DATA ANALYTICS USING R

Contents

Preface

R is *lingua franca* of statistics. I started writing this book for one simple reason that I got to teach R for business analytics students. I never went any academy nor to a Guru for learning R. My first encounter with analytics software is R. I taught "business statistics" several times during my stint as academic. I always used Excel to teach statistics in my classes. However, over the period of time I got bored using Excel in my classes. R turned as my choice to make a changeover.

I turned into data analyst due to my being struck accidentally across open source software when I was in Ethiopia. I got to use GNU/Linux due to a very simple reason that as I got to find out a way to keep my lappy away from virus infection in the university. I crashed Windows a couple of times when I was working in office. I came to know that GNU/Linux is a wonderful solution to handle viruses. My first experience of GNU/Linux, perhaps, is *Jaunty Jackalope*, I suppose. This must be Ubuntu 9.04, an LTS version, if I am not wrong. However, my involvement in GNU/Linux became a serious affair though *Karmic Koala.* I still remember few of my colleagues used to visit my home for OS installation in those days.

I came across R in Ubuntu Software Applications. I notice this little yet uppercase R in *package manager* while I was searching for statistical software tools in Ubuntu. R changed not only my understanding of Statistics but the very way of learning the same. I addicted to R so much so that for every pretty little calculations I used to refer R manuals. I must be the first academic to introduce R in south Indian business management curriculum, yet remained unknown. I used to write reply mails, in my early years of R practice, denouncing commercial tools whenever I used to get invitations related to the usage of same in academics. At first I was never understood here in academic fraternity until certain time. After a couple years I started seeing the response in neighborhood universities and institutes. Today, R is one of the best tools for statistical learning and practice in India.

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 data analysis. There is only very short description on programming concepts such as control flow, loops, functions etc. However, there is sufficient information on practice of analytics. While coming to contents; this text has 5 chapters and each chapter represents a unique concept of data analytics. 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 50 practice based exercises and five case investigations to help academics do their work better.

This book has a companion repository located at `https://github.com/Kamakshaiah/R-Book-Material`. Though, there is not much script associated with this text, but few data sets which are used in this book are available for download. Obtain those data sets and practice while reading code chunks. Feel free to reach me using kamakshaiah.m@gmail.com for doubts and any other queries.

Happy reading · · ·

Dr. M. Kamakshaiah
Author

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.

Chapter 1

Introduction to R

In real open source, you have the right to control your own destiny.

- Linus Torvalds

1.1 Open Source Software

Open-source software (OSS) is a type of computer software in which source code is released under a license by which the copyright holder grants users the rights to use, study, change, and distribute the software to anyone and for any purpose. [1] [2] Open-source software may be developed in a collaborative and public manner. Open-source software is a prominent example of open collaboration. [3]

In the early days of computing, programmers and developers shared software in order to learn from each other and evolve the field of computing. Eventually, the open-source notion moved to the way side of commercialization of software in the years 1970-1980. However, academics still often developed software collaboratively. For example, Donald Knuth in 1979 with the TeX typesetting system or Richard Stallman in 1983 with the GNU operating system. [4] In 1997, Eric Raymond published The Cathedral and the Bazaar, a reflective analysis of the hacker community and free-software principles. The paper received significant attention in early 1998, and was one factor in motivating Netscape Communications Corporation to release their popular Netscape Communicator Internet suite as free software. This source code subsequently became the basis behind SeaMonkey, Mozilla Firefox, Thunderbird and KompoZer.

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. A Microsoft executive publicly stated in 2001 that "open source is an intellectual property destroyer. I can't imagine something that could be worse than this for the software business and the intellectual-property business." [5] However, while 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, Google, and State Farm 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 FOSS.

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. [6] 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.

The Open Source Definition presents an open-source philosophy and further defines the terms of use, modification and redistribution of open-source software. Software licenses grant rights to users which would otherwise be reserved by copyright law to the copyright holder. Several open-source software licenses have qualified within the boundaries of the Open Source Definition. The most prominent and popular example is the GNU General Public License (GPL), which "allows free distribution under the condition that further developments and applications are put under the same licence", thus also free.

1.1.1 UNIX

Today most of the developments in open source computing world can be attributable to two individuals namely (1) Linus Torvalds and (2) Richard M. Stallman. If these two had not thought about software differently, we could have lost lot of developments in this world.

I should mention UNIX before I do that about Linux. UNIX is a family of multitasking, multiuser computer operating systems that derive from the original AT&T Unix, development starting in the 1970s at the Bell Labs research center by Ken Thompson, Dennis Ritchie, and others. Perhaps, Unix is the first OS characterized by a modular design that is sometimes called the "Unix philosophy". At first Unix was named as *Unics*, Uniplexed Information and Computing Service as pun on *Multics.* Multics was an operating system in 1960s and it was a serious project for a group of organizations such as MIT, Bell Labs, GE. Ken Thompson, Dennis Ritchie in Bell had started working on Unix when Multics was wound up officially everywhere. The leisure work done by these two individuals turned as great fortune to the world. The operating system was originally written in assembly language, but in 1973, Version 4 Unix was rewritten in C. Bell Labs produced several versions of Unix that are collectively referred to as "Research Unix". In 1975, the first source license for UNIX was sold to Donald B. Gillies at the University of Illinois Department of Computer Science. During the late 1970s and early 1980s, the influence of Unix in academic circles led to large-scale adoption of Unix (BSD and System V) by commercial startups, including Sequent, HP-UX, Solaris, AIX, and Xenix. In the 1990s, Unix and Unix-like systems grew in popularity as BSD and Linux distributions were developed through collaboration by a worldwide network of programmers. In 2000, Apple released Darwin, also a Unix system, which became the core of the Mac OS X operating system, which was later renamed macOS.

1.1.2 Linux

Linux is a family of open source Unix like operating systems based on the Linux kernel, an operating system kernel first developed by Linus Torvalds. Linux is typically packaged in a Linux distribution.

Distributions include the Linux kernel and supporting system soft-

Figure 1.1: Ken Thompson (sitting) and Dennis Ritchie working together at a PDP-11

ware and libraries, many of which are provided by the GNU Project. Many Linux distributions use the word “Linux” in their name, but the Free Software Foundation uses the name GNU/Linux to emphasize the importance of GNU software. Popular Linux distributions include Debian, Fedora, and Ubuntu. Commercial distributions include Red Hat Enterprise Linux and SUSE Linux Enterprise Server. Desktop Linux distributions include a windowing system such as X11 or Wayland, and a desktop environment such as GNOME or KDE.

Linux was originally developed for personal computers based on the Intel x86 architecture, but has since been ported to more platforms than any other operating system. Linux is the leading operating system on servers and other big iron systems such as mainframe computers, and the only OS used on TOP500 supercomputer.

Linux also runs on embedded systems, i.e. devices whose operating system is typically built into the firmware and is highly tailored to the system. This includes routers, automation controls, televisions, digital video recorders, video game consoles, and smart watches. Many smartphones and tablet computers run Android and other Linux derivatives. Because of the dominance of Android on smartphones, Linux has the largest installed base of all general-purpose operating systems.

In 1991, while attending the University of Helsinki, Torvalds became curious about operating systems. Frustrated by the licensing of MINIX, which at the time limited it to educational use only, he began to work on his own operating system kernel, which eventually became the Linux kernel.

Torvalds began the development of the Linux kernel on MINIX and applications written for MINIX were also used on Linux. Later, Linux matured and further Linux kernel development took place on Linux systems. GNU applications also replaced all MINIX components, because it was advantageous to use the freely available code from the GNU Project with the fledgling operating system; code licensed under the GNU GPL can be reused in other computer programs as long as they also are released under the same or a compatible license. Torvalds initiated a switch from his original license, which prohibited commercial redistribution, to the GNU GPL. Developers worked to integrate GNU components with the Linux kernel, making a fully functional and free operating system.

Adoption of Linux in production environments, rather than being used only by hobbyists, started to take off first in the mid-1990s in the super computing community, where organizations such as NASA started to replace their increasingly expensive machines with clusters of inexpensive commodity computers running Linux. Commercial use began when Dell and IBM, followed by Hewlett-Packard, started offering Linux support to escape Microsoft's monopoly in the desktop operating system market.

Today, Linux systems are used throughout computing, from embedded systems to virtually all supercomputers, and have secured a place in server installations such as the popular LAMP application stack. Use of Linux distributions in home and enterprise desktops has been growing. Linux distributions have also become popular in the netbook market, with many devices shipping with customized Linux distributions installed, and Google releasing their own Chrome OS designed for netbooks.

Linux's greatest success in the consumer market is perhaps the mobile device market, with Android being one of the most dominant operating systems on smartphones and very popular on tablets and,

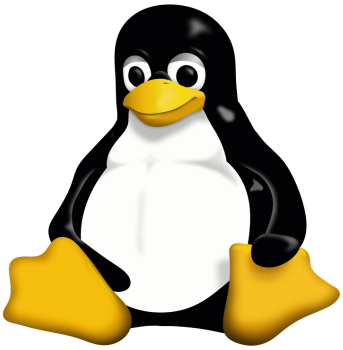


Figure 1.2: Linux Torvalds and Linux Mascot

more recently, on wearables. Linux gaming is also on the rise with Valve showing its support for Linux and rolling out its own gaming oriented Linux distribution.

1.2 Installation & Editors

Anything starts from installation at least in software. Installation provides environment and access to tool level resources. Installing R is pretty easy task unlike many other software suites. There are binaries for three different platforms namely, (1) Windows, (2) Linux and (3) MacOS. For others, they can obtain sources and compile R for their respective platforms.

1.2.1 About R

R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing. The R language is widely used among statisticians and data miners for developing statistical software and data analysis. Polls, data mining surveys, and studies of scholarly literature databases show substantial increases in popularity; as of September 2019, R ranks 19th in the TIOBE index, a measure of popularity of programming languages. [7]

R is a GNU software, source code for the R software environment

Figure 1.3: John Chambers

is written primarily in C, FORTRAN, and R itself [8] and is freely available under the GNU General Public License. Pre-compiled binary versions are provided for various operating systems. Although R has a command line interface, there are several graphical user interfaces, such as RStudio, an integrated development environment.

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. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team (of which Chambers is 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 in 2000.

R and its libraries implement a wide variety of statistical and graphical techniques, including linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, and others. R is easily extensible through functions and extensions, and the R community is noted for its active contributions in terms of packages. Many of R's standard functions are written in R itself, which makes it easy for users to follow the algorithmic choices made. For computationally intensive tasks, C, C++, and FORTRAN code can be linked and called at run time. Advanced users can write C, C++, Java, .NET or Python code to manipulate R objects directly. R is highly extensible through the use of user-submitted packages for specific functions or specific areas of study. Due to its S heritage, R has stronger object-oriented programming facilities than most statistical computing languages. Extending R is also eased by its lexical scoping rules. Another strength of R is static graphics, which can produce publication-quality graphs, including mathematical symbols. Dynamic and interactive graphics are available through additional packages.

R Packages

The capabilities of R are extended through user-created packages, which allow specialized statistical techniques, graphical devices, import/export capabilities, reporting tools (Rmarkdown, knitr, Sweave), etc. These packages are developed primarily in R, and sometimes in Java, C, C++, and FORTRAN. Many of R packages are available in total together with data sets and vignettes.

A core set of packages is included with the installation of R, with more than 15,000 additional packages (as of September 2018) available at the Comprehensive R Archive Network (CRAN), Bioconductor, Omegahat, GitHub, and other repositories.

Users and developers use these packages as per requirement. There is also a mechanism to install and use packages as by domain. The mechanism is CRAN Task Views also known as `ctv`. CRAN stands for "Comprehensive R Archive Network". CRAN is the official repository that hosts lot of data related to R. "Task Views" page on the CRAN website lists a wide range of tasks (in fields such as Finance, Genetics, High Performance Computing, Machine Learning, Medical Imaging, Social Sciences and Spatial Statistics) to which R has been applied and for which packages are available. R has also been identified by the FDA as suitable for interpreting data from clinical research.

Other places like Crantastic and R-Forge are community based platforms for the collaborative development of R packages, R-related software, and projects. The Bioconductor project provides R packages for the analysis of genomic data. This includes object-oriented data-handling and analysis tools for data from Affymetrix, cDNA microarray, and next-generation high-throughput sequencing methods. There is another novel place for neuroscientists, it is known as https://www.neuroconductor.org/. This place is greatly helpful to both beginners and advanced programmers to learn and practice neuroscience.

Editors

The most specialized integrated development environment (IDE) for R is RStudio. Visit https://rstudio.com/. RStudio today is not just editor but grown as an ecosystem for R based programming and web development. A similar development interface is R Tools for Visual Studio. Some generic IDEs like Eclipse, also offer features to work with R. Graphical user interfaces with more of a point-and-click approach include Rattle GUI, R Commander, and RKWard.

Some of the more common editors with varying levels of support for R include Emacs (Emacs Speaks Statistics), Vim (Nvim-R plugin), Neovim (Nvim-R plugin), Kate, LyX, Notepad++, Visual Studio Code, WinEdt, and Tinn-R.

R functionality is accessible from several scripting languages such as Python, Perl, Ruby, *F#*, and Julia. Interfaces to other, high-level programming languages, like Java and .NET *C#* are available as well.

RStudio

RStudio is an integrated development environment (IDE) for R, a programming language for statistical computing and graphics. The RStudio IDE is developed by RStudio, Inc., a commercial enterprise founded by JJ Allaire, creator of the programming language ColdFusion. RStudio, Inc. has no formal connection to the R Foundation, a not for profit organization located in Vienna Austria, which is responsible for overseeing development of the R environment for statistical computing. RStudio is available in two formats: RStudio Desktop, where the program is run locally as a regular desktop application; and RStudio Server, which allows accessing RStudio using a web browser while it is running on a remote Linux server.

Work on RStudio started around December 2010, and the first public beta version (v0.92) was officially announced in February 2011.

Version 1.0 was released on 1 November 2016. Version 1.1 was released on 9 October 2017. RStudio is available with the GNU Affero General Public License version 3. The AGPL v3 is an open source license that guarantees the freedom to share the code. [9] RStudio Desktop and RStudio Server are both available in free and commercial editions. OS support depends on the format/edition of the IDE. Prepackaged distributions of RStudio Desktop are available for Windows, macOS, and Linux. RStudio Server and Server Pro run on Debian, Ubuntu, Red Hat Linux, CentOS, openSUSE and SLES.

RStudio is partly written in the C++ programming language and uses the Qt framework for its graphical user interface. The bigger percentage of the code is written in Java. JavaScript is also amongst the languages used. In April 2019, RStudio released the RStudio Job Launcher, an adjunct to RStudio Server. The launcher provides the ability to start processes within various batch processing systems (e.g. Slurm) and container orchestration platforms (e.g. Kubernetes). This function is only available in RStudio Server Pro (fee-based application).

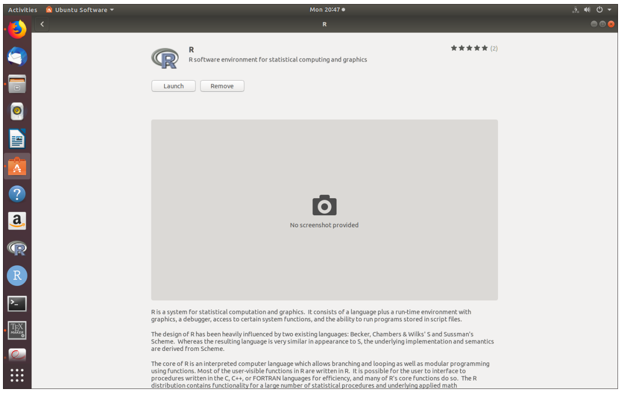
RStudio and its team have contributed to many R packages. These include:

1. Tidyverse - R packages for data science, including `ggplot2, dplyr, tidyr`, and `purrr`.
2. Shiny - An interactive web technology
3. RMarkdown - Insert R code into markdown documents
4. knitr - Dynamic reports combining R, TeX, Markdown & HTML
5. packrat - Package dependency tool
6. devtools - Package development tool

1.2.2 R Installation

Installing R is straight in both Windows and Linux. I use Ubuntu and if you are in Ubuntu, you just need to open *package manager* or *software center* to install the software. [10] You will find a window in Ubuntu like the one in 1.4.

You can press the button “install”, the rest is taken care by Ubuntu



R is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.

The design of R has been heavily influenced by two existing languages: Becker, Chambers & Wilks' S and Sussman's Scheme. Whereas the resulting language is very similar in appearance to S, the underlying implementation and semantics are derived from Scheme.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. Most of the user-visible functions in R are written in R. It is possible for the user to interface to procedures written in the C, C++, or FORTRAN languages for efficiency, and many of R's core functions do so. The R distribution contains functionality for a large number of statistical procedures and underlying applied math

Figure 1.4: Ubuntu Software Center

OS. There is other alternative way to install R i.e. from Terminal. Press `Ctl+Alt+T` to open Ubuntu or Linux Terminal. [11] Then issue the following command.

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

This will take care of entire installation. I mean the base package, dependencies *et cetera*. Since Ubuntu uses Debian package manager, the activity is very simple. Ubuntu packaging system is very great and gigantic. You find everything for all of your needs. In Ubuntu there is also another way to install R but by using *Debian package manager* also known as `dpkg`. This method is highly useful for `.deb` packages. Sometimes, we also try to install *.deb* packages, just like `.exe` in Windows. `.deb` packages are ready-made installers available for Debian platform. Suppose you have `.deb` package such as the one provided at https://packages.debian.org/sid/all/r-base/download, you need to use `dpkg`.

```
dpkg -i r-base_3.6.1-6_all.deb
```

However, this is not advisable. It is always better to use "Software Center" to install packages, in stead of using `.deb` files. One potential reason is Software Center knows how to fetch and install all recommended dependencies automatically.

I use a couple of other Linux based OSes such as Fedora, SUSE, CentOS. Fedora, CentoS and SUSE belongs to RPM platform. RPM stands for *Redhat Package Management*. RPM is very old yet robust package management systems available for most of the OSes in Linux world. Suppose if you would like to install R in Fedora and CentOS; follow the following command in Terminal. By the way, the short cut keys to open Terminal in Fedora is `Ctl+Alt+F2`.

```
sudo yum install R r-core
```

Fedora too has a package manager known as `dnf`. DNF is a software package manager that installs, updates, and removes packages on RPM-based Linux distributions. It automatically computes dependencies and determines the actions required to install packages. DNF also makes it easier to maintain groups of machines, eliminating the need to manually update each one using rpm. Introduced in Fedora 18, it has been the default package manager since Fedora 22. DNF or *Dandified yum* is the next generation version of yum in Fedora. [12] Installing using `dnf` is also straight forward. Use the following statement in Fedora Terminal. [13]

```
dnf install R r-core
```

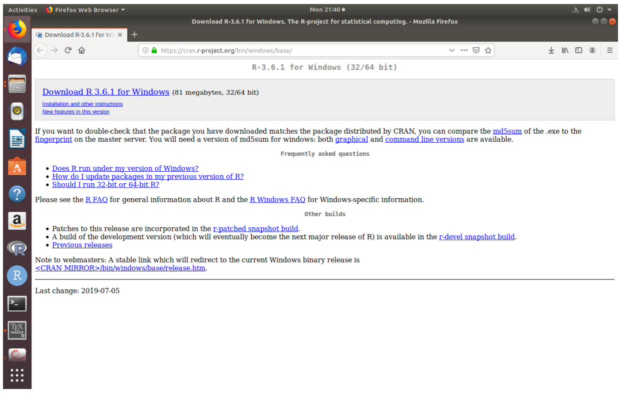
While coming to SUSE; SUSE is known as most viable commercial Linux OS. There is open source OS known as openSUSE. [14] In SUSE installing software is bit easy and mimics like Windows. There is a particular mechanism known as "one click installer". R is also available as one-click installer. Visit https://cran.r-project.org/bin/linux/suse/README.html for these installers. You just have to know openSUSE version. Otherwise, suppose if you want to install using Terminal, use the following command in the Terminal.

```
zypper install R-base R-base-devel
```

Zypper is a command line package manager for installing, updating and removing packages as well as for managing repositories. It is especially useful for accomplishing remote software management tasks or managing software from shell scripts.

In Windows

Now coming to Windows; installing R is more hassle free and straight forward approach. You just have to get `.exe` file. Where do you get it? Just simple! It is available from CRAN. There is more information about CRAN in forthcoming section. Just go to https://cran.r-project.org/bin/windows/base/ or if you


Activities
Firefox Web Browser
Mon 21:40
Download R-3.6.1 for Windows. The R-project for statistical computing. - Mozilla Firefox
Download R-3.6.1 for Wi
https://cran.r-project.org/bin/windows/base/

R-3.6.1 for Windows (32/64 bit)

Download R 3.6.1 for Windows (81 megabytes, 32/64 bit)

Installation and other instructions

New features in this version

If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the md5sum of the .exe to the fingerprint on the master server. You will need a version of md5sum for windows: both graphical and command line versions are available.

Frequently asked questions

- Does R run under my version of Windows?
- How do I update packages in my previous version of R?
- Should I run 32-bit or 64-bit R?

Please see the R FAQ for general information about R and the R Windows FAQ for Windows-specific information.

Other builds

- Patches to this release are incorporated in the r-patched snapshot build.
- A build of the development version (which will eventually become the next major release of R) is available in the r-devel snapshot build.
- Previous releases

Note to webmasters: A stable link which will redirect to the current Windows binary release is <CRAN MIRROR>/bin/windows/base/release.htm.

Last change: 2019-07-05

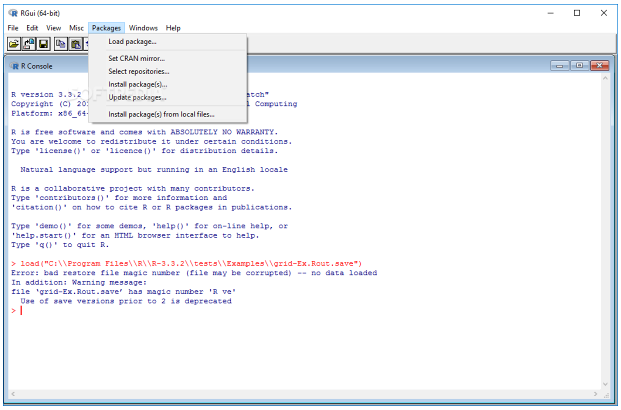
Figure 1.5: R for Windows.

can't remember this URL just search for "R download", in default browser of your OS, using google search engine. This will get you the aforementioned URL. Download the `.exe` file and install the same just like any other software.

1.2.3 Open R

Go to Windows Start, search for "R" and launch the application. In windows, R has R-GUI. This is really a big blessing for Windows platform. Windows uses MinGW environment for opening or closing R. MinGW is a Linux environment available for Windows OS, a contraction of "Minimalist GNU for Windows", and a minimalist development environment for native Microsoft Windows applications. [15] The other such environment is *Cygwin*. Cygwin is a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows.[16]

RGUI, refers Figure 1.6, is just a minimal editor for programming. RGUI, as a minimalistic editor, has few menus along with a provision for R Console, which is main part of the window and is located at the left bottom corner of the main window. We just have to start typing commands or statements at *R command prompt* which is just a little ">" sign in the R Console.



```
R version 3.3.2                                        atch"
Copyright (C) 20                                       l Computing
Platform: x86_64-

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  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.

> load("C:\\Program Files\\R\\R-3.3.2\\tests\\Examples\\grid-Ex.Rout.save")
Error: bad restore file magic number (file may be corrupted) -- no data loaded
In addition: Warning message:
file 'grid-Ex.Rout.save' has magic number 'R ve'
  Use of save versions prior to 2 is deprecated
> |
```

Figure 1.6: R GUI for Windows

1.2.4 Scripts & Commands

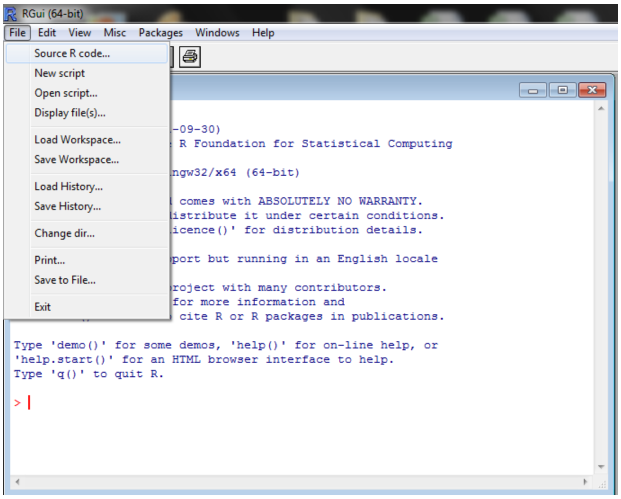
A computer script is a list of commands that are executed by a certain program or scripting language. The commands within the script are executed when sufficient resources, through its computing environment, were available through its language. A language, or script language, is a programming language for a special run-time environment that automates the execution of tasks; the tasks could alternatively be executed one-by-one by a human operator. Scripting languages are often interpreted, rather than compiled. [17]

Scripts in Windows

Writing programs or commands in Linux or Windows is same. The only difference is Editor. Go to "File -> New script". Refer to Figure 1.7.

You will see a new sub-window getting opened in the same main-window, just as shown in the Figure 1.8.

