



Community Experience Distilled

What you need to know about R

Kick-start your journey with R

Dipanjan Sarkar
Raghav Bali

[PACKT]
PUBLISHING

What you need to know about R

Kick-start your journey with R

Raghav Bali
Dipanjan Sarkar



BIRMINGHAM - MUMBAI

What you need to know about R

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First Published: June 2016

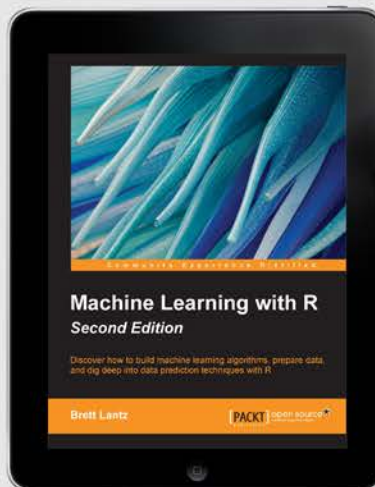
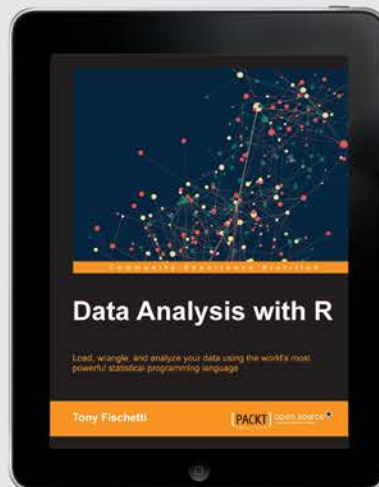
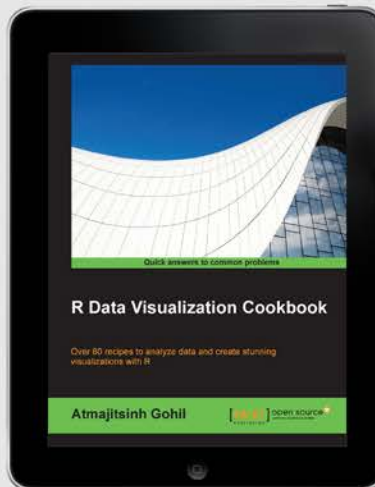
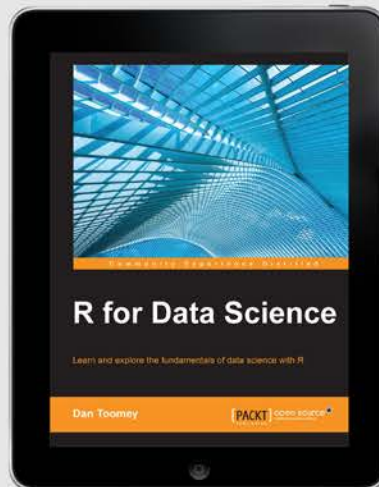
Production reference: 1060616

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

www.packtpub.com

Get
50%
Off

Your next eBook or Video



Use the following code to
apply your exclusive discount

R50

About the Authors

Raghav Bali has a master's degree (gold medalist) in Information Technology from International Institute of Information Technology, Bangalore. He is an IT engineer at Intel, the world's largest silicon company, where he works on analytics, business intelligence, and application development to develop scalable machine learning-based solutions. He has worked as an analyst and developer on domains, such as ERP, Finance, and BI with some of the top companies of the world.

Raghav is a technology enthusiast who loves reading and playing around with new gadgets and technologies. He recently co-authored a book on Machine Learning titled *R Machine Learning by Example*, Packt Publishing. He is a shutterbug, capturing moments when he isn't busy solving problems.

I would like to express my gratitude to my family, teachers, and friends, who have encouraged, supported and taught me over the years. Special thanks to my classmate, friend and colleague, Dipanjan Sarkar, who co-authored this book and made this journey wonderful through his inputs and eye for detail.

I would like to thank Tushar Gupta, Merint Mathew, and Packt Publishing for the opportunity and their support throughout this journey. Last but not the least, thanks to the R community for the amazing stuff that they do!

Dipanjan Sarkar is an IT engineer at Intel, the world's largest silicon company, which is on a mission to make the world more connected and productive. He primarily works on analytics, business intelligence, application development, and building large scale machine learning systems. He received his master's degree in Information Technology from the International Institute of Information Technology, Bangalore. His area of specialization includes software engineering, data science, machine learning, and text analytics.

Dipanjan's interests include learning about new technology, disruptive start-ups, data science, and more recently deep learning. In his spare time, he loves reading, writing, gaming, and watching popular sitcoms. He co-authored a book on Machine Learning titled *R Machine Learning by Example*, Packt Publishing, and he also acted as a technical reviewer for several books on Machine Learning and Data Science from Packt Publishing.

I am indebted to my family, friends, teachers, and colleagues for always standing by my side and supporting me in all my endeavors. Your support keeps me going day in and day out to take on new challenges! I would also like to thank my good friend and fellow colleague, Raghav Bali, who co-authored this book and made this experience more enjoyable. Last, but never the least, I would like to thank Tushar Gupta, Merint Mathew, and Packt Publishing for giving me this wonderful opportunity to share my knowledge with the machine learning, and R enthusiasts out there who are doing truly amazing things every day.

www.PacktPub.com

Support files, eBooks, discount offers, and more

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books, eBooks, and videos.



<https://www.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

R Ecosystem	1
Setting up the R ecosystem	1
Installation	1
Configuration	2
Startup modes	2
Workspace	3
Exploring the basic constructs of R	3
Operators	3
Data types	4
Data structures	5
Installing packages	5
Getting help	6
Integrated Development Environments	6
RStudio	6
Other IDEs	7
RPubs – Publishing through R	8
Shiny – Web apps using R	10
Data Analysis	11
Data analysis workflow	11
Understanding our current objective	14
Acquiring and understanding data	14
Preparing the data	17
Exploratory data analysis	18
Statistical inference	31
Statistical modeling with regression	34

R Cheat Sheets	40
Data processing and transformation	40
Data handling	40
Basic data types	41
Data structures	41
General utilities	43
Math and modeling	44
Math and modeling utilities	44
Math and modeling packages	45
Plotting	46
Plotting packages	47
Summary	47
What to do next?	48
Broaden your horizons with Packt	48

What you need to know about R

This eGuide is designed to act as a brief, practical introduction to R. It is full of practical examples which will get you up a running quickly with the core tasks of R.

We assume that you know a bit about what R is, what it does, and why you want to use it, so this eGuide won't give you a history lesson in the background of R. What this eGuide will give you, however, is a greater understanding of the key basics of R so that you have a good idea of how to advance after you've read the guide. We can then point you in the right direction of what to learn next after giving you the basic knowledge to do so.

What you need to know about R will:

- Cover the fundamentals and the things you really need to know, rather than niche or specialized areas.
- Assume that you come from a fairly technical background and so understand what the technology is and what it broadly does.
- Focus on what things are and how they work.
- Include practical examples to get you up, running, and productive quickly.

Overview

R is a scripting language that is aimed at performing statistical analysis. It draws inspiration from S, a statistical programming language that was developed by AT&T. It also provides a multitude of options, tools, and libraries to make statistical analysis easy and effective. R has grown over the years as a result of its open source nature. It is a community-driven language that provides powerful tools for data processing, manipulation, visualization, and publishing. It continues to evolve with an ever-increasing list of packages and libraries, along with constant improvements to the overall language.

Statistical analysis was the reason for R's inception and it has grown both in importance and functionality over the years to become a go-to language. Data scientists and statisticians alike use it to quickly prototype as well as to build complex models and analysis. R finds applications for financial analysis and modeling, food and drug data analysis, clinical trial analysis, and so on.

R is maintained by the **Comprehensive R Archival Network** (which is better known as **CRAN**). CRAN maintains the latest and past version binaries and the source code for R for most OS platforms, such as Windows, Mac, and Linux. As mentioned earlier, R is a free and open source platform. Due to its increased popularity and capabilities, commercial versions of R are also available with enterprise support. R is also being enhanced and tweaked to make it work with Big Data technologies, such as **Hadoop** with offerings available from Oracle, IBM, and so on. Platforms such as **Mathematica**, **MATLAB**, **SPSS**, and others offer connection capabilities to R as well.

In this guide, we will go through the following topics:

- A quick introduction to the R ecosystem
- Understanding its capabilities and working on a hands-on example using R as a data analysis tool
- A cheat sheet for neat tricks and to provide a quick reference guide to enhance the powers of R

R Ecosystem

This is an introductory section, and it will get you started with the basics of R along with its ecosystem. It will also prepare you for some exciting features and examples in the coming section. In this section, we will cover the following topics:

- **R**: Getting started and basic constructs
- **RStudio**: The de facto development environment for R
- **RPubs**: Publish straight from R and share your work with ease
- **R Shiny**: Develop web applications using the power of R

In this book, we will consider Windows as the default working environment. Setting up and using R on other platforms is also quite easy and similar to the Windows platform. You need to follow the standard guidelines or documentation for platform-specific issues.

Setting up the R ecosystem

In this section, we will set up R on our systems. We will also touch upon different modes and settings to tweak this powerful tool. Let's begin with the installation of R itself.

Installation

R is an open source and free software environment (and an interpreted language) that is available for all major operating systems, such as UNIX and LINUX, Windows, and OS X. As of writing, the current version is 3.2.5 (code named **Very Very Secure Dishes**), and it is available at <https://www.r-project.org/>.



The chronology and evolution of R:

The following link presents the history and evolution of R along with important features and versions clearly marked out:

http://timelyportfolio.github.io/rCharts_timeline_r/

R's setup is straightforward and nicely outlined at the preceding link. For the Windows environment, the setup requires downloading the setup file and following the instructions from the executable. For Unix and Unix-like environments, R can be installed from the prompt directly (wherever Unix-like binaries are available). Enthusiasts can also build R from source as outlined in the steps mentioned in the FAQ section of the *r-project*.

Configuration

R can be configured in a simple way to personalize startup. A file named `Rprofile.site` exists in the installation directory. This simple R script file is checked each time R is loaded into memory; hence, it executes any instructions (for instance, functions, default directories, and so on) that are mentioned in this file. Another level of customization can be achieved, where each user of the system can personalize R's startup by adding a file, named `Rprofile`, to their home directory.

Startup modes

R supports the two following execution modes:

- **Interactive Mode:** R is a scripting language, and it is interpreted line by line. Similarly to other scripting languages, R has an interactive mode, which provides a `>` prompt, which we can use to execute commands directly.
- **Batch Mode:** For scenarios where interactive output isn't required and/or R works in an automated manner to produce certain results, the batch mode comes into play. In this mode, R scripts are invoked from the command prompt (or the OS shell prompt).

Workspace

R has robust memory management, where it allocates and keeps track of all the objects in the environment. R's workspace is nothing but the current working environment, which holds user-defined objects, such as variables, data, functions, and so on. R provides various utility functions to manipulate the workspace.

Some of the standard utilities are as follows:

- `ls()`: This lists all objects in the workspace
- `rm()`: This removes mentioned objects from the workspace
- `getwd()`: This prints the current working directory
- `setwd()`: This sets a directory as the current working directory
- `history()`: This prints all commands that have executed since the start of the session
- `save.image()`: This saves the current workspace in a `.RData` file for future use

There are many other utility functions that are available. Readers are urged to explore them using the `help()` command and R's documentation.

Exploring the basic constructs of R

Every programming or scripting language has certain syntax and other constructs that make it unique yet powerful. The following are the major building blocks of the R programming language.

