Business Analytics in R

Introduction to Statistical Programming

TIMOTHY WONG[1]
JAMES GAMMERMAN[2]

July 15, 2019

[1]timothy.wong@hotmail.co.uk
[2]jgammerman@gmail.com

Contents

List of Figures

List of Tables

Preface

R is an open source language for statistical programming. It is widely used among statisticians, academic researchers, and business analysts across the world. This book is part of a training course designed for business analysts.

Web Access

The LaTeX source code and the compiled PDF version of this book can be digitially accessed at `https://github.com/timothywong731/r-training`

Acknowledgements

- A special word of thanks goes to the open source R community.
- Some of the examples in chapter 3 were adopted from Wickham and Grolemund's book [2].
- Thanks to all contributors for reviewing this book:
  - Artemis Maipa
  - Phuong Pham
  - Laura Shemilt
  - Nyala Noe

Chapter 1

R Ecosystem

R is a language for statistical programming. The language is open source and free for personal, academic and commerical use. It is available on multiple platforms including Windows, Linux and MacOS. In recent years, R has gained popularity and become one of the fastest growing programming languages in the world[1]. In this chapter, we will go through the basics of R.

1.1 Programming Enviornment

RStudio[2] is a popular Integrated Development Environment (IDE) for the R language. It is a very powerful editor which enables programmers to interact with the R language. RStudio is open-sourced and the desktop version is free for personal use.

Inside RStudio, the display is divided into different tabs which users can customise. The following ones are particularly important:

Source Editor Inside RStudio, users can access the source editor for R code. This can be treated as a powerful text editor for the various forms of R code, such as standard R Script, R Markdown, R Notebook, R Sweave and shiny etc. The source editor does not execute any code, it purely helps users edit it efficiently.

Console The R interpreter is shown in the Console window. This is the place where R code is processed. User can highlight a segment of R code in the main text editor and press Ctrl + Enter to send it to execution. Once the code is

[1] https://stackoverflow.blog/2017/10/10/impressive-growth-r/
[2] https://www.rstudio.com

sent, it will be executed in the `R` interpreter and the results will be displayed. Alternatively, users can just place the blinking cursor on a line and press the same keys. This sends the entire line to the interpreter for execution. Users can press `Ctrl + L` to clear existing screen output.

Environment Users can access all the variables created in the current `R` session. The workspace which includes all variables can be exported as `.RData` file in the environment window. Users can also import workspace from an existing `.RData` file.

Files It allows users to browse files on the local storage system. Users can navigate to their own home directory using the ~ path.

Plots All graphical outputs produced by the `R` interpreter are displayed in the plots tab. Users can export the graph as file in this tab.

Packages Users can view the installed packages in this tab and load them manually.

Help Documentation can be searched and viewed in this tab. Users can open documentation of a particular function using `?` character in the interpreter. For instance, `?sum` will open up the documentation for the `sum()` function.

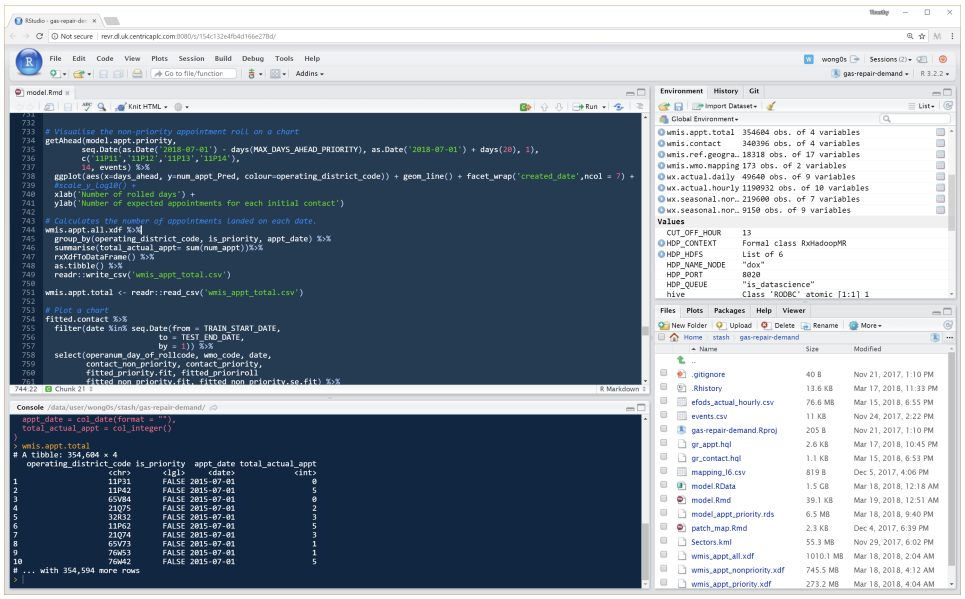


Figure 1.1: RStudio Server Pro (RSP)

Exercise 1 Using RStudio IDE

- Use your interactive console to print your name using : `print("hello <my name>")`
- Write the source code to print your name and run it.

1.2 Packages

Functionality in `R` can be extended through packages. Most packages are open-sourced with a few exceptions. They are published on community-maintained repositories such as CRAN and Bioconductor.

The function `install.packages()` can be used to install new packages from CRAN (or a CRAN-like source).

R Example 1.2.1

```
# Return all installed packages
installed.packages()
# Install a new package and all its dependencies from CRAN
# This will install to the default library location
install.packages("ggplot2")
# Load an installed package
# (both lines are identical)
library(ggplot2)
library("ggplot2")
```

The number of packages on CRAN has grown exponentially in the past few years. To help users select relevant packages for certain statistical topics, users can refer to the CRAN Task View[3]. It provides a curated list of packages which usually serves as good topical guide.

[3] `https://cran.r-project.org/web/views/`

Chapter 2

Programming Concepts

In R, everything begins with variables. Users can assign value to variable using the <− symbol. The # character is used for commenting.

To name a variable, it is very common to name variables in R using the 'camel case' convention. Sometimes 'snake case' and 'dot case' are also used. There is a detailed journal article [1] explaining the rationale behind each naming style.

When creating a new variable, it is important to avoid using reserved words. Reserved words can be checked by running the command ?Reserved.

R Example 2.0.1

```
# Assign variables
myVarX <- 5
myVarY <- 20
# Perform multiplication
myVarX * myVarY
# Look at the reserved words
?Reserved
# Camal case
myCamalCaseVar <- "this is camal case"
# Other less commonly-used styles
my_snake_case_var <- "this is lower snake case"
MY_UPPER_SNAKE_CASE_VAR <- "this is upper snake case"
my.dot.case.var <- "this is dot case"
MyPascalCaseVar <- "this is pascal case"
```

2.1 Vector

R is a vectorised programming language. A vector of consecutive integers can be created using the `:` character. Alternatively, a vector of discrete values can be created using the function `c()`. All elements of a vector must have the same data type.

R Example 2.1.1

```
# Create a vector of integers one to ten
myVec1 <- 1:10
# Find out the length of vector
length(myVec1)
# Reverse the vector
# This does not change the value of myVec1
rev(myVec1)
# Create a custom vector of 10, 15, 20, 25, 30
myVec2 <- c(10, 15, 20, 25, 30)
# Create a vector of sequential numbers with increment 0.5
myVec3 <- seq(from = -2, to = 2, by = 0.5)
# Elements of a vector can be named
myVec4 <- c(`New York` = 8.5,
            `London` = 8.6,
            `Moscow` = 11.9)
```

Users can select elements of a vector by subsetting using index number. In R, index numbers start from $1, 2, 3, \ldots$. For example, `myVec2[2]` will return the second element of the vector.

R Example 2.1.2

```
# Select the second element of the vector
myVec2[2]
# Subset a range from the vector
myVec2[2:4]
# Subset specified elements
myVec2[c(4,2,3)]
# Subset named element of a vector
myVec4["New York"]
```

Most operations in R are vectorised. Vectorisation means the operation is applied to the whole vector instead of individual elements.

R Example 2.1.3

```
# Arithmetic operations
myVec1 + 10
myVec1 - 10
```

```
myVec1 * 2
myVec1 / 2
myVec1 ^ 2
log(myVec1)
```

2.2 Character and Datetime

Users can create vectors of other data types easily in R. Example 2.1.3 shows how to create a character vector, a date vector and a POSIXct(date/time) vector.

R Example 2.2.1

```
# Vector can contain character objects
myVec4 <- c("Bill", "Mark", "Steve", "Jeff", "Larry")
# Constant character vectors in R
LETTERS
letters
month.name
month.abb
# This is a vector of Date objects
myVec5 <- as.Date(c("2017-07-13",
                    "2017-10-11",
                    "2017-11-21",
                    "2018-01-16",
                    "2018-03-27"))
# Load the lubridate package
# Use the function ymd_hms() to parse date/time with timezone
# Returns a vector of POSIXct (date/time) object
library(lubridate)
myVec6 <- ymd_hms(c("2017-07-13 09:30:00",
                    "2017-10-11 08:00:00",
                    "2017-11-21 10:00:00",
                    "2018-01-16 11:30:00",
                    "2018-03-27 12:00:00"),
                  tz = "Europe/London")
# Date/time manipulation applied to a vector
myVec7 <- myVec6 + hours(1) + minutes(30)
# Compute the day of week - returns a vector of characters
weekdays(myVec7)
```

2.3 Factor

Categorical data can be represented in R using `factor`. This is an efficient to store repetitive values.

Factors can either be ordered or unordered. Ordered factors can be compared using logical operators.

R Example 2.3.1

```
# Create an ordered factor variable
myValues <- c("Low", "High", "High", "Medium", "Medium")
myOrder <- c("Low", "Medium", "High")
myFactor <- factor(myValues,
                   levels = myOrder,
                   ordered = TRUE)
# Print the factor
myFactor
# Check the number of levels in the factor variable
nlevels(myFactor)
# Print the labels of the factors
levels(myFactor)
# Compare the levels of individual observations
myFactor[1] > myFactor[2]
myFactor[2] > myFactor[1]
myFactor[2] == myFactor[3]
```

2.4 Logical Operator

Logical operators can be applied on vector objects. It always returns a vector of logical values with the same length as the input vector. Examples 2.4.1

R Example 2.4.1

```
# Find all values greater than 5 - returns a vector of logical values
myVec1 > 5
# Find all values equal to 7
myVec1 == 7
# Find all values matching 2,4,6 and 8
myVec1 %in% c(2,4,6,8)
# Find all values between 2 and 7
myVec1 >= 2 & myVec1 <= 7
# Find all values equal to 7 or equal to 8
myVec1 == 7 | myVec1 == 8
```

2.5 Special Numbers

There are special numbers in `R`, which are shown in Example 2.5.1. For instance, the variable `pi` is a constant 3.14159.... In most cases, missing values are usually indicated as `NA` (Not Available). All operations involving an `NA` input always produce an `NA` output.

On the other hand, values returned by computational error are `NaN` (Not-a-Number). Mathematical infinities are indicated by `Inf` and `-Inf`.

R Example 2.5.1

```
# Pi is constant 3.14159...
pi
# One divided by zero is infinity
1/0
# Negative number divided by zero is negative infinity
-1/0
# Infinity divided by infinity is Not-a-Number (NaN)
Inf/Inf
# Not available (NA) plus one is still NA
NA + 1
# Effects of different special numbers
c(5, 10, 15, NA, 25, 30, NaN, 35, 40, Inf, 50, -Inf, 60) / 5
```

2.6 List

Objects with different data types can be held together in a `list`. Example 2.6.1 illustrates a list containing objects of several data types. Elements of a `list` can be optionally named for easier subsetting.

R Example 2.6.1

```
myFavBook <- list(`title` = "R for Data Science",
                  `authors` = c("Garrett Grolemund", "Hadley Wickham"),
                  `publishDate` = as.Date("2016-12-12"),
                  `price` = 18.17,
                  `currency` = "USD",
                  `edition` = 1,
                  `isbn` = 1491910399)
# Select a named element of a list
# Use the dollar sign, followed by name without bracket
myFavBook$title
# Use double squared brackets with element's name as character
myFavBook[["authors"]]
# Select the fourth element in the list
myFavBook[[4]]
```

2.7 Data Frame and Tibble

A data frame is a list of variables of the same number of rows with unique row names. In many cases, datasets extracted from CSV file or SQL server are returned

as a data frame object. Example 2.7.1 demonstrates how to construct a data frame.

R Example 2.7.1

```
myFavMovies1 <- data.frame(`title` = c("Dr. No",
                                       "Goldfinger",
                                       "Diamonds are Forever",
                                       "Moonraker",
                                       "The Living Daylights",
                                       "GoldenEye",
                                       "Casino Royale"),
                           `year` = c(1962, 1964, 1971, 1979,
                                      1987, 1995, 2006),
                           `box` = c(59.5, 125, 120, 210.3,
                                     191.2, 355, 599),
                           `bondActor` = c("Sean Connery",
                                           "Sean Connery",
                                           "Sean Connery",
                                           "Roger Moore",
                                           "Timothy Dalton",
                                           "Pierce Brosnan",
                                           "Daniel Craig"))
```

In modern `R`, `tibble` is the enhanced version of the traditional data frame. More functions are available for `tibble` objects. Example 2.7.2 shows how to construct a `tibble` and append an extra row at the end.

R Example 2.7.2

```
library(tibble)
myFavMovies2 <- tibble(`title` = c("Dr. No",
                                   "Goldfinger",
                                   "Diamonds are Forever",
                                   "Moonraker",
                                   "The Living Daylights",
                                   "GoldenEye",
                                   "Casino Royale"),
                       `year` = c(1962, 1964, 1971, 1979,
                                  1987, 1995, 2006),
                       `box` = c(59.5, 125, 120, 210.3,
                                 191.2, 355, 599),
                       `bondActor` = c("Sean Connery",
                                       "Sean Connery",
                                       "Sean Connery",
                                       "Roger Moore",
                                       "Timothy Dalton",
                                       "Pierce Brosnan",
                                       "Daniel Craig"))
# Append an extra row at the end of the tibble
# Rewrite the original tibble object
myFavMovies2 <- add_row(myFavMovies2,
        title = "Spectre", year = 2015, box = 880.7,
        bondActor = "Daniel Craig")
```

Example 2.7.3 shows that a `tibble` can be subsetted in various ways. The most common operation is selecting a column by name.

R Example 2.7.3

```
# Get one column by name
myFavMovies2[["title"]]
myFavMovies2$title
# Get a range of columns by position ID
myFavMovies2[, 1:2]
myFavMovies2[1:2]
# Get rows 1 to 3
myFavMovies2[1:3, ]
# Get the "year" variable of row 1-3
myFavMovies2[1:3, "year"]
# Get the "title" and "year" variables of row 4-7
myFavMovies2[4:7, c("title","year")]
```

Exercise 2 Working with Data Frames

- Create your own favourite movies data frame (Your data frame must contain more than one type.)
- Return a variable (column) of your data frame.
- Can you add another variable (column) to your data frame?
- Can you return an observation (row)?

2.8 Function

User can create custom functions in `R`. In example 2.8.1, a new function `is.odd()` is created. The result can be explicitly returned using `return()`. Alternatively, the value of the function's last line is implicitly returned.

R Example 2.8.1

```
# Defines a custom function
is.odd <- function(x) {
  # The modulo operator %% returns the remainder
  # If a number divide by 2 gives remainder 1, then it is an odd number
  remainder <- x %% 2
  equalToOne <- remainder == 1
  return(equalToOne)
```

```
}
# Execute the function with one input
is.odd(5)
# Execute the function with an integer vector
is.odd(1:10)
# Define another function
is.even <- function(x) {
  !is.odd(x)
}
# Return true for even numbers
is.even(1:10)
```

Exercise 3 Working with Functions

- Write a new function called `square.it()` that returns the square of a number.
- Can you apply this to a numerical column in your dataframe?

2.9 Flow Control

2.9.1 If-Else

An `if-else` statement is controlled by the condition. The `if` part will be executed if the contition is `TRUE`. Alternatively, the `else` part will be executed. Example 2.9.1 shows a simple `if-else` control statement.

R Example 2.9.1

```
# Loads the lubridate package for additional date/time functions
library(lubridate)
# Find out what day is today
myWeekday <- weekdays(today())
# Check whether today is Saturday or Sunday
if (myWeekday %in% c("Saturday","Sunday")) {
  myGreeting <- "Have a nice weekend"
} else {
  myGreeting <- "Go back to work"
}
# Prints the message
myGreeting
```

The statement can be further extended to consider multiple conditions. It checks the conditions sequentially and returns once a condition is met. This is shown in example 2.9.2.

R Example 2.9.2

```
library(lubridate)
myWeekday <- weekdays(today())
# Checks multiple conditions
if (myWeekday %in% c("Saturday","Sunday")) {
  myGreeting <- "Have a nice weekend"
} else if(myWeekday == "Friday"){
  myGreeting <- "It's Friday!"
} else if(myWeekday == "Monday"){
  myGreeting <- "Oh no..."
} else {
  myGreeting <- "Go back to work"
}
myGreeting
```

2.9.2 While

The `while` statement loops as long as the condition stays `TRUE`. Example 2.9.3 demonstrates how a `while` loop can be implemented.

R Example 2.9.3

```
myCounter <- 100
while (myCounter > 0) {
  myCounter <- myCounter - 5
  print(myCounter)
}
```

Example 2.9.4 shows that an early exit can be implemented in a `while` statement using the `break` operator. The `next` operator can be used to skip one of the iterations.

R Example 2.9.4

```
myCounter <- 100
while (myCounter > 0) {
  myCounter <- myCounter - 5
  if (myCounter > 50) {
    # Skips all iterations if the counter value is greater than 50
    next
  }
  if (myCounter == 10) {
    # Early stop if the counter value matches 10
    break
  }
  print(myCounter)
}
```

2.9.3 For

`For` loops can be used to execute certain operations again and again. In example 2.9.5, the `for` loop calculates the sum expression $\sum_{i=1}^{100} i^2$ by running the code sequentially many times.

R Example 2.9.5

```
myResult <- 0
for (i in 1:100) {
  myResult <- myResult + i ^ 2
}
myResult
```

Exercise 4 Operational and Conditional Statements

- Write a for loop that runs through a list of numbers `1:20` and prints only the even numbers.
- Can you change this loop so it prints just the odd numbers?

2.10 Apply

Functions in the apply family can be used to run operations over multiple values. In example 2.10.1, the `apply()` function receives a `tibble` object and iterates over each row. Variables of each row are merged in a character message and returned.

R Example 2.10.1

```
# The second argument 1 inidicates iterate over rows
myMessages1 <- apply(myFavMovies2, 1, function(row){
  sprintf("%s was released in %s.", row["title"], row["year"])
})
myMessages1
```

There are several useful functions in the apply family. In example 2.10.2, the `lapply()` function receives an input and returns a `list` as output.

R Example 2.10.2

```
myMessages2 <- lapply(myFavMovies2$title,
                      function(x){ sprintf("%s is a great movie!", x) })
# Checks the data type of the result
typeof(myMessages2)
# Check whether it is a list
is.list(myMessages2)
# Select the 6th element of the list
myMessages2[[6]]
```

The `sapply()` function works in a similar way. It returns a vector as the result. Example 2.10.3 shows how it can be used.

R Example 2.10.3

```
myMessages3 <- sapply(myFavMovies2$title,
                      function(x){ sprintf("%s is a great movie!", x) })
# Check whether it is a list
is.list(myMessages3)
myMessages3
```

There are other useful members in the apply family, such as `vapply()`, `tapply()`, `mapply()`, `rapply()` and `eapply()`. Table 2.1 shows a brief summary of the functions.

Table 2.1: Functions in the apply family

Function	Brief Description
`apply()`	Apply function over a rows or columns of data frame.
`lapply()`	Apply function over a vector and returns a list.
`sapply()`	Apply function over a vector and returns a vector.
`vapply()`	Apply function over a vector and returns a fixed result.
`tapply()`	Apply function over a vector by groups.
`mapply()`	Multivariate version of `sapply()`.
`rapply()`	Recursive version of `lapply()`.
`eapply()`	Apply function over named values in an `environment`.

Vectorisation can speed up code very significantly, therefore it is always a good idea to vectorise code for maximum performance.

In `R`, looping can be slow because they are not vectorised. Moreover, `R` objects are stored in memory. If the operation in the loop is altering the size of an object,

it will force `R` to reallocate the object to a new memory address. For this reason, avoiding to modify object size in loops can always speed up the code.

In example 2.10.4, the `system.time()` function is used to capture the total time used to execute the code. The curly brackets `{}` wrap multi-line code in a single expression.

R Example 2.10.4

```
# Vectorised operation is fast
system.time({
  myResult <- 1:100000 * 2
})
# Looping is quite slow
system.time({
  myResult <- sapply(1:100000, function(x){ x * 2 })
})
# Appending to vector is much slower
system.time({
  myResult <- c()
  for(i in 1:100000){
    myResult <- c(myResult, i * 2)
  }
})
```

Exercise 5 Working with the apply family

- Write the for loop from the previous task, (runs through a list of numbers `1:20` and prints only the even numbers) as an apply function.
- (Advanced) Use the `sys.time()` function to find out if `apply` is faster than `for`.

Chapter 3

Data Transformation

The `tidyverse` is a coherent system of packages for data manipulation, exploration and visualisation that share a common design philosophy. These were mostly developed by the prolific R developer Hadley Wickham[1], but they are now being expanded by several contributors. The `tidyverse` packages are intended to make data scientists and statisticians more productive by guiding them through the workflow of a typical project. This is illustrated in figure 3.

You may begin the analysis by importing data into R. This typically means that you take data stored in a file, database, or web API and load it into R as an object known as a `data frame` or a `tibble`.

Once the data is imported, it is a good idea to tidy it. In brief, when your data is tidy, each column is a variable and each row is an observation. This is important because a consistent structure lets you focus your struggle on questions about the data, not fighting to get the data into the right form for different functions.

Once you have tidy data, the usual next step is to transform it. This includes narrowing in on observations of interest, creating new variables that are functions of existing variables and calculating summary statistics.

There are two main engines of knowledge generation: 1) visualisation and 2) modelling . These have complementary strengths and weaknesses so a good analysis will iterate between them many times.

Data visualisation is fundamentally a human activity. A good visualisation will show you things that you did not expect, or raise new questions about the data. It might also cast doubt on the research hypothesis or hint that you need to collect different data. Visualisation does not scale particularly well because it requires

[1] `http://hadley.nz`

human interpretation.

Models are complementary tools to visualisation. Once the research hypotheses are sufficiently precise, you can use a model to address them. Models are fundamentally mathematical or computational tools, so they generally scale well.

The last step of data science is communication, which is a critical part of any data analysis project. It does not matter how well your models and visualisation have led you to understand the data, unless you can also communicate your results to others.

Surrounding all these tools is programming. This is a tool that you use in every part of the analysis.

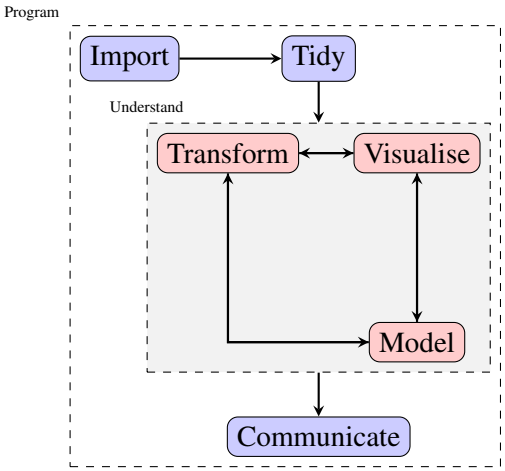


Figure 3.1: Stages of data analysis

This list provides an overview of the main `tidyverse` packages and how they fit into this typical workflow.

Import Reading datasets from various data sources.

- `readr`
- `readxl`
- `haven`
- `httr`
- `rvest`
- `xml2`

Tidy Clean up datasets.

- `tibble`
- `tidyr`

Transform Aggregate, change variable format and derive new variables.

- `dplyr`
- `forcats`
- `hms`
- `lubridate`
- `stringr`

Visualise Creating charts using the Grammar of Graphics.

- `ggplot2`

Model Train and test statistical models.

