Business Analytics in R

Introduction to Statistical Programming

TIMOTHY WONG[1]
JAMES GAMMERMAN[2]

March 19, 2018

[1] timothy.wong@hotmail.co.uk
[2] person@organisation.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/timothy-wong/r-training`

Tim
`timothy.wong@hotmail.co.uk`

Chapter 1

Introduction to Data Analysis

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 1.

You may being 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 a fundamentally 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 do not scale particularly well because it requires human

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

interpretation.

Models are complementary tools to visualisation. Once the research hypothesis 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.

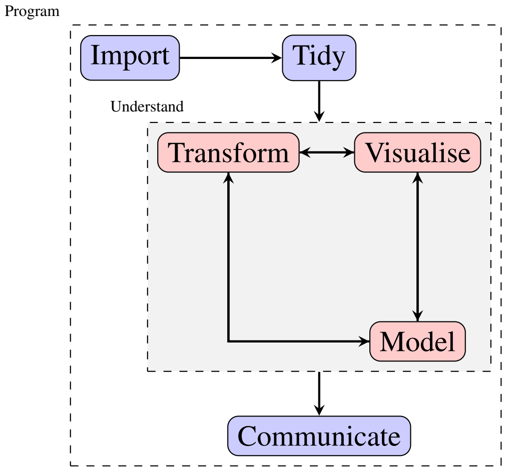


Figure 1.1: Stages of data analysis

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`

1.1 Data Transformation

In most of the time, data does not come to you in exactly the right format. Often you need to compute new variables, or to summaries 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.

Exercise 1 Filtering Observations

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.

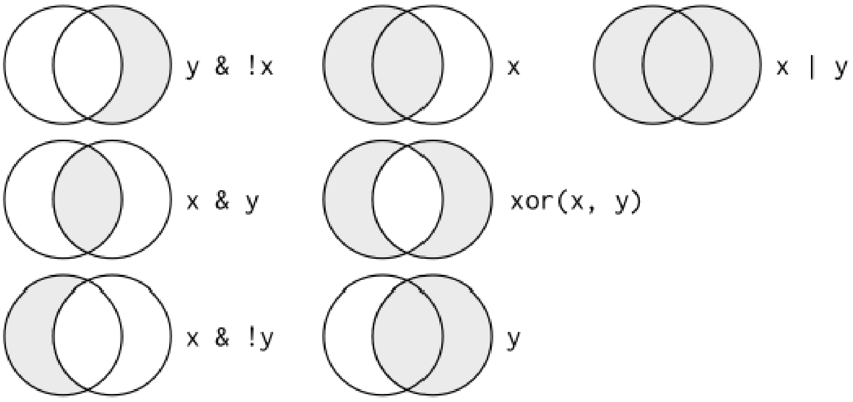


Figure 1.2: Logical operators in R

Missing values are represented in R as `NA`. They are contagious because any operation involving an unknown value will produce unknown results. For instance, 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 1.1.1, We will learn how to use `tidyverse` using a dataset on flights departing from New York City in 2013.

R Example 1.1.1

```
# load the required packages
library(dplyr)
library(nycflights13)
# Inspect the data, noting the type of each variable
# You also can use View(flights) to open the dataset
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)
```

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, American or Delta.
4. Departed in summer (July, August and September)
5. Arrived more than two hours late, but did not leave late.
6. Were delayed by at 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?

Exercise 2 Arranging Observations

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 preceding variable. Use `desc()` to reorder variable in descending order. Missing values are always sorted at the end. Example 1.1.2 shows how the `arrange()` function can be used.

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

R Example 1.1.2

```
# 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))
```

Sort the `flights` dataset to find:

1. The fastest flights.
2. The most delayed flights.
3. The flights that left earliest.
4. Which flights travelled the longest?

Exercise 3 Selecting 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.

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 1.1.3:

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 1.1.3

```r
# 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 = talinum)
# Reorder columns using the everything() helper
select(flights, time_hour, air_time, everything())
```

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?

Exercise 4 Computing New 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 ranks

In example 1.1.4, we will start by creating a narrower dataset so we can see the new variables.

R Example 1.1.4

```
# Select several columns only
flights_sml <- select(flights,
                      year:day,
                      ends_with("delay"),
                      distance,
                      air_time)
# Use these the smaller data frame 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)
min_rank(desc(y))
```

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 new variable `air_time` as `arr_time - dep_time`. What do you expect to see? What problem do you see? What do you need to do to fix it?

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

Exercise 5 Grouped Summaries

The last key function is `summarise()` or `summarize()`. It collapses a data frame into a single row. 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.

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.

Useful summary functions:

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()`

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 1.1.5.

R Example 1.1.5

```
# Example of summarise() function alone
# You can also use summarize()
# These two 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))
```

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 2

Regression Models

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

2.1 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 as the intercept, while the β_1 value is called the slope.

Simple linear model can be extended to include a high-order polynomial term of variable x. It provides higher model flexibility so that 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 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 residual.

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 independent variable x to predict dependent variable y in the data frame object `myData`.

Dummy variable 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 model 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. It 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 model can be expressed as equation (2.1).

$$\hat{y}_i = \underbrace{\beta_0}_{\text{Intercept}} + \underbrace{\beta_1 x_{1,i} + \beta_2 x_{1,i}^2 + \beta_3 x_{1,i}^3}_{\text{Cubic polynomial term}} + \underbrace{\beta_4 x_{2,i}}_{\text{Linear term}} \tag{2.1}$$

Interaction refers to the combined effect[2] of more than one independent variables. For example, independent variable `x1` and `x2` might have no effect on 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 interaction term and the arguments will be treated as simple arithmetic operations.

Exercise 6 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 2.1.1,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 2.1.1

```r
#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 2.1.2 shows how you can use `ggplot2` to visualise the dataset.

R Example 2.1.2

```
# 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 an univariate regression model. The function `summary(myModel1)` would print out all the key results of the model. This is demonstrated in example 2.1.3.

R Example 2.1.3

```
# 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 2.1 lists the key items alongside a short description.

Table 2.1: 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 above 0.05 are considered statistical 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 2.1.4 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 `factor` data type, the `lm()` regression function would handle it automatically without having to state it in the formula.

R Example 2.1.4

```r
# 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, wou can load the `car` package and use the function `avPlots()` to view the partial regression plot[3]. This would graphically display the effect of individual variables while keeping others in control. This is shown in example 2.1.5.

R Example 2.1.5

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

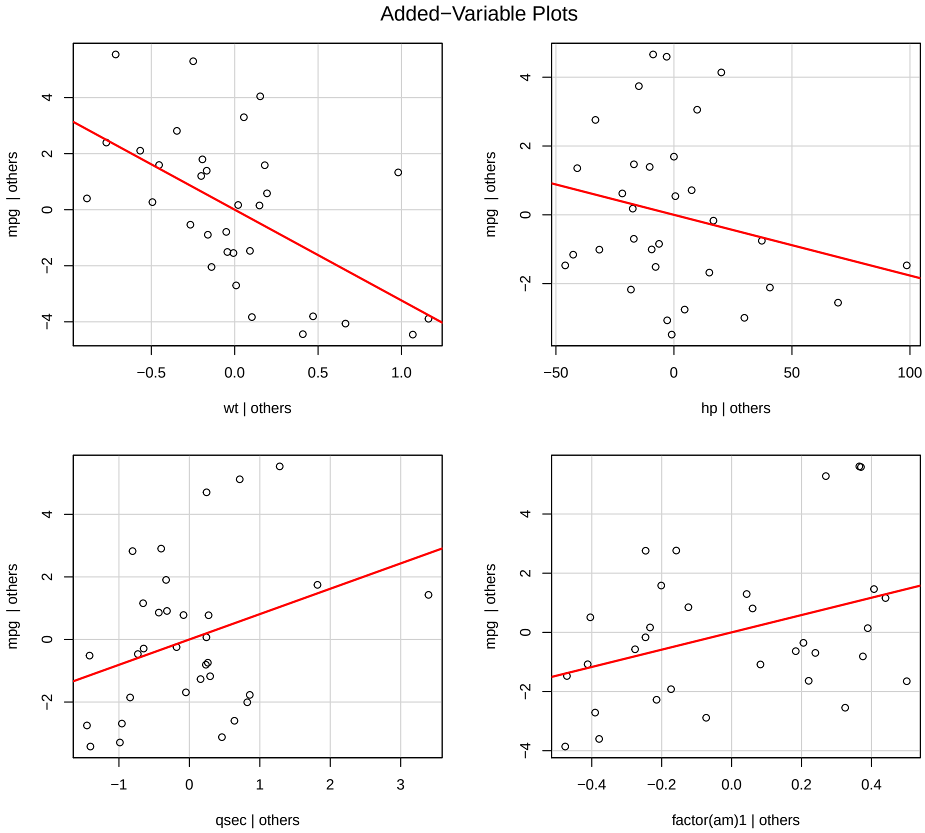


Figure 2.1: Partial Regression plots

As the input predictor variables usually have different scale, the model coefficient are not directly comparable. To fairly compare the effect magnitude of the predictor variables, they need to be standardised first. The `QuantPsyc` package has a function `lm.beta()` which performs coefficient standardisation. The magni-

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

tude of the standardised coefficient indicates the relative influence of each predictor variable. You can follow example 2.1.6 for coefficient standardisation.