Operators

Programming languages require operators to perform actions, such as calculations, transformations, and so on. Similar to other programming languages, R supports operators for *Logical*, *Mathematical*, and *Conditional* operations, and so on. As R was designed keeping statistical analysis in mind, the operators are robust enough to handle not just basic data types, such as *integers*, *floats*, and *characters*, but they can also handle *matrices*, *vectors*, *strings*, and *arrays*.

The only difference, as compared to other languages, is the assignment operator. R provides the following different assignment operators:

- `=`: This is used interchangeably with the `<-` operator for assignment; R's standards reserve the usage of the `=` operator for parameter passing only. This operator assigns a value to parameters of a function without creating a variable in the user's workspace. The following is an example of this:

```
> foo(x=1)
> foo_1(y<-2)
> ls()
```

In this example, `y` would be present in the list of variables but `x` will not be!

- `<-` or `->`: This is the default assignment operator that is used in R. This is part of R's lineage from the days of S. This operator works both sides as the following example states:

```
> x <- 3
> 15 -> y
```

In this snippet, `x` is assigned a value of 3, while `y` gets a value of 15 assigned to it.

There's even an `<<-` operator to access objects in the parent scope. Readers are encouraged to go through R's documentation for more details.

Data types

To enable statistical analysis, R supports all basic data types, such as `numeric` (integer, double), `character`, `boolean`, and so on. Apart from these basic data types, R also provides a data type called `factor` for categorical data and `complex` for the storage of complex numbers. This also handles missing values and nonexistent objects differently, using the `NA` and `NULL` keywords, respectively. You should not confuse `NA` with `NaN`, where `NA` denotes missing values, while `NaN` (**NaN** stands for **not a number** and is a keyword in R) is used to represent undefined or unrepresentable values. This also handles infinity using the `Inf` keyword (`-Inf` for negative infinity). Each of these data types has a number of utility functions to check missing values, length, and so on. A common set of functions for each data type are `as` and `is`, which help in typecasting and checking the data type, respectively. For instance, `as.character()` typecasts the input as a character, while `is.character()` is used to check whether the input is of the character type or not.

Data structures

Data structures are at the core of R, and they provide a very powerful foundation to the language. Any object or variable is `vector` (as in the mathematical vector) by default unless specified otherwise. Lists, arrays (n-dimensional), matrices, and so on are available out of the box. Lists are recursive in nature, that is, they can contain other lists as elements, while vectors, arrays, and matrices are atomic in nature. R also provides a unique tabular data structure called **data frames**. Data frames represent a two-dimensional structure just like a matrix. However, unlike matrices, a data frame can have different columns containing different data types. All the components of a data frame must be of equal length. Consider the following example:

```
>book.sections<- c("section 1", "section 2", "section 3")
>section.pages<- c(6,26,10)
```

```
>dataFrame<- data.frame(book.sections, section.pages,
stringsAsFactors=FALSE)
```

```
>dataFrame #print the data frame
```

We will observe the following output:

	book.sections	section.pages
1	section 1	6
2	section 2	26
3	section 3	10

More on each of these data structures are covered in the upcoming sections.

Installing packages

As mentioned earlier, R is a community-driven language, and it owes its immense power to an ever-increasing list of packages that add on to the capabilities of the platform. In the R community, the term *library* is used instead of the term *package*. R provides the following utilities to handle packages (and many more):

- `install.packages(<package_name>, [<library_path>])`: This installs a package from **Comprehensive R Archive Network (CRAN)**. CRAN helps maintain and distribute R's various versions and documentation.
- `libPaths(<library_path>)`: This adds this library path to R.
- `installed.packages(lib.loc = <library_path>)`: This lists installed packages.

- `update.packages(lib.loc = <library_path>)`: This updates a package.
- `remove.packages (<package_name>)`: This removes a package.
- `path.package ()`: This is the package loaded for the session.
- `library (<package_name>)`: This loads a package in a script to use its functions or utilities.
- `library (help=<package_name>)`: This lists the functions in a package.

Getting help

It is very quick and simple to check documentation or get help related to R, its packages, or utilities. The following utilities are available to get help:

- `help (<any_R_object>)` or `?<any_R_object>`: This provides help on any R object, such as functions, packages, data types, and so on
- `example (<function_name>)`: This provides a quick example for the mentioned function
- `apropos (<any_string>)`: This lists all functions containing the `any_string` term

Integrated Development Environments

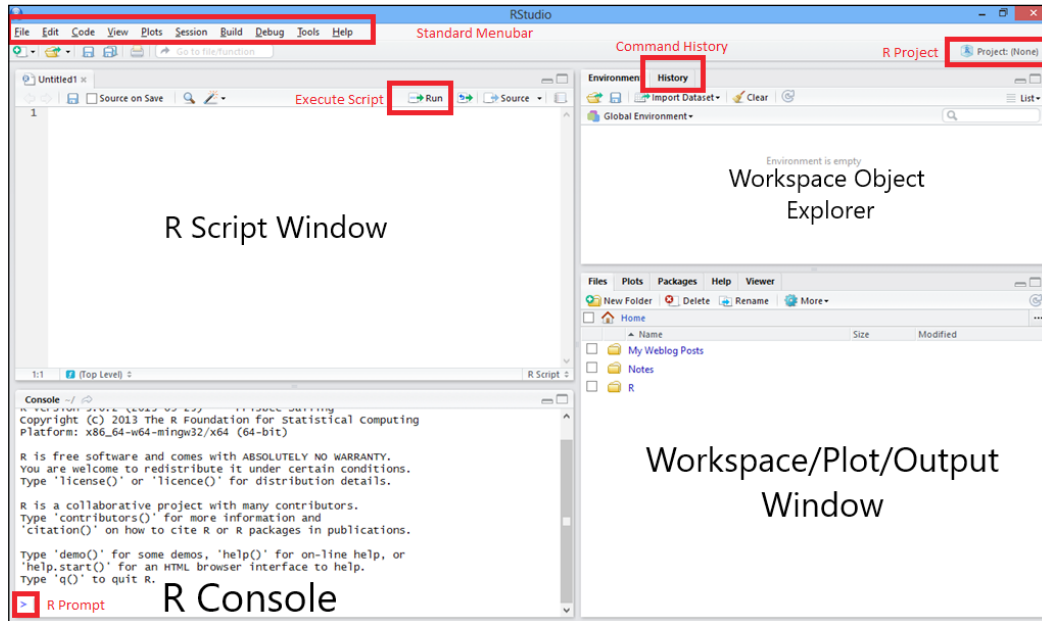
For ease of development and code maintenance, IDEs are available for all programming languages. R comes bundled with a standard interface called the **RGui**. R works seamlessly with multiple IDEs though the most popular and widely-used is the **RStudio**.

RStudio

RStudio provides various features; some of them are as follows:

- Code highlighting and completion
- Multiple windows to write, execute, view objects, graphs, and so on in one place
- A help browser and package handler

The following screenshot displays RStudio's standard interface with different windows, menus, and icons flagged:



The RStudio standard interface

Other IDEs


Apart from RStudio, various other IDEs are also available:

- **RCommander:** <http://www.rcommander.com/>
- **Eclipse R StatET:** <http://www.walware.de/goto/statet>
- **ESS or Emacs Speaks Stats:** <http://ess.r-project.org/>

There are many more specialized and fully-loaded R IDEs. Similar to Eclipse, Visual Studio also provides R and R-related plugins for .NET developers. Use the one that suits you the best. For the purpose of this book, we will stick with RStudio.

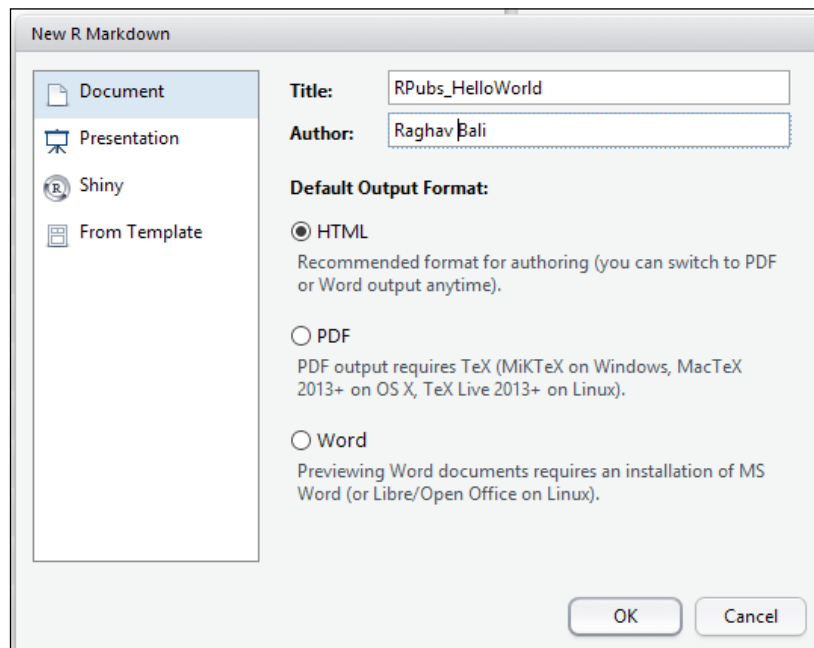
RPubs – Publishing through R

RPubs is a free publishing service from RStudio. Using RPubs, users can share their research, code, and analysis written in R Markdown straight from RStudio itself. Once published, the research work is publicly available for anybody to go through, comment on, and share. This is a step towards reproducible research and sharing knowledge.

 **RMarkdown** is a derivation or the markdown tool for text to HTML conversion. RMarkdown uses the syntax of markdown and adds on additional features to easily integrate and regenerate R code to publish it. More details can be found at <http://rmarkdown.rstudio.com/>.

The following is a quick introduction to RPubs:

From RStudio, go to the **File Menu**, select **New File** as **RMarkdown**. Then, choose the output format from the dialog box and provide a title and author name:



RMarkdown format selection

Add content in standard Markdown syntax in the script pane or window, and click on the **Knit HTML** icon:

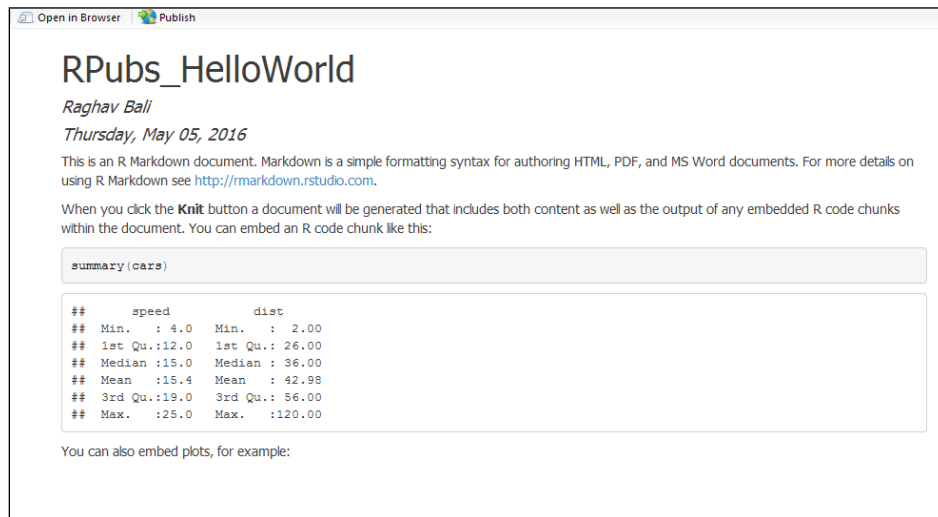
```

1  ---
2  title: "RPubs_HelloWorld"
3  author: "Raghav Bali"
4  date: "Thursday, May 05, 2016"
5  output: html_document
6  ---
7
8  This is an R Markdown document. Markdown is a simple formatting syntax for
9  authoring HTML, PDF, and MS Word documents. For more details on using R Markdown
10 see <http://rmarkdown.rstudio.com>.
11
12 When you click the Knit button a document will be generated that includes both
13 content as well as the output of any embedded R code chunks within the document.
14 You can embed an R code chunk like this:
15
16 ```{r}
17 summary(cars)
18 ```
19
20 You can also embed plots, for example:
21
22 ```{r, echo=FALSE}
23 plot(cars)
24 ```

```

RMarkdown sample script

The markdown script is processed, and a preview window pops up. You can publish to the Web directly from this pop-up by clicking on the **Publish** button:



RMarkdown sample preview

RPubs works without RStudio as well. This requires the installation of packages such as `knitr`, `rmarkdown`, and so on.

