Lightweight Machine Learning Classics with R

Marek Gagolewski

DRAFT v0.1 2020-03-01 14:47 (e066e08)

Contents

{

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!

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 a deep dive into some of the most fundamental algorithms which stood the test of time and which still form the basis for the state-of-the-art solutions of the modern-era 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, (Venables, Smith, and the R Core Team 2020; Peng 2019; Wickham and Grolemund 2017).

We will provide solid underpinnings for further studies related to statistical learning, machine learning data science, data analytics and artificial intelligence, including (James et al. 2017; Hastie, Tibshirani, and Friedman 2017; Bishop 2006). We will appreciate the vital role of mathematics as a commonly accepted

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 enough, introduction to the topic. For a nice, machine learning-focused introduction to mathematics alone, see, e.g., (Deisenroth, Faisal, and Ong 2020).

About Me

I'm currently a Senior Lecturer in Applied AI at Deakin University in Melbourne, Australia and an Associate Professor in Data Science at Warsaw University of Technology, Poland, where I teach various courses related to R and Python programming, algorithms, data science and machine learning. This book was also influenced by my teaching experience at Data Science Retreat in Berlin, Germany.

I'm an author of several R and Python packages, including stringi, which is among the top 20 most often downloaded R extensions. I'm an author of more than 70 publications, my research interests include machine learning and optimisation algorithms, data aggregation and clustering, statistical modelling and scientific computing.

Acknowledgements

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

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

During the writing of this book, I've been mostly 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 new knowledge will be acquired. 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)
- **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 post-factum feedback on the decision made (e.g., learn to play The Witcher 7)

1.2 Supervised Learning

1.2.1 Formalism

Let $\mathbf{X} = \{\mathfrak{X}^{(1)}, \ldots, \mathfrak{X}^{(n)}\}$ 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.).

. . .

Of course, this setting is *abstract* in the way that there might be different realities hidden behind those 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 – let's spend a moment contemplating how many real-world situations fit into this framework.

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

> How would you represent a patient in a clinic?
>
> How would you represent a car in an insurance company's database?
>
> How would you represent a student in an university?

But, e.g., 1920×1080 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 $\mathbf{X} \in \mathbb{R}^{n \times d}$.

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

Example: The famous Fisher's Iris flower dataset, see `?iris` in R and 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, $j = 1, \ldots, p$, $i = 1, \ldots, n$.

For instance:

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

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

The third observation (data point, row in $\mathbf{X}$) consists of items $(x_{3,1}, \ldots, x_{3,p})$ 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
```

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

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

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

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

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

```r
X[,2]
```

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

```r
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:

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

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

$$\mathbf{x}_{i,\cdot} = \begin{bmatrix} x_{i,1} & x_{i,2} & \cdots & x_{i,p} \end{bmatrix}.$$

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

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

where $\cdot^T$ 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 to the desired one as possible. The given data sample will be used to "train" this "model".

Usually the reference outputs are encoded as single 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 $[\mathbf{X}\ \mathbf{y}]$ – where each object is represented as a row vector $[\mathbf{x}_{i,\cdot}\ y_i]$, $i = 1, \ldots, n$:

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

where

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

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., $y_i = 0$/healthy or 1/ill with $\mathbf{x}_{i,\cdot}$ = a patient's health data

- **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

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?

- Email spam detection
- Market stock price prediction
- Predict the likeability of a new ad
- Credit risk assessment
- Detect existence of 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 collisions with pedestrians

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

Examples – the `wines` dataset:

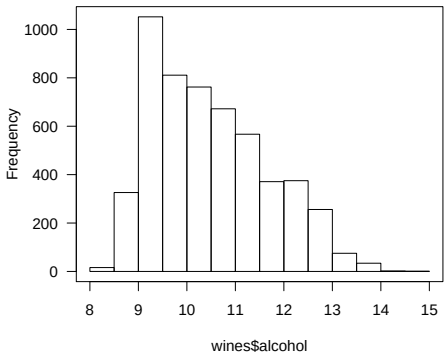
```
wines <- read.csv("datasets/winequality-all.csv", comment="#")
wines[1,]
```

```
##   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        3   red
```

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

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    8.00    9.50   10.40   10.55   11.40   14.90
```

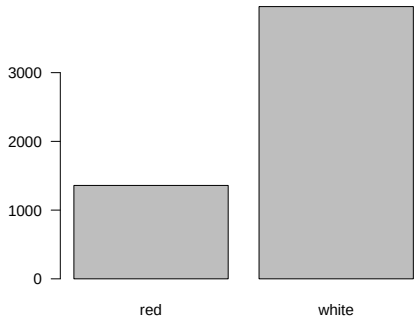
```
hist(wines$alcohol, las=1, main=""); box()
```




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

```
##
##   red white
##  1359  3961
```

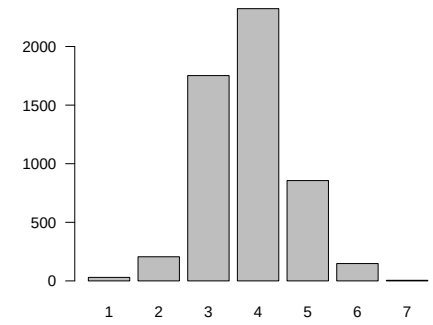
```
barplot(table(wines$color), las=1)
```




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

```
##
##    1    2    3    4    5    6    7
##   30  206 1752 2323  856  148    5
```

```
barplot(table(wines$response), las=1)
```



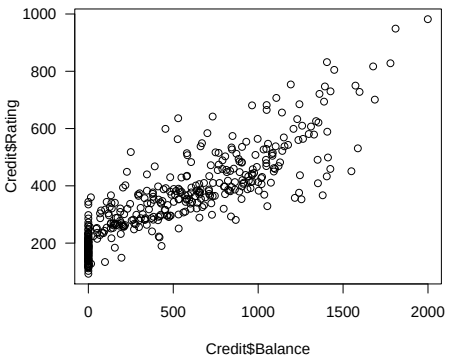
1.3 Simple Regression

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 the concepts graphically.

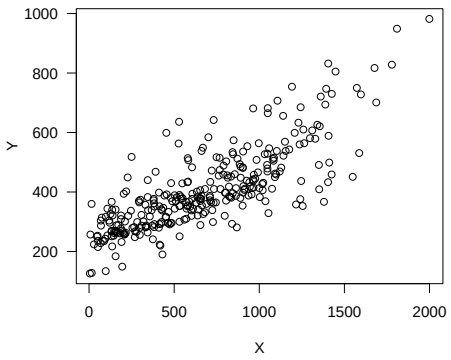
```
library("ISLR") # Credit dataset
plot(Credit$Balance, Credit$Rating, las=1) # scatter plot
```



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.

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

```
plot(X, Y, las=1)
```



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:
  - 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., $E = mc^2$
  or $i\hbar\frac{d}{dt}|\Psi(t)\rangle = \hat{H}|\Psi(t)\rangle$

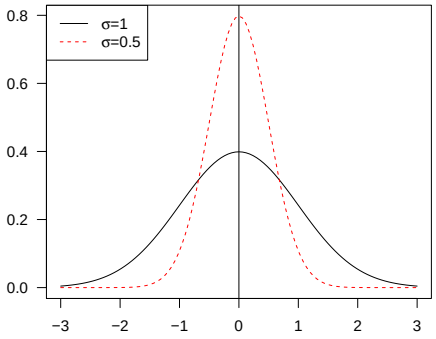
Mathematically, we will assume that there is some "true" function that models data (true relationship between Y and X), but the observed outputs are subject to **additive error**:

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

ε 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.

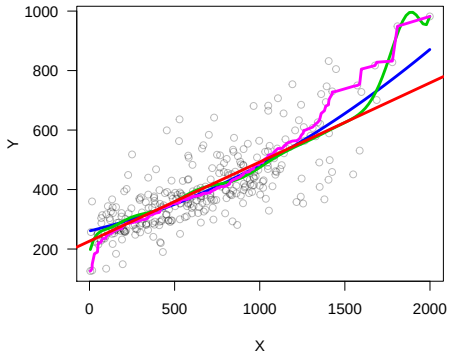
We denote this as $\varepsilon \sim \mathcal{N}(0, \sigma)$ (read: random variable ε follows a normal distribution with expect value of 0 and standard deviation of σ for some $\sigma \geq 0$).

σ controls the amount of noise (and hence, uncertainty). Here is the plot of the probability distribution function (PDFs, densities) of $\mathcal{N}(0, \sigma)$ for different σs:



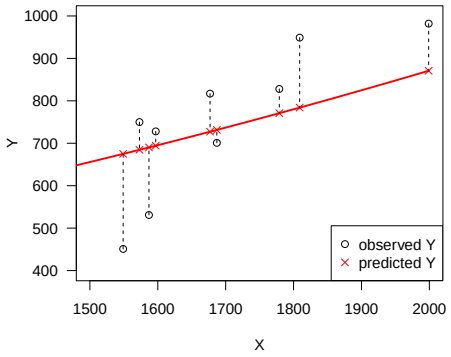
1.3.2 Search Space and Objective

There are many different functions that can be **fitted** into the observed $(\mathbf{x}, \mathbf{y})$



Thus, we need a **model selection criterion**.

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 $\hat{y}_i - y_i$.



Top choice: sum of squared residuals:

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

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 $z_1 + z_2 + \cdots + z_n$.

The notation $\mathrm{SSR}(f|\mathbf{x},\mathbf{y})$ means that it is the error measure corresponding to the model (f) *given* our data.
We could've denoted it with $\mathrm{SSR}_{\mathbf{x},\mathbf{y}}(f)$ or even $\mathrm{SSR}(f)$ to emphasise that $\mathbf{x},\mathbf{y}$ are just fixed values and we are not interested in changing them at all.

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.
- $\mathcal{F} = \{x \mapsto ax + b\}$ – 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**.

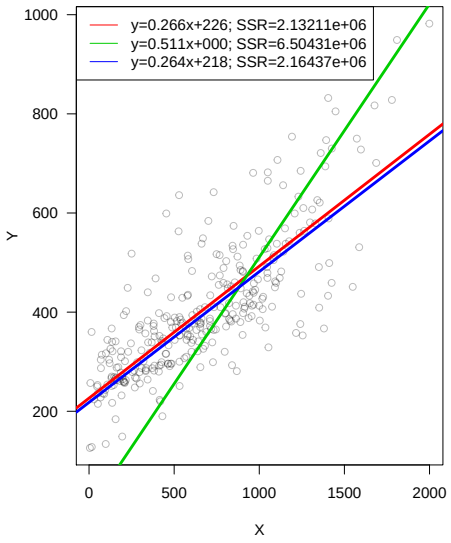
Our problem becomes:

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

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.



Which of these is the least squares solution?

1.4.2 Solution in R

Let's fit the linear model minimising the SSR in R. For convenience, let us store both $\mathbf{x}$ and $\mathbf{y}$ in a data frame:

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

```
##     X   Y
## 1 333 283
## 2 903 483
## 3 580 514
```

The `lm()` function (`l`inear `m`odels) has a convenient *formula*-based interface.

> 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.

```
f <- lm(Y~X, data=XY)
```

> Note that here `X` and `Y` refer to column names in the `XY` data frame.

```
print(f)
```

```
##
## Call:
```

```
## lm(formula = Y ~ X, data = XY)
##
## Coefficients:
## (Intercept)            X
##    226.4711       0.2661
```

Hence, the fitted model is:

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

Coefficient a (slope):

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

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

Coefficient b (intercept):

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

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

SSR:

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

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

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

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

To make a prediction:

```
Xnew <- data.frame(X=c(1500, 2000, 2500))
f$coefficient[2]*Xnew$X + f$coefficient[1]
```

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

or:

```
predict(f, Xnew)
```

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

However:

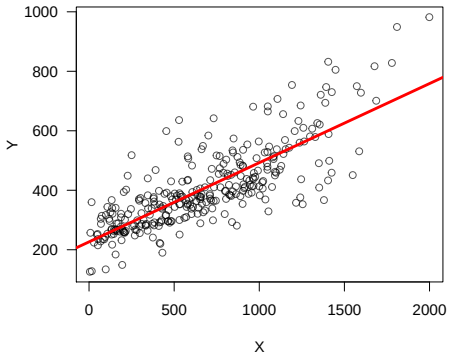
```
predict(f, data.frame(X=c(5000)))
```

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

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

Plotting:

```
plot(X, Y, col="#000000aa", las=1)
abline(f, col=2, lwd=3)
```



Note that our $Y = aX + b$ 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 Derivation of the Solution (**)

> (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:

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

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., $E'_a(a^*, b^*) = 0$ and $E'_b(a^*, b^*) = 0$.

We have:

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

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 $\frac{\partial E}{\partial b}(a,b) = 0$.

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

This is a system of 2 linear equations. Easy.

Rearranging like back in the school days:

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

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

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

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

Sanity check:

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

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

(**) 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:

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

```
##           [,1]
## [1,] 0.2661459
```

```
(b <- mean(Y)-a*mean(X))
```

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

1.5 Outro

1.5.1 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.

1.5.2 Further Reading

Recommended further reading:

- (James et al. 2017: Chapters 1, 2 and 3)

Other:

- (Hastie, Tibshirani, and Friedman 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 $y_i \in \mathbb{R}$, i.e., it is a real number.

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

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

2.1.2 Simple Linear Regression - Recap

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

We restricted ourselves to linear models that minimised the sum of squared residuals, i.e.,

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

The solution is:

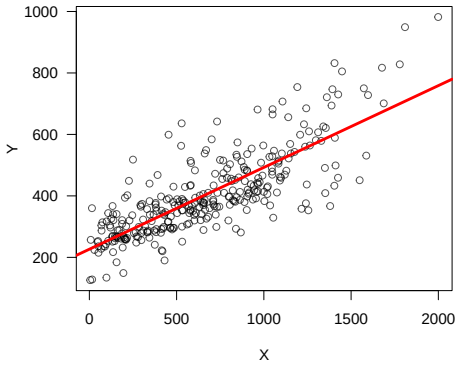
$$\begin{cases} a^* = \dfrac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} y_i \sum_{i=1}^{n} x_i}{n\sum_{i=1}^{n} x_i x_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} x_i} \\ b^* = \dfrac{1}{n}\sum_{i=1}^{n} y_i - a^* \dfrac{1}{n}\sum_{i=1}^{n} x_i \end{cases}$$

Fitting in R:

```
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
```

```
plot(X, Y, col="#000000aa", las=1)
abline(f, col=2, lwd=3)
```



2.2 Multiple Linear Regression

2.2.1 Problem Formulation

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

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

$$Y = f(X_1, \ldots, X_p) \qquad (+\varepsilon)$$

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

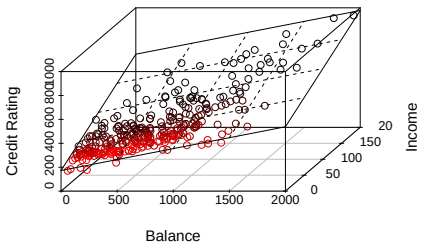
$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p.$$

Above we studied the case where $p = 1$ with $\beta_1 = a$ and $\beta_0 = b$.

The above equation defines:

- $p = 1$ — a line
- $p = 2$ — a plane
- $p \geq 3$ — a hyperplane

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`.

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

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

```
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 # ß0, ß1, ß2
```

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

By the way, the above 3D scatter plot was generated by calling:

```
par(mar=c(4, 4, 0.5, 0.5))
library("scatterplot3d")
s3d <- scatterplot3d(X1, X2, Y,
  angle=60, # change angle to reveal more
  highlight.3d=TRUE, xlab="Balance", ylab="Income",
  zlab="Credit Rating", las=1)
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

Consider the three following models.

