Multivaraite Analytics

using R

Kamakshaiah Musunuru

Contents

1 Introduction to R **11**

1.1 Free software . . . 11

1.1.1 About GNU . . . 12

1.2 Open Source Software . . . 13

1.2.1 History of OSS . . . 13

1.3 About R . . . 14

1.3.1 History of R . . . 15

1.3.2 Installation . . . 17

1.3.3 Editor . . . 18

1.4 Operators . . . 20

1.4.1 Arithmetic operators . . . 21

1.4.2 Relational operators . . . 22

1.4.3 Logical operators . . . 22

1.4.4 Assignment operators . . . 23

2 Data Types **29**

2.1 Programming paradigm . . . 30

2.1.1 R . . . 32

2.2 Help in R . . . 33

2.2.1 `help` function . . . 34

2.2.2 `??` . . . 35

2.3 `RSiteSearch` . . . 36

2.3.1 Vignettes . . . 36

2.3.2 Searching for functions . . . 37

2.4 Data types . . . 40

2.4.1 Data type vs data structure . . . 41

2.5 Vectors . . . 48
2.6 Factors . . . 50
2.7 Lists . . . 52
2.8 Data matrix . . . 54
2.8.1 Matrix operations . . . 56
2.8.2 Identity matrix . . . 58
2.8.3 Matrix multiplications . . . 59
2.9 Data Frame . . . 60
2.9.1 Indexing data frames . . . 62
2.9.2 Data frame vs data matrix . . . 64

3 Functions 67
3.1 Introduction . . . 67
3.2 Arguments . . . 71
3.3 Anonymous functions . . . 73
3.4 Nested Functions . . . 75
3.5 Applications . . . 77
3.5.1 Vector manipulation . . . 77
3.5.2 Random matrix . . . 79
3.5.3 Zeros matrix . . . 79
3.5.4 Ones matrix . . . 80
3.5.5 Stacks and queues . . . 81

4 Control Flow 85
4.1 Conditional Statements . . . 86
4.1.1 Vectorized if statement . . . 87
4.1.2 if-else statement . . . 88
4.1.3 Nested if conditions . . . 88
4.2 Applications - Conditional statements . . . 90
4.2.1 Data transformations . . . 91
4.2.2 Random matrix . . . 92
4.2.3 Positive semidefinit matrices . . . 92
4.2.4 Probability distributions . . . 93
4.3 Loops . . . 95
4.3.1 `for` . . . 95
4.3.2 `repeat` . . . 96
4.3.3 `while` . . . 97
4.3.4 `switch()` . . . 98
4.4 Applications - Loops . . . 99

4.4.1 Loops for Lists . . . 99
4.4.2 Loops for summaries . . . 100
4.4.3 Loops for matrix operations . . . 100

5 Multivariate Analysis **105**
5.0.1 Techniques . . . 106
5.1 Data Simulations . . . 107
5.1.1 Numeric data . . . 108
5.1.2 Non-numeric data . . . 110
5.1.3 Multivariate heterogeneous data . . . 112
5.2 Missing data . . . 113
5.2.1 Substitution . . . 114
5.2.2 Deletion . . . 115
5.3 Correlation . . . 116
5.3.1 Multiple correlation . . . 118
5.3.2 Polychoric correlation . . . 120
5.3.3 Partial Correlation . . . 124
5.3.4 Canonical Correlation . . . 127
5.4 Regression . . . 130
5.4.1 Linear regression . . . 130
5.4.2 Multiple regression . . . 131
5.4.3 Multivariate regression . . . 133
5.5 Multidimensional Scaling . . . 134
5.5.1 Metric/Classical MDS . . . 136
5.5.2 Non-Metric MDS . . . 140
5.6 Correspondence Analysis . . . 143
5.6.1 Simple correspondence analysis . . . 144
5.6.2 MCA . . . 146
5.6.3 MJCA . . . 148
5.7 Exploratory Factor Analysis . . . 150
5.7.1 Methodology . . . 151
5.7.2 Internal consistency . . . 151
5.7.3 Statistical diagnosis . . . 152
5.7.4 Structure diagram . . . 154
5.8 Confirmatory factor analysis . . . 155
5.8.1 Maximum Likelihood Estimation . . . 156
5.8.2 SEM using `Political Democracy` dataset . . . 157
5.8.3 SEM using `multivardata.csv` . . . 158
5.9 Cluster analysis . . . 160

5.9.1 Types of techniques 160
5.9.2 Determine number of clusters 161
5.9.3 K-Means cluster analysis 162
5.9.4 Hierarchical agglomerative clustering 162
5.9.5 Multiscale bootstrap resampling (PVCLUST) 163
5.9.6 Model Based 163
5.9.7 Plotting Cluster Solutions 164

About the Author

Dr. M. Kamakshaiah is a open source software evangelist and enterprise architect. He has been teaching QT and IT practices related to business management for few decades. He has traveled to a couple of countries on teaching and research assignments. He has been teaching data science and analytics using open source software tools like R, Python for survey data analytics and Hadoop for big data analytics. He is also expert of real time data analytics using AVR uC and ARM processor. He is a full stack R, Python developer. Creating web applications for teaching and learning practices is his hobby. All his applications are free and open source. Please visit `https://github.com/Kamakshaiah` for his software applications.

Foreward

I learned R by self study. I mean, through very informal personal learning. All my learning is from online resources. However, I could produce close to three hundred data analysts through my formal classes by the end of 2019. I am writing all this not to show myself as a valiant learner, but to demonstrate how can a novice and a naive enthusiast like me can learn programming tools like R. Today, I am offering a couple of courses to teach Python, Hadoop and IoT, that is all by passion. So, I would like to give confidence to the readers that you don't need any formal learning in computer science or information technology to learn data science and adopt it as profession. However, you need tons and tons of patience and passion.

This book is a sleek and slender manual for beginners. The primary aim of this book is not programming, but to provide a comprehensive manual for practice of R. There is only very short description on programming concepts such as control flow, loops, functions etc. However, there is sufficient information on practice of numerical simulations and data visualization. While coming to contents; this text has 5 chapters and each chapter represents a unique concept of related to practice of code in R. The idea behind 5 chapters is that the learner may be able to pick R with in few weeks and use skills for basic data analysis. This book might be much helpful for practitioners, academics, scholars and students to acquire knowledge on R in a very short time.

This book has a companion repository located at `https://github.com/Kamakshaiah/r-for-mva`. Though, there is not much code associated with this text, but data sets together with a couple of scripts can help the learner practice programming for multivariate data analysis. Obtain these scripts and practice while reading code chunks in this text. Feel free to reach me using kamakshaiah.m@gmail.com for doubts and any other queries.

Happy reading $\cdots$

Dr. M. Kamakshaiah
Author

Chapter 1

Introduction to R

God is not free but his creation is free. Free means absolute freedom but not restricted freedom. In the beginning everything was free, say soil, water, wood, wood. Today, only air is free rest of all has a price. I never bought water in my childhood. Today, it is not possible to buy water, never ever think of that. The problem is not about price but a matter of hygiene. I am sure, we may have to buy air in future. Software was free in the yore. However, slowly became a corporate offering by the time passed.

1.1 Free software

Free software means the users have the freedom to run, copy, distribute, study, change and improve the software. Free software is a matter of liberty, not price. To understand the concept, you should think of "free" as in "free speech", not as in "free beer". [1] More precisely, free software means users of a program have the four essential freedoms:

1. The freedom to run the program as you wish, for any purpose (freedom 0).
2. The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access

to the source code is a precondition for this.

3. The freedom to redistribute copies so you can help others (freedom 2).
4. The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

Developments in technology and network use have made these freedoms even more important now than they were in 1983. Nowadays the free software movement goes far beyond developing the GNU system. The free software movement campaigns to win for the users of computing the freedom that comes from free software. Free software puts its users in control of their own computing. Nonfree software puts its users under the power of the software's developer.

1.1.1 About GNU

GNU is a Unix-like operating system. That means it is a collection of many programs, say applications, libraries, developer tools, even games. The development of GNU, started in January 1984, is known as the GNU Project. Many of the programs in GNU are released under the auspices of the GNU Project. The name "GNU" is a recursive acronym for "GNU's Not Unix." GNU is pronounced *g'noo*, as one syllable, like saying "grew" but replacing the r with n.

The program in a Unix-like system that allocates machine resources and talks to the hardware is called the "kernel". GNU is typically used with a kernel called Linux. This combination is the GNU/Linux operating system. GNU/Linux is used by millions, though many call it "Linux" by mistake. GNU's own kernel, *The Hurd*, was started in 1990 (before Linux was started). The Hurd is still under development and continue to remain as it is.

GNU is an operating system that is free software that is, it respects users' freedom. The GNU operating system consists of GNU packages (programs specifically released by the GNU Project) as well as free software released by third parties. The development of GNU made it possible to use a computer without software that would trample your freedom.

1.2 Open Source Software

Open-source software (OSS) is computer software that is released under a license in which the copyright holder grants users the rights to use, study, change, and distribute the software and its source code to anyone and for any purpose.[2][3] Open-source software may be developed in a collaborative public manner. Open-source software is a prominent example of open collaboration, meaning any capable user is able to participate online in development, making the number of possible contributors indefinite. The ability to examine the code facilitates public trust in the software. [4]

Open-source software development can bring in diverse perspectives beyond those of a single company. A 2008 report by the Standish Group stated that adoption of open-source software models has resulted in savings of about $60 billion per year for consumers.[5]

Open source code can be used for studying and allows capable end users to adapt software to their personal needs in a similar way user scripts and custom style sheets allow for web sites, and eventually publish the modification as a fork for users with similar preferences, and directly submit possible improvements as pull requests.

1.2.1 History of OSS

Netscape's act prompted Raymond and others to look into how to bring the Free Software Foundation's free software ideas and perceived benefits to the commercial software industry. They concluded that FSF's social activism was not appealing to companies like Netscape, and looked for a way to rebrand the free software movement to emphasize the business potential of sharing and collaborating on software source code. The new term they chose was *open source*, which was soon adopted by Bruce Perens, publisher Tim O'Reilly, Linus Torvalds, and others. The Open Source Initiative was founded in February 1998 to encourage use of the new term and evangelize open source principles. [6]

While the Open Source Initiative sought to encourage the use of the new term and evangelize the principles it adhered to, commercial software vendors found themselves increasingly threatened by the concept of freely distributed software and universal access to an application's source code. Free and open-source software has historically played a role outside of the mainstream of private software development, companies as large as Microsoft have begun to develop official open-source presences on the Internet. IBM, Oracle and Google are just a few of the companies with a serious public stake in today's competitive open source market. There has been a significant shift in the corporate philosophy concerning the development of Free Open Source Software (FOSS) or Free and Libre Open Source Software (FLOSS).

The free software movement was launched in 1983. In 1998, a group of individuals advocated that the term free software should be replaced by open-source software (OSS) as an expression which is less ambiguous and more comfortable for the corporate world. Software developers may want to publish their software with an open source license, so that anybody may also develop the same software or understand its internal functioning. With open source software, generally, anyone is allowed to create modifications of it, port it to new operating systems and instruction set architectures, share it with others or, in some cases, market it.

1.3 About R

R is a programming language and free software environment for statistical computing and graphics. R project is supported by the R Foundation. The R language is widely used among statisticians and data miners for developing statistical software and data analysis. R is widely used for processing data gathered through Polls, data mining surveys, and studies of scholarly literature databases. R became so popular in academic and scholarly circles, as of June 2020, R ranks 9^{th} in the TIOBE index. [7]

1.3.1 History of R

R is a GNU software. The official R software environment is written primarily in C, Fortran, and R itself (thus, it is partially self-hosting) and is freely available under the GNU General Public License. [8] Pre-compiled executables are provided for various operating systems. Although R has a command line interface, there are several third-party graphical user interfaces, such as RStudio, an integrated development environment, and Jupyter, a notebook interface.

R is an implementation of the S programming language combined with lexical scoping semantics, inspired by Scheme. S was created by John Chambers in 1976, while at Bell Labs. There are some important differences, but much of the code written for S runs unaltered. [9] At first, it was very difficult find his data online. I was having one simple image of him, see his image 1.1, which I used in my first book. There were hardly a couple of online talks. However, when I was writing this piece of text somewhere in monsoon, 2022, I was so delighted to find his latest talk online. Please visit `https://www.youtube.com/watch?v=qWG_MLrxKps` to hear to him.

Figure 1.1: John M Chambers

R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is developed by the R Development Core Team, of which, as of August 2018, Chambers was a member. R is named partly after the first names of the first two R authors and partly as a play on the name of S. The project was conceived in 1992, with an initial version released in 1995 and a stable beta version (v1.0) on 29 February 2000. Though the very first version of R was released in February, 2000 albeit of the fact that the source code was rampantly used by academics as a matter of testing. [10] Ross Ihaka's wrote a very first paper on R, it is still available at Auckland University's archives for readers. [11] Most of the academics only take the official date of release in to consideration but not the presence of software. There is lot of online information in support of the fact that the efforts of R were started early in 1990s. [12]

Figure 1.2: Ross Ihaka & Robert Gentleman

R is GNUs S. R is lingua franca of statistics. The project site http://www.r-project.org/ is the place where official information about R is available. There are other places where plenty of other information is available. They are (1) github, (2) Rforge. All these three places are active in all respects. [13] R is not only renowned for statistical computing but also respected by experts as research managemet system. R is used in almost all domains of knowledge, viz., geology, genomics, biology, chemistry, mathematics etc. The way it is used is not superficial, but marvelous. R can perform better than any proprietary tool irrespective of domain. The power of R lies in the way programming can be done. There are other statistical computing suits of the same nature but they are conventional in nature. For example, SAS, the another programming environment for statistical and mathematical analysis, is not programmer friendly. [14]. R has a project portal, `http://cran.r-project.org/`, where there is a treasure like information related to downloads, installation, community and et. There is another important place without which any discussion on R is incomplete, that is CRAN. You will read more about CRAN in upcoming sections.

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. It is a GNU project and similar to the S language which was developed at Bell Laboratories (formerly ATT, now Lucent Technologies) by John Chambers and his colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R. R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time series analysis, classification, clustering are only few to name) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.

One of R's strengths is the ease with which well designed publication quality plots can be produced, including mathematical symbols and formula where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control. R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms

1.3.2 Installation

The user need to install software in order to use the same. Installation is a typical computer charade. There are two main OSes in the world namely Windows and Linux. Of course, I respect other OSes such as MacOS and (Free) BSD. In the area of desktop and laptop computers, Microsoft Windows is generally above 70% in most markets and at 77% to 87.8% globally, Apple's macOS is at around 9.6% to 13%, Google's Chrome OS is up to 6% (in the US) and other Linux distributions are at around 2%. All these figures vary somewhat in different markets, and depending on how they are gathered. [15] Linux is there everywhere, but tracking the same is difficult. It is possible to enumerate proprietary deployments but not open source operating systems. Users normally just download or obtain OSOS and it is difficult to audit such deployments. I guess, the actual deployments of Linux go beyond the actual statistics. For instance, most of the softphones has Android OS and it is Linux based OS.

Coming back to the topic. Installing R in any OS is pretty straight forward. Some extent, it is easy to install and setup R in Windows compared to Linux. However, Linux is getting popular in development communities. Mostly developers try software in Linux based OSs.

R installation is pretty simple. Just obtain right build for your Windows version from https://cran.r-project.org/bin/windows/base/, save anywhere in your computer, open the file, just keep clicking "next", "next" and "next" as and when prompted. There are lot of online sources as how to obtain and install R in Windows.

Installing Linux in Linux is also not so difficult but one need to do this from Terminal. Ubuntu is very popular GNU/Linux operating system. Ubuntu is fifth popular OS as per average number of hits per day (HPD). [16] Ubuntu is Debian based distribution, and *.deb* is format for all executables. There are two ways to install software in Ubuntu. *First*, most of the software is available from *Ubuntu Software* Manager. There are thousands of software packages which are freely available for user practice. Just open software manager, search for "R" and install. *Second*, just open the terminal and execute the following statement.

```
sudo apt-get install r-base
```

This is safest method among all. Ubuntu take care of dependencies.

1.3.3 Editor

There are tons of text editors available and many of them have modes for highlighting R syntax. We will be using RStudio for this text. RStudio is an IDE available for Windows/Mac/Linux that integrates editor, console, file tree, graphics windows, has highlighting, helps you keep track of your functions and variables. Above all, it is free and open source. RStudio works well with Sweave, which is a popular TeX package for reproduciable research. However, RStudio is not the only editor there are few but differ in functionality.

Editor	About
Crimson Editor	Good replacement for Notepad, it also offers many powerful features for programming languages such as HTML, C/C++, Perl and Java.
Tinn-R	Open source (GNU General Public License) and free project, exclusively meant for R environment.
Emacs	An extensible, customizable, free/libre text editor - in fact, much more.
JGR	Java GUI for R. Pronounced “Jaguar”. is a universal and unified Graphical User Interface for R.
RKWard	Combine the power of the R-language with the (relative) ease of use of commercial statistics tools.

Table 1.1: R Editors

I will be using RStudio for this text, despite of all the above editors. RStudio has mature to a larger extent for variegated needs of users. Today, it is possible to deploy R in web applications that is due to the advent of Shiny frameword from the labs of RStudio. R joined in the legion of those tools which are in high demand for web development. Anyway, I am not going to discuss any of development in this text. I will be dealing with very few methods for writing User Defined Functions (UDF).

This text is not for absolute beginners and I assume that the reader knows basics of R. By basics I mean: (1) obtaining and installing R, (2) using editors (at least RStudio), (3) managing project folders (paths) and (3) writing and executing scripts. The main goal of this text is to explore the power of functional programmig approach and its utility for computational statistics. There are few resources online as how to obtain and set R as well as it's editors.

Inside the Console

R has a Terminal type utility which does Read, Eval, Print, Loop (REPL) kind of activity. Few people call it R Console may be because RStudio mentions this terminal as “Console”, which you find at the left bottom corner. [17] The main activity of R starts either by opening a script for code or straitly executing code in the Console. Console is meant for testing of code, whereas, scripts meant for development. Scripts are highly useful for reproduciability of work and future reference. Writing scripts is always a best way of working with programming languages. We shall see working with scripts in forthcoming chapters, but for now let me tackle Console.

As it was said, Console is available from left-bottom corner of RStudio. Let us explore R for pretty little calculations.

```
> 1+2
```

```
[1] 3
```

```
> a=1; b=2; c=a+b
> print(c)
```

```
[1] 3
```

```
> a <- 1; b <- 2; c <- a+b
> c
```

```
[1] 3
```

The output is self explanatory. In R, there is difference between `=` and `<-`. The second symbol is called *left assignment operator*. The equals sign is reserved for internal assignments which we will see in forthcoming sections.

1.4 Operators

Programming languages typically support a set of operators. Operators are constructs which behave generally like functions, but differ syntactically or semantically from usual functions. Common simple examples include arithmetic (addition with +), comparison (such as >), and logical operations (such as AND or &&). More involved examples include assignment (usually = or :=), field access in a record or object (usually .), and the scope resolution operator (often :: or .). Languages usually define a set of built-in operators, and in some cases allow users to add new meanings to existing operators or even define completely new operators.

Syntactically operators usually contrast to functions. In most languages, functions may be seen as a special form of prefix operator with fixed precedence level and associativity, often with compulsory parentheses e.g. Func(a) (or (Func a) in Lisp). Most languages support programmer defined functions, but cannot really claim to support programmer defined operators, unless they have more than prefix notation and more than a single precedence level. Semantically operators can be seen as special form of function with different calling notation and a limited number of parameters.

There are different types of operators depending on needs of the user. In R there are variety of operators. However, I shall introduce three very essential and important operators namely

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators

1.4.1 Arithmetic operators

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

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

```
> a = 4; b = 8
> b %% a

[1] 0

> b %/% a

[1] 2
```

So, `%%` gives remainder, whereas `%/%` gives quotient.

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

1.4.2 Relational operators

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

Suppose imagine that there are two variables [18] namely *a* and *b*. Let us assign two values say, 12, 23. Relational operators are useful greatly if we want to compare these two variables (objects). The output for above operators is going to be

```
> a = 12; b = 23
> a<b

[1] TRUE

> a>b

[1] FALSE

> a=b
> a<=b

[1] TRUE

> a>=b

[1] TRUE

> a!=b

[1] FALSE
```

1.4.3 Logical operators

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

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

Let us take Boolean values such as *TRUE* and *FALSE* for two different objects namely a and b. [19] Try testing logical operators.

```
> a=TRUE; b=FALSE
> a & b

[1] FALSE

> a && b

[1] FALSE

> a | b

[1] TRUE

> a || b

[1] TRUE
```

1.4.4 Assignment operators

R assignment operators are used to assign values to variables. These operators are a bit more influential while writing code unlike other operators.

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

The operators $< -$ and $=$ can be used, almost interchangeably, in the same environment. The $<< -$ operator is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used. Let us try:

```
> x <- 5
> x
[1] 5
> x <<- 6
> x
[1] 6
> x = 9
> x
[1] 9
```

They are all seems to be same at rudimentary level. Believe me there are differences among them. $<<-$ is used while using global variables. To understand global vs. local you need to know a concept known as *scoping*. Usually this parameter is used to create child function inside a parent function. The best example can be:

```
> a <- 1
> myfunct <- function(){
+   b <- 2
+   a <- 3
+   print(c(a, b, a+b, const <- 4))
+ }
> myfunct()
[1] 3 2 5 4
> print(a)
[1] 1
```

The concept functions is not discussed till now. The object `a` remains same with global value, in spite of it being changed inside the function `myfunct()`. I will bring few changes in the code by using $<<-$ for `a` and `=` for object `const`.

```
> a <- 1
> myfunct <- function(){
+   b <- 2
+   a <<- 3
+   print(c(a, b, a+b, const = 4))
+ }
> myfunct()

                         const
    3       2       5       4

> print(a)

[1] 3
```

The object `a` got updated gloabally just by changing assignment operator. This is what is called *scoping*. We shall discuss this in the forthcoming sections. The object `const` is not just a variables but a named object.

```
> myfunct()[1]

                   const
    3      2      5      4
3

> myfunct()[2]

                   const
    3      2      5      4
2

> myfunct()[3]

                   const
    3      2      5      4
5

> myfunct()[4]

                   const
    3      2      5      4
const
    4

> myfunct()["const"]

                   const
    3      2      5      4
const
    4
```

It is possible to call the 4th element by name. The operator `=` is mostly used for naming objects. For instance, I often remove objects in the R session using `rm(list = ls())` statement.

```
> ls()

[1] "a"        "b"        "c"        "myfunct" "x"

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

character(0)
```

The operator `<-` will not work in the above statement. R will throw error

```
Error in rm(list <- ls()) : ... must contain names or character strings
```

Notes

[1]Obtained as it is from https://www.gnu.org/home.en.html

[2]St. Laurent, Andrew M. (2008). Understanding Open Source and Free Software Licensing. O'Reilly Media. p. 4. ISBN 9780596553951.

[3]Corbly, James Edward (25 September 2014). "The Free Software Alternative: Freeware, Open Source Software, and Libraries". Information Technology and Libraries. 33 (3): 65. doi:10.6017/ital.v33i3.5105. ISSN 2163-5226.

[4]Levine, Sheen S.; Prietula, Michael J. (30 December 2013). "Open Collaboration for Innovation: Principles and Performance". Organization Science. 25 (5): 14141433.

[5]Rothwell, Richard (5 August 2008). "Creating wealth with free software". Free Software Magazine. Archived from the original on 8 September 2008. Retrieved 8 September 2008.

[6]History of the OSI. Available from https://opensource.org/history.

[7]R was 22^{nd} at the end of June, 2019. Visit https://www.tiobe.com/tiobe-index/ for more details.

[8]Wrathematics (27 August 2011). "How Much of R Is Written in R". librestats. Archived from the original on 12 June 2018. Retrieved 7 August 2018.

[9]R (programming language). Obtained from https://en.wikipedia.org/wiki/R_(programming_language)

[10]Please visit CRAN repository https://cran.r-project.org/src/base/R-0/ for old sources.

[11]Please read Ihaka's paper from https://www.stat.auckland.ac.nz/~ihaka/downloads/Interface98.pdf

[12]Visit https://blog.revolutionanalytics.com/2016/03/16-years-of-r-history.html for more details. There is very interesting information related to R history at this place.

[13]To know the best place for R; you may view nice discussion at http://stackoverflow.com/questions/7416949/r-forge-vs-rforge

[14]Prof. Ravelle mention that SAS is 11 years behind R

[15]Operating System Market Share. NetMarketShare. NetApplications. Retrieved 24 September 2019.

[16]Available at https://distrowatch.com/dwres.php?resource=popularity

[17]RStudio has four panes, (1) top-left: mostly used scripts, (2) top-left: Console, Terminal (OS), etc. (3) right-bottom: File browser (has few other meant for browsing) and (4) top-right: consists few sub-panes for history, environment, connections (if in network).

[18]In R everything is an object.

[19]In R Boolean values are capital or upper case letter unline in JS and Python.

Chapter 2

Data Types

All programming languages are not same. Usage of programming Languages depends on design strategy. Every programming language has got a creator being it. The design strategy depends on the way designer, say creator, conceives it. There are two creators behind R, viz. Ross and Robert. In computer science, the principle that govern design of the programing language is known as *Programming Language Theory (PLT)* (λ). PLT is a branch of computer science that deals with the design, implementation, analysis, characterization, and classification of programming languages and their individual features. [20] PLT encompasses mathematics, software engineering, linguistics and even cognitive science. It is a well recognized branch of computer science, and an active research area, with results published in numerous journals dedicated to PLT, as well as in general computer science and engineering publications.

The lambda calculus, developed by Alonzo Church and Stephen Cole Kleene in the 1930s, is considered by some to be the world's first programming language, even though it was intended to model computation rather than a means for programmers to describe algorithms to a computer system. Many modern functional programming languages have been described through lambda calculus. The first programming language to be invented was Plankalk̈l, which was designed by Konrad Zuse in the 1940s, but not publicly known until 1972 (and not implemented until 1998). The first widely known and successful high-level programming language was FORTRAN, developed from 1954 to 1957 by a team of IBM researchers led by John Backus. The success of FORTRAN led to the formation of a committee of scientists to develop a "universal" computer language; the result of their effort was ALGOL 58. Separately, John McCarthy of MIT developed the LISP programming language (based on the lambda calculus), the first language with origins in academia to be successful. With the success of these initial efforts, programming languages became an active topic of research in the 1960s and beyond.

2.1 Programming paradigm

Computer programming is the process of designing and building an executable computer program to accomplish a specific computing result. A language helps programmer to speak to computer and get the things done through it. The source code of a program is written in one or more languages that are intelligible to programmers, rather than machine code, which is directly executed by the central processing unit. The purpose of programming is to find a sequence of instructions that will automate the performance of a task on a computer, often for solving a given problem. A programmer is expected to be all rounder. Proficient programming thus often requires expertise in several different subjects, including knowledge of the application domain, specialized algorithms, and formal logic.

At surface the word programming seems to be very simple. However, tasks related to programming include testing, debugging, source code maintenance, implementation of build systems, and management of derived artifacts, such as the machine code of computer programs. These stemps might be considered part of the programming process. The term software development is used for such processes.

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms. Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar. Common programming paradigms include:

1. *Imperative* in which the programmer instructs the machine how to change its state,
    (a) *procedural* which groups instructions into procedures. Examples: FORTRAN, ALGOL, COBOL, BASIC, Pascal and C.
    (b) *object-oriented* which groups instructions together with the part of the state they operate on. Examples: Java, C++, C, Python and many more.
2. *Declarative* in which the programmer merely declares properties of the desired result, but not how to compute it
    (a) *functional* in which the desired result is declared as the value of a series of function applications. Examples: LISP, Perl, PHP, Python, and Scala.
    (b) *logic* in which the desired result is declared as the answer to a question about a system of facts and rule, mathematical in which the desired result is declared as the solution of an optimization problem. Example: Prolog.