This is a very good practice to write programs in R. The advantage is we can save these scripts for later use. R uses `.R` file format to save, open and use script files.



```
-09-30)
 R Foundation for Statistical Computing
ngw32/x64 (64-bit)
 comes with ABSOLUTELY NO WARRANTY.
istribute it under certain conditions.
icence()' for distribution details.
port but running in an English locale
roject with many contributors.
for more information and
 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.

> |
```

Figure 1.7: File Menu in RGUI in Windows OS

Scripts in Windows CMD

Whatever may be the programming language and the grimmer associated with it, end of the day every program is compiled or interpreted using Command Line Interpreter (CLI). Take for example, C, Java, Python, Perl and many more. All these languages support plain word file to write programs. We don't need any special editor to execute (compile/interpret) programs. For instance, if you are writing a Java program or Python program, it is just *notepad* that comes handy for work. There is another text editor known as *notepad++*. However, one special care that every programmer needs is to save the file in language specific format. Suppose if we are writing programs in Java, the file format is `.java`. Same fashion, if we are writing programs using Python, the file format will be `.py`. Do you notice websites, few websites ends with *.html*, this is none other than a file format. The server tries retrieve a file that ends with extension `.html`.

CMD stands for COMMAND in Windows. This is also known as Windows Command Prompt. You need to open CMD using Windows START button. You will find this the left bottom side as . Write CMD in search. You may find CMD as shown in the Figure 1.9.

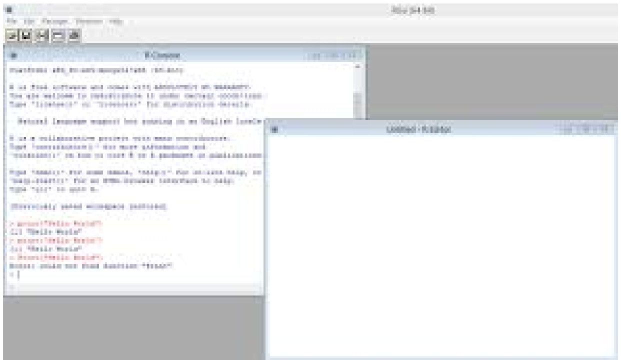


Figure 1.8: New Script in RGUI in Windows OS

You can start writing a file using *notepad* text editor. Save the file in your computer at some valid location. Open CMD. Execute the file using the following command.

```
Rscript.exe r_program.R
```

`Rscript.exe` is the R interpreter which executes R programs. In the above example `r_program.R` is the script name. Notice, the above code works for only R "script".

Windows native executable format is `.bat`. This format supports a language convention known as *batch*. Batch can help you write OS specific programs using this extension and execute them in CMD. `.bat` is treated on par with `.exe` for execution in Windows. Suppose you want to mix Windows specific commands with R and intent to execute both instructions together, then R offers a unique way, i.e. using `R.exe`. Suppose that you have a file `r_program.bat` in which you have both *batch* and R instructions. You can do as shown below to execute such files.

```
r_program.bat
```

The press enter button in your keyboard. That's all. Because, Windows knows that it is batch file and doesn't need any predecessor nor follower to execute the file.

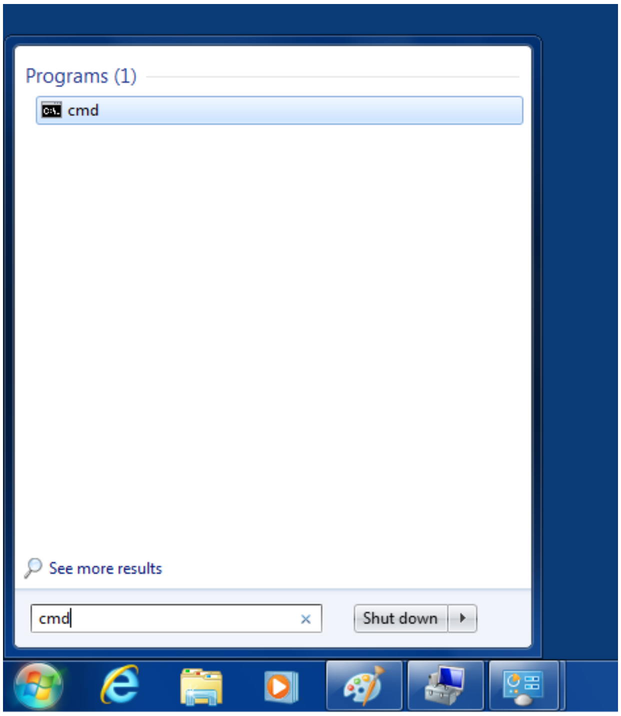
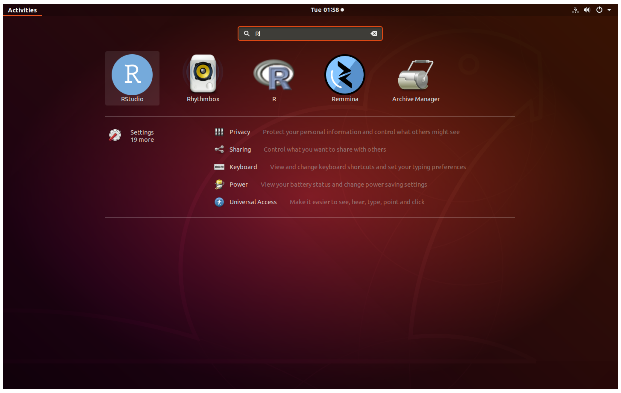


Figure 1.9: CMD Prompt in Windows Start



Privacy Protect your personal information and control what others might see
Sharing Control what you want to share with others
Keyboard View and change keyboard shortcuts and set your typing preferences
Power View your battery status and change power saving settings
Universal Access Make it easier to see, hear, type, point and click

Figure 1.10: Dash in Ubuntu

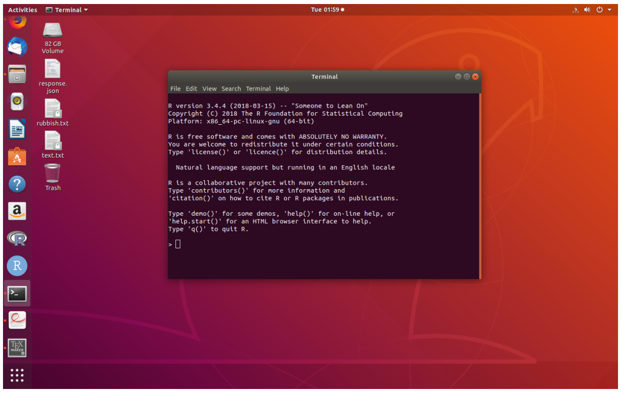
Scripts in Linux Terminal

Linux terminal perhaps is more strong and powerful compared to Windows CMD. This is not a biased statement. In Unix culture every thing is done using Terminal. I remember writing CD/DVDs using Linux Terminal in Ubuntu in my earlier days. I used to do this only out of curiosity. Linux people are mostly character users. They use very little mouse but lots of keyboard. This may be due to the reason that they belong to hacking community and hackers are power users who need to code in tons.

In Linux R opens in Terminal as a default mechanism but not in RGUI. In Linux culture when any language is opened in Terminal it is referred to REPL. REPL stands for Read-Eval-Print-Loop. Suppose if you go to DASH, refer Figure 1.10, R open in Terminal, refer to Figure 1.11.

You can start writing statements in the Console. However, R Console doesn't support scripts. We need a word file just like in Windows. There are number of option for writing scripts in Linux, namely, *nano, vim, gedit, touch* and many more. I always prefer *gedit* for it has little GUI like features. Also it looks like *notepad* of Windows. That way people find it easy to grasp. There are three ways to execute just as it is explained under Windows section.

Imagine that your program is available in a file named *r_program.R*. Use the following command Suppose you want the output thrown to Terminal.



```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  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.

> 
```

Figure 1.11: R Console in Ubuntu Terminal

```
Rscript r_program.R
```

You want the output collected in a file. The you can do as below:

```
R CMD BATCH r_program.R
```

This will create an output file with a name "r_program.Rout". You need to open using any other application just as

```
cat r_program.Rout
```

The other alternative is to execute the file using R command. Suppose you have a script with a name *r_program.R* and you can execute it with following command.

```
R < r_program.R --no-save
```

You can also use `-save` at the end of the above statement.

The fourth alternative is Linux style. This is called the very famous *dot slash (./)*. Linux has a programming language called BASH. BASH is terminal level programming language which allows users to define functions known as User Defined Functions (UDF). These functions are executed using `./`. This not only for BASH, Linux allows executing any programs by their executable paths. You might be knowing about this, if you had worked with Java or Python. Anyway, you need to add an additional line for path for executable file. Linux creates a path variables and adds to the environment when any application is installed. The common path for these executables is `/user/bin/env`. So there must be `Rscript` available from there for R too. Suppose you have R programs in a file named `r_program.R`. You need to add the following line atop of the program, which means the starting line in the script.

```
#!/usr/bin/env Rscript
```

Now it is possible to execute this file using *dot slash* convention.

```
./r_program
```

Suppose you have certain R function in side this script file such `say_hell()`. Now you can execute the function directly without using file name.

```
./say_hello()
```

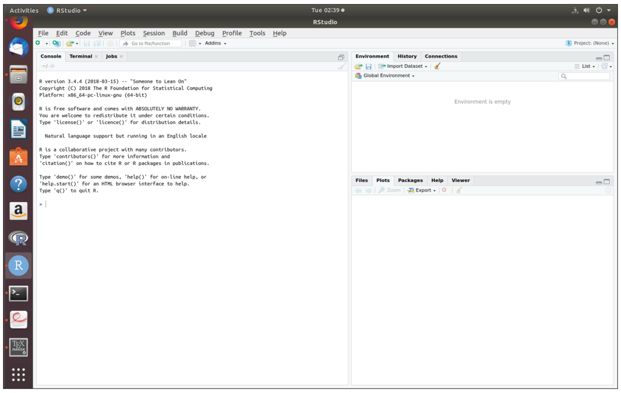
Whatever may be the output that will be thrown to Terminal.

1.2.5 RStudio

We had rather very little introduction to RStudio in previous section. RStudio is not an addin nor a GUI as most of the people confuse. It is a full fledged Editor also known as Integrated Development Environment (IDE). What is the Editor/IDE? An Editor/IDE for any programming language is *a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools, and a debugger.*

RStudio is an integrated development environment (IDE) for R, a programming language for statistical computing and graphics. RStudio today is not just an IDE for R but much more. The `shiny` package developed by RStudio turned R into a web development tool rather than a statistics tool. It is possible to develop web applications using R, just as in Java and Python, but you need Shiny.

Now in this section we will look into installation and usage of the RStudio. Go to https://rstudio.com/products/rstudio/download/. Obtain executable file for your platform. RStudio is available for few OSes such as Ubuntu, Debian, Fedora, MacOS, OpenSUSE, and Windows. You need to get the file suitable for



```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  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.

> 
```

Figure 1.12: RStudio in Ubuntu

your platform. If you are in Windows, just download the `.exe` file to certain valid place (path). Just install as usually as any other software. If you are in Linux just like Ubuntu, you need to download `.deb` file from aforementioned URL. Open with software center and click install. Ubuntu does rest of the work for you. You can't find RStudio available from Software Center in Ubuntu. You need to install only from downloaded (.deb) file. Refer to Section 1.2.2.

Once after installing RStudio, it just doesn't matter if you are in Windows or Linux or MacOS. Any OS can be as best as the other. Because, RStudio has everything that we need to do with R. Figure 1.12 is sample screenshot taken in Ubuntu Linux.

Go through all the menus and try to understand each of sub-menu. For instance, you can use File menu to create new scripts, open saved scripts. Edit menu helps in Undo, Redo, Cut, Paste and etc. There is Tools menu which is useful to install packages.

Panes in RStudio

The top-left pane: This pane is useful to write scripts and at times executing and compiling different types of files. This windows or pane is useful to create reports also. I use *R Sweave* which is highly efficient addin in RStudo to create LaTeX related documents. Many times I use this package for reports. At times I do created books in stead of reports and articles. This package needs

TeX system in your OS.

The top-right pane: This is pane that shows Environment, History and Connections. We will know these things rather more detail in forthcoming sections.

The bottom-left pane: This is rather more important and crux of RStudio. Here you have three other things (1) Console, (2) Terminal and (3) Jobs. Console is R Console. This is none other than R. You can do whatever that can be done in R REPL. The Terminal is the Linux Terminal. Jobs is not very much related to any of our jobs right now. So I leave this for now.

The bottom-right pane: This pane has few very important ingredients such as Files, Plots, Packages, Help and Viewer. We use Plots and Packages quite often from here. Now that we know basics, it is high time to jump into real R.

1.3 About CRAN & Help

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. The Comprehensive R Archive Network (CRAN) is the main repository for R packages. (If your package concerns computational biology or bioinformatics, you might be interested in Bioconductor, instead.) The main advantage to getting your package on CRAN is that it will be easier for users to install (with `install.packages`). Your package will also be tested daily on multiple systems. There are roughly more than 6000 packages for almost all needs of R users. CRAN is used by all types of individuals such as users, developers. Users use CRAN to download R and those 6000 and odd associated packages, install and use. Developers find CRAN as highly useful for submitting their packages so as to make them available for rest of the community. [18]

CRAN portal has so many sections, at left side on page, to explore. The very first section is “Mirrors”. This section has quite a few ftp mirrors for different locations. People can choose mirrors as by their location. The second section is “What’s new?”. This section has all latest information related to R. New features and bug fixes of the latest release version of R are documented in the file NEWS (also contained in the R sources). R Core Team makes announcement as and when required to be necessary. These announcements appears under this section. All the announcements are sorted in descending order by year. The third section is "CRAN Task Views" also know as CTV in short. CRAN task views aim to provide some guidance which packages on CRAN are relevant for tasks related to a certain topic. They give a brief overview of the included packages and can be automatically installed using the *ctv package*. As on October, 2019 there were 41 task views for various needs of analysts. This package need to be installed using `install.packages("ctv")`.

```
> library(ctv)
> length(available.views())
```

CRAN Task Views (CTV)

The next section is *Search*. R user community (without which R would not be what it is today) there are other possibilities to search in R web pages and mail archives. There are number of options to search R information. And most of the data or information is unorganized. There is tons of information available for R. However, there are few ways for organized search. They are (1) R Site Search: This search will allow to search the contents of the R functions, package vignettes, and task views from online resources. Go to http://finzi.psych.upenn.edu/search.html, (2) R Seek: This is just a wraper to Google search. Visit https://rseek.org/ to know more. The third one in the list is the *Nabble R Forum*. It is an innovative search engine for R messages. Visit https://r.789695.n4.nabble.com/ for more information. R has lot of internal mechanisms to search for information that we will discuss in next section. The other important place is perhaps *R Journal*. The R Journal is the open access, refereed journal of the R project for statistical computing. It features short to medium length articles covering topics that should be of interest to users or developers of R. Visit https://journal.r-project.org/ for more information.

The other important sections under CARN are *Packages* and *Manuals*. There are abundant of packages available for R user community. You need to visit CTV for more information. Alternatively visit https://cran.r-project.org/.

```
> url = "https://cloud.r-project.org/"
> df.pkgs <- as.data.frame(available.packages(repos = url))
> names(df.pkgs)
```

```
 [1] "Package"               "Version"               "Priority"
 [4] "Depends"               "Imports"               "LinkingTo"
 [7] "Suggests"              "Enhances"              "License"
[10] "License_is_FOSS"       "License_restricts_use" "OS_type"
[13] "Archs"                 "MD5sum"                "NeedsCompilation"
[16] "File"                  "Repository"
```

```
> dim(df.pkgs)
```

```
[1] 18669    17
```

```
> length(df.pkgs[, 1])
```

```
[1] 18669
```

Above code requires a package named `curl` to be installed properly in both Ubuntu and R. If `curl` is installed properly, the execution creates an object `df.pkgs` in your *workspace*. You may know the number of package by simply executing another statement `dim(df.pkgs)`. This statement brings two numbers first one represents number of rows and second one represents number of columns. There are 15774 packages by the time of writing this book (some day, in June, 2020). [19]

The last section that I would like to highlight is *Manuals*. Manuals are highly useful resources for R users. Users at first may not be knowing anything about usage. Hence, these manuals comes very handy. Manuals for R were created on Debian Linux and may differ from the manuals for Mac or Windows on platform-specific pages, but most parts will be identical for all platforms. The correct version of the manuals for each platform are part of the respective R installations.

There are quite a few things which I am not able to provide through this text. Please visit CRAN repo of R and explore more personally.

1.3.1 Help in R

As I mentioned there are certain functions that helps to know about R. I have to move a bit further or extend this topic beyond *help*. R has a system and every user need to know the very system of R. For instance, when we open R, the R creates an environment for us. This is known as Global Environment. The very first function to execute after installation perhaps is `license()`. This statement gets the following information.

```
This software is distributed under the terms of the GNU General
Public License, either Version 2, June 1991 or Version 3, June 2007.
The terms of version 2 of the license are in a file called COPYING
which you should have received with
this software and which can be displayed by RShowDoc("COPYING").
Version 3 of the license can be displayed by RShowDoc("GPL-3").
```

```
Copies of both versions 2 and 3 of the license can be found
at https://www.R-project.org/Licenses/.
A small number of files (the API header files listed in
R_DOC_DIR/COPYRIGHTS) are distributed under the
LESSER GNU GENERAL PUBLIC LICENSE, version 2.1 or later.
This can be displayed by RShowDoc("LGPL-2.1"),
or obtained at the URI given.
Version 3 of the license can be displayed by RShowDoc("LGPL-3").
'Share and Enjoy.'
```

Frankly we don't need to execute this command because R shows this information in the Console when we open R. However, there is another statement, mind that it is not a function, which gives further more information is `version`. Following is the sample information.

```
> version

               _
platform       x86_64-w64-mingw32
arch           x86_64
os             mingw32
crt            ucrt
system         x86_64, mingw32
status
major          4
minor          2.1
year           2022
month          06
day            23
svn rev        82513
language       R
version.string R version 4.2.1 (2022-06-23 ucrt)
nickname       Funny-Looking Kid
```

This statement is rather more useful to anybody for making sense. I can retrieve specific information as by the parameter. For instance,

```
> ver <- version
> ver\$platform

[1] "x86_64-w64-mingw32"

> ver$os

[1] "mingw32"

> ver$version.string
```

```
[1] "R version 4.2.1 (2022-06-23 ucrt)"
```

I am using R with version R version 3.4.4 (2018-03-15) with in linux-gnu. Why do one need to know all this in formation at all. One reason is, of course, after all a user is a user! It is the duty to know as what a user is using for his/her work. Worrying with a language without knowing anything about it is ridiculous. The another reason is: at time we might use version information while working in different OSes. I got to do this while I was writing few functions related to big data and parallel processing in R. In parallel processing, the user have to know as in which OS he/she is working.

```
> cat("I am using", ver$version.string, "in", ver$os)
```

```
I am using R version 4.2.1 (2022-06-23 ucrt) in mingw32
```

R is programming language and every programming language use *workspace* when we start the program. This particular concept together with *namespace* is highly important for developers. [20] There is guy called *Hadley Wickham*, he is very popular for his contributions to R community. He did write few very useful yet popular packages for very pretty needs of data analysis. One of his package *ggplot2* is highly popular and also has very efficient methods to plot variety of visuals. He is just like celebrity in R community. He has very nice blog or website. Visit http://hadley.nz/ to know more about him. He had written a book known as *Advanced R*, you can read it at http://adv-r.had.co.nz/. He has provided very nice tutorial on workspace and namespace in R. I may not be able to bring this topic for discussion here, because it is beyond the scope of this book. However, I can help you know in which *Environment* you are in and you can do that by using a command or function `sessionInfo()`. The following is the sample information.

```
R version 3.4.4 (2018-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 18.04.3 LTS

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.
    ↪ so.3.7.1
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/
    ↪ liblapack.so.3.7.1
```

```
locale:
 [1] LC_CTYPE=en_IN.UTF-8        LC_NUMERIC=C
   ↪                    LC_TIME=en_IN.UTF-8
 [4] LC_COLLATE=en_IN.UTF-8      LC_MONETARY=
   ↪ en_IN.UTF-8     LC_MESSAGES=en_IN.UTF-8
 [7] LC_PAPER=en_IN.UTF-8        LC_NAME=C
   ↪                       LC_ADDRESS=C
[10] LC_TELEPHONE=C
   ↪ LC_MEASUREMENT=en_IN.UTF-8
   ↪ LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils
   ↪ datasets  methods   base

loaded via a namespace (and not attached):
[1] compiler_3.4.4 tools_3.4.4
```

This information represents entire information about Global Environment. You will find a word *namespace* at the bottom of the message. If you are global environment then the function `environment()` should be able to get you the following information.

```
> environment()
```

```
<environment: R_GlobalEnv>
```

Do you find there is little more detailed information in the above sample message. *attached base packages.* These are the base packages that were installed when we installed *R Base.* These packages are highly useful for basic statistical analysis. If you want to know what are other packages that were installed in your computer. You need to execute `installed.packages()`.

```
> ip <- as.data.frame(installed.packages())
> dim(ip)
```

```
[1] 259  16
```

```
> dim(df.pkgs)
```

```
[1] 18669    17
```

```
> dim(ip)[1]/dim(df.pkgs)[1]
```

```
[1] 0.01387327
```

Only 246 packages out of 15774 packages available from the CRAN were installed in my computer. So, I just have 0.01 percent of stuff written by the whole community inside my machine. Yet I do all that advanced analysis using very few may be one or two of them.

1.3.2 Packages & functions

Packages are very special in R. When R is installed it creates a simple system in the computer. This system supports most of the basic needs of statistical analysis. The Package is a collection of several ingredients thy are *functions, data sets, manuals, vignettes* and several other things. Suppose the user wish to do some analysis such as arithmetic mean. How do he or she know. One way is to attend classes or by using Google. R has pretty well defined methods for searching functions. Imagine that I would like to find a function to compute arithmetic mean and many people call it simply *mean* for arithmetic average is a *simple mean.* I might be able to do as below:

```
> grep("mean", ls("package:base"))

[1] 720 721 722 723 724 725

> ls("package:base")[c(grep("mean", ls("package:base")))]

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

The main function is `ls()` which is highly sought after function in Linux systems. `ls()` is used for most of the needs while searching for contents in the directory, especially in Linux systems. The function `ls()` is just a wraper for OS function. When we install R in Windows it has MinGW system through which it has access to these Linux based functions. The statement `ls("package:base")` gets all the functions available from package with a name *base.* Do you remember this is one of the package names we have notice while executing `sessionInfo()` above. The statement `grep("mean", ls("package:base"))` retrieves all the functions available from package *base.* However, this function retrieves information as a list of numbers. This means the function we are searching is available in those positions in the list created by this statement. From the above sample output it is clear that the function *mean* is available from the package *base.*

Now the challenge is how do we need to use this function. For that we have another function called *help()*. This function is highly useful to know about any function in R system. Suppose we would like to know about our newly found out function i.e. *mean*. We can do as below.

```
> help("mean")
```

Those quotes are very important, because the word is a string. This statement will bring up a manual right side under section "Help" in RStudio. This manual has a content called "example" at the bottom. Suppose if you are interested in knowing only example, then it can be done by using a function `example()`.

```
> example("mean")
```

This output really helps in understanding what is all about "mean". In this example it is clear that; the *mean* is a function and it expects an argument this can be a vector of numbers.

1.3.3 Installing packages

The package is an appropriate way to organize the work and share it with others. Typically, a package will include code, documentation for the package and the functions inside, some tests to check everything works as it should, and data sets.

Packages in R Programming language are a set of R functions, compiled code, and sample data. These are all together called as a "library" within the R environment. [21] By default, R installs a group of packages during installation. Once we start the R console, only the default packages are available by default. Other packages that are already installed need to be loaded explicitly to be utilized by the R program thats getting to use them.

Repositories

A repository is a place where packages are located and stored so you can install packages from it. Organizations and Developers have a local repository, typically they are online and accessible to everyone. Some of the most popular repositories for R packages are:

CRAN: Comprehensive R Archive Network(CRAN) is the official repository, it is a network of ftp and web servers maintained by the R community around the world. The R community coordinates it, and for a package to be published in CRAN, the Package needs to pass several tests to ensure that the package is following CRAN policies.

Bioconductor: Bioconductor is a topic-specific repository, intended for open source software for bioinformatics. Similar to CRAN, it has its own submission and review processes, and its community is very active having several conferences and meetings per year in order to maintain quality.

Github: Github is the most popular repository for open source projects. Its popular as it comes from the unlimited space for open source, the integration with git, a version control software, and its ease to share and collaborate with others.

Install packages

There are multiple ways to install R Package, some of them are,

Installing Packages From CRAN: For installing Package from CRAN we need the name of the package and use the following command:

```
install.packages("package name")
```

Installing Package from CRAN is the most common and easiest way as we just have to use only one command. In order to install more than a package at a time, we just have to write them as a character vector in the first argument of the `install.packages()` function.