Formula	Equation
Rating ~ Balance + Income	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$
Rating ~ Balance	$Y = aX_1 + b$ $(\beta_0 = b, \beta_1 = a, \beta_2 = 0)$
Rating ~ Income	$Y = aX_2 + b$ $(\beta_0 = b, \beta_1 = 0, \beta_2 = a)$

```
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
```

```
## (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?

"Best" — with respect to what kind of measure?

So far we were fitting w.r.t. SSR, as the multiple regression model generalise the two simple ones, the former must yield a not-worse SSR.

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

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

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

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

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

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

We get that $f_{12} \succeq f_2 \succeq f_1$ but these error values are meaningless.

> Interpretability in ML has always been an important issue, think the EU GDPR, amongst others.

The quality of fit can be assessed by performing some descriptive statistical analysis of the residuals, $\hat{y}_i - y_i$.

Interestingly, the mean of residuals (this can be shown analytically) in the least squared fit is always equal to 0:

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

> (*) 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] -2.086704e-16
```

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

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

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 == sum / 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
```

However, if original Ys are, say, in metres [m], MSR is expressed in metres squared [m^2].

Root mean squared error (RMSE):

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

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

```
## [1] 33.99528
```

```
sqrt(mean(f1$residuals^2))
```

```
## [1] 82.93231
```

```
sqrt(mean(f2$residuals^2))
```

```
## [1] 76.69531
```

(compare variance vs. standard deviation)

Still there's a problem with interpreting this values.

Mean absolute error (MAE):

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

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

```
## [1] 22.86342
```

```
mean(abs(f1$residuals))
```

```
## [1] 61.48892
```

```
mean(abs(f2$residuals))
```

```
## [1] 64.1506
```

"On average, the predicted rating differs from the observed one by..."

Descriptive statistics for residuals:

```
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 include:

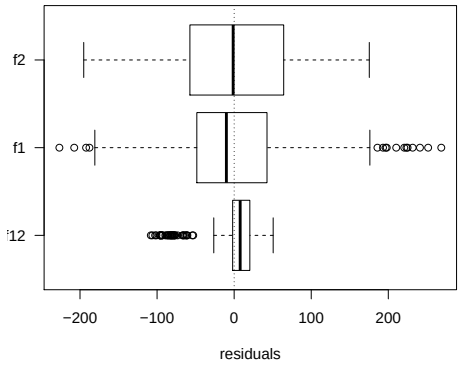
- `Min.` – sample minimum
- `1st Qu.` – 1st quartile == 25th percentile == quantile of order 0.25
- `Median` – median == 50th percentile == quantile of order 0.5
- `3rd Qu.` – 3rd quartile = 75th percentile == quantile of order 0.75
- `Max.` – sample maximum

See `?quantile` in R.

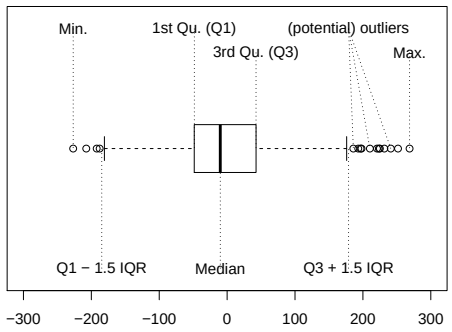
For example 1st quartile is the observation q such that 25% values are $\leq q$ and 75% values are $\geq q$.

A picture is worth a thousand words:

```
boxplot(las=1, horizontal=TRUE, xlab="residuals",
  list(f12=f12$residuals, f1=f1$residuals, f2=f2$residuals))
abline(v=0, lty=3)
```



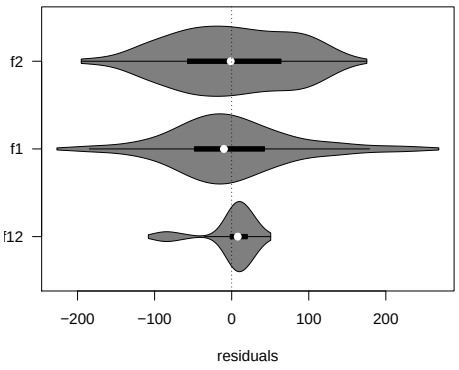
Box and whisker plot:



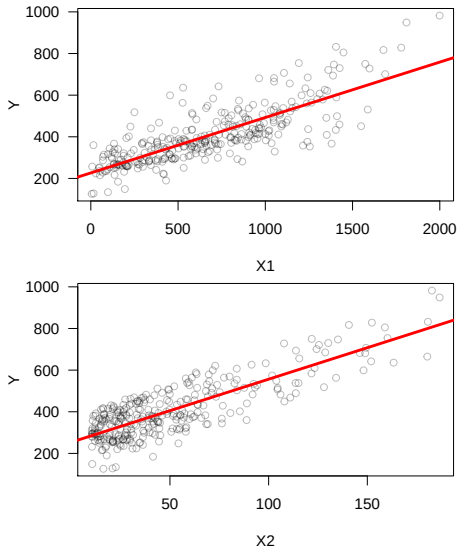
- IQR == Interquartile range == Q3−Q1 (box width)
- The box contains 50% of the "most typical" observations
- Box and whiskers altogether have width ≤ 4 IQR
- Outliers == observations potentially worth inspecting (is it a bug or a feature?)

Violin plot – a blend of a box plot and a (kernel) density estimator (histogram-like):

```
library("vioplot")
vioplot(las=1, horizontal=TRUE, xlab="residuals",
  list(f12=f12$residuals, f1=f1$residuals, f2=f2$residuals))
abline(v=0, lty=3)
```



By the way, this is Rating (Y) as function of Balance (X_1, top subfigure) and Income (X_2, bottom subfigure).



Descriptive statistics for absolute values of residuals:

```
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))
```

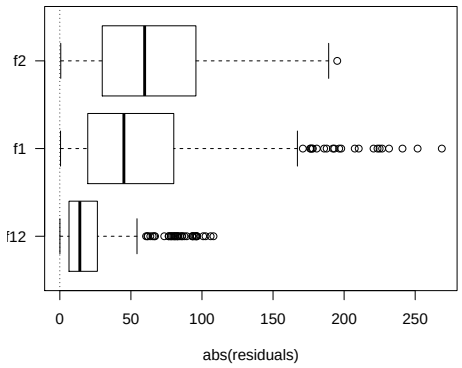
```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##   0.5056  19.6640  45.0716  61.4889  80.1239 268.7377
```

```
summary(abs(f2$residuals))
```

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

This picture is worth $1000:

```
boxplot(las=1, horizontal=TRUE, xlab="abs(residuals)",
  list(f12=abs(f12$residuals), f1=abs(f1$residuals),
       f2=abs(f2$residuals)))
abline(v=0, lty=3)
```



The (unadjusted) R^2 **score** (the coefficient of determination):

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

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

```
(r12 <- 1 - sum(f12$residuals^2)/sum((Y-mean(Y))^2) )
```

```
## [1] 0.9390901
```

```
(r1  <- 1 - sum(f1$residuals^2)/sum((Y-mean(Y))^2)  )
```

```
## [1] 0.6375085
```

```
(r2  <- 1 - sum(f2$residuals^2)/sum((Y-mean(Y))^2)  )
```

```
## [1] 0.6899812
```

$R^2(f) \simeq 1$ indicates a perfect fit – it is the proportion of variance of the dependent variable explained by independent variables in the model.

Unfortunately, R^2 tends to automatically increase as the number of independent variables increase.

To correct for this phenomenon, we can consider the **adjusted** R^2:

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

```
n <- length(x)
1 - (1 - r12)*(n-1)/(n-3)
```

```
## [1] 0.9386933
```

```
1 - (1 - r1 )*(n-1)/(n-2)
```

```
## [1] 0.6363316
```

```
1 - (1 - r2 )*(n-1)/(n-2)
```

```
## [1] 0.6889747
```

The adjusted R^2 penalises for more complex models.

> (*) Side note – results of some statistical tests (e.g., significance of coefficients) are reported by the `summary()` function — 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
```

2.3.2 Variable Selection

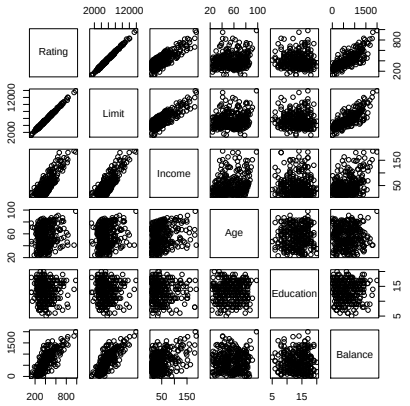
Consider all quantitative (numeric-continuous) variables in the `Credit` data set.

```
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
```

Let's draw a *pair plot* – a matrix of scatter plots for every pair of variables:

```
pairs(C)
```



Seems like `Rating` almost linearly depends on `Limit`...

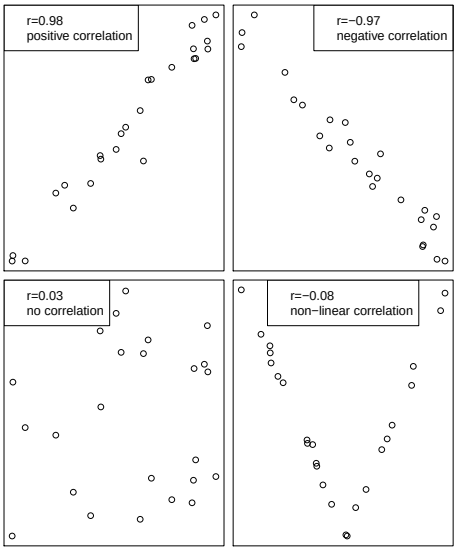
Pearson's r – linear correlation coefficient:

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

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

Interpretation:



Compute Pearson's r between all pairs of variables:

```
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, but both seem to describe the same thing.

For practical purposes, you'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?

The best model:

- has high predictive power,
- is simple.

These 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 RSS.

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

It turns out that the adjusted R^2, despite its interpretability, is not suitable for this task.

Instead, here we'll be using **the Akaike Information Criterion** (AIC).

For a model f with p independent variables:

$$\mathrm{AIC}(f) = 2(p+1) + n\log(\mathrm{RSS}(f)/n)$$

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

For p variables, the number of possible combinations is 2^p (grows exponentially with p).

For large p, an extensive search is impractical.

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 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 2 until AIC decreases

  (*) There are of course many other methods, e.g., lasso regression whose side effect is variable selection.

```
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
```

```
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
```

```
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
```

```
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
```

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.,

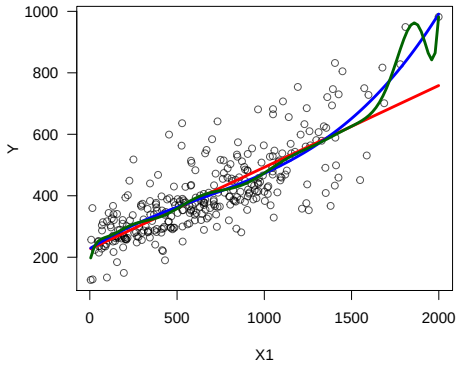
$$Y = be^{aX} \quad \rightarrow \quad \log Y = \log b + aX \quad \rightarrow \quad Y' = aX + b'$$

This is an example of a model's **linearisation**.

For example, here's a series of simple polynomial regression models of the form `Rating~poly(Balance)`:

```
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))
```

```
plot(X1, Y, las=1, 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)
```



2.3.4 Predictive vs. Descriptive Power

The above high-degree polynomial model (`f1_14`) is a clear example of 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 actually observed data, but fails to produce good results for the points that are yet to come.

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

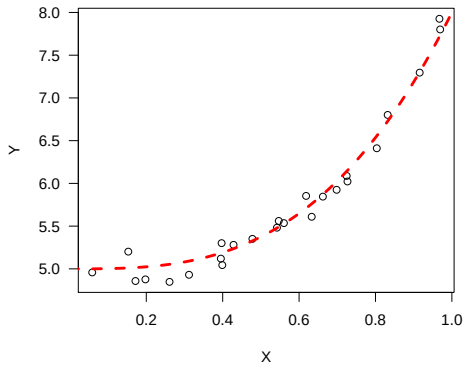
Assume our true model is of the form:

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

And we generate the following random sample from this model (with Y subject to error):

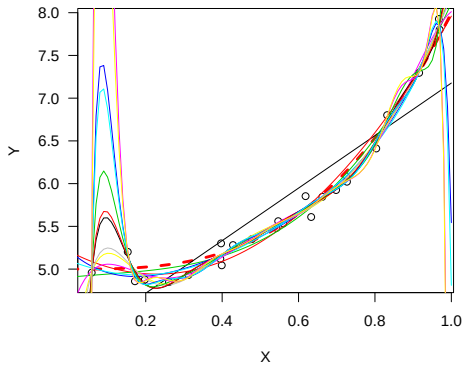
```
n <- 25
X <- runif(n, min=0, max=1)
Y <- true_model(X)+rnorm(n, sd=0.1)
```

```
plot(X, Y, las=1)
x <- seq(0, 1, length.out=101)
lines(x, true_model(x), col=2, lwd=3, lty=2)
```



Let's fit polynomials of different degrees:

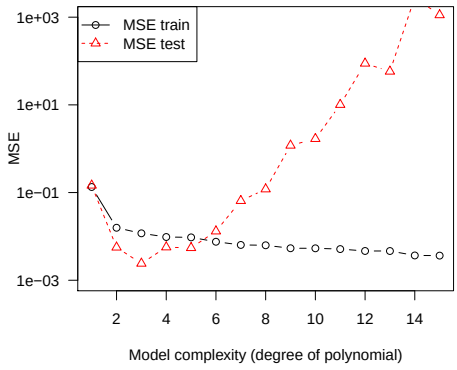
```
plot(X, Y, las=1)
lines(x, true_model(x), col=2, lwd=3, lty=2)

dmax <- 15
MSE_train <- numeric(dmax)
MSE_test  <- numeric(dmax)
for (d in 1:dmax) {
    f <- lm(Y~poly(X, d))
    y <- predict(f, data.frame(X=x))
    lines(x, y, col=d)

    MSE_train[d] <- mean(f$residuals^2)
    MSE_test[d]  <- mean((y-true_model(x))^2)
}
```



Compare the mean squared error (MSE) for the observed vs. future data points:

```
matplot(1:dmax, cbind(MSE_train, MSE_test), type='b',
    las=1, ylim=c(1e-3, 1e3), log="y", pch=1:2,
    xlab='Model complexity (degree of polynomial)',
    ylab="MSE")
legend("topleft", legend=c("MSE train", "MSE test"),
    lty=1:2, col=1:2, pch=1:2)
```

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.