Symbolic techniques such as reflection, which allow the program to refer to itself, might also be considered as a programming paradigm. However, this is compatible with the major paradigms and thus is not a real paradigm in its own right. For example, languages that fall into the *imperative paradigm* have two main features: they state the order in which operations occur, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code. The communication between the units of code is not explicit. Meanwhile, in *object-oriented programming*, code is organized into objects that contain a state that is only modified by the code that is part of the object. Most object-oriented languages are also imperative languages. In contrast, languages that fit the *declarative paradigm* do not state the order in which to execute operations. Instead, they supply a number of operations that are available in the system, along with the conditions under which each is allowed to execute. The implementation of the language's execution model tracks which operations are free to execute and chooses the order on its own. More at Comparison of multi-paradigm programming languages.

2.1.1 R

It is slightly difficult to analyze R design paradigm. R, at core, is a functional programming language mixed with few characteristics of logic and procedural design. There are even sources that implore that considerable amount of C and FORTRAN is involved in R development. [21] [22] Both these languages belongs to procedural paradigm. Though, R seems to be functional in nature, but shares its characteristics with other two paradigms. There are R celebrities who wrote about ways to use R for Objective Oriented Programming (OOP). Hadley Wickham, is a noted R programmer, and regarded as celebrity in R community, has written as how to use R for developing classes in his book titled "Advanced R". [23]

Anyway, I am not going discuss R from OOP point of view. Most of the statisticians prefer R because they are not from core computer sciences. Much of the OOP is rampantly used in computer science but not in math and statistics. Functional programming offers an intermediate approach to people who did not have formal computer science background. That is why, R cought the attention in mathematics and statistics circles. Jave and Pytho are slowly rising due to computer science people being entered into data science and analytics. In the beginning R was considered as the only way to do with math and stat. That is why, the slogan, *R is language of statistics* is still vogue in data science community.

Writing programs using R also follows the same process. There are ways to write programs in R and depends on writing style. Some prefer to write in Console and it is called testing. Others might prefer to write in scripts, it is called programming. That is why the word "programming", is a bit more than freaking out time. In RStudio, the menu *File -> New File -> R Script* is used to write scripts. In fact, there are number of things one can do using this menu. The disucssion is beyond the scope of this book.

2.2 Help in R

R has number of ways to access help. Help is, of course, very important for beginners. After all, how is it possible for anybody to start writing programming statements? Obviously, either through a teacher or through self study (just like me). Today, there are number of sources to get information. Frankly, it is not required to attend any class nor to consult a mentor. Everything is possible through Google and Wikipedia. As I said, no need to shy to use Wikipedia, as many teachers in academics tout to their wards. Believe me, *Wikipedia is a open source knowledge management platform.* No need to cringe or shy for using Wikipedia. Knowledge is the property of society. Using copyrights is a crime in open source communities. Just a joke! However, this has nothing to do with *license*. There are variety of licenses to sell your software. This section provide very little and thorough idea on help.

There are many ways to known about R. Following are few functions a beginner can follow to acquaint with R.

1. `help`
2. `??`
3. `RSiteSearch`
4. Vignettes

All these functions are executed internally, inside R, to search for inforamation. There are few other methods to search for inforamtion. These methods requires users to use browser.

1. R Seek (`https://rseek.org/`)
2. Quick R `https://www.statmethods.net/`

3. R-Bloggers (https://www.r-bloggers.com/)

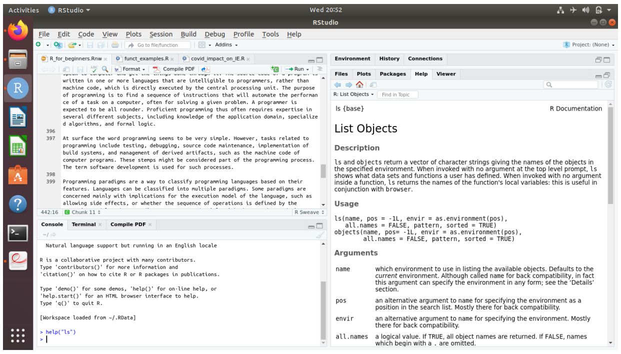
There resources are highly useful for beginners to acquaint themselves with R. R Seek is google like search engine. It retrieves information from the whole amount of data living online. Quick R is a portal maintained by certain data analytics firm, known as *DataCamp*, but designed and created by an interesting individual named *Robert I. Kabacoff*. Read about him at https://www.statmethods.net/about/author.html. He is also R addict like me and who seems to do everything from heart.

2.2.1 `help` function

R has a big fat function called **`help`**. This function must be darling of beginners. Why? Beacause, a beginner is someone who has no knowledge about R. R is developed and tested using UNIX like OS. So, it is possible to use few UNIX like commands in R. There is certain function known as **`ls`** in UNIX OS. This function return *list* of items in present directory.

```
> help("ls")
```

This statement will get you *R Documentation* related to function **`ls`** in the right bottom pane in RStudio. The output looks like the one that appears in Figure 2.1



```
written in one or more languages that are intelligible to programmers, rather than
machine code, which is directly executed by the central processing unit. The purpose
of programming is to find a sequence of instructions that will automate the performan
ce of a task on a computer, often for solving a given problem. A programmer is
expected to be all rounder. Proficient programming thus often requires expertise in
several different subjects, including knowledge of the application domain, specialize
d algorithms, and formal logic.

At surface the word programming seems to be very simple. However, tasks related to
programming include testing, debugging, source code maintenance, implementation of
build systems, and management of derived artifacts, such as the machine code of
computer programs. These stemps might be considered part of the programming process.
The term software development is used for such processes.

Programming paradigms are a way to classify programming languages based on their
features. Languages can be classified into multiple paradigms. Some paradigms are
concerned mainly with implications for the execution model of the language, such as
allowing side effects, or whether the sequence of operations is defined by the
```

```
  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> help("ls")
>
```

ls {base} R Documentation

List Objects

Description

ls and objects return a vector of character strings giving the names of the objects in the specified environment. When invoked with no argument at the top level prompt, ls shows what data sets and functions a user has defined. When invoked with no argument inside a function, ls returns the names of the function's local variables: this is useful in conjunction with browser.

Usage

```
ls(name, pos = -1L, envir = as.environment(pos),
   all.names = FALSE, pattern, sorted = TRUE)
objects(name, pos= -1L, envir = as.environment(pos),
        all.names = FALSE, pattern, sorted = TRUE)
```

Arguments

name	which environment to use in listing the available objects. Defaults to the *current* environment. Although called name for back compatibility, in fact this argument can specify the environment in any form; see the 'Details' section.
pos	an alternative argument to name for specifying the environment as a position in the search list. Mostly there for back compatibility.
envir	an alternative argument to name for specifying the environment. Mostly there for back compatibility.
all.names	a logical value. If TRUE, all object names are returned. If FALSE, names which begin with a . are omitted.

Figure 2.1: `ls` function in RStudio editor.

Don't be panic after looking at the Figure 2.1. I use Ubuntu 18.04

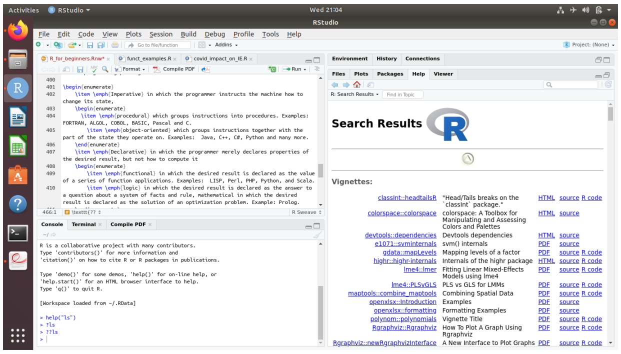
for my work. I am open source software evangelist. RStudio looks same either irrespective of OS. I mean, the look, functionality and performance has nothing to do with host OS. Right bottom side (pane) has documentation related to `ls` function. This way it is possible to get any documentation as long as it is available from namespace or base system. Use the `ls` function on acouple of other methods such as *mean, median* etc.

2.2.2 ??

Using special characters is quite natural in today's communication. Say, WhatsApp, Facebook, for that matter any chat or messenger like utilities. I use `?`, in WhatsApp, whenever I would like to ask a question just below the response or text. `?` is used in R community right from it's invention. `?` is a symbol, but in R it is a function. You can use `?` in the place of `help`.

```
> ?ls
```

This function retrieves exactly the same information as it was shown for `help` function. There is not a whit of difference. However, there is another function i.e., `??` which does things a bit differently. This function shows really abundant of information but as given in hyperlinks. Look at the Figure 2.2



```
\begin{enumerate}
    \item \emph{Imperative} in which the programmer instructs the machine how to
change its state,
    \begin{enumerate}
      \item \emph{procedural} which groups instructions into procedures. Examples:
FORTRAN, ALGOL, COBOL, BASIC, Pascal and C.
        \item \emph{object-oriented} which groups instructions together with the
part of the state they operate on. Examples:  Java, C++, C#, Python and many more.
    \end{enumerate}
    \item \emph{Declarative} in which the programmer merely declares properties of
the desired result, but not how to compute it
    \begin{enumerate}
        \item \emph{functional} in which the desired result is declared as the value
of a series of function applications. Examples:  LISP, Perl, PHP, Python, and Scala.
        \item \emph{logic} in which the desired result is declared as the answer to
a question about a system of facts and rule, mathematical in which the desired
result is declared as the solution of an optimization problem. Example: Prolog.
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> help("ls")
> ?ls
> ??ls
> 
```

Vignettes:

classInt::headtailsR	"Head/Tails breaks on the \`classInt\` package."	HTML	source	R code
colorspace::colorspace	colorspace: A Toolbox for Manipulating and Assessing Colors and Palettes	HTML	source	
devtools::dependencies	Devtools dependencies	HTML	source	
e1071::svminternals	svm() internals	PDF	source	
gdata::mapLevels	Mapping levels of a factor	PDF	source	R code
highr::highr-internals	Internals of the highr package	HTML	source	R code
lme4::lmer	Fitting Linear Mixed-Effects Models using lme4	PDF	source	R code
lme4::PLSvGLS	PLS vs GLS for LMMs	PDF	source	R code
maptools::combine_maptools	Combining Spatial Data	PDF	source	R code
openxlsx::Introduction	Examples	PDF	source	
openxlsx::formatting	Formatting Examples	PDF	source	
polynom::polynomials	Vignette Title	PDF	source	R code
Rgraphviz::Rgraphviz	How To Plot A Graph Using Rgraphviz	PDF	source	R code
Rgraphviz::newRgraphvizInterface	A New Interface to Plot Graphs	PDF	source	R code

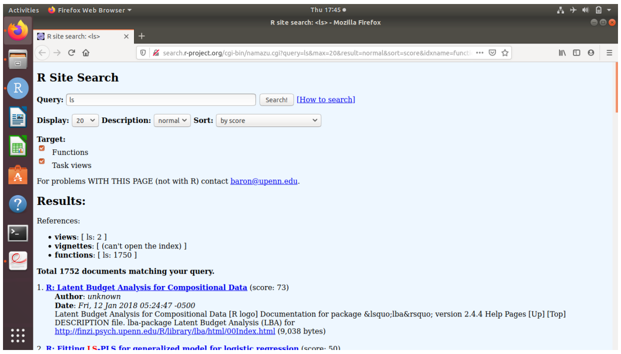
Figure 2.2: `??` function in RStudio.

The right bottom pane has almost all information related to `ls`.

This is rather more useful compared to aforementioned method. However, `help` retrieves or shows inforamton in fraction of seconds and it will be mostly rather succinct rather than lucid. This method is useful in case if it is required to be read for vignettes. Vignettes is a kind of R object helps in getting deeper idea about the question in hand.

2.3 RSiteSearch

This command shows information of search string in browser outside RStudio, default web browser of OS. Figure 2.3 shows the details of R command `ls`.


Activities
Firefox Web Browser
Thu 17:45
R site search: <ls> - Mozilla Firefox
R site search: <ls>
search.r-project.org/cgi-bin/namazu.cgi?query=ls&max=20&result=normal&sort=score&idxname=functi

R Site Search

Query: ls Search! [How to search]

Display: 20 **Description:** normal **Sort:** by score

Target:
✓ Functions
✓ Task views

For problems WITH THIS PAGE (not with R) contact baron@upenn.edu.

Results:

References:

- **views**: [ls: 2]
- **vignettes**: [(can't open the index)]
- **functions**: [ls: 1750]

Total 1752 documents matching your query.

1. **R: Latent Budget Analysis for Compositional Data** (score: 73)
   Author: *unknown*
   Date: *Fri, 12 Jan 2018 05:24:47 -0500*
   Latent Budget Analysis for Compositional Data [R logo] Documentation for package &lsquo;lba&rsquo; version 2.4.4 Help Pages [Up] [Top] DESCRIPTION file. lba-package Latent Budget Analysis (LBA) for
   http://finzi.psych.upenn.edu/R/library/lba/html/00Index.html (9,038 bytes)

2. **R: Fitting LS-PLS for generalized model for logistic regression** (score: 50)

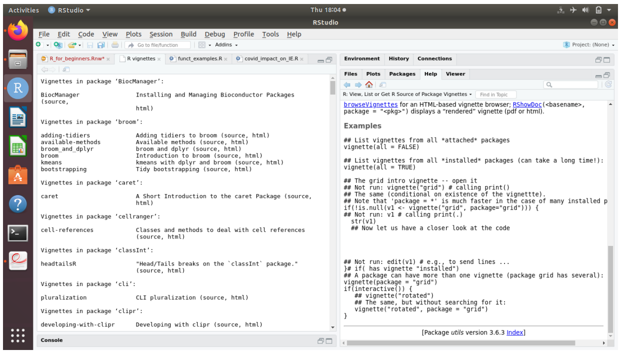
Figure 2.3: Results of `RSiteSearch` command.

Try for few other comamnds. R Studio must be able to show the details in external browser. [24]

2.3.1 Vignettes

The command `vignette` is useful to view, list or get R Source of Package Vignettes. Vignettes are highly extended documentations which includes description, code, examples, use cases and many more. Vignettes are highly useful for people who would like to use R statements rather more efficiently while working with practical data sets. Packages, functions are only for knowledge. Some times, we may actually needs certain information as how those things work in practice. Vignettes are really wounderful entities which provide practical information related to R procedures. Vignettes are associated with packages. To browse vignettes R needs to load a package of concern. The statement is very simple. Following statement in R Console could bring Figure 2.4 in RStudio.

```
> vignette(all = TRUE)
```


Activities
RStudio
Thu 18:04
RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
R_for_beginners.Rnw*
R vignettes
funct_examples.R
covid_impact_on_IE.R
Environment History Connections
Files Plots Packages Help Viewer
R: View, List or Get R Source of Package Vignettes
Find in Topic
Console

```
Vignettes in package ‘BiocManager’:

BiocManager                 Installing and Managing Bioconductor Packages
(source,
                            html)

Vignettes in package ‘broom’:

adding-tidiers              Adding tidiers to broom (source, html)
available-methods           Available methods (source, html)
broom_and_dplyr             broom and dplyr (source, html)
broom                       Introduction to broom (source, html)
kmeans                      kmeans with dplyr and broom (source, html)
bootstrapping               Tidy bootstrapping (source, html)

Vignettes in package ‘caret’:

caret                       A Short Introduction to the caret Package (source,
                            html)

Vignettes in package ‘cellranger’:

cell-references             Classes and methods to deal with cell references
                            (source, html)

Vignettes in package ‘classInt’:

headtailsR                  "Head/Tails breaks on the `classInt` package."
                            (source, html)

Vignettes in package ‘cli’:

pluralization               CLI pluralization (source, html)

Vignettes in package ‘clipr’:

developing-with-clipr       Developing with clipr (source, html)
```

browseVignettes for an HTML-based vignette browser; RShowDoc(<basename>, package = "<pkg>") displays a “rendered” vignette (pdf or html).

Examples

```
## List vignettes from all *attached* packages
vignette(all = FALSE)

## List vignettes from all *installed* packages (can take a long time!):
vignette(all = TRUE)

## The grid intro vignette -- open it
## Not run: vignette("grid") # calling print()
## The same (conditional on existence of the vignettte).
## Note that 'package = *' is much faster in the case of many installed p
if(!is.null(v1 <- vignette("grid", package="grid"))) {
## Not run: v1 # calling print(.)
  str(v1)
  ## Now let us have a closer look at the code



## Not run: edit(v1) # e.g., to send lines ...
}# if( has vignette "installed")
## A package can have more than one vignette (package grid has several):
vignette(package = "grid")
if(interactive()) {
   ## vignette("rotated")
   ## The same, but without searching for it:
   vignette("rotated", package = "grid")
}
```

[Package *utils* version 3.6.3 Index]

Figure 2.4: Results for `vignette` command in RStudio.

You can use `help("vignette")` in R Console for user manual for this command.

2.3.2 Searching for functions

All the above functions are only to know about a particular function of any aspect of programming. However, at times a user, especially a beginner, would like to know what are all different ways of doing a particular task in R. Till now I discussed as how to know about function, such as the one, `ls`. I will be using the same function but to retrieve information about any other functions as per need. I mean, suppose, if someone wants to know that if there exists any functions to compute arithmetic mean, how to know about it? There is a procedure. First I will show you some certain useful command called `sessionInfo()`. R creates a user session, each time when R is opened by the user. This function is highly useful to know about all that information associated to user session.

```
> sessionInfo()
```

```
R version 4.0.3 (2020-10-10)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 19041)
Matrix products: default
locale:
[1] LC_COLLATE=English_India.1252  LC_CTYPE=English_India.1252
[3] LC_MONETARY=English_India.1252 LC_NUMERIC=C
[5] LC_TIME=English_India.1252
attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base
loaded via a namespace (and not attached):
[1] compiler_4.0.3 tools_4.0.3
```

```
> ls()
```

```
character(0)
```

If you observe carefully, there are few packages which are attached to the user session. They are listed under section called *attached base packages*. These are called base packs.

```
> length(sessionInfo())
```

```
[1] 12
```

```
> sessionInfo()[6]
```

```
$basePkgs
[1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"
[7] "base"
```

Whenever, this function, i.e., `ls` is used in R Console, it gets all those, user and package level, objects in R session. Right now I don't have a single object. The output `character(0)` in the above output indicates that there is not any object associated with the present user session. Let me create a couple of objects and then execute the same function.

```
> a = 1; b = 2
> ls()
```

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

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

```
character(0)
```

You see! The command `ls` lists `a, b` after they being defined in the session. Now, just like the above task, it is also possible to list out objects in any given entity of R. A package is a valid R entity. In the session information, we found that there are few packages attached with the session. Right? Now let me pick up one of the packagres, say *stats* and see what are all different functions available in it.

```
> pcks <- ls("package:stats")
> typeof(pcks)
```

```
[1] "character"
```

```
> pcks[1:6]
```

```
[1] "acf"        "acf2AR"     "add.scope"  "add1"        "addmargins"
[6] "aggregate"
```

The function `ls` can be useful to know functions inside any given pacakge, but it should be attached to the user session. To know functions inside any package use `ls("package:package_name")`. **package_name** is the name of the package. The object `pcks` is just a user object in which all the output generated by the statement `ls("package:stats")` do exists. It is possible to know or drill further using the same objects in R. The function `typeof` is useful to known about the type of any object in R session. In this case the object `pcks` is character type. So, it is possible to index the elements inside this object. Eventually, the command `pcks[1:6]` retrieves the very first 6 elements in this object.

Now, suppose I am interested in searching for a function which will assist me calculating arithmetic mean. How to do that? One simple way is to go online, by using any search engine like, Google and search for "arithmetic mean in R". The browser will fetch you tons and tons of links. However, it is also possible to search for functions inside R Console.

```
> pcks <- ls("package:base")
> gr <- grep("mean", pcks)
> gr
```

```
[1] 713 714 715 716 717 718
```

In R, the function `grep` is useful to search for required information in any given object. In the above code chunk, the function `grep` shows numbers, which are indexes of those words in which the search word, i.e., "mean", might exists. How do we know that?

```
> pcks[c(gr)]
```

```
[1] "mean"          "mean.Date"     "mean.default"  "mean.difftime"
[5] "mean.POSIXct"  "mean.POSIXlt"
```

You see the very first function is the required function. I just used the technique *indexing* in R.

```
> mean(1:10)
```

```
[1] 5.5
```

It works!

2.4 Data types

Above section has shown various ways to search for information There are different types of data available for R programmers. A programmer need to know different types of data available for programming. Each language has a specific way to deal with data. Languages differ in the way data is being declared and used for data processing. However, the concept of data is same. I mean, every language has to have some certain ways of using different types of data for programming. So, the language designer need to provide methods to deal with different types of data.

Coming to data, as a word, it means *anything which represents real time phenomena.*. The word data is plural and *datum* is used for its singular form. Data can be collection of characters viz., symbols, numbers, and alphabets. Data can also be collection of images, videos, icons and a others. Whatever, may be the type of data, but it will have certain format. There are two very widely used econding formats namely, Unicode Transformation Format (UTF) and American Standard Code for Information Interchange (ASCII) among many other types. [25] A user or programmer of a language need not known all these details. The language can take care of data encoding and it is also part of language design. A user of programming language need to know what are the different types of data that a given language can support *ala*, numbers, alphabets, special characters etc.

2.4.1 Data type vs data structure

A data type is the most basic form of data. It is this through which the compiler gets to know the form or the type of information that will be used throughout the code. So basically data type is a type of information transmitted between the programmer and the compiler/interpreter where the programmer informs the compiler/interpreter about what type of data is to be stored and also tells how much space it requires in the memory.

A data structure is a colle ction of different forms and different types of data that has a set of specific operations that can be performed. It is a collection of data types. It is a way of organizing the items in terms of memory, and also the way of accessing each item through some defined logic. Some examples of data structures are *stacks, queues, linked lists, binary tree* and many more.

Data structures perform some special operations only like insertion, deletion and traversal. For example, you have to store data for many employees where each employee has his name, employee id and a mobile number. So this kind of data requires complex data management, which means it requires data structure comprised of multiple primitive data types. So data structures are one of the most important aspects when implementing coding concepts in real world applications.

R has very wide variety of data types and structures. R is primarily used for data science and analytics and mostly in academic circles. R is not fully used in development circles. This is why, R is not that strong as its companion languages like Python and Java as far as data structures are concerned. I will be describing as how to deal with different types of data in forthcoming sections. However, let me make rather very little description to stacks and queues in R.

Stacks & queues

It is not possible to deal stacks and queues directly in R just like Python. We need certain package called *dequer*. You can pronounce as "decker". [26]

A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called called *push* and *pop*. In python a stack can be managed using base type *list* as shown below.

```
>>> stack = []
>>> stack.append('hi')
>>> stack.append('this is')
>>> stack.append('a stack')
>>> len(stack)
3
>>> stack.pop()
'a stack'
>>> len(stack)
2
>>> stack.pop()
'this is'
>>> len(stack)
1
>>> stack.pop()
'hi'
>>> len(stack)
0
```

I just created a stack using a base data type called *list* in Python. The object `stack` updated with few words. In python methods like `append` and `pop` are useful to *push* and *pop* data. We can also push or pop elements using loops.

```
>>> s = []
>>> for i in range(3):
  s.append(i)

>>> print(s)
[0, 1, 2]

>>> for i in range(3):
  s.pop()
2
1
0
>>> len(s)
0
```

We have not discussed about loops till now. Loops are explained in one of the forthcoming chapters (Chapter **??**). For now just ignore usage of loops and just concentrate on results. Let us see how to make or destroy stakcs in R.

```
> library(dequer)
> s = stack()
> for (i in 1:3){
+   push(s, i)
+ }
> print(s)
```

```
A stack with 3 elements
```

```
> str(s)
```

```
stack of 3
 $ : int 3
 $ : int 2
 $ : int 1
```

Now lets see how to manage *queue* in R.

```
> q <- queue()
> for (i in 1:3){
+   pushback(q, i)
+ }
> str(q)
```

```
queue of 3
 $ : int 1
 $ : int 2
 $ : int 3
```

```
> for (i in 1:3){
+   pop(q)
+ }
> str(q)
```

```
 queue()
```

As a beginner, you may be worried as why we are interested in this skill. These type of advanced methods or data structures are quite useful while developing software applications. Few fields such as in real time data analytics, the data need to be updated as by time may be in seconds or minutes. Objects need to be appended or removed, with data points, based on criteria (say information or noise). Imagine that you got to manage some certain real time sensor based data. What will you do? You have to depend on data structures such as stacks and queues to ensure efficiency.

These type of data structures are not quite useful in R. At least, I never come across any necessity of such data structures in my experience. We will try to use a particulr data type called *list* for advanced operations such as appending or removing data elements from R objects. Before we talk about data types, it is better to learn few commands such as `typeof` and `mode`. These commands are highly useful to know about the type of data as and when used in the programs. Basically, as per numerical analysis, there are two types of data.

1. numeric
2. non-numeric

```
> a = 1; b = "a"
> typeof(a)

[1] "double"

> typeof(b)

[1] "character"

> mode(a)

[1] "numeric"

> mode(b)

[1] "character"
```

These methods i.e., *type* and *mode* are different. Numbers are not unique. The object *a* is *double* type and *numeric* mode, whereas the object *b* is character in both type and mode.

```
> f = 1.5
> typeof(f)

[1] "double"
```

```
> mode(f)
```

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

Even decimal values belongs to *double* type. What is this double? Use `help("double")`. You will know more information about this type. Generally, in mathematics, *integer* is any value which is not float (i.e., decimal). However, in computer science, the difference between integer and decimal is a matter of bytes. So, you may do as shown below, in case if you are interested in knowing about various sizes of data types in R.

```
> noquote(unlist(format(.Machine)))
```

```
              double.eps             double.neg.eps                double.xmin
            2.220446e-16               1.110223e-16              2.225074e-308
             double.xmax                double.base              double.digits
           1.797693e+308                          2                         53
         double.rounding               double.guard          double.ulp.digits
                       5                          0                        -52
   double.neg.ulp.digits            double.exponent             double.min.exp
                     -53                         11                      -1022
          double.max.exp                integer.max                sizeof.long
                    1024                 2147483647                          4
         sizeof.longlong          sizeof.longdouble             sizeof.pointer
                       8                         16                          8
          longdouble.eps         longdouble.neg.eps          longdouble.digits
            1.084202e-19               5.421011e-20                         64
     longdouble.rounding           longdouble.guard      longdouble.ulp.digits
                       5                          0                        -63
longdouble.neg.ulp.digits       longdouble.exponent         longdouble.min.exp
                     -64                         15                     -16382
      longdouble.max.exp
                   16384
```

The command `.Machine` is is a variable holding information on the numerical characteristics of the machine R is running on, such as the largest double or integer and the machine's precision. It is possible to know about a particular data type by using the same command with its attributes.

```
> c(.Machine$double.xmin, .Machine$double.xmax)
```

```
[1] 2.225074e-308 1.797693e+308
```

Oh! that's not a simple range. So I have possibility to use almost from as less as $2.2250738585072e{-}308$ to as big as $1.79769313486232e+308$ using data type *double*. There are few other attributes for this cammand `.Machine`. You can try checking with other data types such as integer, long etc. You know, the maximum value for double is $(2 - 1/2^{52}) * 2^{1023}$, you can check in R Console.

```
> (2 - 1/2^52) * 2^1023
```

```
[1] 1.797693e+308
```

This expression is equalent to $2^{1024} - 2^{971}$ but it is not possible to compute this expression in R; you can check R shows you Inf, which means infinity. Because, 1024 in power is unreasonable value for computer. It needs to be at least $2^{1024} - 2$. Each byte is 8 bits and possible range for an integer is 0 to 255 but not 1 to 256.

```
> (2^8-1)
```

```
[1] 255
```

But for integer the formula is going to be like this.

```
> (2^(4*8))/2 - 1
```

```
[1] 2147483647
```

This represents a formula $2^{(n*m)} - 1$ where n stands for number of bits and m stands for bytes. For integer it takes 8 bytes and each byte is equal to 4 bits. Let us check this with `.Machine` command.

```
> .Machine$integer.max
```

```
[1] 2147483647
```

Well, if you don't understand. Just don' care. You are not going to loose anything from your brain. Finally, what is that I am talking here? You know computers always understand user data in bits and bytes. In R there is a definit range for different types of data namely integer, double, long etc. You may try as below to understand bits and bytes for user data.

```
> rawToBits(as.raw(1))
```

```
[1] 01 00 00 00 00 00 00 00
```

```
> rawToBits(as.raw(2))
```

```
[1] 00 01 00 00 00 00 00 00
```

```
> rawToBits(as.raw(3))
```

```
[1] 01 01 00 00 00 00 00 00
```

You see, the number 1 shifs from left to right. For instance, the value 2 can be calculated as $0 * 2^0 + 1 * 2^1$. The value 3 can be calculated as $1*2^0+1*2^1$, which obviously 3. So on and so forth ... For instance, if I want to know how much memory is being required by R for a number like 2, I can do as shown below.

```
> object.size(2)
```

```
56 bytes
```

```
> 2^6 - 8
```

```
[1] 56
```

One byte (2^3) less to 64 bits (2^6) bits. I have 64 bit processor in my computer. $2^(2*3)$ is equal to 64 bits and $2^(3)$ one bit. How to know memory size for character type?

```
> object.size("a")
```

```
112 bytes
```

```
> (56*8)/4
```

```
[1] 112
```

This is one-forth of 56 bytes size i.e., I again iterate; don't care! if you don't understand this math. We need to understand as how R process data. R has methods to work with quite a few data types. For the sake of simplicity we can discuss the following methods.