- `broom`
- `modelr`

Program Coding in pipeline-style.

- `magrittr`
- `purrr`

In most cases, data does not come to you in exactly the right format. Often you need to compute new variables, or to summarise and rename the original ones. In some cases, you may have to reorder the observations to make the data easier to work with. An excellent package for these tasks is the `dplyr` package. There are five key functions in this package that allow you to solve the vast majority of data manipulation challenges.

- Subset the observations by criteria - `filter()`
- Reorder the observations - `arrange()`
- Pick variables by name - `select()`
- Compute new variables as a function of existing variables - `mutate()`
- Collapse many values down to a single summary value - `summarise()` or `summarize()`

All aforementioned functions work similarly. The first argument is a `data frame` containing the source data. The subsequent arguments describe what to do with the data. All the functions return the processed data in a new `data frame`. These can all be used in conjunction with the `group_by()` function, which changes the scope of each function from operating on the entire dataset to operating on it group-by-group.

3.1 Filtering

You can use the function `filter()` to subset observations based on their values. The first argument is a `data frame`, while the subsequent arguments are the filtering expressions.

For filtering you can use the standard comparison operators (>, >=, <, <=, != (not equal), and == (equal)). To combine multiple filtering arguments, you can separate them with a comma or use the `&` symbol. In addition, logical operators can be used for more complex combinations.

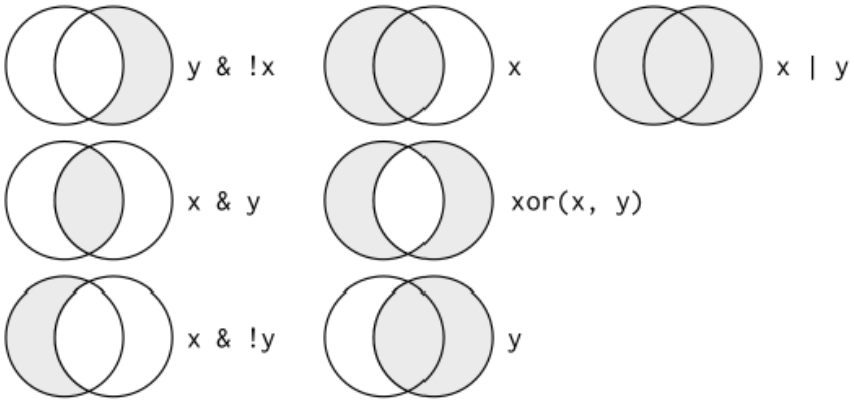


Figure 3.2: Logical operators in R

Missing values are represented in R as `NA`. They are contagious because any operation involving an unknown value will produce unknown results.

The function `filter()` only includes rows where the condition is `TRUE`. It excludes both `FALSE` condition and `NA` values.

You can use the function `is.na()` to determine whether a value is missing. You can also use this to explicitly include missing values in a `filter()` call.

In example 3.1.1, We will learn how to use `tidyverse` using a dataset on flights departing from New York City in 2013.

R Example 3.1.1

```
# load the required packages
library(dplyr)
library(nycflights13)
# Inspect the data, noting the type of each variable
flights
# View the flights dataset interactively
View(flights)
# Select all flights on January 1st
# Assigning them to a new variable jan1
jan1 <- filter(flights, month == 1, day == 1)
jan1
# Select all flights from November or December
filter(flights, month == 11 | month == 12)
# A useful shorthand for this problem is x %in% y
# It selects every row where x is one of the values in y
filter(flights, month %in% c(11, 12))
# Let's select only the flights that weren't delayed
# (on arrival or departure) by more than two hours:
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

Exercise 6 Filtering Observations

Find all flights in the `flights` dataset that:

1. Had an arrival delay of two or more hours.
2. Flew to Houston (IAH or HOU).
3. Were operated by United (`UA`), American (`AA`) or Delta (`DL`).
4. Departed in summer (July, August and September)
5. Arrived more than two hours late, but did not leave late.
6. Had a departure delay of least an hour, but made up over 30 minutes in flight.
7. How many flights have a missing `dep_time`? What other variables are missing?

3.2 Sorting

The function `arrange()` changes the order of observations in a dataset. It sorts the observations by a specified set of variable names[2] in ascending order.

If you provide more than one variable name, each additional variable will be used to break ties in the values of the preceding variable. Use `desc()` to reorder a variable in descending order. Missing values are always sorted at the end. Example 3.2.1 shows how the `arrange()` function can be used.

R Example 3.2.1

```
# Arrange flights by year, then month, then day
arrange(flights, year, month, day)
# Use desc() to reorder by a column in descending order
arrange(flights, desc(arr_delay))
# Missing values are always sorted at the end:
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
arrange(df,desc(x))
```

Exercise 7 Arranging Observations

Sort the `flights` dataset to find:

1. The fastest (in terms of flight time) flights.
2. The fastest (in terms of average speed in mph).
3. The flights that departed the most ahead of time.
4. The flights that were most delayed on departure.
5. Which flights travelled the furthest?

3.3 Subsetting Variables

It is not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you are actually interested in.

[2] More complicated expressions can also be used for ordering.

The `select()` function allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

There are a number of helper functions you can use within `select()`. These are also shown in example 3.3.1:

1. `starts_with("abc")` matches variable names that begin with `"abc"`.
2. `ends_with("xyz")` matches variable names that end with `"xyz"`.
3. `contains("ijk")` matches variable names that contain `"ijk"`.
4. `num_range("x", 1:3)` matches `x1`, `x2`, and `x3`.

The function `select()` can in principle be used to rename variables, but it drops all of the variables not explicitly mentioned. Therefore it is better to use the `rename()` function which keeps all variables not explicitly mentioned.

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you want to move several variables to the start of the `data frame`.

R Example 3.3.1

```
# Select columns by name
select(flights, year, month, day)
# Select all columns between 'year' and 'day' (inclusive)
select(flights, year:day)
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
# Rename a variable using rename()
rename(flights, tail_num = tailnum)
# Select all columns starting with "sched"
select(flights, starts_with("sched"))
# Reorder columns using the everything() helper
select(flights, time_hour, air_time, everything())
```

Exercise 8 Selecting Variables

1. Can you find two ways to select `dep_time`, `dep_delay`, `arr_time` and `arr_delay` from the `flights` dataset in one line of code?
2. What happens if you include the name of a variable multiple times in a `select()` call?

3. 3) Run the following code: `select(flights, contains("TIME"))`. Note the result. What happens when you add the argument `ignore.case = TRUE` in the `contains()` function? What does this tell you about this helper function?

3.4 Compute Variables

In the data transformation process, it is often useful to compute new variables that are functions of existing ones. For this we can use the `mutate()` function. This function always adds new variables at the end of the dataset. Alternatively, you can use the `transmute()` function if you only want to keep the newly-computed variables and remove the old ones. For both `mutate()` and `transmute()`, you can refer to columns that you have just created in the same function call.

There are many useful creation functions you can use with `mutate()` to create new variables:

Arithmetic operators The operators +, −, ⋆, /, ^ are all vectorised using the so-called 'recycling rules' (i.e. if one parameter is shorter than the other, it will be automatically extended to be the same length.)

Modular arithmetic `%/%` for integer division (discards remainder) and `%%` for remainder only (modulo).

Logs Very useful for dealing with data that ranges across multiple orders of magnitude.

Logical comparison <, <=, >, >=, !=

Ranking There are several of these – the most common one is `min_rank()` which does the most usual type of ranking (e.g. first, second, third, fourth) and gives the smallest values the smallest ranks.

In example 3.4.1, we will start by creating a narrower dataset so we can see the new variables.

R Example 3.4.1

```
# Select several columns only
flights_sml <- select(flights,
                      year:day,
```

```
                      ends_with("delay"),
                      distance,
                      air_time)
# Use this smaller data frame to derive new columns
mutate(flights_sml,
       gain = arr_delay - dep_delay,
       speed = distance / air_time * 60)
# You can refer to columns that you have just created
mutate(flights_sml,
      gain = arr_delay - dep_delay,
      hours = air_time / 60,
      gain_per_hour = gain / hours)
# Keep only the new variables using transmute()
transmute(flights,
          gain = arr_delay - dep_delay,
          hours = air_time / 60,
          gain_per_hour = gain / hour)
# Modular arithmetic: modulo
7 %% 2
# Modular arithmetic: integer division
7 %/% 2
# Compute hour and minute from dep_time
transmute(flights,
          dep_time,
          hour = dep_time %/% 100,
          minute = dep_time %% 100)
# Example of ranking
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)
```

Exercise 9 Computing New Variables

1. Currently `dep_time` and `sched_dep_time` are convenient to look at but hard to compute with because they're not really continuous numbers. Convert them to a more convenient representation of number of minutes since midnight.

2. Compare `air_time` with `arr_time - dep_time` (you will need to convert these to units of minutes after midnight - see Q1). What do you expect to see? What problem do you see? Why might this be?

3. What does `1:3 + 1:10` return and why? How about `1:3 + 1:9`?

3.5 Summarising

The last key function is `summarise()` or `summarize()`. It collapses a set of values into one. This function is particularly useful when used in conjunction with `group_by()`. This changes the unit of analysis from the whole dataset to

individual groups. Then you can use functions on the grouped `data frame` in order to obtain grouped summaries. These summary functions can be used in the `summarise()` function:

Measures of location Arithmetic average `mean()` and median `median()`.

Measures of spread Standard deviation `sd()` and interquartile range `IQR()`.

Measures of rank Minimum value `min()`, maximum value `max()` as well as the quantiles `quantile()`

Measures of position `first()` and `last()`

Whenever doing any aggregation, it is always a good idea to include either the total count `n()` or the count of non-missing values `sum(!is.na(x))`. To count the number of unique values, use `n_distinct()`. By including count statistics, you can make sure that you are not drawing conclusions based on small amount of data.

When combining several operations, it is usually better to join them together using the pipe operator `%>%` rather than repeatedly making new variables. This is illustrated in example 3.5.1.

R Example 3.5.1

```
# Example of summarise() function alone
# You can also use summarize() - they are equivalent
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
# Combining summarise() with group_by() and a dplyr verb
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
# Multiple operations without a pipe
by_dest <- group_by(flights, dest)
delay <- summarise(by_dest,
                   count = n(),
                   dist = mean(distance, na.rm = TRUE),
                   delay = mean(arr_delay, na.rm = TRUE))
filter(delay, count > 20)
# There are three steps to prepare the flight delay data:
# 1) Group flights by destination
# 2) Summarise average distance, delay and number of flights
# 3) Filter to remove noisy points
# They can be chained together using pipeline '%>%'
mySummary <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)) %>%
  filter(count > 20)
# Using summarise() to measure standard deviation
```

```
# Distance to some destination has larger spread than others
flights %>%
        group_by(dest) %>%
        summarise(dist_sd = sd(distance, na.rm = TRUE)) %>%
        arrange(desc(dist_sd))
```

Exercise 10 Grouped Summaries

1. Which carrier has the worst delays?
2. For each destination, compute the total minutes of delay. For each flight, compute the proportion of the total delay for its destination.

Chapter 4

Regression Models

The regression model is one of the most widely used statistical techniques in the realm of supervised learning. It quantifies the relationship between a dependent variable Y and an independent variable X, where $Y \subseteq \mathbb{R}$ and $X = \{x_1, x_2, ...x_M\}$.

4.1 Linear Regression

Linear regression[1] is a very popular parametric statistical technique. It can quantify the effects of each input variable. It also informs users whether the effects are statistically significant or not.

In a univariate scenario where x is the only input, it provides the best fit for the model equation $\hat{y}_i = \beta_0 + \beta_1 x_i$. The parameter β_0 is normally referred to as the intercept, while the β_1 value is called the slope.

The simple linear model can be extended to include a high-order polynomial term of a variable x. It provides higher model flexibility so that the linear model fits the data better. For example, an M^{th} order polynomial term has been fitted to the dataset on the next chart. The choice of M is subjective but usually a small value of M is desirable because it helps avoid overfitting.

The model assumes that the model residuals $\epsilon_i = y_i - \hat{y}_i$ are drawn from Gaussian distribution (i.e. having a bell-shaped curve). The goal of linear regression is to minimise the sum of squared residuals.

In the R language you can call the function `lm()` to perform linear regres-

[1] Also called ordinary least square (OLS) regression.

sion. It requires at least two arguments (`data` and `formula`). For example, `lm(y ~ x , myData)` will perform a simple univariate linear model using the independent variable x to predict the dependent variable y in the data frame object `myData`.

Dummy variables can be easily created from categorical labels. Users can simply use the syntax `lm(y ~ x1 + x2, myData)` where `x1` is a numeric variable and `x2` either a factor or a character variable The `lm` function will automatically create dummy variables on-the-fly. Normally the first category in the column will be used as reference level.

Simple linear models are often not flexible enough to model complex variable effects. In light of this, linear models can be made more flexible by including polynomial terms. This can be expressed as `lm(y~poly(x1,3)+x2,myData)`. In this case, we are defining a cubic relationship with variable `x1` and a linear relationship with variable `x2` (Five coefficients in total will be estimated, four from the regression terms plus one from the intercept). Such a model can be expressed as equation (4.1).

$$\hat{y}_i = \underbrace{\beta_0}_{\text{Intercept}} + \underbrace{\beta_1 x_{1,i} + \beta_2 x_{1,i}^2 + \beta_3 x_{1,i}^3}_{\text{Cubic polynomial term}} + \underbrace{\beta_4 x_{2,i}}_{\text{Linear term}} \tag{4.1}$$

Interaction refers to the combined effect[2] of more than one independent variables. For example, the independent variables `x1` and `x2` might have no effect on the dependent variable y alone. However, the effect on y can become prominent when these two variables are combined. In R language you can use the syntax `lm(y ~ x1*x2)` to represent the relationship. The function `I(x1*x2)` can be used to supress the interaction term and the arguments will be treated as simple arithmetic operations.

Exercise 11 Simple Linear Regression

In this exercise, we are going to predict car efficiency using the `mtcars` teaching dataset. The dataset is embedded in open source R and can be called directly by `mtcars`. In example 4.1.1,you can peek at the dataset by executing `head(mtcars)` or `tail(mtcars)`. You can also read the dataset definition by executing the command `?mtcars`.

[2] Also known as synergy effect.

R Example 4.1.1

```
#Load the dataset into your local environment.
data(mtcars)
# Browse the top few rows
head(mtcars)
# Browse the last few rows
tail(mtcars)
```

Before running any models, we can explore the dataset a bit further through visualisation. The R package `ggplot2` is a very popular visualisation add-in. It builds charts by stacking layers of graphics on it. Example 4.1.2 shows how you can use `ggplot2` to visualise the dataset.

R Example 4.1.2

```
# Create a simple histogram using base R plot
hist(mtcars$mpg)
# Using dplyr pipeline style code (equivalent output)
library(dplyr)
mtcars$mpg %>% hist()
# Plot histogram using ggplot2 package
library(ggplot2)
mtcars %>%
  ggplot(aes(x=mpg)) +
  geom_histogram() +
  labs(x="Miles-per-gallon",
       y="Count",
       title="Histogram showing the distribution of car performance")
# Scatterplot showing the relationship between mpg and wt
library(ggplot2)
mtcars %>%
  ggplot(aes(x=wt, y=mpg, colour=factor(cyl))) +
  geom_point() +
  labs(x="Weight",
       y="Miles-per-gallon",
       colour="Number of cylinders",
       title="Scatterplot showing car weight against performance")
# Create a boxplot showing mpg distribution of different gear types
mtcars %>%
  ggplot(aes(x=factor(am,
                      levels = c(0,1),
                      labels = c("Automatic", "Manual")), y=mpg)) +
  geom_boxplot() +
  labs(x="Gear type",
       y="Miles-per-gallon")
# Draw a scatterplot with facets to visualise multiple variables
mtcars %>%
  ggplot(aes(x=hp, y=mpg, colour=factor(gear), size=disp)) +
  geom_point() +
  facet_grid(. ~ cyl) +
  labs(x="Horsepower",
       y="Miles-per-gallon",
       colour="Number of gears",
       size="Displacement")
# Draw a matrix scatterplot
pairs(mtcars)
# Load the package GGally
# It is an extension of the ggplot2 package
# Use ggpairs to draws a prettier matrix scatterplot
library(GGally)
ggpairs(mtcars)
```

Now let us try to analyse car efficiency using the `mpg` column as dependent variable. The hypothesis is that heavier cars have lower miles-per-gallon. We can investigate this by building a univariate linear model using the `lm()` function and analyse the results. The following code fits a univariate regression model. The function `summary(myModel1)` prints out all the key results of the model. This is demonstrated in example 4.1.3.

R Example 4.1.3

```
# Build a univariate linear model
myModel1 <- lm(mpg ~ wt, mtcars)
# Read the model summary
summary(myModel1)
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.5432 -2.3647 -0.1252  1.4096  6.8727
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
## wt           -5.3445     0.5591  -9.559 1.29e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.046 on 30 degrees of freedom
## Multiple R-squared:  0.7528,Adjusted R-squared:  0.7446
## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

The model summary contains a lot of useful information. Table 4.1 lists the key items alongside a short description.

Table 4.1: Summary of key model information

Term	Description
Residuals	This is the unexplained bit of the model, defined as observed value minus fitted value ($\epsilon = y_i - \hat{y}_i$). If the model's parametric assumption is correct, the mean and median values of the residuals should be very close to zero. The distribution of the residuals should have equal tails on both ends.
Estimate	Coefficient of the corresponding independent variable (i.e. the β_m values).
Standard error	Standard deviation of the estimate.
t-value	The number of standard deviations away from zero (i.e. the null hypothesis).
$P(>\|t\|)$	p-value of the model estimate. In general, variables with p-value below 0.05 are considered statistically significant.
Multiple R^2	Pearson's correlation squared which indicates strength of relationship between the observed and fitted values.
Adjusted R^2	Adjusted version of R^2.
F-statistic	Global hypothesis for the model as a whole.

You can also build more complex multivariate models using the same `lm()` function Example 4.1.4 shows how the function deals with nominal and ordinal variables. You can either force the variable to become categorical by explicitly state `factor(myVar)` in the formula. Alternatively, if the variable already belongs to the `factor` data type, the `lm()` regression function would handle it automatically without having to state it in the formula.

R Example 4.1.4

```
# Dummy variables are automatically created on-the-fly.
# The variable am has two categories
myModel2 <- lm(mpg ~ wt + hp + qsec + factor(am), mtcars)
summary(myModel2)
# Build a multivariate linear model with polynomial terms.
# Interaction effect can be added as well
myModel3 <- lm(mpg ~ wt + qsec + factor(am) +
                 factor(cyl) * disp + poly(hp, 3) +
                 factor(gear), mtcars)
summary(myModel3)
```

To further analyse the effects of individual variables, we can load the `car` package and use the function `avPlots()` to view the partial regression plots[3]. This would graphically display the effect of individual variables while keeping others in control. This is shown in example 4.1.5.

R Example 4.1.5

```
library(car)
avPlots(myModel2)
```

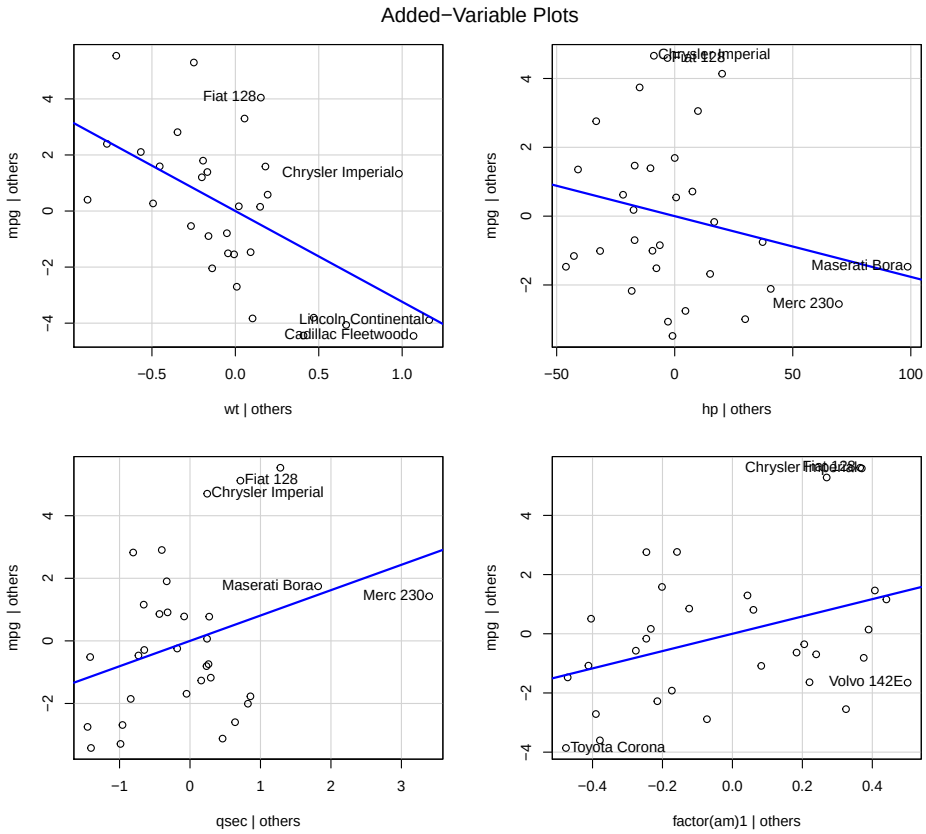


Figure 4.1: Partial regression plots

You can also compare nested models[4] using ANOVA technique. Example 4.1.6 compares three nested models by applying the Chi-square test on the model residuals

[3] It is also called the added-variable plot.

[4] Refers to models having additional predictor terms. For example, $\hat{y} = x_1 + x_2 + x_3$ and $\hat{y} = x_1 + x_2 + x_3 + x_4$ are both nested models of $\hat{y} = x_1 + x_2$.

to check for statistically significant differences. In most cases, the simplest model is prefereable if the candidate models are not significantly different.

R Example 4.1.6

```
# Compare linear regression models using Chi-square test
# Testing whether myModel2 and myModel3 are different from myModel1
anova(myModel1, myModel2, myModel3, test="Chisq")
```

Exercise 12 Regression Diagnostics

Linear regression is a parametric statistical method which has strong underlying assumptions. Analysing the model's diagnostic measurements would help us assess whether the assumptions are sufficiently met. You may use the `plot()` function to create a series of regression diagnostic plots. The command in example 4.1.7 generates several diagnostic plots for a standard linear regression model object.

R Example 4.1.7

```
plot(myModel3)
```

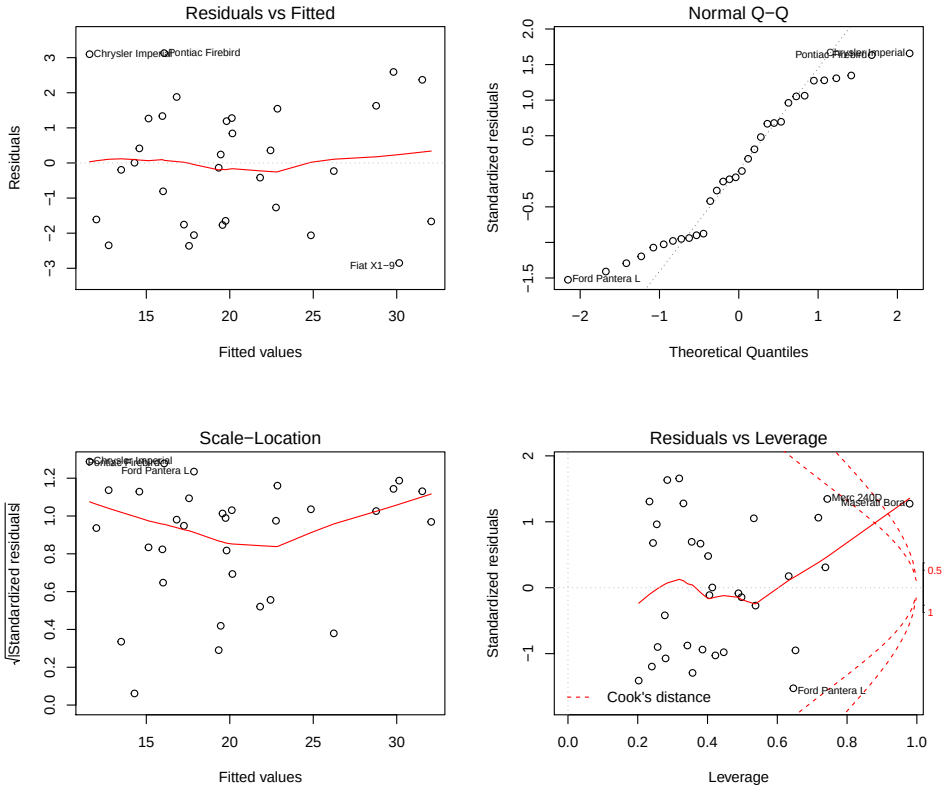


Figure 4.2: Regression diagnostic plots

Residuals vs Fitted Checks for non-linear relationship. For a good linear fit, residuals are normally distributed along a straight line centred at zero (i.e. a horizontal line). Contrarily, curvy lines indicate poor model fit with possible non-linear effects.