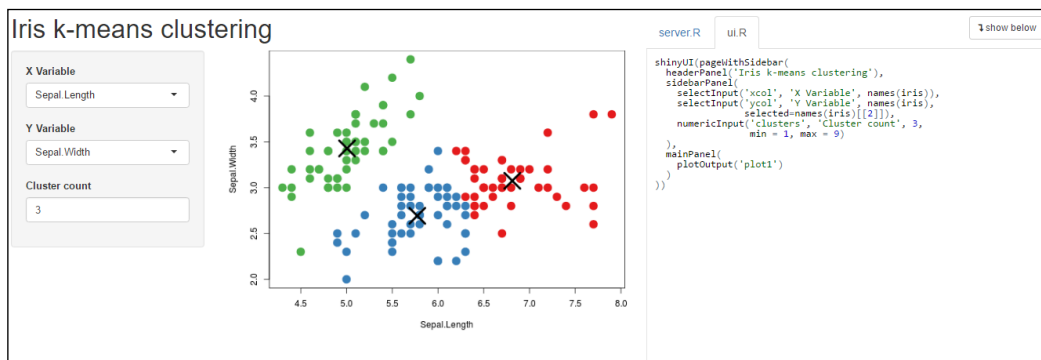
Shiny – Web apps using R

In a world that is leaning more and more towards the Internet, R is no exception. **R Shiny** is a web application framework from RStudio that enables developers to create web applications using just R without detailed knowledge of HTML, CSS, or other web technologies. A simple Shiny app requires just two components:

- `ui.R`: This script contains instructions to render the web application in a browser-like environment
- `server.R`: This is where the actual processing or analysis happens

R Shiny requires the `shiny` package to be added to the list of packages. The documentation for R Shiny is fairly detailed and easy to understand. Refer to the tutorial and examples mentioned at the official website at <http://shiny.rstudio.com/tutorial/>.

The following screenshot displays a sample Shiny app with its rendered output and code side by side:



A sample R Shiny web application (<http://shiny.rstudio.com/gallery/kmeans-example.html>)

Data Analysis

In the previous section, you got a quick glance into the entire ecosystem of tools and frameworks that R offers to analyze data and present your findings in various ways, including reproducible Markdown documents as well as web applications. R is a programming language at heart; it is also a software environment, was primarily built for the statistical analysis of data leveraging a wide variety of statistical techniques and graphical methods to visualize results. In this section, we will look at what a typical data analysis workflow looks like, and then we will analyze a real dataset using exploratory and statistical analysis techniques.

 R has an annual international conference, named **useR**. This is an international event to discuss and present the advancements, applications, and issues that are related to R and general statistical topics.

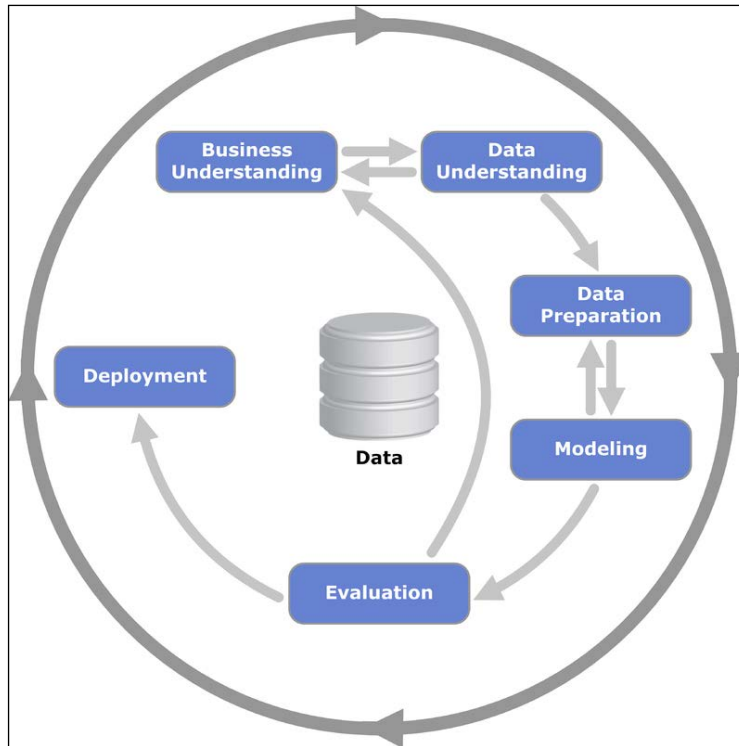
Data analysis workflow

Analyzing data is not only an art, but it is also a science. It has a defined set of steps, which are usually executed in sequence, and several steps are often repeated if they are necessary. There is an industry standard that is widely followed for data analysis, known as **CRISP-DM**, which expands to **Cross Industry Standard Process for Data Mining**. This is a standard data analysis and mining process workflow that describes how to break up any particular data analysis problem into six major stages.

The main stages in the CRISP-DM model are as follows:

- **Business Understanding:** This is the initial stage that focuses on the business context of the problem that has to be solved at hand and uses domain and business knowledge to plan out the main objectives and results that are intended from the data analysis workflow.
- **Data Acquisition and Understanding:** This stage's main focus is to acquire data of interest and understand the meaning and semantics of the various data points and attributes that are present in the data. Some initial exploration of the data may also be done at this stage.
- **Data Preparation:** This stage usually involves data munging, cleaning, and transformation. Data quality issues are also dealt with in this stage. The final dataset is usually used for analysis and modeling.
- **Modeling and Analysis:** This stage mainly focuses on analyzing the data and building models using specific techniques. Often, we need to apply further data transformations that are based on different modeling algorithms.
- **Evaluation:** This is perhaps one of the most crucial stages. Building models and analyzing the data for patterns and insights are not the end of the analysis. In this stage, we evaluate the results that are obtained from different techniques and iterations, and then we select the best possible method or analysis, which gives us the insights that we need based on our business requirements. Often, this stage involves reiterating through the previous two steps to reach a final agreement based on the results.
- **Deployment:** This is the final model where decision systems that are based on analysis are deployed so that end users can start consuming the results and utilizing them. This deployed system can be as complex as a real-time prediction system or as simple as an ad-hoc report.

The following figure shows the relationship between the various stages in the CRISP-DM model:



The relationship between the different stages in CRISP-DM. Source: www.wikipedia.org

In principle, the CRISP-DM model is very clear and concise, and this makes it easy for data science practitioners and analysts to follow in their daily processes. We will look at a real dataset in the next section, and apply some of these principles in our data analysis process to get valuable insights from analyzing data.

Understanding our current objective

Before we start using R and dive into the implementations of analyzing our data, we will first talk about what data we are going to analyze and what our main objectives are in this analysis. The reason that we do this is so that we do not lose focus when applying data analysis techniques to gather insights from our dataset.

Our main objective is to explore and analyze the `mtcars` dataset, which is readily available in R. This dataset contains data about several automobiles, specifically cars and various attributes that are related to each car. Some of our main objectives are listed, as follows:

- Understanding various features in the dataset
- Exploring the relationships between different features
- Visualizing insights using different charts and graphs
- Performing statistical tests to gain specific insights on features that are related to the vehicles
- Building regression models to view the relationship between different features and the mileage of cars
- Understanding and exploring specific concepts in regression modeling, evaluation, and predictions

Next, we will focus on getting our dataset and understanding the semantics of each attribute in the dataset and what they indicate.

Acquiring and understanding data

We will start with getting the necessary dataset that we want to analyze. The dataset that we will look at is called `mtcars`, and is available directly in R without the need to access any specific website or database. Fire up your R console, and type the following command to load the dataset into memory:

```
# load the dataset
data(mtcars)
```

Next, we will inspect some details about the dataset, which can be done using the `str` command, as follows:

```
# see details about the dataset
str(mtcars)
```

This gives us the following output, which tells us the various attributes in the dataset and gives us a quick peek at their values.

```
'data.frame':  32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num   6  6  4  6  8  6  8  4  4  6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num   3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num   2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num   0  0  1  1  0  1  0  1  1  1 ...
 $ am  : num   1  1  1  0  0  0  0  0  0  0 ...
 $ gear: num   4  4  4  3  3  3  3  4  4  4 ...
 $ carb: num   4  4  1  1  2  1  4  2  2  4 ...
```

We observe that the dataset is stored in a data structure of the `data.frame` type, which is basically a two-dimensional tabular structure that is similar to a spreadsheet, where each row is a particular data point that consists of different attributes that are represented by different columns. In our dataset, we have 32 observations or data points that form the rows and 11 attributes that form the columns. Each data point or row is for a particular car, and each column is a specific attribute that is related to the car, such as `mpg`, which indicates the Miles per Gallon of this car. We will now understand the data in more detail by accessing the dataset metadata information using the `help` command, as follows:

```
# detailed information about the dataset
help(mtcars)
```

This command displays detailed information regarding the data, which was originally extracted from the 1974 issue of Motor Trend US magazine. This data has information for a total of 32 cars. Each car is described by a total of 11 attributes, which are described in detail in the following snapshot. They are pretty self-explanatory except perhaps the `vs` attribute, which actually indicates whether the car has a V-engine or a Straight engine:

Description The data was extracted from the 1974 <i>Motor Trend</i> US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).
Usage <code>mtcars</code>
Format A data frame with 32 observations on 11 variables. [1] mpg Miles/(US) gallon [2] cyl Number of cylinders [3] disp Displacement (cu.in.) [4] hp Gross horsepower [5] drat Rear axle ratio [6] wt Weight (1000 lbs) [7] qsec 1/4 mile time [8] vs V/S [9] am Transmission (0 = automatic, 1 = manual) [10] gear Number of forward gears [11] carb Number of carburetors

We will now look at what the actual data looks like in the dataset using the following command:

```
# view the raw data  
head(mtcars, 5)
```

This command shows us the top five rows in the dataset, which are shown in the following snapshot:

	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

Now that we have a good understanding of the data and what it looks like, we will proceed to the next step of data preparation before analyzing it.

Preparing the data

Preparing the data usually involves imputation or filling in missing values (often denoted by `NA` in R) and then performing data transformation, scaling, or data type conversions as needed. Luckily for us, our data is quite clean and does not have any missing values.

We will focus on datatype conversions.

If you closely observe the attribute data types from `str(mtcars)`, which we executed earlier, you will see that each attribute has been declared as `num`, which is a numeric type, by R. However, in reality several variables are not of the numeric type, and we have to change this based on the variable semantics and values. If you have taken a basic course on statistics, you might know that usually we deal with two types of variable or attribute most of the time:

- **Numeric variables:** The values of these variables carry some mathematical meaning. This enables you to carry out mathematical operations on them, such as addition, subtraction, and so on. Some examples from our dataset are `mpg`, `disp`, `wt`, and so on.
- **Categorical variables:** The values of these variables do not have any mathematical significance, and performing mathematical operations on them does not make sense. Each value in this variable belongs to a specific class or category. Some examples from our dataset are `cyl`, `vs`, `am`, and so on.