1. character
2. numeric (real or decimal)
3. integer
4. logical
5. complex

We need to check the four properties i.e., *type, mode, class* and *attributes*, whenever we deal with data. These methods really helps a lot while processing data.

```
> b = "a"; a = 2
> typeof(a)
```

```
[1] "double"
```

```
> is.numeric(a)
```

```
[1] TRUE
```

```
> is.integer(a)
```

```
[1] FALSE
```

```
> aint <- as.integer(a)
> is.integer(aint)
```

```
[1] TRUE
```

```
> a == aint
```

```
[1] TRUE
```

```
> c <- complex(real = 1, imaginary = 2)
> is.complex(c)
```

```
[1] TRUE
```

I will describe rest of the methods i.e., *mode, class* and *attributes* in the forthcoming sections.

2.5 Vectors

In data processing, the primary input for data analytics is data distribution, which is a variable and it can be a collection of few elements. A data set is collection of data in rows and columns. Each row or column can be a data distribution also known as variable. The magnitude of the vector is the numeric value of elements in any given position and direction refers to position of the same element.

```
> vec <- 1:5
> is.vector(vec)
```

```
[1] TRUE
```

So, whenever we create a linear number series in R, it is simply as vector. The special symbol *:* is called colon operator. There is one more operator which helps in organizing data and it is called *c* operator.

```
> vec <- c(1, 2, 3, 4, 5)
> is.vector(vec)
```

```
[1] TRUE
```

However, one potential advantage of c operator is that it is possible to create a vector of variety of elements. Usually referred to heterogeneous collection.

```
> vec <- c(1, 2, "a", "b", 3:5)
> is.vector(vec)
```

```
[1] TRUE
```

It is possible to index vectors using numbers.

```
> length(vec)
```

```
[1] 7
```

```
> vec[1]; vec[6]
```

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

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

We will see how to convert this vector into other data type called "list" in later sections. One pretty thing about R is that the object `vec` can be used as stack, explained in Section 2.4.1. For intance, it is possible to append or remove elements from this object.

```
> vec
```

```
[1] "1" "2" "a" "b" "3" "4" "5"
```

```
> vec <- c(vec, 6)
> length(vec)
```

```
[1] 8
```

```
> vec[length(vec)]
```

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

```
> vec <- vec[-length(vec)]
> vec
```

```
[1] "1" "2" "a" "b" "3" "4" "5"
```

We simply reinstated the object `vec` to its original position. In fact, we can think of using this logic to create our own functions to operate stacks. In which case, it is possible to avoid installing and using depedencies like *dequer* like packages. We will try this in forthcoming section 3.5.5. [27]

2.6 Factors

A factor is any collection of elements which has direction but not magnitude. This means elements in side a factor has no numeric value but has valid positions. In R, a factor refers to any collection which has elements having no numeric value. In R, whenever we create a factor it is just a collection of characters.

```
> fac <- c("a", "b", "c", "d", "e")
> is.vector(fac)

[1] TRUE

> typeof(fac)

[1] "character"

> class(fac)

[1] "character"
```

The object `fac` is still a vector. I remind of the fact, every collection is a vector in R workspace. We can use the method `as.factor` to convert a vector into a factor.

```
> fact <- as.factor(fac)
> levels(fact)

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

Let me do more meaningful task in different fashion. I mean in R style.

```
> lets <- letters[1:2]
> vect <- sample(lets, 10, replace = TRUE)
> fac <- factor(vect, levels=c("a", "b"))
> fac
```

```
 [1] b a b a b b b a b a
Levels: a b
```

I just used a base function `letters` to create an object named `lets` in which there are two letters *a* and *b*. The object `vect` is just a collection of letters, in the object `lets`, for a length of 10 using another function known as `sample`.

This is how vector based data can be converted into factors in R. The factor refers to any data variable in which there are characters. In data analytics, several times data analysts encounter situations where there will be plenty of *non-numerical yet categorical data which is simply called as factors.* Especially in social sciences, few data variables such as *gender, education, marital status* are measured using levels. For instance, let us see as how to simulated a gender variable with two levels i.e., male and female.

```
> genlevels <- c("male", "female")
> gender <- sample(genlevels, 5, replace = TRUE)
> gender <- factor(gender)
> levels(gender)
```

```
[1] "female" "male"
```

As mentioned before factors has no magnitude but has direction. We can study the property of direction using indexing.

```
> gender[c(1, 3, 5)]
```

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

The only way to add numerical value to this object i.e., *gender* is to obtain summaries of the variable using a base function called `table`.

```
> gentab <- table(gender)
> gentab
```

```
gender
female   male
     4      1
```

```
> gentab[1]/2
```

```
female
     2
```

The base function `table` is again just a function somebody written for convenience. We will try to develop our own function in Section 4.4.2 using some certain logic. It is possible to make summaries by sebsetting factors just like indexing vectors. For instance, it is possible to seperate all the elements which are either male and or female as by category. However, the way of working with these subsets is not exactly as that of vectors. It needs a binary operator called *equal to* (==) for this task.

```
> gender == "male"

[1] FALSE FALSE FALSE FALSE  TRUE

> gender == "female"

[1]  TRUE  TRUE  TRUE  TRUE FALSE
```

It is possible to make summaries using this binary operator.

```
> sum(gender == "male")

[1] 1

> sum(gender == "female")

[1] 4
```

This way it is possible to simulate or create any type of data variable in R. Refer Section 5.1.2 for more details.

2.7 Lists

List is another type of data which is also belongs to collections just like matrices, frames, arrays etc. A list is a collection of other data types. Lists act as containers. This means a list can contain numeric, character, vector, factor and many other types of data. Lists are used whenever there is a need for containing multiple types of data. Lists are sometimes called generic vectors, because the elements of a list can by of any type of R object, even lists containing further lists. This property makes them fundamentally different from atomic vectors. A list is a special type of vector. Each element can be a different type.

In R a list can be created using `list` or coerce other objects using `as.list`. An empty list of the required length can be created using `vector`. Use `help("list")` in R Console to know more about lists. Let us create a list of a vector and a factor.

```
> lis <- list()
> lis$perception <- sample(1:5, 5, replace = TRUE)
> lis$satisfaction <- sample(c("yes", "no"), 5, replace = TRUE)
> print(lis)

$perception
[1] 3 5 2 5 5
$satisfaction
[1] "no"  "no"  "no"  "no"  "yes"
```

The above list object i.e., `lis` is called named list. These type of lists are very much useful while createing or playing with data matrices or frames. It is possible to index lists but by using double square brackets.

```
> lis[[1]]

[1] 3 5 2 5 5

> lis[[2]]

[1] "no"  "no"  "no"  "no"  "yes"
```

As a matter of fact it is even possible to carry out any kind of calculations based on type of the data a list is holding.

```
> c(max(lis[[1]]), mean(lis[[1]]), min(lis[[1]]))

[1] 5 4 2

> table(lis[[2]])

 no yes
  4   1
```

There is also a way to create dummy lists and make them usable in code using `vector` function.

```
> lis1 <- vector("list", 2)
> lis1[[1]] <- 1:5
> lis1[[2]] <- letters[1:5]
> lis1

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

[[2]]
[1] "a" "b" "c" "d" "e"
```

These type of lists are known as unnamed lists and these type of procedures are highly useful while working with conditions and loops. One important thing about lists is that the lists has no separate class but data matrix and data frame has.

```
> typeof(lis)

[1] "list"

> mode(lis)

[1] "list"

> class(lis)

[1] "list"
```

Lists return same information for above all attributes. Lists are highly dynamic while workign with user defined functions (UDF). Lists can be very helpful while returning results or output. We will learn more about using lists in functions in forthcoming chapters.

2.8 Data matrix

A matrix is a collection of data arranged in rows and columns. That is why matrices has order and usually represented by $r \times c$, where r stands for rows and c stands for columns. Matrices are highly useful to make collections and also serves as containers.

In R matrices can be created with the help of a function called `matrix`. However, this function has few arguments which needs our attention. The output of matrix function in R depends on the way argument, `byrow`, is used. Use `help("matrix")` in R Console to know more.

```
> mt <- matrix()
> typeof(mt)
```

```
[1] "logical"
```

```
> class(mt)
```

```
[1] "matrix" "array"
```

You see! Matrix has different information for both `typeof` and `class` attributes. Same is true even for data frame. That is why data matrix and data frames are special objects in R.

```
> matrix(1:16, 4, 4, byrow = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
```

```
> matrix(1:16, 4, 4, byrow = FALSE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

This much code is sufficient in R to create matrix. However, a user can use different other arguments to customize the output. Columns in matrix are referred to dimensions. We can assign names to these dimensions using `dimnames` argument. The value to this argument must be list otherwise R will do all murky job.

```
> datf <- matrix(1:16, 4, 4,
+                  byrow = TRUE,
+                  dimnames = list(1:4, c("V1", "V2", "V3", "V4")))
> class(datf)
```

```
[1] "matrix" "array"
```

```
> datf
```

```
  V1 V2 V3 V4
1  1  2  3  4
2  5  6  7  8
3  9 10 11 12
4 13 14 15 16
```

You see? I created a data matrix which looks like a data frame. That is the beauty of working with R. If see the class attribute it is still a matrix. I will show you how to convert this matrix in to a data frame in the forthcoming section.

You can use any type of R logic for `dimnames`. For instance, if you have 1000 rows and 200 columns, writing all the names for rows and columns really give pain in the brain. It is better to use little logic to create list of row names and column names.

```
> rns <- vector(); cns <- vector()
> for (i in 1:4){
+   rns[i] <- paste("r", i, sep = "")
+   cns[i] <- paste("v", i, sep = "")
+ }
> print(rns)
```

```
[1] "r1" "r2" "r3" "r4"
```

```
> print(cns)
```

```
[1] "v1" "v2" "v3" "v4"
```

```
> matrix(1:16, 4, 4, byrow = TRUE,
+        dimnames = list(rns, cns))
```

```
   v1 v2 v3 v4
r1  1  2  3  4
r2  5  6  7  8
r3  9 10 11 12
r4 13 14 15 16
```

2.8.1 Matrix operations

Matrix is a valid data collection which is organized in rows and columns as such it is possible to perform any type of operations related to matrix algebra. Let us see few arithmetic operations using matrix A and B.

```
> A <- matrix(1:16, 4, 4)
> B <- matrix(1:16, 4, 4)
> A + B
```

```
     [,1] [,2] [,3] [,4]
[1,]    2   10   18   26
[2,]    4   12   20   28
[3,]    6   14   22   30
[4,]    8   16   24   32
```

```
> A - B
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
```

```
> A * B
```

```
     [,1] [,2] [,3] [,4]
[1,]    1   25   81  169
[2,]    4   36  100  196
[3,]    9   49  121  225
[4,]   16   64  144  256
```

```
> A / B
```

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

There are interesting results. The subtraction yields a matrix known as zeros matrix and division yields a matrix known as ones matrix. [28] These matrices are highly significant in statistics community. For instance, zero matrices are used in few bivariate and multivariate statistics to evaluate model assumptions. [29] Ones matrix is used in few applications related to polynomial regression and graph theory. [30]

There are few packages in R to create these matrices. For instance, the package *optimbase* has a method called `ones` which creates ones matrix and there is a function `zeros` in a package called *phonTools*. There is another packages called *Matrix* which has quite a few methods and also algorithms to study matrix algebra. [31] It is also possible to create own functions for these methods. Visit Section 3.5.1 in one of the forthcoming chapters for methods to create zeros and ones matrices.

2.8.2 Identity matrix

Identity matrix is a special case in matrix algebra. This matrix has few interesting characteristics. The identity matrix, or sometimes ambiguously called a unit matrix, of size n is the $n \times n$ square matrix with ones on the main diagonal and zeros elsewhere. More technically, any matrix A $(n \times n)$ satisfies the below property to call another matrix I $(n \times n)$ as identity matrix.

$$I_m A = A I_n = A. \tag{2.1}$$

In R there is function called `diag` to create identity matrices.

```
> diag(4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

Let us test the definition of identity matrix.

```
> dm <- diag(4)
> A <- matrix(round(rnorm(16), 2), 4, 4)
> A == dm%*%A%*%dm
```

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

This function `diag` is also useful to make diagonal elements of any given matrix into a vector of required values. In statistics, there is a matrix called correlation matrix. The correlation matrix has 1s' in diagonal position and other values in off-diagonal positions.

Sometimes, I quickly make cor matrices without being using actual data when I am developing code for multivariate analytics.

```
> A <- matrix(round(runif(16), 2), 4, 4)
> diag(A) <- 1
> A
```

```
     [,1] [,2] [,3] [,4]
[1,] 1.00 0.92 0.85 0.86
[2,] 0.35 1.00 0.87 0.64
[3,] 0.24 0.05 1.00 0.83
[4,] 0.74 0.03 0.63 1.00
```

Beautiful the matrix A looks like correlation matrix. All the diagonal values are one and off-diagonal values are 0 to 1 which is the probability space for coefficient r. I used base function called `runif` to simulate values between 0 to 1.

2.8.3 Matrix multiplications

There are number of methods for matrix multiplication. The multiplication that we did just before is called element-wise multiplication also known as dot product. This can be achieved by simply using multiplication symbol (*). There is another product which is called just as *matrix multiplication*. Such multiplication can be achieved using a collection of special symbol $\% * \%$.

```
> mat1 <- matrix(1:9, 3, 3)
> mat2 <- matrix(1:9, 3, 3)
> mat1
```

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

```
> mat2
```

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

```
> mat1 %*% mat2

     [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
```

This is called inner product. The cross product for first row of matrix *mat1* and first column of matrix *mat2* is as follows.

```
> mat1[1, ]; mat2[, 1]

[1] 1 4 7

[1] 1 2 3

> crossprod(mat1[1, ], mat2[, 1])

     [,1]
[1,]   30
```

So what is happening here? Whenever, we perform inner product or cross product, the function $\% * \%$ just executes inner loops. Look at Section 4.4.3 for more detailed understanding. The function `crossprod` just computes a sum of products. Visit Section 3.5.1 for more detials.

2.9 Data Frame

Data frame is another collection available for holding heterogeneous data variables. Data frame more or less arrangement of rows and columns with multifarious data. Each row or column can be a vector and a collection of such vectors must be equal in length. One fundamental differece between data matrix and data frame is that a data matrix mostly has similar type of data eg., either numerical or character, whereas data frame can hold both numeric and character based data bound together.

It is not extraneous if I say that R is a bit unique compared to other programming languages as far as data analytics is concerned. This is because the data type "data frame" was introduced by R community to the world of data analytics. There is some certain and similar function in Python community introduced by *pandas* developers but by then R was fully operational in the analytics field. [32] [33] R has a unique function called `data.frame` with its helper functions such as `is.data.frame`, `as.data.frame` and `cbind.data.frame`. Let us see few examples as how to manage data frames in R. I described as how to deal with vectors and factors in previous sections. I will try to use these methods to manage data frames.

```
> vec1 <- 1:10
> fac1 <- letters[1:10]
> vec2 <- round(rnorm(10), 2)
> dataf <- data.frame(vec1, fac1, vec2)
> dataf

   vec1 fac1  vec2
1     1    a -0.12
2     2    b  0.03
3     3    c  1.18
4     4    d  0.98
5     5    e  0.93
6     6    f -0.73
7     7    g  1.10
8     8    h -0.23
9     9    i  0.76
10   10    j -0.35
```

It is that simple. As far as attributes are concerned, data frame is different for `typeof` and `class`. This means data frames has a special attribute for function `class`

```
> typeof(dataf)

[1] "list"

> class(dataf)

[1] "data.frame"
```

2.9.1 Indexing data frames

Indexing data frames is highly required skill, especially in data analytics community. Most of the anlayses were done using either by rows or by columns. So the user is expected to know as how to play with rows and columns of a data frame. For instance

```
> dim(dataf)
```

```
[1] 10  3
```

```
> dataf[1, 1]
```

```
[1] 1
```

```
> dataf[dim(dataf)[1], dim(dataf)[2]]
```

```
[1] -0.35
```

The characters `[` and `]` are used to index data points inside a data frame. The statment `dataf[1, 1]` yields the very first value which belongs to first row and first column, whereas the statement `dataf[dim(dataf)[1], dim(dataf)[2]]` yields the very last value in the data frame. It is also possible to manipulate data in any data frame using assignment operators.

```
> dataf[1, 1] <- NA
> head(dataf)
```

```
  vec1 fac1  vec2
1   NA    a -0.12
2    2    b  0.03
3    3    c  1.18
4    4    d  0.98
5    5    e  0.93
6    6    f -0.73
```

The value at the address $(1,1)$, in above data frame, is now changed into NA. Data frames are meant for analyses and this data structure is highly dynamic for such analyses.

```
> summary(dataf)
```

```
      vec1         fac1                 vec2
 Min.   : 2   Length:10           Min.   :-0.7300
 1st Qu.: 4   Class :character   1st Qu.:-0.2025
 Median : 6   Mode  :character   Median : 0.3950
 Mean   : 6                      Mean   : 0.3550
 3rd Qu.: 8                      3rd Qu.: 0.9675
 Max.   :10                      Max.   : 1.1800
 NA's   :1
```

The function **summary** is a base function and it is used to calculate or know summary statistics. It is also possible to work through rows as well as columns.

```
> dataf[1, 1] <- 1
> summary(dataf)
```

```
      vec1             fac1                vec2
 Min.   : 1.00   Length:10          Min.   :-0.7300
 1st Qu.: 3.25   Class :character   1st Qu.:-0.2025
 Median : 5.50   Mode  :character   Median : 0.3950
 Mean   : 5.50                      Mean   : 0.3550
 3rd Qu.: 7.75                      3rd Qu.: 0.9675
 Max.   :10.00                      Max.   : 1.1800
```

```
> apply(dataf, 2, is.numeric)
```

```
 vec1  fac1  vec2
FALSE FALSE FALSE
```

```
> apply(dataf, 2, function(x) sum(as.numeric(x)))
```

```
 vec1  fac1  vec2
55.00    NA  3.55
```

The function inside `apply` statement is an anonymous function which converts each vector in `dataf` into numeric type and sum up the values. It is even possible to plot the data using base `plot` function. Look at the figure 2.5.

R has wounderful methods to make visuals using base methods. This topic is covered rather sufficiently in the last Chapter with title *Data Visualizations*. Read R manual using `help("plot")` function. [34] [35] R supports highly advanced methods for ploting data such as 3D plots and animations. [36]

```
> layout(c(1, 2))
> plot(dataf[, 1], type="b"); plot(dataf[, 3], type="b");
```

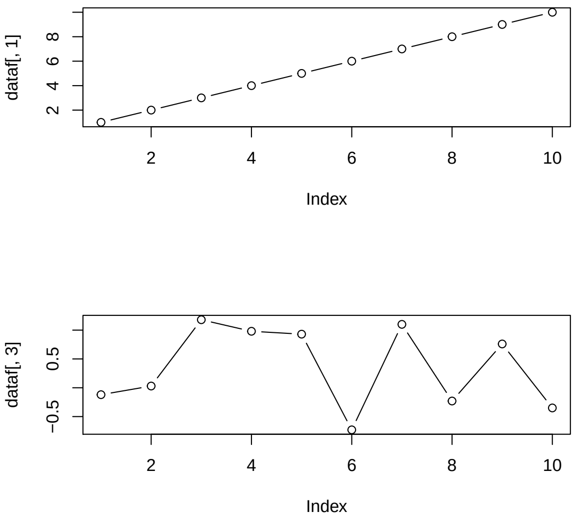


Figure 2.5: Base plot in R.

2.9.2 Data frame vs data matrix

There are methods to convert matrices into frames. This is another skill which is required to manage data frames in R. R is just not a language of statistics but research management system. R can fulfil almost all requirements of a researcher. Say from data entry to reports. R has a base function `scan` which is used to enter data from sheets manually. These manually entered data later can be converted into either vectors or factors and such data types will be collected as data matrices or frames. This topic is rather a lengthy one and comprises into a separate course or fit into volumes of books. I shall just explain as how to convert data matrix into a data frame thoroughly using few base functions in R. This method is mostly used in numerical simulations by software developers using

R.

```
> dmat <- matrix(round(rnorm(10), 2), 10, 3)
> dframe <- data.frame(dmat)
> head(dframe)
```

```
     X1    X2    X3
1 -0.30 -0.30 -0.30
2  1.41  1.41  1.41
3  1.48  1.48  1.48
4  0.17  0.17  0.17
5  0.56  0.56  0.56
6 -0.36 -0.36 -0.36
```

You see! We don't need to worry of setting up column names while converting matrices into frames. R can take care of that. In case if I wish to have different names then I can do as shown below.

```
> colnms <- paste("V", 1:3, sep = "")
> colnames(dframe) <- colnms
> head(dframe)
```

```
     V1    V2    V3
1 -0.30 -0.30 -0.30
2  1.41  1.41  1.41
3  1.48  1.48  1.48
4  0.17  0.17  0.17
5  0.56  0.56  0.56
6 -0.36 -0.36 -0.36
```

Notes

[20]Programming language theory. Available at https://en.wikipedia.org/wiki/Programming_language_theory

[21]Wrathematics (27 August 2011). "How Much of R Is Written in R"?. librestats. Archived from the original on 12 June 2018. Retrieved 7 August 2018.

[22]There is an write-up "How Much of R Is Written in R"?, dated August 27th, 2011 at 12:44 am. The blog has wounderful methods for analyzing R code using BASH together with a couple of tools. Visit http://librestats.com/2011/08/27/how-much-of-r-is-written-in-r/ for more details. The blog shows that only 22% of R is being used to develop R. More or less 50% of R source code is developed in C followed by FORTRAN ($\approx$ 25%)

[23]Wickham, H. OOP systems. Available at https://adv-r.hadley.nz/oo.html

[24]RStudio has an internal browser. Whenever, user request information, the internal browser opens and shows details. This internal browswer is known as "Viewer Pane". Ian Pylvainen written a nice blog as how to use this browser for variaous needs of web development.

[25]Wikipedia. Character encoding. https://en.wikipedia.org/wiki/Character_encoding

[26]Schmidt D (2017). "dequer: Stacks, Queues, and Deques for R." R package version 2.0-1, https://cran.r-project.org/package=dequer.

[27]That is why, it is said, "every thing is a function in R". It is possible to write code for any type of task by writng UDFs.

[28]Ones matrix is also called as unit matrix but the name unit matrix is also used to refer identity matrix in few statistics spheres.

[29]Read about zero matrix at https://en.wikipedia.org/wiki/Zero_matrix. It is highly interesting to know few mathematical properties of zero matrix.

[30]Read about importance of ones matrix in statistica at https://en.wikipedia.org/wiki/Matrix_of_ones

[31]Visit https://cran.r-project.org/web/packages/Matrix/ for more details.

[32]First version of R was released on 29th February, 2000 but Pandas was released in 11th January, 2008.

[33]The source code of R was available from 23 April, 1997 from CRAN, albeit of the fact that the first version was released in February, 2000. Visit https://cran.r-project.org/src/base/R-0/ for old sources.

[34]There are lot of online sources as how to plot data using R. Frank McCown wrote certain basic methods at https://sites.harding.edu/fmccown/r/. It may be nice place to learn basics of ploting data.

[35]There is one webiste known as https://www.r-graph-gallery.com/index.html where there are enormous examples on plotting data in R.

[36]There is a package called *rgl* which is used for 3D Visualization Using OpenGL. There is another package called *animations* which is used for playing animations in R.

Chapter 3

Functions

Functions serves as scaffolding to R. R belongs to the legion of functional programming languages. *Everything that happens in R is function call.* It is easy to say so but requires a bit of pulp in brain to understand the same. R is not suitable for objective orientation in programming despite of the sources and few people working in those lines.[37] For that matter, R has few highly dynamic pathways to do object oriented programming compared to its competitors. However, R is still considered for functional programming in the community. Hadley, one of the celebrities, in R community too feel the same. He wrote as below in one of his books on R programming.

> "Generally in R, functional programming is much more important than object-oriented programming, because you typically solve complex problems by decomposing them into simple functions, not simple objects." [38]

3.1 Introduction

The goal of this text is to introduce functional programming using R. "Functions are a fundamental building block of R", says Hadely in his book *Advanced R.* [39] A function is any set of instructions which executes logic and performs certain task. Every function has certain goal behind it. Scientifically, a program is a collection of statements which commands computer to perform certain task. Computer allocates resources such as memory and processing while helping a program to achieve its goal. An intermediary program which helps a language to borrow resources while dealing with such instructions is called either a compiler or interpreter or assember. These entities helps a programming language in translating high level (human readable) instructions into low level (machine readable) instructions.A function is an object which will be compiled or interpreted for user requests. R uses interpreter, which means it is possible to execute line by line in the code unlike other languages such as VBA and JAVA.

In simplest words, a function is a code block or chunk written by a user to accomplish a particular task in hand. R has efficient mechanisms to define and execute functions. Whenever, a user defines a function, it is called User Defined Function (UDF). In fact, all those base functions available from R Base are actually, in a way, written by developers for easy execution of routines. In R, whenever a function is used from base it is called as either a built-in function or base function. A function written by users are known as user defined functions.

All R functions have three parts:

1. the `body()`, the code inside the function.
2. the `formals()`, the list of arguments which controls how you can call the function.
3. the `environment()`, the "map" of the location of the function's variables.

All the above components are exceptions to a particular type of functions called *primitive* functions. Primitive functions are only found in the base package, and they operate at a low level. Functions, such as `length, +, -, sum`, are primitive and they contain no R code. That is why, these functions don't exhibit above mentioned components.

```
> primfuncs <- ls("package:base")
> which(primfuncs == "length")
```

```
[1] 650
```

```
> primfuncs[644]
```

```
[1] "lapply"
```

```
> is.primitive(length)
```

```
[1] TRUE
```

The function `length` belongs to *base* package.

```
> length
```

```
function (x)  .Primitive("length")
```

This don't return any code. However, the function `mean` could give information regarding components of function, because it is not primitive function.