Normal Quantile-Quantile One of the main assumptions of linear regression is that the residual is drawn from a zero-centred Gaussian distribution $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. To verify whether the proposed model satisfies this assumption, we can use a normal quantile-quantile plot (Q-Q plot) to perform a quick check. It aligns the model residuals against a theoretical normal distribution. If the residuals spread along a straight line on the Q-Q plot, it suggests that the residuals are normally distributed. Alternatively, if the data points deviate from the line it indicates vice versa. In this case, the parametric model assumption does not hold and you might have to consider improving your model.

Scale-Location Shows the distribution of the standardised residuals along the

range of fitted values. As the standard linear model is assumed to be homoscedastic, the residual variance should not vary along the range of fitted values (i.e. expect a near-horizontal line). If the standardised residuals form a distinguishable pattern (e.g. fanning out or curvy), then the model may be heteroscedastic and hence violate the underlying assumption.

Residual vs Leverage (Cook's Distance) Observations having high leverage pose greater influence to the model. This means that the model estimates are strongly affected by these cases. If such obserations have high residuals (i.e. large Cook's distance), they can sometimes be considered as outliers. On the other hand, most observations would have low leverage and short Cook's distance. This means that the model estimates would not have varied a lot if few such observations were to be added or discarded.

Exercise 13 Model Overfitting

Linear regression can be made more flexible by increasing the order of the polynomial terms. This allows the linear model to capture non-linear effects. However, a flexible model also risks overfitting the data. This means that the model might appear to have very good fit during training, but it may fit poorly when it is tested on new data. In general, overfitted models have very little inferential power and are ungeneralisable. The code snippet in example 4.1.8 runs a linear model with variable level of flexibility.

R Example 4.1.8

```
# Bivariate linear model with polynomial terms
# You can change the values here.
J <- 3
K <- 2
myModel4 <- lm(mpg ~ poly(wt,J) + poly(hp,K), mtcars)
summary(myModel4)
# Create the base axes as continuous values
wt_along <- seq(min(mtcars$wt), max(mtcars$wt), length.out = 50)
hp_along <- seq(min(mtcars$hp), max(mtcars$hp), length.out = 50)
# Use the outer product of wt and hp to run predictions for every
# point on the plane
f <- function(k1, k2, model){ z <- predict(model, data.frame(wt=k1, hp=k2 )) }
myPrediction <- outer(wt_along, hp_along, f, model = myModel4)
# Draw the 3D plane
myPlane <- persp(x = wt_along, xlab = "Weight",
                 y = hp_along, ylab = "Horsepower",
                 z = myPrediction, zlab = "Miles-per-Gallon",
                 main = "Fitted vs observed values in 3D space",
                 theta = 30, phi = 30, expand = 0.5, col = "lightblue")
# Add original data as red dots on the plane
myPoints <- trans3d(x = mtcars$wt,
                    y = mtcars$hp,
                    z = mtcars$mpg,
                    pmat=myPlane)
points(myPoints, col="red")
```

$$\hat{y} = \beta_0 + \sum_{j=1}^{3} \beta_{wt_j} x_{wt}^j + \sum_{k=1}^{2} \beta_{hp_k} x_{hp}^k$$

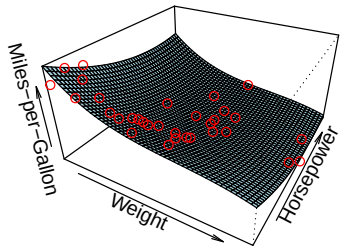


Figure 4.3: A less flexible model showing better generalisability

$$\hat{y} = \beta_0 + \sum_{j=1}^{8} \beta_{wt_j} x_{wt}^j + \sum_{k=1}^{5} \beta_{hp_k} x_{hp}^k$$

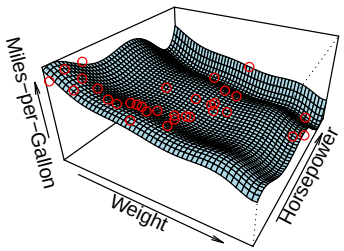


Figure 4.4: A more flexible model illutrating the risk of overfitting

4.2 Poisson Regression

In the previous section, the simple linear regression model assumes that the dependent variable y follows a Gaussian distribution which spans the range $(-\infty, +\infty)$. Yet sometimes we would like to estimate the number of discrete events which often a positive integer ($\mathbb{N} = \{0, 1, 2, 3, ...\}$). In this case, Poisson regression can be used. It is based on the Poisson distribution[5] which can be found in everyday life. The following are typical examples:

- Number of children in a household.
- Number of bank notes in a wallet.

The Poisson regression model can take into account multiple predictor variables. Equation (4.2) shows a Poisson regression model with $\hat{y}$ as the dependent variable and M predictor variables.

$$\hat{y} = e^{\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_M x_M} \tag{4.2}$$

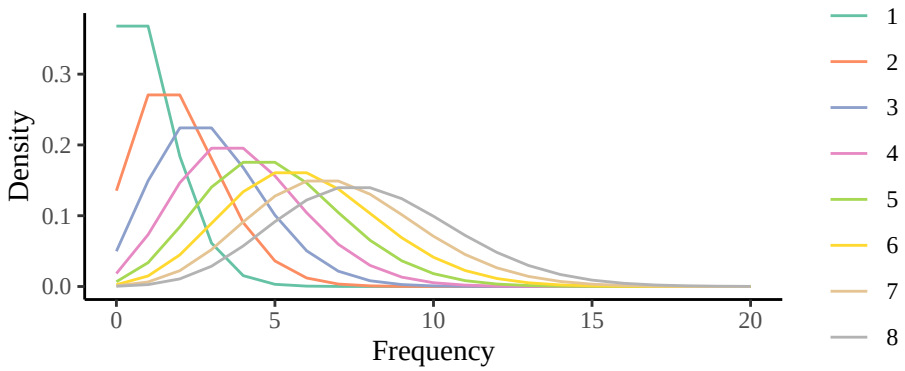


Figure 4.5: Poisson distribution with different λ values

Exercise 14 Testing for Poisson Distribution

Using the `mtcars` dataset, we can estimate the number of carburetors (i.e. the variable `carb`) in different cars. In example 4.2.1, you can use the command `hist(mtcars$carb)` to draw a simple histogram. You should find that this

[5] A Poisson distribution is defined by a single parameter $y \sim Poisson(\lambda)$, where the mean μ and variance σ^2 are equal. i.e. $\lambda = \mu$ and $\lambda = \sigma^2$

variable 1) never goes below zero and 2) has a long but thin tail towards the positive side . These are key signatures of a Poisson distribution.

R Example 4.2.1

```
# Draw a simple histogram.
hist(mtcars$carb)
# Compute the mean and variance.
mean(mtcars$carb)
var(mtcars$carb)
```

To robustly check whether a variable is truly drawn from a Poisson distribution, you can perform a Chi-squared goodness-of-fit test. It fits the input data against a theoretical Poisson distribution. Example 4.2.2 visualises the results. The vertical bars would fill the positive space if the input data fitted well against the Poisson distribution. This can be statistically examined by analysing the p-value of the Chi-square test. If the p-value is small enough, we can then accept the hypothesis that the variable is drawn from a Poisson distribution.

R Example 4.2.2

```
# Performs the Chi-squared goodness-of-fit test.
# It checks whether the variable is drawn from a Poisson distribution.
library(vcd)
gf <- goodfit(mtcars$carb, type= "poisson", method= "ML")
# Plots the observed frequency vs theoretical Poisson distribution.
# The hanging bars should fill the space if it is perfectly Poisson.
plot(gf)
# Checks the statistical p-value of the goodness-of-fit test.
# If p<=0.05 then it is safe to say that the variable is Poisson.
summary(gf)
```

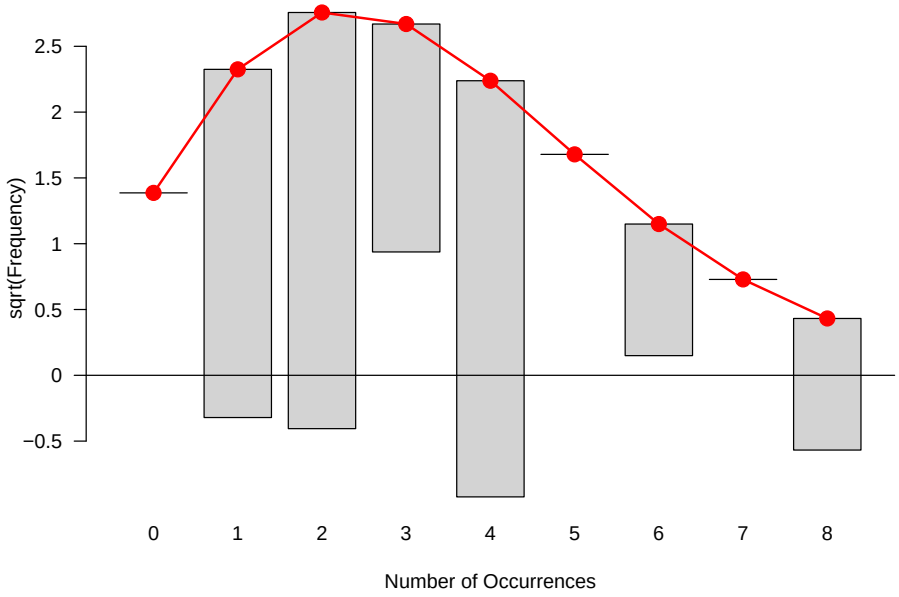


Figure 4.6: Goodness-of-fit test for Poisson distribution

Exercise 15 Building a Poisson Model

Example 4.2.3 trains a Poisson regression model using the `mtcars` dataset. The variable `carb` is used as the dependent variable while `hp`, `wt` and `am` are used as independent predictor variables. The regression output can be interpreted in a similar way as the ones from the linear model.

R Example 4.2.3

```
# Build a Poisson model to predict the number of carburetors in a car.
myPoissonModel <- glm(carb ~ hp + wt + factor(am),
                      family="poisson",
                      data=mtcars)
# Read the model summary
summary(myPoissonModel)
# Read the model diagnostic
plot(myPoissonModel)
# Visualise the observed / fitted values as a table
tibble(observed = mtcars$carb,
       fitted = myPoissonModel$fitted.values) %>% View()
```

4.3 Logistic Regression

Logistic regression can be used if the dependent variable is binary. This refers to when the outcome can either be Y or $\neg Y$. The model estimates outcome likelihood using the logistic function as outlined in equation (4.3). The logistic function transforms a real-valued number X into the range $(0, 1)$ which represents the outcome probability $P(Y) \in (0, 1)$.

$$P(Y) = \frac{1}{1 + e^{-X}} \tag{4.3}$$

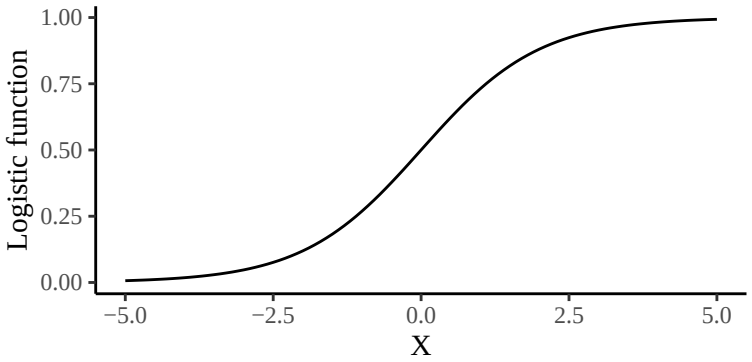


Figure 4.7: Graph showing the range of a logistic function

In a univariate scenario, the logistic function can be expressed as an equation (4.4), where β_0 represents the intercept while $\beta_1, \beta_2, \beta_3, ..., \beta_M$ refer to the regression coefficients of the variables $x_1, x_2, x_3, ..., x_M$. The output $P(Y)$ indicates the likelihood of true outcome.

$$P(Y) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_M x_M)}} \tag{4.4}$$

One of the most powerful features of logistic regression is the odds ratio. It quantifies the effects of each independent variable. It is a value indicating the change of likelihood of the event when an independent predictor variable is increased by 1 unit. The odds ratio is defined in equation (4.5).

$$OR(x_1) = \frac{odds(x_1 + 1)}{odds(x_1)} = \frac{e^{\beta_0 + \beta_1 (x_1 + 1) + \beta_2 x_2 + ... + \beta_M x_M}}{e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_M x_M}} = e^{\beta_1} \tag{4.5}$$

Logistic regression can only handle binary classification problems. If the dependent variable Y has more than two outcomes, we can use another algorithm called multinomial logistic regression[6]. Such problem can also be analysed using artificial neural networks in a much more sophisticated way which we will cover in a later section.

Exercise 16 Building a Logistic Model

In this exercise, we will continue to use the `mtcars` dataset. We will build a logistic regression model to predict whether the vehicle has automatic or manual transmission system (using the `am` variable as dependent variable).

In the example 4.3.1, the model has three independent variables `mpg`, `hp` and `disp`. You may run the example code to build the logistic regression model. You may also calculate the odds ratios and analyse the effects of each variable. The last part of the code is to calculate the model accuracy.

R Example 4.3.1

```
# Build a logistic regression model to predict the dependent variable am
# (1=manual; 0=auto)
myLogisticModel <- glm(am ~ mpg + hp + disp, family="binomial", data=mtcars)
summary(myLogisticModel)
# Calculate the odds-ratios by taking the exponential of the coefficients
# Can you explain the effects of each of these independent variables?
exp(myLogisticModel$coefficients)
# You may also calculate the 95% confidence interval of the odds-ratio
exp(cbind(oddsratio=myLogisticModel$coefficients, confint(myLogisticModel)))
# Returns the modelled probability
myProb <- predict(myLogisticModel, mtcars, type="response")
# Turn the probability into a binary class (i.e. TRUE vs FALSE)
# Probability > 0.5 means the vehicle likely to have manual transmission
myPrediction <- myProb > 0.5
# Construct a contingency table to check correct & incorrect predictions
table(myPrediction, observed=(mtcars$am == 1))
# Calculate model accuracy
# (defined as the percentage of correct prediction)
myAccuracy <- sum(myPrediction==(mtcars$am == 1))/nrow(mtcars)
myAccuracy
```

The logistic model described in example 4.3.1 can be expressed as equation (4.6).

$$\begin{aligned} P(manual) &= \frac{1}{1+e^{-(\beta_0+\beta_1 x_{mpg}+\beta_2 x_{hp}+\beta_3 x_{disp})}} \\ P(automatic) &= P(\neg manual) \\ &= 1 - P(manual) \end{aligned} \tag{4.6}$$

[6] `http://www.ats.ucla.edu/stat/r/dae/mlogit.htm`

Chapter 5

Tree-based Methods

Tree-based algorithms belong to the supervised learning discipline. For a given set of labelled objects, decision tree can produce a set of sequential prediction rules based on categorical and numeric predictor variables. It is based on the concept of prediction region (denoted as $\mathcal{R}_i$) which refers to a subset of the original object space. Objects situation within the same region share the same prediction.

At the heart of tree-based method is a concept called binary recursive partitioning. It is a top-down process which starts from initial object space. At the first recursion, the algorithm would split the master region $\mathcal{R}_1$ into two at a cut-off point. This produces two corresponding new regions $\mathcal{R}_2 \subseteq \mathcal{R}_1$ and $\mathcal{R}_3 \subseteq \mathcal{R}_1$ with two distinct prediction values. The cutoff point s determines where to slice the master region. All objects within $\{X|X_1 < s\}$ belong to $\mathcal{R}_2$ and those with $\{X|X_1 \geq s\}$ belong to $\mathcal{R}_3$. This process runs recursively until it hits the termination criteria.

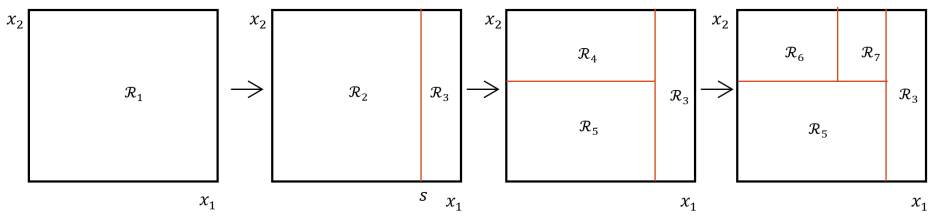


Figure 5.1: Recursive partitioning

5.1 Decision Trees

Decision tree is the simplest form of any tree-based model. It uses standard recursive partitioning to produce prediction regions. Decision tree is a very generic algorithm which fits both regression and classification problems. This means it can predict both continuous real numbers and discrete categories depending on the type of problem. The region prediction for a classification tree is decided by the majority class, while the prediction for a regression tree is defined as the simple average of all members within the region. The tree-splitting structure is called the topology, which can be interpreted graphically in most cases.

On the downside, recursive partitioning tends to produces very complex trees which may overfit the data. Various control parameters can be used to mitigate the risk of overfitting. For example, recursion can terminate once all regions are small enough to contain less than 5 objects.

Exercise 17 Growing a Decision Tree

In the R language, there are many packages which implement tree-based algorithm. In exercise 5.1.1, we will use the `rpart` function in the rpart package to build a simple decision tree for a regression problem. The aim of this exercise is to predict car efficiency (`mpg` variable) using the `mtcars` dataset.

R Example 5.1.1

```
# Load the rpart package for recursive partitioning
# Load the rpart.plot package for tree visualisation
library(rpart)
library(rpart.plot)
# Build a decision tree to predict mpg
myTree <- rpart(formula = mpg ~ wt + hp + factor(carb) + factor(am),
                data = mtcars,
                control = rpart.control(minsplit=5))
# Read the tree topology
myTree
# Read the detailed summary of the tree
summary(myTree)
# Visualise the decision tree
rpart.plot(myTree)
```

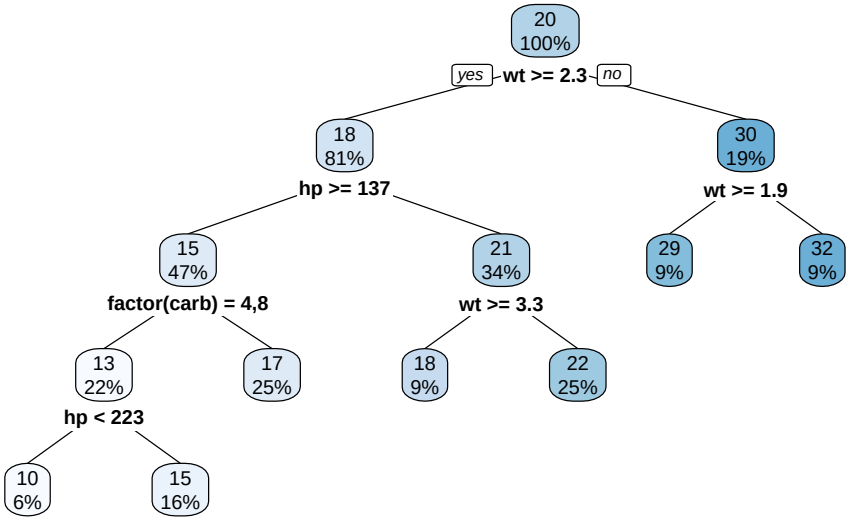


Figure 5.2: Decision tree for a regression problem

Exercise 18 Tree Pruning

It is a common practice to grow a complex decision tree first, and then decide how to prune it afterwards. Removing weaker branches of the tree usually enhances the model's predictive power which eventually makes it more generalisable.

In exercise 5.1.2, the command `printcp(myTree)` returns the relative error of the tree at each and every node. It is defined as $1 - R^2$ and therefore always starts with 1 at the top level. As the tree grows, the R^2 value increases and approaches 1 therefore the corresponding relative error will diminish towards zero. You can use the function `prune()` to remove weak branches. The complexity parameter `cp` refers to the amount of error reduced when a region is split. We can specify a threshold `cp` value so that branches with weak predictive power can be pruned.

R Example 5.1.2

```
# View the cp values
plotcp(myTree)
printcp(myTree)
# Prune the tree at a certain threshold cp value
# You can change the threshold value
?prune
myNewTree <- prune(myTree, cp = 0.03)
rpart.plot(myNewTree, fallen.leaves = FALSE)
```

5.2 Random Forest

Random forest is a collection of many tiny decision trees. All trees in the forest are trained using the same dataset, but with randomly selected predictor variables. In a random forest with P independent variables, only $p < P$ variables are randomly selected at each split. Such randomess causes variation among the trees. Some trees will have strong prediction power while some others will be weaker.

Once all the trees are grown, the random forest algorithm combines the output of all trees and uses the simple average as prediction value if it is regression problem. Alternatively if it is a classification problem, the majority label of the region becomes the prediction value.

It is widely recognised that the prediction accuracy of random forest is far better than that of an individual decision tree. However, as a trade-off, random forest is often harder to interpret manually as the decision rule becomes more complicated.

Exercise 19 Planting a Random Forest

Example 5.2.1 demonstrates how to train a random forest model. Each tree in the forest would randomly select a few variables for assessment. You can use the `randomForest` package to build random forest rapidly.

The importance of each predictor variable is indicated by the decrease of node impurity. A powerful predictor would substantially decrease node impurity. For regression, it is measured by the residual sum of squares. For classification, the node impurity is measured by the Gini index. You can use the function `importance(myForest)` or `varImpPlot(myForest)` to calculate the importance measurement.

R Example 5.2.1

```
library(randomForest)
library(dplyr)
# Build a random forest with 1000 trees
# Each tree has 2 randomly selected variables
# You can change the parameters
myForest <- randomForest(mpg ~ wt + hp + carb + am,
                         ntree = 1000,
                         mtry = 2,
                         data = mtcars %>% mutate(carb = factor(carb),
                                                  am = factor(am)))
# Plot the error as the forest expands
plot(myForest)
# Plot the distribution of tree size
treesize(myForest) %>% hist()
# Model summary
myForest
# Relative importance of each independent variable
importance(myForest)
varImpPlot(myForest)
```

Chapter 6

Neural Networks

Artificial neural networks (ANN) are mathematical algorithms inspired by the structure of the biological brains. It processes incoming information through a non-linear mechanism in a neuron and passes on the output to another neuron. When this process repeats many times via multiple layers of neurons, it becomes an artificial neural network. Neural networks having a complex structure are usually trained iteratively using backpropagation techniques over a long period of time with massive computational power. Nowadays, many modern applications are based on state-of-the-art neural networks, such as video analysis, speech recognition, chatbots and machine translation.

In an ANN, each hidden neuron carries a non-linear activation function $f(x)$. The sigmoid function is a traditional choice of activation function for ANNs(6.1a). It takes the weighted sum of input plus the bias unit and squashes everything into the range $(0, 1)$ with a characteristic sigmoidal 'S' shape. As the sigmoid function is differentiable and easy to compute, it soon became a popular choice for the ANN activation function. However, it suffers from a weak gradient when the input is far away from zero (i.e. the neuron saturates), which makes the ANN learn very slow.