As all the variables or attributes in our dataset were converted to numeric by default, we will only need to convert the categorical variables from numeric data types to factors, which is how R represents categorical attributes.

We will first implement our own utility function to carry out this data type conversion using the following code snippet. A function is basically a block of code that usually takes some input, performs some operations, and may or may not return an output:

The `##` data type conversion to factors is as follows:

```
to.factors<- function(df, variables){
  for (variable in variables) {
    df[[variable]] <- as.factor(df[[variable]])
  }
  return(df)
}
```

We will use this function on our existing `mtcars` data frame to transform the `cyl`, `vs`, `am`, `gear`, and `carb` attributes into categorical attributes using the following code snippet:

```
## perform data type transformation
categorical.vars<- c("cyl", "vs", "am", "gear", "carb")
mtcars<- to.factors(mtcars, categorical.vars)
```

Now, we will observe whether this data type transformation was successful using the following snippet:

```
# verify transformation
str(mtcars)
```

We can then see the attribute details in the data frame with the transformed data types in the following snapshot, which indicates that our transformations were successful:

```
'data.frame':  32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : Factor w/ 3 levels "4","6","8": 2 2 1 2 3 2 3 1 1 2 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : Factor w/ 2 levels "0","1": 1 1 2 2 1 2 1 2 2 2 ...
 $ am  : Factor w/ 2 levels "0","1": 2 2 2 1 1 1 1 1 1 1 ...
 $ gear: Factor w/ 3 levels "3","4","5": 2 2 2 1 1 1 1 2 2 2 ...
 $ carb: Factor w/ 6 levels "1","2","3","4",...: 4 4 1 1 2 1 4 2 2 4 ...
```

This brings us to the end of our data preparation stage, and we will now perform some analysis on our dataset in the next section.

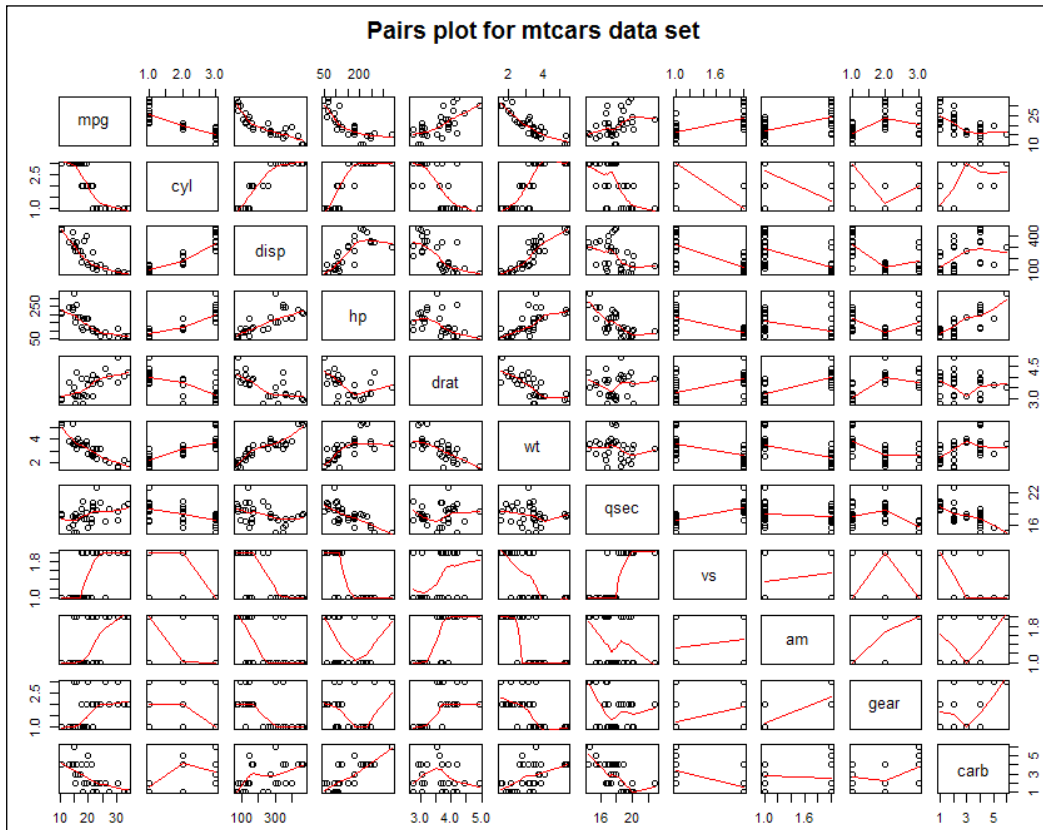
Exploratory data analysis

There are various data analysis techniques that can be applied to a dataset, depending on the problem that has to be solved and the insights we want to gather. However, in all cases, exploratory data analysis is somewhat of a prerequisite before jumping into further advanced analyses. Exploratory data analysis is a good way to gain a deeper understanding of our data, relationships, patterns between different attributes, and to detect anomalies. Besides descriptive analysis, which includes generating summary statistics, we also use visualization techniques to depict various patterns and statistics about the data, which help us in understanding our data better. We will use some graphical methods here to visualize various statistics that are related to our dataset.

One basic visualization includes scatter plots where we usually have an attribute on the x and y axes, and we plot the various data points in the two-dimensional space to see the relationship between the attributes. We will plot a pairs scatterplot between all possible attributes in our `mtcars` dataset with the following code snippet:

```
# pairs plot observing relationships between variables
pairs(mtcars, panel = panel.smooth,
main = "Pairs plot for mtcars data set")
```

This gives us the following scatterplot, which shows the relationship between each pair of attributes in the dataset:

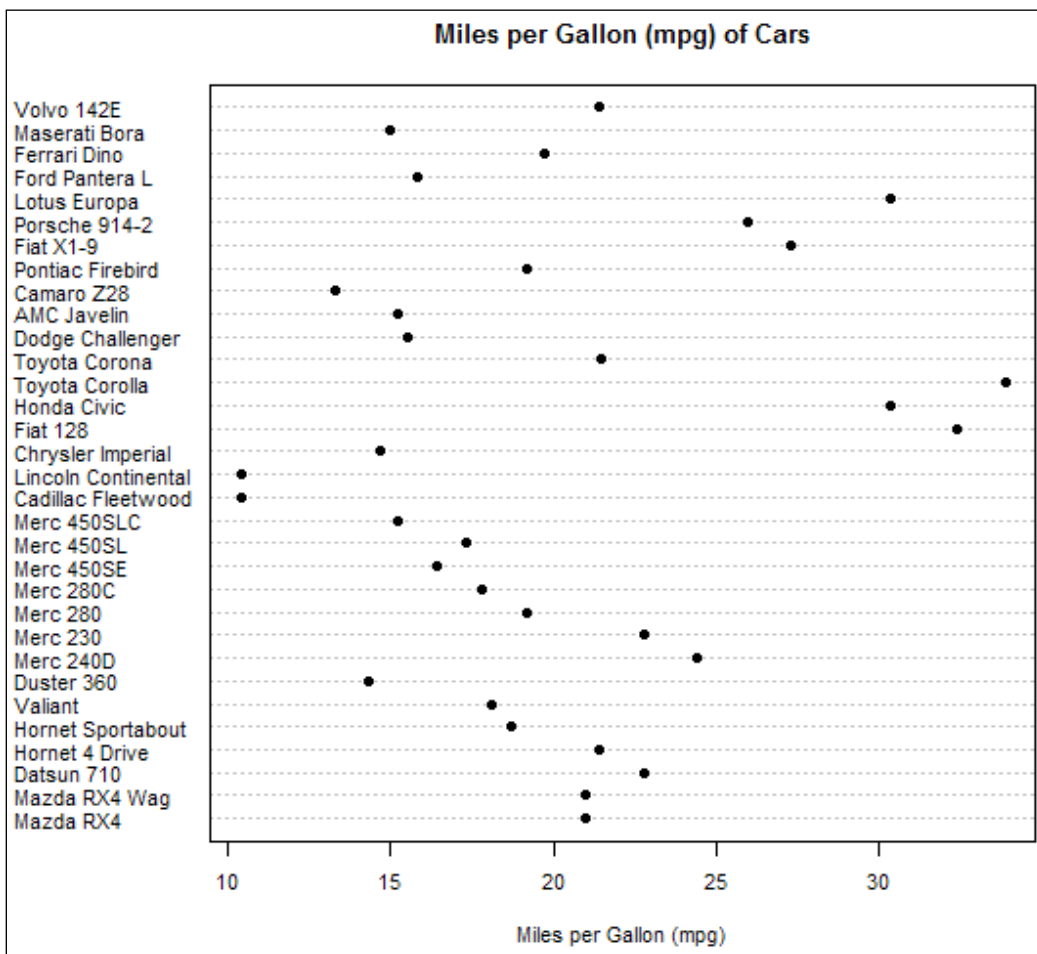


A pairs scatterplot between different attributes of the mtcars dataset

Next, we will leverage the use of a dot chart to plot the Miles per Gallon (mpg) value for all the cars in our dataset using the following code snippet:

```
# mpg of cars
dotchart(mtcars$mpg, labels=row.names(mtcars),
cex=0.7, pch=16,
main="Miles per Gallon (mpg) of Cars",
xlab = "Miles per Gallon (mpg)")
```

This gives us a dot plot of the mpg values of each car, as shown in the following figure:



Some interesting insights that we see from the previous chart is that the top two cars with the least mpg are Lincoln Continental and Cadillac Fleetwood. Similarly, the top two cars with the highest mpg are Toyota Corolla and Fiat 128. We can also compute this using R to prove our observations using the following code snippets, where we use the `order` function to sort the mpg values before filtering out the necessary data:

```
head(mtcars[order(mtcars$mpg),], 2)
```

This gives us the top two cars with the least Miles per Gallon, as seen in the following snapshot:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Cadillac Fleetwood	10.4	8	472	205	2.93	5.250	17.98	0	Automatic	3	4
Lincoln Continental	10.4	8	460	215	3.00	5.424	17.82	0	Automatic	3	4

To get the top two cars with the maximum Miles per Gallon, we use the following code snippet:

```
tail(mtcars[order(mtcars$mpg),], 2)
```

This gives us the following output:

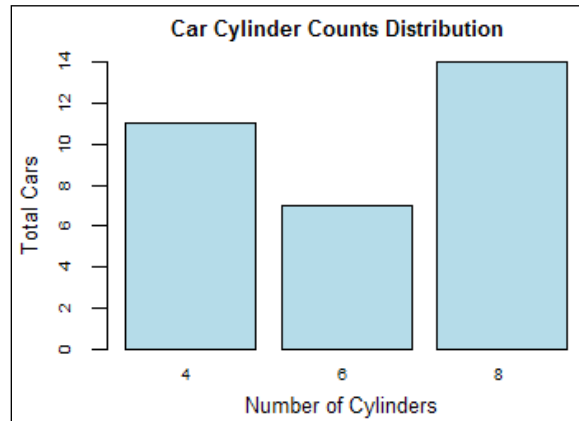
	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	Manual	4	1
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	Manual	4	1

Thus, we see that our observations from the visualization were correct, and we got the same results from our R code snippets.