```
install.packages(c("vioplot", "MASS"))
```

Installing Bioconductor Packages: In Bioconductor, the standard way to install a package is by first executing the following script:

```
source("https://bioconductor.org/biocLite
    ↪ .R")
```

This will install some basic functions which are needed to install *bioconductor* packages, such as the `biocLite()` function. To install the core packages of *Bioconductor* just type it without further arguments:

```
biocLite()
```

If we just want a few particular packages from this repository then type their names directly as a character vector:

```
biocLite(c("GenomicFeatures", "
    ↪ AnnotationDbi"))
```

Update, Remove and Check installed packages

To check what packages are installed on your computer, type this command:

```
installed.packages()
```

To update all the packages, type this command:

```
update.packages()
```

To update a specific package, type this command:

```
install.packages("PACKAGE NAME")
```

Installing Packages Using RStudio UI

In R Studio goto *Tools* $->$ *Install Package*, and there we will get a pop-up window to type the package you want to install:

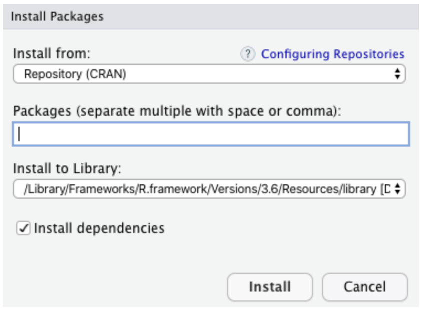


Figure 1.13: Install package menu window in R Studio

Under Packages, type, and search Package which we want to install and then click on install button.

Notes

[1]Most the content obtained from "Open-source software", https://en.wikipedia.org/wiki/Open-source_software. Wikipedia is a open source and collaborative knowledge management platform.

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

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

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

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

[6]Eric S. Raymond. "Goodbye, 'free software; hello, 'open source. http://www.catb.org/~esr/open-source.html"

[7]Please visit https://www.tiobe.com/tiobe-index/ for more information.

[8] Wrathematics, (27 August 2011). "How Much of R Is Written in R". librestats. http://librestats.com/2011/08/27/how-much-of-r-is-written-in-r/

[9]Visit https://www.gnu.org/licenses/agpl-3.0.en.html for details.

[10]At the time of writing this book I was using Ubuntu 18.04 the code name was "Bionic Beaver".

[11]You can also use Ubuntu launcher to open Terminal. Press *Windows Super Button*, the button located in the left bottom place at your keyboard. Then search for Terminal. This will show you Terminal Icon. Just click to open Ubuntu or Linux Terminal.

[12]Source: https://fedoraproject.org/wiki/DNF?rd=Dnf

[13]There is more documentation on `dnf` at https://docs.fedoraproject.org/en-US/quick-docs/dnf/

[14]Visit https://www.opensuse.org/ for more details.

[15]Visit http://www.mingw.org/ for more details on MinGW.

[16]Read more about Cygwin at https://www.cygwin.com/

[17]Obtained from https://en.wikipedia.org/wiki/Scripting_language

[18]Visit https://cran.r-project.org/ for more details on CRAN.

[19]There is really very useful information in R Bloggers with a name "Studying CRAN package names". This article has methods to retrieve and plot package information from CRAN. Visit https://www.r-bloggers.com/studying-cran-package-names/ for more information.

[20]Visit for more information on workspace and namespace.

[21]Use `help('library'` or simply ?library for documentation.

Chapter 2

Programming in R

2.1 Operators & Data Types in R

Programming languages typically support a set of operators. Operators are constructs which behave generally like functions, but which 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.

2.1.1 Assignment operators

R assignment operators. These operators are used to assign values to variables.

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

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

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:

```
> new_counter <- function() {
+     i <- 0
+     function() {
+         i <<- i + 1
+         i
+     }
+ }
> new_counter1 <- function() {
+     i <- 0
+     function() {
+         i <- i + 1
+         i
+     }
+ }
```

We have not really discussed anything about functions. There is some certain description on functions in forthcoming sections of this Chapter. For now, we try to execute this code using `Ctl+Enter` in RStudio. Now let us verify the code. We will try to create two objects `counter` and `counter1` to test above function and observe the behavior of these functions.

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

```
> counter <- new_counter()
> counter1 <- new_counter1()
> counter()

[1] 1

> counter()

[1] 2

> counter()

[1] 3

> counter1()

[1] 1

> counter1()

[1] 1
```

`counter()` and `counter1()` are two new objects created using the function `new_counter()`. When we execute `counter()` it gives different counters whereas `counter1()` gives a unique value i.e. 1. This is due to a particular scoping rule known as *functional scoping.* `i` in `new_counter()` is global so every time the function executes, it remembers `i`, whereas the `i` in `new_counter1()` is local. Every time, the value of `i` set back to 1.

2.1.2 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 [22] 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
```

2.1.3 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.

2.1.4 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*. [23] Try testing logical operators.

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

[1] FALSE

> a && b

[1] FALSE

> a | b

[1] TRUE

> a || b

[1] TRUE
```

2.2 Data types

To make the best of the R language, one needs a strong understanding of the basic data types and data structures and how to operate on them. Data structures are very important to understand because these are the objects you will manipulate on a day-to-day basis in R. Dealing with object conversions is one of the most common sources of frustration for beginners. Everything in R is an object. R has 6 basic data types.

1. Character
2. Numeric (real or decimal)
3. Integer
4. Logical
5. Complex

Elements of these data types may be combined to form data structures. R has few very valid data structures.

1. Atomic vector
2. Factors
3. List
4. Matrix
5. Data frame

R provides quite a few feature functions to examine above types. They are:

1. `class()` - what kind of object is it (high-level)?
2. `typeof()` - what is the object's data type (low-level)?
3. `length()` - how long is it? What about two dimensional objects?
4. `attributes()` - does it have any metadata?

Now let us try and understand few of the above mentioned data types

```
> a = "this is a character"
> is.character(a)
```

```
[1] TRUE
```

```
> b = 123
> is.numeric(b)
```

```
[1] TRUE
```

```
> c = 12
> is.integer(c)
```

```
[1] FALSE
```

```
> typeof(c)
```

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

```
> d = TRUE
> is.logical(d)
```

```
[1] TRUE
```

```
> e = complex(1, 2)
> is.complex(e)
```

```
[1] TRUE
```

That is why R has a special function `complex()` to deal with imaginary numbers. The syntax for the same is as follows:

```
complex(length.out = 0, real = numeric(), imaginary = numeric(),
        modulus = 1, argument = 0)
```

2.2.1 Strings

Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string within double quotes, even when they are created with single quote.

Rules for string construction

- The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
- Double quotes can be inserted into a string starting and ending with single quote.
- Single quote can be inserted into a string starting and ending with double quotes.
- Double quotes can not be inserted into a string starting and ending with double quotes.
- Single quote can not be inserted into a string starting and ending with single quote.

Examples of valid strings

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

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

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

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

When the above code is run we get the following output

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

Examples of invalid strings

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

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

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

When the above code executed R produces errors .

String concatenation

Many strings in R are combined using the `paste()` function. It can take any number of arguments to be combined together. Use `help('pase')` for more information or for syntax.

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

Above code produce the following output

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

Formatting numbers & strings

Numbers and strings can be formatted to a specific style using `format()` function.

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

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

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

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

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

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

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

```
print(result)
```

Following is the output for the above code.

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

Counting number of characters in a string

`nchar()` function counts the number of characters including spaces in a string.

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

Above code produces the following result

```
[1] 30
```

Changing the case

`toupper()` and `tolower()` functions change the case of characters of a string.

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

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

Above code produces the following result

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

Extracting parts of a string

`substring()` function extracts parts of a String.

```
# Extract characters from 5th to 7th position
   ↪ .
result <- substring("Extract", 5, 7)
print(result)
```

Above code produces the following result

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

String substitutions

`gsub()` function in R Language is used to replace all the matches of a pattern from a string. If the pattern is not found the string will be returned as it is.

```
# R program to illustrate the use of gsub()
   ↪ function

# Create a string
> x <- 'hi I am fine'
> gsub('I am', 'Are you', x)
[1] "hi Are you fine"
> y <- gsub('I am', 'Are you', x)
> y
[1] "hi Are you fine"
```

Another example

```
> courses <- c('R course', 'Python course', '
   ↪ Java course')
> programming <- gsub('course', 'programming
   ↪ ', courses)
> programming
[1] "R programming"      "Python programming"
   ↪  "Java programming"
```

Another example

```
> employees <- c('M. Kamakshaiah', 'C.
   ↪ Bhaskar', 'L. Narayana')
```

```
> employees <- gsub('M', 'Musunuru',
    ↪ employees)
> employees
[1] "Musunuru. Kamakshaiah" "C. Bhaskar"
    ↪             "L. Narayana"
> employees <- gsub('C.', 'Challa', employees
    ↪ )
> employees
[1] "Musunuru. Kamakshaiah" "Challa Bhaskar"
    ↪         "L. Narayana"
> employees <- gsub('L.', 'Laggi', employees)
> employees
[1] "Musunuru. Kamakshaiah" "Challa Bhaskar"
    ↪         "Laggi Narayana"
```

Text transformation

Text mining is one of the influential activity in data analysis. Text mining, also referred to as text data mining, similar to text analytics, is the process of deriving high-quality information from text. It involves "the discovery by computer of new, previously unknown information, by automatically extracting information from different written resources". The process of text mining involves conversion of text into sentences and words so that such conversion results into a reasonable *term vectors* or *document term matrices*.

This section deals with certain text known as *lorem ipsum*. The text is being split into corresponding words and then these words will be enumerated so as to create a term vector.

```
> string_obj <- "Lorem Ipsum is simply dummy
    ↪ ..... Lorem Ipsum."
> words <- strsplit(string_obj, " ", fixed =
    ↪ T)
```

"..." is used in the above code to indicate missing text for brevity of code. Following are the properties of string object (`string_obj`).

```
 typeof(string_obj)
[1] "character"
> length(string_obj)
[1] 1
> is.vector(string_obj)
```

```
[1] TRUE
```

The string object is a vector of length 1. This means the whole text, which is just a collection of words is being treated as one single entity. Now let us divide the whole sentence into a list of words and then those words need to be converted to a data frame to make this textual data compatible for data analysis.

```
> words <- strsplit(string_obj, " ", fixed =
    ↪ T)
> length(words[[1]])
[1] 91
> head(words[[1]])
[1] "Lorem"   "Ipsum"   "is"      "simply"  "
    ↪ dummy"  "text"
> tail(words[[1]])
[1] "PageMaker" "including" "versions"  "of"
    ↪        "Lorem"     "Ipsum."
```

However, the vector "words" is still a list but not a data frame. However, it is possible to convert this list of words into data frame with the help of following procedure.

```
> term_vector <- data.frame(table(words))
> typeof(term_vector)
[1] "list"
> class(term_vector)
[1] "data.frame"
> head(term_vector)
   words Freq
1 1500s,    1
2  1960s    1
3      a    2
4  Aldus    1
5   also    1
6     an    1
> tail(term_vector)
        words Freq
64 unchanged.    1
65    unknown    1
66   versions    1
67        was    1
68       when    1
```

```
69        with    2
```

Since *term_vector* is a data frame, it is possible to perform any statistical analysis based on the type of the data frame.

```
> max(term_vector[, 2])
[1] 6
> min(term_vector[, 2])
[1] 1
> mean(term_vector[, 2])
[1] 1.318841
> median(term_vector[, 2])
[1] 1
> sd(term_vector[, 2])
[1] 0.8659023
```

Let us make a simple scatter diagram using the data columns in the *term vector*.

```
> layout(matrix(1:2, 1, 2))
> plot(term_vector[, 2]); text(term_vector[,
   ↪ 1])
> plot(term_vector[, 2]); text(term_vector[,
   ↪ 2], as.character(term_vector[, 1]))
```

The function `layout()` allows the user to make the plots in an order as defined in `matrix()` function inside `layout()` function. The other function `plot()` is a generic function to make plots for given data. This code produce a graph with canvas with two distinct plots for data inside *term_vector* data set. There will be more discussion on plots in the forthcoming chapters related to statistical analysis. The above code produces a figure 2.1.

It is possible to find out outliers with the help of box plot.

```
> library(car)
> box_plot <- Boxplot(term_vector[, 2], id.
   ↪ method = term_vector[, 1])
> box_plot
 [1] 58 34 38  7 26  3 14 20 30 57
> term_vector[box_plot, ]
   words Freq
58   the    6
34 Lorem    4
```

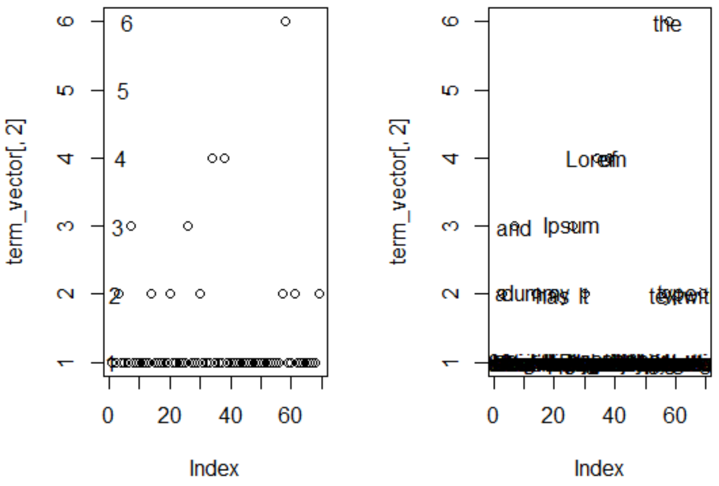


Figure 2.1: Term vector data matrix

```
38     of     4
7     and     3
26  Ipsum     3
3       a     2
14  dummy     2
20    has     2
30     It     2
57   text     2
```

Rows *58, 34, 38, 7, 26, 3, 14, 20, 30, 57* are outliers. The plot will be as shown at 2.2.

2.2.2 Vectors

A vector is the most common and basic data structure in R and is pretty much the workhorse of R. Technically, vectors can be one of two types: (1) atomic vectors and (2) lists. Although the term *vector* most commonly refers to the atomic types not to lists. Although we have a special function `vector()`, we normally don't

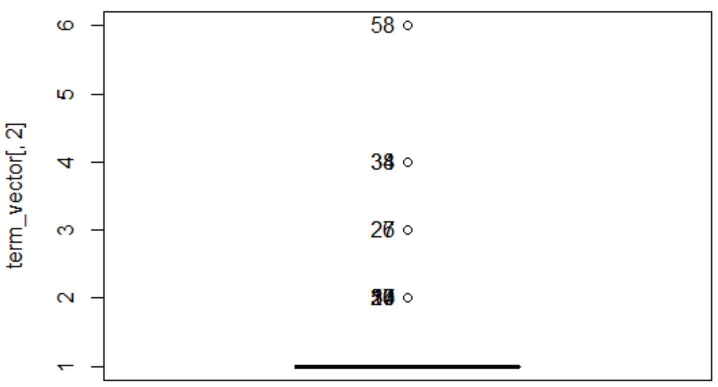


Figure 2.2: Box plot for term vector

make much use of it, but we have a couple of other functions to do same. They are `c()` and colon operator (`:`).

```
> x <- c(1:10)
> typeof(x)
```

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

```
> y <- c("a", "b", 1:10)
> typeof(y)
```

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

The `c` function can also be used to append the existing data vector.

```
> c(x, 11)
```

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

There is one more way to create number sequences that is by using `seq()` function. It is rather more dynamic compared to colon operator.

```
> z <- seq(1, 10, 0.5)
> length(z)
```

```
[1] 19
```

Indexing vectors

Vectors can be indexed using [] (square brackets) in R. For instance

```
> z[1]; z[2]
[1] 1
[1] 1.5
```

Vectors are unidimensional or one-dimensional data structure in R. So arguments inside brackets can be single numbers. How ever it is possible to use any other commands such as colon (:), *c*(), and *seq*().

```
> z[1:5]
[1] 1.0 1.5 2.0 2.5 3.0
> z[seq(1, 2)]
[1] 1.0 1.5
> z[c(1, 3, 5)]
[1] 1 2 3
```

Please note R doesn't follow zero based indexing rule such as in other conventional programming lanauges like Java and Python. It is also use the indexing dynamically using other operators such as relational, logical etc.

```
> z[z<5]
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5
> z[z>=5]
 [1]  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5
    ↪  9.0  9.5 10.0
> z[z!=5:7]
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5
    ↪  5.0  5.5  6.5  7.0  7.5  8.0  8.5
    ↪ 9.0
[17]  9.5 10.0
```

%in% operator

It is also possible compare and retrieve values from one vector with another vector. Suppose if there is a vector which has a sequence of linear number series from 1 to 5 with 0.5 increment and another vector with just a linear number series from 1 to 3. It is possible to compare both and retrieve values as shown below.

```
> vec <- seq(1, 5, 0.5)
> vec
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> vec1 <- 1:3
> vec2 <- 6:8
> vec %in% vec1
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
   ↪  FALSE FALSE
> vec[vec %in% vec1]
[1] 1 2 3
> vec1 %in% vec
[1] TRUE TRUE TRUE
> vec[vec1 %in% vec]
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

There are a couple of other functions such as `setdiff`, `union`, `intersection`, `merge`, `...` to take care of advanced data processing.

Vector arithmetic

It is possible to make arithmetic calculations such as addition, subtraction, multiplicatin and division on vectors.

```
> vec <- 1:10
> vec + 1:3
 [1]  2  4  6  5  7  9  8 10 12 11
Warning message:
In vec + 1:3 :
  longer object length is not a multiple of
     ↪ shorter object length
```

Though it is possible to add vectors with different lengths but will produce warning message. Notice, the addition will repeat across the length of the first vector (vec). This is called *"vector recycling"* in R. However, it is not so with vectors with same lenghts

```
> vec + 2:11
 [1]  3  5  7  9 11 13 15 17 19 21
> vec - 2:11
```

```
 [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
> vec * 2:11
 [1]   2   6  12  20  30  42  56  72  90 110
> vec / 2:11
 [1] 0.5000000 0.6666667 0.7500000 0.8000000
    ↪ 0.8333333 0.8571429 0.8750000
    ↪ 0.8888889
 [9] 0.9000000 0.9090909
```

2.2.3 Factors

Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values. Like ”Male, ”Female” and True, False etc. They are useful in data analysis for statistical modeling. Factors are created using the `factor()` function by taking a vector as input.

```
> factdat <- sample(c("male", "female"), 10, replace = TRUE)
> length(factdat)

[1] 10

> is.factor(factdat)

[1] FALSE

> summary(factdat)

   Length     Class      Mode
       10 character character
```

I used a function `sample()` to simulate gender like data variable. Use `help('sample')` to know about this function. The user object `factdat` is not a *factor*. Why? Because, everything is a vector in R unless defined by other methods. Now let us convert `factdat` into a valid factor using `factor()` method.

```
> factdatch <- factor(factdat)
> length(factdatch)

[1] 10

> is.factor(factdatch)

[1] TRUE

> summary(factdatch)

female   male
     4      6
```

The variable `factdatch` is a factor and it has two levels namely male and *female*. The command `is.factor()` shows that it is “TRUE”. The attribute *levels* differentiate factor from vector.

2.2.4 Lists

A List, in R, is a collection of various other data structures. This means a List can contain elements of different types like: numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using `list()` function. Below code snippet shows as to how list are created and indexed.

```
> a <- 'this is a character'
> b <- 123
> c <- 12
> li <- list(a, b, c)
> li <- list(a, b, c)
> li[[1]]
[1] "this is a character"
> li[[2]]
[1] 123
> li[[3]]
[1] 12
> li[1]
[[1]]
[1] "this is a character"

> li[2]
[[1]]
[1] 123

> li[3]
[[1]]
[1] 12
```

The object `li` is user object of the type *list*, since it was created using the function `list()`. The number inside square brackets represents list *index*. The indexing works with both ways as by using double square brackets and also with single. Loops structures like `for(), while()` can be used even for such tasks.

```
> li1 <- list()
> li1[1] <- a
> li1[2] <- b
> li1[3] <- c
> li1

[[1]]
[1] "this is a character"

[[2]]
[1] 123

[[3]]
[1] 12
```

This time we created a user object with a name `li1` and assigned *a*, *b* and *c* each at a time. We can also use loops to accomplish above task in case of lengthy lists. Lists are highly dynamic objects in R. Lists are used while calculating outputs while executing functions.

Vectors to List

Now that it is clear about vectors and lists let us focus on creating a list using vectors. It is possible to create a list using a couple of vectors. For instance, if there are two vectors with different types of data they can be made into a List using `list()` function.

```
> x <- round(runif(10)*100, )
> is.vector(x)
[1] TRUE

> y <- sample(c('male', 'female'), 10,
   ↪ replace = T)
> is.vector(y)
[1] TRUE

> dataset <- list()
> dataset
list()
> class(dataset)
[1] "list"
> dataset$salary <- x
> dataset$gender <- y

> dataset
```

```
$salary
 [1] 18 57 77 77 26 85 21 85 84 39

$gender
 [1] "female" "female" "male"   "male"   "
    ↪ female" "female" "male"   "male"
 [9] "male"   "female"
```

Both *salary* and *gender* in above code are vectors. These vectors are passed into the list object which is *dataset*. This list can be converted to *data frame*, which is a special data structure in R. We will see that procedure in the forthcoming section.

2.2.5 Matrix

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. Matrix is similar to *vector* but additionally contains the dimension attribute. All attributes of an object can be checked with the `attributes()` function. Dimension can be checked directly with the `dim()` function. Dimension of the matrix can be defined by passing appropriate value for arguments *nrow* and *ncol*. The following script creates a matrix of the order 3×3.

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

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

We can see that the matrix is filled column-wise. This can be reversed to row-wise filling by passing TRUE to the argument byrow.

```
> matrix(1:9, nrow = 3, byrow = TRUE)

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

This time matrix gets values by row in stead of by column. It is possible to name the rows and columns of matrix during creation by passing a 2 element list to the argument `dimnames`.

```
> x <- matrix(1:9, nrow = 3, dimnames = list(c("X", "Y", "Z"),
+     c("A", "B", "C")))
> colnames(x)

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

> rownames(x)

[1] "X" "Y" "Z"

> colnames(x) <- c("v1", "v2", "v3")
> rownames(x) <- 1:3
> typeof(x)

[1] "integer"

> x

  v1 v2 v3
1  1  4  7
2  2  5  8
3  3  6  9

> is.matrix(x)

[1] TRUE
```

The object x is still a matrix.

2.2.6 Data Frames

Data frame is a two dimensional data structure in R. It is a special case of a list which has each component of equal length. Each component form the column and contents of the component form the rows. In short, the data frame is also a list of vectors which are of equal length. A matrix contains only one type of data whereas a data frame can contain different data types. Data frames can be accessed like a matrix by providing index for row and column. The data frames when indexed using row names or column names it is called “slicing”.

```
> x <- data.frame(x)
> typeof(x)

[1] "list"
```

The object x, in the above chunk, is a matrix which was created in previous section. Earlier x was *integer* when it was a matrix, but now it is *list* after converting into a `data.frame()`.

Slice Data Frame

It is possible to slice contents of a Data Frame. Usually numbers corresponding to positions of rows and columns of a data frame are used to retrieve data available in rows and columns. Suppose if a data frame is composed of rows and columns, such as `df[m, n]`, where m represents number of rows and n represents the columns, then it is possible to retrieve data from rows and columns using square brackets correspondingly using numbers of those rows and columns.

```
> dataf <- data.frame(airquality)
> head(dataf)

  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6

> dataf[1:6, ]

  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

I just created an user object with a name `dataf`. This object was assigned with a data set *airquality* which is available from a package known as *datasets*. Data frames has attribute `names()` which will reveal column names of the data frame.

```
> names(dataf)

[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"
```

```
> names(airquality)

[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"
```

Column names for both *airquality* and *dataf* are same. There is a function called `subset()`, which can be very much helpful while using data frames. You know in `airquality` data set, there are missing values. Let us use `subset()` to offset all missing data.

```
> sum(is.na(dataf["Solar.R"]))

[1] 7

> head(subset(dataf, subset = !is.na(Solar.R)))

  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5   8
```

I not only created a subset but with a filter such as the subset created by above statement has all the values that are no missing in column `Solar.R` inside the object dataf. By the way, there are 7 such rows in which the variable `Solar.R` has missing values.

2.2.7 Special types

R language supports several *null-able* values and it is relatively important to understand how these values behave, when making data pre-processing and data munging.

In general, R supports:

- NULL
- NA
- NaN
- Inf / -Inf

NULL is an object and is returned when an expression or function results in an undefined value. In R language, NULL (capital letters) is a reserved word and can also be the product of importing data with unknown data type. *NA* is a logical constant of length 1 and is an indicator for a missing value. NA (capital letters) is a reserved word and can be coerced to any other data type vector (except raw) and can also be a product when importing data. NA and "NA" (as presented as string) are not interchangeable. *NA stands for Not Available. NaN* stands for *Not A Number* and is a logical vector of a length 1 and applies to numerical values, as well as real and imaginary parts of complex values, but not to values of integer vector. NaN is a reserved word. *Inf* and *-Inf* stands for infinity (or negative infinity) and is a result of storing either a large number or a product that is a result of division by zero. *Inf* is a reserved word and is, in most cases, product of computations in R language and therefore very rarely a product of data import. Infinite also tells you that the value is not missing and a number.