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

2.4 Outro

2.4.1 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.

2.4.2 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.

2.4.3 Derivation of the Solution (**)

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

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

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

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

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

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

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

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

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

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

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

can be rewritten as:

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

and further as:

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

Note that all 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.

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

2.4.4 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

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

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:

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

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

$$\begin{cases} \frac{\partial E}{\partial \beta_0}(\boldsymbol{\beta}) &= 0 \\ \frac{\partial E}{\partial \beta_1}(\boldsymbol{\beta}) &= 0 \\ \vdots & \\ \frac{\partial E}{\partial \beta_p}(\boldsymbol{\beta}) &= 0 \end{cases}$$

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

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

Here, ∇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:

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

can be rewritten in matrix terms as:

$$\begin{cases} \beta_0 \dot{\mathbf{x}}_{\cdot,0}^T \dot{\mathbf{x}}_{\cdot,0} + \beta_1 \dot{\mathbf{x}}_{\cdot,0}^T \dot{\mathbf{x}}_{\cdot,1} + \cdots + \beta_p \dot{\mathbf{x}}_{\cdot,0}^T \dot{\mathbf{x}}_{\cdot,p} &= \dot{\mathbf{x}}_{\cdot,0}^T \mathbf{y} \\ \beta_0 \dot{\mathbf{x}}_{\cdot,1}^T \dot{\mathbf{x}}_{\cdot,0} + \beta_1 \dot{\mathbf{x}}_{\cdot,1}^T \dot{\mathbf{x}}_{\cdot,1} + \cdots + \beta_p \dot{\mathbf{x}}_{\cdot,1}^T \dot{\mathbf{x}}_{\cdot,p} &= \dot{\mathbf{x}}_{\cdot,1}^T \mathbf{y} \\ & \vdots \\ \beta_0 \dot{\mathbf{x}}_{\cdot,p}^T \dot{\mathbf{x}}_{\cdot,0} + \beta_1 \dot{\mathbf{x}}_{\cdot,p}^T \dot{\mathbf{x}}_{\cdot,1} + \cdots + \beta_p \dot{\mathbf{x}}_{\cdot,p}^T \dot{\mathbf{x}}_{\cdot,p} &= \dot{\mathbf{x}}_{\cdot,p}^T \mathbf{y} \end{cases}$$

This can be restated as:

$$\begin{cases} \left(\dot{\mathbf{x}}_{\cdot,0}^T \dot{\mathbf{X}}\right) \boldsymbol{\beta} &= \dot{\mathbf{x}}_{\cdot,0}^T \mathbf{y} \\ \left(\dot{\mathbf{x}}_{\cdot,1}^T \dot{\mathbf{X}}\right) \boldsymbol{\beta} &= \dot{\mathbf{x}}_{\cdot,1}^T \mathbf{y} \\ \vdots & \\ \left(\dot{\mathbf{x}}_{\cdot,p}^T \dot{\mathbf{X}}\right) \boldsymbol{\beta} &= \dot{\mathbf{x}}_{\cdot,p}^T \mathbf{y} \end{cases}$$

which in turn is equivalent to:

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

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

> (***) 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
```

2.4.5 Pearson's r in Matrix Form (**)

Recall the Pearson linear correlation coefficient:

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

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

Rewriting the above yields:

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

which is exactly:

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

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
```

2.4.6 Further Reading

Recommended further reading:

- (James et al. 2017: Chapters 1, 2 and 3)

Other:

- (Hastie, Tibshirani, and Friedman 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.

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 interested in **classification** tasks; we assume that each y_i is a discrete label.

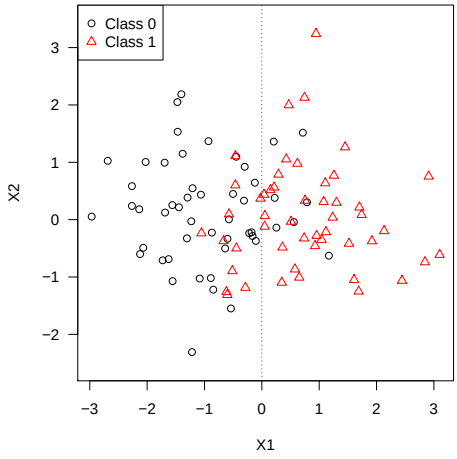
Most commonly, we are faced with **binary classification** tasks where there are only two possible labels.

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.



3.1.2 Factor Data Type

On a side note, `factor` type in R is a very convenient means to store categorical data.

```
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 $\{1, \ldots, M\}$, where M is the number of possible levels.

Labels, used to "decipher" the numeric codes, are stored separately.

```
as.integer(f) # 2nd label, 1st label, 1st label etc.
```

```
## [1] 2 1 1 2 1
```

```
levels(f)
```

```
## [1] "no"  "yes"
```

```
nlevels(f)
```

```
## [1] 2
```

```
levels(f) <- c("failure", "success") # re-encode
f
```

```
## [1] success failure failure success failure
## Levels: failure success
```

3.1.3 Data

For illustration, let's consider the `wines` dataset.

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

```
## [1] 5320
```

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

```
X <- as.matrix(wines[,1:11])
dim(X)
```

```
## [1] 5320   11
```

```
head(X, 2) # first two rows
```

```
##      fixed.acidity volatile.acidity citric.acid residual.sugar
## [1,]           7.4             0.70           0            1.9
## [2,]           7.8             0.88           0            2.6
##      chlorides free.sulfur.dioxide total.sulfur.dioxide density
## [1,]     0.076                  11                   34  0.9978
## [2,]     0.098                  25                   67  0.9968
##        pH sulphates alcohol
## [1,] 3.51      0.56     9.4
## [2,] 3.20      0.68     9.8
```

The `response` variable is an ordinal one, giving each wine's rating as assigned by a sommelier.

Here: 0 == a very bad wine, 10 == a very good one.

We will convert this dependent variable to a binary one:

- 0 == `response < 5` == bad
- 1 == `response >= 5` == good

```
# recall that TRUE == 1
Y <- factor(as.integer(wines$response >= 5))
table(Y)
```

```
## Y
##    0    1
## 4311 1009
```

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

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

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

70/30% train-test split in R:

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

```
## [1] 2463 2511 2227  526 4291 2986
```

```
# first 70% of the indexes (they are arranged randomly)
# will constitute the train sample:
train_indexes <- random_indexes[1:floor(n*0.7)]
X_train <- X[train_indexes,]
Y_train <- Y[train_indexes]
# the remaining indexes (30%) go to the test sample:
X_test  <- X[-train_indexes,]
Y_test  <- Y[-train_indexes]
```

3.1.5 Discussed Methods

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

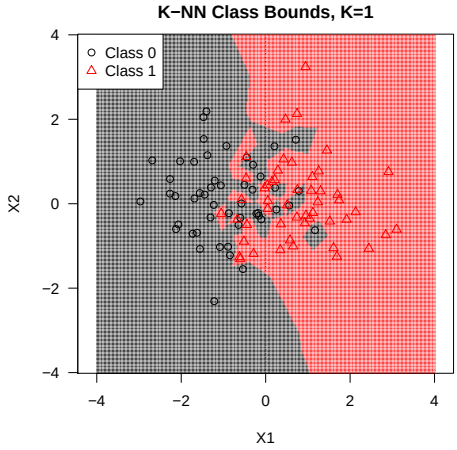
3.2.1 Introduction

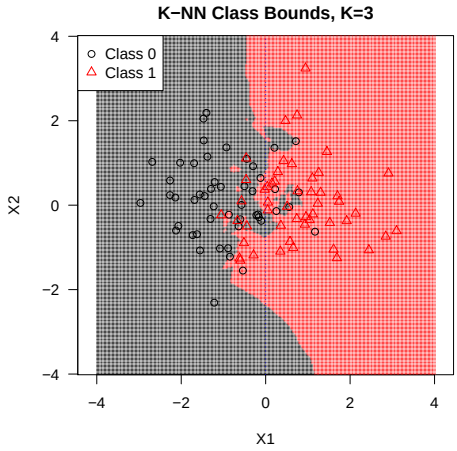
> "If you don't know what to do in a situation, just act like the people around you"

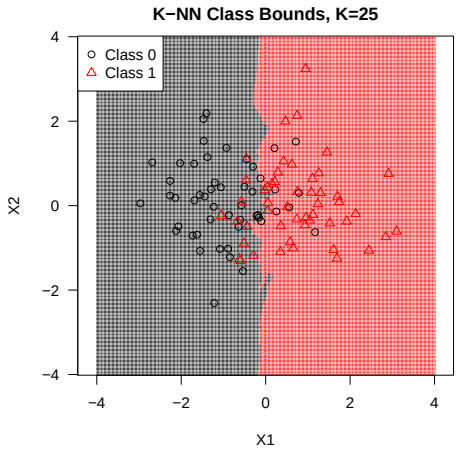
For some integer $K \geq 1$, the **K-Nearest Neighbour (*K-NN*) Classifier** proceeds as follows.

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 $\mathbf{x}_{i_1,\cdot}, \ldots, \mathbf{x}_{i_K,\cdot}$:
   a. compute the Euclidean distances between $\mathbf{x}'$ and each $\mathbf{x}_{i,\cdot}$ form the train set,
   $$d_i = \|\mathbf{x}' - \mathbf{x}_{i,\cdot}\|$$
   b. order d_is in increasing order, $d_{i_1} \leq d_{i_2} \leq \cdots \leq d_{i_K}$
   c. pick first K indexes (these are the *nearest* neighbours)
2. fetch the corresponding reference labels $y_{i_1}, \ldots, y_{i_K}$
3. return their *mode* as a result, i.e., the most frequently occurring label (a.k.a. *majority vote*)


K−NN Class Bounds, K=1
Class 0
Class 1
X2
X1
4
2
0
−2
−4
−4
−2
0
2
4


K−NN Class Bounds, K=3
Class 0
Class 1
X2
X1
4
2
0
−2
−4
−4
−2
0
2
4


K−NN Class Bounds, K=25
Class 0
Class 1
X2
X1
4
2
0
−2
−4
−4
−2
0
2
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 us make prediction using the 5-nn classifier:

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

```
##  [1] 0 0 0 1 0 1 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_knn5, 28) # Predicted 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
## Levels: 0 1
```

```
mean(Y_test == Y_knn5) # accuracy
```

```
## [1] 0.7896055
```

10-nn classifier:

```
Y_knn10 <- knn(X_train, X_test, Y_train, k=10)
head(Y_test, 28) # True Ys
```

```
##  [1] 0 0 0 1 0 1 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_knn10, 28) # Predicted 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
## Levels: 0 1
```

```
mean(Y_test == Y_knn10) # accuracy
```

```
## [1] 0.8008766
```

3.2.3 Different Metrics (*)

The Euclidean distance is just one particular example of many possible **metrics**.

Mathematically, we say that d is a metric on a set X (e.g., $\mathbb{R}^p$), whenever it is a function $d : X \times X \to [0, \infty]$ 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)

  (*) 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)

$$d_\infty(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_\infty = \max_{i=1,\ldots,p} |x_i - x_i'|$$

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.

3.2.4 Standardisation of Independent Variables

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).

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

Let us 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 <- 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))
```

Let us 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] 0.8215404
```

```
Y_knn10s <- knn(Z_train, Z_test, Y_train, k=10)
mean(Y_test == Y_knn10s) # accuracy
```

```
## [1] 0.8334377
```

3.3 Implementing a K-NN Classifier (*)

3.3.1 Main Routine (*)

Let us 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
    # ...
}
```

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 <- nlevels(Y_train)
```

Therefore,

`X_train` $\in \mathbb{R}^{\texttt{n\_train} \times \texttt{p}}$, `X_test` $\in \mathbb{R}^{\texttt{n\_test} \times \texttt{p}}$ and `Y_train` $\in \{1, \ldots, M\}^{\texttt{n\_train}}$

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_indexes of size n_test*k,
# where nn_indexes[i,j] denotes the index of
# X_test[i,]'s j-th nearest neighbour in X_train.
# (It is the point X_train[nn_indexes[i,j],]).
nn_indexes <- 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 <- integer(n_test) # vector of length n_test
# For now we will operate on the integer labels in {1,...,M}
Y_train_int <- as.integer(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_indexes[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)))
```

3.3.2 Mode

To implement the mode, we can use the `tabulate()` function.

> 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.

> 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
```

3.3.3 NN Search Routines (*)

Last but not least, we should implement the `our_get_knnx()` function.

It is the function responsible for seeking the indexes 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_indexes of size n_test*k,
# where nn_indexes[i,j] denotes the index of
# X_test[i,]'s j-th nearest neighbour in X_train.
# (It is the point X_train[nn_indexes[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_indexes <- 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_indexes[i,] <- order(d)[1:k]
    }
    nn_indexes
}
```

A comparison with `FNN:knn()`:

```
system.time(Ya <- knn(X_train, X_test, Y_train, k=5))
```

```
##    user  system elapsed
##   0.125   0.000   0.125
```

```
system.time(Yb <- our_knn(X_train, X_test, Y_train, k=5))
```

```
##    user  system elapsed
##  19.270   0.000  19.288
```

```
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 us substitute our naive implementation with the equivalent one, but written in C++ (available in the `FNN` package).

> (*) Note that we could 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
##   0.137   0.000   0.136
```

```
system.time(Yb <- our_knn(X_train, X_test, Y_train, k=5))
```

```
##    user  system elapsed
##   0.062   0.000   0.063
```

```
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_{test} and n_{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).

> (*) 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(
        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
## 0.36965016 0.01554856
```

```
test_speed(10000, 5, 5)
```

```
##      brute    kd_tree
## 0.52196965 0.09352585
```

```
test_speed(10000, 10, 5)
```

```
##     brute   kd_tree
## 0.8759096 0.9721890
```

```
test_speed(10000, 20, 5)
```

```
##     brute   kd_tree
##  1.968974 16.579919
```

3.4 Outro

3.4.1 Remarks

Note that K-NN is suitable for any kind of multiclass classification.

Some algorithms we are going to discuss in the next part are restricted to binary (0/1) outputs. They will have to be extended somehow to allow for more classes.

In the next part we will try to answer the question of how to choose the best K, and hence how to evaluate and pick the best model.