R Example 2.1.6

```
library(QuantPsyc)
lm.beta(myModel1)
lm.beta(myModel2)
lm.beta(myModel3)
```

You can also compare nested models[4] using ANOVA technique. Example 2.1.7 compares three nested models by applying Chi-square test on the model residuals to check for statistically significant differences. In most cases, the simpler model is prefereable if the candidate models are not significantly different.

R Example 2.1.7

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

Exercise 7 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 2.1.8 generates several diagnostic plots for a standard linear regression model object.

R Example 2.1.8

```
plot(myModel3)
```

[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 $\hat{y} = x_1 + x_2$.

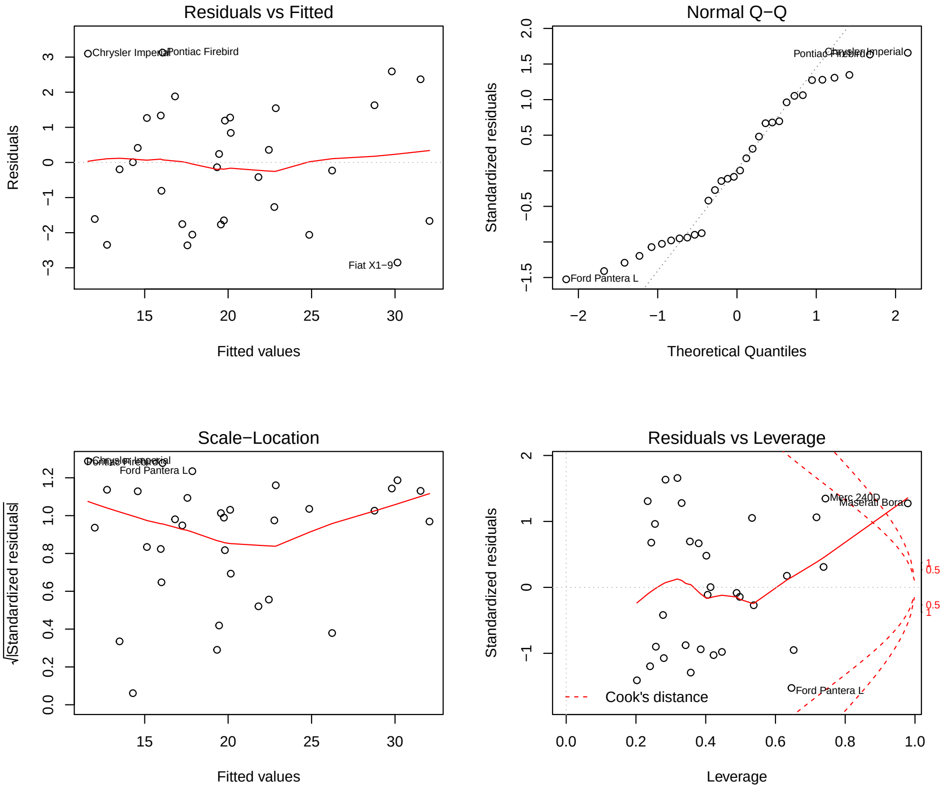


Figure 2.2: Regression diagnostic plots

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 line indicate poor model fit with possible non-linear effects.

Normal Quantile-Quantile One the of main assumptions of linear regression is that the residual is drawn from 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 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, 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 standard linear model is assumed to be homoscedastic, the residual variance should not vary along the range of fitted values (i.e. expects a near-horizontal line). If the standardised residual forms a distinguishable patten (e.g. fanning out or curvy), then the model may be heteroscedastic and hence violates 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 observations 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 8 Model Overfitting

Linear regression can be made more flexible by increasing the order of the polynomial terms. This allows 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 inference and are ungeneralisable. The code snippet in example 2.1.9 runs a linear model with variable level of flexibility.

R Example 2.1.9

```r
# 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')
```

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

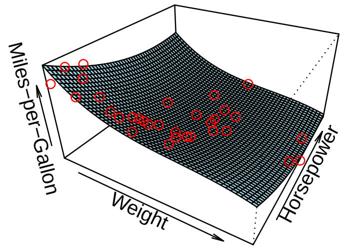


Figure 2.3: A less flexible model showing better generalisability

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

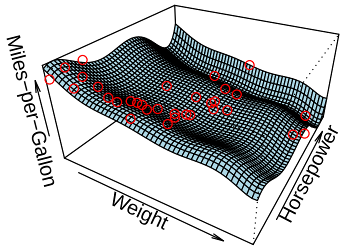


Figure 2.4: A more flexible model illutrating the risk of overfitting

2.2 Poisson Regression

In the previous section, simple linear regression model assumes 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 where it is often a positive integer ($\mathbb{N} = \{0, 1, 2, 3, ...\}$). In this case, Poisson regression can be used. It is based on 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.

Poisson regression model can take into account multiple predictor variables. Equation (2.2) shows a Poisson regression model with $\hat{y}$ as dependent variable and M predictor variables.

$$\hat{y} = e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_M x_M} \tag{2.2}$$

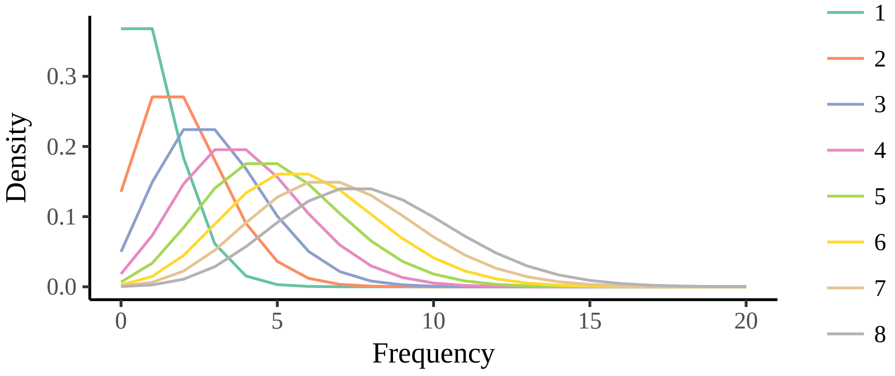


Figure 2.5: Poisson distribution with different λ values

Exercise 9 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 2.2.1, 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 2.2.1

```r
# 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 2.2.2 visualises the results. The vertical bars would fill the positive space if the input data fits 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 2.2.2

```r
# 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 was 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)
```

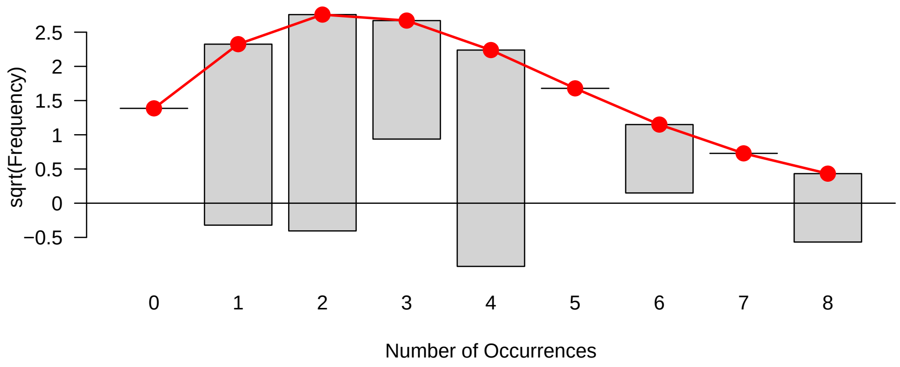


Figure 2.6: Goodness-of-fit test for Poisson distribution

Exercise 10 Building a Poisson Model

Example 2.2.3 trains a Poisson regression model using the `mtcars` dataset. The variable `carb` is used as 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 linear model.

R Example 2.2.3