```
> mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x0000000012e215f0>
<environment: namespace:base>
```

The first line i.e., `function (x, ...)` represents first component which is formals, second statement i.e., `UseMethod("mean")` is body and rest of the output statements belongs to third component called environment. This is because, the function `mean` is written using class method. The response `UseMethod` belongs to the jargon of OOP in R. An R object is a data object which has a class attribute. A class attribute is a character vector giving the names of the classes from which the object inherits. If the object does not have a class attribute, it has an implicit class. Matrices and arrays have class "matrix" or "array" followed by the class of the underlying vector. Most vectors have class the result of mode(x), except that integer vectors have class c("integer", "numeric") and real vectors have class c("double", "numeric").

However, a user defined function may not return UseMethod unless it is defined through OOP.

```
> myfunc <- function(x){
+   print(x)
+ }
> myfunc("MK")

[1] "MK"

> body(myfunc)

{
    print(x)
}
```

In R a function is defined by using a statement called `function` and this statement is a function in itself. The item `x` is user argument also known as parameter by few. The logic must be enclosed inside curly braces. The function `myfunc` is an UDF and it returns logic for the `body()`. I will convert `myfunc` as method and assign a class to object `x`.

```
> myfunc <- function(x){
+   UseMethod("myfunc")
+ }
> myfunc.classOne <- function(x){
+   print(x)
+ }
> myfunc.classTwo <- function(x){
+   print(x)
+ }
> obj <- "MK"
> class(obj) <- c('classOne')
> myfunc(obj)

[1] "MK"
attr(,"class")
[1] "classOne"

> class(obj) <- c('classTwo')
> myfunc(obj)

[1] "MK"
attr(,"class")
[1] "classTwo"
```

```
> myfunc

function(x){
  UseMethod("myfunc")
}
<bytecode: 0x0000000019ae50e8>
```

I defined two classes namely `classOne` and `classTwo`. Now the argument `x` is an object i.e., `obj`. This object has a class. Moreover, it is an *attribute*. What a hell is this? Everything is reverse in R. If you have any knowledge or experience related to object oriented programming, I am sure you will loose the brain. In spite of all this confusion, there is logic in R way of dealing with OOP. What is happening here? As per the conventions, we first define class then methods and somewhere we deal with attributes when required. Most importantly, the object is known as instance. In R everything goes reverse. We first define method and then assign class to it and it is attribute. That is why we say, *everything is a function in R*! A method is a function, a class is a function. In the end even an object is a function by call.

3.2 Arguments

An argument is a value passed by the user to any given function. Arguments are names which later substitutes user values inside the body of a function. Functions can take any of the following arguments in R.

1. Formal arguments
2. Default arguments
3. Expressions or formulas

In R functions can have arguments and also sometimes not. It is absolutely possible to write functions with empty arguments.

```
> func <- function(){
+   print("This is empty function")
+ }
> func()
```

```
[1] "This is empty function"
```

We don't need to pass any arguments to function `func`. It is defined with empty arguments. Let me explain a function with all the three types of arguments.

```
> exFunc <- function(a, b, p = c, q = a+p){
+   c = a + b
+   return(q)
+ }
> exFunc(1, 2)
```

```
[1] 4
```

The above code is self explantory. The function `exFunc` has two user arguments, one default and one expression. If you observe carefully the argument *c* is computed inside the body of the function but R consider as valid argument calling it into farmals. Coming to q, it is an expression which depends on one user argument (a) and a formal called from the body. It is also possible to make open call for arguments using three dots (...) known as ellipses. This will enable R to wrap or unwrap arguments as by the need.

```
> dotsTest <- function(...){
+   print(...)
+ }
> dotsTest("hello! how do you do?")
```

```
[1] "hello! how do you do?"
```

```
> dotsTest(letters[1:5])
```

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

The function `dotsTest` just echos whatever written by the user. I will explain the utility of ellipses (dots) more objectively.

```
> testDots <- function(x, FUN = NA, ...){
+   out <- FUN(x, ...)
+   return(out)
+ }
> inFunc <- function(x, trim = FALSE, d = NA){
+   if(trim){
+     return(round(x+x, d))
+   } else {
+     return(x+x)
+   }
+ }
> testDots(10.221012, inFunc)

[1] 20.44202

> testDots(10.221012, inFunc, TRUE)

[1] NA

> testDots(10.221012, inFunc, TRUE, d = 2)

[1] 20.44
```

The function `testDots` requires two compulsory arguments namely `x` and `FUN` and optional arguments through *dots*. The argument `FUN` is, in deed, a user function. The user function i.e., `inFunc` has three arguments namely `x, trim` and `d`. The argument `x` is a variable whereas others are default. Above function `testDots` requires both arguments i.e., a value for `trim` and `d` to execute rounded value otherwise simply returns full value for sum.

3.3 Anonymous functions

Every function is an object in R. For that matter anything defined by the user is considered as object in its own form. Say, a number, a character, an expression or may be a function. Those functions which are defined by names are called *anonymous functions*. Writing anonymous functions in R is a pretty cool job. Any function which has no name is simpy an anonymous function unlike many other languages like C, Python and JAVA.

```
> (function(x) x^2)(2)

[1] 4

> (function(x) sin(x)^2 + pi/2)(1:5)

[1] 2.278870 2.397618 1.590711 2.143546 2.490332
```

Both functions in the above code block can be considered as anonimous for these functions were not given names. These type of methods are very handy while writing applications and developing projects. The other example can be:

```
> (function(x) plot(x, sin(x),  type = "b"))(1:10)
```

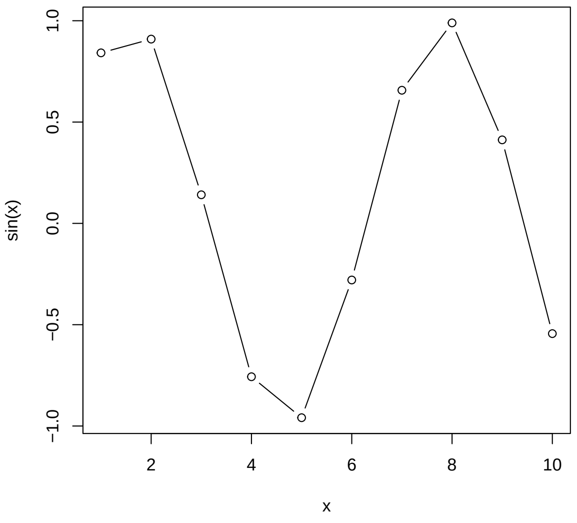


Figure 3.1: Sine Wave

Above code block has the procedure to plot sine curve using anonymous function. These type of curves can be plotted using anonymous functions very thoroughly, then and there, whenever they are required inside a project or application. Anonymous functions are instantaneous and useful whenever they are not required elsewhere in the application or project.

3.4 Nested Functions

R helps in defining functions inside a function. Necessity for nested functions arise whenever there exists a requirement for processing data using unique set of instructions again ana again. Nested functions are routines in which a function holds one or few other functions.

```
> mainFunc <- function(a, b){
+   c <- function(x) 2*x
+   d <- function(x) 3*x
+   e <- c(a) + d(b)
+   return(e)
+ }
> mainFunc(1, 2)
```

```
[1] 8
```

This is very small and insignificant example. The above function process *a, b* as defined by inner functions `c, d`. Let me show you rather more meaningful example.

> Suppose I am teaching a course and at times the total evaluation goes beyong 100. I would like to reduce the marks of the students to 100 for different totals. I may be able develop a function using *total* and *marks* as arguments.

```
> makeHundFrom <- function(total){
+   function(x){
+     x * (100/total)
+   }
+ }
```

I am going to simulate data using R base function known as `runif`, which is useful to create data for lower and upper limits. Use `help("runif")` for more details. This data is processed for two different totals i.e., 120 and 200.

```
> x <- abs(runif(5, 40, 120))
> x
```

```
[1] 55.52285 98.92112 85.70751 50.69477 87.05016
```

```
> reduceToHund <- makeHundFrom(120)
> reduceToHund(x)

[1] 46.26904 82.43426 71.42292 42.24564 72.54180
```

Above code block shows as how to reduce marks (x) form a grand total 120 to 100. The function `makeHundFrom` is a parent function and the function `reduceToHund` is child.

```
> x <- abs(runif(5, 40, 200))
> x

[1] 138.53173 100.48865  47.78772  68.35594  93.28560

> reduceToHund <- makeHundFrom(200)
> reduceToHund(x)

[1] 69.26586 50.24433 23.89386 34.17797 46.64280
```

Above code block shows as how to reduce marks (x) form a grand total 200 to 100. There are few values beyond 100 in input data x (assumed as marks) in both of the above code blocks. These values are standaredized to 100 using respective totals. What is happending here? The function `makeHundFrom` is defined by a variable (totals marks) for processing a vector called x which represents marks. Thse marks tend to vary for various totals say, 120, 150, 200 and so on. Whenever, the need arises for standardization (100), it is possible to define another abstract function using `makeHundFrom`. So, it is not required to write different functions for different totals. Very cool job. This type of activity in R is referred to ***closures***.

A closure is the one used as anonymous function which inturn used to create small functions that are not worth naming. The utility of such closures is to create functions written by other functions. *Closures get their name because they enclose the environment of the parent function and can access all its variables.* [40] This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work. [41]

3.5 Applications

Functions as building blocks can be used for various ways while processing data. As it was mentioned previously, everything is a function in R. These functions can be used very dynamically to manipulate or process data. In this section I shall explain few examples as how to use functions for various use cases such as processing vectors and matrices. Hoever, all this discussion is very limitted for the scope of R scales to nowhere. Writing few useful funtions is the starting point for software development. It is possible to develop packages and software applications using functions but that is beyond the scope of this text.

3.5.1 Vector manipulation

Vectors are basic types of data in R. Mostly the data, if it is either matrix or frame, will be converted into vectors and factors in R while processing. It is possible to write few functions to manipulate vectors. In Section 2.8.3 I mentioned about *sum of the product* during the discussion related to matrix multiplications. This statement i.e., *sum of the product* is not a simple phrase but highly influential in statistics. The entire logic in simple linear regression depends on a particular principle called *least squares*, which is nothing but sum of the product of squared values, as far as logic is concerned. Now let me try to make a UDF for sum of the products later I will use the same logic to make a function for sum of the squares.

```
> x <- 1:5
> y <- 2:6
> sumProd <- function(x, y){
+   out  <- sum(`*`(x, y))
+   return(out)
+ }
> sumProd(x, y)

[1] 70

> crossprod(x, y)
```

```
     [,1]
[1,]   70
```

So, it is possible to write a function equivalent to `crossprod` which is a base function in R. This means someboody wrote a function for *sum of the products* to make our life happy. Coming to the logic of `sumProd`, I just used infix operator "+" as a function to compute sum of products. Now let me show you how to create a UDF for *sum of the squares.* By definition sum of the squres is as follows:

$$\text{ESS} = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2 \tag{3.1}$$

In the above eqution $\hat{y}$ is known as fit values and $\bar{y}$ is known as mean of the vector y. This is a special case in regression analysis. There are different sums of squares in regression analysis such as Total Sum of Squres (TSS), Explained Sum of Squares (ESS), Residual Sum of Squares (RSS); So that

$$TSS = ESS + RSS \tag{3.2}$$

I may not be able to explain entire regression process here. That is beyond the scope of this text. I shall show you how to create a dummy function for two user arguments such as x, y.

```
> sumSquares <- function(x, y){
+   out  <- sum((`-`(x, y))^2)
+   return(out)
+ }
> sumSquares(x, y)
```

```
[1] 5
```

Imagine that x represents fit values and y represents a vector of means or simply a mean value for y. I again used an infix operator i.e., "-" as in *function call* to achieve this operation.

3.5.2 Random matrix

Functions can be used to manipulate matrices. In the previous chapter I provided rather significant discussion on matrices. I shall explain as how to use functions to process matrices. I can develop or write a function, in case, if I like to create a matrix with random elements.

```
> randMat <- function(m, n, isround = TRUE){
+     out <- matrix(rnorm(m*n), m, n)
+     return(out)
+ }
> randMat(4, 4)
```

```
           [,1]       [,2]         [,3]        [,4]
[1,]  0.3675223  0.8035818 -0.244267922  0.51846161
[2,] -1.4237235 -0.8206489  0.341485411 -0.02880838
[3,] -1.1077059 -1.3397326 -0.299563438 -1.42148452
[4,]  0.2088098  1.3283802  0.001154735 -0.73996263
```

The function `randMat` is useful to create an abstract matrix with all random elements. Two of the base functions i.e., `matrix, rnorm` were used in the body of the function to achive the purpose. This function is defined for two user arguments namely *m, n* which stands fro number of rows and number columns respectively. The other argument `isround` is a default arguments, with a default value TRUE), which is useful to round the resultant values or elements in the matrix. The above function can be fine tuned using conditional statements. Look into the code given in Section 4.2.2.

3.5.3 Zeros matrix

There is certain discussion related to a method called `matrix` in the previous Chapter in the Section 2.8.1. Following code block shows the logic for creating zeros matrix.

Zero matrix is highly useful in data processing. For instance, the zero matrix is used in bivariate and multivarite data analysis for

1. data transformation, like *linear transformations*,

2. *ordinary least squares regression* to check if annihilator matrix is zero or not for a best fit.
3. a perticular property called *idempotent*, meaning that when it is multiplied by itself the result is itself.

Above are very few use cases. The following code block shows the method for defining a function for zero matrix.

```
> zerosMat <- function(m, n){
+   out <- matrix(0, m, n)
+   return(out)
+ }
```

Zero matrix is required to check a particular property of matrices called additive identity. Let us test additive identity of matrix using zeros matrix.

```
> z <- zerosMat(4, 4)
> z + A == A + z
```

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

3.5.4 Ones matrix

Ones matrices are highly useful in quantitative analysis. This matrix exhibits several influencial properties in data processing. One matrix is useful for evaluating few assumptions sush as: invariance of configuration matrix (input data) using characteristic polynomials; Also used to check neutrality in Hamdard product which is used to compute Kronecker delta in generalized linear models. One of the most influential utility is in latent variable analysis where ones matrix is used to figure out that whether or not an input matrix is positive-semi definit or not. These calculations are beyond the scope of this text. So I shall just confine my discussion to basics.

```
> onesMat <- function(m, n){
+   out <- matrix(1, m, n)
+   return(out)
+ }
```

Now let us check the property of Hamdard Product which is highly used in structural equation modelling and confirmatory factor analysis. The definition for positive definit matrix is as follows:

> "An $n \times n$ symmetric real matrix M is said to be positive-definite if $x^T Mx > 0$ for all non-zero x in R^n." [42]

```
> randMat <- matrix(round(rnorm(16), 2), 4, 4)
> x <- onesMat(1, 4)
> out <- x%*%randMat%*%t(x)
> out

     [,1]
[1,] 3.38
```

The value of `out` tend to differ because I used a base function called `rnorm`, which is used to simulate random data points from normal distribution. The matrix `randMat` is positive-definit if the object *out* is > 0; positive-semi-definit if the value is ≥ 0. We can use *if* condition to check the value. Read Section 4.2.3

3.5.5 Stacks and queues

The other concept which I would like to use for demonstrating the power of functions is *Stacks and Queues*. Functions are enormously useful to define methods related to these data structures. We dealt with stacks and queues in Section 2.4.1. We will try to create two methods called append and remove using the logic explained in Section 2.5.

```
> append <- function(obj, ele){
+   obj <- c(obj, ele)
+   return(obj)
+ }
> remove <- function(obj, ele){
+   eleins <- grep(ele, obj)
+   obj <- obj[-eleins]
+   return(obj)
+ }
```

```
> vec <- c(1:3, "a", "b")
> vec <- append(vec, "c")
> vec

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

> vec <- remove(vec, "c")
> vec

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

Stacks and queues are highly dynamic data structures. These are highly useful while developing applications. These data structures are very famous owing their properties in dealing with data processing. Stacks and queues are considered to be dynamic while appending and removing data as by the need. Above methods `append` and `remove` are two sample and user defined functions that are useful to add or remove elements to the existing data respectively.

Notes

[37]Read Hadley's very detailed text on "Object Oriented Programming" in his book *Advanced R*. Full book is available in HTML text at https://adv-r.hadley.nz/.

[38]Read more at https://adv-r.hadley.nz/oo.html#oop-systems.

[39]Visit http://adv-r.had.co.nz/Functions.html#function-components.

[40]Wickham, H. Functional programming. *Advanced R*. Retrieved from http://adv-r.had.co.nz/Functional-programming.html#functional-programming.

[41]*ibid.*

[42]Source: https://en.wikipedia.org/wiki/Definite_symmetric_matrix

Chapter 4

Control Flow

Control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. The emphasis on explicit control flow distinguishes an imperative programming language from a declarative programming language. Within an imperative programming language, a control flow statement is a statement that results in a choice being made as to which of two or more paths to follow. For non-strict functional languages, functions and language constructs exist to achieve the same result, but they are usually not termed control flow statements. A set of statements is in turn generally structured as a block, which in addition to grouping, also defines a lexical scope.

Following are few choices for control flow in programming.

- If-else statements
- Case and switch statements
- Loops
  - Count-controlled loops
  - Condition-controlled loops
  - Collection-controlled loops

4.1 Conditional Statements

Every programming language has certain building blocks for it to be upto user expectations. Programming is all about practice of coding. Coding has few important ingredients for users. One of the very important ingredient is conditional statements. Conditional statements very essential in programming. Conditional statements together with loops make decision making possible. Conditions can be implemented using *if* statements.

R has its own way of using conditional statements and that way is highly intuitive. Execute `help("if")` statement in R Console, you may find the following syntax for *if* statement.

```
if(cond) expr
if(cond) cons.expr  else  alt.expr
```

If statement is the part of *control flow*. R has quite a few ways to control flow. There are approximately 7 such statements which control the flow in R.

1. if statement
2. if else statement
3. for loop
4. while
5. repeat statement
6. switch statement
7. break statement
8. next statement

The if statements with loops are highly useful while regulating control flow in R. This Chapter provides description on only conditional statements. Rest of the statement together with Loops were explained in the next Chapter. While coming to if statement; there are different types of if statments.

1. vectorized if statement
2. if-else statement

3. nested if-else statement

Let us look at few examples.

4.1.1 Vectorized if statement

Vectorized if statement is simply called as ifelse statement and it is function which is different from conventional if-else statement. This function behaves just like *if()* function in Excel spreadsheet application. [43]

```
> studMarks <- c(55, 61, 76, 87, 99)
> grade <- vector(length = 5)
> grade <- ifelse(studMarks > 40 & studMarks < 60, "B",
+                 ifelse(studMarks > 60 & studMarks < 75, "A",
+                        ifelse(studMarks > 75 & studMarks < 90, "A+",
+                               ifelse(studMarks > 90, "O", "F"))))
> grade
```

```
[1] "B"  "A"  "A+" "A+" "O"
```

Above example demonstrates as how to process student marks. The object `studMarks` is a variable which holds few student's marks. The object `grade` is defined as vector. The vectorised ifelse statement was used to process the marks. After execution the variable `grade` has the information related to grades as defined in the ifelse statement. This is just an example. It is not required to write such a lengthy statements as and when if it required to process marks as this. One solution is to convert above logic into a function.

```
> convMarksIntoGrades <- function(marks){
+   grade <- vector(length= length(marks))
+   grade <- ifelse(studMarks > 40 & studMarks < 60, "B",
+                 ifelse(studMarks > 60 & studMarks < 75, "A",
+                        ifelse(studMarks > 75 & studMarks < 90, "A+",
+                               ifelse(studMarks > 90, "O", "F"))))
+   return(grade)
+ }
```

Now let us create a dummy marks vector and try the above function.

```
> studMarks <- round(runif(5, 50, 100))
> studMarks
```

```
[1] 62 87 69 97 85
```

```
> convMarksIntoGrades(studMarks)
```

```
[1] "A"  "A+" "A"  "O"  "A+"
```

The base function `runif` is handy in creating a vector of random numbers with lower and upper threashold vaues. The object `studMarks` in the above code block has few abstract yet simulated values created by `runif` function. The function `convMarks` convert `studMarks` into grades. That is the advantage of learning functions in R.

4.1.2 if-else statement

This is very general form of if statement. R has supports if-else statement together with it's usualy curly braces style.

```
> a = 1; b = 2
> if (a < b){
+   print(paste(a, "is less than", b))
+ } else {
+   print(paste(a, "is greater than", b))
+ }
```

```
[1] "1 is less than 2"
```

The above code block compares two input variables *a, b* and they are evaluated as by the condition used for if statement. Moreover, the above block also demonstrates as how to use if-else statement for alternative evaluations. The below code block is an example for nested if-else statement.

```
> if (TRUE){
+   print("this is if part")
+ } else {
+   print("this is else part")
+ }
```

```
[1] "this is if part"
```

4.1.3 Nested if conditions

Nested if conditions are used whenever there is any necessity for if conditions inside another if condition. This situation may arise several times while dealing with data processing in few areas like data science and analytics. Especially in machine learning and artificial intelligence.

```
> if (TRUE){
+   if(TRUE){
+     print("TRUE-TRUE Scenario")
+   }
+ } else {
+   if(FALSE){
+     print("FALSE-FALSE Scenario")
+   }
+ }
```

```
[1] "TRUE-TRUE Scenario"
```

Above code block always prints the very first print statement. Logic never encounter the alternative scenario. However, it is possible to use the alternative scenario by writing UDF through user arguments.

```
> TrueFalseTest <- function(arg1, arg2){
+   if (arg1){
+     if(arg2){
+       print(paste(arg1, "-", arg2, "Scenario"))
+     } else {
+       print(paste(arg1, "-", arg2, "Scenario"))
+     }
+ } else {
+     if(arg1){
+       print(paste(arg1, "-", arg2, "Scenario"))
+     } else {
+       print(paste(arg1, "-", arg2, "Scenario"))
+     }
+ }
+ }
> TrueFalseTest(TRUE, TRUE)
```

```
[1] "TRUE - TRUE Scenario"
```

```
> TrueFalseTest(TRUE, FALSE)
```

```
[1] "TRUE - FALSE Scenario"
```

```
> TrueFalseTest(FALSE, TRUE)

[1] "FALSE - TRUE Scenario"

> TrueFalseTest(FALSE, FALSE)

[1] "FALSE - FALSE Scenario"
```

In the above code block there are three if-else statements. In each branch i.e., either in if or in else (main conditional statement), there is another if-else statement (sub condition). The first branch i.e., in if block (of main condition) the logic is being tested for TRUE, FALSE given TRUE for (main) if statement. The if-else statement inside second (sub) branch i.e., in else part of first (main) if-else statement tests TRUE, FALSE for FALSE for main else branch. This logic seems to be confusing, but the code is simpler than the description. *This type of logic is used several occassions in machine learning algorithms.* The concept is beyond the scope of this book. 44

4.2 Applications - Conditional statements

All the above examples shows only simple ways of using If statements. If statement can be used more meaningfully different ways while writing functions. For that matter, using if alone without loops and functions is rather meaningless. By definition control flow is all about regulating program's execution and it requires conditional statements and loops. Let us look at more meaningful yet practical examples for if statement. In the previous Chapter in Secton 3.2 there was some certain discussion on using dots (ellipses) while writing functions. Let me show rather more meaningful example for if-else for writing a simple function using dots in arguments.

```
> yesNoFunc <- function(...){
+   dots <- c(...)
+   if ('yes' %in% dots){
+     print(paste("there is yes in ", dots))
+   } else {
+     print("are you sure?")
+   }
+ }
> yesNoFunc('yes')
```

```
[1] "there is yes in  yes"
```

```
> yesNoFunc('no')
```

```
[1] "are you sure?"
```

The above function i.e., `yesNoFunc` describes utility of if-else in handling dots inside function logic. This function just return either *yes* or *no* in assessing user response.

4.2.1 Data transformations

The vectorized if statement can be used to explain using data transformations which belongs to data analytics.

```
> convToIntervals <- function(x, cat = 5){
+
+   ll <- min(x)
+   ul <- max(x)
+   int <- round((ul - ll)/cat)
+
+   bin1 <- seq(ll, ul, int)
+   bin2 <- c(bin1[-1], bin1[length(bin1)]+int)
+
+   cats <- vector(length= length(x))
+   cats <- ifelse(x >= bin1[1] & x < bin2[1], "cat 1",
+                  ifelse(x > bin1[2] & x < bin2[2], "cat 2",
+                         ifelse(x > bin1[3] & x < bin2[3], "cat 3",
+                                ifelse(x > bin1[4] & x < bin2[4], "cat 4", "cat 5"))))
+   return(cats)
+ }
```

There is lot of logic in this function. The function `convToIntervals` is useful for converting numeric data variable into non-numeric but categorical data variable. This type of activity is quite common in the area of data science and analytics especially for few machine learning algorithms which depends on categorical response variables. The above function computes categories for any user input variable (x) based on a particular method of statistics called *Sturges' formula.* [45] This formula is useful while calculating bin vectors (through data binning) for ifelse statement. [46] Data bins are used by vectorized if statements inside this function.

```
> x <- round(runif(5, 20, 100))
> x
```

```
[1] 68 45 22 50 62
```

```
> convToIntervals(x)

[1] "cat 5" "cat 3" "cat 1" "cat 4" "cat 5"
```

4.2.2 Random matrix

There was rather significant discussion in the Chapter 3 in Section 3.5.2 related to random matrices. The below function has methods which are useful to fine tune the same function using if-else statement.

```
> randMat <- function(m, n, isround = TRUE){
+   if (isround){
+     out <- matrix(round(rnorm(m*n), 2), m, n)
+     return(out)
+   } else {
+     out <- matrix(rnorm(m*n), m, n)
+     return(out)
+   }
+ }
> randMat(4, 4)

      [,1]  [,2]  [,3]  [,4]
[1,]  0.01  0.56 -0.99 -0.23
[2,]  1.29  0.68  1.73 -0.12
[3,] -0.65 -0.25  1.74 -0.70
[4,] -0.83 -0.32  1.59 -0.48
```

4.2.3 Positive semidefinit matrices

There is certain description about positive semidefinit property of matrices in Section 3.5.4. This can be a good example to to evaluate ***nested if condition***.

```
> if (out < 0){
+   print("The matrix A is negative-definit")
+ } else if (out <= 0){
+   print("The matrix A is negative-semi-definit")
+ } else if (out > 0){
+   print("The matrix A is positive-definit")
+ } else if (out >= 0){
```

```
+   print("The matrix A is positive-semi-definit")
+ }
```

```
[1] "The matrix A is positive-definit"
```

Suppose if you are to check this condition too often, then it is better to create a user defined function. I call it `psdMatrix`.

```
> psdMatrix <- function(A){
+   m <- dim(A)[1]
+   n <- dim(A)[2]
+   x <- onesMat(1, n)
+   out <- x%*%A%*%t(x)
+
+   if (out < 0){
+   print("The input matrix is negative-definit")
+ } else if (out <= 0){
+   print("The input matrix is negative-semi-definit")
+ } else if (out > 0){
+   print("The input matrix A is positive-definit")
+ } else if (out >= 0){
+   print("The input matrix is positive-semi-definit")
+ }
+
+ }
```

The above logic looks like a bit nagging, but it is very simple and believe me. I just used an attribute function `dim` to save the number of columns as m, n because I need n for creating ones matrix. Rest of the logic is already explained well before.

```
> A <- randMat(4, 4)
> psdMatrix(A)
```

```
[1] "The input matrix is negative-definit"
```

4.2.4 Probability distributions

I shall explain as how to work with probability distributions. R is highly powerful as far as probability (theoretical) distributions are concerned. R supports quite a few theoretical distributions. [47]

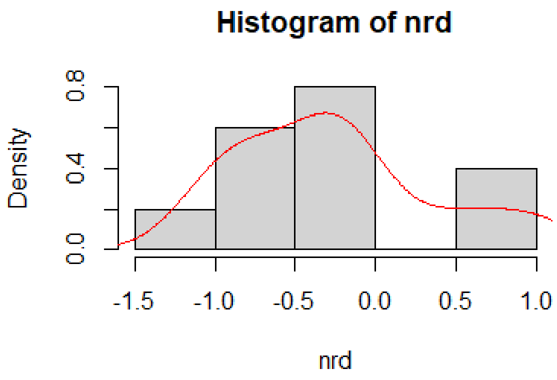


Figure 4.1: Scatter plot for Normal probability distribution

We can use if-else statements for checking type of the distribution and make simulations. Following code snippet shows the way it can be done using R UDF.

```
> printRandomNumbers <- function(typeofthedist, n){
+   if (typeofthedist == 'normal'){
+     out <- rnorm(n)
+     } else if(typeofthedist == 'uniform'){
+       out <- runif(n, 1, n)
+     } else if (typeofthedist == 'binomial'){
+       out <- rbinom(n, n, 0.5)
+     } else if (typeofthedist == 'poisson'){
+       out <- rpois(n, 0.5)
+     }
+   return(out)
+ }
>
> #usage
> nrd <- printRandomNumbers('normal', 10)
> hist(nrd, freq = FALSE); lines(density(nrd), col = '
   ↪ red')
```

The resultant plot looks as below.