We will now plot some simple bar charts for car frequencies that are related to several attributes in the dataset. The next plot shows us the car counts grouped by cylinders (`cyl`) using the following code. You can look at further details, such as graph positioning and alignment, which we perform using the `cex` parameters by checking the documentation using the `?barplot` command:

```
# cylinder counts
barplot(table(mtcars$cyl),
col="lightblue",
main="Car Cylinder Counts Distribution",
xlab="Number of Cylinders", ylab="Total Cars",
cex.main = 0.8, cex.axis=0.6,
cex.names=0.6, cex.lab=0.8)
```

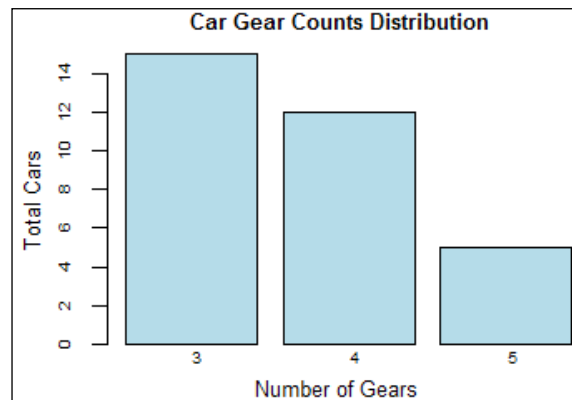
This gives us the following bar chart:



We observe that cars with eight cylinders are the most numerous followed by cars with four cylinders. Next, we plot a similar bar plot of car counts that are grouped by gear using the following code:

```
# gear counts
barplot(table(mtcars$gear),
col="lightblue",
main="Car Gear Counts Distribution",
xlab="Number of Gears", ylab="Total Cars",
cex.main = 0.8, cex.axis=0.6,
cex.names=0.6, cex.lab=0.8)
```

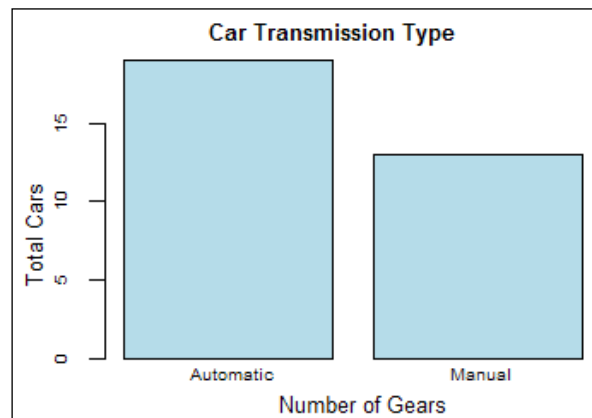
This gives us the following bar plot, where we observe cars with three gears are most numerous:



The last simple bar chart will depict car counts that are grouped by the type of transmission. For this, we relabel the factor variable levels from 0 and 1 to Automatic and Manual first, and then we plot the chart, as shown in the following code:

```
# transmission counts
mtcars$am<- factor(mtcars$am,labels=c('Automatic','Manual'))
barplot(table(mtcars$am),
col="lightblue",
main="Car Transmission Type",
xlab="Number of Gears", ylab="Total Cars",
cex.main = 0.8, cex.axis=0.6,
cex.names=0.6, cex.lab=0.8)
```

This gives us the following plot, which clearly depicts that there are more cars with automatic transmission in our dataset:

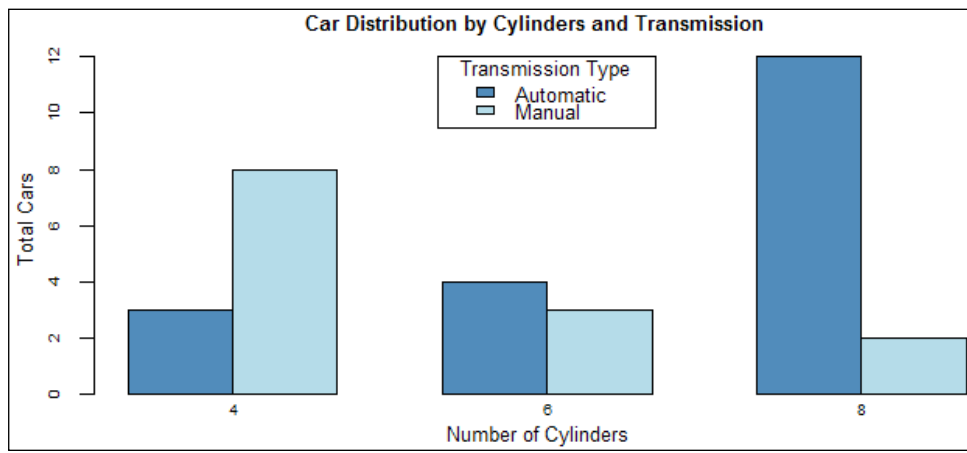


We will now visualize the data using some more complex visualizations in this segment. To start off, let's visualize the car distribution by cylinders as well as transmission using the following code snippet:

```
# visualizing cars distribution by cylinders and transmission
counts<- table(mtcars$am, mtcars$cyl)
barplot(counts, main="Car Distribution by Cylinders and Transmission",
xlab="Number of Cylinders", ylab="Total Cars",
col=c("steelblue","lightblue"),
legend=rownames(counts), beside=TRUE,
```

```
args.legend=list(x="top", title="Transmission Type",  
cex=0.8),  
cex.main = 0.8, cex.axis=0.6,  
cex.names=0.6, cex.lab=0.8)
```

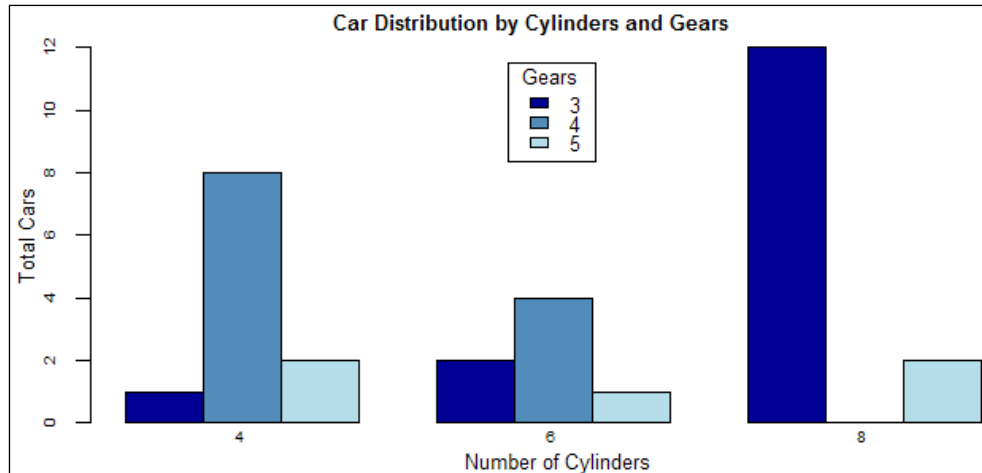
This gives us the following plot, where we see that most cars with automatic transmission have eight cylinders and most cars with manual transmission have four cylinders:



In the next visualization, we observe car distributions by cylinders as well as gears. The following code snippet enables us to visualize this:

```
counts<- table(mtcars$gear, mtcars$cyl)  
barplot(counts, main="Car Distribution by Cylinders and Gears",  
xlab="Number of Cylinders", ylab="Total Cars",  
col=c("darkblue", "steelblue","lightblue"),  
legend=rownames(counts), beside=TRUE,  
args.legend=list(x="top", title="Gears", cex=0.8),  
cex.main = 0.8, cex.axis=0.6,  
cex.names=0.6, cex.lab=0.8)
```

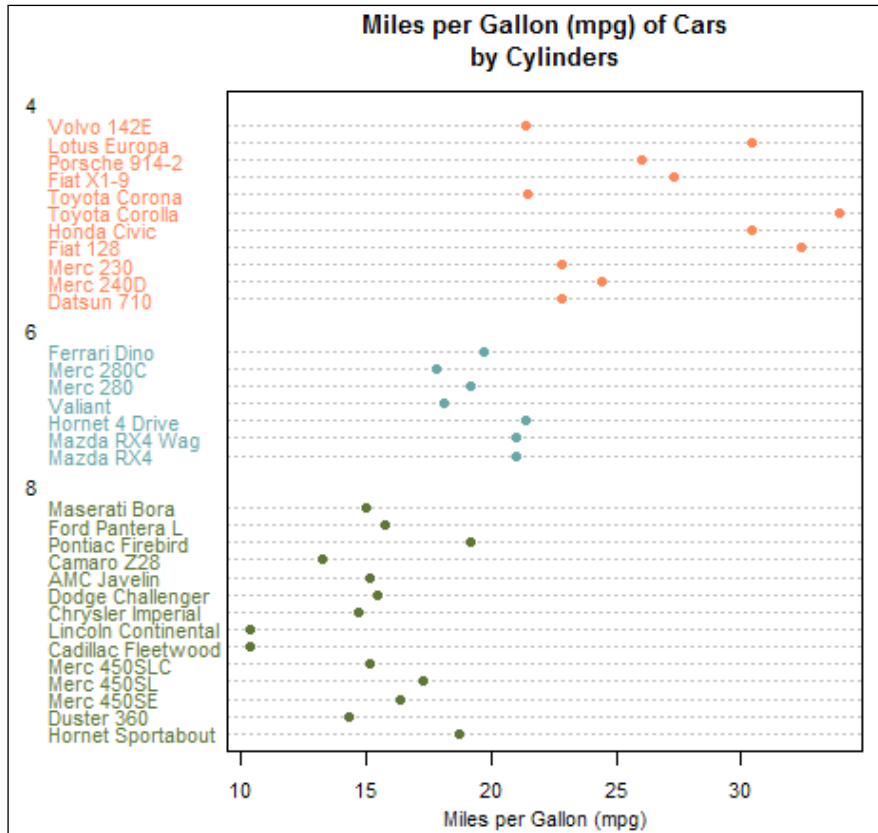
This gives us the following plot where we observe most cars with three gears have eight cylinders:



Now, we will create a grouped dot plot showing the miles per gallon of various cars that are grouped by number of cylinders using the following code snippet:

```
# visualizing car mpg distribution by cylinder
# add a color column within the data frame for plotting
mtcars<- within(mtcars, {
  color<- ifelse(cyl == 4, "coral", ifelse(cyl == 6,
    "cadetblue", "darkolivegreen"))
})
dotchart(mtcars$mpg, labels=row.names(mtcars),
groups=mtcars$cyl,
color=mtcars$color,
cex=0.7, pch=16,
main="Miles per Gallon (mpg) of Cars\nby Cylinders",
xlab = "Miles per Gallon (mpg)")
# remove the color column within the data frame after plotting
mtcars<- within(mtcars, rm("color"))
```

This gives us the following dot plot where we observe cars with four cylinders tend to have the maximum miles per gallon compared to other cars!