We will also discuss some other noteworthy classifiers:

- *Decision trees*
- *Logistic regression*

3.4.2 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 $\mathbf{x}_{i_1,\cdot}, \ldots, \mathbf{x}_{i_K,\cdot}$,
2. fetch the corresponding reference outputs $y_{i_1}, \ldots, y_{i_K}$,
3. return their arithmetic mean as a result,

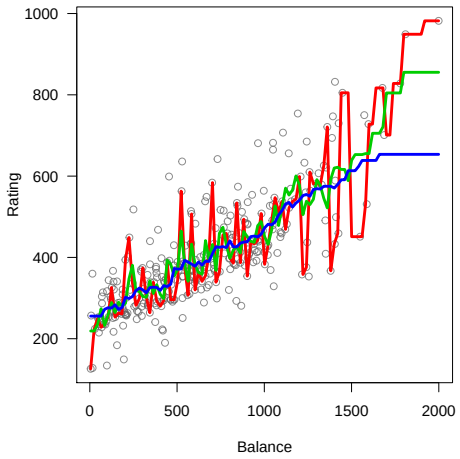
$$\hat{y} = \frac{1}{K} \sum_{j=1}^{K} y_{i_j}.$$

Recall our modelling of the Credit Rating (Y) as a function of the average Credit Card Balance (X) based on the `ISLR::Credit` data set.

```
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
```

```
plot(Xc, Yc, las=1, 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)
```



3.4.3 Further Reading

Recommended further reading:

- (Hastie, Tibshirani, and Friedman 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.

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.

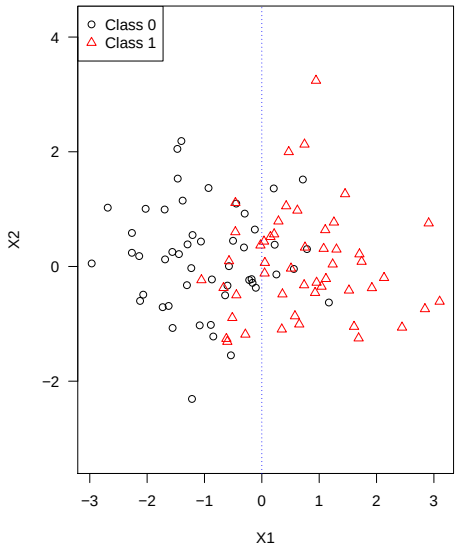
In this chapter we are still interested in **classification** tasks; we assume that each y_i is a discrete label.

In this part we 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



4.1.2 Data

For illustration, let's consider the `wines` dataset again.

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

```
## [1] 5320
```

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

```
X <- as.matrix(wines[,1:11])
dim(X)
```

```
## [1] 5320   11
```

```
head(X, 2) # first two rows
```

```
##      fixed.acidity volatile.acidity citric.acid residual.sugar
## [1,]           7.4             0.70           0            1.9
## [2,]           7.8             0.88           0            2.6
##      chlorides free.sulfur.dioxide total.sulfur.dioxide density
## [1,]     0.076                  11                   34  0.9978
## [2,]     0.098                  25                   67  0.9968
##        pH sulphates alcohol
## [1,] 3.51      0.56     9.4
## [2,] 3.20      0.68     9.8
```

The `response` variable is an ordinal one, giving each wine's rating as assigned by a sommelier.

Here: 0 == a very bad wine, 10 == a very good one.

We will convert this dependent variable to a binary one:

- 0 == `response  < 5` == bad
- 1 == `response >= 5` == good

```
# recall that TRUE == 1
Y <- as.integer(wines$response >= 5)
table(Y)
```

```
## Y
##    0    1
## 4311 1009
```

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

70/30% train-test split:

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

```
## [1] 2463 2511 2227  526 4291 2986
```

```
# first 70% of the indexes (they are arranged randomly)
# will constitute the train sample:
train_indexes <- random_indexes[1:floor(n*0.7)]
X_train <- X[train_indexes,]
Y_train <- Y[train_indexes]
# the remaining indexes (30%) go to the test sample:
X_test  <- X[-train_indexes,]
Y_test  <- Y[-train_indexes]
```

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))
```

4.1.3 Discussed Methods

We are soon going to discuss the following simple and educational (yet practically useful) classification algorithms:

- *decision trees*,
- *logistic regression*.

Before that happens, let's go back to the K-NN algorithm.

```
library("FNN")
Y_knn5   <- knn(X_train, X_test, Y_train, k=5)
Y_knn10  <- knn(X_train, X_test, Y_train, k=10)
Y_knn5s  <- knn(Z_train, Z_test, Y_train, k=5)
Y_knn10s <- knn(Z_train, Z_test, Y_train, k=10)
c(
    mean(Y_test == Y_knn5),
    mean(Y_test == Y_knn10),
    mean(Y_test == Y_knn5s),
    mean(Y_test == Y_knn10s)
)
```

```
## [1] 0.7896055 0.8008766 0.8215404 0.8334377
```

We should answer the question regarding the optimal choice of K first.

How should we do that?

4.2 Model Assessment and Selection

4.2.1 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):

.	$y_i = 0$	$y_i = 1$
$\hat{y}_i = 0$	**True Negative**	False Negative (Type II error)
$\hat{y}_i = 1$	False Positive (Type I error)	**True Positive**

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 <- Y_knn10s
(C <- table(Y_pred, Y_test))
```

```
##       Y_test
## Y_pred    0    1
##      0 1220  187
##      1   79  111
```

For example,

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

```
## [1] 1220
```

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

```
## [1] 79
```

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.

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

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] 0.8334377
```

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

```
## [1] 0.8334377
```

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.

> 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.

> 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

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

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] 0.5842105
```

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

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

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] 0.3724832
```

> 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:

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

> Show that the above equality holds.

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

```
## [1] 0.454918
```

```
get_metrics <- function(Y_test, Y_pred)
{
    C <- table(Y_pred, Y_test) # confusion matrix
    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
}
```

4.2.2 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 us 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_test, Y_pred)
}
```

For example:

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

```
##          Acc         Prec          Rec            F
##    0.8215404    0.5234657    0.4865772    0.5043478
##           TN           FN           FP           TP
## 1167.0000000  153.0000000  132.0000000  145.0000000
```

Example call to evaluate metrics as a function of different Ks:

```
Ks <- 1:10
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
    )))
```

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

We transpose this result, `t()`, in order to get each metric corresponding to different columns in the result.

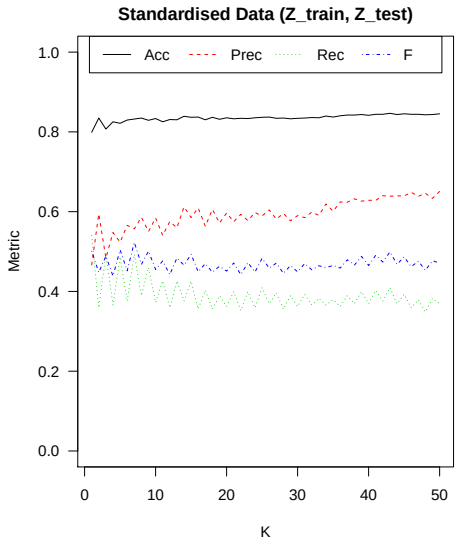
As usual, if you keep wondering, e.g., why `t()`, play with the code yourself – it's fun fun fun.

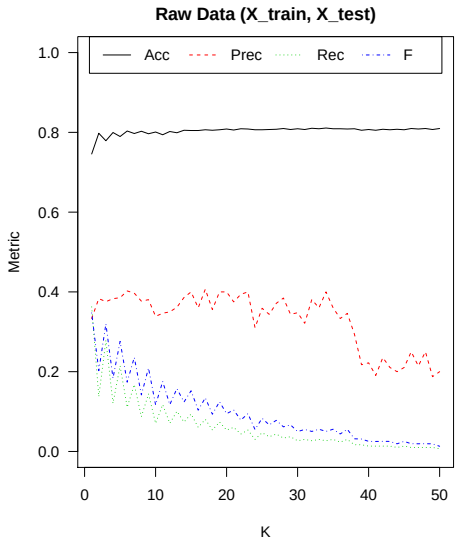
Example results:

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

```
##     K  Acc Prec  Rec    F   TN  FN  FP  TP
## 1   1 0.80 0.47 0.54 0.50 1115 137 184 161
## 2   2 0.83 0.59 0.36 0.45 1226 191  73 107
## 3   3 0.81 0.48 0.49 0.49 1142 151 157 147
## 4   4 0.83 0.55 0.37 0.44 1209 189  90 109
## 5   5 0.82 0.52 0.49 0.50 1167 153 132 145
## 6   6 0.83 0.57 0.38 0.45 1213 186  86 112
## 7   7 0.83 0.56 0.49 0.52 1182 151 117 147
## 8   8 0.83 0.59 0.39 0.47 1217 182  82 116
## 9   9 0.83 0.55 0.46 0.50 1187 161 112 137
## 10 10 0.83 0.58 0.37 0.45 1220 187  79 111
```

A picture is worth a thousand tables though (see `?matplot` in R).





4.2.3 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 metric?

- Which metric should be used?

  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

  (*) If our dataset is too small, we can use various *crossvalidation* techniques instead of a train-validate-test split.

An example way to perform a train-validation-test split:

```
set.seed(123) # reproducibility matters
random_indexes <- sample(n)
n1 <- floor(n*0.6)
n2 <- floor(n*0.8)
X2_train <- X[1:n1,]
Y2_train <- Y[1:n1]
X2_valid <- X[(n1+1):n2,]
Y2_valid <- Y[(n1+1):n2]
X2_test  <- X[(n2+1):n,]
Y2_test  <- Y[(n2+1):n]
stopifnot(nrow(X2_train)+nrow(X2_valid)+nrow(X2_test)
    == nrow(X))
```

4.3 Decision Trees

4.3.1 Introduction

Note that a K-NN classifier is **model-free**. The whole training set must be stored and referred to at all times.

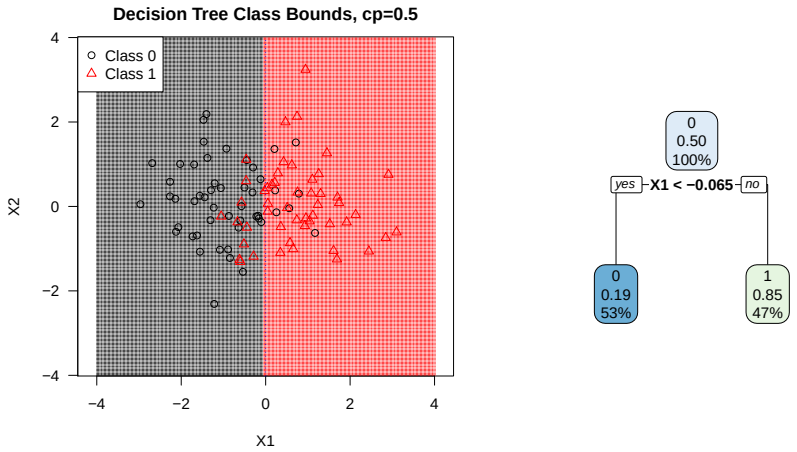
Therefore, it doesn't *explain* the data we have – we may use it solely for the purpose of *prediction*.

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** $\hat{y}_i = 1$**.**

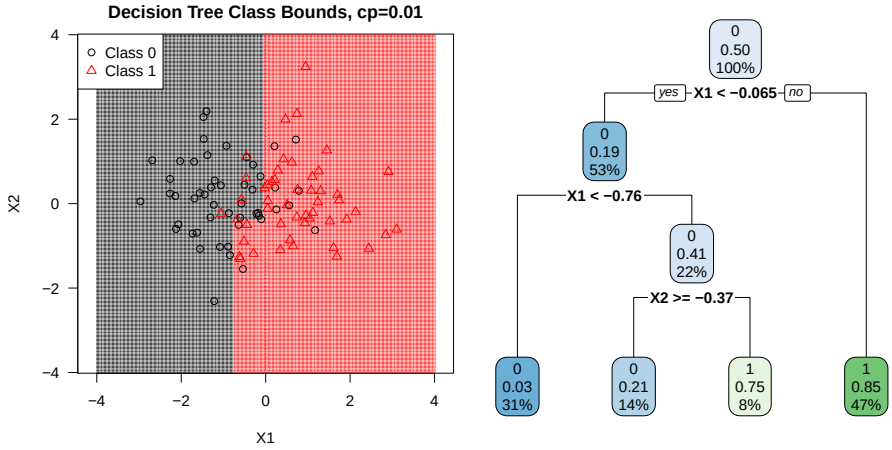
These can be organised into a **hierarchy** for greater readability.

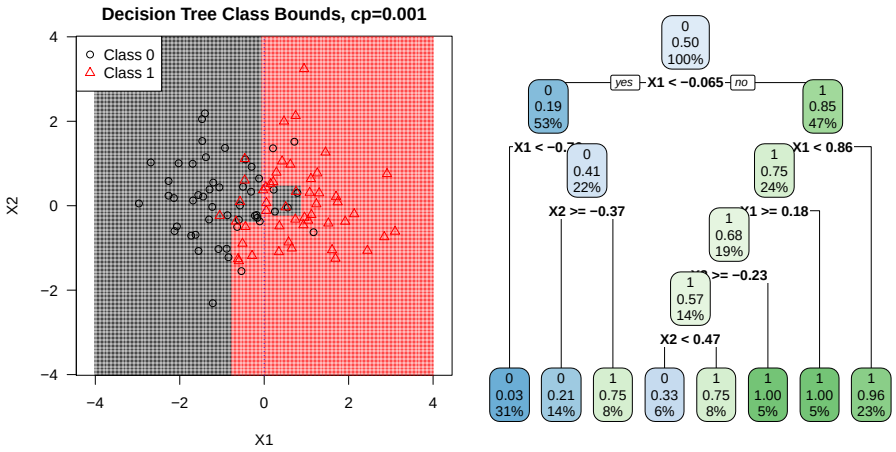
This idea inspired the notion of **decision trees** (Breiman et al. 1984).



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





4.3.2 Example in R

We will use the `rpart()` function from the `rpart` package to build a classification tree.

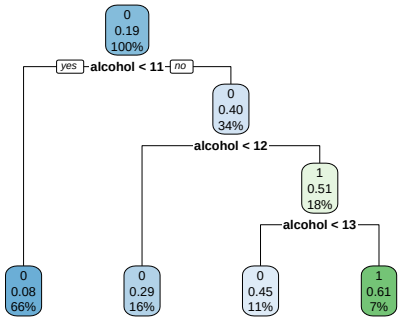
```
library("rpart")
library("rpart.plot")
set.seed(123)
```

`rpart()` uses a formula (`~`) interface, hence it will be easier to feed it with data in a data.frame form.

```
XY_train <- as.data.frame(cbind(X_train, Y=Y_train))
XY_test <- as.data.frame(cbind(X_test, Y=Y_test))
```

Fit and plot a decision tree:

```
t1 <- rpart(Y~., data=XY_train, method="class")
rpart.plot(t1)
```

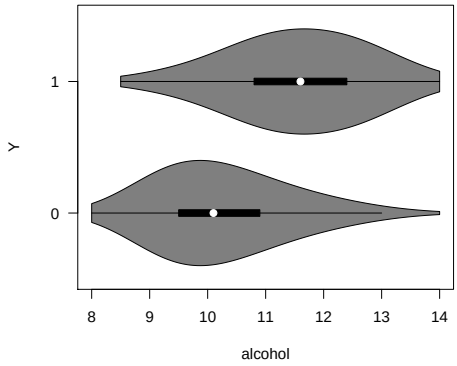


The fitted model is rather... simple.