To address the problem of weak gradient, alternative activation functions have been proposed. For instance, the hyperbolic tangent function can be used(6.1b). It shares the same sigmoidal shape but further stretches the output to the range $(-1, 1)$, therefore providing stronger gradient. Yet, the gradient still suffers from saturation when the input is too small or too large.

Different activation functions can provide stronger gradients while maintaining non-linearity. For instance, the softplus function has a strong gradient (i.e. unsaturable) for any positive input (6.1c). However, it has been considered computationally costly as it contains logarithmic and exponential terms. In light of this, a simplified version call rectified linear unit (ReLU) is usually used instead

(6.1d). The shape of ReLU is very similar to softplus with the exception that it has a threshold at zero. This means only positive input can lead to activation. However, the weighted sum input can change to a negative value during training, therefore causing the ReLU neuron to cease training. This is called the dying ReLU problem. To avoid this, more advanced activation functions incorporate a very small gradient in the negative range to allow the neuron to recover. The output of common activation functions are visualised in figure 6.1.

Sigmoid activation

$$f(x) = \frac{1}{1 + e^{-x}} \tag{6.1a}$$

Hyperbolic tangent activation

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{6.1b}$$

Softplus activation

$$f(x) = \ln(1 + e^{x}) \tag{6.1c}$$

Rectified linear unit (ReLU)

$$f(x) = \max(0, x) \tag{6.1d}$$

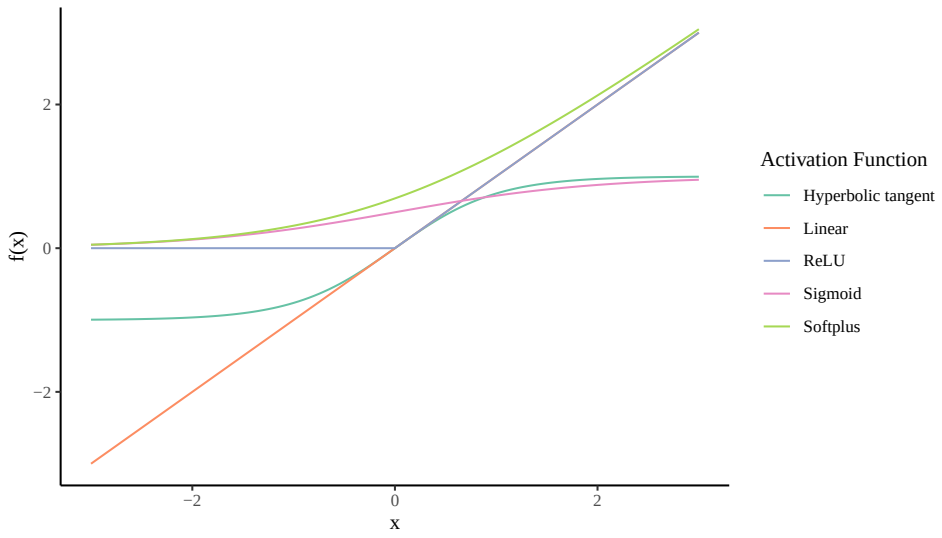


Figure 6.1: Common neural activation functions

The output prediction is made at the network's final layer. Each neuron at this layer combines the hidden neurons' activation though a weighted sum and adds

a bias adjustment term (6.2a). For regression problems, a linear activation layer is usually used to map the output vector back into an unbounded real value range (6.2b). For classification problems, the softmax function is used at the final output layer. It maps the hidden layers' activations into the range $(0, 1)$ where the sum of the entire output vector is restricted to 1 in order to represent class probabilities (6.2c).

Weighted sum of hidden vector with bias adjustment

$$y_k = \beta_k + \sum_{h=1}^{H} w_{h,k} h_h, k = 1, ..., K \tag{6.2a}$$

Linear output

$$\hat{Y_k} = y_k \tag{6.2b}$$

Softmax output

$$\hat{Y_k} = \frac{e^{y_k}}{\sum_{k'=1}^{K} e^{y_{k'}}} \tag{6.2c}$$

6.1 Multilayer Perceptron

A multilayer perceptron (MLP) is the simplest form of all neural networks. It consists of several stacked hidden layers, where all neurons in the hidden layers are fully interconnected.

It is common practice to use zero-centred values for neural network training[1]. To achieve this, you can normalise variables into z-scores (6.3a) so that they have similar range. For any individual value x_i, the z-score can be calculated as the distance from the arithmetic mean $\bar{x}$ divided by the standard deviation σ of the variable.

$$z_i = \frac{x_i - \bar{x}}{\sigma} \tag{6.3a}$$

At the training phase, the network weights are usually initialised randomly. They are then optimised through backpropagation to achieve gradient descent. The weights improve gradually according to a predefined learning rate until they ultimately converge to the minimum value. One of the drawbacks is that backpropagation does not guarantee reaching the global minimum if there are multiple minima across the parameter space. Such problem is usually mitigated by using advanced optimisers with adaptive learning rate.

[1] Zero-centred values have a stronger gradient, thus speed up optimisation through gradient descent.

Exercise 20 Training MLP for Regression Problem

There are many packages which implement neural networks. Traditional packages include `neuralnets`, `nnet`, `RSNNS`, `caret`... etc. Modern deep learning frameworks such as `h2o`, `MXNet` and `keras` are also available in R, but they usually require premium hardware set-up. In general, all neural network packages implement the same underlying algorithm and the differences usually lie in syntax, execution speed and hardware compatibility.

We will continue to use the `mtcars` dataset in this exercise. The objective of this exercise is to predict the `mpg` value of each car given all other known attributes of it. We would use the `neuralnet` package to create a simple multilayer perceptron (MLP) model. The code in example 6.1.1 trains a fully-connected multilayer perceptron with two hidden layers.

R Example 6.1.1

```
library(dplyr)
library(neuralnet)
# The mtcars dataset has a mixture of numeric and categorical variables
# For numeric variables we need to normalise them
mtcars_numeric <- mtcars %>% select(mpg, disp, hp, drat, wt, qsec)
# foreach numeric variable, we calculate the mean and standard deviation
mtcars_mean <- mtcars_numeric %>% lapply(mean)
mtcars_sd <- mtcars_numeric %>% lapply(sd)
# Convert the numeric variables into z-scores using the mean and sd
mtcars_numeric_normalised <- (mtcars_numeric - mtcars_mean) / mtcars_sd
# Construct a two layers MLP using all numeric variables.
# By default it uses sigmoid active function
myNN1 <- neuralnet(formula = mpg ~ disp + hp + drat + wt + qsec,
                   data = mtcars_numeric_normalised,
                   hidden = c(4,3),
                   linear.output = TRUE,
                   lifesign = "full")
# Visualise the network topology
plot(myNN1)
```

Example 6.1.2 shows that you can load the package `NeuralNetTools` to create a better topology plot. In addition, you can plot the observed data against network predictions to visualise the error. For a regression problem, the mean squared error (MSE) defined as $\frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2$ is usually used as the error measurement.

R Example 6.1.2

```
# Use a helper package for prettier graphics (optional)
library(NeuralNetTools)
plotnet(myNN1)
# Calculate the network prediction
myNNResult1 <- compute(myNN1, mtcars_numeric_normalised %>% select(-mpg))
# The predicted values are in scaled format (z-score)
# Need to convert it back to original scale for comparison
myNNPred1 <- myNNResult1$net.result[,1] *
                mtcars_sd[["mpg"]] +
                mtcars_mean[["mpg"]]
# Visualise the results on a scatterplot
qplot(mtcars$mpg, myNNPred1) +
  labs(x="Observed MPG",
       y="Predicted MPG")
# Calculate model error using mean squared error (MSE)
myNNError1 <- mean((myNNPred1 - mtcars$mpg)^2)
```

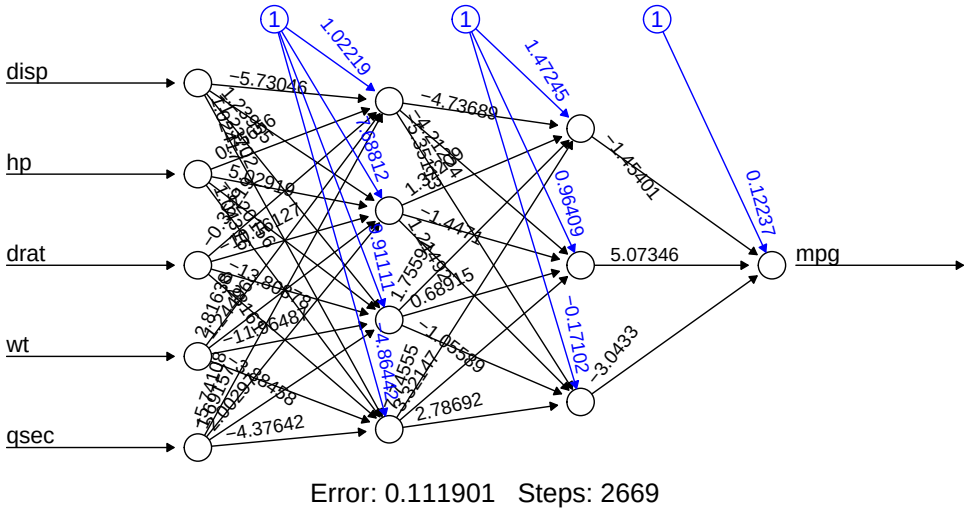


Figure 6.2: MLP model with two hidden layers

The sigmoid function is traditionally used in shallow networks. It can suffer from a weak gradient when stacked in deep networks. You can use the code in example 6.1.3 to customise the activation function.

R Example 6.1.3

```
# Build the second model
# Keep everything the same but change to softplus activation
myNN2 <- neuralnet(formula = mpg ~ disp + hp + drat + wt + qsec,
          data = mtcars_numeric_normalised,
          hidden = c(4,3),
          linear.output = TRUE,
          act.fct = function(x) { log(1+exp(x)) },
          lifesign = "full")
# Calculate the network prediction
myNNResult2 <- compute(myNN2, mtcars_numeric_normalised %>% select(-mpg))
# Convert the predicted values back to original scale
myNNPred2 <- myNNResult2$net.result[,1] *
                mtcars_sd[["mpg"]] +
                mtcars_mean[["mpg"]]
# Calculate model error (MSE)
myNNError2 <- mean((myNNPred2 - mtcars$mpg)^2)
```

Neural networks can also deal with categorical inputs. They are usually converted into one-hot encoding to feed into the model[2]. The code in example 6.1.4 converts all categorical variables in the `mtcars` dataset into one-hot encoding. The encoded values are then bound to numeric values and jointly used for training.

[2]For a categorical variable with K unique values, one-hot encoding would produce K new variables. Each new variable will have value $\{1, 0\}$. Please note that this is different from dummy encoding in statistical modelling.

R Example 6.1.4

```
# Loads the purrr package to access the imap function
library(purrr)
# Selecting all the categorical variables in the dataset
# Use the imap function to iterate through all columns
# Convert all into one-hot encoding and binds back into a tibble
mtcars_encoded <- mtcars %>% select(cyl,vs,am,gear,carb) %>%
  imap(function(myCol, myName) {
    myUniqueValues <- unique(myCol)
    myTib <- sapply(myUniqueValues,
           function(myValue){ (myValue == myCol) * 1 }) %>% as_tibble
    colnames(myTib) <- paste0(myName, "_", myUniqueValues)
    return(myTib)
  }) %>% bind_cols()
# Combines all numeric and categorical variables
mtcars_all <- bind_cols(mtcars_numeric_normalised, mtcars_encoded)
# Train the third model by including encoded categorical variables
myNN3 <- neuralnet(formula = mpg ~
                     disp + hp + drat + wt + qsec +
                     cyl_6 + cyl_4 + cyl_8 +
                     vs_0 + vs_1 +
                     am_1 + am_0 +
                     gear_4 + gear_3 + gear_5 +
                     carb_4 + carb_1 + carb_2 + carb_3 + carb_6 + carb_8,
          data = mtcars_all,
          hidden = c(4,3),
          linear.output = TRUE,
          act.fct = function(x) { log(1+exp(x)) },
          lifesign = "full")
# Visualise the network topology
plot(myNN3)
# Calculate the network prediction
myNNResult3 <- compute(myNN3, mtcars_all %>% select(-mpg))
# Convert the predicted values back to original scale
myNNPred3 <- myNNResult3$net.result[,1] *
               mtcars_sd[["mpg"]] +
               mtcars_mean[["mpg"]]
# Calculate model error (MSE)
myNNError3 <- mean((myNNPred3 - mtcars$mpg)^2)
# Compare the error of the three models
myNNError1
myNNError2
myNNError3
```

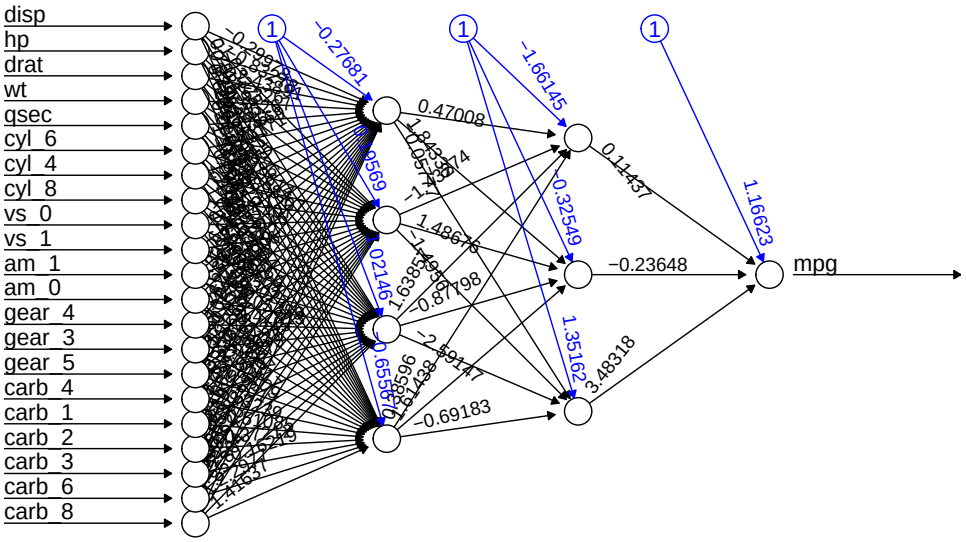


Figure 6.3: MLP model with one-hot encoded categorical variables

Chapter 7

Time Series Analysis

Many datasets have a temporal dimension. Time series refers to a chronologically-ordered sequence of observations. There are two main types of time series data: 1) regularly-sampled[1], and 2) irregularly-sampled[2] In this chapter, we would focus on time series data sampled at regularly-spaced intervals.

Temporal properties are often the point-of-interest in time series analysis. These include trend, seasonality, or temporal dependency between different variables. Studying these properties can offer useful insights. For example, extrapolating the trend can create forecasts for future scenarios. Alternatively, analysing the seasonality can help users understand the nature of recurring patterns.

7.1 Auto-Correlation Function

The auto-correlation function (ACF) measures the correlation of a single variable along the temporal dimension between x_t and x_{t+h}. In other words, it shows the correlation of the variable over different lag periods.

In the R language, you may use the `Acf(x)` function within the package `forecast` to plot the ACF correlogram. For most time series variables, correlation is usually strong at lag $h = 1$ and it gradually diminishes as the lag period increases. A cyclic pattern in the correlogram suggests possible seasonality which you can analyse further.

On the other hand, the partial auto-correlation function (PACF) is similar to the

[1]Regularly sampled time series has observations taken at fixed interval. This includes examples like heart rate, network traffic, daily weather... etc.

[2]Refers to observations that are not recorded at fixed interval, such as incidents of earthquakes.

ACF in the sense that it also measures the correlation between different lag periods. The difference is that it controls the correlation across the temporal dimension so that only the contribution of an individual lag is reflected.

Exercise 21 Loading Datasets

In this exercise, we will use an external dataset. This dataset contains daily electricity generation and demand data published by a German transmission network called Amprion[3]. The first column is a date-time variable and the rest are demand and generation data, each sampled at 15 minutes granularity. The code in example 7.1.1 shows how to load the dataset from the CSV file.

Table 7.1: Description of the Amprion dataset

Variable	Measurement Unit	Description
datetime	%Y-%m-%d %H:%M:%S	Date and time
demand	Megawatt	Demand in control area
pv	Megawatt	Photovoltaic feed-in
wp	Megawatt	Wind power feed-in

R Example 7.1.1

```
# Read the Amprion dataset from CSV file
# You might have to modify the path to point to your file location
library(readr)
amprion <- read_csv("amprion.csv")
# View the dataset
amprion
```

```
## # A tibble: 140,240 x 4
##    datetime            demand    pv    wp
##    <dttm>               <dbl> <dbl> <dbl>
##  1 2014-01-01 00:00:00  16231     0  1498
##  2 2014-01-01 00:15:00  16125     0  1449
##  3 2014-01-01 00:30:00  16066     0  1411
##  4 2014-01-01 00:45:00  16137     0  1438
##  5 2014-01-01 01:00:00  16045     0  1256
##  6 2014-01-01 01:15:00  15851     0  1242
##  7 2014-01-01 01:30:00  15729     0  1216
##  8 2014-01-01 01:45:00  15498     0  1312
##  9 2014-01-01 02:00:00  15417     0  1473
## 10 2014-01-01 02:15:00  15471     0  1676
## # ... with 140,230 more rows
```

[3]Amprion - Demand in Conrrol Area https://www.amprion.net/Grid-Data/Demand-in-Control-Area/

One of the main drivers of power demand and generation is the weather. The weather dataset is published by the Deutscher Wetterdienst[4]. Weather observations are recorded every hour at the Bremen weather station. You can follow the code in example 7.1.2 to load the dataset.

Table 7.2: Description of the Bremen weather dataset

Variable	Measurement Unit	Description
datetime	%Y-%m-%d %H:%M:%S	Date and time
airtemp	Degree Celsius	Air temperature
sun	$Jcm^{-}1$	Short-wave global radiation
windspd	$msec^{-}1$	Wind speed
winddir	Bearing	Wind direction
soil10	Degree Celsius	Soil temperature at 10cm depth
soil20	Degree Celsius	Soil temperature at 20cm depth
soil50	Degree Celsius	Soil temperature at 50cm depth
soil100	Degree Celsius	Soil temperature at 100cm depth

R Example 7.1.2

```
# Load the Bremen weather dataset
bremen <- read_csv("bremen.csv")
# View the dataset
bremen
```

```
## # A tibble: 79,669 x 9
##    datetime            airtemp   sun windspd winddir soil10 soil20 soil50
##    <dttm>                <dbl> <dbl>   <dbl>   <dbl>  <dbl>  <dbl>  <dbl>
##  1 2009-01-01 01:00:00    -2.5     0     3.8     350   -1.8   -1.5   -0.6
##  2 2009-01-01 02:00:00    -2.7     0     3.8     350   -1.7   -1.4   -0.6
##  3 2009-01-01 03:00:00    -2.8     0     1.6      40   -1.7   -1.3   -0.6
##  4 2009-01-01 04:00:00    -2.4     0     1.2     220   -1.6   -1.3   -0.5
##  5 2009-01-01 05:00:00    -2.2     0     1.8     260   -1.5   -1.2   -0.5
##  6 2009-01-01 06:00:00    -1.9     0     2.7     260   -1.5   -1.2   -0.5
##  7 2009-01-01 07:00:00    -1.6     0     2.6     250   -1.4   -1.1   -0.5
##  8 2009-01-01 08:00:00    -1.3     0     3       250   -1.3   -1.1   -0.5
##  9 2009-01-01 09:00:00    -0.8     2     2.6     260   -1.1   -1     -0.5
## 10 2009-01-01 10:00:00     0      11     3       250   -0.9   -0.9   -0.5
## # ... with 79,659 more rows, and 1 more variable: soil100 <dbl>
```

Before moving on to further analyses, we need to aggregate the two datasets into the same granularity. Once this is done, we can join the two datasets together

[4]DWD Climate Data Center https://www.dwd.de/EN/climate_environment/cdc/cdc_node.html

to form one table containing all variables. Example 7.1.3 shows how to use an SQL-like pipeline in `dplyr` to aggregate and join the two datasets.

R Example 7.1.3

```
# Load the lubridate package to access more datetime functions
library(lubridate)
# Load the dplyr package for data wrangling
library(dplyr)
# Aggregate the amprion dataset from 15 minutes to daily level.
amprion_daily <- amprion %>%
                  mutate(date = datetime %>%
                           ymd_hms() %>%
                           floor_date("day") %>%
                           as.Date()) %>%
                  group_by(date) %>%
                  summarise(total_demand = sum(demand),
                            total_pv = sum(pv),
                            total_wp = sum(wp))
# Aggregate the bremen dataset from hourly to daily.
bremen_daily <- bremen %>%
                  mutate(date = datetime %>%
                           ymd_hms() %>%
                           floor_date("day") %>%
                           as.Date()) %>%
                  group_by(date) %>%
                  summarise(mean_airtemp = airtemp %>% mean(),
                            max_sun = sun %>% max(),
                            mean_windspd = windspd %>% mean(),
                            mean_soil10 = soil10 %>% mean(),
                            mean_soil20 = soil20 %>% mean(),
                            mean_soil50 = soil50 %>% mean(),
                            mean_soil100 = soil100 %>% mean())
# Join the two daily datasets together into a common table
myTable <- amprion_daily %>%
  left_join(bremen_daily, by = "date")
# Plots the daily total demand
myTable %>%
  ggplot(aes(x=date, y=total_demand)) +
  geom_line() +
  labs(x = "Date",
       y = "Power Demand (MW)")
```

Exercise 22 Analysing Temporal Correlation

You can use the code in example 7.1.4 to create the ACF and PACF correlograms. In addition, you can use the `Ccf()` or `ggCcf()` function to create a cross-correlation function (CCF) correlogram. It analyses the temporal correlation between two variables.