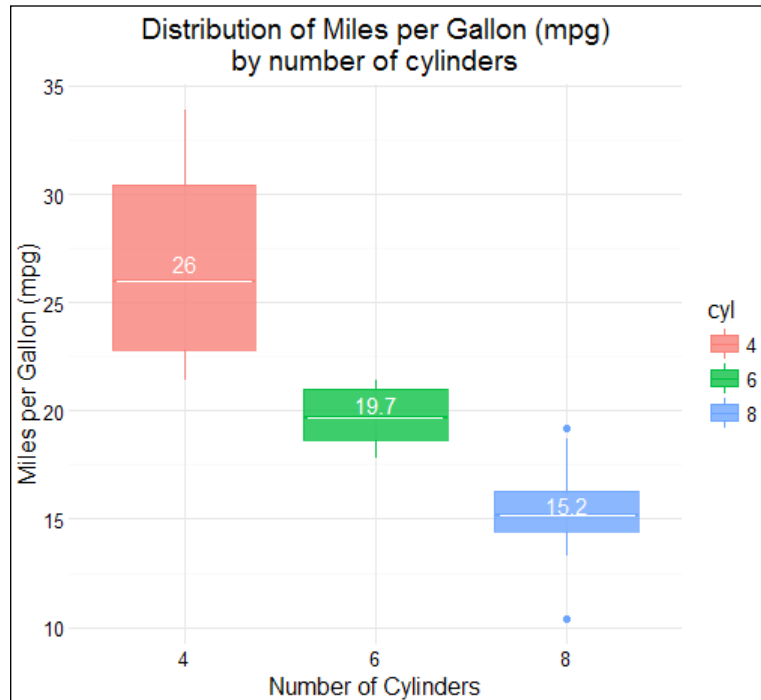
We will now leverage a visualization library, called `ggplot2`, to plot some boxplots that display the relationship of `mpg` with some other car attributes. Here, `mpg` is our variable of interest. We will try to perform some statistical inference and regression modeling later taking `mpg` as the response variable, which we will try to predict based on the other attributes of the various cars. The `ggplot2` visualization library is an excellent library and is a plotting system used in R. It is based on the grammar of graphics, which helps in building extremely complex plots with minimal efforts. It is often used to produce publication-quality plots.

We will start by observing car mpg distributions over the number of cylinders using the following code snippets:

```
# load visualization dependencies
library(ggplot2)
theme<- theme_set(theme_minimal())

# Car MPGs by number of cylinders visualization
ggplot(mtcars,
mapping=aes_string(y = "mpg", x = "cyl")) +
xlab("Number of Cylinders") +
ylab("Miles per Gallon (mpg)") +
ggtitle("Distribution of Miles per Gallon (mpg)\nby number of cylinders")
+
geom_boxplot(outlier.colour = NULL,
aes_string(colour="cyl", fill="cyl"), alpha=0.8) +
stat_summary(geom = "crossbar",
width=0.70,
fatten=0.5,
color="white",
fun.data = function(x) {
  return(c(y=median(x),
ymin=median(x),
ymax=median(x)))
}
) +
stat_summary(fun.data = function(x) {
  return(c(y = median(x)*1.03,
label = round(median(x),2)))
},
geom = "text",
fun.y = mean,
colour = "white")
```

This gives us the following visualization, where we see that the median mpg of cars with four cylinders is the maximum, followed by cars with six and eight cylinders, respectively.



We can also prove that our observations are programmatically correct using the aggregate function in the following code snippet:

```
# insights into average\median mpg of cars by cylinder
aggregate(list(mpg=mtcars$mpg),
list(cylinders=mtcars$cyl),
FUN=function(mpg) {
  c(avg=mean(mpg),
  median=median(mpg))
})
)
```

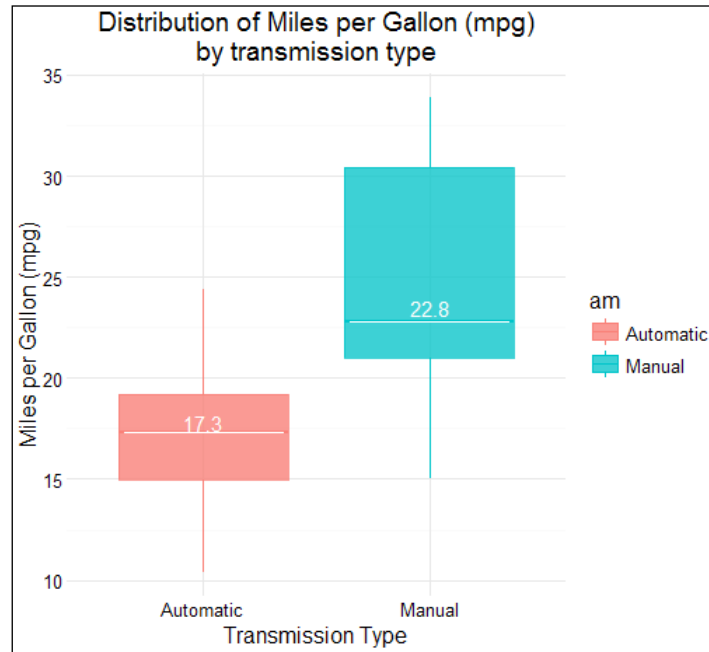

This gives us the following table where we observe that the median values are the same as the ones that we observed in the plot, and the mean values are also quite similar to the median:

	cylinders	mpg.avg	mpg.median
1	4	26.66364	26.00000
2	6	19.74286	19.70000
3	8	15.10000	15.20000

We will now observe car mpg distributions over transmission type, using the following code snippet:

```
# Car MPGs by Transmission type visualization
ggplot(mtcars,
  mapping=aes_string(y = "mpg", x = "am")) +
  xlab("Transmission Type") +
  ylab("Miles per Gallon (mpg)") +
  ggtitle("Distribution of Miles per Gallon (mpg)\nby
  transmission type") +
  geom_boxplot(outlier.colour = NULL,
  aes_string(colour="am",
  fill="am"), alpha=0.8) +
  stat_summary(geom = "crossbar",
  width=0.70,
  fatten=0.5,
  color="white",
  fun.data = function(x) {
    return(c(y=median(x),
    ymin=median(x),
    ymax=median(x)))
  }
) +
  stat_summary(fun.data = function(x) {
    return(c(y = median(x)*1.03,
    label = round(median(x),2)))
  },
  geom = "text",
  fun.y = mean,
  colour = "white")
```

This gives us the following chart where we see the median mpg for cars with manual transmission is **22.8**, which is much higher than **17.3**, the median mpg for cars with automatic transmission:



We can verify these statistics using the aggregate function, as we did earlier using the following code:

```
# insights into average\median mpg of cars by transmission
aggregate(list(mpg=mtcars$mpg),
list(transmission=mtcars$am),
FUN=function(mpg) {
  c(avg=mean(mpg),
    median=median(mpg))
})
)
```

This gives us the following table, where we clearly observe that the mean and median mpg for cars with manual transmission is higher than for automatic transmission cars:

	transmission	mpg.avg	mpg.median
1	Automatic	17.14737	17.30000
2	Manual	24.39231	22.80000

We will now try to prove a hypothesis, which is based on the preceding data, that the mpg statistic (mean) is different for cars with manual and automatic transmission using the principles of statistical inference in the next section.

Statistical inference

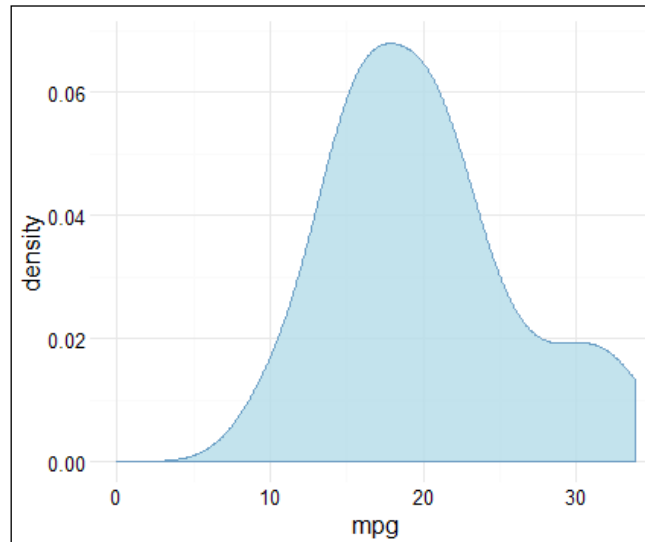
Statistical inference is the process of inferring or deducing patterns, insights, and properties of a dataset using methods such as hypothesis testing. In the previous segment, we used visualizations and aggregations to see that the average miles per gallon were significantly different for cars with automatic and manual transmission. We will now use a statistical test to prove this.

We will start off with a hypothesis (H_0) that the difference in mpg means for automatic and manual transmission cars is zero. Our alternate hypothesis will be that the difference in means is not zero. As our sample size is quite small, using a t-test would be appropriate here.

A t-test, often called a **Student's t-test**, is a statistical hypothesis test that can be used to determine whether two sets of data are significantly different from each other using a test statistic, which is the average mpg in our case. An underlying assumption also is that this test statistic follows a normal distribution. We will start by viewing the data distribution for car miles per gallon using the following code snippet:

```
# view data distribution
ggplot(mtcars, aes(x=mpg)) +
  geom_density(colour="steelblue",
  fill="lightblue", alpha=0.8) +
  expand_limits(x = 0, y = 0)
```

This gives us the following distribution of car mpg values in the form of a density plot, and we see that the distribution is almost a perfect bell-shaped distribution, which is the characteristic of a normal distribution.



We will now perform the t -test using the following code snippet:

```
# t-test  
t.test(mpg ~ am, data = mtcars)
```

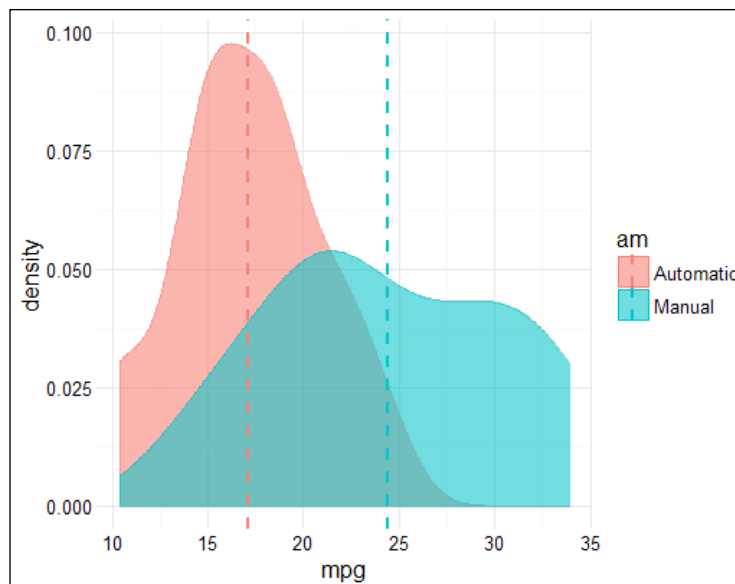
This gives us the following output, which clearly shows us that the mean mpg in the automatic transmission group of cars is much lower as compared to the mean mpg in the manual transmission group of cars:

```
welch Two sample t-test  
data: mpg by am  
t = -3.7671, df = 18.332, p-value = 0.001374  
alternative hypothesis: true difference in means is not equal to 0  
95 percent confidence interval:  
-11.280194 -3.209684  
sample estimates:  
mean in group Automatic    mean in group Manual  
17.14737                    24.39231
```

We can also visualize the t-test results using some density plots using the following code:

```
# visualizing t-test results
aggr<- aggregate(list(mpg=mtcars$mpg),
list(transmission=mtcars$am),
FUN=function(mpg){c(avg=mean(mpg))})
ggplot(mtcars, aes(x=mpg)) +
geom_density(aes(group=am, colour=am, fill=am),
alpha=0.6) +
geom_vline(data=aggr, aes(xintercept=mpg, color=transmission),
linetype="dashed", size=1)
```

This gives us the following density plot where the dotted lines indicate the mean mpg for manual and automatic transmission cars; these are the exact figures that we obtained earlier from our t-test and aggregations:



This finalizes our segment about statistical inference. Next, we will focus on the final section of our analysis, which is related to statistical modeling.

Statistical modeling with regression

In this segment, we will focus on building some regression models to try and predict car miles per gallon (`mpg`) values that are based on the other attributes of the car. We will use multivariate linear regression here to build our models. The linear regression modeling approach usually consists of a response variable, which we want to predict (which is also known as the outcome variable—`mpg` in our case) and several input variables. It also assumes that there is a linear relationship between the input variables and our response variable. Mathematically, this can be denoted as follows.