Only the `alcohol` variable is taken into account.

Well note how these two distributions are shifted:

```
vioplot::vioplot(alcohol~Y, data=XY_train,
    horizontal=TRUE, las=1)
```



Make predictions:

```
Y_pred <- predict(t1, XY_test, type="class")
get_metrics(Y_test, Y_pred)
```

```
##          Acc         Prec          Rec            F
##    0.8202880    0.5447154    0.2248322    0.3182898
##           TN           FN           FP           TP
## 1243.0000000  231.0000000   56.0000000   67.0000000
```

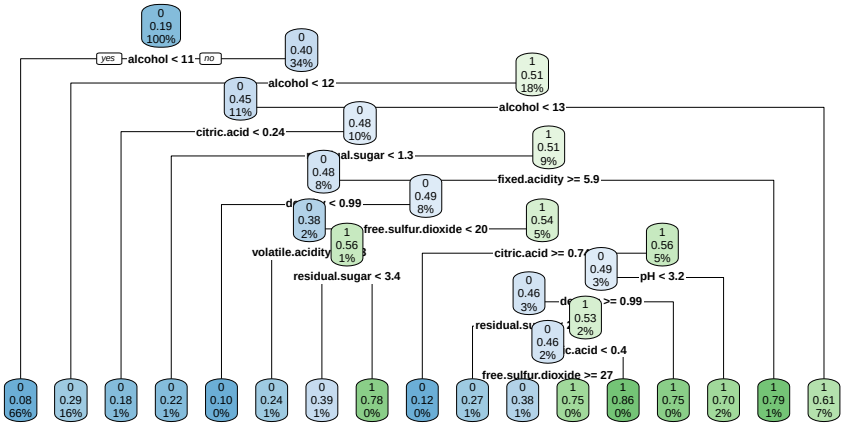
(*) Interestingly, `rpart()` also provides us with information about the importance degrees of each independent variable.

```
t1$variable.importance/sum(t1$variable.importance)
```

```
##              alcohol              density            chlorides
##         0.5664666547         0.2580265845         0.1196468644
##        fixed.acidity     volatile.acidity            sulphates
##         0.0187346795         0.0138674578         0.0115060641
##       residual.sugar total.sulfur.dioxide  free.sulfur.dioxide
##         0.0078854172         0.0036807388         0.0001855391
```

We can build a much more complex tree by playing with the `cp` parameter.

```
# cp = complexity parameter, smaller → more complex tree
t2 <- rpart(Y~., data=XY_train, method="class", cp=0.007)
rpart.plot(t2, tweak=2.5, compress=FALSE)
```




```
Y_pred <- predict(t2, XY_test, type="class")
get_metrics(Y_test, Y_pred)
```

```
##          Acc         Prec          Rec            F
##    0.8159048    0.5100000    0.3422819    0.4096386
##           TN           FN           FP           TP
## 1201.0000000  196.0000000   98.0000000  102.0000000
```

4.3.3 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 and Atkinson 2019).

4.4 Binary Logistic Regression

4.4.1 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.

Theoretically, we could treat the class labels as numeric 0s and 1s and apply regression models in a binary classification task.

```
XY_train_r <- as.data.frame(cbind(X_train,
    Y=as.numeric(as.character(Y_train)) # 0.0 or 1.0
))
f <- lm(Y~., data=XY_train_r)
```

```
Y_pred <- predict(f, as.data.frame(X_test))
summary(Y_pred)
```

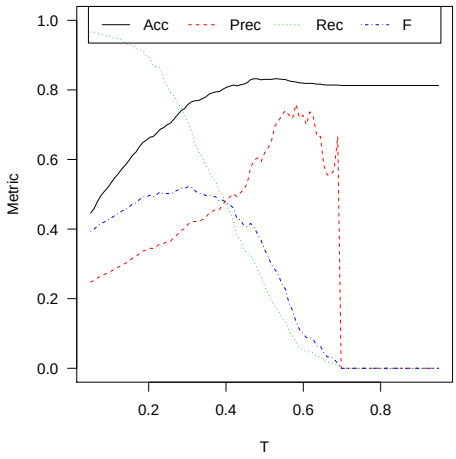
```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.76785   0.03625   0.18985   0.20120   0.34916   0.95385
```

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.integer(Y_pred>0.5)
get_metrics(Y_test, Y_pred)
```

```
##          Acc         Prec          Rec            F
##    0.8303068    0.6194690    0.2348993    0.3406326
##           TN           FN           FP           TP
## 1256.0000000  228.0000000   43.0000000   70.0000000
```

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).



4.4.2 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 $Y \in \mathbb{R}$.

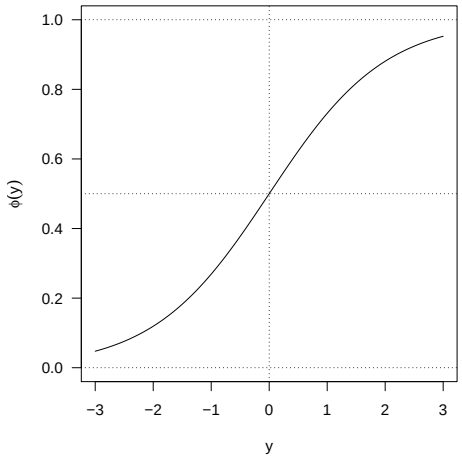
However, we could transform the real-valued outputs by means of some function $\phi : \mathbb{R} \to [0, 1]$ (preferably S-shaped == sigmoid), so as to get:

$$\Pr(Y = 1|\mathbf{X}, \boldsymbol{\beta}) = \phi(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p)$$

> The above reads as "Probability that Y is from class 1 given $\mathbf{X}$ and $\boldsymbol{\beta}$".

A popular choice is the **logistic sigmoid function**,

$$\phi(y) = \frac{1}{1 + e^{-y}} = \frac{e^y}{1 + e^y}$$



Hence our model becomes:

$$Y = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p)}}$$

We call it a **generalised linear model** (glm).

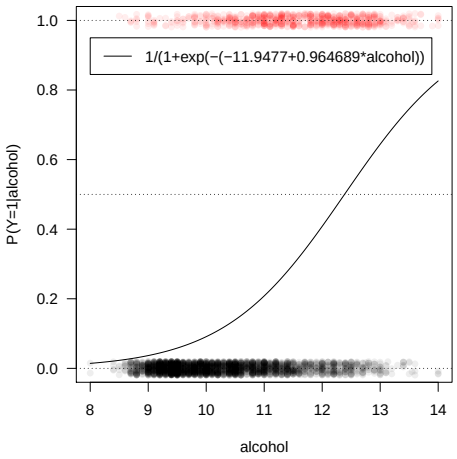
4.4.3 Example in R

Let us first fit a simple (i.e., $p = 1$) logistic regression model using the `alcohol` variable.

"logit" below denotes the inverse of the logistic sigmoid function.

```
f <- glm(Y~alcohol, data=XY_train, family=binomial("logit"))
summary(f)
```

```
##
## Call:
## glm(formula = Y ~ alcohol, family = binomial("logit"), data = XY_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.8703  -0.5994  -0.3984  -0.2770   2.7463
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -11.94770    0.46819  -25.52   <2e-16 ***
## alcohol       0.96469    0.04166   23.16   <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: 3630.9  on 3722  degrees of freedom
## Residual deviance: 2963.4  on 3721  degrees of freedom
## AIC: 2967.4
##
## Number of Fisher Scoring iterations: 5
```



Some predicted probabilities:

```
head(predict(f, XY_test, type="response"))
```

```
##           1           2           3           4           5
## 0.07629709 0.07629709 0.05316971 0.05824087 0.13961713
##           6
## 0.13961713
```

We classify Y as 1 if the corresponding membership probability is greater than 0.5.

```
Y_pred <- as.integer(predict(f, XY_test, type="response")>0.5)
get_metrics(Y_test, Y_pred)
```

```
##          Acc         Prec          Rec            F
##    0.8127740    0.4971751    0.2953020    0.3705263
##           TN           FN           FP           TP
## 1210.0000000  210.0000000   89.0000000   88.0000000
```

And now a fit based on all the input variables:

```
f <- glm(Y~., data=XY_train, family=binomial("logit"))
Y_pred <- as.integer(predict(f, XY_test, type="response")>0.5)
get_metrics(Y_test, Y_pred)
```

```
##          Acc         Prec          Rec            F
##    0.8202880    0.5287958    0.3389262    0.4130879
##           TN           FN           FP           TP
## 1209.0000000  197.0000000   90.0000000  101.0000000
```

4.4.4 Loss Function

The fitting of the model can be written as an optimisation task:

$$\min_{\beta_0,\beta_1,\ldots,\beta_p\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n} e\left(\frac{1}{1+e^{-(\beta_0+\beta_1 x_{i,1}+\cdots+\beta_p x_{i,p})}}, y_i\right)$$

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**),

$$e(\hat{y}, y) = -\left(y \log \hat{y} + (1-y)\log(1-\hat{y})\right)$$

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 $t \in (0, 1)$.

$$e(\hat{y}, y) = -\left(y \log \hat{y} + (1-y)\log(1-\hat{y})\right)$$

- 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 $-\log(t) \to 0$ as $t \to 1$
  - however, if the classifier is totally wrong, i.e., it thinks that $\hat{y} \simeq 0$, then the penalty will be very high, as $-\log(t) \to +\infty$ as $t \to 0$
- 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)

(*) Interestingly, there is no analytical formula for the optimal set of parameters $(\beta_0, \beta_1, \ldots, \beta_p)$ minimising the log-loss.

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.

4.5 Outro

4.5.1 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`.

4.5.2 Further Reading

Recommended further reading:

- (James et al. 2017: Chapters 4 and 8)

Other:

- (Hastie, Tibshirani, and Friedman 2017: Chapters 4 and 7 as well as (*) Chapters 9, 10, 13, 15)

Chapter 5

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.

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

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

With each input $\mathbf{x}_{i,.}$ we associate the desired output y_i which is a categorical label – hence we will be dealing with **classification** tasks again.

In **binary logistic regression** we were modelling the probabilities that a given input belongs to either of the two classes:

$$\begin{aligned} \Pr(Y = 1|\mathbf{X}, \boldsymbol{\beta}) = &\quad \phi(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p) \\ \Pr(Y = 0|\mathbf{X}, \boldsymbol{\beta}) = &\quad 1 - \phi(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p) \end{aligned}$$

where $\phi(z) = \frac{1}{1+e^{-z}}$ is the logistic sigmoid function.

It holds:

$$\begin{aligned}\Pr(Y=1|\mathbf{X},\boldsymbol{\beta}) &= \frac{1}{1+e^{-(\beta_0+\beta_1 X_1+\cdots+\beta_p X_p)}}\\ \Pr(Y=0|\mathbf{X},\boldsymbol{\beta}) &= \frac{e^{-(\beta_0+\beta_1 X_1+\cdots+\beta_p X_p)}}{1+e^{-(\beta_0+\beta_1 X_1+\cdots+\beta_p X_p)}}\end{aligned}$$

The fitting of the model was performed by minimising the cross-entropy (log-loss):

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^{p+1}} -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log \Pr(Y=1|\mathbf{x}_{i,\cdot},\boldsymbol{\beta}) + (1-y_i)\log \Pr(Y=0|\mathbf{x}_{i,\cdot},\boldsymbol{\beta})\right).$$

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:

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^{p+1}} -\frac{1}{n}\sum_{i=1}^{n}\log \Pr(Y=y_i|\mathbf{x}_{i,\cdot},\boldsymbol{\beta}).$$

In this chapter we will generalise the binary logistic regression model:

- First we will consider the case of multiclass classification.
- Then we will note that 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/

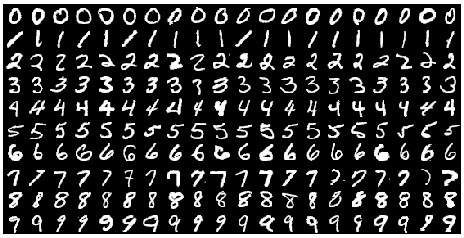
It consists of 28×28 pixel images of handwritten digits:

- `train`: 60,000 training images,
- `t10k`: 10,000 testing images.

There are 10 unique digits, so this is a multiclass classification problem.

> 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.

A few image instances from each class:



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 28×28 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 28×28, respectively.

These are greyscale images, with 0 = black, ..., 255 = white:

```
range(X_train)
```

```
## [1]   0 255
```

It is better to convert the colour values to 0.0 = black, ..., 1.0 = 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
```

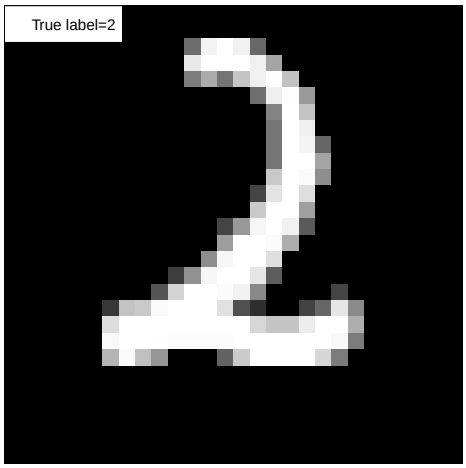
```
table(Y_train) # label distribution in train sample
```

```
## 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 test sample
```

```
## Y_test
##    0    1    2    3    4    5    6    7    8    9
##  980 1135 1032 1010  982  892  958 1028  974 1009
```

```
id <- 123 # which image 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 28×28, 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)
```

The classifiers studied here do not take the "spatial" positioning of the pixels into account anyway.

> (*) See, however, convolutional neural networks (CNNs), e.g., in (Goodfellow, Bengio, and Courville 2016).

Hence, now we're back to our "comfort zone".

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" the outputs of a linear model (so that we obtain a value in $[0, 1]$).

In the multinomial case, we can use a separate linear model for each digit so that $\Pr(Y = k|\mathbf{X}, \mathbf{B})$ is given as a function of

$$\beta_{0,k} + \beta_{1,k}X_1 + \dots + \beta_{p,k}X_p.$$

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: upper case of β is B.

Then, these 10 numbers will have to be normalised so as to they are positive and sum to 1.