All four null/missing data types have accompanying logical functions available in base R; returning the TRUE/FALSE for each of particular function: `is.null(), is.na(), is.nan(), is.infinite()`. General understanding of all values by simply using following code:

```
##reading documentation on all data types:
?NULL
?NA
?NaN
?Inf

##populating variables
a <- "NA"
b <- "NULL"
c <- NULL
d <- NA
e <- NaN
f <- Inf

## Check if variables are same?
identical(a,d)
# [1] FALSE
# NA and NaN are not identical
identical(d,e)
# [1] FALSE

##checking length of data types
length(c)
# [1] 0
```

```
length(d)
# [1] 1
length(e)
# [1] 1
length(f)
# [1] 1

##checking data types
str(c); class(c);
#NULL
#[1] "NULL"
str(d); class(d);
#logi NA
#[1] "logical"
str(e); class(e);
#num NaN
#[1] "numeric"
str(f); class(f);
#num Inf
#[1] "numeric"
```

Nullable data types can have a different behavior when propagated to e.g.: list or or vectors or data.frame types. We can test this by creating NULL or NA or NaN vectors and dataframes and observe the behaviour:

```
#empty vectors for NULL, NA and NaN
v1 <- c(NULL, NULL, NULL)
v2 <- NULL
str(v1); class(v1); mode(v1)
str(v2); class(v2); mode(v2)

v3 <- c(NA, NA, NA)
v4 <- NA
str(v3); class(v3); mode(v3)
str(v4); class(v4); mode(v4)

v5 <- c(NaN, NaN, NaN)
v6 <- NaN
str(v5); class(v5); mode(v5)
str(v6); class(v6); mode(v6)
```

Clearly, it is evident that the NULL vector will always be an empty one, regardless of the elements it can hold. With NA and NaN, it will be the length of the elements it holds, with a slight difference, that NA will be a vector of class Logical, whereas NaN will be a vector of class numeric. NULL vector will not change the size but class when combined with a mathematical operation:

```
#operation on NULL Vector
v1 <- c(NULL, NULL, NULL)
str(v1)
# NULL
v1 <- v1+1
str(v1)
# num(0)
```

This will only change the class but not the length and still any of the data will not persist in the vector. With data.frames it is relatively the same behavior.

```
#empty data.frame
df1 <- data.frame(v1=NA,v2=NA, v3=NA)
df2 <- data.frame(v1=NULL, v2=NULL, v3=NULL)
df3 <- data.frame(v1=NaN, v2=NaN, V3=NaN)
str(df1); str(df2);str(df3)
```

Dataframe consisting of NULL values for each of the column will presented as dataframe with 0 observations and 0 variables (0 columns and 0 rows). Dataframe with NA and NaN will be of 1 observation and 3 variables, of logical data type and of numerical data type, respectively. When adding new observations to data frames, different behavior when dealing with NULL, NA or NaN. Adding to “NA” data.frame:

```
# adding new rows to existing dataframe
df1 <- rbind(df1, data.frame(v1=1, v2=2,v3=3)
   ↪ )
#explore data.frame
df1
```

It is clear that new row is added, and when adding a new row (vector) of different size, it will generate error, since the dataframe definitions holds the dimensions. Same behavior is expected when dealing with NaN value. On the other hand, different results when using NULL values:

```
#df2 will get the dimension definition
df2 <- rbind(df2, data.frame(v1=1, v2=2))
#this will generate error since the dimension
   ↪ definition is set
df2 <- rbind(df2, data.frame(v1=1, v2=NULL))
#and with NA should be fine
df2 <- rbind(df2, data.frame(v1=1, v2=NA))
```

with first assignment, the df2 will get the dimension definition, albeit the first construction of df2 was a nullable vector with three elements. NULLable is also a result when we are looking in the vector element that is not existent, due to the fact that is out of boundaries:

```
l <- list(a=1:10, b=c("a","b","c"), c=seq
   ↪ (0,10,0.5))
l$a
# [1]  1  2  3  4  5  6  7  8  9 10
l$c
# [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5
   ↪   4.0  4.5  5.0  5.5  6.0  6.5  7.0
   ↪ 7.5  8.0  8.5  9.0  9.5 10.0
l$r
# NULL
```

We are calling the sublist r of list l, which is a NULL value, but is not missing or not existing, it is NULL, which in fact is rather contradictory, since the definition is not set. Different results (Not Available) would be returned when calling a vector element:

```
v <- c(1:3)
v[4]
#[1] NA
```

Boundaries in list and in vector are defined differently for NA and NULL data types.

2.2.8 Type conversions

Whenever, an object is created in R, R attach certain information to it. For instance,

```
> x <- 2
> mode(x)
[1] "numeric"
> typeof(x)
[1] "double"
> class(x)
[1] "numeric"
```

In the above code the object x is a variable that contains 2. The moment x is created R infers qualities of that variable automatically and attach certain information. For instance, the function `mode()` will let us know about the base type of that variable. The function `typeof()` will let us know whether it is integer, double or anything else. These two commands deals with a single property of the variable know as *storage mode*. The storage mode determines the very type of the data. Every objects has a kind of storage mode and this is very essential for data transformation. Look into the below chunk

```
> is.numeric(x)
[1] TRUE
> is.character(x)
[1] FALSE
> x <- as.character(x)
> is.numeric(x)
[1] FALSE
> is.character(x)
[1] TRUE
```

What's happening here? We know that R attache certain base information to objects at the same time R also gives the capability to the user to manipulate this information with the help of certain helper function that starts as with `is.` and `as.`. For instance, if the object is numeric, it can be ascertained using `is.numeric` and if needs to converted to the other data type known as character then we can use `as.character`. In the above code the object x is numeric and it was converted to character using `as.character`. This is called *data type conversion*.

The same act can be done to other data types and structures. For instance, let us deal with matrix to data frame and *vice versa*.

```
> x <- matrix(1:9, 3, 3)
> is.matrix(x)
```

```
[1] TRUE
> rownames(x)
NULL
> colnames(x)
NULL
```

The object x is a matrix and it has now row names `rownames()` or column names `colnames()`. Let us convert that using the function `data.frame` (also possible to use `as.data.frame()`)

```
> y <- data.frame(x)
> is.matrix(y)
[1] FALSE
> is.data.frame(y)
[1] TRUE
> colnames(y)
[1] "X1" "X2" "X3"
> rownames(y)
[1] "1" "2" "3"
```

The object has both row names and column names after data conversion.

```
> y
  X1 X2 X3
1  1  4  7
2  2  5  8
3  3  6  9
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

We already used a function called `class()` in the previous sections. This function returns the base data type or structure when used on R objects. This function is a bit different compared to *mode()* and *typeof()*. The function `class()` is required while writing OOP programs, which is beyond the objectives of this text. However, `class()` together with `typeof()` is pretty much useful to know about the type or structure of data.

```
> class(2)
```

```
[1] "numeric"
> class(x <- 2)
[1] "numeric"
> class(1:5)
[1] "integer"
> class(pi)
[1] "numeric"
> class(x)
[1] "matrix" "array"
> .class2(x)
[1] "matrix"  "array"   "integer" "numeric"
> ?.class2 # help on function
```

First few statements in the above code chunk are related atomic vectors. The object x is a matrix created in the earlier section. The function `.calss2()` gives detailed information about the base object.

2.2.9 Data Import

R supports variety of data files such as Excel, Minitab, SPSS, Text, and CSV. For simplicity sake I shall describe CSV file format because CSV is open file format and every software has mechanisms to convert data sets into CSV files.

In CSV file each cell inside such data file is separated by a special character, which usually is a *comma*, although other characters can be used as well. The first row of the data file should contain the column names instead of the actual data.

There is a function known as `read.csv()` in R. The syntax for this function is as follows:

```
read.csv(file, header = TRUE, sep = ",", quote = "\"",
              dec = ".", fill = TRUE, comment.char = "", ...)
```

We need to pass few parameters such as *file, header, sep, quote, dec, fill, comment.char*. Actually we don't need to care all these parameters if we have the data structured correctly.

```
> markdf <- read.csv("D:/Work/Books/R/R-for-data-analytics/Datasets/mark_dat.csv")
> names(markdf)

 [1] "gender"         "employment"     "marital.status" "salary"
 [5] "age"            "sat1"           "sat2"           "sat3"
 [9] "per1"           "per2"           "per3"           "att1"
[13] "att2"           "att3"

> typeof(markdf)
```

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

But is this a data frame? Yes.

```
> is.data.frame(markdf)
```

```
[1] TRUE
```

Is this data frame matrix? No.

```
> is.matrix(markdf)
```

```
[1] FALSE
```

So the function `read.csv()` not only imports data but converts the same as *data frame* and makes ready for analysis.

2.3 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.[24] 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." [25]

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.* [26] 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] 644

> primfuncs[644]

[1] "length"

> 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: 0x5621bad5f0b8>
<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: 0x5621bcf81110>
```

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.

2.3.1 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.

2.3.2 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 2.3: 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.

2.3.3 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]  97.65685 101.64638 111.76556 117.94612 118.26682

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

[1] 81.38071 84.70532 93.13797 98.28844 98.55568
```

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]  91.30515 198.25592  53.73670  43.84888  44.57360

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

```
[1] 45.65258 99.12796 26.86835 21.92444 22.28680
```

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.* [27] 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. [28]

2.4 User defined functions (UDF)

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.

2.4.1 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 **??** 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:

$$ESS = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2 \qquad (2.1)$$

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

$$TSS = ESS + RSS \qquad (2.2)$$

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.

2.4.2 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,] -0.11861137 -0.6682503  0.2252302 -0.5188027
[2,] -1.09420659 -0.2965178  0.2479521  0.8529760
[3,] -0.03130682 -0.4773959  0.7322009  1.5843060
[4,] -1.39548935 -1.0344873 -0.9712821 -2.3255532
```

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 2.5.2.

2.4.3 Zeros matrix

There is certain discussion related to a method called `matrix` in the previous Chapter in the Section **??**. 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
```

2.4.4 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." [29]

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

```
      [,1]
[1,] -2.85
```

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 2.5.2

2.4.5 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 **??**. We will try to create two methods called append and remove using the logic explained in Section **??**.

```
> 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.

2.4.6 Statistical analysis

Above procedure only helps in picking functions as by the need and such functions are known as *in-built functions*. However, at times it may not be possible to get functions for all the needs of users. So, every programming language has got certain procedure to define user functions. When a user defines a function for his/her needs it is known as *User Defined Function* or simply *UDF*. R has wonderful methods to write UDFs. Writing or defining functions is beyond the scope of this book. However, I shall explain quickly as how to make functions in R.

R in-built function `mean()` can make calculation for a particular argument called `trim`. This argument can trim the data vector either side by a value supplied by the user. Let us tinker this functionality a bit differently. Suppose I am interested in finding arithmetic mean which is above or below a particular value in a given vector, then I might be able to define UDF as below:

```
> meanAboveBelow <- function(x, opt = "above") {
+     m = mean(x)
+     if (opt == "above") {
```

```
+         x = x[x > m]
+         return(mean(x))
+     }
+     else if (opt == "below") {
+         x = x[x < m]
+         return(mean(x))
+     }
+ }
```

The above function i.e. `meanAboveBelow` is going to calculate average of those values above average for user option (`opt`) “above” and calculate average for those values below average for user option (`opt`) “below”.

```
> meanAboveBelow(1:10)

[1] 8

> mean(6:10)

[1] 8

> meanAboveBelow(1:10, "below")

[1] 3

> mean(1:5)

[1] 3
```

2.5 Control structures

2.5.1 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.

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. [30]

```
> 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] 88 75 56 87 71
```

```
> convMarksIntoGrades(studMarks)
```

```
[1] "A+" "F"  "B"  "A+" "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.

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"
```

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. 31

2.5.2 Applications

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 2.3.1 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.

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 nonnumeric 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.* [32] This formula is useful while calculating bin vectors (through data binning) for ifelse statement. [33] Data bins are used by vectorized if statements inside this function.

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

[1] 50 88 62 80 83

> convToIntervals(x)

[1] "cat 1" "cat 5" "cat 2" "cat 4" "cat 5"
```

Random matrix

There was rather significant discussion in the Chapter 3 in Section 2.4.2 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,]  1.40 -0.31  2.15  0.80
[2,] -0.22  0.05 -0.76 -1.12
[3,] -0.04 -0.77 -0.78  1.54
[4,] -0.07 -1.93 -0.84 -0.96
```

Positive semidefinit matrices

There is certain description about positive semidefinit property of matrices in Section 2.4.4. 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 negative-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"
```

2.5.3 Loops

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.

```
> 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. [34] 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.

`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
```

`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.

2.5.4 Applications

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.

Loops for Lists

There is certain discussion on named and unnamed lists in Section **??**. 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]  0.33 -1.56  1.13 -1.35 -0.63

[[2]]
[1] -1.24 -0.03 -0.14 -0.77 -1.14

[[3]]
[1] -0.72 -0.57  0.22  0.05 -1.69
```

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.

Loops for summaries

In section **??**, 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] 3 2
```

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.

Loops for matrix operations

There was certain discussion related to matrix multiplication in Section **??**. 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.

2.6 Data sets

In R there are quite a few data sets. Most of these data sets are available from their respective packages. For instance, the data set *mtcars* is available in *cars* package. The function `data()` can list out all data sets available from one of the base package *datasets*. The package *datasets* is loaded when R open first time. Visit https://vincentarelbundock.github.io/Rdatasets/datasets.html for list of data sets available from package `datasets`. [35] These data sets can be downloadable either in CSV or DOC formats. Try the following command in the Console.

```
> head(data()[[3]][, 3])
```

```
[1] "AirPassengers"          "BJsales"                "BJsales.lead (BJsales)"
[4] "BOD"                    "CO2"                    "ChickWeight"
```

Above sample output retrieves first five data sets from the list of those data sets available in R base. For instance, the data set `AirPassengers` is a very famous one. Execute `help("AirPassengers")` to know more about this data set. RStudio opens a document file right bottom pane. Use `AirPassengers` to load data into workspace. Let us try data set with a name *BOD*. Use `help("BOD"` to know more about this data set. This data set is related to *Biochemical Oxygen Demand Description.* The BOD data frame has 6 rows and 2 columns giving the biochemical oxygen demand versus time in an evaluation of water quality. So this data set could be a better to start with.

```
> typeof(BOD)

[1] "list"

> names(BOD)

[1] "Time"   "demand"

> head(BOD)

  Time demand
1    1    8.3
2    2   10.3
3    3   19.0
4    4   16.0
5    5   15.6
6    7   19.8

> tail(BOD)

  Time demand
1    1    8.3
2    2   10.3
3    3   19.0
4    4   16.0
5    5   15.6
6    7   19.8
```

Functions `head()` and `tail()` retrieves top 6 and bottom 6 records of that particular data set respectively. Let us try for other data set *mtcars*. This data set has information related “Motor Trend Car Road Tests”. The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973-74 models).

```
> typeof(mtcars)

[1] "list"

> head(mtcars)

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

```
> tail(mtcars)
```

```
                mpg cyl  disp  hp drat    wt qsec vs am gear carb
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

This data set seems to be highly reasonable for analysis. We will try to use this data set in our forthcoming section. i.e. Visualization in R.

2.6.1 Simulating data sets

R really has wonderful data sets and these data sets highly useful while writing programs or testing code during the time of modeling. However, at times it is not possible to find required data sets for testing programs. I do write lot of programs and these programs at times possess novel methods and it may not be possible to find right data inputs for testing such programs. Programmers needs right kind of data inputs to test their code. Do you see how I developed a method `meanAboveBelow()` in section 2.4.6? I first defined a method and then I used a simple data vector simulated by a function called *colon operator*. I don't need to either find or import right type of data. In stead, it is easy to simulated data for just in time purposes. Suppose if I would like to create certain data set with few socioeconomic characteristics such as *gender, education, income* and few study characteristics such as *satisfaction (sat1, sat2, sat3)*, we might be able to do as below:

```
> gender <- sample(c("male", "female"), 10, replace = TRUE)
> education <- sample(c("educated", "uneducated"), 10, replace = TRUE)
> income <- runif(10, 10, 100)
> sat1 <- sample(1:5, 10, replace = TRUE)
> sat2 <- sample(1:5, 10, replace = TRUE)
> sat3 <- sample(1:5, 10, replace = TRUE)
> mydf <- data.frame(gender, education, income, sat1, sat2, sat3)
> names(mydf)
```

```
[1] "gender"    "education" "income"    "sat1"      "sat2"      "sat3"
```

```
> head(mydf)
```

```
  gender education   income sat1 sat2 sat3
1   male  educated 29.38215    1    5    1
2   male  educated 55.30326    1    2    5
3 female  educated 90.17175    3    5    5
4 female  educated 37.79188    5    5    3
5   male  educated 95.88960    2    2    1
6   male  educated 50.48423    4    4    4
```

```
> meanAboveBelow(mydf$income)
```

```
[1] 84.48651
> meanAboveBelow(mydf$income, "below")
[1] 44.56336
```

Those numbers that you see at the end of the listing are mean income figures for above average and below average samples. This is how programming can be done in R. R is highly powerful for methods related to functional programming. For that matter, everything is a function in R. Say colon operator, *c* operator, what not; anything and everything.

Notes

[22]In R everything is objective

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

[24]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/.

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

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

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

[28]*ibid.*

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

[30]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*.

[31]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.

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

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

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

[35]Visit https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html for a comprehensive view about datasets package.

2.7 Exercises

1. Know about history and background of open source software. Write a brief note on OSS and its legacy in cultivating open culture in various fields.

2. Study profiles of various individuals who created OSS momentum in software industry.

3. Visit www.gnu.org. Study about GNOME Operating System. Try to know as why appropriate way of calling Linux as GNU/Linux.

4. Study the history of operating systems. Contrast and compare UNIX with BSD and also a couple of other OSes.

5. Study different Desktop Environments (DE) such as GNOME, KDE etc.

Study and enumerate historical developments brought by DEs in computer industry.

6. Go to R portal (`www.r-project.org`). Study about various developments and also other useful information.

7. Go to CRAN section (menu) in R portal and study about different aspects of the same. Read and understand as how to install packages from CRAN.

8. Study about different types of editors available for R. Compare them using advantages and disadvantages.

9. Know about utility of IDE in writing code in general and R Studio in particular.

10. What is RGUI? Understand various menus of RGUI in Windows OS.

11. Know about importance of Linux BASH and its utility to do various user related activity in Linux.

12. Study different types of operators in R. Practice each of them using examples.

13. Write summary on data types and structure available for programming in R.

14. Differentiate vector from factor. Practice same with the help of examples.

15. Distinguish data matrix from data frame. Practice slicing data frame using examples.

16. Study about various types of data formats. Practice importing of different data files with variety of file formats.

17. What are delimited files? Write about importing delimited files in R.

18. Understand R `function()` statement. Write few User Defined Functions (UDF) for few statistics viz. mean, median, mode etc.

19. Study different types of data sets available from R package `datasets`. Also understand utility of these data sets for practical uses.

2.8 Case

New Delhi-headquartered Bharti Airtel has over 257 million subscribers across the country, as per the recent data released by the Telecom Authority of India (TRAI). It is also one of the youngest firms to use open source software on a large scale. Airtel has deployed a large number of open source technologies to improve efficiency. These include solutions such as Redis, Nginx, Apache Tomcat, Drools, JBoss Wildfly and Spring. Besides this, the company's vast user data is stored primarily on community-based offerings like AeroSpikeDB, MongoDB, OrientDB and PostgresDB.

Today, Airtel has as many as six major projects based on open source solutions. There is the Mitra app for retailers to acquire new customers and reverse wrongly made recharges; the Fibre Force app to help NMT engineers measure a patroller's performance; the Decision Tree app for call centre employees; the Hive social network app and the Project Leap open network initiative. Also, the telecom giant has recently developed its eCAF (electronic customer acquisition form) to digitize customer acquisition using Aadhaar biometric data.

Investigate as how Bharati Airtel has switched from proprietary software to open source. Enumerate course of events related to technology transformation and its impact on business success.

Chapter 3

Data Visualization

"Plot and plan like all good generals."

- E.A. Bucchianeri, Vocation of a Gadfly

Infographics, relatively a new word in computing and programming, are graphic visual representations of information, data, or knowledge intended to present information quickly and clearly. They can improve cognition by utilizing graphics to enhance the human visual system's ability to see patterns and trends. Similar pursuits are information visualization, data visualization, statistical graphics, information design, or information architecture. Infographics have evolved in recent years to be for mass communication, and thus are designed with fewer assumptions about the readers' knowledge base than other types of visualizations.

R has wonderful ways to plot data for it is language of statistics. I used to use a particular blog Frank McCown's write up titled "Producing Simple Graphs with R" in my earlier days of teaching R. It has really very simple methods for making very simple plots as by the type of the data. [36] As I mentioned earlier, there are endless opportunities in R for graphing and plotting. There are a couple of hundreds of packages for plotting data in R. Please visit cran task view "Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization" for more details. There is also a cran task view meant for computer vision and this is available at `https://cran.r-project.org/web/views/gR.html`. [37]

I have to mention a particular package called *ggplot2*, which is highly popular in R community. [38] Hadley Wickham is a celebrity in R community. He has earned his Ph.D from *Dianne H Cook* a world renowned academic and maverick of computer vision. Hadley wrote quite a few, yet very useful packages for community. Make it sure to visit his personal site at `http://hadley.nz/` to know more about him. In 2006 he was awarded the John Chambers Award for Statistical Computing for his work developing tools for data reshaping and visualization. I also have to mention another popular R programmer named *Yihui Xie*, who has earned his Ph.D from same professor along with Hadley. Yihui did very good works related to animations and documentation. His package *animation* is mind blowing. [39]

3.1 Graphic devices & Base plots

R follows a concept called *device* to plot graphs. A graphics device is something where you can make a plot appear. Examples include:

1. A window on your computer (screen device)
2. A PDF file (file device)
3. A PNG or JPEG file (file device)
4. A scalable vector graphics (SVG) file (file device)

When you make a plot in R, it has to be "sent" to a specific graphics device. The most common place for a plot to be "sent" is the screen device. On a Mac the screen device is launched with the `quartz()` function, on Windows the screen device is launched with `windows()` function, and on Unix/Linux the screen device is launched with `x11()` function.

When making a plot, you need to consider how the plot will be used to determine what device the plot should be sent to. The list of devices supported by your installation of R is found in `?Devices`. There are also graphics devices that have been created by users and these are available through packages on CRAN. Few devices used in R Base are:

1. pdf - Write PDF graphics commands to a file
2. postscript - Writes PostScript graphics commands to a file

3. xfig - Device for XFIG graphics file format
4. bitmap - bitmap pseudo-device via Ghostscript (if available).
5. pictex - Writes TeX/PicTeX graphics commands to a file (of historical interest only)

How Does a Plot Get Created?

There are two basic approaches to plotting. The first is most common. This involves

1. Call a plotting function like plot, xyplot, or qplot
2. The plot appears on the screen device
3. Annotate the plot if necessary
4. Enjoy

Multiple Open Graphics Devices

It is possible to open multiple graphics devices (screen, file, or both), for example when viewing multiple plots at once. Plotting can only occur on one graphics device at a time, though.

The currently active graphics device can be found by calling function `dev.cur()` in R. Every open graphics device is assigned an integer starting with 2 (there is no graphics device 1). You can change the active graphics device with `dev.set(<integer>)` where `<integer>` is the number associated with the graphics device you want to switch to.

The `plot()` function

The core plotting and graphics engine in R is encapsulated in the following packages:

1. `graphics`: contains plotting functions for the "base" graphing systems, including `plot, hist, boxplot` and many others.
2. `grDevices`: contains all the code implementing the various graphics devices, including *X11, PDF, PostScript, PNG*, etc.

The function `plot()` has automatic plotting mechanism. This functions knows as how to make plot based on data type. Suppose you want to plot a data set say *cars* using `plot()`

```
> plot(mtcars)
```

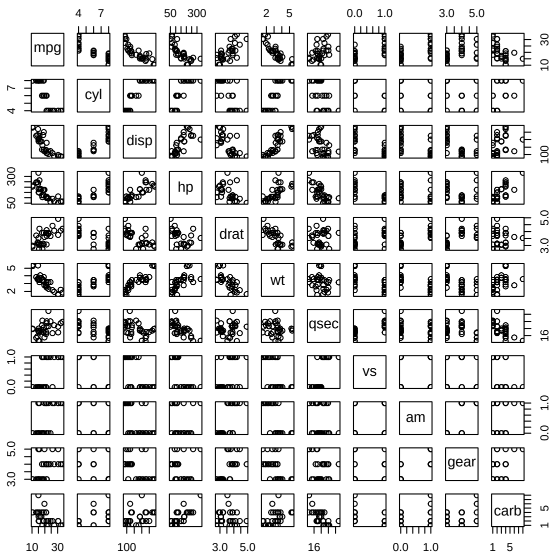


Figure 3.1: Plot for `mtcars` data set

The statement `plot(mtcars)` brings a graph, the one like the Figure 3.1, at right bottom pane of RStudio. Each variable is plotted against other variables. The column has names of the data set. I can produce the Figure 3.2 if I am interested in plotting scatter diagram for two of the variables in *mtcars* namely `mpg` and `hp`. My assumption is that *mileage (mgp)* depends on *horse power (hp)*.

From the Figure 3.2 it is clear that there seems to be inverse relationship between mileage and horse power. Let us try same assumption to difference variables i.e. *mileage (mpg)* and *weight (wt)*.

Scatter plot

Scatter plots can be made using `scatter.smooth()` function of *graphics* package. This time let me make two graphs in single canvas.

The statement `par(mfrow=c(1, 2))` is a parameter for graph device. `par()` can be used to set or query graphical parameters. We

```
> with(mtcars, plot(mtcars$mpg, mtcars$hp))
> abline(lm(mtcars$mpg ~ mtcars$hp, data = mtcars))
> lines(lowess(mtcars$mpg, mtcars$hp), col = "blue")
```

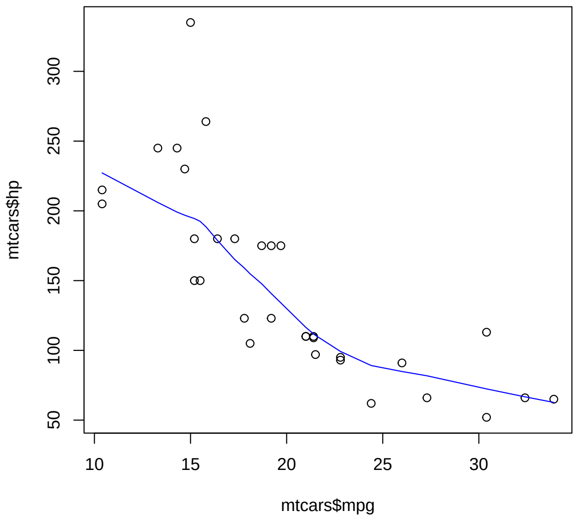


Figure 3.2: Plot for *mp* vs *hp* variables of `mtcars`

are instructing device to make two different plots in a single canvas. 1 represents row and 2 represents columns. Each device has its own set of graphical parameters. If the current device is the null device, `par` will open a new device before querying/setting parameters. Parameters are queried by giving one or more character vectors of parameter names to `par`. Several options are available for `par`. You can read by executing `help("par")`

Let us try another example using `airquality`.

Generally, the `plot()` function takes two vectors of numbers: one for the x-axis coordinates and one for the y-axis coordinates. However, `plot()` is what's called a generic function in R, which means its behavior can change depending on what kinds of data are passed to the function. We won't go into detail about that behavior for now. The remainder of this chapter will focus on the default behavior of the `plot()` function.