4.3 Loops

In computer programming, a loop is a sequence of instructions that is continually repeated until a certain condition is reached or a condition is declared FALSE. Typically, a certain process is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number. If it hasn't, the next instruction in the sequence is an instruction to return to the first instruction in the sequence and repeat the sequence. If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop. A loop is a fundamental programming idea that is commonly used in writing programs. An infinite loop is one that lacks a functioning exit routine . The result is that the loop repeats continually until the operating system senses it and terminates the program with an error or until some other event occurs.

4.3.1 `for`

Loops are highly important for programming. Loops together with conditional statements form logic in coding. Loops are inherent in many base functions in R. For instance, the function `sum` use loop inside its logic to sum up input elements. The below code snippet demonstrates as how to use `for` look for summing input data vector.

```
> sum(1:10)
```

```
[1] 55
```

```
> Sum <- function(x){
+   s = 0
+   for (i in 1:length(x)){
+     s = s + x[i]
+   }
+   return(s)
+ }
> Sum(1:10)
```

```
[1] 55
```

For that matter, the *colon operator* some extent iterates values to create vectors. R uses a technique known as vectorization to deal with loops. Learning to use vectorized operations is a key skill in R. For that matter it is possible to create custom operators.

```
> `%--%` <- function(sv, ev){
+   v <- vector(length = ev - sv)
+   for(i in sv:ev){
+     v[i] <- i
+   }
+   return(v)
+ }
> `%--%`(1, 10)
```

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

```
> 1%--%10
```

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

The above function behaves like *colon operator*. The symbols – were converted as infix operator using % symbol. Infix operators are helpful to define custom opertors in R. [48] However, it doesn't make sense to use *:* in the code while defining alternative to the same. Few functions such as `seq, assign` are advanced functions that depends on colon operator. The above code block demonstrates as how to use for loops for iterating over values within the logic.

4.3.2 `repeat`

R has quite a few ways to deal with loops unlike other languages. It is possible to use few alternatives such as while, repeat in lieu of for statement.

```
> x <- 1
> repeat{
+   print(x)
+ }
```

The above code block prints the value of x until user interruption. This loop can be controlled with help of control flow statement such as `break`.

```
> x <- 1
> repeat{
+   print(x)
+   x <- x + 1
+   if(x == 10){
+     break
+   }
+ }
```

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

4.3.3 `while`

The function `repeat` explained in above subsection is nothing so special, it is just like a While loop. Same job can be accomplished by using While loop.

```
> x <- 1
> while(x){
+   print(x)
+   x <- x + 1
+   if (x == 10){
+     break
+   }
+ }
```

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

```
[1] 6
[1] 7
[1] 8
[1] 9
```

The function `repeat` seems to be a bit simpler compared to `while` loop and that is the only difference.

4.3.4 `switch()`

The `switch()` function in R tests an expression against elements of a list. If the value evaluated from the expression matches item from the list, the corresponding value is returned. Following is the syntax for R `switch()` function.

```
switch (expression, list)
```

In a more trivial way a `switch()` statement can be used as below.

```
> switch(1, 'a', 'b', 'c', 'd', 'e')
[1] "a"
> switch(2, 'first', 'second', 'third', 'fourth', 'fifth
    ↪ ')
[1] "second"
```

Suppose if I have to use it in more meaningful way

```
> switch(
+   'mean',
+   'mean' = mean(1:10),
+   'sd' = sd(1:10),
+   'max' = max(1:10),
+   'min' = min(1:10)
+ )
[1] 5.5
```

Now that we know as how to use `switch()` statement we may proceed further for creating a user defined function (UDF) for summary statistics.

```
> findSummary <- function(vec, stat){
+   out <- switch(
+     stat,
+     'mean' = mean(vec),
+     'sd' = sd(vec),
+     'max' = max(vec),
+       'min' = min(vec)
+    )
+    return(out)
+ }
> findSummary(1:10, 'mean')
[1] 5.5
> findSummary(1:10, 'sd')
[1] 3.02765
> findSummary(1:10, 'max')
[1] 10
> findSummary(1:10, 'min')
[1] 1
```

Isn't it look nice? to make our own methods for statistics. This is where programming or power of R emanates.

4.4 Applications - Loops

Loops are widely useful while writing user defined functions (UDF). Some extent, it is not possible to escape from loops and conditional statements while writing programs. Everything is a control flow in coding. The following subsections are only code blocks related to few concepts dealt in the previous chapters.

4.4.1 Loops for Lists

There is certain discussion on named and unnamed lists in Section 2.7. It is possible to handle these lists using loops.

```
> lis2 <- list()
> for (i in 1:3){
+   lis2[[i]] <- round(rnorm(5), 2)
+ }
> print(lis2)

[[1]]
[1]  0.57 -0.14  0.35  1.67  1.04

[[2]]
[1]  0.74  0.34 -0.48 -0.98 -1.26
```

```
[[3]]
[1] -0.58 -0.59 -0.34 -0.82  0.06
```

The above list object `lis2` was created by just iterating (i) inside loop to create an unnamed list using for loop. This can be one of the very simple example that demonstrates utility of for loop in R.

4.4.2 Loops for summaries

In section 2.6, there was a discussion on obtaining summaries using a base function called `table`. It is possible to write a user defined function (UDF) using For loop and Factor indexing methods.

```
> Table <- function(fact){
+   factlevels <- levels(fact)
+   out <- 1:length(factlevels)
+   for (i in 1:length(factlevels)){
+     out[i] <- sum(fact == factlevels[i])
+   }
+   return(out)
+ }
> Table(gender)
```

```
[1] 4 1
```

The name `Table` was used in the above function, lest the name `table` already exists in R and name confilicts might arise in the work flow if the same name is used. The logic is very simple. The code has three steps

1. Obtained length of levels (`factlevels`).
2. Created a dummy object to parse sums for levels `out`.
3. Calculated respective sums using base function `sum` inside `for` loop.

4.4.3 Loops for matrix operations

There was certain discussion related to matrix multiplication in Section 2.8.3. It is possible to use inner loops or nested loops for dot product.

```
> resmat <- matrix(NA, dim(mat1)[1], dim(mat1)[2], TRUE)
> for (i in 1:dim(mat1)[1]){
+   for (j in 1:dim(mat2)[2]){
+     resmat[i, j] <- mat1[i, j] * mat2[i, j]
+   }
+ }
> resmat
```

```
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

Same logic with little twist can be used for cross product.

```
> resmat <- matrix(0, dim(mat1)[1], dim(mat1)[2])
> for (i in 1:dim(mat1)[1]){
+   for (j in 1:dim(mat2)[2]){
+     for (k in 1:dim(mat1)[2]){
+       resmat[i, j] <- resmat[i, j] +
+         mat1[i, k] * mat2[k, j]
+     }
+   }
+ }
> resmat
```

```
     [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
```

```
> matmulres1 <- mat1 %*% mat2
> resmat == matmulres1
```

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

I used three inner loops to achieve cross product of matrices. The calculations done by loops match with that of internal method. This can be achieved rather more easily using `crossprod` function.

```
> resmat <- matrix(0, dim(mat1)[1], dim(mat1)[2])
> for (i in 1:dim(mat1)[1]){
+   for (j in 1:dim(mat2)[2]){
+     resmat[i, j] <- crossprod(mat1[i, ], mat2[, j])
+   }
+ }
> resmat
```

```
     [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
```

The above code is rather easy for comprehension and also number of loops reduced to two in stead of three. The function `crossprod` clears the clutter of iterators i, j and k in the logic.

Notes

[43]Excel has a if function which has a syntax like `IF(condition, true response, false response)`. This statement can be used like R's ifelse statement by using if in the place of *false response*.

[44]A peculiar method known as *confusion matrix* depends on this type of logic. Perhaps, the name confusion matrix has come due to the way logic is being handled by nested-if conditions. Visit `https://en.wikipedia.org/wiki/Confusion_matrix` for more details on confusion matrix.

[45]Read more about *Sturges' formula* at `https://en.wikipedia.org/wiki/Histogram#Sturges'_formula`.

[46]Read more about Data binning at Wikipedia from `https://en.wikipedia.org/wiki/Data_binning`.

[47]I request the readers to learn about probability distributions from online resources. For instance, visit `https://cran.r-project.org/web/views/Distributions.html` for probability distributions.

[48]There is a detailed guide on infix operators at `http://applied-r.com/data-infix-operators-in-r/`.

Chapter 5

Multivariate Analysis

Multivariate analytics or analysis is a subdivision of statistics encompassing the simultaneous observation and analysis of more than one outcome variable. Multivariate analysis (MVA) is based on the principles of multivariate statistics. Multivariate analysis concerns understanding the different aims and background of each of the different forms of multivariate statistics, and how they relate to each other. Multivariate analysis may involve several types of univariate and multivariate analyses in order to understand the relationships between variables and their relevance to the problem being studied. In addition, multivariate analysis is concerned with multivariate probability distributions, in terms of both how these can be used to represent the distributions of observed data; how they can be used as part of statistical inference, particularly where several different quantities are of interest to the same analysis. Certain types of problems involving multivariate data, for example simple linear regression and multiple regression, are not usually considered to be special cases of multivariate statistics because the analysis is dealt with by considering the (univariate) conditional distribution of a single outcome variable given the other variables. Typically, MVA is used to address the situations where multiple measurements are made on each experimental unit and the relations among these measurements and their structures are important. A modern, overlapping categorization of MVA includes: [49]

- Normal and general multivariate models and distribution theory
- The study and measurement of relationships
- Probability computations of multidimensional regions
- The exploration of data structures and patterns

Typically, MVA is used to address the situations where multiple measurements are made on each experimental unit and the relations among these measurements are being treated as important for decision making. Categorization of MVA includes:

1. Distribution theories (such as χ^2)
2. The study of relationships (such as Canonical models)
3. The exploration of patterns (such as multidimensional scaling)
4. Probability computations of regions (such as Cluster analysis)

5.0.1 Techniques

Various multivariate techniques are in vogue. However, for the sake of simplicity only the following techniques were covered in this text. [1]

1. (Multivariate) Correaltion
2. (Multivariate) Regression
3. Multidimensional Scaling
4. Correspondence Analysis
5. Principal Components Analysis
6. (Exploratory & Confirmatory) Factor Analysis
7. Cluster Analysis

[1]Greenacre, M. Correspondence Analysis and Related Methods. Available at http://statmath.wu.ac.at/courses/CAandRelMeth/CARME1.pdf.

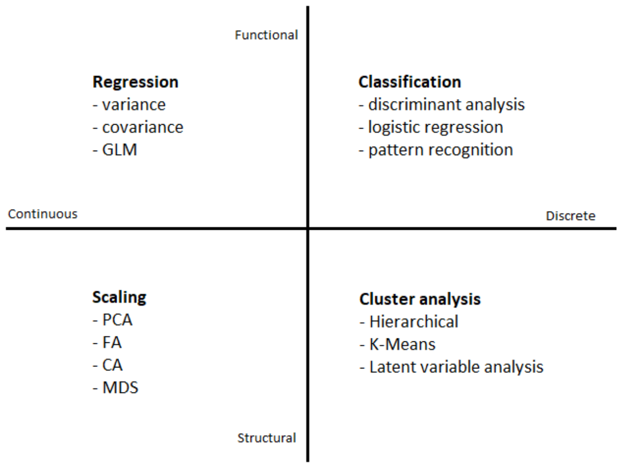


Figure 5.1: Various multi-variate techniques

5.1 Data Simulations

This text has few MVA techniques that requires data simulations. Data simulations usually are very handy for software testing and development. It is not possible to get data in required format every time when in need. Moreover, **a learner can gain huge amount of practice and also understanding on various charades of R programming while learning data simulations**. You will know the significance of this statement in the forthcoming sections.

Data simulation is a procedure that uses certain techniques related to probability theory. There are different types of probability distributions. Few distributions like, (1) normal, (2) binomial, (3) Poisson and etc. are highly useful to simulate data sets. Data simulations depends on certain theory called numerical simulations. Numerical simulation deals with simulating numbers using few rules. It is possible to simulate numerical and non-numerical data using few base functions in R.

5.1.1 Numeric data

Numerical data as the name implies is any data which is numerical. Numerical data is highly useful while testing programs written by the programmer. For instance, it is possible to use *normal distribution* or *theory of normality* to simulated abstract numeric data with random data points. In R there are procedures to simulate random data using aforementioned probability distributions. For instance, it is possible to simulate 10 random data points from normal distribution using a base function `rnorm()`. Numeric data can be simulated as below.

```
> dat <- rnorm(5)
> print(dat)
```

```
[1]  1.8749423  0.5584716  1.1030169  0.5349918 -0.5469772
```

The above technique is the crux of entire gimmick involved in numeric simulations. In social sciences, much of the data represents socioeconomic characteristics. For instance, few socioeconomic characteristics like salary, age are continous and ratio level data. This means the decimal point is valid while dealing with such numeric data. Few variables like attitude, satisfaction are not continuous but categorical. [50] For instance, attitude can be measured as *highly likely, ... highly unlikely.* This type of data variables are not strictly numerical but can be represented as so if required. Otherwise, it is not possible to carry numerical analysis.

```
> salary <- runif(5, 20, 80)
> mean(salary)
```

```
[1] 30.26669
```

In the above code chunk the variable *salary* is continuous but not categorical, despite of the fact that the same variable can be converted to categories. Also it is possible to employ any method related to numerical analysis, such as *mean.* In R, there is certain base function `cut` to convert any continuous data into categorical.

```
> salaryint <- cut(salary, c(20, 40, 60, 80))
> salaryint[order(salaryint)]
> typeof(salaryint)
> mean(salaryint)
> table(salaryint)
```

The object `salaryint` is not numerical, in spite of the fact that it is still "*integer*" in type. However, it is possible to make summary table of frequencies for it is still interval data (or discrete data).

```
> data.frame(table(salint))
    salint  Freq
1   (10,20]     2
2   (20,40]     8
3   (40,60]     5
4   (60,80]     5
5  (80,100]    10
```

Based on above practice it is possible to create a couple of functions which can simulate data sets either in continuous or with intervals. Suppose if I want to create a data variable such as the one explained in salary example above. I might be able to write a function for that as below.

```
> vecData <- function(len, min=20, max=100){
+   out <- runif(len, min, max)
+   return(out)
+ }
> salary <- round(vecData(10), 2)
> salary
 [1] 60.09 64.93 53.31 36.81 55.75 36.85 69.36 42.67
    ↪ 86.45 68.22
```

The data variable `salary` is created using a UDF `vecData()` which process 4 different arguments for R base function `runif()`, which is used to create *uniform distribution*. [51]

These type of functions are quietly useful for creating or simulating data sets with multiple variables i.e., multivariate data. Suppose I would like to simulate a data set with salaries for 3 employees for 12 months. I can do as shown below.

```
> makeMonths <- function(n){
+   monthslist <- c('january', 'february', 'march', '
    ↪ april', 'may', 'june', 'july', 'august', '
    ↪ septermber', 'october', 'november', 'december')
+   if(n <=12){
+     out <- monthslist[1:n]
+   } else{
+     print('invalid number')
+   }
+   return(out)
+ }

> multiVarNumData <- function(rows, cols){
+   len <- rows * cols
+   out <- matrix(runif(len), rows, cols)
+   return(out)
+ }

> salaries <- round(multiVarNumData(12, 3)*100, 2)

> head(salaries)
      [,1]  [,2]  [,3]
[1,] 26.06 64.87 91.77
[2,]  2.72  1.69 77.21
[3,] 57.12 57.44 23.55
[4,] 48.85 56.41 84.46
[5,]  7.64 61.91 91.64
[6,] 86.20 11.69 68.07

> colnames(salaries) <- paste('emp.', 1:3, sep = '')
> rownames(salaries) <- makeMonths(12)

> head(salaries)
         emp.1 emp.2 emp.3
january  26.06 64.87 91.77
february  2.72  1.69 77.21
march    57.12 57.44 23.55
april    48.85 56.41 84.46
may       7.64 61.91 91.64
june     86.20 11.69 68.07
```

The function `makeMonths()` creates a vector composed with month names for argument n. The function `multiVarNumData()` creates a mltivariate data set for two arguments; (1) rows: number of rows, (2) cols: number of columns. The object `salaries` has a data which is 12×3 order matrix. Functions `colnames()` and `rownames()` are two of the R base functions which can be used for setting up column and row wise names. Finally the data set `salaries` is ready with appropriate names for columns and rows.

5.1.2 Non-numeric data

Non numerical data is any data without numerical value. In social sciences there are several instances where variables like gender, family level, education level etc. were used while performing numerical analysis. This type of data is highly importance while dealing with study or research assumptions. For instance, it is possible to evaluate *inter gender differences, inter family level differences, inter education level differences* and etc. There are quite a few techniques to take care of such assumptions. Anyway, this topic is beyond the scope of this text. There was some certain discussion on factors in Section 2.6. Let us use this knowledge to simulated data related to multivariate socioeconomic data for few arguments supplied by user.

```
> factData <- function(len, cat){
+   out <- sample(cat, len, replace = TRUE)
+   return(out)
+ }
> factData(5, c("male", "female"))

[1] "female" "female" "female" "female" "male"

> factData(5, c("employed", "unemployed"))

[1] "employed"   "employed"   "employed"   "unemployed" "unemployed"
```

The above function `factData` can simulate any data variable for two user arguments i.e., `cat` which represents categories and `len` which represents length of the variable. Using this function we may be able to create another abstract function to simulated multivariate data. I will name it `hetMultiVarData`, which stands for *heterogeneous multivariate data.*

```
> multiVarCatData <- function(len, ...){
+   cats <- list(...)
+   out <- list()
+   for (i in 1:length(cats)){
+     out[[i]] <- factData(len, cats[[i]])
+   }
+   return(out)
+ }
```

The logic for above function is rather simple to understand. There is no limit for number of categories for it purely depends on user requirements. That is why, the argument $\cdots$ used in above function. This notation in arguments is called *ellipsis*. This is a special facility in R for providing or accepting indefinit number of arguments. This arguments may be converted into definit list inside the body of the function using base `list` function. The logic is as follows:

1. Convert the categories given by the user into list (`cats`).
2. Create a dummy list (`out`).
3. Simulate or create factors using base `sample` function for argument `len`.

Now its time to test the function.

```
> cat1 <- c("male", "female")
> cat2 <- c("married", "unmarried")
> multivardata <- data.frame(multiVarCatData(10, cat1, cat2))
> colnames(multivardata) <- c("gender", "marital.status")
> head(multivardata)
```

```
  gender marital.status
1 female      unmarried
2 female      unmarried
3 female        married
4   male      unmarried
5 female      unmarried
6 female        married
```

Two of the steps, in the above chunk, like converting the output into data frame and setting column names can be avoided by customizing the function further. It is not difficult but I would like to leave it to reader.

5.1.3 Multivariate heterogeneous data

Multivariate heterogeneous data is a data set which is composed of both numerical and non-numerical data. This type of data sets are rampantly used in multivariate analysis. Usually, multivariate analysis is uses data which is mostly, but not necessarily, uses survey based data, at least in social sciences. It is possible to simulate multivariate data set using above practice. For instance, the below code snippet shows as how to simulates a data sets with few variables such as `gender`, `education`, `salary`, `age`. First two variables are categorical and the other two are numerical.

```
> gender <- factData(30, c('male', 'female'))
> education <- factData(30, c('primary', 'secondary', '
    ↪ higher'))
> salary <- round(vecData(30), 2)
> age <- round(vecData(30), 2)
> hetdataset <- data.frame(gender, education, salary,
    ↪ age)
> head(hetdataset)
  gender education salary   age
1 female   primary  76.11 59.12
2 female   primary  72.29 72.75
3 female   primary  52.46 33.11
4   male secondary  25.37 24.23
5 female secondary  54.13 37.69
6   male    higher  93.24 92.39
```

You see how easy it is to simulate a data set using all those previously written user defined functions (UDFs). That is the power of a programming language. These simulated data sets mimics real time data. **However, there is one big caution that we don't create data. Data arise from the work.** I would like to make some certain discussion on simulations for it has two vital benefits for beginners; (1) it enable learner to practice programming, (2) saves time and no need to search online for data sets. However, in the forthcoming sections, i.e., multivariate analysis, I had used few realtime data sets (survey based) only to show few import/export methods.

5.2 Missing data

Data often are missing in research in economics, sociology, and political science because governments or private entities choose not to, or fail to, report critical statistics, or because the information is not available. Sometimes missing values are caused by the researcher; for example, when data collection is done improperly or mistakes are made in data entry.

Missing data, or missing values, occur when no data value is stored for the variable in an observation. Missing data are a common occurrence and can have a significant effect on the conclusions that can be drawn from the data. Missing data can occur because of nonresponse, which means, no information is provided for one or more items or for a whole unit ("subject"). Some items are more likely to generate a nonresponse than others. For example items about private subjects such as income. Attrition is a type of missingness that can occur in longitudinal studiesfor instance studying development where a measurement is repeated after a certain period of time. Missingness occurs when participants drop out before the test ends and one or more measurements are missing.

These forms of missingness take different types, with different impacts on the validity of conclusions from research. *Missing completely at random, missing at random, and missing not at random.* There are number of strategies to handle missing data; namely, (1) substitution, (2) imputation, (3) interpolation, (4) deletion.

5.2.1 Substitution

Following is the simple explanation for substitution.

```
> x <- round(rnorm(5)*10, )
> x
[1]  -7  13  -3   9 -12
> sort(x)
[1] -12  -7  -3   9  13
> x[2] <- NA
> x
[1]  -7  NA  -3   9 -12
> is.na(x)
[1] FALSE  TRUE FALSE FALSE FALSE
> x[is.na(x)]
[1] NA
> x[is.na(x)]
[1] NA
> is.na(x)
[1] FALSE  TRUE FALSE FALSE FALSE
> x[5] <- NA
> sum(is.na(x))
[1] 2
> mean(x)
[1] NA
```

It is not possible to compute *mean* due to missing data. We need to use `na.rm=T` argument to compute arithmetic mean for data variable that has missing data.

```
> mean(x, na.rm = TRUE)
[1] -0.3333333
```

It is possible to substitute the missing data in R and it is very easy.

```
> x[is.na(x)] <- mean(x, na.rm = T)
> x
[1] -7.0000000 -0.3333333 -3.0000000  9.0000000
    ↪ -0.3333333
> mean(x)
[1] -0.3333333
```

Data frames

We need loops to handle missing data in data frames.

```
data_set <- matrix(runif(40, 1, 5), 10, 4)
data_frame <- data.frame(data_set)
data_frame <- round(data_frame, 2)
data_frame[sample(1:3, 3, replace = F), sample(1:3, 3,
    ↪ replace = F)] <- NA

data_frame.2[, 1][is.na(data_frame.2[, 1])]

for (i in 1:ncol(data_frame.1)){
  data_frame.1[, i][is.na(data_frame.1)[, i]] <- mean(
    ↪ data_frame.1[,i], na.rm = T)
  }
```

5.2.2 Deletion

List-wise deletion

List-wise deletion is a method for handling missing data. In this method, an entire record is excluded from analysis if any single value is missing. It is possible to handle list-wise deletion in two ways using `na.omit()`, `complete.cases()`. **List-wise is deletion advisable when variables are important and sample size is not a matter for analysis**.

```
data_frame.1 <- data_frame

na.omit(data_frame.1)
data_frame.1[complete.cases(data_frame.1), ]
```

List-wise deletion affects statistical power of the tests conducted. Statistical power relies in part on high sample size. Because List-wise deletion excludes data with missing values, it reduces the sample which is being statistically analyzed.

Pair-wise deletion

Pairwise deletion involves deleting a case when it is missing a variable required for a particular analysis, but including that case in analyses for which all required variables are present. When pairwise deletion is used, the total N for analysis will not be consistent across parameter estimations. Because of the incomplete N values at some points in time, while still maintaining complete case comparison for other parameters, pairwise deletion can introduce impossible mathematical situations such as correlations that are over 100%. **List-wise deletion is advisable when case are important and loosing variables is not a matter for analysis**. Pair-wise deletion can be achieved using `complete.cases()` function.

```
> data_frame.pair_wise <- data_frame.1[complete.cases(t(
    ↪ data_frame.1))]
> data_frame.pair_wise
          X4
1   1.806785
2   2.801667
3   2.414237
4   1.229590
5   4.189680
6   1.426970
7   1.088893
8   1.055957
9   3.636893
10  2.172764
```

5.3 Correlation

Correlation is any statistical relationship, whether causal or not, between two or more random variables. In the broadest sense correlation is can also represent statistical association, though it commonly refers to the degree to which a pair of variables are linearly related.

A correlation coefficient is a numerical measure of some type of correlation, meaning a statistical relationship between two variables. The variables may be two columns of a given data set of observations, often called a sample, or two components of a multivariate random variable with a known distribution. Several types of correlation coefficient exist, each with their own definition and own range of usability and characteristics. They all assume values in the range from 1 to +1, where ś1 indicates the strongest possible agreement and 0 the strongest possible disagreement. [52]

There are different types of methods to measure relationships. Following are the most commonly used measures.

- Ratio data
  - Pearson product-moment correlation coefficient
  - Pearson's chi-squared test
- Ordinal data
  - Spearman's rank correlation coefficient
  - Kendall tau rank correlation coefficient
  - Goodman and Kruskal's gamma
- Nominal data
  - Odds ratio
  - Phi coefficient
  - Crammer's C & V coefficient
  - Goodman and Kruskal's lambda
  - Yule's Q

Following is the most generic experssion for calculating correlation coefficient (r).

$$\rho_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$$

All the above techniques are meant for bivariate analysis. For instance, Karl Pearson's Correlation Coefficeint for any two variables can be done as shown below.

