", 2, TRUE, NA)
> attributes(l)
NULL
```

Setting an attribute:

```
> attr(l, 'color') <- 'red'
> attributes(l)
$color
[1] "red"

> attr(l, 'color')
[1] "red"
```

The class of an object enables object oriented programming and allows same function to behave differently for different classes.

Querying the class of an object:

```
> class(1:10)
[1] "integer"
> class(matrix(1:10, nrow=2))
[1] "matrix"
> class(list(1,2,3))
[1] "list"
```

Removing the class of an object (temporarily):

```
> unclass(object)
```

3.9 Coercion

Integers to strings:

```
> as.character(10:14)
[1] "10" "11" "12" "13" "14"
```

Strings to integers:

```
> as.integer(c("10", "11", "12", "13"))
[1] 10 11 12 13
```

Convert an array to a vector:

```
> as.vector(arr)
```

3.10 Sorting and Searching

Searching in a vector:

```
> which (v == 5)
[1] 5
> which (v > 5)
```

```
[1]  6  7  8  9 10
> which (v > 5 & v < 8)
[1] 6 7
```

Searching in a matrix:

```
> m <- matrix(1:10, nrow=2)
> m == 4
      [,1]  [,2]  [,3]  [,4]  [,5]
[1,] FALSE FALSE FALSE FALSE FALSE
[2,] FALSE  TRUE FALSE FALSE FALSE
> which(m == 4)
[1] 4
```

Sorting a vector in ascending order:

```
> x = sample(1:10)
> x
 [1]  6  5  8 10  2  4  1  3  7  9
> sort(x)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Finding unique elements:

```
> v <- c(1, 4, 4, 3, 4, 4, 3, 3, 1, 2, 3, 4, 2, 3, 1, 3, 5, 6)
> unique(v)
[1] 1 4 3 2 5 6
```

3.11 Basic Mathematical Functions

Trigonometric functions:

```
> theta = pi/2
> sin(theta)
[1] 1
> cos(theta)
[1] 6.123032e-17
> tan(theta)
[1] 1.633124e+16
> asin(1)
[1] 1.570796
> acos(1)
[1] 0
> atan(1)
[1] 0.7853982
> atan(1) * 2
[1] 1.570796
```

Exponentiation:

```
> exp(1)
[1] 2.718282
```

Logarithms:

```
> log(exp(1))
[1] 1
> log(exp(4))
[1] 4
> log10(10^4)
[1] 4
> log2(8)
[1] 3
> log2(c(8,16,256,1024, 2048))
[1]  3  4  8 10 11
```

Square root:

```
> sqrt(4)
[1] 2
> sqrt(-4)
[1] NaN
Warning message:
In sqrt(-4) : NaNs produced
> sqrt(-4+0i)
[1] 0+2i
```

3.12 Built-in Constants

π:

```
> pi
```

[1] 3.141593 >

Month names:

```
> month.name
 [1] "January"   "February"  "March"     "April"     "May"       "June"      "July"
↪    "August"
 [9] "September" "October"   "November"  "December"
```

Month name abbreviations:

```
> month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

English letters:

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
↪"u" "v" "w" "x" "y" "z"
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T"
↪"U" "V" "W" "X" "Y" "Z"
```

3.13 Converting Numerical Data to Factor

Numerical data may need to be binned into a sequence of intervals.

Breaking data into intervals of equal length:

```
> data <- sample(0:20, 10, replace = TRUE)
> data
 [1] 10  0 20  3 13 13 16  2  1 10
> cut (data, breaks=4)
 [1] (5,10]     (-0.02,5] (15,20]    (-0.02,5] (10,15]    (10,15]    (15,20]    (-0.02,5]␣
↪(-0.02,5] (5,10]
Levels: (-0.02,5] (5,10] (10,15] (15,20]
```

Each interval is by default open on left side and closed on right side. Closed on left and open on right intervals can be created by using the parameter right=FALSE.

Frequency of categories:

```
> table(cut (data, breaks=4))

(-0.02,5]     (5,10]    (10,15]    (15,20]
        4          2          2          2
```

Making sure that the factors are ordered:

```
> cut (data, breaks=4, ordered_result = TRUE)
 [1] (5,10]     (-0.02,5] (15,20]    (-0.02,5] (10,15]    (10,15]    (15,20]    (-0.02,5]␣
↪(-0.02,5] (5,10]
Levels: (-0.02,5] < (5,10] < (10,15] < (15,20]
```

Using our own labels for the factors:

```
> cut (data, breaks=4, labels=c("a", "b", "c", "d"))
 [1] b a d a c c d a a b
Levels: a b c d
```

Specifying our own break-points (intervals) for cutting:

```
> cut (data, breaks=c(-1, 5,10, 20))
 [1] (5,10]  (-1,5]  (10,20] (-1,5]  (10,20] (10,20] (10,20] (-1,5]  (-1,5]  (5,10]
Levels: (-1,5] (5,10] (10,20]
```

Including the lowest value in the first interval:

```
> cut (data, breaks=c(0, 5,10, 20), include.lowest = TRUE)
 [1] (5,10]  [0,5]   (10,20] [0,5]   (10,20] (10,20] (10,20] [0,5]   [0,5]   (5,10]
Levels: [0,5] (5,10] (10,20]
```

3.14 Apply Family of Functions

Sample data:

```
> m <- matrix(1:8, nrow=2)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

Summing a matrix over rows:

```
> apply(m, 1, sum)
[1] 16 20
```

Summing a matrix over columns:

```
> apply(m, 2, sum)
[1]  3  7 11 15
```

Median for each row and column:

```
> apply(m, 1, median)
[1] 4 5
> apply(m, 2, median)
[1] 1.5 3.5 5.5 7.5
```

`lapply` applies a function on each element of a list and returns the values as a list.

Let's prepare a list of matrices:

```
> A <- matrix(c(1,1,1,3,0,2), nrow=3)
> B <- matrix(c(0,7,2,0,5,1), nrow=3)
> l <- list(A, B)
> l
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    1    0
[3,]    1    2

[[2]]
     [,1] [,2]
[1,]    0    0
[2,]    7    5
[3,]    2    1
```

Extracting first row from each matrix:

```
> lapply(l, '[', 1,)
[[1]]
[1] 1 3

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

Extracting second column from each matrix:

```
> lapply(l, '[', , 2)
[[1]]
[1] 3 0 2

[[2]]
[1] 0 5 1
```

Extracting the element at position [1,2] from each matrix:

```
> lapply(l, '[', 1,2)
[[1]]
[1] 3
```

```
[[2]]
[1] 0
> unlist(lapply(l, '[', 1,2))
[1] 3 0
```

Computing the mean of each column in the mtcars dataset:

```
> lapply(mtcars, 'mean')
$mpg
[1] 20.09062

$cyl
[1] 6.1875

$disp
[1] 230.7219

$hp
[1] 146.6875

$drat
[1] 3.596563

$wt
[1] 3.21725

$qsec
[1] 17.84875

$vs
[1] 0.4375

$am
[1] 0.40625

$gear
[1] 3.6875

$carb
[1] 2.8125
```

`sapply` can help achieve the combination of unlist and lapply easily:

```
> sapply(l, '[', 1,2)
[1] 3 0
```

It basically attempts to simplify the result of `lapply` as much as possible.

Computing the mean of each column in mtcars:

```
> sapply(mtcars, 'mean')
       mpg        cyl       disp         hp       drat         wt       qsec
↪vs         am
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750   0.
↪437500   0.406250
      gear       carb
  3.687500   2.812500
```

The same for iris dataset:

```
> sapply(iris, 'mean')
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
    5.843333     3.057333     3.758000     1.199333           NA
Warning message:
In mean.default(X[[i]], ...) :
  argument is not numeric or logical: returning NA
```

Printing class of each column in a data frame:

```
> sapply(iris, class)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
   "numeric"    "numeric"    "numeric"    "numeric"     "factor"
```

`mapply` applies a function repetitively to elements from a pair of lists or vectors:

```
> v1 <- c(1,2,3)
> v2 <- c(3,4,5)
> mapply(v1, v2, sum)
[1] 4 6 8
```

Applying `rep` to each element of a vector and constructing a matrix of repeated rows:

```
> mapply(rep,1:4,4)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    2    3    4
[3,]    1    2    3    4
[4,]    1    2    3    4
```

This is equivalent to:

```
> matrix(c(rep(1, 4), rep(2, 4), rep(3, 4), rep(4, 4)),4,4)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    2    3    4
[3,]    1    2    3    4
[4,]    1    2    3    4
```

Repeating a list of characters into a matrix:

```
> l <- list("a", "b", "c", "d")
> mode(l)
[1] "list"
> class(l)
[1] "list"
> mode(l[[1]])
[1] "character"
> class(l[[1]])
[1] "character"
> m <- mapply(rep, l, 4)
> m
     [,1] [,2] [,3] [,4]
[1,] "a"  "b"  "c"  "d"
[2,] "a"  "b"  "c"  "d"
[3,] "a"  "b"  "c"  "d"
[4,] "a"  "b"  "c"  "d"
```

```
> mode(m)
[1] "character"
> class(m)
[1] "matrix"
```

One more example:

```
> l <- list("aa", "bb", "cc", "dd")
> m <- mapply(rep, l, 4)
> m
     [,1] [,2] [,3] [,4]
[1,] "aa" "bb" "cc" "dd"
[2,] "aa" "bb" "cc" "dd"
[3,] "aa" "bb" "cc" "dd"
[4,] "aa" "bb" "cc" "dd"
```

Coercion is applied when necessary:

```
> l <- list(1, "bb", T, 4.5)
> m <- mapply(rep, l, 4)
> m
     [,1] [,2] [,3]   [,4]
[1,] "1"  "bb" "TRUE" "4.5"
[2,] "1"  "bb" "TRUE" "4.5"
[3,] "1"  "bb" "TRUE" "4.5"
[4,] "1"  "bb" "TRUE" "4.5"
```

3.15 Missing Data

R has extensive support for missing data.

A vector with missing values:

```
> x <- c(1, -1, 1, NA, -2, 1, -3, 4, NA, NA, 3, 2, -4, -3, NA)
```

Identifying entries in x which are missing:

```
> is.na(x)
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE␣
↪FALSE  TRUE
```

Extracting non-missing values from x:

```
> x[!is.na(x)]
 [1]  1 -1  1 -2  1 -3  4  3  2 -4 -3
```

By defaulting summing NA values gives us NA:

```
> sum(x)
[1] NA
```

We can ignore missing values while calculating the sum:

```
> sum(x, na.rm = T)
[1] -1
```

Ignoring missing values for calculating mean:

```
> mean(x)
[1] NA
> mean(x, na.rm = T)
[1] -0.09090909
```

Ignoring missing values for calculating variance:

```
> var(x)
[1] NA
> var(x, na.rm = T)
[1] 7.090909
```

Recording a missing value:

```
> x[1] <- NA
```

Creating a new dataset without the missing data:

```
> y<-na.omit(x)
> y
 [1] -1  1 -2  1 -3  4  3  2 -4 -3
attr(,"na.action")
[1]  1  4  9 10 15
attr(,"class")
[1] "omit"
```

Failing and error out in presence of missing values:

```
> na.fail(x)
Error in na.fail.default(x) : missing values in object
> na.fail(y)
 [1] -1  1 -2  1 -3  4  3  2 -4 -3
attr(,"na.action")
[1]  1  4  9 10 15
attr(,"class")
[1] "omit"
```

3.16 Classes

A generic function performs a task or action on its arguments specific to the class of the argument itself. If the argument doesn't have a class attribute, then the default version of the generic function is called.

Various versions of the generic function `plot`:

```
> methods(plot)
 [1] plot.acf*           plot.bclust*        plot.data.frame*    plot.decomposed.ts*␣
↪plot.default
 [6] plot.dendrogram*    plot.density*       plot.ecdf           plot.factor*        ␣
↪plot.formula*
[11] plot.function       plot.hclust*        plot.histogram*     plot.HoltWinters*   ␣
↪plot.ica*
[16] plot.isoreg*        plot.lm*            plot.medpolish*     plot.mlm*           ␣
↪plot.ppr*
[21] plot.prcomp*        plot.princomp*      plot.profile.nls*   plot.raster*        ␣
↪plot.SOM*
```

```
[26] plot.somgrid*       plot.spec*          plot.stepfun        plot.stft*          
↪plot.stl*
[31] plot.svm*           plot.table*         plot.ts             plot.tskernel*      
↪plot.TukeyHSD*
[36] plot.tune*
```

Generic methods associated with `matrix` class:

```
> methods(class="matrix")
 [1] anyDuplicated as.data.frame as.raster     boxplot       coerce          
↪determinant   duplicated
 [8] edit          head          initialize    isSymmetric   Math            Math2
↪   Ops
[15] relist        subset        summary       tail          unique
```

Generic methods associated with `table` class:

```
> methods(class="table")
 [1] [             aperm         as.data.frame Axis          coerce        head
↪   initialize
 [8] lines         plot          points        print         show          
↪slotsFromS3   summary
[15] tail
```

Some of the functions may not be visible. They are marked with *:

```
> methods(coef)
[1] coef.aov*     coef.Arima*   coef.default* coef.listof*  coef.maov*    coef.nls*
```

3.17 Defining our own operators

Let us define a distance operator:

```
> `%d%` <- function(x, y) { sqrt(sum((x-y)^2)) }
```

Let us use the operator for calculating distances between points:

```
> c(1,0, 0) %d% c(0,1,0)
[1] 1.414214
> c(1,1, 0) %d% c(0,1,0)
[1] 1
> c(1,1, 1) %d% c(0,1,0)
[1] 1.414214
```

CHAPTER 4

File System and Simple IO

Working directory:

```
> getwd()
[1] "C:/Users/Shailesh"
```

Changing working directory:

```
> setwd('..')
> getwd()
[1] "C:/Users"
> setwd('shailesh')
> getwd()
[1] "C:/Users/shailesh"
```

4.1 Printing

printing a vector:

```
> x <-10
> print(x)
[1] 10

> print(1:3)
[1] 1 2 3
```

Printing data in a loop:

```
> for (month in month.abb[1:4]){print(month)}
[1] "Jan"
[1] "Feb"
[1] "Mar"
[1] "Apr"
```

Printing factors:

```
> fruits = c("apple", "banana", "pear", "watermelon")
> fruit_list <- sample(fruits, 20, replace=TRUE)
> fruit_factors <- factor(fruit_list, levels=fruits)
> fruit_factors
 [1] banana     apple      apple      pear       apple      apple      apple      ␣
↪banana     apple
[10] watermelon banana     watermelon apple      banana     apple      pear       ␣
↪banana     banana
[19] banana     pear
Levels: apple banana pear watermelon
> print(fruit_factors)
 [1] banana     apple      apple      pear       apple      apple      apple      ␣
↪banana     apple
[10] watermelon banana     watermelon apple      banana     apple      pear       ␣
↪banana     banana
[19] banana     pear
Levels: apple banana pear watermelon
```

Factors can be printed with quotes:

```
> print(fruit_factors, quote=TRUE)
 [1] "banana"     "apple"      "apple"      "pear"       "apple"      "apple"
↪"apple"      "banana"
 [9] "apple"      "watermelon" "banana"     "watermelon" "apple"      "banana"
↪"apple"      "pear"
[17] "banana"     "banana"     "banana"     "pear"
Levels: "apple" "banana" "pear" "watermelon"
```

Skipping the levels while printing:

```
> print(fruit_factors, quote=TRUE, max.levels = 0)
 [1] "banana"     "apple"      "apple"      "pear"       "apple"      "apple"
↪"apple"      "banana"
 [9] "apple"      "watermelon" "banana"     "watermelon" "apple"      "banana"
↪"apple"      "pear"
[17] "banana"     "banana"     "banana"     "pear"
```

Controlling the number of digits in floating point numbers:

```
> print(pi)
[1] 3.141593
> print(pi, digits=15)
[1] 3.14159265358979
> print(pi, digits=4)
[1] 3.142
```

CHAPTER 5

Text Processing

5.1 Strings or Character Vectors

Forming a character vector:

```
> 'abc'
[1] "abc"
> "abc"
[1] "abc"
```

A vector of character vectors:

```
> c("abc", "xyz")
[1] "abc" "xyz"
```

Pasting

`paste` takes multiple character vectors as input, combines them element by element using a separator, and collapses the resulting character vector if required.

Combining two character vectors:

```
> paste(c('a', 'b', 'c'), c('x', 'y', 'z'))
[1] "a x" "b y" "c z"
```

The default separator is blank space. Changing the separator:

```
> paste(c('a', 'b', 'c'), c('x', 'y', 'z'), sep='')
[1] "ax" "by" "cz"
> paste(c('a', 'b', 'c'), c('x', 'y', 'z'), sep='-')
[1] "a-x" "b-y" "c-z"
>
```

By default the result is another character vector. It is possible to collapse the result into a single string:

```
> paste(c('a', 'b', 'c'), c('x', 'y', 'z'), sep='', collapse=' ')
[1] "ax by cz"
> paste(c('a', 'b', 'c'), c('x', 'y', 'z'), sep='', collapse='')
[1] "axbycz"
> paste(c('a', 'b', 'c'), c('x', 'y', 'z'), sep='', collapse='o')
[1] "axobyocz"
> paste(c('a', 'b', 'c'), c('x', 'y', 'z'), sep='-', collapse='o')
[1] "a-xob-yoc-z"
```

The smaller vector is recycled if necessary:

```
> paste(c("a", "b", "c"), 1:8)
[1] "a 1" "b 2" "c 3" "a 4" "b 5" "c 6" "a 7" "b 8"
> paste(c("a", "b", "c"), 1:8, sep="")
[1] "a1" "b2" "c3" "a4" "b5" "c6" "a7" "b8"
```

Two strings can be combined easily:

```
> paste('hello', 'world')
[1] "hello world"
> paste('hello', 'world', sep='')
[1] "helloworld"
> paste('hello', 'world', sep='-')
[1] "hello-world"
> paste('hello', 'world', collapse='')
[1] "hello world"
```

It automatically converts any type into strings:

```
> paste(1:4, 5:8)
[1] "1 5" "2 6" "3 7" "4 8"
```

Can work with more than two vectors too:

```
> paste(1:5, 11:15, 21:25, 31:35, 41:45)
[1] "1 11 21 31 41" "2 12 22 32 42" "3 13 23 33 43" "4 14 24 34 44" "5 15 25 35 45"
> paste(1:5, 11:15, 21:25, 31:35, 41:45, sep='')
[1] "111213141" "212223242" "313233343" "414243444" "515253545"
```

This approach can be used for combining multiple strings into one:

```
> paste("a", "b", "c", sep="")
[1] "abc"
```

It is also possible to collapse a character vector into a single string easily:

```
> paste(c('a', 'b', 'c'))
[1] "a" "b" "c"
> paste(c('a', 'b', 'c'), collapse='')
[1] "abc"
> paste(c("abc", "xyz"), collapse="")
[1] "abcxyz"
```

Splitting strings

The function `strsplit` is the workhorse for splitting strings. It takes a character vector (a vector of strings) and returns a list of character vectors. Each input string corresponds to one output character vector in the output list.

Simple examples:

```
> strsplit('hello world', split=' ')
[[1]]
[1] "hello" "world"

> strsplit('hello world again', split=' ')
[[1]]
[1] "hello" "world" "again"

> strsplit('hello-world-again', split='-')
[[1]]
[1] "hello" "world" "again"

> strsplit('hello-world-again', split=' ')
[[1]]
[1] "hello-world-again"
```

Multiple character strings in input:

```
> strsplit(c('hello world', 'amazing world'), split=' ')
[[1]]
[1] "hello" "world"

[[2]]
[1] "amazing" "world"
```

It is possible to create a unified character vector at output:

```
> unlist(strsplit(c('hello world', 'amazing world'), split=' '))
[1] "hello"   "world"   "amazing" "world"
> unlist(strsplit(rep('c a', 10), split=' '))
 [1] "c" "a" "c" "a" "c" "a" "c" "a" "c" "a" "c" "a" "c" "a" "c" "a"
[17] "c" "a" "c" "a"
```

The split argument supports regular expressions:

```
> unlist(strsplit('hello world-again', split='[ -]'))
[1] "hello" "world" "again"
```

Splitting a name:

```
> unlist(strsplit("Ryan, Mr. Edward" , split='[, .]'))
[1] "Ryan"   ""       "Mr"     ""       "Edward"
```

Removing the blank strings:

```
> parts <- unlist(strsplit("Ryan, Mr. Edward" , split='[, .]'))
> parts
[1] "Ryan"   ""       "Mr"     ""       "Edward"
> ?filter
> parts[parts != ""]
[1] "Ryan"   "Mr"     "Edward"
```

White Space

```
> trimws(' hello')
[1] "hello"
> trimws(' hello ')
[1] "hello"
> trimws('hello  ')
[1] "hello"
> trimws(' hello ', which='left')
[1] "hello "
> trimws(' hello ', which='right')
[1] " hello"
> trimws(' hello ', which='both')
[1] "hello"
```

5.2 Pattern Matching and Replacement

`sub` replaces the first match of a pattern with replacement string.

Trimming whitespace at the beginning:

```
> sub(' ', '', ' hello')
[1] "hello"
> sub(' ', '', ' hello ')
[1] "hello "
```

CHAPTER 6

Data Manipulation

6.1 Basic Data Manipulation

6.1.1 Editing

Two functions are available for basic editing of data: `edit` and `fix`. They work on objects like vectors, matrices, data frames, etc.. Both of them open an editor for the user to make changes in the data object. After the editing is done, `fix` changes the object in place while `edit` returns a new object.

For updating an object using `edit` function:

```
> obj <- edit(obj)
```

This is same as

```
> fix(obj)
```

The result of edit operation can be assigned to a new object too:

```
> obj2 <- edit(obj)
```

6.1.2 Loading and Saving R Objects

Saving an R object:

```
save(obj, file='<filename>.rda')
```

Loading an R object:

```
load('<filename>.rda')
```

6.1.3 Reading and Writing Matrices and Vectors

Put this data into a file named `simple_3x4_matrix.txt`:

```
0 1 2 3
4 5 6 7
8 9 10 11
```

Reading the numbers as a vector

```
> scan('simple_3x4_matrix.txt')
Read 12 items
 [1]  0  1  2  3  4  5  6  7  8  9 10 11
```

Reading as a matrix via scan:

```
> matrix(scan('simple_3x4_matrix.txt'), nrow=3)
Read 12 items
     [,1] [,2] [,3] [,4]
[1,]    0    3    6    9
[2,]    1    4    7   10
[3,]    2    5    8   11
> matrix(scan('simple_3x4_matrix.txt'), nrow=3, byrow = T)
Read 12 items
     [,1] [,2] [,3] [,4]
[1,]    0    1    2    3
[2,]    4    5    6    7
[3,]    8    9   10   11
```

6.1.4 Reading and Writing Tables

Writing a data frame to file:

```
> d = data.frame(a=rnorm(3), b=rnorm(3))
> d
           a          b
1  0.9914006 -0.4930738
2 -0.5068710  0.5471745
3 -1.9964106  0.2247440
> write.table(d, file="tst0.txt", row.names=FALSE)
```

Reading a data frame from file:

```
> d2 = read.table(file="tst0.txt", header=TRUE)
```

Reading a table and converting it to a matrix:

```
> read.table('simple_3x4_matrix.txt')
  V1 V2 V3 V4
1  0  1  2  3
2  4  5  6  7
3  8  9 10 11
> as.matrix(read.table('simple_3x4_matrix.txt'))
     V1 V2 V3 V4
[1,]  0  1  2  3
[2,]  4  5  6  7
[3,]  8  9 10 11
```

Functions for reading tabular data and their defaults

Function	Purpose	Header	Separator	Decimal
`read.table`	Read tabular data from file	Absent		.
`read.csv`	Read from comma separated value files	Present	,	.
`read.csv2`	Read from semicolon separated value files	Present	;	,
`read.delim`	Read from tab delimited files	Present	\t	.
`read.delim2`	Read from tab delimited files	Present	\t	,

Suggestions

- Use `nrows=` argument to read only a few rows first to ensure that all the options to the reading function have been provided correctly.

Reading a table from a text string

Let the tabular data be stored in a character string text:

```
> my.data <- '
+ x y z
+ 1 2 3
+ 2 2 3
+ 3 2 4
+ 4 2 1
+ '
```

Let's prepare a connection to the text string:

```
> con <- textConnection(my.data)
```

Let us read the table from the text connection:

```
> read.table(con, header=T)
  x y z
1 1 2 3
2 2 2 3
3 3 2 4
4 4 2 1
```

6.2 dplyr

`dplyr` is a grammar for data manipulation. It provides a consistent set of operations which are sufficient to take care of most data manipulation tasks.