R Example 7.1.4

```
# Load the forecast package
library(forecast)
# Plots the ACF correlogram only
# There are several ways to create plots.
ggAcf(myTable$total_demand) # More pretty
Acf(myTable$total_demand)   # Standard base R plot
# Plots the PACF correlogram only.
ggPacf(myTable$total_demand)
Pacf(myTable$total_demand)
# Draw a CCF correlogram which find the correlation between two variables.
# You can try swapping variables here.
ggCcf(x = myTable$mean_airtemp,
      y = myTable$total_demand)
Ccf(x = myTable$mean_airtemp,
    y = myTable$total_demand)
# Constructs the several key plots in one go.
ggtsdisplay(myTable$total_demand)
tsdisplay(myTable$total_demand)
# Create a lag plot
gglagplot(myTable$total_demand, lags = 28)
lag.plot(myTable$total_demand, lags = 28)
```

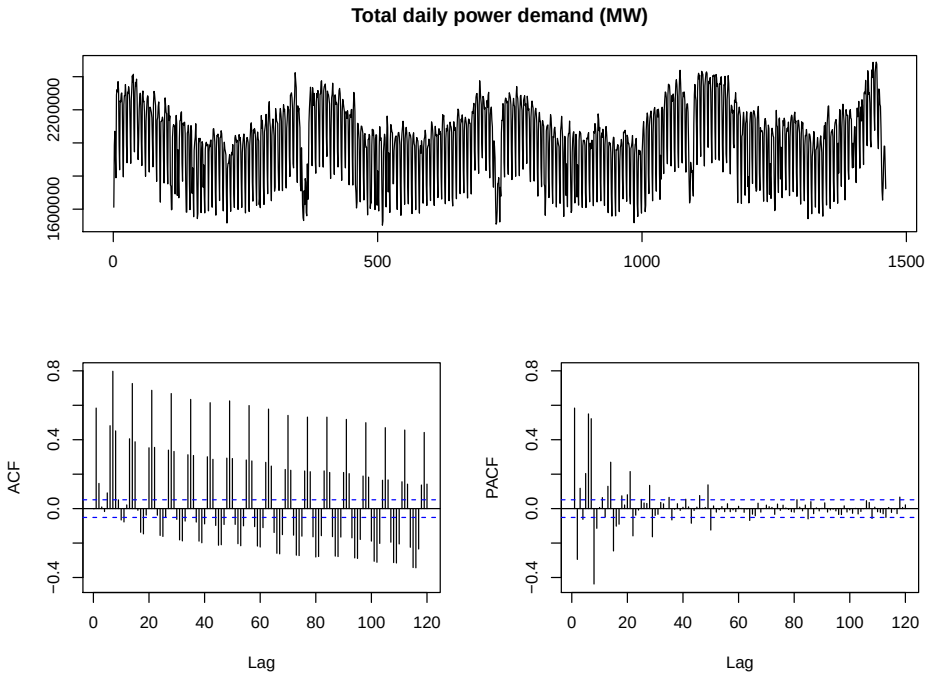


Figure 7.1: ACF and PACF correlograms

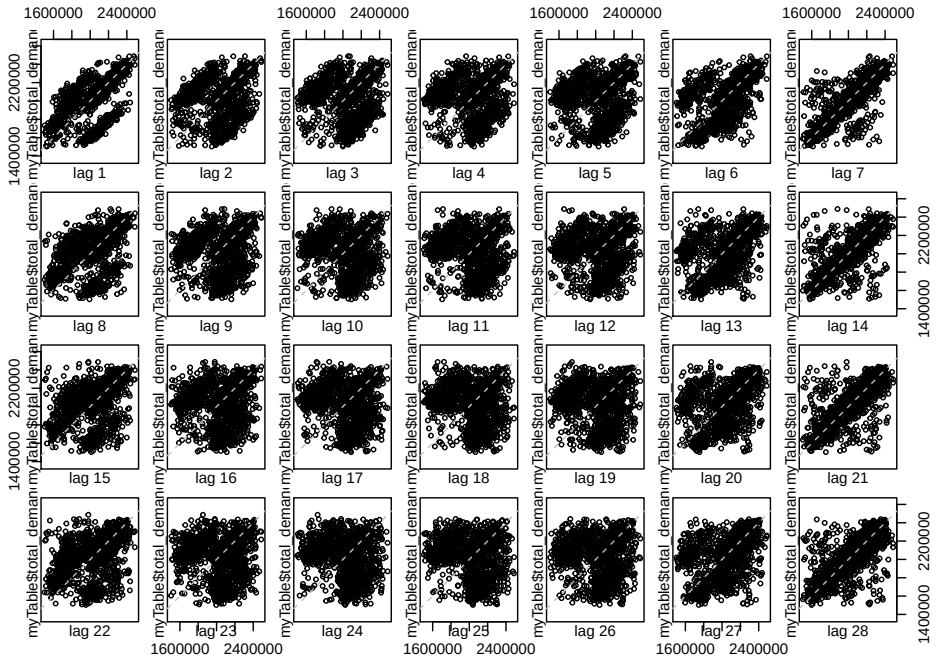


Figure 7.2: Lag plots showing the correlation of various lag periods

7.2 Decomposition

Time series can be either additive or multiplicative. Additive time series can be generally described as $X_t = S_t + T_t + \epsilon_t$ where S_t refers to the seasonality at time t while T_t refers to the trend component at time t. The observed data X_t is simply the sum total of the trend, seasonal and error components. Alternatively, a multiplicative time series is defined as $X_t = S_t \times T_t \times \epsilon_t$. These components can be easily decomposed from the observed values.

Exercise 23 Identifying the Trend and Seasonal Components

To analyse the seasonality of a time series, you need to find out the ideal frequency of the seasonal component. Example 7.2.1 uses the function `findfrequency()` to identify the frequency for a given time series. It uses spectral analysis to identify the frequency with the strongest spectral density. Once the frequency is calculated, we can build a `ts` object using the calculated frequency value. The function `decompose()` converts the observed time series into a trend component $T_t \in [1, T]$, a seasonal component $S_t \in [1, T]$ and random residuals $\epsilon_t \in [1, T]$.

The code also divides the dataset into a training and a testing set. The training set is used to run analyses and train models. Once the models are trained, they are applied to the testing set to assess model performance.

R Example 7.2.1

```
# Divide the dataset into training and testing set
TEST_SET_BEGIN <- "2017-01-01"
myTrainingSet <- myTable %>% filter(date < TEST_SET_BEGIN)
myTestingSet <- myTable %>% filter(date >= TEST_SET_BEGIN)
# Automatically search for ideal frequency using training data
# We would expect the frequency to be 7 (weekly pattern)
myFreq <- findfrequency(myTrainingSet$total_demand)
# Check the calculated frequency
myFreq
# Define a seasonal time series object using the frequency value
myTs <- ts(data = myTrainingSet$total_demand,
           frequency = myFreq)
# Decompose the time series into its constituent components
myDecomp <- decompose(myTs,
                      type = "additive")
# View the decomposed components
autoplot(myDecomp)
plot(myDecomp)
```

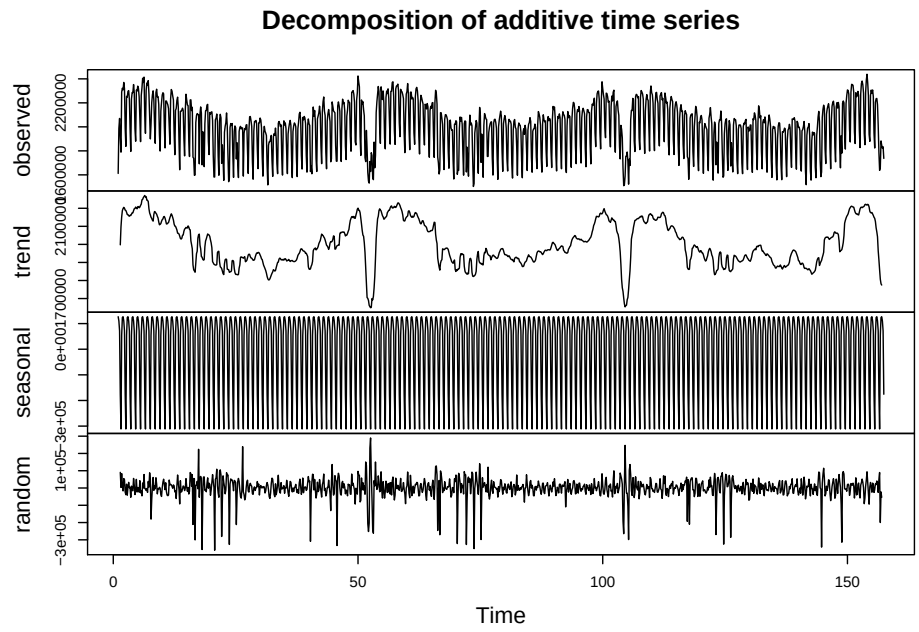


Figure 7.3: Decomposing an additive time series

Exercise 24 Linear Time Series Forecast

In this exercise, we will build a simple forecasting model using the trend and seasonal components as independent variables. The mathematical formula of a linear model with M predictor variables can be expressed as (7.1). The code in example 7.2.2 shows how to build a time series linear regression model with several covariate variables as predictors. It also visualises the forecast output as a chart. The coloured area surrounding the line represents the confidence interval of your prediction.

$$X_t = \beta_0 + \beta_{trend} T_t + \beta_{seasonal} S_t + \sum_{m=1}^{M} (\beta_m x_{mt}) + \epsilon_t \tag{7.1}$$

R Example 7.2.2

```
library(forecast)
# Perform linear regression model with decomposed time series components
# You can also add interaction and polynomial terms here
myTsModel1 <- tslm(myTs ~ trend + season +
                     mean_airtemp * mean_windspd +
                     poly(max_sun,degree = 2) +
                     mean_soil10 + mean_soil20,
                   data = myTrainingSet)
# View model summary
summary(myTsModel1)
# Produce forecast using the testing set
myTsForecast1 <- forecast(object = myTsModel1,
                          newdata = myTestingSet)
# Visualise the forecast
autoplot(myTsForecast1)
plot(myTsForecast1)
# Calculate the model performance by comparing with the testing set
# Using mean squared error (MSE) here.
myTestError1 <- mean((myTsForecast1$mean - myTestingSet$total_demand)^2)
```

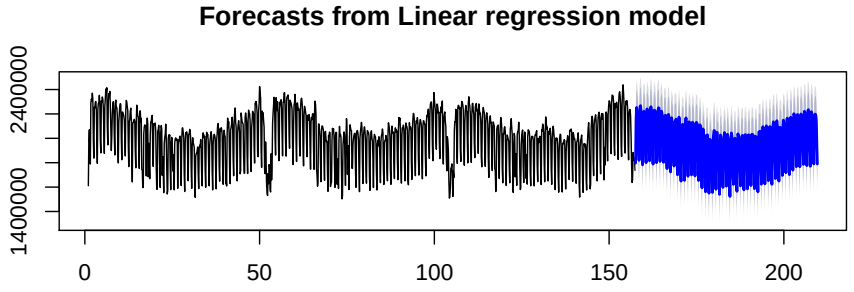


Figure 7.4: Linear time series forecasting with trend and seasonal components

To compare the time series regression model output with standard regression, you can run a simple linear regresion model using the same set of predictor variables with the `lm()` function. This is shown in example 7.2.3.

R Example 7.2.3

```
myTsModel2 <- lm(myTs ~ mean_airtemp * mean_windspd +
                   poly(max_sun,degree = 2) +
                   mean_soil10 + mean_soil20,
                   data = myTrainingSet)
# View model summary
summary(myTsModel2)
# Calculate model prediction using testing set
myTsForecast2 <- predict(object = myTsModel2,
                         newdata = myTestingSet)
# Calculate testing MSE
myTestError2 <- mean((myTsForecast2 - myTestingSet$total_demand)^2)
# Compare the testing error (MSE) of these two models
# Which one is a better model?
myTestError1
myTestError2
# Visualise the predictions of the two models
ggplot() +
  geom_line(aes(x=date, y=total_demand), myTrainingSet) +
  geom_line(aes(x=date, y=myTsForecast1$mean, colour="blue"),
            myTestingSet) +
  geom_line(aes(x=date, y=myTsForecast2, colour="green"),
            myTestingSet) +
  scale_colour_manual(guide = "legend", name = "Model",
         values =c("blue"="blue",
                   "green"="green"),
         labels = c("Time Series Regression Model",
                    "Simple Linear Regression")) +
  labs(x="Date",
       y="Total Demand (MW)") +
  theme(legend.position = "bottom")
```

7.3 ARIMA Model

ARIMA is the acronym for Auto-Regressive Integrative Moving Average model. It is a statistical technique which incorporates lag within the model. It can be described as the combination of three separate parts: autoregression, integration and moving average. ARIMA has three corresponding parameters p,d,and q which is normally expressed as $ARIMA(p,d,q)$ or as separate terms $AR(p)$, $I(d)$ and $MA(q)$.

The $AR(p)$ part suggests that observation X_t is dependent on the linear combination of lagged terms up to p lag periods. A pure $AR(p)$ model is expressed as $X_t = \sum_{i=i}^{p}(\phi_i X_{t-i})$. The moving average part $MA(q)$ indicates the residual is inherited from up to q lag periods. A pure $MA(q)$ model can be expressed as as $X_t = \sum_{i=1}^{q}(\theta_i \epsilon_{t-i}) + \epsilon_t$. As a result, a simple $ARMA(p,q)$ model can be

expressed as the following where ϕ_i and θ_i are model coefficients, while X_{t-i} represents observed data at i^{th} lag step and ϵ_{t-i} refers to the random error at the i^{th} lag step (7.2).

$$\underbrace{X_t}_{\text{Observation}} = \underbrace{\beta_0}_{\text{intercept}} + \underbrace{\sum_{i=i}^{p}(\phi_i X_{t-1})}_{\text{AR(p)}} + \underbrace{\sum_{i=1}^{q}(\theta_i \epsilon_{t-i})}_{\text{MA(q)}} + \underbrace{\epsilon_t}_{\text{residual}} \tag{7.2}$$

The ARIMA model assumes that the time series conforms to stationarity[5]. The integrative component $I(d)$ ensures stationarity by taking d number of integrative steps over time. A first order integrative model $I(1)$ is simply the difference between the current step and the immediate previous lag step. It is expressed as $X_t^{'} = X_t - X_{t-1}$. Similarly, a second order integrative model $I(2)$ is expressed as $X_t^{''} = X_t^{'} - X_{t-1}^{'} = X_t - 2X_{t-1} + X_{t-2}$.

Time series data with seasonality can be expressed as $ARIMA(p,d,q)(P,D,Q)_m$ where the uppercase parameters represent the seasonal component of the model. The m value is a positive non-zero integer indicating the frequency of the seasonality. The estimates $AR(P)$, $I(D)$ and $MA(Q)$ are linearly combined together with the non-seasonal part to create the seasonal ARIMA (SARIMA) model.

Exercise 25 Automated ARIMA

The standard ARIMA implementation accepts six parameters p, d, q, P, D and Q which produces a seasonal time series model. In many cases, these values are usually not known to the user and all possible values are examined case-by-case to get the best fit.

In the `forecast` package[6], you may use the function `Arima()` to experiment parameters manually. Alternatively, it is quite common to use an automated method to search for good parameters. The method `auto.arima()` tries all parameter values within the given constraints. It can also fit linear regression using predictor variables if the `xreg` attribute is supplied to the function. This is considerably slower than the `Arima()` function due to overhead for parameter search. Example 7.3.1 demonstrates parameter searching using automated ARIMA.

[5] A stationary time series has consistent statistical properties across all time, such as equal mean and variance.

[6] The package author has published a detailed book: `https://www.otexts.org/fpp/`

R Example 7.3.1

```
library(forecast)
library(dplyr)
# Build an ARIMA model automatically
# Keeping the maximum order (p+d+P+D) small
# Search for seasonal model only
myTsModel3 <- auto.arima(y = myTs,
                         max.order = 5,
                         seasonal = TRUE,
                         xreg = myTrainingSet %>%
                                 select(mean_airtemp,
                                        mean_windspd,
                                        max_sun,
                                        mean_soil10,
                                        mean_soil20,
                                        mean_soil50,
                                        mean_soil100) %>%
                           as.matrix(),
                         trace = TRUE)
# View the ARIMA(p,d,q)(P,D,Q) estimates and their coefficients
summary(myTsModel3)
# Run the forecast
# Apply the ARIMA model to testing set
myTsForecast3 <- forecast(myTsModel3,
                          xreg = myTestingSet %>%
                            select(mean_airtemp,
                                   mean_windspd,
                                   max_sun,
                                   mean_soil10,
                                   mean_soil20,
                                   mean_soil50,
                                   mean_soil100) %>%
                            as.matrix())
# Visualise the forecast
autoplot(myTsForecast3)
plot(myTsForecast3)
# Calculate the MSE error using the testing set
myTestError3 <- mean((myTsForecast3$mean - myTestingSet$total_demand)^2)
```

Exercise 26 Custom ARIMA

After running the automated ARIMA model, you might realise that the forecast tends to flat out when the forecast horizon increases. This is because the `auto.arima()` function selects the best parameters based on an indicator called AIC. It maximises the log-likelihood of the training data and gives preference to simpler models. We can manually tweak the ARIMA model with custom parameters using the `Arima()` function instead. This is demonstrated in example 7.3.2.

R Example 7.3.2

```
# Use custom parameters for the ARIMA model
# In this case we can try ARIMA(2,0,0)(1,1,1)
# You can change the parameters here
myTsModel4 <- Arima(y = myTs,
                    xreg = myTrainingSet %>%
                           select(mean_airtemp,
                                  mean_windspd,
                                  max_sun,
                                  mean_soil10,
                                  mean_soil20,
                                  mean_soil50,
                                  mean_soil100) %>%
                           as.matrix(),
                    order = c(2,0,0),
                    seasonal = c(1,1,1))
# View the model summary
summary(myTsModel4)
# Apply the ARIMA model to test set
myTsForecast4 <- forecast(myTsModel4,
                          xreg = myTestingSet %>%
                            select(mean_airtemp,
                                   mean_windspd,
                                   max_sun,
                                   mean_soil10,
                                   mean_soil20,
                                   mean_soil50,
                                   mean_soil100) %>%
                            as.matrix())
# Visualise the forecast
autoplot(myTsForecast4)
plot(myTsForecast4)
# Calculate MSE error using the testing set
myTestError4 <- mean((myTsForecast4$mean - myTestingSet$total_demand)^2)
```

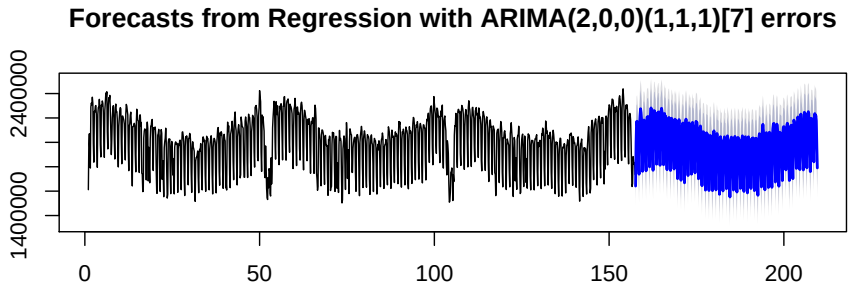


Figure 7.5: Forecast generated from a seasonal ARIMA model.

Exercise 27 Model-based Simulation

With a full $ARIMA(p,d,q)(P,D,Q)_m$ model, model-based simulation can be created very easily. The code in example 7.3.3 will generate 100 simulated runs and plot the average of all runs as point forecast on a chart.

R Example 7.3.3

```
# Use one of the trained ARIMA model for simulation
# Wrapping the simulation in a lapply loop
mySimulation <- lapply(1:100, function(i){
  tibble(date = myTestingSet$date,
         run = i,
         value = simulate(object = myTsModel4,
                          xreg = myTestingSet %>%
                            select(mean_airtemp,
                                   mean_windspd,
                                   max_sun,
                                   mean_soil10,
                                   mean_soil20,
                                   mean_soil50,
                                   mean_soil100) %>%
                            as.matrix()) %>%
                  as.numeric())})
# Combines all tibbles together to form a large tibble
mySimulationAll <- do.call(rbind, mySimulation)
# Calculate the mean of all simulated runs
myTsForecast5 <- mySimulationAll %>%
                     group_by(date) %>%
                     summarise(fcast = mean(value))
# Visualise the simulated forecast data
ggplot() +
  geom_line(aes(x=date, y=total_demand), myTrainingSet) +
  geom_line(aes(x=date, y=value, group=run), mySimulationAll, alpha=0.02) +
  stat_summary(aes(x=date, y=value), mySimulationAll,
               fun.y = mean,
               geom = "line",
               colour ="blue") +
  labs(x="Date",
       y="Power Demand")
# Calculate the MSE error
myTestError5 <- mean((myTsForecast5$fcast - myTestingSet$total_demand)^2)
```

At last, you can compare the performance of various models:

```
# Linear time series model
myTestError1
# Simple linear regression (not time series model)
myTestError2
# Auto ARIMA model
myTestError3
# ARIMA with custom parameters
myTestError4
# Simulated ARIMA model
myTestError5
```

Chapter 8

Survival Analysis

Events occuring at irregular time intervals can be studied through survival analysis. It is commonly used to analyse time-to-event in many research areas, such as medicine, economics, engineering and biology. For example, survival analysis is traditionally used in clinical research to analyse the effects of different drugs on sustaining patient's life. In this case, the time to death is used an indicator for drug performance. We will go through several techniques in this chapter.

8.1 Kaplan-Meier Estimator

The Kaplan-Meier estimator is used to measure how many subjects have survived in a clinical trial since treatment began. At time $t \leqslant T$, the estimator is given by equation (8.1) where $d_{t'}$ represents the number of events and $n_{t'}$ represents the number of subjects at risk.

$$\hat{S}_t = \prod_{t'=1}^{t} \left(1 - \frac{d_{t'}}{n_{t'}}\right) \tag{8.1}$$

Exercise 28 Fitting a Kaplan-Meier Curve

There are many implementations for survival analysis in the R language. The most commonly used one is the `survival` package. You can use the `survfit()` function to fit a Kaplan-Meier curve with categorical predictors.

In this exercise, we use the `lung` dataset within the `survival` package which contains lung cancer patients' survival time. You can use the command `?lung` to

read the detailed dataset description. To fit a Kaplan-Meier curve, we need to define the target event (i.e. death, in this example) and the time-to-event. The code in example 8.1.1 shows how to define the `Surv` object using the `Surv(time, event)` function. The `survfit` function fits a Kaplan-Meier curve against the target event using the supplied variables.

R Example 8.1.1

```
# Load the survival package for curve fitting
library(survival)
# Use the survminer package for better graphics
library(survminer)
# Load the lungs dataset into current environment
data(lung)
# Read the dataset description
?lung
# Build an empty model
# We are interested in death cases only (status = 2)
# This model has no predictor variable
mySurvFit1 <- survfit(Surv(time, status==2) ~ 1,
                      data = lung)
# Plot the fited curve
ggsurvplot(mySurvFit1)
# Use patient's sex as predictor
mySurvFit2 <- survfit(Surv(time, status==2) ~ sex,
                      data = lung)
# Plot the curve with confidence interval and p-value
ggsurvplot(mySurvFit2,
           conf.int = TRUE,
           pval = TRUE)
# The predictor needs to be categorical variable
# Use age as predictor by encoding into age group categories
mySurvFit3 <- survfit(Surv(time, status==2) ~ cut(age, c(40,50,60,70)),
                      data = lung)
ggsurvplot(mySurvFit3,pval = TRUE)
```

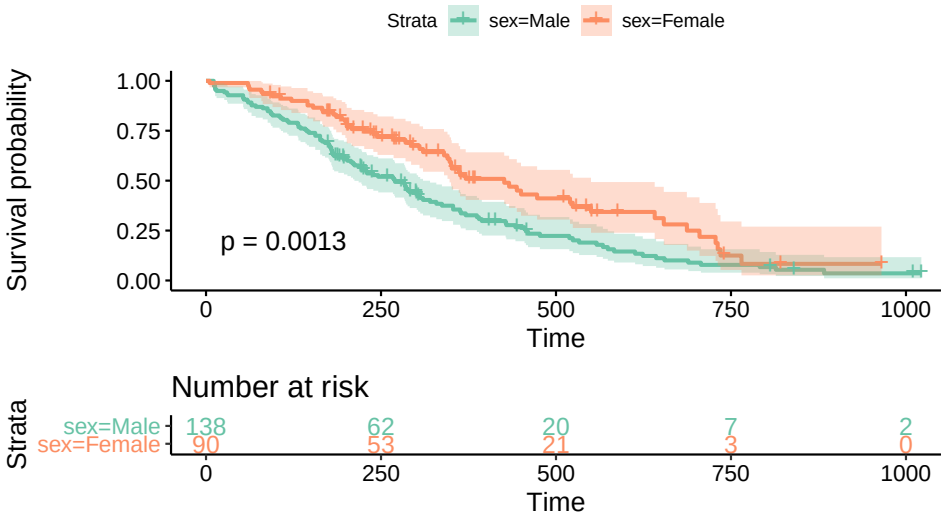


Figure 8.1: Kaplan-Meier curves showing two strata

8.2 Cox Proportional Hazards Model

To investigate the statistical effects of multiple predictor variables on survival probability, a technique named Cox regression can be used. Cox regression can take into account categorical, ordinal as well as numerical range variables. It analyses the effects of multiple variables on survival and assumes that the effects of these covariates are time-independent.