To maintain the spirit of the original model, we can apply $e^{-(\beta_{0,k}+\beta_{1,k}X_1+\dots+\beta_{p,k}X_p)}$ 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:

$$\begin{aligned}
\Pr(Y=0|\mathbf{X},\mathbf{B}) &= \frac{e^{-(\beta_{0,0}+\beta_{1,0}X_1+\cdots+\beta_{p,0}X_p)}}{\sum_{k=0}^{9} e^{-(\beta_{0,k}+\beta_{1,k}X_1+\cdots+\beta_{p,k}X_p)}},\\
\Pr(Y=1|\mathbf{X},\mathbf{B}) &= \frac{e^{-(\beta_{0,1}+\beta_{1,1}X_1+\cdots+\beta_{p,1}X_p)}}{\sum_{k=0}^{9} e^{-(\beta_{0,k}+\beta_{1,k}X_1+\cdots+\beta_{p,k}X_p)}},\\
&\vdots\\
\Pr(Y=9|\mathbf{X},\mathbf{B}) &= \frac{e^{-(\beta_{0,9}+\beta_{1,9}X_1+\cdots+\beta_{p,9}X_p)}}{\sum_{k=0}^{9} e^{-(\beta_{0,k}+\beta_{1,k}X_1+\cdots+\beta_{p,k}X_p)}}.
\end{aligned}$$

Note that we get the binary logistic regression if we fix $\beta_{0,0} = \beta_{1,0} = \cdots = \beta_{p,0} = 0$ as $e^0 = 1$ and consider only the classes 0 and 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, 0, 9,  0, 0, 8))
round(y_pred, 2) # probabilities of class 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
```

> `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 $\arg\max_{k=1,\ldots,K} y_k$.

To make processing the outputs of a logistic regression model more convenient, we will apply the **one-hot-encoding** of the labels.

Here, each label will be represented as a 0-1 probability vector – with probability 1 corresponding to the true class only.

For example:

```
y <- 2 # true class (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 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
```

> 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_train` and `Y_test`:

```
Y_train2 <- one_hot_encode(Y_train)
Y_test2 <- one_hot_encode(Y_test)
```

5.2.5 Cross-entropy Revisited

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:

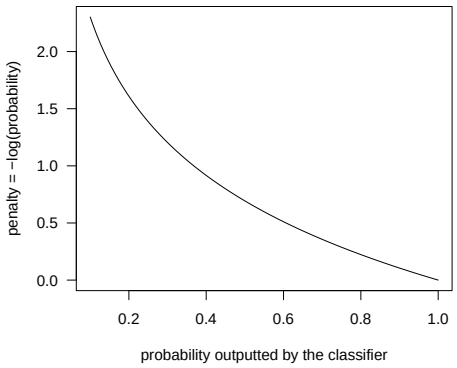
```
y2
```

```
##  [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:



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 $\{1, 2, \ldots, K\}$ 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:

$$\begin{aligned}
\hat{y}_{i,1} = \Pr(Y=1|\mathbf{x}_{i,\cdot},\mathbf{B}) &= \frac{e^{\beta_{0,1}+\beta_{1,1}x_{i,1}+\cdots+\beta_{p,1}x_{i,p}}}{\sum_{k=1}^{K} e^{\beta_{0,k}+\beta_{1,k}x_{i,1}+\cdots+\beta_{p,k}x_{i,p}}},\\
&\vdots\\
\hat{y}_{i,K} = \Pr(Y=K|\mathbf{x}_{i,\cdot},\mathbf{B}) &= \frac{e^{\beta_{0,K}+\beta_{1,K}x_{i,1}+\cdots+\beta_{p,K}x_{i,p}}}{\sum_{k=1}^{K} e^{\beta_{0,k}+\beta_{1,k}x_{i,1}+\cdots+\beta_{p,k}x_{i,p}}}.
\end{aligned}$$

> We have dropped the minus sign in the exponentiation for the 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:

- $x_{i,j}$ – i-th observation, j-th feature;
- $\hat{y}_{i,k}$ – i-th observation, k-th class probability;
- $\beta_{j,k}$ – coefficient for the j-th feature when computing the k-th class.

Note that by augmenting $\dot{\mathbf{X}} = [\mathbf{1}\ \mathbf{X}] \in \mathbb{R}^{n\times(p+1)}$, 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:

$$\begin{aligned}
t_{i,1} &= \textstyle\sum_{j=0}^{p} \dot{x}_{i,j}\,\beta_{j,1} &&= \dot{\mathbf{x}}_{i,\cdot}\,\boldsymbol{\beta}_{\cdot,1}\\
&\vdots\\
t_{i,K} &= \textstyle\sum_{j=0}^{p} \dot{x}_{i,j}\,\beta_{j,K} &&= \dot{\mathbf{x}}_{i,\cdot}\,\boldsymbol{\beta}_{\cdot,K}
\end{aligned}$$

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, 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 $\mathbf{T}$:

$$\hat{\mathbf{Y}} = \text{softmax}\left(\dot{\mathbf{X}}\,\mathbf{B}\right).$$

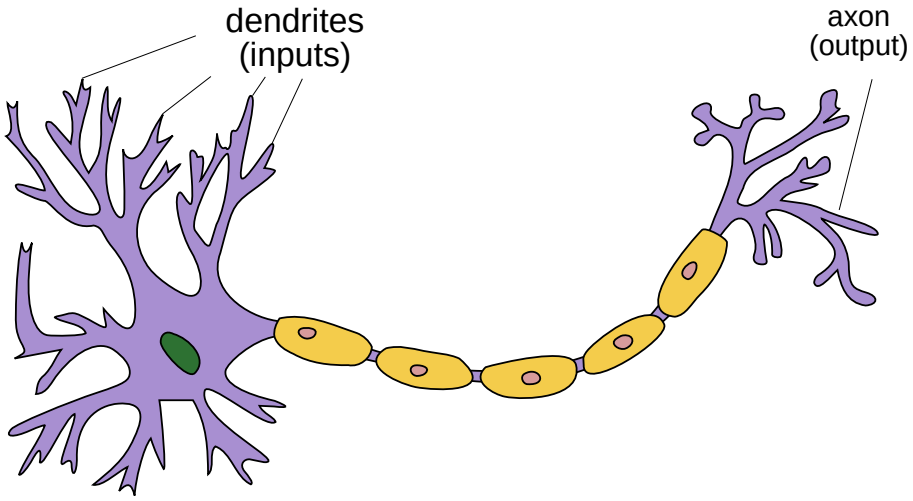
That's it. Take some time to appreciate the elegance of this notation.

Methods for minimising crossentropy expressed in matrix form will be discussed in the next chapter.

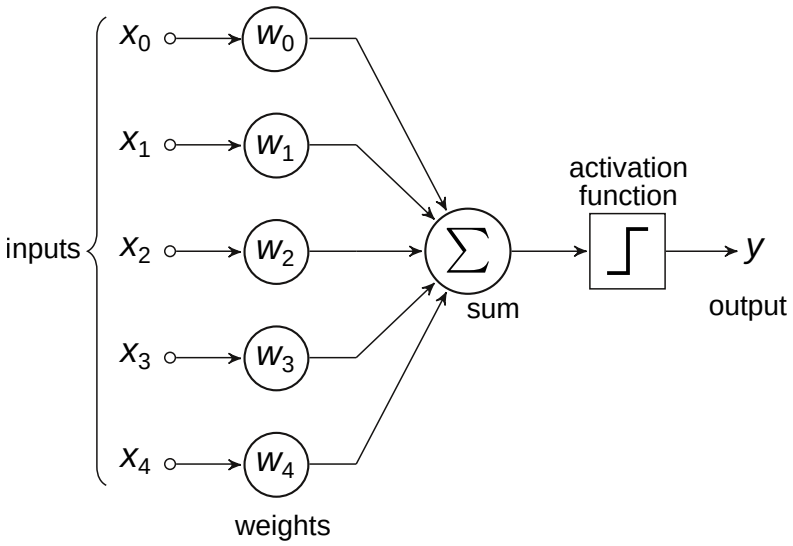
5.3 Artificial Neural Networks

5.3.1 Artificial Neuron

A neuron as a mathematical function:



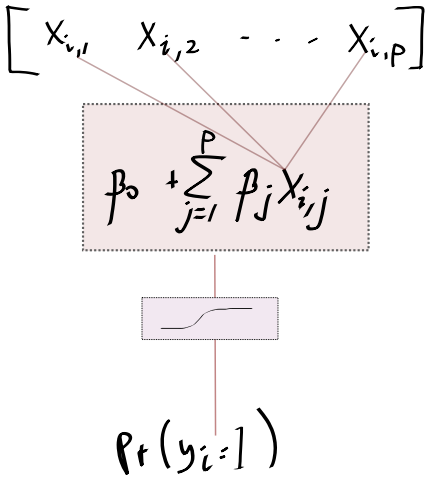
The **perceptron** (Frank Rosenblatt, 1958) was amongst the first models of artificial neurons:



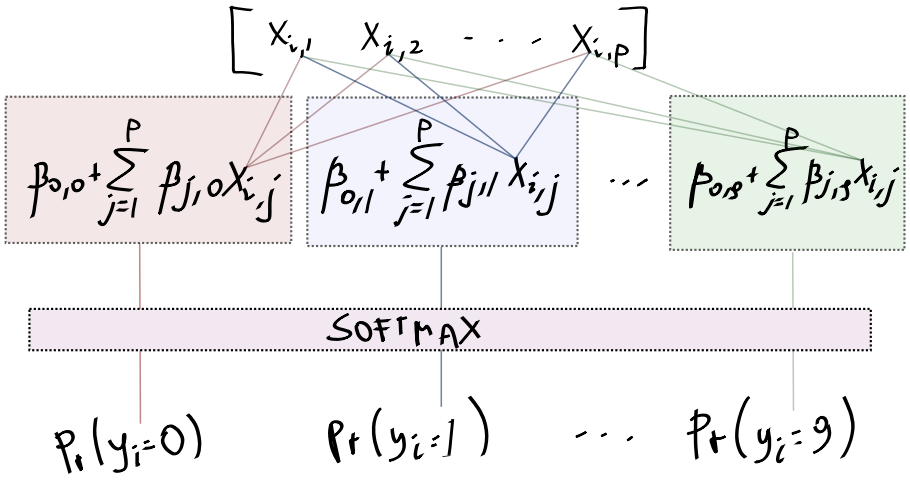
5.3.2 Logistic Regression as a Neural Network

The above resembles our binary logistic regression model!

We determine a linear combination (a weighted sum) of 784 inputs and then transform it using the logistic sigmoid "activation" function.



A multiclass logistic regression can be depicted as:



This is 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 neurons),
- **feed-forward** (outputs are generated by processing the inputs directly, 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 (fit a multinomial logistic regression model), we will use the keras package, a wrapper around the state-of-the-art, GPU-enabled TensorFlow library.

```
# Start with an empty model
model <- keras_model_sequential()
# Add a single layer with 10 units and softmax activation
layer_dense(model, units=10, activation='softmax')
# We will be minimising the cross-entropy,
# sgd == stochastic gradient descent, see the next chapter
compile(model, optimizer='sgd',
        loss='categorical_crossentropy')
# Fit the model
fit(model, X_train2, Y_train2, epochs=5)
```

Predict over the test set and one-hot-decode the output probabilities:

```
Y_pred2 <- predict(model, 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 0.89 0.02 0.00 0.01 0.06 0.00 0.01  0.00
## [3,] 0.00 0.94 0.02 0.01 0.00 0.00 0.01 0.01 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 0.89 0.00 0.01 0.02 0.01  0.05
## [6,] 0.00 0.97 0.00 0.01 0.00 0.00 0.00 0.00 0.01  0.00
```

```
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)
```

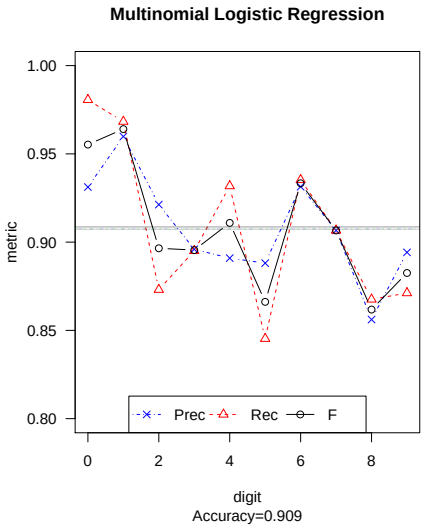
```
## [1] 0.9086
```

Performance metrics for each digit separately:

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9910	0.9312016	0.9806122	0.9552684	8949	19	71	961
1	0.9918	0.9598253	0.9682819	0.9640351	8819	36	46	1099
2	0.9792	0.9212679	0.8730620	0.8965174	8891	131	77	901
3	0.9789	0.8959366	0.8950495	0.8954928	8885	106	105	904
4	0.9821	0.8909445	0.9317719	0.9109009	8906	67	112	915
5	0.9767	0.8881037	0.8452915	0.8661689	9013	138	95	754
6	0.9872	0.9313929	0.9352818	0.9333333	8976	62	66	896
7	0.9808	0.9066148	0.9066148	0.9066148	8876	96	96	932
8	0.9729	0.8561297	0.8675565	0.8618052	8884	129	142	845
9	0.9766	0.8942014	0.8711596	0.8825301	8887	130	104	879

Note how misleading the individual accuracies are! Averages:

```
##       Acc      Prec       Rec         F
## 0.9817200 0.9075618 0.9074682 0.9072667
```

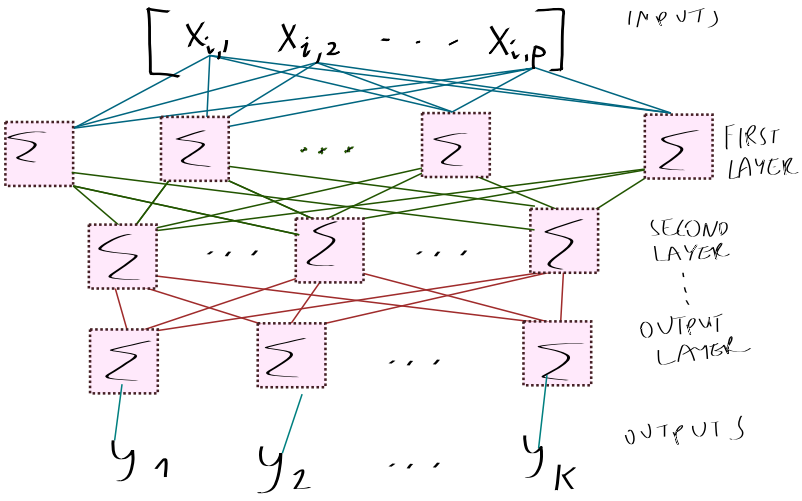


5.4 Deep Neural Networks

5.4.1 Introduction

In a brain, a neuron's output is an input to another neuron.

We could try aligning neurons into many interconnected layers.