In a data frame we have columns representing different variables and rows representing different records.

The typical data manipulation tasks are:

- Selecting few variables (columns) from a data set
- Filtering the rows (cases) based on the values of specific variables (columns)
- Summarizing (reducing) the dataset

CHAPTER 7

Linear Algebra

7.1 Matrix Properties

Determinant of a matrix:

```
> det(matrix(c(1:4), nrow=2))
[1] -2
```

The rank of a matrix can be found through its QR decomposition:

```
> m <- matrix(c(1,2,3, 2,4,6, 3, 3, 3), nrow=3)
> qr(m)$rank
[1] 2
> det(m)
[1] 0
```

Trace of a square matrix:

```
> A <- matrix(c(1, 2, -1, 3, -6, 9, -1, 2, 1), nrow=3)
> A
     [,1] [,2] [,3]
[1,]    1    3   -1
[2,]    2   -6    2
[3,]   -1    9    1
> sum(diag(A))
[1] -4
```

7.2 Linear Equations

Solving a linear equation:

```
> A <- matrix(1:4, nrow=2)
> v <- c(1:2)
> b <- A %*% v
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> v
[1] 1 2
> b
     [,1]
[1,]    7
[2,]   10
> solve(A, b)
     [,1]
[1,]    1
[2,]    2
```

Computing the inverse of a matrix through solving the equation $AX = I$:

```
> A <- matrix(1:4, nrow=2)
> solve(A)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Computing the quadratic form $x^T A^{-1} x$:

```
> A <- matrix(1:4, nrow=2)
> x <- c(2,3)
> x %*% solve(A) %*% x
     [,1]
[1,]  2.5
> x %*% solve(A, x)
     [,1]
[1,]  2.5
```

The second approach is much more efficient and reliable.

7.3 Eigen Value Decomposition

Eigen value decomposition:

```
> A <- matrix(1:9, nrow=3)
> A <- A + t(A)
> eigen(A)
eigen() decomposition
$values
[1]  3.291647e+01 -5.329071e-15 -2.916473e+00

$vectors
           [,1]       [,2]       [,3]
[1,] -0.3516251  0.4082483  0.8424328
[2,] -0.5533562 -0.8164966  0.1647127
[3,] -0.7550872  0.4082483 -0.5130074
```

Let us verify the decomposition:

```
> A
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    6   10   14
[3,]   10   14   18


> e <- eigen(A)
> lambda <- diag(e$values)
> U <- e$vectors
> U %*% lambda %*% t(U)
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    6   10   14
[3,]   10   14   18
```

Computing only the eigen values:

```
> eigen(A, only.values = TRUE)
$values
[1]  3.291647e+01 -1.787388e-15 -2.916473e+00

$vectors
NULL
```

7.4 Singular Value Decomposition

SVD (Singular Value Decomposition) stands for splitting a matrix A into a product $A = USV^H$ where U and V are unitary matrices and S is a diagonal matrix consisting of singular values on its main diagonal arranged in non-increasing order where all the singular values are non-negative.

Computing the singular value decomposition of a matrix:

```
> A <- matrix(c(1, 2, -1, 3, -6, 9, -1, 2, 1), nrow=3)
> svd(A)
$d
[1] 11.355933  2.475195  1.707690

$u
           [,1]       [,2]       [,3]
[1,]  0.2526715 -0.1073216 -0.9615816
[2,] -0.5565826  0.7968092 -0.2351827
[3,]  0.7914373  0.5946235  0.1415976

$v
            [,1]      [,2]       [,3]
[1,] -0.14546854 0.3602437 -0.9214464
[2,]  0.98806904 0.1005140 -0.1166899
[3,] -0.05058143 0.9274273  0.3705672
```

Largest singular value:

```
svd(A)$d[1]
```

Smallest singular value:

```
> tail(svd(A)$d, n=1)
[1] 1.70769
```

Absolute value of determinant as product of singular values:

```
> det(A)
[1] -48
> prod(svd(A)$d)
[1] 48
```

7.5 QR Decomposition

We split a matrix A into a product $A = QR$ where Q is a matrix with unit norm orthogonal vectors and R is an upper triangular matrix.

An example matrix:

```
A <- matrix(c(1,2,3, 2,4,6, 3, 3, 3), nrow=3)
```

Computing the QR decomposition:

```
> QR <- qr(A)
```

Rank of the matrix:

```
> QR$rank
[1] 2
```

The Q factor:

```
> Q <- qr.Q(QR); Q
           [,1]       [,2]       [,3]
[1,] -0.2672612  0.8728716  0.4082483
[2,] -0.5345225  0.2182179 -0.8164966
[3,] -0.8017837 -0.4364358  0.4082483
```

The R factor:

```
> R <- qr.R(QR); R
          [,1]      [,2]      [,3]
[1,] -3.741657 -4.810702 -7.483315
[2,]  0.000000  1.963961  0.000000
[3,]  0.000000  0.000000  0.000000
```

We can reconstruct the matrix A from its decomposition as follows:

```
> qr.X(QR)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    3
[3,]    3    6    3
```

A method is also available to compute Qy for any vector y:

```
> qr.qy(QR, c(1,0, 0))
[1] -0.2672612 -0.5345225 -0.8017837
> qr.qy(QR, c(0,0, 1))
[1]  0.4082483 -0.8164966  0.4082483
> qr.qy(QR, c(0,1, 0))
[1]  0.8728716  0.2182179 -0.4364358
> qr.qy(QR, c(0,1, 1))
[1]  1.28111985 -0.59827869 -0.02818749
```

Another method is available to compute $Q^T y$ for any vector y:

```
> qr.qty(QR, c(1,0,0))
[1] -0.2672612  0.8728716  0.4082483
> qr.qty(QR, c(0,1,0))
[1] -0.5345225  0.2182179 -0.8164966
> qr.qty(QR, c(0,0,1))
[1] -0.8017837 -0.4364358  0.4082483
> qr.qty(QR, c(0,1,1))
[1] -1.3363062 -0.2182179 -0.4082483
```

Checking whether an object is a QR decomposition of a matrix:

```
> is.qr(A)
[1] FALSE
> is.qr(QR)
[1] TRUE
```

Solving a linear system of equations using the QR decomposition

Consider the linear system of equations $y = Ax$

```
> A <- matrix(c(3, 2, -1, 2, -2, .5, -1, 4, -1), nrow=3); A
     [,1] [,2] [,3]
[1,]    3  2.0   -1
[2,]    2 -2.0    4
[3,]   -1  0.5   -1
> x <- c(1, -2, -2); x
[1]  1 -2 -2
> y <- A %*% x ; y
     [,1]
[1,]    1
[2,]   -2
[3,]    0
```

Compute the QR decomposition of A

```
> QR <- qr(A)
> Q <- qr.Q(QR)
> R <- qr.R(QR)
```

Computing $b = Q^T y$:

```
> b <- crossprod(Q, y); b
           [,1]
[1,]  0.2672612
```

```
[2,]  2.1472519
[3,] -0.5638092
```

This can also be achieved as follows:

```
> qr.qty(QR, y)
           [,1]
[1,]  0.2672612
[2,]  2.1472519
[3,] -0.5638092
```

Solving the upper triangular system $Rx = b$:

```
> backsolve(R, b)
     [,1]
[1,]    1
[2,]   -2
[3,]   -2
```

Solving the equation in a single step:

```
> backsolve(R, crossprod(Q, y))
     [,1]
[1,]    1
[2,]   -2
[3,]   -2
```

The process of solving a linear equation using a QR decomposition can be performed in a single function call too:

```
> solve.qr(QR, y)
     [,1]
[1,]    1
[2,]   -2
[3,]   -2
```

7.6 Cholesky Decomposition

Let's choose a symmetric positive definite matrix:

```
> A <- matrix(c(4, 12, -16, 12, 37, -43, -16, -43, 98), nrow=3)
> A
```

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

[1,] 4 12 -16 [2,] 12 37 -43 [3,] -16 -43 98

Let's perform its Cholesky decomposition:

```
> U <- chol(A)
> U
     [,1] [,2] [,3]
[1,]    2    6   -8
[2,]    0    1    5
[3,]    0    0    3
```

Let's verify the correctness of the decomposition:

```
> t(U) %*% U
     [,1] [,2] [,3]
[1,]    4   12  -16
[2,]   12   37  -43
[3,]  -16  -43   98
```

Alternative way:

```
> crossprod(U)
     [,1] [,2] [,3]
[1,]    4   12  -16
[2,]   12   37  -43
[3,]  -16  -43   98
```

CHAPTER 8

Real Analysis

8.1 Metric Spaces and Distances

A metric space is a set of points equipped with a distance function.

8.1.1 Euclidean Distance

This is also known as l2-distance. For $x, y \in \mathbb{R}^n$, it is defined as:

$$d(x,y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$$

Here is a simple implementation in R:

```
distance.l2 <- function(x,y){
  sqrt(sum((abs(x - y))^2 ))
}
```

Let us use this function:

```
> x <- c(1,0,0)
> y <- c(0,1,0)
> distance.l2(x, y)
[1] 1.414214
```

8.1.2 Manhattan Distance

This is also known as city-block distance or l1-distance. For $x, y \in \mathbb{R}^n$, it is defined as:

$$d(x,y) = \sum_{i=1}^{n} |x_i - y_i|$$

Here is a simple implementation in R:

```
distance.l1 <- function(x,y){
  sum(abs(x-y))
}
```

Using this function:

```
> distance.l1(x, y)
[1] 2
```

8.1.3 Max Distance

For $x, y \in \mathbb{R}^n$, it is defined as:

$$d(x, y) = \max_{i=1}^{n} |x_i - y_i|$$

Here is a simple implementation in R:

```
distance.linf <- function(x,y){
  max(abs(x - y))
}
```

Using this function:

```
> distance.linf(x, y)
[1] 1
```

8.1.4 Minkowski Distance

This distance is a generalization of the l1, l2, and max distances. For $x, y \in \mathbb{R}^n$, the Minkowski distance of order p is defined as:

$$d(x, y) = \left[\sum_{i=1}^{n} |x_i - y_i|^p \right]^{\frac{1}{p}}$$

For $p = 1$, it reduces to city-block distance. For $p = 2$, it reduces to Euclidean distance. In the limit of $p \to \infty$, it reduces to max distance.

Here is a simple implementation in R:

```
distance.lp <- function(x, y, p){
  (sum((abs(x - y))^p))^(1/p)
}
```

Using this function:

```
> distance.lp(x, y, p=2)
[1] 1.414214
> distance.lp(x, y, p=.5)
[1] 4
> distance.lp(x, y, p=3)
[1] 1.259921
```

8.1.5 Canberra Distance

Canberra Distance is a weighted version of the Manhattan distance. For $x, y \in \mathbb{R}^n$, it is defined as:

$$d(x,y) = \sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i| + |y_i|}$$

Here is a simple implementation in R:

```
distance.canberra <- function(x, y){
  a1 <-  abs(x - y)
  a2 <- abs(x) + abs(y)
  a3 = a1 / a2
  sum(a1 / a2, na.rm=T)
}
```

Using this function:

```
> distance.canberra(x, y)
[1] 2
```

8.1.6 Binary or Jaccard Distance

Jaccard distance (a.k.a. binary distance) measures the dissimilarity between sample sets. For $x, y \in \mathbb{R}^n$ we identify $A = \{i : x_i \neq 0\}$ and $B = \{i : x_i \neq 0\}$. Then the distance is defined as:

$$d(x,y) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} = 1 - \frac{|A \cap B|}{|A \cup B|}.$$

Here is a simple implementation in R:

```
distance.binary <- function(x, y){
  x <- x != 0
  y <- y != 0
  a <- sum(x & y)
  b <- sum (x | y)
  1 - (a / b)
}
```

Using this function:

```
> a <- c(1, 0, 3, 0)
> b <- c(0, 2, 4, 0)
> distance.binary(a, b)
[1] 0.6666667
```

8.1.7 dist Function

R provides a function named `dist` which can compute all the distances described above. This function works on a data frame or a matrix. Every row is treated as a separate point in space. If the data frame has n rows, then the function computes $n(n-1)/2$ distances. It returns an object of which `dist` can be used to find out distances between each pair of points. The dist object can be converted into a $n \times n$ symmetric matrix containing the distances. By default, it computes Euclidean distances.

We will compute distances between unit-vectors in 3-dimensional space:

```
> eye <- diag(3)
> eye
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Computing the distances:

```
> d <- dist(eye)
```

The distances in the form of a symmetric matrix:

```
> as.matrix(d)
         1        2        3
1 0.000000 1.414214 1.414214
2 1.414214 0.000000 1.414214
3 1.414214 1.414214 0.000000
```

Computing Manhattan distances:

```
> d1 <- dist(eye, method='manhattan')
> as.matrix(d1)
  1 2 3
1 0 2 2
2 2 0 2
3 2 2 0
```

Computing maximum distances:

```
> dinf <- dist(eye, method='maximum')
> as.matrix(dinf)
  1 2 3
1 0 1 1
2 1 0 1
3 1 1 0
```

Minkowski distances:

```
> as.matrix(dist(eye, 'minkowski', p=0.5))
  1 2 3
1 0 4 4
2 4 0 4
3 4 4 0
> as.matrix(dist(eye, 'minkowski', p=3))
         1        2        3
1 0.000000 1.259921 1.259921
2 1.259921 0.000000 1.259921
3 1.259921 1.259921 0.000000
```

Canberra distances:

```
> as.matrix(dist(eye, 'canberra'))
  1 2 3
1 0 3 3
2 3 0 3
3 3 3 0
```

It is also straightforward to compute distance between two points as follows:

```
> a <- c(1, 0, 3, 0)
> b <- c(0, 2, 4, 0)
> dist(rbind(a, b))
         a
b 2.44949
> dist(rbind(a, b), method='manhattan')
  a
b 4
> dist(rbind(a, b), method='maximum')
  a
b 2
```

Computing the binary distance:

```
> dist(rbind(a, b), method='binary')
          a
b 0.6666667
```

Understanding the dist object

The `dist` function returns an object of class `dict`.

Let us create 4 points for this exercise:

```
> points <- diag(c(1,2,3,4))
> points
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

Let us compute the city block distances between these points:

```
> distances <- dist(points, method='manhattan')
```

Check the class of the returned value:

```
> class(distances)
[1] "dist"
```

Let us print the distances:

```
> distances
  1 2 3
2 3
3 4 5
4 5 6 7
> as.matrix(distances)
  1 2 3 4
1 0 3 4 5
2 3 0 5 6
3 4 5 0 7
4 5 6 7 0
```

If you note carefully, you can see that the distances object contains the lower triangle of the distance matrix [below the diagonal]. For 4 points, it stores 6 distances (1 + 2 + 3 = 4 * 3 / 2 = 6).

The number of points for which distances were calculated can be accessed from the dist object as follows:

```
> attr(distances, 'Size')
[1] 4
```

The dist object is a one dimensional vector. Assuming that there are n-points, then the distance between i-th point and j-th point where (1 <= i < j <= n) is stored at p-th index in the dist object where p is given by the following formula:

$$p = n(i-1) - i(i-1)/2 + j - i$$

Let us get n first:

```
> n <- attr(distances, 'Size')
```

Let us say we want to find the distance between 2nd and 4th point:

```
> i <- 2; j <- 4;
> distances[n*(i-1) - i*(i-1)/2 + j-i]
[1] 6
```

You can match the same with the distance matrix presented above. I guess it is much easier to first convert the dist object into a distance matrix and then work with it.

CHAPTER 9

Discrete Mathematics

9.1 Permutations and Combinations

9.1.1 Factorial

```
> factorial(4)
[1] 24
> factorial(1:4)
[1]  1  2  6 24
```

There is also a method to compute the log of a factorial:

```
> lfactorial(1:4)
[1] 0.0000000 0.6931472 1.7917595 3.1780538
> exp(lfactorial(1:4))
[1]  1  2  6 24
```

9.1.2 Gamma Function

This is a generalization of factorial `gamma(n) = factorial(n-1)`.

```
> gamma(1:10)
 [1]      1      1      2      6     24    120    720   5040  40320 362880
> factorial(0:9)
 [1]      1      1      2      6     24    120    720   5040  40320 362880
> gamma(1.5)
[1] 0.8862269
> factorial(.5)
[1] 0.8862269
```

9.1.3 Binomial coefficients

Number of ways to choose 2 balls from an urn containing 4 balls:

```
> choose(4, 2)
[1] 6
```

Binomial coefficients:

```
> n <- 1; choose(n, 0:n)
[1] 1 1
> n <- 2; choose(n, 0:n)
[1] 1 2 1
> n <- 3; choose(n, 0:n)
[1] 1 3 3 1
> n <- 4; choose(n, 0:n)
[1] 1 4 6 4 1
> n <- 5; choose(n, 0:n)
[1]  1  5 10 10  5  1
```

Another way using foreach:

```
> n <- 1; unlist(foreach(i=0:n) %do% choose(n,i))
[1] 1 1
> n <- 2; unlist(foreach(i=0:n) %do% choose(n,i))
[1] 1 2 1
> n <- 3; unlist(foreach(i=0:n) %do% choose(n,i))
[1] 1 3 3 1
> n <- 4; unlist(foreach(i=0:n) %do% choose(n,i))
[1] 1 4 6 4 1
> n <- 5; unlist(foreach(i=0:n) %do% choose(n,i))
[1]  1  5 10 10  5  1
> n <- 6; unlist(foreach(i=0:n) %do% choose(n,i))
[1]  1  6 15 20 15  6  1
> n <- 7; unlist(foreach(i=0:n) %do% choose(n,i))
[1]  1  7 21 35 35 21  7  1
> n <- 8; unlist(foreach(i=0:n) %do% choose(n,i))
[1]  1  8 28 56 70 56 28  8  1
```

Working with natural log of choose(n,k):

```
> choose(10, 0:10)
 [1]   1  10  45 120 210 252 210 120  45  10   1
> lchoose(10, 0:10)
 [1] 0.000000 2.302585 3.806662 4.787492 5.347108 5.529429 5.347108 4.787492 3.806662␣
↪2.302585 0.000000
> exp(lchoose(10, 0:10))
 [1]   1  10  45 120 210 252 210 120  45  10   1
```

9.1.4 Permutations

All permutations of 3 elements:

```
> permutations(3)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    1    3
```

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

9.1.5 Combinations

Listing all combinations of k elements chosen from n elements.

6 ways to choose 2 out of 4 elements:

```
> combn(4, 2)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    1    2    2    3
[2,]    2    3    4    3    4    4
```

4 ways to choose 3 out of 4 elements:

```
> combn(4, 3)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    2
[2,]    2    2    3    3
[3,]    3    4    4    4
```

Only one way to choose 4 out of 4 elements:

```
> combn(4,4)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

9.2 References

1. Combinatorics

CHAPTER 10

Probability

10.1 Random Numbers

Generate a vector of standard normal random numbers:

```
rnorm(10)
```

A Gaussian random matrix:

```
> matrix(rnorm(4*3), nrow=4, ncol=3)
           [,1]       [,2]       [,3]
[1,] -1.1436534  0.9533856 -0.6523511
[2,] -1.6479827  2.4177261  0.4608755
[3,] -0.5903960 -0.3371174  0.6652128
[4,]  0.3527609 -0.5014484  0.2769601
```

Uniformly distributed numbers between 0 and 1:

```
> runif(4)
[1] 0.8272435 0.9034066 0.5614103 0.1499100
```

Uniformly distributed numbers between 1 and 4:

```
> runif(4, min=1, max=4)
[1] 1.788217 3.501510 2.996803 1.741222
```

Seeding the random number sequence:

```
> set.seed(10)
> rnorm(10)
 [1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513  0.38979430 -1.
↪20807618 -0.36367602
 [9] -1.62667268 -0.25647839
> rnorm(10)
 [1]  1.10177950  0.75578151 -0.23823356  0.98744470  0.74139013  0.08934727 -0.
↪95494386 -0.19515038
```

```
 [9]  0.92552126  0.48297852
> set.seed(10)
> rnorm(10)
 [1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513  0.38979430 -1.
↪20807618 -0.36367602
 [9] -1.62667268 -0.25647839
> rnorm(10)
 [1]  1.10177950  0.75578151 -0.23823356  0.98744470  0.74139013  0.08934727 -0.
↪95494386 -0.19515038
 [9]  0.92552126  0.48297852
```

10.2 Coin Tossing

Bernoulli trials can be generated from the binomial distribution. Here is a way to generate a sequence of trials:

```
> rbinom(10, size=1, prob=0.5)
 [1] 0 1 0 1 1 1 0 0 1 1
```

We can use `table` to verify the distribution of 0s and 1s:

```
> table(rbinom(1000, size=1, prob=0.5))

  0   1
513 487
```

Varying the probability of 1 has the desired impact:

```
> table(rbinom(1000, size=1, prob=0.4))

  0   1
629 371
```

10.3 Sampling Data

The population from which samples will be created:

```
> v = 1:10
```

Sample one value from a vector:

```
> sample(v, 1)
[1] 9
> sample(v, 1)
[1] 1
> sample(v, 1)
[1] 2
> sample(v, 1)
[1] 9
```

Sampling multiple values without replacement:

```
> sample(v, 4)
[1]  1  7 10  3
> sample(v, 4)
[1]  6 10  5  3
> sample(v, 4)
[1] 10  4  1  9
> sample(v, 4)
[1] 5 2 4 3
```

Sampling all values without replacement:

```
> sample(v)
 [1]  8 10  3  5  2  7  4  9  1  6
> sample(v)
 [1]  6  7  1 10  4  5  3  9  2  8
```

This is essentially a random permutation of the original vector.

Sampling with replacement:

```
> sample(v, replace=TRUE)
 [1]  5  1  5  5  3  7  9 10  5  6
> sample(v, replace=TRUE)
 [1]  4  3 10  9 10  9  6  8  6  3
```

Notice that some values are repeating and some values are missing.

We can sample as many values as we want with replacement:

```
> sample(v, 20, replace=TRUE)
 [1]  8  6  1  8  7 10  4  4  2  2  9  5  9  7  7  6  1  3  9  6
```

10.4 Normal Distribution

The normal density function is given by

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

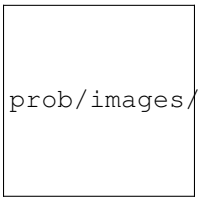
Probability Density Function

Evaluating the density function for different values of x, μ and σ:

```
> dnorm(x=0, mean = 0, sd = 1)
[1] 0.3989423
> dnorm(x=-4:4, mean = 0, sd = 1)
[1] 0.0001338302 0.0044318484 0.0539909665 0.2419707245 0.3989422804 0.2419707245 0.
↪0539909665 0.0044318484
[9] 0.0001338302
> dnorm(x=-3:5, mean = 1, sd = 1)
[1] 0.0001338302 0.0044318484 0.0539909665 0.2419707245 0.3989422804 0.2419707245 0.
↪0539909665 0.0044318484
[9] 0.0001338302
```

We can use `dnorm` to plot the PDF of normal distribution:

```
> x <- seq(-4,4,by=0.01)
> y <- dnorm(x)
> plot(x,y, 'l', main ='Normal Distribution')
```



Cumulative Distribution Function

The function `pnorm()` is used to compute the CDF of normal distribution up to any point on the real line:

```
> pnorm(0)
[1] 0.5
> pnorm(1)
[1] 0.8413447
> pnorm(-1)
[1] 0.1586553
> pnorm(1, mean=1)
[1] 0.5
> pnorm(-4:4)
[1] 3.167124e-05 1.349898e-03 2.275013e-02 1.586553e-01 5.000000e-01 8.413447e-01 9.
↪772499e-01 9.986501e-01
[9] 9.999683e-01
```

By default `pnorm` gives the integral of the PDF from :math:`-\infty` to `q`. It is also possible to compute the integral from `q` to ∞ using the `lower.tail` parameter:

```
> pnorm(0, lower.tail = FALSE)
[1] 0.5
> pnorm(1, lower.tail = FALSE)
[1] 0.1586553
> pnorm(-1, lower.tail = FALSE)
[1] 0.8413447
```

Note that `pnorm(x) + pnorm(x, lower.tail=FALSE)=1`.

Quantile or Inverse Cumulative Distribution Function

We can use the `qnorm` function to compute the z-score for a given quantile value:

```
> qnorm(c(0, .25, .5, .75, 1))
[1]       -Inf -0.6744898  0.0000000  0.6744898        Inf
> qnorm(.5, mean=1)
[1] 1
> qnorm(pnorm(-3:3))
[1] -3 -2 -1  0  1  2  3
```

Finally, we use `rnorm` for generating random numbers from the normal distribution.