```
# 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()
```

2.3 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 (2.3). Logistic function transforms a real-valued number X into the range $(0,1)$ which represents the outcome probability $P(Y) \in (0,1)$.

$$P(Y) = \frac{1}{1+e^{-X}} \tag{2.3}$$

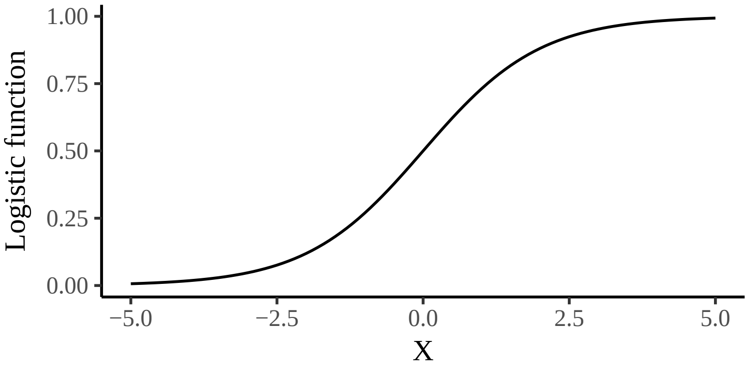


Figure 2.7: Graph showing the range of a logistic function

In a univariate scenario, the logistic function can be expressed as equation (2.4), where β_0 represents the intercept while $\beta_1, \beta_2, \beta_3, ..., \beta_M$ refers to the regression coefficient of variables $x_1, x_2, x_3, ..., x_M$. The output $P(Y)$ indicates the likelihood of true outcome.

$$P(Y) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_M x_M)}} \tag{2.4}$$

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. Odds ratio is defined in equation (2.5).

$$OR(x_1) = \frac{odds(x_1+1)}{odds(x_1)} = \frac{e^{\beta_0+\beta_1(x_1+1)+\beta_2 x_2+...+\beta_M x_M}}{e^{\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_M x_M}} = e^{\beta_1} \tag{2.5}$$

Logistic regression can only handle binary problem. 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.

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

Exercise 11 Building a Logistic Model

In this exercise, we would 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 2.3.1, 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 ratio and analyse the effects of each variable. The last part of the code is to calculate the model accuracy.

R Example 2.3.1

```r
# 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 2.3.1 can be expressed as equation (2.6).

$$\begin{aligned} P(manual) &= \frac{1}{1+e^{-(\beta_0+\beta_1 x_{mpg}+\beta_2 x_{hp}+\beta_3 x_{disp})}} \\ P(automatic) &= P(\neg manual) \\ &= 1 - P(manual) \end{aligned} \tag{2.6}$$

Chapter 3

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.

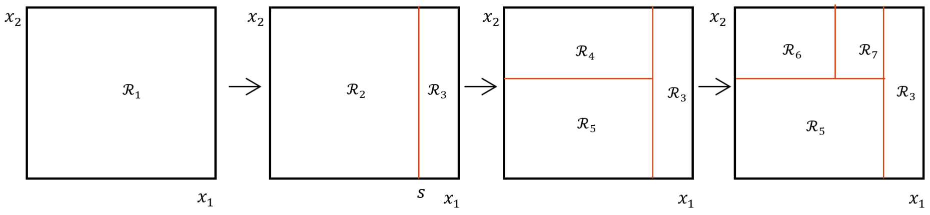


Figure 3.1: Recursive partitioning

3.1 Decision Trees

Decision tree is the simplest form of any tree-based models. It uses standard recursive partitioning to produce prediction regions. Decision tree is a very generic algorithm which fits both regression and classification problem. 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 12 Growing a Decision Tree

In the R language, there are many packages which implement tree-based algorithm. In exercise 3.1.1, we will use the `rpart` function in the rpart package to build a simple decision trees for a regression problem. The aim of this exercise is to predict car efficiency (`mpg` variable) using the `mtcars` dataset.

R Example 3.1.1

```r
# 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)
```

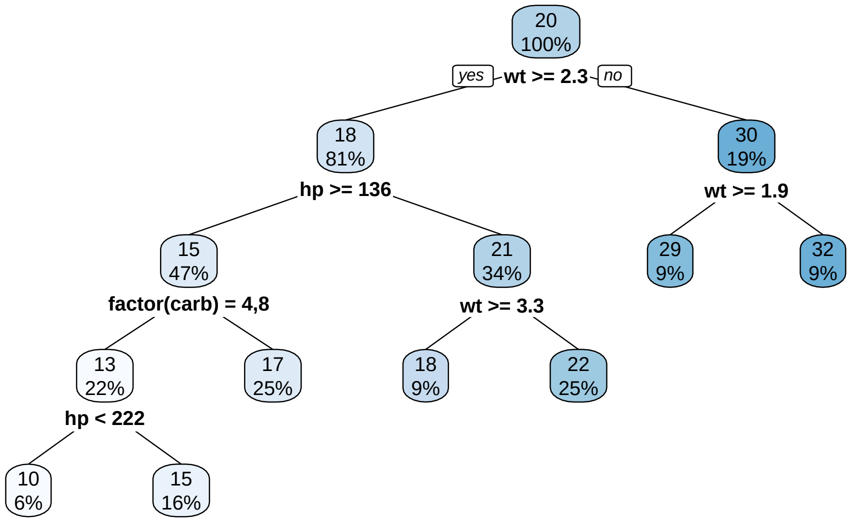


Figure 3.2: Decision tree for a regression problem

Exercise 13 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 3.1.2, 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 approachs 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 3.1.2

```
# 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)
```

3.2 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 would have strong prediction power while some others would 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 prediction accuracy of random forest is far better than 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 14 Planting a Random Forest

Example 3.2.1 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 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 3.2.1

```
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 4

Neural Networks

Artificial neural networks (ANN) are mathematical algorithms inspired by the structure of the biological brains. It process incoming information through non-linear mechanism in a neuron and pass 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 complex structure are usually trained iteratively using backpropagation techniques over 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 $f(x)$. Sigmoid function is a traditional choice of activation function for ANNs(4.1a). 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 becomes a popular choice for ANN activation function. However, it suffers from 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(4.1b). It shares the same sigmoidal shape but further stretches the output to the range $(-1, 1)$ therefore provides 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 strong gradient (i.e. unsaturable) for any positive input (4.1c). However, it has been considered computationally costly as it contains logarithm and exponential terms. In light of this, a simplified version call rectified linear unit (ReLU) is usually used instead (4.1d).

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 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 4.1.

Sigmoid activation

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.1a}$$

Hyperbolic tangent activation

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4.1b}$$

Softplus activation

$$f(x) = \ln(1 + e^{x}) \tag{4.1c}$$

Rectified linear unit (ReLU)

$$f(x) = \max(0, x) \tag{4.1d}$$

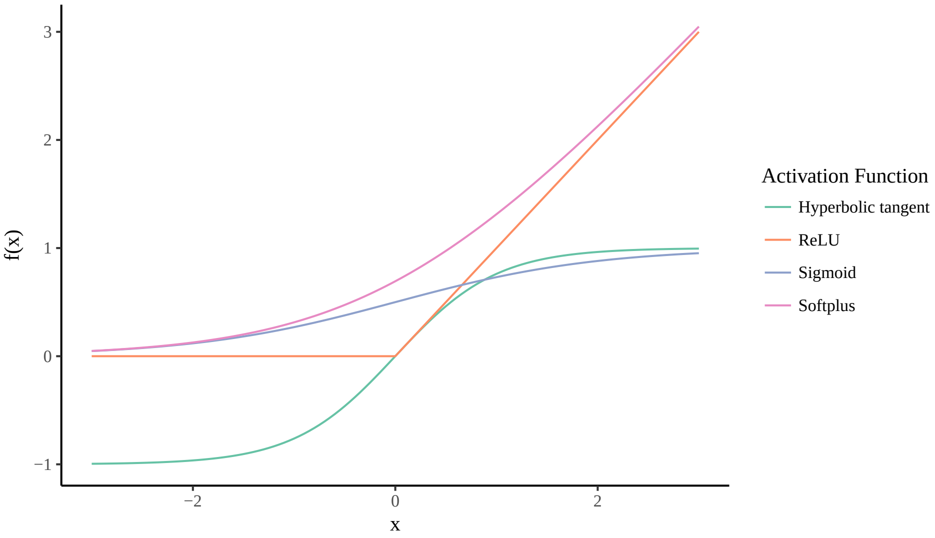


Figure 4.1: Common neural activation functions

The output prediction is made at the network's final layer. Each neuron at this layer combines the hidden neuron's activation though weighted sum and a bias adjustment unit (4.2a). For regression problems, a linear activation layer is usually used to map the output vector back into unbounded real value range (4.2b). For classification problem, softmax function is used as the final output layer. It maps the hidden layer's activations into the range $(0, 1)$ where the sum of the entire output vector is restricted to1 in order to represent class probabilities (4.2c).

Weighted sum of hidden vector with bias adjustment

$$y_k = \beta_k + \sum_{h=1}^{H} w_{h,k} h_h, k = 1, ..., K \tag{4.2a}$$

Linear output

$$\hat{Y_k} = y_k \tag{4.2b}$$

Softmax output

$$\hat{Y_k} = \frac{e^{y_k}}{\sum_{k'=1}^{K} e^{y_{k'}}} \tag{4.2c}$$

4.1 Multilayer Perceptron

A multilayer perceptron (MLP) is the simplest form of all neural network. It consists of several stacked hidden layers, where every 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-score (4.3a) 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 standard deviation σ of the variable.

$$z_i = \frac{x_i - \bar{x}}{\sigma} \tag{4.3a}$$

At the training phase, network weights are usually initialised randomly. They are then optimised through backpropagation to achieve gradent descent. The weights improves gradually according to a predefined learning rate until they ultimately converge at the minimum value. One of the drawbacks is that backpropagation does not guarantee reaching 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 stronger gradient, thus speed up optimisation through gradient descent.

Exercise 15 Training MLP for Regression Problem

There are many packages which implements neural network in the R language. Traditional packages include neuralnets, nnet, RSNNS, caret... etc. Modern deep learning frameworks such as h2o, MXNet and kerasR 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 difference usually lies 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. Code in example 4.1.1 trains a fully-connected multilayer perceptron with two hidden layers.

R Example 4.1.1