```
> par(mfrow = c(1, 2))
> scatter.smooth(mtcars$mpg ~ mtcars$wt)
> scatter.smooth(mtcars$mpg ~ mtcars$hp)
```

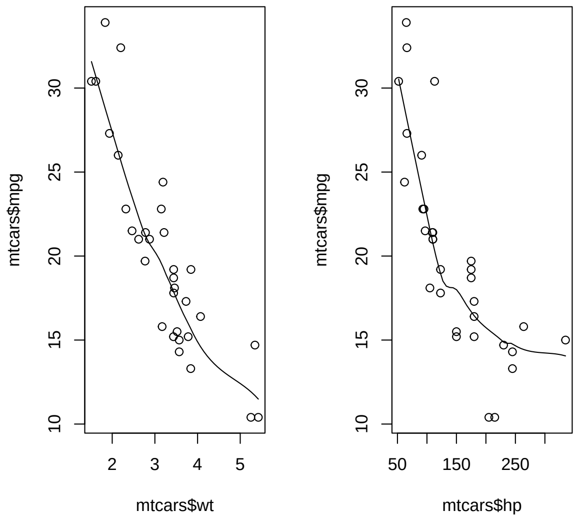


Figure 3.3: `scatter smooth` plot for *mileage* and *weight*

One thing to note here is that although we did not provide labels for the x- and the y-axis, labels were automatically created from the names of the variables (i.e. "Wind" and "Ozone"). This can be useful when you are making plots quickly, but it demands that you have useful descriptive names for the your variables and R objects.

3.2 Basic Plots

There are several methods to plot very basic visuals in R. In fact, R as language of statistics, offers endless opportunities in plotting data. However, use do need to know a precise strategy for plotting. Making visuals is not a burlesque task, visualization has a system behind it. Visuals or plots depend on the type of the data. When a researcher plan for data, he/she has a measurement strategy in mind. So, data does not appear just like that. Suppose, if the

```
> with(airquality, plot(Wind, Ozone))
> model <- lm(Ozone ~ Wind, airquality)
> abline(model, lwd = 2)
```

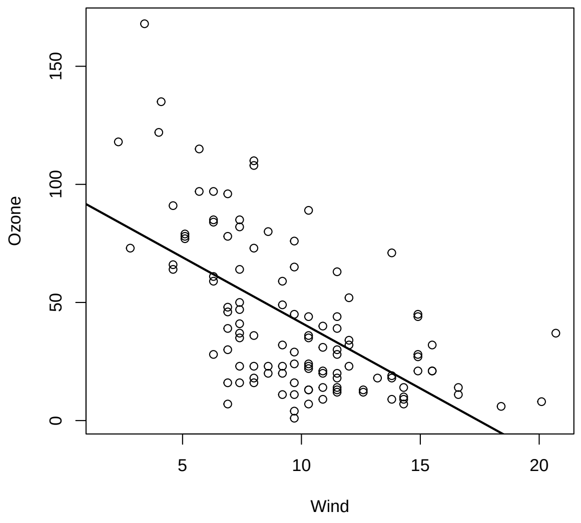


Figure 3.4: Scatter plot for `airquality` data set

research would like to collect gender information, where gender is going to be a valid data variable for analysis. Such variables need to be defined using a precise strategy and that pretty much depends on the needs of the analyst. In this case, since gender is dichotomous variable. It required to be measured using levels, such as *male* and *female*. So the variable gender turns out to be nominal in nature. Why I say nominal? If the analysis does not require gender to be analyzed using numerical calculations, then analyst may not require to convert responses i.e. *male* and *female*, otherwise, the analyst need to code this variable by assigning certain numeric to each of these responses. May be 1 for *male* and 2 *female*. In such case the data turns out to be numeric. However, the data is not ordinal. Why? Because, it is not possible to calculate how different a *male* respondent from *female* respondent such as with time, and distance. The following strategy might hold valid for obtaining visuals based on type of the data (measurement level).

Data	Plot
Nominal	Pie
Ordinal	Bar
Interval or Ratio	Histogram

Table 3.1: Type of Plots

3.2.1 Histogram

A histogram is an approximate representation of the distribution of numerical or categorical data. It was first introduced by Karl Pearson. A technique called “binning” is used to construct a histogram. It is all about dividing the entire range of values into a series of intervals and then count how many values fall into each interval. The bins are usually specified as consecutive, non-overlapping intervals of a variable. The bins (intervals) must be adjacent, and are often (but not required to be) of equal size.

Histograms give a rough sense of the density of the underlying distribution of the data, and often for density estimation: estimating the probability density function of the underlying variable. The total area of a histogram used for probability density is always normalized to 1. If the length of the intervals on the x-axis are all 1, then a histogram is identical to a relative frequency plot.

A histogram can be thought of as a simplistic kernel density estimation, which uses a kernel to smooth frequencies over the bins. This yields a smoother probability density function, which will in general more accurately reflect distribution of the underlying variable. The density estimate could be plotted as an alternative to the histogram, and is usually drawn as a curve rather than a set of boxes. Histograms are nevertheless preferred in applications, when their statistical properties need to be modeled.

Histograms are highly useful visuals for evaluating normality of data variables. Here is an example of a simple histogram made using the `hist()` function in the graphics package. Run this code and your graphics window is not already open, it should open once you call the `hist()` function.

Figure 3.5 shows that the data variable mileage (mpg) seems to be roughly normal.

```
> hist(mtcars$mpg, freq = FALSE)
> lines(density(mtcars$mpg), col = "red")
```

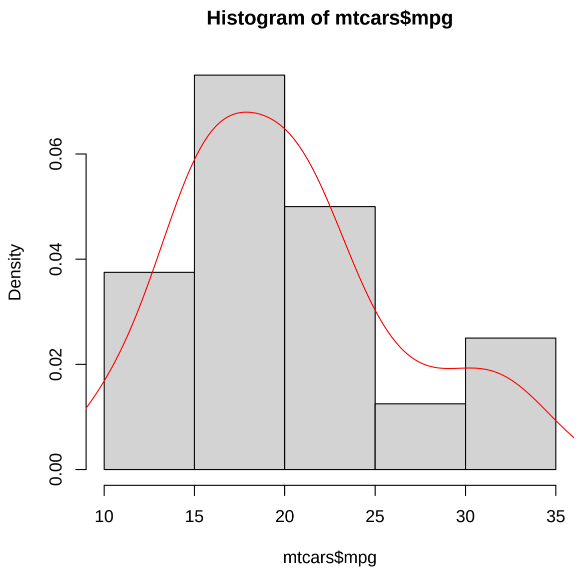


Figure 3.5: Histogram for *mpg* in `mtcars`

3.2.2 Box Plots

A box plot or boxplot is a method for graphically depicting groups of numerical data through their quartiles. Boxplots are useful in comparing categories of factorial data. Box plots may also have lines extending from the boxes, also known as whiskers, indicating variability outside the upper and lower quartiles, hence the term *box-and-whisker* plot is in vogue. Outliers, in boxplots, are plotted as individual points.

Box plots are non-parametric. They display variation in samples of a statistical population without making any assumptions of the underlying statistical distribution. The spacing between different parts of the box indicate the degree of dispersion (spread) and skewness in the data, and show outliers. Box plots can be drawn either horizontally or vertically. Box plots received their name from the box in the middle.

Boxplots can be made in R using the `boxplot()` function, which takes as its first argument a formula. The formula has form of $y \sim x$. Anytime you see a $\sim$ in R, it's a formula. Here, we are plotting ozone levels in New York by month, and the right hand side of the $\sim$ indicate the month variable. However, we first have to transform the month variable in to a factor before we can pass it to `boxplot()`, or else `boxplot()` will treat the month variable as continuous.

```
> airquality <- transform(airquality, Month = factor(Month))
> boxplot(airquality$Ozone ~ airquality$Month, airquality)
```

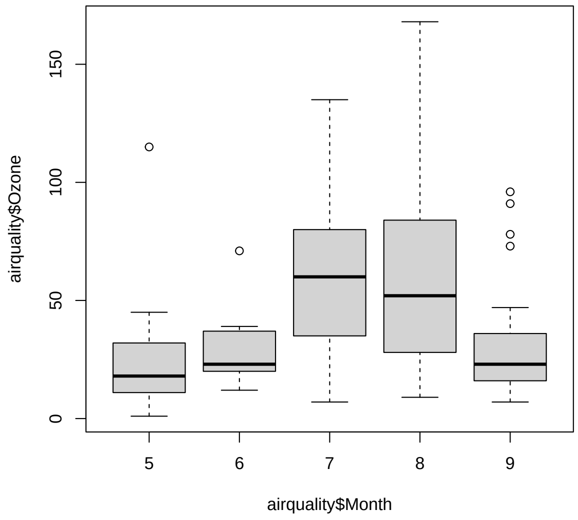


Figure 3.6: Boxplot for `airquality` data.

In Figure 3.6, each boxplot shows the median, 25th and 75th percentiles of the data (the "box"), as well as +/− 1.5 times the interquartile range (IQR) of the data (the "whiskers"). Any data points beyond 1.5 times the IQR of the data are indicated separately with circles.

In this case the monthly boxplots show some interesting features. First, the levels of ozone tend to be highest in July and August. Second, the variability of ozone is also highest in July and August. This phenomenon is common with environmental data where the mean and the variance are often related to each other.

From Figure 3.6 it is clear that there are outliers for months 5, 6, 8 and 9. Months 7 and 8 did not have any outliers for Ozone. This means the ozone levels are highly unsteady for all the months except 7th and 8th months. We need to look into data set to interpret more accurately if we can look into information. Use `help("airquality"` to understand measurement level for "Ozone".

3.2.3 Bar charts

A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. A vertical bar chart is sometimes called a column chart. [40]

Bar charts have a discrete domain of categories, and are usually scaled so that all the data can fit on the chart. When there is no natural ordering of the categories being compared, bars on the chart may be arranged in any order. Bar charts arranged from highest to lowest incidence are called Pareto charts.

Bar graphs/charts provide a visual presentation of categorical data. Categorical data is a grouping of data into discrete groups, such as months of the year, age group, shoe sizes, and animals. These categories are usually qualitative. In a column bar chart, the categories appear along the horizontal axis; the height of the bar corresponds to the value of each category.

Bar graphs can also be used for more complex comparisons of data with grouped bar charts and stacked bar charts. In a grouped bar chart, for each categorical group there are two or more bars. These bars are color coded to represent a particular grouping. The other variant of bar chart is stacked bar chart. The stacked bar chart stacks bars that represent different groups on top of each other. The height of the resulting bar shows the combined result of the groups. However, stacked bar charts are not suited to data sets where some groups have negative values. In such cases, grouped bar chart are preferable.

A bar graph shows comparisons among discrete categories. That is why they are highly suitable for ordinal data. One axis of the chart shows the specific categories being compared, and the other axis represents a measured value. Some bar graphs present bars clustered in groups of more than one, showing the values of more than one measured variable. There is a function `barplot()` in R which is useful for plotting bar plots. I shall show as how to plot bar charts using `mtcars` data set.

```
> counts <- table(mtcars$gear)
> barplot(counts, main = "Car Distribution", xlab = "Number of Gears")
```

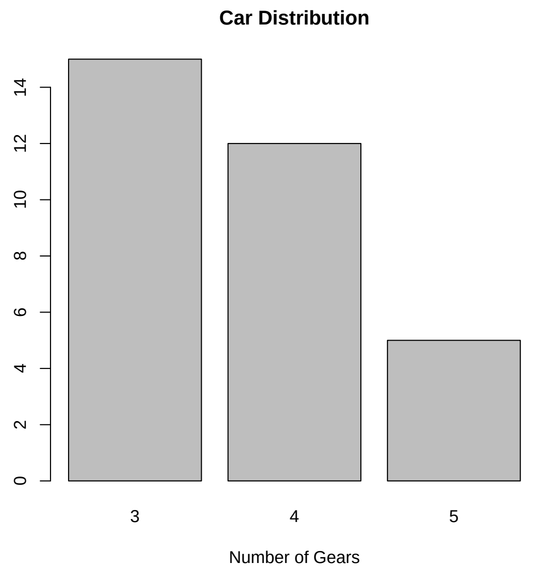


Figure 3.7: Bar chart for `mtcars` data set.

There are number of ways to plot bar charts, especially by arranging bars in the chart. There is certain argument `horiz` in `barplot()` function which defines the orientation of bars in the plot.

3.2.4 Pie charts

A pie chart, also known as circle chart, is a circular statistical graphic, which is divided into slices to illustrate numerical proportion. In a pie chart, the arc/sector length of each slice (and consequently its central angle and area), is proportional to the quantity it represents. While it is named for its resemblance to

```
> counts <- table(mtcars$vs, mtcars$gear)
> barplot(counts, main = "Car Distribution by Gears and VS", xlab = "Number of Gears",
+     col = c("blue", "pink"), legend = rownames(counts))
```

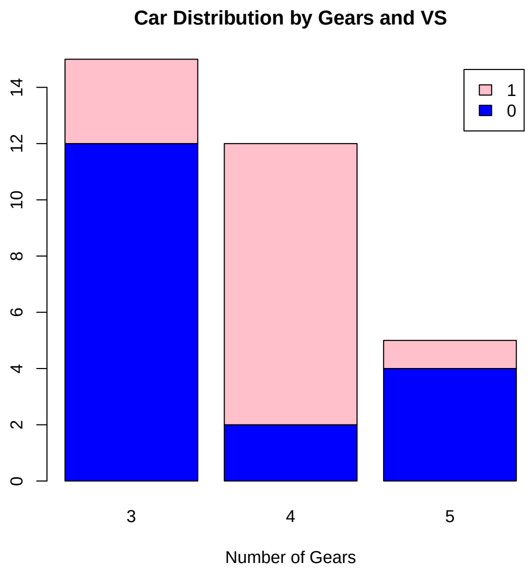


Figure 3.8: Bar chart for `mtcars` data set.

a pie which has been sliced, there are variations on the way it can be presented. The earliest known pie chart is generally credited to William Playfair's Statistical Breviary of 1801.

Pie charts are very widely used in the business world and the mass media in spite of criticism and disagreement in user community. Pie charts can be replaced in most cases by other plots such as the bar chart, box plot, dot plot, etc.

In spite of criticism, pie charts are still offers best ways for displaying information related to nominal data. Some extent, pie charts can be extensible to ordinal data. Let me try pie chart using BOD data set, because this data set has 6 rows and 2 columns. That makes it very simple for pie chart.

We can stretch this pie chart further by adding labels for each of the pies.

All the techniques or methods explained in this chapter deals with

```
> pie(BOD$demand, labels = BOD$Time)
```

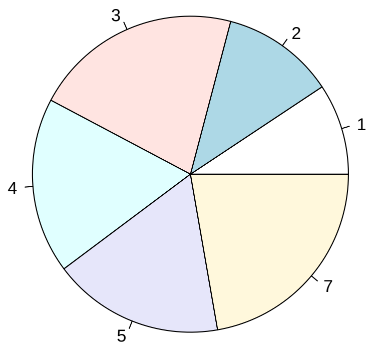


Figure 3.9: Pie chart for `BOD` data set.

only few basic ways of plotting data. R offers highly extensible environment for various ways to plot data. Few basic plots like line diagrams, contours, perspective maps were not described in this text. Scope of this book does not permit all such visuals. R has ways to plot data as by the technique in hand. This means it is possible perform visual investigation whenever analysis is done on certain data. For instance, R offers very nice ways to obtain visuals for regression. Regression is explained in bivariate analysis in this book. So there will be appropriate description to these methods related to obtaining visuals/plots while performing regression analysis.

```
> bodlabels <- round(BOD$demand/sum(BOD$demand) * 100, 1)
> bodlabels <- paste(bodlabels, "%", "")
> pie(BOD$demand, labels = bodlabels)
```

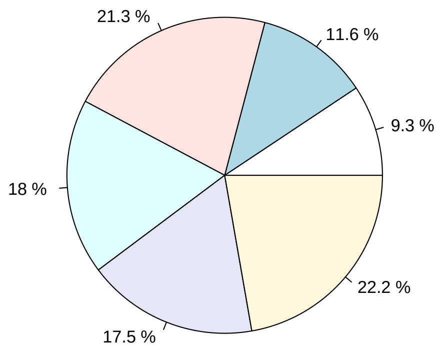


Figure 3.10: Pie chart for `BOD` data set.

Notes

[36]Frank McCown, Producing Simple Graphs with R. Available from https://sites.harding.edu/fmccown/r/

[37]CRAN Task View: gRaphical Models in R. Available at gRaphicalModelsinR.

[38]Hadley Wickham, ggplot2: Create Elegant Data Visualizations Using the Grammar of Graphics. Available at https://cran.r-project.org/web/packages/ggplot2/index.html.

[39]Yihui Xie, et al., animation: A Gallery of Animations in Statistics and Utilities to Create Animations. Available at https://cran.r-project.org/web/packages/animation/index.html

[40]Bar charts. Available from https://en.wikipedia.org/wiki/Bar_chart

3.3 Exercises

1. What is Graph Device? How do you call or off a new or active device respectively. Practice using R statements.
2. Study about R base function `plot()`. Practice various parameters or user arguments of this function in practice.
3. Study about suitable plots for data by its measurement levels.
4. Practice few plotting techniques like contour maps, perspective maps using R base functions.
5. Study about normality plot. Practice as how to shade area under normal curve.
6. Identify few packages available for advanced visualization such as 3D plots.
7. Learn few techniques related to animations of graphs using different types of data.
8. Find out few packages from CRAN which are meant for advanced visualization techniques related to computer vision.
9. Identify few packages which are useful for image manipulation and editing.

3.4 Case

Looking back to the start of the year 2019, we could never have predicted what would befall our world as a result of COVID 19. Back then, the coronavirus was spreading in China, and while there were warnings of its potential to escalate across the world, few could imagine the tremendous shift it would bring to the status quo. Today, we find ourselves living in a new normal. Working from home has become standard, the global economy is uncertain, hospitals are working harder than ever, and the world is waiting for a vaccine to provide reassurance that we can stop social distancing.

Different firms, say Legz.io, at large has came up with variegated solutions to address information crunch regarding COVID 19. Few firms, even started offering their services in open source style. A company called Elastic offers a viable solution for creating dashboards using open source tech stack, they call it Elastic stack. Taking cues from time to time changes of disease and pandemic, WHO has came up with a unique dashboard for providing latest updates on COVID 19.

What is a dashboard? Study about different ways to create dashboards. Are these solutions, offered by tech firms truly open source? Is there any commercial interest behind such efforts? Investigate.

Chapter 4

Univariate Analysis

"If your experiment needs a statistician, you need a better experiment."

- Ernest Rutherford

4.1 Introduction to statistics

Statistics has no positive fame in academics as well as in industry. There is tons of information related to hatred on statistics online. There are also people who love statistics. Ironically hatred travels faster than love. Whatever, world might think, but to me it is not possible to discuss statistics in one go. It is continuous learning process. Most of the time people dislike statistics because Statistics is not strictly mathematical in nature. Mathematics is the mother of all sciences. Coming to statistics, it is not strictly science and there is little pseudoscience in it. Just a joke! May be my being stated as such is due to the way I experience events around me when I am at work. One potential reason for this restlessness might be due to lack of very strict scientific implementations in Statistics. Science is respected by its practicality. Science is a fiction as long as it is in theory and untested. At lest to me!

This Chapter deals with few very familiar statistics. I shall discuss basics such as *univariate, bivariate* and *multivariate* statistics in this text. There is no comprehensive method to classify statistics.

Methods are abundant. For instance, as it was mentioned earlier, statistics can be classified as below:

1. Univariate statistics
2. Bivariate statistics
3. Multivariate statistics

and this classification is based on variables. *Univariate* analysis deals with only one variable and methods associated with this are: (1) summaries, (2) descriptives, (3) statistical tests (such as normality tests) and many more. *Bivariate* analysis deals with statistics which attempts to explain relationships or dependencies between two variables. These techniques might include (1) cross tabulations, (2) correlation, (3) regression. *Multivariate* analysis deals with those statistical techniques which analyze data sets with more than two variables. There is another classification based on type of analyses that is being done on data sets.

1. Descriptive analytics
2. Predictive analytics
3. Prescriptive analytics

This classification consider type of analysis as criteria. *Descriptive analytics* is just like the summary category in the above classification. Descriptive statistics deals with measures that explains location (mean) and scale (dispersion). Briefly descriptive statistics try to cover three types of measures related to *center, dispersion* and *shape*. Coming to *predictive analytics*, it deals with all bivariate methods related to regression (mostly). Many experts mention regression as "hydrogen bomb" in the arsenal of statistics. Coming to *prescriptive analytics*, it deals with much of mathematics and less statistics. This phrase is relatively a new one used in data analytics community. Most of the numerical optimization is covered under prescriptive analytics. The third classification I would like to highlight is as follows:

1. Summary statistics
2. Inferential statistics
3. Exploratory statistics

Few texts follows the above classification. Again, *summary* statistics is just like as descriptive statistics but not exactly. Summaries deals with aggregations. Aggregation literally means *any process which information is explained by summaries.* Data is processed for few summary statistics such as *sum, maximum, minimum, range* and etc. It is all confusing right! that is why you don't get people who like statistics all the time. While coming to *inferential* statistics, they are pretty much same like *predictive in second classification.* Inference literally means: *"a conclusion reached on the basis of evidence and reasoning."* These statistics most of the time found to be strictly connected with a very high and grand technical word known as *"hypothesis"*. If at all anybody refer statistics to science, that is because of this *hypothesis* testing. There is also lot of criticism on testing hypotheses in the area of Statistics. [41] Now it is time to discuss about methods. Please be careful and cautious of your brain. Absolutely no guarantee or warranty what so ever. Reader beware!

4.2 Summary statistics

Summary statistics are used to summarize a set of observations, in order to communicate the largest amount of information as simply as possible. Statisticians commonly try to describe the observations in

1. a measure of central tendency
2. a measure of dispersion
3. a measure of the shape

A common collection of order statistics used as summary statistics are the *five number summary*, sometimes extended to a *seven number summary*, and the associated box plot. Entries in an analysis of variance table can also be regarded as summary statistics.

There are two very important functions which helps a lot while computing summaries. They are (1) with and (2) by. `with` is a function that evaluates an R expression in an environment constructed from data, possibly modifying (a copy of) the original data. The syntax for this function is as follows:

```
with(data, expr, ...)
```

`with` is a generic function that evaluates *expr* in a local environment constructed from data. The other one is `by` and syntax for this function is as follows:

```
    by(data, INDICES, FUN, ..., simplify = TRUE)
```

`by` require *indices* to execute *FUN*. We shall use these two functions to obtain summary statistics. Let us take data set we used in Chapter 1 named `markdf`. We might be able to obtain summaries as shown below:

```
> with(markdf, mean(markdf$salary))

[1] 55

> with(markdf, {
+     cbind(mean(markdf$salary), sd(markdf$salary))
+ })

     [,1]     [,2]
[1,]   55 26.24421
```

Suppose we wish to know all other measures in detail. We might be able to use a special but highly generic function `summary()`

```
> summary(markdf$gender)

   Length     Class      Mode
       30 character character

> summary(markdf$age)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  19.00   25.25   42.00   40.60   55.00   59.00

> summary(markdf$salary)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  12.00   35.00   48.50   55.00   80.75   95.00
```

We are missing one measure out of those seven that is *mode.* The mode is a frequently occurring observation in the given data distribution. As such *Max.* can serve as *Mode* but is wrong. Mode requires categories. It does not make sense in computing mode for salary. However, I can convert salary into an interval data distribution, and then compute mode. For instance, I might be able to compute Mode for *gender* in `markdf` data set as in following manner.

```
> with(markdf, max(table(markdf$gender)))

[1] 17
```

Both *age* and *salary* need to be converted to categorical data before computing mode. And the way to do it is using *if()* statement. Let us see how to put `if()` statement into action.

```
> markdf$salary2 <- with(markdf, ifelse(salary >= 0 & salary <
+     20, "0 to 20", ifelse(salary >= 20 & salary < 40, "20 to 40",
+     ifelse(salary >= 40 & salary < 60, "40 to 60", ifelse(salary >=
+         60 & salary < 80, "60 to 80", "80 to 100")))))
```

And the additional variable `salary2` will appear as below:

```
> names(markdf)
```

```
 [1] "gender"         "employment"     "marital.status" "salary"
 [5] "age"            "sat1"           "sat2"           "sat3"
 [9] "per1"           "per2"           "per3"           "att1"
[13] "att2"           "att3"           "salary2"
```

```
> markdf$salary2[1:5]
```

```
[1] "20 to 40"  "80 to 100" "40 to 60"  "20 to 40"  "60 to 80"
```

Applying `ifelse()` every time whenever I need above conversion may give pain in the brain. I may require this operations wherever continuous data arises in analysis. In which case, it is better to create a UDF for such operation.

```
> makeCats <- function(dat) {
+     max <- max(dat)
+     min <- min(dat)
+     q1 <- quantile(dat, 0.25)
+     q2 <- quantile(dat, 0.5)
+     q3 <- quantile(dat, 0.75)
+     cats <- ifelse(dat > min & dat <= q1, 1, ifelse(dat < q1 &
+         dat <= q2, 2, ifelse(dat < q2 & dat <= q3, 3, ifelse(dat <
+         q3 & dat <= max, 4, 5))))
+     return(cats)
+ }
> makeCats(markdf$salary)
```

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

There is another way to compute summaries using *quantiles.* There is one function known as `quantile()` in R to take care of all quantiles such as quartiles, deciles and percentiles etc. First we need to make quartiles. Later these quartiles can be used to make a new variable possibly `age1` in same data set.