Hazard function

Hazard function is given by $H(x) = -log(1 - F(x))$.

This can be computed as follows:

```
> q =1
> -log(pnorm(q, lower.tail = FALSE))
[1] 1.841022
```

Log likelihood

Log likelihood function is given by $log(f(x))$.

This can be computed by:

```
> dnorm(x, log=TRUE)
[1] -0.9189385
```

10.4.1 Bivariate Normal Distribution

In this section, we will look at different ways to generate samples from bivariate normal distribution.

Let our random variable be denoted as X = (X1, X2). Let the number of samples to be generated be N.

The simplest case is when both X1 and X2 are independent standard normal variables:

```
> N <- 1000
> set.seed(123)
> samples <- matrix(rnorm(N*2), ncol=2)
> colMeans(samples)
[1] 0.01612787 0.04246525
> cov(samples)
           [,1]       [,2]
[1,] 0.9834589 0.0865909
[2,] 0.0865909 1.0194419
```

The next case is when the two variables are independent but have different means:

```
> mu1 <- 1
> mu2 <- 2
> samples <- cbind(rnorm(N, mean=mu1), rnorm(N, mean=mu2))
> colMeans(samples)
[1] 0.9798875 1.9908396
> cov(samples)
            [,1]       [,2]
[1,] 0.95718335 0.04908825
[2,] 0.04908825 0.98476186
```

There is a function called `mvrnorm` in the `MASS` package which is very flexible:

```
> mu1 <- 1
> mu2 <- 2
> mu <- c(mu1, mu2)
> sd1 <- 2
> sd2 <- 4
```

```
> corr <- 0.6
> Sigma <- matrix(c(sd1, corr, corr, sd2), nrow=2)
> library(MASS)
> N <- 10000
> samples <- mvrnorm(N, mu=mu, Sigma=Sigma)
> colMeans(samples)
[1] 0.9976949 2.0208528
> cov(samples)
          [,1]      [,2]
[1,] 1.9889508 0.6005303
[2,] 0.6005303 4.0516402
```

10.5 Hyper-Geometric Distribution

We have an urn which has m white balls and n black balls. We draw k balls from it. We want to find the probability of picking x white balls.

Let's examine the simple case with m=1,n=1, k=1:

```
> m <- 1
> n <- 1
> k <- 1
> x <- 0:2
> dhyper(x, m, n, k)
[1] 0.5 0.5 0.0
```

There is 50% chance that the first ball is black, 50% chance that first ball is white, and 0% chance that we have drawn 2 white balls.

Now, let us examine the situation for k=2:

```
> k <- 2
> x <- 0:3
> dhyper(x, m, n, k)
[1] 0 1 0 0
```

In two trials, both white and black balls will come out. There will be exactly one white ball.

Let's consider the case where m=2, n=2, k=2:

```
> m <- 2
> n <- 2
> k <- 2
> x <-0:4
> dhyper(x, m, n, k)
[1] 0.1666667 0.6666667 0.1666667 0.0000000 0.0000000
```

10.6 Discrete Distributions

Let us define our sample space:

```
> sample.space <- c(1,2,3, 4)
```

Let us define the probability mass function over the sample space:

```
> pmf <- c(0.25, 0.3, 0.35, 0.1)
> sum(pmf)
[1] 1
```

Let's draw some samples from this distribution:

```
> sample(sample.space, size=10, replace=T, prob=pmf)
 [1] 2 3 3 3 3 4 2 1 3 2
```

Let's tabulate them for large number of samples:

```
> table(sample(sample.space, size=10000, replace=T, prob=pmf))

   1    2    3    4
2578 3059 3383  980
```

Let's verify their proportions:

```
> prop.table(table(sample(sample.space, size=10000, replace=T, prob=pmf)))

     1      2      3      4
0.2522 0.3029 0.3505 0.0944
```

Note that this matches quite well with the original probability mass function.

10.7 Standard Probability Distributions

Functions for several probability distributions are provided as part of R. Some distribution are available in the global space. Additional distributions are available through some packages.

Each distribution has a name in R (e.g. norm, beta, t, f, etc.). For each distribution following functions are provided:

- Probability Density (Mass) Function (dnorm, dbeta, dt, df, etc.)
- Cumulative Distribution Function (pnorm, pbeta, , pt, pf, etc.)
- Quantile or Inverse Cumulative Distribution Function (qnorm, qbeta, qt, qf, etc.)
- Random Number Generator for the given distribution (rnorm, rbeta, rt, rf, etc.)

Distribution	R name	Additional arguments
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
chi-squared	chisq	df, ncp
exponential	exp	rate
F	f	df1, df2, ncp
gamma	gamma	shape, scale
geometric	geom	prob
hypergeometric	hyper	m, n, k
log-normal	lnorm	meanlog, sdlog
logistic	logis	location, scale
negative binomial	nbinom	size, prob
normal	norm	mean, sd
Poisson	pois	lambda
signed rank	signrank	n
Student's t	t	df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

10.8 Kernel Density Estimation

A kernel is a special type of probability density function (PDF) with the added property that it must be even. Thus, a kernel is a function with the following properties

- real-valued
- non-negative
- even
- its definite integral over its support set must equal to 1

A bump is assigned to each data point. The size of the bump is proportional to the number of points at that value. The estimated density function is the average of bumps over all data points.

The density() function in R computes the values of the kernel density estimate.

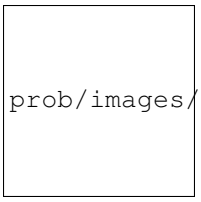
Let us estimate and plot the PDF of eruptions from faithful dataset:

```
> plot(density(faithful$eruptions))
```

prob/images/faithful_eruptions_density_estimate.png

A more reliable approach for automatic estimation of bandwidth:

```
> plot(density(faithful$eruptions, bw='SJ'))
```


prob/images/faithful_eruptions_density_estimate_sj.png

CHAPTER 11

Statistics

11.1 Basic Statistics

Summary Statistics

Some data:

```
> x <- rnorm(100)
```

Maximum value:

```
> max(x)
[1] 3.251759
```

Minimum value:

```
> min(x)
[1] -2.340614
```

Sum of all values:

```
> sum(x)
[1] 8.446345
```

Mean of all values:

```
> mean(x)
[1] 0.08446345
```

Median of all values:

```
> median(x)
[1] 0.0814703
```

Range (min, max) of all values:

```
> range(x)
[1] -2.340614  3.251759
```

Parallel max and min:

```
> a <- sample(1:10)
> b <- sample(1:10)
> a
 [1]  8  5  9 10  4  2  6  1  7  3
> b
 [1]  4  6  9 10  2  8  7  1  3  5
> pmax(a,b)
 [1]  8  6  9 10  4  8  7  1  7  5
> pmin(a,b)
 [1]  4  5  9 10  2  2  6  1  3  3
```

Summary statistics for all variables in a data frame can be obtained simultaneously:

```
> data("mtcars")
> summary(mtcars)
      mpg              cyl             disp             hp             drat
↪ wt
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0   Min.   :2.760   Min.
↪   :1.513
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080   1st
↪Qu.:2.581
 Median :19.20   Median :6.000   Median :196.3   Median :123.0   Median :3.695
↪Median :3.325
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7   Mean   :3.597   Mean
↪   :3.217
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920   3rd
↪Qu.:3.610
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0   Max.   :4.930   Max.
↪   :5.424
      qsec             vs              am              gear            carb
 Min.   :14.50   Min.   :0.0000   Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:16.89   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :17.71   Median :0.0000   Median :0.0000   Median :4.000   Median :2.000
 Mean   :17.85   Mean   :0.4375   Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:18.90   3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :22.90   Max.   :1.0000   Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Variance, Covariance

Computing the sample variance:

```
> var(mtcars$mpg)
[1] 36.3241
```

The mean square value:

```
> n <- length(mtcars$mpg)
> ms <- sum(mtcars$mpg^2) / n
> ms
[1] 438.8222
```

Verifying the variance and mean square value relationship:

```
> ms - mean(mtcars$mpg)^2
[1] 35.18897
> var(mtcars$mpg)  * (n - 1) / n
[1] 35.18897
```

Computing the variance of each variable in a data frame:

```
> round(sapply(mtcars, var), digits=2)
     mpg      cyl     disp       hp     drat       wt     qsec       vs       am
↪gear     carb
   36.32     3.19 15360.80  4700.87     0.29     0.96     3.19     0.25     0.25
↪0.54     2.61
```

Variances of selected columns:

```
> sapply(mtcars[, c('cyl', 'disp', 'wt')], var)
        cyl         disp           wt
   3.189516 15360.799829     0.957379
```

Computing the covariance matrix for all variables in a data frame:

```
> round(cov(mtcars), digits=2)
         mpg    cyl     disp       hp   drat     wt   qsec     vs     am   gear  carb
mpg    36.32  -9.17  -633.10 -320.73   2.20  -5.12   4.51   2.02   1.80   2.14 -5.36
cyl    -9.17   3.19   199.66  101.93  -0.67   1.37  -1.89  -0.73  -0.47  -0.65  1.52
disp -633.10 199.66 15360.80 6721.16 -47.06 107.68 -96.05 -44.38 -36.56 -50.80 79.07
hp   -320.73 101.93  6721.16 4700.87 -16.45  44.19 -86.77 -24.99  -8.32  -6.36 83.04
drat    2.20  -0.67   -47.06  -16.45   0.29  -0.37   0.09   0.12   0.19   0.28 -0.08
wt     -5.12   1.37   107.68   44.19  -0.37   0.96  -0.31  -0.27  -0.34  -0.42  0.68
qsec    4.51  -1.89   -96.05  -86.77   0.09  -0.31   3.19   0.67  -0.20  -0.28 -1.89
vs      2.02  -0.73   -44.38  -24.99   0.12  -0.27   0.67   0.25   0.04   0.08 -0.46
am      1.80  -0.47   -36.56   -8.32   0.19  -0.34  -0.20   0.04   0.25   0.29  0.05
gear    2.14  -0.65   -50.80   -6.36   0.28  -0.42  -0.28   0.08   0.29   0.54  0.33
carb   -5.36   1.52    79.07   83.04  -0.08   0.68  -1.89  -0.46   0.05   0.33  2.61
```

Computing the covariance matrix of selected variables:

```
> cov(mtcars[, c('cyl', 'disp', 'wt')])
            cyl       disp         wt
cyl    3.189516   199.6603   1.367371
disp 199.660282 15360.7998 107.684204
wt     1.367371   107.6842   0.957379
```

Computing the standard deviation:

```
> sd(mtcars$mpg)
[1] 6.026948
```

Standard deviation of each variable in a data frame:

```
> sapply(mtcars, sd)
       mpg         cyl        disp          hp        drat          wt        qsec
↪         vs
  6.0269481   1.7859216 123.9386938  68.5628685   0.5346787   0.9784574   1.7869432
↪0.5040161
        am        gear        carb
  0.4989909   0.7378041   1.6152000
```

Pearson Correlation

Pearson correlation coefficients are useful in estimating dependence between different (numeric) variables. The value varies from 0 to 1. This corresponds between no correlation to complete correlation.

Pearson coefficient	Interpretation
0.00 - 0.19	very weak correlation
0.20 - 0.39	weak correlation
0.40 - 0.59	moderate correlation
0.60 - 0.79	strong correlation
0.80 - 1.00	very strong correlation

Computing Pearson correlation coefficients for all variables in a data frame:

```
> round(cor(mtcars), digits=2)
       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
mpg   1.00 -0.85 -0.85 -0.78  0.68 -0.87  0.42  0.66  0.60  0.48 -0.55
cyl  -0.85  1.00  0.90  0.83 -0.70  0.78 -0.59 -0.81 -0.52 -0.49  0.53
disp -0.85  0.90  1.00  0.79 -0.71  0.89 -0.43 -0.71 -0.59 -0.56  0.39
hp   -0.78  0.83  0.79  1.00 -0.45  0.66 -0.71 -0.72 -0.24 -0.13  0.75
drat  0.68 -0.70 -0.71 -0.45  1.00 -0.71  0.09  0.44  0.71  0.70 -0.09
wt   -0.87  0.78  0.89  0.66 -0.71  1.00 -0.17 -0.55 -0.69 -0.58  0.43
qsec  0.42 -0.59 -0.43 -0.71  0.09 -0.17  1.00  0.74 -0.23 -0.21 -0.66
vs    0.66 -0.81 -0.71 -0.72  0.44 -0.55  0.74  1.00  0.17  0.21 -0.57
am    0.60 -0.52 -0.59 -0.24  0.71 -0.69 -0.23  0.17  1.00  0.79  0.06
gear  0.48 -0.49 -0.56 -0.13  0.70 -0.58 -0.21  0.21  0.79  1.00  0.27
carb -0.55  0.53  0.39  0.75 -0.09  0.43 -0.66 -0.57  0.06  0.27  1.00
```

Computing Pearson correlation coefficients for selected variables:

```
> cor(mtcars[, c('cyl', 'disp', 'wt')])
           cyl      disp        wt
cyl  1.0000000 0.9020329 0.7824958
disp 0.9020329 1.0000000 0.8879799
wt   0.7824958 0.8879799 1.0000000
```

Here is a way to map the actual correlation values to 5 ranges.

Let us compute the correlation coefficients for numerical variables in iris dataset:

```
> iris.correlations <- cor(iris[, -c(5)])
> iris.correlations
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
```

Note that we have left out the Species variable which is a factor variable.

Let us use `cut` to break it into 5 ranges:

```
iris.correlation.levels <- cut(abs(iris.correlations), breaks=c(0, .2, .4, .6, .8, 1.
↪0), include.lowest = T, labels = c('VW', 'WK', 'MD', 'ST', 'VS'))
```

Cut returns a vector. We need to reshape it into a matrix:

```
> iris.correlation.levels<- matrix(iris.correlation.levels, nrow=4)
> iris.correlation.levels
     [,1] [,2] [,3] [,4]
[1,] "VS" "VW" "VS" "VS"
[2,] "VW" "VS" "MD" "WK"
[3,] "VS" "MD" "VS" "VS"
[4,] "VS" "WK" "VS" "VS"
```

We are still missing the row names and column names. Let us get them back too:

```
> rownames(iris.correlation.levels) <- rownames(iris.correlations)
> colnames(iris.correlation.levels) <- colnames(iris.correlations)
> iris.correlation.levels
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length "VS"         "VW"        "VS"         "VS"
Sepal.Width  "VW"         "VS"        "MD"         "WK"
Petal.Length "VS"         "MD"        "VS"         "VS"
Petal.Width  "VS"         "WK"        "VS"         "VS"
```

Some interesting exercises:

```
> x <- rnorm(100)
> cor(x, x)
[1] 1
> cor(x, abs(x))
[1] 0.03242731
> cor(x, x^2)
[1] -0.01069063
> cor(abs(x), x^2)
[1] 0.9333162
> cor(x, x^3)
[1] 0.7631594
> cor(abs(x), abs(x)^3)
[1] 0.8048567
> cor(abs(x), x^4)
[1] 0.6817026
> cor(abs(x), log(abs(x)))
[1] 0.8360999
```

Spearman Correlation

The Spearman correlation method is suitable for computing correlation of a factor variable with other variables. In the next example, we will compute the Spearman correlation of Species variable with other variables in the iris dataset.

In order to compute the correlation, a factor variable needs to be cast as a numeric variable:

```
> as.numeric(iris$Species)
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1␣
↪1 1 1 1 1 1 1
 [48] 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2␣
↪2 2 2 2 2 2 2
 [95] 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3␣
↪3 3 3 3 3 3 3
[142] 3 3 3 3 3 3 3 3 3
```

We can use this for computing the correlations:

```
> cor(as.numeric(iris$Species), iris[, -5], method='spearman')
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]    0.7980781  -0.4402896    0.9354305   0.9381792
```

Note that we have removed the Species variable from the second parameter for computing the correlations.

This may look a bit cumbersome if the number of variables is large. Here is a cleaner look:

```
> t(cor(as.numeric(iris$Species), iris[, -5], method='spearman'))
                   [,1]
Sepal.Length  0.7980781
Sepal.Width  -0.4402896
Petal.Length  0.9354305
Petal.Width   0.9381792
```

It can be made a little bit more beautiful:

```
> iris.species.correlations <- t(cor(as.numeric(iris$Species), iris[, -5], method=
↪'spearman'))
> colnames(iris.species.correlations) <- c('correlation')
> iris.species.correlations
             correlation
Sepal.Length   0.7980781
Sepal.Width   -0.4402896
Petal.Length   0.9354305
Petal.Width    0.9381792
```

We note that petal length and width are very strongly correlated with the species.

Tukey Five Number Summary

The five numbers include: minimum, lower-hinge, median, upper-hinge, maximum:

```
> fivenum(mtcars$mpg)
[1] 10.40 15.35 19.20 22.80 33.90
```

Quantiles

Computing the quantiles of a given data:

```
> quantile(mtcars$mpg)
    0%    25%    50%    75%   100%
10.400 15.425 19.200 22.800 33.900
> quantile(sort(mtcars$mpg))
    0%    25%    50%    75%   100%
10.400 15.425 19.200 22.800 33.900
> quantile(mtcars$mpg, probs=c(0.1, 0.2, 0.4, 0.8, 1.0))
  10%   20%   40%   80%  100%
14.34 15.20 17.92 24.08 33.90
```

```
> IQR(mtcars$mpg)
[1] 7.375
```

Median Absolute Deviation

```
> mad(mtcars$mpg)
[1] 5.41149
```

Skewness

This is available in `e1071` library:

```
> library(e1071)
> skewness(mtcars$mpg)
[1] 0.610655
> skewness(discoveries)
[1] 1.2076
```

Kurtosis

This is available in `e1071` library:

```
> library(e1071)
> kurtosis(mtcars$mpg)
[1] -0.372766
> kurtosis(discoveries)
[1] 1.989659
```

- Samples with negative kurtosis value are called *platykurtic*.
- Samples with positive kurtosis values are called *leptokurtic*.
- Samples with kurtosis very close to 0 are called *mesokurtic*.

Scaling or Standardizing a Variable

Let us pick a variable and check its mean and variance:

```
> x <- mtcars$mpg
> mean(x)
[1] 20.09062
> var(x)
[1] 36.3241
```

Let us now scale it to zero mean unit variance:

```
> y <- scale(x)
> mean(y)
[1] 7.112366e-17
> var(y)
     [,1]
[1,]    1
```

Scaling whole data frame:

```
> mtcars2 <- scale(mtcars)
```

Let us verify the means:

```
> colMeans(mtcars)
       mpg        cyl       disp         hp       drat         wt       qsec
↪vs
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750   0.
↪437500
        am       gear       carb
  0.406250   3.687500   2.812500
> colMeans(mtcars2)
          mpg           cyl          disp            hp          drat            wt
 7.112366e-17 -1.474515e-17 -9.085614e-17  1.040834e-17 -2.918672e-16  4.682398e-17
         qsec            vs            am          gear          carb
 5.299580e-16  6.938894e-18  4.510281e-17 -3.469447e-18  3.165870e-17
```

Note that the original means are still maintained inside the scaled data frame as an attribute:

```
> attr(mtcars2, 'scaled:center')
       mpg        cyl       disp         hp       drat         wt       qsec
↪vs
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750   0.
↪437500
        am       gear       carb
  0.406250   3.687500   2.812500
```

And so are original standard deviations:

```
> attr(mtcars2, 'scaled:scale')
       mpg        cyl       disp         hp       drat         wt       qsec
↪       vs
  6.0269481   1.7859216 123.9386938  68.5628685   0.5346787   0.9784574   1.7869432
↪0.5040161
        am       gear       carb
  0.4989909   0.7378041   1.6152000
```

Verifying that the scaled data frame indeed has unit variance:

```
> apply(mtcars, 2, sd)
       mpg        cyl       disp         hp       drat         wt       qsec
↪       vs
  6.0269481   1.7859216 123.9386938  68.5628685   0.5346787   0.9784574   1.7869432
↪0.5040161
        am       gear       carb
  0.4989909   0.7378041   1.6152000
> apply(mtcars, 2, var)
          mpg           cyl          disp            hp          drat            wt
↪qsec
3.632410e+01 3.189516e+00 1.536080e+04 4.700867e+03 2.858814e-01 9.573790e-01 3.
↪193166e+00
           vs            am          gear          carb
2.540323e-01 2.489919e-01 5.443548e-01 2.608871e+00
> apply(mtcars2, 2, var)
 mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
   1    1    1    1    1    1    1    1    1    1    1
```

Scaling a data frame which contains a mixture of numeric and factor variables is a bit more involved. We will work

with iris dataset for this example:

```
> iris2 <- iris
```

We first identify the variables which are numeric:

```
> ind <- sapply(iris2, is.numeric)
> ind
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
        TRUE         TRUE         TRUE         TRUE        FALSE
```

Next we scale these variables:

```
> iris2.scaled <- scale(iris2[ind])
> attr(iris2.scaled, 'scaled:center')
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
> attr(iris2.scaled, 'scaled:scale')
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   0.8280661    0.4358663    1.7652982    0.7622377
```

Time to merge it back:

```
> iris2[ind] <- iris2.scaled
```

Verify that the numeric columns have indeed been scaled:

```
> sapply(iris2[ind], mean)
 Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
-4.484318e-16  2.034094e-16 -2.895326e-17 -3.663049e-17
> sapply(iris2[ind], sd)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           1            1            1            1
> sapply(iris2[ind], var)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           1            1            1            1
```

11.1.1 Group Wise Statistics

We will compute summary statistics for each species in iris database:

```
> data("iris")
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
> tapply(iris$Petal.Length, iris$Species, summary)
$setosa
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.400   1.500   1.462   1.575   1.900

$versicolor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
   3.00    4.00    4.35    4.26    4.60    5.10

$virginica
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.500   5.100   5.550   5.552   5.875   6.900
```

We can compute individual group-wise statistics too:

```
> tapply(iris$Petal.Length, iris$Species, mean)
    setosa versicolor  virginica
     1.462      4.260      5.552
> tapply(iris$Petal.Length, iris$Species, max)
    setosa versicolor  virginica
       1.9        5.1        6.9
> tapply(iris$Petal.Length, iris$Species, var)
    setosa versicolor  virginica
0.03015918 0.22081633 0.30458776
> tapply(iris$Petal.Length, iris$Species, min)
    setosa versicolor  virginica
       1.0        3.0        4.5
```

Frequency Tables

When we factor a list into levels, we can compute the frequency table from the factors as follows:

```
> states <- sample(datasets::state.name[1:10], 20, replace=TRUE)
> statesf <- factor(states)
> table(statesf)
statesf
    Alabama      Alaska     Arizona  California    Colorado Connecticut    Delaware
↪  Florida     Georgia
          1           1           1           2           2           2           3
↪         5           3
```

Looking at the tabulation in proportional terms:

```
> states <- sample(datasets::state.name[1:10], 20, replace=TRUE)
> statesf <- factor(states)
> prop.table(table(statesf))
statesf
   Alabama     Alaska   Arkansas California   Colorado   Delaware    Florida
↪Georgia
      0.05       0.10       0.25       0.15       0.05       0.15       0.15       0.
↪10
```

Building a two-dimensional frequency table

Here is a simple example. We will extract the fuel type and vehicle class from the mpg data set and tabulate the co-occurrence of pairs of these two variables:

```
> table(mpg[, c('fl', 'class')])
   class
fl  2seater compact midsize minivan pickup subcompact suv
  c       0       0       0       0      0          1   0
```

```
  d       0       1       0       0       0          2   2
  e       0       0       0       1       3          0   4
  p       5      21      15       0       0          3   8
  r       0      25      26      10      30         29  48
```

Let's convert this table into a simple data frame:

```
> df <- as.data.frame(table(mpg[, c('fl', 'class')]))
> df[df$Freq != 0,]
   fl      class Freq
4   p    2seater    5
7   d    compact    1
9   p    compact   21
10  r    compact   25
14  p    midsize   15
15  r    midsize   26
18  e    minivan    1
20  r    minivan   10
23  e     pickup    3
25  r     pickup   30
26  c subcompact    1
27  d subcompact    2
29  p subcompact    3
30  r subcompact   29
32  d        suv    2
33  e        suv    4
34  p        suv    8
35  r        suv   48
```

Note that only 18 of the rows (or combinations of fuel type and vehicle class) have non-zero entries.

US states income data:

```
> incomes <- datasets::state.x77[,2]
> summary(incomes)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3098    3993    4519    4436    4814    6315
```

Categorizing the income data:

```
> incomes_fr <- cut(incomes, breaks=2500+1000*(0:4), dig.lab = 4)
> incomes_fr
 [1] (3500,4500] (5500,6500] (4500,5500] (2500,3500] (4500,5500] (4500,5500] (4500,
↪5500] (4500,5500]
 [9] (4500,5500] (3500,4500] (4500,5500] (3500,4500] (4500,5500] (3500,4500] (4500,
↪5500] (4500,5500]
[17] (3500,4500] (3500,4500] (3500,4500] (4500,5500] (4500,5500] (4500,5500] (4500,
↪5500] (2500,3500]
[25] (3500,4500] (3500,4500] (4500,5500] (4500,5500] (3500,4500] (4500,5500] (3500,
↪4500] (4500,5500]
[33] (3500,4500] (4500,5500] (4500,5500] (3500,4500] (4500,5500] (3500,4500] (4500,
↪5500] (3500,4500]
[41] (3500,4500] (3500,4500] (3500,4500] (3500,4500] (3500,4500] (4500,5500] (4500,
↪5500] (3500,4500]
[49] (3500,4500] (4500,5500]
Levels: (2500,3500] (3500,4500] (4500,5500] (5500,6500]
```

Tabulating the income data frequencies:

```
> table(incomes_fr)
incomes_fr
(2500,3500] (3500,4500] (4500,5500] (5500,6500]
          2          22          25           1
```

US states illiteracy data:

```
> illiteracy <- datasets::state.x77[,3]
> summary(illiteracy)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.500   0.625   0.950   1.170   1.575   2.800
```

Categorizing the illiteracy data:

```
> illiteracy_fr <- cut(illiteracy, breaks=c(0, .5, 1.0, 1.5, 2.0,2.5, 3.0))
> illiteracy_fr
 [1] (2,2.5] (1,1.5] (1.5,2] (1.5,2] (1,1.5] (0.5,1] (1,1.5] (0.5,1] (1,1.5] (1.5,2]␣
↪(1.5,2] (0.5,1] (0.5,1]
[14] (0.5,1] (0,0.5] (0.5,1] (1.5,2] (2.5,3] (0.5,1] (0.5,1] (1,1.5] (0.5,1] (0.5,1]␣
↪(2,2.5] (0.5,1] (0.5,1]
[27] (0.5,1] (0,0.5] (0.5,1] (1,1.5] (2,2.5] (1,1.5] (1.5,2] (0.5,1] (0.5,1] (1,1.5]␣
↪(0.5,1] (0.5,1] (1,1.5]
[40] (2,2.5] (0,0.5] (1.5,2] (2,2.5] (0.5,1] (0.5,1] (1,1.5] (0.5,1] (1,1.5] (0.5,1]␣
↪(0.5,1]
Levels: (0,0.5] (0.5,1] (1,1.5] (1.5,2] (2,2.5] (2.5,3]
```

Tabulating the illiteracy data frequencies:

```
> table(illiteracy_fr)
illiteracy_fr
(0,0.5] (0.5,1] (1,1.5] (1.5,2] (2,2.5] (2.5,3]
      3      23      11       7       5       1
```

Tabulating income vs illiteracy

```
> table(incomes_fr, illiteracy_fr)
             illiteracy_fr
incomes_fr     (0,0.5] (0.5,1] (1,1.5] (1.5,2] (2,2.5] (2.5,3]
  (2500,3500]       0       0       0       1       1       0
  (3500,4500]       1      10       2       4       4       1
  (4500,5500]       2      13       8       2       0       0
  (5500,6500]       0       0       1       0       0       0
```

Aggregation

Computing mean of sepal length for each species in iris:

```
> aggregate(iris$Sepal.Length, by=list(iris$Species), FUN=mean)
     Group.1     x
1     setosa 5.006
2 versicolor 5.936
3  virginica 6.588
```

Computing mean mileage per gallon of cars aggreated by their number of cylinders and V/S:

```
> unique(mtcars$cyl)
[1] 6 4 8
> unique(mtcars$vs)
[1] 0 1
> aggregate(mtcars$mpg, by=list(mtcars$cyl,mtcars$vs),
+           FUN=mean, na.rm=TRUE)
  Group.1 Group.2        x
1       4       0 26.00000
2       6       0 20.56667
3       8       0 15.10000
4       4       1 26.73000
5       6       1 19.12500
```

Computing mean all attributes of cars aggregated by their number of cylinders and V/S:

```
> aggregate(mtcars, by=list(mtcars$cyl,mtcars$vs),
+           FUN=mean, na.rm=TRUE)
  Group.1 Group.2      mpg cyl   disp       hp     drat       wt     qsec vs        am
1       4       0 26.00000   4 120.30  91.0000 4.430000 2.140000 16.70000  0 1.0000000
2       6       0 20.56667   6 155.00 131.6667 3.806667 2.755000 16.32667  0 1.0000000
3       8       0 15.10000   8 353.10 209.2143 3.229286 3.999214 16.77214  0 0.1428571
4       4       1 26.73000   4 103.62  81.8000 4.035000 2.300300 19.38100  1 0.7000000
5       6       1 19.12500   6 204.55 115.2500 3.420000 3.388750 19.21500  1 0.0000000
      gear     carb
1 5.000000 2.000000
2 4.333333 4.666667
3 3.285714 3.500000
4 4.000000 1.500000
5 3.500000 2.500000
```

Aggregation using formula

```
> aggregate(mpg~cyl+vs, data=mtcars, FUN=mean)
  cyl vs      mpg
1   4  0 26.00000
2   6  0 20.56667
3   8  0 15.10000
4   4  1 26.73000
5   6  1 19.12500
```

11.2 Statistical Tests

In hypothesis testing, a p-value helps us determine the significance of the results. There is a null hypothesis and an alternate hypothesis. A hypothesis test uses p-value to weigh the strength of the evidence is support of alternative hypothesis.

- A small p-value (typically 0.05) indicates strong evidence against the null hypothesis, so you reject the null hypothesis.
- A large p-value (> 0.05) indicates weak evidence against the null hypothesis, so you fail to reject the null hypothesis.
- p-values very close to the cutoff (0.05) are considered to be marginal (could go either way). Always report the p-value so your readers can draw their own conclusions.

11.2.1 Shapiro-Wilk Normality Test

This test is applicable to check the normality of a univariate sample:

```
> shapiro.test(ToothGrowth$len)

        Shapiro-Wilk normality test

data:  ToothGrowth$len
W = 0.96743, p-value = 0.1091
```

The null-hypothesis here is that the distribution is normally distributed. If the p-value is > 0.05, we have weak evidence against the null hypothesis. Hence we infer that the distribution of the data is not significantly different from normal distribution.

Applying the test on uniform data:

```
> shapiro.test(runif(100))

        Shapiro-Wilk normality test

data:  runif(100)
W = 0.9649, p-value = 0.009121
```

Clearly the p-value is quite low, so we reject the contention that the data is normally distributed.

For exponentially distributed data:

```
> shapiro.test(rexp(100))

        Shapiro-Wilk normality test

data:  rexp(100)
W = 0.86374, p-value = 3.952e-08
```

Gamma distribution:

```
> shapiro.test(rgamma(100, shape=1))

        Shapiro-Wilk normality test

data:  rgamma(100, shape = 1)
W = 0.86356, p-value = 3.89e-08
```

Binomial distribution:

```
> shapiro.test(rbinom(100, size = 15, prob=0.1))

        Shapiro-Wilk normality test

data:  rbinom(100, size = 15, prob = 0.1)
W = 0.90717, p-value = 3.124e-06
```

11.2.2 Unpaired T-Test

For testing the equality of means

Let us generate some data from standard normal distribution:

```
> x <- rnorm(100)
```

Let us run the test to verify that the mean is indeed zero:

```
> t.test(x)

        One Sample t-test

data:  x
t = -0.75685, df = 99, p-value = 0.4509
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.2401063  0.1075127
sample estimates:
 mean of x
-0.0662968
```

Here the null hypothesis is that the mean is 0. The alternative hypothesis is that mean is not zero.

The p-value is 0.45 which is much larger than 0.05. We have very weak evidence against the null hypothesis.

Let's try again with a non-zero mean sample:

```
> x <- rnorm(100, mean=1)
> t.test(x)

        One Sample t-test

data:  x
t = 10.577, df = 99, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.871659 1.274210
sample estimates:
mean of x
 1.072934
```

11.2.3 Classical T-Test

For testing equality of means under the assumption of equal variances.

TODO

11.2.4 F-Test

For testing the equality of variances

TODO

11.2.5 Two Sample Wilcoxon Test

TODO

11.2.6 Kolmogorov-Smirnov Test

TODO

11.3 Maximum Likelihood Estimation

11.3.1 Estimating the Probability of Flipping Coins

We will generate a sequence of Bernoulli trials and attempt to estimate the probability of 1.

Our sample sequence of coin flips:

```
> x <- rbinom(100, size=1, prob=0.4)
```

The log likelihood function to be optimized:

```
> log_likelihood <- function(p, x) sum(dbinom(x, size=1, prob = p, log=T))
```

The maximum likelihood estimation procedure:

```
> optimize(log_likelihood, interval=c(0, 1), x=x, maximum=T)
$maximum
[1] 0.3599941

$objective
[1] -65.34182
```

We see significant deviation from the ideal value of 0.4. Let's try with another dataset:

```
> x <- rbinom(100, size=1, prob=0.4)
> optimize(log_likelihood, interval=c(0, 1), x=x, maximum=T)
$maximum
[1] 0.410003

$objective
[1] -67.68585
```

The likelihood estimate becomes stable as the number of samples increase:

```
> x <- rbinom(10000, size=1, prob=0.4)
> optimize(log_likelihood, interval=c(0, 1), x=x, maximum=T)
$maximum
[1] 0.3984011

$objective
[1] -6723.576

> x <- rbinom(10000, size=1, prob=0.4)
> optimize(log_likelihood, interval=c(0, 1), x=x, maximum=T)
$maximum
[1] 0.4080028

$objective
[1] -6761.223

> x <- rbinom(10000, size=1, prob=0.4)
```

```
> optimize(log_likelihood, interval=c(0, 1), x=x, maximum=T)
$maximum
[1] 0.398301

$objective
[1] -6723.164
```

11.3.2 Estimating the Mean of Normal Distribution

Let us draw some samples from a normal distribution with mean 1:

```
> x <- rnorm(100, mean=4)
```

Let us define our log-likelihood function:

```
> log_likelihood <- function(mu, x) sum(dnorm(x, mean=mu, log=T))
```

Let's find the maximum likelihood estimate of the mean:

```
> optimize(log_likelihood, interval=c(-100, 100), x=x, maximum=T)
$maximum
[1] 3.869241

$objective
[1] -139.3414
```

This is not too far from the actual value of 4.

Let's try again with more data:

```
> x <- rnorm(10000, mean=4)
> optimize(log_likelihood, interval=c(-100, 100), x=x, maximum=T)
$maximum
[1] 3.986473

$objective
[1] -14266.71
```

We get an estimate which is very close to 4.

CHAPTER 12

Graphics

R comes with a number of packages for creating graphics. A detailed overview of relevant packages is covered at CRAN.

12.1 Simple Charts

This section focuses on functionality provided by the graphics package.

- We build a graph by one main command.
- We follow it by a sequence of auxiliary commands to add more elements to the graphics.

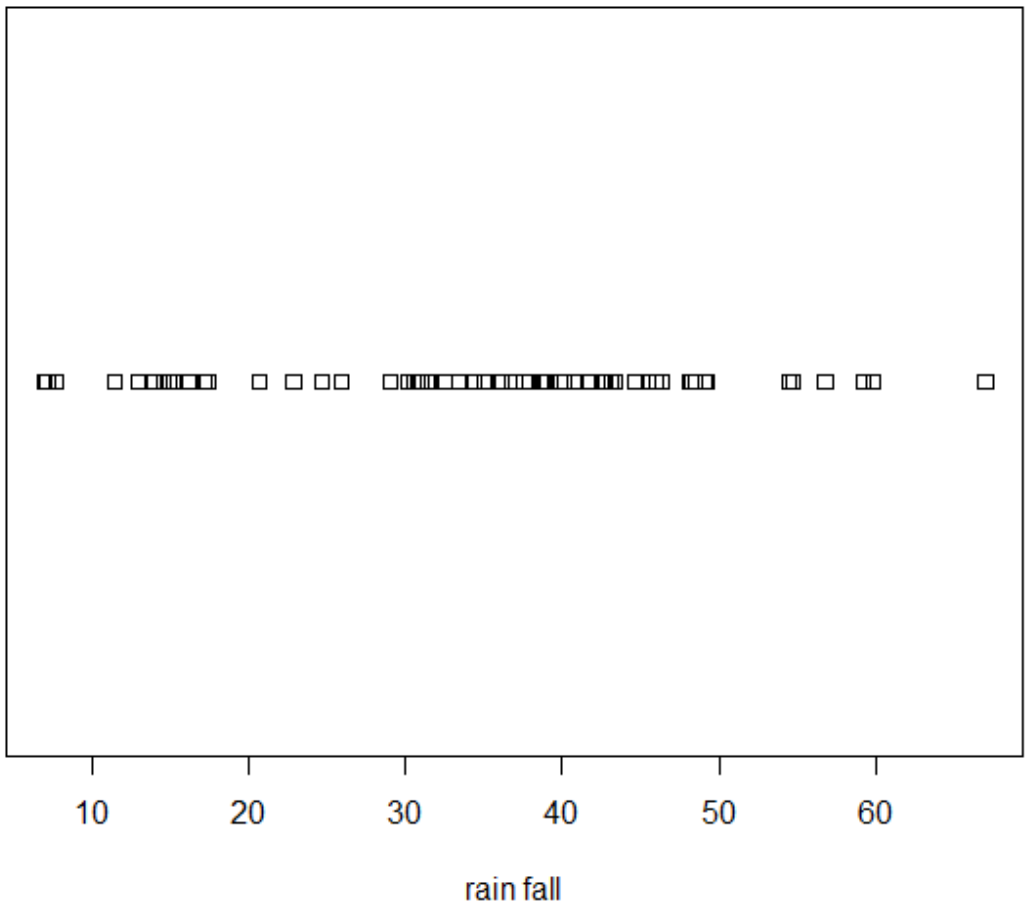
Plotting and charting functions support a set of commonly used parameters. They are listed here for quick reference. You will see them being used in rest of the section.

Parameter	Purpose
xlab	Label of X Axis
ylab	Label of Y Axis
xlim	Limits for the range of values in X Axis
ylim	Limits for the range of values in Y Axis
main	Main title for the plot

12.1.1 Strip Chart

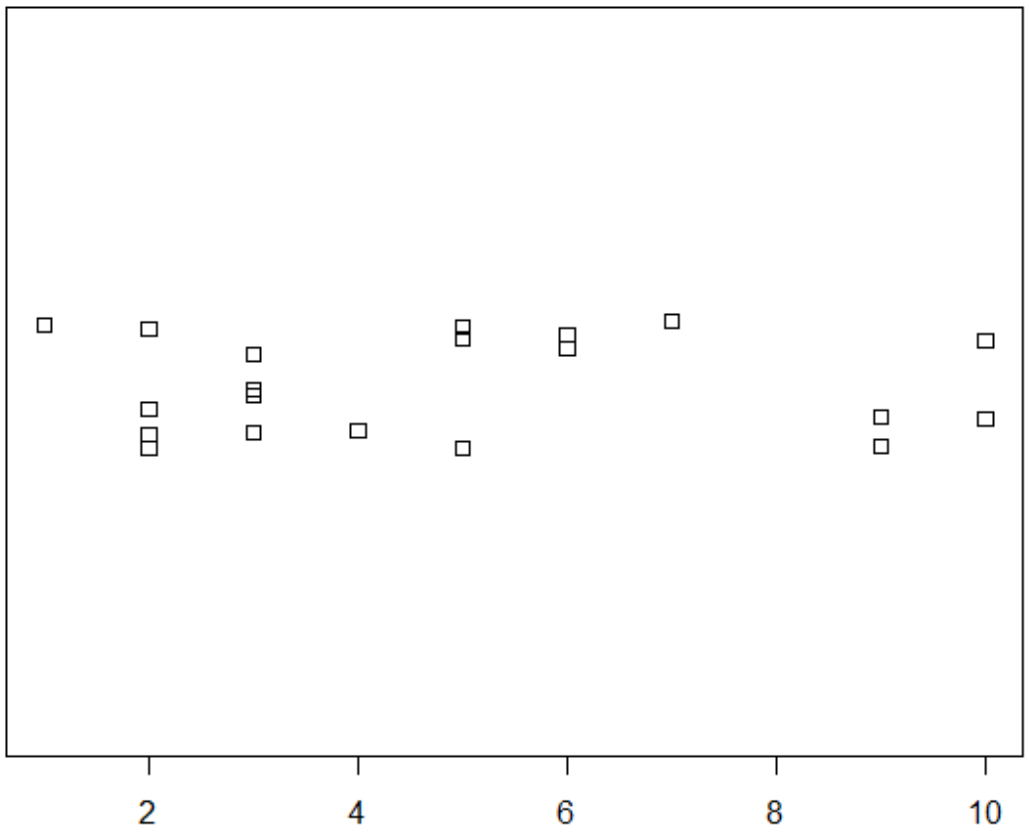
Over Plot Method

```
> stripchart(precip, xlab='rain fall')
```



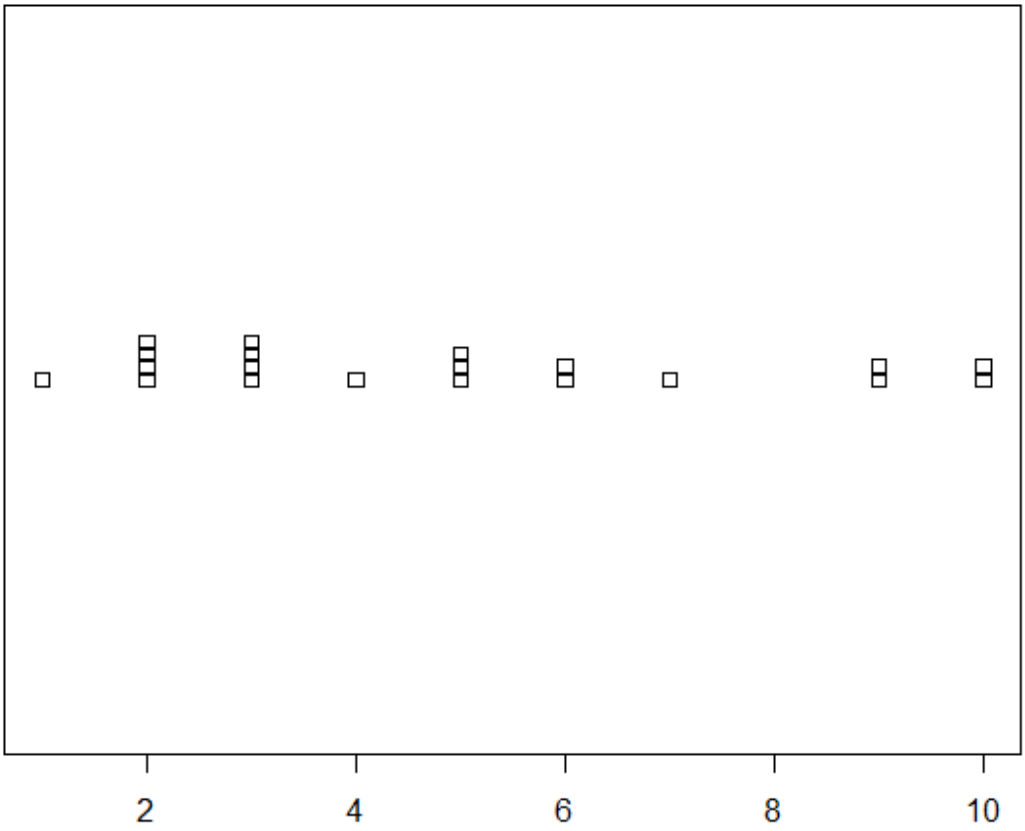
Jitter Method

```
> data <- sample(1:10, 20, replace=TRUE)
> data
 [1]  9  7 10  2  3  3  6  9  4  5  2  5 10  3  1  2  5  2  3  6
> stripchart(data, method='jitter')
> sort(data)
 [1]  1  2  2  2  2  3  3  3  3  4  5  5  5  6  6  7  9  9 10 10
```



Stack Method

```
> stripchart(data, method='stack')
```



12.1.2 Histograms

Plotting a histogram:

```
hist(rnorm(100))
hist(rnorm(100), col="red")
```

12.1.3 Stem and Leaf Plots

These are completely textual plots. A numeric vector is plotted as follows. From each number, the first and last digits are taken. First digit becomes the stem, last digit becomes the left. Stems go to the left of | and leaves go to the right of |.

:: > stem(c(10, 11, 21, 22, 23, 24, 25, 30, 31, 41,42,43,44,45,46,47, 60, 70))

The decimal point is 1 digit(s) to the right of the |

0 | 01 2 | 1234501 4 | 1234567 6 | 00

Scale parameter can be used to expand the plot:

```
> stem(c(10, 11, 21, 22, 23, 24, 25, 30, 31, 41,42,43,44,45,46,47, 60, 70), scale=2)

  The decimal point is 1 digit(s) to the right of the |

  1 | 01
  2 | 12345
  3 | 01
  4 | 1234567
  5 |
  6 | 0
  7 | 0
```

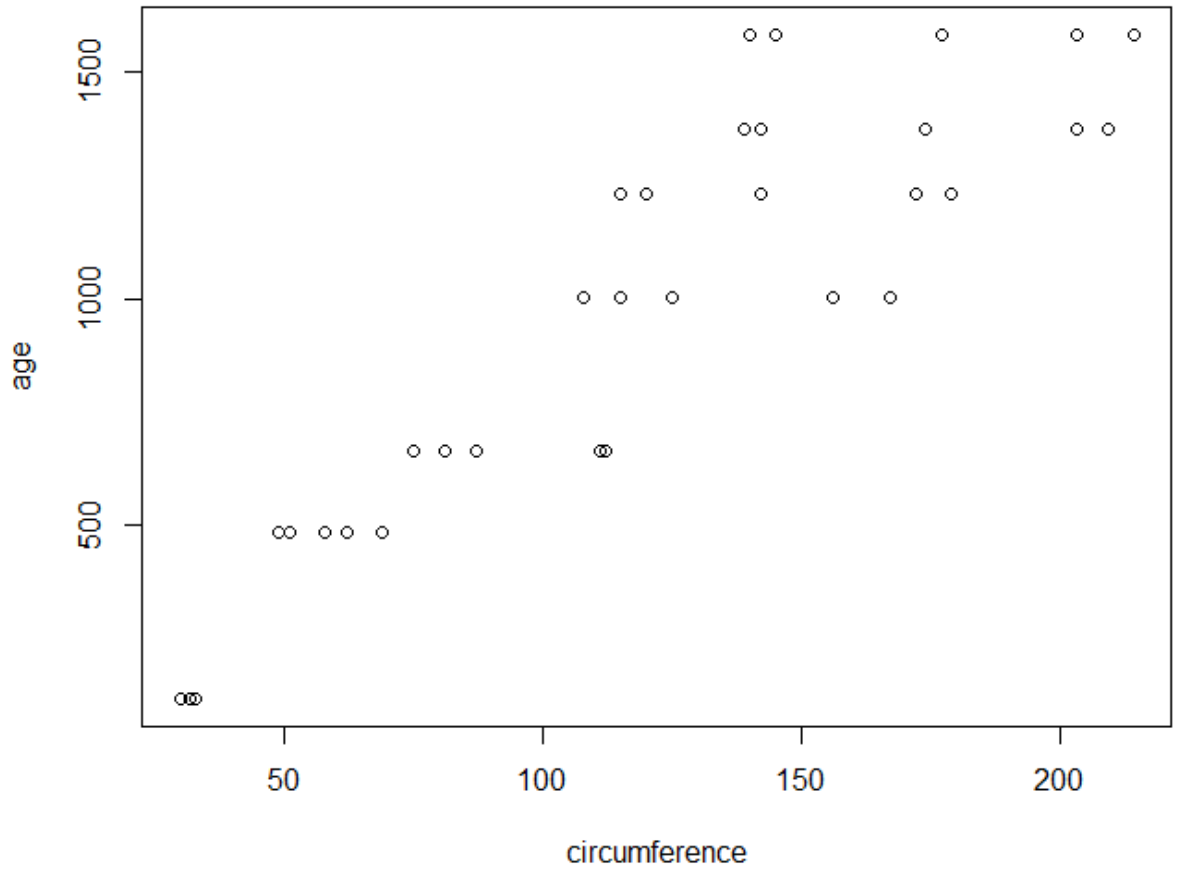
12.1.4 Bar Charts

12.1.5 Pie Charts

12.1.6 Scatter Plots

Plotting age vs circumference from the Orange dataset:

```
> plot(age~circumference, data=Orange)
```



Scatter Plot with Linear Model and Fitted Curve

We can overlay a linear model fit on top of our scatter plot.

Let us first create our scatter plot:

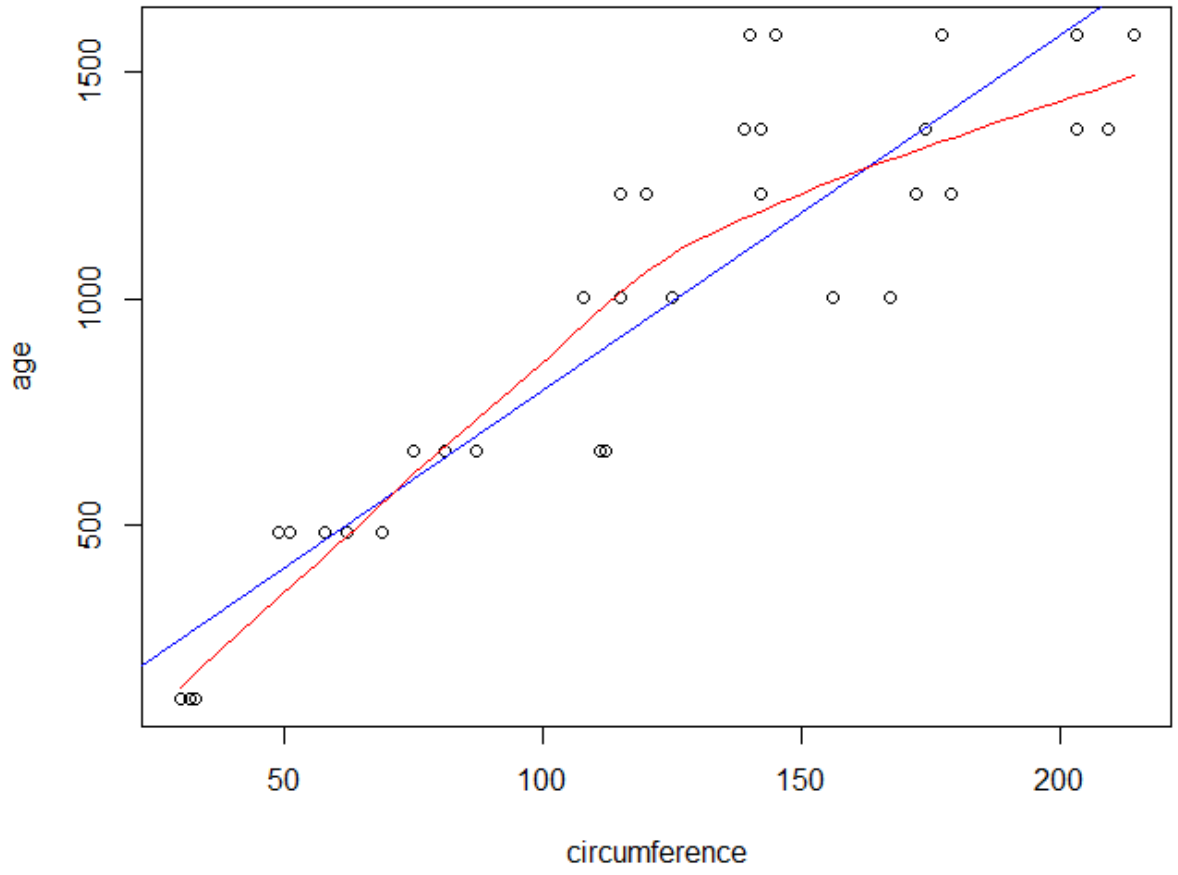
```
> plot(age~circumference, data=Orange)
```

Let us now create a linear model between age and circumference and plot the fitted model:

```
> abline(lm(age~circumference, dat=Orange), col='blue')
```

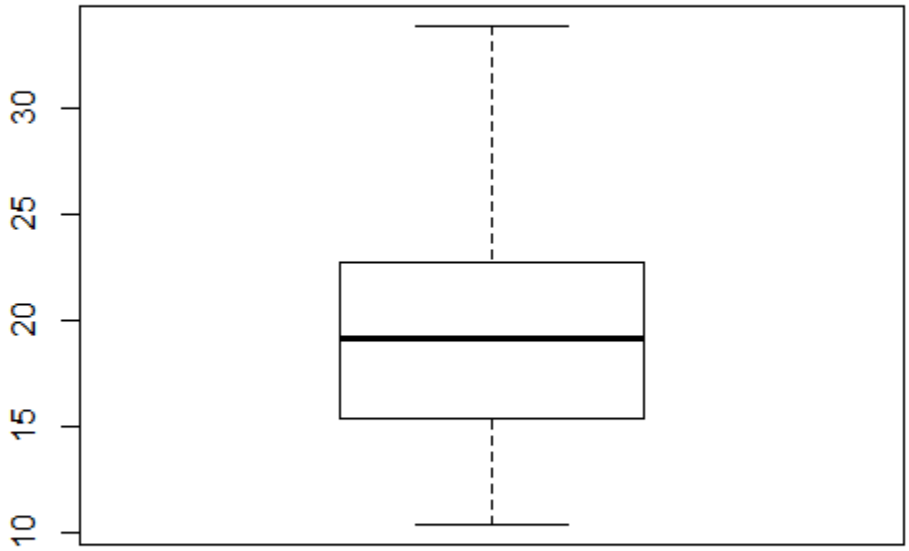
Finally, let us draw a smooth curve fitting the given data:

```
> lines(loess.smooth(Orange$circumference, Orange$age), col='red')
```



12.1.7 Box Plots

```
> boxplot(mtcars$mpg)
```



A box plot covers following statistics of the data:

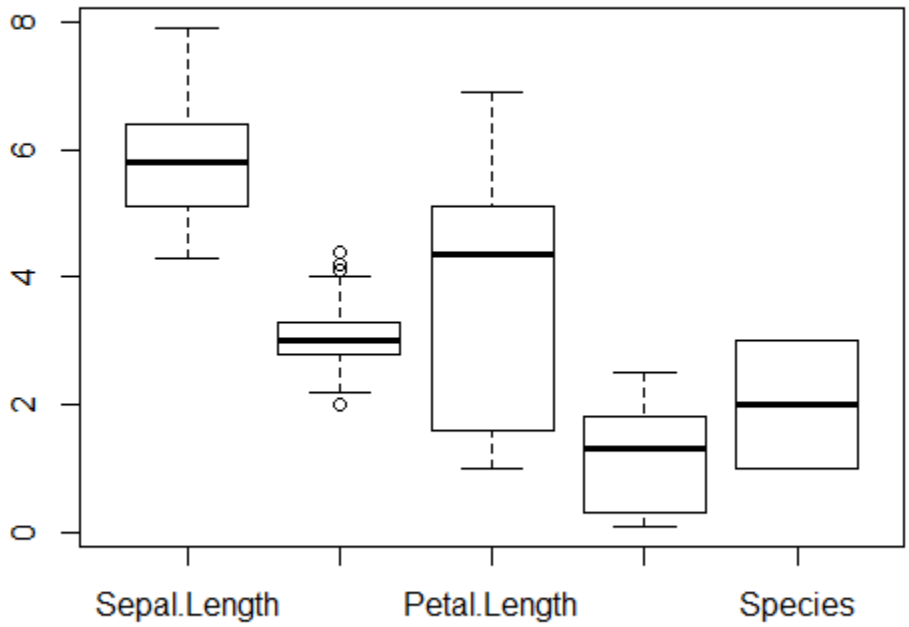
- Lower and Upper hinges making up the box
- Median making up the line in the middle of the box
- Whiskers extending from the box up to the maximum and minimum values in the data

The outliers in data are identified and drawn separately as circles beyond the maximum and minimum values [calculated after removing outliers].

- A longer whisker (in one direction) indicates skewness in that direction.

Plotting multiple variables from a data frame:

```
> boxplot(iris)
```



Outliers

- A *potential outlier* falls beyond 1.5 times the width of the box on either side.
- A *suspected outlier* falls beyond 3 times the width of the box on either side.
- Both are drawn as circle in the box plot in R.

Finding the list of outliers:

```
> boxplot.stats(precip)
$stats
   Phoenix  Milwaukee  Pittsburg Providence     Mobile
      11.5       29.1       36.6       42.8       59.8

$n
[1] 70

$conf
[1] 34.01281 39.18719

$out
     Mobile     Phoenix        Reno Albuquerque     El Paso
       67.0         7.0         7.2         7.8         7.8
```

The `$out` variable gives the list of outliers.

Finding the list of suspected outliers:

```
> boxplot.stats(rivers, coef=3)$out
[1] 2348 3710 2315 2533 1885
```

12.1.8 QQ Plots

12.1.9 Index Plots

Spikes

Points

Line Charts

A line chart:

```
x = rnorm(10); plot(x, type="l", col="blue")
```

12.1.10 More about Plot Function

Controlling the labels on x-axis

Let's prepare some data:

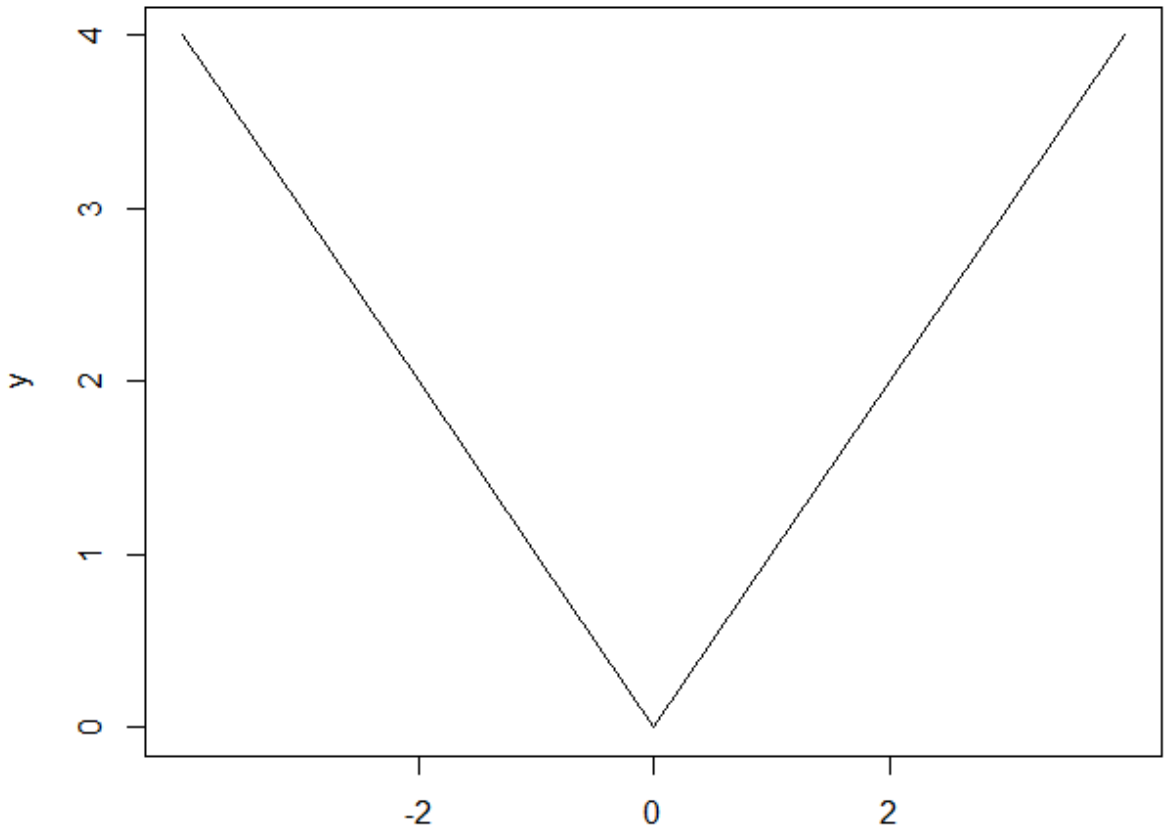
```
> x <- -4:4
> y <- abs(x)
```

Let's plot the `y` data without any labels on x-axis:

```
> plot(y, type='l', xaxt='n', xlab='')
```

Let's specify labels for specific values of `y`:

```
> axis(1, at=which(y==0), labels=c(0))
> axis(1, at=which(y==2), labels=c(-2,2))
```



Adding a Rug to a Plot

12.1.11 Exporting Plots

Some data to be plotted:

```
x <- rnorm(1000000)
y <- rnorm(1000000)
```

Preparing a PNG device attached to a file for plotting:

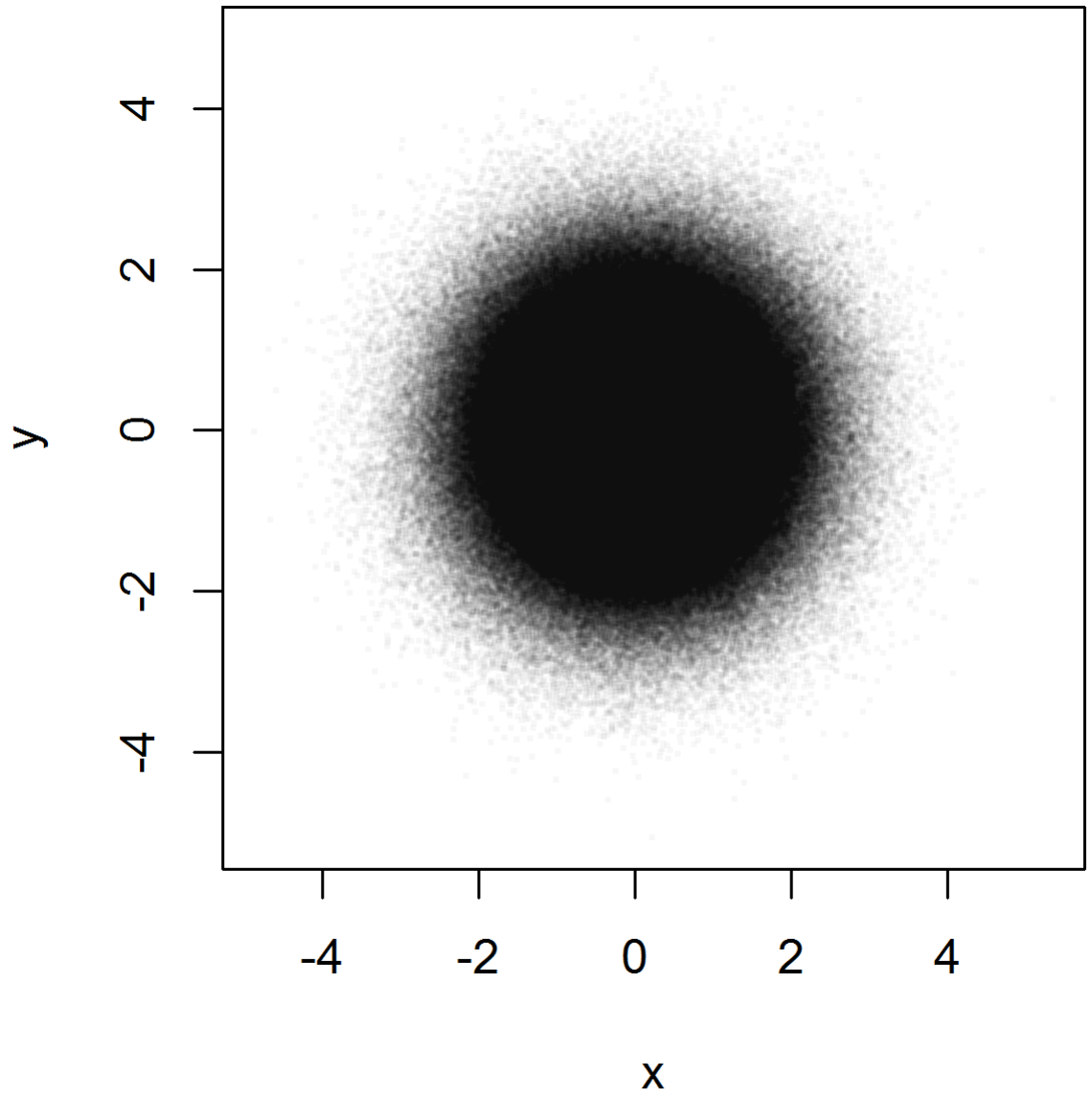
```
png("plot_export_demo.png", width=4, height=4, units="in", res=300)
par(mar=c(4,4,1,1))
```

Plotting the data:

```
plot(x,y,col=rgb(0,0,0,0.03), pch=".", cex=2)
```

Closing the device to finish the export:

```
> dev.off()
```



12.1.12 Heat Maps

12.2 GGPlot2

The grammar of graphics has following ideas:

- data set
- coordinate systems
- scales
- geometrical objects (points, lines, boxes, bars, etc.)

- aesthetics (size, color, transparency, etc.)
- facets

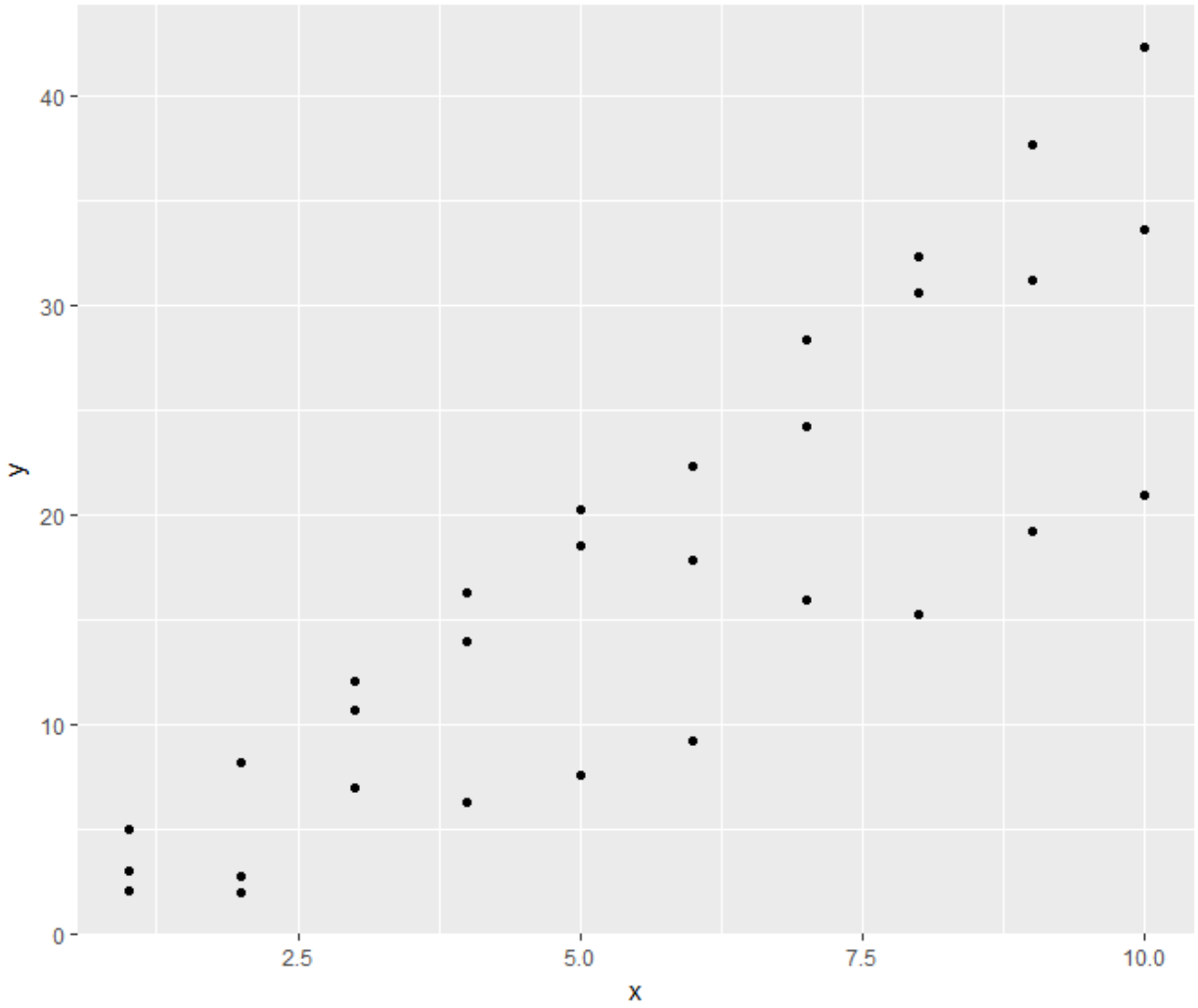
List of aesthetics

Aesthetic	Description	Details
color	color of points	
size	size of points	
shape	shape of points	6 shapes by default
fill	fill color for bars, etc.	
alpha	transparency	0 - totally transparent, 1 - totally opaque
stroke		
linetype		

We will prepare a data set consisting of points from three noisy lines:

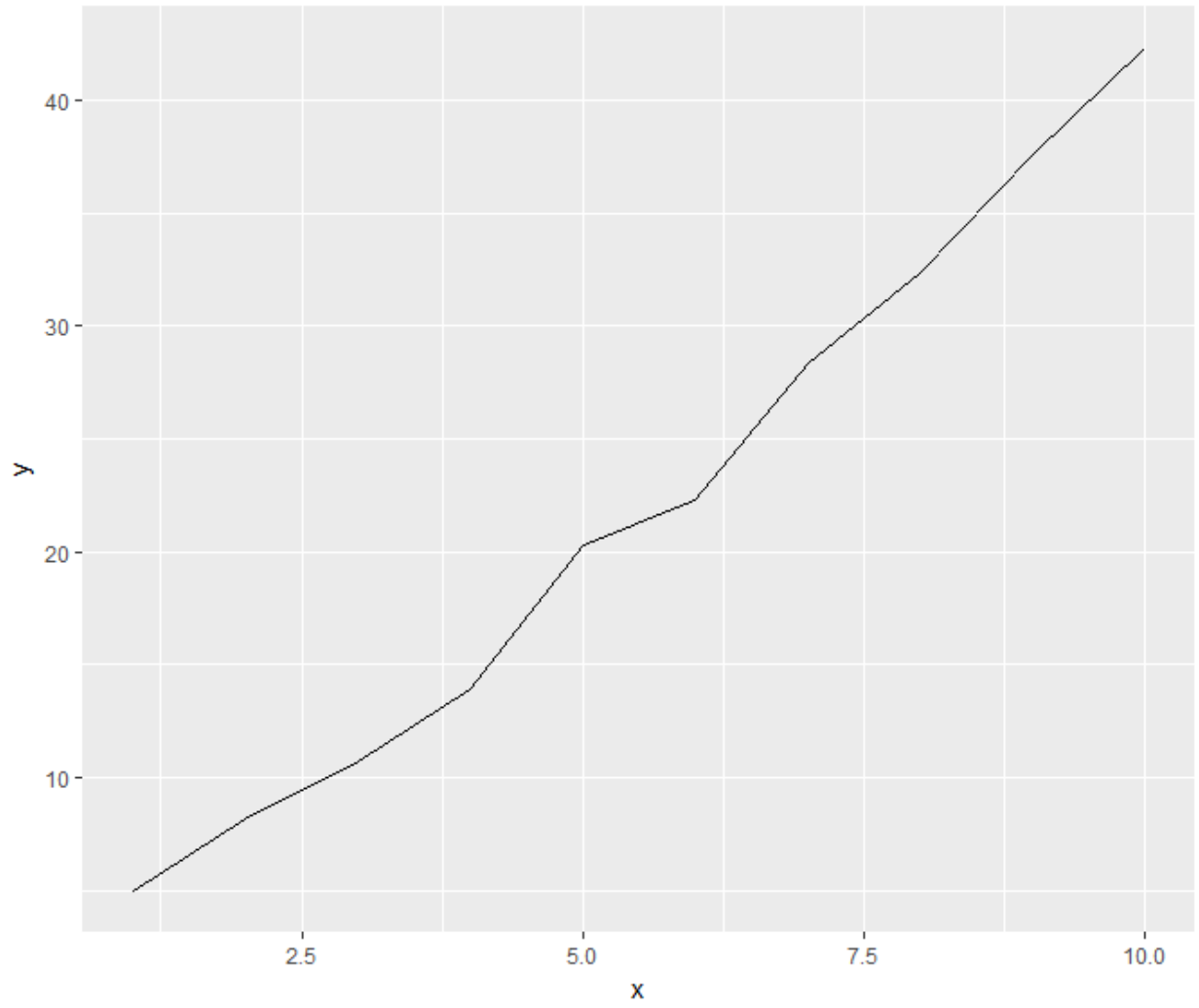
```
> x <- c(1:10)
> y1 <- 4 * x + 2*rnorm(10)
> y2 <- 3.5 * x + 2*rnorm(10)
> y3 <- 2 * x + 2*rnorm(10)
> y <- c(y1, y2, y3)
> x <- rep(x, 3)
> index <- rep(1:3, each=10)
> df <- data.frame(index=index, x=x, y=y)
> df
   index  x         y
1      1  1  4.970771
2      1  2  8.178185
3      1  3 10.698853
4      1  4 13.931963
5      1  5 20.264946
6      1  6 22.301773
7      1  7 28.370019
8      1  8 32.313593
9      1  9 37.628974
10     1 10 42.292327
11     2  1  3.052891
12     2  2  2.792227
13     2  3 12.092148
14     2  4 16.325021
15     2  5 18.509915
16     2  6 17.871619
17     2  7 24.220242
18     2  8 30.617569
19     2  9 31.180454
20     2 10 33.617135
21     3  1  2.024496
22     3  2  1.985061
23     3  3  6.995560
24     3  4  6.292036
25     3  5  7.542600
26     3  6  9.227137
27     3  7 15.913043
28     3  8 15.263772
29     3  9 19.253074
30     3 10 20.927419
```