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:

$$\phi(t) = 1/(1 + \exp(-t)),$$

- `tanh` – The hyperbolic function:

$$\tanh(t) = (\exp(t) - \exp(-t))/(\exp(t) + \exp(-t)).$$

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

2-layer Neural Network 784-800-10

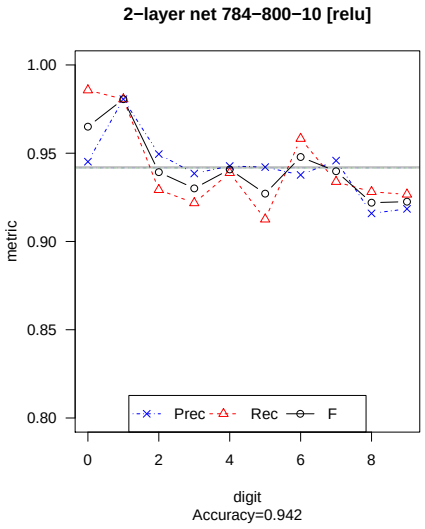
```
model <- keras_model_sequential()
layer_dense(model, units=800, activation='relu')
layer_dense(model, units=10,  activation='softmax')
compile(model, optimizer='sgd',
        loss='categorical_crossentropy')
fit(model, X_train2, Y_train2, epochs=5)

Y_pred2 <- predict(model, 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] 0.9422
```

Performance metrics for each digit separately:

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9930	0.9452055	0.9857143	0.9650350	8964	14	56	966
1	0.9956	0.9806167	0.9806167	0.9806167	8843	22	22	1113
2	0.9876	0.9495050	0.9292636	0.9392752	8917	73	51	959

i	Acc	Prec	Rec	F	TN	FN	FP	TP
3	0.9860	0.9385081	0.9217822	0.9300699	8929	79	61	931
4	0.9884	0.9427403	0.9389002	0.9408163	8962	60	56	922
5	0.9872	0.9421296	0.9125561	0.9271071	9058	78	50	814
6	0.9899	0.9376915	0.9582463	0.9478575	8981	40	61	918
7	0.9877	0.9458128	0.9338521	0.9397944	8917	68	55	960
8	0.9847	0.9159068	0.9281314	0.9219786	8943	70	83	904
9	0.9843	0.9184676	0.9266601	0.9225456	8908	74	83	935



5.4.4 Example in R - 6 Layers

6-layer *Deep* Neural Network 784-2500-2000-1500-1000-500-10

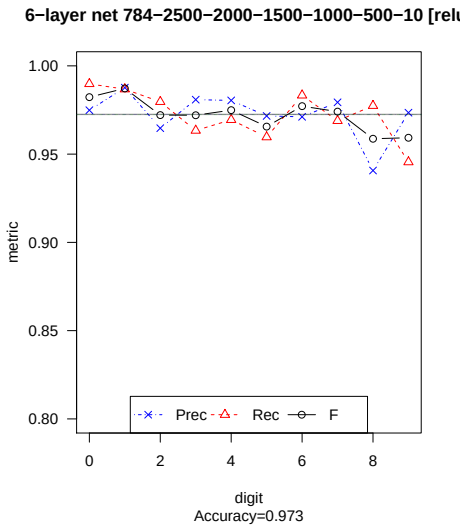
```
model <- keras_model_sequential()
layer_dense(model, units=2500, activation='relu')
layer_dense(model, units=2000, activation='relu')
layer_dense(model, units=1500, activation='relu')
layer_dense(model, units=1000, activation='relu')
layer_dense(model, units=500,  activation='relu')
layer_dense(model, units=10,   activation='softmax')
compile(model, optimizer='sgd',
        loss='categorical_crossentropy')
fit(model, X_train2, Y_train2, epochs=5)
```

```
Y_pred2 <- predict(model, 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] 0.9726
```

Performance metrics for each digit separately:

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9965	0.9748744	0.9897959	0.9822785	8995	10	25	970
1	0.9971	0.9876543	0.9867841	0.9872190	8851	15	14	1120
2	0.9942	0.9646947	0.9796512	0.9721154	8931	21	37	1011
3	0.9944	0.9808468	0.9633663	0.9720280	8971	37	19	973
4	0.9951	0.9804325	0.9694501	0.9749104	8999	30	19	952
5	0.9939	0.9716232	0.9596413	0.9655950	9083	36	25	856
6	0.9956	0.9711340	0.9832985	0.9771784	9014	16	28	942
7	0.9947	0.9793510	0.9688716	0.9740831	8951	32	21	996
8	0.9918	0.9407115	0.9774127	0.9587110	8966	22	60	952
9	0.9919	0.9734694	0.9454906	0.9592760	8965	55	26	954



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 – those 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 Deskewing

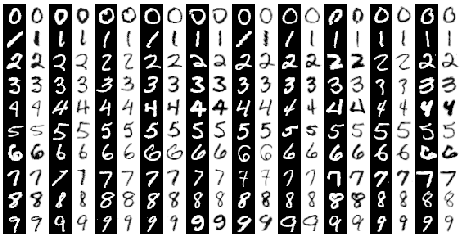
Deskewing of images ("straightening" of the digits) is amongst the most typical transformations that can be applied on MNIST.

Unfortunately, we don't have 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)
```



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.

Deskew everything:

```
Z_train <- X_train
for (i in 1:dim(Z_train)[1]) {
    Z_train[i,,] <- deskew(Z_train[i,,])
}
Z_train2 <- matrix(Z_train, ncol=28*28)


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)
```

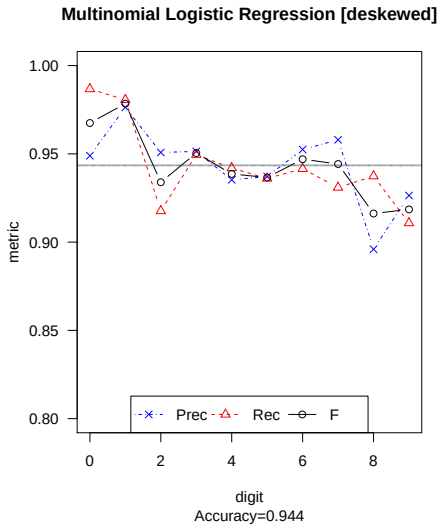
Multinomial logistic regression model (1-layer NN):

```
model <- keras_model_sequential()
layer_dense(model, units=10, activation='softmax')
compile(model, optimizer='sgd',
        loss='categorical_crossentropy')
fit(model, Z_train2, Y_train2, epochs=5)

Y_pred2 <- predict(model, 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] 0.9437
```

Performance metrics for each digit separately:

i	Acc	Prec	Rec	F	TN	FN	FP	TP
0	0.9935	0.9489696	0.9867347	0.9674837	8968	13	52	967
1	0.9951	0.9763158	0.9806167	0.9784615	8838	22	27	1113
2	0.9866	0.9508032	0.9176357	0.9339250	8919	85	49	947
3	0.9900	0.9513889	0.9495050	0.9504460	8941	51	49	959
4	0.9879	0.9352882	0.9419552	0.9386098	8954	57	64	925
5	0.9887	0.9371493	0.9360987	0.9366237	9052	57	56	835
6	0.9899	0.9524815	0.9415449	0.9469816	8997	56	45	902
7	0.9887	0.9579580	0.9309339	0.9442526	8930	71	42	957
8	0.9833	0.8959764	0.9373717	0.9162067	8920	61	106	913
9	0.9837	0.9264113	0.9108028	0.9185407	8918	90	73	919



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.

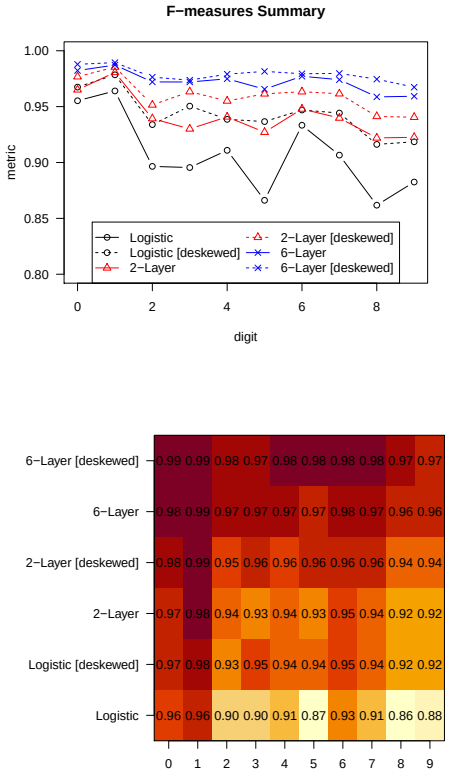
Main drawbacks of deep neural networks:

- 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 will be tackled in more advanced AI/ML courses and books, such as (Goodfellow, Bengio, and Courville 2016).

5.6.2 Beyond MNIST



The MNIST dataset is a classic, although its use in research is discouraged nowadays – 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)
- (Goodfellow, Bengio, and Courville 2016)

Other:

- keras package tutorials available at: https://cran.r-project.org/web/packages/keras/index.html and https://keras.rstudio.com

Chapter 6

Optimisation with Iterative Algorithms

6.1 Introduction

6.1.1 Optimisation Problem

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:

$$f : D \to \mathbb{R}$$

called an **objective function**.

D is the **search space** (problem domain, feasible set) – if defines the set of possible solution candidates.

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

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

In **binary logistic regression** we were minimising the cross-entropy:

$$\min_{(\beta_0,\beta_1,\ldots,\beta_p)\in\mathbb{R}^{(p+1)}} -\frac{1}{n}\sum_{i=1}^{n} \left(\begin{array}{c} y_i \log\left(\frac{1}{1+e^{-(\beta_0+\beta_1 x_{i,1}+\cdots+\beta_p x_{i,p}}}\right) + \\ +(1-y_i)\log\left(\frac{e^{-(\beta_0+\beta_1 x_{i,1}+\cdots+\beta_p x_{i,p}}}{1+e^{-(\beta_0+\beta_1 x_{i,1}+\cdots+\beta_p x_{i,p})}}\right) \end{array} \right) .$$

6.1.3 Types of Minima and Maxima

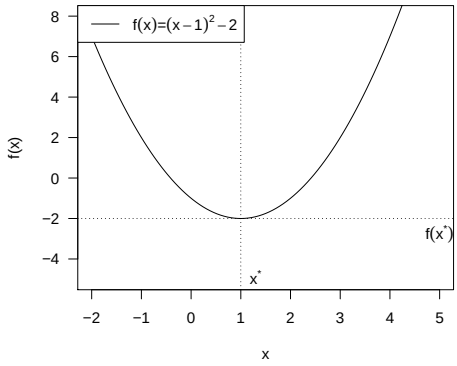
Note that minimising f is the same as maximising $\bar{f} = -f$.

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 $D = \mathbb{R}$, here is an example objective function, $f : \mathbb{D} \to \mathbb{R}$, that has a minimum at $x^* = 1$ with $f(x^*) = -2$.

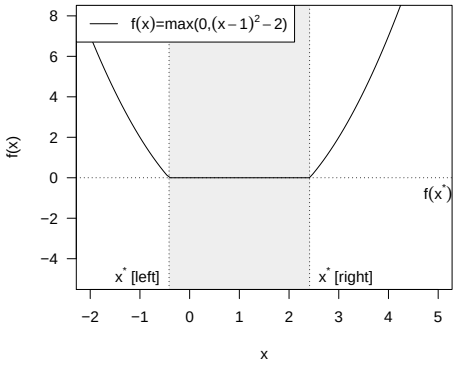


$\min_{x\in\mathbb{R}} f(x) = -2$ (value of f at the minimum)

$\arg\min_{x\in\mathbb{R}} f(x) = 1$ (location of the minimum)

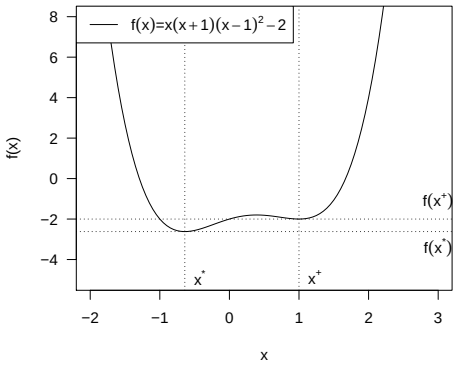
By definition, a minimum/maximum **might not necessarily be unique**. This depends on a problem.

Assuming that $D = \mathbb{R}$, here is an example objective function, $f : \mathbb{D} \to \mathbb{R}$, that has multiple minima; every $x^* \in [1 - \sqrt{2}, 1 + \sqrt{2}]$ yields $f(x^*) = 0$.



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**.



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 $\mathbf{x} \in B(\mathbf{x}^+)$.

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., $(x - r, x + r)$

(*) 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).

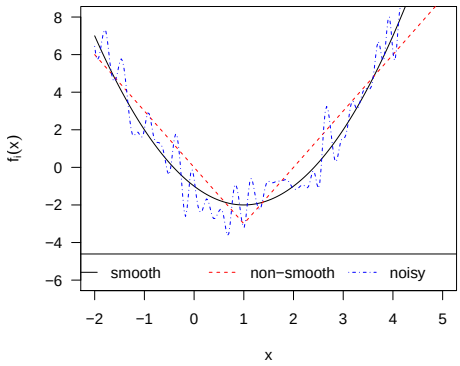
To avoid ambiguity, the "true" minimum (a point x^* such that $f(x^*) \leq f(x)$ for all $x \in D$) is sometimes also referred to as a **global** minimum.

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?

Smooth vs non-smooth vs noisy objectives:



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:

$$g(x_1, x_2) = \log\left((x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\ldots\right)$$

```
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:

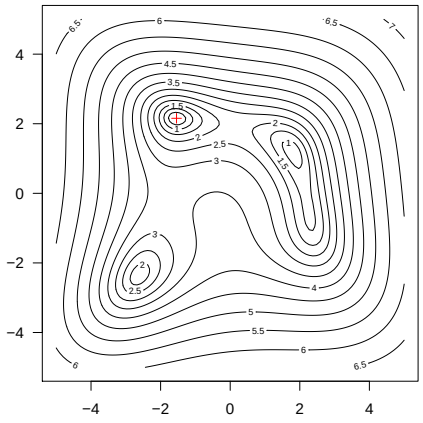
```
g(-1.542255693195422641930153, 2.156405289793087261832605)
```

```
## [1] 0
```

Let's explore various ways of depicting f.