```
> min(markdf$age)
```

```
[1] 19
```

```
> quantile(markdf$age, 0.25)
```

```
  25%
25.25
```

```
> quantile(markdf$age, 0.5)

50%
 42

> quantile(markdf$age, 0.75)

75%
 55

> max(markdf$age)

[1] 59
```

The above chunk helps in calculating four measures such as $min, < 25, < 75, < 100, max$, just like those seen in summaries. Median is known as 50^{th} quantile.

```
> quantile(markdf$age, 0.5)

50%
 42

> median(markdf$age)

[1] 42
```

4.3 Apply functions

R has few other methods to try summaries using peculiar functions such as `apply, tapply, sapply`, and `lapply`. These functions are simply referred to *apply* functions.

The `apply()` collection is bundled with R Base. These functions are available immediately after installing R. The `apply()` function can be feed with many functions to perform redundant application on a collection of object (data frame, list, vector, etc.). The purpose of `apply()` is primarily to avoid explicit uses of loop constructs. They can be used for an input list, matrix or array and apply a function. Any function can be passed into `apply()`

Suppose we want to find mean and that need to be done for all variables in the data set. The only option in R is to make use of `if()` statement. Such things might be palatable for programmers, but most of the R users are statisticians and they are not into much of programming activity. This problem can be simply achieved using `apply()` function without using `if()` statement.

```
> names(markdf)

 [1] "gender"         "employment"     "marital.status" "salary"
 [5] "age"            "sat1"           "sat2"           "sat3"
 [9] "per1"           "per2"           "per3"           "att1"
[13] "att2"           "att3"           "salary2"

> apply(markdf[, 6:14], 2, median)

sat1 sat2 sat3 per1 per2 per3 att1 att2 att3
 3.0  3.0  3.0  3.5  2.5  4.0  4.0  3.5  4.0
```

Above chunk calculates *median* for all those variables that are numerical. Please notice we indexed data i.e. only those variables that are ordinal and also study variables.

Next function is `tapply()`. This function is highly useful whenever we wish to make calculations across the table. Suppose we intent to see *mean* salary for gender. Please notice *salary* is not categorical whereas *gender* is categorical. Earlier we converted non-categorical variables into categorical to make calculations. Actually we can escape all that work by using `tapply()`.

```
> tapply(markdf$salary, markdf$gender, mean)

  female     male
58.35294 50.61538
```

Next functions are `sapply` and `lapply`. In fact, these two functions does same activity but the difference is only in the way they output results. The `sapply()` output result as in array, whereas `tapply()` output results as in list. Let us try for study variables in the data set *markdf*.

```
> sapply(markdf[, 6:14], median)

sat1 sat2 sat3 per1 per2 per3 att1 att2 att3
 3.0  3.0  3.0  3.5  2.5  4.0  4.0  3.5  4.0

> lapply(markdf[, 6:14], median)[1:3]

$sat1
[1] 3

$sat2
[1] 3

$sat3
[1] 3
```

These *apply* functions are highly useful for those whose primary goal is data analysis. Programmers may not like it because, they may be interested in using raw commands just as conditional statements coupled with loops. For instance, if I am interested in average salary for gender levels, I might be able to define a function such as

```
> groupMeans <- function(x, y) {
+     gpdata <- data.frame(x, y)
+     l <- levels(gpdata$x)
+     gs <- list()
+     for (i in 1:length(l)) {
+         gs[i] <- mean(gpdata$y[gpdata$x == l[i]])
+     }
+     return(t(matrix(gs, dimnames = list(l))))
+ }
> groupMeans(markdf$gender, markdf$salary)
```

```
     [,1]
[1,] NaN
```

```
> tapply(markdf$salary, markdf$gender, mean)
```

```
  female     male
58.35294 50.61538
```

The above program computes mean salary for gender groups. The results were verified with `tapply()` function. You see! That is the advantage of known programming. The question is why? For pretty calculations it is okay to use base functions. What if it requires to do gigantic calculations, for which it is not possible to find a predefined method of function. Everyone, one day or the after should turn into programming. It is just a matter of experience. So the function `apply()` is just a base function that somebody written for users. This way, it is possible to rewrite remaining functions such as `apply`, `sapply` and `lapply` using user logic.

4.4 Frequency & Cross tables

The other aspect of summary statistics is frequency tables. Frequency tables are summary tables for categorical data. There is one very simple function called as `table()` to make summary tables in R. We can try making a table *gender*.

```
> tab <- table(markdf$gender, markdf$salary2)
> tab

        0 to 20 20 to 40 40 to 60 60 to 80 80 to 100
 female       2        4        3        2         6
 male         1        4        4        1         3

> margin.table(tab, 1)

female   male
   17     13
```

The frequencies for bivariate data is called a cross table. For instance, the table of frequencies for *gender* defined by *salary2* is a bivariate frequency table also known as cross table. `margin.table()` calculates sum of all salary-wise categories against *gender*. 16 is the total of all the category wise values for salary4. The opposite scenario i.e. sums of *gender* against *salary* can be done by using following statement.

```
> margin.table(tab, 2)

 0 to 20  20 to 40  40 to 60  60 to 80 80 to 100
       3         8         7         3         9
```

`margin.table()` is useful for calculating sums. But in case if we wish to calculate percentages we might use `prop.table()`.

```
> prop.table(tab)

            0 to 20   20 to 40   40 to 60   60 to 80  80 to 100
 female 0.06666667 0.13333333 0.10000000 0.06666667 0.20000000
 male   0.03333333 0.13333333 0.13333333 0.03333333 0.10000000

> prop.table(tab, 1)

            0 to 20   20 to 40   40 to 60   60 to 80  80 to 100
 female 0.11764706 0.23529412 0.17647059 0.11764706 0.35294118
 male   0.07692308 0.30769231 0.30769231 0.07692308 0.23076923

> prop.table(tab, 2)

          0 to 20  20 to 40  40 to 60  60 to 80 80 to 100
 female 0.6666667 0.5000000 0.4285714 0.6666667 0.6666667
 male   0.3333333 0.5000000 0.5714286 0.3333333 0.3333333
```

The first table is cell percentages, which means frequencies of the cross table is divided by grand total. The second table shows proportions by row totals. Third table shows proportions by column totals. There is another function known as `ftable()`. This function gives rather more information adding little decoration to the frequency table. Suppose we are interested in more than two variables, then the `table()` function may not be a best option for it gives murky output. `ftable()` gives rather attractive tables.

```
> ftable(markdf$gender, markdf$salary2, markdf$employment)

                  low middle top

female 0 to 20      0      1   1
       20 to 40     3      1   0
       40 to 60     2      0   1
       60 to 80     1      1   0
       80 to 100    4      2   0
male   0 to 20      0      1   0
       20 to 40     1      2   1
       40 to 60     2      2   0
       60 to 80     0      1   0
       80 to 100    0      2   1
```

There is even other function known as `xtabs()` but this function does nothing more compared to `table()`.

4.4.1 Tests of independence

For 2 way tables you can use *chisq.test()* to test independence of the row and column variable. By default, the *p-value* is calculated from the asymptotic *chisquared* distribution of the test statistic. Suppose if we assume that salary depends on gender, which means there are inter-gender level differences to salary. This assumption can serve as null hypothesis for our test. This hypothesis can be tested using chi-squared test. Usually for chi-squared test the null hypothesis is *the variables under study are independent.*

```
> tab

          0 to 20 20 to 40 40 to 60 60 to 80 80 to 100
  female        2        4        3        2         6
  male          1        4        4        1         3

> chisq.test(tab)

        Pearson's Chi-squared test

data:  tab
X-squared = 1.2993, df = 4, p-value = 0.8615
```

```
> fisher.test(tab)

        Fisher's Exact Test for Count Data

data:  tab
p-value = 0.9026
alternative hypothesis: two.sided
```

Unfortunately there isn't evidence in support of our hypothesis. The P Value is 0.8035 this is far beyond compared to threshold value 0.05. So, we fail to reject null hypothesis and the assumption is not true. Hence we infer that the *salary* is independent of *gender*. The gender discrimination is not found in the data.

What do we need to do for 3 way tables? There is a function `mantelhaen.test()`. this function is useful to find inter-categorical differences to another categorical variable provided the third dimension has levels of strata. Suppose if we wish to find *gender* level differences to *salary* and *employment* serves as stratum. We can perform contingency test as shown below.

```
> A <- markdf$gender
> B <- markdf$salary2
> C <- markdf$employment
> mantelhaen.test(A, B, C)

        Cochran-Mantel-Haenszel test

data:  A and B and C
Cochran-Mantel-Haenszel M^2 = 2.8892, df = 4, p-value = 0.5765
```

I converted variables into A, B and C, where A stands for *gender*, B stands for *salary2* and C stands for *employment*. The function `mantelhaen.test()` is useful to perform a *Cochran-Mantel-Haenszel chi-squared test* of the null hypothesis that two nominal variables are conditionally independent in each stratum, assuming that there is no three-way interaction. This function requires 3 dimensional data, where the last dimension refers to the strata. So, now looking at the P Value it is possible for us to infer that the conditional independence does not exit in data.

4.4.2 Mutual Independence

A log-linear model is a mathematical model that takes the form of a function whose logarithm equals a linear combination of the parameters of the model, which makes it possible to apply (possibly multivariate) linear regression. That is, it has the general form. Suppose if we imagine that *employment*, *gender* and *salary* exhibit pair wise dependence (H_0). We may be able to test using `loglm()` from *MASS* library.

```
> xtab <- xtabs(~A + B + C, markdf)
> library(MASS)
> loglm(~A + B + C, xtab)

Call:
loglm(formula = ~A + B + C, data = xtab)

Statistics:
                      X^2 df P(> X^2)
Likelihood Ratio 21.72839 22 0.476197
Pearson          16.26060 22 0.802745
```

Both *Likelihood Ratio* and *Pearson r* are statistically insignificant (P Value > 0.05). So we fail to reject H_o that *employment, gender* and *salary* are independent as by pairs. So there is no evidence in support of *mutual dependence.*

4.4.3 Partial Independence

Partial Independence assumes that A is partially independent of B and C (i.e., A is independent of the composite variable BC). Let us assume that *salary* is independent of *gender* and *marital status.*

```
> llt <- loglin(xtab, list(c(1, 2), c(2, 3), c(3, 1)))

5 iterations: deviation 0.04989516

> 1 - pchisq(llt$lrt, llt$df)

[1] 0.3886252
```

The P Value is greater than 0.05, so we fail to reject null hypothesis. *gender, employment* and *salary* are independent as by their respective margins.

4.4.4 Conditional independence

Conditional independence is just like A is independent of B, given C. As usual, A stands for *gender*, B stands for *salary2* and C stands for *employment.* Suppose if we assume that *salary* is independent of *gender* that is because of *employment.*

```
> loglm(~A + B + C + B * A + C * A, xtab)

Call:
loglm(formula = ~A + B + C + B * A + C * A, data = xtab)

Statistics:
                      X^2 df  P(> X^2)
Likelihood Ratio 16.2808 16 0.4335409
Pearson          12.9500 16 0.6764101
```

So we don't have evidence against the hypothesis that *salary* depends on *gender* and *employment.* This gives us a rough idea that there may not be any interaction at all in our data. Because there is no any type of dependence as far as *gender, employment* and *salary* are concerned. However, our suspicion must be tested using *3 way independence test.*

4.4.5 3 way independence

In three way independence all the three variables assumed to be independence from each other. In this case, the study variables gender (A), salary (B), and employment (C) are independence from each other.

```
> loglm(~A + B + C + B * C + B * A + C * A, xtab)

Call:
loglm(formula = ~A + B + C + B * C + B * A + C * A, data = xtab)
Statistics:
                      X^2 df  P(> X^2)
Likelihood Ratio 8.479293  8 0.3881039
Pearson               NaN  8       NaN
```

The P Value is > 0.05. So now there is no evidence in support of the assumption that these variables are totally independent from each other.

4.5 Statistical diagnosis

Parametric statistics is a branch of statistics which assumes that sample data come from a population that can be adequately modeled by a probability distribution that has a *fixed* set of parameters. Conversely a non-parametric model differs precisely in that the parameter set is not fixed and can increase, or even decrease, if new relevant information is collected. The normal family of distributions all have the same general shape and are parameterized by mean and standard deviation. That means that if the mean and standard deviation are known and if the distribution is normal, the probability of any future observation lying in a given range is known.

The very base for these parametric tests is *probability distributions*. A probability distribution describes how the values of a random variable is distributed. For example, the collection of all possible outcomes of a sequence of coin tossing is known to follow the binomial distribution. Whereas the means of sufficiently large samples of a data population are known to resemble the normal distribution. Since the characteristics of these theoretical distributions are well understood, they can be used to make statistical inferences on the entire data population as a whole. Few important distributions are as follows:

1. Binomial Distribution
2. Poisson Distribution
3. Continuous Uniform Distribution
4. Exponential Distribution
5. Normal Distribution
6. Chi-squared Distribution
7. Student t Distribution
8. F Distribution

Many of the distributions like *binomial, Poisson, uniform, exponential* etc. are situational in nature.

4.5.1 Parametric tests

Parametric tests are a branch of statistics which assumes that sample data come from a population that can be adequately modeled by a probability distribution that has a fixed set of parameters. [42] Conversely a non-parametric model differs precisely in that the parameter set is not fixed and can increase, or even decrease, if new relevant information is collected.

R has functions to deal with all types of probability distributions. Suppose if I ask a question that is there 50-50 chance for gender if we randomly pick them from 30. The success is 0.5.

```
> dbinom(1, 2, 1/length(levels(markdf$gender)))
```

```
[1] NaN
```

Suppose if we ask a question what is the probability of getting a person whose salary is less than 50 (which means response 2) out of 30 individuals? This is a Poisson distribution. I know that the probability of success is roughly 0.2 (1/5).

```
> 1/5
```

```
[1] 0.2
```

```
> q2 <- quantile(markdf$salary, 0.25)
> dpois(2, mean(makeCats(markdf$salary)))
```

```
[1] 0.2113045
```

The answer is 21 % likely. Coming to *normal distribution*. Suppose if we wish to test the probability of getting a random individual whose salary is greater than 75 thousand or less from a sample of 30 given their mean and standard deviation.

```
> dnorm(75, mean(markdf$salary), sd(markdf$salary))
```

```
[1] 0.01137018
```

So it is 1 % likely. Try using minimum and maximum values. You might find that the likeliness increases with minimum salary, and likeliness decreases with maximum salary.

We can also model our satisfaction variable using *exponential* distribution. If we assume that increase or decrease of satisfaction behaves like exponential distribution. Suppose if the mean satisfaction of our survey respondents is known.

```
> satisfaction <- apply(markdf[, 6:8], 1, mean)
> dexp(mean(satisfaction))
```

```
[1] 0.05502322
```

It is 5 % likely that a randomly chosen individual might be identified as less than indifferent category. This means the probability of individual whose response is less than 3 on scale of 1 to 5 is just 5 percent.

4.5.2 T Test

The T test is statistical hypothesis test in which the test statistic follows a Student's t-distribution under the null hypothesis. A t-test is most commonly applied when the test statistic would follow a normal distribution if the value of a scaling term in the test statistic were known. When the scaling term is unknown and is replaced by an estimate based on the data, the test statistics (under certain conditions) follow a Student's t distribution. The t-test can be used, for example, to determine if the means of two sets of data are significantly different from each other.

Suppose if I wish to compute one sample t-test for *salary.* I choose $H_0 = 0$.

```
> t.test(markdf$salary, mu = 0)

        One Sample t-test

data:  markdf$salary
t = 11.479, df = 29, p-value = 2.645e-12
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 45.20025 64.79975
sample estimates:
mean of x
       55
```

Oh! The P value is significant. So there is evidence in support of H_a and there by H_0 can be rejected. The mean value of salary is significantly different from zero (μ). The CI is the shock absorber for mean value.

4.6 Univariate normality

Normality tests are used to determine if a data set is well modeled by a normal distribution and to compute how likely it is for a random variable underlying the data set to be normally distributed. An informal approach to testing normality is to compare a histogram of the sample data to a normal probability curve. The empirical distribution of the data (the histogram) should be bell shaped and resemble the normal distribution. This might be difficult to see if the sample is small.

A graphical tool for assessing normality is the normal probability plot, a quantile quantile plot (QQ plot) of the standardized data against the standard normal distribution. Here the correlation between the sample data and normal quantiles (a measure of the goodness of fit) measures how well the data are modeled by a normal distribution. For normal data the points plotted in the QQ plot should fall approximately on a straight line, indicating high positive correlation. These plots are easy to interpret and also have the benefit that outliers are easily identified. QQPlot for *age* distribution in *markdf* data set. [43]

```
> qqnorm(markdf$age)
> qqline(markdf$age)
```

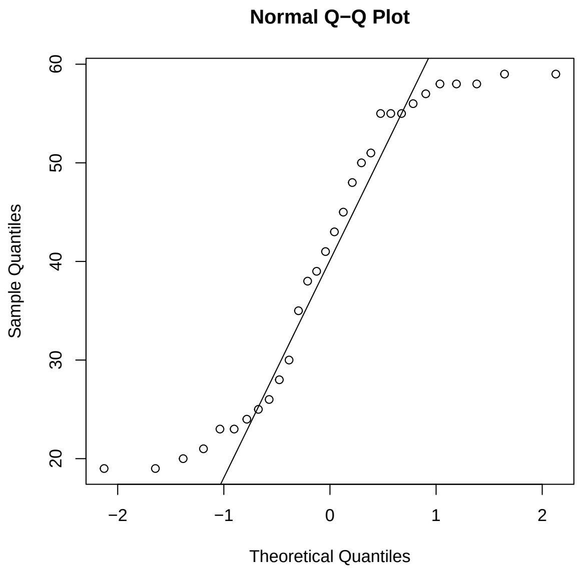


Figure 4.1: QQ Plot for *marketing data set* (`markdf`)

Look at Figure 4.1 it is strictly not normal. However, this might be naive way of coming to conclusion that the data distribution *age* is non-normal just by looking at visuals. The departure what-so-ever we are perceiving might be due to *sampling error*. This may not be true to in population. If someone ask this type of questions we need to show evidence. And to show evidence we need to perform some statistical test. In R there are quite a few statistical tests for univariate data. Following is the list of very few of such tests:

1. Shapiro Wilk test
2. Anderson Darling test
3. Cramer-von-Mises test
4. Shapiro Francia test
5. Jarque Bera test
6. Kolmogorov Smirnov test
7. Lilliefors test
8. Pearson χ^2 test

Each of which is special in it's own way. Few of them depends on mean and others on different other measures. Cramer, Lilliefors, Cramer etc. are basically variance based tests. There is another test which is not listed in this are *Bartlett test* which tests a particular aspect called *spherecity* and it depends on variance. For this section let us understand two tests namely *Kolmogorov Smirnov Test* and *Shapiro Wilk Test*. These two tests are highly robust and also simple to understand. Now let us gather evidence in support of earlier understanding related to *age*.

```
> shapiro.test(markdf$age)

        Shapiro-Wilk normality test

data:  markdf$age
W = 0.87647, p-value = 0.002341

> ks.test(markdf$age, "pnorm")

        Asymptotic one-sample Kolmogorov-Smirnov test

data:  markdf$age
D = 1, p-value < 2.2e-16
alternative hypothesis: two-sided
```

The H_0 is that the variable under study is normally distributed. Both tests confirms that the *age* is not normally distributed. We have sufficient evidence to show in support of H_a (P value < 0.05).

4.6.1 Non-parametric tests

Non-parametric statistics is the branch of statistics that is not based solely on parameterized families of probability distributions (common examples of parameters are the mean and variance). Non-parametric statistics is based on either being distribution-free or having a specified distribution but with the distribution's parameters unspecified. Non-parametric statistics includes both descriptive statistics and statistical inference.

Non-parametric (or distribution-free) inferential statistical methods are mathematical procedures for statistical hypothesis testing which, unlike parametric statistics, make no assumptions about the probability distributions of the variables being assessed.

Sign test & Prop Test

A Sign Test is used to decide whether a binomial distribution has the equal chance of success and failure. Performs an exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment. The statistical test used for Sign Test is *binomial test*. By definition, the binomial test is an exact test of the statistical significance of deviations from a theoretically expected distribution of observations into two categories. One common use of the binomial test is in the case where the null hypothesis is that two categories are equally likely to occur.

We have different proportions for *gender* in our study data. Suppose if we suspect gender bias that can be tested using `binom.test()` in R. There is also another test known as `prop.test` if there are more number of proportions in the test data. Know more about this test by executing `help("binom.test")` in R Console. In *markdf* data there is one dichotomous variable i.e. *gender*. It might be possible to use *binomial test* to know if gender levels are equally likely or not.

```
> length(which(markdf$gender == "male"))

[1] 13

> length(which(markdf$gender == "female"))

[1] 17

> 17 - 13/30

[1] 16.56667

> nm <- length(which(markdf$gender == "male"))
> binom.test(nm, 30)
```

```
        Exact binomial test
data:  nm and 30
number of successes = 13, number of trials = 30, p-value = 0.5847
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.2546075 0.6257265
sample estimates:
probability of success
            0.4333333
```

There is almost 16% of difference between gender levels but that difference is not significant in the data (for P value is > 0.05). Same way we can also make test for polytomous data. Such as *employment, marital status, satisfaction* etc. but we need to use *prop.test()* in stead of `binom.test`. Suppose if I wish to find out level wise differences for employment status I might be able to do as shown below:

```
> nm1 <- length(which(markdf$employment == "top"))
> nm2 <- length(which(markdf$employment == "middle"))
> nm3 <- length(which(markdf$employment == "low"))
> c(nm1, nm2, nm3)
```

```
[1]  4 13 13
```

```
> pttop <- prop.test(nm1, 30)
> ptmid <- prop.test(nm2, 30)
> ptlow <- prop.test(nm3, 30)
> c(pttop$p.value, ptmid$p.value, ptlow$p.value)
```

```
[1] 0.0001260465 0.5838824208 0.5838824208
```

The very first level i.e. *top* level employees are significant in the data. This means that the probability (proportion) that a randomly picked individual belongs to top level is close to zero. This seems to be true if you notice frequencies for employment levels.

Notes

[41] Gerd Gigerenzer, Stefan Krauss, and Oliver Vitouch wrote a popular article with a name “The Null Ritual What You Always Wanted to Know About Significance Testing but Were Afraid to Ask”; this will help demystify few doubts about hypothesis testing. Available at https://library.mpib-berlin.mpg.de/ft/gg/GG_Null_2004.pdf

[42] Geisser, S. (2006), Modes of Parametric Statistical Inference, John Wiley Sons

[43] Clay Ford, Understanding Q-Q Plots. Retrieved from https://data.library.virginia.edu/understanding-q-q-plots/

4.7 Exercises

1. Define statistics. Understand criteria for classification of statistics.
2. Understand summary statistics and distinguish them from descriptive statistics.
3. Know the difference between logistic and exponential data distributions. Write a couple of functions to plot these distributions.
4. Study R control flow. Use the same for writing function which convert a randomly sampled data into logistic and exponential data distributions.
5. Try to understand continuous vs discrete data. Write UDF for converting continuous data variable into discrete data variable.
6. Practice `apply` functions using different types of data sets.
7. Try to write few UDFs for apply functions.
8. What is cross table or contingency table? Understand various tests meant for contingency tables.
9. Understand various types of independence tests. Practice them using R code.
10. Distinguish parametric tests from non-parametric tests. Perform them using R base functions.

4.8 Case

A descriptive statistic is a summary statistic that quantitatively describes or summarizes features from a collection of information, while descriptive statistics is the process of using and analyzing those statistics. Descriptive statistics is distinguished from inferential statistics by its aim to summarize a sample, rather than use the data to learn about the population that the sample of data is thought to represent.

Visit WHO and try to find out different indices related to public health. Collect data of different countries and try to find out outliers through such data using few descriptive statistics.

Chapter 5

Bivariate Analysis

"I have seen the future and it is very much like the present, only longer."

- Kehlog Albran, The Profit

Predictive analytics encompasses a variety of statistical techniques from data mining, predictive modelling, and machine learning, that analyze current and historical facts to make predictions about future or otherwise unknown events. [44] [45] In business, predictive models exploit patterns found in historical and transactional data to identify risks and opportunities. Models capture relationships among many factors to allow assessment of risk or potential associated with a particular set of conditions, guiding decision-making for candidate transactions.

Predictive analytics is used in many fields such as actuarial science, marketing, financial services, insurance, telecommunications, retail, travel, mobility, healthcare, pharmaceuticals, capacity planning, social networking and other fields. In finance, predictive analytics used for one of the best known applications that is credit scoring, which is used throughout financial services. Scoring models process a customer's credit history, loan application, customer data, etc., in order to rank-order individuals by their likelihood of making future credit payments on time.

Most of the predictive models falls under the area of bivariate analysis. Few bivariate techniques such as cross tables discussed in

Chapter 4. This Chapter has few techniques like bivariate normality, correlation and regression. *Normality tests and correlation analysis does not predict data.* However, they might serve as prerequisites for regression in which real activity related to prediction arises. There are quite a few ways to perform bivariate normality tests. Few of them can be *t test, ks test, wilcoxon test* and many more. T test is useful in testing if the variables under study has come from same population or not. This type of test is known as "test of independence". T test is also useful to test if two samples are paired or not. We will use a finance related data set for this section. Please refer to companion portal associated with this text at `https://github.com/Kamakshaiah/R-Book-Material` for data sets. I will use two data sets namely *BHEL* and *CIL* related to Maharatna sector in India. Each of these data sets has 12 columns and 12 rows. Rows represents years starting from 2005 to 2019. Columns represents measures and following are measures.

1. Independent variables
   - Total debt
   - Total assets
   - Financial leverage (D/E)
2. Dependent variables
   - Net profit margin (NPM)
   - Return on assets (RoA)
   - Return on equity (RoE)
3. Control variables
   - Size
   - Age
   - Growth
   - Inflation rate

The very fist activity is importing data sets. I shall do that using `read.csv()` function to get data into R environment.