```
> qplot(x, y, data=df)
```

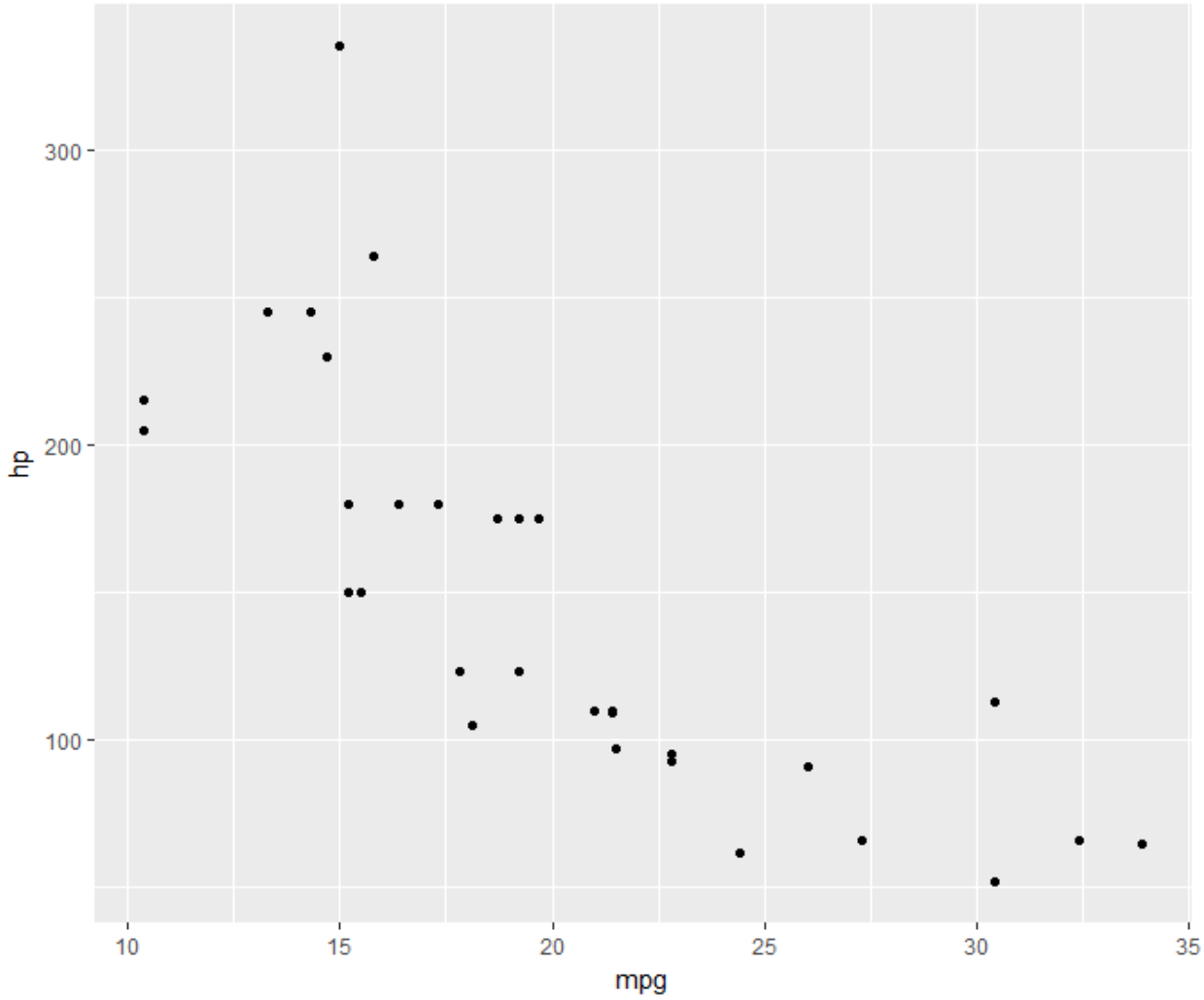


Plotting data from line 1 as a line:

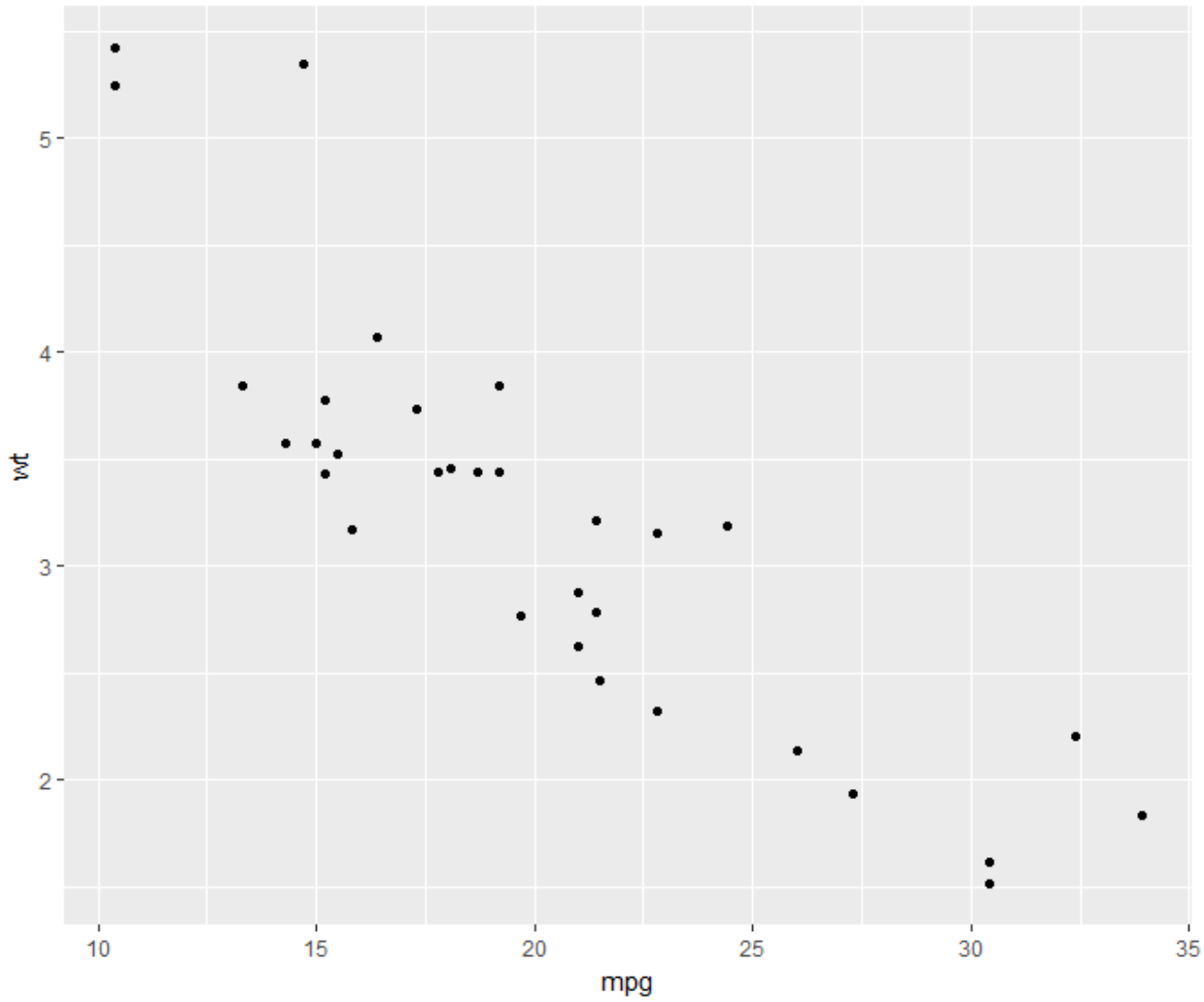
```
> df[df$index==1, ]
  index  x         y
1      1  1  4.970771
2      1  2  8.178185
3      1  3 10.698853
4      1  4 13.931963
5      1  5 20.264946
6      1  6 22.301773
7      1  7 28.370019
8      1  8 32.313593
9      1  9 37.628974
> qplot(x, y, data=df[df$index == 1, ], geom='line')
```




```
> qplot(mpg, hp, data=mtcars)
```




```
> qplot(mpg, wt, data=mtcars)
```



..:

```
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
```

```
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
.. image:: images/plt.png
```

12.3 Special Charts

12.3.1 Missing Data Map

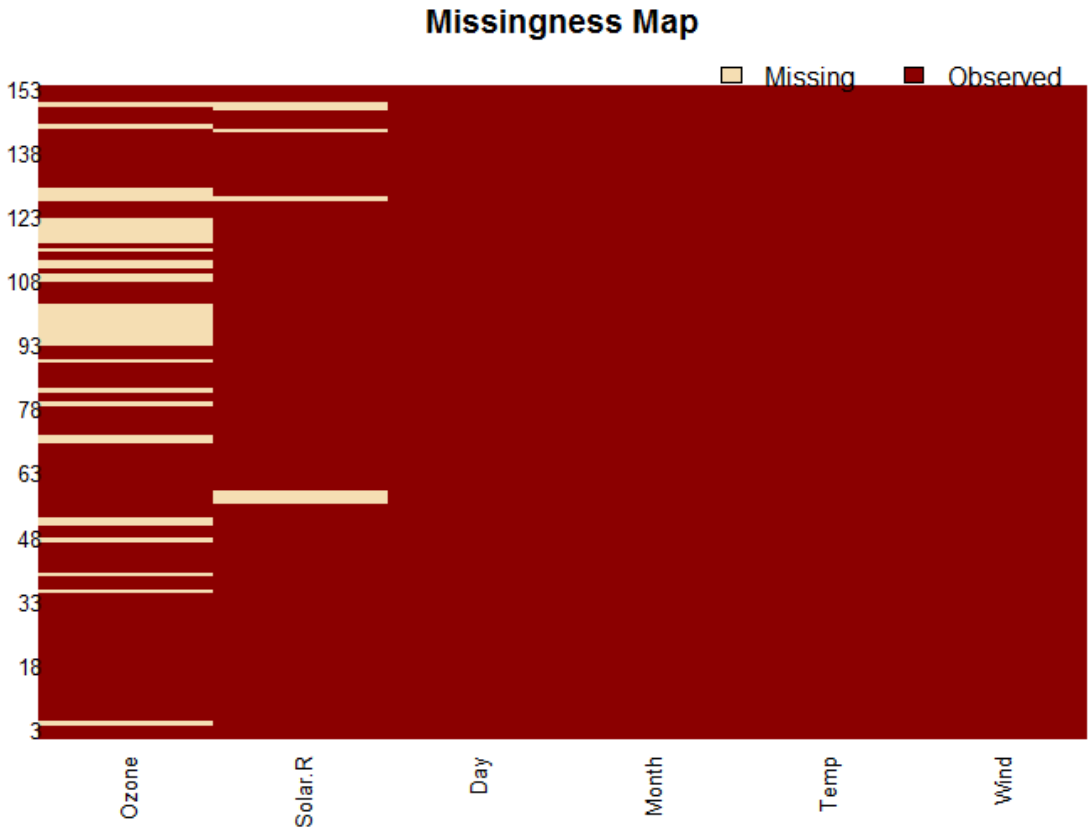
While doing exploratory data analysis, it is important to identify variables which have a lot of missing data.

Let's see the variation of missing data in air quality dataset:

```
> sapply(airquality, function(x){sum(is.na(x))})
  Ozone Solar.R    Wind    Temp   Month     Day
     37       7       0       0       0       0
```

Amelia library provides a function to visualize this:

```
> require(Amelia)
> mismap(airquality)
```



12.4 Parameters

It is possible to ask user to hit return before drawing the next graphics on device.

To switch on the asking:

```
> par(ask=T)
```

To switch off asking:

```
> par(ask=F)
```

When we change a parameter, it is good to store the old value somewhere and restore it later:

```
> oldpar <- par(ask=T)
> oldpar
$ask
[1] FALSE

> par(oldpar)
```

Parameter	Purpose
ask	Ask user before plotting next graphics on device

CHAPTER 13

Optimization

13.1 Minimization, Maximization

The R function `optimize` can be used for both minimization and maximization. We start with discussing univariate functions.

13.1.1 Minimization

A univariate function $f(x)$ is to be minimized over an interval $[a, b]$. The value of the function $f(x)$ at a particular value of x is called the objective. The minimization procedure will identify a point $c \in [a, b]$ such that $f(c)$ is lower than or equal to all other values of $f(x) : x \in [a, b]$. The minimization procedure will return:

- The objective value at the point of minimization
- The value of x at which the minimum is achieved.

Minimizing a function over a given interval:

```
> f <- function (x) { x^ 2}
> f(2)
[1] 4
> f(-2)
[1] 4
> optimize(f, c(-2, 2))
$minimum
[1] -5.551115e-17

$objective
[1] 3.081488e-33

> f <- function (x) { (x-1)^ 2}
> optimize(f, c(-2, 2))
$minimum
[1] 1
```

```
$objective
[1] 0
```

In the second case, $f(x)$ is symmetrical around $x = 1$. It is possible to make this parameterized for $x = a$:

```
> f <- function (x, a) { (x-a)^ 2}
> optimize(f, c(-10, 10), a=3)
$minimum
[1] 3

$objective
[1] 0

> optimize(f, c(-10, 10), a=4)
$minimum
[1] 4

$objective
[1] 0

> optimize(f, c(-10, 10), a=5)
$minimum
[1] 5

$objective
[1] 0
```

13.1.2 Maximization

Our function to be maximized:

```
> f <- function(x, a) {sin(x-a)}
```

Maximization procedure:

```
> optimize(f, c(0, pi), a=pi/4, maximum = T)
$maximum
[1] 2.35619

$objective
[1] 1

> pi/4 + pi/2
[1] 2.356194
```

13.2 Least Squares

The least squares problem $y = Ax + e$.

Computing the least squares (without an intercept term):

```
> A <- matrix(c(1, 2, -1, 3, -6, 9, -1, 2, 1), nrow=3)
> x <- c(1,2,3)
```

```
> y <- A %*% x
> lsfit(A, y, intercept=FALSE)
$coefficients
X1 X2 X3
 1  2  3

$residuals
[1] 0 0 0

$intercept
[1] FALSE

$qr
$qt
[1]  9.797959 16.970563  6.928203

$qr
             X1         X2         X3
[1,] -2.4494897  7.3484692 -0.8164966
[2,]  0.8164966  8.4852814  0.0000000
[3,] -0.4082483 -0.9120956  2.3094011

$qraux
[1] 1.408248 1.409978 2.309401

$rank
[1] 3

$pivot
[1] 1 2 3

$tol
[1] 1e-07

attr(,"class")
[1] "qr"
```

CHAPTER 14

Statistical Models

14.1 Linear Models

14.1.1 Cooked-up Examples

We start with a simple linear model $y = 2x$. We will generate some test samples containing pairs of (x,y) values. Our goal would be to train a simple linear model $y = wx$ with the test data and estimate the value of w.

Let us prepare some test data:

```
> x <- 1:10
> y <- 2*x
```

Let us fit a model `y~x` meaning `y` is linearly dependent on `x` (with a possible bias term) as

$$y = wx + c$$

The goal of fitting the model is to compute the weight coefficient `w` and the intercept or bias or offset term `c`:

```
> lmfit <- lm(y~x)
```

We can print the model:

```
> lmfit

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
  2.247e-15    2.000e+00
```

The coefficients can be extracted separately:

```
> coefficients(lmfit)
 (Intercept)            x
2.246933e-15 2.000000e+00
```

The fitted y values for every value of x:

```
> fitted.values(lmfit)
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 10 12 14 16 18 20
```

This looks too good to be true. Let's add an intercept term and also introduce some noise:

```
> n <- length(x)
> e <- rnorm(n, sd = 0.1)
> y <- 2*x + 3 + e
```

Fitting the linear mode:

```
> lmfit <- lm(y~x)
> lmfit

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      2.933        2.008
```

The intercept value is very close to 3 and the weight is also pretty close to 2.

Let's compare the fitted values with actual values of y:

```
> fitted.values(lmfit)
        1         2         3         4         5         6         7         8
↪  9
 4.941002  6.949160  8.957318 10.965476 12.973634 14.981792 16.989951 18.998109 21.
↪006267
       10
23.014425
> y
 [1]  4.941495  6.895725  9.110955 10.874365 12.980755 14.998704 16.899220 19.066500
 [9] 20.932681 23.076735
```

We can also see the residual values:

```
> residuals(lmfit)
           1            2            3            4            5            6
 0.0004929328 -0.0534350207  0.1536365592 -0.0911108850  0.0071203406  0.0169111935
           7            8            9           10
-0.0907301344  0.0683915230 -0.0735859540  0.0623094450
```

Let's look at more complicated models.

A polynomial in x:

```
> y <- 3 + 4*x + 5 *x^2 + rnorm(n, sd=0.1)
> lmfit <- lm(y~1+x+I(x^2))
> lmfit
```

```
Call:
lm(formula = y ~ 1 + x + I(x^2))

Coefficients:
(Intercept)            x       I(x^2)
      2.945        3.965        5.005
```

Multiple sinusoidal variables:

```
> x <- sin(pi *seq(0, 2, by=0.05))
> y <- cos(pi *seq(0, 2, by=0.05))
> n <- length(x)
> z <- 2 + 3*x + 4*y + rnorm(n, sd=0.1)
> lmfit <- lm(z~x+y)
> lmfit

Call:
lm(formula = z ~ x + y)

Coefficients:
(Intercept)            x            y
      2.014        2.982        4.015
```

With just 41 samples, the estimate is pretty good.

Fitting without the intercept term:

```
> z <- 2 + 3*x + 4*y + rnorm(n, sd=0.1)
> lm(z~0+x+y)

Call:
lm(formula = z ~ 0 + x + y)

Coefficients:
    x      y
2.982  4.111
```

We note that the original formula had an intercept term. This had an undesired effect on the estimate of the weights of x and y.

Let us consider an example where the construction of z doesn't have an intercept term:

```
> z <- 3*x + 4*y + rnorm(n, sd=0.1)
> lmfit <- lm(z~0+x+y)
> lmfit

Call:
lm(formula = z ~ 0 + x + y)

Coefficients:
    x      y
3.003  3.984
> coefficients(lmfit)
       x        y
3.003388 3.983586
```

We see that the weights of x and y are calculated correctly.

Fitting on a data frame

Let's explore the relationship between the mpg and disp variables in the mtcars dataset.

Attaching the dataset:

```
> attach(mtcars)
```

Exploring a polynomial dependency:

```
> lmfit <- lm(disp~1+mpg+I(mpg^2))
> lmfit

Call:
lm(formula = disp ~ 1 + mpg + I(mpg^2))

Coefficients:
(Intercept)          mpg     I(mpg^2)
   941.2143     -53.0598       0.8101
```

Measuring the quality of estimation as ratio between residuals and fitted values:

```
> mean(abs(residuals(lmfit) / fitted.values(lmfit) ))
[1] 0.1825118
```

14.1.2 Cars Data Set

`cars` dataset is shipped with R distribution. It captures the speed of cars vs the distance taken to stop the car. This data was recorded in 1920s. The data set consists of 50 records:

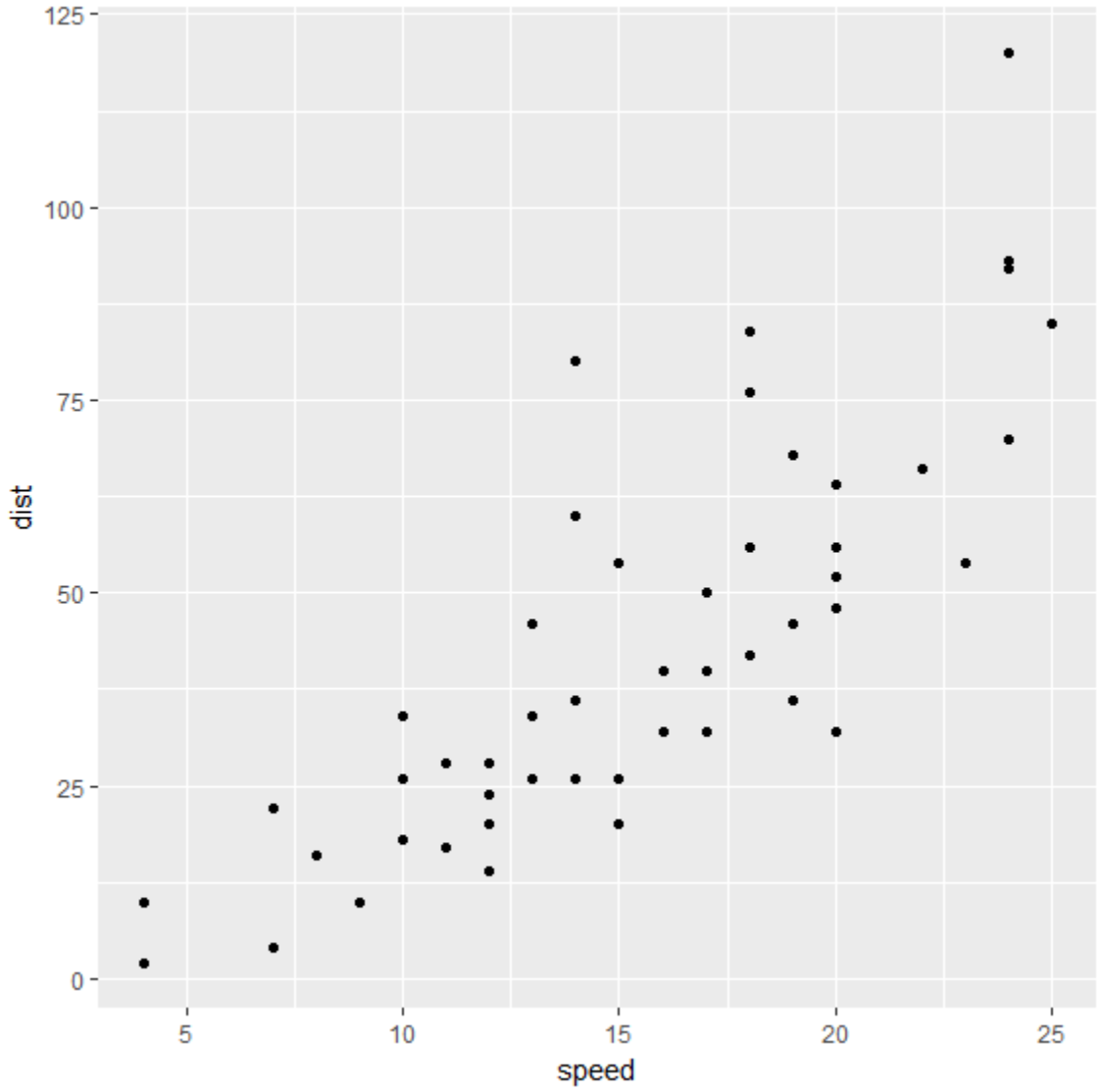
```
> dim(cars)
[1] 50  2
```

Summary statistics:

```
> summary(cars)
     speed           dist
 Min.   : 4.0   Min.   :  2.00
 1st Qu.:12.0   1st Qu.: 26.00
 Median :15.0   Median : 36.00
 Mean   :15.4   Mean   : 42.98
 3rd Qu.:19.0   3rd Qu.: 56.00
 Max.   :25.0   Max.   :120.00
```

Let us create a scatter plot of speed vs distance data:

```
> qplot(speed, dist, data=cars)
```



There appears to be a good correlation between speed and distance and the relationship appears to be quite linear. Let us verify the correlation between the two variables too:

```
> cor(cars)
          speed      dist
speed 1.0000000 0.8068949
dist  0.8068949 1.0000000
```

Let us fit a linear model `dist = a + b speed` to this dataset:

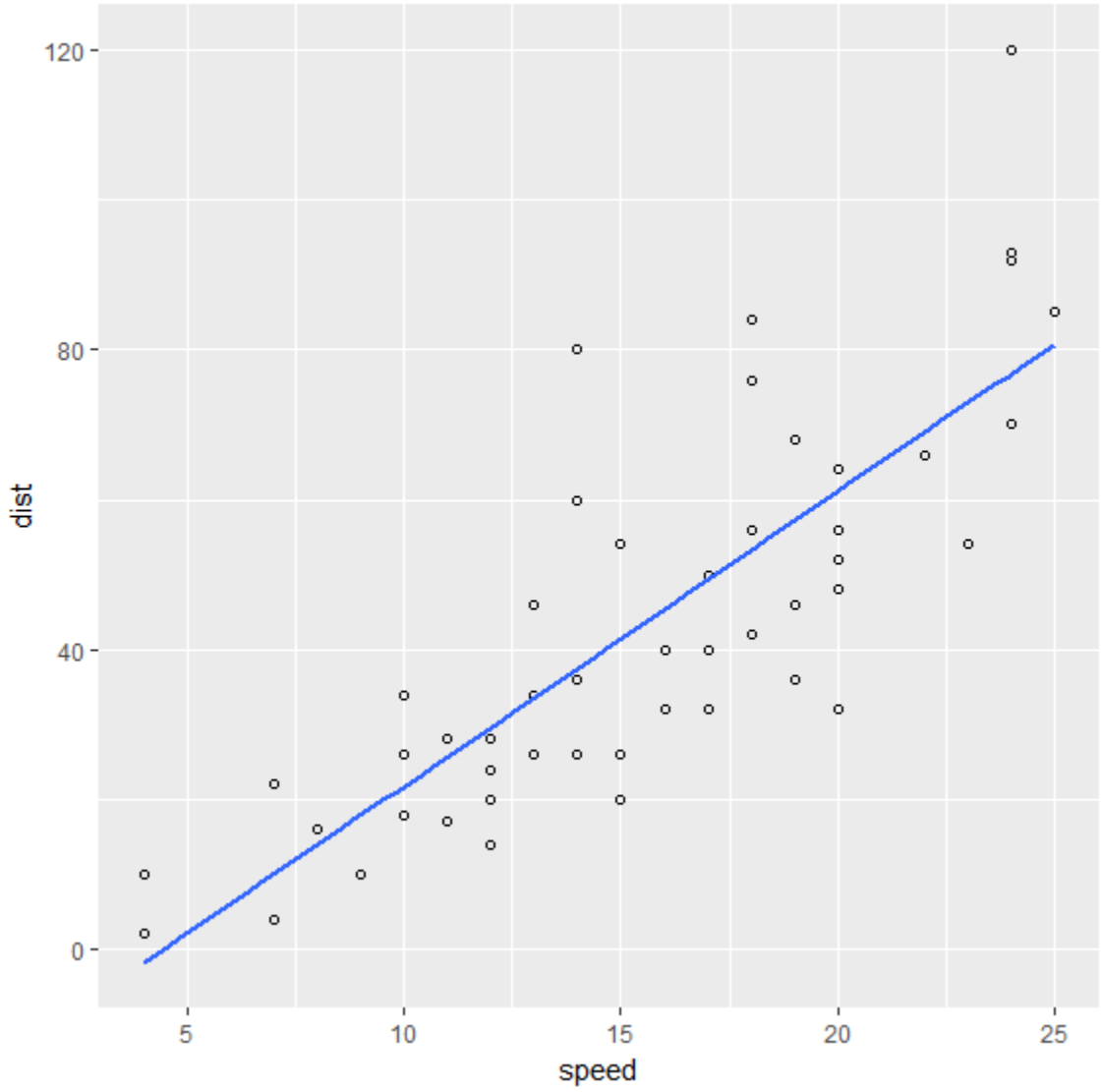
```
> fit <- lm(dist~speed, data=cars)
> fit

Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
(Intercept)        speed
    -17.579        3.932
```

The fitted values for distance are:

```
> round(fitted.values(fit), 2)
    1     2     3     4     5     6     7     8     9    10    11    12    13    14
↪ 15    16    17    18    19
-1.85 -1.85  9.95  9.95 13.88 17.81 21.74 21.74 21.74 25.68 25.68 29.61 29.61 29.61
↪29.61 33.54 33.54 33.54 33.54
   20    21    22    23    24    25    26    27    28    29    30    31    32    33
↪ 34    35    36    37    38
37.47 37.47 37.47 37.47 41.41 41.41 41.41 45.34 45.34 49.27 49.27 49.27 53.20 53.20
↪53.20 53.20 57.14 57.14 57.14
   39    40    41    42    43    44    45    46    47    48    49    50
61.07 61.07 61.07 61.07 61.07 68.93 72.87 76.80 76.80 76.80 76.80 80.73
```

We can plot the fitted values as follows:

```
> ggplot(data=cars, mapping=aes(x=speed, y=dist)) + geom_point(shape=1) + geom_
↪smooth(method='lm', se=FALSE)
```



Diagnostics

We can print the summary statistics for the linear model as follows:

```
> fit.summary <- summary(fit); fit.summary

Call:
lm(formula = dist ~ speed, data = cars)

Residuals:
    Min      1Q  Median      3Q     Max
-29.069  -9.525  -2.272   9.215  43.201

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -17.5791     6.7584  -2.601   0.0123 *
speed         3.9324     0.4155   9.464 1.49e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 15.38 on 48 degrees of freedom
Multiple R-squared:  0.6511,    Adjusted R-squared:  0.6438
F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Several important statistics can be separated out.

R Squared:

```
> fit.summary$r.squared
[1] 0.6510794
```

Adjusted R Squared:

```
> fit.summary$adj.r.squared
[1] 0.6438102
```

F Statistic:

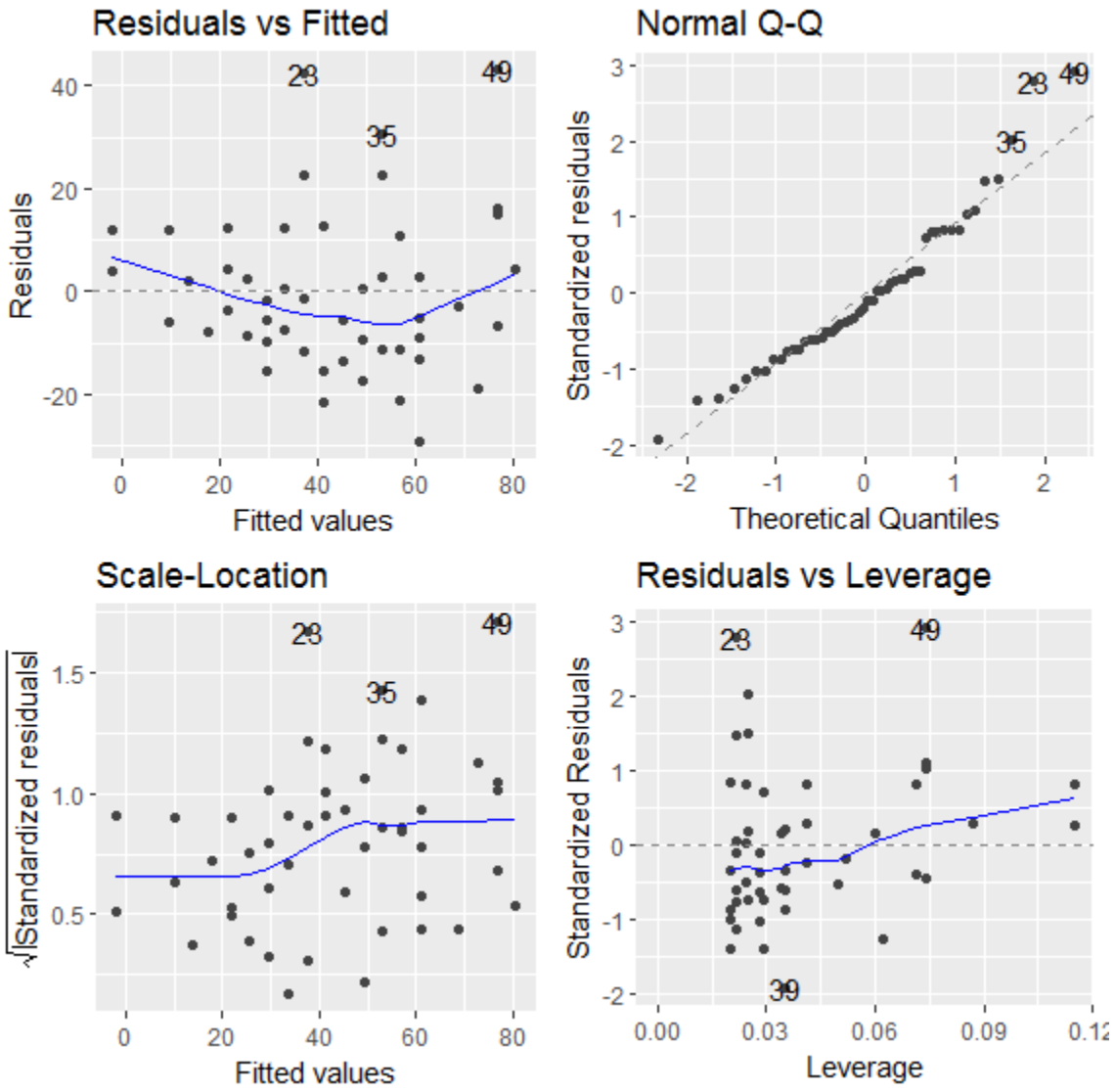
```
> fit.summary$fstatistic
   value    numdf    dendf
89.56711  1.00000 48.00000
```

The `ggfortify` library provides functions to let ggplot interpret linear models:

```
> install.packages("ggfortify")
> library(ggfortify)
```

We are now ready to plot the diagnostics for this model:

```
> autoplot(fit)
```



14.1.3 Cats Data Set

This data set is available in MASS package. The data set records body weight and heart weight for 144 cats along with their sex.

Loading the dataset:

```
> data(cats, package='MASS')
```

Variables in the data set:

```
> str(cats)
'data.frame':   144 obs. of  3 variables:
 $ Sex: Factor w/ 2 levels "F","M": 1 1 1 1 1 1 1 1 1 1 ...
```

```
 $ Bwt: num  2 2 2 2.1 2.1 2.1 2.1 2.1 2.1 2.1 ...
 $ Hwt: num  7 7.4 9.5 7.2 7.3 7.6 8.1 8.2 8.3 8.5 ...
```

Few rows from data set:

```
> head(cats)
  Sex Bwt Hwt
1   F 2.0 7.0
2   F 2.0 7.4
3   F 2.0 9.5
4   F 2.1 7.2
5   F 2.1 7.3
6   F 2.1 7.6
```

Summary statistics:

```
> summary(cats)
 Sex          Bwt             Hwt
 F:47   Min.   :2.000   Min.   : 6.30
 M:97   1st Qu.:2.300   1st Qu.: 8.95
        Median :2.700   Median :10.10
        Mean   :2.724   Mean   :10.63
        3rd Qu.:3.025   3rd Qu.:12.12
        Max.   :3.900   Max.   :20.50
```

Predicting the heart weight from body weight:

```
> fit <- lm(Hwt~Bwt, data=cats)
> fit

Call:
lm(formula = Hwt ~ Bwt, data = cats)

Coefficients:
(Intercept)          Bwt
    -0.3567       4.0341
```

Summary statistics for the fitted model:

```
> summary(fit)

Call:
lm(formula = Hwt ~ Bwt, data = cats)

Residuals:
    Min      1Q  Median      3Q     Max
-3.5694 -0.9634 -0.0921  1.0426  5.1238

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.3567     0.6923  -0.515    0.607
Bwt           4.0341     0.2503  16.119   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.452 on 142 degrees of freedom
Multiple R-squared:  0.6466,    Adjusted R-squared:  0.6441
F-statistic: 259.8 on 1 and 142 DF,  p-value: < 2.2e-16
```

The t-value for intercept term is small. This is indicative that if body weight is 0, then heart weight will also be 0.

Let us check if including the sex of the cat provides any help in improving the model:

```
> fit2 <- lm(Hwt~Bwt+Sex, data=cats)
> fit2

Call:
lm(formula = Hwt ~ Bwt + Sex, data = cats)

Coefficients:
(Intercept)          Bwt         SexM
    -0.4150       4.0758      -0.0821

> summary(fit2)

Call:
lm(formula = Hwt ~ Bwt + Sex, data = cats)

Residuals:
    Min      1Q  Median      3Q     Max
-3.5833 -0.9700 -0.0948  1.0432  5.1016

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.4149     0.7273  -0.571    0.569
Bwt           4.0758     0.2948  13.826   <2e-16 ***
SexM         -0.0821     0.3040  -0.270    0.788
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.457 on 141 degrees of freedom
Multiple R-squared:  0.6468,    Adjusted R-squared:  0.6418
F-statistic: 129.1 on 2 and 141 DF,  p-value: < 2.2e-16
```

We note that the residual standard error has increased and the F statistic has decreased. We see that t-value for the sex parameter is very small. It doesn't add value to the model.

CHAPTER 15

Clustering

15.1 K-Means Clustering

15.1.1 Iris Data Set

We will work with the iris dataset in this section. Our goal is to automatically cluster the measurements in the data set so that measurements from the same species fall into the same cluster. The species variable in the data set works as the ground truth. We will use the other variables to perform clustering.

Let us estimate the correlation between different variables and the species variable:

```
> cor(iris$Sepal.Length, as.integer(iris$Species), method='spearman')
[1] 0.7980781
> cor(iris$Sepal.Width, as.integer(iris$Species), method='spearman')
[1] -0.4402896
> cor(iris$Petal.Length, as.integer(iris$Species), method='spearman')
[1] 0.9354305
> cor(iris$Petal.Width, as.integer(iris$Species), method='spearman')
[1] 0.9381792
```

The correlations suggest that petal width and petal length are very strongly correlated with the species of the flower. We will use these variables for the clustering purpose.

Let us prepare the subset data frame on which clustering will be applied

```
> iris2 <- iris[, c("Petal.Length", "Petal.Width")]
```

We are ready to run the k-means clustering algorithm now:

```
> num_clusters <- 3
> set.seed(1111)
> result <- kmeans(iris2, num_clusters, nstart=20)
```

We can see the centers for the 3 clusters as follows:

```
> result$centers
  Petal.Length Petal.Width
1     5.595833    2.037500
2     1.462000    0.246000
3     4.269231    1.342308
```

We can see the assignment of cluster to individual measurements as follows:

```
> result$cluster
  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2␣
↪2 2 2 2
 [45] 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 3 3 3 3 3 1␣
↪3 3 3 3
 [89] 3 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 3 1␣
↪1 1 1 1
[133] 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1 1 1 1
```

Let us see how well the clustering has happened:

```
> table(result$cluster, iris$Species)

    setosa versicolor virginica
  1      0          2        46
  2     50          0         0
  3      0         48         4
```

6 measurements have been mis-clustered.

Within cluster sum of squares by cluster:

```
> result$withinss
[1] 16.29167  2.02200 13.05769
```

At times, it is useful to first center and scale each variable before clustering:

```
> iris3 <- scale(iris2)
> result <- kmeans(iris3, num_clusters, nstart=20)
> table(result$cluster, iris$Species)

    setosa versicolor virginica
  1      0         48         4
  2     50          0         0
  3      0          2        46
>
```

CHAPTER 16

Utilities

16.1 Date Time

Current date and time:

```
> date()
[1] "Tue Oct 31 12:43:00 2017"
```

This is a character object.

Current date:

```
> Sys.Date()
[1] "2017-10-31"
```

This is a Date object.

Current time:

```
> Sys.time()
[1] "2017-10-31 12:44:11 IST"
```

Current time is a POSIXct object.

Following options are available in R for date time manipulation:

- Built-in `as.Date` function supports dates (but no time)
- `chron` library supports date and time but no time-zones
- `POSIXct` and `POSIXlt` classes provide support for time zone too.

Using `asDate` function:

```
> as.Date('1982-03-18')
[1] "1982-03-18"
> as.Date('1982/03/18')
```

```
[1] "1982-03-18"
> d <- as.Date('1982/03/18')
> typeof(d)
[1] "double"
> class(d)
[1] "Date"
> mode(d)
[1] "numeric"
```

The default format is YYYY-MM-DD where the separator can be changed.

Specifying a custom format:

```
> as.Date('18/03/1982', format='%d/%m/%Y')
[1] "1982-03-18"
```

The codes for the date format are listed in the table below.

Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated)
%B	Month (full name)
%y	Year (2 digit)
%Y	Year (4 digit)

Number of dates since Jan 1, 1970:

```
> as.numeric(as.Date('1982/3/18'))
[1] 4459
```

Date arithmetic:

```
> d <- as.Date('1982/03/18')
> d
[1] "1982-03-18"
> d + 1
[1] "1982-03-19"
> d + 30
[1] "1982-04-17"
> d - 1
[1] "1982-03-17"
> d - 30
[1] "1982-02-16"
```

Preparing a list of dates:

```
> d + seq(1, 10, by=3)
[1] "1982-03-19" "1982-03-22" "1982-03-25" "1982-03-28"
> d2 <- d + seq(1, 10, by=3)
```

Identifying corresponding weekdays:

```
> weekdays(d2)
[1] "Friday"   "Monday"   "Thursday" "Sunday"
```

Corresponding months:

```
> months(d2)
[1] "March" "March" "March" "March"
```

Corresponding quarters:

```
> quarters(d2)
[1] "Q1" "Q1" "Q1" "Q1"
```

CHAPTER 17

Object Oriented Programming

R provides multiple approaches to object oriented programming. In this chapter, we cover S3 and S4 classes. This chapter may be skipped on initial reading.

17.1 S3 Classes

S3 system builds up on two concepts:

- A class attribute attached to an object
- Class specific implementation for generic methods

In the following, we provide an implementation of the generic method `print` for an object of class `mylist`:

```
print.mylist <- function(lst, ...){
  for (name in names(lst)){
      cat(name); cat(': '); cat(lst[[name]]); cat(' ')
  }
  cat('\n')
}
```

Now let us create a list object:

```
> l <- list(a=1, b=2, c=1:4)
```

Standard printing of list objects:

```
> l
$a
[1] 1

$b
[1] 2
```

```
$c
[1] 1 2 3 4
```

Let us change the class of `l` now:

```
> class(l) <- 'mylist'
```

Print `l` will now pick the new implementation:

```
> print(l)
a: 1 b: 2 c: 1 2 3 4
> l
a: 1 b: 2 c: 1 2 3 4
```

17.1.1 A Modified Gram Schmidt Algorithm Implementation

The function below implements the Modified Gram Schmidt. While the algorithm implementation is straightforward and not important for the discussion in this section, look at the returned object. The function is returning a list object whose class has been set to `mgs`:

```
mgs <- function(X){

  # Ensure that X is a matrix
  X <- as.matrix(X)
  # Number of rows
  m <- nrow(X)
  # Number of columns
  n <- ncol(X)
  if (m < n) {
    stop('Wide matrices are not supported.')
  }
  # Construct the empty Q and R matrices
  Q <- matrix(0, m, n)
  R <- matrix(0, n, n)
  for (j in 1:n){
    # Pick up the j-th column of X
    v <- X[, j]
    # compute the projection of v on existing columns
    if (j > 1){
      for (i in 1:(j-1)){
        # pick up the i-th Q vector
        q <- Q[, i]
        #cat('q: '); print(q)
        # compute projection of v on qi
        projection <- as.vector(q %*% v)
        #cat('projection: '); print(projection)
        # Store the projection in R
        R[i, j] <- projection
        # subtract the projection from v
        v <- v - projection * q
      }
    }
    # cat('v: ') ; print(v)
    # Compute the norm of the remaining vector
    v.norm <- sqrt(sum(v^2))
    # cat('v-norm: '); print(v.norm)
```

```
    # Store the norm in R
    R[j,j] <- v.norm
    # Place the normalized vector in Q
    Q[, j] <- v / v.norm
  }
  # Prepare the result
  result <- list(Q=Q, R=R, call=match.call())
  # Set the class of the result
  class(result) <- 'mgs'
  result
}
```

We can now provide implementations of generic methods for the objects returned by this object. For example, let us implement print for objects of type `mgs`:

```
print.mgs <- function(mgs){
  cat("Call: \n")
  print(mgs$call)
  cat('Q: \n')
  print(mgs$Q)
  cat('R: \n')
  print(mgs$R)
}
```

Let us compute the QR decomposition of a matrix using this algorithm:

```
> A <- matrix(c(3, 2, -1, 2, -2, .5, -1, 4, -1), nrow=3)
> res <- mgs(A)
```

When we print the result object, `print.mgs` will be called:

```
> res
Call:
mgs(X = A)
Q:
           [,1]       [,2]      [,3]
[1,]  0.8017837  0.5901803 0.0939682
[2,]  0.5345225 -0.7785358 0.3288887
[3,] -0.2672612  0.2134695 0.9396820
R:
         [,1]      [,2]       [,3]
[1,] 3.741657 0.4008919  1.6035675
[2,] 0.000000 2.8441670 -3.9177929
[3,] 0.000000 0.0000000  0.2819046
```

We can implement other (non-generic) methods for this class too:

```
mgs.q <- function(mgs){
  mgs$Q
}

mgs.r <- function(mgs){
  mgs$R
}

mgs.x <- function(mgs){
  mgs$Q %*% mgs$R
}
```

Note that these methods don't verify that the object being passed is of class `mgs`.

17.2 S4 Classes

CHAPTER 18

R Packages

18.1 Standard Packages

The standard R distribution comes with following packages.

Package	Description
base	Base R functions
compiler	R byte code compiler
corrplot	For plotting correlation matrix nicely
datasets	Base R datasets
dlm	Maximum likelihood and Bayesian dynamic linear models
fftw	Fast Fourier Transform
grDevices	Graphics devices for base and grid graphics
graphics	R functions for base graphics
grid	A rewrite of the graphics layout capabilities, plus some support for interaction
methods	Formally defined methods and classes for R objects, plus other programming tools, as described in the Green Book
parallel	Support for parallel computation, including by forking and by sockets, and random-number generation
rattle	R Analytic Tool To Learn Easily (Rattle) provides a Gnome (RGtk2) based interface to R functionality for data science
splines	Regression spline functions and classes
signal	Signal Processing [ported from Octave]
stats	R statistical functions
stats4	Statistical functions using S4 classes
tcltk	Interface and language bindings to Tcl/Tk GUI elements
tools	Tools for package development and administration
utils	R utility functions
waved	Deconvolution of noisy signals using wavelet methods
wavelets	Wavelet filters, transforms, multi-resolution analysis
waveslim	1, 2, 3 dimensional signal processing with wavelets, book: introduction to wavelets and other filtering methods in finance and economics
wavethresh	Book: Wavelet methods in statistics with R

18.2 Contributed Packages

This section lists some of the contributed packages which are useful in day to day R development work.

Package	Description	Remarks
boot		
DescTools		
foreach		
FSA		
iterators	Provides iterator functionality	
plyr		
psych		
rattle		Requires GTK+
RColorBrewer		
Rmisc		
rpart	Recursive partitioning and regression trees	
rpart.plot		
randomForest		
party	Conditional inference trees	
ggpubr	publication ready plots based on ggplot2	

CHAPTER 19

Data Sets

19.1 Standard Data Sets

R ships with a set of built-in datasets.