```
> sales <- multiVarNumData(12, 3)
> colnames(sales) <- paste('Product.', LETTERS[1:3], sep
  ↪ = '')
> rownames(sales) <- makeMonths(12)
> sales <- round(sales*100, 2)
> head(sales)
```

```
          Product.A Product.B Product.C
january       29.11     18.21     21.32
february      13.59     49.22      6.99
march          0.46     64.37     40.83
april         48.21      0.63     59.85
may           74.85     32.91     87.11
june          94.22     23.14     12.11

> cor(sales, method = c('pearson'))
             Product.A  Product.B   Product.C
Product.A  1.00000000 -0.3035630 -0.08458243
Product.B -0.30356300  1.0000000 -0.14329559
Product.C -0.08458243 -0.1432956  1.00000000
```

I just used `multiVarNumData()` to create multivariate data set composing of sales for three different products namely, *Product.A*, *Product.B* and *Product.C* for 12 different months from *january* to *december*. The function `makeMonths()` is used to set month names (rows) for data set. [2] `cor()` is the R base function to compute Karl Pearson's Correlation r coefficient. You can use any acceptable convensions to interpret the data. For instance, sales performance of *Product.A* negatively associated with both *Product.B* and *Product.C*. However, this is highly plausible yet very light minded way of interpreting data. There is also *Correlation Significance Test* in bivariate correlation analysis. That is beyond the scope of this text. I recommend user to pick material from online resources for the correlation significance test.

5.3.1 Multiple correlation

Bivariate correlations are useful for finding relationships between any two distinct data variables. The association is confined to only those two variables of interest. However, there might be situations where association of one variables versus a bunch of other variables becomes necessary. Suppose, the satisfaction of customers and its dependency over other few socioeconomic variables might be a best example for this scenario. In which, case bivariate correlation may not provide sufficient basis for evaluation of relationships over dependencies. However, this scenario can be tackled using multiple correlation. Multiple correlation is a measure of how well a given

[2] Look at the section 5.1.1 for `makeMonths()` function.

variable can be predicted using a linear function of a set of other variables.

$R^2 = \mathbf{c}^\top R_{xx}^{-1}\,\mathbf{c}$

In the above expression R^2 is known as *coefficient of multiple correlation*. The coefficient of multiple correlation takes values between 0 and 1. Higher values indicate higher predictability of the dependent variable from the independent variables, with a value of 1 indicating that the predictions are exactly correct and a value of 0 indicating that no linear combination of the independent variables is a better predictor than is the fixed mean of the dependent variable. Following is the method of calculating *multiple correlation*.

```
# simulation of data sets
> satisfaction <- sample(c(0, 1), 30, replace = TRUE)
> age <- round(vecData(30), )
> education <- sample(c(0, 1), 30, replace = TRUE)
> marital_status <- sample(c(0, 1), 30, replace = TRUE)
```

Actually I don't need to create an object `marital_status` but I need it to prove the consistency of the methods employed in multiple correlation using multiple regression in forthcoming sections. Now let me calculate multiple correlation coefficient and evaluate the level of dependency between dependent variable (satisfaction) with independent variables (`age`, `education` and `marital_status`).

```
> ryx <- matrix(NA, dim(x)[2], 1)
> for (i in 1:length(names(x))){
+   ryx[i] <- cor(y, x[, i])
+ }
> rxx <- cor(x)
> ## multiple correlation using matrix multiplication
> r2 <- t(ryx) %*% solve(rxx) %*% ryx

> r2
          [,1]
[1,] 0.4772947
```

The multiple correlation coefficient or measure seems to be positive and also indicates moderate association with independent variables. However, keeping procedure aside; what is the consistency of the method? Can we believe in this procedure? The answer is YES!

I will show you the alternative procedure to compute R^2 measure using multiple regression in forthcoming sections.

5.3.2 Polychoric correlation

The polychoric correlation coefficient measures association between two ordered-categorical variables. It's technically defined as the estimate of the Pearson correlation coefficient one would obtain if:

1. The two variables were measured on a continuous scale, instead of as ordered-category variables.
2. The two continuous variables followed a bivariate normal distribution.

When both variables are dichotomous instead of ordered-categorical, the polychoric correlation coefficient is called the tetrachoric correlation coefficient.

This technique is frequently applied when analyzing items on self-report instruments such as personality tests and surveys that often use rating scales with a small number of response options (e.g., strongly disagree to strongly agree). The smaller the number of response categories, the more a correlation between latent continuous variables will tend to be attenuated.

Imagine that there are two variables which represents *satisfaction* in stead of one. Let us assume that the variables `age`, `education` and `marital_status` represents a hidden dimension known as *socioeconomic profile*. Now Polychoric correlation can be used to evaluate if these variables are bivariate normal or not while assessing associations.

```
> # simulation
> sat1 <- sample(c(0, 1), 30, replace = TRUE)
> sat2 <- sample(c(0, 1), 30, replace = TRUE)
> age <- round(vecData(30), )
> # age_cat <- cut(age, c(20, 40, 60, 80, 100))
> education <- sample(c(0, 1), 30, replace = TRUE)
> marital_status <- sample(c(0, 1), 30, replace = TRUE)
> multicordata <- data.frame(sat1, sat2, age, education,
    ↪ marital_status)
> head(multicordata)
  sat1 sat2 age education marital_status
```

```
1     1     1   45          1               0
2     1     1   61          0               1
3     0     1   71          1               1
4     1     0   82          0               0
5     0     1   25          0               0
6     1     1   98          1               0
> round(cor(multicordata), 2)
                sat1  sat2   age education marital_
    ↪ status
sat1            1.00  0.22 -0.02     -0.41
    ↪ -0.14
sat2            0.22  1.00 -0.02     -0.09
    ↪ 0.21
age            -0.02 -0.02  1.00      0.00
    ↪ 0.18
education      -0.41 -0.09  0.00      1.00
    ↪ 0.17
marital_status -0.14  0.21  0.18      0.17
    ↪ 1.00
```

The simulations worked out perfectly, and we got meaningful data for analysis. For instance, if you look at the variables `sat1, sat2` they are positively correlated. Other variables related to socioeconomic profile seems to be positively correlated. This means there are two perfect latent dimensions in the data; the first, *satisfaction* represented by `sat1` and `sat2`. Second, *socioeconomic profile* represented by `age, education` and `marital_status`. The object `mulcordata` is just a data frame object in which all the variables of interest are columns.

```
> hetcor(multicordata, ML=TRUE)

Maximum-Likelihood Estimates

Correlations/Type of Correlation:
                   sat1     sat2       age  education
    ↪ marital_status
sat1                  1  Pearson   Pearson   Pearson
    ↪     Pearson
sat2             0.2182        1   Pearson   Pearson
    ↪     Pearson
age            -0.01504 -0.02363         1   Pearson
    ↪     Pearson
education       -0.4082 -0.08909 -0.004912         1
    ↪     Pearson
marital_status  -0.1361   0.2079    0.1762    0.1667
    ↪          1
```

```

Standard Errors:
                 sat1   sat2    age education
sat1
sat2           0.1737
age            0.1821 0.1821
education      0.1525 0.1807 0.1821
marital_status 0.1789 0.1745 0.1767    0.1772

n = 30

P-values for Tests of Bivariate Normality:
                    sat1      sat2       age education
sat1
sat2            6.94e-13
age            8.205e-06 1.641e-06
education      8.259e-12 5.361e-13 1.185e-05
marital_status 4.624e-12  3.35e-13 1.134e-05 3.701e-12
```

The function `hetcor()` in `polycor` library can estimate bivariate normality of relationships in the sampled data. All the coefficients are *pearson* values and they are slightly different from the earlier correlation coefficients. This is due to the argument `ML=TRUE` in `hetcor()` function. The correlation coefficient values estimated by `hetcor()` function are maximum likelihood measures. All the P Values are close to zero. This means none of the relationships are, in fact, bivariate normal in the sampled data. [3] The conclusion is that the latent variables or dimensions in the data are vividly identified.

One important observation is the correlations types or methods that were used by the `hetcor()` function. All coefficients were calculated by using *pearson* method. This will not be the case if the variables are categorical. For instance, I will convert the variable age into categorical by using R base function `cut`.

```
> # polyserial
> age_cat <- cut(age, c(20, 40, 60, 80, 100))
> multicordata <- data.frame(sat1, sat2, age_cat,
    ↪ education, marital_status)
> hetcor(multicordata, ML=TRUE)

```

[3]Simulated data sets always shows significant results. Readers are advised to test these methods on real time data sets for analysis.

```
Maximum-Likelihood Estimates

Correlations/Type of Correlation:
                   sat1     sat2    age_cat  education
    ↪  marital_status
sat1                  1  Pearson Polyserial    Pearson
    ↪        Pearson
sat2             0.2182        1 Polyserial    Pearson
    ↪        Pearson
age_cat        -0.07687   -0.147          1 Polyserial
    ↪     Polyserial
education       -0.4082 -0.08909   0.007479          1
    ↪        Pearson
marital_status  -0.1361   0.2079     0.2199     0.1667
    ↪              1

Standard Errors:
                 sat1   sat2 age_cat education
sat1
sat2           0.1737
age_cat        0.1985 0.1938
education      0.1525 0.1807  0.2005
marital_status 0.1789 0.1745   0.188    0.1772

n = 30

P-values for Tests of Bivariate Normality:
                    sat1      sat2   age_cat education
sat1
sat2            6.94e-13
age_cat        1.606e-05 2.555e-06
education      8.259e-12 5.361e-13 9.477e-06
marital_status 4.624e-12  3.35e-13 7.274e-06 3.701e-12
```

The variable `age` is converted to categorical variable. Now observe the methods used by the `hetcor()` function. All the coefficients for `age_cat` were calculated by using *polyserial* correlation. This means; correlation (and its standard error) between a `age_cat` and all other variables were calculated for base assumption that the joint distribution of the latent continuous variable underlying the `age_cat` and rest of the variables is bivariate normal. However, it is not the case in the data for the P Value is close to zero.

What is the difference between pearson's r and polyserial correlation coefficient? Polyserial correlation coefficient (rps) is just $r = r * (sqrt(n-1)/n)\sigma y / \sum(zpi)$ where zpi are the ordinates of the normal curve at the normal equivalent of the cut point boundaries between the item responses. [53]

5.3.3 Partial Correlation

Partial

Partial correlation measures the degree of association between two random variables, with the effect of a set of controlling random variables removed. If we are interested in finding to what extent there is a numerical relationship between two variables of interest, using their correlation coefficient will give misleading results if there is another, *confounding*, variable that is numerically related to both variables of interest. Visit http://faculty.cas.usf.edu/mbrannick/regression/Part3/Partials.html for more description on partial correlation.

I would like to show you as how to practice partial correlations using covid19 world pandemic. Let us assume relationship between *personal care* and *covid19 infection*. I would like to define *personal care* using two variables namely; *sanitation*, and *social.distance*; these variables are assumed to possess negative relationship with the third variable i.e., *covid19.infection*. Given such scenario, it might be possible to assume partial correlation from *sanitation* and *social.distance* over *covid19.infection*. In which case we can keep either of *sanitation* or *social.distance* as constant while assessing their relationship with *covid19.infection*. Let us simulate data as in ordinal scale. *social.distance* and *sanitation* were assumed to be measured using 5 point Likart scale, whereas *covid19.infection* is measured using dichotomous response such as *yes* or *no*.

```
> codeData <- function(dat){
+   out <- unclass(as.factor(dat))
+   return(out)
+ }
```

I defined the above UDf for converting factor data into numeric data variable.

```
> sanitation <- sample(1:5, 30, replace = TRUE)
> social.distance <- sample(1:5, 30, replace = TRUE)
> covid19.infection <- sample(c('yes', 'no'), 30,
   ↪ replace = 30)
```

```
>
> covid19.dataset <- data.frame(sanitation, social.
    ↪ distance, covid19 = as.numeric(codeData(covid19.
    ↪ infection)))
>
> cor(covid19.dataset)
                sanitation social.distance     covid19
sanitation       1.0000000     -0.10434322 -0.19075676
social.distance -0.1043432      1.00000000 -0.08297134
covid19         -0.1907568     -0.08297134  1.00000000
```

`sanitation` and `social.distance` are simulated for Likart 5 point scale whereas `covid19.infection` is being simulated for dichotomous responses namely (1) yes, and (2) no. While coming to bivariate correlations; both `sanitation` and `social.distance` are negatively correlated to `covid19.infection`. This means it seems that *personal care* do mitigate *covid19 infection*. However, as mentioned earlier if we infer certain *confounding* or *surrogate* relationships, such relationships can be evaluated using partial correlations.

```
> library(ppcor)
> pcor(covid19.dataset)
$estimate
                sanitation social.distance    covid19
sanitation       1.0000000      -0.1228421 -0.2012025
social.distance -0.1228421       1.0000000 -0.1053751
covid19         -0.2012025      -0.1053751  1.0000000

$p.value
                sanitation social.distance   covid19
sanitation       0.0000000       0.5255343 0.2952845
social.distance  0.5255343       0.0000000 0.5864279
covid19          0.2952845       0.5864279 0.0000000

$statistic
                sanitation social.distance    covid19
sanitation       0.0000000      -0.6431774 -1.0673057
social.distance -0.6431774       0.0000000 -0.5506107
covid19         -1.0673057      -0.5506107  0.0000000

$n
[1] 30

$gp
[1] 1

$method
```

```
[1] "pearson"
```

All partial correlations are as expected. The confounding effect does not seems to be statistically significant for all the P Values are above the threshold value ($\alpha = 0.05$). You may do as below if you wants to know about the partial correlations.

```
> pcor.test(covid19.dataset$sanitation, covid19.dataset$
    ↪ covid19, covid19.dataset\$social.distance)
   estimate   p.value statistic  n gp  Method
1 0.2611631 0.1711777  1.405833 30  1 pearson
```

These statistics are identical to respective relation from the `pcor()` results above, i.e., the relationship between *sanitation* versus *covid19* given *social distance* is constant.

Semipartial

The semipartial (or part) correlation statistic is similar to the partial correlation statistic. Both compare variations of two variables after certain factors are controlled for, but to calculate the semipartial correlation one holds the third variable constant for either X or Y but not both, whereas for the partial correlation one holds the third variable constant for both. The semipartial correlation compares the unique variation of one variable (having removed variation associated with the Z variable(s)), with the unfiltered variation of the other, while the partial correlation compares the unique variation of one variable to the unique variation of the other.

The semipartial (or part) correlation can be viewed as more practically relevant "because it is scaled to (i.e., relative to) the total variability in the dependent (response) variable." Conversely, it is less theoretically useful because it is less precise about the role of the unique contribution of the independent variable.

Goging back to Covid19 data set; imagine that I wants to verify influence of *social distance* over *covid19 infection* while keeping only the relationship between *covid19 infection* and *social distance* as constant in stead for both variables of personal care, then I can do as below.

```
> spcor.test(covid19.dataset$social.distance, covid19.
    ↪ dataset$covid19, covid19.dataset$sanitation)
```

```
   estimate   p.value statistic  n gp  Method
1 0.2479536 0.1946657  1.329936 30  1 pearson
> spcor.test(covid19.dataset$sanitation, covid19.dataset
    ↪ $covid19, covid19.dataset$social.distance)
    estimate   p.value  statistic  n gp  Method
1 -0.1143904 0.5546145 -0.5983175 30  1 pearson
```

There isn't any change in coefficients nor the P Values are statistically significant. So, finally we can conclude that the *personal care* found to impact *covid19 infection* and there are not any suspicious relationships among the study variables.

5.3.4 Canonical Correlation

canonical-correlation analysis (CCA), also called canonical variates analysis, is a way of inferring information from cross-covariance matrices. If we have two vectors X = (X1, ..., Xn) and Y = (Y1, ..., Ym) of random variables, and there are correlations among the variables, then canonical-correlation analysis will find linear combinations of X and Y which have maximum correlation with each other. [54]

Definition

Given two column vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ of random variables with finite second moments, one may define the cross-covariance $\Sigma_{XY} = \mathrm{cov}(X, Y)$ to be the $n \times m$ matrix whose (i, j) entry is the covariance $\mathrm{cov}(x_i, y_j)$. In practice, we would estimate the covariance matrix based on sampled data from $XandY$ (i.e. from a pair of data matrices).

Canonical-correlation analysis seeks vectors a ($a \in R^n$) and b ($b \in R^m$) such that the random variables $a^T X$ and $b^T Y$ maximize the correlation $\rho = \mathrm{corr}(a^T X, b^T Y)$. The random variables $U = a^T X$ and $V = b^T Y$ are the first pair of canonical variables. Then one seeks vectors maximizing the same correlation subject to the constraint that they are to be uncorrelated with the first pair of canonical variables; this gives the second pair of canonical variables. This procedure may be continued up to $\min\{m, n\}$ times.

$$(a', b') = \underset{a,b}{\operatorname{argmax}} \operatorname{corr}(a^T X, b^T Y)$$

Hypothesis testing

Each row can be tested for significance with the following method. Since the correlations are sorted, saying that row i is zero implies all further correlations are also zero. If we have p independent observations in a sample and $\widehat{\rho}_i$ is the estimated correlation for $i = 1, \ldots, \min\{m, n\}$. For the i^{th} row, the test statistic is:

$$\chi^2 = -\left(p - 1 - \tfrac{1}{2}(m + n + 1)\right) \ln \prod_{j=i}^{\min\{m,n\}} (1 - \widehat{\rho}_j^2)$$

which is asymptotically distributed as a chi-squared with $(m - i + 1)(n - i + 1)$ degrees of freedom for large p.[55] Since all the correlations from $\min\{m, n\}$ to p are logically zero (and estimated that way also) the product for the terms after this point is irrelevant. Note that in the small sample size limit with $p < n+m$ then we are guaranteed that the top $m + n - p$ correlations will be identically 1 and hence the test is meaningless. [56]

Data simulation

Lets see how to find association/relationship between sales of two regions such as *south* and *north*. The sales data assumed to be arranged in city-wise namely, *city 1*, *city 2*, *city 3* and *city 4*.

```
> southsales <- multiVarNumData(10, 4)
> colnames(southsales) <- paste('city', 1:4)
> head(southsales)
     city 1 city 2 city 3 city 4
[1,]   0.86   0.40   0.24   0.13
[2,]   0.86   0.52   0.77   0.53
[3,]   0.66   0.20   0.93   0.28
[4,]   0.72   0.33   0.01   0.86
[5,]   0.02   0.98   0.82   0.29
[6,]   0.30   0.42   0.73   0.77
>
> westsales <- multiVarNumData(10, 4)
> colnames(westsales) <- paste('city', 1:4)
> head(westsales)
     city 1 city 2 city 3 city 4
[1,]   0.20   0.14   0.66   0.85
[2,]   0.99   0.33   0.76   0.98
[3,]   0.44   0.81   0.90   0.31
```

```
[4,]    0.25    0.88    0.60    0.36
[5,]    0.88    0.41    0.85    0.32
[6,]    0.30    0.93    0.99    0.98
```

I just used *multiVarNumData()* function defined in section 5.1.1 to simulate data sets of sales for two different regions namely *south* and *north*. The sales were created in ration scale. The following procedure shows as how to perform *canonical correlation* using R base function `cancor()`.

```
> out <- cancor(southsales, westsales)
> out
$cor
[1] 0.80380682 0.60483331 0.49781569 0.04561462

$xcoef
              [,1]       [,2]       [,3]       [,4]
city 1 -0.7856959 -1.2407582 -0.5048359  0.6649262
city 2 -0.7571980 -0.9479509 -0.3519891 -0.8389841
city 3  0.3910514 -0.2086043 -1.1553523  0.5527363
city 4  1.0048376 -0.7032582  0.2153108  0.2958516

$ycoef
              [,1]        [,2]       [,3]       [,4]
city 1  0.1997096  0.09247335 -1.2781059 -0.2216730
city 2  1.1061750 -0.34837355 -0.4702877 -0.3016660
city 3  0.5636997  0.60812867  0.2152717  0.9332916
city 4 -0.2249277 -0.86353579 -0.3219558  0.5558739

$xcenter
city 1 city 2 city 3 city 4
 0.540  0.486  0.639  0.488

\$ycenter
city 1 city 2 city 3 city 4
 0.480  0.586  0.661  0.533
```

The matrices `xcoef` and `ycoef` are not correlation coefficients but canonical correlation coefficients which needs to be interpreted using regression convention. This means one unit increase in *city 1* south sales leads to 19.19 % increase in north sales or *vice versa*. In fact, each column within these matrices i.e., `xcoef` and `ycoef`, are called as canonical variables which are referred as *U*, *V* at section 5.3.4.

5.4 Regression

Multivariate Regression is a method used to measure the degree at which more than one independent variable (predictors) and more than one dependent variable (responses), are linearly related. The method is broadly used to predict the behavior of the response variables associated to changes in the predictor variables, once a desired degree of relation has been established. For the sake of understanding we will focus on all three regression analysis viz. simple linear regression, multiple regression and multivariate regression. [57]

5.4.1 Linear regression

Simple linear regression is a linear approach to modeling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables). The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression.

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = x_i^T \beta + \varepsilon_i$$

Where

$$\beta = \frac{\sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i^2}$$

Suppose if we assume that “ice cream sales” can depend on daily “temperature”. That will gives rise to a hypothesis that $H_0 : \beta_{icecreamsales} \neq 0; \beta_{temperature} \neq 0$. Let us test this hypothesis with simulated data sets.

Data simulation

```
> temperature <- round(runif(10)*100, )
> icecreamesales <- round(runif(10)*100, )
> data.frame(icecreamsales, temperature)
   icecreamsales temperature
1              9          71
2             28          10
3             17           4
4             49          82
5             43          53
6             69          24
7             84          83
```

```
8               44            75
9               12             9
10              45             6
```

Not bad! Simple linear regression can be performed as shown below.

```
> summary(lm(icecreamsales ~ temperature))

Call:
lm(formula = icecreamsales ~ temperature)

Residuals:
    Min      1Q  Median      3Q     Max
-39.114 -10.725  -2.691  11.132  33.902

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  28.4521    12.2475   2.323   0.0487 *
temperature   0.2769     0.2319   1.194   0.2666
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 23.77 on 8 degrees of freedom
Multiple R-squared:  0.1513,  Adjusted R-squared:
    ↪ 0.04522
F-statistic: 1.426 on 1 and 8 DF,  p-value: 0.2666
```

If we assume *ice cream sales* as y and *temperature* as x, then the regression equation for the above results is going to be $y = 28.4521 + (0.2769) \times x$. The intercept is significant (P Value = 0.0487) at 5% significance level. However, the fitness seems to be a matter of concern, $R^2 = 0.1513$. We can reject null hypothesis.

5.4.2 Multiple regression

Multiple regression also known as *General linear model.* In the more general multiple regression model, there are p independent variables:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i,$$

where x_{ij} is the $i^t h$ observation on the $j^t h$ independent variable. If the first independent variable takes the value 1 for all i, $x_{i1} = 1$, then β_1 is called the regression intercept.

The least squares parameter estimates are obtained from p normal equations. The residual can be written as

$$\varepsilon_i = y_i - \hat{\beta}_1 x_{i1} - \cdots - \hat{\beta}_p x_{ip}.$$

In matrix notation, the normal equations are written as

$$(\mathbf{X}^\top \mathbf{X})\hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{Y},$$

where the ij element of $\mathbf{X}$ is x_{ij}, the i element of the column vector Y is y_i, and the j element of $\hat{\boldsymbol{\beta}}$ is $\hat{\beta}_j$. Thus $\mathbf{X}$ is $n \times p$, Y is $n \times 1$, and $\hat{\boldsymbol{\beta}}$ is $p \times 1$. The solution is

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y}$$

Data simulation

```
> gender <- factData(30, c('male', 'female'), type='
    ↪ numerical')
> education <- factData(30, c('primary', 'secondary', '
    ↪ higher'), type='numerical')
> salary <- round(vecData(30), 2)
> age <- round(vecData(30), 2)
> satisfaction <- round(vecData(30), 0)
> hetdataset <- data.frame(satisfaction, gender,
    ↪ education, salary, age)
> head(hetdataset)
  satisfaction gender education salary  age
1            4      1         1   2.25 3.46
2            5      2         1   1.04 1.82
3            3      1         1   4.52 4.99
4            3      1         3   1.44 3.42
5            4      2         2   3.14 2.16
6            3      1         3   1.76 1.30
```

Following block has the code and results for multiple regressino.

```
> summary(lm(satisfaction ~ gender + education + salary
    ↪ + age), data = hetdataset)

Call:
lm(formula = satisfaction ~ gender + education + salary
    ↪ + age)

Residuals:
    Min      1Q  Median      3Q     Max
-2.9662 -0.7899  0.1241  0.8152  1.7822

```

```
Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)      3.24578    1.42313   2.281   0.0314 *
gender           0.63256    0.47058   1.344   0.1910
education       -0.28224    0.28702  -0.983   0.3349
salary          -0.19405    0.19789  -0.981   0.3362
age              0.05594    0.23399   0.239   0.8130
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 1.239 on 25 degrees of freedom
Multiple R-squared:  0.1343,  Adjusted R-squared:
    ↪ -0.004268
F-statistic: 0.9692 on 4 and 25 DF,  p-value: 0.4418
```

Except intercept nothing is statistically significant. Satisfaction does not depend on other variables in this simulated data set.

5.4.3 Multivariate regression

MANOVA

Multivariate analysis of variance (MANOVA) is a procedure for comparing multivariate sample means. As a multivariate procedure, it is used when there are two or more dependent variables, and is often followed by significance tests involving individual dependent variables separately.

MANOVA is based on the product of model variance matrix, $\sum_{model}$ and inverse of the error variance matrix, $\sum_{res}^{-1}$, or $A = \sum_{model} \times \sum_{res}^{-1}$. The hypothesis that $\sum_{model} = \sum_{residual}$ implies that the product $A \sim I$. Invariance considerations imply the MANOVA statistic should be a measure of magnitude of the singular value decomposition of this matrix product, but there is no unique choice owing to the multi-dimensional nature of the alternative hypothesis.

The most common statistics are summaries based on the roots (or eigenvalues) λ_pA matrix:

- Samuel Stanley Wilks $\Lambda_{Wilks} = \det(I+A)^{-1} = \det(\Sigma_{res})/\det(\Sigma_{res} + \Sigma_{model})$
- the K. C. Sreedharan Pillai-M. S. Bartlett trace, $\Lambda_{Pillai} = tr(A(I+A)^{-1})$

- the Lawley-Hotelling trace, $\Lambda_{LH} = tr(A)$
- Roy's greatest root $\Lambda_{Roy} = \max_p(\lambda_p) = \|A\|_\infty$

Suggested reading:

Title : "Effects of the COVID-19 Pandemic and Nationwide Lockdown on Trust, Attitudes Toward Government, and Well-Being"

URL : https://doi.apa.org/fulltext/2020-39514-001.html

```
> multivardata <- read.csv(paste("D:/Work/Books/R/MultiVarAnal/Data/multivardata.csv", sep=""))
> names(multivardata)[1:5]

[1] "age"       "income"    "family"    "education" "sat1"

> fit <- manova(cbind(sat1, sat2, sat3) ~ education + family,
+               data = multivardata )
> summary(fit)

          Df  Pillai approx F num Df den Df  Pr(>F)
education  2 0.51002  2.85249      6     50 0.01815 *
family     1 0.09755  0.86477      3     24 0.47286
Residuals 26
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

5.5 Multidimensional Scaling

Multidimensional scaling (MDS) is a means of visualizing the level of similarity of individual cases of a dataset. MDS is used to translate "information about the pairwise 'distances' among a set of n objects or individuals" into a configuration of n points mapped into an abstract Cartesian space. More technically, MDS refers to a set of related ordination techniques used in information visualization, in particular to display the information contained in a distance matrix. It is a form of non-linear dimensionality reduction. Given a distance matrix with the distances between each pair of objects in a set, and a chosen number of dimensions, N, an MDS algorithm places each object into N-dimensional space (a lower-dimensional representation) such that the between-object distances are preserved as well as possible. For $N = 1, 2$, and 3, the resulting points can be visualized on a scatter plot. Core theoretical contributions to MDS were made by *James O. Ramsay* of McGill University, who is also regarded as the founder of functional data analysis. The actual distance between two points i and j may be computed numerically using the Euclidean distance formula:

$$d_{ij} = \sqrt{\sum_{k=1}^{p}(x_{ik} - x_{jk})}$$

Goodness-of-Fit

In the case of MDS, you are trying to model the distances. Hence, the most obvious choice for a goodness-of-fit statistic is one based on the differences between the actual distances and their predicted values. Such a measure is called stress and is calculated as values:

$$stree = \sqrt{\frac{\sum(d_{ij} - d_{ij})^2}{d_{ij}}^2}$$

In his original paper on MDS, Kruskal (1964) gave following advise about stress values based on his experience: [4]

Stress	Goodness- of-fit
0.200	poor
0.100	fair
0.050	good
0.025	excellent
0.000	perfect

Table 5.1: Criteria to assess GoF for cMDS

R provides functions for both classical and non-metric multidimensional scaling. Assume that we have N objects measured on p numeric variables. We want to represent the distances among the objects in a parsimonious (and visual) way (i.e., a lower k-dimensional space).