```
> bhel <- read.csv("D:/Work/Books/R/R-for-data-analytics/Datasets/BHEL.csv")
> cil <- read.csv("D:/Work/Books/R/R-for-data-analytics/Datasets/CIL.csv")
> names(bhel)
```

```
 [1] "Year"                  "Total.Debt"             "Total.Assets"
 [4] "Financial.Levergae"    "X"                      "Net.Profit.Margin"
 [7] "Return.on.Assets.ROA." "Return.on.Equity.ROE." "X.1"
[10] "Size"                  "Age"                    "Inflation.Rate"
```

```
> names(cil)
```

```
 [1] "Year"                  "Total.Debt"             "Total.Assets"
 [4] "Financial.Levergae"    "X"                      "Net.Profit.Margin"
 [7] "Return.on.Assets.ROA." "Return.on.Equity.ROE." "X.1"
[10] "Size"                  "Age"                    "Growth"
[13] "Inflation.Rate"
```

I used absolute paths for these data sets. We have two unwanted data variables in data viz. *X* and *X.1*. These are blank columns in those CSV files. We now have to get rid of those blank columns.

```
> bhel <- bhel[, !names(bhel) %in% c("X", "X.1")]
> cil <- cil[, !names(cil) %in% c("X", "X.1")]
> names(bhel)
```

```
 [1] "Year"                  "Total.Debt"            "Total.Assets"
 [4] "Financial.Levergae"    "Net.Profit.Margin"     "Return.on.Assets.ROA."
 [7] "Return.on.Equity.ROE." "Size"                  "Age"
[10] "Inflation.Rate"
```

```
> names(cil)
```

```
 [1] "Year"                  "Total.Debt"            "Total.Assets"
 [4] "Financial.Levergae"    "Net.Profit.Margin"     "Return.on.Assets.ROA."
 [7] "Return.on.Equity.ROE." "Size"                  "Age"
[10] "Growth"                "Inflation.Rate"
```

5.1 T test & Bivariate normality

I explained T test in Section 4.5.2. The type of the test that was explained there is called *one sample T test*. There are few other types of tests namely; (1) paired sample T test, and (2) independent sample T test. These tests are known as *two sample T tests*. In this section I shall explain the later two types.

For instance, I can test profitability of BHEL by using *net profit* and *leverage*. The null hypothesis i.e. $H_0 : \mu_{NetProfit} = \mu_{Leverage} = 0$.

```
> t.test(bhel$Financial.Levergae, cil$Net.Profit.Margin)

        Welch Two Sample t-test
data:  bhel$Financial.Levergae and cil$Net.Profit.Margin
t = 5.0456, df = 16.767, p-value = 0.0001038
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.1934779 0.4720696
sample estimates:
mean of x mean of y
0.5897662 0.2569924
```

P value is close to zero (0.0001038). Null hypothesis can be rejected. There is evidence in support of alternative hypothesis i.e.

the difference between means of these variables is significant. Coming to inference, company's financial leverage seems to be good.

Suppose if I wish to assume that assets and debt are bivariate normal. I might be able to check as shown below.

```
> ks.test(bhel$Total.Debt, bhel$Total.Assets)

        Exact two-sample Kolmogorov-Smirnov test

data:  bhel$Total.Debt and bhel$Total.Assets
D = 0.66667, p-value = 0.001837
alternative hypothesis: two-sided
```

These variables does not seems to follow normality. It is better to check through visuals.

```
> qqplot(bhel$Total.Debt, bhel$Total.Assets)
```

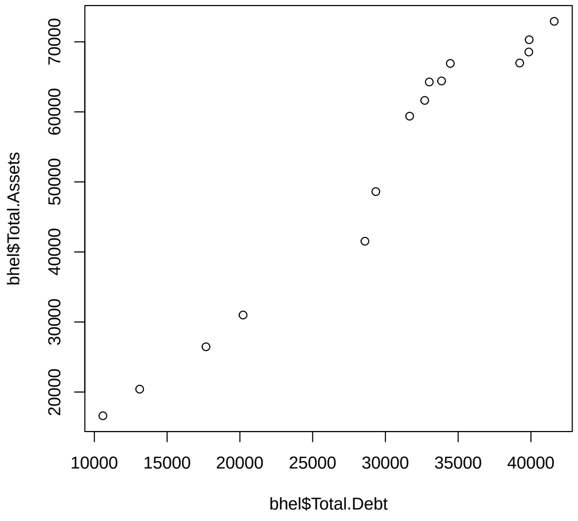


Figure 5.1: Bivariate normality plot for *assets* and *debt* for BHEL

Figure 5.1 shows visual representation for bivariate normality. From the figure it is clear that though there seems to be relationship but that doesn't seems to be strictly normal.

5.2 Correlation

In the broadest sense correlation is any statistical association, though it commonly refers to the degree to which a pair of variables are linearly related. The most familiar measure of dependence between two quantities is the *Pearson product moment correlation* coefficient, or *Pearson's correlation coefficient*, commonly called simply "the correlation coefficient". It is obtained by dividing the co-variance of the two variables by the product of their standard deviations. Karl Pearson developed the coefficient from a similar but slightly different idea by *Francis Galton.*

The correlation coefficient is +1 in the case of a perfect direct (increasing) linear relationship (correlation), 1 in the case of a perfect decreasing (inverse) linear relationship (anticorrelation), and some value in the open interval $(1, 1)$ in all other cases, indicating the degree of linear dependence between the variables. As it approaches zero there is less of a relationship. The closer the coefficient is to either 1 or 1, the stronger the correlation between the variables.

In R there is base function known as `cor()`. This works for all types of correlations namely (1) Karl Pearson r, (2) Spearman Rho, (3) Kendal's Tau. Use `help()` to know more about this function. The default method is *Pearson's r*, *Rho* is useful to compute correlation for ranked data. *Tau* is used to compute correlations if there are *ties* in ranked data.

Coming to financial data sets; I assume that firms BHEL and CIL are similar in using their financial leverage. Frankly, I don't if such assumption is valid or not. The assumption being both firms use their leverage in similar fashion. There is a function `cor` in R Base which is useful to calculate all those measures of correlation.

```
> cor(bhel$Financial.Levergae, cil$Financial.Levergae)
```

```
[1] 0.8156881
```

Since, the data is non-categorical and numerical, it can be interpret using *Pearson's r.* The r value is 0.81 so the relationship seems to be positive and also strong. Since, the data is not ranked so the other methods are not valid.

Now coming to correlation value. This value represents only the sample but not the population. Our interpretation is only applicable to the data that we collected some certain sampling strategy.

Obviously, there arises a question: is this relationship holds true in the population? This mean if taken all Maharathna units; is it possible to observer same type of relationship? That needs correlation significance test to answer this question.

```
> cor.test(bhel$Financial.Levergae, cil$Financial.Levergae)

        Pearson's product-moment correlation

data:  bhel$Financial.Levergae and cil$Financial.Levergae
t = 5.0839, df = 13, p-value = 0.0002096
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.5212119 0.9365970
sample estimates:
      cor
0.8156881
```

`cor.test()` in R does t test for null hypothesis $\rho = 0$. This means the association/relationship in the population is zero. The T value (difference between means) seems to be statistically significant. So, null hypothesis can be safely rejected. In other words, there is evidence in support of alternative hypothesis. This means, it is *possible* to observe same level of relationship in the population.

5.3 Simple linear regression

Simple linear regression is a linear regression model with a single explanatory variable. That is, it concerns two-dimensional sample points with one independent variable and one dependent variable (conventionally, the x and y coordinates in a Cartesian coordinate system) and finds a linear function (a non-vertical straight line) that, as accurately as possible, predicts the dependent variable values as a function of the independent variables. The adjective simple refers to the fact that the outcome variable is related to a single predictor. Consider the model function

$$y = \alpha + \beta x \tag{5.1}$$

which describes a line with slope β and y intercept α. In general such a relationship may not hold exactly for the largely unobserved population of values of the independent and dependent variables;

we call the unobserved deviations from the above equation the errors. Suppose we observe n data pairs and call them (x_i, y_i), i = 1, ..., n. We can describe the underlying relationship between y_i and x_i involving this error term ε_i by

$$y_i = \alpha + \beta x_i + \varepsilon_i \tag{5.2}$$

This relationship between the true (but unobserved) underlying parameters α and β and the data points is called a linear regression model. The goal is to find estimated values $\widehat{\alpha}$ and $\widehat{\beta}$ for the parameters α and β which would provide the *best* fit in some sense for the data points. As mentioned in the introduction, in this article the *best* fit will be understood as in the least squares approach: a line that minimizes the sum of squared residuals $\widehat{\varepsilon}_i$ (differences between actual and predicted values of the dependent variable y), each of which is given by, for any candidate parameter values α and β,

$$\widehat{\varepsilon}_i = y_i - \alpha - \beta x_i \tag{5.3}$$

There are few assumptions which need to be satisfied before performing *simple linear regression.* As a matter of caution we first need to do few tests for following assumptions.

1. Normality of residuals
2. Linearity of the data
3. Homogeneity of residual variance

These assumptions can be checked either before or after performing regression analysis. Usually, in R we need to obtain the model to check these assumptions. In BHEL and CIL data sets three are few dependent and independent variables. It is possible to make an assumption that *net profit* depends on *financial leverage* of the firm. In regression, it is not possible to make null hypothesis as by the subject. This means, the statement: (H_0) net profit depends on financial leverage seems to be meaningful, but does not make sense. This is because, regression tests the estimates known as *intercept* and *slope.*

```
> bhelregfit <- lm(bhel$Net.Profit.Margin ~ bhel$Financial.Levergae,
+     data = bhel)
> summary(bhelregfit)$coefficients
```

```
                           Estimate Std. Error   t value     Pr(>|t|)
(Intercept)              -0.2879885 0.06515624 -4.419968 6.917635e-04
bhel$Financial.Levergae   0.6436481 0.10961066  5.872130 5.481697e-05

> summary(bhelregfit)$r.squared

[1] 0.7262116

> summary(bhelregfit)$fstatistic

   value    numdf    dendf
34.48192  1.00000 13.00000

> summary(bhelregfit)$residuals[1:5]

            1             2             3             4             5
-0.0019090906 -0.0012483172 -0.0008173696 -0.0733115893  0.0106926069
```

I tried retrieving information from regression object `bhelregfit` bits and pieces only to keep the text tidy. It is possible to retrieve information related to model using `summary()` function.

From the regression analysis it is clear that both the estimates are statistically significant. P values for β_0 and β_1 are significantly different from zero. This means, **it is possible to reject null hypotheses:** $H_0 : \beta_0 = \beta_1 = 0$. What does it mean? if β_0 is zero, then the regression line must be passing through origin. In which case, the fixed effect in regression model is simply zero. Coming to β_1, if this value is zero, then the slope of the regression line is going to be zero. That is not possible in any case. So, the inference in regression always revolves round the estimates in stead of variables under study. However, it is possible to extend such interpretation to the variables but only using intuition. Since, estimates are not equal to zero, it is possible to infer dependency between these variables. Going back to values β_0 is negative but a moderate value for which P value is significant. So, the effect (may be fixed) whatever is being explained by β_0 is valid for decisions. Coming to β_1, the value if positive but strong and also significant. So, whatever, the effect (may be variable) is being explained by β_1 is valid for decisions. Finally, it is possible to infer a significant dependency relationship between these two study variables. In other words, the explanatory variable i.e. Leverage is able to explain dependent variable which is Net Profit in the regression model. While coming to fitness of the model; both values i.e. R Squared (0.72) and F Statistic (5.482e-05) seems to show fitness in the model.

The object `bhelregfit` represents the regression model. This is an object which has attributes associated with it. For instance, *fitted values* can be obtained by performing *tab search*, this means write the name of the object in R Console followed by *dollar* sign ($)

and then press tab to see what attributes are associated with this object. The *fitted values* and *residuals* can be obtained as shown below.

```
> bhelregfit$fitted.values

         1          2          3          4          5          6          7
0.04190909 0.02914832 0.01841737 0.03771159 0.03560739 0.06398520 0.07691474
         8          9         10         11         12         13         14
0.11181324 0.13721506 0.14493182 0.15513238 0.13191468 0.14188292 0.12544945
        15
0.12216675
```

```
> bhelregfit$residuals

            1             2             3             4             5
-0.0019090906 -0.0012483172 -0.0008173696 -0.0733115893  0.0106926069
            6             7             8             9            10
 0.0231148047  0.0578852616  0.0372867580  0.0073849399 -0.0137318224
           11            12            13            14            15
-0.0354323788  0.0137853161 -0.0017829188  0.0001505537 -0.0220667543
```

These two objects are very important for regression diagnostics. These model objects can be used to make few plots. These plots helps in verifying regression assumptions.

```
> par(mfrow = c(2, 2))
> plot(bhelregfit)
```

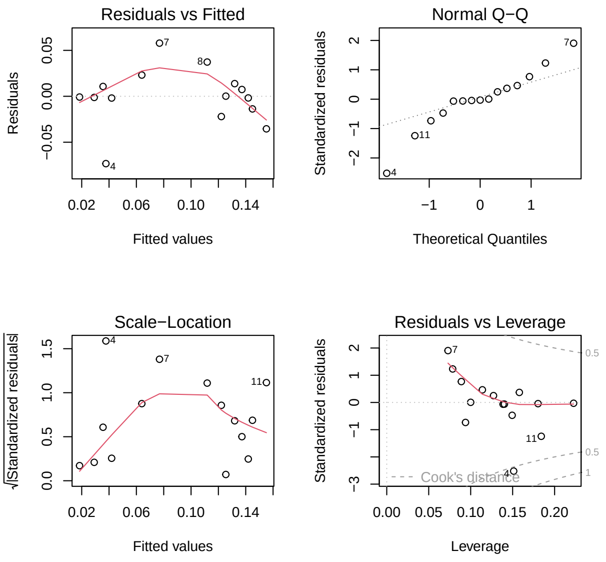


Figure 5.2: Regression diagnostics for BHEL regression fit.

Above visual speaks quite a bit of information related to regression model. *First*, plot *Residual vs Fitted* must not have any distinct patter and points should hang along horizontal line. This type of visual satisfies assumptions related to linearity. In the sample data most of the points are above and below this line and there is a little *fan effect.* This type of effect shows heteroskedasticity. Heteroskedasticity refers unreasonable dispersion in the data. *Second* plot is *Normal QQ* plot. The points in this plot should wander around the diagonal line. The proximity of points to diagonal line determines the extent to which this data is bivariate normal. *Third* plot is *Scale Location* plot. The points in this plot should be distributed roughly equal above and blow the horizontal line. Otherwise, it is subjected to heteroscedasticity. *Last* plot is *Residual vs Leverage*, this plot is rather more crucial for regression analysis. This plot helps us in identifying *outliers.* Outlier is a point that is extreme in outcome variable. The presence of outliers may affect the interpretation of the model, because it increases the RSE. RSE is known as *Residual Standard Error* which helps in evaluating fitness of the model to data. Outliers can be identified by examining the standardized residual (or studentized residual), which is the residual divided by its estimated standard error. Standardized residuals can be interpreted as the number of standard errors away from the regression line. Observations whose standardized residuals are greater than 3 in absolute value are possible outliers. A data point has high leverage, if it has extreme predictor x values. This can be detected by examining the leverage statistic or the *hat value.* A value of this statistic above $2(p+1)/n$ indicates an observation with high leverage; where, p is the number of predictors and n is the number of observations. In regression analysis, points 4, 7 and 11 are outlier as well as influential. What are these points.

```
> bhel[c(4, 7, 11), ]
```

```
   Year Total.Debt Total.Assets Financial.Levergae Net.Profit.Margin
4  2016    33859.2      66912.5          0.5060220           -0.0356
7  2013    39854.2      70298.3          0.5669298            0.1348
11 2009    28591.9      41530.7          0.6884522            0.1197
   Return.on.Assets.ROA. Return.on.Equity.ROE.     Size Age Inflation.Rate
4                -0.0137               -0.0276 4.825507  52            4.9
7                 0.0941                0.2173 4.846945  49           10.9
11                0.0756                0.2425 4.618369  45           10.9
```

The data related to years 2009, 2013 and 2016 are outliers. Something, went wrong in these years with respect to study variables. In other words, firm must have suffered leveraging its finances during these years. These assumptions can be evaluated by redoing entire regression analysis excluding these rows from the data. It is possible to infer that whether these rows are truly influential or not based on R Square value from the model.

```
> bhelwithout <- bhel[c(4, 7, 11), ]
> out <- lm(bhelwithout$Net.Profit.Margin ~ bhelwithout$Financial.Levergae,
+     data = bhelwithout)
> summary(out)$r.squared
```

```
[1] 0.4921416
```

R Squared decreased, which means these years are not truly influential in analysis. This may happens when there is random error in the data. One possible solution could be increasing sample size. Such strategy might be able to improve the fitness of the model.

5.3.1 Prediction

A prediction, or forecast, is an activity that enable analyst to foresee the future event. They are often, but not always, based upon experience or knowledge. In regression, estimates which were obtained through analysis were used to predict future events. For instance, in above analysis the intercept and slope were two estimates the model identified and the instances of *dependent* variables can be predicted using *independent* variable. The regression equation for prediction is as follows:

$$Netprofitmargin = -0.28799 + 0.64365 * Financialleverage \tag{5.4}$$

It is possible to obtain various values of *net profit margin* using different values of *financial leverage.* This can be done using a base function `predict()` in R.

```
> predData <- cbind.data.frame(bhel$Net.Profit.Margin, bhel$Financial.Levergae)
> predData
```

```
   bhel$Net.Profit.Margin bhel$Financial.Levergae
1                  0.0400               0.5125435
2                  0.0279               0.4927178
3                  0.0176               0.4760457
4                 -0.0356               0.5060220
5                  0.0463               0.5027529
6                  0.0871               0.5468419
7                  0.1348               0.5669298
8                  0.1491               0.6211496
9                  0.1446               0.6606150
10                 0.1312               0.6726041
11                 0.1197               0.6884522
12                 0.1457               0.6523801
13                 0.1401               0.6678672
14                 0.1256               0.6423354
15                 0.1001               0.6372353
```

```
> predValues <- predict(bhelregfit, predData)
> length(predValues)
```

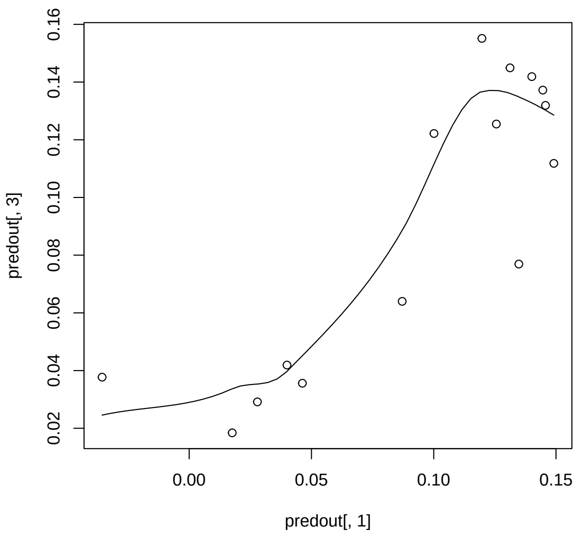
```
[1] 15
```

```
> cbind.data.frame(predData, predValues)
```

```
   bhel$Net.Profit.Margin bhel$Financial.Levergae predValues
1                  0.0400               0.5125435 0.04190909
2                  0.0279               0.4927178 0.02914832
3                  0.0176               0.4760457 0.01841737
4                 -0.0356               0.5060220 0.03771159
5                  0.0463               0.5027529 0.03560739
6                  0.0871               0.5468419 0.06398520
7                  0.1348               0.5669298 0.07691474
8                  0.1491               0.6211496 0.11181324
9                  0.1446               0.6606150 0.13721506
10                 0.1312               0.6726041 0.14493182
11                 0.1197               0.6884522 0.15513238
12                 0.1457               0.6523801 0.13191468
13                 0.1401               0.6678672 0.14188292
14                 0.1256               0.6423354 0.12544945
15                 0.1001               0.6372353 0.12216675
```

The object `predValues` is subset, composing of variables *Net Profit Margin* and *Financial Leverage* of *bhel* data set. All predicted values has close resemblance with actual values. How to know that?

```
> predout <- cbind.data.frame(predData, predValues)
> r <- cor(predout[, 1], predout[, 3])
> c(r, r^2)

[1] 0.8521805 0.7262116

> scatter.smooth(predout[, 1], predout[, 3])
```



Correlation coefficient is strong and also positive. The values has 72% (R^2) of resemblance. This is what is called *prediction accu-*

racy. There are number of ways to evaluate prediction accuracy in regression. Measures such as

1. Min-Max Accuracy
2. Mean Absolute Percentage Error (MAPE) [46]
3. Mean Absolute Error (MAE)
4. Root Mean Square Error (RMSE)
5. Normalized Root Mean Square Error (NRMSE)

These are, in fact, very few which helps in assessing accuracy of prediction but not the model. There is one gigantic technique known as *Confusion Matrix*, it is widely used in few operational areas like machine learning and artificial intelligence. This technique is used mostly on classification problems where the data is categorical in nature. That is why, confusion matrix is popular in logistic regression, where dependent variable is categorical. It may not be possible to explain above all methods, I shall explain only first two methods in the above list. Expressions MM Accuracy and MAPE are as follows:

$$MMAccuracy = mean\left[\frac{min(Actuals, Predicted)}{max(Actuals, Predicted)}\right] \tag{5.5}$$

$$MAPE = mean\left[abs\left(\frac{Predicted - Actuals}{Actuals}\right)\right] \tag{5.6}$$

```
> actuals <- bhel$Net.Profit.Margin
> predicted <- predValues
> mins <- apply(cbind.data.frame(actuals, predicted), 1, min)
> maxs <- apply(cbind.data.frame(actuals, predicted), 1, max)
> mmacuracy <- mean(mins/maxs)
> mape <- mean(abs((predicted - actuals)/actuals))
> c(mmacuracy, mape)
```

```
[1] 0.7389458 0.2769850
```

Why I am interested in explaining MM Accuracy and MAPE? Because, MM Accuracy explains the accuracy of predictions and MAPE explains error in predictions.

```
> mmacuracy + mape
```

```
[1] 1.015931
```

The value should be close to one. I request the reader to evaluate predictions using rest of the methods. It is better to convert above procedure into a functions (UDF) for these two methods appear again in multiple regression in forthcoming Chapter.

```
> MMAccuracy <- function(actuals, predicted) {
+     mins <- apply(cbind.data.frame(actuals, predicted), 1, min)
+     maxs <- apply(cbind.data.frame(actuals, predicted), 1, max)
+     mmaccuracy <- mean(mins/maxs)
+     return(mmaccuracy)
+ }
> MAPE <- function(actuals, predicted) {
+     mape <- mean(abs((predicted - actuals)/actuals))
+     return(mape)
+ }
> checkAccuracy <- function(actuals, predicted) {
+     mmaccuracy <- MMAccuracy(actuals, predicted)
+     mape <- MAPE(actuals, predicted)
+     return(c(mmaccuracy, mape))
+ }
> checkAccuracy(actuals, predValues)
```

```
[1] 0.7389458 0.2769850
```

I wrote three functions viz. `MMAccuracy()`, `MAPE()` and also `check-Accuracy()`. The function `checkAccuracy()` is a composite of first two functions. The function `checkAccuracy()` does exactly well as expected.

Notes

[44]Nyce, Charles (2007), Predictive Analytics White Paper (PDF), American Institute for Chartered Property Casualty Underwriters/Insurance Institute of America, p. 1

[45]Predictive analytics. Obtained from https://en.wikipedia.org/wiki/Predictive_analytics

[46]Read about MAPE at https://en.wikipedia.org/wiki/Mean_absolute_percentage_error

5.4 Exercises

1. Understand mathematics behind one sample T test and two sample T test. Practice `t.test()` function.
2. What is correlation? Understand procedure to test correlation significance.
3. Understand bivariate normality and different tests associated with it. Perform bivariate normality using few R functions.
4. Study about `lm()` function and its use in fitting regression models.
5. Read about ANOVA and perform one factor, two factor ANOVA in R.
6. What is prediction accuracy? Study and evaluate various measures for the same. Write UDFs for few accuracy measures such as MSE, MAE, RMSE etc.

5.5 Case

Businesses today seem to have a multitude of product offerings to choose from predictive analytics vendors in every industry, which can help businesses leverage their historical data store by discovering complex correlations in the data, identifying unknown patterns, and forecasting. This is hardly surprising considering the fact that predictive analytics can help businesses answer questions such as "Are customers likely to buy my product?" Or even "Which marketing strategies might be most successful?"

Instead of simply presenting information about past events to a user, predictive analytics estimate the likelihood of a future outcome based on patterns in the historical data. This allows clinicians, financial experts, and administrative staff to receive alerts about potential events before they happen, and therefore make more informed choices about how to proceed with a decision.

Study about importance of predictive analytics for effective business practices. Investigate as how firms answer few future uncertainties using predictive analytics.

Chapter 6

Multivariate Analysis

"My ears hear colors and my eyes see sounds."

- Suzy Kassem, Rise Up and Salute the Sun: The Writings of Suzy Kassem

Human life is full of choices. Variety is the property of Nature. There is no single solution for any problem. The solution depends on the approach of the seeker. There are multifarious ways to solve a problem as by each approach. Human senses are fickle so the approaches. There are alternatives to every thing that we think or do. Say it is a sound, taste, touch, sight or smell. There is no choice but choices. In research, though it is possible to analyse each variable and make summaries, sometimes it requires all encompassing view. This all encompassing view or sight requires analyses in a single go but not through bits and pieces. Such endeavor is referred to multivariate analysis.

Multivariate analysis is based on the statistical principle of multivariate statistics, which involves observation and analysis of more than one statistical outcome variable at a time. Typically, multivariate analysis 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.

Multivariate analysis has many different models, each with its own type of analysis. Few of them are as follows:

1. Multivariate correlation (MC)
2. Canonical correlation (Cancor)
3. Multiple regression (MR)
4. Multivariate analysis of variance (MANOVA)
5. Multivariate regression (GLM)
6. Correspondence analysis (CA)
7. Principal components analysis (PCA)
8. Factor analysis (FA)
9. Cluster analysis (Clus. A)

These techniques are very few among those available for analysis. Very few of the above models were discussed in this text.

6.1 Multivariate correlation

The coefficient of multivariate correlation is a measure of how well a given variable can be predicted using a linear function of a set of other variables. It is the correlation between the variable's values and the best predictions that can be computed linearly from the predictive variables. The coefficient of multivariate correlation takes values between 0 and 1; a higher value indicates a better 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.

Performing multivariate correlation in R is very simple. In R both bivariate and multivariate correlation can be performed using same function `cor()`. Suppose if we wish to perform multivariate correlation for BHEL data set.

```
> head(cor(bhel))
```

```
                             Year  Total.Debt Total.Assets Financial.Levergae
Year                    1.0000000  0.66587081    0.8595453         -0.8753358
Total.Debt              0.6658708  1.00000000    0.9364731         -0.3704883
Total.Assets            0.8595453  0.93647307    1.0000000         -0.6714818
Financial.Levergae     -0.8753358 -0.37048833   -0.6714818          1.0000000
Net.Profit.Margin      -0.7317318 -0.14452020   -0.4383073          0.8521805
Return.on.Assets.ROA.  -0.7144300 -0.08847959   -0.3967161          0.8571531
                      Net.Profit.Margin Return.on.Assets.ROA.
Year                         -0.7317318           -0.71443002
Total.Debt                   -0.1445202           -0.08847959
Total.Assets                 -0.4383073           -0.39671605
Financial.Levergae            0.8521805            0.85715307
```

```
Net.Profit.Margin          1.0000000           0.98747920
Return.on.Assets.ROA.      0.9874792           1.00000000
                      Return.on.Equity.ROE.       Size       Age
Year                           -0.7600879  0.8516004  1.0000000
Total.Debt                     -0.1536473  0.9481388  0.6658708
Total.Assets                   -0.4725824  0.9866868  0.8595453
Financial.Levergae              0.9226214 -0.6090168 -0.8753358
Net.Profit.Margin               0.9684636 -0.4003935 -0.7317318
Return.on.Assets.ROA.           0.9846404 -0.3546275 -0.7144300
                      Inflation.Rate
Year                     -0.21804690
Total.Debt                0.37616318
Total.Assets              0.08618236
Financial.Levergae        0.57635505
Net.Profit.Margin         0.63846818
Return.on.Assets.ROA.     0.67292428
```

The function `head()` helps in bringing first 6 rows of information in R. This output can be used to identify few strong, moderate and poor relationships. For instance, the correlation between *Total Debts* observed to have strong and positive relationship with *Total Assets, Size* and *Age*. Interestingly *Assets* and *Debts* negatively associated with *Financial Leverage* and *Net Profit*. Interestingly, *Net Profit Margin* strongly correlated with *Financial Leverage* and *Returns*. *Age* of the unit negatively associated with *Leverage* and *Profit*. Interestingly *Inflation Rate* is positively correlated with most of the variables except *Age* of the unit. Usually analyst try to pick up those relations which adds value to his/her research/reports.

6.2 Canonical correlation

Canonical-correlation analysis (CCA) is a special case of multivariate correlation. CCA is also called Canonical Variates Analysis,(CVA) is a way of inferring information from cross co-variance 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. Canonical correlation analysis, is a general procedure for investigating the relationships between two sets of variables. The method was first introduced by Harold Hotelling in 1936, although in the context of angles between flats the mathematical concept was published by Jordan in 1875.

Performing Canonical Correlation Analysis in R is very simple and we can do that using a special function `cancor()`. This function is available from base package *stats* and this package is available right after installing R Base. There is no necessity to install using `install.packages()` command.

I am going to use certain data set related to HR domain. This data set is available at text book companion repository at https://github.com/Kamakshaiah/R-Book-Material. The data set has 9 columns and 30 rows. Rows represents individuals (employees). Columns represents three factors namely (1) training, (2) evaluation, (3) performance. Each of these factors were constructed using three variables. Let us try to see if *training* as a factor can be correlated with *performance*.

```
> hrdf <- read.csv("D:/Work/Books/R/R-for-data-analytics/Datasets/hrtraining.csv")
> dim(hrdf)

[1] 30  9

> names(hrdf)

[1] "train1"  "train2"  "train3"  "perf1"   "perf2"   "perf3"   "impact1"
[8] "impact2" "impact3"

> cor(hrdf[, 1:3], hrdf[, 4:6])

             perf1      perf2        perf3
train1 -0.12882163 -0.2685613 -0.014684893
train2 -0.23450694 -0.3229367 -0.151661214
train3 -0.03696987 -0.1532965 -0.001147568

> cancor(hrdf[, 1:3], hrdf[, 4:6])

$cor
[1] 0.35170636 0.27339990 0.06121762
$xcoef
                [,1]         [,2]         [,3]
train1 -0.0238489214  0.038463238  0.017443311
train2  0.0377710754 -0.022605623 -0.003267819
train3  0.0003829919 -0.004677821 -0.023657298
$ycoef
             [,1]        [,2]        [,3]
perf1 -0.05052452  0.01099669 -0.20943539
perf2 -0.06564757 -0.13317881  0.09108505
perf3 -0.06274374  0.14242555  0.11968668
$xcenter
  train1   train2   train3
28.60000 29.50000 30.63333
$ycenter
   perf1    perf2    perf3
2.800000 2.600000 3.233333
```

Correlations obtained using `cor()` function are different from those of `cancor()`. Bivariate correlations are negative whereas canonical correlations are not. From the result it is clear that the association at very first i.e. *training1 vs performance1* is strong and the association gradually reduced to third training activity. The effect of training on employee performance at the end is not good. It is also possible to try visuals for this analysis.

The Figure 6.1 adds visualization to the above canonical correlation analysis. There is one package called *CCA* which has methods to obtain advanced visualizations from canonical correlation analysis. [47] [48]

```
> out <- cancor(hrdf[, 1:3], hrdf[, 4:6])
> barplot(out$cor)
```

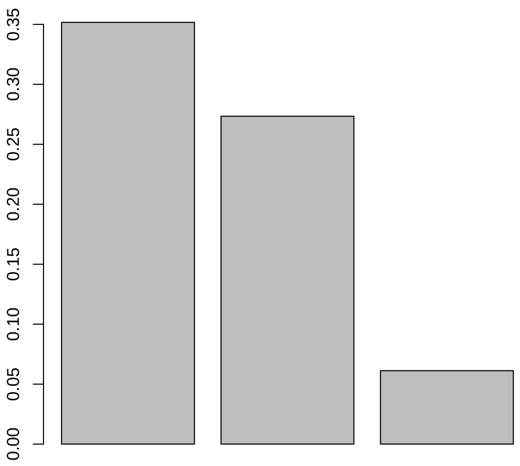


Figure 6.1: Barplot for canonical correlations for training vs performance

6.3 Multiple regression

Multiple linear regression attempts to model the relationship between two or more explanatory variables and a response variable by fitting a linear equation to observed data. Every value of the independent variable x is associated with a value of the dependent variable y. The population regression line for p explanatory variables $x1, x2, ..., xp$ is defined to be $y = 0 + 1x1 + 2x2 + ... + pxp$. This line describes how the mean response y changes with the explanatory variables. The observed values for y vary about their means y and are assumed to have the same standard deviation. The fitted values $b0, b1, ..., bp$ estimate the parameters $0, 1, ..., p$ of the population regression line.

Coming to BHEL data set, for instance, it is possible to verify assumption that *Net Profits* are influenced by *Financial Leverage* and *Inflation Rate.*

```
> bhelmrout <- lm(bhel$Net.Profit.Margin ~ bhel$Financial.Levergae *
+     bhel$Inflation.Rate, data = bhel)
> summary(bhelmrout)

Call:
lm(formula = bhel$Net.Profit.Margin ~ bhel$Financial.Levergae *
    bhel$Inflation.Rate, data = bhel)
Residuals:
      Min        1Q    Median        3Q       Max
-0.067071 -0.012431  0.005056  0.015492  0.035938
Coefficients:
                                            Estimate Std. Error t value
(Intercept)                                 -0.50579    0.16580  -3.051
bhel$Financial.Levergae                      0.95750    0.28940   3.309
bhel$Inflation.Rate                          0.04268    0.02452   1.741
bhel$Financial.Levergae:bhel$Inflation.Rate -0.06307    0.04025  -1.567
                                            Pr(>|t|)
(Intercept)                                  0.01104 *
bhel$Financial.Levergae                      0.00697 **
bhel$Inflation.Rate                          0.10961
bhel$Financial.Levergae:bhel$Inflation.Rate  0.14545
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 0.02912 on 11 degrees of freedom
Multiple R-squared:  0.8027,       Adjusted R-squared:  0.7489
F-statistic: 14.92 on 3 and 11 DF,  p-value: 0.0003417
```

Both *Intercept* and *Slope* for *Financial leverage* are statistically significant. *Inflation Rate* is not statistically significant. The interaction between *Financial Leverage* and *Inflation rate* is also not significant in the model. This apart, estimates for intercept and interaction are negative. Coming to model fit, both R^2 and F Statistic seems to support the model. So, it is possible to infer that profits of BHEL were influenced by financial leverage but not inflation rate. The assumption that financial leverage together with inflation influence profits of the firm can be safely rebuked for interaction is not statistically significant in the model. Let us check regression diagnostics.

A quick investigation reveals that there are few outliers and the relationship is non-normal. I may be able to predict using following procedure.

```
> mrpreds <- predict(bhelmrout, bhel)
> mractuals <- apply(cbind.data.frame(bhel$Financial.Levergae,
+     bhel$Inflation.Rate), 1, min)
> checkAccuracy(mractuals, mrpreds)

[1] 0.1469858 0.8530142
```

The object `mractuals` represents actuals for accuracy check. Why? Because, in multiple regression there are two or more variables in right side of the regression equation.

Net Profits = -0.50579 + 0.95750 * Financial Leverage
+ 0.04268 * Inflation Rate -0.06307 * (Financial

```
> par(mfrow = c(2, 2))
> plot(bhelmrout)
```

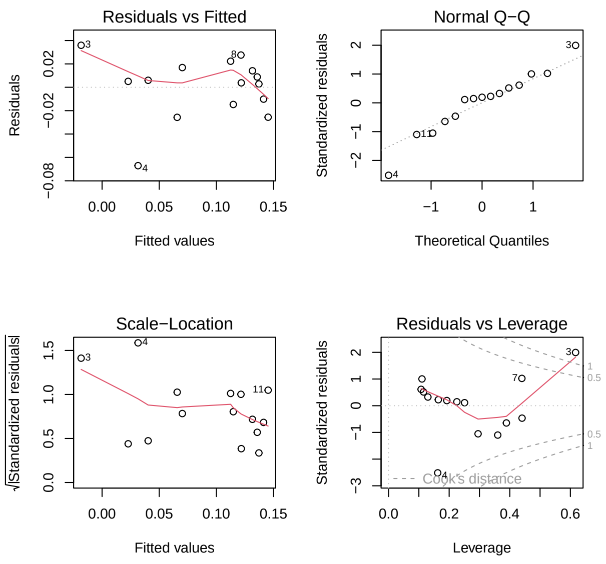


Figure 6.2: Multiple regression diagnostics for BHEL.

Leverage : Inflation Rate)

Whereas, methods `MMAccuracy` and `MAPE()` accepts inputs as single column vectors. However, in multiple regression actuals has two or more vectors. So I had to convert actuals into a single vector using `apply()` function. Confusing? Okay, the total for accuracy check must be one. Isn't it? Let us try.

```
> checkAccuracy(mractuals, mrpreds)[1] + checkAccuracy(mractuals,
+     mrpreds)[2]

[1] 1
```

6.3.1 Formulae for regression in R

Doing multiple regression in R is highly intuitive process. We need to understand the way *formula* is being used in the `lm()` function.

There are different ways to perform multiple regression in R using variety of symbols in model definition.

Symbol	Example	Meaning
+	+X	include this variable
-	-X	delete this variable
:	X : Z	include the interaction between these variables
*	X * Y	include these variables and the interactions between them
|	X|Z	conditioning: include x given z
^	$(X + Z + W)^3$	include these variables and all interactions up to three way
I	I(X * Z)	as is: include a new variable consisting of these variables multiplied
1	X-1	intercept: delete the intercept (regress through the origin)

Table 6.1: Notations for regression formula.

6.4 Factor Analysis

Factor analysis is a statistical method used to describe variability among observed, correlated variables in terms of a potentially lower number of unobserved variables called factors. Factor analysis searches for joint variations in response to unobserved latent variables. The observed variables are modeled as *linear combinations* of the potential factors, plus *error* terms. Factor analysis aims to find independent latent variables. [49]

Factor analysis is related to principal component analysis (PCA), but the two are not identical. There has been significant controversy in the field over differences between the two techniques. PCA can be considered as a more basic version of exploratory factor analysis (EFA). PCA was developed in the early days prior to the advent of high speed computers. Both PCA and factor analysis aim to reduce the dimensionality of a set of data, but the approaches taken to do so are different for the two techniques. Factor analysis is clearly designed with the objective to identify certain factors from the observed variables, whereas PCA does not directly address this objective; at best, PCA provides an approximation to the required factors.

Definition

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 writable as a linear combination of the common factors:

$$x_i - \mu_i = l_{i1}F_1 + \cdots + l_{ik}F_k + \varepsilon_i \tag{6.1}$$

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

$$x - \mu = LF + \varepsilon. x - \mu = LF + \varepsilon \tag{6.2}$$

If we have n observations, then we will have the dimensions $x_{p\times n}$, $L_{p\times k}$, and $F_{k\times n}$. Each column of x and F denotes values for one particular observation, and matrix L does not vary across observations. Any solution of the above set of equations following the constraints for F is defined as the factors, and L as the loading matrix. The onus of analyst at the end is to solve the above equations and find loadings matrix.

I will start analysis with PCA and then slowly get into FA. By the way words such as component in PCA, factor in FA, dimensions in multidimensional analysis (MDA) are refer to one particular aspect called *latent construct*.

6.4.1 PCA

I shall apply PCA on *hrdf* in which there are three constructs viz. training (train), performance (perf), and impact (Impact). Each of these components has three variables. In other words, these components were measured using each of three variables (also known as research questions). Now for PCA there will be two questions. Are these components are truly unique? What are the relationships among these components.

There is one very important base function `princomp()` in R which is useful to perform PCA. Use `help("princomp")` in R console to know more about this function. In help-doc go strait to down and find the example. Otherwise, it is also possible to use `example(princomp)` which will show examples with executed results in the console. I am going to use this function using another argument

```
> par(mfrow = c(1, 2))
> plot(hrpcaout)
> plot(summary(hrpcaout)$sdev, type = "b", col = "red")
```

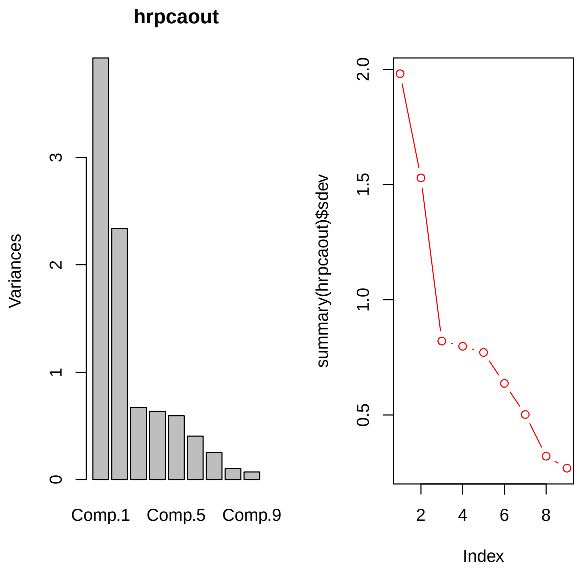


Figure 6.3: PCA for *hrdf* data set.

`cor = TRUE`. This will make sure that PCA is performed using correlations in stead of co-variances. However, such argument is not mandatory for performing PCA.

```
> hrpcaout <- princomp(hrdf, cor = TRUE)
> hrpcaout

Call:
princomp(x = hrdf, cor = TRUE)
Standard deviations:
   Comp.1    Comp.2    Comp.3    Comp.4    Comp.5    Comp.6    Comp.7    Comp.8
1.9809168 1.5287935 0.8207259 0.7981096 0.7714035 0.6370217 0.5018834 0.3209707
   Comp.9
0.2691096
 9  variables and  30 observations.

> summary(hrpcaout)

Importance of components:
                          Comp.1    Comp.2     Comp.3     Comp.4     Comp.5
Standard deviation     1.9809168 1.5287935 0.82072592 0.79810960 0.77140345
Proportion of Variance 0.4360035 0.2596900 0.07484345 0.07077544 0.06611814
Cumulative Proportion  0.4360035 0.6956934 0.77053689 0.84131232 0.90743047
                           Comp.6     Comp.7     Comp.8      Comp.9
Standard deviation     0.63702173 0.50188338 0.32097074 0.269109619
Proportion of Variance 0.04508852 0.02798744 0.01144691 0.008046665
Cumulative Proportion  0.95251899 0.98050642 0.99195333 1.000000000
```

Output will not return much results. Function `princomp()` gets standard deviation by default but it is possible to get other useful information by using `summary()`. As far as plots are concerned; left side plot of Figure 6.3 is a barplot obtained using variances for components. The right side plot in the figure is called a *scree plot*. Scree plot is useful to know as how may components are really useful for interpretation. Scree plots are interpreted using a dent in curve called *elbow*. From the scree plot it is clear that there are three potential components in the data. So the analysis support analyst assumption. Actually, what is important in PCA is loadings matrix. It is possible to retrieve loadings matrix using the following statement.

```
> hrpcaout$loadings

Loadings:
        Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6 Comp.7 Comp.8 Comp.9
train1   0.210  0.571                       0.228         0.112  0.748
train2   0.260  0.501  0.103                0.506  0.128 -0.210 -0.592
train3   0.154  0.488 -0.407  0.296 -0.168 -0.626         0.114 -0.207
perf1   -0.438  0.178 -0.106        -0.171  0.127 -0.669 -0.514
perf2   -0.352         0.245  0.750 -0.374  0.177  0.276
perf3   -0.373  0.225  0.153         0.621 -0.292  0.383 -0.408
impact1 -0.301  0.247  0.453 -0.533 -0.487 -0.285  0.157  0.116
impact2 -0.432  0.203                0.384  0.168 -0.287  0.692 -0.186
impact3 -0.364        -0.722 -0.244 -0.178  0.240  0.442
               Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6 Comp.7 Comp.8 Comp.9
SS loadings     1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000
Proportion Var  0.111  0.111  0.111  0.111  0.111  0.111  0.111  0.111  0.111
Cumulative Var  0.111  0.222  0.333  0.444  0.556  0.667  0.778  0.889  1.000
```

Loadings for Component 1, 2, 3 and also 6 seems to be valid for interpretation. Loadigns does not seems to be strong. Most of the values are close to 0.3. However, that is not a matter for observations. Mostly value of loadings depends on data and the context. As far as interpretation goes, Component 1 seems to be interesting compared to other components. All the variables related to *training* has positive loadings while others has negative. This means, there is clearly some certain opposing content in between variables. Training is not necessarily seems to impact performance. It is possible to make endless insights from loadings matrix by taking each variable against its respective component.

Now let me perform Exploratory Factor Analysis (EFA) with the help of a base function `factanal()` in R using same data. Use `help("factanal)` in the console to know about this function. The function `factanal()` requires two arguments viz. data set and *factors*.

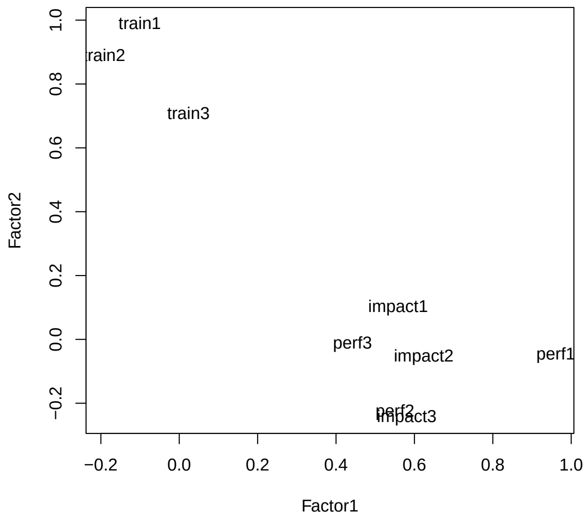
```
> hrfactanal <- factanal(hrdf, 3)
> hrfactanal

Call:
```

```
factanal(x = hrdf, factors = 3)
Uniquenesses:
 train1  train2  train3   perf1   perf2   perf3 impact1 impact2 impact3
  0.005   0.170   0.489   0.005   0.610   0.344   0.636   0.005   0.545
Loadings:
        Factor1 Factor2 Factor3
train1  -0.101   0.990
train2  -0.192   0.890
train3           0.709
perf1    0.961           0.264
perf2    0.550  -0.228   0.187
perf3    0.442           0.678
impact1  0.559           0.206
impact2  0.624           0.776
impact3  0.580  -0.245   0.241
               Factor1 Factor2 Factor3
SS loadings      2.507   2.403   1.281
Proportion Var   0.279   0.267   0.142
Cumulative Var   0.279   0.546   0.688
Test of the hypothesis that 3 factors are sufficient.
The chi square statistic is 4.45 on 12 degrees of freedom.
The p-value is 0.974
```

Factor analysis on the other hand outputs all required information unlike PCA. Output from FA is more or less similar to PCA. The very first factor is the best in explaining all the variables. The opposing behavior between training, impact and performance are distinct from the analysis. The χ^2 statistic is test of spherecity. This means, χ^2 tries to answer the question: does this model has fitness? The P Value is close to one. So it is not possible to reject null hypothesis. Whereby, concluding that the data fits 3 factor solution. [50]

It is possible to make text plot using `factanal()` results unlike the one in PCA.

```
> loads <- hrfactanal$loadings[, 1:3]
> plot(loads, type = "n")
> text(loads, labels = rownames(loads))
```



The plot adds visualization to realizations from numerical analysis. There are two clusters in the visual. All the training variables are grouped to left top corner and other variables are grouped to left bottom corner of the graph. Again in those groups variables *train3* and *perf1* seems to disagree with other members in their groups. There is numerical evidence in support of this observation from the numerical output. So the opposing behavior is so distinct from the plot. [51]

Notes

[47] Maciak, Canonical Correlations in R. Available at http://www.karlin.mff.cuni.cz/~maciak/NMST539/cvicenie11.html

[48] Visit https://cran.r-project.org/web/packages/CCA/ for more details on CCA package.

[49] Factor Analysis. Retrieved from https://en.wikipedia.org/wiki/Factor_analysis

[50] Revelle, W. Find various goodness of fit statistics for factor analysis and principal components. Available from https://personality-project.org/r/psych/help/factor.stats.html

[51] I am not trying to interpret numerical outputs for it is endless. It is possible to interpret each number for a pair of variable and factor. It takes lot of textual interpretation. So, I request the reader to get insights from the output here or from that of personal practice.

6.5 Exercises

1. Study about the area of multivariate analysis and its significance in statistics.
2. Understand multivariate correlation and try to perform the same using various data sets.
3. Write User Defined Function (UDF) for computing P values for correlation significance tests using R code.
4. Write User Defined Functions (UDF) for computing various measures involved in multiple regression.
5. Understand difference between multiple regression and MANOVA. Try to perform MANOVA in R using base functions.
6. Understand the difference between procedures related to prediction accuracy for simple linear regression and multiple regression.
7. Find the difference between PCA and Exploratory Factor Analysis (EFA). Try to perform factor analysis using different methods such as gls, wls, mle etc.

6.6 Case

Methods dealing with only one variable are called univariate methods. Methods dealing with more than one variable at once are called multivariate methods. Using univariate methods natural systems cannot be described satisfactorily. Nature is multivariate. That means that any particular phenomenon studied in detail usually depends on several factors.

The main objectives of multivariate data analysis are exploratory data analysis, classification and parameter prediction. One of the reports published by World Bank mentions that global waste generation could increase 70% by 2050. Currently, about 2.01 billion metric tons of municipal solid waste (MSW) are produced annually worldwide. An estimated 13.5% of today's waste is recycled and 5.5% is composted. The report estimates that between one-third and 40% of waste generated worldwide is not managed properly and instead dumped or openly burned. These are all facts generated from descriptive analysis.

Collect multivariate data sets online related to global waste. Investigate hazards of growing waste on the globe, using multivariate data analysis.

Index