Viewing the list of datasets:

```
> data()
```

Loading a dataset:

```
> data(Orange)
```

Description of the dataset:

```
> ?Orange
```

Lets assign a generic name to this dataset:

```
> ds <- Orange
```

Seeing the first few rows of the dataset:

```
> head(ds)
  Tree  age circumference
1    1  118            30
2    1  484            58
3    1  664            87
4    1 1004           115
5    1 1231           120
6    1 1372           142
```

Seeing the last few rows:

```
> tail(ds)
   Tree  age circumference
30    5  484            49
31    5  664            81
32    5 1004           125
33    5 1231           142
34    5 1372           174
35    5 1582           177
```

Number of records:

```
> nrow(ds)
[1] 35
```

Sampling some random rows from the dataset:

```
> rows <- sample(nrow(ds), 10)
> rows <- sort(rows)
> ds[rows,]
   Tree  age circumference
2     1  484            58
3     1  664            87
10    2  664           111
13    2 1372           203
14    2 1582           203
22    4  118            32
23    4  484            62
27    4 1372           209
29    5  118            30
31    5  664            81
```

`dplyr` library provides similar functionality to sample records:

```
> dplyr::sample_n(ds, 10)
   Tree  age circumference
24    4  664           112
21    3 1582           140
18    3 1004           108
2     1  484            58
5     1 1231           120
30    5  484            49
19    3 1231           115
33    5 1231           142
15    3  118            30
25    4 1004           167
```

Finding the column names (variable names) in the dataset:

```
> names(ds)
[1] "Tree"          "age"           "circumference"
```

Computing averages:

```
> mean(ds$age)
[1] 922.1429
> mean(ds$circumference)
[1] 115.8571
```

Computing variances:

```
> var(ds$age)
[1] 241930.7
> var(ds$circumference)
[1] 3304.891
```

Summary of the dataset:

```
> summary(ds)
 Tree       age         circumference
 3:7   Min.   : 118.0   Min.   : 30.0
 1:7   1st Qu.: 484.0   1st Qu.: 65.5
 5:7   Median :1004.0   Median :115.0
 2:7   Mean   : 922.1   Mean   :115.9
 4:7   3rd Qu.:1372.0   3rd Qu.:161.5
       Max.   :1582.0   Max.   :214.0
```

Finding class of each variable in the dataset:

```
> sapply(Orange, class)
$Tree
[1] "ordered" "factor"

$age
[1] "numeric"

$circumference
[1] "numeric"
```

This works better for other datasets like iris, mtcars:

```
> sapply(iris, class)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
   "numeric"    "numeric"    "numeric"    "numeric"     "factor"

> sapply(mtcars, class)
      mpg       cyl      disp        hp      drat        wt      qsec        vs
↪ am
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
↪"numeric"
     gear      carb
"numeric" "numeric"
```

19.1.1 mtcars Data Set

Loading:

```
> data("mtcars")
```

Basic info:

```
> nrow(mtcars)
[1] 32
> ncol(mtcars)
[1] 11
> head(mtcars)
```

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

Summary:

```
> summary(mtcars)
      mpg              cyl             disp             hp             drat           
↪ wt
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0   Min.   :2.760   Min.
↪   :1.513
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080   1st
↪Qu.:2.581
 Median :19.20   Median :6.000   Median :196.3   Median :123.0   Median :3.695
↪Median :3.325
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7   Mean   :3.597   Mean
↪   :3.217
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920   3rd
↪Qu.:3.610
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0   Max.   :4.930   Max.
↪   :5.424
      qsec             vs              am              gear            carb      
 Min.   :14.50   Min.   :0.0000   Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:16.89   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :17.71   Median :0.0000   Median :0.0000   Median :4.000   Median :2.000
 Mean   :17.85   Mean   :0.4375   Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:18.90   3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :22.90   Max.   :1.0000   Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Scaling the variables:

```
> mtcars.scaled <- scale(mtcars)
```

Computing the covariance matrix:

```
> mtcars.scaled.cov <- cov(mtcars.scaled)
```

Identifying variable pairs with significant covariance:

```
> abs(mtcars.scaled.cov) > 0.8
       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
mpg   TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
cyl   TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
disp  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
hp   FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
drat FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
wt    TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
qsec FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
vs   FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
am   FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
gear FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
carb FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

Relationship between miles per gallon and displacement:

```
> ggplot(mtcars) + geom_point(mapping=aes(x=mpg, y=disp))
```

19.1.2 iris Data Set

Loading:

```
> data("iris")
```

Basic info:

```
> nrow(iris)
[1] 150
> ncol(iris)
[1] 5
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Summary:

```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

Unique species:

```
> table(iris$Species)

    setosa versicolor  virginica
        50         50         50
```

19.1.3 ToothGrowth Data Set

Loading:

```
> data("ToothGrowth")
```

Basic info:

```
> nrow(ToothGrowth)
[1] 60
> ncol(ToothGrowth)
[1] 3
> head(ToothGrowth)
   len supp dose
1  4.2   VC  0.5
2 11.5   VC  0.5
3  7.3   VC  0.5
4  5.8   VC  0.5
5  6.4   VC  0.5
6 10.0   VC  0.5
```

Summary:

```
> summary(ToothGrowth)
      len        supp         dose
 Min.   : 4.20   OJ:30   Min.   :0.500
 1st Qu.:13.07   VC:30   1st Qu.:0.500
 Median :19.25           Median :1.000
 Mean   :18.81           Mean   :1.167
 3rd Qu.:25.27           3rd Qu.:2.000
 Max.   :33.90           Max.   :2.000
```

19.1.4 PlantGrowth Data Set

Loading:

```
> data("PlantGrowth")
```

Basic info:

```
> nrow(PlantGrowth)
[1] 30
> ncol(PlantGrowth)
[1] 2
> head(PlantGrowth)
  weight group
1   4.17  ctrl
2   5.58  ctrl
3   5.18  ctrl
4   6.11  ctrl
5   4.50  ctrl
6   4.61  ctrl
```

Summary:

```
> summary(PlantGrowth)
     weight       group
 Min.   :3.590   ctrl:10
 1st Qu.:4.550   trt1:10
 Median :5.155   trt2:10
 Mean   :5.073
 3rd Qu.:5.530
 Max.   :6.310
```

19.1.5 USArrests Data Set

Loading:

```
> data('USArrests')
```

Basic info:

```
> nrow(USArrests)
[1] 50
> ncol(USArrests)
[1] 4
> head(USArrests)
           Murder Assault UrbanPop Rape
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
Colorado      7.9     204       78 38.7
```

Summary:

```
> summary(USArrests)
     Murder          Assault         UrbanPop          Rape
 Min.   : 0.800   Min.   : 45.0   Min.   :32.00   Min.   : 7.30
 1st Qu.: 4.075   1st Qu.:109.0   1st Qu.:54.50   1st Qu.:15.07
 Median : 7.250   Median :159.0   Median :66.00   Median :20.10
 Mean   : 7.788   Mean   :170.8   Mean   :65.54   Mean   :21.23
 3rd Qu.:11.250   3rd Qu.:249.0   3rd Qu.:77.75   3rd Qu.:26.18
 Max.   :17.400   Max.   :337.0   Max.   :91.00   Max.   :46.00
```

19.1.6 Titanic dataset

It is a 4-dimensional array resulting from cross-tabulating 2201 observations on 4 variables.

The 4 dimensions are:

1. Class: 1st, 2nd, 3rd, Crew
2. Sex: Male Female
3. Age: Child Adult
4. Survived: No, Yes

```
> dim(Titanic)
[1] 4 2 2 2
```

19.1.7 Datasets in the datasets Package

Dataset	Description
AirPassengers	Monthly Airline Passenger Numbers 1949-1960
BJsales	Sales Data with Leading Indicator

Continued on next page

Table 19.1 – continued from previous page

Dataset	Description
BJsales.lead (BJsales)	Sales Data with Leading Indicator
BOD	Biochemical Oxygen Demand
CO2	Carbon Dioxide Uptake in Grass Plants
ChickWeight	Weight versus age of chicks on different diets
DNase	Elisa assay of DNase
EuStockMarkets	Daily Closing Prices of Major European Stock Indices, 1991-1998
Formaldehyde	Determination of Formaldehyde
HairEyeColor	Hair and Eye Color of Statistics Students
Harman23.cor	Harman Example 2.3
Harman74.cor	Harman Example 7.4
Indometh	Pharmacokinetics of Indomethacin
InsectSprays	Effectiveness of Insect Sprays
JohnsonJohnson	Quarterly Earnings per Johnson & Johnson Share
LakeHuron	Level of Lake Huron 1875-1972
LifeCycleSavings	Intercountry Life-Cycle Savings Data
Loblolly	Growth of Loblolly pine trees
Nile	Flow of the River Nile
Orange	Growth of Orange Trees
OrchardSprays	Potency of Orchard Sprays
PlantGrowth	Results from an Experiment on Plant Growth
Puromycin	Reaction Velocity of an Enzymatic Reaction
Seatbelts	Road Casualties in Great Britain 1969-84
Theoph	Pharmacokinetics of Theophylline
Titanic	Survival of passengers on the Titanic
ToothGrowth	The Effect of Vitamin C on Tooth Growth in Guinea Pigs
UCBAdmissions	Student Admissions at UC Berkeley
UKDriverDeaths	Road Casualties in Great Britain 1969-84
UKgas	UK Quarterly Gas Consumption
USAccDeaths	Accidental Deaths in the US 1973-1978
USArrests	Violent Crime Rates by US State
USJudgeRatings	Lawyers' Ratings of State Judges in the US Superior Court
USPersonalExpenditure	Personal Expenditure Data
UScitiesD	Distances Between European Cities and Between US Cities
VADeaths	Death Rates in Virginia (1940)
WWWusage	Internet Usage per Minute
WorldPhones	The World's Telephones
ability.cov	Ability and Intelligence Tests
airmiles	Passenger Miles on Commercial US Airlines, 1937-1960
airquality	New York Air Quality Measurements
anscombe	Anscombe's Quartet of 'Identical' Simple Linear Regressions
attenu	The Joyner-Boore Attenuation Data
attitude	The Chatterjee-Price Attitude Data
austres	Quarterly Time Series of the Number of Australian Residents
beaver1 (beavers)	Body Temperature Series of Two Beavers
beaver2 (beavers)	Body Temperature Series of Two Beavers
cars	Speed and Stopping Distances of Cars
chickwts	Chicken Weights by Feed Type
co2	Mauna Loa Atmospheric CO2 Concentration
crimtab	Student's 3000 Criminals Data

Continued on next page

Table 19.1 – continued from previous page

Dataset	Description
discoveries	Yearly Numbers of Important Discoveries
esoph	Smoking, Alcohol and (O)esophageal Cancer
euro	Conversion Rates of Euro Currencies
euro.cross (euro)	Conversion Rates of Euro Currencies
eurodist	Distances Between European Cities and Between US Cities
faithful	Old Faithful Geyser Data
fdeaths (UKLungDeaths)	Monthly Deaths from Lung Diseases in the UK
freeny	Freeny's Revenue Data
freeny.x (freeny)	Freeny's Revenue Data
freeny.y (freeny)	Freeny's Revenue Data
infert	Infertility after Spontaneous and Induced Abortion
iris	Edgar Anderson's Iris Data
iris3	Edgar Anderson's Iris Data
islands	Areas of the World's Major Landmasses
ldeaths (UKLungDeaths)	Monthly Deaths from Lung Diseases in the UK
lh	Luteinizing Hormone in Blood Samples
longley	Longley's Economic Regression Data
lynx	Annual Canadian Lynx trappings 1821-1934
mdeaths (UKLungDeaths)	Monthly Deaths from Lung Diseases in the UK
morley	Michelson Speed of Light Data
mtcars	Motor Trend Car Road Tests
nhtemp	Average Yearly Temperatures in New Haven
nottem	Average Monthly Temperatures at Nottingham, 1920-1939
npk	Classical N, P, K Factorial Experiment
occupationalStatus	Occupational Status of Fathers and their Sons
precip	Annual Precipitation in US Cities
presidents	Quarterly Approval Ratings of US Presidents
pressure	Vapor Pressure of Mercury as a Function of Temperature
quakes	Locations of Earthquakes off Fiji
randu	Random Numbers from Congruential Generator RANDU
rivers	Lengths of Major North American Rivers
rock	Measurements on Petroleum Rock Samples
sleep	Student's Sleep Data
stack.loss (stackloss)	Brownlee's Stack Loss Plant Data
stack.x (stackloss)	Brownlee's Stack Loss Plant Data
stackloss	Brownlee's Stack Loss Plant Data
state.abb (state)	US State Facts and Figures
state.area (state)	US State Facts and Figures
state.center (state)	US State Facts and Figures
state.division (state)	US State Facts and Figures
state.name (state)	US State Facts and Figures
state.region (state)	US State Facts and Figures
state.x77 (state)	US State Facts and Figures
sunspot.month	Monthly Sunspot Data, from 1749 to "Present"
sunspot.year	Yearly Sunspot Data, 1700-1988
sunspots	Monthly Sunspot Numbers, 1749-1983
swiss	Swiss Fertility and Socioeconomic Indicators (1888) Data
treering	Yearly Treering Data, -6000-1979
trees	Girth, Height and Volume for Black Cherry Trees

Continued on next page

Table 19.1 – continued from previous page

Dataset	Description
uspop	Populations Recorded by the US Census
volcano	Topographic Information on Auckland's Maunga Whau Volcano
warpbreaks	The Number of Breaks in Yarn during Weaving
women	Average Heights and Weights for American Women

19.1.8 US states facts and figures

Names of states:

```
> datasets::state.name
 [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"       "California"␣
↪    "Colorado"
 [7] "Connecticut"    "Delaware"       "Florida"        "Georgia"        "Hawaii"         ␣
↪    "Idaho"
[13] "Illinois"       "Indiana"        "Iowa"           "Kansas"         "Kentucky"       ␣
↪    "Louisiana"
[19] "Maine"          "Maryland"       "Massachusetts"  "Michigan"       "Minnesota"      ␣
↪    "Mississippi"
[25] "Missouri"       "Montana"        "Nebraska"       "Nevada"         "New␣
↪Hampshire"  "New Jersey"
[31] "New Mexico"     "New York"       "North Carolina" "North Dakota"   "Ohio"           ␣
↪    "Oklahoma"
[37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina" "South Dakota
↪"   "Tennessee"
[43] "Texas"          "Utah"           "Vermont"        "Virginia"       "Washington"␣
↪    "West Virginia"
[49] "Wisconsin"      "Wyoming"
```

Abbreviations of states:

```
> datasets::state.abb
 [1] "AL" "AK" "AZ" "AR" "CA" "CO" "CT" "DE" "FL" "GA" "HI" "ID" "IL" "IN" "IA" "KS"
↪"KY" "LA" "ME" "MD" "MA"
[22] "MI" "MN" "MS" "MO" "MT" "NE" "NV" "NH" "NJ" "NM" "NY" "NC" "ND" "OH" "OK" "OR"
↪"PA" "RI" "SC" "SD" "TN"
[43] "TX" "UT" "VT" "VA" "WA" "WV" "WI" "WY"
```

Longitudes and latitudes:

```
> datasets::state.center
$x
 [1]  -86.7509 -127.2500 -111.6250  -92.2992 -119.7730 -105.5130  -72.3573  -74.9841 ␣
↪-81.6850  -83.3736
[11] -126.2500 -113.9300  -89.3776  -86.0808  -93.3714  -98.1156  -84.7674  -92.2724 ␣
↪-68.9801  -76.6459
[21]  -71.5800  -84.6870  -94.6043  -89.8065  -92.5137 -109.3200  -99.5898 -116.8510 ␣
↪-71.3924  -74.2336
[31] -105.9420  -75.1449  -78.4686 -100.0990  -82.5963  -97.1239 -120.0680  -77.4500 ␣
↪-71.1244  -80.5056
[41]  -99.7238  -86.4560  -98.7857 -111.3300  -72.5450  -78.2005 -119.7460  -80.6665 ␣
↪-89.9941 -107.2560

$y
 [1] 32.5901 49.2500 34.2192 34.7336 36.5341 38.6777 41.5928 38.6777 27.8744 32.3329␣
↪31.7500 43.5648 40.0495
```

```
[14] 40.0495 41.9358 38.4204 37.3915 30.6181 45.6226 39.2778 42.3645 43.1361 46.3943␣
↪32.6758 38.3347 46.8230
[27] 41.3356 39.1063 43.3934 39.9637 34.4764 43.1361 35.4195 47.2517 40.2210 35.5053␣
↪43.9078 40.9069 41.5928
[40] 33.6190 44.3365 35.6767 31.3897 39.1063 44.2508 37.5630 47.4231 38.4204 44.5937␣
↪43.0504
```

Divisions:

```
> datasets::state.division
 [1] East South Central Pacific            Mountain           West South Central␣
↪Pacific
 [6] Mountain           New England        South Atlantic     South Atlantic     ␣
↪South Atlantic
[11] Pacific            Mountain           East North Central East North Central West␣
↪North Central
[16] West North Central East South Central West South Central New England       ␣
↪South Atlantic
[21] New England        East North Central West North Central East South Central West␣
↪North Central
[26] Mountain           West North Central Mountain           New England       ␣
↪Middle Atlantic
[31] Mountain           Middle Atlantic    South Atlantic     West North Central East␣
↪North Central
[36] West South Central Pacific            Middle Atlantic    New England       ␣
↪South Atlantic
[41] West North Central East South Central West South Central Mountain            New␣
↪England
[46] South Atlantic     Pacific            South Atlantic     East North Central␣
↪Mountain
9 Levels: New England Middle Atlantic South Atlantic East South Central ... Pacific
> table(datasets::state.division)

       New England    Middle Atlantic     South Atlantic East South Central West␣
↪South Central
                 6                   3                   8                   4          ␣
↪       4
East North Central West North Central           Mountain            Pacific
                 5                   7                   8                   5
```

Area in square miles:

```
> datasets::state.area
 [1]  51609 589757 113909  53104 158693 104247   5009   2057  58560  58876   6450 ␣
↪83557  56400  36291  56290
[16]  82264  40395  48523  33215  10577   8257  58216  84068  47716  69686 147138 ␣
↪77227 110540   9304   7836
[31] 121666  49576  52586  70665  41222  69919  96981  45333   1214  31055  77047 ␣
↪42244 267339  84916   9609
[46]  40815  68192  24181  56154  97914
```

Regions:

```
> datasets::state.region
 [1] South         West          West          South         West          West      ␣
↪   Northeast
 [8] South         South         South         West          West          North␣
↪Central North Central
```

```
[15] North Central North Central South         South         Northeast     South
↪    Northeast
[22] North Central North Central South         North Central West          North
↪Central West
[29] Northeast     Northeast     West          Northeast     South         North
↪Central North Central
[36] South         West          Northeast     Northeast     South         North
↪Central South
[43] South         West          Northeast     South         West          South
↪    North Central
[50] West
Levels: Northeast South North Central West
> table(datasets::state.region)

    Northeast         South North Central          West
            9            16            12            13
```

Several statistics for the states:

```
> head(datasets::state.x77)
           Population Income Illiteracy Life Exp Murder HS Grad Frost   Area
Alabama          3615   3624        2.1    69.05   15.1    41.3    20  50708
Alaska            365   6315        1.5    69.31   11.3    66.7   152 566432
Arizona          2212   4530        1.8    70.55    7.8    58.1    15 113417
Arkansas         2110   3378        1.9    70.66   10.1    39.9    65  51945
California      21198   5114        1.1    71.71   10.3    62.6    20 156361
Colorado         2541   4884        0.7    72.06    6.8    63.9   166 103766

> summary(datasets::state.x77)
   Population        Income       Illiteracy       Life Exp         Murder
↪HS Grad
 Min.   :  365   Min.   :3098   Min.   :0.500   Min.   :67.96   Min.   : 1.400   Min.
↪   :37.80
 1st Qu.: 1080   1st Qu.:3993   1st Qu.:0.625   1st Qu.:70.12   1st Qu.: 4.350   1st
↪Qu.:48.05
 Median : 2838   Median :4519   Median :0.950   Median :70.67   Median : 6.850
↪Median :53.25
 Mean   : 4246   Mean   :4436   Mean   :1.170   Mean   :70.88   Mean   : 7.378   Mean
↪   :53.11
 3rd Qu.: 4968   3rd Qu.:4814   3rd Qu.:1.575   3rd Qu.:71.89   3rd Qu.:10.675   3rd
↪Qu.:59.15
 Max.   :21198   Max.   :6315   Max.   :2.800   Max.   :73.60   Max.   :15.100   Max.
↪   :67.30
     Frost             Area
 Min.   :  0.00   Min.   :  1049
 1st Qu.: 66.25   1st Qu.: 36985
 Median :114.50   Median : 54277
 Mean   :104.46   Mean   : 70736
 3rd Qu.:139.75   3rd Qu.: 81163
 Max.   :188.00   Max.   :566432
```

19.2 Public Data Sets

1. Datasets from the book: A Handbook of Small Data Sets

19.3 Small Data Sets

These data sets have been picked up from Datasets from the book: A Handbook of Small Data Sets .

19.3.1 Germinating Seeds

This dataset studies the effect of different amounts of water on germination of seeds.

- There are two experiments: one with covered boxes and one with uncovered boxes.
- Each experiment has four boxes.
- There are six levels of water (coded as level 1 to 6).
- Each box starts with 100 seeds.
- At the end of the experiment, the number of seeds germinating in the box.
- The columns in the data set represent water levels.
- The rows represent the box numbers.
- First four rows capture the results on uncovered boxes.
- Second four rows capture the results on covered boxes.
- One entry is missing in last row.

Following code shows how to read the data:

```
read.germinating.seeds <- function(){
    df <- read.table('germin.dat', na.strings = '*')
    dfa <- df[1:4, ]
    dfb <- df[5:8, ]
    list(
        starting.seeds=100,
        uncovered.box.germination.data=dfa,
        covered.box.germination.data=dfb,
        water.levels=factor(1:6),
        boxes=factor(1:4)
        )
}
```

19.4 Test Data

This section contains some simple vectors / matrices etc. useful for examples in this document.

19.4.1 Matrices

General

```
> matrix(c(1,1,1,3,0,2), nrow=3)
     [,1] [,2]
[1,]    1    3
[2,]    1    0
[3,]    1    2
> matrix(c(0,7,2,0,5,1), nrow=3)
```

```
     [,1] [,2]
[1,]    0    0
[2,]    7    5
[3,]    2    1
```

Permutation matrices:

```
> diag(3)[c(1,3,2), ]
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    0    1
[3,]    0    1    0
> x <- c (3,4,1,2)
> diag(length(x))[x, ]
     [,1] [,2] [,3] [,4]
[1,]    0    0    1    0
[2,]    0    0    0    1
[3,]    1    0    0    0
[4,]    0    1    0    0
```

Orthogonal matrices:

```
> diag(1,nrow=2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> theta <- pi/4; matrix(c(cos(theta), sin(theta), -sin(theta), cos(theta)), nrow=2)
          [,1]       [,2]
[1,] 0.7071068 -0.7071068
[2,] 0.7071068  0.7071068
```

Symmetric positive definite matrices:

```
> A <- matrix(c(5,1,1,3),2,2)
> A
     [,1] [,2]
[1,]    5    1
[2,]    1    3
> eigen(A)$values
[1] 5.414214 2.585786
> A <- matrix(c(4, 12, -16, 12, 37, -43, -16, -43, 98), nrow=3)
> A
     [,1] [,2] [,3]
[1,]    4   12  -16
[2,]   12   37  -43
[3,]  -16  -43   98
> eigen(A)$values
[1] 123.47723179  15.50396323   0.01880498
```

Upper triangular matrices:

```
> A <- matrix(c(2, 0, 0, 6, 1, 0, -8, 5, 3), nrow=3)
> A
     [,1] [,2] [,3]
[1,]    2    6   -8
[2,]    0    1    5
[3,]    0    0    3
```

CHAPTER 20

Tips

This chapter contains some general tips for performing the data analysis. These are based on my own learnings, and will evolve over time.

20.1 Exploratory data analysis

- Understand all the variables in their data set.
- Separate out factor variables and numerical variables.
- Distinguish between response variables and independent variables.
- Look at the histogram of numerical variables.
- If the histogram doesn't look normal, see if some data transformation can make the histogram look so.
- Look at the Pearson correlations between numerical variables. Categorize them between very weak, weak, moderate, strong, very strong correlations.
- Compute Spearman correlation between a factor variable with other numerical/factor variables.
- Compute factor-wise box plots of numerical variables. Examine them to see if the box plots for different levels of a factor are significantly different.

Handling NA data

- Make sure that you look at raw data and identify the patterns used for entering NA values. It can be NA, na, blank space, *, etc.
- Count the number of NA entries in each column of data set.
- Identify variables with very high NA percentage. Consider if you should totally eliminate the variable from further data analysis.
- If there are very few NA entries, one approach can be to eliminate the corresponding rows.

- One way of filling NA values is by computing median / mean of the corresponding variable and using that value in all NA slots for that variable.
- Alternatively, one can use the non-NA entries in the variable and fit a linear / non-linear model for that variable from other variables which have good quality data. Then, one can use this model to predict the NA entries.
- Make sure that your data-set is cleaned of NA values before serious modeling is done.

CHAPTER 21

Kaggle

21.1 References

1. Titanic: Getting Started With R

CHAPTER 22

Indices and tables

- genindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z