```r
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 4.1.2 shows that you can load the package NeuralNetTools to create better topology plot. In addition, you can plot the observed data against network predictions to visualise the error. For a regression problem, 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 4.1.2

```
# 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)
```

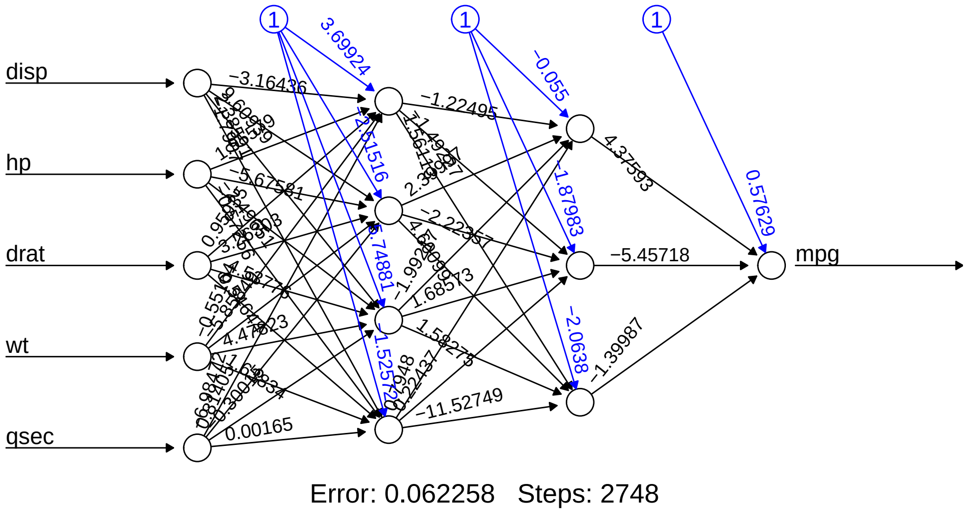


Figure 4.2: MLP model with two hidden layers

Sigmoid function is traditionally used in shallow networks. It can suffer from weak gradient when stacked in deep networks. You can use the code in example 4.1.3 to customise the activation function.

R Example 4.1.3

```
# 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]. Code in example 4.1.4 converts all categorical variables in the `mtcars` dataset into one-hot encoding. The encoded values afre then binded to the 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 variables would have value $\{1, 0\}$. Please note that this is different from dummy encoding in statistical modelling.

R Example 4.1.4

```r
# 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)
# View the dataset
View(mtcars_all)
# 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
```

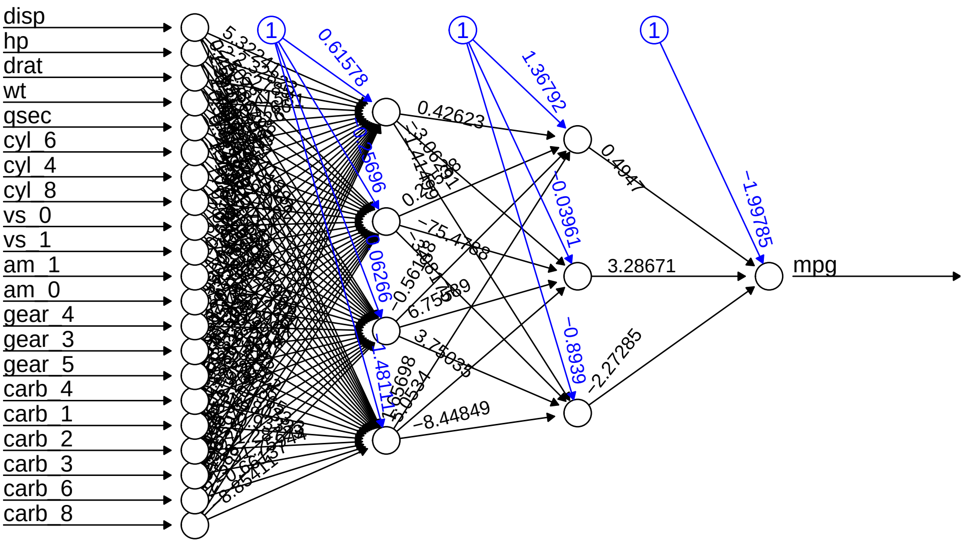


Figure 4.3: MLP model with one-hot encoded categorical variables

Chapter 5

Time Series Analysis

Many datasets have 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. This include trend, seasonality, or temporal dependency between different variables. Studying these properties can gain useful insights. For example, extrapolating trend can create forecast for the future. Alternatively, analysing seasonality can help users understand the nature of recurring patterns.

5.1 Auto-Correlation Function

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 lag period increases. 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 when observations 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 dimsnion so that only the contribution of an individual lag is reflected.

Exercise 16 Loading Datasets

In this exercise, we would 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. Code in example 5.1.1 show how to load the dataset from CSV file.

Table 5.1: 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

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

R Example 5.1.1

```
# Read the Amprion dataset from CSV file
# You might have to modify the path to point to your file location
amprion <- read.csv('amprion.csv',
                    colClasses = c('character',
                                   'numeric',
                                   'numeric',
                                   'numeric')) %>% as_tibble()
# View the dataset
amprion
```

```
## # A tibble: 140,240 x 4
##                datetime demand     pv     wp
##                   <chr>  <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
```

One of the main drivers of power demand and generation is 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 5.1.2 to load the dataset.

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

Table 5.2: Description of the Breman 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 5.1.2

```
# Load the Breman weather dataset
bremen <- read.csv('bremen.csv',
                   colClasses = c('character',
                                  'numeric',
                                  'numeric',
                                  'numeric',
                                  'factor',
                                  'numeric',
                                  'numeric',
                                  'numeric',
                                  'numeric')) %>% as_tibble()
# View the dataset
bremen
```

```
## # A tibble: 79,669 x 9
##                datetime airtemp   sun windspd winddir soil10 soil20 soil50
##                   <chr>   <dbl> <dbl>   <dbl>  <fctr>  <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     3.0     250   -1.3   -1.1   -0.5
##  9 2009-01-01 09:00:00    -0.8     2     2.6     260   -1.1   -1.0   -0.5
## 10 2009-01-01 10:00:00     0.0    11     3.0     250   -0.9   -0.9   -0.5
## # ... with 79,659 more rows, and 1 more variables: soil100 <dbl>
```

Before moving on to further analysis, we need to aggregate the two datasets into the same granularity first. Once this is done, we can then join the two datasets together to form one table containing all variables. Example 5.1.3 shows how to use SQL-like pipeline in `dplyr` to aggregate and join the two datasets.

R Example 5.1.3

```
# 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')
# View the aggregated datasets
View(myTable)
# Plots the daily total demand
myTable %>%
  ggplot(aes(x=date, y=total_demand)) +
  geom_line() +
  labs(x = 'Date',
       y = 'Power Demand (MW)')
```

Exercise 17 Analysing Temporal Correlation

You can use the code in example 5.1.4 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 5.1.4

```r
# 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)
```

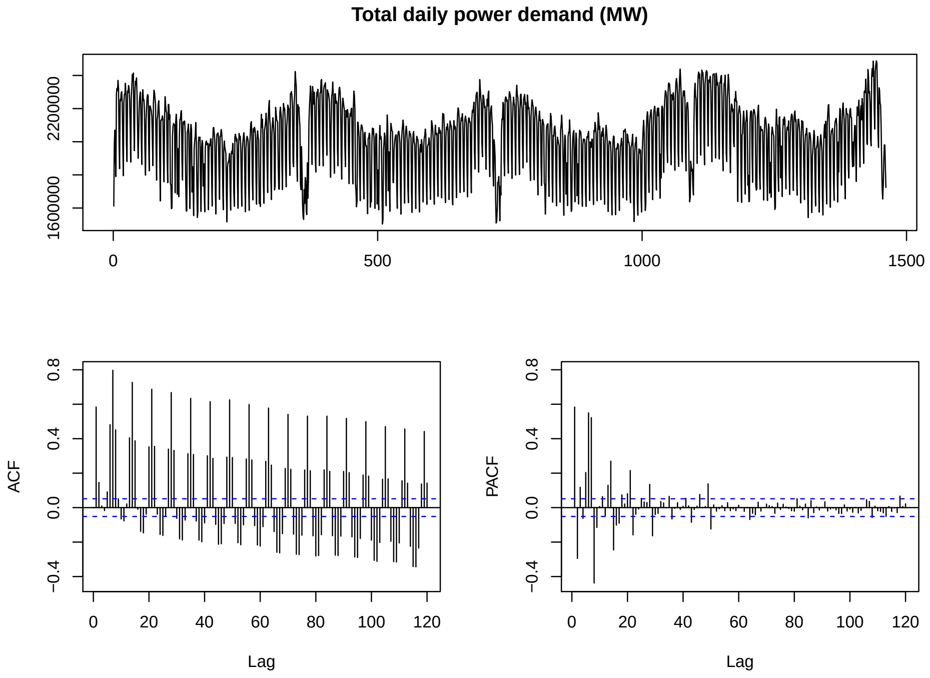


Figure 5.1: ACF and PACF correlograms

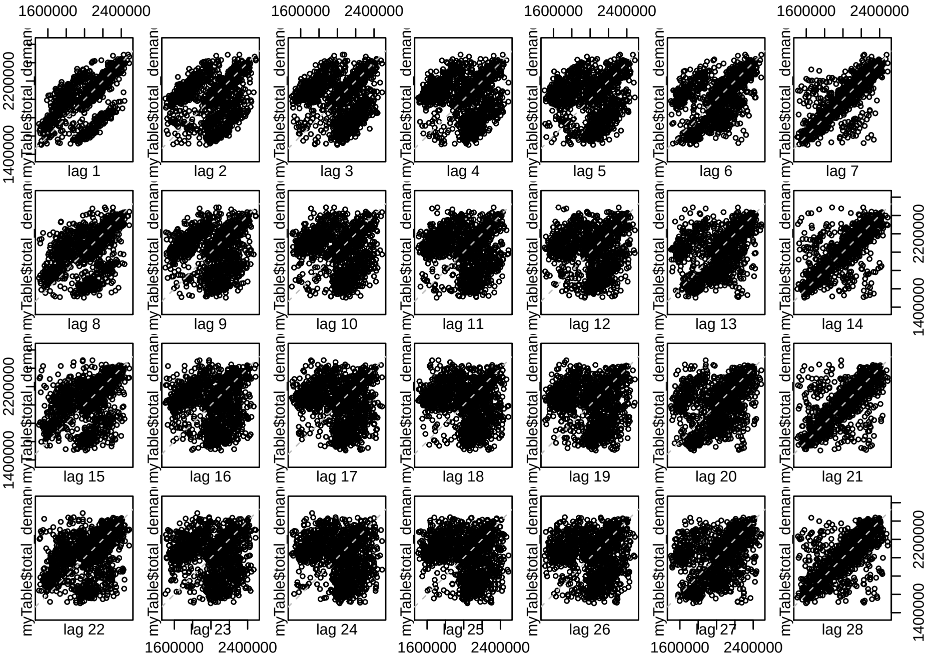


Figure 5.2: Lag plot showing the correlation of various lag periods

5.2 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. The observed data X_t is simply the sum total of trend, seasonal and error components. Alternative, 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 18 Identifying 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 5.2.1 uses the function `findfrequency()` to identify the frequency for a given time series. It uses spectal analysis to identify the frequency with 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 trend component $T_t \in [1, T]$, seasonal component $S_t \in [1, T]$ and random residuals $\epsilon_t \in [1, T]$.

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

R Example 5.2.1

```r
# 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)
```

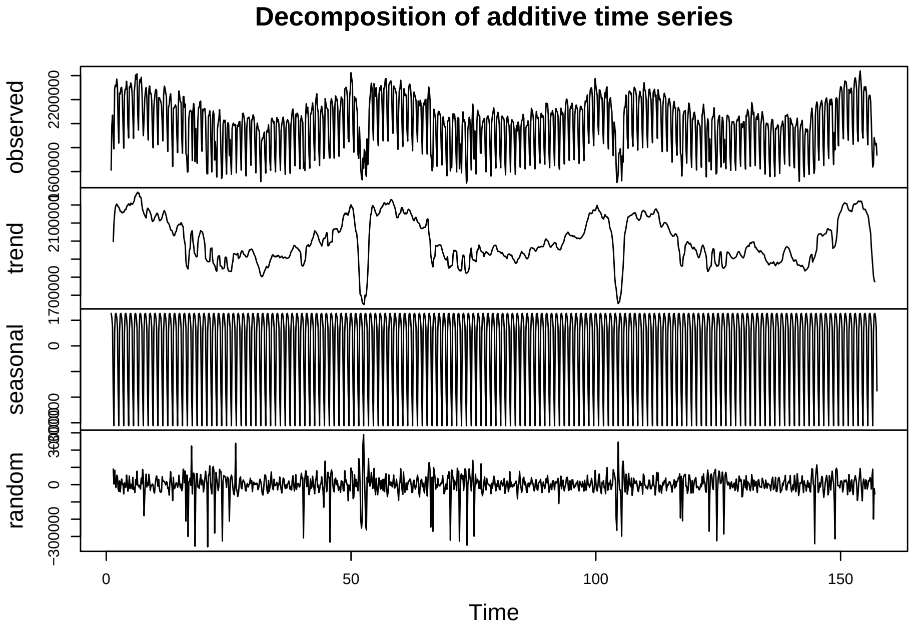


Figure 5.3: Decomposing an additive time series

Exercise 19 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 (5.1). Code in example 5.2.2 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.

$$X_t = \beta_0 + \beta_{trend}T_t + \beta_{seasonal}S_t + \sum_{m=1}^{M}(\beta_m x_{mt}) + \epsilon_t \tag{5.1}$$

R Example 5.2.2

```
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)
```

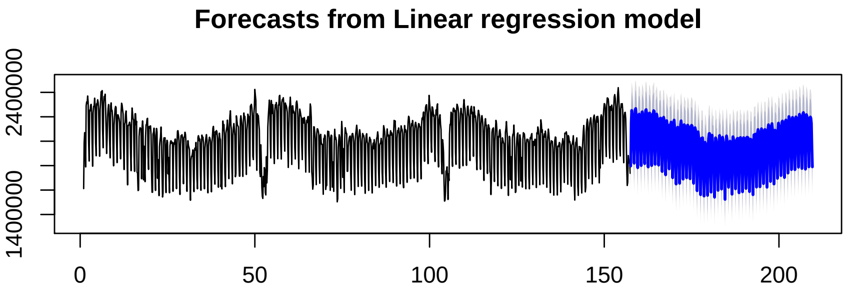


Figure 5.4: Linear time series forecasting with trend and seasonal components

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 5.2.3.

R Example 5.2.3

```
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')
```

5.3 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} represent observed data at i^{th} lag step and ϵ_{t-i} refers to the random error at the i^{th} lag step (5.2).

$$\underbrace{X_t}_{\text{Observation}} = \underbrace{\beta_0}_{\text{intercept}} + \underbrace{\sum_{i=i}^{p}(\phi_i X_{t-1})}_{\text{AR(p)}} + \underbrace{\sum_{i=1}^{q}(\theta_i\epsilon_{t-i})}_{\text{MA(q)}} + \underbrace{\epsilon_t}_{\text{residual}} \tag{5.2}$$

ARIMA model assumes the time series conforms 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 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 20 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 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 5.3.1 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 5.3.1

```r
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),
                         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))
# Visualise the forecast
autoplot(myTsForecast3)
plot(myTsForecast3)
# Calculate the MSE error using the testing set
myTestError3 <- mean((myTsForecast3$mean - myTestingSet$total_demand)^2)
```

Exercise 21 Custom ARIMA

After running the automated ARIMA forecast, you might realise the forecast tends to flat out when forecast horizon increases. This is because the `auto.arima()` function selects the best parameters based on an indicated 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 5.3.2.

R Example 5.3.2

```
# 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),
                    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))
# Visualise the forecast
autoplot(myTsForecast4)
plot(myTsForecast4)
# Calculate MSE error using the testing set
myTestError4 <- mean((myTsForecast4$mean - myTestingSet$total_demand)^2)
```

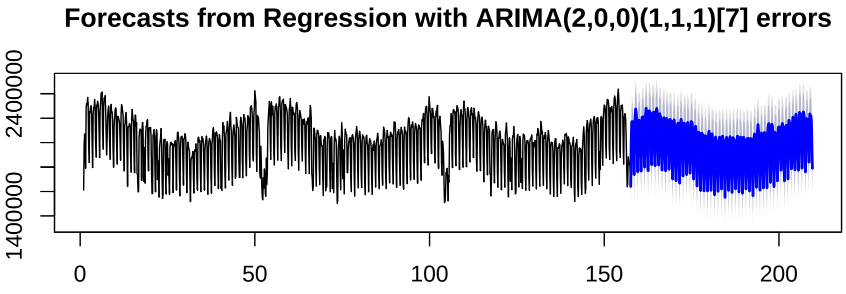


Figure 5.5: Forecast generated from a seasonal ARIMA model.

Exercise 22 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 5.3.3 will generate 100 simulated runs and plot the average of all runs as point forecast on a chart.

R Example 5.3.3

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

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

```r
# 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 6

Survival Analysis

Events occuring at irregular time interval 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.

6.1 Kaplan-Meier Estimator

Kaplan-Meier estimator is used to measure how many subjects survives in a clinical trial since treatment began. At time $t \leqslant T$, the estimator is given by equation (6.1) where $d_{t'}$ represents the number of events and $n_{t'}$ represents the number of subjects at risk.

$$\hat{S}_t = \prod_{t'=1}^{t} \left(1 - \frac{d_{t'}}{n_{t'}}\right) \tag{6.1}$$

Exercise 23 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. Code in example 6.1.1 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 6.1.1

```
# 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)
```

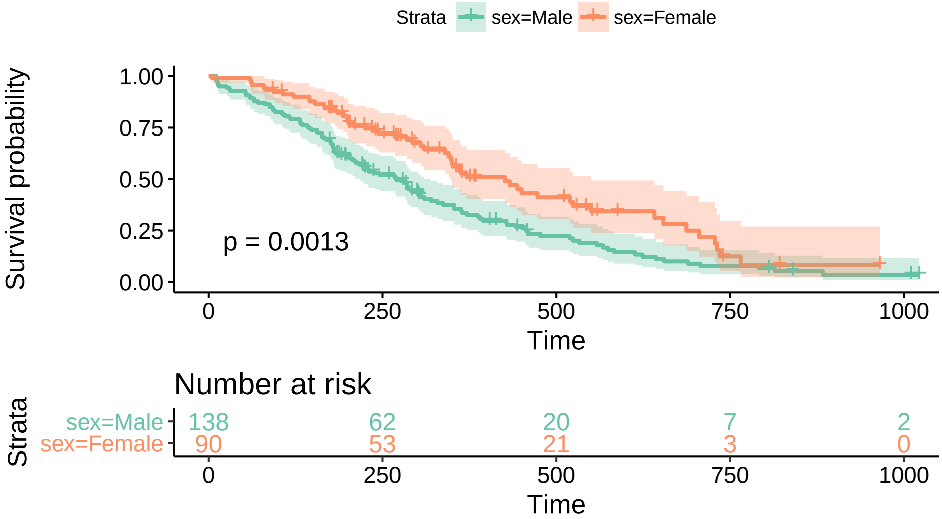


Figure 6.1: Kaplan-Meier curves showing two strata

6.2 Cox Proportional Harzard 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 survivial and assumes that the effects of these covariates are time-independent.

The harzard function h_t is give by equation (6.2). The term $h_{0,t}$ in the equation represents the baseline harzard 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 harzard ratio of the covariate variable x_m. If $e^{\beta_m} > 1$, the corresponding covariate is positively correlated with increase in hazard. On the other hand, x_m is negatively correlated with harzard if $e^{\beta_m} < 1$. In the case where $e^{\beta_m} = 1$, the variable x_m has no effects on harzard.

$$h_t = h_{0,t} \times e^{\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + ... + \beta_M x_M} \tag{6.2}$$

Cox model assumes variable effects are time-independent. To test whether the assumptions hold, we can analyse the Schoenfeld residuals for each covariate variable. Equation (6.3) 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 risks at time t. If there are observable temporal patterns in the residual plot, it suggests the proportional harzard assumptions may have been violated. In this case, you can consider adding interaction effects with time to mitigate the problem.

$$s_{i,k} = x_{i,k} - \sum_{i=1}^{j \in R(t)} x_{i,m} \hat{p_j} \tag{6.3}$$

Exercise 24 Training a Cox Regression Model

Cox regression model can be trained using the `coxph()` function in the `survival` package. Example 6.2.1 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 6.2.1

```
# 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
## factor(sex)2 -0.55085214535  0.57645837470  0.20083299516 -2.74284
## age           0.01064919149  1.01070609595  0.01161113439  0.91715
## ph.ecog       0.73417669164  2.08376570435  0.22327092554  3.28828
## ph.karno      0.02245506374  1.02270907641  0.01123988543  1.99780
## pat.karno    -0.01241655133  0.98766021598  0.00805415700 -1.54163
## meal.cal      0.00003329025  1.00003329081  0.00025946690  0.12830
## wt.loss      -0.01433061217  0.98577158230  0.00777132668 -1.84404
##               Pr(>|z|)
## factor(sex)2 0.0060911 **
## age          0.3590623
## ph.ecog      0.0010080 **
## ph.karno     0.0457381 *
## pat.karno    0.1231629
## meal.cal     0.8979096
## wt.loss      0.0651778 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##              exp(coef) exp(-coef) lower .95 upper .95
## factor(sex)2 0.5764584  1.7347306 0.3888827 0.8545103
## age          1.0107061  0.9894073 0.9879648 1.0339709
## ph.ecog      2.0837657  0.4799004 1.3452417 3.2277318
## ph.karno     1.0227091  0.9777952 1.0004254 1.0454891
## pat.karno    0.9876602  1.0124940 0.9721916 1.0033750
## meal.cal     1.0000333  0.9999667 0.9995249 1.0005420
## wt.loss      0.9857716  1.0144338 0.9708706 1.0009013
##
## Concordance= 0.651  (se = 0.031 )
## Rsquare= 0.155   (max possible= 0.998 )
## Likelihood ratio test= 28.33  on 7 df,   p=0.0001918421
## Wald test            = 27.58  on 7 df,   p=0.0002615887
## Score (logrank) test = 28.41  on 7 df,   p=0.0001848923
```

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

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

that having sex=2 would change the patient's hazard by $e^{-0.551} - 1 = -42.4\%$. In other words, sex=2 is benefitial to the patient's wellbeing.

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

Exercise 25 Cox Regression Diagnostics

Example 6.2.2 tests the Cox proportional hazard assumption by calculating the Schoenfeld residuals. If there are observable patterns along the temporal dimension, the model assumptions may have been violated.

R Example 6.2.2

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

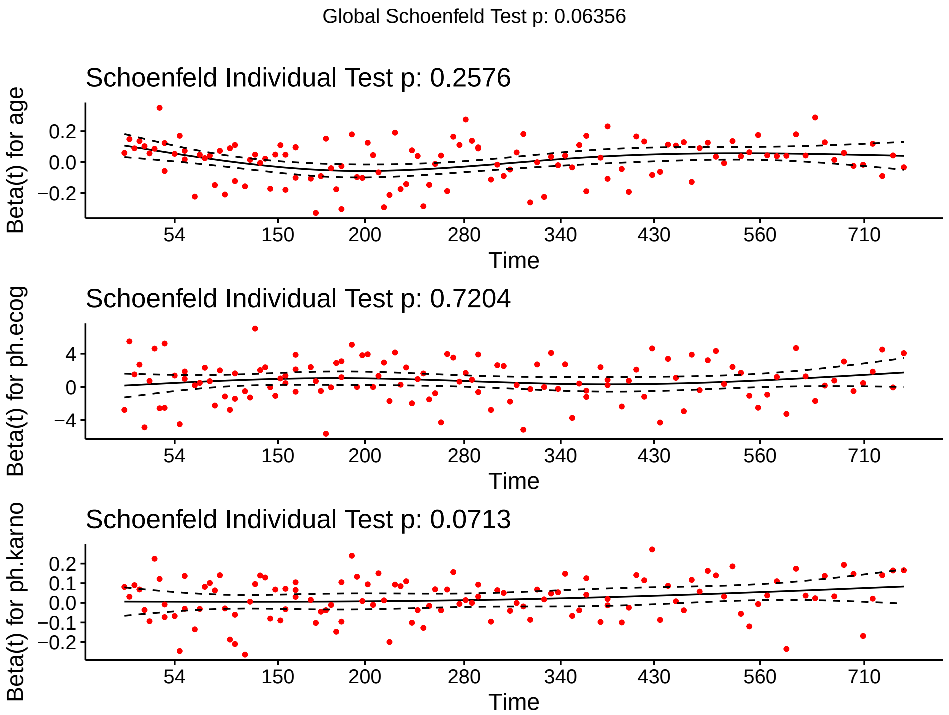


Figure 6.2: Scaled Schoenfeld residuals of a selected set of variables plotted against time

Chapter 7

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 7.1 below shows the effects of choosing differennt number of clusters.

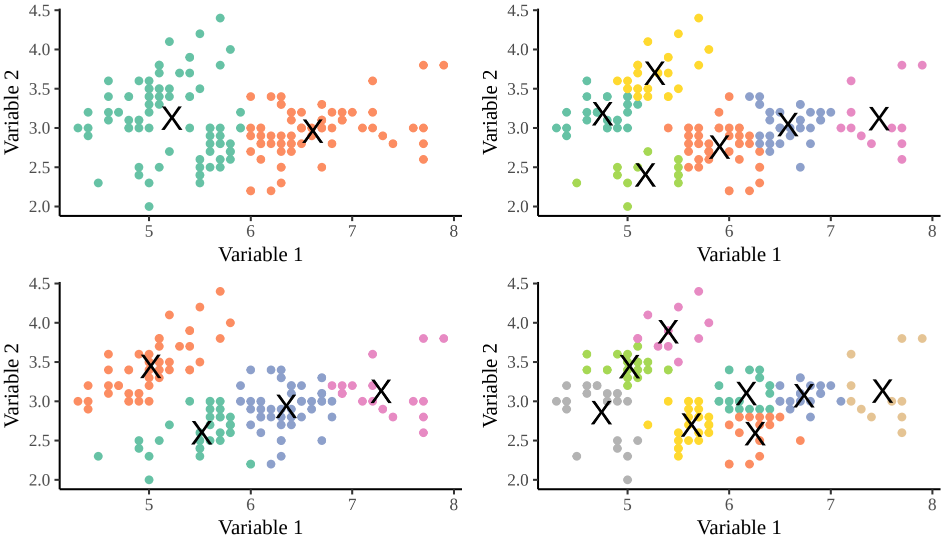


Figure 7.1: Different ways to cluster a set of unlabelled objects

7.1 K-means Clustering

K-means clustering 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 equation (7.1).

$$d(\vec{x_j}, \vec{x_k}) = \sqrt{\sum_{p=1}^{P}(x_{j,p} - x_{k,p})^2} \tag{7.1}$$

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
        $k \leftarrow \underset{k'}{\text{argmin}}\ d(\vec{\mu}_{k'}, \vec{x}_n), k \in 1, 2, 3, ..., K$ ;
        $\omega_k \leftarrow \omega_k \cup \{\vec{x}_n\}$ ;
    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 cluster 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 26 Dimensionality Reduction

In example 7.1.1, 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 K-means algorithm, it is very common to normalise the numeric variables using z-scores if they are recorded in different units. Normalisation would ensure 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 would execute PCA and visualise the top two PCs on a scatterplot.

R Example 7.1.1

```r
# 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)
```

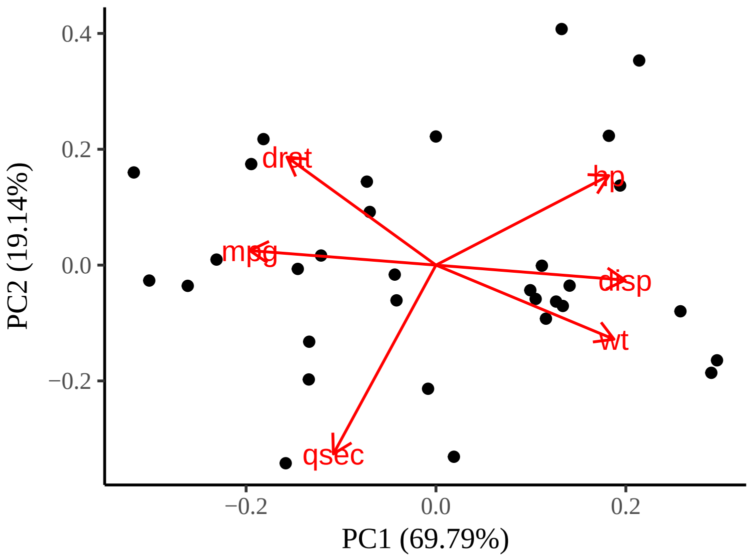


Figure 7.2: Biplot showing the first and second principal components

Exercise 27 K-means Clustering

With a $P = 6$ dimensional dataset, we can apply K-means algorithm on it to compute the clusters. Since the number of cluster K is unknown, we can experiement different values in this example 7.1.2. The clustering results can be visualised in a low-dimensional space with two principal components.

R Example 7.1.2

```r
# 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')
```

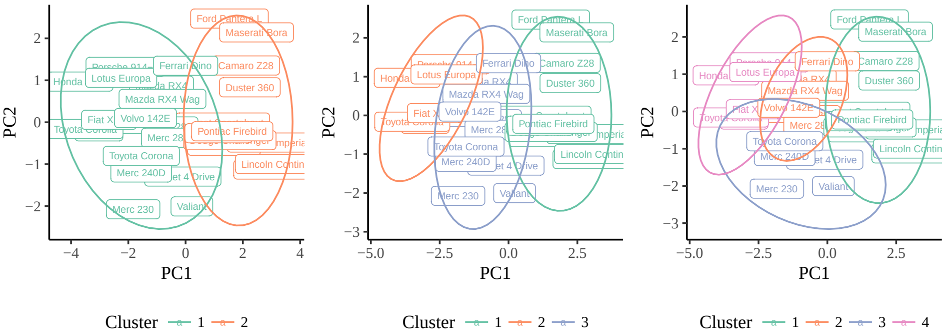


Figure 7.3: Comparing K-means clustering results using different K values

7.2 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 agglomerative hierarchical clustering algorithm works.

In hierarchical clustering, the closeness metric between two clusters ω_i and ω_j is denoted as function $D(\omega_i, \omega_j)$. There are several common choices of 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 (7.2a). 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 (7.2b). This tends to produce clusters with consistent size. The two aforementioned measurements are prone to outlier influence. To mitigate such problem, average linkage can be used. It uses the arithmetic average of all pairwise distance as the cluster closeness measurement (7.2c). Similarly, we can make use of cluster centroid to measure closeness (7.2d). The centroid method is also resilient to outlier influence. Nonetheless, Ward's method compares the change in sum of squares between cluster members and their centroid when they are merged (7.2e).

Single linkage

$$D(\omega_i, \omega_j) = \min_{\vec{x}_i \in \omega_i, \vec{x}_j \in \omega_j} d(\vec{x}_i, \vec{x}_j) \tag{7.2a}$$

Complete linkage

$$D(\omega_i, \omega_j) = \max_{\vec{x}_i \in \omega_i, \vec{x}_j \in \omega_j} d(\vec{x}_i, \vec{x}_j) \tag{7.2b}$$

Average linkage

$$D(\omega_i, \omega_j) = \underbrace{\frac{1}{|\omega_i|} \frac{1}{|\omega_j|} \sum_{\vec{x}_i \in \omega_i} \sum_{\vec{x}_j \in \omega_j} d(\vec{x}_i, \vec{x}_j)}_{\text{Average pairwise distance between } \omega_i \text{ and } \omega_j} \tag{7.2c}$$

Centroid

$$D(\omega_i, \omega_j) = d\Bigg(\underbrace{\Big(\frac{1}{|\omega_i|} \sum_{\vec{x}_i \in \omega_i} \vec{x}_i \Big)}_{\text{Centroid of } \omega_i}, \underbrace{\Big(\frac{1}{|\omega_j|} \sum_{\vec{x}_j \in \omega_j} \vec{x}_j \Big)}_{\text{Centroid of } \omega_j} \Bigg) \tag{7.2d}$$

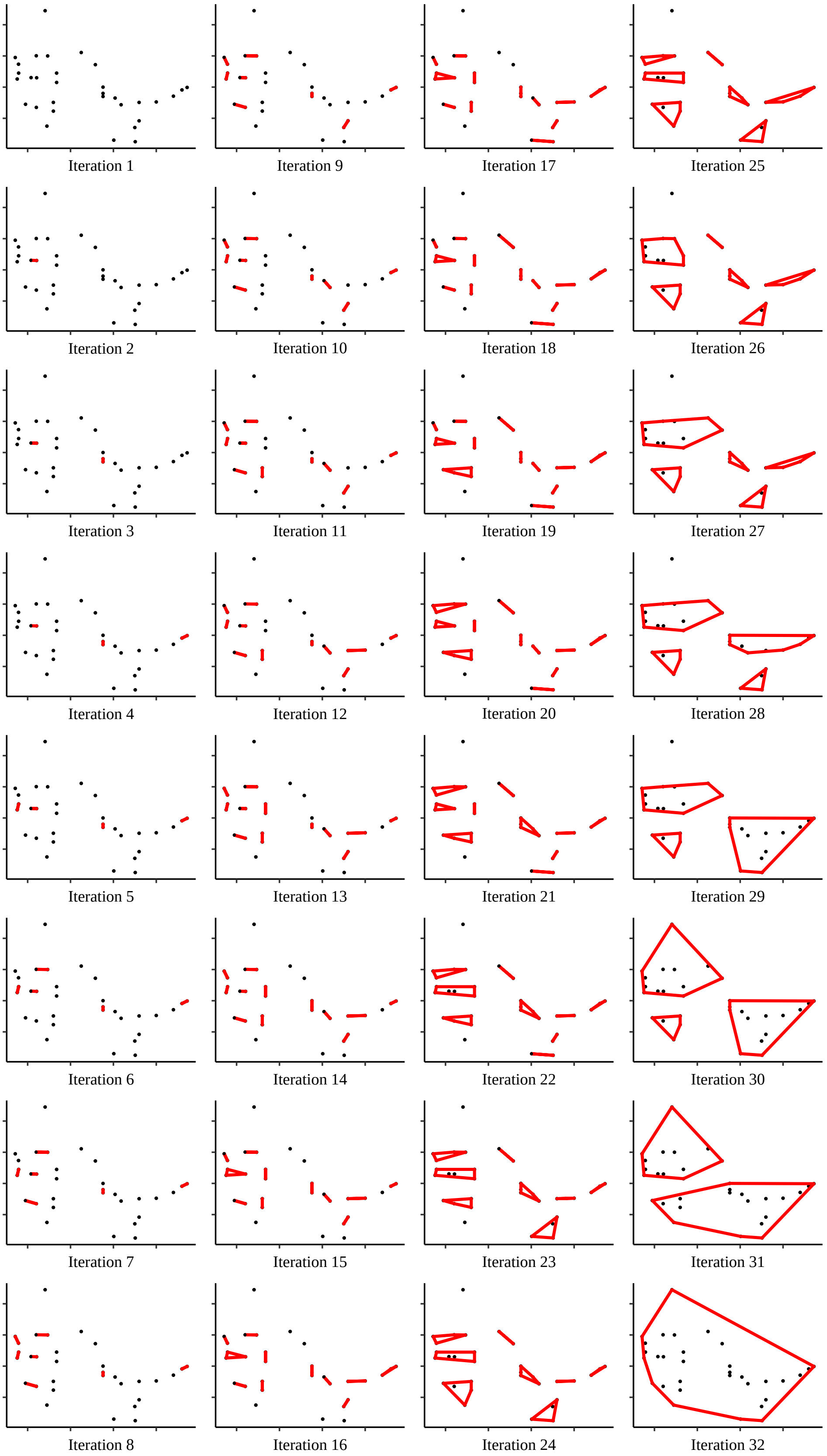
Ward's method

$$\begin{aligned} D(\omega_i, \omega_j) = & \underbrace{\sum_{k \in \omega_i \cup \omega_j} \Big(\vec{x}_k - \big(\frac{1}{|\omega_i \cup \omega_j|} \sum_{\vec{x}_{k'} \in \omega_i \cup \omega_j} \vec{x}_{k'} \big) \Big)^2}_{\text{Sum of squares of } \omega_i \cup \omega_j} - \\ & \underbrace{\sum_{i \in \omega_i} \Big(\vec{x}_i - \big(\frac{1}{|\omega_i|} \sum_{\vec{x}_{i'} \in \omega_i} \vec{x}_{i'} \big) \Big)^2}_{\text{Sum of squares of } \omega_i} - \\ & \underbrace{\sum_{j \in \omega_j} \Big(\vec{x}_j - \big(\frac{1}{|\omega_j|} \sum_{\vec{x}_{j'} \in \omega_j} \vec{x}_{j'} \big) \Big)^2}_{\text{Sum of squares of } \omega_j} \end{aligned} \tag{7.2e}$$

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
    $\{i, j\} \leftarrow \underset{i,j}{\text{argmin}}\ \Omega^{'}$ ;
    $\omega_{ij} \leftarrow \Omega^{'}_i \cup \Omega^{'}_j$ ;
    $\Omega \leftarrow \Omega^{'} \setminus \Omega^{'}_i \setminus \Omega^{'}_j \cup \omega_{ij}$ ;
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 objective 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.



Iteration 1 Iteration 9 Iteration 17 Iteration 25

Iteration 2 Iteration 10 Iteration 18 Iteration 26

Iteration 3 Iteration 11 Iteration 19 Iteration 27

Iteration 4 Iteration 12 Iteration 20 Iteration 28

Iteration 5 Iteration 13 Iteration 21 Iteration 29

Iteration 6 Iteration 14 Iteration 22 Iteration 30

Iteration 7 Iteration 15 Iteration 23 Iteration 31

Iteration 8 Iteration 16 Iteration 24 Iteration 32

Figure 7.4: Iterative steps of agglomerative hierarchical clustering

Exercise 28 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 7.2.1 performs hierarchical clustering and visualises the result as a simple dendrogram.

R Example 7.2.1

```r
# 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)
```

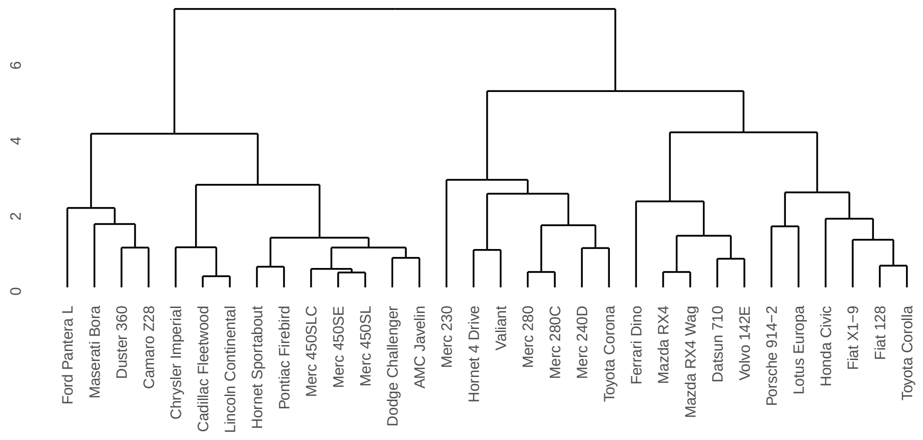


Figure 7.5: Dendrogram illustrating hierarchical clustering using complete linkage

Exercise 29 Cutting a Dendrogram

Dendrogram can be cut to remove smaller clusters at lower height. This can be achieved using the `cutree()` function in R. You can either specify the number of clusters to retain using parameter `k`, this would 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 7.2.2 shows how to cut a dendrogram. It also demonstrates various ways to visualise a dendrogram.

R Example 7.2.2

```r
# 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])
```

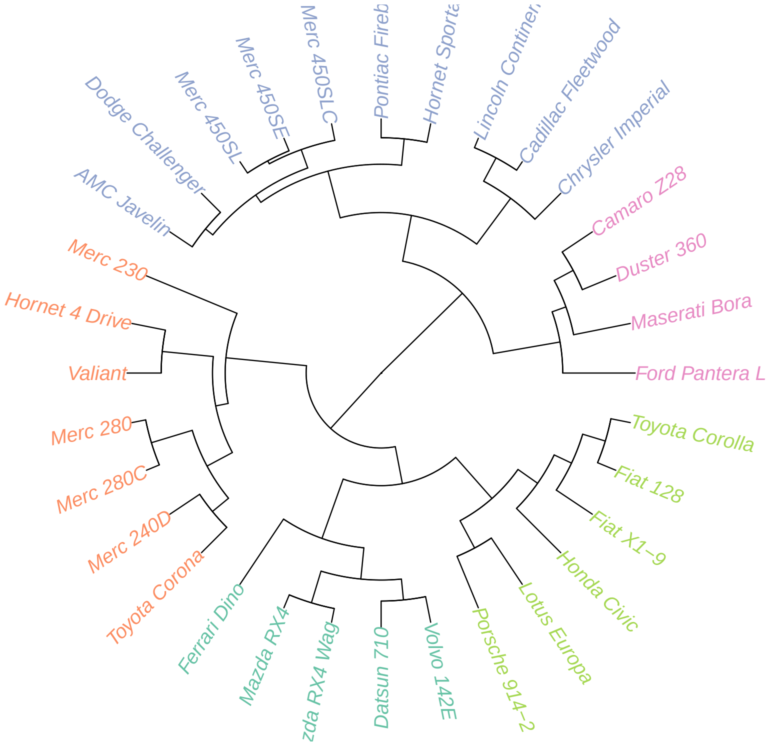


Figure 7.6: Fan phylogram showing hierarchical clusters