The Hazard function h_t is given by equation (8.2). The term $h_{0,t}$ in the equation represents the baseline Hazard when all covariates are zero. The linear terms $x_1, x_2, x_3, ..., x_M$ are the predictor variables, while $\beta_1, \beta_2, \beta_3, ..., \beta_M$ are their corresponding coefficients. For each coefficient β_m, the exponential term e^{β_m} represents the Hazard ratio of the covariate variable x_m. If $e^{\beta_m} > 1$, the corresponding covariate is positively correlated with an increase in hazard. On the other hand, x_m is negatively correlated with Hazard if $e^{\beta_m} < 1$. In the case where $e^{\beta_m} = 1$, the variable x_m has no effects on Hazard.

$$h_t = h_{0,t} \times e^{\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + ... + \beta_M x_M} \tag{8.2}$$

The Cox model assumes that variable effects are time-independent. To test whether the assumptions holds, we can analyse the Schoenfeld residuals for each covariate variable. Equation (8.3) defines the Schoenfeld residual $s_{i,k}$ of covariate k

of observation i. It is the difference between covariate $x_{i,k}$ and the sum of weighted likelihood of failure of all subjects at risk at time t. If there are observable temporal patterns in the residual plot, it suggests that the proportional Hazard assumption may have been violated. In this case, you can consider adding interaction effects with time to mitigate the problem.

$$s_{i,k} = x_{i,k} - \sum_{i=1}^{j \in R(t)} x_{i,m} \hat{p_j} \tag{8.3}$$

Exercise 29 Training a Cox Regression Model

Cox regression model can be trained using the `coxph()` function in the `survival` package. Example 8.2.1 builds a Cox regression model and analyses the effects of different covariate variables on the time-to-death of a group of cancer patients.

R Example 8.2.1

```
# Build a Cox model with predictor variables
myCoxModel1 <- coxph(Surv(time, status==2) ~ factor(sex) + age +
                       ph.ecog + ph.karno +
                       pat.karno +
                       meal.cal + wt.loss, data = lung)
# Read the model summary
summary(myCoxModel1)
```

```
## Call:
## coxph(formula = Surv(time, status == 2) ~ factor(sex) + age +
##     ph.ecog + ph.karno + pat.karno + meal.cal + wt.loss, data = lung)
##
##   n= 168, number of events= 121
##    (60 observations deleted due to missingness)
##
##                      coef  exp(coef)   se(coef)      z Pr(>|z|)
## factor(sex)2 -5.509e-01  5.765e-01  2.008e-01 -2.743  0.00609 **
## age           1.065e-02  1.011e+00  1.161e-02  0.917  0.35906
## ph.ecog       7.342e-01  2.084e+00  2.233e-01  3.288  0.00101 **
## ph.karno      2.246e-02  1.023e+00  1.124e-02  1.998  0.04574 *
## pat.karno    -1.242e-02  9.877e-01  8.054e-03 -1.542  0.12316
## meal.cal      3.329e-05  1.000e+00  2.595e-04  0.128  0.89791
## wt.loss      -1.433e-02  9.858e-01  7.771e-03 -1.844  0.06518 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##              exp(coef) exp(-coef) lower .95 upper .95
## factor(sex)2    0.5765     1.7347    0.3889    0.8545
## age             1.0107     0.9894    0.9880    1.0340
## ph.ecog         2.0838     0.4799    1.3452    3.2277
## ph.karno        1.0227     0.9778    1.0004    1.0455
## pat.karno       0.9877     1.0125    0.9722    1.0034
## meal.cal        1.0000     1.0000    0.9995    1.0005
## wt.loss         0.9858     1.0144    0.9709    1.0009
##
## Concordance= 0.651  (se = 0.029 )
## Likelihood ratio test= 28.33  on 7 df,   p=2e-04
## Wald test            = 27.58  on 7 df,   p=3e-04
## Score (logrank) test = 28.41  on 7 df,   p=2e-04
```

The model output above shows the statistical effects of different covariates. For instance, `sex`[1] is a statistically significant variable for predicting time-to-death of lung cancer patients. This variable has coefficient $\beta_{sex=2} = -0.551$, which means that having `sex=2` would change the patient's hazard by $e^{-0.551} - 1 = -42.4\%$. In other words, `sex=2` is beneficial to the patient's wellbeing.

Likewise, `ph.ecog` is also a significant variable. Each unit increase in `ph.ecog` would change the patient's Hazard by $e^{0.734} - 1 = 108.4\%$. This implies that a higher `ph.ecog` score significantly increases a patient's risk of

[1] It is encoded as male=1 and female=2.

death.

Exercise 30 Cox Regression Diagnostics

Example 8.2.2 tests the Cox proportional hazards assumption by calculating the Schoenfeld residuals. If there are observable patterns along the temporal dimension, the models' assumption may have been violated.

R Example 8.2.2

```
# Test the proportional Hazard assumption of a Cox regression
myCoxZph1 <- cox.zph(myCoxModel1)
# Print the results of the test
myCoxZph1
# Plot the Schoenfeld residuals
ggcoxzph(myCoxZph1)
```

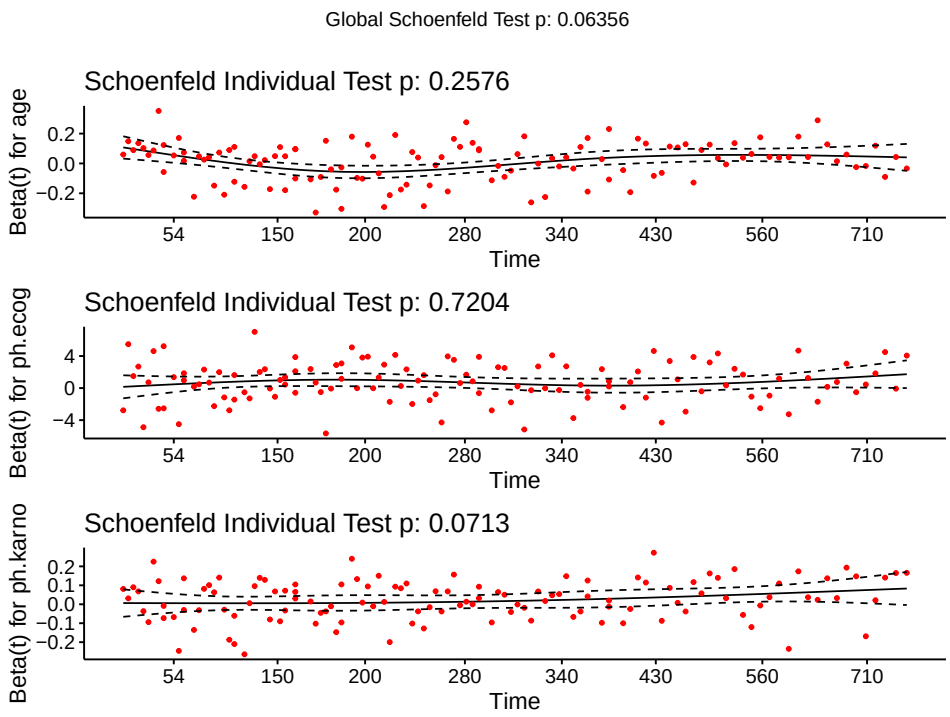


Figure 8.2: Scaled Schoenfeld residuals of a selected set of variables plotted against time

Chapter 9

Unsupervised Learning

Unsupervised learning identifies the underlying structure of an unlabelled dataset. Clustering is one of the most common applications of unsupervised learning, which aims at allocating similar objects into common groups.

In a given set of unlabelled objects, there can be different ways to produce clusters. Figure 9.1 below shows the effects of choosing different number of clusters.

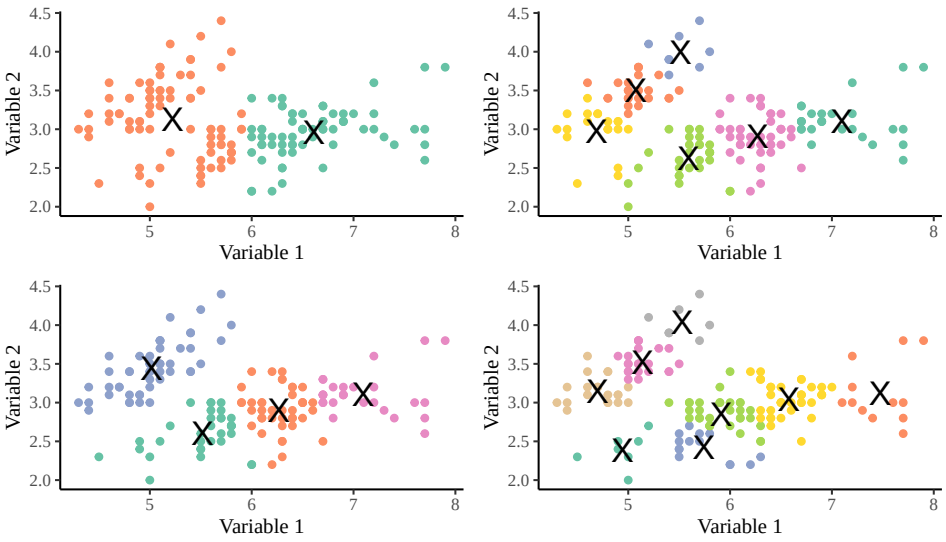


Figure 9.1: Different ways to cluster a set of unlabelled objects

9.1 K-means Clustering

K-means clustering is a very common clustering algorithm due to its intrinsic simplicity. It produces clusters by minimising the Euclidean distance between objects and the centroid of their own cluster. The Euclidean distance between two P-dimensional vectors $\vec{x_j}$ and $\vec{x_k}$ is defined as in equation (9.1).

$$d(\vec{x_j}, \vec{x_k}) = \sqrt{\sum_{p=1}^{P} (x_{j,p} - x_{k,p})^2} \tag{9.1}$$

Algorithm 1: K-means clustering

Input : Set of unlabelled object $X = \{\vec{x}_1, \vec{x}_2, \vec{x}_3, ..., \vec{x}_N\}$
Input : Number of clusters K
Initialise
$\{\vec{\mu}_1, \vec{\mu}_2, \vec{\mu}_3, ..., \vec{\mu}_K\} \leftarrow Randomise(\{\vec{x}_1, \vec{x}_2, \vec{x}_3, ..., \vec{x}_N\}, K), K < N$;
while *true* **do**
  for $k \leftarrow \{1, 2, 3, ..., K\}$ **do**
    $\omega_k \leftarrow \{\}$;
  end
  for $n \leftarrow \{1, 2, 3, ..., N\}$ **do**
    $k \leftarrow \underset{k'}{\operatorname{argmin}}\, d(\vec{\mu}_{k'}, \vec{x}_n), k \in 1, 2, 3, ..., K$;
    $\omega_k \leftarrow \omega_k \cup \{\vec{x}_n\}$;
  end
  for $k \leftarrow \{1, 2, 3, ..., K\}$ **do**
    $\vec{\mu}_k^{'} \leftarrow \frac{1}{|\omega_k|} \sum_{\vec{x} \in \omega_k} \vec{x}$;
  end
  if $\vec{\mu}_k^{'} = \vec{\mu}_k, k = 1, 2, 3, ...K$ **then**
    break;
  else
    $\vec{\mu}_k = \vec{\mu}_k^{'}$;
  end
end
return Cluster centroids $\{\vec{\mu}_1, \vec{\mu}_2, \vec{\mu}_3, ..., \vec{\mu}_K\}$;
return Object cluster assignment $\{\omega_1, \omega_2, \omega_3, ..., \omega_K\}$

The algorithm starts with a randomly select subset of K objects as initial cluster centroids such as $\{\vec{\mu}_1, \vec{\mu}_2, \vec{\mu}_3, ..., \vec{\mu}_K\}$. The Euclidean distance between initial centroids and each object in the unlabelled set is calculated. The cluster assignment of an object belongs to the centroid with shortest distance. Once cluster assignment is completed for all objects, the centroid is recomputed as the mean of the cluster.

This process iterates until the new cluster assignment is identical to the one at the previous iteration.

Since the dataset is unlabelled, the true number of clusters is unknown. The K value which represents the number of clusters is usually experimented one-by-one and the best value is determined from the output.

In the R language, the K-means clustering algorithm is implemented very efficiently. You can use the `kmeans()` function in the `stats` package to perform K-means clustering.

Exercise 31 Dimensionality Reduction

In example 9.1.1, we will use the `mtcars` dataset. This dataset contains six numeric variables. In other words, car can be represented as $P = 6$ dimensional objects. In practical applications of the K-means algorithm, it is very common to normalise the numeric variables using z-scores if they are recorded in different units. Normalisation ensures that all variables are fairly represented.

You can use dimensionality reduction techniques such as principal component analysis (PCA) to visualise the data. PCA converts input variables into principal components (PCs) in the order of maximum variance. The following code will execute PCA and visualise the top two PCs on a scatterplot.

R Example 9.1.1

```
# Select the numeric variables from the mtcars dataset
mtcars_numeric <- mtcars %>% select(mpg, disp, hp, drat, wt, qsec)
# Calculate the mean and standard deviation for each variables
mtcars_mean <- mtcars_numeric %>% lapply(mean)
mtcars_sd <- mtcars_numeric %>% lapply(sd)
# Convert the numeric variables into z-scores using the mean and sd
mtcars_numeric_normalised <- (mtcars_numeric - mtcars_mean) / mtcars_sd
# There are six variables in this dataset
# We can use principal component analysis (PCA) to reduce the dimensionity
myPca <- prcomp(mtcars_numeric_normalised)
library(ggfortify)
autoplot(myPca, loadings.label = TRUE)
```

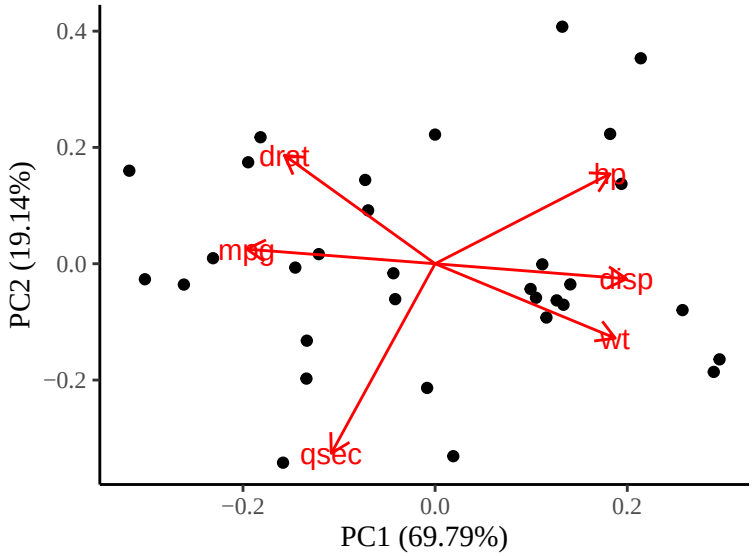


Figure 9.2: Biplot showing the first and second principal components

Exercise 32 K-means Clustering

With a $P = 6$ dimensional dataset, we can apply the K-means algorithm on it to compute the clusters. Since the number of clusters K is unknown, we experiement with different values in example 9.1.2. The clustering results can be visualised in a low-dimensional space with two principal components.

R Example 9.1.2

```
# We know there are three types of of flowers, so let's start with K=3
# You can try different values
myKClust <- kmeans(mtcars_numeric_normalised, centers = 3)
# Visualise the clusters
ggplot(myPca$x, aes(x = PC1,
                    y = PC2,
                    colour = factor(myKClust$cluster))) +
  geom_point() +
  geom_label(aes(label=mtcars %>% rownames())) +
  stat_ellipse() +
  labs(colour="Cluster")
```

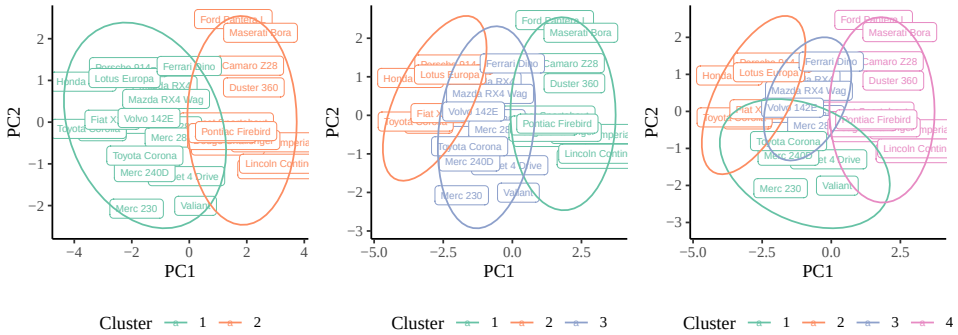


Figure 9.3: Comparing K-means clustering results using different K values

9.2 Hierarchical Clustering

In a given unlabelled dataset containing objects $\vec{x}_i, i = \{1, 2, 3, ..., N\}$, the maximum number of cluster is N, where each cluster contains only 1 member object. In this case, the cluster centroids denoted as $\vec{\mu}_i$ contain rich information which perfectly describes their member objects as $\vec{\mu}_i = \vec{x}_i$. However, such information would be practically useless. To improve this, we can start with N clusters and merge the closest two clusters into one. This would have obtained $N - 1$ clusters by adding the least amount of error into the system. This merging process can be iteratively repeated until there are no more clusters left to merge. This is how the agglomerative hierarchical clustering algorithm works.

In hierarchical clustering, the closeness metric between two clusters ω_i and ω_j is denoted as a function $D(\omega_i, \omega_j)$. There are several common choices including 1) single linkage, 2) complete linkage, 3) average linkage, 4) centroid and 5) Ward's method . For single linkage, the distance between two clusters is defined as the closest distance between their member objects (9.2a). In many cases, this tends to produce long chains with similar objects merging sequentially into the same cluster. On the other hand, complete linkage uses the distance between farthest objects between two clusters as the cluster closeness metric (9.2b). This tends to produce clusters with consistent size. The two aforementioned measurements are prone to outlier influence. To mitigate this problem, average linkage can be used. It uses the arithmetic average of all pairwise distances as the cluster closeness measurement (9.2c). Similarly, we can make use of the cluster centroid to measure closeness (9.2d). The centroid method is also resilient to outlier influence. Ward's method compares the change in the sum of squares between cluster members and their

centroid when they are merged (9.2e).

Single linkage

$$D(\omega_i, \omega_j) = \min_{\vec{x}_i \in \omega_i, \vec{x}_j \in \omega_j} d(\vec{x}_i, \vec{x}_j) \tag{9.2a}$$

Complete linkage

$$D(\omega_i, \omega_j) = \max_{\vec{x}_i \in \omega_i, \vec{x}_j \in \omega_j} d(\vec{x}_i, \vec{x}_j) \tag{9.2b}$$

Average linkage

$$D(\omega_i, \omega_j) = \underbrace{\frac{1}{|\omega_i|}\frac{1}{|\omega_j|}\sum_{\vec{x}_i \in \omega_i}\sum_{\vec{x}_j \in \omega_j} d(\vec{x}_i, \vec{x}_j)}_{\text{Average pairwise distance between } \omega_i \text{ and } \omega_j} \tag{9.2c}$$

Centroid

$$D(\omega_i, \omega_j) = d\Bigg(\Big(\underbrace{\frac{1}{|\omega_i|}\sum_{\vec{x}_i \in \omega_i} \vec{x}_i}_{\text{Centroid of } \omega_i}\Big), \Big(\underbrace{\frac{1}{|\omega_j|}\sum_{\vec{x}_j \in \omega_j} \vec{x}_j}_{\text{Centroid of } \omega_j}\Big)\Bigg) \tag{9.2d}$$

Ward's method

$$\begin{aligned} D(\omega_i, \omega_j) = &\underbrace{\sum_{k \in \omega_i \cup \omega_j} \Big(\vec{x}_k - \big(\frac{1}{|\omega_i \cup \omega_j|}\sum_{\vec{x}_{k'} \in \omega_i \cup \omega_j} \vec{x}_{k'}\big)\Big)^2}_{\text{Sum of squares of } \omega_i \cup \omega_j} - \\ &\underbrace{\sum_{i \in \omega_i} \Big(\vec{x}_i - \big(\frac{1}{|\omega_i|}\sum_{\vec{x}_{i'} \in \omega_i} \vec{x}_{i'}\big)\Big)^2}_{\text{Sum of squares of } \omega_i} - \\ &\underbrace{\sum_{j \in \omega_j} \Big(\vec{x}_j - \big(\frac{1}{|\omega_j|}\sum_{\vec{x}_{j'} \in \omega_j} \vec{x}_{j'}\big)\Big)^2}_{\text{Sum of squares of } \omega_j} \end{aligned} \tag{9.2e}$$

Algorithm 2: Agglomerative hierarchical clustering

Input : Set of unlabelled object $X = \{\vec{x}_1, \vec{x}_2, \vec{x}_3, ..., \vec{x}_N\}$
Input : Linkage function $D(\omega_i, \omega_j)$
for $n \in \{1, 2, 3, ..., N\}$ **do**
  $\omega_n \leftarrow \{\vec{x}_n\}$;
end
$\Omega \leftarrow \{\omega_1, \omega_2, \omega_3, ..., \omega_N\}$;
while $|\Omega| > 1$ **do**
  $\Omega^{'} = \{\}$;
  for $i \in \{1, 2, 3, ..., |\Omega|\}$ **do**
    $\Omega_i^{'} \leftarrow D(\omega_i, \omega_j), j = \{1, 2, 3, ..., |\Omega|\}$;
  end
  $\{i, j\} \leftarrow \underset{i,j}{\operatorname{argmin}}\, \Omega^{'}$;
  $\omega_{ij} \leftarrow \Omega_i^{'} \cup \Omega_j^{'}$;
  $\Omega \leftarrow \Omega^{'} \setminus \Omega_i^{'} \setminus \Omega_j^{'} \cup \omega_{ij}$;
end

The result of hierarchical clustering can be visualised using a tree-like structure called dendrogram. The merging sequence of clusters as well as object closeness can be easily read from the dendrogram. The height of the node at the dendrogram indicates the closeness metric of the two clusters when they are merged. After analysing the dendrogram, users can decide how many clusters to retain. This is usually an subjective decision. Once decided, the dendrogram can be cut to obtain the desired number of clusters. Alternatively, we can cut the dendrogram at a certain fixed height to discard trivial clusters at the bottom.

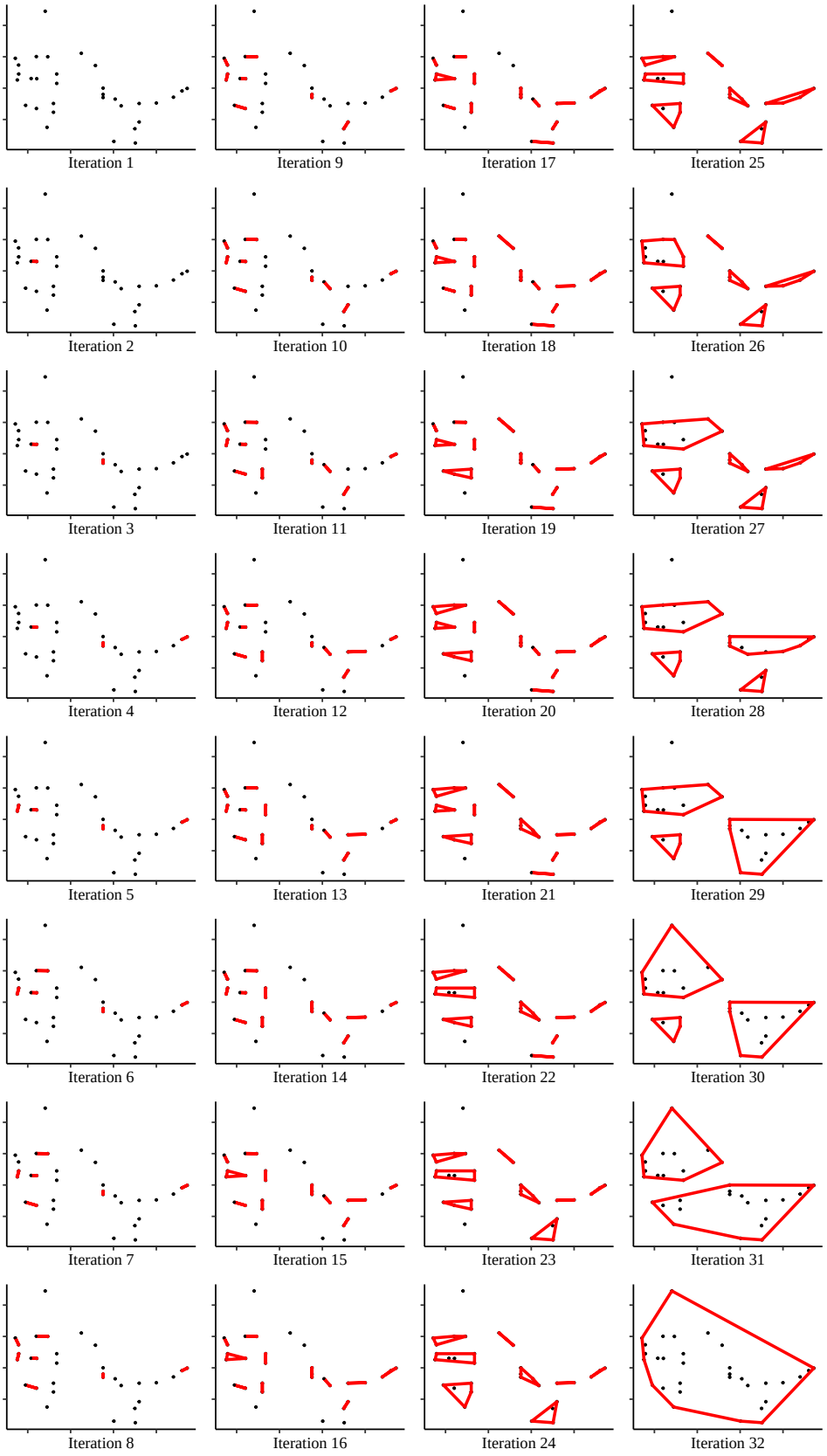


Figure 9.4: Iterative steps of agglomerative hierarchical clustering

Exercise 33 Constructing a Dendrogram

In the R language, hierarchical clustering can be performed using the `hclust()` function which is included in the default `stats` package. The function requires a distance matrix of objects which is normally pre-computed using the `dist()` function. The `hclust()` function uses complete linkage by default if the `method` parameter is not specified. You can change the linkage function and check the difference in output results. The code snippet in example 9.2.1 performs hierarchical clustering and visualises the result as a simple dendrogram.