Suggested reading:

Title : "Data mining and analysis of scientific research data records on Covid-19 mortality, immunity, and vaccine development - In the first wave of the Covid-19 pandemic" URL : https://www.sciencedirect.com/science/article/pii/S1871402120302332

[4]Visit https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Multidimensional_Scaling.pdf for more information.

5.5.1 Metric/Classical MDS

It is a super-set of classical MDS that generalizes the optimization procedure to a variety of loss functions and input matrices of known distances with weights and so on. A useful loss function in this context is called stress, which is often minimized using a procedure called *stress majorization.* Metric MDS minimizes the cost function called "stress" which is a residual sum of squares:

$$\text{Stress}_D(x_1, x_2, ..., x_N) = \sqrt{\sum_{i \neq j = 1,...,N} \left(d_{ij} - \|x_i - x_j\|\right)^2}.$$

Metric scaling uses a power transformation with a user-controlled exponent p: d_{ij}^p and $-d_{ij}^{2p}$ for distance. In classical scaling $p = 1$. Non-metric scaling is defined by the use of isotonic regression to non-parametrically estimate a transformation of the dissimilarities.

Classical MDS

It is also known as Principal Coordinates Analysis (PCoA), Torgerson Scaling or TorgersonGower scaling. It takes an input matrix giving dissimilarities between pairs of items and outputs a coordinate matrix whose configuration minimizes a loss function called strain. [58] For example, given the Euclidean aerial distances d_{ij} between various cities indexed by i and j, you want to find the coordinates (x_i, y_i) of the cities such that $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. In this example, an exact solution is possible (assuming the Euclidean distances are exact). In practice, this is usually not the case, and MDS therefore seeks to approximate the lower-dimensional representation by minimising a loss function. General forms of loss functions are called stress in distance MDS and strain in classical MDS. The strain is given by:

$$\text{Strain}_D(x_1, x_2, ..., x_N) = \left(\frac{\sum_{i,j} \left(b_{ij} - x_i^T x_j\right)^2}{\sum_{i,j} b_{ij}^2}\right)^{1/2},$$

where x_i now denote vectors in N-dimensional space, $x_i^T x_j$ denotes the scalar product between x_i and x_j, and b_{ij} are the elements of the matrix B defined on step 2 of the following algorithm, which are computed from the distances.

Steps of a Classical MDS algorithm

Classical MDS uses the fact that the coordinate matrix X can be derived by eigenvalue decomposition from $B = XX'$. And the matrix B can be computed from proximity matrix D by using double centering.

1. Set up the squared proximity matrix $D^{(2)} = [d_{ij}^2]$.
2. Apply double centering: $B = -\frac{1}{2}CD^{(2)}C$ using the centering matrix $C = I - \frac{1}{n}J_n$, where n is the number of objects, I is the $n \times n$ identity matrix, and J_n is an $n \times n$ matrix of all ones.
3. Determine the m largest eigenvalues $\lambda_1, \lambda_2, ..., \lambda_m$ and corresponding eigen-vectors $e_1, e_2, ..., e_m$ of B (where m is the number of dimensions desired for the output).
4. Now, $X = E_m \Lambda_m^{1/2}$, where E_m is the matrix of m eigenvectors and Λ_m is the diagonal matrix of m eigenvalues of B.

Classical MDS can be performed using the `cmdscale()` function. Following snippet shows summaries for Covid-19 dataset.

```
> coviddata <- read.csv(paste("D:/Work/Books/R/MultiVarAnal/Data/covid19.csv", sep = ""))
> summary(coviddata)
```

```
     Sate           Active.Cases        Cured            Deaths     
 Length:35          Min.   :   393   Min.   :   461   Min.   :    0  
 Class :character   1st Qu.:  1420   1st Qu.:  3006   1st Qu.:   32  
 Mode  :character   Median :  9442   Median : 25205   Median :  252  
                    Mean   : 20124   Mean   : 68702   Mean   : 1668  
                    3rd Qu.: 20890   3rd Qu.: 78627   3rd Qu.: 1188  
                    Max.   :168443   Max.   :502490   Max.   :22465  
```

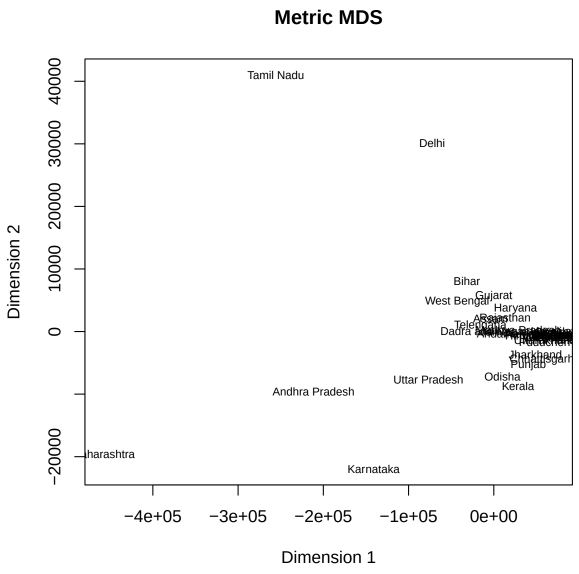
Classical MDS assumes Euclidean distances. So this is not applicable for direct dissimilarity ratings. Following snippet shows as to how classical MDS can be performed in R using Covid-19 dataset.

```
> d <- dist(coviddata[, 2:4])
> fit <- cmdscale(d, eig=TRUE, k=3)
> fit
```

```
$points
              [,1]         [,2]        [,3]
 [1,]   69379.38438   -342.009821   562.24814
 [2,] -211532.29585  -9619.861818 -5467.84985
 [3,]   69030.06222   -344.069482   520.97115
 [4,]   -3920.65202   2121.300184 -1450.17854
 [5,]  -31560.67920   8104.332003 -1806.41002
 [6,]   69644.00848  -1019.612156   542.90493
 [7,]   56346.36186  -4450.503248   137.18359
 [8,]   69782.48559    -59.822322   554.46184
 [9,]  -72334.34475  30171.807280  1667.70936
[10,]   60271.90742    -46.147938   397.92178
[11,]      73.19357   5716.836828  1495.18319
[12,]   25390.02419   3651.439212   -86.46819
[13,]   67748.09936   -598.380877   492.93680
[14,]   45374.04539      9.606117   401.88114
[15,]   48814.05880  -3661.046080    55.81435
[16,] -140980.87569 -21996.302779 -2317.42256
[17,]   28554.42745  -8668.745815  -906.67723
[18,]   69927.24664   -517.451693   559.99320
[19,]   28662.32826     40.471293   493.10554
[20,] -458464.81589 -19598.231102  5342.31537
[21,]   67584.63474   -693.267704   476.54135
[22,]   70515.39010  -1096.369572   538.28222
[23,]   71024.58289   -531.350780   571.34975
[24,]   68938.17051   -763.916841   503.58988
[25,]   10193.21903  -7135.567969 -1239.66991
[26,]   63883.64228  -1818.563818   451.23404
[27,]   40481.32200  -5391.086955   514.15379
[28,]   13247.33021   2082.717796  -188.21831
[29,]   70566.28345   -413.251794   564.92745
[30,] -255655.64682  41043.050259 -1397.35908
[31,]  -15814.00765    976.131002 -1361.11489
[32,]   64689.81762   -959.349844   428.59195
[33,]   59890.82232  -1320.676252   388.82622
[34,]  -76744.48058  -7682.189543 -1464.94247
[35,]  -43005.05031   4810.084231    24.18402
$eig
 [1]  4.348008e+11  3.957915e+09  8.709485e+07  9.330910e-05  3.151277e-05
 [6]  2.072057e-05  1.718355e-05  1.633135e-05  1.360152e-05  1.089191e-05
[11]  8.032513e-06  7.965913e-06  7.911800e-06  7.643109e-06  4.922343e-06
[16]  2.826598e-06  1.172814e-06  3.711233e-07  8.853394e-08 -1.470212e-07
[21] -2.383580e-06 -2.989610e-06 -3.498419e-06 -4.241086e-06 -4.503812e-06
[26] -4.776480e-06 -7.181333e-06 -9.130648e-06 -1.337912e-05 -1.372633e-05
[31] -1.470755e-05 -2.527781e-05 -3.014738e-05 -3.912410e-05 -7.327016e-05
$x
NULL
$ac
[1] 0
$GOF
[1] 1 1
```

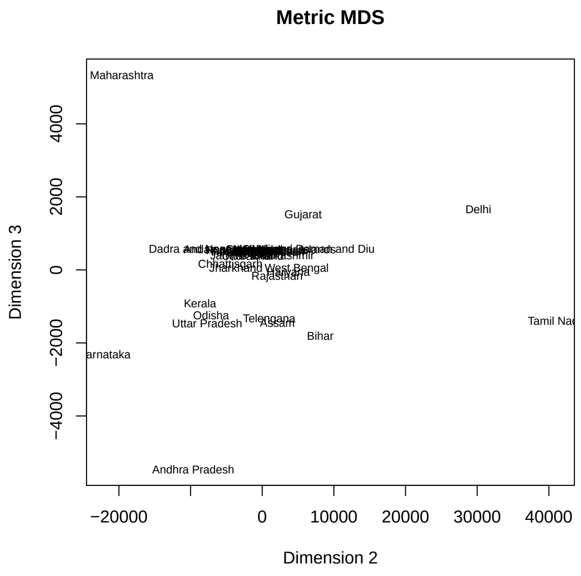
Plot solution for Dim 1 and Dim 2.

```
> x <- fit$points[, 1]
> y <- fit$points[, 2]
> plot(x, y, xlab="Dimension 1", ylab="Dimension 2",
+   main="Metric MDS", type="n")
> #text(x, y, labels = row.names(coviddata), cex=.7)
> text(x, y, labels = coviddata[, 1], cex=.7)
```



Plot solution for Dim2 and Dim 3.

```
> x <- fit$points[, 2]
> y <- fit$points[, 3]
> plot(x, y, xlab="Dimension 2", ylab="Dimension 3",
+   main="Metric MDS", type="n")
> #text(x, y, labels = row.names(coviddata), cex=.7)
> text(x, y, labels = coviddata[, 1], cex=.7)
$
```



5.5.2 Non-Metric MDS

In contrast to metric MDS, non-metric MDS finds both a non-parametric monotonic relationship between the dissimilarities in the item-item matrix and the Euclidean distances between items, and the location of each item in the low-dimensional space. The relationship is typically found using isotonic regression: let x denote the vector of proximities, $f(x)$ a monotonic transformation of x, and d the point distances; then coordinates have to be found, that minimize the so-called stress,

$$\text{Stress} = \sqrt{\frac{\sum \left(f(x) - d\right)^2}{\sum d^2}}.$$

A few variants of this cost function exist. MDS programs automatically minimize stress in order to obtain the MDS solution. The core of a non-metric MDS algorithm is a twofold optimization process. First the optimal monotonic transformation of the proximities has to be found. Secondly, the points of a configuration have to be optimally arranged, so that their distances match the scaled proximities as closely as possible. The basic steps in a non-metric MDS algorithm are:

1. Find a random configuration of points, e. g. by sampling from a normal distribution.

2. Calculate the distances d between the points.

3. Find the optimal monotonic transformation of the proximities, in order to obtain optimally scaled data $f(x)$.

4. Minimize the stress between the optimally scaled data and the distances by finding a new configuration of points.

5. Compare the stress to some criterion. If the stress is small enough then exit the algorithm else return to 2.

Louis Guttman's smallest space analysis (SSA) is an example of a non-metric MDS procedure.

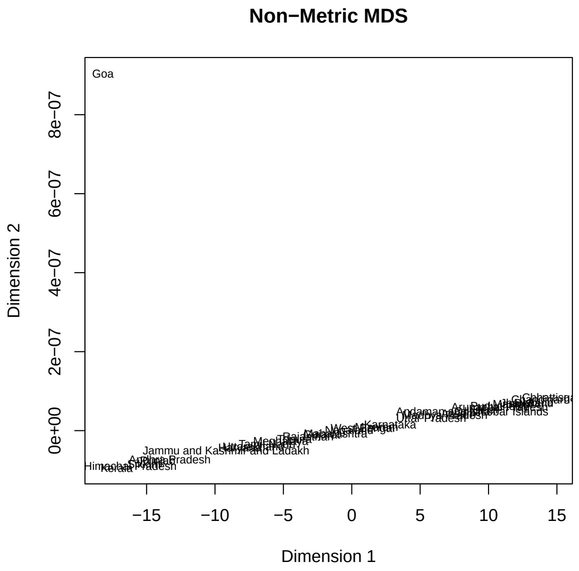
The data set used for this analysis is rather measured in ordinal scale (ranked data). So it is suitable for Non-metric MDS.

```
> indianpoverty <- read.csv(paste("D:/Work/Books/R/MultiVarAnal/Data/indianpoverty_ranked.csv", sep = ""))
> names(indianpoverty)

[1] "State"   "Poverty" "Income"

> summary(indianpoverty)

    State              Poverty          Income
 Length:32          Min.   : 1.00   Min.   : 1.000
 Class :character   1st Qu.: 6.75   1st Qu.: 4.750
 Mode  :character   Median :15.50   Median : 7.000
                    Mean   :15.81   Mean   : 7.188
                    3rd Qu.:23.25   3rd Qu.: 9.500
                    Max.   :34.00   Max.   :19.000
```

```
> d <- dist(indianpoverty[, 2])
> fit <- cmdscale(d, eig=TRUE, k=2)
> fit

$points
           [,1]          [,2]
 [1,] -18.1875  9.048111e-07
 [2,] -17.1875 -9.484685e-08
 [3,] -16.1875 -8.932849e-08
 [4,] -15.1875 -8.381013e-08
 [5,] -14.1875 -7.829176e-08
 [6,] -13.1875 -7.277340e-08
 [7,]  -9.1875 -5.069995e-08
 [8,]  -8.1875 -4.518159e-08
 [9,]  -7.1875 -3.966323e-08
[10,]  -6.1875 -3.414487e-08
[11,]  -5.1875 -2.862650e-08
[12,]  -4.1875 -2.310814e-08
[13,]  -3.1875 -1.758978e-08
[14,]  -2.1875 -1.207142e-08
[15,]  -1.1875 -6.553055e-09
```

```
[16,]  -0.1875 -1.034693e-09
[17,]   0.8125  4.483669e-09
[18,]   1.8125  1.000203e-08
[19,]   2.8125  1.552039e-08
[20,]   5.8125  3.207548e-08
[21,]   6.8125  3.759384e-08
[22,]   7.8125  4.311221e-08
[23,]   8.8125  4.863057e-08
[24,]   9.8125  5.414893e-08
[25,]  10.8125  5.966729e-08
[26,]  11.8125  6.518565e-08
[27,]  12.8125  7.070402e-08
[28,]  14.8125  8.174074e-08
[29,]   8.8125  4.863057e-08
[30,]  10.8125  5.966729e-08
[31,]  12.8125  7.070402e-08
[32,]  13.8125  7.622238e-08
$eig
 [1]  3.312875e+03  9.094947e-13  1.542703e-14  1.098080e-14  6.655461e-15
 [6]  5.222297e-15  3.443514e-15  3.010069e-15  2.614184e-15  1.805111e-15
[11]  1.324417e-15  9.039237e-16  8.447577e-16  8.143609e-16  2.689242e-16
[16]  4.342534e-30 -9.001270e-31 -1.308673e-17 -4.690867e-17 -1.050371e-16
[21] -1.846287e-16 -5.406627e-16 -6.140540e-16 -8.395124e-16 -1.278795e-15
[26] -1.974773e-15 -2.007523e-15 -2.553419e-15 -4.176164e-15 -4.955027e-15
[31] -2.989017e-14 -3.884943e-13
$x
NULL
$ac
[1] 0
$GOF
[1] 1 1
```

```
> x <- fit$points[, 1]
> y <- fit$points[, 2]
> plot(x, y, xlab="Dimension 1", ylab="Dimension 2",
+   main="Non-Metric MDS", type="n")
> #text(x, y, labels = row.names(coviddata), cex=.7)
> text(x, y, labels = indianpoverty[, 1], cex=.7)
$
```



5.6 Correspondence Analysis

Correspondence analysis (CA) or *reciprocal averaging* is a multivariate statistical technique proposed by Herman Otto Hartley and later developed by Jean-Paul Benzécri. It is conceptually similar to principal component analysis, but applies to categorical rather than continuous data. In a similar manner to principal component analysis, it provides a means of displaying or summarising a set of data in two-dimensional graphical form.

All data should be on the same scale for CA to be applicable, keeping in mind that the method treats rows and columns equivalently. It is traditionally applied to contingency tables - CA decomposes the chi-squared statistic associated with this table into orthogonal factors. Because CA is a descriptive technique, it can be applied to tables whether or not the χ^2 statistic is appropriate.

Correspondence analysis provides a graphic method of exploring the relationship between variables in a *contingency table*. There are many options for correspondence analysis in R. *ca* package by Nenadic and Greenacre supports supplimentary points, subset analyses, and comprehensive graphics. It is possible to obtain the package from CRAN.

5.6.1 Simple correspondence analysis

Correspondence analysis can be done using *ca* package using the command `ca`.

```
> satisfaction <- sample(c("satisfied", "indifferent",
+                          "dissatisfied"), 10, replace = TRUE)
> perception <- sample(c("reasonable", "don't know",
+                        "irreasonable"), 10, replace = TRUE)
> tab <- table(satisfaction, perception)
> tab
```

```
              perception
satisfaction   don't know irreasonable reasonable
  dissatisfied          2            1          0
  indifferent           2            2          0
  satisfied             1            1          1
```

```
> library(ca)
> prop.table(tab, 1) # row percentages
```

```
              perception
satisfaction   don't know irreasonable reasonable
  dissatisfied  0.6666667    0.3333333  0.0000000
  indifferent   0.5000000    0.5000000  0.0000000
  satisfied     0.3333333    0.3333333  0.3333333
```

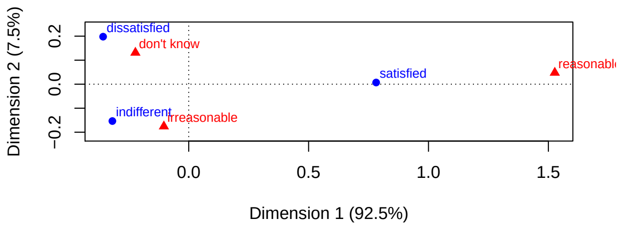
```
> prop.table(tab, 2) # col percentages
```

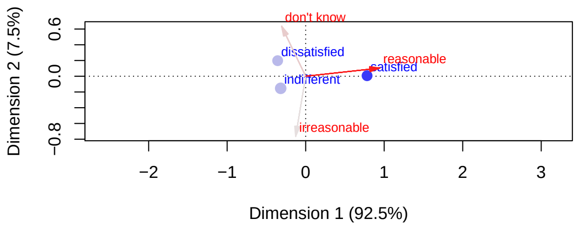
```
              perception
satisfaction   don't know irreasonable reasonable
  dissatisfied       0.40         0.25       0.00
  indifferent        0.40         0.50       0.00
  satisfied          0.20         0.25       1.00
```

```
> fit <- ca(tab)
> #summary(fit)
```

Plots

```
> par(mfrow = c(2, 1))
> plot(fit)
> plot(fit, mass = TRUE, contrib = "absolute", map =
+    "rowgreen", arrows = c(FALSE, TRUE)) # asymmetric map
```





The first graph is the standard symmetric representation of a simple correspondence analysis with rows and column represented by points. Row points (column points) that are closer together have more similar column profiles (row profiles). Keep in mind that you can not interpret the distance between row and column points directly.

The second graph is asymmetric , with rows in the principal coordinates and columns in reconstructions of the standarized residuals. Additionally, mass is represented by points and columns are represented by arrows. Point intensity (shading) corresponds to the absolute contributions for the rows. This example is included to highlight some of the available options. [5] [6]

[5]Read about CA on (default `author` data at https://www.gastonsanchez.com/visually-enforced/how-to/2012/07/19/Correspondence-Analysis/)

[6]Read about CA on FTA at https://programminghistorian.org/en/lessons/correspondence-analysis-in-R.

5.6.2 MCA

Multiple correspondence analysis (MCA) is a data analysis technique for *nominal categorical data*, used to detect and represent underlying structures in a data set. It does this by representing data as points in a low-dimensional Euclidean space. The procedure thus appears to be the counterpart of principal component analysis for categorical data. MCA can be viewed as an extension of simple correspondence analysis (CA) in that it is applicable to a large set of categorical variables. More technically, Multiple CA is done by either performing an eigenvalue decomposition on the *Burt-matrix*, or by performing a singular value decomposition on the i*ndicator matrix*. In the standard case, MCA is based on the *SVD* of the indicator matrix. The Burt table is the result of the inner product of a design or indicator matrix, and the multiple correspondence analysis results are identical to the results you would obtain for the column points from a simple correspondence analysis of the indicator or design matrix.

Suggested reading:

Title : "Multiple categorical variables: Multiple Correspondence Analysis" URL : http://www.econ.upf.edu/~michael/vienna/CARME5_BW.pdf

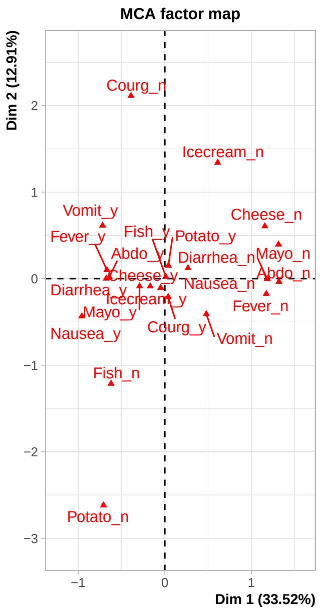
7

Following code snippet shows as to how MCA is performed on certain data set called "*poison*". Use `help('poison')` for more details. [59]

```
> library(FactoMineR)
> data(poison)
> poisonForAnal <- poison[1:55, 5:15]
> resMca <- MCA(poisonForAnal, graph=TRUE)
> summary(resMca)

Call:
MCA(X = poisonForAnal, graph = TRUE)
Eigenvalues
                       Dim.1   Dim.2   Dim.3   Dim.4   Dim.5   Dim.6   Dim.7
Variance               0.335   0.129   0.107   0.096   0.079   0.071   0.060
% of var.             33.523  12.914  10.735   9.588   7.883   7.109   6.017
Cumulative % of var.  33.523  46.437  57.172  66.760  74.643  81.752  87.769
                       Dim.8   Dim.9  Dim.10  Dim.11
```

[7]Read certain useful material on MCA at http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/114-mca-multiple-correspondence-analysis-in-r-essentials/.

```
Variance               0.056   0.041   0.013   0.012
% of var.              5.577   4.121   1.304   1.229
Cumulative % of var.  93.346  97.467  98.771 100.000
Individuals (the 10 first)
               Dim.1    ctr   cos2    Dim.2    ctr   cos2    Dim.3    ctr   cos2
1           | -0.453  1.111  0.347 | -0.264  0.982  0.118 |  0.172  0.498  0.050
2           |  0.836  3.792  0.556 | -0.032  0.014  0.001 | -0.072  0.088  0.004
3           | -0.448  1.089  0.548 |  0.135  0.258  0.050 | -0.225  0.856  0.138
4           |  0.880  4.204  0.748 | -0.085  0.103  0.007 | -0.021  0.007  0.000
5           | -0.448  1.089  0.548 |  0.135  0.258  0.050 | -0.225  0.856  0.138
6           | -0.359  0.701  0.025 | -0.436  2.677  0.037 | -1.209 24.770  0.281
7           | -0.448  1.089  0.548 |  0.135  0.258  0.050 | -0.225  0.856  0.138
8           | -0.641  2.226  0.615 | -0.005  0.000  0.000 |  0.113  0.216  0.019
9           | -0.453  1.111  0.347 | -0.264  0.982  0.118 |  0.172  0.498  0.050
10          | -0.141  0.107  0.039 |  0.122  0.209  0.029 | -0.227  0.872  0.101
1           |
2           |
3           |
4           |
5           |
6           |
7           |
8           |
9           |
10          |
Categories (the 10 first)
              Dim.1    ctr   cos2 v.test    Dim.2    ctr   cos2 v.test    Dim.3
Nausea_n    |  0.267  1.516  0.256  3.720 |  0.121  0.811  0.053  1.689 | -0.266
Nausea_y    | -0.958  5.432  0.256 -3.720 | -0.435  2.906  0.053 -1.689 |  0.952
Vomit_n     |  0.479  3.734  0.344  4.311 | -0.409  7.072  0.251 -3.683 |  0.084
Vomit_y     | -0.719  5.601  0.344 -4.311 |  0.614 10.608  0.251  3.683 | -0.127
Abdo_n      |  1.318 15.418  0.845  6.755 | -0.036  0.029  0.001 -0.183 | -0.005
Abdo_y      | -0.641  7.500  0.845 -6.755 |  0.017  0.014  0.001  0.183 |  0.002
Fever_n     |  1.172 13.541  0.785  6.509 | -0.175  0.783  0.017 -0.972 |  0.097
Fever_y     | -0.670  7.738  0.785 -6.509 |  0.100  0.447  0.017  0.972 | -0.056
Diarrhea_n  |  1.183 13.797  0.799  6.570 | -0.003  0.000  0.000 -0.015 | -0.083
Diarrhea_y  | -0.676  7.884  0.799 -6.570 |  0.002  0.000  0.000  0.015 |  0.047
              ctr   cos2 v.test
Nausea_n    4.670  0.253 -3.694 |
Nausea_y   16.734  0.253  3.694 |
Vomit_n     0.363  0.011  0.760 |
Vomit_y     0.544  0.011 -0.760 |
Abdo_n      0.001  0.000 -0.026 |
Abdo_y      0.000  0.000  0.026 |
Fever_n     0.291  0.005  0.540 |
Fever_y     0.167  0.005 -0.540 |
Diarrhea_n  0.212  0.004 -0.461 |
Diarrhea_y  0.121  0.004  0.461 |
Categorical variables (eta2)
              Dim.1 Dim.2 Dim.3
Nausea      | 0.256 0.053 0.253 |
Vomiting    | 0.344 0.251 0.011 |
Abdominals  | 0.845 0.001 0.000 |
Fever       | 0.785 0.017 0.005 |
Diarrhae    | 0.799 0.000 0.004 |
Potato      | 0.029 0.396 0.264 |
Fish        | 0.007 0.027 0.252 |
Mayo        | 0.383 0.035 0.039 |
Courgette   | 0.015 0.446 0.053 |
Cheese      | 0.194 0.053 0.012 |
```



MCA is useful to detect the influence of outlying points, the scaling of row and column coordinates in the maps, whether solutions should be rotated, the statistical significance of the results, and the "horseshoe" effect. [8]

5.6.3 MJCA

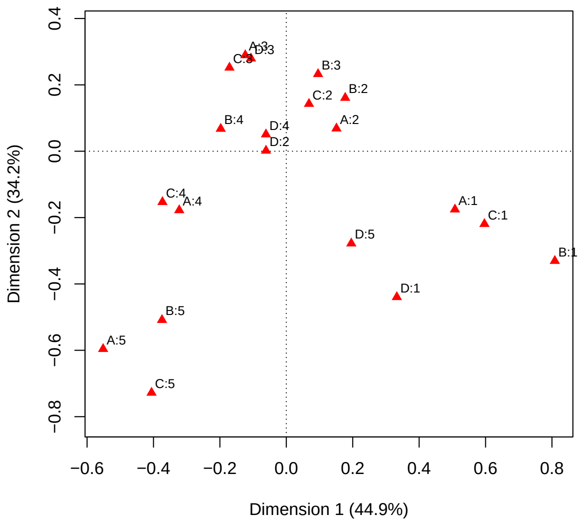
Multiple and Joint Correspondence Analysis (MJCA) can be done with `lambda="JCA"` as argument inside the command `mjca` (*ca* package). This gives a joint correspondence analysis, which uses an iterative algorithm that optimally fits the off-diagonal submatrices of the Burt matrix. The JCA solution does not have strictly nested dimensions, so the percentage of inertia explained is given for the whole solution of chosen dimensionality, not for each dimension, but this percentage is optimal. In simple words, *MJCA give solution which is optimal solution for chosen dimensionality.*

[8]Greenacre, M. TYING UP THE LOOSE ENDS IN SIMPLE, MULTIPLE AND JOINT CORRESPONDENCE ANALYSIS. Available at http://www.econ.upf.edu/~michael/work/LooseEndsCOMPSTAT.pdf.

```
> data("wg93")
> out <- mjca(wg93[,1:4])
> summary(out)
```

```
Principal inertias (eigenvalues):
 dim    value      %   cum%   scree plot
 1      0.076455  44.9  44.9  *************
 2      0.058220  34.2  79.1  **********
 3      0.009197   5.4  84.5  **
 4      0.005670   3.3  87.8  *
 5      0.001172   0.7  88.5
 6      7e-06000   0.0  88.5
        -------- -----
 Total: 0.170246
Columns:
     name   mass  qlt  inr    k=1 cor ctr    k=2 cor ctr
1  |  A:1 |   34  963   55 |  508 860 115 | -176 103  18 |
2  |  A:2 |   92  659   38 |  151 546  28 |   69 113   7 |
3  |  A:3 |   59  929   47 | -124 143  12 |  289 786  84 |
4  |  A:4 |   51  798   50 | -322 612  69 | -178 186  28 |
5  |  A:5 |   14  799   60 | -552 369  55 | -596 430  84 |
6  |  B:1 |   20  911   62 |  809 781 174 | -331 131  38 |
7  |  B:2 |   50  631   47 |  177 346  21 |  161 285  22 |
8  |  B:3 |   59  806   45 |   96 117   7 |  233 690  55 |
9  |  B:4 |   81  620   41 | -197 555  41 |   68  65   6 |
10 |  B:5 |   40  810   60 | -374 285  74 | -509 526 179 |
11 |  C:1 |   44  847   60 |  597 746 203 | -219 101  36 |
12 |  C:2 |   91  545   38 |   68 101   6 |  143 444  32 |
13 |  C:3 |   57  691   48 | -171 218  22 |  252 473  62 |
14 |  C:4 |   44  788   52 | -373 674  80 | -153 114  18 |
15 |  C:5 |   15  852   60 | -406 202  32 | -728 650 136 |
16 |  D:1 |   17  782   56 |  333 285  25 | -440 497  57 |
17 |  D:2 |   67  126   42 |  -61 126   3 |    2   0   0 |
18 |  D:3 |   58  688   48 | -106  87   9 |  280 601  78 |
19 |  D:4 |   65  174   43 |  -61 103   3 |   51  71   3 |
20 |  D:5 |   43  869   50 |  196 288  22 | -278 581  57 |
```

```
> plot(out)
```



5.7 Exploratory Factor Analysis

Exploratory factor analysis (EFA) is a statistical method used to uncover the underlying structure of a relatively large set of variables. EFA is a technique within factor analysis whose overarching goal is to *identify the underlying relationships between measured variables*. It is commonly used by researchers when *developing a scale* (a scale is a collection of questions used to measure a particular research topic) and serves to *identify a set of latent constructs* underlying a battery of measured variables. It should be used when the researcher has *no a priori hypothesis about factors or patterns of measured variables*. Measured variables are any one of several attributes of people that may be observed and measured. Examples of measured variables could be the physical height, weight, and pulse rate of a human being. Usually, researchers would have a large number of measured variables, which are assumed to be related to a smaller number of *unobserved factors*. Researchers must carefully consider the number of measured variables to include in the analysis. EFA procedures are more accurate when each factor is represented by multiple measured variables in the analysis.