The different variables in the equation are explained, as follows:

- Y : This is the dependent or response variable (`mpg` in our case)
- x_i : This denotes the input variables, for example, $i=1, 2, 3, \dots, n-1$ (`am`, `cyl`, and so on, in our case)
- C_0 : This constant is the value of y when x_i is 0 and is called the **intercept**
- C_i : This denotes the coefficients for the input variables, ($i=1, 2, 3, \dots, n-1$)
- ϵ : This denotes the error term or noise that captures all other factors influence the responding variable besides the input variables

As we have a far smaller number of samples in our dataset, we will train our model on almost all the samples and try to predict the `mpg` value for one sample. First, we will prepare our datasets using the following code:

```
# prepare datasets
car.to.predict<- mtcars[15, ]
training.data<- mtcars[-15, ]
```

Next, we will build our first regression model on only the training data using the following code:

```
# build initial model
initial_model<- lm(mpg ~ ., data = training.data)
```

Now, we can view the details of the model we just built using the `summary(initial_model)` command, which gives us detailed information regarding the model, the coefficients of the various variables, and different metrics. You will observe that the **adjusted R-squared** value is **0.7832**, which indicates that **78.32%** of variation in our response variable (`mpg`) is explained by our input variables. The higher this value is, then the better our model will be because it will be able to explain most of the variability that is observed in the response variable, which we want to predict.

Now, we will try to build a series of regression models and select the best model from them on the basis of an evaluation metric called **Akaike Information Criterion (AIC)**, which we will inspect in detail. The following code snippet steps through multiple regression models and finally selects the best one:

```
# best model selection
best_model<- step(initial_model, direction = "both")
```

This generates an output for a series of steps, and we depict the output of the final step by selecting the best model in the following snapshot:

Step:	AIC=60.51			
mpg ~ wt + qsec + am				
	Df	Sum of Sq	RSS	AIC
<none>			168.66	60.510
+ hp	1	9.893	158.76	60.636
+ disp	1	4.181	164.47	61.732
+ drat	1	1.304	167.35	62.270
+ vs	1	0.001	168.66	62.510
+ cyl	2	10.508	158.15	62.516
- am	1	26.563	195.22	63.044
+ gear	2	0.266	168.39	64.461
+ carb	5	11.262	157.40	68.368
- qsec	1	107.932	276.59	73.845
- wt	1	141.465	310.12	77.392

We see that *wt*, *qsec*, and *am* were the most important attributes, used as input variables to build the best regression model, and the AIC value of the model is **60.51**, which is the least of all the models that were generated. It is used as a measurement to evaluate the quality of various regression models against each other and then select the best model, which minimizes the loss of information and has a minimum number of parameters. We can view the details of the best model using the following code:

```
# view model details
summary(best_model)
```

This gives us the following information, where we observe that the three input variables that were used to create the final model were `wt`, `qsec`, and `am`. We also notice that the **adjusted R-squared** value is **0.8179**, which is better than the value that we obtained in our initial model. We can observe this in the following output snapshot:

```
Call:
lm(formula = mpg ~ wt + qsec + am, data = training.data)

Residuals:
    Min       1Q   Median       3Q      Max
-3.4950 -1.6339 -0.7351  1.4651  4.5774

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   8.9085     7.4186   1.201 0.240246
wt            -3.8076     0.8001  -4.759 5.82e-05 ***
qsec           1.2449     0.2995   4.157 0.000292 ***
amManual       3.0518     1.4799   2.062 0.048939 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.499 on 27 degrees of freedom
Multiple R-squared:  0.8361,    Adjusted R-squared:  0.8179
F-statistic: 45.92 on 3 and 27 DF,  p-value: 9.767e-11
```

We will now look at the car whose `mpg` value we want to predict using the following code:

```
# MPG of car to predict
print(data.frame(car.to.predict=data.matrix(
list(rownames(car.to.predict),
car.to.predict[, "mpg"])
)), row.names = FALSE)
```

This gives us the following output, showing the actual `mpg` of the car:

```
car.to.predict
Cadillac Fleetwood
10.4
```

We now predict the value of `mpg` for this car using our initial model, as follows:

```
# initial model prediction
predict(initial_model, car.to.predict)
```

This gives us the following `mpg` value as the predicted output:

```
Cadillac Fleetwood
16.0991
```


Next, we make another prediction using our best model using the following code:

```
# best model prediction
predict(best_model, car.to.predict)
```

This gives us the predicted mpg value, as follows:

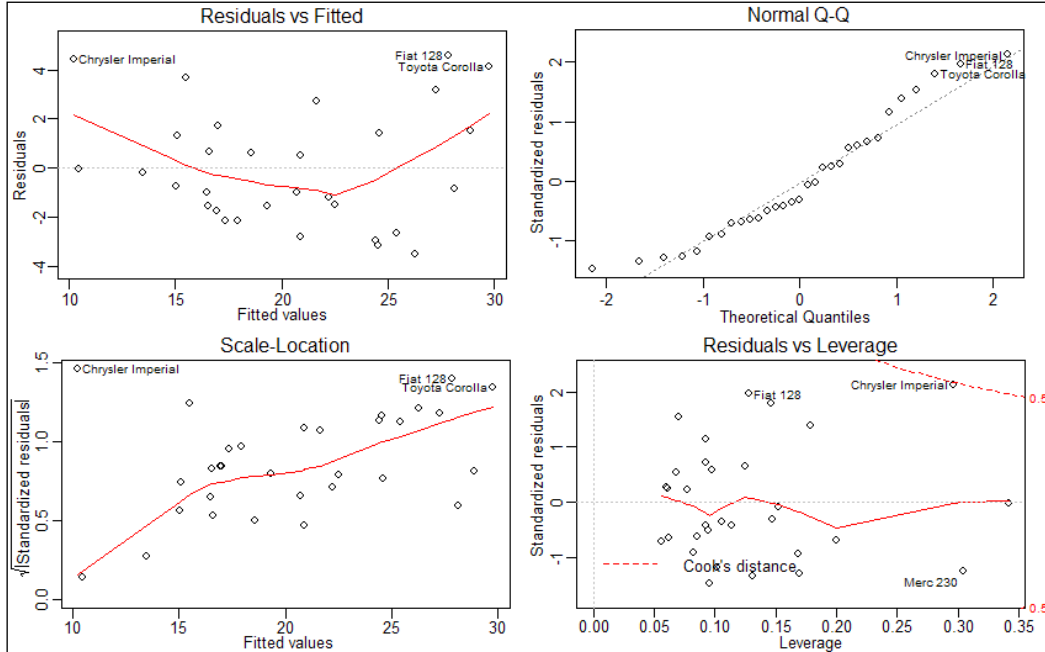
Cadillac Fleetwood
11.30242

We observe that our best model predicts much more effectively than our initial model and the predicted value of mpg for Cadillac Fleetwood (**11.3**) is much closer to its true mpg value (**10.4**), as compared to the predicted mpg value (**16.1**) that was obtained from the initial model.

Finally, we will look at some regression model diagnostics and residual plots using the following code snippet:

```
# best model diagnostics
par(mfrow = c(2, 2))
plot(best_model)
```

This gives us the following set of plots:



We can make the following observations from the plots:

- The points in the **Residuals vs. Fitted** plot seem to be randomly scattered on the plot, and they verify homoscedasticity, which indicates the variance of error is uniform across all x values
- The **Normal Q-Q** plot consists of the points that mostly fall on the line, indicating that the residuals are almost normally distributed
- The **Scale-Location** plot consists of points that are scattered in a constant band pattern, indicating constant variance
- There are also some distinct points of interest (outliers or leverage points) in the plots, which we shall discuss next

You may have noticed some specific data points with the names of cars mentioned in the preceding plots. These points are often known as outliers, and they can be separated into two types of points: influence and leverage points.

Influential points are data points that, if removed from the dataset, they change the parameter estimates of the regression model by a significant amount and cause a notable change in the computation results, based on changing the position of the regression line. We can compute the influential data points in our dataset using the following code snippet:

```
# influence points
influential<- dfbetas(best_model)
tail(sort(influential[,4]),4)
```

The `dfbetas` function is usually used to find out the extent to which one of these influential points has affected the estimate of the regression line and corresponding coefficients. This gives us the top four influential points, as shown in the following snapshot:

Toyota Corolla	Toyota Corona	Fiat 128	Chrysler Imperial
0.2947699	0.4069629	0.4643860	0.7345127

Leverage points are data points that always have high or extreme values of the independent variables such that they might have a greater ability to move the regression line, based on its position as compared to the other data points. These points can also be influential if they fall outside the general pattern of the other data points, thus, greatly affecting the position of the regression line. We will compute the high leverage data points in our dataset using the following code snippet:

```
# leverage points
leverage<- hatvalues(best_model)
tail(sort(leverage),4)
```

We use the `hatvalues` function to get the following top-four high-leverage points, that are depicted in the following snapshot:

```
Maserati Bora Chrysler Imperial Merc 230 Lincoln Continental
0.1995797          0.2960847 0.3048291          0.3416256
```

You will notice that several of these cars are depicted in the diagnostic plots of our model, which we saw earlier.

This brings us to the end of our data analysis process. By now, you have seen the benefits of exploratory analysis, visualizations, and modeling to get interesting insights from data.

R Cheat Sheets

We just learned about the CRISP-DM model while utilizing and understanding various R constructs and libraries to solve a real-world problem. This concluding section presents various utilities, tricks, and techniques in the form of cheat sheets to facilitate a quick look-up.

In this section, we will present cheat sheets that are organized in the following manner:

- Data processing and transformation
- Math and modeling
- Plotting
- Important links

Data processing and transformation

For any kind of analysis, input/output and transformation of data are core tasks. R is a robust platform with many features that we will cover in the following sections.

Data handling

To extract and load data for any kind of analysis, R provides pretty powerful and easy-to-use utility functions. Some of these are listed, as follows:

- `read.csv(<file_name>)`: This imports a standard `.csv` file
- `write.csv(<object_name>, <file_name>)`: This exports to a `.csv` file
- `data(<dataset_name>)`: This loads R's built-in dataset
- `head(<object>)`: This prints the first few entries of the data imported
- `names(<object>)`: This lists variables in an object
- `read.table(<file_name>)`: This reads contents from an ASCII file

Basic data types

Data types form the basic constructs for R—or any other language as a matter of fact. What makes R special is an extended list of basic data types to handle varied data types. These are as follows:

- `numeric` (`integer` and `double`) and `character`: These are data types that are available in R
- `factor`: This allows you to store categorical data while a complex data type is used for complex numbers
- `is.<data_type>` and `as.<data_type>`: These are used to check data types and type conversion, respectively
- `length(<variable>)`: This gives you a count of characters in a variable

Data structures

R provides many data structures out of the box, which we discuss in the following subsections.

Vectors

This is the most basic data structure in R. It is similar to a mathematical vector. The following are ways to interact with a vector in R:

- `r[1]`: This allows you to access elements using square braces. The element count begins from 1.
- `r[x > 100]`: These vectors support logical expressions as indices.
- `r[5:10]`: These vectors support subselection. The given example returns vector values between the index 5 to 10.
- `r[-1]`: This returns all indices except 1.
- `factor(x)`: This converts a vector `x` to factor.
- `which.max(x)` and `which.min(x)`: These return the maximum and minimum values of `x`, respectively.
- `rev(x)`: This reverses the elements of `x`.
- `table(x)`: This gives you the frequency table for elements of the `x` vector.
- `match(a, b)`: This returns values from `a` which exist in `b`; otherwise, this is not applicable.