R Example 9.2.1

```
# Calculate distance matrix
# Using Euclidean distance here but you can change it
myDist <- mtcars_numeric_normalised %>% dist(method = "euclidean")
# Perform hierarchical clustering using complete linkage
myHClust <- myDist %>% hclust(method = "complete")
# You can change the closeness measurement
# Read the documentation of the hclust function
?hclust
# Visualise the dendrogram
plot(myHClust)
# You can use ggdendrogram to plot a prettier dendrogram
library(ggdendro)
ggdendrogram(myHClust)
```

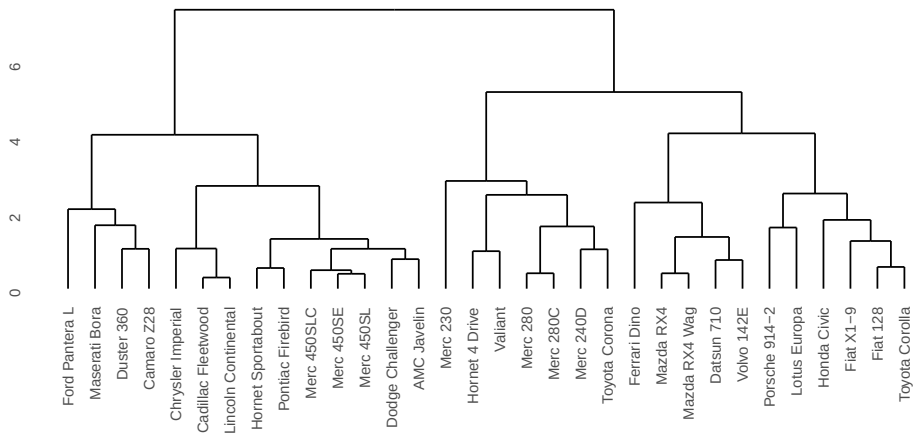


Figure 9.5: Dendrogram illustrating hierarchical clustering using complete linkage

Exercise 34 Cutting a Dendrogram

The dendrogram can be cut to remove smaller clusters at lower heights. This can be achieved using the `cutree()` function in R. You can either specify the number

of clusters to retain using the parameter `k`, this will retain the top `k` clusters of the dendrogram. Alternatively, you can use the `h` parameter to specify at which height the dendrogram should be cut. Example 9.2.2 shows how to cut a dendrogram. It also demonstrates various ways to visualise a dendrogram.

R Example 9.2.2

```
# Cut the dendrogram by specifying how many clusters to retain
# You can change the number of clusters here
myCutClusters1 <- cutree(myHClust, k = 5) %>% factor()
# Alternatively, cut the dendrogram at a certain height
myCutClusters2 <- cutree(myHClust, h = 6) %>% factor()
# Use the ape package to plot pretty dendrograms
# The RColorBrewer package generates colour palette
library(ape)
library(RColorBrewer)
# Obtain colour definition
myColours <- brewer.pal(n = 5, name="Set1")
# Convert the hierarchical cluster result into a phylogram object
myPhylo <- myHClust %>% as.phylo()
# Draw some plots
# This is a phylogenic tree
plot(myPhylo,
     type = "phylogram",
     tip.color = myColours[myCutClusters1])
# This is a cladogram
plot(myPhylo,
     type = "cladogram",
     tip.color = myColours[myCutClusters1])
# This is a unrooted phylogenic tree
plot(myPhylo,
     type = "unrooted",
     tip.color = myColours[myCutClusters1])
# This is a fan phylogram
plot(myPhylo,
     type = "fan",
     tip.color = myColours[myCutClusters1])
# This is a radial phylogram
plot(myPhylo,
     type = "radial",
     tip.color = myColours[myCutClusters1])
```

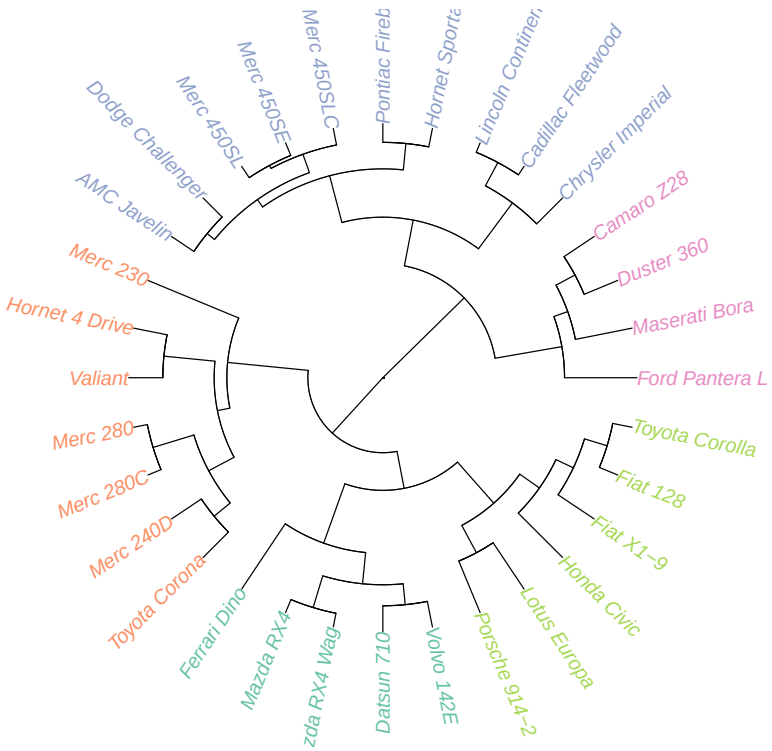


Figure 9.6: Fan phylogram showing hierarchical clusters

Chapter 10

Extending R

There are many ways to extend R functionalities. We will introduce some of them in this chapter.

10.1 R Markdown

R Markdown is a way to produce documents dynamically. It supports various output formats such as HTML, PDF, LaTeX and Beamer. The document is self-contained and fully reproducible which makes it easy to share.

RStudio is integrated with R Markdown so that users can manipulate the documents very easily. Users can launch a R Markdown template by navigating to **File > New File > R Markdown**. This creates a `.Rmd` template. It contains three types of content:

YAML header Surrounded by `---` at the top of the document. Users can specify key parameters here, such as output format.

Text Standard Markdown format.

Code chunk Surrounded by ```` ``` ````, optional arguments can be provided in the trailing curly bracket `{}`.

A standard YAML header can render basic properties such as document title and author names in the HTML output file. Additional properties can be supplied to modify style and enable advanced features.

The text body is written in standard markdown format. Markdown is a lightweight markup language with plain text formatting syntax. Small pieces of inline `R` code is surrounded by the `` `r `` mark. This will print the `R` output in the compiled output.

A large chunk of multi-line `R` code is embraced by the ```` ```{r} ```` symbol. Additional arguments can be passed to each code chunk. For example, `message=FALSE` would supress package loading message. On the other hand, `echo=FALSE` would hide the `R` code while still executing the chunk.

Exercise 35 Dynamically Generating a Report

This example shows how to dynamically generate an HTML document using `R` Markdown. In RStudio, user can compile a `.Rmd` file by clicking the **Knit** button or pressing `Ctrl + Shift + K`. This will send the document to the `knitr` engine for compilation. The compiled output is shown in Figure 35.

````
---
title: "Car Assessment Report"
author: "James Bond"
output: html_document
---
Vehicle Analysis
=======

I have analysed `r nrow(mtcars)` cars systematically. The
following vehicles have the largest `horsepower`:

```{r, message=FALSE}
library(dplyr)
mtcars %>%
  mutate(name = rownames(.)) %>%
  arrange(desc(hp)) %>%
  select(name, hp) %>%
  head(2)
```

The following graph shows that these cars have impressive
**horsepower** and **1/4 mile time**.

```{r, echo=FALSE, warning=FALSE, message=FALSE}
# Create a ggplot
library(ggplot2)
myPlot <- mtcars %>%
  mutate(car_name = rownames(.),
````
````

```
          hp_rank = dense_rank(desc(hp)),
          top2 = hp_rank <= 2) %>%
  ggplot(aes(x=qsec, y=hp), name=car_name) +
  stat_smooth(method = "lm") +
  geom_point(aes(colour=top2)) +
  labs(x="1/4 Mile Time (Seconds)",
       y="Horsepower",
       colour="Top 2 Horsepower")
# Convert ggplot to plotly
library(plotly)
myPlot %>% ggplotly(tooltip = "name")
```

```
## Selection Criteria

Essential gadgets are required for Health and Safety reasons:

* Rocket launchers (_front_ and _back_)
* Ejectable seats
* Bulletproof screen
```

Car Assessment Report

James Bond

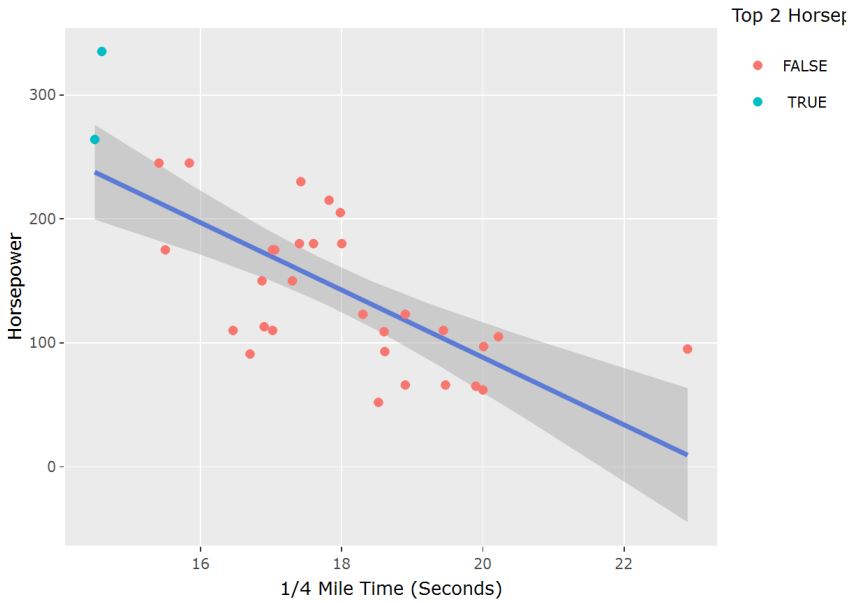
Vehicle Analysis

I have analysed 32 cars systematically. The following vehicles have the largest `horsepower`:

```
library(dplyr)
mtcars %>%
  mutate(name = rownames(.)) %>%
  arrange(desc(hp)) %>%
  select(name, hp) %>%
  head(2)
```

```
##             name  hp
## 1  Maserati Bora 335
## 2 Ford Pantera L 264
```

The following graph shows that these cars have impressive **horsepower** and **1/4 mile time**.



Selection Criteria

Essential gadgets are required for Health and Safety reasons:

- Rocket launchers (*front* and *back*)
- Ejectable seats
- Bulletproof screen

Figure 10.1: Compiled R Markdown output in HTML format

10.1.1 R Notebook

R Notebook is a special type of an R Markdown document with chunks that can be executed independently and interactively, with output visible immediately beneath the input.

Users can launch an R Notebook template by navigating to **File > New File > R Notebook**. It has the same `.Rmd` file extension and the Notebook is configured by the `output: html_notebook` parameter in the YAML header.

The **Preview** button shows the rendered copy of the Notebook. Unlike **Knit** for standard R Markdown documents, **Preview** does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.

10.2 Shiny Web Application

`shiny` is a package for creating interactive web applications. Users can use it to integrate statistical models in R with dashboard elements such as chart, drop down menu, checkbox, etc. It is a commonly used to disseminate analytical output to non-technical users.

In a `shiny` application, the web page layout is coded in R and automatically rendered. There is no need to write any HTML, CSS or JavaScript anymore. This enables users to focus more on analytical elements and develop dashboards rapidly.

User can launch a template for a `shiny` web application in RStudio by navigating to **File > New Project > Shiny Web App**. This creates a new project containing an `app.R` file[1][2]. There are two main components inside the file:

ui This defines the user interface (UI) for the `shiny` application, e.g. the page layout, location of control widgets, page title, etc.

server The server-side application which contains the logic. The analytical code is located in this part.

The `ui` of the default `shiny` template is shown in example 10.2.1. The page layout is defined by the `fluidPage()` function. In this example, the

[1] Do not change the file name, otherwise the system would not recognise it as a shiny web application.

[2] Users can also create two files ui.R and server.R seperately. This is suitable for more complex application with longer code.

`titlePanel()` function defines the page title which is located at the top. The `sidebarLayout()` function divides the screen layout into two unequal parts. The smaller part is defined by `sidebarPanel()` which contains input controls. On the other hand, the larger part is defined by `mainPanel()` which contains the output.

Many types of control widgets are supported in `shiny`. They are shown in figure 10.2. A detailed description can be found at `https://shiny.rstudio.com/tutorial/written-tutorial/lesson3/`.

R Example 10.2.1

```
# Define UI for application that draws a histogram
ui <- fluidPage(
   # Application title
   titlePanel("Old Faithful Geyser Data"),
   # Sidebar with a slider input for number of bins
   sidebarLayout(
      sidebarPanel(
         sliderInput("bins",
                     "Number of bins:",
                     min = 1,
                     max = 50,
                     value = 30)),
      # Show a plot of the generated distribution
      mainPanel(
         plotOutput("distPlot")
      )
   )
)
```

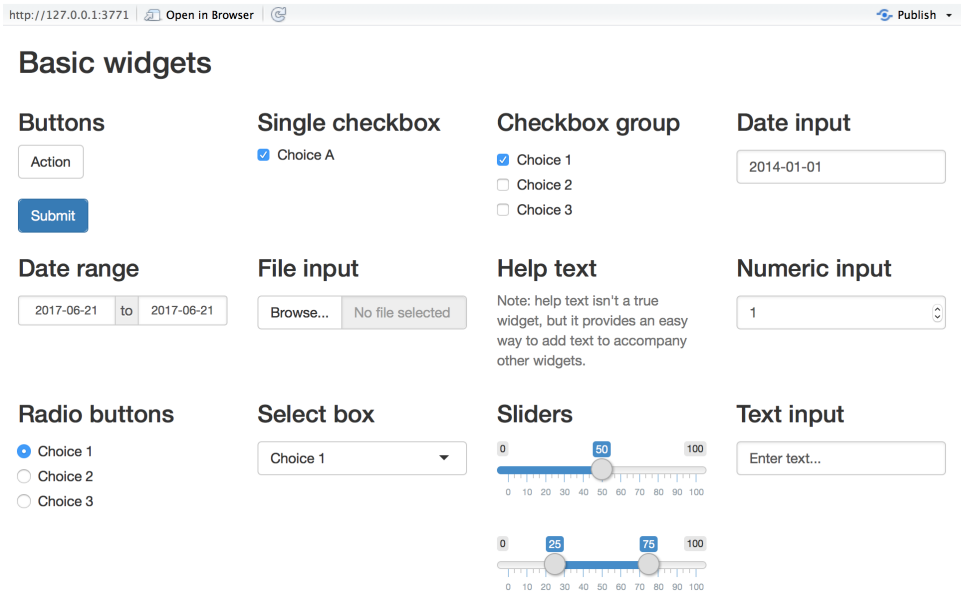


Figure 10.2: Control widgets in shiny

Example 10.2.2 shows the `server` component in the default `shiny` template. It is defined as a simple function which accepts two arguments `input` and `output`. In the function, named elements can be added to the `output` and visualised through the `ui`. In this case, the function retrieves variable `input$bins` and renders a histogram, which is eventually returned as a graph as the `output$distPlot` element.

R Example 10.2.2

```
# Define server logic required to draw a histogram
server <- function(input, output) {
   output$distPlot <- renderPlot({
      # generate bins based on input$bins from ui.R
      x    <- faithful[, 2]
      bins <- seq(min(x), max(x), length.out = input$bins + 1)
      # draw the histogram with the specified number of bins
      hist(x, breaks = bins, col = 'darkgray', border = 'white')
   })
}
```

To run a `shiny` application, users can click on the **Run App** button in RStudio. This will render the application. Figure 10.2 shows the `shiny` application rendered from the default template.

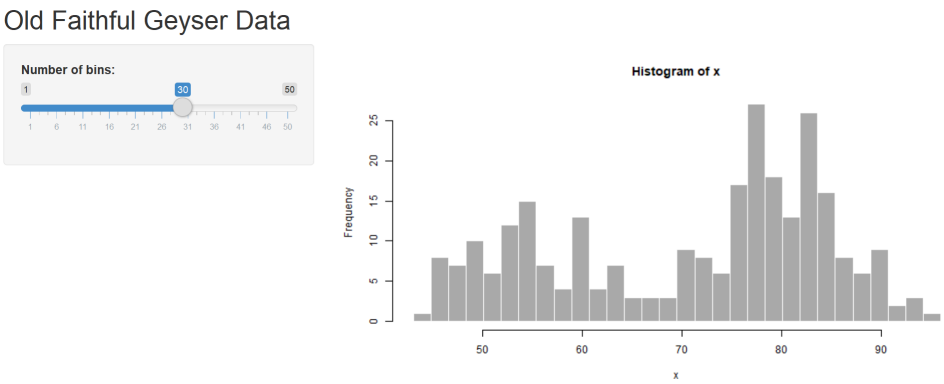


Figure 10.3: Default shiny template

Exercise 36 Design an Analytics Dashboard

In this exercise, the objective is to design a simple analytics dashboard which displays predicted flight departure delay. Users can use shiny to make predictions in real-time. The app includes serveral basic widgets such as date picker, drop-down menu and radio button. A sample screenshot is displayed in figure 36

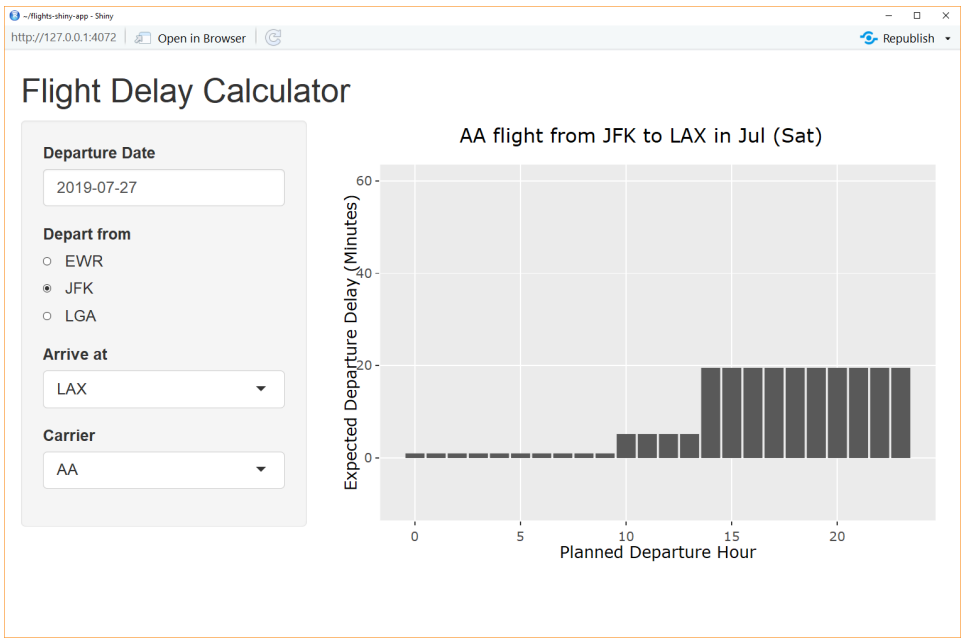


Figure 10.4: Interactive application displaying predicted flight departure delay.

Please load the package `nycflights13` and use the `flights` dataset. Steps are indicated in example 10.2.3.

R Example 10.2.3

```
# Load the package
library(nycflights13)
# Browse the flights dataset
# Aii flights departing from New York airports in 2013.
flights
# Browse the airlines dataset
# This is a small reference table containing full names of airlines
airlines
```

To begin with, users can launch the default `shiny` template and start editing. The code chunk in example 10.2.4 shows how the underlying model works. This is the main logic of the application and therefore should be placed in the `server` component of the application.

R Example 10.2.4

```
library(dplyr)
library(lubridate)
library(ggplot2)
# Build a simple model to predict flight departure delay
myFlightModel <- rpart(dep_delay ~
                         months(date) + weekdays(date) +
                         hour + origin + dest + carrier,
                       data = flights %>% mutate(date = make_date(year, month, day)),
                       na.action = na.omit,
                       control = rpart.control(cp = 0.0005))
# Predict the departure delay of a new flight
# Use shiny control widgets to capture user input
myNewFlight <- tibble(date = make_date(2019,5,10),
                      hour= 0:23,
                      origin="JFK",
                      dest = "SFO",
                      carrier="AA")
# Run the prediction
myNewFlightDelay <- predict(myFlightModel, myNewFlight)
# Visualise the prediction
tibble(hour=0:23, dep_delay = myNewFlightDelay) %>%
  ggplot(aes(x=hour, y=dep_delay)) +
  geom_col() +
  labs(x="Hours", y="Departure Delay (Minutes)")
```

10.3 Writing Packages

In R, functions are wrapped in packages so that they can be re-used and transported. All functions are documented in a standard way and usually include working examples.

Exercise 37 Create a Package

The objective of this exercise is to create a simple R package. To launch a package template, navigate to **File > New Project > New Directory > R Package**. Enter a package name such as `mypackage` then click **Create Project** as shown in figure 37. This should launch a package template.