5.7.1 Methodology

Suppose we have a set of p observable random variables, $x_1, \ldots, x_p$ with means $\mu_1, \ldots, \mu_p$. Suppose for some unknown constants l_{ij} and k unobserved random variables F_j (called "common factors," because they influence all the observed random variables), where $i \in 1, \ldots, p$ and $j \in 1, \ldots, k$, where $k < p$, we have that the terms in each random variable (as a difference from that variable's mean) should be writeable as a linear combination of the common factors :

$$x_i - \mu_i = l_{i1}F_1 + \cdots + l_{ik}F_k + \varepsilon_i$$

Here, the ε_i are unobserved stochastic error terms with zero mean and finite variance, which may not be the same for all i. In matrix terms, we have

$$x - \mu = LF + \varepsilon$$

Assumptions on F:

1. F and $\varepsilon$are independent.
2. E(F)=0. (E is Expectation)
3. Cov(F) = I (Cov is the cross-covariance matrix, to make sure that the factors are uncorrelated).

Any solution of the above set of equations following the constraints for F is defined as the factors, and L as the loading matrix.

5.7.2 Internal consistency

Internal consistency is typically a measure based on the correlations between different items on the same test. It measures whether several items that propose to measure the same general construct produce similar scores. Internal consistency is usually measured with *Cronbach's alpha*, a statistic calculated from the pairwise correlations between items. Internal consistency ranges between negative infinity and one. Coefficient alpha will be negative whenever there is greater within-subject variability than between-subject variability. A commonly accepted rule of thumb for describing internal consistency is as follows:

Cronbach's alpha	Internal consistency
$0.9 \leq \alpha$	Excellent
$0.8 \leq \alpha < 0.9$	Good
$0.7 \leq \alpha < 0.8$	Acceptable
$0.6 \leq \alpha < 0.7$	Questionable
$0.5 \leq \alpha < 0.6$	Poor
$\alpha < 0.5$	Unacceptable

Table 5.2: Criteria for alpha interpretation

Cronbach's alpha can be computed using function `alpha()` available in the package *pysch*.

```
> library(psych)
> summary(alpha(multivardata[, 5:13]))[1]

Reliability analysis
 raw_alpha std.alpha G6(smc) average_r S/N  ase mean sd median_r
      0.87      0.88    0.98      0.45 7.2 0.04  2.8  1     0.43
 raw_alpha
 0.8727446
```

5.7.3 Statistical diagnosis

EFA requires two fundamental tests viz. (1) Bartlett's test and (2) KMO test.

Bartlett's test, named after Maurice Stevenson Bartlett, is used to test homoscedasticity, that is, if multiple samples are from populations with equal variances. Some statistical tests, such as the analysis of variance, assume that variances are equal across groups or samples, which can be verified with Bartlett's test. In a Bartlett test, we construct the null and alternative hypothesis. For this purpose several test procedures have been devised. The test procedure due to M.S.E (Mean Square Error) is most wide used procedure. This test procedure is based on the statistic whose sampling distribution is approximately a Chi-Square distribution with (k 1) degrees of freedom, where k is the number of random samples, which may vary in size and are each drawn from independent normal distributions. Bartlett's test is sensitive to departures from normality. That is, if the samples come from non-normal distributions, then Bartlett's test may simply be testing for non-normality. Levene's test and the BrownForsythe test are alternatives to the Bartlett test that are less sensitive to departures from normality.

Bartlett's test is used to test the null hypothesis, H0 that all k population variances are equal against the alternative that at least two are different. If there are k samples with sizes n_i and sample variances S_i^2 then Bartlett's test statistic is

$$\chi^2 = \frac{(N-k)\ln(S_p^2) - \sum_{i=1}^{k}(n_i-1)\ln(S_i^2)}{1+\frac{1}{3(k-1)}\left(\sum_{i=1}^{k}(\frac{1}{n_i-1}) - \frac{1}{N-k}\right)}$$

where $N = \sum_{i=1}^{k} n_i$ and $S_p^2 = \frac{1}{N-k}\sum_i (n_i-1)S_i^2$ is the pooled estimate for the variance.

The test statistic has approximately a χ^2_{k-1} distribution. Thus, the null hypothesis is rejected if $\chi^2 > \chi^2_{k-1,\alpha}$ (where $\chi^2_{k-1,\alpha}$ is the upper tail critical value for the χ^2_{k-1} distribution). Bartlett's test is a modification of the corresponding likelihood ratio test designed to make the approximation to the χ^2_{k-1} distribution better.

Why Bartlett's Test is required for EFA? Actually, the Bartlett's Test is required to test the Sphericity in the data, which requires comparing observed correlation matrix of sample data against identity matrix. Essentially it checks to see if there is a certain redundancy between the variables that we can summarize with a few number of factors. The null hypothesis of the test is that the variables are orthogonal, i.e. not correlated.

KaiserMeyerOlkin (KMO) is a statistical measure to determine how suited data is for factor analysis. The test measures sampling adequacy for each variable in the model and the complete model. The statistic is a measure of the proportion of variance among variables that might be common variance. Henry Kaiser introduced a Measure of Sampling Adequacy (MSA) of factor analytic data matrices in 1970. [60] Kaiser and Rice then modified it in 1974. [61] The measure of sampling adequacy is calculated for each indicator as

$$MSA_j = \frac{\sum\limits_{k\neq j} r_{jk}^2}{\sum\limits_{k\neq j} r_{jk}^2 + \sum\limits_{k\neq j} p_{jk}^2}$$

and indicates to what extent an indicator is suitable for a factor analysis. These tests can be done using a package called *psych* in R. The *psych* package has two functions `cortest.bartlett()` and `KMO()` for Bartlett's test and KMO test respectively.

```
> cortest.bartlett(cor(multivardata[, 5:13]))

$chisq
[1] 1562.579
$p.value
[1] 2.120978e-305
\$df
[1] 36

> KMO(multivardata[, 5:13])

Kaiser-Meyer-Olkin factor adequacy
Call: KMO(r = multivardata[, 5:13])
Overall MSA =  0.65
MSA for each item =
sat1 sat2 sat3 per1 per2 per3 att1 att2 att3
0.56 0.96 0.57 0.61 0.78 0.58 0.84 0.59 0.62
```

Suggested reading:

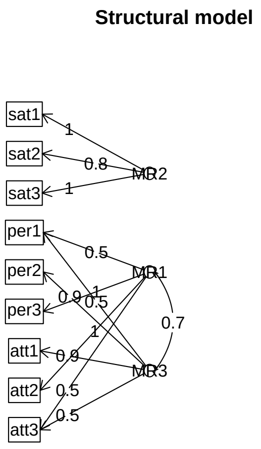
Title : "R practice: Factor analysis *by* Minato Nakazawa"
[9] URL : https://minato.sip21c.org/swtips/factor-in-R.pdf

5.7.4 Structure diagram

The *psych* package provides several functions to make visuals for factor analysis. For instance, the function `structure.diagram()` yields nice visuals for EFA outputs.

```
> faout <- fa(cor(multivardata[, 5:13]), 3)
> structure.diagram(faout)
```

[9]His profile is available at https://minato.sip21c.org/index-e.html



5.8 Confirmatory factor analysis

confirmatory factor analysis (CFA) is a special form of factor analysis, most commonly used in social research. It is used to test whether measures of a construct are consistent with a researcher's understanding of the nature of that construct (or factor). As such, the objective of confirmatory factor analysis is to test whether the data fit a hypothesized measurement model. This hypothesized model is based on theory and/or previous analytic research. CFA was first developed by Jöreskog.

In confirmatory factor analysis, the researcher first develops a hypothesis about what factors they believe are underlying the measures used and may impose constraints on the model based on these a priori hypotheses. By imposing these constraints, the researcher is forcing the model to be consistent with their theory. Model fit measures could then be obtained to assess how well the proposed model captured the covariance between all the items or measures in the model. If the constraints the researcher has imposed on the model are inconsistent with the sample data, then the results of statistical tests of model fit will indicate a poor fit, and the model will be rejected. If the fit is poor, it may be due to some items measuring multiple factors. It might also be that some items within a factor are more related to each other than others.

Suggested reading:

Title : “Confirmatory factor analysis using Microsoft Excel” URL : https://link.springer.com/content/pdf/10.3758/BF03192739.pdf

5.8.1 Maximum Likelihood Estimation

The onus of the analyst is to solve a set of equations for a given objective function using MLE.

$$F_{ml} = log|\sum| + tr(S\sum^{1}) - log|S| - p$$

Given a model

$$x = \Lambda\xi + \delta$$

Model fit

$$\chi^2 = F_{ml} * (n-1)$$

The above measure follow χ^2 distribution with $\frac{k(k-1)}{2} - 2 * k$; where k represents number of variables and p represents number of parameters.

Decision related to fitness is taken based on

1. χ^2 statistic (using P Value)
2. RMSEA ($RMSEA \leq 0.05 - Closefit; 0.05 < RMSEA \leq 0.08 - Reasonablefit; RMSEA > 0.1 - Poorfi$)
3. Fit indices (e.g. GFI, NFI, CLI, TLI and many more; all indices must bemore than or at least close to 0.09)

5.8.2 SEM using `Political Democracy` dataset

R has few packages for SEM. Two of the packages that widely used by analysts are (1) *sem*, (2) *lavaan*. It is possible to produce visuals (path diagrams) together with a package called *SemPaths*. We will try *lavaan* in this section. However, there is abundant of documentation available on SEM in R.

The lavaan package is developed to provide *useRs*, researchers and teachers a free open-source, but commercial-quality package for latent variable modeling. You can use lavaan to estimate a large variety of multivariate statistical models, including path analysis, confirmatory factor analysis, structural equation modeling and growth curve models. Install the package using `install.pacakges('lavaan')` and start using it. [62] Following code is SEM analysis done on certain dataset called *Politcal Democracy data*. This dataset is available strait after installing *lavaan* package in R.

```
model <- '
  # latent variable definitions
    ind60 =~ x1 + x2 + x3
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ ind60
    dem65 ~ ind60 + dem60
  # residual (co)variances
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
'
fit <- sem(model,
           data = PoliticalDemocracy)

coef(fit)
```

The above code was executed in RStudio script pane. The summary of fitness can be accessed using `summary()` function.

```
> summary(fit)
lavaan 0.6-9 ended normally after 68 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                          31

  Number of observations                              75

Model Test User Model:

  Test statistic                                  38.125
  Degrees of freedom                                  35
  P-value (Chi-square)                             0.329
```

The above output shows the data fits to model defined in 5.8.2. It is possible to retrieve output in more user readable fashion using the below code.

```
fit_summary <- data.frame(summary(fit)\$PE)
typeof(fit_summary)
head(fit_summary)
```

Lavaan has the capability to compute close to a 40 *fit measures*. Analysts are so crazy for these fit measures.

```
head(data.frame(fitMeasures(fit)))
               fitMeasures.fit.
npar                 31.0000000
fmin                  0.2541681
chisq                38.1252182
df                   35.0000000
pvalue                0.3291804
baseline.chisq      730.6540854
> dim(data.frame(fitMeasures(fit)))
[1] 42  1
```

We can plo the path diagram using the following procedure.

```
library('semPlot')
semPaths(fit,"std",edge.label.cex=0.5, curvePivot = TRUE
    ↪ )
```

5.8.3 SEM using `multivardata.csv`

This section shows performing SEM on *multivariate data set* available in github repository accompanied with this textbook.

```
> library(lavaan)
> # define model
> model <- '
+   f1 =~ sat1 + sat2 + sat3
```

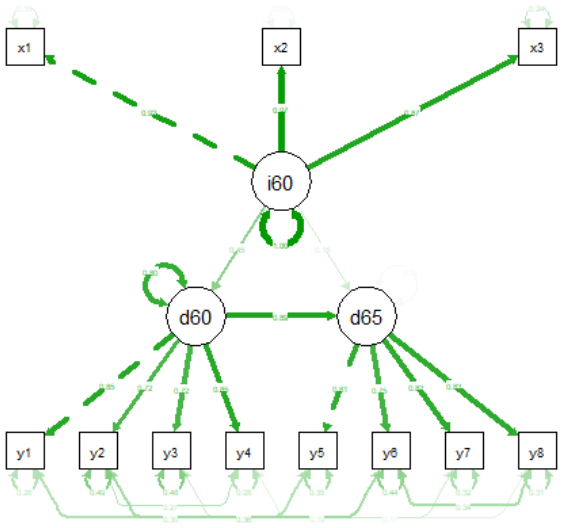


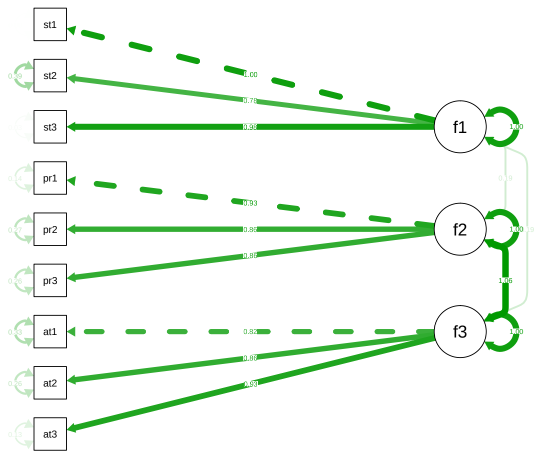
Figure 5.2: Path diagram for Political Democracy dataset.

```
+   f2 =~ per1 + per2 + per3
+   f3 =~ att1 + att2 + att3
+ '
> # fit sem
> semfit <- lavaan(model, data=multivardata, auto.var=TRUE,
+                  auto.fix.first=TRUE, auto.cov.lv.x=TRUE)
> # summary statistics
> summary(semfit)

lavaan 0.6-7 ended normally after 63 iterations
  Estimator                                         ML
  Optimization method                           NLMINB
  Number of free parameters                         21

  Number of observations                            30
Model Test User Model:
  Test statistic                               143.452
  Degrees of freedom                                24
  P-value (Chi-square)                           0.000

Parameter Estimates:
  Standard errors                             Standard
  Information                                 Expected
  Information saturated (h1) model          Structured
Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)
  f1 =~
    sat1              1.000
    sat2              0.774    0.116    6.686    0.000
    sat3              1.049    0.045   23.183    0.000
  f2 =~
    per1              1.000
    per2              0.965    0.125    7.739    0.000
    per3              0.883    0.113    7.824    0.000
  f3 =~
    att1              1.000
    att2              0.978    0.159    6.150    0.000
    att3              1.088    0.155    7.018    0.000
Covariances:
                   Estimate  Std.Err  z-value  P(>|z|)
  f1 ~~
    f2                0.368    0.369    0.997    0.319
    f3                0.337    0.344    0.980    0.327
  f2 ~~
    f3                1.583    0.457    3.464    0.001
Variances:
                   Estimate  Std.Err  z-value  P(>|z|)
   .sat1              0.005    0.061    0.080    0.937
   .sat2              0.876    0.229    3.824    0.000
   .sat3              0.080    0.070    1.138    0.255
   .per1              0.255    0.068    3.739    0.000
   .per2              0.548    0.135    4.056    0.000
   .per3              0.444    0.110    4.051    0.000
   .att1              0.681    0.167    4.071    0.000
   .att2              0.457    0.114    4.019    0.000
   .att3              0.241    0.066    3.635    0.000
    f1                2.257    0.587    3.844    0.000
    f2                1.607    0.477    3.372    0.001
    f3                1.381    0.500    2.764    0.006
```

Plotting the solution [10]

```
> library(semPlot)
> semPaths(semfit, "std", edge.label.cex = 0.5, curvePivot = TRUE, rotation=4)
```

[10] Read more about plotting at http://sachaepskamp.com/semPlot/examples.



5.9 Cluster analysis

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including pattern recognition, image analysis, information retrieval, bioinformatics, data compression, computer graphics and machine learning.

5.9.1 Types of techniques

1. Connectivity models: for example, hierarchical clustering builds models based on distance connectivity.
2. Centroid models: for example, the k-means algorithm represents each cluster by a single mean vector.

3. Distribution models: clusters are modeled using statistical distributions, such as multivariate normal distributions used by the expectation-maximization algorithm.
4. Density models: for example, DBSCAN and OPTICS defines clusters as connected dense regions in the data space.

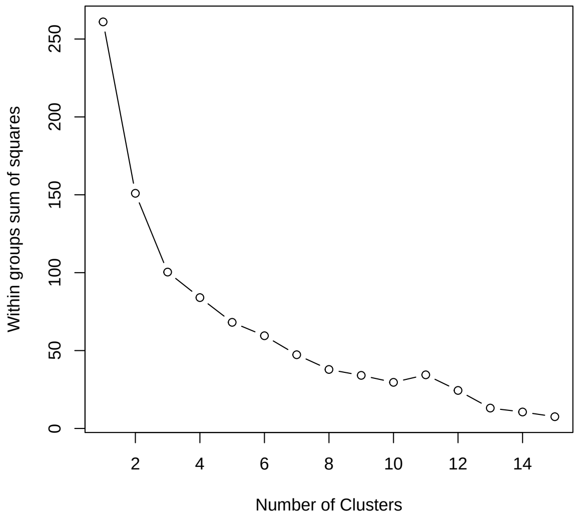
```
> sum(is.na.data.frame(multivardata))
```

```
[1] 0
```

```
> clusdata <- scale(multivardata[, 5:13])
```

5.9.2 Determine number of clusters

```
> wss <- (nrow(clusdata)-1)*sum(apply(clusdata, 2, var))
> for (i in 2:15) wss[i] <- sum(kmeans(clusdata,
+                                       centers=i)$withinss)
> plot(1:15, wss, type="b", xlab="Number of Clusters",
+      ylab="Within groups sum of squares")
```



5.9.3 K-Means cluster analysis

```
> fit <- kmeans(clusdata, 3) # 3 cluster solution
> aggregate(clusdata, by=list(fit$cluster), FUN=mean)
```

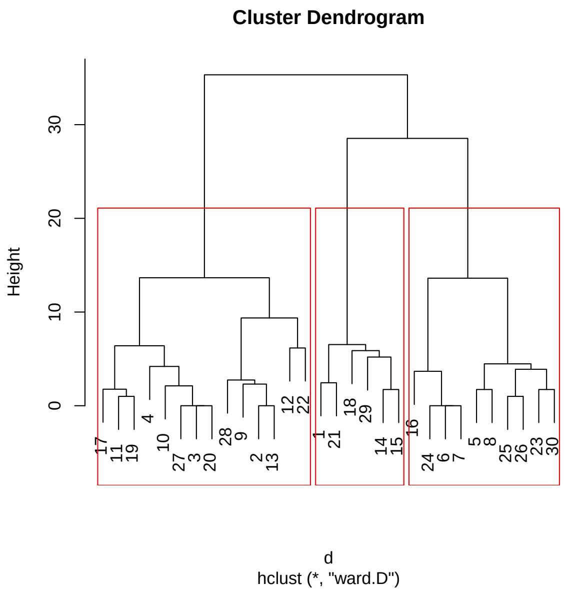
```
  Group.1       sat1       sat2       sat3       per1       per2       per3
1       1  0.3649760 -0.1153052  0.4041043  1.3629102  1.2774866  1.1251947
2       2 -0.8373436 -0.6682994 -0.8564381 -0.5283528 -0.4779322 -0.4238563
3       3  1.1003754  1.2848291  1.0946624 -0.4382927 -0.4411052 -0.3834462
        att1       att2       att3
1  1.0383907  1.1487468  1.3815784
2 -0.3064638 -0.4703807 -0.5536567
3 -0.5020790 -0.3255806 -0.4126793
```

```
> head(data.frame(clusdata, fit$cluster))
```

```
        sat1       sat2       sat3       per1       per2       per3       att1
1  1.2637975 -1.1859961  1.2481197  1.6330906  1.5352754  1.7852251  1.5518806
2 -1.3509559 -1.1859961 -1.2071977  0.1921283  0.1604019  0.2765842  0.1825742
3 -0.6972676 -0.5271094 -0.5933684 -0.5283528 -0.5270348 -0.4777363 -0.5020790
4 -0.6972676 -0.5271094 -1.2071977  0.1921283 -1.2144716  0.2765842  0.1825742
5  0.6101091  0.7906641  0.6342903 -0.5283528 -0.5270348 -0.4777363 -0.5020790
6  1.2637975  1.4495508  1.2481197  0.1921283  0.1604019  0.2765842  0.1825742
        att2       att3 fit.cluster
1  1.7937651  1.6507171           1
2  0.3194376  0.2153109           2
3 -0.4177261 -0.5023921           2
4 -1.1548898 -1.2200952           2
5 -0.4177261 -0.5023921           3
6  0.3194376  0.2153109           3
```

5.9.4 Hierarchical agglomerative clustering

```
> d <- dist(multivardata[, 5:13], method = "euclidean")
> fit <- hclust(d, method="ward")
> plot(fit)
> groups <- cutree(fit, k=3)
> rect.hclust(fit, k=3, border="red")
```



5.9.5 Multiscale bootstrap resampling (PVCLUST)

The `pvclust()` function in the pvclust package provides p-values for hierarchical clustering based on *multiscale bootstrap resampling*. Clusters that are highly supported by the data will have large p values. Be aware that *pvclust* clusters columns, not rows. Transpose your data before using.

```
> # Ward Hierarchical Clustering with Bootstrapped p values
> library(pvclust)
> fit <- pvclust(multivardata[, 5:13], method.hclust="ward",
+    method.dist="euclidean")
> plot(fit) # dendogram with p values
> # add rectangles around groups highly supported by the data
> pvrect(fit, alpha=.95)
```

5.9.6 Model Based

Model based approaches assume a variety of data models and apply maximum likelihood estimation and Bayes criteria to identify the most likely model and number of clusters. Specifically, the

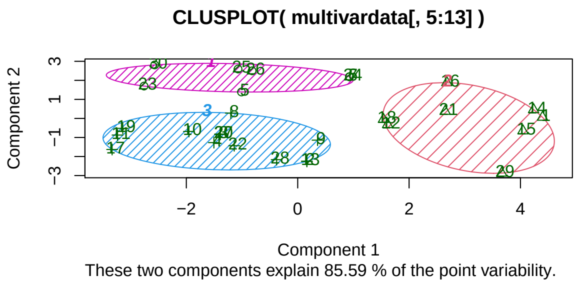
`Mclust()` function in the *mclust* package selects the optimal model according to *BIC* for *EM* initialized by hierarchical clustering for parameterized *Gaussian mixture models*. **One chooses the model and number of clusters with the largest BIC**. Use the function `help("mclustModelNames")` to know the details on best models.

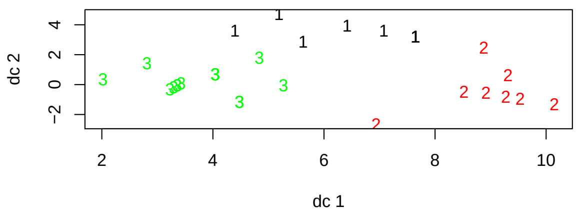
```
> # Model Based Clustering
> library(mclust)
> fit <- Mclust(multivardata[, 5:13])
> plot(fit) # plot results
> summary(fit) # display the best model
```

5.9.7 Plotting Cluster Solutions

It is always a good idea to look at the cluster results.

```
> # K-Means Clustering with 5 clusters
> fit <- kmeans(multivardata[, 5:13], 3)
> par(mfrow=c(2, 1))
> # Cluster Plot against 1st 2 principal components
> # vary parameters for most readable graph
> library(cluster)
> clusplot(multivardata[, 5:13], fit$cluster, color=TRUE, shade=TRUE,
+    labels=2, lines=0)
> # Centroid Plot against 1st 2 discriminant functions
> library(fpc)
> plotcluster(multivardata[, 5:13], fit$cluster)
```


CLUSPLOT(multivardata[, 5:13])
Component 2
Component 1
These two components explain 85.59 % of the point variability.


dc 2
dc 1

Key words