Arrays and matrices

R supports multidimensional arrays. A matrix is a two-dimensional array. The following are access patterns for these data structures:

- `array (<vector>, <vector_dimensions>)`: This generates an array from an input vector
- `%o%`: This gives you the outer or cross product of two arrays
- `x[a, b, c]`: This is when the dimensions of an array are comma-separated and accessed from within square braces
- `matrix(<vector>, nrow=r, ncol=c)`: This generates an $r \times c$ matrix with values from `<vector>`
- `t(<matrix>)`: This is the transpose of a matrix
- `diag(<matrix>)`: This gives the diagonal of a matrix
- `colsum(<matrix>)` and `rowsum(<matrix>)`: This calculates the sum of columns and rows of a matrix, respectively
- `colmeans(<matrix>)` and `rowmeans(<matrix>)`: This calculates the sum of columns and rows of a matrix, respectively
- `%*%`: This is a matrix multiplication operator
- `lower.tri(<matrix>)`: This returns a vector with values from the lower triangle of a matrix

Lists

A list is an ordered collection of named or unnamed objects, which may or may not be homogenous. These are recursive data structures; that is, a list's element can itself be a list. A list can be manipulated using the following:

- `list(<object_1>, <object_2>, ...)`: This generates a list of objects that are separated by a comma
- `L[[i]]`: This is when double-square brackets are used to access elements at the i^{th} index of the list
- `length(<list>)`: This returns the count of the topmost elements of a list
- `L$<name>`: This is when the `$` operator allows access to the `<named>` element of list `L`; this is the same as `L[[i]]`

Data frames

Data frames are tabular structures that can have columns of different data types and attributes. A data frame may contain components of the numeric, character, factor, or list types, or it may contain other data frames. The following utilities help in manipulating data frames:

- `data.frame(col1=<object1>, col2=<object2>, ...)`: This generates a data frame with n columns or components, which have values from corresponding objects
- `attach(<data.frame>)`: This exposes components of a data frame in a search path for easy access
- `merge(x, y)`: This combines two data frames that are based on common columns or row names

General utilities

Apart from the utilities and the other constructs that we just discussed, R provides a rich set of general utilities to make data analysis even easier. Check out the following utilities:

- `c(1:5)`: This is a generic function that concatenates values. The given example would generate a vector with values 1 to 5.
- `rep(<value>, <count>)`: This generates a vector with repeating `<value>` elements of the `<count>` size.
- `seq(to, from)`: This generates a sequence vector starting with `to` and ending with `from`. You can also specify increments; the default is 1.
- `sort(c(10, 9, 8, 7))`: This returns a sorted vector 7, 8, 9, 10.
- `order(10, 9, 1, 2)`: This returns indices in ascending order as 3, 4, 2, 1.
- `rank(10, 5, 6, 9)`: This returns the rank order of elements as 4, 1, 2, 3.
- `summary(<object>)`: This has summary details, such as min, max, mean, median, and so on, for the object.
- `choose(n, k)`: This returns the combination of k in n repetitions.
- `na.omit(x)`: This suppresses all the missing values (`nas`) from `x`.
- `na.fail(x)`: This errors out if `x` contains even a single missing value.
- `unique(x)`: This returns only distinct or unique values of `x`. This works with vectors and data frames.
- `paste(...)`: This converts objects to strings and concatenates them.

- `substr(cv, start, stop)`: This substrings from the `cv` character vector from the `start` to the `stop` position.
- `grep(ptrn, cv)`: This searches for the `ptrn` patterns in the `cv` vector.
- `gsub(ptrn, rep, cv)`: This replaces match for the `ptrn` pattern with the `rep` replacement in the `cv` vector.
- `tolower` and `toupper`: This converts character vector elements to lowercase and uppercase, respectively.

Math and modeling

R has a rich set of inbuilt functions and packages to perform mathematical and modeling operations.

Math and modeling utilities

As R is a statistical language, it provides a rich set of mathematical functions that are available right out of the box (while more can be added using additional libraries or packages):

- `sum(x)`: This is the sum of the elements of `x`.
- `cumsum(x)`: This calculates the cumulative sum of the elements of `x`.
- `diff(x)`: This is the pair-wise difference between the elements of vector `x`.
- `prod(x)`: This is the product of the elements of `x`.
- `mean(x)` and `median(x)`: This is the mean and median of `x`, respectively.
- `var(x, y)`: This is the variance between the elements of `x` and `y`. It works with matrices and data frames as well. This is the same as `cov(x, y)`.
- `quantile(x, probs)`: This returns the `quantile` breakup of `x` for given probabilities.
- `sd(x)`: This is the standard deviation for `x`.
- `weighted.mean(x, w)`: This returns the weighted mean of `x` using the `w` weight vector.
- `cor(x, y)`: This is the linear correlation between `x` and `y`.
- `round(x, n)`: This rounds the elements of `x` to `n` digits.
- `log(a, b)`: This calculates the log of `a` for base `b`.
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, and so on: These are Trigonometric functions.
- `exp(x)`: This exponentiates each element of the `x` vector.
- `scale(m)`: This centers or scales the elements of an `m` numeric matrix.

- `union(x, y)`, `intersect(x, y)`, and `is.element(e, x)`: These are Set functions that are also available.
- `Conj(c)`: This returns the conjugate of the `c` complex number.
- `rnorm`, `rpois`, `rgamma`, `rexp`, `rcauchy`, `rt`, and so on: These can be used to generate Gaussian, Poisson, Gamma, Exponential, Cauchy, and Student distributions.
- `fft(x)`: This calculates Fast Fourier Transform of the elements of `x`.
- `apply(m, INDEX, FUNC)`: This applies the `FUNC` function on the `INDEX` index of the `m` matrix.
- `lapply(l, FUNC)`: This applies the `FUNC` function on the `l` list.
- `optim(params, func, mtds)`: This is the general-purpose method to optimize a `func` function for the `params` parameters using the `mtds` methods.
- `lm(frml)`: This fits a linear model on the `frml` formula. This is used for regression and covariance analysis. Also, check `glm` for generalized linear models.
- `nls(fml)`: This fits nonlinear least squares estimates for nonlinear models.
- `spline(s)`: This calculates the cubic spline.
- `predict(fit, [...])`: This is a generic function to test model fitting on input data.
- `df.residual(fit)`: This calculates the degrees of residual freedom from `fit`.
- `coef`, `residuals`, and `deviance`: These return coefficients, residuals, and deviance of models fitted.
- `logLik(fit)`: This calculates the log likelihood of the model fitted.
- `aov(frml)`: This performs analysis of variance model calculations on `frml`.
- `Anova(fit, [...])`: This performs analysis of variance of models fitted.

Math and modeling packages

The following is a list of popular and mature sets of packages, which enhance the power of R:

- `arules`: This is association rule mining
- `cluster`, `fpc`, `mclust`: This is clustering and classification
- `DmwR`, `dprep`, `rlof`: This is outlier detection
- `multicore`, `snow`: This is a multiprocessing library
- `nlme`: This is regression, linear, and nonlinear modeling

- `TraMiner`: This is sequential pattern mining
- `party` and `rpart`: These are recursive partitioning, decision trees, and survival analysis
- `nnet`: This is neural networks
- `kernlab` and `e1071`: These support Vector Machines, PCA, Naive Bayes, fuzzy clustering, and so on.
- `stats`, `ast`, `forecast`: This is for time series analysis
- `RgoogleMaps`, `ggmap`, `plotKML`, and `spdep`: These are for spatial analysis
- `sna`, `network`, and `igraph`: These are for social network analysis
- `tm`, `lda`, `topicmodels`, `RTextTools`, and `tau`: These are for text mining

Plotting

Statistical analysis and data science are way too difficult without graphs and visualization. R has a rich set of utilities and libraries for plotting. Let's have a look at a few of these:

- `plot(y)`: This plots the values of y on the y axis ordered by indices on the x axis.
- `plot(x, y)`: This plots values on the x and y axis, respectively.
- `barplot(x)`: This is a bar plot of the values of x .
- `hist(x)`: This is a histogram of frequencies of the elements of x .
- `pie(x)`: This is a pie chart for the elements of x .
- `boxplot(x)`: This is a boxplot for the elements of x .
- `plot.ts(x)`: This is a plot with respect to time.
- `mosaicplot(x)`: This is a mosaic graph of residuals of a log-linear regression.
- `contour(x, y, z)`: This is a contour plot of x and y , where x and y must be vectors and z should be a matrix of the $x \times y$ dimension.
- `qqplot(x, y)`: This is a quantile plot of y with respect to x .
- `abline(m, c)`: This draws a line with the m slope and the c intercept. This can also be used to draw horizontal, vertical, and regression lines.
- `rect(x1, y1, x2, y2)`: This draws a rectangle, based on the top-left $(x1, y1)$ and bottom-right $(x2, y2)$ coordinates.
- `polygon(x, y)`: This draws a polygon, connecting the elements of x and y .
- `xlim, ylim`: These are the x and y limits of a graph.

- `col()`: This is the line or symbol color.
- `text()`, `title()`, and `legend()`: These are for text, title, and legends on a graph.

Plotting packages

Let's now take a look at some plotting packages.

- `ggplot2`: This is the de facto graphics grammar for R
- `ggvis`: This is a rich and powerful plotting library
- `googleVis`: This brings the power of Google Visualization APIs to R
- `lattice`: This is specialized for multivariate data
- `iplots`: These are interactive plots

Summary

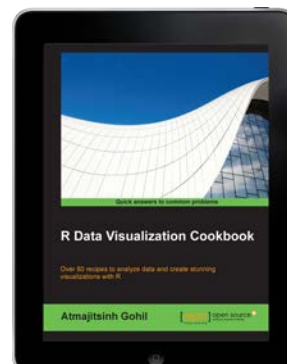
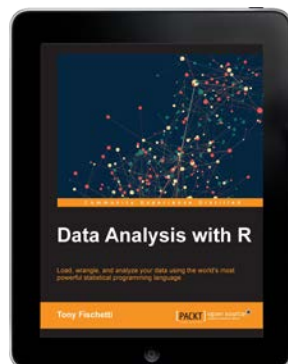
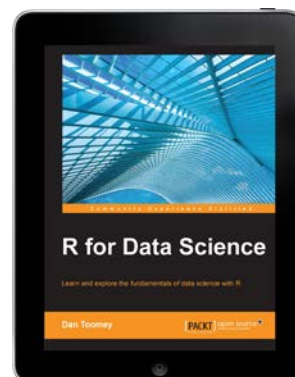
Using this guide, we went on a journey from the origins of R to using it to analyze the `mtcars` dataset available in R. Throughout the guide, we learned about the R ecosystem, its tools, and services, and we understood the basic constructs of R along with the CRISP-DM data analysis model or life cycle to perform different analyses. We performed exploratory analysis, and we went on to draw relationships and insights using various packages for regression modeling and visualization. We also looked at the model evaluation methods for such models. We concluded the guide by listing neat tips and tricks, along with popular and standard sets of packages and utilities, for quick reference.

As R is a community-driven language and software platform, it thrives and improves on user contributions. Hence, we urge our readers to learn and then contribute back to the community by way of blogs, tutorials, libraries, bug fixes, and so on.

What to do next?

Broaden your horizons with Packt

If you're interested in R, then you've come to the right place. We've got a diverse range of products that should appeal to budding as well as proficient specialists in the field of R.



To learn more about R and find out what you want to learn next, visit the R technology page at <https://www.packtpub.com/tech/r>

If you have any feedback on this eBook, or are struggling with something we haven't covered, let us know at customer-care@packtpub.com.

Get a 50% discount on your next eBook or video from www.packtpub.com using the code:

