

# Lightweight Machine Learning Classics with R

Marek Gagolewski

DRAFT v0.2 2020-04-28 21:58 (378d7a3)



# Contents

{	11
<b>1 Simple Linear Regression</b>	<b>13</b>
1.1 Machine Learning . . . . .	13
1.1.1 What is Machine Learning? . . . . .	13
1.1.2 Main Types of Machine Learning Problems . . . . .	13
1.2 Supervised Learning . . . . .	14
1.2.1 Formalism . . . . .	14
1.2.2 Desired Outputs . . . . .	16
1.2.3 Types of Supervised Learning Problems . . . . .	17
1.3 Simple Regression . . . . .	20
1.3.1 Introduction . . . . .	20
1.3.2 Search Space and Objective . . . . .	23
1.4 Simple Linear Regression . . . . .	26
1.4.1 Introduction . . . . .	26
1.4.2 Solution in R . . . . .	26
1.4.3 Analytic Solution . . . . .	29
1.4.4 Derivation of the Solution (**). . . . .	30
1.5 Exercises in R . . . . .	32
1.5.1 The Anscombe Quartet . . . . .	32
1.6 Outro . . . . .	38
1.6.1 Remarks . . . . .	38
1.6.2 Further Reading . . . . .	38
<b>2 Multiple Regression</b>	<b>39</b>
2.1 Introduction . . . . .	39
2.1.1 Formalism . . . . .	39
2.1.2 Simple Linear Regression - Recap . . . . .	40
2.2 Multiple Linear Regression . . . . .	41
2.2.1 Problem Formulation . . . . .	41
2.2.2 Fitting a Linear Model in R . . . . .	41
2.3 Finding the Best Model . . . . .	43
2.3.1 Model Diagnostics . . . . .	43
2.3.2 Variable Selection . . . . .	53

2.3.3	Variable Transformation . . . . .	61
2.3.4	Predictive vs. Descriptive Power . . . . .	63
2.4	Exercises in R . . . . .	66
2.4.1	Ansccombe's Quartet Revisited . . . . .	66
2.4.2	Countries of the World – Simple models involving the GDP per capita . . . . .	68
2.4.3	Countries of the World – Most correlated variables (*) . . . . .	74
2.4.4	Countries of the World – A non-linear model based on the GDP per capita . . . . .	79
2.4.5	Countries of the World – A multiple regression model for the per capita GDP . . . . .	85
2.5	Outro . . . . .	89
2.5.1	Remarks . . . . .	89
2.5.2	Other Methods for Regression . . . . .	89
2.5.3	Derivation of the Solution (**) . . . . .	90
2.5.4	Solution in Matrix Form (***) . . . . .	91
2.5.5	Pearson's r in Matrix Form (**) . . . . .	93
2.5.6	Further Reading . . . . .	94
<b>3</b>	<b>Classification with K-Nearest Neighbours</b>	<b>95</b>
3.1	Introduction . . . . .	95
3.1.1	Classification Task . . . . .	95
3.1.2	Data . . . . .	96
3.1.3	Training and Test Sets . . . . .	97
3.1.4	Discussed Methods . . . . .	98
3.2	K-nearest Neighbour Classifier . . . . .	99
3.2.1	Introduction . . . . .	99
3.2.2	Example in R . . . . .	100
3.2.3	Feature Engineering . . . . .	101
3.3	Model Assessment and Selection . . . . .	104
3.3.1	Performance Metrics . . . . .	104
3.3.2	How to Choose K for K-NN Classification? . . . . .	107
3.3.3	Training, Validation and Test sets . . . . .	108
3.4	Implementing a K-NN Classifier (*) . . . . .	110
3.4.1	Factor Data Type . . . . .	110
3.4.2	Main Routine (*) . . . . .	110
3.4.3	Mode . . . . .	112
3.4.4	NN Search Routines (*) . . . . .	113
3.4.5	Different Metrics (*) . . . . .	116
3.5	Outro . . . . .	117
3.5.1	Remarks . . . . .	117
3.5.2	Side Note: K-NN Regression . . . . .	117
3.5.3	Further Reading . . . . .	118
<b>4</b>	<b>Classification with Trees and Linear Models</b>	<b>119</b>
4.1	Introduction . . . . .	119

4.1.1	Classification Task . . . . .	119
4.1.2	Data . . . . .	120
4.2	Decision Trees . . . . .	123
4.2.1	Introduction . . . . .	123
4.2.2	Example in R . . . . .	124
4.2.3	A Note on Decision Tree Learning . . . . .	128
4.3	Binary Logistic Regression . . . . .	128
4.3.1	Motivation . . . . .	128
4.3.2	Logistic Model . . . . .	129
4.3.3	Example in R . . . . .	130
4.3.4	Loss Function . . . . .	133
4.4	Outro . . . . .	134
4.4.1	Remarks . . . . .	134
4.4.2	Further Reading . . . . .	134
<b>5</b>	<b>Shallow and Deep Neural Networks</b>	<b>135</b>
5.1	Introduction . . . . .	135
5.1.1	Binary Logistic Regression: Recap . . . . .	135
5.1.2	Data . . . . .	136
5.2	Multinomial Logistic Regression . . . . .	138
5.2.1	A Note on Data Representation . . . . .	138
5.2.2	Extending Logistic Regression . . . . .	139
5.2.3	Softmax Function . . . . .	140
5.2.4	One-Hot Encoding and Decoding . . . . .	141
5.2.5	Cross-entropy Revisited . . . . .	142
5.2.6	Problem Formulation in Matrix Form (**). . . . .	144
5.3	Artificial Neural Networks . . . . .	145
5.3.1	Artificial Neuron . . . . .	145
5.3.2	Logistic Regression as a Neural Network . . . . .	147
5.3.3	Example in R . . . . .	147
5.4	Deep Neural Networks . . . . .	151
5.4.1	Introduction . . . . .	151
5.4.2	Activation Functions . . . . .	151
5.4.3	Example in R - 2 Layers . . . . .	152
5.4.4	Example in R - 6 Layers . . . . .	153
5.5	Preprocessing of Data . . . . .	155
5.5.1	Introduction . . . . .	155
5.5.2	Image Deskewing . . . . .	155
5.5.3	Summary of All the Models Considered . . . . .	157
5.6	Outro . . . . .	159
5.6.1	Remarks . . . . .	159
5.6.2	Beyond MNIST . . . . .	160
5.6.3	Further Reading . . . . .	160
<b>6</b>	<b>Continuous Optimisation with Iterative Algorithms</b>	<b>161</b>
6.1	Introduction . . . . .	161

6.1.1	Optimisation Problems . . . . .	161
6.1.2	Example Optimisation Problems in Machine Learning . . . . .	162
6.1.3	Types of Minima and Maxima . . . . .	162
6.1.4	Example Objective over a 2D Domain . . . . .	166
6.2	Iterative Methods . . . . .	168
6.2.1	Introduction . . . . .	168
6.2.2	Example in R . . . . .	168
6.2.3	Convergence to Local Optima . . . . .	170
6.2.4	Random Restarts . . . . .	171
6.3	Gradient Descent . . . . .	173
6.3.1	Function Gradient (*) . . . . .	173
6.3.2	Three Facts on the Gradient . . . . .	174
6.3.3	Gradient Descent Algorithm (GD) . . . . .	175
6.3.4	Example: MNIST (*) . . . . .	179
6.3.5	Stochastic Gradient Descent (SGD) (*) . . . . .	183
6.4	Outro . . . . .	186
6.4.1	Remarks . . . . .	186
6.4.2	Further Reading . . . . .	187
<b>7</b>	<b>Clustering</b>	<b>189</b>
7.1	Unsupervised Learning . . . . .	189
7.1.1	Introduction . . . . .	189
7.1.2	Main Types of Unsupervised Learning Problems . . . . .	190
7.1.3	Definitions . . . . .	191
7.2	K-means Clustering . . . . .	193
7.2.1	Example in R . . . . .	193
7.2.2	Problem Statement . . . . .	195
7.2.3	Algorithms for the K-means Problem . . . . .	198
7.3	Agglomerative Hierarchical Clustering . . . . .	201
7.3.1	Introduction . . . . .	201
7.3.2	Example in R . . . . .	202
7.3.3	Linkage Functions . . . . .	205
7.3.4	Cluster Dendograms . . . . .	206
7.4	Outro . . . . .	209
7.4.1	Remarks . . . . .	209
7.4.2	Further Reading . . . . .	210
<b>8</b>	<b>Optimisation with Genetic Algorithms</b>	<b>211</b>
8.1	Introduction . . . . .	211
8.1.1	Recap . . . . .	211
8.1.2	K-means Revisited . . . . .	212
8.1.3	optim() vs. kmeans() . . . . .	213
8.2	A Note on Convex Optimisation (*) . . . . .	217
8.2.1	Introduction . . . . .	217
8.2.2	Convex Combinations (*) . . . . .	217
8.2.3	Convex Functions (*) . . . . .	219

8.2.4 Examples . . . . .	220
8.3 Genetic Algorithms . . . . .	221
8.3.1 Introduction . . . . .	221
8.3.2 Overview of the Method . . . . .	222
8.3.3 Example Implementation - GA for K-means . . . . .	222
8.4 Outro . . . . .	226
8.4.1 Remarks . . . . .	226
8.4.2 Further Reading . . . . .	227
<b>9 Recommender Systems</b>	<b>229</b>
9.1 Introduction . . . . .	229
9.1.1 What is a Recommender System? . . . . .	229
9.1.2 The Netflix Prize . . . . .	229
9.1.3 Main Approaches . . . . .	230
9.1.4 Formalism . . . . .	231
9.2 Collaborative Filtering . . . . .	231
9.2.1 Example . . . . .	231
9.2.2 Similarity Measures . . . . .	233
9.2.3 User-Based Collaborative Filtering . . . . .	233
9.2.4 Item-Based Collaborative Filtering . . . . .	234
9.3 MovieLens Dataset (*) . . . . .	236
9.3.1 Dataset . . . . .	236
9.3.2 Data Cleansing . . . . .	237
9.3.3 Item-Item Similarities . . . . .	238
9.3.4 Example Recommendations . . . . .	238
9.3.5 Clustering . . . . .	239
9.4 Outro . . . . .	241
9.4.1 Remarks . . . . .	241
9.4.2 Issues . . . . .	241
9.4.3 Further Reading . . . . .	242
}	243
<b>A Setting Up the R Environment</b>	<b>247</b>
A.1 Installing R . . . . .	247
A.2 Installing an IDE . . . . .	247
A.3 Installing Recommended Packages . . . . .	248
A.4 First R Script in RStudio . . . . .	248
<b>B Vector Algebra in R</b>	<b>251</b>
B.1 Motivation . . . . .	251
B.2 Numeric Vectors . . . . .	253
B.2.1 Creating Numeric Vectors . . . . .	253
B.2.2 Vector-Scalar Operations . . . . .	256
B.2.3 Vector-Vector Operations . . . . .	257
B.2.4 Aggregation Functions . . . . .	258

B.2.5	Special Functions . . . . .	259
B.2.6	Norms and Distances . . . . .	260
B.2.7	Dot Product (*) . . . . .	261
B.2.8	Missing and Other Special Values . . . . .	263
B.3	Logical Vectors . . . . .	264
B.3.1	Creating Logical Vectors . . . . .	264
B.3.2	Logical Operations . . . . .	265
B.3.3	Comparison Operations . . . . .	265
B.3.4	Aggregation Functions . . . . .	266
B.4	Character Vectors . . . . .	267
B.4.1	Creating Character Vectors . . . . .	267
B.4.2	Concatenating Character Vectors . . . . .	267
B.4.3	Collapsing Character Vectors . . . . .	268
B.5	Vector Subsetting . . . . .	268
B.5.1	Subsetting with Positive Indices . . . . .	268
B.5.2	Subsetting with Negative Indices . . . . .	269
B.5.3	Subsetting with Logical Vectors . . . . .	269
B.5.4	Replacing Elements . . . . .	270
B.5.5	Other Functions . . . . .	270
B.6	Named Vectors . . . . .	271
B.6.1	Creating Named Vectors . . . . .	271
B.6.2	Subsetting Named Vectors with Character String Indices . . . . .	272
B.7	Factors . . . . .	273
B.7.1	Creating Factors . . . . .	273
B.7.2	Levels . . . . .	273
B.7.3	Internal Representation (*) . . . . .	274
B.8	Lists . . . . .	276
B.8.1	Creating Lists . . . . .	276
B.8.2	Named Lists . . . . .	277
B.8.3	Subsetting and Extracting From Lists . . . . .	278
B.8.4	Common Operations . . . . .	278
B.9	Further Reading . . . . .	280
<b>C</b>	<b>Matrix Algebra in R</b>	<b>281</b>
C.1	Creating Matrices . . . . .	282
C.1.1	<code>matrix()</code> . . . . .	282
C.1.2	Stacking Vectors . . . . .	282
C.1.3	Beyond Numeric Matrices . . . . .	283
C.1.4	Naming Rows and Columns . . . . .	283
C.1.5	Other Methods . . . . .	284
C.1.6	Internal Representation (*) . . . . .	286
C.2	Common Operations . . . . .	287
C.2.1	Matrix Transpose . . . . .	287
C.2.2	Matrix-Scalar Operations . . . . .	287
C.2.3	Matrix-Matrix Operations . . . . .	288
C.2.4	Matrix Multiplication (*) . . . . .	288

C.2.5	Aggregation of Rows and Columns . . . . .	290
C.2.6	Vectorised Special Functions . . . . .	291
C.2.7	Matrix-Vector Operations . . . . .	292
C.3	Matrix Subsetting . . . . .	293
C.3.1	Selecting Individual Elements . . . . .	293
C.3.2	Selecting Rows and Columns . . . . .	293
C.3.3	Selecting Submatrices . . . . .	295
C.3.4	Selecting Based on Logical Vectors and Matrices . . . . .	295
C.3.5	Selecting Based on Two-Column Matrices . . . . .	296
C.4	Further Reading . . . . .	296
<b>D</b>	<b>Data Frame Wrangling in R</b>	<b>297</b>
D.1	Creating Data Frames . . . . .	298
D.2	Importing Data Frames . . . . .	299
D.3	Data Frame Subsetting . . . . .	300
D.3.1	Each Data Frame is a List . . . . .	300
D.3.2	Each Data Frame is Matrix-like . . . . .	301
D.4	Common Operations . . . . .	303
D.5	Metaprogramming and Formulas (*) . . . . .	305
D.6	Further Reading . . . . .	308
<b>R</b>	<b>References</b>	<b>309</b>



{

This is a draft version (distributed in the hope that it will be useful) of the book *Lightweight Machine Learning Classics with R* by Marek Gagolewski.

Please submit any feature requests, remarks and bug fixes via the project site at [github](#) or by email. Thanks!

Copyright (C) 2020, Marek Gagolewski. This material is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

You can access this book at:

- <https://lmlcr.gagolewski.com/> (a browser-friendly version)
- <https://lmlcr.gagolewski.com/lmlcr.pdf> (PDF)
- <https://github.com/gagolews/lmlcr> (source code)

## Aims and Scope

Machine learning has numerous exciting real-world applications, including stock market prediction, speech recognition, computer-aided medical diagnosis, content and product recommendation, anomaly detection in security camera footages, game playing, autonomous vehicle operation and many others.

In this book we will take an unpretentious glance at the most fundamental algorithms that have stood the test of time and which form the basis for state-of-the-art solutions of modern AI, which is principally (big) data-driven. We will learn how to use the R language (R Development Core Team 2020) for implementing various stages of data processing and modelling activities. For a more in-depth treatment of R, refer to this book's Appendices and, for instance, (Wickham & Grolemund 2017, Peng 2019, Venables et al. 2020).

These pages contain solid underpinnings for further studies related to statistical learning, machine learning data science, data analytics and artificial intelligence, including (Bishop 2006, Hastie et al. 2017, James et al. 2017). We will also appreciate the vital role of mathematics as a universal language for formalising data-intense problems and communicating their solutions. The book is aimed

at readers who are yet to be fluent with university-level linear algebra, calculus and probability theory, such as 1st year undergrads or those who have forgotten all the maths they have learned and need a gentle, non-invasive, yet rigorous introduction to the topic. For a nice, machine learning-focused introduction to mathematics alone, see, e.g., (Deisenroth et al. 2020).

## About Me

I'm currently a Senior Lecturer (equivalent to Associate Professor in the US) in Applied AI at Deakin University in Melbourne, VIC, Australia and an Associate Professor in Data Science at Warsaw University of Technology, Poland. My main passion is in research – my primary interests include machine learning and optimisation algorithms, data aggregation and clustering, statistical modelling and scientific computing. *Explaining* of things matters to me more than merely tuning the knobs so as to increase some performance metric (with consequences to many other ones); the latter belongs to technology and wizardry, not science.

I'm an author of more than 70 publications. I've developed several open source R and Python packages, including `stringi`, which is among the most often downloaded R extensions.

On top of that, I teach various courses related to R and Python programming, algorithms, data science and machine learning – and I'm good at it. This book was also influenced by my teaching experience at Data Science Retreat in Berlin, Germany.

## Acknowledgements

This book has been prepared with pandoc, Markdown and GitBook. R code chunks have been processed with knitr. A little help of bookdown, good ol' Makefiles and shell scripts did the trick.

The following R packages are used or referred to in the text: bookdown, Cairo, fastcluster, FNN, genie, ISLR, keras, knitr, Matrix, microbenchmark, pdist, RColorBrewer, recommenderlab, rpart, rpart.plot, scatterplot3d, stringi, tensorflow, titanic, vioplot.

During the writing of this book, I've been listening to the music featuring John Coltrane, Krzysztof Komeda, Henry Threadgill, Albert Ayler, Paco de Lucia and Tomatito.

# Chapter 1

## Simple Linear Regression

### 1.1 Machine Learning

#### 1.1.1 What is Machine Learning?

An **algorithm** is a well-defined sequence of instructions that, for a given sequence of input arguments, yields some desired output.

In other words, it is a specific recipe for a **function**.

Developing algorithms is a tedious task.

In **machine learning**, we build and study computer algorithms that make *predictions* or *decisions* but which are not manually programmed.

**Learning** needs some material based upon which new knowledge is to be acquired.

In other words, we need **data**.

#### 1.1.2 Main Types of Machine Learning Problems

Machine Learning Problems include, but are not limited to:

- **Supervised learning** – for every input point (e.g., a photo) there is an associated desired output (e.g., whether it depicts a crosswalk or how many cars can be seen on it)
- **Unsupervised learning** – inputs are unlabelled, the aim is to discover the underlying structure in the data (e.g., automatically group customers w.r.t. common behavioural patterns)
- **Semi-supervised learning** – some inputs are labelled, the others are not (definitely a cheaper scenario)

- **Reinforcement learning** – learn to act based on a feedback given after the actual decision was made (e.g., learn to play The Witcher 7 by testing different hypotheses what to do to survive as long as possible)

## 1.2 Supervised Learning

### 1.2.1 Formalism

Let  $\mathbf{X} = \{\mathfrak{X}_1, \dots, \mathfrak{X}_n\}$  be an input sample (“a database”) that consists of  $n$  objects.

Most often we assume that each object  $\mathfrak{X}_i$  is represented using  $p$  numbers for some  $p$ .

We denote this fact as  $\mathfrak{X}_i \in \mathbb{R}^p$  (it is a  *$p$ -dimensional real vector* or a *sequence of  $p$  numbers* or a *point in a  $p$ -dimensional real space* or an *element of a real  $p$ -space* etc.).

If we have “complex” objects on input, we can always try representing them as **feature vectors** (e.g., come up with numeric attributes that best describe them in a task at hand).

#### Exercise.

Consider the following problems:

1. How would you represent a patient in a clinic?
2. How would you represent a car in an insurance company’s database?
3. How would you represent a student in an university?

□

Of course, our setting is *abstract* in the sense that there might be different realities *hidden* behind these symbols.

This is what maths is for – creating *abstractions* or *models* of complex entities/phenomena so that they can be much more easily manipulated or understood. This is very powerful – spend a moment contemplating how many real-world situations fit into our framework.

This also includes image/video data, e.g., a  $1920 \times 1080$  pixel image can be “unwound” to a “flat” vector of length 2,073,600.

(\*) There are some algorithms such as Multidimensional Scaling, Locally Linear Embedding, IsoMap etc. that can do that automagically.

In cases such as this we say that we deal with *structured (tabular) data*

- $\mathbf{X}$  can be written as an  $(n \times p)$ -matrix:

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix}$$

Mathematically, we denote this as  $\mathbf{X} \in \mathbb{R}^{n \times d}$ .

**Remark.** Structured data == think: Excel/Calc spreadsheets, SQL tables etc.

For an example, consider the famous Fisher's Iris flower dataset, see `?iris` in R and [https://en.wikipedia.org/wiki/Iris\\_flower\\_data\\_set](https://en.wikipedia.org/wiki/Iris_flower_data_set).

```
X <- iris[1:6, 1:4] # first 6 rows and 4 columns
X      # or: print(X)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1      5.1       3.5      1.4       0.2
## 2      4.9       3.0      1.4       0.2
## 3      4.7       3.2      1.3       0.2
## 4      4.6       3.1      1.5       0.2
## 5      5.0       3.6      1.4       0.2
## 6      5.4       3.9      1.7       0.4
```

```
dim(X)    # gives n and p
```

```
## [1] 6 4
```

```
dim(iris) # for the full dataset
```

```
## [1] 150 5
```

$x_{i,j} \in \mathbb{R}$  represents the  $j$ -th feature of the  $i$ -th observation,  $j = 1, \dots, p$ ,  $i = 1, \dots, n$ .

For instance:

```
X[3, 2] # 3rd row, 2nd column
```

```
## [1] 3.2
```

The third observation (data point, row in  $\mathbf{X}$ ) consists of items  $(x_{3,1}, \dots, x_{3,p})$  that can be extracted by calling:

```
X[3,]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 3      4.7       3.2      1.3       0.2
```

```
as.numeric(X[3,]) # drops names
```

```
## [1] 4.7 3.2 1.3 0.2
```

```
length(X[3,])
```

```
## [1] 4
```

Moreover, the second feature/variable/column is comprised of  $(x_{1,2}, x_{2,2}, \dots, x_{n,2})$ :

```
X[,2]
```

```
## [1] 3.5 3.0 3.2 3.1 3.6 3.9
```

```
length(X[,2])
```

```
## [1] 6
```

We will sometimes use the following notation to emphasise that the  $\mathbf{X}$  matrix consists of  $n$  rows or  $p$  columns:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_{1,\cdot} \\ \mathbf{x}_{2,\cdot} \\ \vdots \\ \mathbf{x}_{n,\cdot} \end{bmatrix} = [\mathbf{x}_{\cdot,1} \quad \mathbf{x}_{\cdot,2} \quad \cdots \quad \mathbf{x}_{\cdot,p}] .$$

Here,  $\mathbf{x}_{i,\cdot}$  is a *row vector* of length  $p$ , i.e., a  $(1 \times p)$ -matrix:

$$\mathbf{x}_{i,\cdot} = [x_{i,1} \quad x_{i,2} \quad \cdots \quad x_{i,p}] .$$

Moreover,  $\mathbf{x}_{\cdot,j}$  is a *column vector* of length  $n$ , i.e., an  $(n \times 1)$ -matrix:

$$\mathbf{x}_{\cdot,j} = [x_{1,j} \quad x_{2,j} \quad \cdots \quad x_{n,j}]^T = \begin{bmatrix} x_{1,j} \\ x_{2,j} \\ \vdots \\ x_{n,j} \end{bmatrix} ,$$

where  $.^T$  denotes the *transpose* of a given matrix – think of this as a kind of rotation; it allows us to introduce a set of “vertically stacked” objects using a single inline formula.

### 1.2.2 Desired Outputs

In supervised learning, apart from the inputs we are also given the corresponding reference/desired outputs.

The aim of supervised learning is to try to create an “algorithm” that, given an input point, generates an output that is as *close* as possible to the desired one. The given data sample will be used to “train” this “model”.

Usually the reference outputs are encoded as individual numbers (scalars) or textual labels.

In other words, with each input  $\mathbf{x}_{i,\cdot}$  we associate the desired output  $y_i$ :

```
# in iris, iris[, 5] gives the Ys
iris[sample(nrow(iris), 3), ] # three random rows
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 14      4.3       3.0      1.1       0.1    setosa
## 50      5.0       3.3      1.4       0.2    setosa
## 118     7.7       3.8      6.7       2.2 virginica
```

Hence, our dataset is  $[\mathbf{X} \mathbf{y}]$  – each object is represented as a row vector  $[\mathbf{x}_{i,\cdot} \ y_i]$ ,  $i = 1, \dots, n$ :

$$[\mathbf{X} \mathbf{y}] = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} & y_1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} & y_n \end{bmatrix},$$

where:

$$\mathbf{y} = [y_1 \ y_2 \ \cdots \ y_n]^T = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

### 1.2.3 Types of Supervised Learning Problems

Depending on the type of the elements in  $\mathbf{y}$  (the domain of  $\mathbf{y}$ ), supervised learning problems are usually classified as:

- **regression** – each  $y_i$  is a real number  
e.g.,  $y_i$  = future market stock price with  $\mathbf{x}_{i,\cdot}$  = prices from  $p$  previous days
- **classification** – each  $y_i$  is a discrete label  
e.g.,  $y_i$  = healthy (0) or ill (1) with  $\mathbf{x}_{i,\cdot}$  = a patient's health record
- **ordinal regression** (a.k.a. ordinal classification) – each  $y_i$  is a rank  
e.g.,  $y_i$  = rating of a product on the scale 1–5 with  $\mathbf{x}_{i,\cdot}$  = ratings of  $p$  most similar products

#### **Exercise.**

*Example Problems – Discussion:*

*Which of the following are instances of classification problems? Which of them are regression tasks?*

*What kind of data should you gather in order to tackle them?*

- Detect email spam
- Predict a market stock price
- Predict the likeability of a new ad
- Assess credit risk
- Detect tumour tissues in medical images
- Predict time-to-recovery of cancer patients
- Recognise smiling faces on photographs
- Detect unattended luggage in airport security camera footage
- Turn on emergency braking to avoid a collision with pedestrians

□

A single dataset can become an instance of many different ML problems.

Examples – the `wines` dataset:

```
wines <- read.csv("datasets/winequality-all.csv", comment="#")
wines[1,]

##   fixed.acidity volatile.acidity citric.acid residual.sugar
## 1          7.4            0.7         0        1.9
##   chlorides free.sulfur.dioxide total.sulfur.dioxide density
## 1      0.076                  11           34  0.9978
##   pH sulphates alcohol response color
## 1 3.51      0.56     9.4       5 red
```

`alcohol` is a numeric (quantitative) variable (see Figure 1.1 for a histogram depicting its empirical distribution):

```
summary(wines$alcohol) # continuous variable

##    Min. 1st Qu. Median    Mean 3rd Qu.    Max.
##    8.00   9.50  10.30  10.49  11.30  14.90

hist(wines$alcohol, main="", col="white"); box()
```

`color` is a quantitative variable with two possible outcomes (see Figure 1.2 for a bar plot):

```
table(wines$color) # binary variable

##
##   red white
## 1599 4898

barplot(table(wines$color), col="white", ylim=c(0, 6000))
```

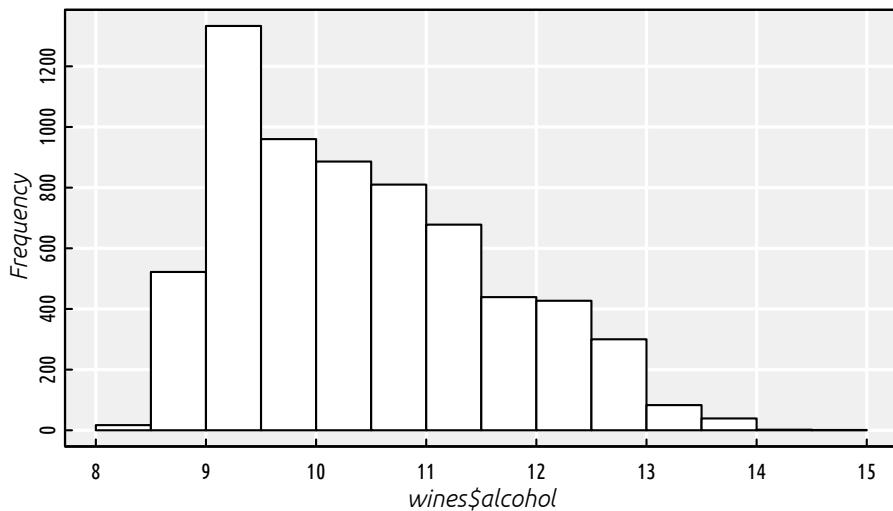


Figure 1.1: Quantitative (numeric) outputs lead to regression problems

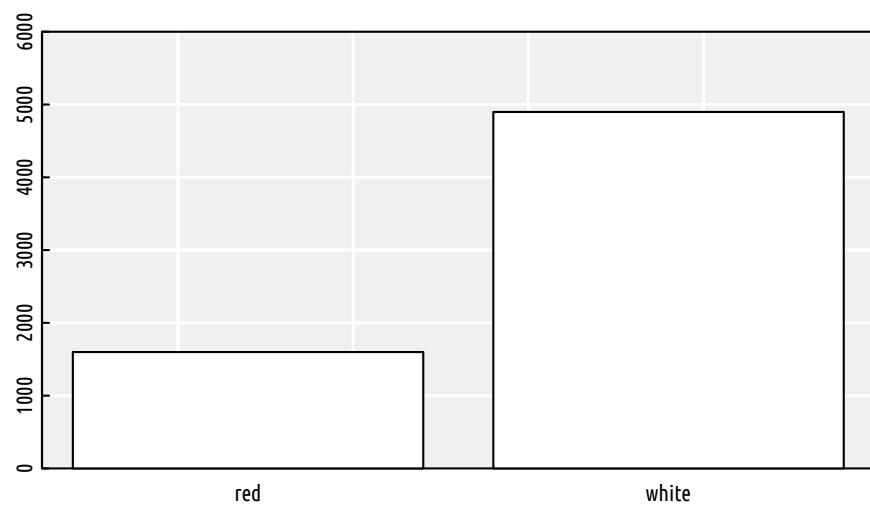


Figure 1.2: Quantitative outputs lead to classification tasks

Moreover, `response` is an ordinal variable, representing a wine's rating as assigned by a wine expert (see Figure 1.3 for a barplot). Note that although the ranks are represented with numbers, we they are not continuous variables. Moreover, these ranks are something more than just labels – they are linearly ordered, we know what's the smallest rank and what's the greatest one.

```
table(wines$response) # ordinal variable

## 
##   3    4    5    6    7    8    9
## 30  216 2138 2836 1079 193   5

barplot(table(wines$response), col="white", ylim=c(0, 3000))
```

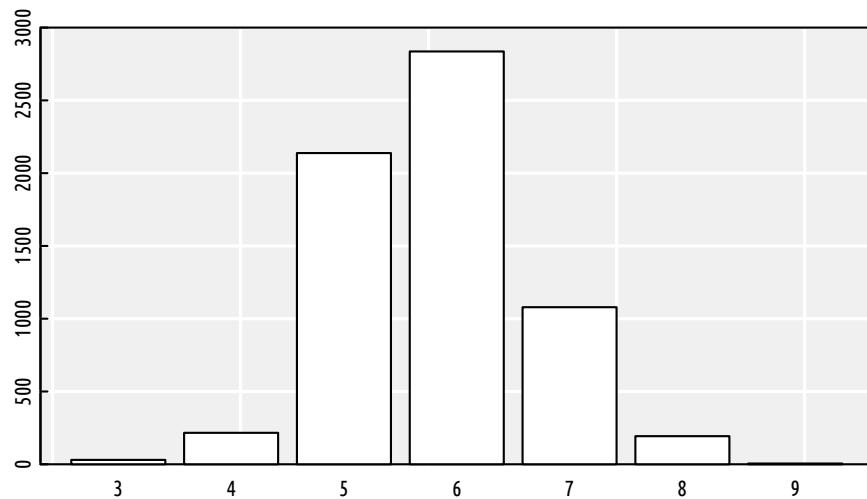


Figure 1.3: Ordinal variables constitute ordinal regression tasks

## 1.3 Simple Regression

### 1.3.1 Introduction

**Simple regression** is the easiest setting to start with – let's assume  $p = 1$ , i.e., all inputs are 1-dimensional. Denote  $x_i = x_{i,1}$ .

We will use it to build many intuitions, for example, it'll be easy to illustrate all the concepts graphically.

```
library("ISLR") # Credit dataset
plot(Credit$Balance, Credit$Rating) # scatter plot
```

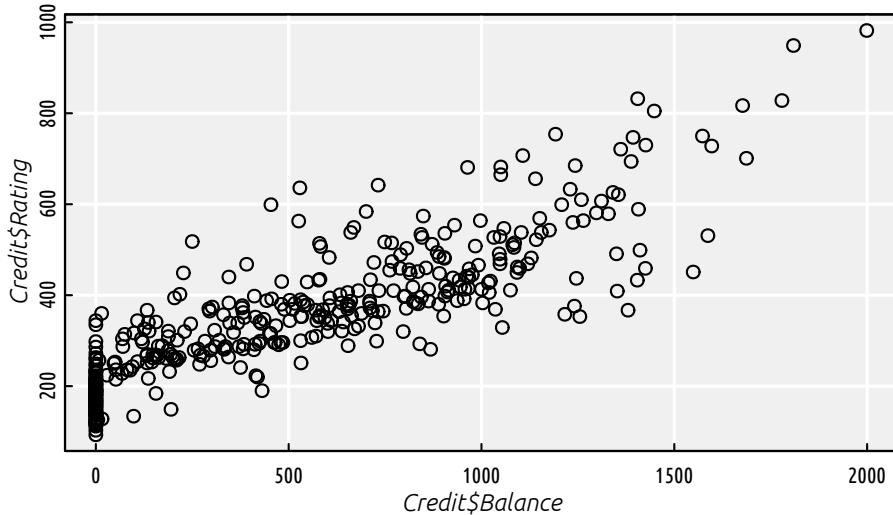


Figure 1.4: A scatter plot of Rating vs. Balance

In what follows we will be modelling the Credit Rating ( $Y$ ) as a function of the average Credit Card Balance ( $X$ ) in USD for customers *with positive Balance only*. It is because it is evident from Figure 1.4 that some customers with zero balance obtained a credit rating based on some external data source that we don't have access to in our very setting.

```
X <- as.matrix(as.numeric(Credit$Balance[Credit$Balance>0]))
Y <- as.matrix(as.numeric(Credit$Rating[Credit$Balance>0]))
```

Figure 1.5 gives the updated scatter plot with the zero-balance clients “taken care of”.

```
plot(X, Y, xlab="X (Balance)", ylab="Y (Rating)")
```

Our aim is to construct a function  $f$  that **models** Rating as a function of Balance,  $f(X) = Y$ .

We are equipped with  $n = 310$  reference (observed) Ratings  $\mathbf{y} = [y_1 \cdots y_n]^T$  for particular Balances  $\mathbf{x} = [x_1 \cdots x_n]^T$ .

Note the following naming conventions:

- Variable types:
  - $X$  – independent/explanatory/predictor variable
  - $Y$  – dependent/response/predicted variable
- Also note that:

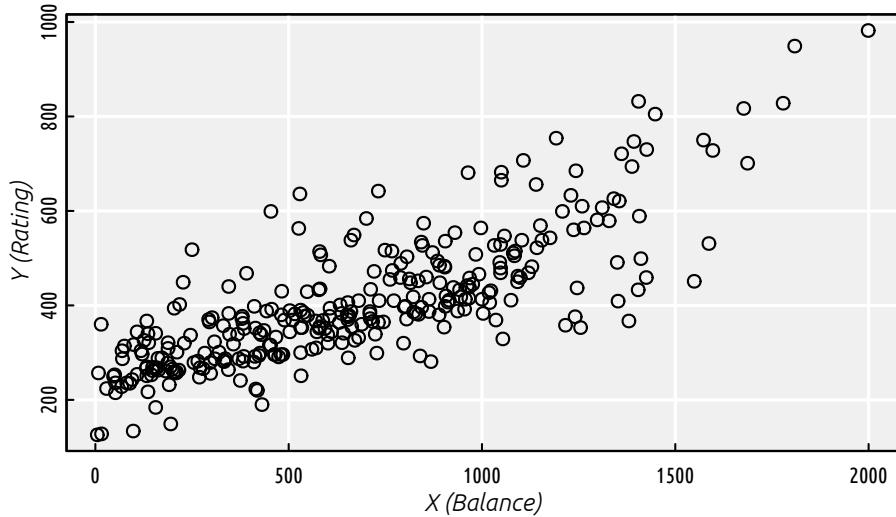


Figure 1.5: A scatter plot of Rating vs. Balance with clients of Balance=0 removed

- $Y$  – idealisation (any possible Rating)
- $\mathbf{y} = [y_1 \ \cdots \ y_n]^T$  – values actually observed

The model will not be ideal, but it might be usable:

- We will be able to **predict** the rating of any new client.  
What should be the rating of a client with Balance of \$1500?  
What should be the rating of a client with Balance of \$2500?
- We will be able to **describe** (understand) this reality using a single mathematical formula so as to infer that there is an association between  $X$  and  $Y$

Think of “data compression” and laws of physics, e.g.,  $E = mc^2$ .

(\*) Mathematically, we will assume that there is some “true” function that models the data (true relationship between  $Y$  and  $X$ ), but the observed outputs are subject to **additive error**:

$$Y = f(X) + \varepsilon.$$

$\varepsilon$  is a random term, classically we assume that errors are independent of each other, have expected value of 0 (there is no systematic error = unbiased) and that they follow a normal distribution.

(\*) We denote this as  $\varepsilon \sim \mathcal{N}(0, \sigma)$  (read: random variable  $\varepsilon$  follows a normal distribution with expected value of 0 and standard deviation of  $\sigma$  for some  $\sigma \geq 0$ ).

$\sigma$  controls the amount of noise (and hence, uncertainty). Figure 1.6 gives the plot of the probability distribution function (PDFs, densities) of  $\mathcal{N}(0, \sigma)$  for different  $\sigma$ s:

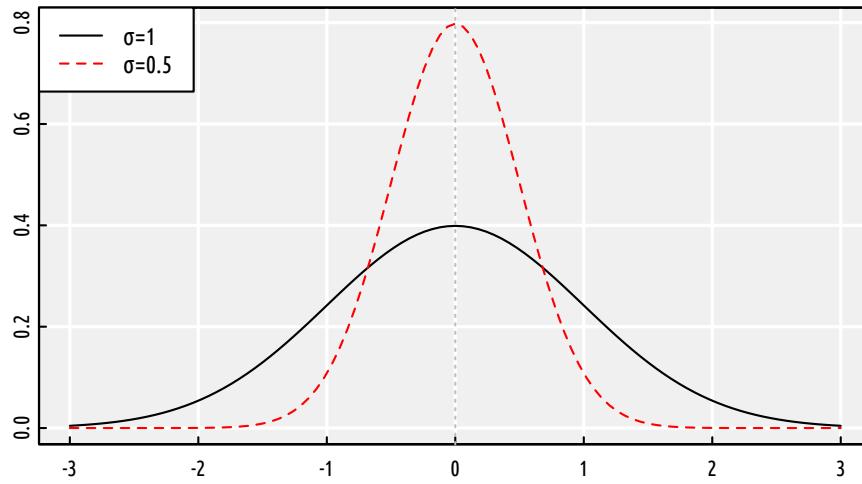


Figure 1.6: Probability density functions of normal distributions with different standard deviations  $\sigma$ .

### 1.3.2 Search Space and Objective

There are many different functions that can be **fitted** into the observed  $(\mathbf{x}, \mathbf{y})$ , compare Figure 1.7. Some of them are better than the other (with respect to different aspects, such as fit quality, simplicity etc.).

Thus, we need a formal **model selection criterion** that could enable us to tackle the model fitting task on a computer.

Usually, we will be interested in a model that minimises appropriately aggregated **residuals**  $f(x_i) - y_i$ , i.e., **predicted outputs minus observed outputs**, often denoted with  $\hat{y}_i - y_i$ , for  $i = 1, \dots, n$ .

In Figure 1.8, the residuals correspond to the lengths of the dashed line segments – they measure the discrepancy between the outputs generated by the model (what we get) and the true outputs (what we want).

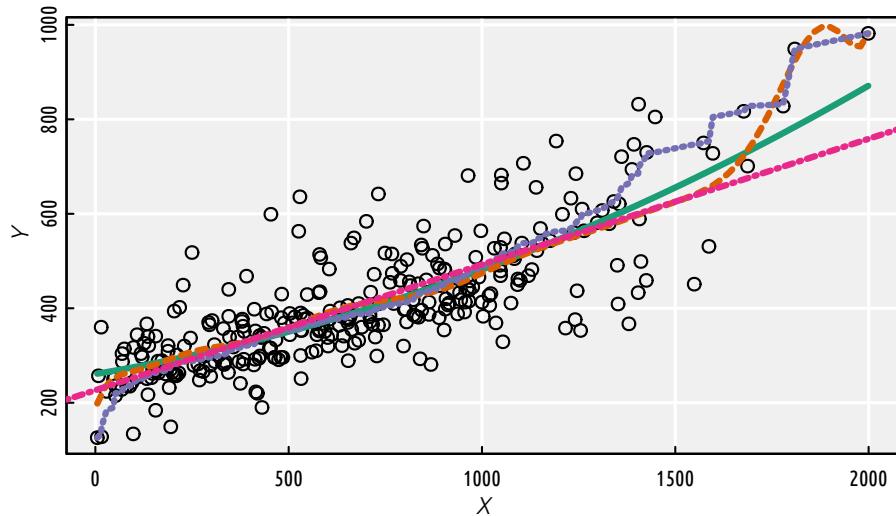


Figure 1.7: Different polynomial models fitted to data

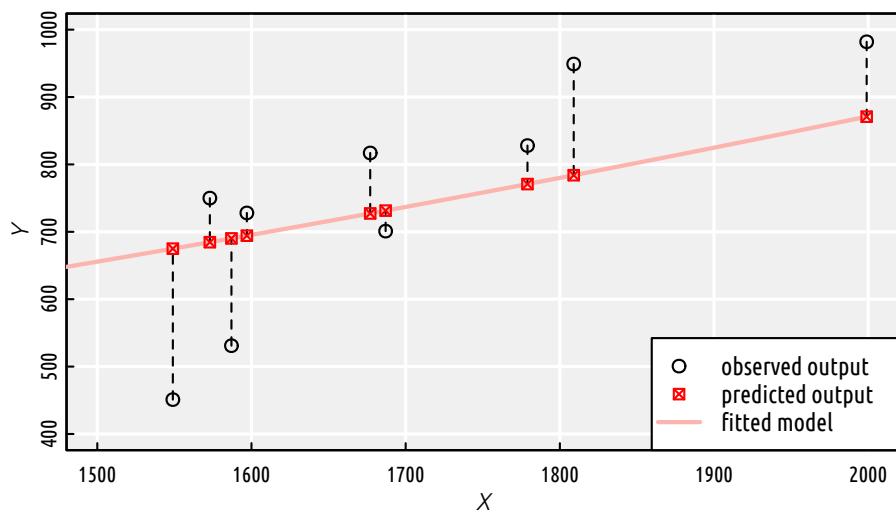


Figure 1.8: Residuals are defined as the differences between the predicted and observed outputs  $\hat{y}_i - y_i$

Top choice: sum of squared residuals:

$$\begin{aligned}\text{SSR}(f|\mathbf{x}, \mathbf{y}) &= (f(x_1) - y_1)^2 + \cdots + (f(x_n) - y_n)^2 \\ &= \sum_{i=1}^n (f(x_i) - y_i)^2\end{aligned}$$

**Remark.** Read “ $\sum_{i=1}^n z_i$ ” as “the sum of  $z_i$  for  $i$  from 1 to  $n$ ”; this is just a shorthand for  $z_1 + z_2 + \cdots + z_n$ .

**Remark.** The notation  $\text{SSR}(f|\mathbf{x}, \mathbf{y})$  means that it is the error measure corresponding to the model  $(f)$  given our data.

We could've denoted it with  $\text{SSR}_{\mathbf{x}, \mathbf{y}}(f)$  or even  $\text{SSR}(f)$  to emphasise that  $\mathbf{x}, \mathbf{y}$  are just fixed values and we are not interested in changing them at all (they are “global variables”).

We enjoy SSR because (amongst others):

- larger errors are penalised much more than smaller ones  
(this can be considered a drawback as well)
- (\*\*\*) statistically speaking, this has a clear underlying interpretation  
(assuming errors are normally distributed, finding a model minimising the SSR is equivalent to maximum likelihood estimation)
- the models minimising the SSR can often be found easily  
(corresponding optimisation tasks have an analytic solution – studied already by Gauss in the late 18th century)

(\*\*\*) Other choices:

- regularised SSR, e.g., lasso or ridge regression (in the case of multiple input variables)
- sum or median of absolute values (robust regression)

Fitting a model to data can be written as an optimisation problem:

$$\min_{f \in \mathcal{F}} \text{SSR}(f|\mathbf{x}, \mathbf{y}),$$

i.e., find  $f$  minimising the SSR (**seek “best”  $f$** ) amongst the set of admissible models  $\mathcal{F}$ .

Example  $\mathcal{F}$ s:

- $\mathcal{F} = \{\text{All possible functions of one variable}\}$  – if there are no repeated  $x_i$ 's, this corresponds to data *interpolation*; note that there are many functions that give SSR of 0.

- $\mathcal{F} = \{x \mapsto x^2, x \mapsto \cos(x), x \mapsto \exp(2x + 7) - 9\}$  – obviously an ad-hoc choice but you can easily choose the best amongst the 3 by computing 3 sums of squares.
- $\mathcal{F} = \{x \mapsto a + bx\}$  – the space of linear functions of one variable
- etc.

(e.g.,  $x \mapsto x^2$  is read “ $x$  maps to  $x^2$ ” and is an elegant way to define an inline function  $f$  such that  $f(x) = x^2$ )

## 1.4 Simple Linear Regression

### 1.4.1 Introduction

If the family of admissible models  $\mathcal{F}$  consists only of all linear functions of one variable, we deal with a **simple linear regression**.

Our problem becomes:

$$\min_{a,b \in \mathbb{R}} \sum_{i=1}^n (a + bx_i - y_i)^2$$

In other words, we seek best fitting line in terms of the squared residuals.

This is the **method of least squares**.

This is particularly nice, because our search space is just  $\mathbb{R}^2$  – easy to handle both analytically and numerically.

**Exercise.**

Which of lines in Figure 1.9 is the least squares solution?

□

### 1.4.2 Solution in R

Let's fit the linear model minimising the SSR in R. The `lm()` function (linear models) has a convenient *formula*-based interface.

```
f <- lm(Y~X)
```

In R, the expression “`Y~X`” denotes a formula, which we read as: variable `Y` is a function of `X`. Note that the dependent variable is on the left side of the formula. Here, `X` and `Y` are two R numeric vectors of identical lengths.

Let's print the fitted model:

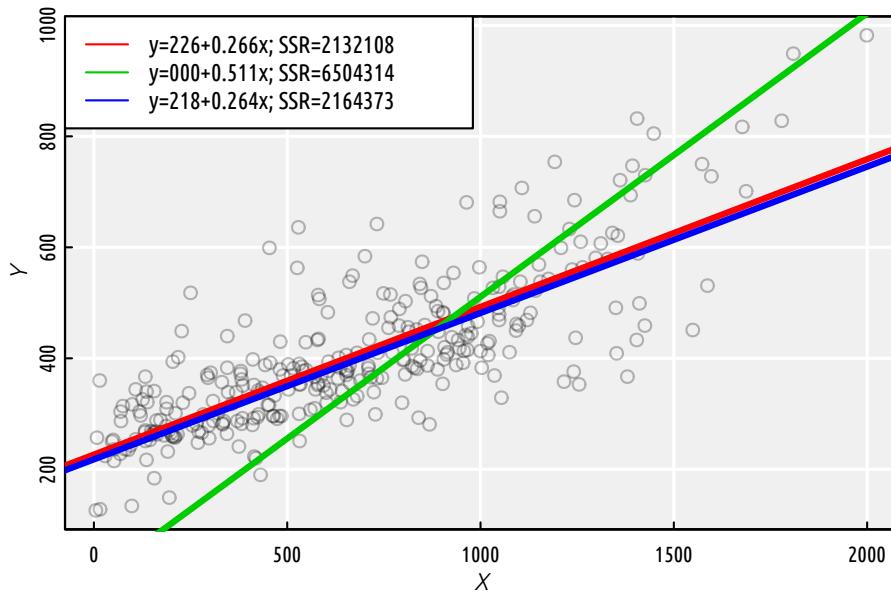


Figure 1.9: Three simple linear models together with the corresponding SSRs

```

print(f)

##
## Call:
## lm(formula = Y ~ X)
##
## Coefficients:
## (Intercept)          X
##   226.4711      0.2661

```

Hence, the fitted model is:

$$Y = f(X) = 226.4711446 + 0.2661459X \quad (+\varepsilon)$$

Coefficient  $a$  (intercept):

```
f$coefficient[1]
```

```
## (Intercept)
##   226.4711
```

Coefficient  $b$  (slope):

```
f$coefficient[2]
```

```
##          X
## 0.2661459
```

Plotting, see Figure 1.10:

```
plot(X, Y, col="#000000aa")
abline(f, col=2, lwd=3)
```

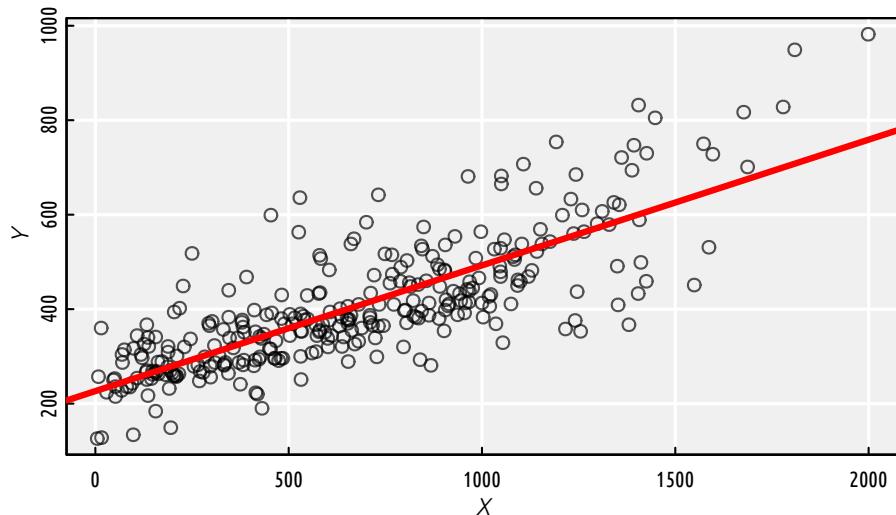


Figure 1.10: Fitted regression line

SSR:

```
sum(f$residuals^2)

## [1] 2132108
sum((f$coefficient[1]+f$coefficient[2]*X-Y)^2) # equivalent

## [1] 2132108
```

We can predict the model's output for yet-unobserved inputs by writing:

```
X_new <- c(1500, 2000, 2500) # example inputs
f$coefficient[1] + f$coefficient[2]*X_new
```

```
## [1] 625.6900 758.7630 891.8359
```

Note that linear models can also be fitted based on formulas that refer to a data frame's columns. For example, let us wrap both **x** and **y** inside a data frame:

```
XY <- data.frame(Balance=X, Rating=Y)
head(XY, 3)
```

```
##   Balance Rating
## 1     333    283
## 2     903    483
## 3     580    514
```

By writing:

```
f <- lm(Rating~Balance, data=XY)
```

now `Balance` and `Rating` refer to column names in the `XY` data frame, and not the objects in R's "workspace".

Based on the above, we can make a prediction using the `predict()` function"

```
X_new <- data.frame(Balance=c(1500, 2000, 2500))
predict(f, X_new)
```

```
##      1      2      3
## 625.6900 758.7630 891.8359
```

Interestingly:

```
predict(f, data.frame(Balance=c(5000)))
```

```
##      1
## 1557.201
```

This is more than the highest possible rating – we have extrapolated way beyond the observable data range.

Note that our  $Y = a + bX$  model is **interpretable** and **well-behaving** (not all machine learning models will have this feature, think: deep neural networks, which we rather conceive as *black boxes*):

- we know that by increasing  $X$  by a small amount,  $Y$  will also increase (positive correlation),
- the model is continuous – small change in  $X$  doesn't yield any drastic change in  $Y$ ,
- we know what will happen if we increase or decrease  $X$  by, say, 100,
- the function is invertible – if we want Rating of 500, we can compute the associated preferred Balance that should yield it (provided that the model is valid).

### 1.4.3 Analytic Solution

It may be shown (which we actually do below) that the solution is:

$$\begin{cases} b = \frac{n \sum_{i=1}^n x_i y_i - \sum_{i=1}^n y_i \sum_{i=1}^n x_i}{n \sum_{i=1}^n x_i x_i - \sum_{i=1}^n x_i \sum_{i=1}^n x_i} \\ a = \frac{1}{n} \sum_{i=1}^n y_i - b \frac{1}{n} \sum_{i=1}^n x_i \end{cases}$$

Which can be implemented in R as follows:

```
n <- length(X)
b <- (n*sum(X*Y) - sum(X)*sum(Y)) / (n*sum(X*X) - sum(X)^2)
a <- mean(Y) - b*mean(X)
c(a, b) # the same as f$coefficients
```

```
## [1] 226.4711446 0.2661459
```

#### 1.4.4 Derivation of the Solution (\*\*)

**Remark.** You can safely skip this part if you are yet to know how to search for a minimum of a function of many variables and what are partial derivatives.

Denote with:

$$E(a, b) = \text{SSR}(x \mapsto a + bx | \mathbf{x}, \mathbf{y}) = \sum_{i=1}^n (a + bx_i - y_i)^2.$$

We seek the minimum of  $E$  w.r.t. both  $a, b$ .

**Theorem.** If  $E$  has a (local) minimum at  $(a^*, b^*)$ , then its partial derivatives vanish therein, i.e.,  $\partial E / \partial a(a^*, b^*) = 0$  and  $\partial E / \partial b(a^*, b^*) = 0$ .

We have:

$$E(a, b) = \sum_{i=1}^n (a + bx_i - y_i)^2.$$

We need to compute the partial derivatives  $\partial E / \partial a$  (derivative of  $E$  w.r.t. variable  $a$  – all other terms treated as constants) and  $\partial E / \partial b$  (w.r.t.  $b$ ).

Useful rules – derivatives w.r.t.  $a$  (denote  $f'(a) = (f(a))'$ ):

- $(f(a) + g(a))' = f'(a) + g'(a)$  (derivative of sum is sum of derivatives)
- $(f(a)g(a))' = f'(a)g(a) + f(a)g'(a)$  (derivative of product)
- $(f(g(a)))' = f'(g(a))g'(a)$  (chain rule)
- $(c)' = 0$  for any constant  $c$  (expression not involving  $a$ )

- $(a^p)' = pa^{p-1}$  for any  $p$
- in particular:  $(ca^2 + d)' = 2ca$ ,  $(ca)' = c$ ,  $((ca + d)^2)' = 2(ca + d)c$  (application of the above rules)

We seek  $a, b$  such that  $\frac{\partial E}{\partial a}(a, b) = 0$  and  $\frac{\partial E}{\partial b}(a, b) = 0$ .

$$\begin{cases} \frac{\partial E}{\partial a}(a, b) = 2 \sum_{i=1}^n (a + bx_i - y_i) = 0 \\ \frac{\partial E}{\partial b}(a, b) = 2 \sum_{i=1}^n (a + bx_i - y_i) x_i = 0 \end{cases}$$

This is a system of 2 linear equations. Easy.

Rearranging like back in the school days:

$$\begin{cases} b \sum_{i=1}^n x_i + an = \sum_{i=1}^n y_i \\ b \sum_{i=1}^n x_i x_i + a \sum_{i=1}^n x_i = \sum_{i=1}^n x_i y_i \end{cases}$$

It is left as an exercise to show that the solution is:

$$\begin{cases} b^* = \frac{n \sum_{i=1}^n x_i y_i - \sum_{i=1}^n y_i \sum_{i=1}^n x_i}{n \sum_{i=1}^n x_i x_i - \sum_{i=1}^n x_i \sum_{i=1}^n x_i} \\ a^* = \frac{1}{n} \sum_{i=1}^n y_i - b^* \frac{1}{n} \sum_{i=1}^n x_i \end{cases}$$

(we should additionally perform the second derivative test to assure that this is the minimum of  $E$  – which is exactly the case though)

(\*\*) In the next chapter, we will introduce the notion of Pearson's linear coefficient,  $r$  (see `cor()` in R). It might be shown that  $a$  and  $b$  can also be rewritten as:

```
(b <- cor(X,Y)*sd(Y)/sd(X))
```

```
## [1] [,1]
## [1,] 0.2661459
(a <- mean(Y)-b*mean(X))

## [1] [,1]
## [1,] 226.4711
```

## 1.5 Exercises in R

### 1.5.1 The Anscombe Quartet

Here is a famous illustrative example proposed by the statistician Francis Anscombe in the early 1970s.

```
print(anscombe) # `anscombe` is a built-in object
```

```
##   x1  x2  x3  x4    y1    y2    y3    y4
## 1 10  10  10   8  8.04 9.14  7.46  6.58
## 2   8   8   8   8  6.95 8.14  6.77  5.76
## 3 13  13  13   8  7.58 8.74 12.74  7.71
## 4   9   9   9   8  8.81 8.77  7.11  8.84
## 5 11  11  11   8  8.33 9.26  7.81  8.47
## 6 14  14  14   8  9.96 8.10  8.84  7.04
## 7   6   6   6   8  7.24 6.13  6.08  5.25
## 8   4   4   4  19  4.26 3.10  5.39 12.50
## 9 12  12  12   8 10.84 9.13  8.15  5.56
## 10  7   7   7   8  4.82 7.26  6.42  7.91
## 11  5   5   5   8  5.68 4.74  5.73  6.89
```

What we see above is a single data frame that encodes four separate datasets: `anscombe$x1` and `anscombe$y1` define the first pair of variables, `anscombe$x2` and `anscombe$y2` define the second pair and so forth.

#### Exercise.

Split the above data (manually) into four data frames `ans1`, ..., `ans4` with columns `x` and `y`.

For example, `ans1` should look like:

```
print(ans1)
```

```
##   x     y
## 1 10  8.04
## 2   8  6.95
## 3 13  7.58
## 4   9  8.81
## 5 11  8.33
## 6 14  9.96
## 7   6  7.24
## 8   4  4.26
## 9 12 10.84
## 10  7  4.82
## 11  5  5.68
```

□

**Solution.**

```
ans1 <- data.frame(x=anscombe$x1, y=anscombe$y1)
ans2 <- data.frame(x=anscombe$x2, y=anscombe$y2)
ans3 <- data.frame(x=anscombe$x3, y=anscombe$y3)
ans4 <- data.frame(x=anscombe$x4, y=anscombe$y4)
print(ans1)
```

**Exercise.**

*Compute the mean of each  $x$  and  $y$  variable.*

**Solution.**

```
mean(ans1$x) # individual column
## [1] 9
mean(ans1$y) # individual column
## [1] 7.500909
sapply(ans2, mean) # all columns in ans2
##      x      y
## 9.000000 7.500909
sapply(anscombe, mean) # all columns in the full anscombe dataset
##      x1      x2      x3      x4      y1      y2      y3
## 9.000000 9.000000 9.000000 9.000000 7.500909 7.500909 7.500000
##      y4
## 7.500909
```

*Comment: This is really interesting, all the means of  $x$  columns as well as the means of  $y$ s are almost identical.*

**Exercise.**

*Compute the standard deviation of each  $x$  and  $y$  variable.*

**Solution.**

The solution is similar to the previous one, just replace `mean` with `sd`. Here, to learn something new, we will use the `knitr::kable()` function that pretty-prints a given matrix or data frame:

```
results <- sapply(anscombe, sd)
knitr::kable(results, col.names="standard deviation")
```

	standard deviation
x1	3.316625
x2	3.316625
x3	3.316625
x4	3.316625
y1	2.031568
y2	2.031657
y3	2.030424
y4	2.030578

*Comment:* This is even more interesting, because the numbers agree up to 2 decimal digits.

■

### Exercise.

Fit a simple linear regression model for each data set. Draw the scatter plots again (`plot()`) and add the regression lines (`lines()` or `abline()`).

□

### Solution.

To recall, this can be done with the `lm()` function explained in Lecture 2.

At this point we should already have become lazy – the tasks are very repetitious. Let's automate them by writing a single function that does all the above for any data set:

```
fit_models <- function(ans) {
  # ans is a data frame with columns x and y
  f <- lm(y~x, data=ans) # fit linear model
  print(f$coefficients) # estimated coefficients
  plot(ans$x, ans$y) # scatter plot
  abline(f, col="red") # regression line
  return(f)
}
```

Now we can apply it on the four particular examples.

```
par(mfrow=c(2, 2)) # four plots on 1 figure (2x2 grid)
f1 <- fit_models(ans1)

## (Intercept)          x
##   3.0000909   0.5000909
```

```
f2 <- fit_models(ans2)

## (Intercept)          x
## 3.000909   0.500000

f3 <- fit_models(ans3)

## (Intercept)          x
## 3.0024545  0.4997273

f4 <- fit_models(ans4)

## (Intercept)          x
## 3.0017273  0.4999091
```

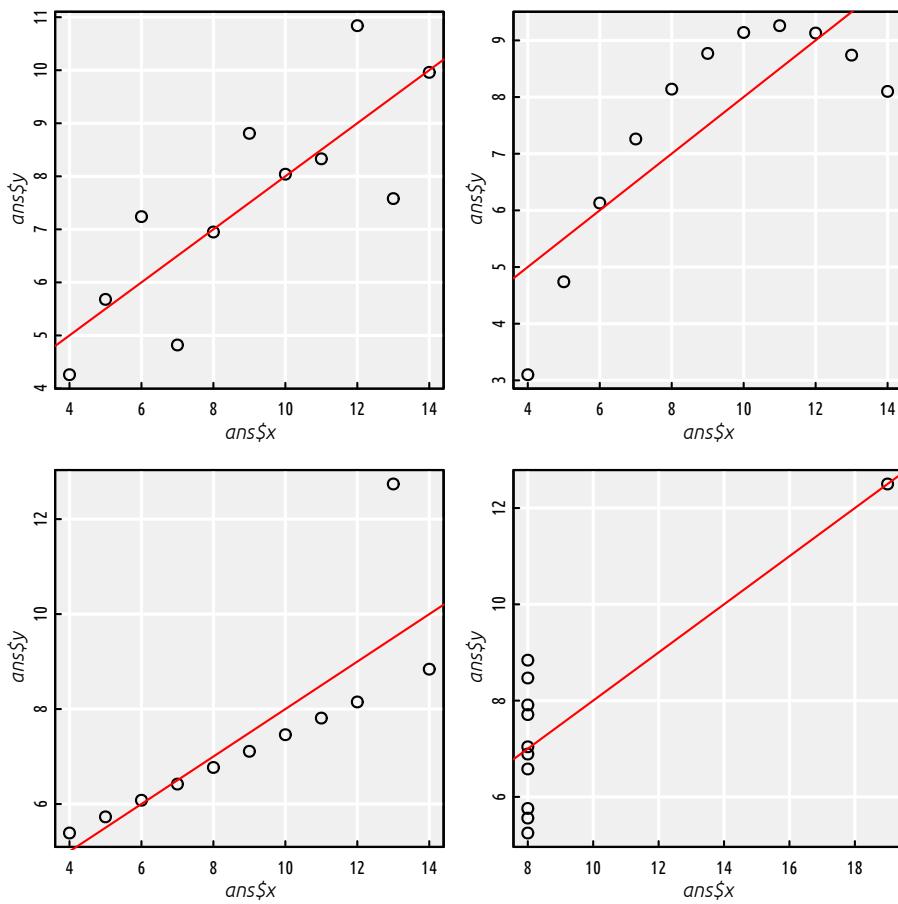


Figure 1.11: Fitted regression lines for the Anscombe quartet

*Comment: All the estimated models are virtually the same, the regression lines are*

$y = 0.5x + 3$ , compare Figure 1.11.



### Exercise.

Create scatter plots of the residuals (predicted  $\hat{y}_i$  minus true  $y_i$ ) as a function of the predicted  $\hat{y}_i = f(x_i)$  for every  $i = 1, \dots, 11$ .



### Solution.

To recall, the model predictions can be generated by (amongst others) calling the `predict()` function.

```
y_pred1 <- predict(f1, ans1)
y_pred2 <- predict(f2, ans2)
y_pred3 <- predict(f3, ans3)
y_pred4 <- predict(f4, ans4)
```

Plots of residuals as a function of the predicted (fitted) values are given in Figure 1.12.

```
par(mfrow=c(2, 2)) # four plots on 1 figure (2x2 grid)
plot(y_pred1, y_pred1-ans1$y)
plot(y_pred2, y_pred2-ans2$y)
plot(y_pred3, y_pred3-ans3$y)
plot(y_pred4, y_pred4-ans4$y)
```

*Comment: Ideally, the residuals shouldn't be correlated with the predicted values – they should “oscillate” randomly around 0. This is only the case of the first dataset. All the other cases are “alarming” in the sense that they suggest that the obtained models are “suspicious” (perhaps data cleansing is needed or a linear model is not at all appropriate).*



### Exercise.

Draw conclusions (in your own words).



### Solution.

We're being taught a lesson here: don't perform data analysis tasks automatically, don't look at bare numbers only, visualise your data first!



### Exercise.

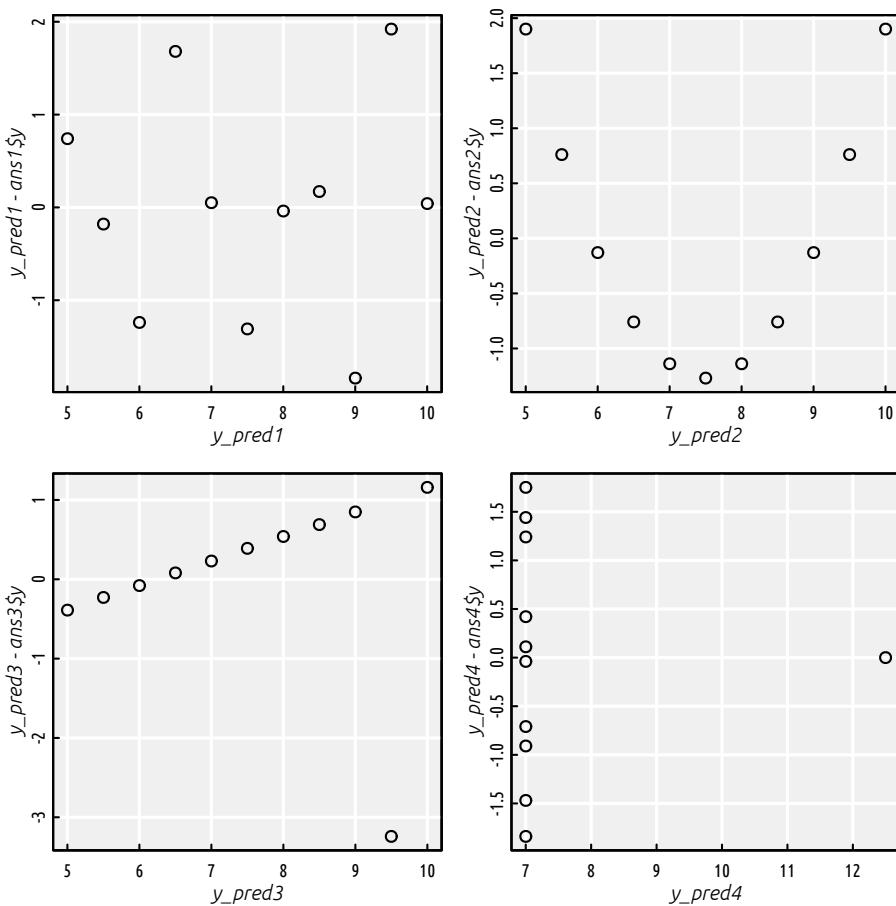


Figure 1.12: Residuals vs. fitted values for the regression lines fitted to the Anscombe quartet

*Read more about Anscombe's quartet at [https://en.wikipedia.org/wiki/Anscombe%27s\\_quartet](https://en.wikipedia.org/wiki/Anscombe%27s_quartet)*

□

## 1.6 Outro

### 1.6.1 Remarks

In supervised learning, with each input point, there's an associated reference output value.

Learning a model = constructing a function that approximates (minimising some error measure) the given data.

Regression = the output variable  $Y$  is continuous.

We studied linear models with a single independent variable based on the least squares (SSR) fit.

In the next part we will extend this setting to the case of many variables, i.e.,  $p > 1$ , called multiple regression.

### 1.6.2 Further Reading

Recommended further reading: (James et al. 2017: Chapters 1, 2 and 3)

Other: (Hastie et al. 2017: Chapter 1, Sections 3.2 and 3.3)

# Chapter 2

## Multiple Regression

### 2.1 Introduction

#### 2.1.1 Formalism

Let  $\mathbf{X} \in \mathbb{R}^{n \times p}$  be an input matrix that consists of  $n$  points in a  $p$ -dimensional space.

In other words, we have a database on  $n$  objects, each of which being described by means of  $p$  numerical features.

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix}$$

Recall that in supervised learning, apart from  $\mathbf{X}$ , we are also given the corresponding  $\mathbf{y}$ ; with each input point  $\mathbf{x}_{i,\cdot}$ , we associate the desired output  $y_i$ .

In this chapter we are still interested in **regression** tasks; hence, we assume that each  $y_i$  it is a real number, i.e.,  $y_i \in \mathbb{R}$ .

Hence, our dataset is  $[\mathbf{X} \ \mathbf{y}]$  – where each object is represented as a row vector  $[\mathbf{x}_{i,\cdot} \ y_i]$ ,  $i = 1, \dots, n$ :

$$[\mathbf{X} \ \mathbf{y}] = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} & y_1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} & y_n \end{bmatrix}.$$

### 2.1.2 Simple Linear Regression - Recap

In a simple regression task, we have assumed that  $p = 1$  – there is only one independent variable, denoted  $x_i = x_{i,1}$ .

We restricted ourselves to linear models of the form  $Y = f(X) = a + bX$  that minimised the sum of squared residuals (SSR), i.e.,

$$\min_{a,b \in \mathbb{R}} \sum_{i=1}^n (a + bx_i - y_i)^2.$$

The solution is:

$$\begin{cases} b = \frac{n \sum_{i=1}^n x_i y_i - \sum_{i=1}^n y_i \sum_{i=1}^n x_i}{n \sum_{i=1}^n x_i x_i - \sum_{i=1}^n x_i \sum_{i=1}^n x_i} \\ a = \frac{1}{n} \sum_{i=1}^n y_i - b \frac{1}{n} \sum_{i=1}^n x_i \end{cases}$$

Fitting in R can be performed by calling the `lm()` function:

```
library("ISLR") # Credit dataset
X <- as.numeric(Credit$Balance[Credit$Balance>0])
Y <- as.numeric(Credit$Rating[Credit$Balance>0])
f <- lm(Y~X) # Y~X is a formula, read: Y is a function of X
print(f)

##
## Call:
## lm(formula = Y ~ X)
##
## Coefficients:
## (Intercept)          X
##     226.4711      0.2661
```

Figure 2.1 gives the scatter plot of  $Y$  vs.  $X$  together with the fitted simple linear model.

```
plot(X, Y, xlab="X (Balance)", ylab="Y (Credit)")
abline(f, col=2, lwd=3)
```

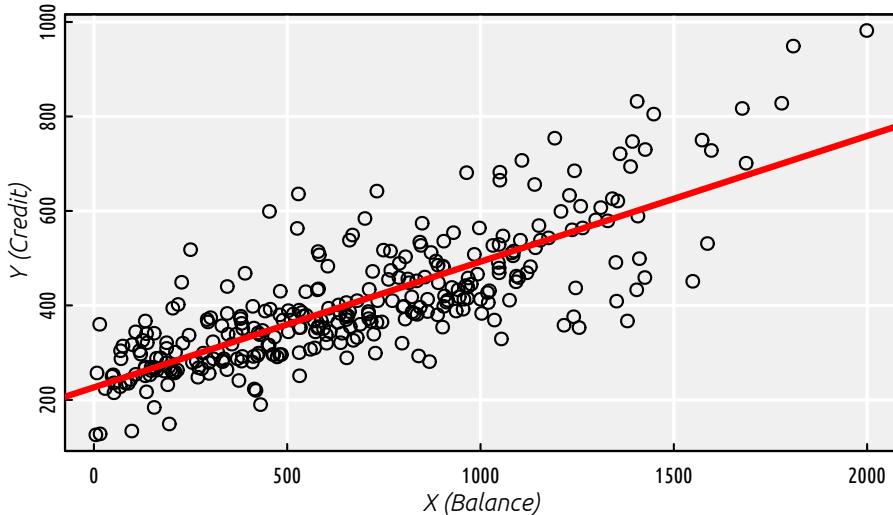


Figure 2.1: Fitted regression line for the Credit dataset

## 2.2 Multiple Linear Regression

### 2.2.1 Problem Formulation

Let's now generalise the above to the case of many variables  $X_1, \dots, X_p$ .

We wish to model the dependent variable as a function of  $p$  independent variables.

$$Y = f(X_1, \dots, X_p) \quad (+\varepsilon)$$

Restricting ourselves to the class of **linear models**, we have

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p.$$

Above we studied the case where  $p = 1$ , i.e.,  $Y = a + bX_1$  with  $\beta_0 = a$  and  $\beta_1 = b$ .

The above equation defines:

- $p = 1$  — a line (see Figure 2.1),
- $p = 2$  — a plane (see Figure 2.2),
- $p \geq 3$  — a hyperplane (well, most people find it difficult to imagine objects in high dimensions, but we are lucky to have this thing called maths).

### 2.2.2 Fitting a Linear Model in R

`lm()` accepts a formula of the form `Y~X1+X2+...+Xp`.

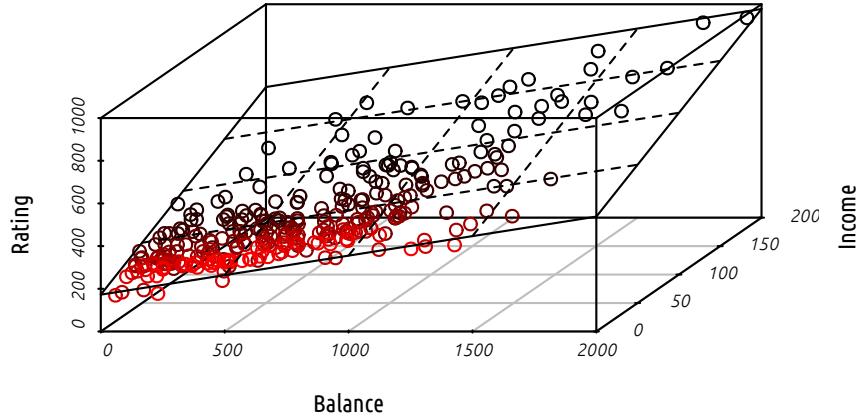


Figure 2.2: Fitted regression plane for the Credit dataset

It finds the least squares fit, i.e., solves

$$\min_{\beta_0, \beta_1, \dots, \beta_p \in \mathbb{R}} \sum_{i=1}^n (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i)^2$$

```
X1 <- as.numeric(Credit$Balance[Credit$Balance>0])
X2 <- as.numeric(Credit$Income[Credit$Balance>0])
Y <- as.numeric(Credit$Rating[Credit$Balance>0])
f <- lm(Y~X1+X2)
f$coefficients # β₀, β₁, β₂

## (Intercept)          X1           X2
## 172.5586670   0.1828011   2.1976461
```

By the way, the 3D scatter plot in Figure 2.2 was generated by calling:

```
library("scatterplot3d")
s3d <- scatterplot3d(X1, X2, Y,
  angle=60, # change angle to reveal more
  highlight.3d=TRUE, xlab="Balance", ylab="Income",
  zlab="Rating")
s3d$plane3d(f, lty.box="solid")
```

(`s3d` is an R list, one of its elements named `plane3d` is a function object – this is legal)

## 2.3 Finding the Best Model

### 2.3.1 Model Diagnostics

Here is Rating ( $Y$ ) as function of Balance ( $X_1$ , lefthand side of Figure 2.3) and Income ( $X_2$ , righthand side of Figure 2.3).

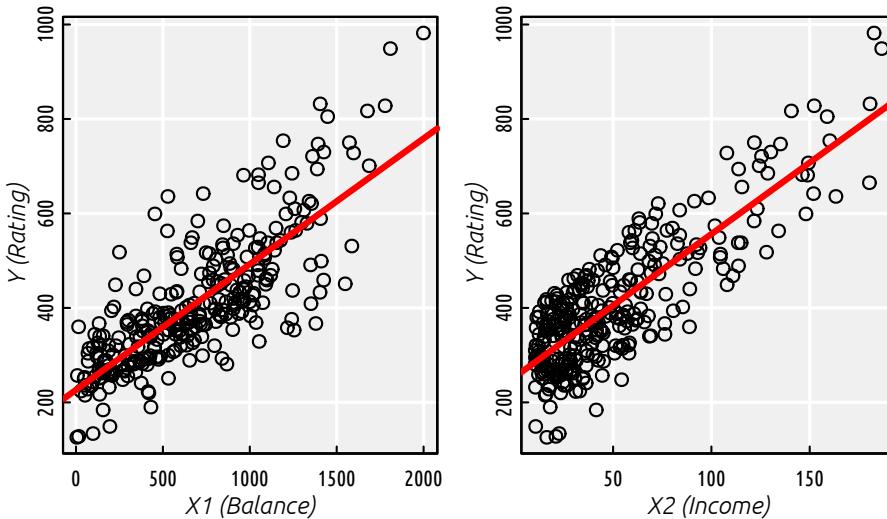


Figure 2.3: Scatter plots of  $Y$  vs.  $X_1$  and  $X_2$

Moreover, Figure 2.4 depicts (in a hopefully readable manner) both  $X_1$  and  $X_2$  with Rating  $Y$  encoded with a colour (low ratings are green, high ratings are red; some rating values are explicitly printed out within the plot).

Consider the three following models.

Formula	Equation
Rating ~ Balance + Income	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$
Rating ~ Balance	$Y = a + b X_1$ ( $\beta_0 = a, \beta_1 = b, \beta_2 = 0$ )
Rating ~ Income	$Y = a + b X_2$ ( $\beta_0 = a, \beta_1 = 0, \beta_2 = b$ )

```
f12 <- lm(Y~X1+X2) # Rating ~ Balance + Income
f12$coefficients

## (Intercept)          X1          X2
## 172.5586670   0.1828011   2.1976461

f1  <- lm(Y~X1)      # Rating ~ Balance
f1$coefficients
```

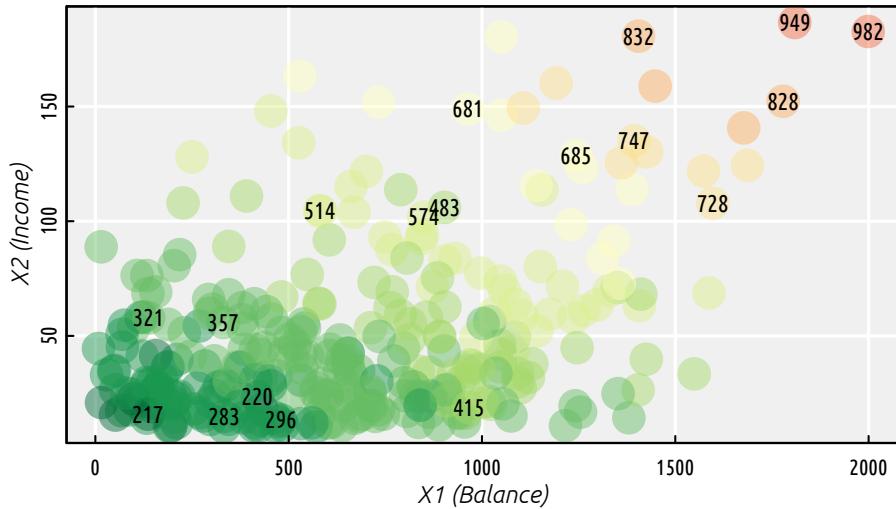


Figure 2.4: A heatmap for Rating as a function of Balance and Income; greens represent low credit ratings, whereas reds – high ones

```
## (Intercept)           X1
## 226.4711446   0.2661459
f2 <- lm(Y~X2)      # Rating ~ Income
f2$coefficients

## (Intercept)           X2
## 253.851416    3.025286
```

Which of the three models is the best? Of course, by using the word “best”, we need to answer the question “best?... but with respect to what kind of measure?”

So far we were fitting w.r.t. SSR, as the multiple regression model generalises the two simple ones, the former must yield a not-worse SSR. This is because in the case of  $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$ , setting  $\beta_1$  to 0 (just one of uncountably many possible  $\beta_1$ s, if it happens to be the *best* one, good for us) gives  $Y = a + bX_2$  whereas by setting  $\beta_2$  to 0 we obtain  $Y = a + bX_1$ .

```
sum(f12$residuals^2)
```

```
## [1] 358260.6
```

```
sum(f1$residuals^2)
```

```
## [1] 2132108
```

```
sum(f2$residuals^2)
```

```
## [1] 1823473
```

We get that, in terms of SSRs,  $f_{12}$  is better than  $f_2$ , which in turn is better than  $f_1$ . However, these error values per se (sheer numbers) are meaningless (not meaningful).

**Remark.** Interpretability in ML has always been an important issue, think the EU General Data Protection Regulation (GDPR), amongst others.

### 2.3.1.1 SSR, MSE, RMSE and MAE

The quality of fit can be assessed by performing some *descriptive statistical analysis of the residuals*,  $\hat{y}_i - y_i$ , for  $i = 1, \dots, n$ .

I know how to summarise data on the residuals! Of course I should compute their arithmetic mean and I'm done with that shtask! Interestingly, the mean of residuals (this can be shown analytically) in the least squared fit is always equal to 0:

$$\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i) = 0.$$

Therefore, we need a different metric.

#### Exercise.

(\*) A proof of this fact is left as an exercise to the curious; assume  $p = 1$  just as in the previous chapter and note that  $\hat{y}_i = ax_i + b$ .

□

```
mean(f12$residuals) # almost zero numerically
```

```
## [1] 4.444473e-16
```

```
all.equal(mean(f12$residuals), 0)
```

```
## [1] TRUE
```

We noted that sum of squared residuals (SSR) is not interpretable, but the mean squared residuals (MSR) – also called mean squared error (MSE) regression loss – is a little better. Recall that mean is defined as the sum divided by number of samples.

$$\text{MSE}(f) = \frac{1}{n} \sum_{i=1}^n (f(\mathbf{x}_{i,\cdot}) - y_i)^2.$$

```
mean(f12$residuals^2)

## [1] 1155.679
mean(f1$residuals^2)

## [1] 6877.768
mean(f2$residuals^2)

## [1] 5882.171
```

This gives an information of how much do we err *per sample*, so at least this measure does not depend on  $n$  anymore. However, if the original  $Y$ s are, say, in metres [m], MSE is expressed in metres squared [ $\text{m}^2$ ].

To account for that, we may consider the root mean squared error (RMSE):

$$\text{RMSE}(f) = \sqrt{\frac{1}{n} \sum_{i=1}^n (f(\mathbf{x}_{i,\cdot}) - y_i)^2}.$$

This is just like with the sample variance vs. standard deviation – recall the latter is defined as the square root of the former.

```
sqrt(mean(f12$residuals^2))

## [1] 33.99528
sqrt(mean(f1$residuals^2))

## [1] 82.93231
sqrt(mean(f2$residuals^2))

## [1] 76.69531
```

The interpretation of the RMSE is rather quirky; it is some-sort-of-averaged *deviance* from the true rating (which is on the scale 0–1000, hence we see that the first model is not that bad). Recall that the square function is sensitive to large observations, hence, it penalises notable deviations more heavily.

As still we have a problem with finding something easily interpretable (your non-technical boss or client may ask you: but what do these numbers mean??), we suggest here that the mean absolute error (MAE; also called mean absolute deviations, MAD) might be a better idea than the above:

$$\text{MAE}(f) = \frac{1}{n} \sum_{i=1}^n |f(\mathbf{x}_{i,\cdot}) - y_i|.$$

```
mean(abs(f12$residuals))

## [1] 22.86342
mean(abs(f1$residuals))

## [1] 61.48892
mean(abs(f2$residuals))

## [1] 64.1506
```

With the above we may say “On average, the predicted rating differs from the observed one by...”. That is good enough.

**Remark.** (\*) You may ask why don’t we fit models so as to minimise the MAE and we minimise the RMSE instead (note that minimising RMSE is the same as minimising the SSR, one is a strictly monotone transformation of the other and do not affect the solution). Well, it is possible. It turns out that, however, minimising MAE is more computationally expensive and the solution may be numerically unstable. So it’s rarely an analyst’s first choice (assuming they are well-educated enough to know about the MAD regression task). However, it may be worth trying it out sometimes.

Sometimes we might prefer MAD regression to the classic one if our data is heavily contaminated by outliers. But in such cases it is worth checking if proper data cleansing does the trick.

### 2.3.1.2 Graphical Summaries of Residuals

If we are not happy with single numerical aggregated of the residuals or their absolute values, we can (and should) always compute a whole bunch of descriptive statistics:

```
summary(f12$residuals)

##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
## -108.100 -1.940   7.812    0.000  20.249   50.623

summary(f1$residuals)

##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
## -226.75 -48.30  -10.08     0.00  42.58  268.74

summary(f2$residuals)

##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
## -195.156 -57.341 -1.284     0.000 64.013 175.344
```

The outputs generated by `summary()` include:

- `Min.` – sample minimum

- 1st Qu. – 1st quartile == 25th percentile == quantile of order 0.25
- Median – median == 50th percentile == quantile of order 0.5
- 3rd Qu. – 3rd quartile = 75th percentile == quantile of order 0.75
- Max. – sample maximum

For example, 1st quartile is the observation  $q$  such that 25% values are  $\leq q$  and 75% values are  $\geq q$ , see `?quantile` in R.

Graphically, it is nice to summarise the empirical distribution of the residuals on a **box and whisker plot**. Here is the key to decipher Figure 2.5:

- IQR == Interquartile range ==  $Q_3 - Q_1$  (box width)
- The box contains 50% of the “most typical” observations
- Box and whiskers altogether have width  $\leq 4$  IQR
- Outliers == observations potentially worth inspecting (is it a bug or a feature?)

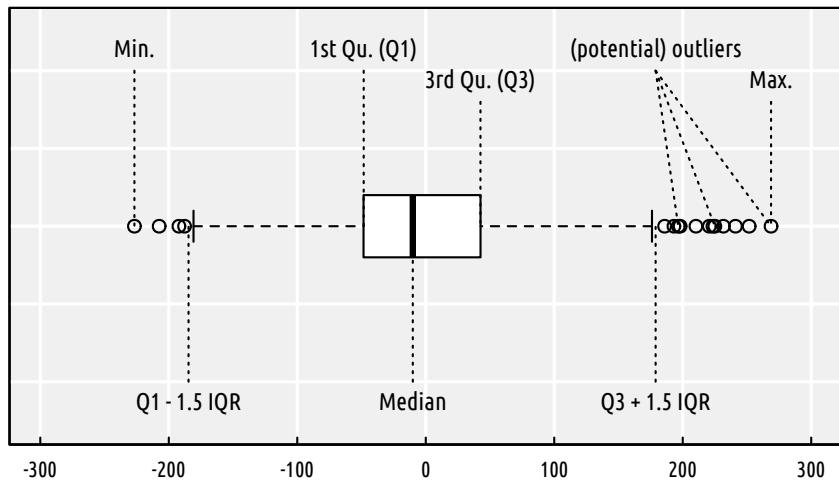


Figure 2.5: An example boxplot

Figure 2.6 is worth a thousand words:

```
boxplot(horizontal=TRUE, xlab="residuals", col="white",
        list(f12=f12$residuals, f1=f1$residuals, f2=f2$residuals))
abline(v=0, lty=3)
```

Figure 2.7 gives a *violin plot* – a blend of a box plot and a (kernel) density estimator (histogram-like):

```
library("vioplot")
vioplot(horizontal=TRUE, xlab="residuals", col="white",
```

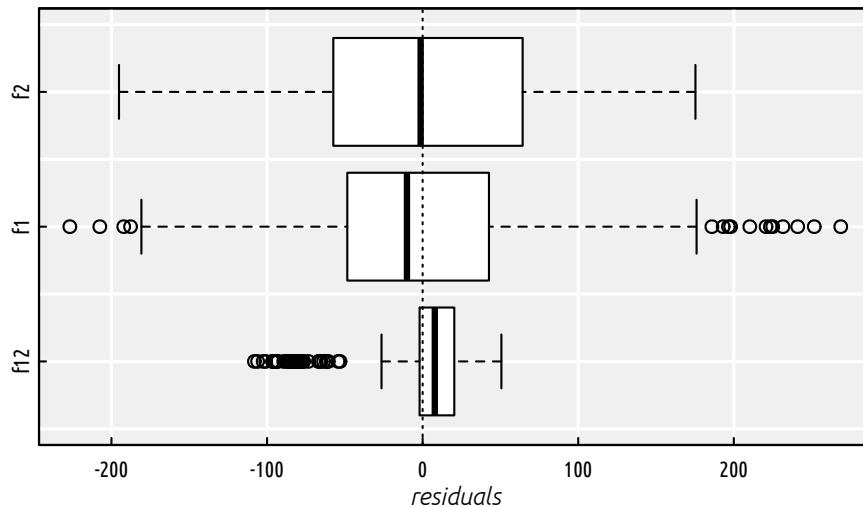


Figure 2.6: Box plots of the residuals for the three models studied

```
list(f12=f12$residuals, f1=f1$residuals, f2=f2$residuals))
abline(v=0, lty=3)
```

We can also take a look at the absolute values of the residuals. Here are some descriptive statistics:

```
summary(abs(f12$residuals))

##      Min.    1st Qu.     Median      Mean    3rd Qu.      Max.
##  0.06457   6.46397  14.07055  22.86342  26.41772 108.09995

summary(abs(f1$residuals))

##      Min.    1st Qu.     Median      Mean    3rd Qu.      Max.
##  0.5056   19.6640   45.0716  61.4889  80.1239 268.7377

summary(abs(f2$residuals))

##      Min.    1st Qu.     Median      Mean    3rd Qu.      Max.
##  0.6545   29.8540   59.6756  64.1506  95.7384 195.1557
```

Figure 2.8 is worth \$1000:

```
boxplot(horizontal=TRUE, col="white", xlab="abs(residuals)",
        list(f12=abs(f12$residuals), f1=abs(f1$residuals),
             f2=abs(f2$residuals)))
abline(v=0, lty=3)
```

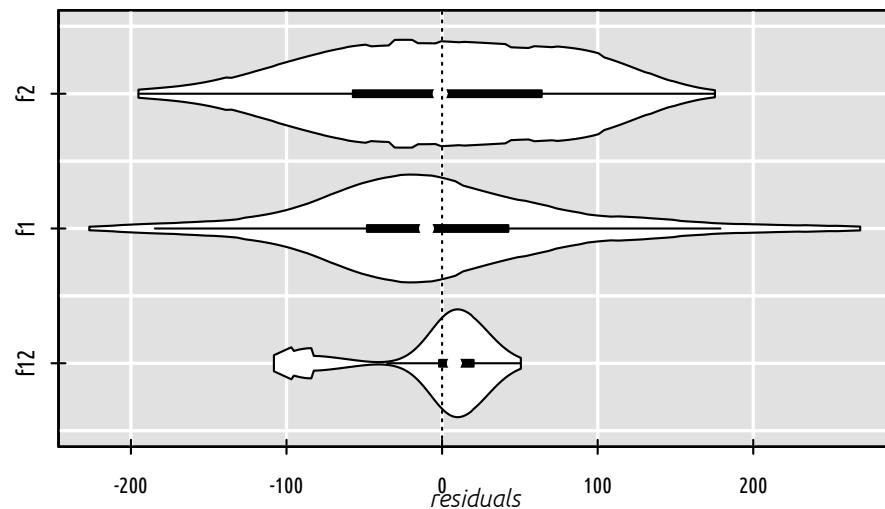


Figure 2.7: Violin plots of the residuals for the three models studied

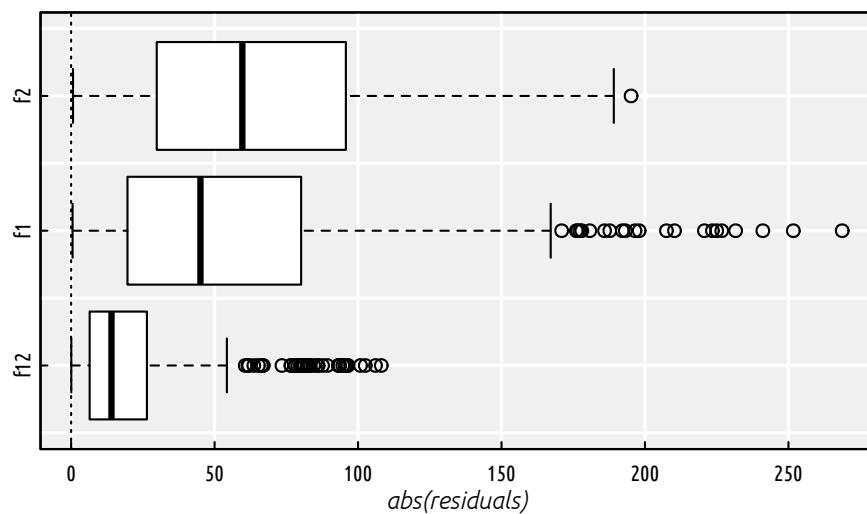


Figure 2.8: Box plots of the modules of the residuals for the three models studied

### 2.3.1.3 Coefficient of Determination (R-squared)

If we didn't know the range of the dependent variable (in our case we do know that the credit rating is on the scale 0–1000), the RMSE or MAE would be hard to interpret.

It turns out that there is a popular *normalised* (unit-less) measure that is somehow easy to interpret with no domain-specific knowledge of the modelled problem. Namely, the (unadjusted)  $R^2$  **score** (the coefficient of determination) is given by:

$$R^2(f) = 1 - \frac{\sum_{i=1}^n (y_i - f(\mathbf{x}_{i,\cdot}))^2}{\sum_{i=1}^n (y_i - \bar{y})^2},$$

where  $\bar{y}$  is the arithmetic mean  $\frac{1}{n} \sum_{i=1}^n y_i$ .

```
(r12 <- summary(f12)$r.squared)

## [1] 0.9390901
1 - sum(f12$residuals^2)/sum((Y-mean(Y))^2) # the same

## [1] 0.9390901
(r1 <- summary(f1)$r.squared)

## [1] 0.6375085
(r2 <- summary(f2)$r.squared)

## [1] 0.6899812
```

The coefficient of determination gives the proportion of variance of the dependent variable explained by independent variables in the model;  $R^2(f) \simeq 1$  indicates a perfect fit. The first model is a very good one, the simple models are “more or less okay”.

Unfortunately,  $R^2$  tends to automatically increase as the number of independent variables increase (recall that the more variables in the model, the better the SSR must be). To correct for this phenomenon, we sometimes consider the **adjusted R<sup>2</sup>**:

$$\bar{R}^2(f) = 1 - (1 - R^2(f)) \frac{n - 1}{n - p - 1}$$

```
summary(f12)$adj.r.squared

## [1] 0.9386933
n <- length(x); 1 - (1 - r12)*(n-1)/(n-3) # the same

## [1] 0.9386933
```

```
summary(f1)$adj.r.squared
## [1] 0.6363316
summary(f2)$adj.r.squared
## [1] 0.6889747
```

In other words, the adjusted  $R^2$  penalises for more complex models.

**Remark.** (\*) Side note – results of some statistical tests (e.g., significance of coefficients) are reported by calling `summary(f12)` etc. — refer to a more advanced source to obtain more information. These, however, require the verification of some assumptions regarding the input data and the residuals.

```
summary(f12)
##
## Call:
## lm(formula = Y ~ X1 + X2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -108.100   -1.940    7.812   20.249   50.623
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.726e+02  3.950e+00   43.69   <2e-16 ***
## X1          1.828e-01  5.159e-03   35.43   <2e-16 ***
## X2          2.198e+00  5.637e-02   38.99   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 34.16 on 307 degrees of freedom
## Multiple R-squared:  0.9391, Adjusted R-squared:  0.9387
## F-statistic: 2367 on 2 and 307 DF, p-value: < 2.2e-16
```

### 2.3.1.4 Residuals vs. Fitted Plot

We can also create scatter plots of the residuals (predicted  $\hat{y}_i$  minus true  $y_i$ ) as a function of the predicted  $\hat{y}_i = f(\mathbf{x}_{i,\cdot})$ , see Figure 2.9.

```
Y_pred12 <- predict(f12, data.frame(X1, X2))
Y_pred1  <- predict(f1, data.frame(X1))
Y_pred2  <- predict(f2, data.frame(X2))
par(mfrow=c(1, 3))
plot(Y_pred12, Y_pred12-Y)
plot(Y_pred1,  Y_pred1 -Y)
```

```
plot(Y_pred2, Y_pred2 - Y)
```

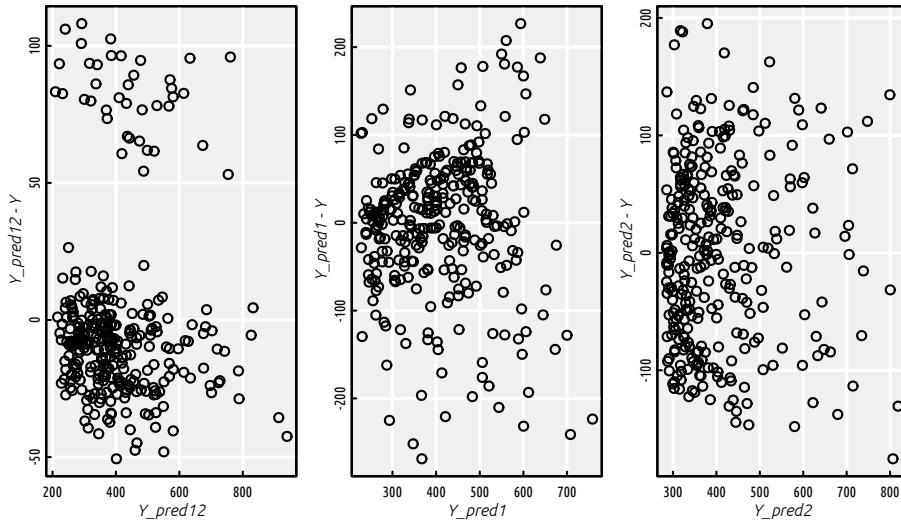


Figure 2.9: Residuals vs. fitted outputs for the three regression models

Ideally (provided that the hypothesis that the dependent variable is indeed a linear function of the dependent variable(s) is true), we would expect to see a point cloud that spread around 0 in a very much unorderly fashion.

### 2.3.2 Variable Selection

Okay, up to now we've been considering the problem of modelling the `Rating` variable as a function of `Balance` and/or `Income`. However, in the `Credit` data set there are other variables possibly worth inspecting.

Consider all quantitative (numeric-continuous) variables in the `Credit` data set.

```
C <- Credit[Credit$Balance>0,
  c("Rating", "Limit", "Income", "Age",
    "Education", "Balance")]
head(C)

##   Rating Limit Income Age Education Balance
## 1    283   3606  14.891  34        11     333
## 2    483   6645 106.025  82        15     903
## 3    514   7075 104.593  71        11      580
## 4    681   9504 148.924  36        11     964
## 5    357   4897  55.882  68        16     331
## 6    569   8047  80.180  77        10    1151
```

Obviously there are many possible combinations of the variables upon which regression models can be constructed (precisely, for  $p$  variables there are  $2^p$  such models). How do we choose the *best* set of inputs?

**Remark.** We should already be suspicious at this point: wait... *best* requires some sort of criterion, right?

First, however, let's draw a matrix of scatter plots for every pair of variables – so as to get an impression of how individual variables interact with each other, see Figure 2.10.

`pairs(C)`

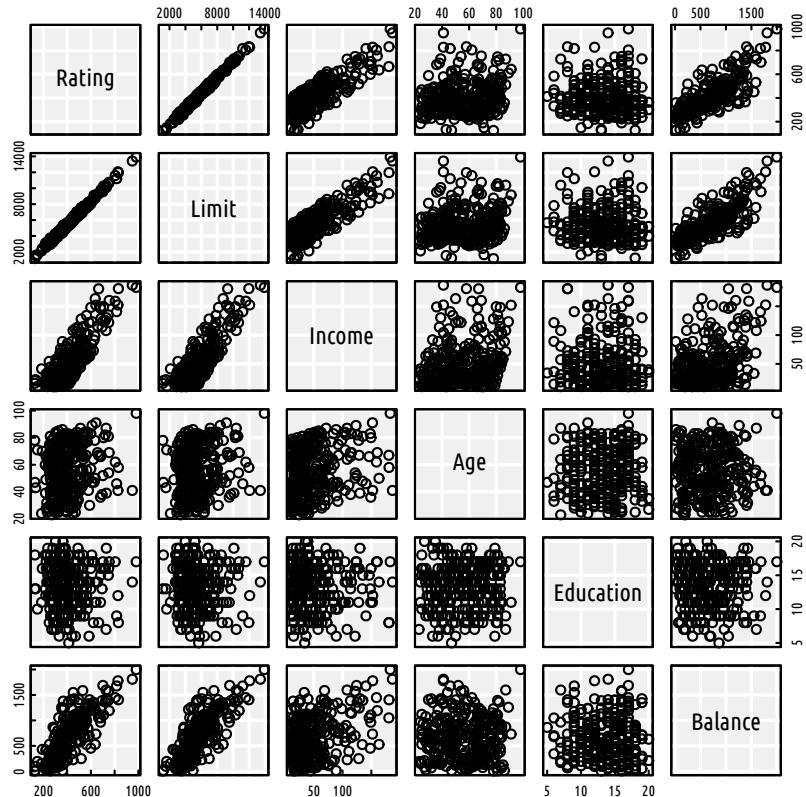


Figure 2.10: Scatter plot matrix for the Credit dataset

It seems like **Rating** depends on **Limit** almost linearly... We have a tool to actually quantify the degree of linear dependence between a pair of variables –

Pearson's  $r$  – the linear correlation coefficient:

$$r(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}.$$

It holds  $r \in [-1, 1]$ , where:

- $r = 1$  – positive linear dependence ( $y$  increases as  $x$  increases)
- $r = -1$  – negative linear dependence ( $y$  decreases as  $x$  increases)
- $r \simeq 0$  – uncorrelated or non-linearly dependent

Figure 2.11 gives an illustration of the above.

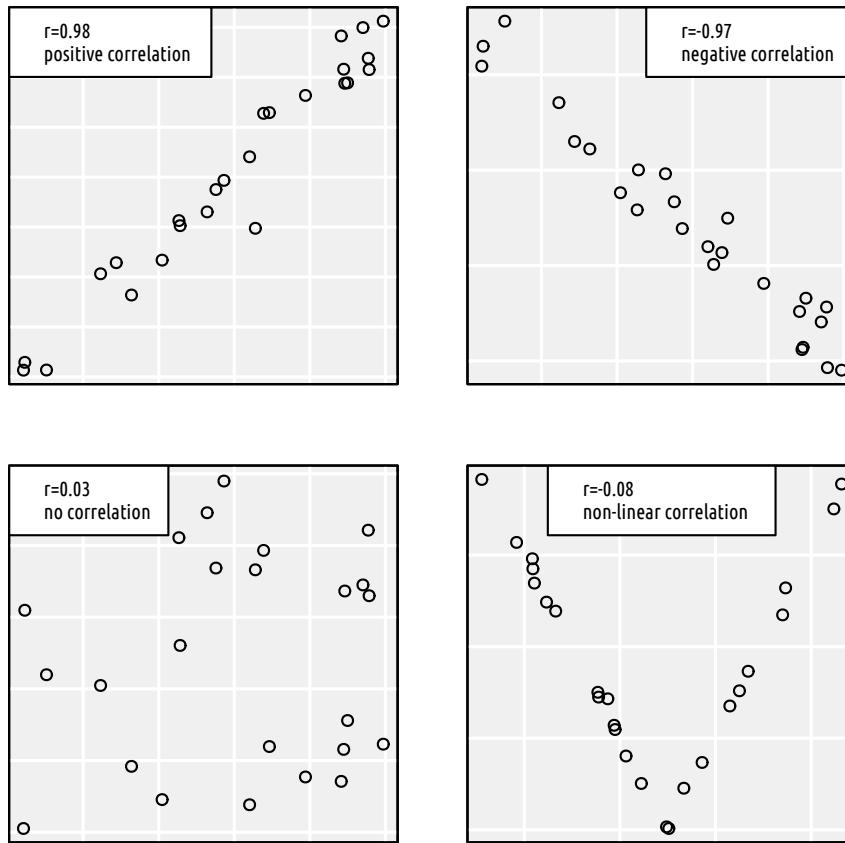


Figure 2.11: Different datasets and the corresponding Pearson's  $r$  coefficients

To compute Pearson's  $r$  between all pairs of variables, we call:

```
round(cor(C), 3)
```

	Rating	Limit	Income	Age	Education	Balance
## Rating	1.000	0.996	0.831	0.167	-0.040	0.798
## Limit	0.996	1.000	0.834	0.164	-0.032	0.796
## Income	0.831	0.834	1.000	0.227	-0.033	0.414
## Age	0.167	0.164	0.227	1.000	0.024	0.008
## Education	-0.040	-0.032	-0.033	0.024	1.000	0.001
## Balance	0.798	0.796	0.414	0.008	0.001	1.000

`Rating` and `Limit` are almost perfectly linearly correlated, and both seem to describe the same thing.

For practical purposes, we'd rather model `Rating` as a function of the other variables. For simple linear regression models, we'd choose either `Income` or `Balance`. How about multiple regression though?

The best model:

- has high predictive power,
- is simple.

These two criteria are often mutually exclusive.

Which variables should be included in the optimal model?

Again, the definition of the “best” object needs a *fitness* function.

For fitting a single model to data, we use the SSR.

We need a metric that takes the number of dependent variables into account.

**Remark.** (\*) Unfortunately, the adjusted  $R^2$ , despite its interpretability, is not really suitable for this task. It does not penalise complex models heavily enough to be really useful.

Here we'll be using the **Akaike Information Criterion** (AIC).

For a model  $f$  with  $p'$  independent variables:

$$\text{AIC}(f) = 2(p' + 1) + n \log(\text{SSR}(f)) - n \log n$$

Our task is to find the combination of independent variables that minimises the AIC.

**Remark.** (\*\*) Note that this is a bi-level optimisation problem – for every considered combination of variables (which we look for), we must solve another problem of finding the best model involving these variables – the

one that minimises the SSR.

$$\min_{s_1, s_2, \dots, s_p \in \{0,1\}} \left( 2 \left( \sum_{j=1}^p s_j + 1 \right) + n \log \left( \min_{\beta_0, \beta_1, \dots, \beta_p \in \mathbb{R}} \sum_{i=1}^n (\beta_0 + s_1 \beta_1 x_{i,1} + \dots + s_p \beta_p x_{i,p} - y_i)^2 \right) \right)$$

We dropped the  $n \log n$  term, because it is always constant and hence doesn't affect the solution. If  $s_j = 0$ , then the  $s_j \beta_j x_{i,j}$  term is equal to 0, and hence is not considered in the model. This plays the role of including  $s_j = 1$  or omitting  $s_j = 0$  the  $j$ -th variable in the model building exercise.

For  $p$  variables, the number of their possible combinations is equal to  $2^p$  (grows exponentially with  $p$ ). For large  $p$  (think big data), an extensive search is impractical (in our case we could get away with this though – left as an exercise to a slightly more advanced reader). Therefore, to find the variable combination minimising the AIC, we often rely on one of the two following greedy heuristics:

- forward selection:

1. start with an empty model
2. find an independent variable whose addition to the current model would yield the highest decrease in the AIC and add it to the model
3. go to step 2 until AIC decreases

- backward elimination:

1. start with the full model
2. find an independent variable whose removal from the current model would decrease the AIC the most and eliminate it from the model
3. go to step 2 until AIC decreases

**Remark.** (\*\*\*) The above bi-level optimisation problem can be solved by implementing a genetic algorithm – see further chapter for more details.

**Remark.** (\*) There are of course many other methods which also perform some form of variable selection, e.g., lasso regression. But these minimise a different objective.

Forward selection example:

```
C <- Credit[Credit$Balance>0,
  c("Rating", "Income", "Age",
    "Education", "Balance")]
step(lm(Rating~1, data=C), # empty model
  scope=formula(lm(Rating~., data=C)), # full model
  direction="forward")

## Start: AIC=3055.75
## Rating ~ 1
```

```

##  

##          Df Sum of Sq      RSS      AIC  

## + Income     1  4058342 1823473 2694.7  

## + Balance    1   3749707 2132108 2743.2  

## + Age        1    164567 5717248 3048.9  

## <none>           5881815 3055.8  

## + Education  1      9631 5872184 3057.2  

##  

## Step:  AIC=2694.7  

## Rating ~ Income  

##  

##          Df Sum of Sq      RSS      AIC  

## + Balance    1  1465212 358261 2192.3  

## <none>           1823473 2694.7  

## + Age        1      2836 1820637 2696.2  

## + Education  1      1063 1822410 2696.5  

##  

## Step:  AIC=2192.26  

## Rating ~ Income + Balance  

##  

##          Df Sum of Sq      RSS      AIC  

## + Age         1     4119.1 354141 2190.7  

## + Education   1     2692.1 355568 2191.9  

## <none>           358261 2192.3  

##  

## Step:  AIC=2190.67  

## Rating ~ Income + Balance + Age  

##  

##          Df Sum of Sq      RSS      AIC  

## + Education   1     2925.7 351216 2190.1  

## <none>           354141 2190.7  

##  

## Step:  AIC=2190.1  

## Rating ~ Income + Balance + Age + Education  

##  

## Call:  

## lm(formula = Rating ~ Income + Balance + Age + Education, data = C)  

##  

## Coefficients:  

## (Intercept)      Income       Balance       Age  

## 173.8300       2.1668       0.1839      0.2234  

## Education  

## -0.9601

```

The full model has been selected.

Backward elimination example:

```
step(lm(Rating~., data=C), # full model
     scope=formula(lm(Rating~1, data=C)), # empty model
     direction="backward")

## Start: AIC=2190.1
## Rating ~ Income + Age + Education + Balance
##
##          Df Sum of Sq    RSS    AIC
## <none>            351216 2190.1
## - Education      1     2926  354141 2190.7
## - Age            1     4353  355568 2191.9
## - Balance         1   1468466 1819682 2698.1
## - Income          1   1617191 1968406 2722.4

##
## Call:
## lm(formula = Rating ~ Income + Age + Education + Balance, data = C)
##
## Coefficients:
## (Intercept)      Income        Age        Education
##           173.8300     2.1668     0.2234     -0.9601
## Balance
##           0.1839
```

The full model is considered the best.

Forward selection example – full dataset:

```
C <- Credit[, # do not restrict to Credit$Balance>0
             c("Rating", "Income", "Age",
               "Education", "Balance")]
step(lm(Rating~1, data=C), # empty model
     scope=formula(lm(Rating~., data=C)), # full model
     direction="forward")

## Start: AIC=4034.31
## Rating ~ 1
##
##          Df Sum of Sq    RSS    AIC
## + Balance      1   7124258 2427627 3488.4
## + Income       1   5982140 3569744 3642.6
## + Age          1   101661  9450224 4032.0
## <none>          9551885 4034.3
## + Education    1     8675  9543210 4036.0
##
## Step: AIC=3488.38
## Rating ~ Balance
```

```

##          Df Sum of Sq    RSS    AIC
## + Income     1   1859749  567878 2909.3
## + Age        1      98562 2329065 3473.8
## <none>           2427627 3488.4
## + Education  1      5130 2422497 3489.5
##
## Step:  AIC=2909.28
## Rating ~ Balance + Income
##
##          Df Sum of Sq    RSS    AIC
## <none>           567878 2909.3
## + Age        1     2142.4 565735 2909.8
## + Education  1     1208.6 566669 2910.4
##
## Call:
## lm(formula = Rating ~ Balance + Income, data = C)
##
## Coefficients:
## (Intercept)      Balance       Income
## 145.3506        0.2129        2.1863

```

This procedure suggests including only the `Balance` and `Income` variables.

Backward elimination example – full dataset:

```

step(lm(Rating~., data=C), # full model
     scope=formula(lm(Rating~1, data=C)), # empty model
     direction="backward")

## Start:  AIC=2910.89
## Rating ~ Income + Age + Education + Balance
##
##          Df Sum of Sq    RSS    AIC
## - Education  1      1238 565735 2909.8
## - Age        1      2172 566669 2910.4
## <none>           564497 2910.9
## - Income     1    1759273 2323770 3474.9
## - Balance    1    2992164 3556661 3645.1
##
## Step:  AIC=2909.77
## Rating ~ Income + Age + Balance
##
##          Df Sum of Sq    RSS    AIC
## - Age        1      2142 567878 2909.3
## <none>           565735 2909.8
## - Income    1    1763329 2329065 3473.8

```

```

## - Balance 1 2991523 3557259 3643.2
##
## Step: AIC=2909.28
## Rating ~ Income + Balance
##
##          Df Sum of Sq      RSS      AIC
## <none>            567878 2909.3
## - Income    1   1859749 2427627 3488.4
## - Balance   1   3001866 3569744 3642.6
##
## Call:
## lm(formula = Rating ~ Income + Balance, data = C)
##
## Coefficients:
## (Intercept)      Income      Balance
##       145.3506     2.1863      0.2129

```

This procedure gives the same results as forward selection (however, for other data sets that is not necessarily the case).

### 2.3.3 Variable Transformation

So far we have been fitting linear models of the form:

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p.$$

What about some non-linear models such as polynomials etc.? For example:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_1^2 + \beta_3 X_1^3 + \beta_4 X_2.$$

Solution: pre-process inputs by setting  $X'_1 := X_1$ ,  $X'_2 := X_1^2$ ,  $X'_3 := X_1^3$ ,  $X'_4 := X_2$  and fit a linear model:

$$Y = \beta_0 + \beta_1 X'_1 + \beta_2 X'_2 + \beta_3 X'_3 + \beta_4 X'_4.$$

This trick works for every model of the form  $Y = \sum_{i=1}^k \sum_{j=1}^p \varphi_{i,j}(X_j)$  for any  $k$  and any univariate functions  $\varphi_{i,j}$ .

Also, with a little creativity (and maths), we might be able to transform a few other models to a linear one, e.g.,

$$Y = b e^{aX} \quad \rightarrow \quad \log Y = \log b + aX \quad \rightarrow \quad Y' = aX + b'$$

This is an example of a model's **linearisation**. However, not every model can be linearised. In particular, one that involves functions that are not invertible.

For example, here's a series of simple ( $p = 1$ ) degree- $d$  polynomial regression models of the form:

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \cdots + \beta_d X^d.$$

Such models can be fitted with the `lm()` function based on the formula of the form `Y~poly(X, d)` or `Y~X+I(X^2)+I(X^3)+...`

```
f1_1 <- lm(Y~X1)
f1_3 <- lm(Y~X1+I(X1^2)+I(X1^3)) # also: Y~poly(X1, 3)
f1_14 <- lm(Y~poly(X1, 14))
```

Above we have fitted the polynomials of degrees 1, 3 and 14. Note that a polynomial of degree 1 is just a line.

Let us depict the three models:

```
plot(X1, Y, col="#000000aa")
x <- seq(min(X1), max(X1), length.out=101)
lines(x, predict(f1_1, data.frame(X1=x)), col="red", lwd=3)
lines(x, predict(f1_3, data.frame(X1=x)), col="blue", lwd=3)
lines(x, predict(f1_14, data.frame(X1=x)), col="darkgreen", lwd=3)
```

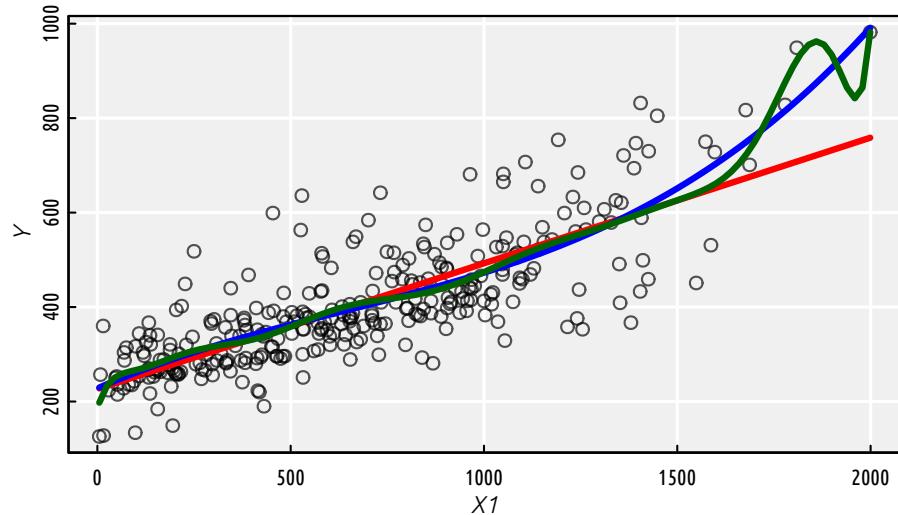


Figure 2.12: Polynomials of different degrees fitted to the Credit dataset

From Figure 2.12 we see that there's clearly a problem with the degree-14 polynomial.

### 2.3.4 Predictive vs. Descriptive Power

The above high-degree polynomial model (`f1_14`) is a typical instance of a phenomenon called an **overfit**.

Clearly (based on our expert knowledge), the `Rating` shouldn't decrease as `Balance` increases.

In other words, `f1_14` gives a better fit to data actually observed, but fails to produce good results for the points that are yet to come.

We say that it **generalises** poorly to unseen data.

Assume our true model is of the form:

```
true_model <- function(x) 3*x^3+5
```

Let's generate the following random sample from this model (with  $Y$  subject to error), see Figure 2.13:

```
n <- 25
X <- runif(n, min=0, max=1)
Y <- true_model(X)+rnorm(n, sd=0.1) # add normally-distributed noise

set.seed(123) # to assure reproducibility
plot(X, Y)
x <- seq(0, 1, length.out=101)
lines(x, true_model(x), col=2, lwd=3, lty=2)
```

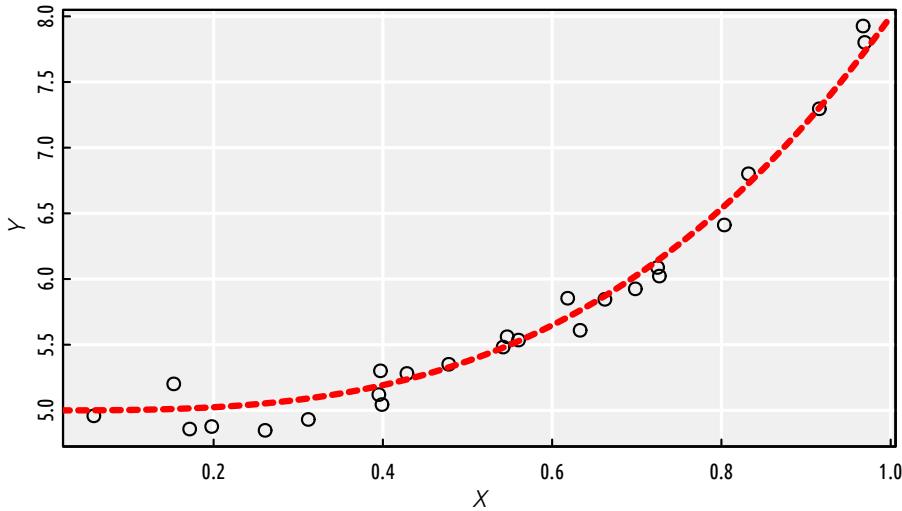


Figure 2.13: Synthetic data generated by means of the formula  $Y = 3x^3 + 5$  (+ noise)

Let's fit polynomials of different degrees, see Figure 2.14.

```
plot(X, Y)
lines(x, true_model(x), col=2, lwd=3, lty=2)

dmax <- 15 # maximal polynomial degree
MSE_train <- numeric(dmax)
MSE_test <- numeric(dmax)
for (d in 1:dmax) { # for every polynomial degree
  f <- lm(Y~poly(X, d)) # fit a d-degree polynomial
  y <- predict(f, data.frame(X=x))
  lines(x, y, col=d)
  # MSE on given random X,Y:
  MSE_train[d] <- mean(f$residuals^2)
  # MSE on many more points:
  MSE_test[d] <- mean((y-true_model(x))^2)
}
}
```

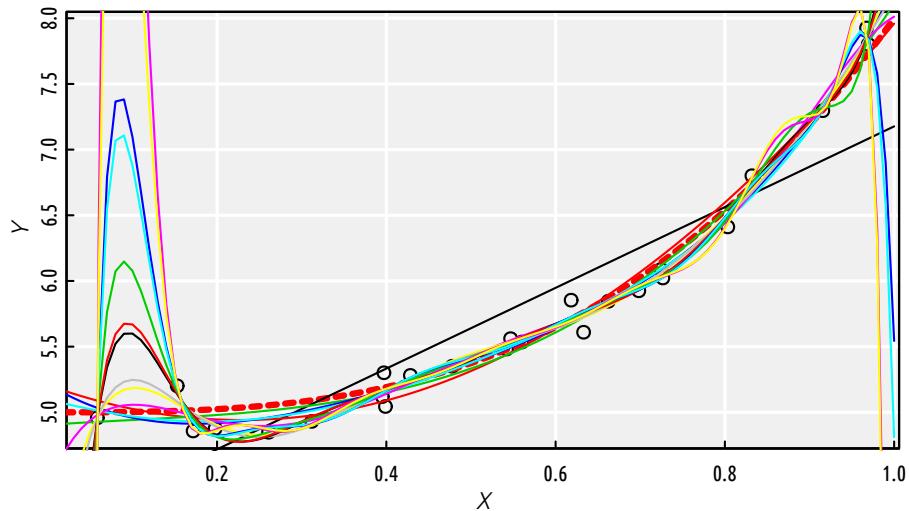


Figure 2.14: Polynomials fitted to our synthetic dataset

Some of the polynomials are fitted too well!

**Remark (\*)** The oscillation of the high-degree polynomials at the domain boundaries is known as the Runge phenomenon.

Compare the mean squared error (MSE) for the observed vs. future data points, see Figure 2.15.

```
matplot(1:dmax, cbind(MSE_train, MSE_test), type="b",
       ylim=c(1e-3, 2e3), log="y", pch=1:2,
```

```

xlab="Model complexity (polynomial degree)",
ylab="MSE")
legend("topleft", legend=c("MSE on original data", "MSE on the whole range"),
lty=1:2, col=1:2, pch=1:2, bg="white")

```

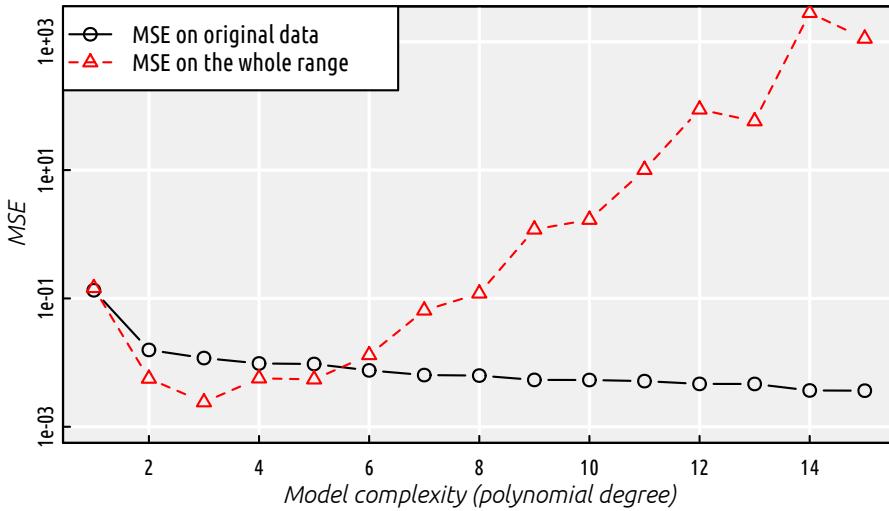


Figure 2.15: MSE on the dataset used to construct the model vs. MSE on a whole range of points as function of the polynomial degree

Note the logarithmic scale on the  $y$  axis.

This is a very typical behaviour!

- A model's fit to observed data improves as the model's complexity increases.
- A model's generalisation to unseen data initially improves, but then becomes worse.
- In the above example, the sweet spot is at a polynomial of degree 3, which is exactly our true underlying model.

Hence, most often we should be interested in the accuracy of the predictions made in the case of unobserved data.

If we have a data set of a considerable size, we can divide it (randomly) into two parts:

- *training sample* (say, 60% or 80%) – used to fit a model
- *test sample* (the remaining 40% or 20%) – used to assess its quality (e.g., using MSE)

More on this issue in the chapter on Classification.

**Remark.** (\*) We shall see that sometimes a train-test-validate split will be necessary, e.g., 60-20-20%.

## 2.4 Exercises in R

### 2.4.1 Anscombe's Quartet Revisited

Consider the `anscombe` database once again:

```
print(anscombe) # `anscombe` is a built-in object
```

```
##   x1  x2  x3  x4    y1    y2    y3    y4
## 1 10  10  10   8  8.04  9.14  7.46  6.58
## 2   8   8   8   8  6.95  8.14  6.77  5.76
## 3 13  13  13   8  7.58  8.74 12.74  7.71
## 4   9   9   9   8  8.81  8.77  7.11  8.84
## 5 11  11  11   8  8.33  9.26  7.81  8.47
## 6 14  14  14   8  9.96  8.10  8.84  7.04
## 7   6   6   6   8  7.24  6.13  6.08  5.25
## 8   4   4   4  19  4.26  3.10  5.39 12.50
## 9 12  12  12   8 10.84  9.13  8.15  5.56
## 10  7   7   7   8  4.82  7.26  6.42  7.91
## 11  5   5   5   8  5.68  4.74  5.73  6.89
```

Recall that in the previous Chapter we have split the above data into four data frames `ans1`, ..., `ans4` with columns `x` and `y`.

#### Exercise.

*In `ans1`, fit a regression line to the data set as-is.*

□

#### Solution.

We've done that already, see Figure 2.16. What a wonderful exercise, thank you – effective learning is often done by repeating stuff.

```
ans1 <- data.frame(x=anscombe$x1, y=anscombe$y1)
f1 <- lm(y~x, data=ans1)
plot(ans1$x, ans1$y)
abline(f1, col="red")
```

■

#### Exercise.

*In `ans2`, fit a quadratic model ( $y = a + bx + cx^2$ ).*

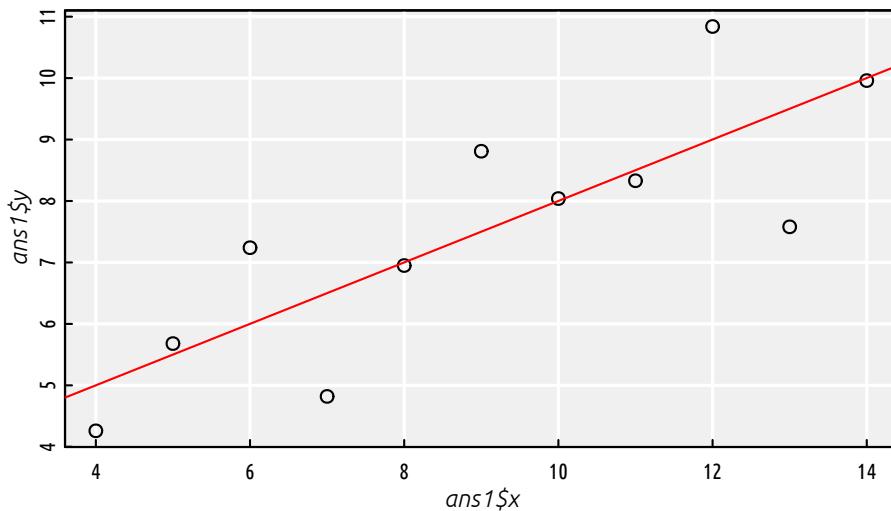


Figure 2.16: Fitted regression line for ans1

□

**Solution.**

How to fit a polynomial model is explained above.

```
ans2 <- data.frame(x=anscombe$x2, y=anscombe$y2)
f2 <- lm(y~x+I(x^2), data=ans2)
plot(ans2$x, ans2$y)
x_plot <- seq(4, 14, by=0.1)
y_plot <- predict(f2, data.frame(x=x_plot))
lines(x_plot, y_plot, col="red")
```

*Comment:* From Figure 2.17 we see that it's an almost-perfect fit! Clearly, the second Anscombe dataset isn't a case of linearly dependent variables.

■

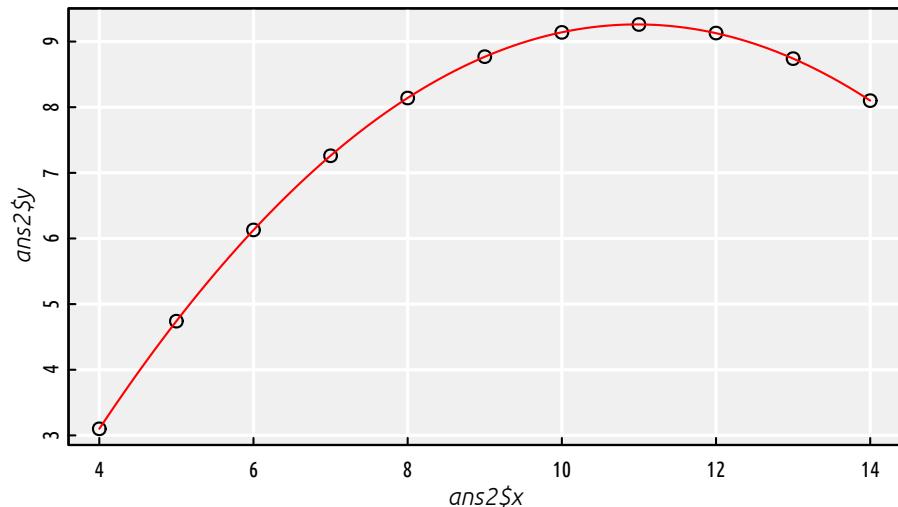
**Exercise.**

In *ans3*, remove the obvious outlier from data and fit a regression line.

□

**Solution.**

Let's plot the data set first, see Figure 2.18.

Figure 2.17: Fitted quadratic model for `ans2`

```
ans3 <- data.frame(x=anscombe$x3, y=anscombe$y3)
plot(ans3$x, ans3$y)
```

Indeed, the observation at  $x \approx 13$  is an obvious outlier. Perhaps the easiest way to remove it is to call:

```
ans3b <- ans3[ans3$y<=12,] # the outlier is definitely at y>12
```

We could also use the condition `y < max(y)`, amongst others.

Now let's fit the linear model:

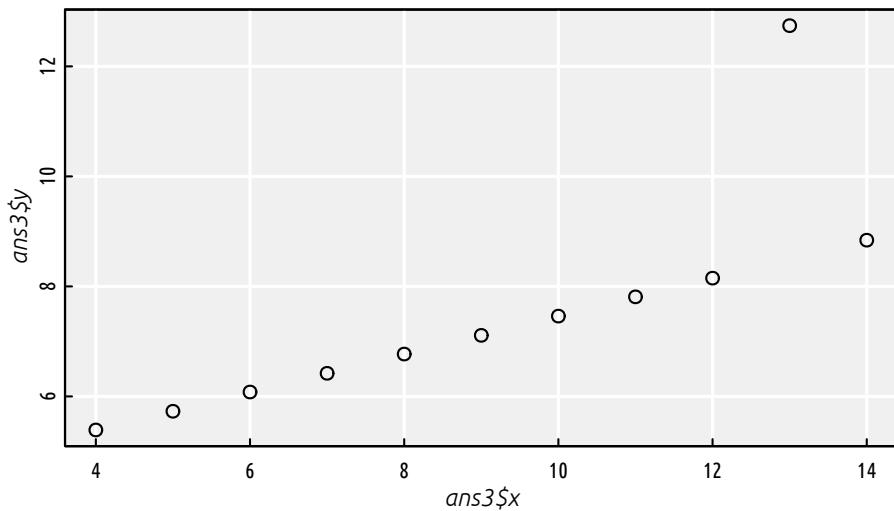
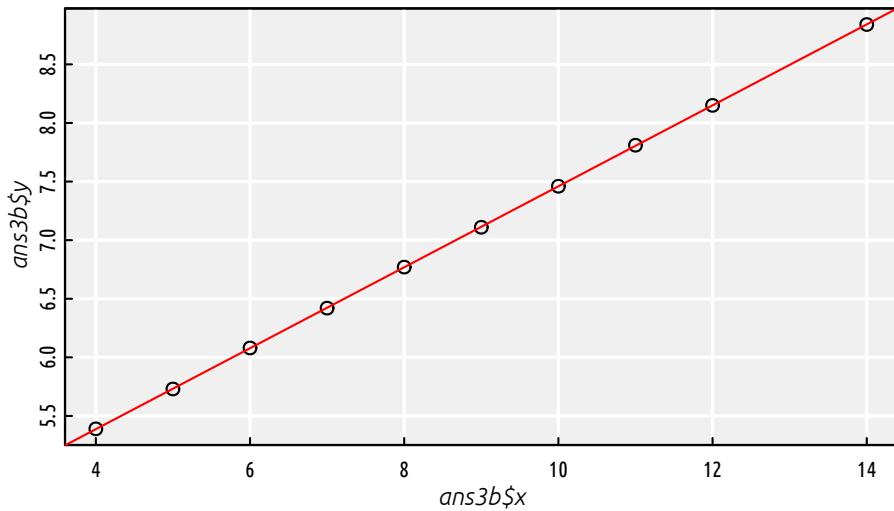
```
f3b <- lm(y~x, data=ans3b)
plot(ans3b$x, ans3b$y)
abline(f3b, col="red")
```

*Comment: Now Figure 2.19 is what we call linearly correlated data. By the way, Pearson's coefficient now equals 0.9999966.*

■

### 2.4.2 Countries of the World – Simple models involving the GDP per capita

Let's consider the World Factbook 2020 dataset (see this book's `datasets` folder). It consists of country names, their population, area, GDP, mortality rates etc. We have scraped it from the CIA website at <https://www.cia.gov/library/publications/the-world-factbook/docs/rankorderguide.html> and compiled into a single file on 3 April 2020.

Figure 2.18: Scatter plot for `ans3`Figure 2.19: Scatter plot for `ans3` with the outlier removed and the fitted linear model

```
factbook <- read.csv("datasets/world_factbook_2020.csv",
  comment="#" , stringsAsFactors=FALSE)
```

Here is a preview of a few features for 3 selected countries (see `help("%in%")`):

```
factbook[factbook$country %in%
  c("Australia", "New Zealand", "United States"),
  c("country", "area", "population", "gdp_per_capita_ppp")]

##           country     area population gdp_per_capita_ppp
## 15      Australia 7741220    25466459          50400
## 169    New Zealand 268838     4925477          39000
## 247 United States 9833517   332639102          59800
```

### Exercise.

*List the 10 countries with the highest GDP per capita.*



### Solution.

To recall, to generate a list of indexes that produce an ordered version of a numeric vector, we need to call the `order()` function.

```
which_top <- tail(order(factbook$gdp_per_capita_ppp, na.last=FALSE), 10)
factbook[which_top, c("country", "gdp_per_capita_ppp")]

##           country gdp_per_capita_ppp
## 113      Ireland        73200
## 35       Brunei        78900
## 114    Isle of Man      84600
## 211    Singapore      94100
## 26      Bermuda      99400
## 141 Luxembourg     105100
## 157      Monaco     115700
## 142      Macau     122000
## 192      Qatar     124100
## 139 Liechtenstein  139100
```

By the way, the reported values are in USD.

*Question: Which of these countries are tax havens?*



### Exercise.

*Find the 5 most positively and the 5 most negatively correlated variables with the `gdp_per_capita_ppp` feature (of course, with respect to the Pearson coefficient).*

□

**Solution.**

This can be solved via a call to `cor()`. Note that we need to make sure that missing values are omitted from computations. A quick glimpse at the manual page (`?cor`) reveals that computing the correlation between a column and all the other ones (of course, except `country`, which is non-numeric) can be performed as follows.

```
r <- cor(factbook$gdp_per_capita_ppp,
          factbook[, !(names(factbook) %in% c("country", "gdp_per_capita_ppp"))],
          use="complete.obs")[1,]
or <- order(r) # ordering permutation (indexes)
r[head(or, 5)] # first 5 ordered indexes

## infant_mortality_rate maternal_mortality_rate
##           -0.7465825      -0.6700486
## birth_rate             death_rate
##           -0.6082201      -0.5721644
## total_fertility_rate
##           -0.5672550

r[tail(or, 5)] # last 5 ordered indexes

## natural_gas_production gross_national_saving
##           0.5689764      0.6113255
## median_age            obesity_adult_prevalence_rate
##           0.6208990      0.6368120
## life_expectancy_at_birth
##           0.7546062
```

*Comment:* “Live long and prosper” just gained a new meaning. Richer countries have lower infant and maternal mortality rates, lower birth rates, but higher life expectancy and obesity prevalence. Note, however, that correlation is not causation: we are unlikely to increase the GDP by asking people to put on weight.

■

**Exercise.**

Fit simple regression models where the per capita GDP explains its four most correlated variables (four individual models). Draw them on a scatter plot. Compute the root mean squared errors (RMSE), mean absolute errors (MAE) and the coefficients of determination ( $R^2$ ).

□

**Solution.**

The four most correlated variables (we should look at the absolute value of the correlation coefficient now – recall that it is the correlation of 0 that means no linear dependence; 1 and -1 show a strong association between a pair of variables) are:

```
(most_correlated <- names(r)[tail(order(abs(r)), 4)])  
  
## [1] "obesity_adult_prevalence_rate"  
## [2] "maternal_mortality_rate"  
## [3] "infant_mortality_rate"  
## [4] "life_expectancy_at_birth"
```

We could take the above column names and construct four formulas manually, e.g., by writing `gdp_per_capita_ppp~life_expectancy_at_birth`, but we are lazy. Being lazy when it comes to computer programming is often a virtue, not a flaw in one's character.

Instead, we will run a `for` loop that extracts the pairs of interesting columns and constructs a formula based on two vectors (`lm(Y~X)`), see Figure 2.20.

```
par(mfrow=c(2, 2)) # 4 plots on a 2x2 grid  
for (i in 1:4) {  
  print(most_correlated[i])  
  X <- factbook[, "gdp_per_capita_ppp"]  
  Y <- factbook[, most_correlated[i]]  
  f <- lm(Y~X)  
  print(cbind(RMSE=sqrt(mean(f$residuals^2)),  
             MAE=mean(abs(f$residuals)),  
             R2=summary(f)$r.squared))  
  plot(X, Y, xlab="gdp_per_capita_ppp",  
        ylab=most_correlated[i])  
  abline(f, col="red")  
}  
  
## [1] "obesity_adult_prevalence_rate"  
##      RMSE      MAE      R2  
## [1,] 11.04095 8.158911 0.06219638  
  
## [1] "maternal_mortality_rate"  
##      RMSE      MAE      R2  
## [1,] 204.9272 146.5324 0.2148087  
  
## [1] "infant_mortality_rate"  
##      RMSE      MAE      R2  
## [1,] 15.74646 12.1665 0.3005046  
  
## [1] "life_expectancy_at_birth"  
##      RMSE      MAE      R2  
## [1,] 5.429182 4.372718 0.4309554
```

Recall that the root mean squared error is the square root of the arithmetic mean of the squared residuals. Mean absolute error is the average of the absolute values of the residuals. The coefficient of determination is given by:  $R^2(f) = 1 - \frac{\sum_{i=1}^n (y_i - f(\mathbf{x}_i, \cdot))^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$ .

*Comment:* Unfortunately, we were misled by the high correlation coefficients between the  $X$ s and  $Y$ s: the low actual  $R^2$  scores indicate that these models should not be deemed trustworthy. Note that 3 of the plots are evidently L-shaped.

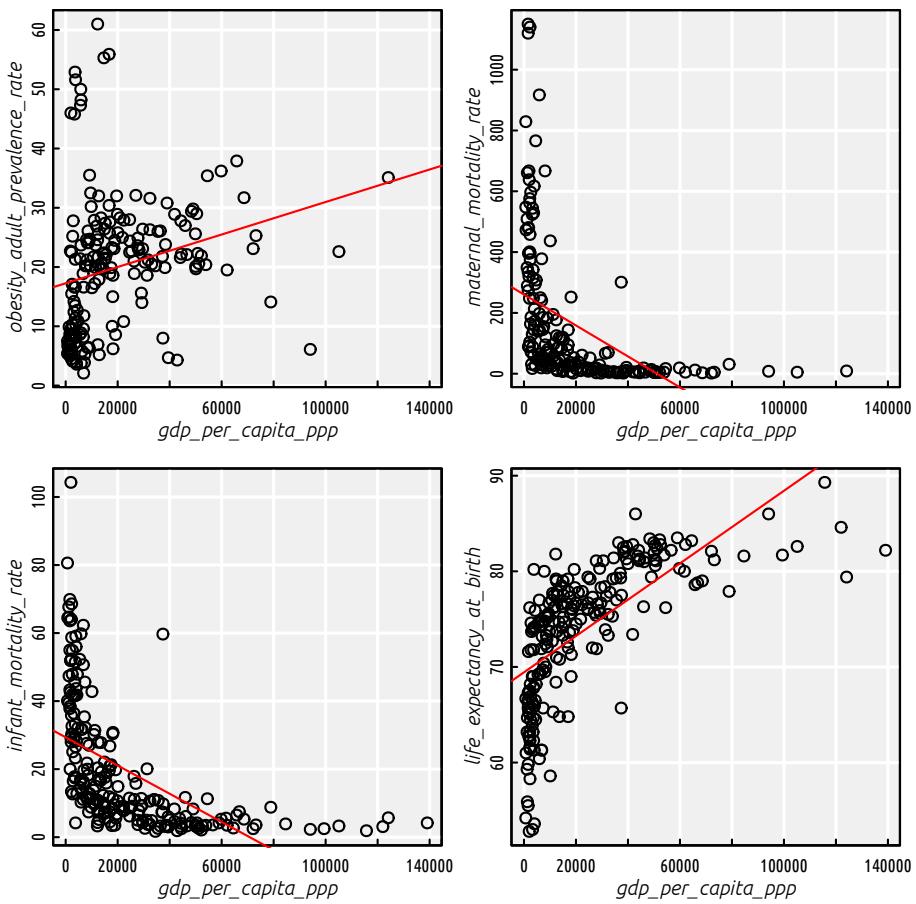


Figure 2.20: A scatter plot matrix and regression lines for the 4 variables most correlated with the per capita GDP

*Fun fact: (\*) Interestingly, it can be shown that  $R^2$  (in the case of the linear models fitted by minimising the SSR) is the square of the correlation between the true  $Y$ s and the predicted  $Y$ s:*

```
X <- factbook[, "gdp_per_capita_ppp"]
Y <- factbook[, most_correlated[i]]
f <- lm(Y~X, y=TRUE)
print(summary(f)$r.squared)

## [1] 0.4309554
print(cor(f$fitted.values, f$y)^2)

## [1] 0.4309554
```

*Side note: Do note that RMSE and MAE are interpretable: for instance, average error of life expectancy prediction based on the GDP is 4-5 years. Recall that you can find the information on the variables' units of measure at <https://www.cia.gov/library/publications/the-world-factbook/docs/rankorderguide.html>.*

■

### 2.4.3 Countries of the World – Most correlated variables (\*)

Let's get back to the World Factbook 2020 dataset (`world_factbook_2020.csv`).

```
factbook <- read.csv("datasets/world_factbook_2020.csv",
comment="#" , stringsAsFactors=FALSE)
```

#### Exercise.

Create a data frame  $C$  with three columns named  $col1$ ,  $col2$  and  $r$  and  $p(p-1)/2$  rows, where  $p$  is the number of numeric features in `factbook`. Every row should represent a unique pair of column names in `factbook` (we do not distinguish between  $a, b$  and  $b, a$ ) of correlation coefficients between them.

□

#### Solution.

First we will solve this exercise considering only 4 numeric features in our dataset, so that we can keep track of how the R expressions we evaluate actually work.

Let us compute the Pearson coefficients between chosen pairs of variables.

```
R <- cor(factbook[,c("area", "median_age", "birth_rate", "exports")],
use="complete.obs") # 4 selected columns
print(R)

##                      area median_age birth_rate   exports
## area      1.00000000  0.04452395 -0.0319946  0.4925857
```

```
## median_age  0.04452395  1.00000000 -0.9215918  0.2997270
## birth_rate -0.03199460 -0.92159182  1.0000000 -0.2429553
## exports      0.49258568  0.29972695 -0.2429553  1.0000000
```

Note that the R matrix has 1.0 on the diagonal (where each entry represents a correlation between a variable and itself). Moreover, it is symmetric around the diagonal –  $R[i, j] == R[j, i]$ , because it is the correlation between the same pair of variables. Hence, from now on we may be interested in the elements below the diagonal. We can get access to them by using `lower.tri()` (“lower triangle”).

```
R[lower.tri(R)]
```

```
## [1]  0.04452395 -0.03199460  0.49258568 -0.92159182  0.29972695
## [6] -0.24295529
```

This is already the 3rd column of the data frame we are asked to generate, which should look like:

```
##      col1      col2      r
## 1 median_age    area  0.04452395
## 2 birth_rate    area -0.03199460
## 3   exports    area  0.49258568
## 4 birth_rate median_age -0.92159182
## 5   exports median_age  0.29972695
## 6   exports birth_rate -0.24295529
```

How to generate `col1` and `col2`? One idea is to take the “lower triangles” of the following matrices:

```
##      [,1]      [,2]      [,3]      [,4]
## [1,] "area"     "area"     "area"     "area"
## [2,] "median_age" "median_age" "median_age" "median_age"
## [3,] "birth_rate"  "birth_rate" "birth_rate" "birth_rate"
## [4,] "exports"    "exports"   "exports"   "exports"
```

and:

```
##      [,1]      [,2]      [,3]      [,4]
## [1,] "area" "median_age" "birth_rate" "exports"
## [2,] "area" "median_age" "birth_rate" "exports"
## [3,] "area" "median_age" "birth_rate" "exports"
## [4,] "area" "median_age" "birth_rate" "exports"
```

Here is a complete solution for all the features in `factbook`:

```
R <- cor(factbook[,-1], use="complete.obs") # skip the `country` column
rrr <- matrix(dimnames(R)[[1]], nrow=nrow(R), ncol=ncol(R))
ccc <- matrix(dimnames(R)[[2]], byrow=TRUE, nrow=nrow(R), ncol=ncol(R))
C <- data.frame(col1=rrr[lower.tri(rrr)],
                 col2=ccc[lower.tri(ccc)],
                 r=R[lower.tri(R)])
```

*Comment:* In “classical” programming languages we would perhaps have used of a double (nested) `for` loop here (a less readable solution).

■

**Exercise.**

*Find the 5 most correlated pairs of variables.*

□

**Solution.**

This can be done by ordering the rows of `C` in decreasing order of absolute values of `C$r`, and then choosing the first 5 rows.

```
C_top <- head(C[order(abs(C$r), decreasing=TRUE),], 5)
knitr::kable(C_top)
```

	col1	col2	r
1687	electricity_installed_generating_capacity	electricity_production	0.9994159
1684	electricity_consumption	electricity_production	0.9992063
88	labor_force	population	0.9986190
1718	electricity_installed_generating_capacity	electricity_consumption	0.9981540
1300	telephones_mobile_cellular	labor_force	0.9979274

*Comment: The most correlated pairs of features are not really “mind-blowing”...*

■

**Exercise.**

*Fit simple regression models for the most correlated pair of variables.*

□

**Solution.**

There is a degree of ambiguity here: should `col1` or rather `col2` be treated as the dependent variable in our model? Let's do it either way.

To learn something new, which is exactly why we are all here, we will create the formulas programmatically, by first concatenating (joining) appropriate strings (note that in order to input a double quotes character, we need to proceed in with a backslash), and then calling the `formula()` function.

```
form <- formula(paste(C_top[1,2], " ~ ", C_top[1,1]))
f <- lm(form, data=factbook)
print(f)

##
## Call:
## lm(formula = form, data = factbook)
```

```

## 
## Coefficients:
##                               (Intercept)
##                               795225381
## electricity_installed_generating_capacity
##                               3625
plot(factbook[,C_top[1,1]], factbook[,C_top[1,2]],
      xlab=C_top[1,1], ylab=C_top[1,2])
abline(f, col="red")

```

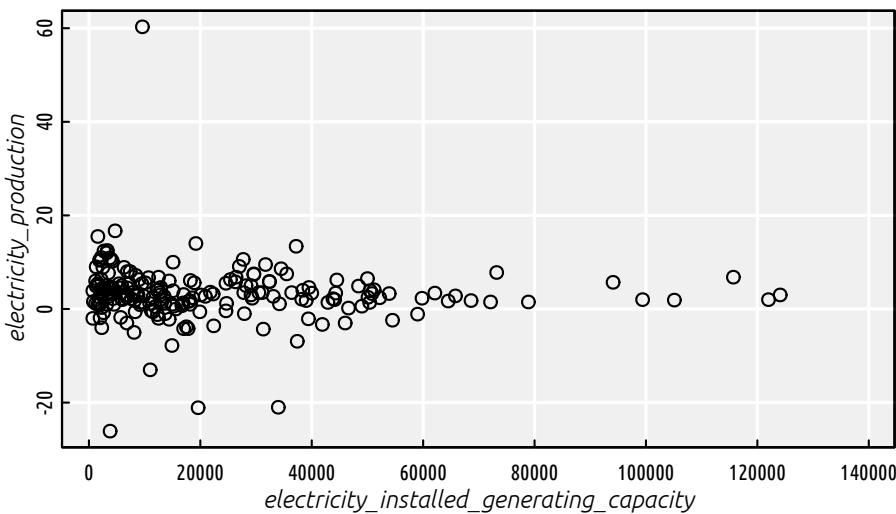


Figure 2.21: Most correlated pair of variables and the invisible regression line

Figure 2.21 is just great.

Wait, hold on a second! The fitted line is not present on the plot!

#### Exercise.

*There is a serious, extremely hard-to-detect bug in the above code. The reader is kindly asked to try to identify it.*



#### Solution.

The author wishes to apologise, but this where he will get slightly... emotional. We are all programmers here. So let's speak like a programmer to a programmer.

Aaaaaaaaaaaaaarg!!!!!!! I've spent 3 hours trying to locate it!!!

So in Appendix related to data frame wrangling I have explicitly asked everyone (including myself, because I tend to forget that!!!) to globally set the `stringsAsFactors=FALSE`

option. But we didn't do so. The problem is where we create the `C` matrix using the `data.frame()` function – `C$col1` and `C$col2` are objects of type:

```
class(C$col1)
## [1] "factor"
class(C$col2)
## [1] "factor"
```

Is this a problem? Well it is, because indexing an object `x` with a factor `f`, `x[f]` means `x[as.numeric(f)]` and not `x[as.character(f)]`.

Therefore, `factbook[,C_top[1,1]]` refers to:

```
names(factbook)[C_top[1,1]]
```

```
## [1] "gdp_per_capita_ppp"
```

and not:

```
as.character(C_top[1,1])
```

```
## [1] "electricity_installed_generating_capacity"
```

Here is a corrected version, see 2.22.

```
C <- data.frame(
  col1=rrr[lower.tri(rrr)],
  col2=ccc[lower.tri(ccc)],
  r=R[lower.tri(R)],
  stringsAsFactors=FALSE ##### !!!!!!!!!!!!!!!!
)
C_top <- head(C[order(abs(C$r), decreasing=TRUE),], 5)
form <- formula(paste(C_top[1,2], "~", C_top[1,1]))
f <- lm(form, data=factbook)
print(f)

##
## Call:
## lm(formula = form, data = factbook)
##
## Coefficients:
##                               (Intercept) 795225381
## electricity_installed_generating_capacity 3625
plot(factbook[,C_top[1,1]], factbook[,C_top[1,2]],
      xlab=C_top[1,1], ylab=C_top[1,2])
abline(f, col="red")
```

Seriously, I really like the R language, but this `stringsAsFactors` thing is it's weakest spot. Just fire up `options(stringsAsFactors=FALSE)` next time! (Which I of course won't). End of story.

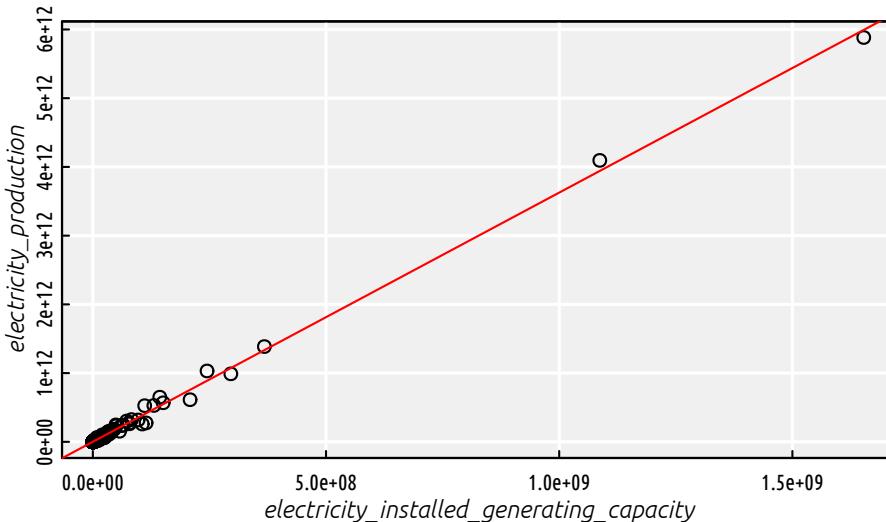


Figure 2.22: A bugfixed version of Figure 2.21



#### 2.4.4 Countries of the World – A non-linear model based on the GDP per capita

Let's revisit the World Factbook 2020 dataset (`world_factbook_2020.csv`).

```
factbook <- read.csv("datasets/world_factbook_2020.csv",
  comment="#", stringsAsFactors=FALSE)
```

##### Exercise.

Draw a histogram of the empirical distribution of the GDP per capita. Moreover, draw a histogram of the logarithm of the GDP/person.



##### Solution.

```
par(mfrow=c(1,2))
hist(factbook$gdp_per_capita_ppp, col="white", main=NA)
hist(log(factbook$gdp_per_capita_ppp), col="white", main=NA)
```

*Comment:* In Figure 2.23 we see that distribution of the GDP is right-skewed: most countries have small GDP. However, few of them (those in the “right tail” of the

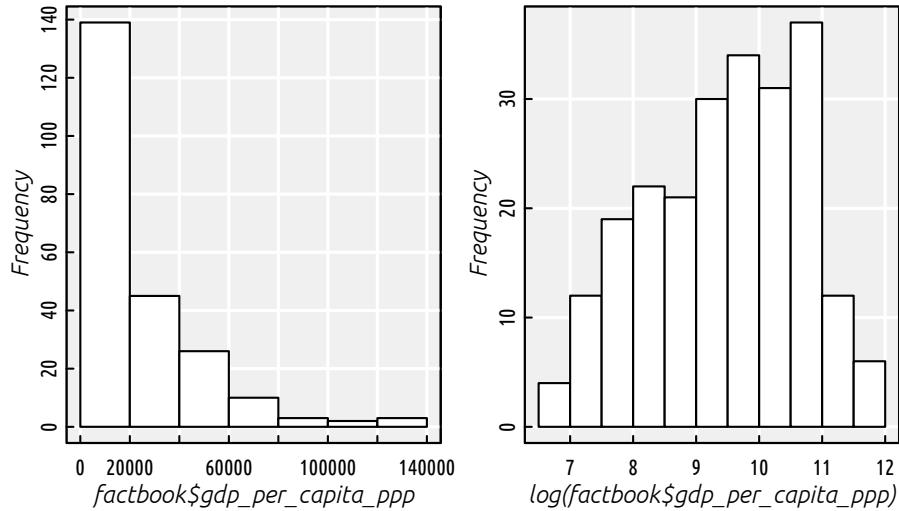


Figure 2.23: Histograms of the empirical distribution of the GDP per capita with linear (left) and log (right) scale on the X axis

*distribution) are very very rich (hey, how about taxing the richest countries?!). There is the famous observation made by V. Pareto stating that most assets are in the hands of the “wealthy minority” (compare: power law, rich-get-richer rule, preferential attachment in complex networks). Interestingly, many real-world-phenomena are distributed similarly (e.g., the popularity of web pages, the number of followers of Instagram profiles). It is frequently the case that the logarithm of the aforementioned variable looks more “normal” (is bell-shaped).*

*Side note: “The” logarithm most often refers to the logarithm base  $e$ ,  $\log x = \log_e x$ , where  $e \approx 2.72$  is the Euler constant, see `exp(1)` in R. Note that you can only compute logarithms of positive real numbers.*

Non-technical audience might be confused when asked to contemplate the distribution of the logarithm of a variable. Let's make it more user-friendly (on the other hand, we could've asked them to harden up...) by nicely re-labelling the X axis, see Figure 2.24.

```
hist(log(factbook$gdp_per_capita_ppp), axes=FALSE,
     xlab="GDP per capita (thousands USD)", main=NA, col="white")
box()
axis(2) # Y axis
at <- c(1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000)
axis(1, at=log(at), labels=at/1000)
```

*Comment: This is still a plot of the logarithm of the distribution of the per capita GDP, but it's somehow “hidden” behind the human-readable axis labels. Nice.*



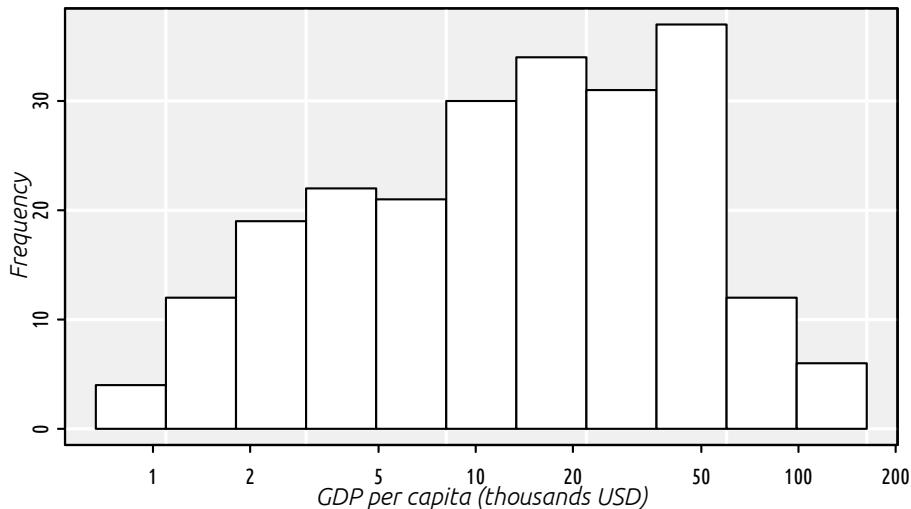


Figure 2.24: Histogram of the empirical distribution of the GDP per capita now with human-readable X axis labels (not the logarithmic scale)

### Exercise.

Fit a simple linear model of `life_expectancy_at_birth` as a function of `gdp_per_capita_ppp`.

□

### Solution.

Easy. We have already done than in one of the previous exercises. Yet, to learn something new, let's note that the `plot()` function accepts formulas as well.

```
f <- lm(life_expectancy_at_birth ~ gdp_per_capita_ppp, data=factbook)
plot(life_expectancy_at_birth ~ gdp_per_capita_ppp, data=factbook)
abline(f, col="purple")
summary(f)$r.squared

## [1] 0.4309554
```

Comment: From Figure 2.25 we see that this is not a good model.

■

### Exercise.

Draw a scatter plot of `life_expectancy_at_birth` as a function `gdp_per_capita_ppp`, with the X axis being logarithmic. Compute the correlation coefficient between `log(gdp_per_capita_ppp)` and `life_expectancy_at_birth`.

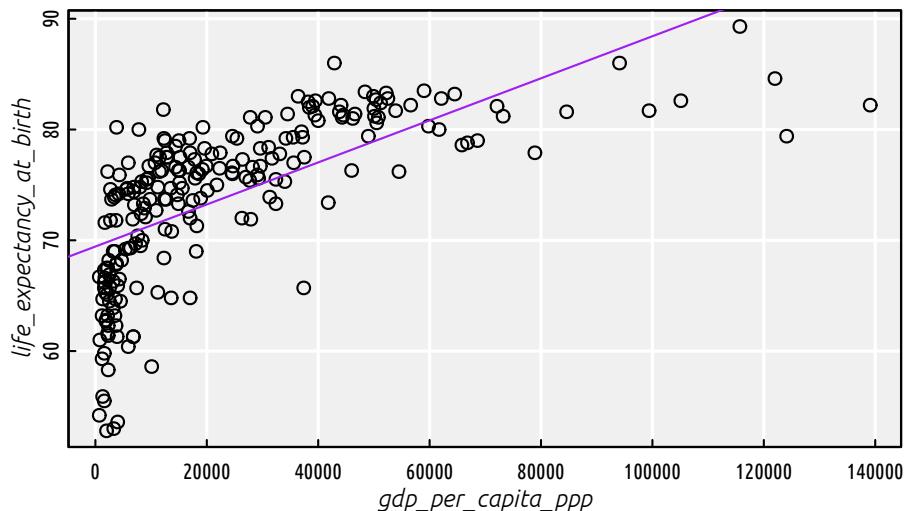


Figure 2.25: Linear model fitted for life expectancy vs. GDP/person

□

### Solution.

We could apply the `log()`-transformation manually and generate fancy X axis labels ourselves. However, the `plot()` function has the `log` argument (see `?plot.default`) which provides us with all we need, see Figure 2.26.

```
plot(factbook$gdp_per_capita_ppp,
      factbook$life_expectancy_at_birth,
      log="x")
```

Here is the *linear* correlation coefficient between the logarithm of the GDP/person and the life expectancy.

```
cor(log(factbook$gdp_per_capita_ppp), factbook$life_expectancy_at_birth,
    use="complete.obs")
```

```
## [1] 0.8066505
```

The correlation is quite high, hence the following task.

■

### Exercise.

*Fit a model predicting `life_expectancy_at_birth` by means of `log(gdp_per_capita_ppp)`.*

□

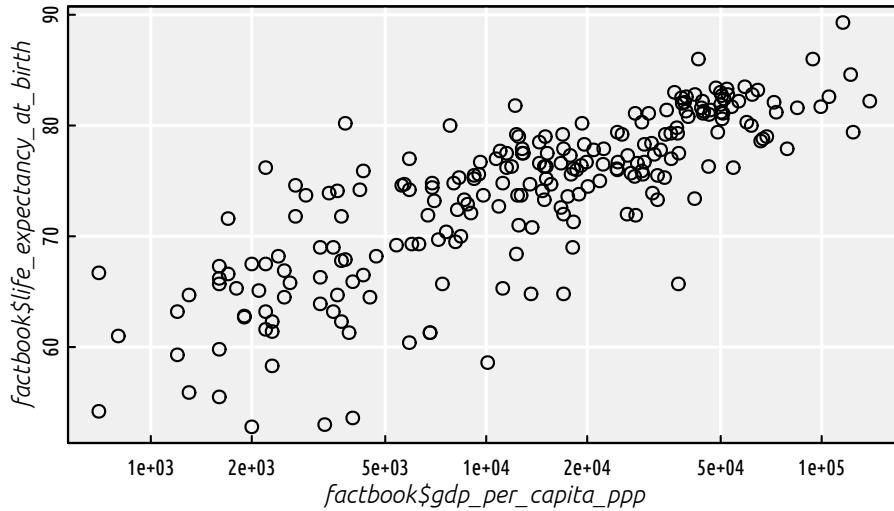


Figure 2.26: Scatter plot of life expectancy vs. GDP/person with log scale on the X axis

### Solution.

We would like to fit a model of the form  $Y = a \log X + b$ . The formula `life_expectancy_at_birth~log(gdp_per_capita_ppp)` should do the trick here.

```
f <- lm(life_expectancy_at_birth~log(gdp_per_capita_ppp), data=factbook)
plot(life_expectancy_at_birth~log(gdp_per_capita_ppp), data=factbook)
abline(f, col="red", lty=3)
f$coefficients

##           (Intercept) log(gdp_per_capita_ppp)
##             28.306385          4.817806
summary(f)$r.squared

## [1] 0.650685
```

*Comment: That is an okay model (in terms of the coefficient of determination), see Figure 2.27.*

■

### Exercise.

*Draw the fitted logarithmic model on a scatter plot with a standard, non-logarithmic X axis.*

□

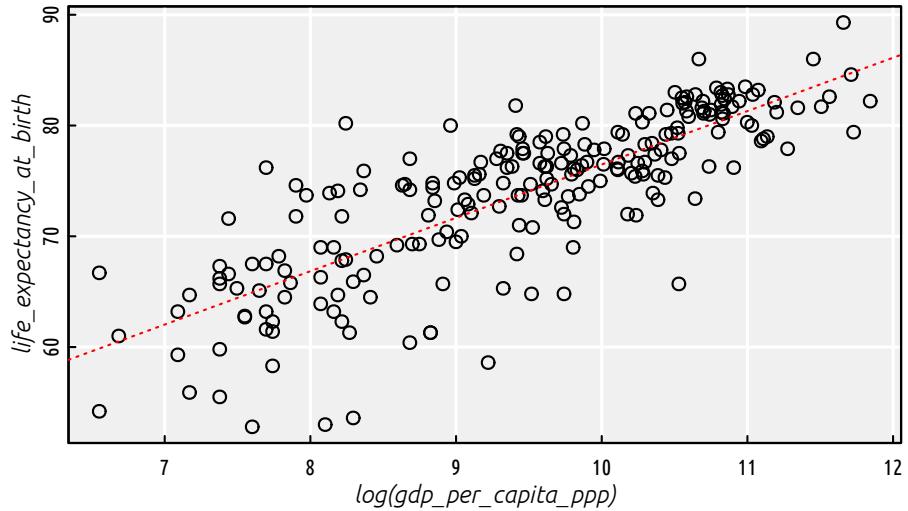


Figure 2.27: Linear model fitted for life expectancy vs. the logarithm of GDP/person

**Solution.**

The model fitted above is of the form  $Y \simeq 4.82 \log X + 28.31$ . To depict it on a plot with linear (non-logarithmic) axes, we can compute this formula on multiple points by hand, see Figure 2.28.

```
plot(factbook$gdp_per_capita_ppp, factbook$life_expectancy_at_birth)

# many points on the X axis:
xxx <- seq(min(factbook$gdp_per_capita_ppp, na.rm=TRUE),
            max(factbook$gdp_per_capita_ppp, na.rm=TRUE),
            length.out=101)
yyy <- f$coefficients[1] + f$coefficients[2]*log(xxx)
lines(xxx, yyy, col="red", lty=3)
```

*Comment:* Well, people are not immortal... The original (linear) model didn't really take that into account. Also, recall that correlation is not causation. Moreover, there is a lot of variability at an individual level. Being born in a less-wealthy country (e.g., not in a tax haven), doesn't mean you don't have the whole life ahead of you. Do the cool stuff, do something for the others. Life's not about money.



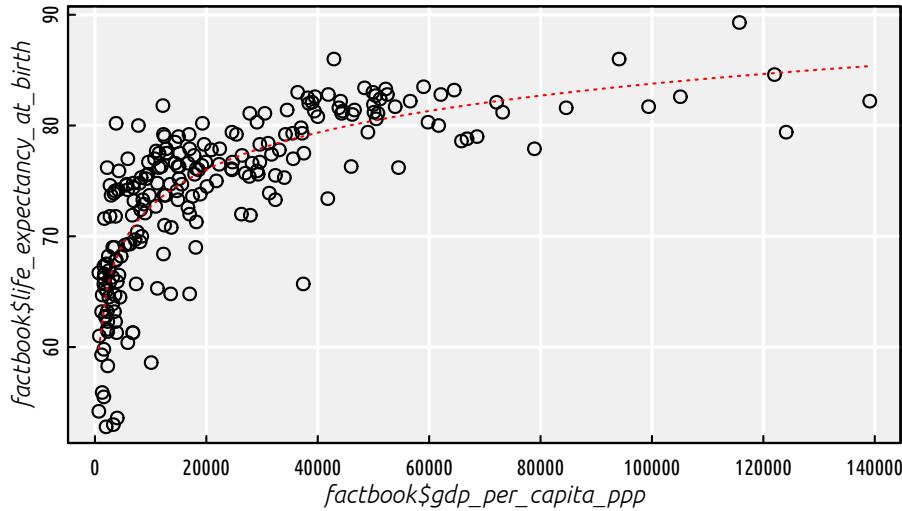


Figure 2.28: Logarithmic model fitted for life expectancy vs. GDP/person

#### 2.4.5 Countries of the World – A multiple regression model for the per capita GDP

Let's play with World Factbook 2020 (`world_factbook_2020.csv`) once again. World is an interesting place, so we're far from being bored with this dataset.

```
factbook <- read.csv("datasets/world_factbook_2020.csv",
  comment="#", stringsAsFactors=FALSE)
```

Let's restrict ourselves to the following columns, mostly related to imports and exports:

```
factbookn <- factbook[c("gdp_purchasing_power_parity",
  "imports", "exports", "electricity_exports",
  "electricity_imports", "military_expenditures",
  "crude_oil_exports", "crude_oil_imports",
  "natural_gas_exports", "natural_gas_imports",
  "reserves_of_foreign_exchange_and_gold")]
```

Let's compute the per capita versions of the above, by dividing all values by each country's population:

```
for (i in 1:ncol(factbookn))
  factbookn[[i]] <- factbookn[[i]]/factbook$population
```

We are going to build a few multiple regression models using the `step()` function, which is not too fond of missing values, therefore they should be removed first:

```
factbookn <- na.omit(factbookn)
c(nrow(factbook), nrow(factbookn)) # how many countries were omitted?

## [1] 261 157
```

**Exercise.**

*Build a model for gdp\_purchasing\_power\_parity as a function of imports and exports (all per capita).*

□

**Solution.**

Let's first take a look at how the aforementioned variables are related to each other, see Figure 2.29.

```
pairs(factbookn[c("gdp_purchasing_power_parity", "imports", "exports")])
cor(factbookn[c("gdp_purchasing_power_parity", "imports", "exports")])

##                                     gdp_purchasing_power_parity
## gdp_purchasing_power_parity                 1.0000000
## imports                               0.8289144
## exports                               0.8189854
##                               imports   exports
## gdp_purchasing_power_parity 0.8289144 0.8189854
## imports                         1.0000000 0.9424084
## exports                         0.9424084 1.0000000
```

They are nicely correlated. Moreover, they are on a similar scale (“tens of thousands of USD per capita”).

Fitting the requested model yields:

```
options(scipen=10) # prefer "decimal" over "scientific" notation
f1 <- lm(gdp_purchasing_power_parity~imports+exports, data=factbookn)
f1$coefficients

## (Intercept)      imports      exports
## 9852.5381318    1.4419398    0.7806693
summary(f1)$adj.r.squared

## [1] 0.6959805
```

■

**Exercise.**

*Use forward selection to come up with a model for gdp\_purchasing\_power\_parity per capita.*

□

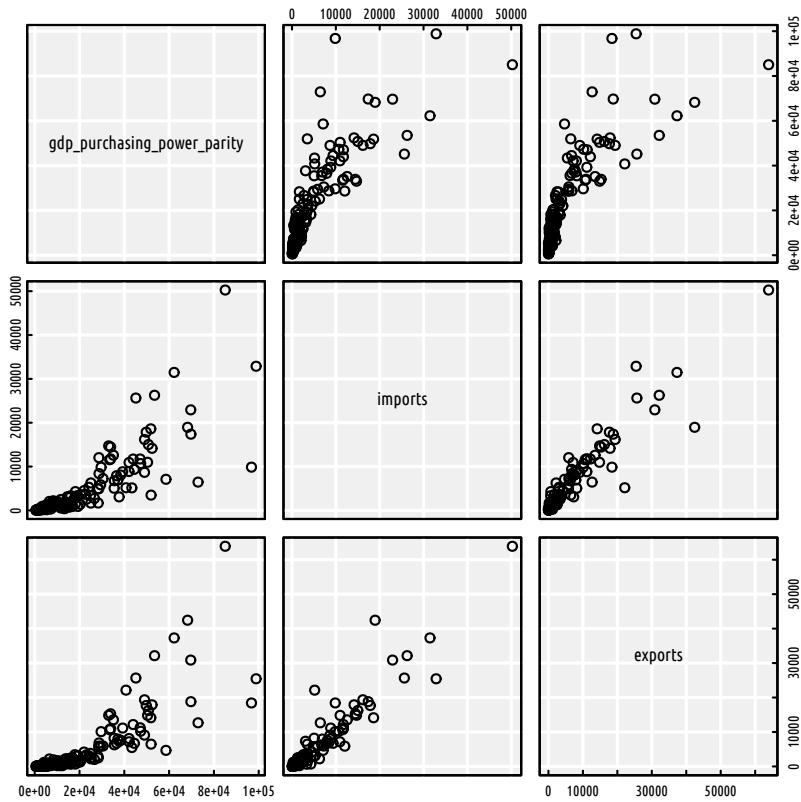


Figure 2.29: Scatter plot matrix for GDP, imports and exports

**Solution.**

```
f2 <- step(lm(gdp_purchasing_power_parity~1, data=factbookn), # empty model
           scope=formula(lm(gdp_purchasing_power_parity~., data=factbookn)), # full model
           direction="forward", trace=0)
f2

## 
## Call:
## lm(formula = gdp_purchasing_power_parity ~ imports + crude_oil_exports +
##      crude_oil_imports + electricity_imports + natural_gas_imports,
##      data = factbookn)
##
## Coefficients:
##             (Intercept)          imports    crude_oil_exports
##                 7603.236          1.775        128472.216
##      crude_oil_imports  electricity_imports natural_gas_imports
##                 100781.638          1.621            3.131
summary(f2)$adj.r.squared

## [1] 0.7865036
```

*Comment:* Interestingly, it's mostly the import-related variables that contribute to the GDP per capita. However, the model is not perfect, so we should refrain ourselves from building a brand new economic theory around this "discovery". On the other hand, you know what they say: all models are wrong, but some might be useful. Note that we used the adjusted  $R^2$  coefficient to correct for the number of variables in the model so as to make it more comparable with the coefficient corresponding to the f1 model.

**Exercise.**

Use backward elimination to construct a model for *gdp\_purchasing\_power\_parity* per capita.

**Solution.**

```
f3 <- step(lm(gdp_purchasing_power_parity~., data=factbookn),
           scope=formula(lm(gdp_purchasing_power_parity~1, data=factbookn)),
           direction="backward", trace=0)
f3

## 
## Call:
## lm(formula = gdp_purchasing_power_parity ~ imports + electricity_imports +
##      crude_oil_exports + crude_oil_imports + natural_gas_imports,
##      data = factbookn)
##
```

```

## Coefficients:
## (Intercept)           imports electricity_imports
## 7603.236              1.775          1.621
## crude_oil_exports    crude_oil_imports natural_gas_imports
## 128472.216            100781.638          3.131
summary(f3)$adj.r.squared

## [1] 0.7865036

```

*Comment: This is the same model as the one found by forward selection, i.e., f2.*

■

## 2.5 Outro

### 2.5.1 Remarks

Multiple regression is simple, fast to apply and interpretable.

Linear models go beyond fitting of straight lines and other hyperplanes!

A complex model may overfit and hence generalise poorly to unobserved inputs.

Note that the SSR criterion makes the models sensitive to outliers.

**Remember:**

good models

=

better understanding of the modelled reality + better predictions

=

more revenue, your boss' happiness, your startup's growth etc.

### 2.5.2 Other Methods for Regression

Other example approaches to regression:

- ridge regression,
- lasso regression,
- least absolute deviations (LAD) regression,
- multiadaptive regression splines (MARS),
- K-nearest neighbour (K-NN) regression, see `FNN::knn.reg()` in R,
- regression trees,
- support-vector regression (SVR),
- neural networks (also deep) for regression.

### 2.5.3 Derivation of the Solution (\*\*)

We would like to find an analytical solution to the problem of minimising of the sum of squared residuals:

$$\min_{\beta_0, \beta_1, \dots, \beta_p \in \mathbb{R}} E(\beta_0, \beta_1, \dots, \beta_p) = \sum_{i=1}^n (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i)^2$$

This requires computing the  $p+1$  partial derivatives  $\partial E / \partial \beta_j$  for  $j = 0, \dots, p$ .

The partial derivatives are very similar to each other;  $\frac{\partial E}{\partial \beta_0}$  is given by:

$$\frac{\partial E}{\partial \beta_0}(\beta_0, \beta_1, \dots, \beta_p) = 2 \sum_{i=1}^n (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i)$$

and  $\frac{\partial E}{\partial \beta_j}$  for  $j > 0$  is equal to:

$$\frac{\partial E}{\partial \beta_j}(\beta_0, \beta_1, \dots, \beta_p) = 2 \sum_{i=1}^n x_{i,j} (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i)$$

Then all we need to do is to solve the system of linear equations:

$$\begin{cases} \frac{\partial E}{\partial \beta_0}(\beta_0, \beta_1, \dots, \beta_p) = 0 \\ \frac{\partial E}{\partial \beta_1}(\beta_0, \beta_1, \dots, \beta_p) = 0 \\ \vdots \\ \frac{\partial E}{\partial \beta_p}(\beta_0, \beta_1, \dots, \beta_p) = 0 \end{cases}$$

The above system of  $p+1$  linear equations, which we are supposed to solve for  $\beta_0, \beta_1, \dots, \beta_p$ :

$$\begin{cases} 2 \sum_{i=1}^n (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i) = 0 \\ 2 \sum_{i=1}^n x_{i,1} (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i) = 0 \\ \vdots \\ 2 \sum_{i=1}^n x_{i,p} (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i) = 0 \end{cases}$$

can be rewritten as:

$$\begin{cases} \sum_{i=1}^n (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}) = \sum_{i=1}^n y_i \\ \sum_{i=1}^n x_{i,1} (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}) = \sum_{i=1}^n x_{i,1} y_i \\ \vdots \\ \sum_{i=1}^n x_{i,p} (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}) = \sum_{i=1}^n x_{i,p} y_i \end{cases}$$

and further as:

$$\left\{ \begin{array}{lcl} \beta_0 n + \beta_1 \sum_{i=1}^n x_{i,1} + \cdots + \beta_p \sum_{i=1}^n x_{i,p} & = & \sum_{i=1}^n y_i \\ \beta_0 \sum_{i=1}^n x_{i,1} + \beta_1 \sum_{i=1}^n x_{i,1} x_{i,1} + \cdots + \beta_p \sum_{i=1}^n x_{i,1} x_{i,p} & = & \sum_{i=1}^n x_{i,1} y_i \\ \vdots \\ \beta_0 \sum_{i=1}^n x_{i,p} + \beta_1 \sum_{i=1}^n x_{i,p} x_{i,1} + \cdots + \beta_p \sum_{i=1}^n x_{i,p} x_{i,p} & = & \sum_{i=1}^n x_{i,p} y_i \end{array} \right.$$

Note that the terms involving  $x_{i,j}$  and  $y_i$  (the sums) are all constant – these are some fixed real numbers. We have learned how to solve such problems in high school.

### Exercise.

Try deriving the analytical solution and implementing it for  $p = 2$ . Recall that in the previous chapter we solved the special case of  $p = 1$ .

□

### 2.5.4 Solution in Matrix Form (\*\*\*)

Assume that  $\mathbf{X} \in \mathbb{R}^{n \times p}$  (a matrix with inputs),  $\mathbf{y} \in \mathbb{R}^{n \times 1}$  (a column vector of reference outputs) and  $\boldsymbol{\beta} \in \mathbb{R}^{(p+1) \times 1}$  (a column vector of parameters).

Firstly, note that a linear model of the form:

$$f_{\boldsymbol{\beta}}(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

can be rewritten as:

$$f_{\boldsymbol{\beta}}(\mathbf{x}) = \beta_0 1 + \beta_1 x_1 + \cdots + \beta_p x_p = \dot{\mathbf{x}}\boldsymbol{\beta},$$

where  $\dot{\mathbf{x}} = [1 \ x_1 \ x_2 \ \cdots \ x_p]$ .

Similarly, if we assume that  $\dot{\mathbf{X}} = [\mathbf{1} \ \mathbf{X}] \in \mathbb{R}^{n \times (p+1)}$  is the input matrix with a prepended column of 1s, i.e.,  $\mathbf{1} = [1 \ 1 \ \cdots \ 1]^T$  and  $\dot{x}_{i,0} = 1$  (for brevity of notation the columns added will have index 0),  $\dot{x}_{i,j} = x_{i,j}$  for all  $j \geq 1$  and all  $i$ , then:

$$\hat{\mathbf{y}} = \dot{\mathbf{X}}\boldsymbol{\beta}$$

gives the vector of predicted outputs for every input point.

This way, the sum of squared residuals

$$E(\beta_0, \beta_1, \dots, \beta_p) = \sum_{i=1}^n (\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_p x_{i,p} - y_i)^2$$

can be rewritten as:

$$E(\boldsymbol{\beta}) = \|\dot{\mathbf{X}}\boldsymbol{\beta} - \mathbf{y}\|^2,$$

where as usual  $\|\cdot\|^2$  denotes the squared Euclidean norm.

Recall that this can be re-expressed as:

$$E(\beta) = (\dot{\mathbf{X}}\beta - \mathbf{y})^T(\dot{\mathbf{X}}\beta - \mathbf{y}).$$

In order to find the minimum of  $E$  w.r.t.  $\beta$ , we need to find the parameters that make the partial derivatives vanish, i.e.:

$$\left\{ \begin{array}{lcl} \frac{\partial E}{\partial \beta_0}(\beta) & = & 0 \\ \frac{\partial E}{\partial \beta_1}(\beta) & = & 0 \\ \vdots & & \\ \frac{\partial E}{\partial \beta_p}(\beta) & = & 0 \end{array} \right.$$

**Remark.** (\*\*\*) Interestingly, the above can also be expressed in matrix form, using the special notation:

$$\nabla E(\beta) = \mathbf{0}$$

Here,  $\nabla E$  (nabla symbol = differential operator) denotes the function gradient, i.e., the vector of all partial derivatives. This is nothing more than syntactic sugar for this quite commonly applied operator.

Anyway, the system of linear equations we have derived above:

$$\left\{ \begin{array}{lcl} \beta_0 n + \beta_1 \sum_{i=1}^n x_{i,1} + \cdots + \beta_p \sum_{i=1}^n x_{i,p} & = & \sum_{i=1}^n y_i \\ \beta_0 \sum_{i=1}^n x_{i,1} + \beta_1 \sum_{i=1}^n x_{i,1}x_{i,1} + \cdots + \beta_p \sum_{i=1}^n x_{i,1}x_{i,p} & = & \sum_{i=1}^n x_{i,1}y_i \\ \vdots & & \\ \beta_0 \sum_{i=1}^n x_{i,p} + \beta_1 \sum_{i=1}^n x_{i,p}x_{i,1} + \cdots + \beta_p \sum_{i=1}^n x_{i,p}x_{i,p} & = & \sum_{i=1}^n x_{i,p}y_i \end{array} \right.$$

can be rewritten in matrix terms as:

$$\left\{ \begin{array}{lcl} \beta_0 \dot{\mathbf{x}}_{:,0}^T \dot{\mathbf{x}}_{:,0} + \beta_1 \dot{\mathbf{x}}_{:,0}^T \dot{\mathbf{x}}_{:,1} + \cdots + \beta_p \dot{\mathbf{x}}_{:,0}^T \dot{\mathbf{x}}_{:,p} & = & \dot{\mathbf{x}}_{:,0}^T \mathbf{y} \\ \beta_0 \dot{\mathbf{x}}_{:,1}^T \dot{\mathbf{x}}_{:,0} + \beta_1 \dot{\mathbf{x}}_{:,1}^T \dot{\mathbf{x}}_{:,1} + \cdots + \beta_p \dot{\mathbf{x}}_{:,1}^T \dot{\mathbf{x}}_{:,p} & = & \dot{\mathbf{x}}_{:,1}^T \mathbf{y} \\ \vdots & & \\ \beta_0 \dot{\mathbf{x}}_{:,p}^T \dot{\mathbf{x}}_{:,0} + \beta_1 \dot{\mathbf{x}}_{:,p}^T \dot{\mathbf{x}}_{:,1} + \cdots + \beta_p \dot{\mathbf{x}}_{:,p}^T \dot{\mathbf{x}}_{:,p} & = & \dot{\mathbf{x}}_{:,p}^T \mathbf{y} \end{array} \right.$$

This can be restated as:

$$\left\{ \begin{array}{lcl} \left( \dot{\mathbf{x}}_{:,0}^T \dot{\mathbf{X}} \right) \beta & = & \dot{\mathbf{x}}_{:,0}^T \mathbf{y} \\ \left( \dot{\mathbf{x}}_{:,1}^T \dot{\mathbf{X}} \right) \beta & = & \dot{\mathbf{x}}_{:,1}^T \mathbf{y} \\ \vdots & & \\ \left( \dot{\mathbf{x}}_{:,p}^T \dot{\mathbf{X}} \right) \beta & = & \dot{\mathbf{x}}_{:,p}^T \mathbf{y} \end{array} \right.$$

which in turn is equivalent to:

$$\left( \dot{\mathbf{X}}^T \dot{\mathbf{X}} \right) \beta = \dot{\mathbf{X}}^T \mathbf{y}.$$

Such a system of linear equations in matrix form can be solved numerically using, amongst others, the `solve()` function.

**Remark.** (\*\*\*) In practice, we'd rather rely on QR or SVD decompositions of matrices for efficiency and numerical accuracy reasons.

Numeric example – solution via `lm()`:

```
X1 <- as.numeric(Credit$Balance[Credit$Balance>0])
X2 <- as.numeric(Credit$Income[Credit$Balance>0])
Y <- as.numeric(Credit$Rating[Credit$Balance>0])
lm(Y~X1+X2)$coefficients

## (Intercept)           X1            X2
## 172.5586670   0.1828011   2.1976461
```

Recalling that  $\mathbf{A}^T \mathbf{B}$  can be computed by calling `t(A) %*% B` or – even faster – by calling `crossprod(A, B)`, we can also use `solve()` to obtain the same result:

```
X_dot <- cbind(1, X1, X2)
solve(crossprod(X_dot, X_dot), crossprod(X_dot, Y))

##          [,1]
## 172.5586670
## X1    0.1828011
## X2    2.1976461
```

## 2.5.5 Pearson's r in Matrix Form (\*\*)

Recall the Pearson linear correlation coefficient:

$$r(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Denote with  $\mathbf{x}^\circ$  and  $\mathbf{y}^\circ$  the centred versions of  $\mathbf{x}$  and  $\mathbf{y}$ , respectively, i.e.,  $x_i^\circ = x_i - \bar{x}$  and  $y_i^\circ = y_i - \bar{y}$ .

Rewriting the above yields:

$$r(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^n x_i^\circ y_i^\circ}{\sqrt{\sum_{i=1}^n (x_i^\circ)^2} \sqrt{\sum_{i=1}^n (y_i^\circ)^2}}$$

which is exactly:

$$r(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x}^\circ \cdot \mathbf{y}^\circ}{\|\mathbf{x}^\circ\| \|\mathbf{y}^\circ\|}$$

i.e., the normalised dot product of the centred versions of the two vectors.

This is the cosine of the angle between the two vectors (in  $n$ -dimensional spaces)!

(\*\*) Recalling from the previous chapter that  $\mathbf{A}^T \mathbf{A}$  gives the dot product between all the pairs of columns in a matrix  $\mathbf{A}$ , we can implement an equivalent version of `cor(C)` as follows:

```

C <- Credit[Credit$Balance>0,
  c("Rating", "Limit", "Income", "Age",
    "Education", "Balance")]
C_centred <- apply(C, 2, function(c) c-mean(c))
C_normalised <- apply(C_centred, 2, function(c)
  c/sqrt(sum(c^2)))
round(t(C_normalised) %*% C_normalised, 3)

##          Rating  Limit Income   Age Education Balance
## Rating     1.000  0.996  0.831 0.167   -0.040  0.798
## Limit      0.996  1.000  0.834 0.164   -0.032  0.796
## Income     0.831  0.834  1.000 0.227   -0.033  0.414
## Age        0.167  0.164  0.227 1.000    0.024  0.008
## Education -0.040 -0.032 -0.033 0.024    1.000  0.001
## Balance    0.798  0.796  0.414 0.008    0.001  1.000

```

### 2.5.6 Further Reading

Recommended further reading: (James et al. 2017: Chapters 1, 2 and 3)

Other: (Hastie et al. 2017: Chapter 1, Sections 3.2 and 3.3)

# Chapter 3

## Classification with K-Nearest Neighbours

### 3.1 Introduction

#### 3.1.1 Classification Task

Let  $\mathbf{X} \in \mathbb{R}^{n \times p}$  be an input matrix that consists of  $n$  points in a  $p$ -dimensional space (each of the  $n$  objects is described by means of  $p$  numerical features).

Recall that in supervised learning, with each  $\mathbf{x}_{i,:}$  we associate the desired output  $y_i$ .

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

In this chapter we are interested in **classification** tasks; we assume that each  $y_i$  is a *label* (e.g., a character string) – it is of quantitative/categorical type.

Most commonly, we are faced with **binary classification** tasks where there are only two possible distinct labels.

We traditionally denote them with 0s and 1s.

For example:

0	1
no	yes
false	true
failure	success
healthy	ill

On the other hand, in **multiclass classification**, we assume that each  $y_i$  takes more than two possible values.

Example plot of a synthetic dataset with the reference binary  $ys$  is given in Figure 3.1. The “true” decision boundary is at  $X_1 = 0$  but the classes slightly overlap (the dataset is a bit noisy).

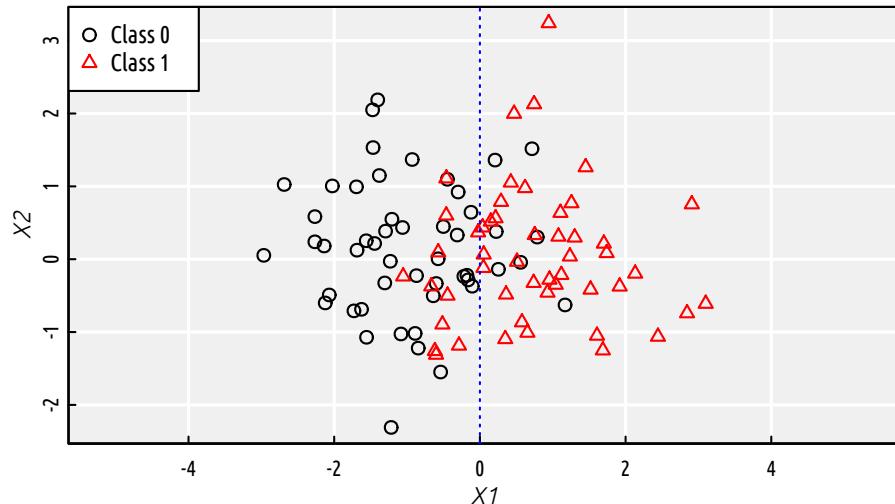


Figure 3.1: A synthetic 2D dataset with the true decision boundary at  $X_1 = 0$

### 3.1.2 Data

For illustration, let’s consider the Wine Quality dataset (Cortez et al. 2009) that can be downloaded from the UCI Machine Learning Repository (<https://archive.ics.uci.edu/ml/datasets/Wine+Quality>) – white wines only.

```
wines <- read.csv("datasets/winequality-all.csv", comment="#")
wines <- wines[wines$color == "white",]
(n <- nrow(wines)) # number of samples

## [1] 4898
```

These are Vinho Verde wine samples from the north of Portugal, see <https://www.vinhoverde.pt/en/homepage>.

There are 11 physicochemical features reported. Moreover, there is a wine rating (which we won't consider here) on the scale 0 (bad) to 10 (excellent) given by wine experts.

The input matrix  $\mathbf{X} \in \mathbb{R}^{n \times p}$  consists of the first 10 numeric variables:

```
X <- as.matrix(wines[, 1:10])
dim(X)

## [1] 4898    10

head(X, 2) # first two rows

##      fixed.acidity volatile.acidity citric.acid residual.sugar
## 1600      7.0          0.27        0.36       20.7
## 1601      6.3          0.30        0.34        1.6
##      chlorides free.sulfur.dioxide total.sulfur.dioxide density
## 1600     0.045           45         170     1.001
## 1601     0.049           14         132     0.994
##      pH sulphates
## 1600 3.0      0.45
## 1601 3.3      0.49
```

The 11th variable measures the amount of alcohol (in %).

We will convert this dependent variable to a binary one:

- 0 == (alcohol < 12) == lower-alcohol wines
- 1 == (alcohol >= 12) == higher-alcohol wines

```
# recall that TRUE == 1
Y <- factor(as.character(as.numeric(wines$alcohol >= 12)))
table(Y)

## Y
##   0   1
## 4085 813
```

Now  $(\mathbf{X}, \mathbf{y})$  is a basis for an interesting (yet challenging) binary classification task.

### 3.1.3 Training and Test Sets

Recall that we are genuinely interested in the construction of supervised learning models for the two following purposes:

- **description** – to explain a given dataset in simpler terms,
- **prediction** – to forecast the values of the dependent variable for inputs that are yet to be observed.

In the latter case:

- we don't want our models to *overfit* to current data,
- we want our models to *generalise* well to new data.

One way to assess if a model has sufficient predictive power is based on a random **train-test split** of the original dataset:

- *training sample* (usually 60-80% of the observations) – used to construct a model,
- *test sample* (remaining 40-20%) – used to assess the goodness of fit.

**Remark.** Test sample must not be used in the training phase! (No cheating!)

60/40% train-test split in R:

```
set.seed(123) # reproducibility matters
random_indices <- sample(n)
head(random_indices) # preview

## [1] 2463 2511 2227  526 4291 2986
# first 60% of the indices (they are arranged randomly)
# will constitute the train sample:
train_indices <- random_indices[1:floor(n*0.6)]
X_train <- X[train_indices,]
Y_train <- Y[train_indices]
# the remaining indices (40%) go to the test sample:
X_test <- X[-train_indices,]
Y_test <- Y[-train_indices]
```

### 3.1.4 Discussed Methods

Our aim is to build a classifier that takes 10 wine physicochemical features and determines whether it's a “strong” wine.

We will discuss 3 simple and educational (yet practically useful) classification algorithms:

- *K-nearest neighbour scheme* – this chapter,
- *Decision trees* – the next chapter,
- *Logistic regression* – the next chapter.

## 3.2 K-nearest Neighbour Classifier

### 3.2.1 Introduction

**Rule.** “If you don’t know what to do in a situation, just act like the people around you”

For some integer  $K \geq 1$ , the **K-Nearest Neighbour (K-NN) Classifier** proceeds as follows.

To classify a new point  $\mathbf{x}'$ :

1. find the  $K$  nearest neighbours of a given point  $\mathbf{x}'$  amongst the points in the train set, denoted  $\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_K}$ :
  - a. compute the Euclidean distances between  $\mathbf{x}'$  and each  $\mathbf{x}_{i,\cdot}$  from the train set,
$$d_i = \|\mathbf{x}' - \mathbf{x}_{i,\cdot}\|$$
- b. order  $d_i$ s in increasing order,  $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_K}$
- c. pick first  $K$  indices (these are the *nearest* neighbours)
2. fetch the corresponding reference labels  $y_{i_1}, \dots, y_{i_K}$
3. return their *mode* as a result, i.e., the most frequently occurring label (a.k.a. *majority vote*)

Here is how  $K$ -NN classifier works on a synthetic 2D dataset. Firstly let’s consider  $K = 1$ , see Figure 3.2. Gray and pink regions depict how new points would be classified. In particular 1-NN is “greedy” in the sense that we just locate the nearest point.

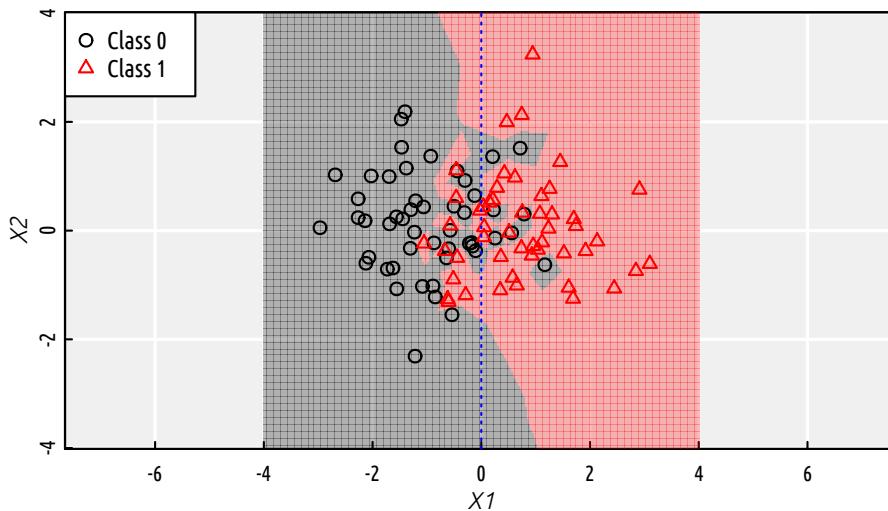


Figure 3.2: 1-NN class bounds for our 2D synthetic dataset

**Remark.** (\*) 1-NN classification is essentially based on a dataset's so-called Voronoi diagram.

Increasing  $K$  somehow smoothens the decision boundary (this makes it less “local” and more “global”). Figure 3.3 depicts the  $K = 3$  case.

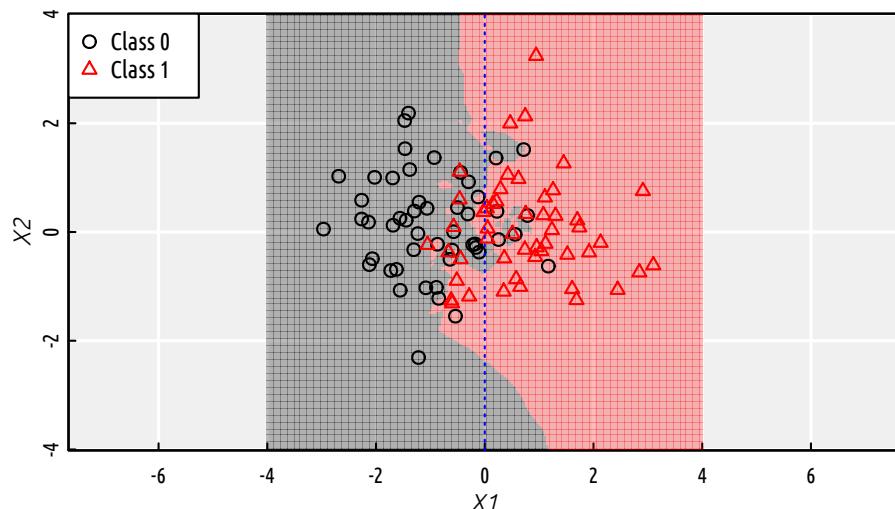


Figure 3.3: 3-NN class bounds for our 2D synthetic dataset

Recall that the “true” decision boundary for this synthetic dataset is at  $X_1 = 0$ . The 25-NN classifier did quite a good job, see Figure 3.4.

### 3.2.2 Example in R

We shall be calling the `knn()` function from package `FNN` to classify the points from the test sample extracted from the `wines` dataset:

```
library("FNN")
```

Let's make prediction using the 5-nn classifier:

```
Y_knn5 <- knn(X_train, X_test, Y_train, k=5)
head(Y_test, 28) # True Ys
```

```
head(Y_knn5, 28) # Predicted Ys
```

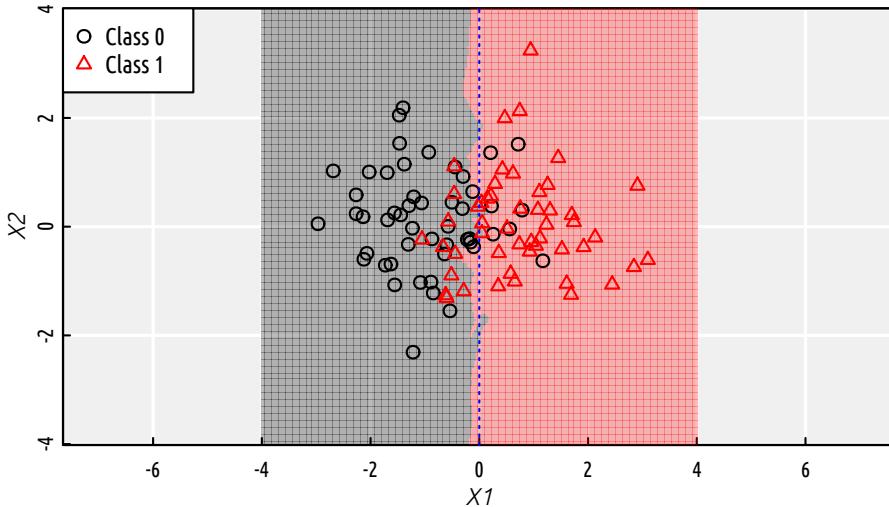


Figure 3.4: 25-NN class bounds for our 2D synthetic dataset

```

mean(Y_test == Y_knn5) # accuracy
## [1] 0.8173469

9-nn classifier:
Y_knn9 <- knn(X_train, X_test, Y_train, k=9)
head(Y_test, 28) # True Ys
## [1] 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
head(Y_knn9, 28) # Predicted Ys
## [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
mean(Y_test == Y_knn9) # accuracy
## [1] 0.8193878

```

### 3.2.3 Feature Engineering

Note that the Euclidean distance that we used above implicitly assumes that every feature (independent variable) is on the same scale.

However, when dealing with, e.g., physical quantities, we often perform conversions of units of measurement (kg → g, feet → m etc.).

Transforming a single feature may drastically change the metric structure of the dataset and therefore highly affect the obtained predictions.

To “bring data to the same scale”, we often apply a trick called **standardization**.

Computing the so-called **Z-scores** of the  $j$ -th feature,  $\mathbf{x}_{\cdot,j}$ , is done by subtracting from each observation the sample mean and dividing the result by the sample standard deviation:

$$z_{i,j} = \frac{x_{i,j} - \bar{x}_{\cdot,j}}{s_{x_{\cdot,j}}}$$

This a new feature  $\mathbf{z}_{\cdot,j}$  that always has mean 0 and standard deviation of 1.

Moreover, it is *unit-less* (e.g., we divide a value in kgs by a value in kgs, the units are cancelled out). This, amongst others, prevents one of the features from dominating the other ones.

Z-scores are easy to interpret, e.g., 0.5 denotes an observation that is 0.5 standard deviations above the mean.

Let’s compute  $\mathbf{Z}_{\text{train}}$  and  $\mathbf{Z}_{\text{test}}$ , being the standardised versions of  $\mathbf{X}_{\text{train}}$  and  $\mathbf{X}_{\text{test}}$ , respectively.

```
means <- apply(X, 2, mean) # column means
sds   <- apply(X, 2, sd)   # column standard deviations
Z_train <- X_train # to be done
Z_test  <- X_test # to be done
for (j in 1:ncol(X)) {
  Z_train[,j] <- (Z_train[,j]-means[j])/sds[j]
  Z_test[,j]  <- (Z_test[,j] -means[j])/sds[j]
}
```

Alternatively:

```
Z_train <- t(apply(X_train, 1, function(c) (c-means)/sds))
Z_test  <- t(apply(X_test, 1, function(c) (c-means)/sds))
```

See Figure 3.5 for an illustration. Note that the righthand figures (histograms of standardised variables) are on the same scale now.

**Remark.** Of course, standardisation is only about shifting and scaling, it preserves the shape of the distribution. If the original variable is right skewed or bimodal, its standardised version will remain as such.

Let’s compute the accuracy of K-NN classifiers acting on standardised data.

```
Y_knn5s <- knn(Z_train, Z_test, Y_train, k=5)
mean(Y_test == Y_knn5s) # accuracy
```

```
## [1] 0.9102041
```

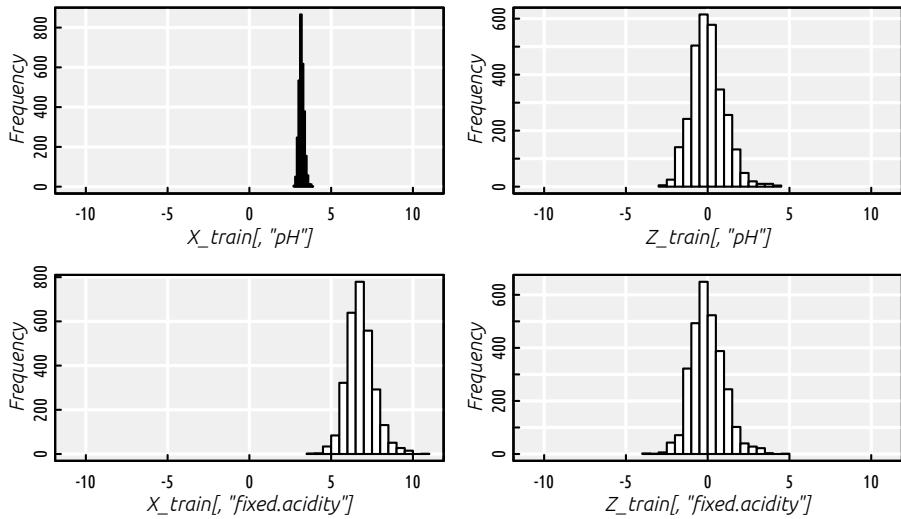


Figure 3.5: Empirical distribution of two variables (pH on the top, fixed.acidity on the bottom) before (left) and after (right) standardising

```
Y_knn9s <- knn(Z_train, Z_test, Y_train, k=9)
mean(Y_test == Y_knn9s) # accuracy

## [1] 0.9122449
```

The accuracy is much better.

Standardisation is an example of *feature engineering*.

Good models rarely work well “straight out of the box” – if that was the case, we wouldn’t need data scientists and machine learning engineers!

To increase models’ accuracy, we often spend a lot of time:

- cleansing data (e.g., removing outliers)
- extracting new features
- transforming existing features
- trying to find a set of features that are relevant

This is the “more art than science” part of data science (sic!), and hence most textbooks are not really eager for discussing such topics (including this one).

Sorry, this is sad but true. The solutions that work well in the case of dataset A may fail in the B case and vice versa. However, the more exercises you solve, the greater the arsenal of ideas/possible approaches you will have at hand when dealing with real-world problems.

Feature selection – example (manually selected columns):

```

features <- c("density", "residual.sugar")
Y_knn5s <- knn(Z_train[,features], Z_test[,features],
                  Y_train, k=5)
mean(Y_test == Y_knn5s) # accuracy

## [1] 0.9168367

Y_knn9s <- knn(Z_train[,features], Z_test[,features],
                  Y_train, k=9)
mean(Y_test == Y_knn9s) # accuracy

## [1] 0.9255102

```

**Exercise.**

Try to find a combination of 2-4 features (by guessing or applying magic tricks) that increases the accuracy of a K-NN classifier on this dataset.

□

## 3.3 Model Assessment and Selection

### 3.3.1 Performance Metrics

Recall that  $y_i$  denotes the true label associated with the  $i$ -th observation.

Let  $\hat{y}_i$  denote the classifier's output for a given  $\mathbf{x}_i$ ..

Ideally, we'd wish that  $\hat{y}_i = y_i$ .

Sadly, in practice we will make errors.

Here are the 4 possible situations (true vs. predicted label):

	$y_i = 0$	$y_i = 1$
$\hat{y}_i = 0$	<b>True Negative</b>	False Negative (Type II error)
$\hat{y}_i = 1$	False Positive (Type I error)	<b>True Positive</b>

Note that the terms **positive** and **negative** refer to the classifier's output, i.e., occur when  $\hat{y}_i$  is equal to 1 and 0, respectively.

A **confusion matrix** is used to summarise the correctness of predictions for the whole sample:

```

Y_pred <- knn(Z_train, Z_test, Y_train, k=9)
(C <- table(Y_pred, Y_test))

##      Y_test

```

```
## Y_pred    0    1
##      0 1606 135
##      1   37 182
```

For example,

```
C[1,1] # number of TNs
```

```
## [1] 1606
```

```
C[2,1] # number of FPs
```

```
## [1] 37
```

**Accuracy** is the ratio of the correctly classified instances to all the instances.

In other words, it is the probability of making a correct prediction.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{1}{n} \sum_{i=1}^n \mathbb{I}(y_i = \hat{y}_i)$$

where  $\mathbb{I}$  is the indicator function,  $\mathbb{I}(l) = 1$  if logical condition  $l$  is true and 0 otherwise.

```
mean(Y_test == Y_pred) # accuracy
```

```
## [1] 0.9122449
```

```
(C[1,1]+C[2,2])/sum(C) # equivalently
```

```
## [1] 0.9122449
```

In many applications we are dealing with **unbalanced problems**, where the case  $y_i = 1$  is relatively rare, yet predicting it correctly is much more important than being accurate with respect to class 0.

**Remark.** Think of medical applications, e.g., HIV testing or tumour diagnosis.

In such a case, *accuracy* as a metric fails to quantify what we are aiming for.

**Remark.** If only 1% of the cases have true  $y_i = 1$ , then a dummy classifier that always outputs  $\hat{y}_i = 0$  has 99% accuracy.

Metrics such as precision and recall (and their aggregated version, F-measure) aim to address this problem.

### Precision

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

If the classifier outputs 1, what is the probability that this is indeed true?

```
C[2,2]/(C[2,2]+C[2,1]) # Precision
```

```
## [1] 0.8310502
```

**Recall** (a.k.a. sensitivity, hit rate or true positive rate)

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

If the true class is 1, what is the probability that the classifier will detect it?

```
C[2,2]/(C[2,2]+C[1,2]) # Recall
```

```
## [1] 0.5741325
```

**Remark.** Precision or recall? It depends on an application. Think of medical diagnosis, medical screening, plagiarism detection, etc. — which measure is more important in each of the settings listed?

As a compromise, we can use the **F-measure** (a.k.a.  $F_1$ -measure), which is the harmonic mean of precision and recall:

$$F = \frac{1}{\frac{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}}{2}} = \left( \frac{1}{2} (\text{Precision}^{-1} + \text{Recall}^{-1}) \right)^{-1} = \frac{\text{TP}}{\text{TP} + \frac{\text{FP}+\text{FN}}{2}}$$

### Exercise.

Show that the above equality holds.

□

```
C[2,2]/(C[2,2]+0.5*C[1,2]+0.5*C[2,1]) # F
```

```
## [1] 0.6791045
```

The following function can come in handy in the future:

```
get_metrics <- function(Y_pred, Y_test)
{
  C <- table(Y_pred, Y_test) # confusion matrix
  stopifnot(dim(C) == c(2, 2))
  c(Acc=(C[1,1]+C[2,2])/sum(C), # accuracy
    Prec=C[2,2]/(C[2,2]+C[2,1]), # precision
    Rec=C[2,2]/(C[2,2]+C[1,2]), # recall
    F=C[2,2]/(C[2,2]+0.5*C[1,2]+0.5*C[2,1]), # F-measure
    # Confusion matrix items:
    TN=C[1,1], FN=C[1,2],
    FP=C[2,1], TP=C[2,2]
```

```

    ) # return a named vector
}

get_metrics(Y_pred, Y_test)

##          Acc        Prec        Rec         F
## 0.9122449  0.8310502  0.5741325  0.6791045
##          TN        FN        FP         TP
## 1606.0000000 135.0000000 37.0000000 182.0000000

```

### 3.3.2 How to Choose K for K-NN Classification?

We haven't yet considered the question which  $K$  yields *the best* classifier.

Best == one that has the highest *predictive power*.

Best == with respect to some chosen metric (accuracy, recall, precision, F-measure, ...)

Let's study how the metrics on the test set change as functions of the number of nearest neighbours considered,  $K$ .

Auxiliary function:

```
knn_metrics <- function(k, X_train, X_test, Y_train, Y_test)
{
  Y_pred <- knn(X_train, X_test, Y_train, k=k) # classify
  get_metrics(Y_pred, Y_test)
}
```

For example:

```
knn_metrics(5, Z_train, Z_test, Y_train, Y_test)

##          Acc        Prec        Rec         F
## 0.9102041  0.8025751  0.5899054  0.6800000
##          TN        FN        FP         TP
## 1597.0000000 130.0000000 46.0000000 187.0000000
```

Example call to evaluate metrics as a function of different  $K$ s:

```
Ks <- seq(1, 19, by=2)
Ps <- as.data.frame(t(
  sapply(Ks, # on each element in this vector
    knn_metrics,      # apply this function
    Z_train, Z_test, Y_train, Y_test # aux args
)))
```

**Remark.** Note that `sapply(X, f, arg1, arg2, ...)` outputs a list  $Y$  such that  $Y[[i]] = f(X[i], arg1, arg2, \dots)$  which is then simplified to a matrix.

**Remark.** We transpose this result, `t()`, in order to get each metric corresponding to different columns in the result. As usual, if you keep wondering, e.g., why `t()`, play with the code yourself – it's fun fun fun.

Example results:

```
round(cbind(K=Ks, Ps), 2)
```

	K	Acc	Prec	Rec	F	TN	FN	FP	TP
## 1	1	0.92	0.76	0.72	0.74	1573	90	70	227
## 2	3	0.92	0.79	0.66	0.72	1588	108	55	209
## 3	5	0.91	0.80	0.59	0.68	1597	130	46	187
## 4	7	0.91	0.82	0.56	0.67	1605	140	38	177
## 5	9	0.91	0.83	0.57	0.68	1606	135	37	182
## 6	11	0.91	0.84	0.57	0.68	1608	135	35	182
## 7	13	0.91	0.83	0.57	0.67	1605	137	38	180
## 8	15	0.91	0.82	0.55	0.66	1606	144	37	173
## 9	17	0.91	0.83	0.54	0.65	1607	147	36	170
## 10	19	0.90	0.80	0.52	0.63	1602	151	41	166

Figure 3.6 is worth a thousand tables though (see `?matplot` in R). The reader is kindly asked to draw conclusions themselves.

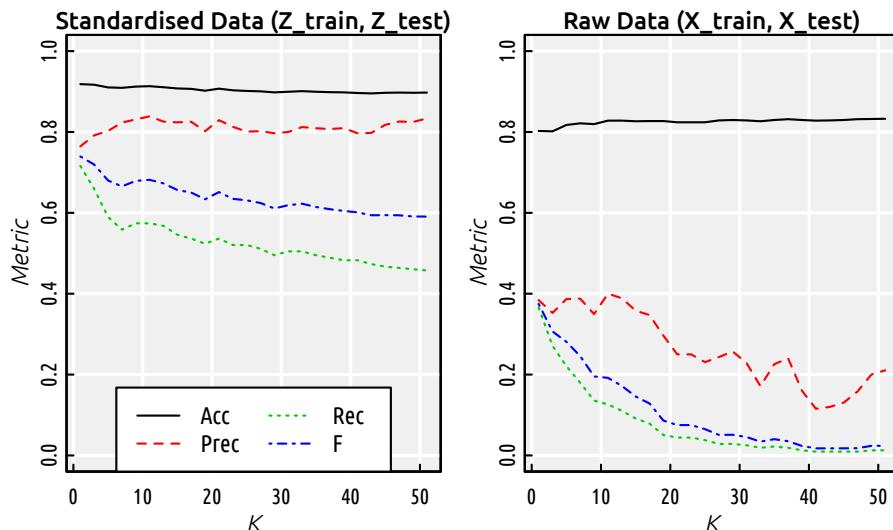


Figure 3.6: Performance of  $K$ -nn classifiers as a function of  $K$  for standardised and raw data

### 3.3.3 Training, Validation and Test sets

In the  $K$ -NN classification task, there are many hyperparameters to tune up:

- Which  $K$  should we choose?
- Should we standardise the dataset?
- Which variables should be taken into account when computing the Euclidean distance?

**Remark.** If we select the best hyperparameter set based on test sample error, we will run into the trap of overfitting again. This time we'll be overfitting to the test set — the model that is optimal for a given test sample doesn't have to generalise well to other test samples (!).

In order to overcome this problem, we can perform a random **train-validation-test split** of the original dataset:

- *training sample* (e.g., 60%) – used to construct the models
- *validation sample* (e.g., 20%) – used to tune the hyperparameters of the classifier
- *test sample* (e.g., 20%) – used to assess the goodness of fit

An example way to perform a 60/20/20% train-validation-test split:

```
set.seed(123) # reproducibility matters
random_indices <- sample(n)
n1 <- floor(n*0.6)
n2 <- floor(n*0.8)
X2_train <- X[random_indices[1:n1], ]
Y2_train <- Y[random_indices[1:n1] ]
X2_valid <- X[random_indices[(n1+1):n2], ]
Y2_valid <- Y[random_indices[(n1+1):n2] ]
X2_test <- X[random_indices[(n2+1):n], ]
Y2_test <- Y[random_indices[(n2+1):n] ]
stopifnot(nrow(X2_train)+nrow(X2_valid)+nrow(X2_test)
          == nrow(X))
```

**Exercise.**

Find the best  $K$  on the validation set and compute the error metrics on the test set.

□

**Remark.** (\*) If our dataset is too small, we can use various *cross-validation* techniques instead of a train-validate-test split.

## 3.4 Implementing a K-NN Classifier (\*)

### 3.4.1 Factor Data Type

Recall that (see Appendix B for more details) `factor` type in R is a very convenient means to encode categorical data (such as `y`):

```
x <- c("yes", "no", "no", "yes", "no")
f <- factor(x, levels=c("no", "yes"))
f

## [1] yes no  no  yes no
## Levels: no yes

table(f) # counts

## f
##  no yes
## 3   2
```

Internally, objects of type `factor` are represented as integer vectors with elements in  $\{1, \dots, M\}$ , where  $M$  is the number of possible levels.

Labels, used to “decipher” the numeric codes, are stored separately.

```
as.numeric(f) # 2nd label, 1st label, 1st label etc.

## [1] 2 1 1 2 1

levels(f)

## [1] "no"  "yes"
levels(f) <- c("failure", "success") # re-encode
f

## [1] success failure failure success failure
## Levels: failure success
```

### 3.4.2 Main Routine (\*)

Let’s implement a K-NN classifier ourselves by using a top-bottom approach.

We will start with a general description of the admissible inputs and the expected output.

Then we will arrange the processing of data into conveniently manageable chunks.

The function’s declaration will look like:

```
our_knn <- function(X_train, X_test, Y_train, k=1) {
  # k=1 denotes a parameter with a default value
```

```
# ...
}
```

Load an example dataset on which we will test our algorithm:

```
wines <- read.csv("datasets/winequality-all.csv", comment="#")
wines <- wines[wines$color == "white",]
X <- as.matrix(wines[,1:10])
Y <- factor(as.character(as.numeric(wines$alcohol >= 12)))
```

Note that Y is now a factor object.

Train-test split:

```
set.seed(123)
random_indices <- sample(n)
train_indices <- random_indices[1:floor(n*0.6)]
X_train <- X[train_indices,]
Y_train <- Y[train_indices]
X_test <- X[-train_indices,]
Y_test <- Y[-train_indices]
```

First, we should specify the type and form of the arguments we're expecting:

```
# this is the body of our_knn() - part 1
stopifnot(is.numeric(X_train), is.matrix(X_train))
stopifnot(is.numeric(X_test), is.matrix(X_test))
stopifnot(is.factor(Y_train))
stopifnot(ncol(X_train) == ncol(X_test))
stopifnot(nrow(X_train) == length(Y_train))
stopifnot(k >= 1)
n_train <- nrow(X_train)
n_test <- nrow(X_test)
p <- ncol(X_train)
M <- length(levels(Y_train))
```

Therefore,

$X_{\text{train}} \in \mathbb{R}^{n_{\text{train}} \times p}$ ,  $X_{\text{test}} \in \mathbb{R}^{n_{\text{test}} \times p}$  and  $Y_{\text{train}} \in \{1, \dots, M\}^{n_{\text{train}}}$

**Remark.** Recall that R factor objects are internally encoded as integer vectors.

Next, we will call the (to-be-done) function `our_get_knnx()`, which seeks nearest neighbours of all the points:

```
# our_get_knnx returns a matrix nn_indices of size n_test*k,
# where nn_indices[i, j] denotes the index of
# X_test[i, ]'s j-th nearest neighbour in X_train.
# (It is the point X_train[nn_indices[i, j], ].)
nn_indices <- our_get_knnx(X_train, X_test, k)
```

Then, for each point in `X_test`, we fetch the labels corresponding to its nearest neighbours and compute their mode:

```
Y_pred <- numeric(n_test) # vector of length n_test
# For now we will operate on the integer labels in {1,...,M}
Y_train_int <- as.numeric(Y_train)
for (i in 1:n_test) {
  # Get the labels of the NNs of the i-th point:
  nn_labels_i <- Y_train_int[nn_indices[i,]]
  # Compute the mode (majority vote):
  Y_pred[i] <- our_mode(nn_labels_i) # in {1,...,M}
}
```

Finally, we should convert the resulting integer vector to an object of type `factor`:

```
# Convert Y_pred to factor:
return(factor(Y_pred, labels=levels(Y_train)))
```

### 3.4.3 Mode

To implement the mode, we can use the `tabulate()` function.

**Exercise.**

*Read the function's man page, see `?tabulate`.*

□

For example:

```
tabulate(c(1, 2, 1, 1, 1, 5, 2))
## [1] 4 2 0 0 1
```

There might be multiple modes – in such a case, we should pick one at random.

For that, we can use the `sample()` function.

**Exercise.**

*Read the function's man page, see `?sample`. Note that its behaviour is different when it's first argument is a vector of length 1.*

□

An example implementation:

```
our_mode <- function(Y) {
  # tabulate() will take care of
  # checking the correctness of Y
```

```

t <- tabulate(Y)
mode_candidates <- which(t == max(t))
if (length(mode_candidates) == 1) return(mode_candidates)
else return(sample(mode_candidates, 1))
}

our_mode(c(1, 1, 1, 1))

## [1] 1
our_mode(c(2, 2, 2, 2))

## [1] 2
our_mode(c(3, 1, 3, 3))

## [1] 3
our_mode(c(1, 1, 3, 3, 2))

## [1] 3
our_mode(c(1, 1, 3, 3, 2))

## [1] 1

```

### 3.4.4 NN Search Routines (\*)

Last but not least, we should implement the `our_get_knnx()` function.

It is the function responsible for seeking the indices of nearest neighbours.

It turns out this function will actually constitute the K-NN classifier's performance bottleneck in case of big data samples.

```

# our_get_knnx returns a matrix nn_indices of size n_test*k,
# where nn_indices[i,j] denotes the index of
# X_test[i,]'s j-th nearest neighbour in X_train.
# (It is the point X_train[nn_indices[i,j],].)
our_get_knnx <- function(X_train, X_test, k) {
  # ...
}

```

A naive approach to `our_get_knnx()` relies on computing all pairwise distances, and sorting them.

```

our_get_knnx <- function(X_train, X_test, k) {
  n_test <- nrow(X_test)
  nn_indices <- matrix(NA_real_, nrow=n_test, ncol=k)
  for (i in 1:n_test) {
    d <- apply(X_train, 1, function(x)

```

```

        sqrt(sum((x-X_test[i,])^2)))
    # now d[j] is the distance
    # between X_train[j,] and X_test[i,]
    nn_indices[i,] <- order(d)[1:k]
}
nn_indices
}

```

A comparison with FNN:knn():

```

system.time(Ya <- knn(X_train, X_test, Y_train, k=5))

##    user  system elapsed
##  0.128   0.000   0.127

system.time(Yb <- our_knn(X_train, X_test, Y_train, k=5))

##    user  system elapsed
## 15.310   0.012 15.322
mean(Ya == Yb) # 1.0 on perfect match

## [1] 1

```

Both functions return identical results but our implementation is “slightly” slower.

FNN:knn() is efficiently written in C++, which is a compiled programming language.

R, on the other hand (just like Python and Matlab) is interpreted, therefore as a rule of thumb we should consider it an order of magnitude slower (see, however, the Julia language).

Let’s substitute our naive implementation with the equivalent one, but written in C++ (available in the FNN package).

**Remark.** (\*) Note that we can write a C++ implementation ourselves, see the Rcpp package for seamless R and C++ integration.

```

our_get_knnx <- function(X_train, X_test, k) {
  # this is used by our_knn()
  FNN::get.knnx(X_train, X_test, k, algorithm="brute")$nn.index
}
system.time(Ya <- knn(X_train, X_test, Y_train, k=5))

##    user  system elapsed
##  0.124   0.000   0.123

system.time(Yb <- our_knn(X_train, X_test, Y_train, k=5))

##    user  system elapsed

```

```

##   0.044   0.000   0.044
mean(Ya == Yb) # 1.0 on perfect match

## [1] 1

```

Note that our solution requires  $c \cdot n_{\text{test}} \cdot n_{\text{train}} \cdot p$  arithmetic operations for some  $c > 1$ . The overall cost of sorting is at least  $d \cdot n_{\text{test}} \cdot n_{\text{train}} \cdot \log n_{\text{train}}$  for some  $d > 1$ .

This does not scale well with both  $n_{\text{test}}$  and  $n_{\text{train}}$  (think – big data).

It turns out that there are special **spatial data structures** – such as *metric trees* – that aim to speed up searching for nearest neighbours in *low-dimensional spaces* (for small  $p$ ).

**Remark.** (\*) Searching in high-dimensional spaces is hard due to the so-called curse of dimensionality.

For example, `FNN::get.knnx()` also implements the so-called kd-trees.

```

library("microbenchmark")
test_speed <- function(n, p, k) {
  A <- matrix(runif(n*p), nrow=n, ncol=p)
  s <- summary(microbenchmark::microbenchmark(
    brute=FNN::get.knnx(A, A, k, algorithm="brute"),
    kd_tree=FNN::get.knnx(A, A, k, algorithm="kd_tree"),
    times=3
  ), unit="s")
  # minima of 3 time measurements:
  structure(s$min, names=as.character(s$expr))
}

test_speed(10000, 2, 5)

##      brute      kd_tree
## 0.29808951 0.01213134

test_speed(10000, 5, 5)

##      brute      kd_tree
## 0.43236915 0.06273565

test_speed(10000, 10, 5)

##      brute      kd_tree
## 0.6641034 0.6511161

test_speed(10000, 20, 5)

##      brute      kd_tree

```

```
## 1.279774 5.349479
```

### 3.4.5 Different Metrics (\*)

The Euclidean distance is just one particular example of many possible **metrics** (metric == a mathematical term, above we have used this term in a more relaxed fashion, when referring to accuracy etc.).

Mathematically, we say that  $d$  is a metric on a set  $X$  (e.g.,  $\mathbb{R}^p$ ), whenever it is a function  $d : X \times X \rightarrow [0, \infty]$  such that for all  $x, x', x'' \in X$ :

- $d(x, x') = 0$  if and only if  $x = x'$ ,
- $d(x, x') = d(x', x)$  (it is symmetric)
- $d(x, x'') \leq d(x, x') + d(x', x'')$  (it fulfils the triangle inequality)

**Remark.** (\*) Not all the properties are required in all the applications; sometimes we might need a few additional ones.

We can easily generalise the way we introduced the K-NN method to have a classifier that is based on a point's neighbourhood with respect to any metric.

Example metrics on  $\mathbb{R}^p$ :

- **Euclidean**

$$d_2(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\| = \|\mathbf{x} - \mathbf{x}'\|_2 = \sqrt{\sum_{i=1}^p (x_i - x'_i)^2}$$

- **Manhattan** (taxicab)

$$d_1(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_1 = \sum_{i=1}^p |x_i - x'_i|$$

- **Chebyshev** (maximum)

$$d_\infty(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_\infty = \max_{i=1, \dots, p} |x_i - x'_i|$$

We can define metrics on different spaces too.

For example, the **Levenshtein distance** is a popular choice for comparing character strings (also DNA sequences etc.)

It is an *edit distance* – it measures the minimal number of single-character insertions, deletions or substitutions to change one string into another.

For instance:

```
adist("happy", "nap")
```

```
##      [,1]
## [1,]    3
```

This is because we need 1 substitution and 2 deletions,

happy → nappy → napp → nap.

See also:

- the Hamming distance for categorical vectors (or strings of equal lengths),
- the Jaccard distance for sets,
- the Kendall tau rank distance for rankings.

Moreover, R package `stringdist` includes implementations of numerous string metrics.

## 3.5 Outro

### 3.5.1 Remarks

Note that K-NN is suitable for any kind of multiclass classification.

However, in practice it's pretty slow for larger datasets – to classify a single point we have to query the whole training set (which should be available at all times).

In the next part we will discuss some other well-known classifiers:

- *Decision trees*
- *Logistic regression*

### 3.5.2 Side Note: K-NN Regression

The K-Nearest Neighbour scheme is intuitively pleasing.

No wonder it has inspired a similar approach for solving a regression task.

In order to make a prediction for a new point  $\mathbf{x}'$ :

1. find the K-nearest neighbours of  $\mathbf{x}'$  amongst the points in the train set, denoted  $\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_K}$ ,
2. fetch the corresponding reference outputs  $y_{i_1}, \dots, y_{i_K}$ ,
3. return their arithmetic mean as a result,

$$\hat{y} = \frac{1}{K} \sum_{j=1}^K y_{i_j}.$$

Recall our modelling of the Credit Rating ( $Y$ ) as a function of the average Credit Card Balance ( $X$ ) based on the `ISLR::Credit` dataset.

```
library("ISLR") # Credit dataset
Xc <- as.matrix(as.numeric(Credit$Balance[Credit$Balance>0]))
Yc <- as.matrix(as.numeric(Credit$Rating[Credit$Balance>0]))
```

```
library("FNN") # knn.reg function
x <- as.matrix(seq(min(Xc), max(Xc), length.out=101))
y1 <- knn.reg(Xc, x, Yc, k=1)$pred
y5 <- knn.reg(Xc, x, Yc, k=5)$pred
y25 <- knn.reg(Xc, x, Yc, k=25)$pred
```

The three models are depicted in Figure 3.7. Again, the higher the  $K$ , the smoother the curve. On the other hand, for small  $K$  we adapt better to what's in a point's neighbourhood.

```
plot(Xc, Yc, col="#666666c0",
      xlab="Balance", ylab="Rating")
lines(x, y1, col=2, lwd=3)
lines(x, y5, col=3, lwd=3)
lines(x, y25, col=4, lwd=3)
legend("topleft", legend=c("K=1", "K=5", "K=25"),
       col=c(2, 3, 4), lwd=3, bg="white")
```

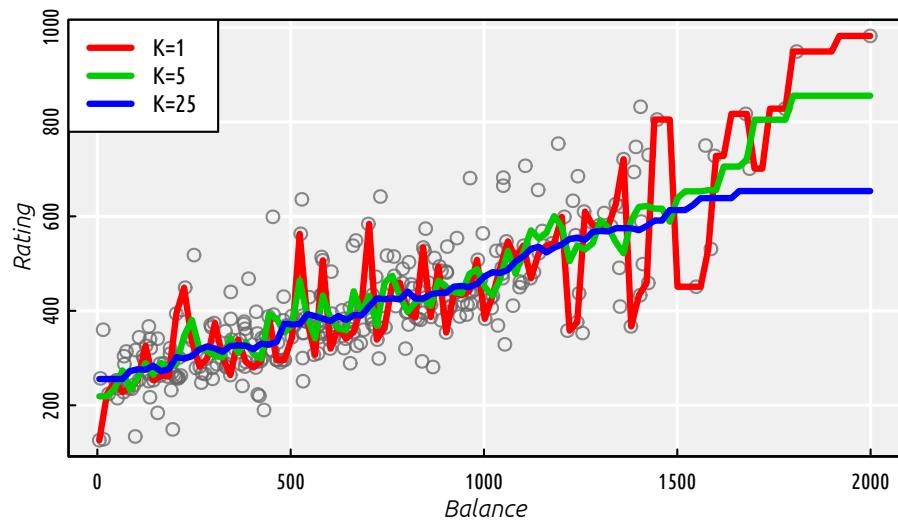


Figure 3.7: K-NN regression example

### 3.5.3 Further Reading

Recommended further reading: (Hastie et al. 2017: Section 13.3)

## Chapter 4

# Classification with Trees and Linear Models

### 4.1 Introduction

#### 4.1.1 Classification Task

Let  $\mathbf{X} \in \mathbb{R}^{n \times p}$  be an input matrix that consists of  $n$  points in a  $p$ -dimensional space (each of the  $n$  objects is described by means of  $p$  numerical features)

Recall that in supervised learning, with each  $\mathbf{x}_{i,:}$  we associate the desired output  $y_i$ .

Hence, our dataset is  $[\mathbf{X} \ \mathbf{y}]$  – where each object is represented as a row vector  $[\mathbf{x}_{i,:} \ y_i]$ ,  $i = 1, \dots, n$ :

$$[\mathbf{X} \ \mathbf{y}] = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} & y_1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} & y_n \end{bmatrix}.$$

In this chapter we are still interested in **classification** tasks; we assume that each  $y_i$  is a descriptive label.

Let's assume that we are faced with **binary classification** tasks.

Hence, there are only two possible labels that we traditionally denote with 0s and 1s.

For example:

0	1
no	yes
false	true
failure	success
healthy	ill

Let's recall the synthetic 2D dataset from the previous chapter (true decision boundary is at  $X_1 = 0$ ), see Figure 4.1.

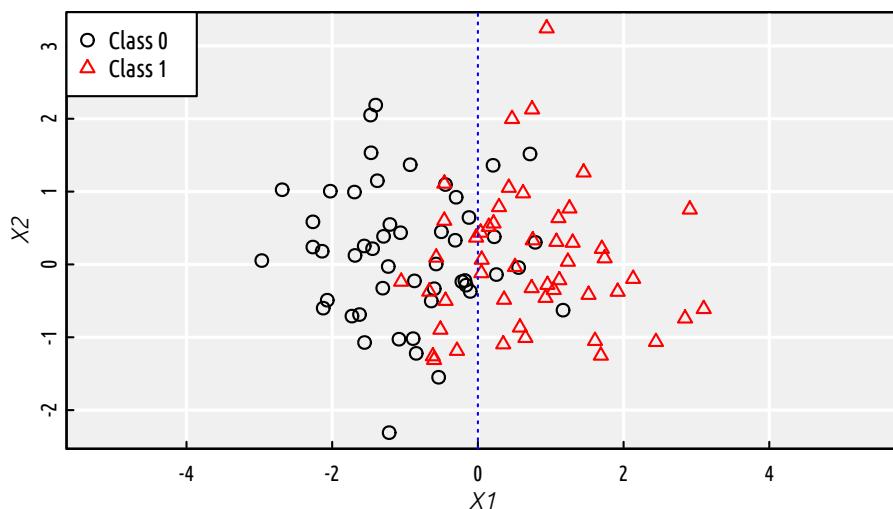


Figure 4.1: A synthetic 2D dataset with the true decision boundary at  $X_1 = 0$

### 4.1.2 Data

For illustration, we'll be considering the Wine Quality dataset (white wines only):

```
wines <- read.csv("datasets/winequality-all.csv", comment="#")
wines <- wines[wines$color == "white",]
(n <- nrow(wines)) # number of samples
```

```
## [1] 4898
```

The input matrix  $\mathbf{X} \in \mathbb{R}^{n \times p}$  consists of the first 10 numeric variables:

```
X <- as.matrix(wines[,1:10])
dim(X)
```

```
## [1] 4898    10
```

```
head(X, 2) # first two rows

##      fixed.acidity volatile.acidity citric.acid residual.sugar
## 1600        7.0          0.27       0.36        20.7
## 1601        6.3          0.30       0.34        1.6
##      chlorides free.sulfur.dioxide total.sulfur.dioxide density
## 1600     0.045           45         170    1.001
## 1601     0.049           14         132    0.994
##      pH sulphates
## 1600 3.0      0.45
## 1601 3.3      0.49
```

The 11th variable measures the amount of alcohol (in %).

We will convert this dependent variable to a binary one:

- 0 == (alcohol < 12) == lower-alcohol wines
- 1 == (alcohol >= 12) == higher-alcohol wines

```
# recall that TRUE == 1
Y <- factor(as.character(as.numeric(wines$alcohol >= 12)))
table(Y)
```

```
## Y
##   0   1
## 4085 813
```

60/40% train-test split:

```
set.seed(123) # reproducibility matters
random_indices <- sample(n)
head(random_indices) # preview

## [1] 2463 2511 2227 526 4291 2986

# first 60% of the indices (they are arranged randomly)
# will constitute the train sample:
train_indices <- random_indices[1:floor(n*0.6)]
X_train <- X[train_indices,]
Y_train <- Y[train_indices]
# the remaining indices (40%) go to the test sample:
X_test <- X[-train_indices,]
Y_test <- Y[-train_indices]
```

Let's also compute Z\_train and Z\_test, being the standardised versions of X\_train and X\_test, respectively.

```
means <- apply(X, 2, mean) # column means
sds   <- apply(X, 2, sd)   # column standard deviations
Z_train <- t(apply(X_train, 1, function(c) (c-means)/sds))
```

```
Z_test <- t(apply(X_test, 1, function(c) (c-means)/sds))

get_metrics <- function(Y_pred, Y_test)
{
  C <- table(Y_pred, Y_test) # confusion matrix
  stopifnot(dim(C) == c(2, 2))
  c(Acc=(C[1,1]+C[2,2])/sum(C), # accuracy
    Prec=C[2,2]/(C[2,2]+C[2,1]), # precision
    Rec=C[2,2]/(C[2,2]+C[1,2]), # recall
    F=C[2,2]/(C[2,2]+0.5*C[1,2]+0.5*C[2,1]), # F-measure
    # Confusion matrix items:
    TN=C[1,1], FN=C[1,2],
    FP=C[2,1], TP=C[2,2]
  ) # return a named vector
}
```

Let's go back to the K-NN algorithm.

```
library("FNN")
Y_knn5 <- knn(X_train, X_test, Y_train, k=5)
Y_knn9 <- knn(X_train, X_test, Y_train, k=9)
Y_knn5s <- knn(Z_train, Z_test, Y_train, k=5)
Y_knn9s <- knn(Z_train, Z_test, Y_train, k=9)
```

Recall the quality metrics we have obtained previously (as a point of reference):

```
cbind(
  Knn5=get_metrics(Y_knn5, Y_test),
  Knn9=get_metrics(Y_knn9, Y_test),
  Knn5s=get_metrics(Y_knn5s, Y_test),
  Knn9s=get_metrics(Y_knn9s, Y_test)
)

##          Knn5        Knn9       Knn5s       Knn9s
## Acc  0.8173469  0.8193878  0.9102041  0.9122449
## Prec 0.3867403  0.3495935  0.8025751  0.8310502
## Rec  0.2208202  0.1356467  0.5899054  0.5741325
## F   0.2811245  0.1954545  0.6800000  0.6791045
## TN  1532.0000000 1563.0000000 1597.0000000 1606.0000000
## FN  247.0000000 274.0000000 130.0000000 135.0000000
## FP  111.0000000 80.0000000 46.0000000 37.0000000
## TP  70.0000000 43.0000000 187.0000000 182.0000000
```

In this chapter we discuss the following simple and educational (yet practically useful) classification algorithms:

- *decision trees*,
- *logistic regression*.

## 4.2 Decision Trees

### 4.2.1 Introduction

Note that a K-NN classifier discussed in the previous chapter is **model-free**. The whole training set must be stored and referred to at all times.

Therefore, it doesn't *explain* the data we have – we may use it solely for the purpose of *prediction*.

Perhaps one of the most interpretable (and hence human-friendly) models consist of decision rules of the form:

**IF**  $x_{i,j_1} \leq v_1$  **AND** ... **AND**  $x_{i,j_r} \leq v_r$  **THEN**  $\hat{y}_i = 1$ .

These can be organised into a **hierarchy** for greater readability.

This idea inspired the notion of **decision trees** (Breiman et al. 1984).

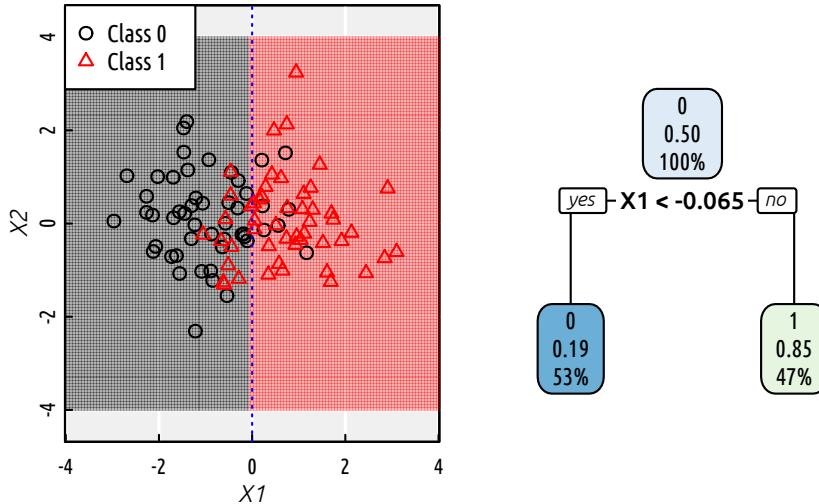


Figure 4.2: The simplest decision tree for the synthetic 2D dataset and the corresponding decision boundaries

Figure 4.3 depicts a very simple decision tree for the aforementioned synthetic dataset. There is only one decision boundary (based on  $X_1$ ) that splits data into the “left” and “right” sides. Each tree node reports 3 pieces of information:

- dominating class (0 or 1)
- (relative) proportion of 1s represented in a node
- (absolute) proportion of all observations in a node

Figures 4.3 and 4.4 depict trees with more decision rules. Take a moment to contemplate how the corresponding decision boundaries changed with the

introduction of new decision rules.

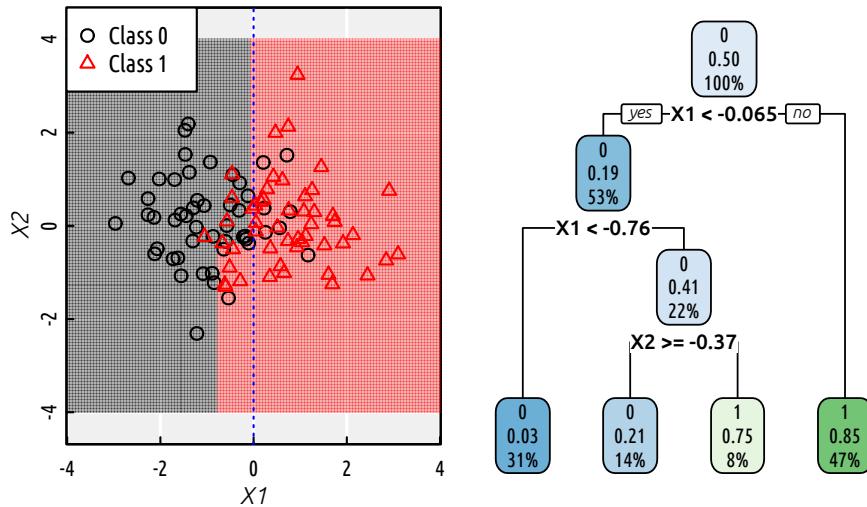


Figure 4.3: A more complicated decision tree for the synthetic 2D dataset and the corresponding decision boundaries

#### 4.2.2 Example in R

We will use the `rpart()` function from the `rpart` package to build a classification tree.

```
library("rpart")
library("rpart.plot")
set.seed(123)
```

`rpart()` uses a formula (`~`) interface, hence it will be easier to feed it with data in a `data.frame` form.

```
XY_train <- cbind(as.data.frame(X_train), Y=Y_train)
XY_test <- cbind(as.data.frame(X_test), Y=Y_test)
```

Fit and plot a decision tree, see Figure 4.5.

```
t1 <- rpart(Y~, data=XY_train, method="class")
rpart.plot(t1, tweak=1.1, fallen.leaves=FALSE, digits=3)
```

We can build less or more complex trees by playing with the `cp` parameter, see Figures 4.6 and 4.7.

```
# cp = complexity parameter, smaller → more complex tree
t2 <- rpart(Y~, data=XY_train, method="class", cp=0.1)
rpart.plot(t2, tweak=1.1, fallen.leaves=FALSE, digits=3)
```

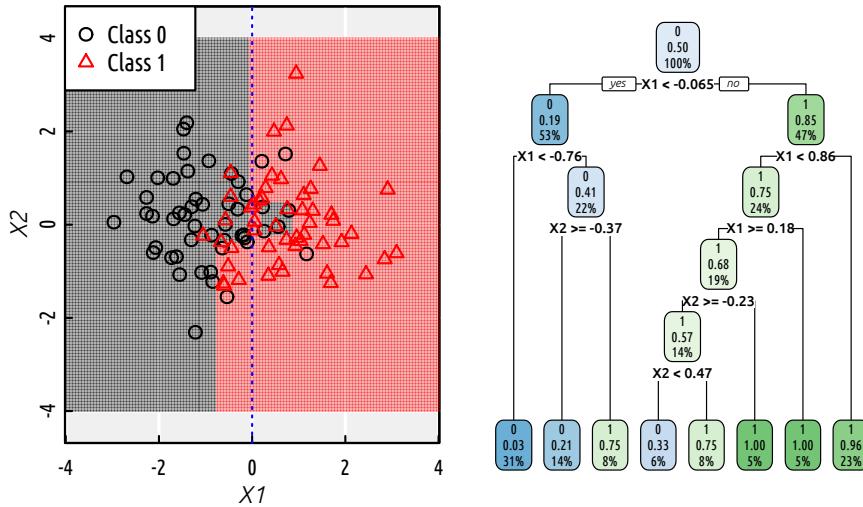


Figure 4.4: An even more complicated decision tree for the synthetic 2D dataset and the corresponding decision boundaries

```
# cp = complexity parameter, smaller → more complex tree
t3 <- rpart(Y~., data=XY_train, method="class", cp=0.00001)
rpart.plot(t3, tweak=1.1, fallen.leaves=FALSE, digits=3)
```

Trees with few decision rules actually are very nicely interpretable. On the other hand, plotting of the complex ones is just hopeless; we should treat them as “black boxes” instead.

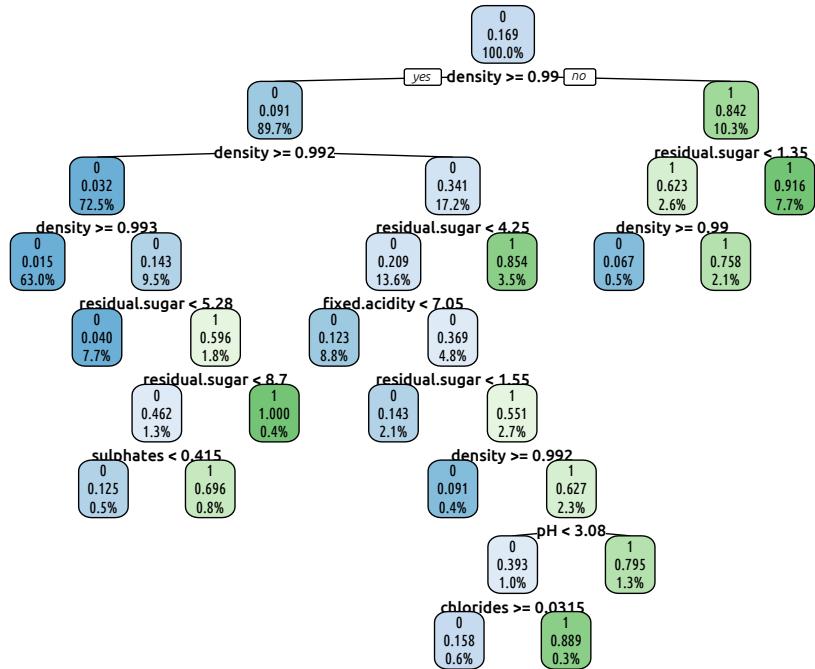
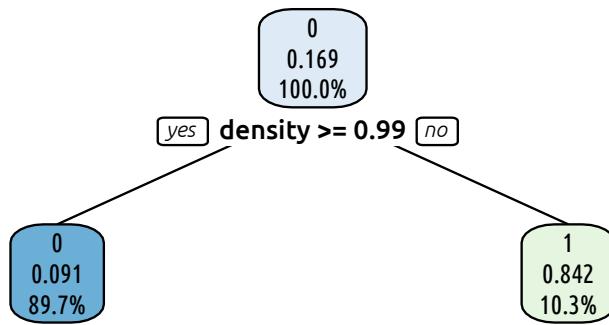
Let’s make some predictions:

```
Y_pred <- predict(t1, XY_test, type="class")
get_metrics(Y_pred, Y_test)
```

```
##          Acc       Prec       Rec        F
## 0.9285714 0.8062284 0.7350158 0.7689769
##      TN      FN      FP      TP
## 1587.0000000 84.0000000 56.0000000 233.0000000
```

```
Y_pred <- predict(t2, XY_test, type="class")
get_metrics(Y_pred, Y_test)
```

```
##          Acc       Prec       Rec        F
## 0.9025510 0.8387097 0.4921136 0.6202783
##      TN      FN      FP      TP
## 1613.0000000 161.0000000 30.0000000 156.0000000
```

Figure 4.5: A decision tree for the `wines` datasetFigure 4.6: A (simpler) decision tree for the `wines` dataset

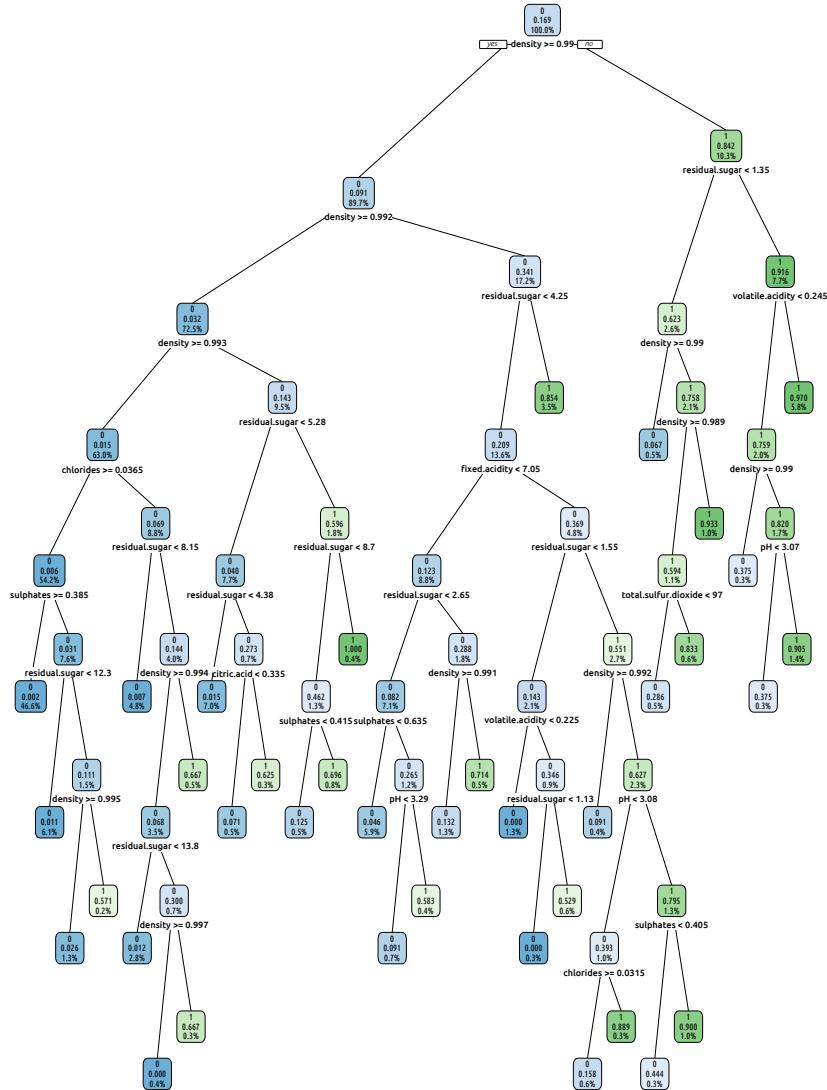


Figure 4.7: A (more complex) decision tree for the `wines` dataset

```
Y_pred <- predict(t3, XY_test, type="class")
get_metrics(Y_pred, Y_test)
```

```
##          Acc        Prec        Rec         F
## 0.9183673 0.7343284 0.7760252 0.7546012
##      TN       FN       FP       TP
## 1554.0000000 71.0000000 89.0000000 246.0000000
```

**Remark.** (\*) Interestingly, `rpart()` also provides us with information about the importance degrees of each independent variable.

```
t1$variable.importance/sum(t1$variable.importance)

##           density      residual.sugar      fixed.acidity
## 0.656248965 0.198422128 0.030516677
##      chlorides      pH      volatile.acidity
## 0.021500846 0.020967824 0.019288047
##      sulphates total.sulfur.dioxide citric.acid
## 0.018429283 0.014048245 0.011920068
## free.sulfur.dioxide
## 0.008657918
```

### 4.2.3 A Note on Decision Tree Learning

Learning an optimal decision tree is a computationally hard problem – we need some heuristics.

Examples:

- ID3 (Iterative Dichotomiser 3) (Quinlan 1986)
- C4.5 algorithm (Quinlan 1993)
- CART by Leo Breiman et al., (Breiman et al. 1984)

(\*\*) Decision trees are most often constructed by a *greedy, top-down recursive partitioning*, see., e.g., (Therneau & Atkinson 2019).

## 4.3 Binary Logistic Regression

### 4.3.1 Motivation

Recall that for a regression task, we fitted a very simple family of models – the linear ones – by minimising the sum of squared residuals.

This approach was pretty effective.

(Very) theoretically, we could treat the class labels as numeric 0s and 1s and apply regression models in a binary classification task.

```

XY_train_r <- cbind(as.data.frame(X_train),
  Y=as.numeric(Y_train)-1 # 0.0 or 1.0
)
XY_test_r <- cbind(as.data.frame(X_test),
  Y=as.numeric(Y_test)-1 # 0.0 or 1.0
)
f_r <- lm(Y~density+residual.sugar+pH, data=XY_train_r)

Y_pred_r <- predict(f_r, XY_test_r)
summary(Y_pred_r)

##      Min.    1st Qu.     Median      Mean    3rd Qu.      Max.
## -3.04678 -0.02106  0.11917  0.16452  0.34906  0.88921

```

The predicted outputs,  $\hat{Y}$ , are arbitrary real numbers, but we can convert them to binary ones by checking if, e.g.,  $\hat{Y} > 0.5$ .

```

Y_pred <- as.numeric(Y_pred_r>0.5)
round(get_metrics(Y_pred, XY_test_r$Y), 3)

##      Acc      Prec      Rec      F      TN      FN      FP
## 0.927  0.865  0.647  0.740 1611.000  112.000  32.000
##      TP
## 205.000

```

**Remark.** (\*) The threshold  $T = 0.5$  could even be treated as a free parameter we optimise for (w.r.t. different metrics over the validation sample), see Figure 4.8.

Despite we can, we shouldn't use linear regression for classification. Treating class labels "0" and "1" as ordinary real numbers just doesn't cut it – we intuitively feel that we are doing something *ugly*. Luckily, there is a better, more meaningful approach that still relies on a linear model, but has the *right* semantics.

### 4.3.2 Logistic Model

Inspired by this idea, we could try modelling the **probability that a given point belongs to class 1**.

This could also provide us with the *confidence* in our prediction.

Probability is a number in  $[0, 1]$ , but the outputs of a linear model are arbitrary real numbers.

However, we could transform those real-valued outputs by means of some function  $\phi : \mathbb{R} \rightarrow [0, 1]$  (preferably S-shaped == sigmoid), so as to get:

$$\Pr(Y = 1 | \mathbf{X}, \boldsymbol{\beta}) = \phi(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p)$$

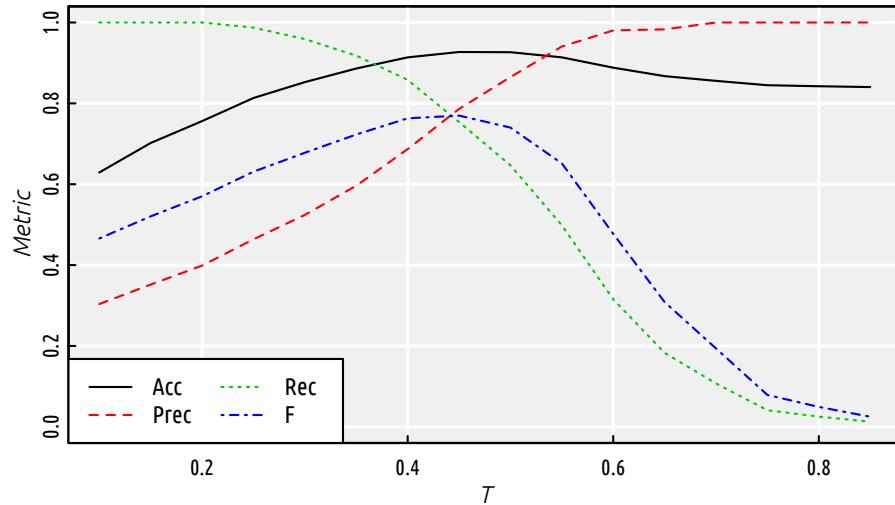


Figure 4.8: Quality metrics for a binary classifier “Classify  $X$  as 1 if  $f(X) > T$  and as 0 if  $f(X) \leq T$ ”

**Remark.** The above reads as “Probability that  $Y$  is from class 1 given  $\mathbf{X}$  and  $\beta$ ”.

A popular choice is the **logistic sigmoid function**, see Figure 4.9:

$$\phi(y) = \frac{1}{1 + e^{-y}} = \frac{e^y}{1 + e^y}$$

Hence our model becomes:

$$Y = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p)}}$$

We call it a **generalised linear model** (glm).

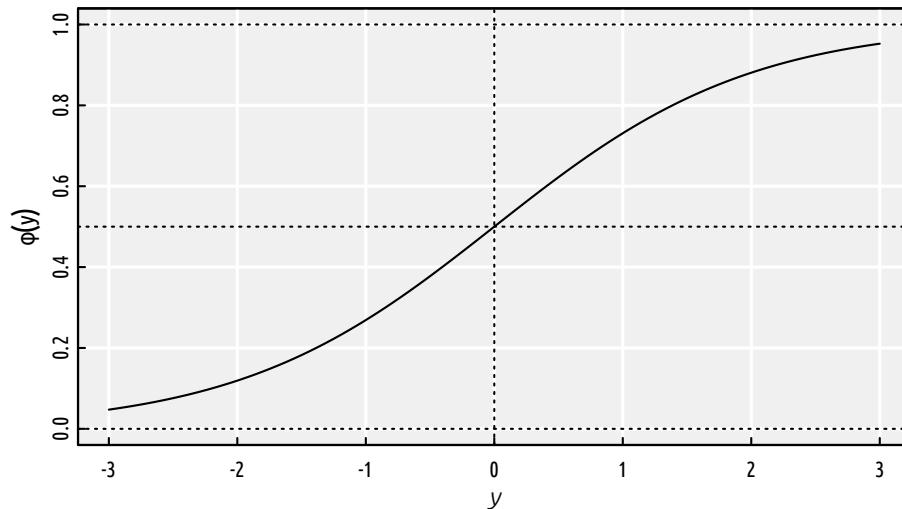
(There are many other possible generalisations)

### 4.3.3 Example in R

Let us first fit a simple (i.e.,  $p = 1$ ) logistic regression model using the `density` variable.

**Remark.** “logit” below denotes the inverse of the logistic sigmoid function.

```
f <- glm(Y~density, data=XY_train, family=binomial("logit"))
summary(f)
```

Figure 4.9: The logistic sigmoid function,  $\varphi$ 

```

## 
## Call:
## glm(formula = Y ~ density, family = binomial("logit"), data = XY_train)
## 
## Deviance Residuals:
##      Min        1Q     Median        3Q       Max
## -2.1591   -0.3653   -0.0934   -0.0154    4.0683
## 
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 1173.21     55.04   21.32 <2e-16 ***
## density     -1184.21     55.53  -21.33 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for binomial family taken to be 1)
## 
## Null deviance: 2667.8 on 2937 degrees of freedom
## Residual deviance: 1418.9 on 2936 degrees of freedom
## AIC: 1422.9
## 
## Number of Fisher Scoring iterations: 7

```

Figure 4.10 depicts the obtained probability  $\Pr(Y = 1|density)$ .

Some predicted probabilities:

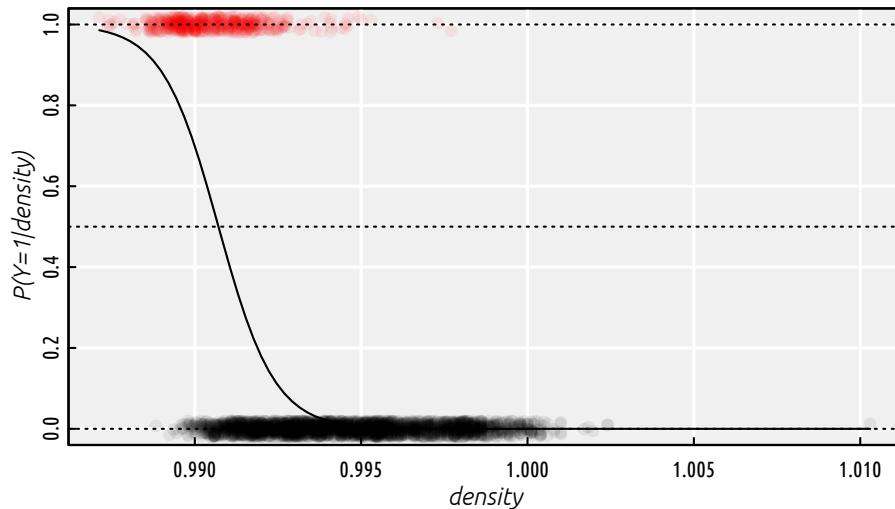


Figure 4.10: The probability that a given wine is a high-alcohol one given its density; black and red points denote the actual observed data points from the class 0 and 1, respectively

```
round(head(predict(f, XY_test, type="response")), 12), 2)
```

```
## 1602 1605 1607 1608 1609 1613 1614 1615 1621 1622 1623 1627
## 0.01 0.01 0.00 0.02 0.03 0.36 0.00 0.31 0.36 0.06 0.03 0.00
```

We classify  $Y$  as 1 if the corresponding membership probability is greater than 0.5.

```
Y_pred <- as.numeric(predict(f, XY_test, type="response")>0.5)
get_metrics(Y_pred, Y_test)
```

```
##          Acc      Prec      Rec      F
## 0.8979592 0.7276265 0.5899054 0.6515679
##      TN      FN      FP      TP
## 1573.0000000 130.0000000 70.0000000 187.0000000
```

And now a fit based on some other input variables:

```
f <- glm(Y~density+residual.sugar+total.sulfur.dioxide,
  data=XY_train, family=binomial("logit"))
Y_pred <- as.numeric(predict(f, XY_test, type="response")>0.5)
get_metrics(Y_pred, Y_test)
```

```
##          Acc      Prec      Rec      F
## 0.9321429 0.8239437 0.7381703 0.7787022
```

##	TN	FN	FP	TP
##	1593.0000000	83.0000000	50.0000000	234.0000000

**Exercise.**

*Try fitting the models based on different combinations of features.*

□

**4.3.4 Loss Function**

The fitting of the model can be written as an optimisation task:

$$\min_{\beta_0, \beta_1, \dots, \beta_p \in \mathbb{R}} \frac{1}{n} \sum_{i=1}^n e \left( \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p})}}, y_i \right)$$

where  $e(\hat{y}, y)$  denotes the penalty that measures the “difference” between the true  $y$  and its predicted version  $\hat{y}$ .

In the ordinary regression, we use the squared residual  $e(\hat{y}, y) = (\hat{y} - y)^2$ .

In **logistic regression** (the kind of a classifier we are interested in right now), we use the **cross-entropy** (a.k.a. **log-loss**),

$$e(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

The corresponding loss function has not only nice statistical properties (\*\*) but also an intuitive interpretation.

Note that the predicted  $\hat{y}$  is in  $(0, 1)$  and the true  $y$  equals to either 0 or 1.

Recall also that  $\log t \in (-\infty, 0)$  for  $t \in (0, 1)$ .

$$e(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

- if true  $y = 1$ , then the penalty becomes  $e(\hat{y}, 1) = -\log(\hat{y})$ 
  - $\hat{y}$  is the probability that the classified input is indeed from class 1
  - we'd be happy if the classifier outputted  $\hat{y} \simeq 1$  in this case; this is not penalised as  $-\log(t) \rightarrow 0$  as  $t \rightarrow 1$
  - however, if the classifier is totally wrong, i.e., it thinks that  $\hat{y} \simeq 0$ , then the penalty will be very high, as  $-\log(t) \rightarrow +\infty$  as  $t \rightarrow 0$
- if true  $y = 0$ , then the penalty becomes  $e(\hat{y}, 0) = -\log(1 - \hat{y})$ 
  - $1 - \hat{y}$  is the predicted probability that the input is from class 0
  - we penalise heavily the case where  $1 - \hat{y}$  is small (we'd be happy if the classifier was sure that  $1 - \hat{y} \simeq 1$ , because this is the ground-truth)

**Remark.** (\*) Interestingly, there is no analytical formula for the optimal set of parameters  $(\beta_0, \beta_1, \dots, \beta_p)$  minimising the log-loss.

**Remark.** In the chapter on optimisation, we shall see that the solution to the logistic regression can be solved numerically by means of quite simple iterative algorithms.

## 4.4 Outro

### 4.4.1 Remarks

Other prominent classification algorithms:

- Naive Bayes and other probabilistic approaches,
- Support Vector Machines (SVMs) and other kernel methods,
- (Artificial) (Deep) Neural Networks.

Interestingly, in the next chapter we will note that the logistic regression model is a special case of a *feed-forward single layer neural network*.

We will also generalise the binary logistic regression to the case of a multiclass classification.

The state-of-the art classifiers called *Random Forests* and *XGBoost* (see also: *AdaBoost*) are based on decision trees. They tend to be more accurate but – at the same time – they fail to exhibit the decision trees' important feature: interpretability.

Trees can also be used for regression tasks, see R package `rpart`.

### 4.4.2 Further Reading

Recommended further reading: (James et al. 2017: Chapters 4 and 8)

Other: (Hastie et al. 2017: Chapters 4 and 7 as well as (\*) Chapters 9, 10, 13, 15)

# Chapter 5

# Shallow and Deep Neural Networks

## 5.1 Introduction

### 5.1.1 Binary Logistic Regression: Recap

Let  $\mathbf{X} \in \mathbb{R}^{n \times p}$  be an input matrix that consists of  $n$  points in a  $p$ -dimensional space.

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix}$$

In other words, we have a database on  $n$  objects. Each object is described by means of  $p$  numerical features.

With each input  $\mathbf{x}_{i,\cdot}$ , we associate the desired output  $y_i$  which is a categorical label – hence we will be dealing with **classification** tasks again.

To recall, in **binary logistic regression** we model the probabilities that a given input belongs to either of the two classes:

$$\begin{aligned} \Pr(Y = 1 | \mathbf{X}, \boldsymbol{\beta}) &= \phi(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p) \\ \Pr(Y = 0 | \mathbf{X}, \boldsymbol{\beta}) &= 1 - \phi(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p) \end{aligned}$$

where  $\phi(z) = \frac{1}{1+e^{-z}}$  is the logistic sigmoid function.

It holds:

$$\Pr(Y = 1|\mathbf{X}, \boldsymbol{\beta}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p)}}$$

$$\Pr(Y = 0|\mathbf{X}, \boldsymbol{\beta}) = \frac{e^{-(\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p)}}{1 + e^{-(\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p)}}$$

The fitting of the model was performed by minimising the cross-entropy (log-loss):

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^{p+1}} -\frac{1}{n} \sum_{i=1}^n \left( y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right).$$

where  $\hat{y}_i = \Pr(Y = 1|\mathbf{x}_{i,.}, \boldsymbol{\beta})$ .

This is equivalent to:

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^{p+1}} -\frac{1}{n} \sum_{i=1}^n \left( y_i \log \Pr(Y = 1|\mathbf{x}_{i,.}, \boldsymbol{\beta}) + (1 - y_i) \log \Pr(Y = 0|\mathbf{x}_{i,.}, \boldsymbol{\beta}) \right).$$

Note that for each  $i$ , either the left or the right term (in the bracketed expression) vanishes.

Hence, we may also write the above as:

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^{p+1}} -\frac{1}{n} \sum_{i=1}^n \log \Pr(Y = y_i|\mathbf{x}_{i,.}, \boldsymbol{\beta}).$$

In this chapter we will generalise the binary logistic regression model:

- First we will consider the case of many classes (multiclass classification). This will lead to the multinomial logistic regression model.
- Then we will note that the multinomial logistic regression is a special case of a feed-forward neural network.

### 5.1.2 Data

We will study the famous classic – the MNIST image classification dataset (Modified National Institute of Standards and Technology database), see <http://yann.lecun.com/exdb/mnist/>

It consists of  $28 \times 28$  pixel images of handwritten digits:

- **train**: 60,000 training images,
- **t10k**: 10,000 testing images.

A few image instances from each class are depicted in Figure 5.1.

There are 10 unique digits, so this is a multiclass classification problem.



Figure 5.1: Example images in the MNIST database

**Remark.** The dataset is already “too easy” for testing of the state-of-the-art classifiers (see the notes below), but it’s a great educational example.

Accessing MNIST via the `keras` package (which we will use throughout this chapter anyway) is easy:

```
library("keras")
mnist <- dataset_mnist()
X_train <- mnist$train$x
Y_train <- mnist$train$y
X_test <- mnist$test$x
Y_test <- mnist$test$y
```

`X_train` and `X_test` consist of  $28 \times 28$  pixel images.

```
dim(X_train)

## [1] 60000    28    28

dim(X_test)

## [1] 10000    28    28
```

`X_train` and `X_test` are 3-dimensional arrays, think of them as vectors of 60000 and 10000 matrices of size  $28 \times 28$ , respectively.

These are grey-scale images, with 0 = black, ..., 255 = white:

```
range(X_train)

## [1] 0 255
```

Numerically, it's more convenient to work with colour values converted to  $0.0 = \text{black}, \dots, 1.0 = \text{white}$ :

```
X_train <- X_train/255
X_test  <- X_test/255
```

`Y_train` and `Y_test` are the corresponding integer labels:

```
length(Y_train)

## [1] 60000

length(Y_test)

## [1] 10000

table(Y_train) # label distribution in the training sample

## Y_train
##   0   1   2   3   4   5   6   7   8   9
## 5923 6742 5958 6131 5842 5421 5918 6265 5851 5949

table(Y_test) # label distribution in the test sample

## Y_test
##   0   1   2   3   4   5   6   7   8   9
## 980 1135 1032 1010 982 892 958 1028 974 1009
```

Here is how we can plot one of the digits (see Figure 5.2):

```
id <- 123 # image ID to show
image(z=t(X_train[id,,]), col=grey.colors(256, 0, 1),
      axes=FALSE, asp=1, ylim=c(1, 0))
legend("topleft", bg="white",
       legend=sprintf("True label=%d", Y_train[id]))
```

## 5.2 Multinomial Logistic Regression

### 5.2.1 A Note on Data Representation

So... you may now be wondering “how do we construct an image classifier, this seems so complicated!”.

For a computer, (almost) everything is just numbers.

Instead of playing with  $n$  matrices, each of size  $28 \times 28$ , we may “flatten” the images so as to get  $n$  “long” vectors of length  $p = 784$ .

```
X_train2 <- matrix(X_train, ncol=28*28)
X_test2  <- matrix(X_test, ncol=28*28)
```

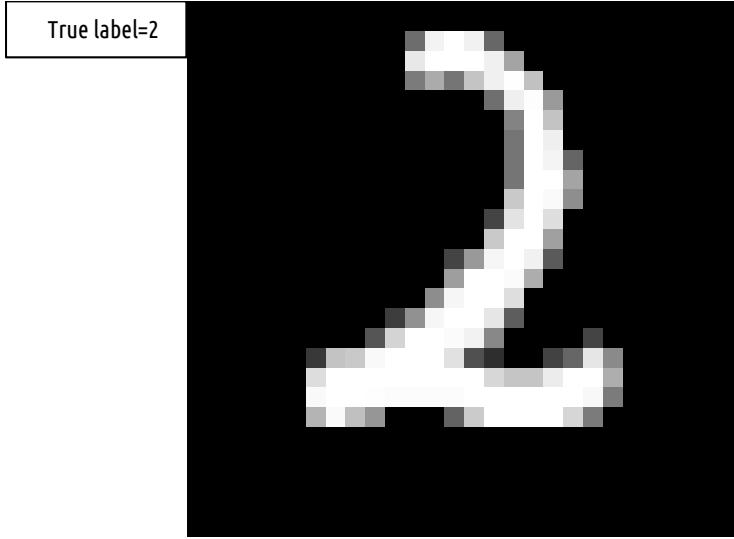


Figure 5.2: Example image from the MNIST dataset

The classifiers studied here do not take the “spatial” positioning of the pixels into account anyway. Hence, now we’re back to our “comfort zone”.

**Remark.** (\*) See, however, convolutional neural networks (CNNs), e.g., in (Goodfellow et al. 2016).

### 5.2.2 Extending Logistic Regression

Let us generalise the binary logistic regression model to a 10-class one (or, more generally,  $K$ -class one).

This time we will be modelling ten probabilities, with  $\Pr(Y = k|\mathbf{X}, \mathbf{B})$  denoting the *confidence* that a given image  $\mathbf{X}$  is in fact the  $k$ -th digit:

$$\begin{aligned} \Pr(Y = 0|\mathbf{X}, \mathbf{B}) &= \dots \\ \Pr(Y = 1|\mathbf{X}, \mathbf{B}) &= \dots \\ &\vdots \\ \Pr(Y = 9|\mathbf{X}, \mathbf{B}) &= \dots \end{aligned}$$

where  $\mathbf{B}$  is the set of underlying model parameters (to be determined soon).

In binary logistic regression, the class probabilities are obtained by “cleverly normalising” (by means of the logistic sigmoid) the outputs of a linear model (so that we obtain a value in  $[0, 1]$ ).

$$\Pr(Y = 1 | \mathbf{X}, \boldsymbol{\beta}) = \phi(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p)}}$$

In the multinomial case, we can use a separate linear model for each digit so that each  $\Pr(Y = k | \mathbf{X}, \mathbf{B})$ ,  $k = 0, 1, \dots, 9$ , is given as a function of:

$$\beta_{0,k} + \beta_{1,k} X_1 + \cdots + \beta_{p,k} X_p.$$

Therefore, instead of a parameter vector of length  $(p + 1)$ , we will need a parameter matrix of size  $(p + 1) \times 10$  representing the model's definition.

**Side note.** The upper case of  $\beta$  is  $B$ .

Then, these 10 numbers will have to be normalised so as to they are all greater than 0 and sum to 1.

To maintain the spirit of the original model, we can apply  $e^{-(\beta_{0,k} + \beta_{1,k} X_1 + \cdots + \beta_{p,k} X_p)}$  to get a positive value, because the co-domain of the exponential function  $t \mapsto e^t$  is  $(0, \infty)$ .

Then, dividing each output by the sum of all the outputs will guarantee that the total sum equals 1.

This leads to:

$$\begin{aligned}\Pr(Y = 0 | \mathbf{X}, \mathbf{B}) &= \frac{e^{-(\beta_{0,0} + \beta_{1,0} X_1 + \cdots + \beta_{p,0} X_p)}}{\sum_{k=0}^9 e^{-(\beta_{0,k} + \beta_{1,k} X_1 + \cdots + \beta_{p,k} X_p)}}, \\ \Pr(Y = 1 | \mathbf{X}, \mathbf{B}) &= \frac{e^{-(\beta_{0,1} + \beta_{1,1} X_1 + \cdots + \beta_{p,1} X_p)}}{\sum_{k=0}^9 e^{-(\beta_{0,k} + \beta_{1,k} X_1 + \cdots + \beta_{p,k} X_p)}}, \\ &\vdots \\ \Pr(Y = 9 | \mathbf{X}, \mathbf{B}) &= \frac{e^{-(\beta_{0,9} + \beta_{1,9} X_1 + \cdots + \beta_{p,9} X_p)}}{\sum_{k=0}^9 e^{-(\beta_{0,k} + \beta_{1,k} X_1 + \cdots + \beta_{p,k} X_p)}}.\end{aligned}$$

This reduces to the binary logistic regression if we consider only the classes 0 and 1 and fix  $\beta_{0,0} = \beta_{1,0} = \cdots = \beta_{p,0} = 0$  (as  $e^0 = 1$ ).

### 5.2.3 Softmax Function

The above transformation (that maps 10 arbitrary real numbers to positive ones that sum to 1) is called the **softmax** function (or *softargmax*).

```
softmax <- function(T) {
  T2 <- exp(T) # ignore the minus sign above
  T2/sum(T2)
}
round(rbind(
```

```

softmax(c(0, 0, 10, 0, 0, 0, 0, 0, 0, 0)),
softmax(c(0, 0, 10, 0, 0, 0, 10, 0, 0, 0)),
softmax(c(0, 0, 10, 0, 0, 0, 9, 0, 0, 0)),
softmax(c(0, 0, 10, 0, 0, 0, 9, 0, 0, 8))), 2)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0 1.00    0    0    0 0.00    0    0    0.00
## [2,]    0    0 0.50    0    0    0 0.50    0    0    0.00
## [3,]    0    0 0.73    0    0    0 0.27    0    0    0.00
## [4,]    0    0 0.67    0    0    0 0.24    0    0    0.09

```

### 5.2.4 One-Hot Encoding and Decoding

The ten class-belongingness-degrees can be decoded to obtain a single label by simply choosing the class that is assigned the highest probability.

```

y_pred <- softmax(c(0, 0, 10, 0, 0, 9, 0, 0, 8))
round(y_pred, 2) # probabilities of Y=0, 1, 2, ..., 9

## [1] 0.00 0.00 0.67 0.00 0.00 0.00 0.24 0.00 0.00 0.09
which.max(y_pred)-1 # 1..10 -> 0..9

## [1] 2

```

**Remark.** `which.max(y)` returns an index  $k$  such that  $y[k]==\max(y)$  (recall that in R the first element in a vector is at index 1). Mathematically, we denote this operation as  $\arg \max_{k=1,\dots,K} y_k$ .

To make processing the outputs of a logistic regression model more convenient, we will apply the so-called **one-hot-encoding** of the labels.

Here, each label will be represented as a 0-1 vector of 10 probabilities – with probability 1 corresponding to the true class only.

For instance:

```

y <- 2 # true class (this is just an example)
y2 <- rep(0, 10)
y2[y+1] <- 1 # +1 because we need 0..9 -> 1..10
y2 # one-hot-encoded y

## [1] 0 0 1 0 0 0 0 0 0 0

```

To one-hot encode *all* the reference outputs in R, we start with a matrix of size  $n \times 10$  populated with “0”s:

```

Y_train2 <- matrix(0, nrow=length(Y_train), ncol=10)

```

Next, for every  $i$ , we insert a “1” in the  $i$ -th row and the  $(Y\_train[i]+1)$ -th column:

```
# Note the "+1" 0..9 -> 1..10
Y_train2[cbind(1:length(Y_train), Y_train+1)] <- 1
```

**Remark.** In R, indexing a matrix A with a 2-column matrix B, i.e.,  $A[B]$ , allows for an easy access to  $A[B[1,1], B[1,2]]$ ,  $A[B[2,1], B[2,2]]$ ,  $A[B[3,1], B[3,2]]$ , ...

Sanity check:

```
head(Y_train)
```

```
## [1] 5 0 4 1 9 2
```

```
head(Y_train2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]     0     0     0     0     0     1     0     0     0     0
## [2,]     1     0     0     0     0     0     0     0     0     0
## [3,]     0     0     0     0     1     0     0     0     0     0
## [4,]     0     1     0     0     0     0     0     0     0     0
## [5,]     0     0     0     0     0     0     0     0     0     1
## [6,]     0     0     1     0     0     0     0     0     0     0
```

Let us generalise the above idea and write a function that can one-hot-encode any vector of integer labels:

```
one_hot_encode <- function(Y) {
  stopifnot(is.numeric(Y))
  c1 <- min(Y) # first class label
  cK <- max(Y) # last class label
  K <- cK-c1+1 # number of classes

  Y2 <- matrix(0, nrow=length(Y), ncol=K)
  Y2[cbind(1:length(Y), Y-c1+1)] <- 1
  Y2
}
```

Encode  $Y_{\text{train}}$  and  $Y_{\text{test}}$ :

```
Y_train2 <- one_hot_encode(Y_train)
Y_test2 <- one_hot_encode(Y_test)
```

### 5.2.5 Cross-entropy Revisited

Our classifier will be outputting  $K = 10$  probabilities.

The true class labels are not one-hot-encoded so that they are represented as vectors of  $K - 1$  zeros and a single one.

How to measure the “agreement” between these two?

In essence, we will be comparing the probability vectors as generated by a classifier,  $\hat{Y}$ :

```
round(y_pred, 2)
## [1] 0.00 0.00 0.67 0.00 0.00 0.00 0.24 0.00 0.00 0.09
```

with the one-hot-encoded true probabilities,  $Y$ :

```
y2
## [1] 0 0 1 0 0 0 0 0 0 0
```

It turns out that one of the definitions of cross-entropy introduced above already handles the case of multiclass classification:

$$E(\mathbf{B}) = -\frac{1}{n} \sum_{i=1}^n \log \Pr(Y = y_i | \mathbf{x}_{i,.}, \mathbf{B}).$$

The smaller the probability corresponding to the ground-truth class outputted by the classifier, the higher the penalty, see Figure 5.3.

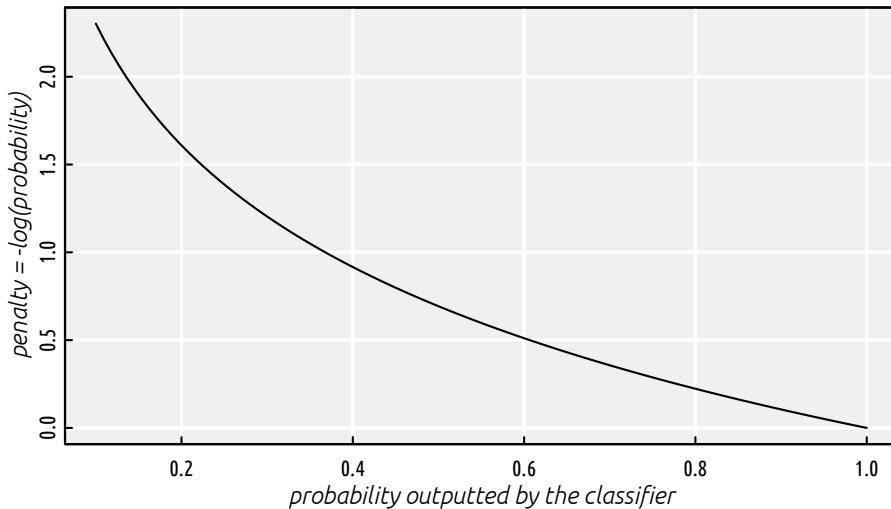


Figure 5.3: The less the classifier is confident about the prediction of the actually true label, the greater the penalty

To sum up, we will be solving the optimisation problem:

$$\min_{\mathbf{B} \in \mathbb{R}^{(p+1) \times 10}} -\frac{1}{n} \sum_{i=1}^n \log \Pr(Y = y_i | \mathbf{x}_{i,.}, \mathbf{B}).$$

This has no analytical solution, but can be solved using iterative methods (see the chapter on optimisation).

(\*) Side note: A single term in the above formula,

$$\log \Pr(Y = y_i | \mathbf{x}_{i,.}, \mathbf{B})$$

given:

- `y_pred` – a vector of 10 probabilities generated by the model:

$$[\Pr(Y = 0 | \mathbf{x}_{i,.}, \mathbf{B}) \ \Pr(Y = 1 | \mathbf{x}_{i,.}, \mathbf{B}) \ \cdots \ \Pr(Y = 9 | \mathbf{x}_{i,.}, \mathbf{B})]$$

- `y2` – a one-hot-encoded version of the true label,  $y_i$ , of the form:

$$[0 \ 0 \ \cdots \ 0 \ 1 \ 0 \ \cdots \ 0]$$

can be computed as:

```
sum(y2*log(y_pred))
```

```
## [1] -0.4078174
```

### 5.2.6 Problem Formulation in Matrix Form (\*\*)

The definition of a multinomial logistic regression model for a multiclass classification task involving classes  $\{1, 2, \dots, K\}$  is slightly bloated.

Assuming that  $\mathbf{X} \in \mathbb{R}^{n \times p}$  is the input matrix, to compute the  $K$  predicted probabilities for the  $i$ -th input,

$$[\hat{y}_{i,1} \ \hat{y}_{i,2} \ \cdots \ \hat{y}_{i,K}],$$

given a parameter matrix  $\mathbf{B}^{(p+1) \times K}$ , we apply:

$$\begin{aligned} \hat{y}_{i,1} = \Pr(Y = 1 | \mathbf{x}_{i,.}, \mathbf{B}) &= \frac{e^{\beta_{0,1} + \beta_{1,1}x_{i,1} + \cdots + \beta_{p,1}x_{i,p}}}{\sum_{k=1}^K e^{\beta_{0,k} + \beta_{1,k}x_{i,1} + \cdots + \beta_{p,k}x_{i,p}}}, \\ &\vdots \\ \hat{y}_{i,K} = \Pr(Y = K | \mathbf{x}_{i,.}, \mathbf{B}) &= \frac{e^{\beta_{0,K} + \beta_{1,K}x_{i,1} + \cdots + \beta_{p,K}x_{i,p}}}{\sum_{k=1}^K e^{\beta_{0,k} + \beta_{1,k}x_{i,1} + \cdots + \beta_{p,k}x_{i,p}}}. \end{aligned}$$

**Remark.** We have dropped the minus sign in the exponentiation for brevity of notation. Note that we can always map  $b'_{j,k} = -b_{j,k}$ .

It turns out we can make use of matrix notation to tidy the above formulas.

Denote the linear combinations prior to computing the softmax function with:

$$\begin{aligned} t_{i,1} &= \beta_{0,1} + \beta_{1,1}x_{i,1} + \cdots + \beta_{p,1}x_{i,p}, \\ &\vdots \\ t_{i,K} &= \beta_{0,K} + \beta_{1,K}x_{i,1} + \cdots + \beta_{p,K}x_{i,p}. \end{aligned}$$

We have:

- $x_{i,j}$  – the  $i$ -th observation, the  $j$ -th feature;
- $\hat{y}_{i,k}$  – the  $i$ -th observation, the  $k$ -th class probability;
- $\beta_{j,k}$  – the coefficient for the  $j$ -th feature when computing the  $k$ -th class.

Note that by augmenting  $\dot{\mathbf{X}} = [\mathbf{1} \ \mathbf{X}] \in \mathbb{R}^{n \times (p+1)}$  by adding a column of 1s, i.e., where  $\dot{x}_{i,0} = 1$  and  $\dot{x}_{i,j} = x_{i,j}$  for all  $j \geq 1$  and all  $i$ , we can write the above as:

$$\begin{aligned} t_{i,1} &= \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,1} = \dot{\mathbf{x}}_{i,\cdot} \boldsymbol{\beta}_{\cdot,1}, \\ &\vdots \\ t_{i,K} &= \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,K} = \dot{\mathbf{x}}_{i,\cdot} \boldsymbol{\beta}_{\cdot,K}. \end{aligned}$$

We can get the  $K$  linear combinations all at once in the form of a row vector by writing:

$$[t_{i,1} \ t_{i,2} \ \cdots \ t_{i,K}] = \mathbf{x}_{i,\cdot} \mathbf{B}.$$

Moreover, we can do that for all the  $n$  inputs by writing:

$$\mathbf{T} = \dot{\mathbf{X}} \mathbf{B}.$$

Yes yes yes! This is a single matrix multiplication, we have  $\mathbf{T} \in \mathbb{R}^{n \times K}$ .

To obtain  $\hat{\mathbf{Y}}$ , we have to apply the softmax function on every row of  $\mathbf{T}$ :

$$\hat{\mathbf{Y}} = \text{softmax}(\dot{\mathbf{X}} \mathbf{B}).$$

That's it. Take some time to appreciate the elegance of this notation.

Methods for minimising cross-entropy expressed in matrix form will be discussed in the next chapter.

## 5.3 Artificial Neural Networks

### 5.3.1 Artificial Neuron

A neuron can be thought of as a mathematical function, see Figure 5.4, which has its specific inputs and an output.

The Linear Threshold Unit (McCulloch and Pitts, 1940s), the Perceptron (Rosenblatt, 1958) and the Adaptive Linear Neuron (Widrow and Hoff, 1960) were amongst the first models of an artificial neuron that could be used for the purpose of pattern recognition, see Figure 5.5. They can be thought of as processing units that compute a weighted sum of the inputs, which is then transformed by means of a nonlinear “activation” function.

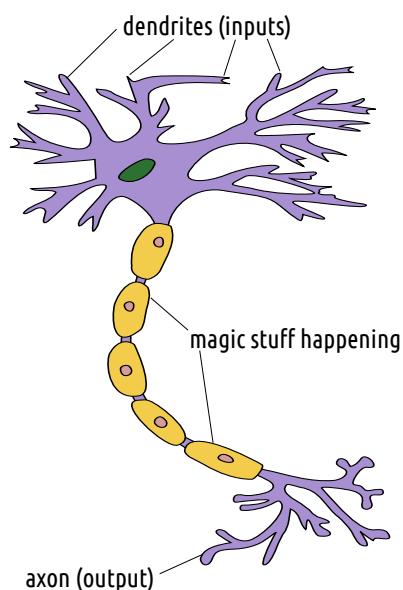


Figure 5.4: Neuron as a mathematical (black box) function; image based on: <https://en.wikipedia.org/wiki/File:Neuron3.png> by Egm4313.s12 at English Wikipedia, licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license

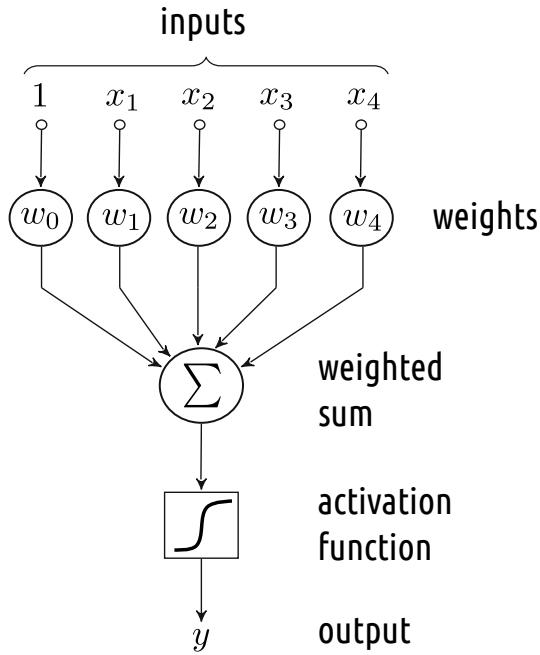


Figure 5.5: A simple model of an artificial neuron

### 5.3.2 Logistic Regression as a Neural Network

The above resembles our binary logistic regression model, where we determine a linear combination (a weighted sum) of  $p$  inputs and then transform it using the logistic sigmoid “activation” function. We can easily depict it in the Figure 5.4-style, see Figure 5.6.

On the other hand, a multiclass logistic regression can be depicted as in Figure 5.7. In fact, we can consider it as an instance of a:

- **single layer** (there is only one processing step that consists of 10 units),
- **densely connected** (all the inputs are connected to all the components below),
- **feed-forward** (the outputs are generated by processing the inputs from “top” to “bottom”, there are no loops in the graph etc.)

*artificial neural network* that uses the softmax as the activation function.

### 5.3.3 Example in R

To train such a neural network (i.e., fit a multinomial logistic regression model), we will use the `keras` package, a wrapper around the (GPU-enabled) TensorFlow library.

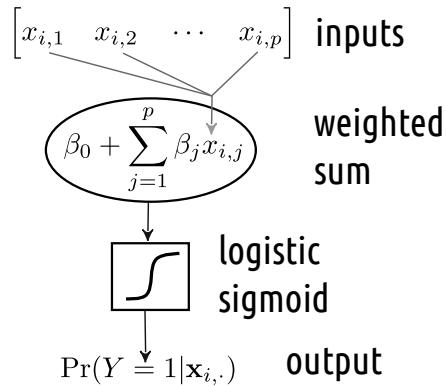


Figure 5.6: Binary logistic regression

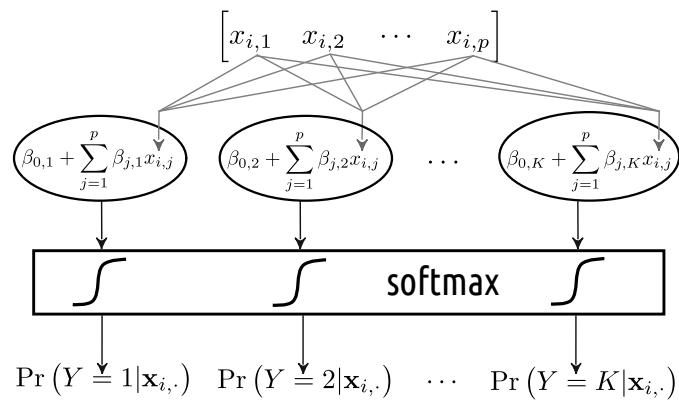


Figure 5.7: Multinomial logistic regression

The training of the model takes a few minutes (for more complex models and bigger datasets – it could take hours/days). Thus, it is wise to store the computed model (the **B** coefficient matrix and the accompanying `keras`'s auxiliary data) for further reference:

```
file_name <- "datasets/mnist_keras_model1.h5"
if (!file.exists(file_name)) { # File doesn't exist -> compute
  set.seed(123)
  # Start with an empty model
  model1 <- keras_model_sequential()
  # Add a single layer with 10 units and softmax activation
  layer_dense(model1, units=10, activation="softmax")
  # We will be minimising the cross-entropy,
  # sgd == stochastic gradient descent, see the next chapter
  compile(model1, optimizer="sgd",
          loss="categorical_crossentropy")
  # Fit the model (slooooow!)
  fit(model1, X_train2, Y_train2, epochs=10)
  # Save the model for future reference
  save_model_hdf5(model1, file_name)
} else { # File exists -> reload the model
  model1 <- load_model_hdf5(file_name)
}
```

Let's make predictions over the test set:

```
Y_pred2 <- predict(model1, X_test2)
round(head(Y_pred2), 2) # predicted class probabilities

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,] 0.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00
## [2,] 0.01 0.00 0.93 0.01 0.00 0.01 0.04 0.00 0.00 0.00
## [3,] 0.00 0.96 0.02 0.00 0.00 0.00 0.00 0.00 0.01 0.00
## [4,] 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
## [5,] 0.00 0.00 0.01 0.00 0.91 0.00 0.01 0.01 0.01 0.05
## [6,] 0.00 0.98 0.00 0.00 0.00 0.00 0.00 0.00 0.01 0.00
```

Then, we can one-hot-decode the output probabilities:

```
Y_pred <- apply(Y_pred2, 1, which.max)-1 # 1..10 -> 0..9
head(Y_pred, 20) # predicted outputs

## [1] 7 2 1 0 4 1 4 9 6 9 0 6 9 0 1 5 9 7 3 4
head(Y_test, 20) # true outputs
```

```
## [1] 7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4
```

Accuracy on the test set:

```
mean(Y_test == Y_pred)
```

```
## [1] 0.9169
```

Performance metrics for each digit separately (see also Figure 5.8):

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9924	0.9466403	0.9775510	0.9618474	8966	22	54	958
1	0.9923	0.9592014	0.9735683	0.9663314	8818	30	47	1105
2	0.9803	0.9221436	0.8837209	0.9025235	8891	120	77	912
3	0.9802	0.8941748	0.9118812	0.9029412	8881	89	109	921
4	0.9833	0.9014778	0.9317719	0.9163746	8918	67	100	915
5	0.9793	0.9141475	0.8475336	0.8795812	9037	136	71	756
6	0.9885	0.9314227	0.9498956	0.9405685	8975	48	67	910
7	0.9834	0.9284294	0.9085603	0.9183874	8900	94	72	934
8	0.9754	0.8647295	0.8860370	0.8752535	8891	111	135	863
9	0.9787	0.9004024	0.8870168	0.8936595	8892	114	99	895

Note how misleading the individual accuracies are! Averaging over the above table's columns gives:

```
##      Acc      Prec      Rec      F
## 0.9833800 0.9162769 0.9157537 0.9157468
```

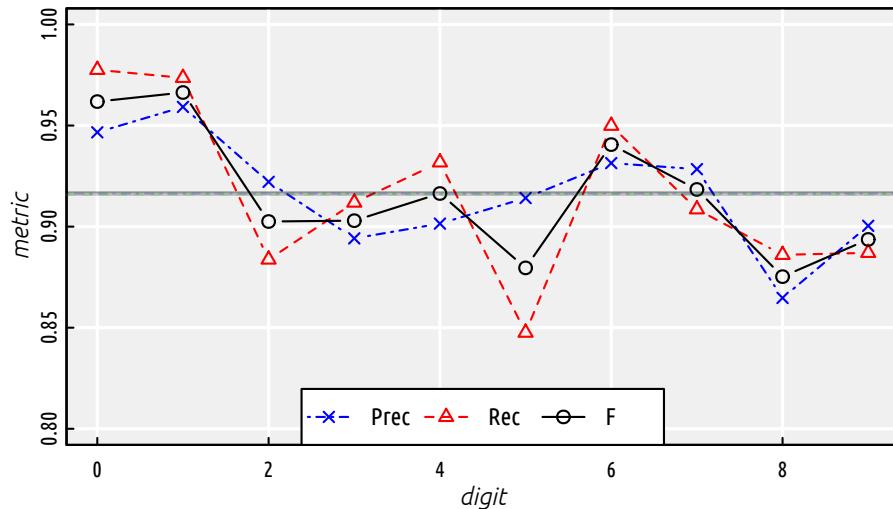


Figure 5.8: Performance metrics for multinomial logistic regression on MNIST

## 5.4 Deep Neural Networks

### 5.4.1 Introduction

In a brain, a neuron's output is an input to a bunch of other neurons. We could try aligning neurons into many interconnected layers. This leads to a structure like the one in Figure 5.9.

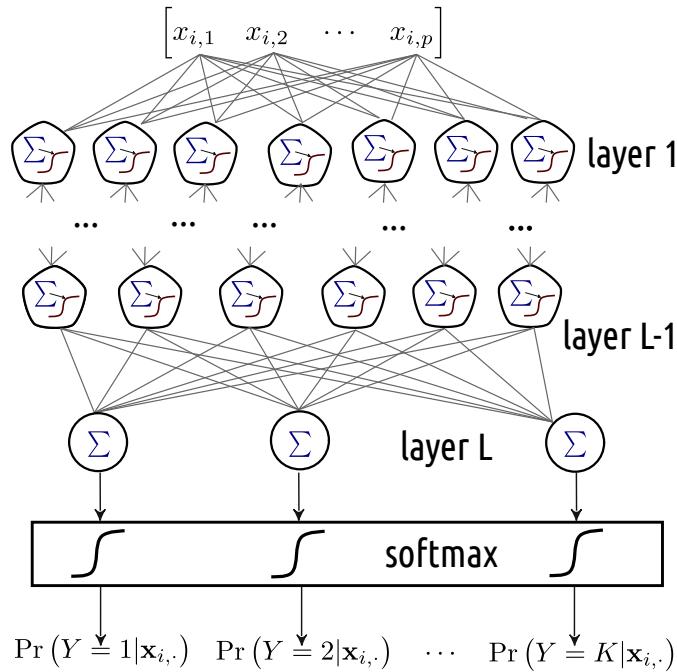


Figure 5.9: A multi-layer neural network

### 5.4.2 Activation Functions

Each layer's outputs should be transformed by some non-linear activation function. Otherwise, we'd end up with linear combinations of linear combinations, which are linear combinations themselves.

Example activation functions that can be used in hidden (inner) layers:

- **relu** – The rectified linear unit:

$$\psi(t) = \max(t, 0),$$

- **sigmoid** – The logistic sigmoid:

$$\phi(t) = \frac{1}{1 + \exp(-t)},$$

- $\tanh$  – The hyperbolic function:

$$\tanh(t) = \frac{\exp(t) - \exp(-t)}{\exp(t) + \exp(-t)}.$$

There is not much difference between them, but some might be more convenient to handle numerically than the others, depending on the implementation.

### 5.4.3 Example in R - 2 Layers

Let's construct a 2-layer Neural Network of the type 784-800-10:

```
file_name <- "datasets/mnist_keras_model2.h5"
if (!file.exists(file_name)) {
  set.seed(123)
  model2 <- keras_model_sequential()
  layer_dense(model2, units=800, activation="relu")
  layer_dense(model2, units=10, activation="softmax")
  compile(model2, optimizer="sgd",
          loss="categorical_crossentropy")
  fit(model2, X_train2, Y_train2, epochs=10)
  save_model_hdf5(model2, file_name)
} else {
  model2 <- load_model_hdf5(file_name)
}

Y_pred2 <- predict(model2, X_test2)
Y_pred <- apply(Y_pred2, 1, which.max)-1 # 1..10 -> 0..9
mean(Y_test == Y_pred) # accuracy on the test set

## [1] 0.9583
```

Performance metrics for each digit separately, see also Figure 5.10:

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9948	0.9621514	0.9857143	0.9737903	8982	14	38	966
1	0.9962	0.9815628	0.9850220	0.9832894	8844	17	21	1118
2	0.9911	0.9600000	0.9534884	0.9567331	8927	48	41	984
3	0.9898	0.9477318	0.9514851	0.9496047	8937	49	53	961
4	0.9919	0.9582909	0.9592668	0.9587786	8977	40	41	942
5	0.9911	0.9546999	0.9450673	0.9498592	9068	49	40	843
6	0.9920	0.9488753	0.9686848	0.9586777	8992	30	50	928
7	0.9906	0.9551657	0.9533074	0.9542356	8926	48	46	980
8	0.9899	0.9542144	0.9414784	0.9478036	8982	57	44	917
9	0.9892	0.9564336	0.9355798	0.9458918	8948	65	43	944

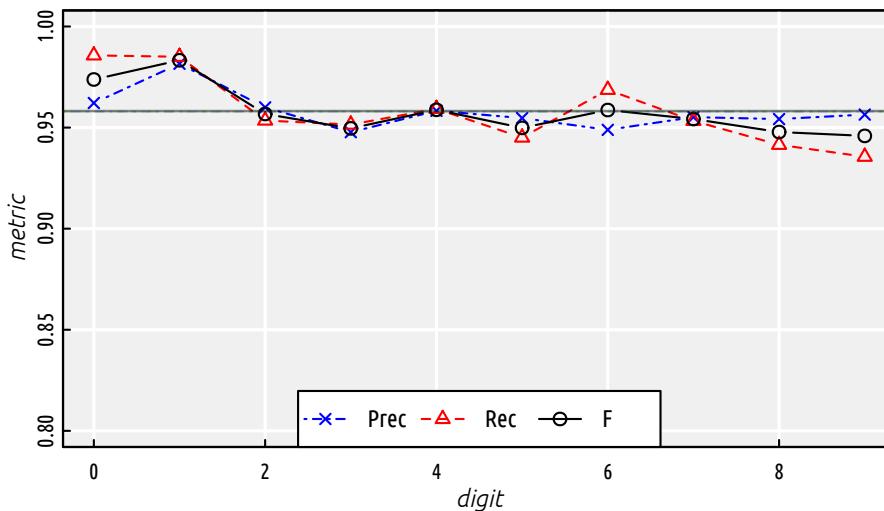


Figure 5.10: Performance metrics for a 2-layer net 784-800-10 [relu] on MNIST

#### 5.4.4 Example in R - 6 Layers

How about a 6-layer Deep Neural Network like 784-2500-2000-1500-1000-500-10?  
Here you are:

```
file_name <- "datasets/mnist_keras_model3.h5"
if (!file.exists(file_name)) {
  set.seed(123)
  model3 <- keras_model_sequential()
  layer_dense(model3, units=2500, activation="relu")
  layer_dense(model3, units=2000, activation="relu")
  layer_dense(model3, units=1500, activation="relu")
  layer_dense(model3, units=1000, activation="relu")
  layer_dense(model3, units=500, activation="relu")
  layer_dense(model3, units=10, activation="softmax")
  compile(model3, optimizer="sgd",
    loss="categorical_crossentropy")
  fit(model3, X_train2, Y_train2, epochs=10)
  save_model_hdf5(model3, file_name)
} else {
  model3 <- load_model_hdf5(file_name)
}

Y_pred2 <- predict(model3, X_test2)
Y_pred <- apply(Y_pred2, 1, which.max)-1 # 1..10 -> 0..9
mean(Y_test == Y_pred) # accuracy on the test set
```

```
## [1] 0.9797
```

Performance metrics for each digit separately, see also Figure 5.11.

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9966	0.9739479	0.9918367	0.9828109	8994	8	26	972
1	0.9975	0.9885563	0.9894273	0.9889916	8852	12	13	1123
2	0.9951	0.9861523	0.9660853	0.9760157	8954	35	14	997
3	0.9964	0.9909274	0.9732673	0.9820180	8981	27	9	983
4	0.9960	0.9700599	0.9898167	0.9798387	8988	10	30	972
5	0.9962	0.9885584	0.9686099	0.9784824	9098	28	10	864
6	0.9963	0.9801877	0.9812109	0.9806990	9023	18	19	940
7	0.9961	0.9833822	0.9785992	0.9809849	8955	22	17	1006
8	0.9939	0.9606458	0.9774127	0.9689567	8987	22	39	952
9	0.9953	0.9743590	0.9791873	0.9767672	8965	21	26	988

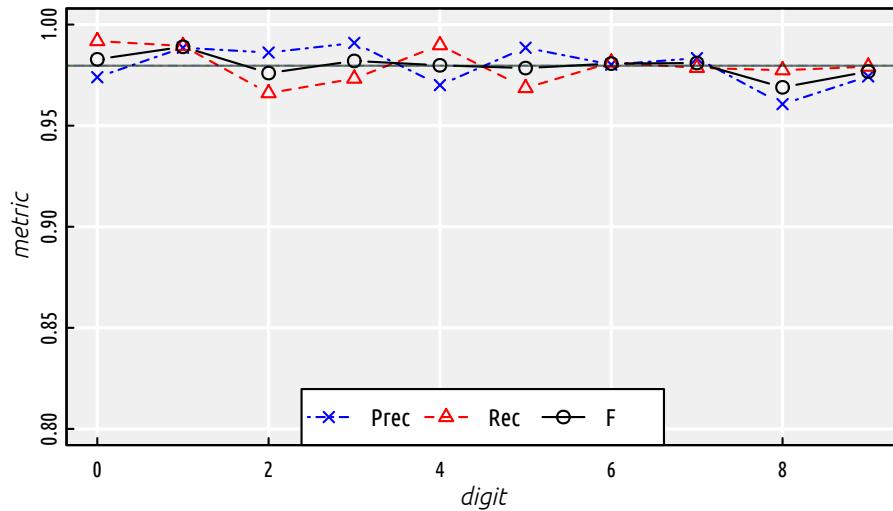


Figure 5.11: Performance metrics for a 6-layer net 784-2500-2000-1500-1000-500-10 [relu] on MNIST

### Exercise.

Test the performance of different neural network architectures (different number of layers, different number of neurons in each layer etc.). Yes, it's more art than science! Many tried to come up with various "rules of thumb", see, for example, the `comp.ai.neural-nets` FAQ (Sarle & others 2002) at <http://www.faqs.org/faqs/ai-faq/neural-nets/part3/preamble.html>, but what works well in one problem might not be generalisable to another one.

□

## 5.5 Preprocessing of Data

### 5.5.1 Introduction

Do not underestimate the power of appropriate data preprocessing — deep neural networks are not a universal replacement for a data engineer’s hard work!

On top of that, they are not interpretable – these are merely black-boxes.

Among the typical transformations of the input images we can find:

- normalisation of colours (setting brightness, stretching contrast, etc.),
- repositioning of the image (centring),
- deskewing (see below),
- denoising (e.g., by blurring).

Another frequently applied technique concerns an expansion of the training data — we can add “artificially contaminated” images to the training set (e.g., slightly rotated digits) so as to be more ready to whatever will be provided in the test test.

### 5.5.2 Image Deskewng

Deskewng of images (“straightening” of the digits) is amongst the most typical transformations that can be applied on MNIST.

Unfortunately, we don’t have (yet) the necessary mathematical background to discuss this operation in very detail.

Luckily, we can apply it on each image anyway.

See the GitHub repository at <https://github.com/gagolews/Playground.R> for an example notebook and the `deskew.R` script.

```
# See https://github.com/gagolews/Playground.R
source("~/R/Playground.R/deskew.R")
# new_image <- deskew(old_image)
```

Let’s take a look at Figure 5.12. In each pair, the left image (black background) is the original one, and the right image (palette inverted for purely dramatic effects) is its deskewed version.

Below we deskew each image in the training as well as in the test sample. This also takes a long time, so let’s store the resulting objects for further reference:

```
file_name <- "datasets/mnist_deskewed_train.rds"
if (!file.exists(file_name)) {
  Z_train <- X_train
  for (i in 1:dim(Z_train)[1]) {
```

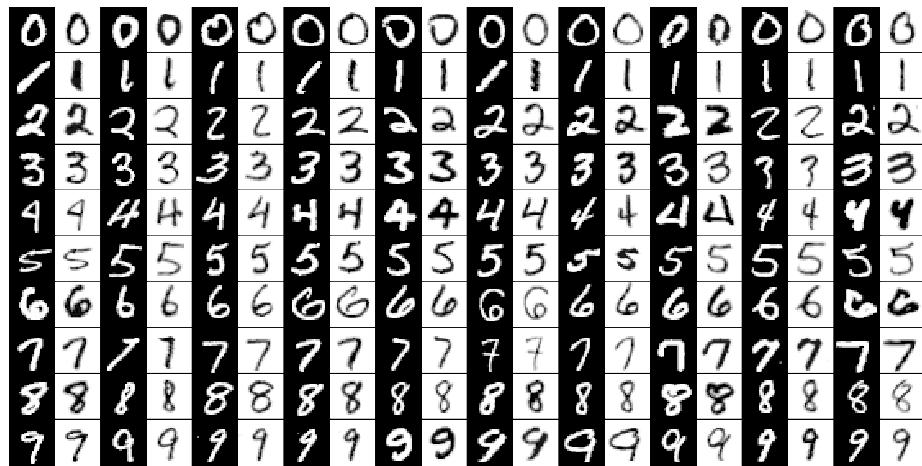


Figure 5.12: Deskewing of the MNIST digits

```

        Z_train[i,,] <- deskew(Z_train[i,,])
    }
    Z_train2 <- matrix(Z_train, ncol=28*28)
    saveRDS(Z_train2, file_name)
} else {
    Z_train2 <- readRDS(file_name)
}

file_name <- "datasets/mnist_deskewed_test.rds"
if (!file.exists(file_name)) {
    Z_test <- X_test
    for (i in 1:dim(Z_test)[1]) {
        Z_test[i,,] <- deskew(Z_test[i,,])
    }
    Z_test2 <- matrix(Z_test, ncol=28*28)
    saveRDS(Z_test2, file_name)
} else {
    Z_test2 <- readRDS(file_name)
}

```

**Remark.** RDS is a compressed file format used by R for object serialisation (quickly storing its verbatim copies so that they can be reloaded at any time).

Multinomial logistic regression model (1-layer NN):

```

file_name <- "datasets/mnist_keras_model1d.h5"
if (!file.exists(file_name)) {

```

```

set.seed(123)
model1d <- keras_model_sequential()
layer_dense(model1d, units=10, activation="softmax")
compile(model1d, optimizer="sgd",
        loss="categorical_crossentropy")
fit(model1d, Z_train2, Y_train2, epochs=10)
save_model_hdf5(model1d, file_name)
} else {
  model1d <- load_model_hdf5(file_name)
}

Y_pred2 <- predict(model1d, Z_test2)
Y_pred <- apply(Y_pred2, 1, which.max)-1 # 1..10 -> 0..9
mean(Y_test == Y_pred) # accuracy on the test set

## [1] 0.9488

```

Performance metrics for each digit separately, see also Figure 5.13.

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9939	0.9545005	0.9846939	0.9693621	8974	15	46	965
1	0.9959	0.9823633	0.9814978	0.9819304	8845	21	20	1114
2	0.9878	0.9540918	0.9263566	0.9400197	8922	76	46	956
3	0.9904	0.9506903	0.9544554	0.9525692	8940	46	50	964
4	0.9888	0.9411765	0.9450102	0.9430894	8960	54	58	928
5	0.9905	0.9442586	0.9495516	0.9468977	9058	45	50	847
6	0.9905	0.9556494	0.9446764	0.9501312	9000	53	42	905
7	0.9892	0.9600000	0.9338521	0.9467456	8932	68	40	960
8	0.9855	0.9116187	0.9425051	0.9268046	8937	56	89	918
9	0.9851	0.9291417	0.9226957	0.9259075	8920	78	71	931

### 5.5.3 Summary of All the Models Considered

Let's summarise the quality of all the considered classifiers. Figure 5.14 gives the F-measures, for each digit separately.

Note that the applied preprocessing of data increased the prediction accuracy.

The same information can also be included on a heat map which is depicted in Figure 5.15 (see the `image()` function in R).

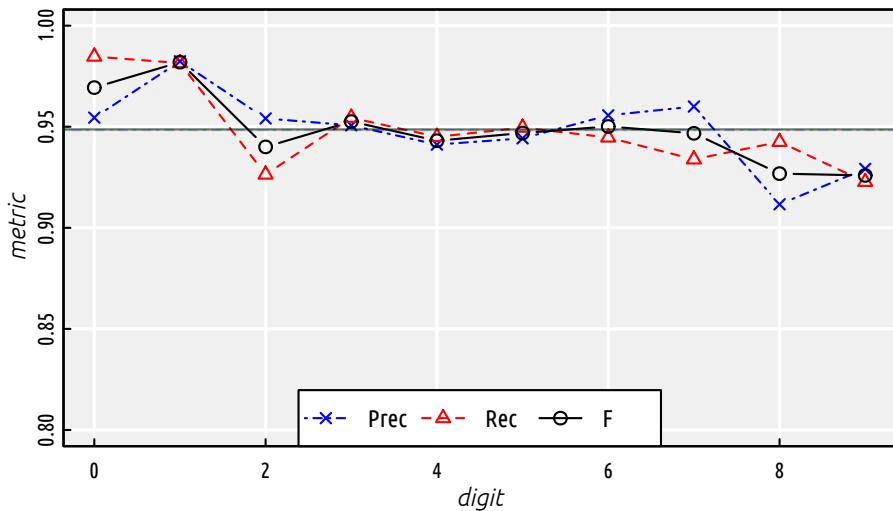


Figure 5.13: Performance of Multinomial Logistic Regression on the deskewed MNIST

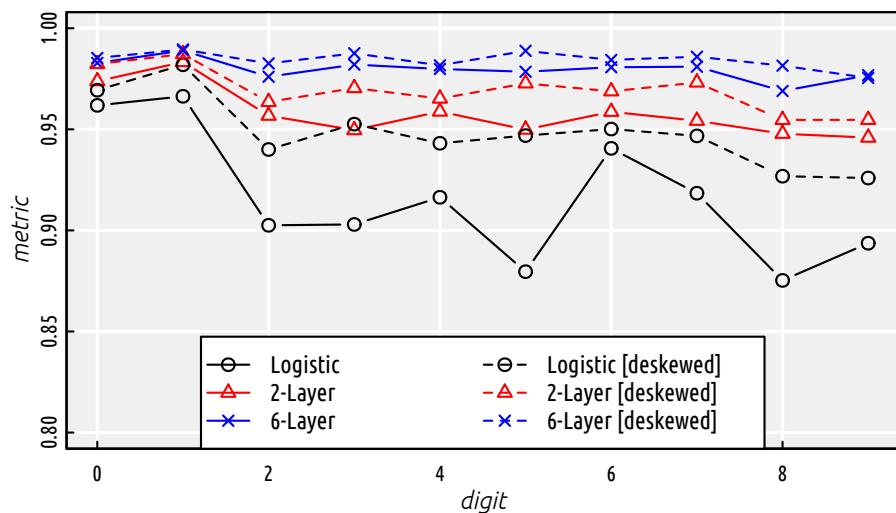


Figure 5.14: Summary of F-measures for each classified digit and every method

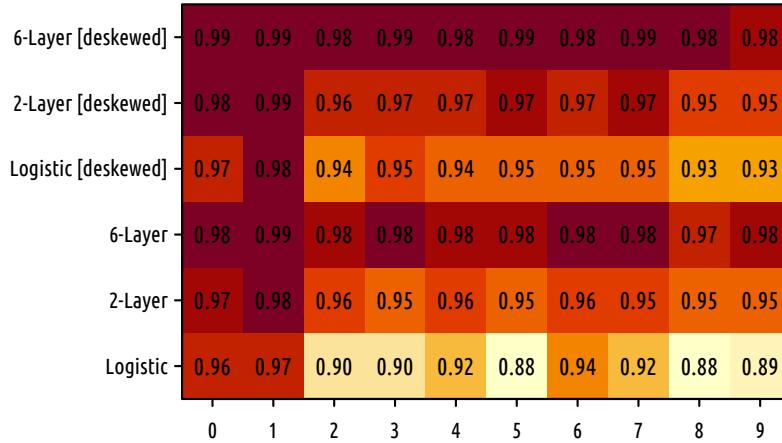


Figure 5.15: A heat map of F-measures for each classified digit and each method

## 5.6 Outro

### 5.6.1 Remarks

We have discussed a multinomial logistic regression model as a generalisation of the binary one.

This in turn is a special case of feed-forward neural networks.

There's a lot of hype (again...) for deep neural networks in many applications, including vision, self-driving cars, natural language processing, speech recognition etc.

Many different architectures of neural networks and types of units are being considered in theory and in practice, e.g.:

- convolutional neural networks apply a series of signal (e.g., image) transformations in first layers, they might actually “discover” deskewing automatically etc.;
- recurrent neural networks can imitate long/short-term memory that can be used for speech synthesis and time series prediction.

Main drawbacks of deep neural networks:

- learning is very slow, especially with very deep architectures (days, weeks);
- models are not explainable (black boxes) and hard to debug;
- finding good architectures is more art than science (maybe: more of a craftsmanship even);

- sometimes using deep neural network is just an excuse for being too lazy to do proper data cleansing and pre-processing.

There are many issues and challenges that are tackled in more advanced AI/ML courses and books, such as (Goodfellow et al. 2016).

### 5.6.2 Beyond MNIST

The MNIST dataset is a classic, although its use in deep learning research is nowadays discouraged – the dataset is not considered challenging anymore – state of the art classifiers can reach 99.8% accuracy.

See Zalando's Fashion-MNIST (by Kashif Rasul & Han Xiao) at <https://github.com/zalandoresearch/fashion-mnist> for a modern replacement.

Alternatively, take a look at CIFAR-10 and CIFAR-100 (<https://www.cs.toronto.edu/~kriz/cifar.html>) by A. Krizhevsky et al. or at ImageNet (<http://image-net.org/index>) for an even greater challenge.

### 5.6.3 Further Reading

Recommended further reading: (James et al. 2017: Chapter 11), (Sarle & others 2002) and (Goodfellow et al. 2016)

See also the `keras` package tutorials available at: <https://cran.r-project.org/web/packages/keras/index.html> and <https://keras.rstudio.com>.

# Chapter 6

## Continuous Optimisation with Iterative Algorithms

### 6.1 Introduction

#### 6.1.1 Optimisation Problems

**Mathematical optimisation** (a.k.a. mathematical programming) deals with the study of algorithms to solve problems related to selecting the *best* element amongst the set of available alternatives.

Most frequently “best” is expressed in terms of an *error* or *goodness of fit* measure:

$$f : D \rightarrow \mathbb{R}$$

called an **objective function**.

$D$  is the **search space** (problem domain, feasible set) – it defines the set of possible candidate solutions.

An **optimisation task** deals with finding an element  $x \in D$  that minimises or maximises  $f$ :

$$\min_{x \in D} f(x) \quad \text{or} \quad \max_{x \in D} f(x),$$

In this chapter, we will deal with **unconstrained continuous optimisation**, i.e., we will assume the search space is  $D = \mathbb{R}^p$  for some  $p$ .

### 6.1.2 Example Optimisation Problems in Machine Learning

In **multiple linear regression** we were minimising the sum of squared residuals

$$\min_{\beta_0, \beta_1, \dots, \beta_p \in \mathbb{R}} \sum_{i=1}^n (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p} - y_i)^2.$$

In **binary logistic regression** we were minimising the cross-entropy:

$$\min_{(\beta_0, \beta_1, \dots, \beta_p) \in \mathbb{R}^{(p+1)}} -\frac{1}{n} \sum_{i=1}^n \left( y_i \log \left( \frac{1}{1+e^{-(\beta_0+\beta_1 x_{i,1}+\dots+\beta_p x_{i,p})}} \right) + (1-y_i) \log \left( \frac{e^{-(\beta_0+\beta_1 x_{i,1}+\dots+\beta_p x_{i,p})}}{1+e^{-(\beta_0+\beta_1 x_{i,1}+\dots+\beta_p x_{i,p})}} \right) \right).$$

### 6.1.3 Types of Minima and Maxima

Note that minimising  $f$  is the same as maximising  $\bar{f} = -f$ .

In other words,  $\min_{x \in D} f(x)$  and  $\max_{x \in D} -f(x)$  represent the same optimisation problems (and hence have identical solutions).

A **minimum** of  $f$  is a point  $x^*$  such that  $f(x^*) \leq f(x)$  for all  $x \in D$ .

A **maximum** of  $f$  is a point  $x^*$  such that  $f(x^*) \geq f(x)$  for all  $x \in D$ .

Assuming that  $D = \mathbb{R}$ , Figure 6.1 shows an example objective function,  $f : \mathbb{D} \rightarrow \mathbb{R}$ , that has a minimum at  $x^* = 1$  with  $f(x^*) = -2$ .

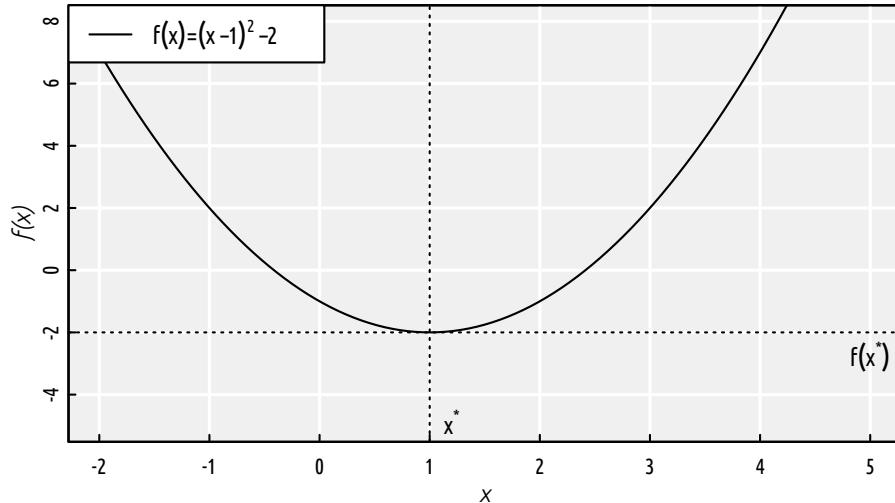


Figure 6.1: A function with the global minimum at  $x^* = 1$

$$\min_{x \in \mathbb{R}} f(x) = -2 \text{ (value of } f \text{ at the minimum)}$$

$$\arg \min_{x \in \mathbb{R}} f(x) = 1 \text{ (location of the minimum)}$$

By definition, a minimum/maximum **might not necessarily be unique**. This depends on a problem.

Assuming that  $D = \mathbb{R}$ , Figure 6.2 gives an example objective function,  $f : \mathbb{D} \rightarrow \mathbb{R}$ , that has multiple minima; every  $x^* \in [1 - \sqrt{2}, 1 + \sqrt{2}]$  yields  $f(x^*) = 0$ .

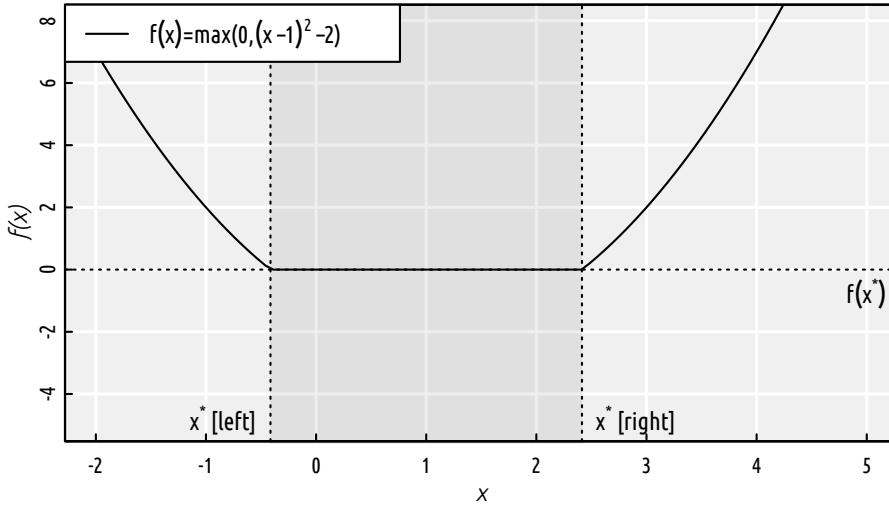


Figure 6.2: A function that has multiple minima

**Remark.** If this was the case of some machine learning problem, it'd mean that we could have many equally well-performing models, and hence many equivalent explanations of the same phenomenon.

Moreover, it may happen that a function has **multiple local minima**, compare Figure 6.3.

We say that  $f$  has a **local minimum** at  $\mathbf{x}^+ \in D$ , if for some neighbourhood  $B(\mathbf{x}^+)$  of  $\mathbf{x}^+$  it holds  $f(\mathbf{x}^+) \leq f(\mathbf{x})$  for each  $\mathbf{x} \in B(\mathbf{x}^+)$ .

**Definition.** If  $D = \mathbb{R}$ , by neighbourhood  $B(x)$  of  $x$  we mean an open interval centred at  $x$  of width  $2r$  for some small  $r > 0$ , i.e.,  $(x - r, x + r)$

**Definition.** (\*) If  $D = \mathbb{R}^p$  (for any  $p \geq 1$ ), by neighbourhood  $B(\mathbf{x})$  of  $\mathbf{x}$  we mean an *open ball* centred at  $\mathbf{x}^+$  of some small radius  $r > 0$ , i.e.,  $\{\mathbf{y} : \|\mathbf{x} - \mathbf{y}\| < r\}$  (read: the set of all the points with Euclidean distances to  $\mathbf{x}$  less than  $r$ ).

To avoid ambiguity, the “true” minimum (a point  $x^*$  such that  $f(x^*) \leq f(x)$  for all  $x \in D$ ) is sometimes also referred to as a **global minimum**.

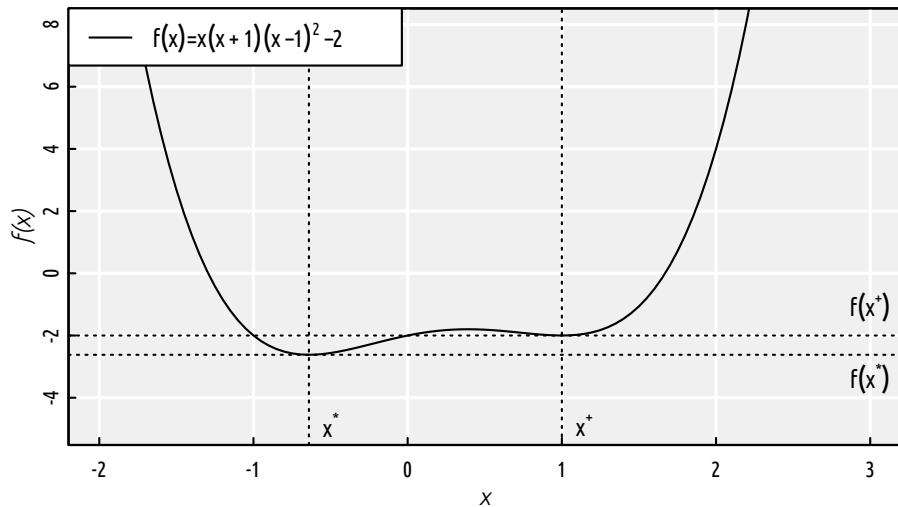


Figure 6.3: A function with two local minima

**Remark.** Of course, the global minimum is also a function's local minimum.

The existence of local minima is problematic as most of the optimisation methods might get stuck there and fail to return the global one.

Moreover, we cannot often be sure if the result returned by an algorithm is indeed a global minimum. Maybe there exists a better solution that hasn't been considered yet? Or maybe the function is very noisy (see Figure 6.4)?

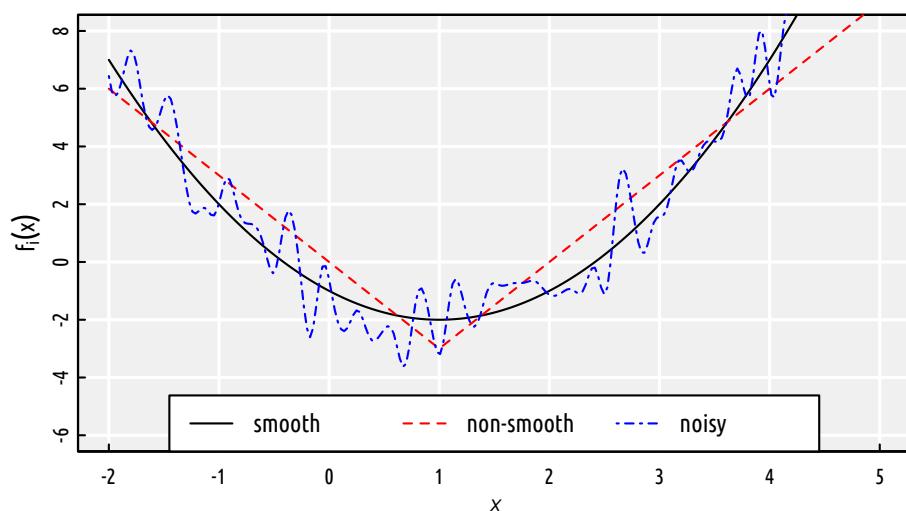


Figure 6.4: Smooth vs. non-smooth vs. noisy objective functions

### 6.1.4 Example Objective over a 2D Domain

Of course, our objective function does not necessarily have to be defined over a one-dimensional domain.

For example, consider the following function:

$$g(x_1, x_2) = \log((x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\dots)$$

```
g <- function(x1, x2)
  log((x1^2+x2-5)^2+(x1+x2^2-3)^2+x1^2-1.60644366086443841)
x1 <- seq(-5, 5, length.out=100)
x2 <- seq(-5, 5, length.out=100)
# outer() expands two vectors to form a 2D grid
# and applies a given function on each point
y <- outer(x1, x2, g)
```

There are four local minima:

x1	x2	f(x1,x2)
2.278005	-0.6134279	1.3564152
-2.612316	-2.3454621	1.7050788
1.798788	1.1987929	0.6954984
-1.542256	2.1564053	0.0000000

The global minimum is at  $(x_1^*, x_2^*)$  as below:

```
g(-1.542255693195422641930153, 2.156405289793087261832605)
```

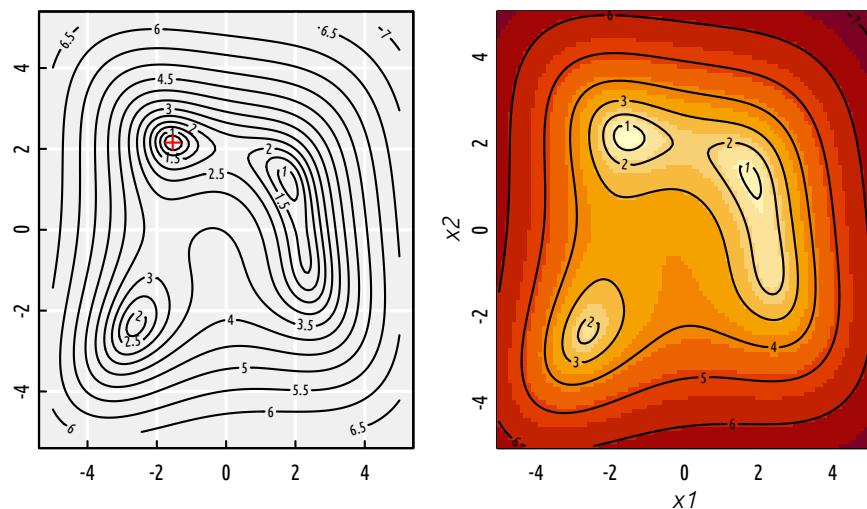
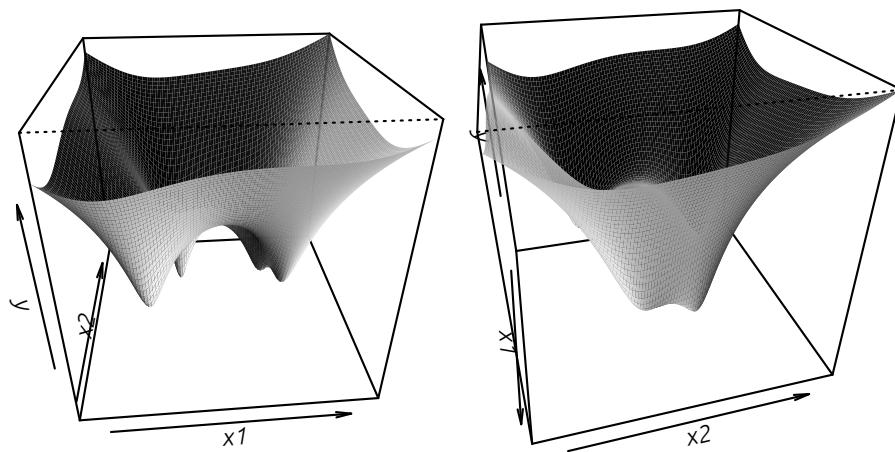
```
## [1] 0
```

Let's explore various ways of depicting  $f$ . A contour plot and a heat map are given in Figure 6.5.

```
par(mfrow=c(1,2)) # 2 in 1
# lefthand plot:
contour(x1, x2, y, nlevels=25)
points(-1.54226, 2.15641, col=2, pch=3)
# righthand plot:
image(x1, x2, y)
contour(x1, x2, y, add=TRUE)
```

Two perspective plots (views from different angles) are given in Figure 6.6.

```
par(mfrow=c(1,2)) # 2 in 1
persp(x1, x2, y, phi=30, theta=-5, shade=2, border=NA)
persp(x1, x2, y, phi=30, theta=75, shade=2, border=NA)
```

Figure 6.5: A contour plot and a heat map of  $g(x_1, x_2)$ Figure 6.6: Perspective plots of  $g(x_1, x_2)$

**Remark.** As usual, depicting functions that are defined over high-dimensional (3D and higher) domains is... difficult. Usually 1D or 2D projections can give us some neat intuitions though.

## 6.2 Iterative Methods

### 6.2.1 Introduction

Many optimisation algorithms are built around the following scheme:

*starting from a random point, perform a walk, in each step deciding where to go based on the idea of where the location of the minimum seems to be.*

**Example.** Imagine we're to cycle from Deakin University's Burwood Campus to the CBD not knowing the route and with GPS disabled – we'll have to ask many people along the way, but we'll eventually (because most people are good) get to some CBD (say, in Perth).

More formally, we are interested in iterative algorithms that operate in a greedy-like fashion:

1.  $\mathbf{x}^{(0)}$  – initial guess (e.g., generated at random)
2. for  $i = 1, \dots, M$ :
  - a.  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + [\text{guessed direction}]$
  - b. if  $|f(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i-1)})| < \varepsilon$  break
3. return  $\mathbf{x}^{(i)}$  as result

Note that there are two stopping criteria, based on:

- $M$  = maximum number of iterations
- $\varepsilon$  = tolerance, e.g.,  $10^{-8}$

### 6.2.2 Example in R

R has a built-in function, `optim()`, that provides an implementation of (amongst others) **the BFGS method** (proposed by Broyden, Fletcher, Goldfarb and Shanno in 1970).

**Remark.** (\*) BFGS uses the assumption that the objective function is smooth – the [guessed direction] is determined by computing the (partial) derivatives (or their finite-difference approximations). However, they might work well even if this is not the case. We'll be able to derive similar algorithms (called quasi-Newton ones) ourselves once we learn about Taylor series approximation by reading a book/taking a course on calculus.

Here, we shall use the BFGS as a *black-box* continuous optimisation method, i.e., without going into how it has been defined (in terms of our assumed math skills,

it might be too early for this). Despite that, will still be able to point out a few interesting patterns.

```
optim(par, fn, method="BFGS")
```

where:

- **par** – an initial guess (a numeric vector of length  $p$ )
- **fn** – an objective function to minimise (takes a vector of length  $p$  on input, returns a single number)

Let us minimise the  $g$  function defined above (the one with the 2D domain):

```
# g needs to be rewritten to accept a 2-ary vector
g_vectorised <- function(x12) g(x12[1], x12[2])
# random starting point with coordinates in [-5, 5]
(x12_init <- runif(2, -5, 5))

## [1] -2.124225  2.883051
(res <- optim(x12_init, g_vectorised, method="BFGS"))

## $par
## [1] -1.542255  2.156405
##
## $value
## [1] 1.413092e-12
##
## $counts
## function gradient
##      101      21
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Note that:

- **par** gives the location of the local minimum found,
- **value** gives the value of  $g$  at **par**,
- **convergence** of 0 is a successful one (we were able to satisfy the  $|f(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i-1)})| < \varepsilon$  condition).

We can even depict the points that the algorithm is “visiting”, see Figure 6.7.

**Remark.** (\*) Technically, the algorithm needs to evaluate a few more points in order to make the decision on where to go next (BFGS approximates the Hessian matrix).

```

g_vectorised_plot <- function(x12) {
  points(x12[1], x12[2], col=2, pch=3) # draw
  g(x12[1], x12[2]) # return value
}
contour(x1, x2, y, nlevels=25)
res <- optim(x12_init, g_vectorised_plot, method="BFGS")

```

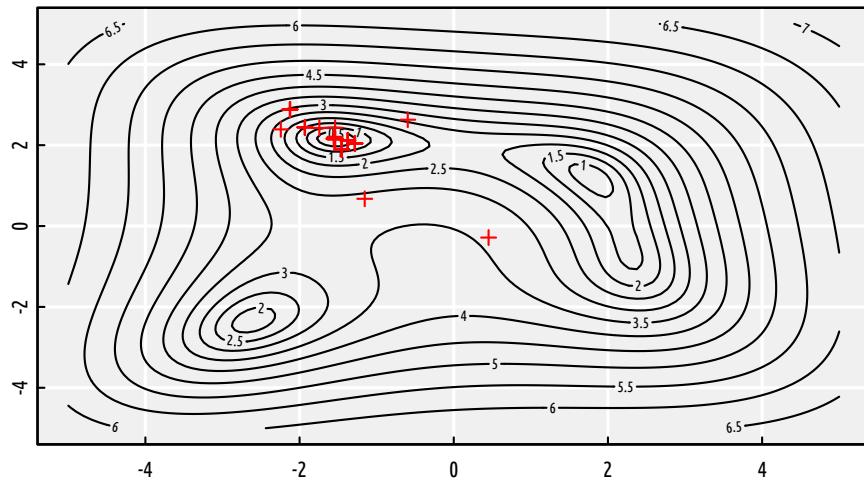


Figure 6.7: Each plotting symbol marks a point where the objective function was evaluated by the BFGS method

### 6.2.3 Convergence to Local Optima

We were lucky, because the local minimum that the algorithm has found coincides with the global minimum.

Let's see where does the BFGS algorithm converge if seek the minimum of the above  $g$  starting from many randomly chosen points uniformly distributed over the square  $[-5, 5] \times [-5, 5]$ :

```

res_value <- replicate(1000, {
  # this will be iterated 100 times
  x12_init <- runif(2, -5, 5)
  res <- optim(x12_init, g_vectorised, method="BFGS")
  res$value # return value from each iteration
})
table(round(res_value,3))

##

```

```
##      0 0.695 1.356 1.705
##    273   352   156   219
```

Unfortunately, we find the global minimum only in  $\sim 25\%$  cases, compare Figure 6.8.

```
hist(res_value, col="white", breaks=100, main=NA); box()
```

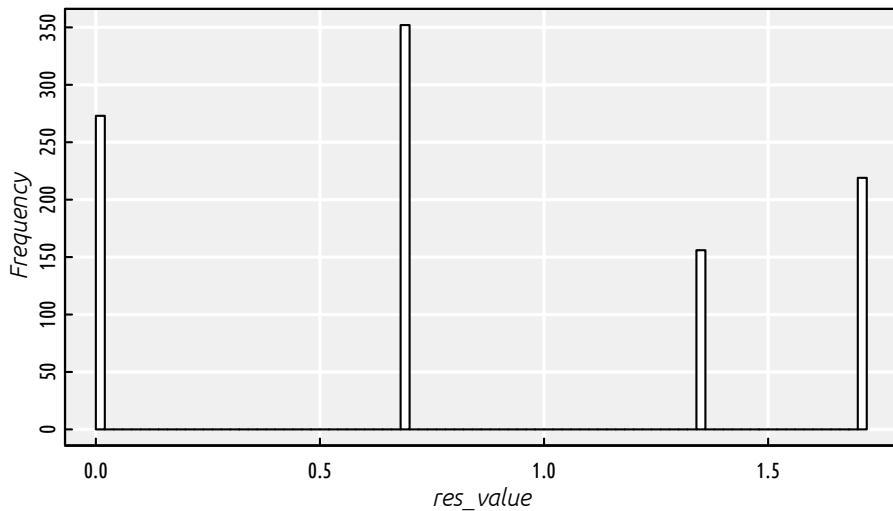


Figure 6.8: A histogram of the objective function’s value at the local minimum found when using a random initial guess

Figure 6.9 depicts all the random starting points and where do we converge from them.

#### 6.2.4 Random Restarts

A kind of “remedy” for the above limitation could be provided by *repeated local search*: in order to robustify an optimisation procedure it is often advised to consider multiple random initial points and pick the best solution amongst the identified local optima.

```
# N           - number of restarts
# par_generator - a function generating initial guesses
# ...          - further arguments to optim()
optim_with_restarts <- function(par_generator, ..., N=10) {
  res_best <- list(value=Inf) # cannot be worse than this
  for (i in 1:N) {
    res <- optim(par_generator(), ...)
    if (res$value < res_best$value)
```

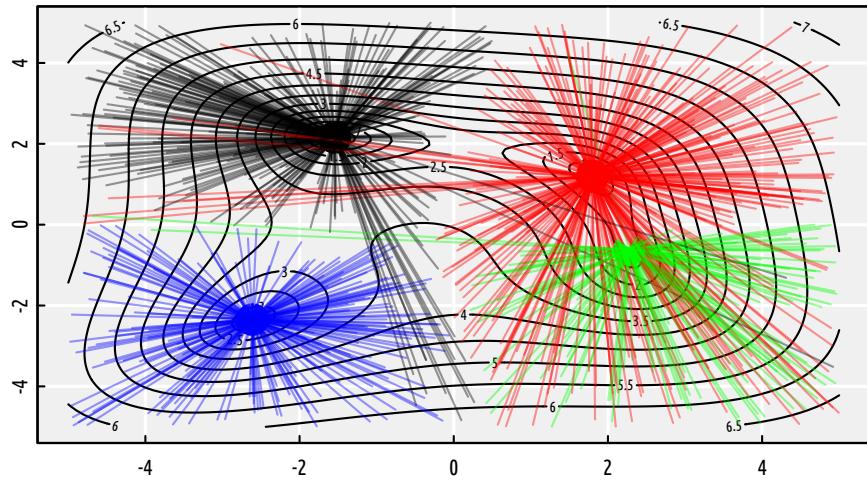


Figure 6.9: Each line segment connect a starting point to the point of BFGS's convergence

```

        res_best <- res # a better candidate found
    }
    res_best
}

optim_with_restarts(function() runif(2, -5, 5),
  g_vectorised, method="BFGS", N=10)

```

```

## $par
## [1] -1.542256  2.156405
##
## $value
## [1] 3.970158e-13
##
## $counts
## function gradient
##       48         17
##
## $convergence
## [1] 0
##
## $message
## NULL

```

**Exercise.**

*Food for thought: Can we guarantee that the global minimum will be found within  $N$  tries?*

□

**Solution.**

Absolutely not.

■

## 6.3 Gradient Descent

### 6.3.1 Function Gradient (\*)

How to choose the [guessed direction] in our iterative optimisation algorithm?

If we are minimising a smooth function, the simplest possible choice is to use the information included in the objective's **gradient**, which provides us with the information about the direction where the function decreases the fastest.

**Definition.** (\*) Gradient of  $f : \mathbb{R}^p \rightarrow \mathbb{R}$ , denoted  $\nabla f : \mathbb{R}^p \rightarrow \mathbb{R}^p$ , is the vector of all its partial derivatives, ( $\nabla$  – nabla symbol = differential operator)

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_p}(\mathbf{x}) \end{bmatrix}$$

If we have a function  $f(x_1, \dots, x_p)$ , the partial derivative w.r.t. the  $i$ -th variable, denoted  $\frac{\partial f}{\partial x_i}$  is like an ordinary derivative w.r.t.  $x_i$  where  $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_p$  are assumed constant.

**Remark.** Function differentiation is an important concept – see how it's referred to in, e.g., the **keras** package manual at <https://keras.rstudio.com/reference/fit.html>.

Recall our  $g$  function defined above:

$$g(x_1, x_2) = \log((x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\dots)$$

It can be shown (\*) that:

$$\begin{aligned} \frac{\partial g}{\partial x_1}(x_1, x_2) &= \frac{4x_1(x_1^2 + x_2 - 5) + 2(x_1 + x_2^2 - 3) + 2x_1}{(x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\dots} \\ \frac{\partial g}{\partial x_2}(x_1, x_2) &= \frac{2(x_1^2 + x_2 - 5) + 4x_2(x_1 + x_2^2 - 3)}{(x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\dots} \end{aligned}$$

```
grad_g_vectorised <- function(x) {
  c(
    4*x[1]*(x[1]^2+x[2]-5)+2*(x[1]+x[2]^2-3)+2*x[1],
    2*(x[1]^2+x[2]-5)+4*x[2]*(x[1]+x[2]^2-3)
  )/(
    (x[1]^2+x[2]-5)^2+(x[1]+x[2]^2-3)^2+x[1]^2-1.60644366086443841
  )
}
```

### 6.3.2 Three Facts on the Gradient

For now, we should emphasise three important facts:

**Fact 1.** If we are incapable of deriving the gradient analytically, we can rely on its finite differences approximation. Each partial derivative can be estimated by means of:

$$\frac{\partial f}{\partial x_i}(x_1, \dots, x_p) \simeq \frac{f(x_1, \dots, x_i + \delta, \dots, x_p) - f(x_1, \dots, x_i, \dots, x_p)}{\delta}$$

for some small  $\delta > 0$ , say,  $\delta = 10^{-6}$ .

**Remark.** (\*) Actually, a function's partial derivative, by definition, is the limit of the above as  $\delta \rightarrow 0$ .

Example implementation:

```
# gradient of f at x=c(x[1],...,x[p])
grad_approx <- function(f, x, delta=1e-6) {
  p <- length(x)
  gf <- numeric(p) # vector of length p
  for (i in 1:p) {
    xi <- x
    xi[i] <- xi[i]+delta
    gf[i] <- f(xi)
  }
  (gf-f(x))/delta
}
```

**Remark.** (\*) Interestingly, some modern vector/matrix algebra frameworks like TensorFlow (upon which keras is built) or PyTorch, feature methods to “derive” the gradient algorithmically (autodiff; automatic differentiation).

Sanity check:

```
grad_approx(g_vectorised, c(-2, 2))
```

```
## [1] -3.186485 -1.365634
```

```
grad_g_vectorised(c(-2, 2))

## [1] -3.186485 -1.365636
grad_approx(g_vectorised, c(-1.542255693, 2.15640528979))

## [1] 1.058842e-05 1.981748e-05
grad_g_vectorised(c(-1.542255693, 2.15640528979))

## [1] 4.129167e-09 3.577146e-10
```

BTW, there is also the `grad()` function in package `numDeriv` that might be a little more accurate (uses a different approximation formula).

**Fact 2a.** The gradient of  $f$  at  $\mathbf{x}$ ,  $\nabla f(\mathbf{x})$ , is a vector that points in the direction of the steepest slope.

**Fact 2b.** Minus gradient,  $-\nabla f(\mathbf{x})$ , is the direction where the function decreases the fastest.

**Remark.** (\*) This can be shown by considering a function's first-order Taylor series approximation.

Therefore, in our iterative algorithm, we may try taking the direction of the minus gradient!

How far in that direction? Well, a bit. We will refer to the desired step size as the **learning rate**,  $\eta$ .

This will be called the **gradient descent** method (GD; Cauchy, 1847).

**Fact 3.** If a function  $f$  has a local minimum at  $\mathbf{x}^*$ , then its gradient vanishes there, i.e.,  $\nabla f(\mathbf{x}^*) = [0, \dots, 0]$ .

Note that the above condition is a necessary, not sufficient one. For example, the gradient also vanishes at a maximum or at a saddle point. In fact, we have what follows.

**Theorem.** (\*\*\*) More generally, a twice-differentiable function has a local minimum at  $\mathbf{x}^*$  if and only if its gradient vanishes there and  $\nabla^2 f(\mathbf{x}^*)$  (Hessian matrix = matrix of all second-order derivatives) is positive-definite.

### 6.3.3 Gradient Descent Algorithm (GD)

Taking the above into account, we arrive at the gradient descent algorithm:

1.  $\mathbf{x}^{(0)}$  – initial guess (e.g., generated at random)
2. for  $i = 1, \dots, M$ :
  - a.  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} - \eta \nabla f(\mathbf{x}^{(i-1)})$

- b. if  $|f(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i-1)})| < \varepsilon$  break  
 3. return  $\mathbf{x}^{(i)}$  as result

where  $\eta > 0$  is a step size frequently referred to as the *learning rate*, because that's much more cool. We usually set  $\eta$  of small order of magnitude, say 0.01 or 0.1.

An implementation of the gradient descent algorithm is straightforward. In essence, it's the `par <- par - eta*grad_g_vectorised(par)` expression run in a loop, until convergence.

```
# par      - initial guess
# fn       - a function to be minimised
# gr       - a function to return the gradient of fn
# eta      - learning rate
# maxit   - maximum number of iterations
# tol      - convergence tolerance
optim_gd <- function(par, fn, gr, eta=0.01,
                      maxit=1000, tol=1e-8) {
  f_last <- fn(par)
  for (i in 1:maxit) {
    par <- par - eta*grad_g_vectorised(par) # update step
    f_cur <- fn(par)
    if (abs(f_cur-f_last) < tol) break
    f_last <- f_cur
  }
  list( # see ?optim, section `Value`
    par=par,
    value=g_vectorised(par),
    counts=i,
    convergence=as.integer(i==maxit)
  )
}
```

Tests of the  $g$  function. First, let's try  $\eta = 0.01$ . Figure 6.10 zooms in the contour plot so that we can see the actual path the algorithm has taken.

```
eta <- 0.01
res <- optim_gd(c(-3, 1), g_vectorised, grad_g_vectorised, eta=eta)
str(res)
```

```
## List of 4
## $ par      : num [1:2] -1.54 2.16
## $ value    : num 1.33e-08
## $ counts   : int 135
## $ convergence: int 0
```

Now let's try  $\eta = 0.05$ .

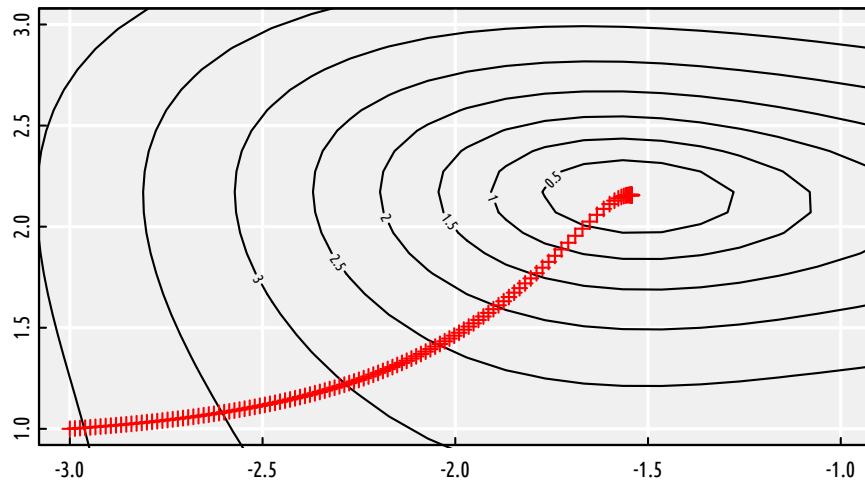


Figure 6.10: Path taken by the gradient descent algorithm with  $\eta = 0.01$

```

eta <- 0.05
res <- optim_gd(c(-3, 1), g_vectorised, grad_g_vectorised, eta=eta)
str(res)

## List of 4
## $ par      : num [1:2] -1.54 2.15
## $ value    : num 0.000203
## $ counts   : int 417
## $ convergence: int 0

```

With an increased step size, the algorithm needed many more iterations (3 times as many), see Figure 6.11 for the path taken.

And now for something completely different:  $\eta = 0.1$ , see Figure 6.12.

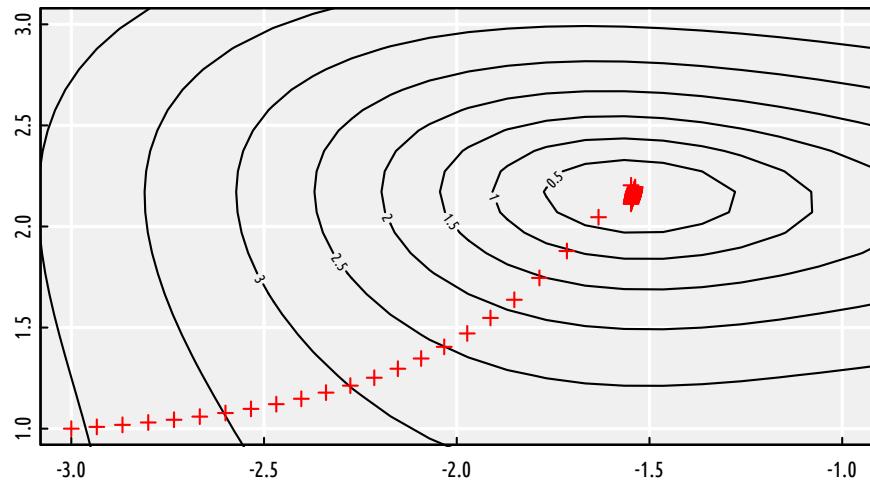
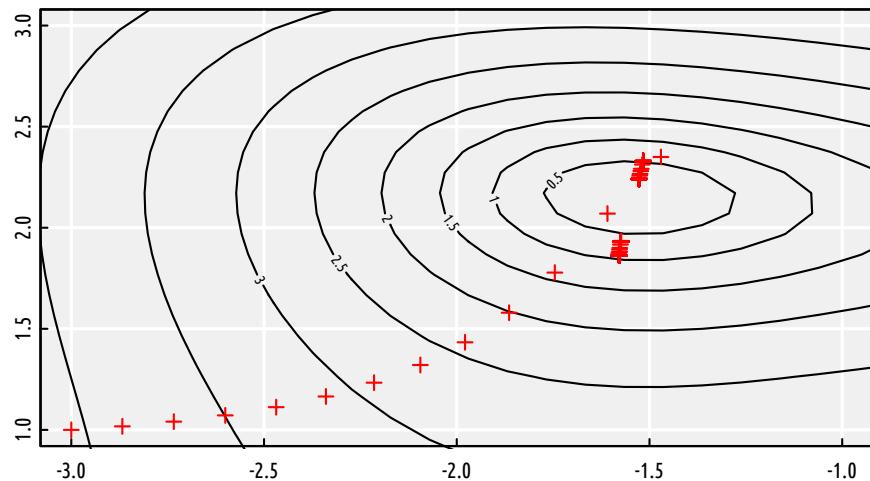
```

eta <- 0.1
res <- optim_gd(c(-3, 1), g_vectorised, grad_g_vectorised, eta=eta)
str(res)

## List of 4
## $ par      : num [1:2] -1.52 2.33
## $ value    : num 0.507
## $ counts   : int 1000
## $ convergence: int 1

```

The algorithm failed to converge.

Figure 6.11: Path taken by the gradient descent algorithm with  $\eta = 0.05$ Figure 6.12: Path taken by the gradient descent algorithm with  $\eta = 0.1$

If the learning rate  $\eta$  is too small, the convergence might be too slow or we might get stuck at a plateau. On the other hand, if  $\eta$  is too large, we might be overshooting and end up bouncing around the minimum.

This is why many optimisation libraries (including `keras`/TensorFlow) implement some of the following ideas:

- *learning rate decay* – start with large  $\eta$ , decreasing it in every iteration, say, by some percent;
- *line search* – determine optimal  $\eta$  in every step by solving a 1-dimensional optimisation problem w.r.t.  $\eta \in [0, \eta_{\max}]$ ;
- *momentum* – the update step is based on a combination of the gradient direction and the previous change of the parameters,  $\Delta\mathbf{x}$ ; can be used to accelerate search in the relevant direction and minimise oscillations.

#### Exercise.

*Try implementing at least the first of the above heuristics yourself. You can set `eta <- eta*0.9` in every iteration of the gradient descent procedure.*

□

#### 6.3.4 Example: MNIST (\*)

In the previous chapter we've studied the MNIST dataset. Let us go back to the task of fitting a multiclass logistic regression model.

```
library("keras")
mnist <- dataset_mnist()

# get train/test images in greyscale
X_train <- mnist$train$x/255 # to [0,1]
X_test  <- mnist$test$x/255  # to [0,1]

# get the corresponding labels in {0,1,...,9}:
Y_train <- mnist$train$y
Y_test  <- mnist$test$y
```

The labels need to be one-hot encoded:

```
one_hot_encode <- function(Y) {
  stopifnot(is.numeric(Y))
  c1 <- min(Y) # first class label
  cK <- max(Y) # last class label
  K <- cK-c1+1 # number of classes
  Y2 <- matrix(0, nrow=length(Y), ncol=K)
  Y2[cbind(1:length(Y), Y-c1+1)] <- 1
```

```

    Y2
}

Y_train2 <- one_hot_encode(Y_train)
Y_test2 <- one_hot_encode(Y_test)

```

Our task is to find the parameters  $\mathbf{B}$  that minimise cross entropy  $E(\mathbf{B})$  over the training set:

$$\min_{\mathbf{B} \in \mathbb{R}^{785 \times 10}} -\frac{1}{n^{\text{train}}} \sum_{i=1}^{n^{\text{train}}} \log \Pr(Y = y_i^{\text{train}} | \mathbf{x}_i^{\text{train}}, \mathbf{B}).$$

In the previous chapter, we've relied on the methods implemented in the `keras` package. Let's do that all by ourselves now.

In order to come up with a working version of the gradient descent procedure for classifying of MNIST digits, we will need to derive and implement `grad_cross_entropy()`. We do that below using matrix notation.

**Remark.** In the first reading, you can jump to the *Safe landing zone* below with no much loss in what we're trying to convey here (you will then treat `grad_cross_entropy()` as a black-box function). Nevertheless, keep in mind that this is the kind of maths you will need to master anyway sooner than later – this is inevitable. Perhaps you should go back to, e.g., the appendix on Matrix Computations with R or the chapter on Linear Regression? Learning maths is not a linear, step-by-step process. Everyone is different and will have a different path to success. The material needs to be frequently revisited, it will “click” someday, don’t you worry; good stuff isn’t built in a day or seven.

Recall that the output of the logistic regression model (1-layer neural network with softmax) can be written in the matrix form as:

$$\hat{\mathbf{Y}} = \text{softmax}(\dot{\mathbf{X}} \mathbf{B}),$$

where  $\dot{\mathbf{X}} \in \mathbb{R}^{n \times 785}$  is a matrix representing  $n$  images of size  $28 \times 28$ , augmented with a column of 1s, and  $\mathbf{B} \in \mathbb{R}^{785 \times 10}$  is the coefficients matrix and softmax is applied on each matrix row separately.

Of course, by the definition of matrix multiplication,  $\hat{\mathbf{Y}}$  will be a matrix of size  $n \times 10$ , where  $\hat{y}_{i,k}$  represents the predicted probability that the  $i$ -th image depicts the  $k$ -th digit.

```

# convert to matrices of size n*784
# and add a column of 1s
X_train1 <- cbind(1.0, matrix(X_train, ncol=28*28))
X_test1  <- cbind(1.0, matrix(X_test, ncol=28*28))

```

The `nn_predict()` function implements the above formula for  $\hat{\mathbf{Y}}$ :

```
softmax <- function(T) {
  T <- exp(T)
  T/rowSums(T)
}

nn_predict <- function(B, X) {
  softmax(X %*% B)
}
```

Let's define the functions to compute the cross-entropy (which we shall minimise) and accuracy (which we shall report to a user):

```
cross_entropy <- function(Y_true, Y_pred) {
  -sum(Y_true*log(Y_pred))/nrow(Y_true)
}

accuracy <- function(Y_true, Y_pred) {
  # both arguments are one-hot encoded
  Y_true_decoded <- apply(Y_true, 1, which.max)
  Y_pred_decoded <- apply(Y_pred, 1, which.max)
  # proportion of equal corresponding pairs:
  mean(Y_true_decoded == Y_pred_decoded)
}
```

It may be shown (\*\*) that the gradient of cross-entropy (with respect to the parameter matrix  $\mathbf{B}$ ) can be expressed in the matrix form as:

$$\frac{1}{n} \dot{\mathbf{X}}^T (\hat{\mathbf{Y}} - \mathbf{Y})$$

```
grad_cross_entropy <- function(X, Y_true, Y_pred) {
  t(X) %*% (Y_pred-Y_true)/nrow(Y_true)
}
```

Of course, we could always substitute the gradient with the finite difference approximation. Yet, this would be much slower).

The more mathematically inclined reader will surely notice that by expanding the formulas given in the previous chapter, we can write cross-entropy in the non-matrix form ( $n$  – number of samples,  $K$  – number of classes,  $p + 1$  – number of model parameters; in our case  $K = 10$  and  $p = 784$ ) as:

$$\begin{aligned}
E(\mathbf{B}) &= -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K y_{i,k} \log \left( \frac{\exp \left( \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,k} \right)}{\sum_{c=1}^K \exp \left( \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,c} \right)} \right) \\
&= \frac{1}{n} \sum_{i=1}^n \left( \log \left( \sum_{k=1}^K \exp \left( \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,k} \right) \right) - \sum_{k=1}^K y_{i,k} \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,k} \right).
\end{aligned}$$

Partial derivatives of cross-entropy w.r.t.  $\beta_{a,b}$  in non-matrix form can be derived (\*\*\*) so as to get:

$$\begin{aligned}
\frac{\partial E}{\partial \beta_{a,b}}(\mathbf{B}) &= \frac{1}{n} \sum_{i=1}^n \dot{x}_{i,a} \left( \frac{\exp \left( \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,b} \right)}{\sum_{k=1}^K \exp \left( \sum_{j=0}^p \dot{x}_{i,j} \beta_{j,k} \right)} - y_{i,b} \right) \\
&= \frac{1}{n} \sum_{i=1}^n \dot{x}_{i,a} (\hat{y}_{i,b} - y_{i,b}).
\end{aligned}$$

**Safe landing zone.** In case you're lost with the above, continue from here. However, in the near future, harden up and revisit the skipped material to get the most out of our discussion.

We now have all the building blocks to implement the gradient descent method. The algorithm below follows exactly the same scheme as the one in the  $g$  function example. This time, however, we play with a parameter matrix  $\mathbf{B}$  (not a parameter vector  $[x_1, x_2]$ ) and we compute the gradient of cross-entropy (by means of `grad_cross_entropy()`), not the gradient of  $g$ .

Note that a call to `system.time(expr)` measures the time (in seconds) spent evaluating an expression `expr`.

```

# random matrix of size 785x10 - initial guess
B <- matrix(rnorm(ncol(X_train1)*ncol(Y_train2)),
             nrow=ncol(X_train1))
eta <- 0.1 # learning rate
maxit <- 100 # number of GD iterations
system.time({ # measure time spent
    # for simplicity, we stop only when we reach maxit
    for (i in 1:maxit) {
        B <- B - eta*grad_cross_entropy(
            X_train1, Y_train2, nn_predict(B, X_train1))
    }
})

```

```

    }
}) # `user` - processing time in seconds:

##   user  system elapsed
## 48.798   7.757 32.784

```

Unfortunately, the method's convergence is really slow (we are optimising over 7850 parameters...) and the results after 100 iterations are disappointing:

```

accuracy(Y_train2, nn_predict(B, X_train1))

## [1] 0.4646167

accuracy(Y_test2, nn_predict(B, X_test1))

## [1] 0.4735

```

Recall that in the previous chapter we obtained much better classification accuracy by using the `keras` package. What are we doing wrong then? Maybe `keras` implements some Super-Fancy Hyper Optimisation Framework (TM) (R) that we could get access to for only \$19.99 per month?

### 6.3.5 Stochastic Gradient Descent (SGD) (\*)

It turns out that there's a very simple cure for the slow convergence of our method.

It might be shocking for some, but sometimes the true global minimum of cross-entropy for the whole training set is not exactly what we *really* want. In our predictive modelling task, we are minimising train error, but what we actually desire is to minimise the test error (which we cannot refer to while training = no cheating!).

It is therefore rational to assume that both the train and the test set consist of random digits independently sampled from the set of “all the possible digits out there in the world”.

Looking at the original objective (cross-entropy):

$$E(\mathbf{B}) = -\frac{1}{n^{\text{train}}} \sum_{i=1}^{n^{\text{train}}} \log \Pr(Y = y_i^{\text{train}} | \mathbf{x}_{i,\cdot}^{\text{train}}, \mathbf{B}).$$

How about we try fitting to different random samples of the train set in each iteration of the gradient descent method instead of fitting to the whole train set?

$$E(\mathbf{B}) \simeq -\frac{1}{b} \sum_{i=1}^b \log \Pr(Y = y_{\text{random\_index}_i}^{\text{train}} | \mathbf{x}_{\text{random\_index}_i,\cdot}^{\text{train}}, \mathbf{B}),$$

where  $b$  is some fixed batch size.

Such a scheme is often called **stochastic gradient descent**.

**Remark.** Technically, this is sometimes referred to as **mini-batch** gradient descent; there are a few variations popular in the literature, we pick the most intuitive now.

Stochastic gradient descent can be implemented very easily:

```
B <- matrix(rnorm(ncol(X_train1)*ncol(Y_train2)),
            nrow=ncol(X_train1))
eta <- 0.1
maxit <- 100
batch_size <- 32
system.time({
  for (i in 1:maxit) {
    wh <- sample(nrow(X_train1), size=batch_size)
    B <- B - eta*grad_cross_entropy(
      X_train1[wh,], Y_train2[wh,],
      nn_predict(B, X_train1[wh,]))
  }
})
##    user  system elapsed
##  0.344   0.007   0.093
accuracy(Y_train2, nn_predict(B, X_train1))

## [1] 0.4619833
accuracy(Y_test2, nn_predict(B, X_test1))

## [1] 0.4693
```

The errors are much worse but at least we got the (useless) solution very quickly. That's the “fail fast” rule in practice.

However, why don't we increase the number of iterations and see what happens? We've allowed the classic gradient descent to scrabble around the MNIST dataset for almost 2 minutes.

```
B <- matrix(rnorm(ncol(X_train1)*ncol(Y_train2)),
            nrow=ncol(X_train1))
eta <- 0.1
maxit <- 10000
batch_size <- 32
system.time({
  for (i in 1:maxit) {
    wh <- sample(nrow(X_train1), size=batch_size)
```

```

        B <- B - eta*grad_cross_entropy(
          X_train1[wh,], Y_train2[wh,],
          nn_predict(B, X_train1[wh,])
        )
      }
}

##   user  system elapsed
## 30.604   0.347   7.743

accuracy(Y_train2, nn_predict(B, X_train1))

## [1] 0.8922167

accuracy(Y_test2, nn_predict(B, X_test1))

## [1] 0.8935

```

Bingo! Let's take a closer look at how the train/test error behaves in each iteration for different batch sizes. Figures 6.13 and 6.14 depict the cases of `batch_size` of 32 and 128, respectively.

The time needed to go through 10000 iterations with batch size of 32 is:

```

##   user  system elapsed
## 75.064   0.032 32.161

```

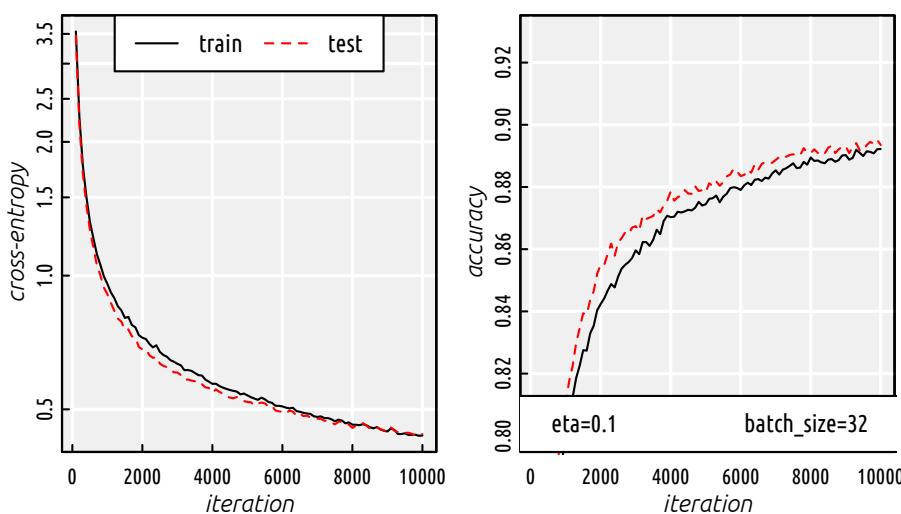


Figure 6.13: Cross-entropy and accuracy on the train and test set in each iteration of SGD; batch size of 32

What's more, batch size of 128 takes:

```
##      user    system elapsed
## 157.655    0.021  52.851
```

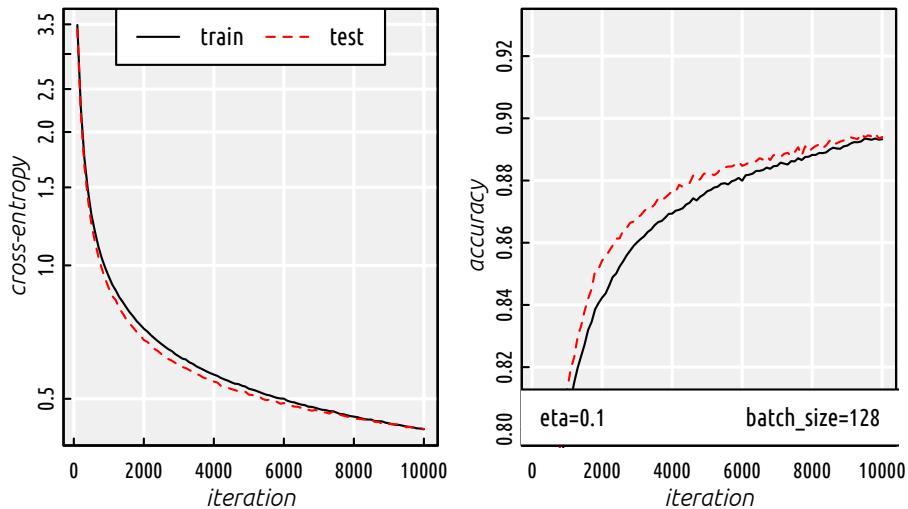


Figure 6.14: Cross-entropy and accuracy on the train and test set in each iteration of SGD; batch size of 128

**Exercise.**

*Draw conclusions.*

□

## 6.4 Outro

### 6.4.1 Remarks

Solving continuous problems with many variables (e.g., deep neural networks) is time consuming – the more variables to optimise over (e.g., model parameters, think the number of interconnections between all the neurons), the slower the optimisation process.

Moreover, it might be the case that the sole objective function takes long to compute (think of image classification with large training samples).

**Remark.** (\*) Although theoretically possible, good luck fitting a logistic regression model to MNIST with `optim()`'s BFGS – there are 7850 variables!

Training *deep* neural networks with SGD is even slower (more parameters), but there is a trick to propagate weight updates layer by layer, called *backpropagation*

(actually used in every neural network library), see, e.g., (Sarle & others 2002) and (Goodfellow et al. 2016). Moreover, `keras` and similar libraries implement automatic differentiation procedures that make its user’s life much easier (swiping some of the tedious math under the comfy carpet).

`keras` implements various optimisers that we can refer to in the `compile()` function, see <https://keras.rstudio.com/reference/compile.html> and <https://keras.io/optimizers/>:

- `SGD` – stochastic gradient descent supporting momentum and learning rate decay,
- `RMSprop` – divides the gradient by a running average of its recent magnitude,
- `Adam` – adaptive momentum

and so on. These are all non-complicated variations of the pure stochastic GD. Some of them are just tricks that work well in some examples and destroy the convergence on other ones. You can get into their details in a dedicated book/course aimed at covering neural networks (see, e.g., (Sarle & others 2002), (Goodfellow et al. 2016)), but we have already developed some good intuitions here.

Keep in mind that with methods such as GD or SGD, there is no guarantee we reach a minimum, but an approximate solution is better than no solution at all. Also sometimes (especially in ML applications) we don’t really need the actual minimum (with respect to the train set). Those “mathematically pure” will find that a bit... unaesthetic, but here we are. Better *a* solution than no solution at all, remember? But... is it really always the case though?

#### 6.4.2 Further Reading

Recommended further reading: (Nocedal & Wright 2006) and (Fletcher 2008).



# Chapter 7

## Clustering

### 7.1 Unsupervised Learning

#### 7.1.1 Introduction

In **unsupervised learning** (learning without a teacher), the input data points  $\mathbf{x}_{1,\cdot}, \dots, \mathbf{x}_{n,\cdot}$  are not assigned any reference labels (compare Figure 7.1).

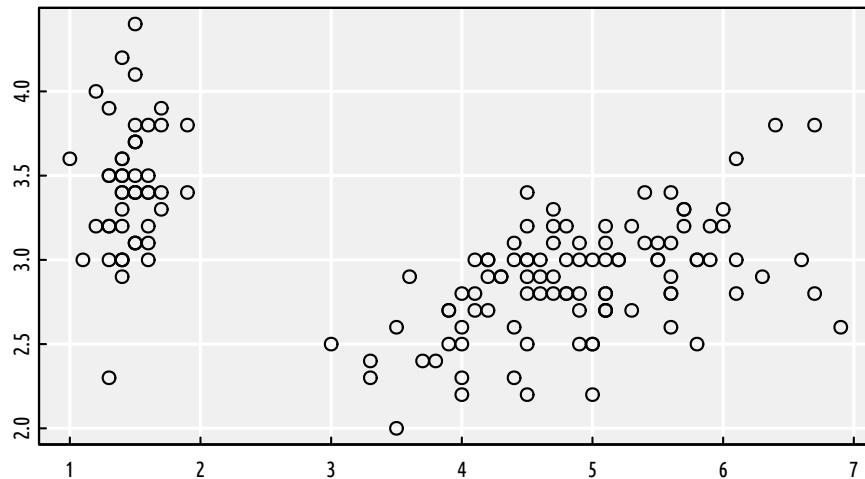


Figure 7.1: Unsupervised learning: “But what it is exactly that I have to do here?”

Our aim now is to discover the **underlying structure in the data**, whatever that means.

### 7.1.2 Main Types of Unsupervised Learning Problems

It turns out, however, that certain classes of unsupervised learning problems are not only intellectually stimulating, but practically useful at the same time.

In particular, in **dimensionality reduction** we seek a meaningful *projection* of a high dimensional space (think: many variables/columns).

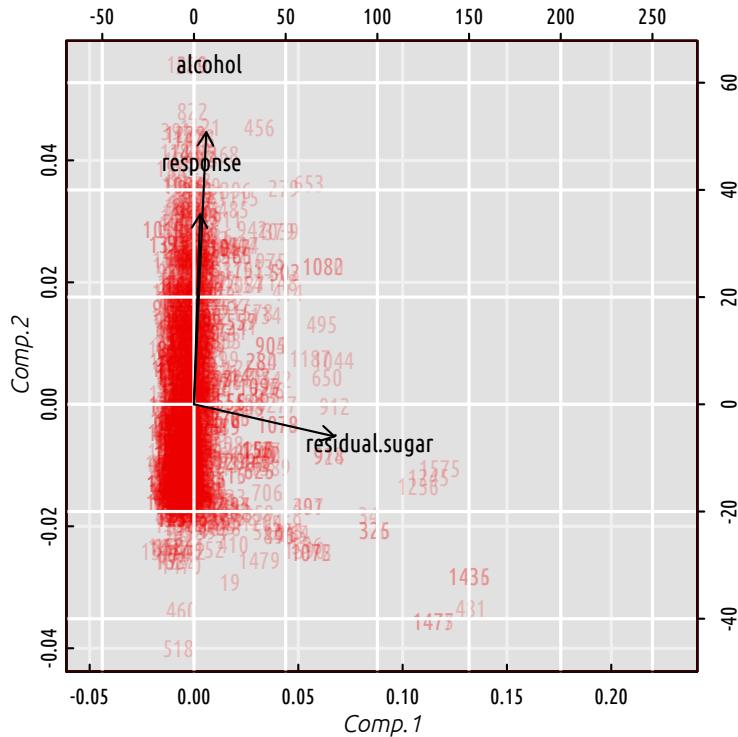


Figure 7.2: Principal component analysis (a dimensionality reduction technique) applied on three features of red wines

For instance, Figure 7.2 reveals that the “alcohol”, “response” and “residual.sugar” dimensions of the Wine Quality dataset that we have studied earlier on can actually be nicely depicted (with no much loss of information) on a two-dimensional plot. It turns out that the wine experts’ opinion on a wine’s quality is highly correlated with the amount of... alcohol in a bottle. On the other hand, sugar is orthogonal (unrelated) to these two.

Amongst example dimensionality reduction methods we find:

- Multidimensional scaling (MDS)
- Principal component analysis (PCA)
- Kernel PCA
- t-SNE
- Autoencoders (deep learning)

See, for example, (Hastie et al. 2017) for more details.

Furthermore, in **anomaly detection**, our task is to identify rare, suspicious, ab-normal or out-standing items. For example, these can be cars on walkways in a park's security camera footage.

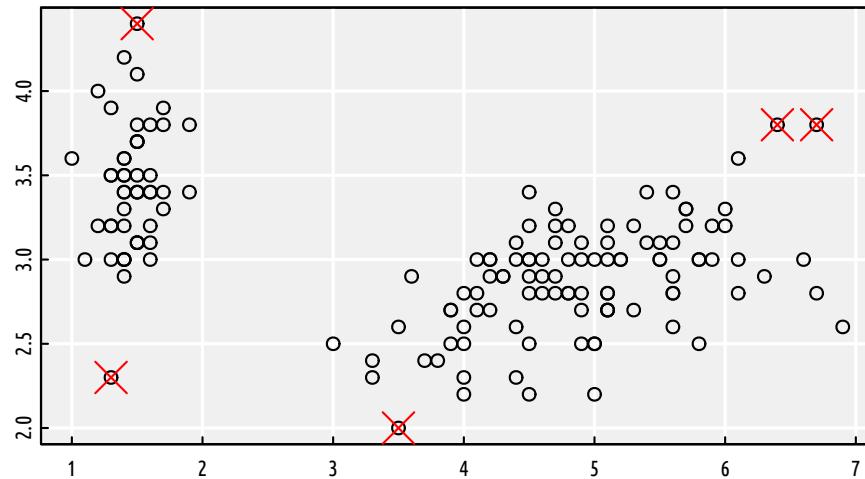


Figure 7.3: Outliers can be thought of anomalies of some sort

Finally, the aim of **clustering** is to automatically discover some *naturally occurring* subgroups in the data set. For example, these may be customers having different shopping patterns (such as “young parents”, “students”, “boomers”).

### 7.1.3 Definitions

Formally, given  $K \geq 2$ , **clustering** aims is to find a *special kind* of a  **$K$ -partition** of the input data set  $\mathbf{X}$ .

**Definition.** We say that  $\mathcal{C} = \{C_1, \dots, C_K\}$  is a  **$K$ -partition** of  $\mathbf{X}$  of size  $n$ , whenever:

- $C_k \neq \emptyset$  for all  $k$  (each set is nonempty),

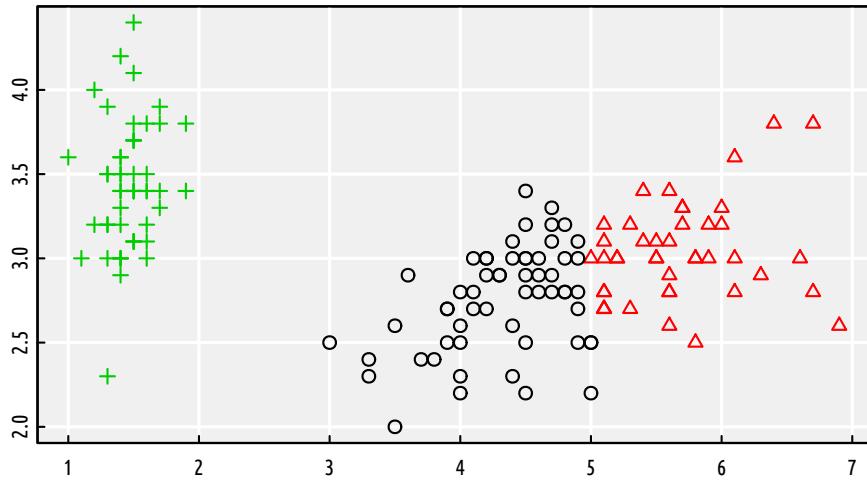


Figure 7.4: NEWS FLASH! SCIENTISTS SHOWED (by writing about it) THAT SOME VERY IMPORTANT THING (Iris dataset) COMES IN THREE DIFFERENT FLAVOURS (by applying the 3-means clustering algorithm)!

- $C_k \cap C_l = \emptyset$  for all  $k \neq l$  (sets are pairwise disjoint),
- $\bigcup_{k=1}^K C_k = \mathbf{X}$  (no point is neglected).

This can also be thought of as assigning each point a unique label  $\{1, \dots, K\}$  (think: colouring of the points, where each number has a colour). We will consider the point  $\mathbf{x}_{i,\cdot}$  as labelled  $j$  if and only if it belongs to cluster  $C_j$ , i.e.,  $\mathbf{x}_{i,\cdot} \in C_j$ .

Example applications of clustering:

- *taxonomisation*: e.g., partition the consumers to more “uniform” groups to better understand who they are and what do they need,
- *image processing*: e.g., object detection, like tumour tissues on medical images,
- *complex networks analysis*: e.g., detecting communities in friendship, retweets and other networks,
- *fine-tuning supervised learning algorithms*: e.g., recommender systems indicating content that was rated highly by users from the same group or learning multiple manifolds in a dimension reduction task.

The number of possible  $K$ -partitions of a set with  $n$  elements is given by *the Stirling number of the second kind*:

$$\left\{ \begin{array}{c} n \\ K \end{array} \right\} = \frac{1}{K!} \sum_{j=0}^K (-1)^{K-j} \binom{K}{j} j^n;$$

e.g., already  $\left\{ \begin{array}{c} n \\ 2 \end{array} \right\} = 2^{n-1} - 1$  and  $\left\{ \begin{array}{c} n \\ 3 \end{array} \right\} = O(3^n)$  – that is a lot. Certainly, we are not just interested in “any” partition – some of them will be more meaningful or valuable than others. However, even one of the most famous ML textbooks provides us with only a vague hint of what we might be looking for:

**“Definition”.** Clustering concerns “segmenting a collection of objects into subsets so that those within each cluster are more **closely related** to one another than objects assigned to different clusters” (Hastie et al. 2017).

It is not uncommon to equate the general definition of data clustering problems with... the particular outputs yield by specific clustering algorithms. In some sense, that sounds fair. From this perspective, we might be interested in identifying the two main types of clustering algorithms:

- **parametric** (model-based):
  - find clusters of specific shapes or following specific multidimensional probability distributions,
  - e.g.,  $K$ -means, expectation-maximisation for Gaussian mixtures (EM), average linkage agglomerative clustering;
- **nonparametric** (model-free):
  - identify high-density or well-separable regions, perhaps in the presence of noise points,
  - e.g., single linkage agglomerative clustering, Genie, (H)DBSCAN, BIRCH.

In this chapter we’ll take a look at two classical approaches to clustering:

- *K-means clustering* that looks for a specific number of clusters,
- *(agglomerative) hierarchical clustering* that outputs a whole hierarchy of nested data partitions.

## 7.2 K-means Clustering

### 7.2.1 Example in R

Let’s begin our clustering adventure by applying the  $K$ -means clustering method to find  $K = 3$  groups in the famous Fisher’s `iris` data set (variables `Sepal.Width` and `Petal.Length` variables only):

```
X <- as.matrix(iris[,c(3,2)])
# never forget to set nstart>>1!
km <- kmeans(X, centers=3, nstart=10)
km$cluster # labels assigned to each of 150 points:
```

**Remark.** Later we'll see that `nstart` is responsible for random restarting the (local) optimisation procedure, just as we did in the previous chapter.

Let's draw a scatter plot that depicts the detected clusters:

```
plot(X, col=km$cluster)
```

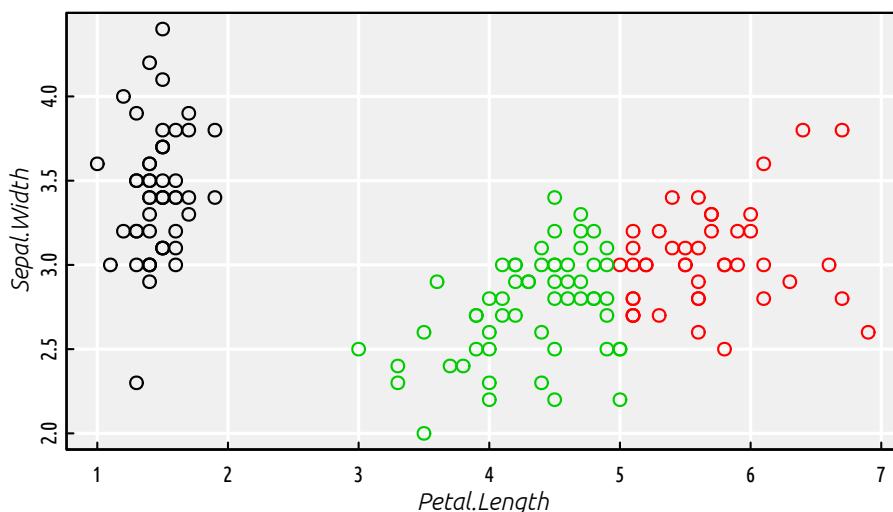


Figure 7.5: 3-means clustering on a projection of the Iris dataset

The colours in Figure 7.5 indicate the detected clusters. The left group is clearly well-separated from the other two.

What can we do with this information? Well, if we were experts on plants (in the 1930s), that'd definitely be something ground-breaking. Figure 7.6 is a version of the aforementioned scatter plot now with the true iris species added.

```
plot(X, col=km$cluster, pch=as.numeric(iris$Species))
```

Here is a contingency table for detected clusters vs. true iris species:

```
(C <- table(km$cluster, iris$Species))
```

```
##          setosa  versicolor  virginica
```

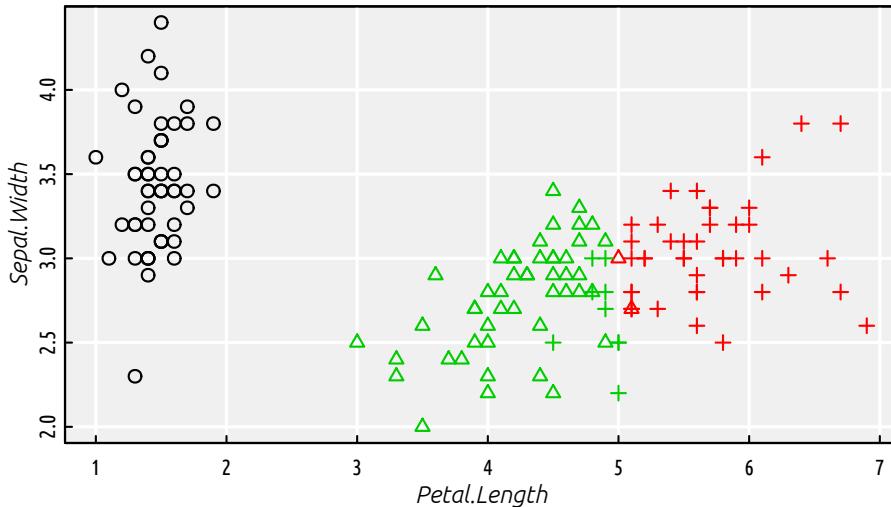


Figure 7.6: 3-means clustering (colours) vs true Iris species (shapes)

```

##   1      50      0      0
##   2      0      2     41
##   3      0     48      9
  
```

It turns out that the discovered partition matches the original iris species very well. We have just made a “discovery” in the field of botany (actually some research fields classify their objects of study into families, genres etc. by means of such tools).

Were the actual Iris species what we had hoped to match? Was that our aim? Well, surely we have had begun our journey with “clear minds” (yet with hungry eyes). Note that the true class labels were not used during the clustering procedure – we’re dealing with an unsupervised learning problem here. The result turned useful, it’s a win.

**Remark.** (\*) There are several indices that assess the similarity of two partitions, for example the Adjusted Rand Index (ARI) or the Normalised Mutual Information Score (NMI), see, e.g., (Hubert & Arabie 1985).

### 7.2.2 Problem Statement

The aim of *K-means clustering* is to find  $K$  “good” cluster centres  $\mu_{1,\cdot}, \dots, \mu_{K,\cdot}$ .

Then, a point  $\mathbf{x}_{i,\cdot}$  will be assigned to the cluster represented by the closest centre. Here, by *closest* we mean the *squared* Euclidean distance.

More formally, assuming all the points are in a  $p$ -dimensional space,  $\mathbb{R}^p$ , we define the distance between the  $i$ -th point and the  $k$ -th centre as:

$$d(\mathbf{x}_{i,\cdot}, \boldsymbol{\mu}_{k,\cdot}) = \|\mathbf{x}_{i,\cdot} - \boldsymbol{\mu}_{k,\cdot}\|^2 = \sum_{j=1}^p (x_{i,j} - \mu_{k,j})^2$$

Then the  $i$ -th point's cluster is determined by:

$$C(i) = \arg \min_{k=1,\dots,K} d(\mathbf{x}_{i,\cdot}, \boldsymbol{\mu}_{k,\cdot}),$$

where, as usual,  $\arg \min$  (argument minimum) is the index  $k$  that minimises the given expression.

In the previous example, the three identified cluster centres in  $\mathbb{R}^2$  are given by (see Figure 7.7 for illustration):

```
km$centers
```

```
##   Petal.Length Sepal.Width
## 1      1.462000    3.428000
## 2      5.672093    3.032558
## 3      4.328070    2.750877
plot(X, col=km$cluster, asp=1) # asp=1 gives the same scale on both axes
points(km$centers, cex=2, col=4, pch=16)
```

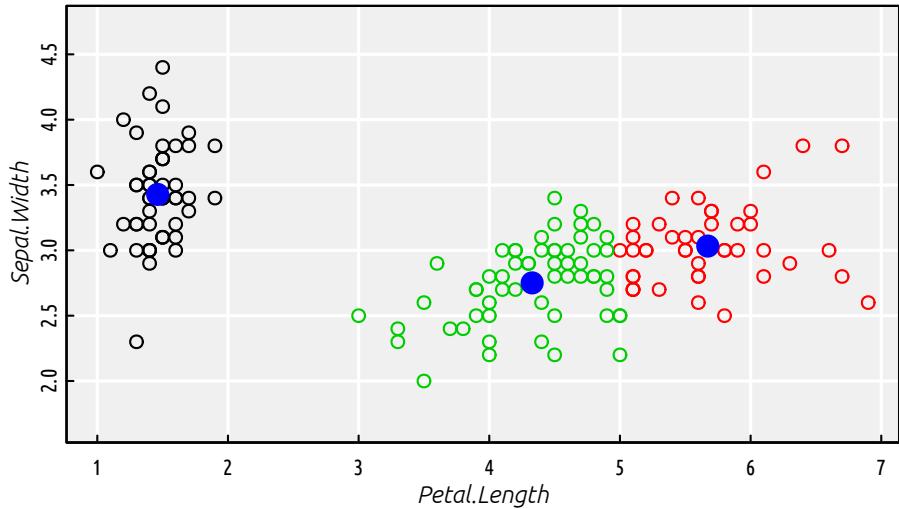


Figure 7.7: Cluster centres (blue dots) identified by the 3-means algorithm

Figure 7.8 depicts the partition of the whole  $\mathbb{R}^2$  space into clusters based on the closeness to the three cluster centres.

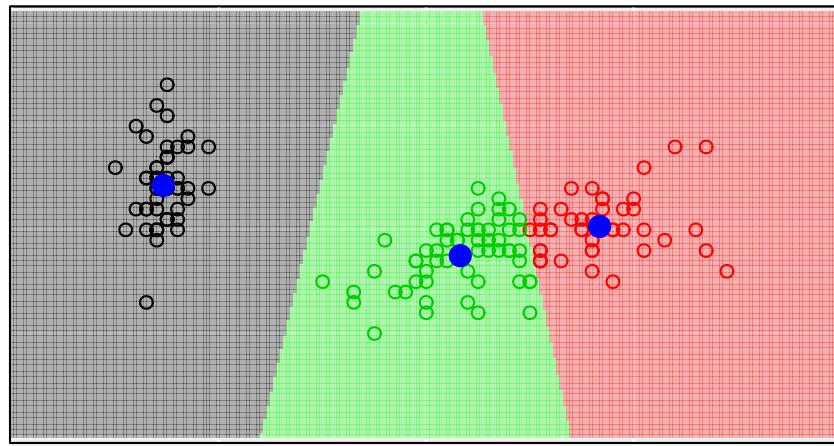


Figure 7.8: The division of the whole space into three sets based on the proximity to cluster centres (a so-called Voronoi diagram)

To compute the distances between all the points and the cluster centres, we may call `pdist::pdist()`:

```
library("pdist")
D <- as.matrix(pdist(X, km$centers)) ^ 2
head(D)
```

	[,1]	[,2]	[,3]
## [1,]	0.009028008	18.46928	9.134779
## [2,]	0.187028001	18.25184	8.635656
## [3,]	0.078227995	19.14323	9.370921
## [4,]	0.109028036	17.41091	8.119867
## [5,]	0.033427977	18.57277	9.294603
## [6,]	0.279428145	16.52998	8.227236

where  $D[i,k]$  gives the squared Euclidean distance between  $\mathbf{x}_{i,:}$  and  $\boldsymbol{\mu}_{k,:}$ .

The cluster memberships ( $\arg \min$ s) can now be determined by:

```
all(km$cluster == idx) # sanity check
## [1] TRUE
```

### 7.2.3 Algorithms for the K-means Problem

All good, but how do we find “good” cluster centres? Good, better, best... yet again we are in a need for a goodness-of-fit metric. In the  $K$ -means clustering, we determine  $\mu_{1,.}, \dots, \mu_{K,.}$  that minimise the total within-cluster distances (distances from each point to each own cluster centre):

$$\min_{\mu_{1,.}, \dots, \mu_{K,.} \in \mathbb{R}^p} \sum_{i=1}^n d(\mathbf{x}_{i,.}, \mu_{C(i),.}),$$

Note that the  $\mu$ s are also “hidden” inside the point-to-cluster belongingness mapping,  $C$ . Expanding the above yields:

$$\min_{\mu_{1,.}, \dots, \mu_{K,.} \in \mathbb{R}^p} \sum_{i=1}^n \left( \min_{k=1, \dots, K} \sum_{j=1}^p (x_{i,j} - \mu_{k,j})^2 \right).$$

Unfortunately, the min operator in the objective function makes this optimisation problem not tractable with the methods discussed in the previous chapter.

The above problem is *hard* to solve (\* more precisely, it is an NP-hard problem). Therefore, in practice we use various heuristics to solve it. The `kmeans()` function itself implements 4 of them: the Hartigan-Wong, Lloyd, Forgy and MacQueen algorithms.

**Remark.** (\*) Technically, there is no such thing as “the K-means algorithm” – all the aforementioned methods are particular heuristic approaches to solving the K-means clustering problem formalised as the above optimisation task. By setting `nstart = 10` above, we ask the (Hartigan-Wong, which is the default one in `kmeans()`) algorithm to find 10 solution candidates obtained by considering different random initial clusterings and choose the best one (with respect to the sum of within-cluster distances) amongst them. This does not guarantee finding the optimal solution, especially for very unbalanced datasets, but increases the likelihood of such.

**Remark.** The *squared* Euclidean distance was of course chosen to make computations easier. It turns out that for any given subset of input points  $\mathbf{x}_{i_1,.}, \dots, \mathbf{x}_{i_m,.}$ , the point  $\mu_{k,.}$  that minimises the total distances to all of them, i.e.,

$$\min_{\boldsymbol{\mu}_{k,\cdot} \in \mathbb{R}^p} \sum_{\ell=1}^m \left( \sum_{j=1}^p (x_{i_\ell,j} - \mu_{k,j})^2 \right),$$

is exactly these points' *centroid* – which is given by the componentwise arithmetic means of their coordinates.

For example:

```
colMeans(X[km$cluster == 1,]) # centroid of the points in the 1st cluster

## Petal.Length Sepal.Width
##      1.462      3.428

km$centers[1,] # the centre of the 1st cluster

## Petal.Length Sepal.Width
##      1.462      3.428
```

Among the various heuristics to solve the K-means problem, Lloyd's algorithm (1957) is perhaps the simplest. This is probably the reason why it is sometimes referred to as “the” K-means algorithm:

1. Start with random cluster centres  $\boldsymbol{\mu}_{1,\cdot}, \dots, \boldsymbol{\mu}_{K,\cdot}$ .
2. For each point  $\mathbf{x}_{i,\cdot}$ , determine its closest centre  $C(i) \in \{1, \dots, K\}$ :

$$C(i) = \arg \min_{k=1, \dots, K} d(\mathbf{x}_{i,\cdot}, \boldsymbol{\mu}_{k,\cdot}).$$

3. For each cluster  $k \in \{1, \dots, K\}$ , compute the new cluster centre  $\boldsymbol{\mu}_{k,\cdot}$  as the centroid of all the point indices  $i$  such that  $C(i) = k$ .
4. If the cluster centres changed since the last iteration, go to step 2, otherwise stop and return the result.

(\*) Here's an example implementation. As the initial cluster centres, let's pick some “noisy” versions of  $K$  randomly chosen points in  $\mathbf{X}$ .

```
set.seed(12345)
K <- 3

# Random initial cluster centres:
M <- jitter(X[sample(1:nrow(X), K),])
M

##      Petal.Length Sepal.Width
## [1,]      5.100369     3.081381
```

```
## [2,]    4.709108   3.186095
## [3,]    3.319589   2.409427
```

In what follows, we will be maintaining a matrix such that  $D[i,k]$  is the distance between the  $i$ -th point and the  $k$ -th centre and a vector such that  $\text{idx}[i]$  denotes the index of the cluster centre closest to the  $i$ -th point.

```
D <- as.matrix(pdist(X, M)) ^ 2
idx <- apply(D, 1, which.min)

repeat {
  # Determine the new cluster centres:
  M <- t(sapply(1:K, function(k) {
    # the centroid of all points in the k-th cluster:
    colMeans(X[idx == k,])
  }))

  # Store the previous cluster belongingness info:
  old_idx <- idx

  # Recompute D and idx:
  D <- as.matrix(pdist(X, M)) ^ 2
  idx <- apply(D, 1, which.min)

  # Check if converged already:
  if (all(idx == old_idx)) break
}
```

Let's compare the obtained cluster centres with the ones returned by `kmeans()`:

```
M # our result

##      Petal.Length Sepal.Width
## [1,]      5.672093   3.032558
## [2,]      4.328070   2.750877
## [3,]      1.462000   3.428000

km$center # result of kmeans()

##      Petal.Length Sepal.Width
## 1      1.462000   3.428000
## 2      5.672093   3.032558
## 3      4.328070   2.750877
```

These two represent exactly the same 3-partitions (note that the actual labels (the order of centres) are not important).

The value of the objective function (total within-cluster distances) at the identified candidate solution is equal to:

```

sum(D[cbind(1:nrow(X),idx)]) # indexing with a 2-column matrix!
## [1] 40.73708
km$tot.withinss # as reported by kmeans()
## [1] 40.73707

```

We would need it if we were to implement the `nstart` functionality, which is left as an:

**Exercise.**

(\*) Wrap the implementation of the Lloyd algorithm into a standalone R function, with a similar look-and-feel as the original `kmeans()`.

□

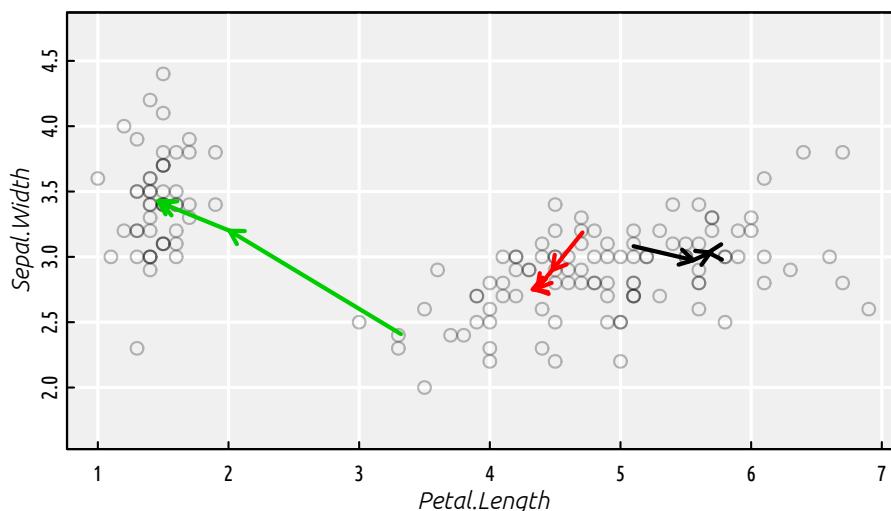


Figure 7.9: The arrows denote the cluster centres in each iteration of the Lloyd algorithm

On a side note, our algorithm needed 4 iterations to identify the (locally optimal) cluster centres. Figure 7.9 depicts its quest for the clustering grail.

## 7.3 Agglomerative Hierarchical Clustering

### 7.3.1 Introduction

In K-means, we need to specify the number of clusters,  $K$ , in advance. What if we don't have any idea how to choose this parameter (which is often the case)?

Also, the problem with K-means is that there is no guarantee that a  $K$ -partition is any “similar” to the  $K'$ -one for  $K \neq K'$ , see Figure 7.10.

```
km1 <- kmeans(X, 3, nstart=10)
km2 <- kmeans(X, 4, nstart=10)
plot(X, col=km1$cluster, pch=km2$cluster, asp=1)
```

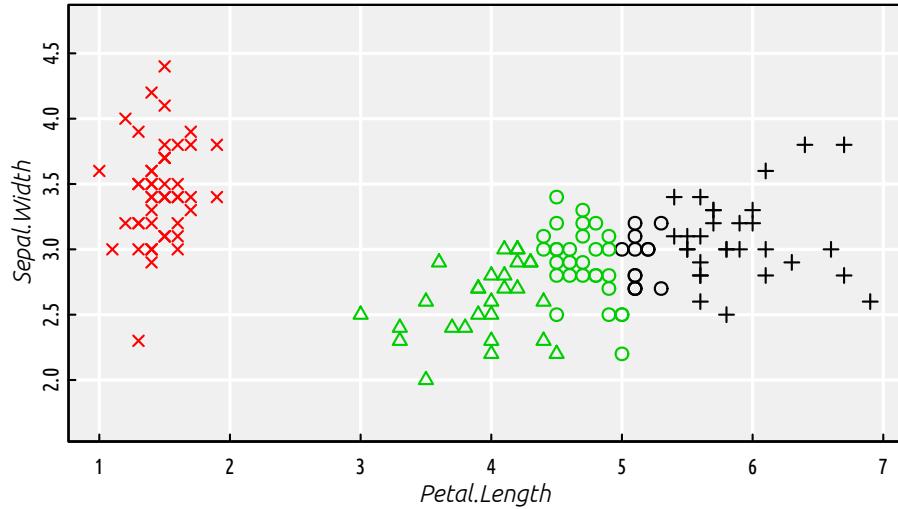


Figure 7.10: 3-means (colours) vs. 4-means (symbols) on example data; the “circle” cluster cannot decide if it likes the green or the black one more

Hierarchical methods, on the other hand, output a whole hierarchy of mutually *nested* partitions, which increase the interpretability of the results. A  $K$ -partition for any  $K$  can be extracted later at any time.

In this book we will be interested in *agglomerative* hierarchical algorithms:

- at the lowest level of the hierarchy, each point belongs to its own cluster (there are  $n$  singletons);
- at the highest level of the hierarchy, there is one cluster that embraces all the points;
- moving from the  $i$ -th to the  $(i+1)$ -th level, we select (somehow; see below) a pair of clusters to be merged.

### 7.3.2 Example in R

The most basic implementation of a few agglomerative hierarchical clustering algorithms is provided by the `hclust()` function, which works on a pairwise distance matrix.

```
# Euclidean distances between all pairs of points:
D <- dist(X)
# Apply Complete Linkage (the default, details below):
h <- hclust(D) # method="complete"
print(h)

##
## Call:
## hclust(d = D)
##
## Cluster method : complete
## Distance       : euclidean
## Number of objects: 150
```

**Remark.** There are  $n(n - 1)/2$  unique pairwise distances between  $n$  points.

Don't try calling `dist()` on large data matrices. Already  $n = 100,000$  points consumes 40 GB of available memory (assuming that each distance is stored as an 8-byte double-precision floating point number); packages `fastcluster` and `genie`, among other, aim to solve this problem.

The obtained hierarchy (`tree`) can be *cut* at an arbitrary level by applying the `cutree()` function.

```
cutree(h, k=3) # extract the 3-partition
```

```
## [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [30] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [59] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [88] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 3 3 3 3 2 3 3 3 3 2 3 3 3 2 2 2 2
## [117] 3 3 3 2 3 2 3 2 3 3 2 2 3 3 3 3 2 3 3 3 3 2 3 3 3 2 2 3 3
## [146] 2 2 2 3 2
```

The cuts of the hierarchy at different levels are depicted in Figure 7.11. The obtained 3-partition also matches the true Iris species quite well. However, now it makes total sense to "zoom" our partitioning in or out and investigate how are the subgroups decomposed or aggregated when we change  $K$ .

```
par(mfrow=c(2,2))
plot(X, col=cutree(h, k=5), ann=FALSE)
legend("top", legend="k=5", bg="white")
plot(X, col=cutree(h, k=4), ann=FALSE)
legend("top", legend="k=4", bg="white")
plot(X, col=cutree(h, k=3), ann=FALSE)
legend("top", legend="k=3", bg="white")
plot(X, col=cutree(h, k=2), ann=FALSE)
legend("top", legend="k=2", bg="white")
```

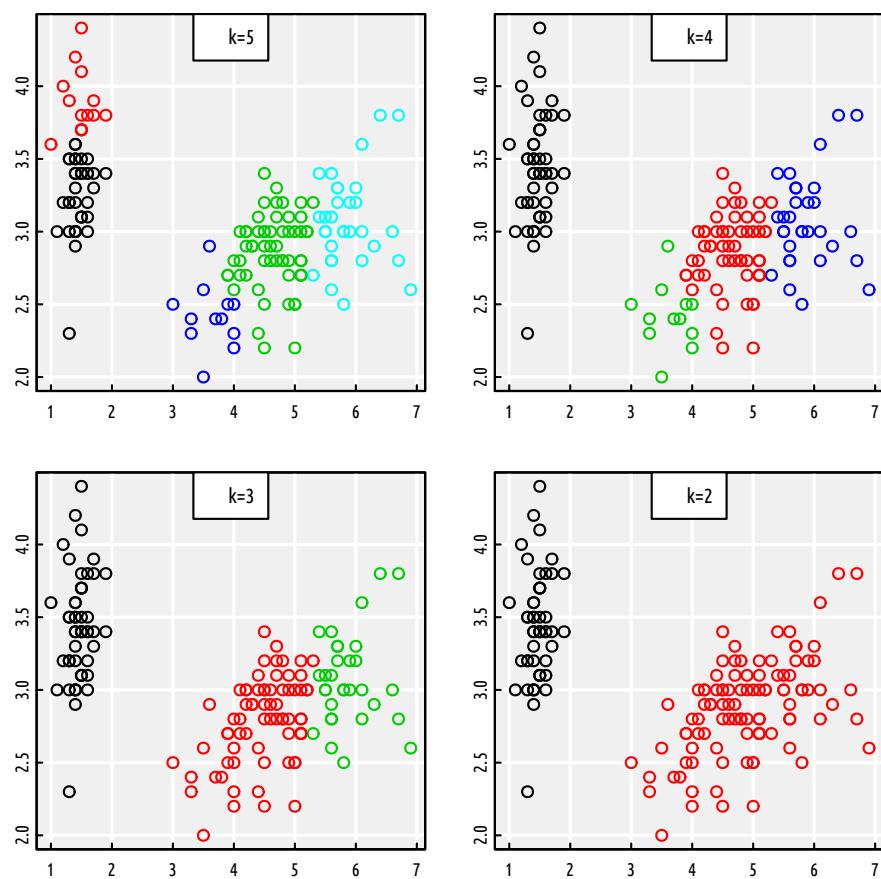


Figure 7.11: Complete linkage – 4 different cuts

### 7.3.3 Linkage Functions

Let's formalise the clustering process. Initially,  $\mathcal{C}^{(0)} = \{\{\mathbf{x}_{1,\cdot}\}, \dots, \{\mathbf{x}_{n,\cdot}\}\}$ , i.e., each point is a member of its own cluster.

While an agglomerative hierarchical clustering algorithm is being computed, there are  $n - k$  clusters at the  $k$ -th step of the procedure,  $\mathcal{C}^{(k)} = \{C_1^{(k)}, \dots, C_{n-k}^{(k)}\}$ .

When proceeding from step  $k$  to  $k + 1$ , we determine the two groups  $C_u^{(k)}$  and  $C_v^{(k)}$ ,  $u < v$ , to be *merged* together so that the clustering at the higher level is of the form:

$$\mathcal{C}^{(k+1)} = \left\{ C_1^{(k)}, \dots, C_{u-1}^{(k)}, C_u^{(k)} \cup C_v^{(k)}, C_{u+1}^{(k)}, \dots, C_{v-1}^{(k)}, C_{v+1}^{(k)}, \dots, C_{n-k}^{(k)} \right\}.$$

Thus,  $(\mathcal{C}^{(0)}, \mathcal{C}^{(1)}, \dots, \mathcal{C}^{(n-1)})$  form a sequence of *nested* partitions of the input dataset with the last level being just one big cluster,  $\mathcal{C}^{(n-1)} = \{\{\mathbf{x}_{1,\cdot}, \mathbf{x}_{2,\cdot}, \dots, \mathbf{x}_{n,\cdot}\}\}$ .

There is one component missing – how to determine the pair of clusters  $C_u^{(k)}$  and  $C_v^{(k)}$  to be merged with each other at the  $k$ -th iteration? Of course this will be expressed as some optimisation problem (although this time, a simple one)! The decision will be based on:

$$\arg \min_{u < v} d^*(C_u^{(k)}, C_v^{(k)}),$$

where  $d^*(C_u^{(k)}, C_v^{(k)})$  is the *distance* between two clusters  $C_u^{(k)}$  and  $C_v^{(k)}$ .

Note that we usually only consider the distances between *individual points*, not sets of points. Hence,  $d^*$  must be a suitable extension of a pointwise distance  $d$  (usually the Euclidean metric) to whole sets.

We will assume that  $d^*(\{\mathbf{x}_{i,\cdot}\}, \{\mathbf{x}_{j,\cdot}\}) = d(\mathbf{x}_{i,\cdot}, \mathbf{x}_{j,\cdot})$ , i.e., the distance between singleton clusters is the same as the distance between the points themselves. As far as more populous point groups are concerned, there are many popular choices of  $d^*$  (which in the context of hierarchical clustering we call *linkage functions*):

- single linkage:

$$d_S^*(C_u^{(k)}, C_v^{(k)}) = \min_{\mathbf{x}_{i,\cdot} \in C_u^{(k)}, \mathbf{x}_{j,\cdot} \in C_v^{(k)}} d(\mathbf{x}_{i,\cdot}, \mathbf{x}_{j,\cdot}),$$

- complete linkage:

$$d_C^*(C_u^{(k)}, C_v^{(k)}) = \max_{\mathbf{x}_{i,\cdot} \in C_u^{(k)}, \mathbf{x}_{j,\cdot} \in C_v^{(k)}} d(\mathbf{x}_{i,\cdot}, \mathbf{x}_{j,\cdot}),$$

- average linkage:

$$d_A^*(C_u^{(k)}, C_v^{(k)}) = \frac{1}{|C_u^{(k)}||C_v^{(k)}|} \sum_{\mathbf{x}_{i,.} \in C_u^{(k)}} \sum_{\mathbf{x}_{j,.} \in C_v^{(k)}} d(\mathbf{x}_{i,.}, \mathbf{x}_{j,.}).$$

An illustration of the way different linkages are computed is given in Figure 7.12.

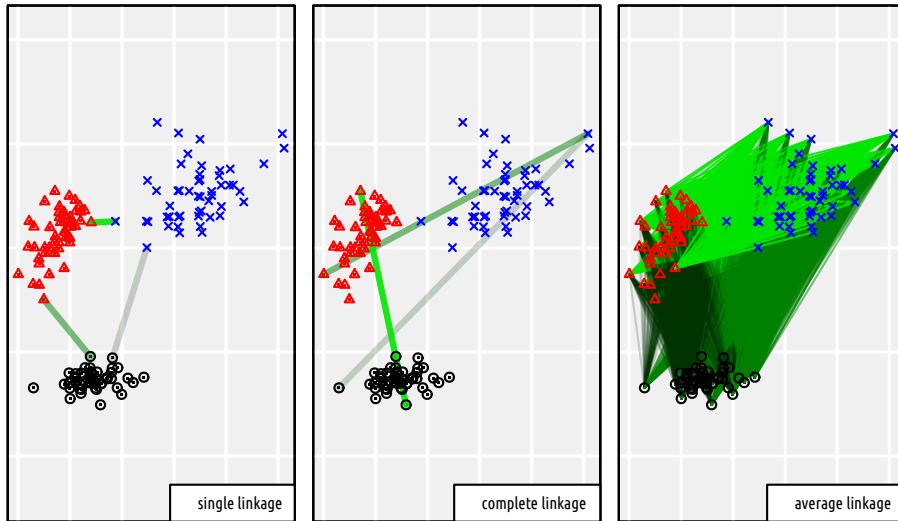


Figure 7.12: In single linkage, we find the closest pair of points; in complete linkage, we seek the pair furthest away from each other; in average linkage, we determine the arithmetic mean of all pairwise distances

Assuming  $d_S^*$ ,  $d_C^*$  or  $d_A^*$  in the aforementioned procedure leads to single, complete or average linkage-based agglomerative hierarchical clustering algorithms, respectively (referred to as single linkage etc. for brevity).

```
hs <- hclust(D, method="single")
hc <- hclust(D, method="complete")
ha <- hclust(D, method="average")
```

Figure 7.13 compares the 5-, 4- and 3-partitions obtained by applying the 3 above linkages. Note that it's in very nature of the single linkage algorithm that it's highly sensitive to outliers.

### 7.3.4 Cluster Dendograms

A *dendrogram* (which we can plot by calling `plot(h)`, where `h` is the result returned by `hclust()`) depicts the distances (as defined by the linkage function)

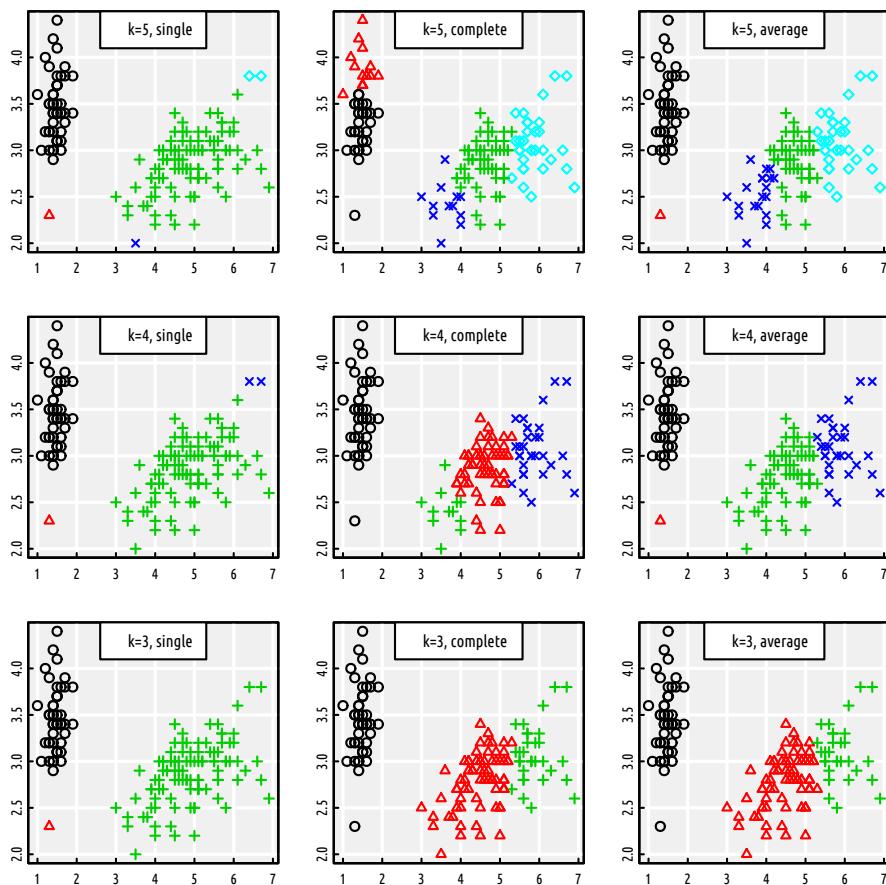


Figure 7.13: 3 cuts of 3 different hierarchies

between the clusters merged at every stage of the agglomerative procedure. This can provide us with some insight into the underlying data structure as well as with hints about at which level the tree could be cut.

Figure 7.14 depicts the three dendrograms that correspond to the clusterings obtained by applying different linkages. Each tree has 150 leaves (at the bottom) that represent the 150 points in our example dataset. Each “edge” (joint) represents a group of points being merged. For instance, the very top joint in the middle subfigure is located at height of  $\simeq 6$ , which is exactly the maximal pairwise distance (complete linkage) between the points in the last two last clusters.

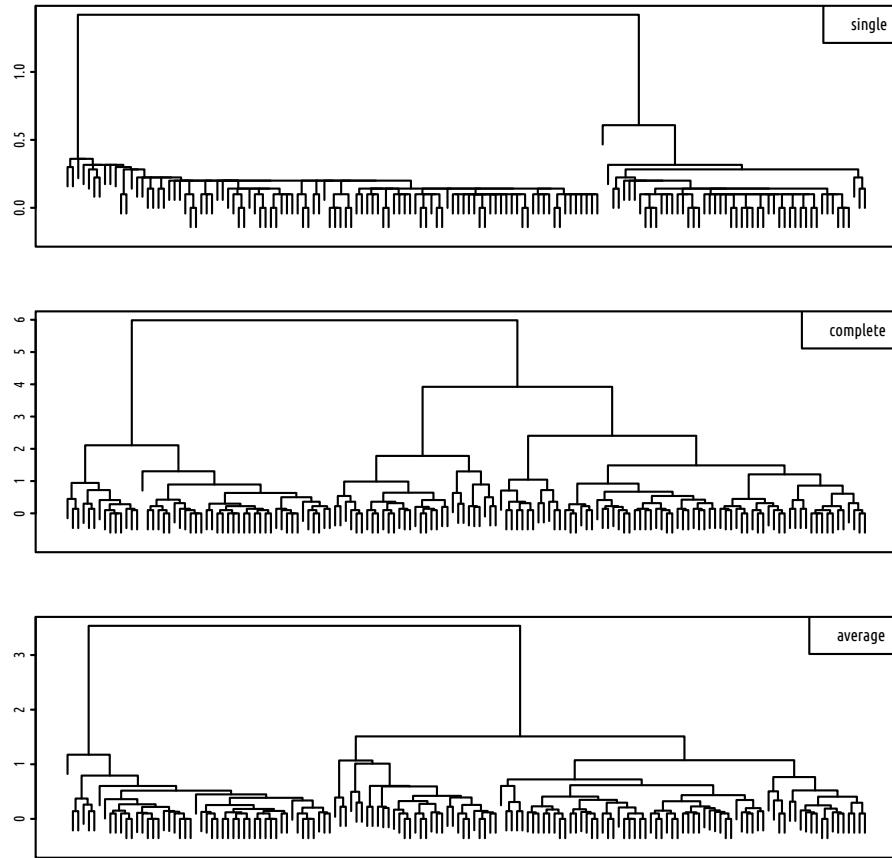


Figure 7.14: Cluster dendrograms for the single, complete and average linkages

## 7.4 Outro

### 7.4.1 Remarks

Unsupervised learning is often performed during the data pre-processing and exploration stage. Assessing the quality of clustering is particularly challenging as, unlike in a supervised setting, we have no access to “ground truth” information.

In practice, we often apply different clustering algorithms and just see where they lead us. There’s no teacher that would tell us what we should do, so whatever we do is awesome, right? Well, not precisely. Most frequently, you, my dear reader, will work for some party that’s genuinely interested in your explaining why did you spent the last month coming up with nothing useful at all. Thus, the main body of work related to proving the use-ful/less-ness will be on you.

Clustering methods can aid us in supervised tasks – instead of fitting a single “large model”, it might be useful to fit separate models to each cluster.

To sum up, the aim of K-means is to find  $K$  clusters based on the notion of the points’ closeness to the cluster centres. Remember that  $K$  must be set in advance. By definition (\* via its relation to Voronoi diagrams), all clusters will be of convex shapes.

However, we may try applying  $K'$ -means for  $K' \gg K$  to obtain a “fine grained” compressed representation of data and then combine the (sub)clusters into more meaningful groups using other methods (such as the hierarchical ones).

Iterative K-means algorithms are very fast (e.g., there is a mini-batch version of the algorithm) even for large data sets, but they may fail to find a desirable solution, especially if clusters are unbalanced.

Hierarchical methods, on the other hand, output a whole family of mutually nested partitions, which may provide us with insight into the underlying structure of data data. Unfortunately, there is no easy way to assign new points to existing clusters; yet, you can always build a classifier (e.g., a decision tree or a neural network) that learns the discovered labels.

A linkage scheme must be chosen with care, for instance, single linkage can be sensitive to outliers. However, it is generally the fastest. The methods implemented in `hclust()` are generally slow; they have time complexity between  $O(n^2)$  and  $O(n^3)$ .

**Remark.** Note that the `fastcluster` package provides a more efficient and memory-saving implementation of some methods available via a call to `hclust()`. See also the `genie` package for a super-robust version of the single linkage algorithm based on the datasets’s Euclidean minimum spanning tree, which can be computed quite quickly.

Finally, note that all the discussed clustering methods are based on the notion of pairwise distances. These of course tend to behave weirdly in high-dimensional spaces (“the curse of dimensionality”). Moreover, some hardcore feature engineering might be needed to obtain meaningful results.

### 7.4.2 Further Reading

Recommended further reading: (James et al. 2017: Section 10.3)

Other: (Hastie et al. 2017: Section 14.3)

Additionally, check out other noteworthy clustering approaches:

- Genie (see R package `genie`) (Gagolewski et al. 2016, Cena & Gagolewski 2020)
- ITM (Müller et al. 2012)
- DBSCAN, HDBSCAN\* (Ling 1973, Ester et al. 1996, Campello et al. 2015)
- K-medoids, K-medians
- Fuzzy C-means (a.k.a. weighted K-means) (Bezdek et al. 1984)
- Spectral clustering; e.g., (Ng et al. 2001)
- BIRCH (Zhang et al. 1996)

## Chapter 8

# Optimisation with Genetic Algorithms

### 8.1 Introduction

#### 8.1.1 Recap

Recall that an **optimisation task** deals with finding an element  $x$  in a **search space**  $D$ , that minimises or maximises an **objective function**  $f : D \rightarrow \mathbb{R}$ :

$$\min_{x \in D} f(x) \quad \text{or} \quad \max_{x \in D} f(x),$$

In one of the previous chapters, we were dealing with **unconstrained continuous optimisation**, i.e., we assumed the search space is  $D = \mathbb{R}^p$  for some  $p$ .

Example problems of this kind: minimising mean squared error in linear regression or cross-entropy in logistic regression.

The class of general-purpose iterative algorithms we've previously studied fit into the following scheme:

1.  $\mathbf{x}^{(0)}$  – initial guess (e.g., generated at random)
2. for  $i = 1, \dots, M$ :
  - a.  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + [\text{guessed direction, e.g., } -\eta \nabla f(\mathbf{x})]$
  - b. if  $|f(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i-1)})| < \varepsilon$  break

3. return  $\mathbf{x}^{(i)}$  as result

where:

- $M$  = maximum number of iterations
- $\varepsilon$  = tolerance, e.g.,  $10^{-8}$
- $\eta > 0$  = learning rate

The algorithms such as gradient descent and BFGS (see `optim()`) give satisfactory results in the case of **smooth and well-behaving objective functions**.

However, if an objective has, e.g., many plateaus (regions where it is almost constant), those methods might easily get stuck in local minima.

The K-means clustering's objective function is a not particularly pleasant one – it involves a nested search for the closest cluster, with the use of the `min` operator.

### 8.1.2 K-means Revisited

In **K-means clustering** we are minimising the squared Euclidean distance to each point's cluster centre:

$$\min_{\boldsymbol{\mu}_{1,\cdot}, \dots, \boldsymbol{\mu}_{K,\cdot} \in \mathbb{R}^p} \sum_{i=1}^n \left( \min_{k=1, \dots, K} \sum_{j=1}^p (x_{i,j} - \mu_{k,j})^2 \right).$$

This is an (NP-)hard problem! There is no efficient exact algorithm.

We need approximations. In the last chapter, we have discussed the iterative Lloyd's algorithm (1957), which is amongst a few procedures implemented in the `kmeans()` function.

To recall, Lloyd's algorithm (1957) is sometimes referred to as “the” K-means algorithm:

1. Start with random cluster centres  $\boldsymbol{\mu}_{1,\cdot}, \dots, \boldsymbol{\mu}_{K,\cdot}$ .
2. For each point  $\mathbf{x}_{i,\cdot}$ , determine its closest centre  $C(i) \in \{1, \dots, K\}$ .
3. For each cluster  $k \in \{1, \dots, K\}$ , compute the new cluster centre  $\boldsymbol{\mu}_{k,\cdot}$  as the componentwise arithmetic mean of the coordinates of all the point indices  $i$  such that  $C(i) = k$ .
4. If the cluster centres changed since last iteration, go to step 2, otherwise stop and return the result.

As the procedure might get stuck in a local minimum, a few restarts are recommended (as usual).

Hence, we are used to calling:

```
kmeans(X, centers=k, nstart=10)
```

### 8.1.3 optim() vs. kmeans()

Let us compare how a general-purpose optimiser such as the BFGS algorithm implemented in `optim()` compares with a customised, problem-specific solver.

We will need some benchmark data.

```
gen_cluster <- function(n, p, m, s) {
  vectors <- matrix(rnorm(n*p), nrow=n, ncol=p)
  unit_vectors <- vectors/sqrt(rowSums(vectors^2))
  unit_vectors*rnorm(n, 0, s)+rep(m, each=n)
}
```

The above function generates  $n$  points in  $\mathbb{R}^p$  from a distribution centred at  $\mathbf{m} \in \mathbb{R}^p$ , spread randomly in every possible direction with scale factor  $s$ .

Two example clusters in  $\mathbb{R}^2$ :

```
# plot the "black" cluster
plot(gen_cluster(500, 2, c(0, 0), 1), col="#00000022", pch=16,
      xlim=c(-3, 4), ylim=c(-3, 4), asp=1, ann=FALSE, las=1)
# plot the "red" cluster
points(gen_cluster(250, 2, c(1.5, 1), 0.5), col="#ff000022", pch=16)
```

Let's generate the benchmark dataset  $\mathbf{X}$  that consists of three clusters in a high-dimensional space.

```
set.seed(123)
p <- 32
Ns <- c(50, 100, 20)
Ms <- c(0, 1, 2)
s <- 1.5*p
K <- length(Ns)

X <- lapply(1:K, function(k)
  gen_cluster(Ns[k], p, rep(Ms[k], p), s))
X <- do.call(rbind, X) # rbind(X[[1]], X[[2]], X[[3]])
```

The objective function for the K-means clustering problem:

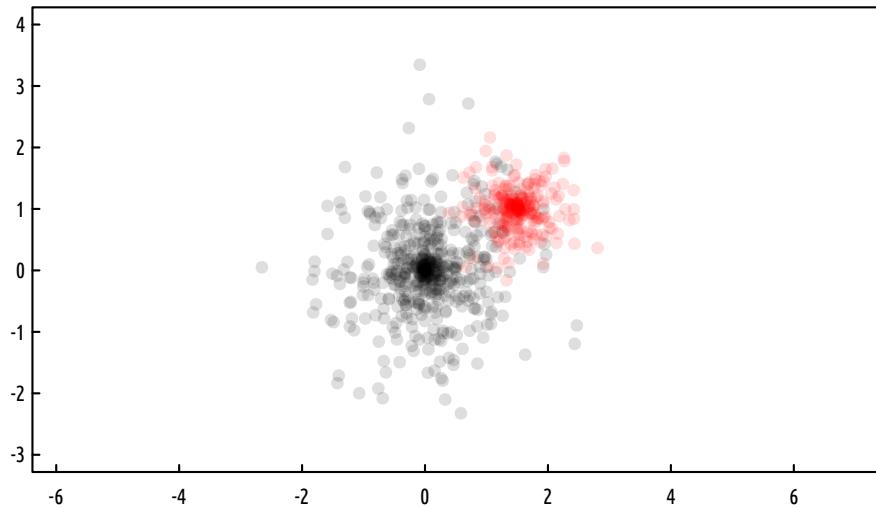


Figure 8.1: plot of chunk gendata\_example

```
library("FNN")
get_fitness <- function(mu, X) {
  # For each point in X,
  # get the index of the closest point in mu:
  memb <- FNN::get.knnx(mu, X, 1)$nn.index

  # compute the sum of squared distances
  # between each point and its closes cluster centre:
  sum((X-mu[memb,])^2)
}
```

Setting up the solvers:

```
min_HartiganWong <- function(mu0, X)
  get_fitness(
    # algorithm="Hartigan-Wong"
    kmeans(X, mu0, iter.max=100)$centers,
    X)
min_Lloyd <- function(mu0, X)
  get_fitness(
    kmeans(X, mu0, iter.max=100, algorithm="Lloyd")$centers,
    X)
min_optim <- function(mu0, X)
  optim(mu0,
        function(mu, X) {
```

```

        get_fitness(matrix(mu, nrow=nrow(mu0)), X)
    }, X=X, method="BFGS", control=list(reltol=1e-16)
)$val

```

Running the simulation:

```

nstart <- 100
set.seed(123)
res <- replicate(nstart, {
  mu0 <- X[sample(nrow(X), K),]
  c(
    HartiganWong=min_HartiganWong(mu0, X),
    Lloyd=min_Lloyd(mu0, X),
    optim=min_optim(mu0, X)
  )
})

```

Notice a considerable variability of the objective function at the local minima found:

```

par(mar=c(2, 6.5, 0.5, 0.5)) # figure margins
boxplot(as.data.frame(t(res)), horizontal=TRUE, las=1)

```

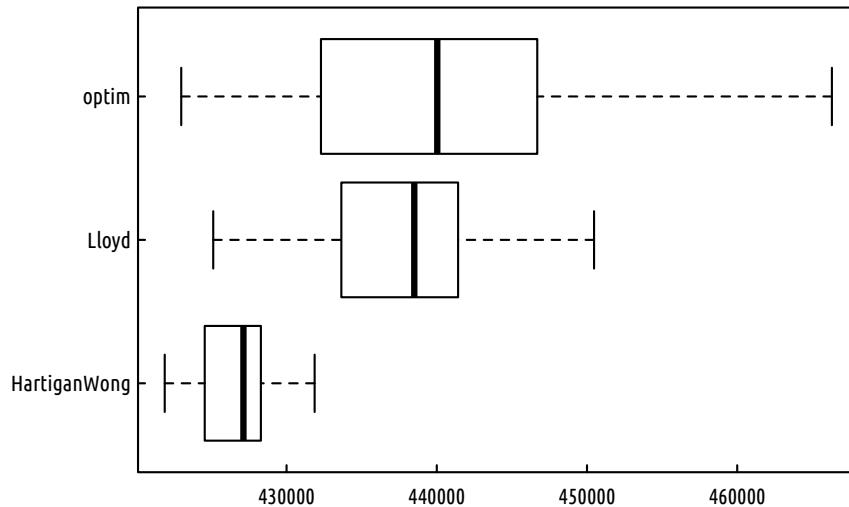


Figure 8.2: plot of chunk gendata5

```

print(apply(res, 1, function(x)
  c(summary(x), sd=sd(x)))
))

##          HartiganWong      Lloyd      optim
## Min.    421889.463 425119.482 422989.2
## 1st Qu. 424662.768 433669.308 432445.6
## Median  427128.673 438502.186 440032.9
## Mean    426557.050 438074.991 440635.3
## 3rd Qu. 428242.881 441381.268 446614.2
## Max.    431868.537 450469.678 466302.5
## sd       2300.955   5709.282 10888.4

```

Of course, we are interested in the smallest value of the objective, because we're trying to pinpoint the global minimum.

```

print(apply(res, 1, min))

## HartiganWong      Lloyd      optim
##        421889.5     425119.5     422989.2

```

The Hartigan-Wong algorithm (the default one in `kmeans()`) is the most reliable one of the three:

- it gives the best solution (low bias)
- the solutions have the lowest degree of variability (low variance)
- it is the fastest:

```

library("microbenchmark")
set.seed(123)
mu0 <- X[sample(nrow(X), K),]
summary(microbenchmark(
  HartiganWong=min_HartiganWong(mu0, X),
  Lloyd=min_Lloyd(mu0, X),
  optim=min_optim(mu0, X),
  times=10
), unit="relative")

##           expr      min       lq     mean      median
## 1 HartiganWong 1.121572 1.137392 1.175882 1.206071
## 2      Lloyd    1.000000 1.000000 1.000000 1.000000
## 3      optim 1638.117974 1670.386988 1670.383165 1706.594265
##           uq      max neval
## 1    1.262584 1.167868    10
## 2    1.000000 1.000000    10
## 3 1730.880290 1634.661002    10

```

```
print(min(res))
```

```
## [1] 421889.5
```

Is it the global minimum?

We don't know, we just didn't happen to find anything better (yet).

Did we put enough effort to find it?

Well, maybe. We can try more random restarts:

```
res_tried_very_hard <- kmeans(X, K, nstart=100000, iter.max=10000)$centers
print(get_fitness(res_tried_very_hard, X))
```

```
## [1] 421889.5
```

Is it good enough?

It depends what we'd like to do with this. Does it make your boss happy? Does it generate revenue? Does it help solve any other problem? Is it useful anyhow? Are you really looking for the global minimum?

## 8.2 A Note on Convex Optimisation (\*)

### 8.2.1 Introduction

Are there cases where we are sure that a local minimum is the global minimum?

Yes. For example when we minimise *convex objective functions*.

Here is just a very brief overview of the topic for the interested, see also (Boyd & Vandenberghe 2004) for more.

For example, the linear regression or the logistic regression have convex objectives – they are very well-behaving.

Note that this doesn't mean that we know an **analytic solution**.

### 8.2.2 Convex Combinations (\*)

A **convex combination** of a set of points  $x_1, \dots, x_n \in D$  is a *linear combination*

$$\theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

for some  $\theta_1, \theta_2, \dots, \theta_n$  that fulfils  $\theta_1, \theta_2, \dots, \theta_n \geq 0$  and  $\theta_1 + \theta_2 + \cdots + \theta_n = 1$ .

Think of this as a weighted arithmetic mean of these points.

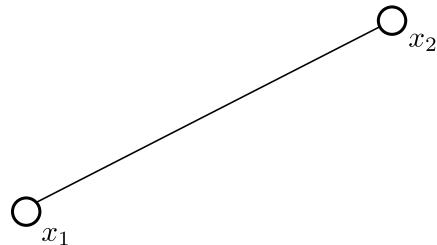


Figure 8.3:

The set of all convex combinations of two points  $x_1, x_2 \in D$ :

$$\{\theta x_1 + (1 - \theta)x_2 : \theta \in [0, 1]\}$$

is just the line segment between these two points.

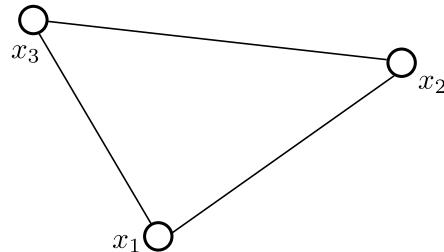


Figure 8.4:

The set of all convex combinations of 3 points yields a triangle (unless  $D$  is one-dimensional).

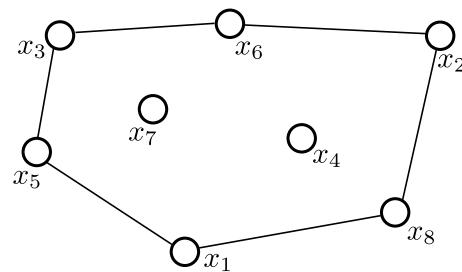


Figure 8.5:

More generally, we get the *convex hull* of a set of points – the smallest set  $C$  enclosing all the points that is convex, i.e., a line segment between any two points in  $C$  is fully included in  $C$ .

In two dimensions, think of a rubber band stretched around all the points like a fence.

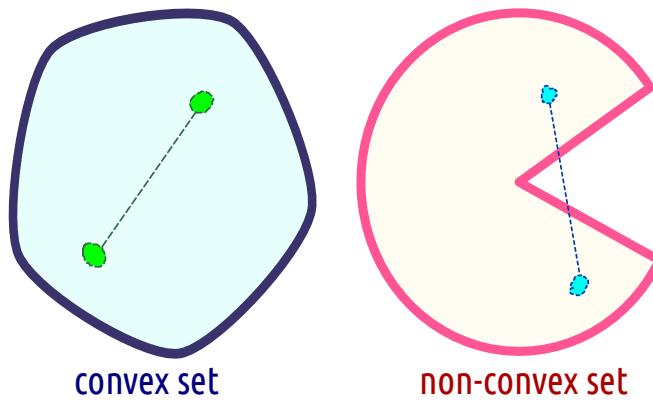


Figure 8.6:

### 8.2.3 Convex Functions (\*)

We call a function  $f : D \rightarrow \mathbb{R}$  **convex**, whenever:

$$(\forall x_1, x_2 \in D)(\forall \theta \in [0, 1]) \quad f(\theta x_1 + (1 - \theta)x_2) \leq \theta f(x_1) + (1 - \theta)f(x_2)$$

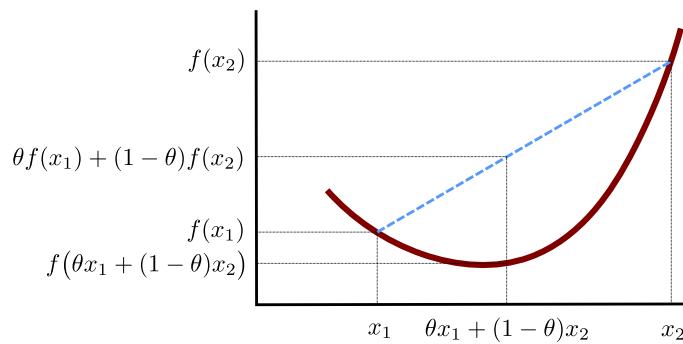


Figure 8.7:

(function value at any convex combination of two points is not greater than that combination of the function values at these two points)

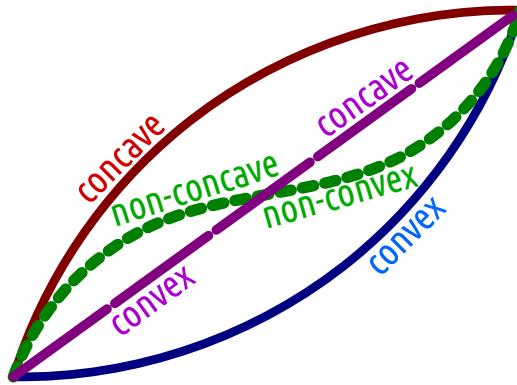


Figure 8.8:

$$\text{Convex: } f(\theta x_1 + (1 - \theta)x_2) \leq \theta f(x_1) + (1 - \theta)f(x_2)$$

$$\text{Concave: } f(\theta x_1 + (1 - \theta)x_2) \geq \theta f(x_1) + (1 - \theta)f(x_2)$$

**Theorem** For any convex function  $f$ , if  $f$  has a local minimum at  $x$  then  $x$  is also its global minimum.

Optimising convex functions is *relatively* easy, especially if they are differentiable.

Methods such as gradient descent or BFGS should work very well (unless there are large regions where a function is constant...).

(\*\*) There is a special class of constrained optimisation problems called linear and quadratic programming that involves convex functions, see (Nocedal & Wright 2006, Fletcher 2008).

(\*\*\*) See also the Karush–Kuhn–Tucker (KKT) conditions for a more general problem of minimisation with constraints.

#### 8.2.4 Examples

- $x^2, |x|, e^x$  are all convex.
- $|x|^p$  is convex for all  $p \geq 1$ .
- if  $f$  is convex, then  $-f$  is concave.
- if  $f_1$  and  $f_2$  are convex, then  $w_1 f_1 + w_2 f_2$  are convex for any  $w_1, w_2 \geq 0$ .

- if  $f_1$  and  $f_2$  are convex, then  $\max\{f_1, f_2\}$  is convex.
- if  $f$  and  $g$  are convex and  $g$  is non-decreasing, then  $g(f(x))$  is convex.
- Sum of squared residuals in linear regression is a convex function of the underlying parameters.
- Cross-entropy in logistic regression is a convex function of the underlying parameters.

## 8.3 Genetic Algorithms

### 8.3.1 Introduction

What if our optimisation problem cannot be solved reliably with gradient-based methods like those in `optim()` and we don't have any custom solver for the task at hand?

There are a couple of useful *metaheuristics* in the literature that can serve this purpose.

Most of them rely on clever randomised search.

They are slow to run and don't guarantee anything, but yet they still might be useful – better a solution than no solution at all.

There is a wide class of **nature-inspired** algorithms (that traditionally belong to the subfield of AI called *computational intelligence* or *soft computing*); see, e.g, (Simon 2013):

- evolutionary algorithms – inspired by the principle of natural selection  
maintain a population of candidate solutions, let the “fittest” combine with each other to generate new “offspring” solutions.
- swarm algorithms  
maintain a herd of candidate solutions, allow them to “explore” the environment, “communicate” with each other in order to seek the best spot to “go to”.

For example:

- ant colony
- bees
- cuckoo search
- particle sward
- krill herd
- other metaheuristics:
  - harmony search

- memetic algorithm
- firefly algorithm

All of these sound fancy, but the general ideas behind them are pretty simple.

### 8.3.2 Overview of the Method

Genetic algorithms (GAs) are amongst the most popular evolutionary approaches. They are based on Charles Darwin's work on evolution by natural selection. See (Goldberg, 1989) for a comprehensive overview and (Simon, 2013) for extensions.

Here is the general idea of a GA (there might be many) to minimise a given objective/fitness function  $f$  over a given domain  $D$ .

1. Generate a random initial population of individuals –  $n_{\text{pop}}$  points in  $D$ , e.g.,  $n_{\text{pop}} = 128$
2. Repeat until some convergence criterion is not met:
  - a. evaluate the fitness of each individual
  - b. select the pairs of the individuals for reproduction, the fitter should be selected more eagerly
  - c. apply crossover operations to create offspring
  - d. slightly mutate randomly selected individuals
  - e. replace the old population with the new one

### 8.3.3 Example Implementation - GA for K-means

Initial setup:

```
set.seed(123)

# simulation parameters:
npop <- 32
niter <- 100

# randomly generate an initial population of size `npop`:
pop <- lapply(1:npop, function(i) X[sample(nrow(X), K),])

# evaluate fitness of each individual:
cur_fitness <- sapply(pop, get_fitness, X)
cur_best_fitness <- min(cur_fitness)
best_fitness <- cur_best_fitness
```

Each individual in the population is just the set of  $K$  candidate cluster centres represented as a matrix in  $\mathbb{R}^{K \times p}$ .

Let's assume that the fitness of each individual should be a function of the rank of the objective function's value (smallest objective == highest rank == best fit).

For the crossover, we will sample pairs of individuals with probabilities proportional to their fitness.

```
selection <- function(cur_fitness) {
  npop <- length(cur_fitness)
  probs <- rank(-cur_fitness)
  probs <- probs/sum(probs)
  left <- sample(npop, npop, replace=TRUE, prob=probs)
  right <- sample(npop, npop, replace=TRUE, prob=probs)
  cbind(left, right)
}
```

An example crossover combines each cluster centre in such a way that we take a few coordinates of the “left” parent and the remaining ones from the “right” parent (see below for an illustration):

```
crossover <- function(pop, pairs, K, p) {
  old_pop <- pop
  pop <- pop[pairs[,2]]
  for (j in 1:length(pop)) {
    wh <- sample(p-1, K, replace=TRUE)
    for (l in 1:K)
      pop[[j]][l,1:wh[l]] <-
        old_pop[[pairs[j,1]]][l,1:wh[l]]
  }
  pop
}
```

Mutation (occurring with a very small probability) substitutes some cluster centre with a random vector from the input dataset.

```
mutate <- function(pop, X, K) {
  for (j in 1:length(pop)) {
    if (runif(1) < 0.025) {
      szw <- sample(1:K, 1)
      pop[[j]][szw,] <- X[sample(nrow(X), length(szw)),]
    }
  }
}
```

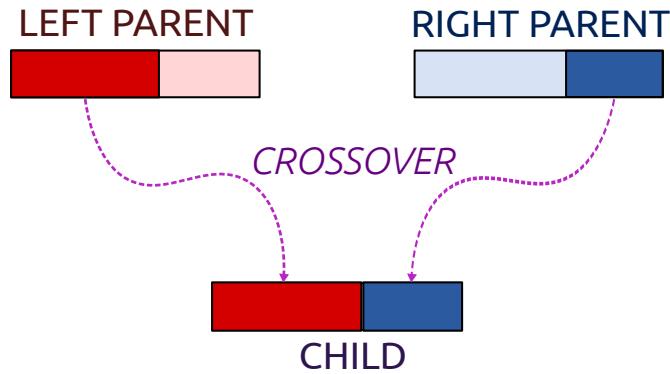


Figure 8.9:

```
    pop
}
```

We also need a function that checks if the new cluster centres aren't too far away from the input points.

If it happens that we have empty clusters, our solution is degenerate and we must correct it.

All “bad” cluster centres will be substituted with randomly chosen points from  $\mathbf{X}$ .

Moreover, we will recompute the cluster centres as the componentwise arithmetic mean of the closest points, just like in Lloyd's algorithm, to speed up convergence.

```
recompute_mus <- function(pop, X, K) {
  for (j in 1:length(pop)) {
    # get nearest cluster centres for each point:
    memb <- get.knnx(pop[[j]], X, 1)$nn.index
    sz <- tabulate(memb, K) # number of points in each cluster
    # if there are empty clusters, fix them:
    szw <- which(sz==0)
    if (length(szw)>0) { # random points in X will be new cluster centres
      pop[[j]][szw,] <- X[sample(nrow(X), length(szw)),]
      memb <- FNN::get.knnx(pop[[j]], X, 1)$nn.index
      sz <- tabulate(memb, K)
    }
    # recompute cluster centres - componentwise average:
    pop[[j]][,] <- 0
  }
}
```

```

for (l in 1:nrow(X))
  pop[[j]][memb[l],] <- pop[[j]][memb[l],] + X[l,]
  pop[[j]] <- pop[[j]]/sz
}
pop
}

```

We are ready to build our genetic algorithm to solve the K-means clustering problem:

```

for (i in 1:niter) {
  pairs <- selection(cur_fitness)
  pop <- crossover(pop, pairs, K, p)
  pop <- mutate(pop, X, K)
  pop <- recompute_mu(pop, X, K)
  # re-evaluate fitness:
  cur_fitness <- sapply(pop, get_fitness, X)
  cur_best_fitness <- min(cur_fitness)
  # give feedback on what's going on:
  if (cur_best_fitness < best_fitness) {
    best_fitness <- cur_best_fitness
    best_mu <- pop[[which.min(cur_fitness)]]
    cat(sprintf("%5d: f_best=%10.5f\n", i, best_fitness))
  }
}

```

```

##      1: f_best=435638.52165
##      2: f_best=428808.89706
##      4: f_best=428438.45125
##      6: f_best=422277.99136
##      8: f_best=421889.46265
print(get_fitness(best_mu, X))

```

```

## [1] 421889.5
print(get_fitness(res_tried_very_hard, X))

```

```

## [1] 421889.5

```

It works! :)

## 8.4 Outro

### 8.4.1 Remarks

For any  $p \geq 1$ , the search space type determines the problem class:

- $D \subseteq \mathbb{R}^p$  – **continuous optimisation**

In particular:

- $D = \mathbb{R}^p$  – continuous unconstrained
- $D = [a_1, b_1] \times \cdots \times [a_n, b_n]$  – continuous with box constraints
- constrained with  $k$  linear inequality constraints

$$a_{1,1}x_1 + \cdots + a_{1,p}x_p \leq b_1, \dots, a_{k,1}x_1 + \cdots + a_{k,p}x_p \leq b_k$$

However, there are other possibilities as well:

- $D \subseteq \mathbb{Z}^p$  ( $\mathbb{Z}$  – the set of integers) – **discrete optimisation**

In particular:

- $D = \{0, 1\}^p$  – 0–1 optimisation (hard!)
- $D$  is finite (but perhaps large, its objects can be enumerated) – **combination optimisation**

For example:

- $D$  = all possible routes between two points on a map.

These optimisation tasks tend to be much harder than the continuous ones.

Genetic algorithms might come in handy in such cases.

Specialised methods, customised to solve a specific problem (like Lloyd's algorithm) will often outperform generic ones (like SGD, genetic algorithms) in terms of speed and reliability.

All in all, we prefer a suboptimal solution obtained by means of heuristics to no solution at all.

Problems that you could try solving with GAs include variable selection in multiple regression – finding the subset of features optimising the AIC (this is a hard problem to and forward selection was just a simple greed heuristic).

Other interesting algorithms:

- Hill Climbing (a simple variation of GD with no gradient use)

- Simulated annealing
- CMA-ES
- Tabu search
- Particle swarm optimisation
- Artificial bee/ant colony optimisation
- Cuckoo Search

#### 8.4.2 Further Reading

Recommended further reading:

- (Goldberg 1989)

Other:

- (Simon 2013)
- (Boyd & Vandenberghe 2004)



# Chapter 9

# Recommender Systems

## 9.1 Introduction

### 9.1.1 What is a Recommender System?

A **recommender (recommendation) system** is a method to predict the rating a **user** would give to an **item**.

For example:

- playlist generators at Spotify, YouTube or Netflix,
- content recommendations on Facebook, Instagram, Twitter or Apple News,
- product recommendations at Amazon or Alibaba.

(Ricci et al. 2011) list the following functions of recommender systems:

- increase the number of items sold,
- sell more diverse items,
- increase users' satisfaction,
- increase users' fidelity,
- better understand what users want.

### 9.1.2 The Netflix Prize

In 2006 Netflix (back then a DVD rental company) released one of the most famous benchmark sets for recommender systems, which helped boost the research on algorithms in this field.

See <https://www.kaggle.com/netflix-inc/netflix-prize-data>; data archived at <https://web.archive.org/web/20090925184737/http://archive.ics.uci.edu/ml/datasets/Netflix+Prize> and [https://archive.org/details/nf\\_prize\\_dataset.tar](https://archive.org/details/nf_prize_dataset.tar)

The dataset consists of:

- 480,189 users
- 17,770 movies
- 100,480,507 ratings in the training sample:
  - `MovieID`
  - `CustomerID`
  - `Rating` from 1 to 5
  - `Title`
  - `YearOfRelease` from 1890 to 2005
  - `Date` of rating in the range 1998-11-01 to 2005-12-31

The *quiz set* consists of 1,408,342 ratings and it was used by the competitors to assess the quality of their algorithms and compute leaderboard scores.

Root mean squared error (RMSE) of predicted vs. true rankings was chosen as a performance metric.

The *test set* of 1,408,789 ratings was used to determine the winner.

On 21 Sept. 2009, the grand prize of US\$1,000,000 was given to the BellKor's Pragmatic Chaos team which improved over the Netflix's *Cinematch* algorithm by 10.06%, achieving the winning RMSE of 0.8567 on the test subset.

### 9.1.3 Main Approaches

Current recommender systems are quite complex and use a fusion of various approaches, also those based on external knowledge bases.

However, we may distinguish at least two core approaches, see (Ricci et al. 2011) for more:

- *Collaborative Filtering*

It is based on the assumption that if two people interact with the same product, they're likely to have other interests in common as well.

John and Mary both like bananas and apples and dislike spinach.  
 John likes sushi. Mary hasn't tried sushi yet. It seems they might have similar tastes, so we recommend that Mary should give sushi a try.

- *Content-based Filtering*

It builds a user's profile that represent information of what kind of products does she/he like.

We have discovered that John likes fruits but dislikes vegetables. An orange is a fruit (an item similar to those he liked in the past) with which John is yet to interact. John should give it a try.

Jim Bennett, vice president of recommendations systems at Netflix on the idea behind the original Cinematch algorithm (see <https://www.technologyreview.com/s/406637/the-1-million-netflix-challenge/>):

First, you collect 100 million user ratings for about 18,000 movies. Take any two movies and find the people who have rated both of them. Then look to see if the people who rate one of the movies highly rate the other one highly, if they liked one and not the other, or if they didn't like either movie. Based on their ratings, Cinematch sees whether there's a correlation between those people. Now, do this for all possible pairs of 65,000 movies.

See also: <https://web.archive.org/web/20070821194257/http://www.netflixprize.com/faq>

Is it an example of collaborative or context-based filtering?

#### 9.1.4 Formalism

Let  $\mathcal{U} = \{U_1, \dots, U_m\}$  denote the set of  $m$  users.

Let  $\mathcal{I} = \{I_1, \dots, I_n\}$  denote the set of  $n$  items.

Let  $R \in \mathbb{R}^{m \times n}$  be a user-item matrix such that:

$$r_{u,i} = \begin{cases} r & \text{if the } u\text{-th user ranked the } i\text{-th item as } r > 0 \\ 0 & \text{if the } u\text{-th user hasn't interacted with the } i\text{-th item yet} \end{cases}$$

Note that here 0 is used to denote a missing value (NA)

In particular, we can assume:

- $r_{u,i} \in \{0, 1, \dots, 5\}$  (ratings on the scale 1–5 or no interaction)
- $r_{u,i} \in \{0, 1\}$  (“Like” or no interaction)

The aim of an recommender system is to predict the rating  $\hat{r}_{u,i}$  that the  $u$ -th user would give to the  $i$ -th item provided that currently  $r_{u,i} = 0$ .

## 9.2 Collaborative Filtering

### 9.2.1 Example

In **user-based collaborative filtering**, we seek users with similar preference profiles/rating patters.

“User A has similar behavioural patterns as user B, so we should suggest A what B likes.”

In **item-based collaborative filtering**, we seek items with similar (dis)likeability structure.

“Users who (dis)liked X also (dis)liked Y”.

.	Apple	Banana	Sushi	Spinach	Orange
Anne	1	5	5		1
Beth	1	1	5	5	1
John	5	5		1	
Kate	1	1	5	5	1
Mark	5	5	1	1	5
Sara	?	5		?	5

Will Sara enjoy her spinach? Will Sara enjoy her apple?

```
R <- matrix(
  c(
    1, 5, 5, 0, 1,
    1, 1, 5, 5, 1,
    5, 5, 0, 1, 0,
    1, 1, 5, 5, 1,
    5, 5, 1, 1, 5,
    0, 5, 0, 0, 5
  ), byrow=TRUE, nrow=6, ncol=5,
  dimnames=list(
    c("Anne", "Beth", "John", "Kate", "Mark", "Sara"),
    c("Apple", "Banana", "Sushi", "Spinach", "Orange")
  )
)
```

R

```
##      Apple Banana Sushi Spinach Orange
## Anne     1      5     5      0      1
## Beth     1      1     5      5      1
## John     5      5     0      1      0
## Kate     1      1     5      5      1
## Mark     5      5     1      1      5
## Sara     0      5     0      0      5
```

### 9.2.2 Similarity Measures

Assuming  $\mathbf{a}, \mathbf{b} \in \mathbb{N}^k$  (in our setting,  $k \in \{n, m\}$ ), let  $S$  be the following similarity measure between two vectors of identical lengths (representing ratings):

$$S(\mathbf{a}, \mathbf{b}) = \frac{\sum_{i=1}^k a_i b_i}{\sqrt{\sum_{i=1}^k a_i^2} \sqrt{\sum_{i=1}^k b_i^2}} \geq 0$$

```
cosim <- function(a, b) sum(a*b)/sqrt(sum(a^2)*sum(b^2))
```

We call it the **cosine similarity**.

(\*) Another frequently considered similarity measure is a version of the Pearson correlation coefficient that ignores all 0-valued observations, see also the `use` argument to the `cor()` function.

### 9.2.3 User-Based Collaborative Filtering

**User-based** approaches involve comparing each user against every other user (pairwise comparisons of the rows in  $R$ ). This yields a similarity degree between the  $i$ -th and the  $j$ -th user:

$$s_{i,j}^U = S(\mathbf{r}_{i,\cdot}, \mathbf{r}_{j,\cdot}).$$

```
SU <- matrix(0, nrow=nrow(R), ncol=nrow(R),
             dimnames=dimnames(R)[c(1,1)]) # and empty m*m matrix
for (i in 1:nrow(R)) {
  for (j in 1:nrow(R)) {
    SU[i,j] <- cosim(R[i,], R[j,])
  }
}
```

```
round(SU, 2)
```

```
##      Anne Beth John Kate Mark Sara
## Anne 1.00 0.61 0.58 0.61 0.63 0.59
## Beth 0.61 1.00 0.29 1.00 0.39 0.19
## John 0.58 0.29 1.00 0.29 0.81 0.50
## Kate 0.61 1.00 0.29 1.00 0.39 0.19
## Mark 0.63 0.39 0.81 0.39 1.00 0.81
## Sara 0.59 0.19 0.50 0.19 0.81 1.00
```

In order to obtain the previously unobserved rating  $\hat{r}_{u,i}$  using the user-based approach, we typically look for the  $K$  most similar users and aggregate their corresponding scores (for some  $K \geq 1$ ).

More formally, let  $\{U_{v_1}, \dots, U_{v_K}\} \in \mathcal{U} \setminus \{U_u\}$  be the set of users maximising  $s_{u,v_1}^U, \dots, s_{u,v_K}^U$  and having  $r_{v_1,i}, \dots, r_{v_K,i} > 0$ . Then

$$\hat{r}_{u,i} = \frac{1}{K} \sum_{\ell=1}^K r_{v_\ell,i}.$$

The arithmetic mean can be replaced with, e.g., the more or a weighted arithmetic mean where weights are proportional to  $s_{u,v_\ell}^U$ .

This is similar to the  $K$ -nearest neighbour heuristic.

```
K <- 2
(sim <- order(SU["Sara"], ,decreasing=TRUE))

## [1] 6 5 1 3 2 4
# sim gives the indices of people in decreasing order
# of similarity to Sara:
dimnames(R)[[1]][sim] # the corresponding names

## [1] "Sara" "Mark" "Anne" "John" "Beth" "Kate"
# Remove those who haven't tried Spinach yet (including Sara):
sim <- sim[ R[sim, "Spinach"]>0 ]
dimnames(R)[[1]][sim]

## [1] "Mark" "John" "Beth" "Kate"
# aggregate the Spinach ratings of the K most similar people:
mean(R[sim[1:K], "Spinach"])

## [1] 1
```

### 9.2.4 Item-Based Collaborative Filtering

**Item-based** schemes rely on pairwise comparisons between the items (columns in  $R$ ). Hence, a similarity degree between the  $i$ -th and the  $j$ -th item is given by:

$$s_{i,j}^I = S(\mathbf{r}_{\cdot,i}, \mathbf{r}_{\cdot,j}).$$

```
SI <- matrix(0, nrow=ncol(R), ncol=ncol(R),
             dimnames=dimnames(R)[c(2,2)]) # an empty n*n matrix
for (i in 1:ncol(R)) {
```

```

    for (j in 1:ncol(R)) {
      SI[i,j] <- cosim(R[,i], R[,j])
    }
  }

round(SI, 2)

##          Apple Banana Sushi Spinach Orange
## Apple     1.00   0.78  0.32    0.38   0.53
## Banana    0.78   1.00   0.45    0.27   0.78
## Sushi     0.32   0.45   1.00    0.81   0.32
## Spinach   0.38   0.27   0.81    1.00   0.29
## Orange    0.53   0.78   0.32    0.29   1.00

```

In order to obtain the previously unobserved rating  $\hat{r}_{u,i}$  using the item-based approach, we typically look for the  $K$  most similar items and aggregate their corresponding scores (for some  $K \geq 1$ )

More formally, let  $\{I_{j_1}, \dots, I_{j_K}\} \in \mathcal{I} \setminus \{I_i\}$  be the set of items maximising  $s_{i,j_1}^I, \dots, s_{i,j_K}^I$  and having  $r_{u,j_1}, \dots, r_{u,j_K} > 0$ . Then

$$\hat{r}_{u,i} = \frac{1}{K} \sum_{\ell=1}^K r_{u,j_\ell}.$$

The arithmetic mean can be replaced with, e.g., a weighted arithmetic mean where weights are proportional to  $s_{i,j_\ell}^I$  or the mode.

```

K <- 2
(sim <- order(SI["Apple",], decreasing=TRUE))

## [1] 1 2 5 4 3
# sim gives the indices of items in decreasing order
# of similarity to Apple:
dimnames(R)[[2]][sim] # the corresponding item types

## [1] "Apple"    "Banana"   "Orange"   "Spinach"  "Sushi"
# Remove these which Sara haven't tried yet (e.g., Apples):
sim <- sim[R["Sara", sim]>0]
dimnames(R)[[2]][sim]

## [1] "Banana"   "Orange"
# aggregate Sara's ratings of the K most similar items:
mean(R["Sara", sim[1:K]])

## [1] 5

```

## 9.3 MovieLens Dataset (\*)

### 9.3.1 Dataset

Let us make a few recommendations based on the MovieLens-9/2018-Small dataset available at <https://grouplens.org/datasets/movielens/latest/>

The dataset consists of ca. 100,000 ratings to 9,000 movies by 600 users. Last updated 9/2018.

This is already a pretty large dataset! We might run into problems with memory usage and run-time.

The following examples are a bit more difficult to follow (programming-wise), therefore we mark them with (\*).

See also <https://movielens.org/> and (Harper & Konstan 2015).

```
options(stringsAsFactors=FALSE)
movies <- read.csv("datasets/ml-9-2018-small/movies.csv")
head(movies, 4)

##   movieId          title
## 1       1    Toy Story (1995)
## 2       2        Jumanji (1995)
## 3       3 Grumpier Old Men (1995)
## 4       4 Waiting to Exhale (1995)
##
##           genres
## 1 Adventure|Animation|Children|Comedy|Fantasy
## 2             Adventure|Children|Fantasy
## 3                   Comedy|Romance
## 4             Comedy|Drama|Romance

nrow(movies)

## [1] 9742

ratings <- read.csv("datasets/ml-9-2018-small/ratings.csv")
head(ratings, 4)

##   userId movieId rating timestamp
## 1      1       1     4 964982703
## 2      1       3     4 964981247
## 3      1       6     4 964982224
## 4      1      47     5 964983815

nrow(ratings)
```

```
## [1] 100836





```

### 9.3.2 Data Cleansing

`movieIds` should be re-encoded, as not every film is mentioned/rated in the database. We will re-map the `movieIds` to consecutive integers.

```
movieId2 <- unique(ratings$movieId) # the list of all rated movieIds
(m <- max(ratings$userId)) # max user Id (these could've been cleaned up too)

## [1] 610
(n <- length(movieId2)) # number of unique movies

## [1] 9724

movies <- movies[movies$movieId %in% movieId2, ] # remove unrated movies
# we shall map movieId2[i] to i for each i=1,...,n
movies$movieId <- match(movies$movieId, movieId2)
ratings$movieId <- match(ratings$movieId, movieId2)
# order the movies by the new movieId so that
# the movie with Id=i is at the i-th row.
movies <- movies[order(movies$movieId),]
stopifnot(all(movies$movieId == 1:n)) # sanity check
```

We will use a sparse matrix data type (from R package `Matrix`) to store ratings data. We don't want to run out of memory!

Sparse == many zeros.

```
library("Matrix")
RML <- Matrix(0.0, sparse=TRUE, nrow=m, ncol=n)
# This is a vectorised operation;
# it is faster than a for loop over each row in ratings:
RML[cbind(ratings$userId, ratings$movieId)] <- ratings$rating

# Preview:
RML[1:6, 1:18]

## 6 x 18 sparse Matrix of class "dgCMatrix"
##
## [1,] 4 4 4 5 5 3 5 4 5 5 5 5 3 5 4 5 3 3
```

```
## [2,] . . . . . . . . . . . .
## [3,] . . . . . . . . . . . .
## [4,] . . . 2 . . . . . . . 2 5 1 .
## [5,] 4 . . . 4 . . 4 . . . . . . 5 2
## [6,] . 5 4 4 1 . 5 4 . 3 4 . 3 . . 2 5
```

### 9.3.3 Item-Item Similarities

To recall, the cosine similarity between  $\mathbf{a}, \mathbf{b} \in \mathbb{R}^m$  is given by:

$$S_C(\mathbf{a}, \mathbf{b}) = \frac{\sum_{i=1}^m a_i b_i}{\sqrt{\sum_{i=1}^m a_i^2} \sqrt{\sum_{i=1}^m b_i^2}}$$

In vector/matrix algebra notation (have you noticed this section is marked with (\*)?), this is:

$$S_C(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}^T \mathbf{b}}{\sqrt{\mathbf{a}^T \mathbf{a}} \sqrt{\mathbf{b}^T \mathbf{b}}}$$

If  $\mathbf{A} \in \mathbb{R}^{m \times n}$  we can “almost” compute the all the  $n$  cosine similarities at once by applying:

$$S_C(\mathbf{a}, \mathbf{B}) = \frac{\mathbf{A}^T \mathbf{A}}{\dots}$$

Cosine item-item similarities:

```
norms <- as.matrix(sqrt(colSums(RML^2)))
RMLx <- as.matrix(crossprod(RML, RML))
SI <- RMLx/tcrossprod(norms)
SI[is.nan(SI)] <- 0 # there were some divisions by zero
```

`crossprod(A,B)` gives  $\mathbf{A}^T \mathbf{B}$

`tcrossprod(A,B)` gives  $\mathbf{A}\mathbf{B}^T$

### 9.3.4 Example Recommendations

```
recommend <- function(i, K, SI, movies) {
  # get K most similar movies to the i-th one
  ms <- order(SI[i,], decreasing=TRUE)
  tibble::tibble(
    Title=movies$title[ms[1:K]],
```

```

SIC=SI[i,ms[1:K]])
}

recommend(1215, 10, SI, movies)

## # A tibble: 10 x 2
##   Title           SIC
##   <chr>          <dbl>
## 1 Monty Python's The Meaning of Life (1983)    1
## 2 Monty Python's Life of Brian (1979)        0.611
## 3 Monty Python and the Holy Grail (1975)      0.514
## 4 House of Flying Daggers (Shi mian mai fu) (2004) 0.493
## 5 Hitchhiker's Guide to the Galaxy, The (2005)    0.455
## 6 Bowling for Columbine (2002)                  0.451
## 7 Shaun of the Dead (2004)                      0.446
## 8 O Brother, Where Art Thou? (2000)            0.445
## 9 Ghost World (2001)                          0.444
## 10 Full Metal Jacket (1987)                 0.443

```

```

recommend(1, 10, SI, movies)

## # A tibble: 10 x 2
##   Title           SIC
##   <chr>          <dbl>
## 1 Toy Story (1995)    1
## 2 Toy Story 2 (1999)  0.573
## 3 Jurassic Park (1993) 0.566
## 4 Independence Day (a.k.a. ID4) (1996) 0.564
## 5 Star Wars: Episode IV - A New Hope (1977) 0.557
## 6 Forrest Gump (1994)    0.547
## 7 Lion King, The (1994)  0.541
## 8 Star Wars: Episode VI - Return of the Jedi (1983) 0.541
## 9 Mission: Impossible (1996) 0.539
## 10 Groundhog Day (1993)   0.534

```

... and so on.

### 9.3.5 Clustering

A cosine similarity matrix can be turned into a dissimilarity matrix:

```

DI <- 1.0-SI
DI[DI<0] <- 0.0 # account for numeric inaccuracies
DI <- as.dist(DI)

```

Which enables us to perform, e.g., the cluster analysis of items:

```
library("genie")
h <- hclust2(DI)
c <- cutree(h, k=20)
```

Example movies in the 3rd cluster:

```
library("stringi")
cat(i, stri_wrap(stri_paste(head(movies$title[c==3], 20),
collapse=" ", ")), sep="\n")

## 5
## Bottle Rocket (1996), Clerks (1994), Star Wars: Episode
## IV - A New Hope (1977), Swingers (1996), Monty Python's
## Life of Brian (1979), E.T. the Extra-Terrestrial (1982),
## Monty Python and the Holy Grail (1975), Star Wars:
## Episode V - The Empire Strikes Back (1980), Princess
## Bride, The (1987), Raiders of the Lost Ark (Indiana
## Jones and the Raiders of the Lost Ark) (1981), Star Wars:
## Episode VI - Return of the Jedi (1983), Blues Brothers,
## The (1980), Duck Soup (1933), Groundhog Day (1993), Back
## to the Future (1985), Young Frankenstein (1974), Indiana
## Jones and the Last Crusade (1989), Grosse Pointe Blank
## (1997), Austin Powers: International Man of Mystery
## (1997), Men in Black (a.k.a. MIB) (1997)
```

Example movies in the 5th cluster:

```
cat(i, stri_wrap(stri_paste(head(movies$title[c==5], 20),
collapse=" ", ")), sep="\n")

## 5
## Blown Away (1994), Flight of the Navigator (1986), Dick
## Tracy (1990), Mighty Aphrodite (1995), Postman, The
## (Postino, Il) (1994), Flirting With Disaster (1996),
## Living in Oblivion (1995), Safe (1995), Eat Drink Man
## Woman (Yin shi nan nu) (1994), Bullets Over Broadway
## (1994), Barcelona (1994), In the Name of the Father
## (1993), Six Degrees of Separation (1993), Maya Lin: A
## Strong Clear Vision (1994), Everyone Says I Love You
## (1996), Rebel Without a Cause (1955), Wings of Desire
## (Himmel über Berlin, Der) (1987), High Noon (1952),
## Afterglow (1997), Bulworth (1998)
```

## 9.4 Outro

### 9.4.1 Remarks

Good recommender systems are perfect tools to increase the revenue of any user-centric enterprise.

Not a single algorithm, but an ensemble (a proper combination) of different approaches is often used in practice, see the Further Reading section below for the detailed information of the Netflix Prize winners.

Recommender systems are an interesting fusion of the techniques we have already studied – linear models, K-nearest neighbours etc.

### 9.4.2 Issues

Building recommender systems is challenging, because data is large yet often sparse;

Here is the ratio of available ratings vs. all possible user-item valuations for the Netflix Prize (obviously, it is just a sample of the complete dataset that Netflix has):

```
100480507 / (480189 * 17770)
```

```
## [1] 0.01177558
```

*Sparse matrix* (many “zeros” = unassigned ratings) data structure is often used for storing of and computing over such data effectively.

Some users are *biased* in the sense that they are more critical or enthusiastic than average users.

Is 3 stars a “bad”, “fair enough” or “good” rating for you? Would you go to a bar/restaurant ranked 3.0 by your favourite Maps app community?

It is particularly challenging to predict the preferences of users that cast few ratings, e.g., those who just signed up (*the cold start problem*).

“Hill et al. [1995] have shown that users provide inconsistent ratings when asked to rate the same movie at different times. They suggest that an algorithm cannot be more accurate than the variance in a user’s ratings for the same item.” (Herlocker et al. 2004: p. 6)

It is good to take into account the temporal (time-based) characteristics of data as well as external knowledge (e.g., how long ago a rating was cast, what is a film’s genre).

The presented approaches are vulnerable to attacks – bots may be used to promote or inhibit selected items.

### 9.4.3 Further Reading

Recommended further reading:

- (Herlocker et al. 2004)
- (Ricci et al. 2011)
- (Lü & others 2012)
- (Harper & Konstan 2015)

Other:

- (Koren 2009)
- (Töscher et al. 2009)
- (Piotte & Chabbert 2009)

Also don't forget to take a look at the R package `recommenderlab` (amongst others).

}

## Abbreviations

a.k.a. == also known as

w.r.t. == with respect to

s.t. == such that

iff == if and only if

e.g. == for example (Latin: *exempli gratia*)

i.e. == that is (Latin: *id est*)

etc. == and so forth (Latin: *et cetera*)

AI == artificial intelligence

GA == genetic algorithm

GD == gradient descent

GLM == generalised linear model

ML == machine learning

NN == neural network

SGD == stochastic gradient descent

IDE = integrated development environment

## Notation Convention – Logic and Set Theory

$\forall$  – for all

$\exists$  – exists

By writing  $x \in \{a, b, c\}$  we mean that “ $x$  is in a set that consists of  $a$ ,  $b$  and  $c$ ” or “ $x$  is either  $a$ ,  $b$  or  $c$ ”

$A \subseteq B$  – set  $A$  is a subset of set  $B$  (every element in  $A$  belongs to  $B$ ,  $x \in A$  implies that  $x \in B$ )

$A \cup B$  – union (sum) of two sets,  $x \in A \cup B$  iff  $x \in A$  or  $x \in B$

$A \cap B$  – intersection (sum) of two sets,  $x \in A \cap B$  iff  $x \in A$  and  $x \in B$

$A \setminus B$  – difference of two sets,  $x \in A \setminus B$  iff  $x \in A$  and  $x \notin B$

$A \times B$  – Cartesian product of two sets,  $A \times B = \{(a, b) : a \in A, b \in B\}$

$A^p = A \times A \times \dots \times A$  ( $p$  times) for any  $p$

### Notation Convention – Symbols

$\mathbf{X}, \mathbf{Y}, \mathbf{A}, \mathbf{I}, \mathbf{C}$  – bold (I use it for denoting vectors and matrices)

$\mathbb{X}, \mathbb{Y}, \mathbb{A}, \mathbb{I}, \mathbb{C}$  – blackboard bold (I sometimes use it for sets)

$\mathcal{X}, \mathcal{Y}, \mathcal{A}, \mathcal{I}, \mathcal{C}$  – calligraphic (I use it for set families = sets of sets)

$X, x, \mathbf{X}, \mathbf{x}$  – inputs (usually)

$Y, y, \mathbf{Y}, \mathbf{y}$  – outputs

$\hat{Y}, \hat{y}, \hat{\mathbf{Y}}, \hat{\mathbf{y}}$  – predicted outputs (usually)

- $X$  – independent/explanatory/predictor variable
- $Y$  – dependent/response/predicted variable

$\mathbb{R}$  – the set of real numbers,  $\mathbb{R} = (-\infty, \infty)$

$\mathbb{N}$  – the set of natural numbers,  $\mathbb{N} = \{1, 2, 3, \dots\}$

$\mathbb{N}_0$  – the set of natural numbers including zero,  $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$

$\mathbb{Z}$  – the set of integer numbers,  $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$

$[0, 1]$  – the unit interval

$(a, b)$  – an open interval;  $x \in (a, b)$  iff  $a < x < b$  for some  $a < b$

$[a, b]$  – a closed interval;  $x \in [a, b]$  iff  $a \leq x \leq b$  for some  $a \leq b$

### Notation Convention – Vectors and Matrices

$\mathbf{x} = (x_1, \dots, x_n)$  – a sequence of  $n$  elements ( $n$ -ary sequence/vector)

if it consists of real numbers, we write  $\mathbf{x} \in \mathbb{R}^n$

$\mathbf{x} = [x_1 \ x_2 \ \dots \ x_p]$  – a row vector,  $\mathbf{x} \in \mathbb{R}^{1 \times p}$  (a matrix with 1 row)

$\mathbf{x} = [x_1 \ x_2 \ \dots \ x_n]^T$  – a column vector,  $\mathbf{x} \in \mathbb{R}^{n \times 1}$  (a matrix with 1 column)

$\mathbf{X} \in \mathbb{R}^{n \times p}$  – matrix with  $n$  rows and  $p$  columns

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix}$$

$x_{i,j}$  – element in the  $i$ -th row,  $j$ -th column

$\mathbf{x}_{i,\cdot}$  – the  $i$ -th row of  $\mathbf{X}$

$\mathbf{x}_{\cdot,j}$  – the  $j$ -th column of  $\mathbf{X}$

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_{1,\cdot} \\ \mathbf{x}_{2,\cdot} \\ \vdots \\ \mathbf{x}_{n,\cdot} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{\cdot,1} & \mathbf{x}_{\cdot,2} & \cdots & \mathbf{x}_{\cdot,p} \end{bmatrix}.$$

$$\mathbf{x}_{i,\cdot} = \begin{bmatrix} x_{i,1} & x_{i,2} & \cdots & x_{i,p} \end{bmatrix}.$$

$$\mathbf{x}_{\cdot,j} = \begin{bmatrix} x_{1,j} & x_{2,j} & \cdots & x_{n,j} \end{bmatrix}^T = \begin{bmatrix} x_{1,j} \\ x_{2,j} \\ \vdots \\ x_{n,j} \end{bmatrix},$$

$T$  denotes the matrix transpose;  $\mathbf{B} = \mathbf{A}^T$  is a matrix such that  $b_{i,j} = a_{j,i}$ .

$\|\mathbf{x}\| = \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$  – the Euclidean norm

### Notation Convention – Functions

$f : X \rightarrow Y$  means that  $f$  is a function mapping inputs from set  $X$  (the domain of  $f$ ) to elements of  $Y$  (the codomain)

$x \mapsto x^2$  denotes a (inline) function mapping  $x$  to  $x^2$ , equivalent to `function(x) x^2` in R

$\exp x = e^x$  – exponential function with base  $e \approx 2.718$

$\log x$  – natural logarithm (base  $e$ )

it holds  $e^x = y$  iff  $\log y = x$

$\log ab = \log a + \log b$

$\log a^c = c \log a$

$\log a/b = \log a - \log b$

$\log 1 = 0$

$$\log e = 1$$

hence  $\log e^x = x$

### Notation Convention – Sums and Products

$$\sum_{i=1}^n x_i = x_1 + x_2 + \cdots + x_n$$

$\sum_{i=1,\dots,n} x_i$  – the same

$\sum_{i \in \{1,\dots,n\}} x_i$  – the same

note display (stand-alone)  $\sum_{i=1}^n x_i$  vs text (in-line)  $\sum_{i=1}^n x_i$  style

$$\prod_{i=1}^n x_i = x_1 x_2 \dots x_n$$

## Appendix A

# Setting Up the R Environment

### A.1 Installing R

R and Python are *the* languages of modern data science. The former is slightly more oriented towards data modelling, analysis and visualisation as well as statistical computing. It has a gentle learning curve, which makes it very suitable even for beginners – just like us!

R is available for Windows as well as MacOS, Linux and other Unix-like operating systems. It can be downloaded from the R project website, see <https://www.r-project.org/> (or installed through system-specific package repositories).

**Remark.** From now on we assume that you have installed the R environment.

### A.2 Installing an IDE

As we wish to make our first steps with the R language as stress- and hassle-free as possible, let's stick to a very user-friendly development environment called RStudio, which can be downloaded from <https://rstudio.com/products/rstudio/> (choose RStudio Desktop Open Source Edition).

**Remark.** There are of course many other options for working with R, both interactive and non-interactive, including Jupyter Notebooks (see <https://irkernel.github.io/>), dynamically generated reports (see <https://yihui.org/knitr/options/>) and plain shell scripts executed from a terminal. However, for now let's leave that to more advanced users.

### A.3 Installing Recommended Packages

Once we get the above up and running, from within RStudio, we need to install a few packages which we're going to use during the course of this course. Execute the following commands in the R console (bottom-left Rstudio pane):

```
pkgs <- c("Cairo", "fastcluster", "FNN", "genie", "ISLR",
         "keras", "Matrix", "microbenchmark", "pdist",
         "RColorBrewer", "recommenderlab", "rpart",
         "rpart.plot", "scatterplot3d", "stringi", "tensorflow",
         "titanic", "vioplot")
install.packages(pkgs)
```

What is more, in order to be able to play with neural networks, we will need some Python environment, for example the Anaconda Distribution Python 3.x, see <https://www.anaconda.com/distribution/>.

**Remark.** Do **not** download Python 2.7.

Installation instructions can be found at <https://docs.anaconda.com/anaconda/install/>. This is required for the R packages tensorflow and keras, see <https://tensorflow.rstudio.com/installation/>. Once this is installed, execute the following R commands in the console:

```
library("tensorflow")
install_tensorflow()
```

### A.4 First R Script in RStudio

Let's open RStudio and perform the following steps:

1. Create a New Project where we will store all the scripts related to this book. Click *File* → *New Project* and then choose to start in a brand new working directory, in any location you like. Choose *New Project* as the project type.

From now on, we are assuming that the project name is *LMLCR* and the project has been opened. All source files we create will be relative to the project directory.

2. Create a new R source file, *File* → *New File* → *R Script*. Save the file as, for example, *sandbox\_01.R*.

The source editor (top left pane) behaves just like any other text editor. Standard keyboard shortcuts are available, such as CTRL+C and CTRL+V (Cmd+C and Cmd+V on MacOS) for copy and paste, respectively.

A list of keyboard shortcuts is available at <https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts>

3. Input the following R code into the editor:

```
# My first R script
# This is a comment

# Another comment

# Everything from '#' to the end of the line
#     is ignored by the R interpreter
print("Hello world") # prints a given character string
print(2+2) # evaluates the expression and prints the result
x <- seq(0, 10, length.out=100) # a new numeric vector
y <- x^2 # squares every element in x
plot(x, y, las=1, type="l") # plots y as a function of x
```

4. Execute the 5 above commands, line by line, by positioning the keyboard cursor accordingly and pressing Ctrl+Enter (Cmd+Return on MacOS).

Each time, the command will be copied to the console (bottom-left pane) and evaluated.

The last line generates a nice plot which will appear in the bottom-right pane.

While you learn, we recommend that you get used to writing your code in an R script and executing it just as we did above.

On a side note, you can execute (source) the whole script by pressing Ctrl+Shift+S (Cmd+Shift+S on MacOS).



## Appendix B

# Vector Algebra in R

This chapter is a step-by-step guide to vector computations in R. It also explains the basic mathematical notation around vectors.

You're encouraged to not only simply *read* the chapter, but also to execute yourself the R code provided. Play with it, do some experiments, get curious about how R works. Read the documentation on the functions you are calling, e.g., `?seq`, `?sample` and so on.

Technical and mathematical literature isn't belletristic. It requires *active* (*pro-active* even) thinking. Sometimes going through a single page can take an hour. Or a day. If you don't understand something, keep thinking, go back, ask yourself questions, take a look at other sources. This is not a *linear* process. This is what makes it fun and creative. To become a good programmer you need a lot of practice, there are no shortcuts. But the whole endeavour is worth the hassle!

### B.1 Motivation

Vector and matrix algebra provides us with a convenient language for expressing computations on sequential and tabular data.

Vector and matrix algebra operations are supported by every major programming language – either natively (e.g., R, Matlab, GNU Octave, Mathematica) or via an additional library/package (e.g, Python with numpy, tensorflow or pytorch; C++ with Eigen/Armadillo; C, C++ or Fortran with LAPACK).

By using matrix notation, we generate more concise and readable code.

For instance, given two vectors  $\mathbf{x} = (x_1, \dots, x_n)$  and  $\mathbf{y} = (y_1, \dots, y_n)$  like:

```
x <- c(1.5, 3.5, 2.3, -6.5)
y <- c(2.9, 8.2, -0.1, 0.8)
```

Instead of writing:

```
s <- 0
n <- length(x)
for (i in 1:n)
  s <- s + (x[i] - y[i])^2
sqrt(s)
```

```
## [1] 9.11592
```

to mean:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2},$$

which denotes the (Euclidean) distance between the two vectors (the square root of the sum of squared differences between the corresponding elements in  $\mathbf{x}$  and  $\mathbf{y}$ ), we shall soon become used to writing:

```
sqrt(sum((x-y)^2))
```

```
## [1] 9.11592
```

or:

$$\sqrt{(\mathbf{x} - \mathbf{y})^T (\mathbf{x} - \mathbf{y})}$$

or even:

$$\|\mathbf{x} - \mathbf{y}\|_2$$

In order to be able to read this notation, we only have to get to know the most common “building blocks”. There are just a few of them, but it’ll take some time until we become comfortable with their use.

What’s more, we should note that vectorised code might be much faster than the `for` loop-based one (a.k.a. “iterative” style):

```
library("microbenchmark")
n <- 10000
x <- runif(n) # n random numbers in [0, 1]
y <- runif(n)
print(microbenchmark(
  t1={
    # "iterative" style
    s <- 0
    n <- length(x)
```

```

for (i in 1:n)
  s <- s + (x[i]-y[i])^2
sqrt(s)
},
t2={
  # "vectorised" style
  sqrt(sum((x-y)^2))
\bigskip
), signif=3, unit='relative')

## Unit: relative
##   expr min  lq  mean median    uq  max neval
##   t1 127 123 102     117 93.9 91.7   100
##   t2    1    1    1      1  1.0  1.0   100

```

## B.2 Numeric Vectors

### B.2.1 Creating Numeric Vectors

First let's introduce a few ways with which we can create numeric vectors.

#### B.2.1.1 c()

The `c()` function combines a given list of values to form a sequence:

```

c(1, 2, 3)

## [1] 1 2 3

c(1, 2, 3, c(4, 5), c(6, c(7, 8)))

## [1] 1 2 3 4 5 6 7 8

```

Note that when we use the assignment operator, `<-` or `=` (both are equivalent), printing of the output is suppressed:

```

x <- c(1, 2, 3) # doesn't print anything
print(x)

```

```
## [1] 1 2 3
```

However, we can enforce it by parenthesising the whole expression:

```
(x <- c(1, 2, 3))
```

```
## [1] 1 2 3
```

In order to determine that `x` is indeed a numeric vector, we call:

```
mode(x)

## [1] "numeric"

class(x)

## [1] "numeric"
```

**Remark.** These two functions might return different results. For instance, in the next chapter we note that a numeric matrix will yield `mode()` of `numeric` and `class()` of `matrix`.

What is more, we can get the number of elements in `x` by calling:

```
length(x)

## [1] 3
```

### B.2.1.2 `seq()`

To create an arithmetic progression, i.e., a sequence of equally-spaced numbers, we can call the `seq()` function

```
seq(1, 9, 2)

## [1] 1 3 5 7 9
```

If we access the function's documentation (by executing `?seq` in the console), we'll note that the function takes a couple of parameters: `from`, `to`, `by`, `length.out` etc.

The above call is equivalent to:

```
seq(from=1, to=9, by=2)

## [1] 1 3 5 7 9
```

The `by` argument can be replaced with `length.out`, which gives the desired size:

```
seq(0, 1, length.out=5)

## [1] 0.00 0.25 0.50 0.75 1.00
```

Note that R supports partial matching of argument names:

```
seq(0, 1, len=5)

## [1] 0.00 0.25 0.50 0.75 1.00
```

Quite often we need progressions with step equal to 1 or -1. Such vectors can be generated by applying the `:` operator.

```
1:10      # from:to (inclusive)
```

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

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

### B.2.1.3 rep()

Moreover, `rep()` replicates a given vector. Check out the function's documentation (see `?rep`) for the meaning of the arguments provided below.

```
rep(1, 5)

## [1] 1 1 1 1 1

rep(1:3, 4)

## [1] 1 2 3 1 2 3 1 2 3 1 2 3
rep(1:3, c(2, 4, 3))

## [1] 1 1 2 2 2 3 3 3
rep(1:3, each=4)

## [1] 1 1 1 1 2 2 2 3 3 3 3
```

### B.2.1.4 Pseudo-Random Vectors

We can also generate vectors of pseudo-random values. For instance, the following generates 5 deviates from the uniform distribution (every number has the same probability) on the unit (i.e.,  $[0, 1]$ ) interval:

```
runif(5, 0, 1)

## [1] 0.5649012 0.5588051 0.4414829 0.2076393 0.6696355
```

We call such numbers pseudo-random, because they are generated arithmetically. In fact, by setting the random number generator's state (also called the *seed*), we can obtain *reproducible* results.

```
set.seed(123)
runif(5, 0, 1) # a,b,c,d,e

## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
runif(5, 0, 1) # f,g,h,i,j

## [1] 0.0455565 0.5281055 0.8924190 0.5514350 0.4566147
set.seed(123)
runif(5, 0, 1) # a,b,c,d,e again!

## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

Note the difference between the uniform distribution on  $[0, 1]$  and the normal distribution with expected value of 0 and standard deviation of 1 (also called the standard normal distribution), see Figure B.1.

```
par(mfrow=c(1, 2)) # align plots in one row and two columns
hist(runif(10000, 0, 1), col="white", ylim=c(0, 2500)); box()
hist(rnorm(10000, 0, 1), col="white", ylim=c(0, 2500)); box()
```

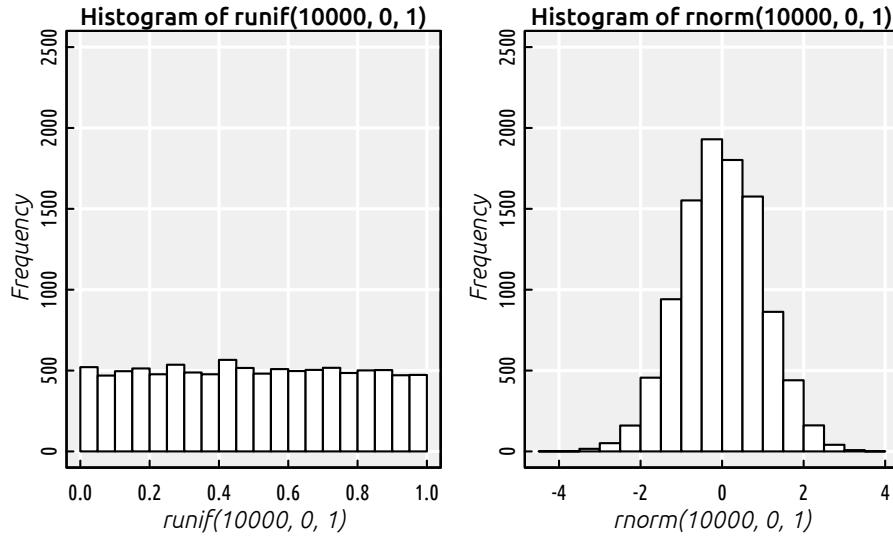


Figure B.1: Uniformly vs. normally distributed random variables

Another useful function samples a number of values from a given vector, either with or without replacement:

```
sample(1:10, 8, replace=TRUE) # with replacement
```

```
## [1] 3 3 10 2 6 5 4 6
```

```
sample(1:10, 8, replace=FALSE) # without replacement
```

```
## [1] 9 5 3 8 1 4 6 10
```

Note that if  $n$  is a single number, `sample(n, ...)` is equivalent to `sample(1:n, ...)`. This is a dangerous behaviour than may lead to bugs in our code. Read more at `?sample`.

## B.2.2 Vector-Scalar Operations

Mathematically, we sometimes refer to a vector that is reduced to a single component as a *scalar*. We are used to denoting such objects with lowercase letters such as  $a, b, i, s, x \in \mathbb{R}$ .

**Remark.** Note that some programming languages distinguish between atomic numerical entities and length-one vectors, e.g., 7 vs. [7] in Python. This is not the case in R, where `length(7)` returns 1.

Vector-scalar arithmetic operations such as  $s\mathbf{x}$  (multiplication of a vector  $\mathbf{x} = (x_1, \dots, x_n)$  by a scalar  $s$ ) result in a vector  $\mathbf{y}$  such that  $y_i = sx_i$ ,  $i = 1, \dots, n$ .

The same rule holds for, e.g.,  $s + \mathbf{x}$ ,  $\mathbf{x} - s$ ,  $\mathbf{x}/s$ .

```
0.5 * c(1, 10, 100)
```

```
## [1] 0.5 5.0 50.0
```

```
10 + 1:5
```

```
## [1] 11 12 13 14 15
```

```
seq(0, 10, by=2)/10
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

By  $-\mathbf{x}$  we will mean  $(-1)\mathbf{x}$ :

```
-seq(0, 1, length.out=5)
```

```
## [1] 0.00 -0.25 -0.50 -0.75 -1.00
```

Note that in R the same rule applies for exponentiation:

```
(0:5)^2 # synonym: (1:5)**2
```

```
## [1] 0 1 4 9 16 25
```

```
2^(0:5)
```

```
## [1] 1 2 4 8 16 32
```

However, in mathematics, we are **not** used to writing  $2^{\mathbf{x}}$  or  $\mathbf{x}^2$ .

### B.2.3 Vector-Vector Operations

Let  $\mathbf{x} = (x_1, \dots, x_n)$  and  $\mathbf{y} = (y_1, \dots, y_n)$  be two vectors of identical lengths.

Arithmetic operations  $\mathbf{x} + \mathbf{y}$  and  $\mathbf{x} - \mathbf{y}$  are performed *elementwise*, i.e., they result in a vector  $\mathbf{z}$  such that  $z_i = x_i + y_i$  and  $z_i = x_i - y_i$ , respectively,  $i = 1, \dots, n$ .

```
x <- c(1, 2, 3, 4)
```

```
y <- c(1, 10, 100, 1000)
```

```
x+y
```

```
## [1] 2 12 103 1004
```

```
x-y
```

```
## [1] 0 -8 -97 -996
```

Although in mathematics we are **not** used to using any special notation for elementwise multiplication, division and exponentiation, this is available in R.

```
x*y
## [1]    1   20  300 4000
x/y
## [1] 1.000 0.200 0.030 0.004
y^x
## [1] 1e+00 1e+02 1e+06 1e+12
```

**Remark.** `1e+12` is a number written in the *scientific notation*. It means “1 times 10 to the power of 12”, i.e.,  $1 \times 10^{12}$ . Physicists love this notation, because they are used to dealing with very small (think sizes of quarks) and very large (think distances between galaxies) entities.

Moreover, in R the **recycling rule** is applied if we perform elementwise operations on vectors of *different* lengths – the shorter vector is recycled as many times as needed to match the length of the longer vector, just as if we were performing:

```
rep(1:3, length.out=12) # recycle 1,2,3 to get 12 values
```

```
## [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

Therefore:

```
1:6 * c(1)
## [1] 1 2 3 4 5 6
1:6 * c(1,10)
## [1] 1 20 3 40 5 60
1:6 * c(1,10,100)
## [1] 1 20 300 4 50 600
1:6 * c(1,10,100,1000)
## Warning in 1:6 * c(1, 10, 100, 1000): longer object length is
## not a multiple of shorter object length
## [1] 1 20 300 4000 5 60
```

Note that a warning is not an error – we still get a result that makes sense.

## B.2.4 Aggregation Functions

R implements a couple of *aggregation* functions:

- `sum(x) =  $\sum_{i=1}^n x_i = x_1 + x_2 + \dots + x_n$`
- `prod(x) =  $\prod_{i=1}^n x_i = x_1 x_2 \dots x_n$`
- `mean(x) =  $\frac{1}{n} \sum_{i=1}^n x_i$`  – arithmetic mean
- `var(x) = sum((x-mean(x))^2)/(length(x)-1) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \frac{1}{n} \sum_{j=1}^n x_j)^2`
- variance
- `sd(x) = sqrt(var(x))` – standard deviation

see also: `min()`, `max()`, `median()`, `quantile()`.

**Remark.** Remember that you can always access the R manual by typing `?functionname`, e.g., `?quantile`.

**Remark.** Note that  $\sum_{i=1}^n x_i$  can also be written as  $\sum_{i=1}^n x_i$  or even  $\sum_{i=1, \dots, n} x_i$ .

These all mean the sum of  $x_i$  for  $i$  from 1 to  $n$ , that is, the sum of  $x_1, x_2, \dots, x_n$ , i.e.,  $x_1 + x_2 + \dots + x_n$ .

```
x <- runif(1000)
mean(x)

## [1] 0.4972778
median(x)

## [1] 0.4899503
min(x)

## [1] 0.0004653491
max(x)

## [1] 0.9994045
```

### B.2.5 Special Functions

Furthermore, R supports numerous mathematical functions, e.g., `sqrt()`, `abs()`, `round()`, `log()`, `exp()`, `cos()`, `sin()`. All of them are vectorised – when applied on a vector of length  $n$ , they yield a vector of length  $n$  in result.

For example, here is how we can compute the square roots of all the integers between 1 and 9:

```
sqrt(1:9)

## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
## [7] 2.645751 2.828427 3.000000
```

Vectorisation is super-convenient when it comes to, for instance, plotting (see Figure B.2).

```
x <- seq(-2*pi, 6*pi, length.out=51)
plot(x, sin(x), type="l")
lines(x, cos(x), col="red") # add a curve to the current plot
```

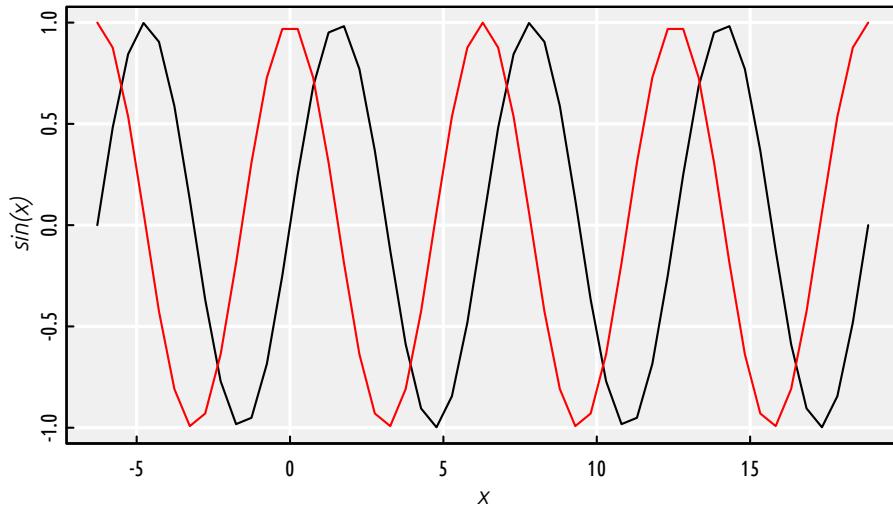


Figure B.2: An example plot of the sine and cosine functions

**Exercise.**

Try increasing the `length.out` argument to make the curves smoother.

□

### B.2.6 Norms and Distances

Norms are used to measure the *size* of an object. Mathematically, we will also be interested in the following norms:

- Euclidean norm:

$$\|\mathbf{x}\| = \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

this is nothing else than the *length* of the vector  $\mathbf{x}$

- Manhattan (taxicab) norm:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

- Chebyshev (maximum) norm:

$$\|\mathbf{x}\|_\infty = \max_{i=1,\dots,n} |x_i| = \max\{|x_1|, |x_2|, \dots, |x_n|\}$$

The above norms can be easily implemented by means of the building blocks introduced above. This is super easy:

```
z <- c(1, 2)
sqrt(sum(z^2)) # or norm(z, "2"); Euclidean

## [1] 2.236068
sum(abs(z))    # Manhattan

## [1] 3
max(abs(z))    # Chebyshev

## [1] 2
```

Also note that all the norms easily generate the corresponding *distances* (metrics) between two given points. In particular:

$$\|\mathbf{x} - \mathbf{y}\| = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

gives the *Euclidean distance* (metric) between the two vectors.

```
u <- c(1, 0)
v <- c(1, 1)
sqrt(sum((u-v)^2))

## [1] 1
```

This is the “normal” distance that you learned at school.

### B.2.7 Dot Product (\*)

What is more, given two vectors of identical lengths,  $\mathbf{x}$  and  $\mathbf{y}$ , we define their *dot product* (a.k.a. *scalar* or *inner product*) as:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i.$$

Let’s stress that this is not the same as the elementwise vector multiplication in R – the result is a single number.

```
u <- c(1, 0)
v <- c(1, 1)
sum(u*v)
```

```
## [1] 1
```

**Remark.** (\*) Note that the squared Euclidean norm of a vector is equal to the dot product of the vector and itself,  $\|\mathbf{x}\|^2 = \mathbf{x} \cdot \mathbf{x}$ .

(\*) Interestingly, a dot product has a nice geometrical interpretation:

$$\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos \alpha$$

where  $\alpha$  is the angle between the two vectors. In other words, it is the product of the lengths of the two vectors and the cosine of the angle between them. Note that we can get the cosine part by computing the dot product of the *normalised* vectors, i.e., such that their lengths are equal to 1.

For example, the two vectors  $\mathbf{u}$  and  $\mathbf{v}$  defined above can be depicted as in Figure B.3.

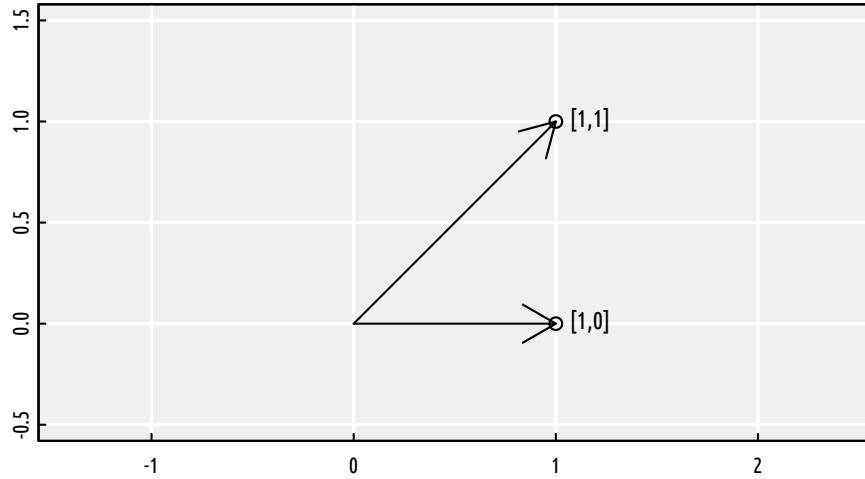


Figure B.3: Example vectors in 2D

We can compute the angle between them by calling:

```
(len_u <- sqrt(sum(u^2))) # length == Euclidean norm
## [1] 1
(len_v <- sqrt(sum(v^2)))
## [1] 1.414214
```

```
(cos_angle_uv <- (sum(u*v)/(len_u*len_v))) # cosine of the angle
## [1] 0.7071068
acos(cos_angle_uv)*180/pi # angle in degs
## [1] 45
```

### B.2.8 Missing and Other Special Values

R has a notion of a missing (not-available) value. It is very useful in data analysis, as we sometimes don't have an information on an object's feature. For instance, we might not know a patient's age if he was admitted to the hospital unconscious.

```
x <- c(1, 2, NA, 4, 5)
```

Operations on missing values generally result in missing values – that makes a lot sense.

```
x + 11:15
## [1] 12 14 NA 18 20
mean(x)
## [1] NA
```

If we wish to compute a vector's aggregate after all, we can get rid of the missing values by calling `na.omit()`:

```
mean(na.omit(x)) # mean of non-missing values
## [1] 3
```

We can also make use of the `na.rm` parameter of the `mean()` function (however, not every aggregation function has it – always refer to documentation).

```
mean(x, na.rm=TRUE)
## [1] 3
```

**Remark.** Note that in R, a dot has no special meaning. `na.omit` is as good of a function's name or variable identifier as `na_omit`, `na0mit`, `NAOMIT`, `naomit` and `Na0mit`. Note that, however, R is a case-sensitive language – these are all different symbols. Read more in the *Details* section of `?make.names`.

Moreover, some arithmetic operations can result in infinities ( $\pm\infty$ ):

```
log(0)
## [1] -Inf
```

```

10^1000 # too large

## [1] Inf

Also, sometimes we'll get a not-a-number, NaN. This is not a missing value, but
a "invalid" result.

sqrt(-1)

## Warning in sqrt(-1): NaNs produced

## [1] NaN

log(-1)

## Warning in log(-1): NaNs produced

## [1] NaN

Inf-Inf

## [1] NaN

```

## B.3 Logical Vectors

### B.3.1 Creating Logical Vectors

In R there are 3 (!) logical values: TRUE, FALSE and geez, I don't know, NA  
maybe?

```

c(TRUE, FALSE, TRUE, NA, FALSE, FALSE, TRUE)

## [1] TRUE FALSE TRUE     NA FALSE FALSE TRUE
(x <- rep(c(TRUE, FALSE, NA), 2))

## [1] TRUE FALSE     NA  TRUE FALSE     NA
mode(x)

## [1] "logical"
class(x)

## [1] "logical"
length(x)

## [1] 6

```

**Remark.** By default, T is a synonym for TRUE and F for FALSE. This may be changed though so it's better not to rely on these.

### B.3.2 Logical Operations

Logical operators such as `&` (and) and `|` (or) are performed in the same manner as arithmetic ones, i.e.:

- they are elementwise operations and
- recycling rule is applied if necessary.

For example,

```
TRUE & TRUE
```

```
## [1] TRUE
TRUE & c(TRUE, FALSE)

## [1] TRUE FALSE
c(FALSE, FALSE, TRUE, TRUE) | c(TRUE, FALSE, TRUE, FALSE)

## [1] TRUE FALSE TRUE TRUE
```

The `!` operator stands for logical elementwise negation:

```
!c(TRUE, FALSE)
```

```
## [1] FALSE TRUE
```

Generally, operations on NAs yield `NA` unless other solution makes sense.

```
u <- c(TRUE, FALSE, NA)
v <- c(TRUE, TRUE, TRUE, FALSE, FALSE, NA, NA, NA)
u & v # elementwise AND (conjunction)

## [1] TRUE FALSE NA FALSE FALSE FALSE NA FALSE NA
u | v # elementwise OR (disjunction)

## [1] TRUE TRUE TRUE TRUE FALSE NA TRUE NA NA
!u # elementwise NOT (negation)

## [1] FALSE TRUE NA
```

### B.3.3 Comparison Operations

We can compare the corresponding elements of two numeric vectors and get a logical vector in result. Operators such as `<` (less than), `<=` (less than or equal), `==` (equal), `!=` (not equal), `>` (greater than) and `>=` (greater than or equal) are again elementwise and use the recycling rule if necessary.

```
3 < 1:5 # c(3, 3, 3, 3, 3) < c(1, 2, 3, 4, 5)
```

```
## [1] FALSE FALSE FALSE TRUE TRUE
```

```
1:2 == 1:4 # c(1,2,1,2) == c(1,2,3,4)

## [1] TRUE TRUE FALSE FALSE
z <- c(0, 3, -1, 1, 0.5)
(z >= 0) & (z <= 1)

## [1] TRUE FALSE FALSE TRUE TRUE
```

### B.3.4 Aggregation Functions

Also note the following operations on *logical* vectors:

```
z <- 1:10
all(z >= 5) # are all values TRUE?

## [1] FALSE
any(z >= 5) # is there any value TRUE?

## [1] TRUE
```

Moreover:

```
sum(z >= 5) # how many TRUE values are there?

## [1] 6
mean(z >= 5) # what is the proportion of TRUE values?

## [1] 0.6
```

The behaviour of `sum()` and `mean()` is dictated by the fact that, when interpreted in numeric terms, TRUE is interpreted as numeric 1 and FALSE as 0.

```
as.numeric(c(FALSE, TRUE))

## [1] 0 1
```

Therefore in the example above we have:

```
z >= 5

## [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
as.numeric(z >= 5)

## [1] 0 0 0 0 1 1 1 1 1 1
sum(as.numeric(z >= 5)) # the same as sum(z >= 5)

## [1] 6
```

Yes, there are 6 values equal to TRUE (or 6 ones after conversion), the sum of zeros and ones gives the number of ones.

## B.4 Character Vectors

### B.4.1 Creating Character Vectors

Individual character strings can be created using double quotes or apostrophes. These are the elements of character vectors

```
(x <- "a string")
## [1] "a string"
mode(x)
## [1] "character"
class(x)
## [1] "character"
length(x)
## [1] 1
rep(c("aaa", 'bb', "c"), 2)
## [1] "aaa" "bb"  "c"    "aaa" "bb"  "c"
```

### B.4.2 Concatenating Character Vectors

To join (concatenate) the corresponding elements of two or more character vectors, we call the `paste()` function:

```
paste(c("a", "b", "c"), c("1", "2", "3"))
## [1] "a 1" "b 2" "c 3"
paste(c("a", "b", "c"), c("1", "2", "3"), sep="")
## [1] "a1" "b2" "c3"
```

Also note:

```
paste(c("a", "b", "c"), 1:3) # the same as as.character(1:3)
## [1] "a 1" "b 2" "c 3"
paste(c("a", "b", "c"), 1:6) # recycling
## [1] "a 1" "b 2" "c 3" "a 4" "b 5" "c 6"
paste(c("a", "b", "c"), 1:6, c("!", "?"))
## [1] "a 1 !" "b 2 ?" "c 3 !" "a 4 ?" "b 5 !" "c 6 ?"
```

### B.4.3 Collapsing Character Vectors

We can also collapse a sequence of strings to a single string:

```
paste(c("a", "b", "c", "d"), collapse="")
## [1] "abcd"
paste(c("a", "b", "c", "d"), collapse=",")
## [1] "a,b,c,d"
```

## B.5 Vector Subsetting

### B.5.1 Subsetting with Positive Indices

In order to extract subsets (parts) of vectors, we use the square brackets:

```
(x <- seq(10, 100))
## [1] 10 20 30 40 50 60 70 80 90 100
x[1]          # the first element
## [1] 10
x[length(x)] # the last element
## [1] 100
```

More than one element at a time can also be extracted:

```
x[1:3] # the first three
## [1] 10 20 30
x[c(1, length(x))] # the first and the last
## [1] 10 100
```

For example, the `order()` function returns the indices of the smallest, 2nd smallest, 3rd smallest, ..., the largest element in a given vector. We will use this function when implementing our first classifier.

```
y <- c(50, 30, 10, 20, 40)
(o <- order(y))
## [1] 3 4 2 5 1
```

Hence, we see that the smallest element in `y` is at index 3 and the largest at index 1:

```
y[o[1]]
```

```
## [1] 10
y[o[length(y)]]
```

```
## [1] 50
```

Therefore, to get a sorted version of  $y$ , we call:

```
y[o] # see also sort(y)
```

```
## [1] 10 20 30 40 50
```

We can also obtain the 3 largest elements by calling:

```
y[order(y, decreasing=TRUE)[1:3]]
```

```
## [1] 50 40 30
```

### B.5.2 Subsetting with Negative Indices

Subsetting with a vector of negative indices, *excludes* the elements at given positions:

```
x[-1] # all but the first
```

```
## [1] 20 30 40 50 60 70 80 90 100
```

```
x[-(1:3)]
```

```
## [1] 40 50 60 70 80 90 100
```

```
x[-c(1:3, 5, 8)]
```

```
## [1] 40 60 70 90 100
```

### B.5.3 Subsetting with Logical Vectors

We may also subset a vector  $x$  of length  $n$  with a logical vector  $l$  also of length  $n$ . The  $i$ -th element,  $x_i$ , will be extracted if and only if the corresponding  $l_i$  is true.

```
x[c(TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 10 50 70 80 100
```

This gets along nicely with comparison operators that yield logical vectors on output.

```
x>50
```

```
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
x[x>50] # select elements in x that are greater than 50
```

```
## [1] 60 70 80 90 100
```

```
x[x<30 | x>70]

## [1] 10 20 80 90 100
x[x<max(x)] # getting rid of the greatest element

## [1] 10 20 30 40 50 60 70 80 90
x[x > min(x) & x < max(x)] # return all but the smallest and greatest one

## [1] 20 30 40 50 60 70 80 90
```

Of course, e.g., `x[x<max(x)]` returns a new, independent object. In order to remove the greatest element in `x` permanently, we can write `x <- x[x<max(x)]`.

#### B.5.4 Replacing Elements

Note that the three above vector indexing schemes (positive, negative, logical indices) allow for replacing specific elements with new values.

```
x[-1] <- 10000
x

## [1] 10 10000 10000 10000 10000 10000 10000 10000 10000 10000
x[-(1:7)] <- c(1, 2, 3)
x

## [1] 10 10000 10000 10000 10000 10000 10000 10000 10000 10000
1 2 3
```

#### B.5.5 Other Functions

`head()` and `tail()` return, respectively, a few (6 by default) first and last elements of a vector.

```
head(x) # head(x, 6)

## [1] 10 10000 10000 10000 10000 10000
tail(x, 3)

## [1] 1 2 3
```

Sometimes the `which()` function can come in handy. For a given logical vector, it returns all the indices where TRUE elements are stored.

```
which(c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, TRUE))

## [1] 1 3 4 7
print(y) # recall

## [1] 50 30 10 20 40
```

```
which(y>30)
```

```
## [1] 1 5
```

Note that `y[y>70]` gives the same result as `y[which(y>70)]` but is faster (because it involves less operations).

`which.min()` and `which.max()` return the index of the smallest and the largest element, respectively:

```
which.min(y) # where is the minimum?
```

```
## [1] 3
```

```
which.max(y)
```

```
## [1] 1
```

```
y[which.min(y)] # min(y)
```

```
## [1] 10
```

`is.na()` indicates which elements are missing values (NAAs):

```
z <- c(1, 2, NA, 4, NA, 6)
is.na(z)
```

```
## [1] FALSE FALSE TRUE FALSE TRUE FALSE
```

Therefore, to remove them from `z` permanently, we can write (compare `na.omit()`, see also `is.finite()`):

```
(z <- z[!is.na(z)])
```

```
## [1] 1 2 4 6
```

## B.6 Named Vectors

### B.6.1 Creating Named Vectors

Vectors in R can be *named* – each element can be assigned a string label.

```
x <- c(20, 40, 99, 30, 10)
names(x) <- c("a", "b", "c", "d", "e")
x # a named vector
```

```
## a b c d e
## 20 40 99 30 10
```

Other ways to create named vectors include:

```
c(a=1, b=2, c=3)
```

```
## a b c
## 1 2 3
structure(1:3, names=c("a", "b", "c"))

## a b c
## 1 2 3
```

For instance, the `summary()` function returns a named vector:

```
summary(x) # NAMED vector, we don't want this here yet
```

```
##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
##      10.0    20.0   30.0    39.8   40.0    99.0
```

This gives the minimum, 1st quartile (25%-quantile), Median (50%-quantile), arithmetic mean, 3rd quartile (75%-quantile) and maximum.

Note that `x` is still a numeric vector, we can perform various operations on it as usual:

```
sum(x)

## [1] 199

x[x>3]

## a b c d e
## 20 40 99 30 10
```

Names can be dropped by calling:

```
unname(x)

## [1] 20 40 99 30 10

as.numeric(x) # we need to know the type of x though

## [1] 20 40 99 30 10
```

## B.6.2 Subsetting Named Vectors with Character String Indices

It turns out that extracting elements from a named vector can *also* be performed by means of a vector of character string indices:

```
x[c("a", "d", "b")]

## a d b
## 20 30 40

summary(x)[c("Median", "Mean")]

## Median      Mean
```

```
##   30.0   39.8
```

## B.7 Factors

Factors are *special* kinds of vectors that are frequently used to store qualitative data, e.g., species, groups, types. Factors are convenient in situations where we have many observations, but the number of distinct (unique) values is relatively small.

### B.7.1 Creating Factors

For example, the following character vector:

```
(col <- sample(c("blue", "red", "green"), replace=TRUE, 10))
```

```
## [1] "green" "green" "green" "red"   "green" "red"   "red"
## [8] "red"   "green" "blue"
```

can be converted to a factor by calling:

```
(fcol <- factor(col))
```

```
## [1] green green green red   green red   red   red   green blue
## Levels: blue green red
```

Note how different is the way factors are printed out on the console.

### B.7.2 Levels

We can easily obtain the list unique labels:

```
levels(fcol)
```

```
## [1] "blue"  "green" "red"
```

Those can be re-encoded by calling:

```
levels(fcol) <- c("b", "g", "r")
fcol
```

```
## [1] g g g r g r r r g b
## Levels: b g r
```

To create a contingency table (in the form of a named numeric vector, giving how many values are at each factor level), we call:

```
table(fcol)
```

```
## fcol
## b g r
## 1 5 4
```

### B.7.3 Internal Representation (\*)

Factors have a look-and-feel of character vectors, however, internally they are represented as integer sequences.

```
class(fcol)

## [1] "factor"
mode(fcol)

## [1] "numeric"
as.numeric(fcol)

## [1] 2 2 2 3 2 3 3 3 2 1
```

These are always integers from 1 to M inclusive, where M is the number of levels. Their meaning is given by the `levels()` function: in the example above, the meaning of the codes 1, 2, 3 is, respectively, `b`, `g`, `r`.

If we wished to generate a factor with a specific order of labels, we could call:

```
factor(col, levels=c("red", "green", "blue"))

## [1] green green green red   green red   red   red   green blue
## Levels: red green blue
```

We can also assign different labels upon creation of a factor:

```
factor(col, levels=c("red", "green", "blue"), labels=c("r", "g", "b"))

## [1] g g g r g r r r g b
## Levels: r g b
```

Knowing how factors are represented is important when we deal with factors that are built around data that *look like* numeric. This is because their conversion to numeric gives the internal codes, not the actual values:

```
(f <- factor(c(1, 3, 0, 1, 4, 0, 0, 1, 4)))

## [1] 1 3 0 1 4 0 0 1 4
## Levels: 0 1 3 4

as.numeric(f) # not necessarily what we want here

## [1] 2 3 1 2 4 1 1 2 4

as.numeric(as.character(f)) # much better

## [1] 1 3 0 1 4 0 0 1 4
```

Moreover, that idea is labour-saving in contexts such as plotting of data that are grouped into different classes. For instance, here is a scatter plot for the Sepal.Length and Petal.Width variables in the `iris` dataset (which is an object

of type `data.frame`, see below). Flowers are of different Species, and we wish to indicate which point belongs to which class:

```
which_preview <- c(1, 11, 51, 69, 101) # indexes we show below
iris$Sepal.Length[which_preview]

## [1] 5.1 5.4 7.0 6.2 6.3

iris$Petal.Width[which_preview]

## [1] 0.2 0.2 1.4 1.5 2.5

iris$Species[which_preview]

## [1] setosa      setosa      versicolor versicolor virginica
## Levels: setosa versicolor virginica
as.numeric(iris$Species)[which_preview]

## [1] 1 1 2 2 3

plot(iris$Sepal.Length, # x (it's a vector)
      iris$Petal.Width, # y (it's a vector)
      col=as.numeric(iris$Species), # colours
      pch=as.numeric(iris$Species)
)
```

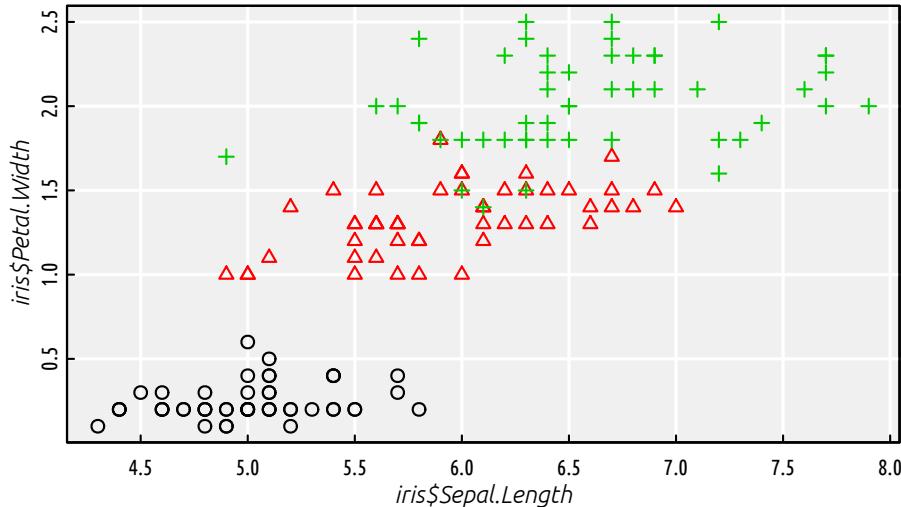


Figure B.4: `as.numeric()` on factors can be used to create different plotting styles

The above (see Figure B.4) was possible because the Species column is a factor

object with:

```
levels(iris$Species)

## [1] "setosa"     "versicolor"  "virginica"
```

and the meaning of pch of 1, 2, 3, ... is “circle”, “triangle”, “plus”, ..., respectively. What’s more, there’s a default palette that maps consecutive integers to different colours:

```
palette()
```

```
## [1] "black"      "red"       "green3"     "blue"      "cyan"      "magenta"
## [7] "yellow"     "gray"
```

Hence, black circles mark irises from the 1st class, i.e., “setosa”.

## B.8 Lists

Numeric, logical and character vectors are *atomic* objects – each component is of the same type. Let’s take a look at what happens when we create an atomic vector out of objects of different types:

```
c("nine", FALSE, 7, TRUE)

## [1] "nine"    "FALSE"   "7"        "TRUE"

c(FALSE, 7, TRUE, 7)

## [1] 0 7 1 7
```

In each case, we get an object of the most “general” type which is able to represent our data.

On the other hand, R *lists* are *generalised* vectors. They can consist of arbitrary R objects, possibly of mixed types – also other lists.

### B.8.1 Creating Lists

Most commonly, we create a generalised vector by calling the `list()` function.

```
(l <- list(1:5, letters, runif(3)))

## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
## [16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
##
## [[3]]
```

```
## [1] 0.9568333 0.4533342 0.6775706
mode(1)

## [1] "list"
class(1)

## [1] "list"
length(1)

## [1] 3
```

There's a more compact way to print a list on the console:

```
str(1)

## List of 3
## $ : int [1:5] 1 2 3 4 5
## $ : chr [1:26] "a" "b" "c" "d" ...
## $ : num [1:3] 0.957 0.453 0.678
```

We can also convert an atomic vector to a list by calling:

```
as.list(1:3)

## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

## B.8.2 Named Lists

List, like other vectors, may be assigned a `names` attribute.

```
names(1) <- c("a", "b", "c")
l

## $a
## [1] 1 2 3 4 5
##
## $b
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
## [16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
##
## $c
## [1] 0.9568333 0.4533342 0.6775706
```

### B.8.3 Subsetting and Extracting From Lists

Applying a square brackets operator creates a sub-list, which is of type list as well.

```
1[-1]

## $b
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
## [16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
##
## $c
## [1] 0.9568333 0.4533342 0.6775706

1[c("a", "c")]

## $a
## [1] 1 2 3 4 5
##
## $c
## [1] 0.9568333 0.4533342 0.6775706

1[1]

## $a
## [1] 1 2 3 4 5
```

Note in the 3rd case we deal with a list of length one, not a numeric vector.

To *extract* (dig into) a particular (single) element, we use double square brackets:

```
1[[1]]

## [1] 1 2 3 4 5

1[["c"]]

## [1] 0.9568333 0.4533342 0.6775706
```

The latter can equivalently be written as:

```
1$c

## [1] 0.9568333 0.4533342 0.6775706
```

### B.8.4 Common Operations

Lists, because of their generality (they can store any kind of object), have few dedicated operations. In particular, it neither makes sense to add, multiply, ... two lists together nor to aggregate them.

However, if we wish to run some operation on each element, we can call list-apply:

```
(k <- list(x=runif(5), y=runif(6), z=runif(3))) # a named list

## $x
## [1] 0.57263340 0.10292468 0.89982497 0.24608773 0.04205953
##
## $y
## [1] 0.3279207 0.9545036 0.8895393 0.6928034 0.6405068 0.9942698
##
## $z
## [1] 0.6557058 0.7085305 0.5440660

lapply(k, mean)

## $x
## [1] 0.3727061
##
## $y
## [1] 0.7499239
##
## $z
## [1] 0.6361008
```

The above computes the mean of each of the three numeric vectors stored inside list `k`. Moreover:

```
lapply(k, range)

## $x
## [1] 0.04205953 0.89982497
##
## $y
## [1] 0.3279207 0.9942698
##
## $z
## [1] 0.5440660 0.7085305
```

The built-in function `range(x)` returns `c(min(x), max(x))`.

`unlist()` tries (it might not always be possible) to unwind a list to a simpler, atomic form:

```
unlist(lapply(k, mean))
```

```
##      x      y      z
## 0.3727061 0.7499239 0.6361008
```

Moreover, `split(x, f)` classifies elements in a vector `x` into subgroups defined by a factor (or an object coercible to) of the same length.

```
x <- c( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
f <- c("a", "b", "a", "a", "c", "b", "b", "a", "a", "b")
split(x, f)

## $a
## [1] 1 3 4 8 9
##
## $b
## [1] 2 6 7 10
##
## $c
## [1] 5
```

This is very useful when combined with `lapply()` and `unlist()`. For instance, here are the mean sepal lengths for each of the three flower species in the famous `iris` dataset.

```
unlist(lapply(split(iris$Sepal.Length, iris$Species), mean))

##      setosa versicolor  virginica
##      5.006     5.936     6.588
```

By the way, if we take a look at the documentation of `?lapply`, we will note that that this function is defined as `lapply(X, FUN, ...)`. Here `...` denotes the optional arguments that will be passed to `FUN`.

In other words, `lapply(X, FUN, ...)` returns a list `Y` of length `length(X)` such that `Y[[i]] <- FUN(X[[i]], ...)` for each `i`. For example, `mean()` has an additional argument `na.rm` that aims to remove missing values from the input vector. Compare the following:

```
t <- list(1:10, c(1, 2, NA, 4, 5))
unlist(lapply(t, mean))

## [1] 5.5 NA
unlist(lapply(t, mean, na.rm=TRUE))

## [1] 5.5 3.0
```

## B.9 Further Reading

Recommended further reading: (Venables et al. 2020)

Other: (Deisenroth et al. 2020), (Peng 2019), (Wickham & Grolemund 2017)

## Appendix C

# Matrix Algebra in R

Vectors are one-dimensional objects – they represent “flat” sequences of values. Matrices, on the other hand, are two-dimensional – they represent tabular data, where values aligned into rows and columns. Matrices (and their extensions – data frames, which we’ll cover in the next chapter) are predominant in data science, where objects are typically represented by means of feature vectors.

Below are some examples of structured datasets in matrix forms.

```
head(as.matrix(iris[,1:4]))
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## [1,]         5.1       3.5        1.4       0.2
## [2,]         4.9       3.0        1.4       0.2
## [3,]         4.7       3.2        1.3       0.2
## [4,]         4.6       3.1        1.5       0.2
## [5,]         5.0       3.6        1.4       0.2
## [6,]         5.4       3.9        1.7       0.4
```

```
WorldPhones
```

```
##      N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
## 1951  45939 21574 2876   1815    1646     89     555
## 1956  60423 29990 4708   2568    2366   1411     733
## 1957  64721 32510 5230   2695    2526   1546     773
## 1958  68484 35218 6662   2845    2691   1663     836
## 1959  71799 37598 6856   3000    2868   1769     911
## 1960  76036 40341 8220   3145    3054   1905    1008
## 1961  79831 43173 9053   3338    3224   2005    1076
```

The aim of this chapter is to cover the most essential matrix operations, both from the computational perspective and the mathematical one.

## C.1 Creating Matrices

### C.1.1 `matrix()`

A matrix can be created – amongst others – with a call to the `matrix()` function.

```
(A <- matrix(c(1, 2, 3, 4, 5, 6), byrow=TRUE, nrow=2))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
class(A)
```

```
## [1] "matrix"
```

Given a numeric vector of length 6, we've asked R to convert to a numeric matrix with 2 rows (the `nrow` argument). The number of columns has been deduced automatically (otherwise, we would additionally have to pass `ncol=3` to the function).

Using mathematical notation, above we have defined  $\mathbf{A} \in \mathbb{R}^{2 \times 3}$ :

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

We can fetch the size of the matrix by calling:

```
dim(A) # number of rows, number of columns
```

```
## [1] 2 3
```

We can also “promote” a “flat” vector to a column vector, i.e., a matrix with one column by calling:

```
as.matrix(1:3)
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

### C.1.2 Stacking Vectors

Other ways to create a matrix involve stacking a couple of vectors of equal lengths along each other:

```
rbind(1:3, 4:6, 7:9) # row bind
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
```

```

## [2,]    4    5    6
## [3,]    7    8    9
cbind(1:3, 4:6, 7:9) # column bind

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

```

These functions also allow for adding new rows/columns to existing matrices:

```

rbind(A, c(-1, -2, -3))

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]   -1   -2   -3

cbind(A, c(-1, -2))

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3   -1
## [2,]    4    5    6   -2

```

### C.1.3 Beyond Numeric Matrices

Note that logical matrices are possible as well. For instance, knowing that comparison such as `<` and `==` are performed elementwise also in the case of matrices, we can obtain:

```

A >= 3

##      [,1] [,2] [,3]
## [1,] FALSE FALSE TRUE
## [2,]  TRUE  TRUE  TRUE

```

Moreover, although much more rarely used, we can define character matrices:

```

matrix(LETTERS[1:12], ncol=6)

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] "A"  "C"  "E"  "G"  "I"  "K"
## [2,] "B"  "D"  "F"  "H"  "J"  "L"

```

### C.1.4 Naming Rows and Columns

Just like vectors could be equipped with `names` attribute:

```
c(a=1, b=2, c=3)
```

```
## a b c
```

```
## 1 2 3
```

matrices can be assigned row and column labels in the form of a list of two character vectors:

```
dimnames(A) <- list(
  c("a", "b"),      # row labels
  c("x", "y", "z") # column labels
)
A

##   x y z
## a 1 2 3
## b 4 5 6
```

### C.1.5 Other Methods

The `read.table()` (and its special case, `read.csv()`), can be used to read a matrix from a text file. We will cover it in the next chapter, because technically it returns a data frame object (which we can convert to a matrix with a call to `as.matrix()`).

`outer()` applies a given (vectorised) function on each pair of elements from two vectors, forming a two-dimensional “grid”. More precisely `outer(x, y, f, ...)` returns a matrix  $Z$  with `length(x)` rows and `length(y)` columns such that  $z_{i,j} = f(x_i, y_j, \dots)$ , where  $\dots$  are optional further arguments to `f`.

```
outer(c(1, 10, 100), 1:5, "*") # apply the multiplication operator

##      [,1] [,2] [,3] [,4] [,5]
## [1,]     1    2    3    4    5
## [2,]    10   20   30   40   50
## [3,]   100  200  300  400  500

outer(c("A", "B"), 1:8, paste, sep="-") # concatenate strings

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] "A-1" "A-2" "A-3" "A-4" "A-5" "A-6" "A-7" "A-8"
## [2,] "B-1" "B-2" "B-3" "B-4" "B-5" "B-6" "B-7" "B-8"
```

`simplify2array()` is an extension of the `unlist()` function. Given a list of vectors, each of length one, it will return an “unlisted” vector. However, if a list of equisized vectors of greater lengths is given, these will be converted to a matrix.

```
simplify2array(list(1, 11, 21))
```

```
## [1] 1 11 21
```

```
simplify2array(list(1:3, 11:13, 21:23))
```

```

##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    2   12   22
## [3,]    3   13   23
simplify2array(list(1, 11:12, 21:23)) # no can do

## [[1]]
## [1] 1
##
## [[2]]
## [1] 11 12
##
## [[3]]
## [1] 21 22 23

sapply(...) is a nice application of the above, meaning simplify2array(lapply(...)).
sapply(split(iris$Sepal.Length, iris$Species), mean)

##      setosa versicolor  virginica
##      5.006      5.936      6.588
sapply(split(iris$Sepal.Length, iris$Species), summary)

##      setosa versicolor  virginica
## Min.   4.300   4.900   4.900
## 1st Qu. 4.800   5.600   6.225
## Median 5.000   5.900   6.500
## Mean   5.006   5.936   6.588
## 3rd Qu. 5.200   6.300   6.900
## Max.   5.800   7.000   7.900

```

Of course, custom functions can also be applied:

```

min_mean_max <- function(x) {
  # returns a named vector with three elements
  # (note that the last expression in a function's body
  # is its return value)
  c(min=min(x), mean=mean(x), max=max(x))
}
sapply(split(iris$Sepal.Length, iris$Species), min_mean_max)

##      setosa versicolor  virginica
## min   4.300   4.900   4.900
## mean  5.006   5.936   6.588
## max   5.800   7.000   7.900

```

Lastly, `table(x, y)` creates a contingency matrix that counts the number of unique pairs of corresponding elements from two vectors of equal lengths.

```

library("titanic") # data on the passengers of the RMS Titanic
table(titanic_train$Survived)

## 
##    0    1
## 549 342

table(titanic_train$Sex)

## 
## female male
##   314   577

table(titanic_train$Survived, titanic_train$Sex)

## 
##      female male
## 0     81   468
## 1    233   109

```

### C.1.6 Internal Representation (\*)

Note that by setting `byrow=TRUE` in a call to the `matrix()` function above, we are reading the elements of the input vector in the row-wise (row-major) fashion. The default is the column-major order, which might be a little unintuitive for some of us.

```

A <- matrix(c(1, 2, 3, 4, 5, 6), ncol=3, byrow=TRUE)
B <- matrix(c(1, 2, 3, 4, 5, 6), ncol=3) # byrow=FALSE

```

It turns out that is exactly the order in which the matrix is stored internally. Under the hood, it is an ordinary numeric vector:

```

mode(B)      # == mode(A)

## [1] "numeric"
length(B)   # == length(A)

## [1] 6
as.numeric(A)

## [1] 1 4 2 5 3 6
as.numeric(B)

## [1] 1 2 3 4 5 6

```

Also note that we can create a different *view* on the same underlying data vector:

```

dim(A) <- c(3, 2) # 3 rows, 2 columns
A

##      [,1] [,2]
## [1,]    1    5
## [2,]    4    3
## [3,]    2    6

dim(B) <- c(3, 2) # 3 rows, 2 columns
B

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6

```

## C.2 Common Operations

### C.2.1 Matrix Transpose

The matrix *transpose* is denoted with  $\mathbf{A}^T$ :

```
t(A)
```

```

##      [,1] [,2] [,3]
## [1,]    1    4    2
## [2,]    5    3    6

```

Hence,  $\mathbf{B} = \mathbf{A}^T$  is a matrix such that  $b_{i,j} = a_{j,i}$ .

In other words, in the transposed matrix, rows become columns and columns become rows. For example:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \quad \mathbf{A}^T = \begin{bmatrix} a_{1,1} & a_{2,1} \\ a_{1,2} & a_{2,2} \\ a_{1,3} & a_{2,3} \end{bmatrix}$$

### C.2.2 Matrix-Scalar Operations

Operations such as  $s\mathbf{A}$  (multiplication of a matrix by a scalar),  $-\mathbf{A}$ ,  $s + \mathbf{A}$  etc. are applied on each element of the input matrix:

```
(A <- matrix(c(1, 2, 3, 4, 5, 6), byrow=TRUE, nrow=2))
```

```

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6

```

```
(-1)*A
```

```
##      [,1] [,2] [,3]
## [1,]    -1   -2   -3
## [2,]    -4   -5   -6
```

In R, the same rule holds when we compute other operations (despite the fact that, mathematically, e.g.,  $\mathbf{A}^2$  or  $\mathbf{A} \geq 0$  might have a different meaning):

```
A^2 # this is not A-matrix-multiply-A, see below
```

```
##      [,1] [,2] [,3]
## [1,]     1     4     9
## [2,]    16    25    36
```

```
A>=3
```

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

### C.2.3 Matrix-Matrix Operations

If  $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times p}$  are two matrices of identical sizes, then  $\mathbf{A} + \mathbf{B}$  and  $\mathbf{A} - \mathbf{B}$  are understood elementwise, i.e., they result in  $\mathbf{C} \in \mathbb{R}^{n \times p}$  such that  $c_{i,j} = a_{i,j} \pm b_{i,j}$ .

```
A-A
```

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

In R (but not when we use mathematical notation), all other arithmetic, logical and comparison operators are also applied in an elementwise fashion.

```
A*A
```

```
##      [,1] [,2] [,3]
## [1,]     1     4     9
## [2,]    16    25    36
```

```
(A>2) & (A<=5)
```

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

### C.2.4 Matrix Multiplication (\*)

Mathematically,  $\mathbf{AB}$  denotes the **matrix multiplication**. It is a very different operation to the elementwise multiplication.

```
(A <- rbind(c(1, 2), c(3, 4)))
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
(I <- rbind(c(1, 0), c(0, 1)))
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
A %*% I # matrix multiplication
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

This is not the same as the elementwise `A*I`.

Matrix multiplication can only be performed on two matrices of *compatible sizes* – the number of columns in the left matrix must match the number of rows in the right operand.

Given  $\mathbf{A} \in \mathbb{R}^{n \times p}$  and  $\mathbf{B} \in \mathbb{R}^{p \times m}$ , their multiply is a matrix  $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{n \times m}$  such that  $c_{i,j}$  is the dot product of the  $i$ -th row in  $\mathbf{A}$  and the  $j$ -th column in  $\mathbf{B}$ :

$$c_{i,j} = \mathbf{a}_{i,\cdot} \cdot \mathbf{b}_{\cdot,j} = \sum_{k=1}^p a_{i,k} b_{k,j}$$

for  $i = 1, \dots, n$  and  $j = 1, \dots, m$ .

### Exercise.

Multiply a few simple matrices of sizes  $2 \times 2$ ,  $2 \times 3$ ,  $3 \times 2$  etc. using pen and paper and checking the results in R.

□

Also remember that, mathematically, *squaring* a matrix is done in terms of matrix multiplication, i.e.,  $\mathbf{A}^2 = \mathbf{AA}$ . It can only be performed on *square* matrices, i.e., ones with the same number of rows and columns. This is again different than R's elementwise `A^2`.

Note that  $\mathbf{A}^T \mathbf{A}$  gives the matrix that consists of the dot products of all the pairs of columns in  $\mathbf{A}$ .

```
crossprod(A) # same as t(A) %*% A
##      [,1] [,2]
## [1,]    10   14
```

```
## [2,] 14 20
```

In one of the chapters on Regression, we note that the Pearson linear correlation coefficient can be beautifully expressed this way.

### C.2.5 Aggregation of Rows and Columns

The `apply()` function may be used to transform or summarise individual rows or columns in a matrix. More precisely:

- `apply(A, 1, f)` applies a given function  $f$  on each *row* of  $\mathbf{A}$ .
- `apply(A, 2, f)` applies a given function  $f$  on each *column* of  $\mathbf{A}$ .

Usually, either  $f$  returns a single value (when we wish to aggregate all the elements in a row/column) or returns the same number of values (when we wish to transform a row/column). The latter case is covered in the next subsection.

Let's compute the mean of each row and column in  $\mathbf{A}$ :

```
(A <- matrix(1:18, byrow=TRUE, nrow=3))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]     1    2    3    4    5    6
## [2,]     7    8    9   10   11   12
## [3,]    13   14   15   16   17   18
apply(A, 1, mean) # synonym: rowMeans(A)

## [1] 3.5 9.5 15.5
apply(A, 2, mean) # synonym: colMeans(A)
```

```
## [1] 7 8 9 10 11 12
```

We can also fetch the minimal and maximal value by means of the `range()` function:

```
apply(A, 1, range)

##      [,1] [,2] [,3]
## [1,]     1    7   13
## [2,]     6   12   18
apply(A, 2, range)

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]     1    2    3    4    5    6
## [2,]    13   14   15   16   17   18
```

Of course, a custom function can be provided as well. Here we compute the minimum, average and maximum of each row:

```
apply(A, 1, function(row) c(min(row), mean(row), max(row)))

##      [,1] [,2] [,3]
## [1,] 1.0  7.0 13.0
## [2,] 3.5  9.5 15.5
## [3,] 6.0 12.0 18.0
```

### C.2.6 Vectorised Special Functions

The special functions mentioned in the previous chapter, e.g., `sqrt()`, `abs()`, `round()`, `log()`, `exp()`, `cos()`, `sin()`, are also performed in an elementwise manner when applied on a matrix object.

```
round(1/A, 2) # rounds every element in 1/A
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] 1.00 0.50 0.33 0.25 0.20 0.17
## [2,] 0.14 0.12 0.11 0.10 0.09 0.08
## [3,] 0.08 0.07 0.07 0.06 0.06 0.06
```

An example plot of the absolute values of sine and cosine functions depicted using the `matplot()` function (see Figure C.1).

```
x <- seq(-2*pi, 6*pi, by=pi/100)
Y <- cbind(sin(x), cos(x)) # a matrix with two columns
Y <- abs(Y) # take the absolute value of every element in Y
matplot(x, Y, type="l")
```

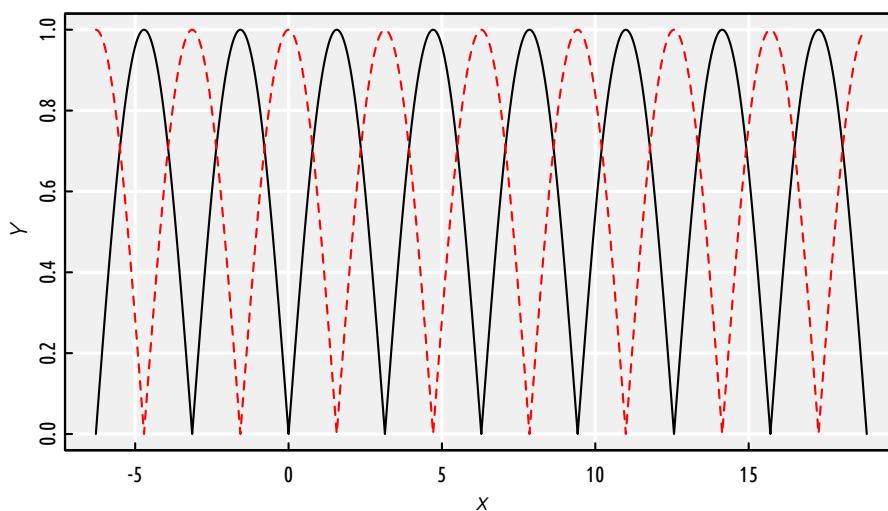


Figure C.1: Example plot with `matplot()`

### C.2.7 Matrix-Vector Operations

Mathematically, there is no generally agreed upon convention defining arithmetic operations between matrices and vectors.

(\*) The only exception is the matrix – vector multiplication in the case where an argument is a column or a row vector, i.e., in fact, a matrix. Hence, given  $\mathbf{A} \in \mathbb{R}^{n \times p}$  we may write  $\mathbf{Ax}$  only if  $\mathbf{x} \in \mathbb{R}^{p \times 1}$  is a column vector. Similarly,  $\mathbf{yA}$  makes only sense whenever  $\mathbf{y} \in \mathbb{R}^{1 \times n}$  is a row vector.

**Remark.** Please take notice of the fact that we consistently discriminate between different bold math fonts and letter cases:  $\mathbf{X}$  is a matrix,  $\mathbf{x}$  is a row or column vector (still a matrix, but a sequence-like one) and  $x$  is an ordinary vector (one-dimensional sequence).

However, in R, we might sometimes wish to vectorise an arithmetic operation between a matrix and a vector in a row- or column-wise fashion. For example, if  $\mathbf{A} \in \mathbb{R}^{n \times p}$  is a matrix and  $\mathbf{m} \in \mathbb{R}^{1 \times p}$  is a row vector, we might want to subtract  $m_i$  from each element in the  $i$ -th column. Here, the `apply()` function comes in handy again.

Example: to create a *centred* version of a given matrix, we need to subtract from each element the arithmetic mean of its column.

```
(A <- cbind(c(1, 2), c(2, 4), c(5, 8)))

##      [,1] [,2] [,3]
## [1,]     1     2     5
## [2,]     2     4     8

(m <- apply(A, 2, mean)) # same as colMeans(A)

## [1] 1.5 3.0 6.5
t(apply(A, 1, function(r) r-m)) # note the transpose here

##      [,1] [,2] [,3]
## [1,] -0.5  -1 -1.5
## [2,]  0.5   1  1.5
```

The above is equivalent to:

```
apply(A, 2, function(c) c-mean(c))

##      [,1] [,2] [,3]
## [1,] -0.5  -1 -1.5
## [2,]  0.5   1  1.5
```

## C.3 Matrix Subsetting

Example matrices:

```
(A <- matrix(1:12, byrow=TRUE, nrow=3))

##      [,1] [,2] [,3] [,4]
## [1,]     1     2     3     4
## [2,]     5     6     7     8
## [3,]     9    10    11    12

B <- A
dimnames(B) <- list(
  c("a", "b", "c"),      # row labels
  c("x", "y", "z", "w") # column labels
)
B

##   x   y   z   w
## a 1  2  3  4
## b 5  6  7  8
## c 9 10 11 12
```

### C.3.1 Selecting Individual Elements

Matrices are two-dimensional structures: items are aligned in rows and columns. Hence, to extract an element from a matrix, we will need two indices. Mathematically, given a matrix  $\mathbf{A}$ ,  $a_{i,j}$  stands for the element in the  $i$ -th row and the  $j$ -th column. The same in R:

```
A[1, 2] # 1st row, 2nd columns

## [1] 2
B["a", "y"] # using dimnames == B[1,2]

## [1] 2
```

### C.3.2 Selecting Rows and Columns

We will sometimes use the following notation to emphasise that a matrix  $\mathbf{A}$  consists of  $n$  rows or  $p$  columns:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_{1,\cdot} \\ \mathbf{a}_{2,\cdot} \\ \vdots \\ \mathbf{a}_{n,\cdot} \end{bmatrix} = \left[ \begin{array}{cccc} \mathbf{a}_{\cdot,1} & \mathbf{a}_{\cdot,2} & \cdots & \mathbf{a}_{\cdot,p} \end{array} \right].$$

Here,  $\mathbf{a}_{i,\cdot}$  is a *row vector* of length  $p$ , i.e., a  $(1 \times p)$ -matrix:

$$\mathbf{a}_{i,\cdot} = [ \ a_{i,1} \ a_{i,2} \ \cdots \ a_{i,p} \ ].$$

Moreover,  $\mathbf{a}_{\cdot,j}$  is a *column vector* of length  $n$ , i.e., an  $(n \times 1)$ -matrix:

$$\mathbf{a}_{\cdot,j} = [ \ a_{1,j} \ a_{2,j} \ \cdots \ a_{n,j} \ ]^T = \begin{bmatrix} a_{1,j} \\ a_{2,j} \\ \vdots \\ a_{n,j} \end{bmatrix},$$

We can extract individual rows and columns from a matrix by using the following notation:

```
A[1,] # 1st row
## [1] 1 2 3 4
A[,2] # 2nd column
## [1] 2 6 10
B["a",] # of course, B[1,] is still legal
## x y z w
## 1 2 3 4
B[, "y"]
## a b c
## 2 6 10
```

Note that by extracting a single row/column, we get an atomic (one-dimensional) vector. However, we can preserve the dimensionality of the output object by passing `drop=FALSE`:

```
A[ 1, , drop=FALSE] # 1st row
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
A[ , 2, drop=FALSE] # 2nd column
##      [,1]
## [1,]    2
## [2,]    6
## [3,]   10
B["a", , drop=FALSE]
## x y z w
## a 1 2 3 4
```

```
B[ , "y", drop=FALSE]
```

```
##      y
## a  2
## b  6
## c 10
```

Now this is what we call proper row and column vectors!

### C.3.3 Selecting Submatrices

To create a sub-block of a given matrix we pass two indexers, possibly of length greater than one:

```
A[1:2, c(1, 2, 4)] # rows 1,2 columns 1,2,4
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    4
## [2,]    5    6    8
B[c("a", "b"), c(1, 2, 4)]
```

```
##   x y w
## a 1 2 4
## b 5 6 8
A[c(1, 3), 3]
```

```
## [1] 3 11
A[c(1, 3), 3, drop=FALSE]
```

```
##      [,1]
## [1,]    3
## [2,]   11
```

### C.3.4 Selecting Based on Logical Vectors and Matrices

We can also subset a matrix with a logical matrix of the same size. This always yields a (flat) vector in return.

```
A[A>8]
```

```
## [1] 9 10 11 12
```

Logical vectors can also be used as indexers:

```
A[c(TRUE, FALSE, TRUE),] # select 1st and 3rd row
```

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

```

## [2,]    9   10   11   12
A[, colMeans(A)>6] # columns with means > 6

##      [,1] [,2]
## [1,]    3    4
## [2,]    7    8
## [3,]   11   12
B[B[, "x"]>1 & B[, "x"]<=9,] # All rows where x is in (1, 9]

##   x   y   z   w
## b 5  6  7  8
## c 9 10 11 12

```

### C.3.5 Selecting Based on Two-Column Matrices

Lastly, note that we can also index a matrix `A` with a 2-column matrix `I`, i.e., `A[I]`. This allows for an easy access to `A[I[1,1], I[1,2]]`, `A[I[2,1], I[2,2]]`, `A[I[3,1], I[3,2]]`, ...

```

I <- cbind(c(1, 3, 2, 1, 2),
            c(2, 3, 2, 1, 4)
)
A[I]

## [1] 2 11 6 1 8

```

This is exactly `A[1, 2]`, `A[3, 3]`, `A[2, 2]`, `A[1, 1]`, `A[2, 4]`.

## C.4 Further Reading

Recommended further reading: (Venables et al. 2020)

Other: (Deisenroth et al. 2020), (Peng 2019), (Wickham & Grolemund 2017)

## Appendix D

# Data Frame Wrangling in R

R `data.frames` are similar to matrices in the sense that we use them to store tabular data. However, in data frames each column can be of different type:

```
head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa

head(rpart::car90, 2)

##                   Country Disp Disp2 Eng.Rev Front.Hd Frt.Leg.Room
## Acura Integra     Japan  112    1.8   2935      3.5        41.5
## Acura Legend      Japan  163    2.7   2505      2.0        41.5
##                   Frt.Shld Gear.Ratio Gear2 HP HP.revs Height
## Acura Integra     53.0     3.26  3.21 130    6000     47.5
## Acura Legend      55.5     2.95  3.02 160    5900     50.0
##                   Length Luggage Mileage Model2 Price Rear.Hd
## Acura Integra     177      16     NA    11950     1.5
## Acura Legend      191      14     20    24760     2.0
##                   Rear.Seating RearShld Reliability Rim Sratio.m
## Acura Integra      26.5     52.0 Much better R14        NA
## Acura Legend       28.5     55.5 Much better R15        NA
##                   Sratio.p Steering Tank Tires Trans1 Trans2
## Acura Integra     0.86    power 13.2 195/60 man.5 auto.4
## Acura Legend      0.96    power 18.0 205/60 man.5 auto.4
##                   Turning Type Weight Wheel.base Width
```

```
## Acura Integra      37  Small   2700      102     67
## Acura Legend      42 Medium   3265      109     69
```

## D.1 Creating Data Frames

Most frequently, we will be creating data frames based on a series of numeric, logical, characters vectors of identical lengths.

```
x <- data.frame(
  u=runif(5),
  v=sample(c(TRUE, FALSE), 5, replace=TRUE),
  w=LETTERS[1:5]
)
print(x)

##           u      v w
## 1 0.1815171 TRUE A
## 2 0.9197226 FALSE B
## 3 0.3117235 FALSE C
## 4 0.0641516 TRUE D
## 5 0.3964216 FALSE E
```

Note that when we create objects of type data frame, strings are automatically converted to factors.

```
class(x$w)

## [1] "factor"
```

Throughout the history of computing with R, this has caused way too many bugs (recall, for instance, what's the result of calling `as.numeric()` on a factor). In order to change this behaviour, either pass `stringsAsFactors=FALSE` argument to `data.frame()` or switch this feature off globally (recommended):

```
options(stringsAsFactors=FALSE)
```

Some objects, such as matrices, can easily be coerced to data frames:

```
(A <- matrix(1:12, byrow=TRUE, nrow=3,
  dimnames=list(
    NULL,       # row labels
    c("x", "y", "z", "w") # column labels
  )))

##      x  y  z  w
## [1,] 1  2  3  4
## [2,] 5  6  7  8
## [3,] 9 10 11 12
```

```

as.data.frame(A)

##   x  y  z  w
## 1 1  2  3  4
## 2 5  6  7  8
## 3 9 10 11 12

Named lists are amongst other candidates for a meaningful conversion:
(l <- lapply(split(iris$Sepal.Length, iris$Species),
  function(x) {
    c(min=min(x), median=median(x), mean=mean(x), max=max(x))
 }))

## $setosa
##   min median   mean   max
## 4.300 5.000 5.006 5.800
##
## $versicolor
##   min median   mean   max
## 4.900 5.900 5.936 7.000
##
## $virginica
##   min median   mean   max
## 4.900 6.500 6.588 7.900

as.data.frame(l)

##      setosa versicolor virginica
## min    4.300     4.900     4.900
## median 5.000     5.900     6.500
## mean   5.006     5.936     6.588
## max    5.800     7.000     7.900

```

## D.2 Importing Data Frames

Many interesting data frames come from external sources, such as csv files, web APIs, SQL databases and so on.

In particular, `read.csv()` (see `?read.table` for a long list of tunable parameters) imports data from plain text files organised in a tabular manner (such as comma-separated lists of values):

```

f <- tempfile() # temporary file name
write.csv(x, f, row.names=FALSE) # save data frame to file
cat(readLines(f), sep="\n") # print file contents

## "u","v","w"

```

```

## 0.181517061544582,TRUE,"A"
## 0.919722604798153, FALSE,"B"
## 0.31172346835956, FALSE,"C"
## 0.0641516039613634, TRUE,"D"
## 0.396421572659165, FALSE,"E"

read.csv(f)

##          u      v w
## 1 0.1815171 TRUE A
## 2 0.9197226 FALSE B
## 3 0.3117235 FALSE C
## 4 0.0641516 TRUE D
## 5 0.3964216 FALSE E

```

Note that CSV is by far the most portable format for exchanging matrix-like objects between different programs (statistical or numeric computing environments, spreadsheets etc.).

## D.3 Data Frame Subsetting

### D.3.1 Each Data Frame is a List

First of all, we should note that each data frame is in fact represented as an ordinary named list:

```

class(x)

## [1] "data.frame"

typeof(x)

## [1] "list"

length(x) # number of columns

## [1] 3

names(x) # column labels

## [1] "u" "v" "w"

x$u # accessing column `u` (synonym: x[["u"]])

## [1] 0.1815171 0.9197226 0.3117235 0.0641516 0.3964216

```

```
x[[2]] # 2nd column
## [1] TRUE FALSE FALSE TRUE FALSE
x[c(1,3)] # a sub-data.frame

##          u   w
## 1 0.1815171 A
## 2 0.9197226 B
## 3 0.3117235 C
## 4 0.0641516 D
## 5 0.3964216 E
sapply(x, class) # apply class() on each column

##          u           v           w
## "numeric" "logical" "factor"
```

### D.3.2 Each Data Frame is Matrix-like

Data frames can be considered as “generalised” matrices. Therefore, operations such as subsetting will work in the same manner.

```
dim(x) # number of rows and columns
## [1] 5 3
x[1:2,] # first two rows

##          u   v   w
## 1 0.1815171 TRUE A
## 2 0.9197226 FALSE B
x[,c(1,3)] # 1st and 3rd column, synonym: x[c(1,3)]

##          u   w
## 1 0.1815171 A
## 2 0.9197226 B
## 3 0.3117235 C
## 4 0.0641516 D
## 5 0.3964216 E
x[,1] # synonym: x[[1]]

## [1] 0.1815171 0.9197226 0.3117235 0.0641516 0.3964216
x[,1,drop=FALSE] # synonym: x[1]

##          u
## 1 0.1815171
## 2 0.9197226
```

```
## 3 0.3117235
## 4 0.0641516
## 5 0.3964216
```

Take a special note of selecting rows based on logical vectors. For instance, let's extract all the rows from `x` where the values in the column named `u` are between 0.3 and 0.6:

```
x[x$u>=0.3 & x$u<=0.6, ]
```

```
##           u      v w
## 3 0.3117235 FALSE C
## 5 0.3964216 FALSE E
```

```
x[!(x[, "u"]<0.3 | x[, "u"]>0.6), ] # equivalent
```

```
##           u      v w
## 3 0.3117235 FALSE C
## 5 0.3964216 FALSE E
```

Moreover, subsetting based on integer vectors can be used to change the order of rows. Here is how we can sort the rows in `x` with respect to the values in column `u`:

```
(x_sorted <- x[order(x$u), ])
```

```
##           u      v w
## 4 0.0641516 TRUE D
## 1 0.1815171 TRUE A
## 3 0.3117235 FALSE C
## 5 0.3964216 FALSE E
## 2 0.9197226 FALSE B
```

Let's stress that the programming style we emphasise on here is very transparent. If we don't understand how a complex operation is being executed, we can always decompose it into smaller chunks that can be studied separately. For instance, as far as the last example is concerned, we can take a look at the manual of `?order` and then inspect the result of calling `order(x$u)`.

On a side note, we can re-set the row names by referring to:

```
row.names(x_sorted) <- NULL
x_sorted
```

```
##           u      v w
## 1 0.0641516 TRUE D
## 2 0.1815171 TRUE A
## 3 0.3117235 FALSE C
## 4 0.3964216 FALSE E
## 5 0.9197226 FALSE B
```

## D.4 Common Operations

We already know how to filter rows based on logical conditions, e.g.:

```
iris[iris$Petal.Width >= 1.2 & iris$Petal.Width <= 1.3,
  c("Petal.Width", "Species")]

##      Petal.Width   Species
## 54          1.3 versicolor
## 56          1.3 versicolor
## 59          1.3 versicolor
## 65          1.3 versicolor
## 72          1.3 versicolor
## 74          1.2 versicolor
## 75          1.3 versicolor
## 83          1.2 versicolor
## 88          1.3 versicolor
## 89          1.3 versicolor
## 90          1.3 versicolor
## 91          1.2 versicolor
## 93          1.2 versicolor
## 95          1.3 versicolor
## 96          1.2 versicolor
## 97          1.3 versicolor
## 98          1.3 versicolor
## 100         1.3 versicolor

iris[iris$Sepal.Length > 6.5 & iris$Species == "versicolor", ]

##      Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 51          7.0        3.2       4.7        1.4 versicolor
## 53          6.9        3.1       4.9        1.5 versicolor
## 59          6.6        2.9       4.6        1.3 versicolor
## 66          6.7        3.1       4.4        1.4 versicolor
## 76          6.6        3.0       4.4        1.4 versicolor
## 77          6.8        2.8       4.8        1.4 versicolor
## 78          6.7        3.0       5.0        1.7 versicolor
## 87          6.7        3.1       4.7        1.5 versicolor
```

and aggregate information in individual columns:

```
sapply(iris[1:4], summary)

##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## Min.       4.300000  2.000000    1.000     0.100000
## 1st Qu.    5.100000  2.800000    1.600     0.300000
## Median    5.800000  3.000000    4.350     1.300000
## Mean      5.843333  3.057333    3.758     1.199333
```

```
## 3rd Qu.      6.400000   3.300000   5.100   1.800000
## Max.       7.900000   4.400000   6.900   2.500000
```

Quite frequently, we will be interested in summarising data within subgroups generated by a list of factor-like variables.

```
aggregate(iris[1:4], iris[5], mean)

##          Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1         setosa      5.006     3.428      1.462     0.246
## 2 versicolor      5.936     2.770      4.260     1.326
## 3 virginica       6.588     2.974      5.552     2.026
ToothGrowth[sample(nrow(ToothGrowth), 5), ] # 5 random rows

##      len supp dose
## 12 16.5  VC  1.0
## 10  7.0  VC  0.5
## 17 13.6  VC  1.0
## 50 27.3  OJ  1.0
## 60 23.0  OJ  2.0

aggregate(ToothGrowth["len"], ToothGrowth[c("supp", "dose")], median)

##    supp dose   len
## 1   OJ  0.5 12.25
## 2   VC  0.5  7.15
## 3   OJ  1.0 23.45
## 4   VC  1.0 16.50
## 5   OJ  2.0 25.95
## 6   VC  2.0 25.95
```

Taking into account that `split()` accepts a data frame input as well, we can perform what follows:

```
sapply(
  # split iris into 3 sub-data.frames:
  split(iris, iris[5]),
  # on each sub-data.frame, apply the following function
  function(df) {
    # compute the mean of first four columns:
    sapply(df[1:4], mean)
  })

##           setosa versicolor virginica
## Sepal.Length 5.006     5.936     6.588
## Sepal.Width  3.428     2.770     2.974
## Petal.Length 1.462     4.260     5.552
## Petal.Width  0.246     1.326     2.026
```

```

sapply(split(iris, iris[5]), function(df) {
  c(Sepal.Length=summary(iris$Sepal.Length),
    Petal.Length=summary(iris$Petal.Length)
  )
})

##           setosa versicolor virginica
## Sepal.Length.Min. 4.300000 4.300000 4.300000
## Sepal.Length.1st Qu. 5.100000 5.100000 5.100000
## Sepal.Length.Median 5.800000 5.800000 5.800000
## Sepal.Length.Mean 5.843333 5.843333 5.843333
## Sepal.Length.3rd Qu. 6.400000 6.400000 6.400000
## Sepal.Length.Max. 7.900000 7.900000 7.900000
## Petal.Length.Min. 1.000000 1.000000 1.000000
## Petal.Length.1st Qu. 1.600000 1.600000 1.600000
## Petal.Length.Median 4.350000 4.350000 4.350000
## Petal.Length.Mean 3.758000 3.758000 3.758000
## Petal.Length.3rd Qu. 5.100000 5.100000 5.100000
## Petal.Length.Max. 6.900000 6.900000 6.900000

```

The above syntax is not super-convenient, but it only uses the building blocks that we have already mastered! That should be very appealing to the minimalists. Note that R packages such as `data.table` and `dplyr` offer more convenient substitutes – you can always learn them on your own (which takes time, but it’s worth the hassle). They simplify the most common data wrangling tasks. Moreover, they have been optimised for speed – they can handle much larger data sets efficiently.

## D.5 Metaprogramming and Formulas (\*)

R (together with a few other programming languages such as Lisp and Scheme, that heavily inspired R’s semantics) allows its programmers to apply some *metaprogramming* techniques, that is, to write programs that manipulate unevaluated R expressions.

For instance, take a close look at the following plot:

```

z <- seq(-2*pi, 2*pi, length.out=101)
plot(z, sin(z), type="l")

```

How did the `plot()` function know that we are plotting `sin` of `z` (see Figure D.1)? It turns out that, at any time, we not only have access to the value of an object (such as the result of evaluating `sin(z)`, which is a vector of 101 reals) but also to the expression that was passed as a function’s argument itself.

```

test_meta <- function(x) {
  cat("x equals to ", x, "\n") # \n == newline
}

```

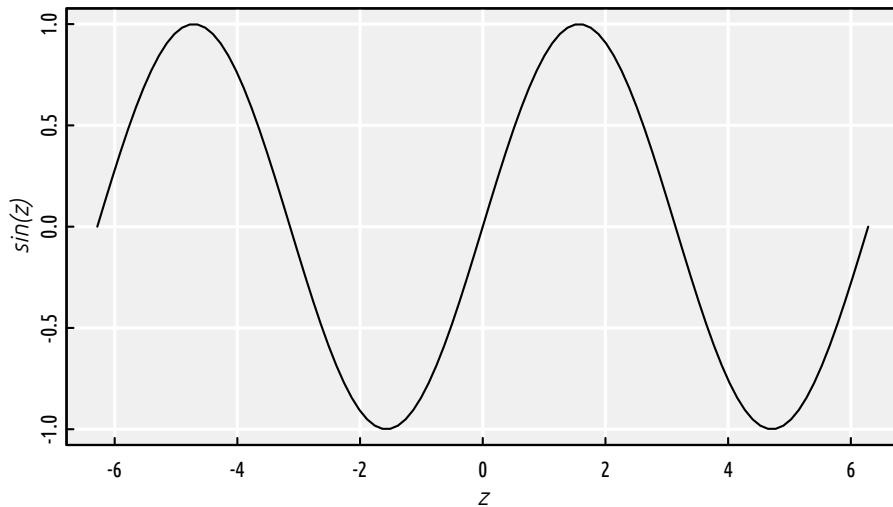


Figure D.1: Metaprogramming in action: Just take a look at the Y axis label

```

cat("x stemmed from ", deparse(substitute(x)), "\n")
}
test_meta(2+7)

## x equals to 9
## x stemmed from 2 + 7

```

This is very powerful and yet potentially very confusing to the users, because we can write functions that don't compute the arguments provided in a way we expect them to (i.e., following the R language specification). Each function can constitute a new micro-verse, where with its own rules – we should always refer to the documentation.

For instance, consider the `subset()` function:

```

head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5         1.4         0.2  setosa
## 2          4.9         3.0         1.4         0.2  setosa
## 3          4.7         3.2         1.3         0.2  setosa
## 4          4.6         3.1         1.5         0.2  setosa
## 5          5.0         3.6         1.4         0.2  setosa
## 6          5.4         3.9         1.7         0.4  setosa
subset(iris, Sepal.Length>7.5, select=-Sepal.Width-Petal.Width))

```

```
##      Sepal.Length  Species
## 106          7.6 virginica
## 118          7.7 virginica
## 119          7.7 virginica
## 123          7.7 virginica
## 132          7.9 virginica
## 136          7.7 virginica
```

Neither `Sepal.Length>6` nor `-(Sepal.Width:Petal.Width)` make sense as standalone R expressions! However, according to the `subset()` function's own rules, the former expression is considered as a row selector (here, `Sepal.Length` refers to a particular column *within* the `iris` data frame). The latter plays the role of a column filter (select everything but all the columns between...).

The `data.table` and `dplyr` packages (which are very popular) rely on this language feature all the time, so we shouldn't be surprised when we see them.

There is one more interesting language feature that is possible thanks to metaprogramming. *Formulas* are special R objects that consist of two unevaluated R expressions separated by a tilde (~). For example:

```
len ~ supp+dose
```

```
## len ~ supp + dose
```

A formula on its own has no meaning. However, many R functions accept formulas as arguments and can interpret them in various different ways.

For example, the `lm()` function that fits a linear regression model, uses formulas to specify the output and input variables:

```
lm(Sepal.Length~Petal.Length+Sepal.Width, data=iris)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Petal.Length + Sepal.Width, data = iris)
##
## Coefficients:
## (Intercept)  Petal.Length  Sepal.Width
##           2.2491        0.4719        0.5955
```

On the other hand, `boxplot()` (see Figure D.2) allows for creating separate box-and-whisker plots for each subgroup given by a combination of factors.

```
boxplot(len~supp+dose, data=ToothGrowth,
        horizontal=TRUE, col="white")
```

The `aggregate()` function supports formulas too:

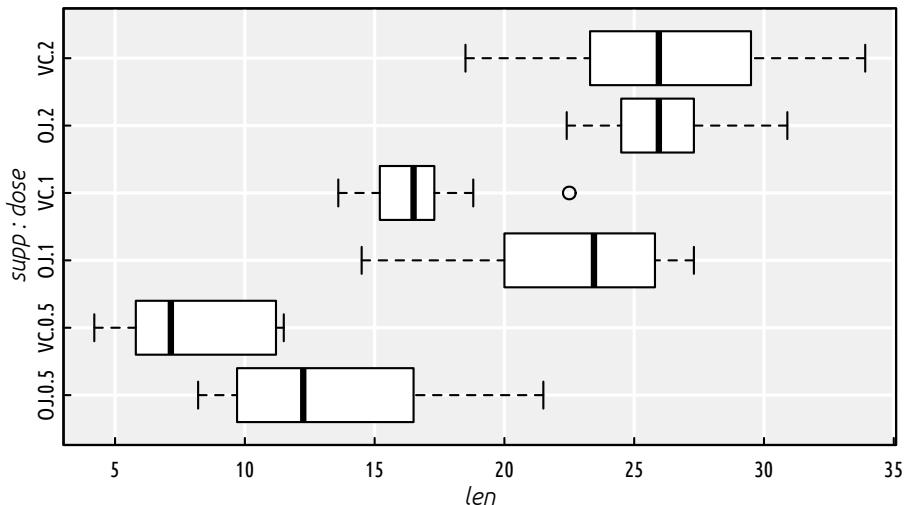


Figure D.2: Example box plot created via the formula interface

```
aggregate(cbind(Sepal.Length, Sepal.Width) ~ Species, data=iris, mean)

##      Species Sepal.Length Sepal.Width
## 1      setosa     5.006     3.428
## 2 versicolor     5.936     2.770
## 3  virginica     6.588     2.974
```

We should therefore make sure that we know how every function interacts with a formula – information on that can be found in `?lm`, `?boxplot`, `?aggregate` and so forth.

## D.6 Further Reading

Recommended further reading: (Venables et al. 2020)

Other: (Peng 2019), (Wickham & Grolemund 2017)

R packages `dplyr` and `data.table` implement the most common data frame wrangling procedures. You may find them very useful. Moreover, they are very fast even for large data sets. Additionally, the `magrittr` package provides a pipe operator, `%>%`, that simplifies the writing of complex, nested function calls. Do note that not everyone is a big fan of these, however.

# References

- Bezdek JC, Ehrlich R, Full W (1984) FCM: The fuzzy c-means clustering algorithm. *Computer and Geosciences* 10, 191–203.
- Bishop C (2006) *Pattern recognition and machine learning*. Springer-Verlag <https://www.microsoft.com/en-us/research/people/cmbishop/>.
- Boyd S, Vandenberghe L (2004) *Convex optimization*. Cambridge University Press [https://web.stanford.edu/~boyd/cvxbook/bv\\_cvxbook.pdf](https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf).
- Breiman L, Friedman J, Stone CJ, Olshen RA (1984) *Classification and regression trees*. Chapman; Hall/CRC.
- Campello RJGB, Moulavi D, Zimek A, Sander J (2015) Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Transactions on Knowledge Discovery from Data* 10, 5:1–5:51.
- Cena A, Gagolewski M (2020) Genie+OWA: Robustifying hierarchical clustering with OWA-based linkages. *Information Sciences* in press, doi:10.1016/j.ins.2020.02.025.
- Cortez P, Cerdeira A, Almeida F, Matos T, Reis J (2009) Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems* 47, 547–553.
- Deisenroth MP, Faisal AA, Ong CS (2020) *Mathematics for machine learning*. Cambridge University Press <https://mml-book.com/>.
- Ester M, Kriegel H-P, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise *Proc. KDD'96*, pp. 226–231.
- Fletcher R (2008) *Practical methods of optimization*. Wiley.
- Gagolewski M, Bartoszuk M, Cena A (2016) Genie: A new, fast, and outlier-resistant hierarchical clustering algorithm. *Information Sciences* 363, 8–23.
- Goldberg DE (1989) *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley.

- Goodfellow I, Bengio Y, Courville A (2016) *Deep learning*. MIT Press <https://www.deeplearningbook.org/>.
- Harper FM, Konstan JA (2015) The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems* 5, 19:1–19:19.
- Hastie T, Tibshirani R, Friedman J (2017) *The elements of statistical learning*. Springer-Verlag <https://web.stanford.edu/~hastie/ElemStatLearn/>.
- Herlocker JL, Konstan JA, Terveen LG, Riedl JT (2004) Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems* 22, 5–53. [https://web.archive.org/web/20070306161407/http://web.engr.oregonstate.edu/~herlock/papers/eval\\_tois.pdf](https://web.archive.org/web/20070306161407/http://web.engr.oregonstate.edu/~herlock/papers/eval_tois.pdf).
- Hubert L, Arabie P (1985) Comparing partitions. *Journal of Classification* 2, 193–218.
- James G, Witten D, Hastie T, Tibshirani R (2017) *An introduction to statistical learning with applications in R*. Springer-Verlag <http://faculty.marshall.usc.edu/gareth-james/ISL/>.
- Koren Y (2009) *The BellKor solution to the Netflix grand prize*. [https://netflixprize.com/assets/GrandPrize2009\\_BPC\\_BellKor.pdf](https://netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf).
- Ling RF (1973) A probability theory of cluster analysis. *Journal of the American Statistical Association* 68, 159–164.
- Lü L, others (2012) Recommender systems. *Physics Reports* 519, 1–49. <https://arxiv.org/pdf/1202.1112.pdf>.
- Müller AC, Nowozin S, Lampert CH (2012) Information theoretic clustering using minimum spanning trees *Proc. German conference on pattern recognition*, <https://github.com/amueller/information-theoretic-mst>.
- Ng AY, Jordan MI, Weiss Y (2001) On spectral clustering: Analysis and an algorithm *Proc. Advances in neural information processing systems 14 (NIPS'01)*, <https://papers.nips.cc/paper/2092-on-spectral-clustering-analysis-and-an-algorithm.pdf>.
- Nocedal J, Wright SJ (2006) *Numerical optimization*. Springer.
- Peng RD (2019) *R programming for data science*. <https://bookdown.org/rdpeng/rprogdatascience/>.
- Piotte M, Chabbert M (2009) *The Pragmatic Theory solution to the Netflix grand prize*. [https://netflixprize.com/assets/GrandPrize2009\\_BPC\\_PragmaticTheory.pdf](https://netflixprize.com/assets/GrandPrize2009_BPC_PragmaticTheory.pdf).
- Quinlan R (1986) Induction of decision trees. *Machine Learning* 1, 81–106.
- Quinlan R (1993) *C4.5: Programs for machine learning*. Morgan Kaufmann Publishers.

- R Development Core Team (2020) *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria <http://www.R-project.org>.
- Ricci F, Rokach L, Shapira B, Kantor P (eds) (2011) *Recommender systems handbook*. Springer <http://www.inf.unibz.it/~ricci/papers/intro-rec-sys-handbook.pdf>.
- Sarle WS, others (eds) (2002) The comp.ai.neural-nets FAQ. <http://www.faqs.org/faqs/ai-faq/neural-nets/part1/>.
- Simon D (2013) *Evolutionary optimization algorithms: Biologically-inspired and population-based approaches to computer intelligence*. Wiley.
- Therneau TM, Atkinson EJ (2019) *An introduction to recursive partitioning using the RPART routines*. <https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf>.
- Tötscher A, Jährer M, Bell RM (2009) *The BigChaos solution to the Netflix grand prize*. [https://netflixprize.com/assets/GrandPrize2009\\_BPC\\_BigChaos.pdf](https://netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf).
- Venables WN, Smith DM, R Core Team (2020) *An introduction to R*. <https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf>.
- Wickham H, Grolemund G (2017) *R for data science*. O'Reilly <https://r4ds.had.co.nz/>.
- Zhang T, Ramakrishnan R, Livny M (1996) BIRCH: An efficient data clustering method for large databases *Proc. ACM SIGMOD international conference on management of data – SIGMOD'96*, pp. 103–114.