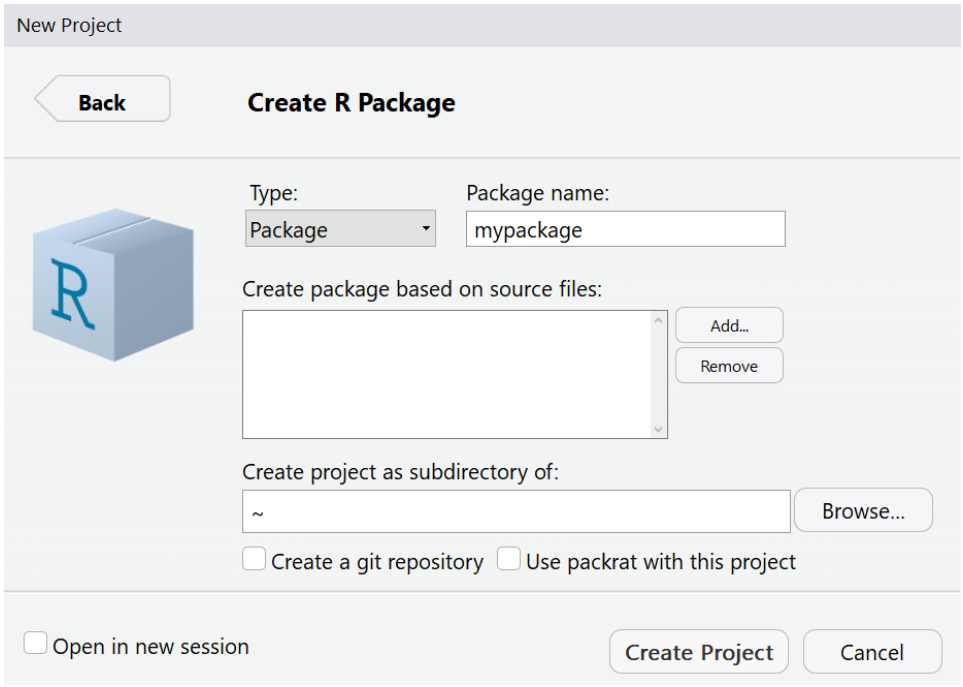


Figure 10.5: Launching an empty package template.

All functions in a package should be documented in a standard way. This can be done through the `roxygen2` package. To make sure `roxygen2` is enabled, navigate to **Build > Configure Build Tools** and check the box **Generate documentation with Roxygen**. Afterwards, click on **Configure** and check all boxes, this will ensure documentation is generated at all places. This is illustrated in figure 37.

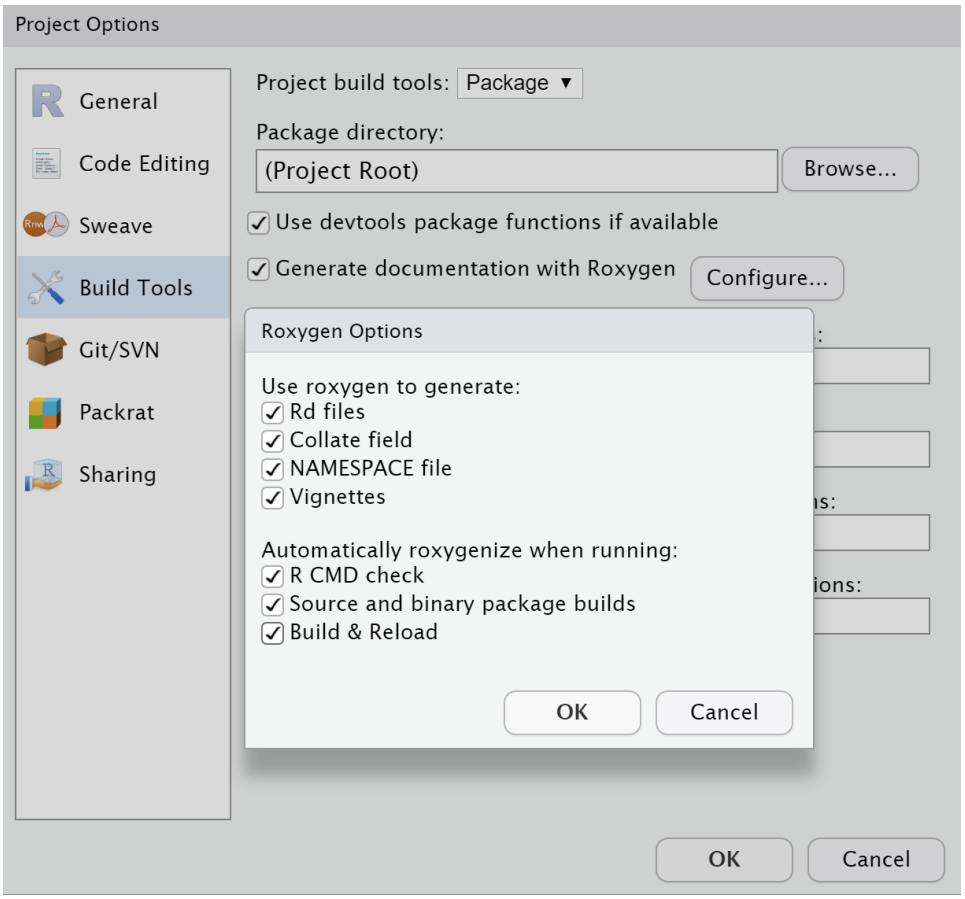


Figure 10.6: Enabling roxygen2 to generate documentation

Functions are located in the `R` directory. We will create a simple function called `is_even()` to check whether a given integer is even or not. Users can begin by nagivating to this directory and create a new file named `is_even.R` and enter the code chunk in example 10.3.1.

R Example 10.3.1

```r
#' Checks even number
#'
#' \code{is_even()} returns \code{TRUE} if the input is an even number.
#'
#' @author James Bond \email{jbond@@universialexports.com}
#'
#' @param x Input integer vector.
#' @return Logical vector
#'
#' @examples
#' myNumbers <- 1:20
#' is_even(myNumbers)
#'
#' @export
is_even <- function(x) {
  if(!is.integer(x)){
    stop("Input is not integer")
  }
  return (x %% 2) == 0
}
```

The `roxygen2` package picks up tags such as `@param` and turns them into standard R documentation. It is particularly important to include `@export` at the end, as this will ensure the function is exported so that users can access it.

Before moving on, users can open up the `DESCRIPTION` file and modify the package details. It is crucial that the details are up-to-date after every code change, especially the version number.

```
Package: mypackage
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
Maintainer: The package maintainer <yourself@somewhere.net>
Description: More about what it does (maybe more than one line)
    Use four spaces when indenting paragraphs within the Description.
License: What license is it under?
Encoding: UTF-8
LazyData: true
RoxygenNote: 6.0.1
```

Once the user has finished editing everything, the source code is ready to be wrapped as a package. To do this, the user can navigate to **Build > Check Package**. This will check the source code and ensure the code is ready for build. Afterwards,

the user can click on **Build > Build Source Package**, which will wrap all the code into a `.tar.gz` package file. The resulting file is a standard R package which can be transported to any user on any platform.

Another way to wrap the package is to click on **Build > Install and Restart**. This builds the package and installs it directly into the user's library. In this case, the user can simply call the library and start using the custom functions rigth away.

R Example 10.3.2

```
library(mypackage)
# This will return a logical vector
is_even(1:20)
# This will error
is_even(3.141)
# Lookup the documentation
?is_even
```

10.4 Reproducibility

The principle of reproducibility suggests that a process should always produce same results if the input remains identical. Reproducible results are crucially important for business analytics for various kind of reasons, such as compliance, quality assurance, KPI measurement... etc. The open source nature of `R` and many of its packages means they get very frequent updates. There can be substantial differences between versions of the same package.

A dependency management system called `packrat` can be used to ensure reproducibility. It creates a private library for the project and searches for packages only in this location. Shared libraries would not be used in a `packrat`-enabled project. This means the project is totally isolated and updating packages in a shared library would not break the project.

Users can enable `packrat` by navigating to **Tools > Project Options > Packages** and enable the option **Use packrat with this project**. The system will then scan for packages and put them in a private library.

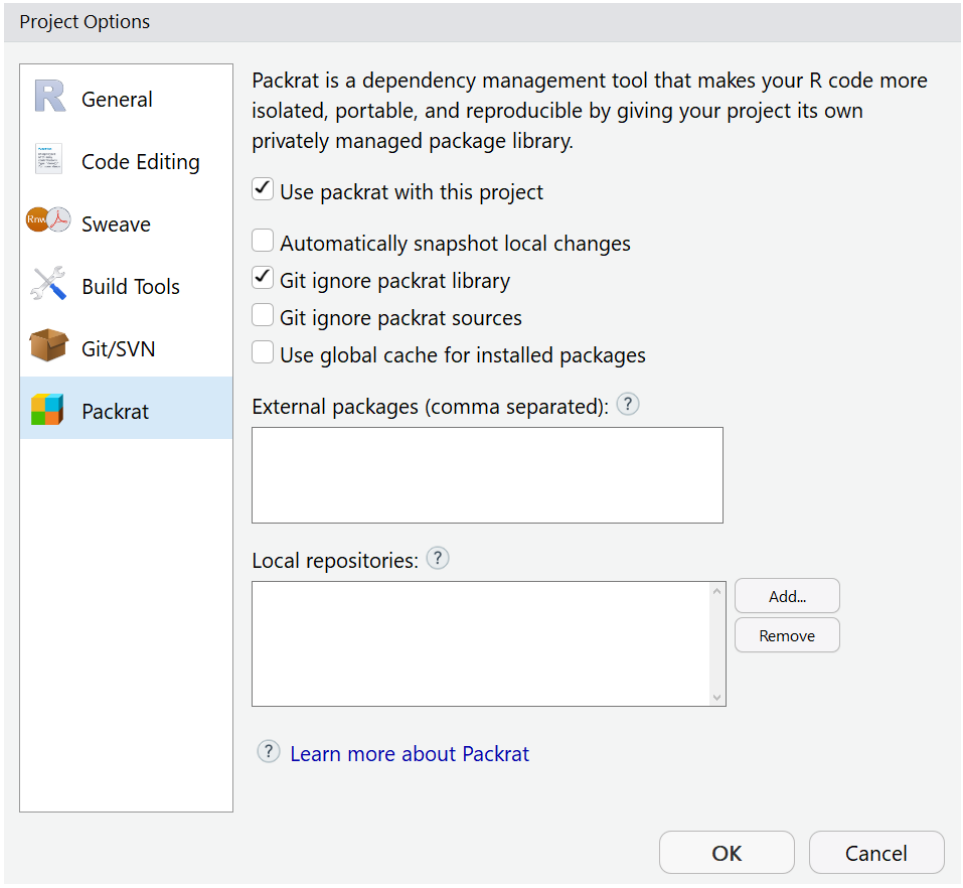


Figure 10.7: Enabling packrat dependency management system

Chapter 11

Efficient Programming

Computational power is a scarce resource, especially when most base R functions run in memory. This chapter dives deeper into memory and CPU management.

11.1 Memory Usage

All integer and numeric values in R are stored as double-precision values. This means each number consumes 64 bits of system memory. Users should notice that significant figures have no effect on the object size. In other words, the values 3.14 and 3.14159265359 have identical object size.

The package pryr can be used to track memory usage, as shown in example 11.1.1.

R Example 11.1.1

```
library(pryr)
# Displays the size of an integer value
# Note the trailing 'L' character, this forces the number to become an integer
object_size(3L)
# Significant figure of a numeric valule has no effect on object size
# Therefore these two values have the same object size
object_size(3.14)
object_size(3.14159265359)
# Creates an integer vector
# This contains 1 million integers
# Consumes 4 MB memory
myBigVec1 <- 1:1e6
object_size(myBigVec1)
```

In general, object size grows proportionally with the length of the object. However, memory is allocated to R objects in large blocks in exchange for faster processing. Example 11.1.2 shows two character objects with different length having the same object size.

R Example 11.1.2

```
# Both character objects have identical object size.
# Each consume 120 bytes.
object_size("My name is Bond.")
object_size("My name is Bond, James Bond.")
```

It is important to use data type correctly in order to conserve memory. Example 11.1.3 compares the object size of a character vector versus a logical vector.

R Example 11.1.3

```
# Object size of logical value is much smaller than character value
# In this case, the character vector is three time larger
object_size(c("Yes", "No", "No", "Yes"))
object_size(c(TRUE, FALSE, FALSE, TRUE))
```

The `mem_used()` function in the `pryr` package identifies the total memory used by all objects. This does not include memory used by the R interpreter itself, the `bash` shell wrapping R, nor memory used by RStudio IDE. Example 11.1.4 shows how the function is used.

R Example 11.1.4

```
# Returns the total memory used by all R objects
mem_used()
```

Useful function such as `mem_change()` can be used to measure how much memory was changed from an operation. This is shown in example 11.1.5.

R Example 11.1.5

```
# Assign a large vector
# This will consume memory (+4 MB)
mem_change({ myBigVec2 <- 1:1e6 })
# Remove a large vector
# This releases memory (-4 MB)
mem_change({ rm(myBigVec2) })
```

11.2 Profiling

R code can be profiled to identify the memory and execution time bottleneck. The package profvis works with RStudio IDE to offer an interactive interface for visualising profile data.

The profvis() function wraps an R operation inside a set of curly brackets {}, as shown in example 11.2.1. This example generates some random numbers, runs a simple linear regression model and views the model summary.

R Example 11.2.1

```
library(profvis)
profvis({
  # Generate some random numbers from normal distribution
  myDataX <- rnorm(1e6)
  myDataY <- rnorm(1e6)
  # Run a linear model
  myModel <- lm(myDataY ~ myDataX)
  # View the model summary
  summary(myModel)
})
```

This produces an interactive output with two parts, as shown in figure 11.2.1. The top part is the code with the profiled memory and time. It shows how much memory was allocated (positive) and deallocated (negative) by the function call. The bottom part is a flame graph which shows the call stack. The time spent in each call stack is indicated by the width of the block.

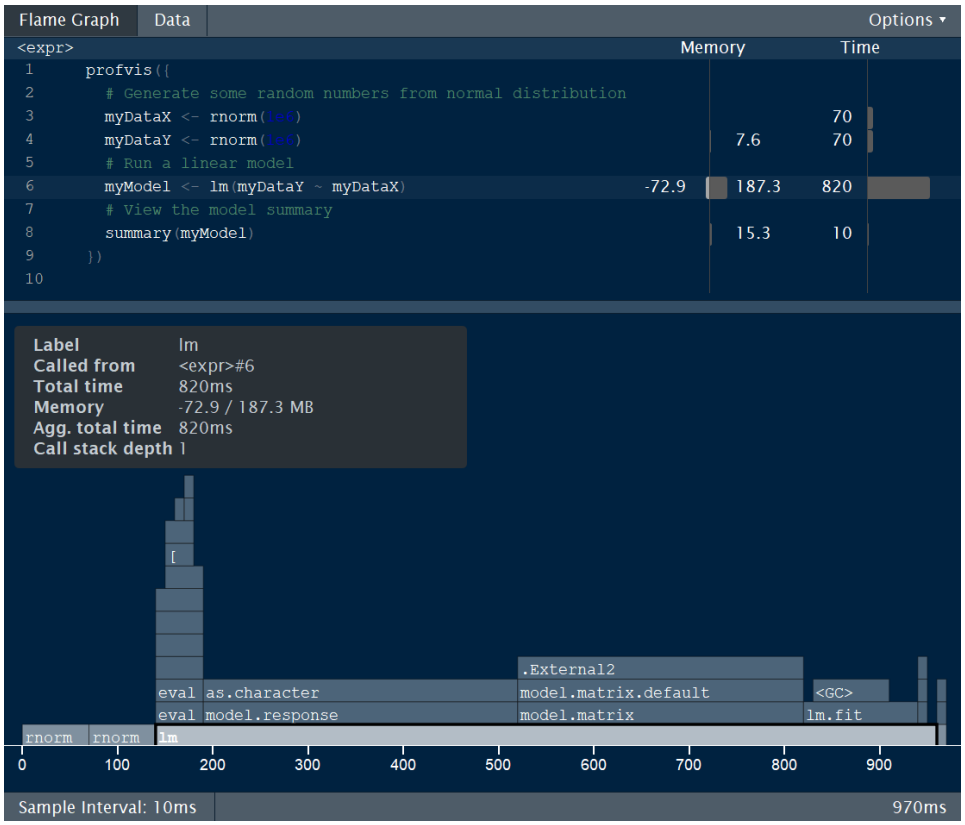


Figure 11.1: Flame graph of profiling output

11.3 Multithreaded Processing

Base R[1] was historically designed to run in a single thread. Today, many computers have multiple cores or even multiple sockets. To make use of the computational power, many functions in R can be parallelised on a multi-core system for faster processing.

Microsoft R Open (MRO)[2] is a popular distribution of R which offers multi-threaded processing. It uses Intel Math Kernel Library (MKL) which provides BLAS and LAPACK subroutines. This allows key mathematical operations to be performed on all threads.

Example 11.3.1 shows how to detect and change MKL threads in MRO. There

[1] GNU-R

[2] https://mran.microsoft.com/open

is no need to modify the MKL thread number under normal circumstances.

R Example 11.3.1

```
# Detects how many threads are in use
getMKLthreads()
# Change to 4 threads
setMKLthreads(4)
# Reset to system default value
setMKLthreads()
```

Multithreading can also be used to optimise `apply()` family functions and `for` loops. Examples 11.3.2 and 11.3.3 show how a simple `sapply()` function can be executed using the `parSapply()` function in the `parallel` package. It sets up a cluster of workers which run simultaneously, where code is executed in parallel. The cluster is stopped to release resources once the work has finished.

R Example 11.3.2

```
system.time({
  sapply(1:10, function(x) {Sys.sleep(0.25)})
})
```

R Example 11.3.3

```
library(parallel)
system.time({
  # Detect how many cores are available on the system
  myCores <- detectCores()
  # Create a parallel cluster
  # Use 25% of system cores
  myCluster <- makeCluster(floor(myCores * 1/4))
  # Run the same process on multicore
  parSapply(myCluster, 1:10, function(x) {Sys.sleep(0.25)})
  # Stop the cluster
  stopCluster(myCluster)
})
```

Multithreading does not always guarentee top speed. This is because it has very significant overhead due to starting and closing the threads. In addition, multithreaded processes may cause serious issues on multi-user systems, as users may compete for computational resources at the same time. Besides, using many threads simultaneously may block other users' processes. This should always be used with great care.

Chapter 12

Distributed Computing

Very large datasets which cannot fit in memory need to be processed differently. There are several big data tools which support large-scale processing. The simplest way is to carve up the large dataset into smaller chunks and process them seperately. Once completed, the overall results can be reconstructed.

A slightly more advanced way is to process it natively in a Hadoop cluster which hosts the large dataset. There are plenty of algorithms which can be executed on Hadoop, these are mainly provided by Apache Spark MLlib.

12.1 Apache Spark

Apache Spark can be accessed in `R` via the `sparklyr` package, which supports the full `dplyr` pipeline. It features low-latency computation by caching the working dataset in memory and performing computations at memory speed. Ideally, Spark can run on a YARN cluster to maximise the benefits of distributed computing. It also supports local mode which runs on a single host. Spark can run on several modes:

Standalone This is also known as local mode. The driver and executors run on the client machine. Although there is no distribution of computing process, prototyping and debugging in the local mode is more convenient.

YARN Client Spark driver is launched in the same process as the client that submits the application. Resource allocation is done by YARN Resource Manager (RM) based on data locality. Driver program on client machine controls the executors on the YARN cluster.

YARN Cluster The Spark client submits an application to the YARN cluster. Both the driver and executors run on the YARN cluster.

In Spark, source data can accessed through different ways. Files in CSV format can be accessed directly in the local file system. Alternatively, various file formats such as CSV, JSON and parquet can be uploaded to HDFS and read from there. Spark also supports other object types, such as `data.frame`, `tibble` objects and even Hive tables in Hadoop as data sources.

Exercise 38 Read / Write in Spark

In this exercise, we will learn how to set up Spark and perform simple aggregation and read/write operations. To begin with, we will check whether Spark is installed or not. If not, we will invoke the installation command `spark_install()`, as shown in example 12.1.1.

R Example 12.1.1

```r
library(sparklyr)
library(dplyr)
# Checks which spark versions are available
spark_available_versions()
# Checks which version is installed.
spark_installed_versions()
# Install spark
# Only do this if Spark is not already installed
spark_install()
```

To open a Spark connection, follow the code in example 12.1.2.

R Example 12.1.2

```r
# Opens the Spark connection through sparklyr
# Using standalone mode (local master)
mySparkConn <- spark_connect(master = "local")
# Checks whether the connection is opened
mySparkConn %>% connection_is_open()
# Checks Spark version
mySparkConn %>% spark_version()
```

Once we have set up Spark connection, we can learn how to convert local data to Spark and perform aggregation pipeline in `dplyr` style. This is shown in example 12.1.3.

R Example 12.1.3

```
# Copy a local R data frame to Spark as SDF
mySparkConn %>% sdf_copy_to(mtcars, overwrite = TRUE)
# Sets up a connection to Spark data frame (SDF)
myTbl <- mySparkConn %>% tbl("mtcars")
# Browse the top rows of the SDF
head(myTbl)
# Performs SQL-style aggregation using dplyr framework
# It runs natively on Spark
myTbl %>%
  group_by(cyl) %>%
  summarise(avg_mpg = mean(mpg, na.rm=TRUE))
```

Spark standalone mode can read and write files in user's local file system. Example 12.1.4 shows

R Example 12.1.4

```
# First, write the mtcars SDF to CSV
myTbl %>% spark_write_csv("file:/home/YOUR_USER_NAME/my_data/", mode = "overwrite")
# Cchange user to your username/lanid
myCsvLocal <- mySparkConn %>%
  spark_read_csv(name = "mtcars_local",
                 path = "file:/home/YOUR_USER_NAME/my_dat/")
myCsvLocal
```

Exercise 39 Data Visualisation in Spark

Large datasets are especially hard to visualise. In Spark, we can user the extension package `dbplot` to visualise very large datasets. The summary statistics is computed in Spark, which means that the visulisation process is fully scalable. Follow the code in example 12.1.5.

R Example 12.1.5

```
library(nycflights13)
library(ggplot2)
library(dbplot)
# Upload the flights dataset to Spark
flights_tbl <- mySparkConn %>% sdf_copy_to(flights, overwrite = TRUE)
# Create a line plot showing the mean departure delay
myChart <- flights_tbl %>%
  dbplot_line(month, mean(dep_delay)) +
  labs(title = "Flights - average departure delay per month",
       ylab = "Average departure delay",
       xlab = "Month") +
  scale_x_continuous(breaks= seq(1,12,1))
# Plot the graph
myChart
# Compute a histogram of the air_time variable
myHistogram <- flights_tbl %>%
  filter(!is.na(air_time)) %>%
  dbplot_histogram(air_time, binwidth = 50) +
  labs(title = "Flights - air time") +
  theme_light()
# Plot the histogram
myHistogram
```

Exercise 40 Train a Random Forest Model

The `sparklyr` package supports Spark MLlib, which is a library for scalable machine learning algorithms. We can run a linear regression model using this package. Example 12.1.6 shows how to run a random forest model on Spark.

R Example 12.1.6

```
# Remove NA data and divide the dataset into two halves
flights_tbl %>%
  na.omit() %>%
  sdf_partition(training = 0.5,
                testing = 0.5) %>%
  sdf_register(c("flights_training", "flights_testing"))
# Get a pointer object for the training and testing datasets
flights_training <- mySparkConn %>% tbl("flights_training")
flights_testing <- mySparkConn %>% tbl("flights_testing")
# Run a random forest model
# This can take some time
myFlightsModel <- flights_training  %>%
  ml_random_forest(dep_delay ~ month + hour + origin + dest + carrier)
# Run the model prediction on both training and testing set
myFlightsPredictionTraining <- myFlightsModel %>% ml_predict(flights_training)
myFlightsPredictionTesting <- myFlightsModel %>% ml_predict(flights_testing)
# Evaluate model performance using Mean-squared error (MSE)
ml_regression_evaluator(myFlightsPredictionTraining,
                        label_col = "dep_delay",
                        metric_name = "mse")
ml_regression_evaluator(myFlightsPredictionTesting,
                        label_col = "dep_delay",
                        metric_name = "mse")
```

Exercise 41 Distributed R

In some cases, users may want to implement custom logic on a large dataset. Spark allows user to run any custom R code. In example 12.1.7, we create a sample dataset in Spark and use the function `spark_apply()` to run an arbitrary R function on it.

R Example 12.1.7

```
# Create a SDF with 100 rows (with values from 1 to 100)
myData <- mySparkConn %>% sdf_len(100)
# Runs arbitrary R code on all Spark executors
# This is a native R function
myData %>% spark_apply(function(df) { df * 10 })
```

Bibliography

[1] Rasmus Bååth. The state of naming conventions in r. *The R journal*, 4(2):74–75, 2012.

[2] Hadley Wickham and Garrett Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc., 1st edition, 2017.