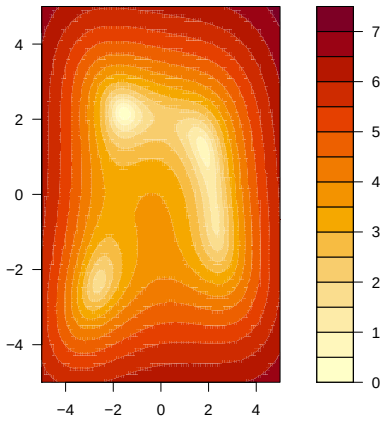
A contour plot:

```
contour(x1, x2, y, las=1, nlevels=25)
points(-1.54226, 2.15641, col=2, pch=3)
```



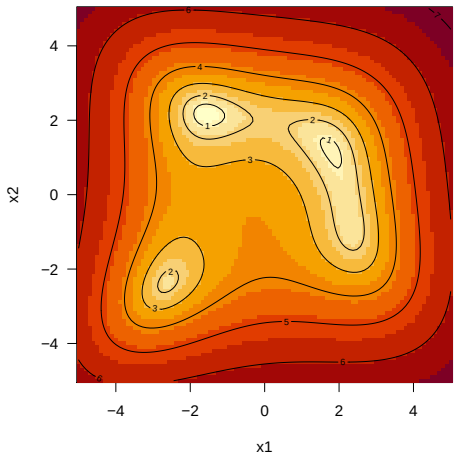
A filled contour plot (a heatmap):

```
filled.contour(x1, x2, y, las=1)
```



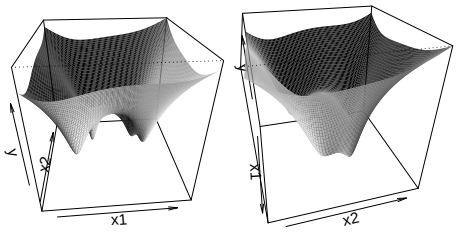
Alternatively:

```
image(x1, x2, y, las=1)
contour(x1, x2, y, add=TRUE)
```



A perspective plot:

```
par(mfrow=c(1,2)) # 2 in 1
persp(x1, x2, y, las=1, phi=30, theta=-5, shade=2, border=NA)
persp(x1, x2, y, las=1, phi=30, theta=75, shade=2, border=NA)
```



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: cycling from the Deakin University's Burwood Campus to the CBD not knowing the route and with GPS disabled – you'll have to ask many people along the way, but you'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 $i = 1, ..., M$:
   a. $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} +$ [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
- ε = tolerance, e.g, 10^{-8}

6.2.2 Example in R

R has a built-in `optim()` function that provides an implementation of (amongst others) **the BFGS method** (proposed by Broyden, Fletcher, Goldfarb and Shanno in 1970).

> (*) 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. You will be able to derive similar algorithms (called quasi-Newton ones) yourself once you get to know about Taylor series approximation by 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 (it's too early for this). However, this will still enable us to identify a few interesting behavioural 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")
```

```
res
```

```
## $par
## [1] -1.542255  2.156405
##
## $value
## [1] 1.413092e-12
##
## $counts
## function gradient
##      101       21
##
## $convergence
## [1] 0
##
## $message
## NULL
```

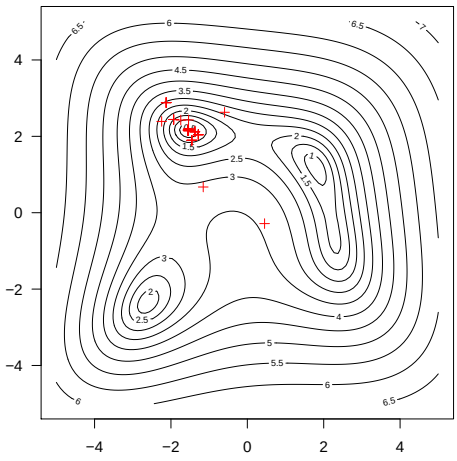
`par` gives the location of the local minimum found

`value` gives the value of g at `par`

We can even depict the points that the algorithm is "visiting":

(*) 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, las=1, nlevels=25)
res <- optim(x12_init, g_vectorised_plot, method="BFGS")
```



6.2.3 Convergence to Local Optima

We were lucky, because the local minimum that the algorithm has found coincides with the global minimum.

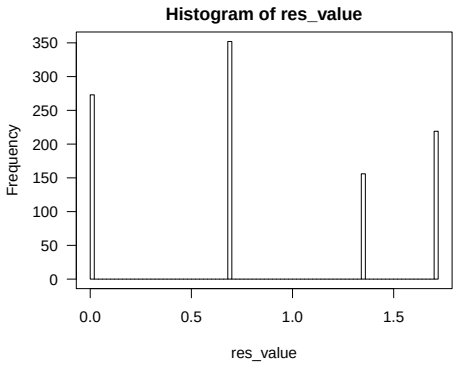
Let's see where does the algorithm converge if we start it from many randomly chosen points uniformly distributed over the square $[-5, 5] \times [-5, 5]$:

```
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))
```

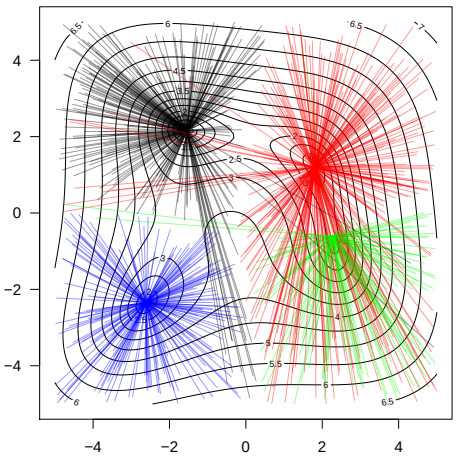
```
##
##     0 0.695 1.356 1.705
##   273   352   156   219
```

We find the global minimum only in $\sim 25\%$ cases! :(

```
hist(res_value, las=1, breaks=100)
box()
```



Here is a depiction of all the random starting points and where do we converge from them:



6.2.4 Random Restarts

A "remedy": 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)
            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
```

Can we guarantee that the global minimum will be found within N tries? **No.**

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 direction where the function decreases the fastest.

(*) Gradient of $f : \mathbb{R}^p \to \mathbb{R}$, denoted $\nabla f : \mathbb{R}^p \to \mathbb{R}^p$ is the vector of

all its partial derivatives, (∇ – 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 $f(x_1, ..., x_p)$, 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 $x_1, ..., x_{i-1}, x_{i+1}, ..., x_p$ are assumed constant.

Function differentiation is an important concept – see how it's referred to in, e.g., the Keras manual at https://keras.rstudio.com/reference/fit.html. Don't worry though – we take our time with this – Melbourne wasn't built in a day.

Recall our g function defined above:

$$g(x_1, x_2) = \log\left((x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\ldots\right)$$

It can be shown (*) that:

$$\begin{aligned} \frac{\partial g}{\partial x_1}(x_1, x_2) &= \frac{4x_1(x_1^2 + x_2 - 5) + 2(x_1 + x_2^2 - 3) + 2x_1}{(x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\ldots} \\ \frac{\partial g}{\partial x_2}(x_1, x_2) &= \frac{2(x_1^2 + x_2 - 5) + 4x_2(x_1 + x_2^2 - 3)}{(x_1^2 + x_2 - 5)^2 + (x_1 + x_2^2 - 3)^2 + x_1^2 - 1.60644\ldots} \end{aligned}$$

```
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 unable to derive the gradient analytically, we can rely on its finite differences approximation:

$$\frac{\partial f}{\partial x_i}(x_1, \ldots, x_p) \simeq \frac{f(x_1, ..., x_i + \delta, ..., x_p) - f(x_1, ..., x_i, ..., x_p)}{\delta}$$

for some small $\delta > 0$, say, $\delta = 10^{-6}$.

Example implementation:

```
# gradient of f at x=c(x[1],...,x[p])
grad <- 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
}
```

> (*) 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(g_vectorised, c(-2, 2))
```

```
## [1] -3.186485 -1.365634
```

```
grad_g_vectorised(c(-2, 2))
```

```
## [1] -3.186485 -1.365636
```

```
grad(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).

Fact 2.

The gradient of f at $\mathbf{x}$, $\nabla f(\mathbf{x})$, is a vector that points in the direction of the steepest slope.

Minus gradient, $-\nabla f(\mathbf{x})$, is the direction where the function decreases the fastest.

> (*) 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**, η.

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 $\nabla f(\mathbf{x}^*) = [0, \dots, 0]$.

> (***) 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)

An implementation of the gradient descent algorithm:

```
# 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
```

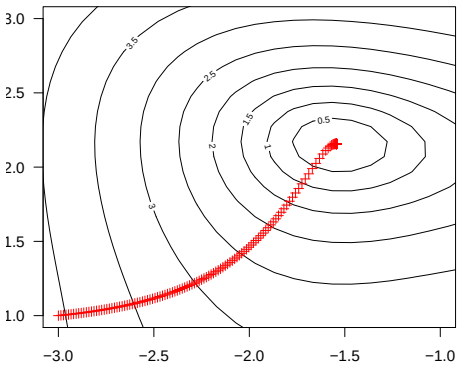
```
        f_last <- f_cur
    }
    list( # see ?optim, section `Value`
        par=par,
        value=g_vectorised(par),
        counts=i,
        convergence=as.integer(i==maxit)
    )
}
```

Tests of the g function:

```
eta <- 0.01
optim_gd(c(-3,1), g_vectorised, grad_g_vectorised, eta=eta)
```

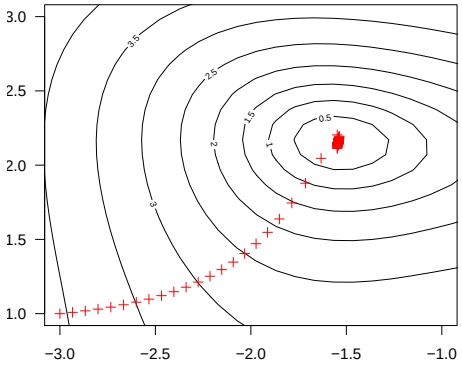
```
## $par
## [1] -1.542291  2.156410
##
## $value
## [1] 1.332582e-08
##
## $counts
## [1] 135
##
## $convergence
## [1] 0
```

Zooming in the contour plot to see the actual path ($\eta = 0.01$):



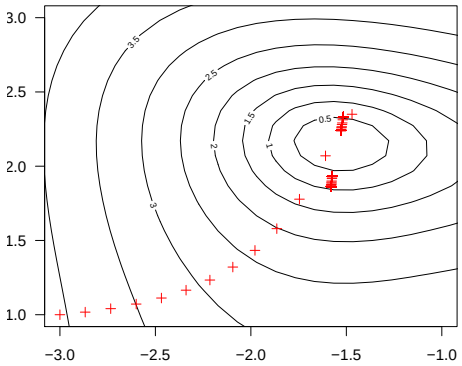

```
## List of 4
##  $ par        : num [1:2] -1.54 2.16
##  $ value      : num 1.33e-08
##  $ counts     : int 135
##  $ convergence: int 0
```

Now with $\eta = 0.05$:




```
## List of 4
##  $ par        : num [1:2] -1.54 2.15
##  $ value      : num 0.000203
##  $ counts     : int 417
##  $ convergence: int 0
```

And now with $\eta = 0.1$:




```
## List of 4
##  $ par        : num [1:2] -1.52 2.33
##  $ value      : num 0.507
##  $ counts     : int 1000
##  $ convergence: int 1
```

If the learning rate η is too small, the convergence might be too slow and we might get stuck in a plateau.

On the other hand, if η 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 η, decreasing it in every iteration, say, by some percent;
- *line search* – determine optimal η 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 too accelerate search in the relevant direction and minimise oscillations.

6.3.4 Example: MNIST

Recall that 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)
```

Recall that the output of the logistic regression model (1-layer neural network with softmax) can be written in the matrix form as:

$$\hat{\mathbf{Y}} = \text{softmax}\left(\dot{\mathbf{X}}\,\mathbf{B}\right),$$

where $\dot{\mathbf{X}} \in \mathbb{R}^{n\times 785}$ is a matrix representing n images of size 28×28, 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))
```

```
softmax <- function(T) {
    T <- exp(T)
    T/rowSums(T)
}

nn_predict <- function(B, X) {
    softmax(X %*% B)
}
```

Define the functions to compute cross-entropy (which we shall minimise) and accuracy (which we shall report to a user):

```
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)
}

cross_entropy <- function(Y_true, Y_pred) {
    -sum(Y_true*log(Y_pred))/nrow(Y_true)
}
```

(*) Cross-entropy in non-matrix form (n – number of samples, K – number of classes, $p+1$ – number of model parameters; in our case $K = 10$ and $p = 784$):

$$\begin{aligned} E(\mathbf{B}) &= -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{i,k}\log\left(\frac{\exp\left(\sum_{j=0}^{p}\dot{x}_{i,j}\beta_{j,k}\right)}{\sum_{c=1}^{K}\exp\left(\sum_{j=0}^{p}\dot{x}_{i,j}\beta_{j,c}\right)}\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(\log\left(\sum_{k=1}^{K}\exp\left(\sum_{j=0}^{p}\dot{x}_{i,j}\beta_{j,k}\right)\right) - \sum_{k=1}^{K} y_{i,k}\sum_{j=0}^{p}\dot{x}_{i,j}\beta_{j,k}\right) \end{aligned}$$

(***) Partial derivative of cross-entropy w.r.t. $\beta_{a,b}$ in non-matrix form:

$$\begin{aligned} \frac{\partial E}{\partial \beta_{a,b}}(\mathbf{B}) &= \frac{1}{n}\sum_{i=1}^{n}\dot{x}_{i,a}\left(\frac{\exp\left(\sum_{j=0}^{p}\dot{x}_{i,j}\beta_{j,b}\right)}{\sum_{k=1}^{K}\exp\left(\sum_{j=0}^{p}\dot{x}_{i,j}\beta_{j,k}\right)} - y_{i,b}\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}\dot{x}_{i,a}\left(\hat{y}_{i,b} - y_{i,b}\right) \end{aligned}$$

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:

$$\frac{1}{n}\dot{\mathbf{X}}^T\left(\hat{\mathbf{Y}} - \mathbf{Y}\right)$$

```
grad_cross_entropy <- function(X, Y_true, Y_pred) {
    t(X) %*% (Y_pred-Y_true)/nrow(Y_true)
}
```

> Luckily, we are not overwhelmed with the above, because we can always substitute the gradient with the finite differences (yet, these will be slower). :)

Let us implement the gradient descent method:

```
# 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
##  81.712  27.060  41.009
```

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
```

6.3.5 Stochastic Gradient Descent (SGD)

In turns out that there's a simple cure for that.

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 really want is to minimise the test error** [which we cannot refer to while training = no cheating!]

It is 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 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?

$$E(\mathbf{B}) = -\frac{1}{b}\sum_{i=1}^{b} \log \Pr(Y = y^{\text{train}}_{\text{random\_index}_i} | \mathbf{x}^{\text{train}}_{\text{random\_index}_i,.}, \mathbf{B}),$$

where b is some fixed batch size.

Such a scheme is often called **stochastic gradient descent**.

> 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:

```
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
##   0.155   0.060   0.084
```

```
accuracy(Y_train2, nn_predict(B, X_train1))
```

```
## [1] 0.40435
```

```
accuracy(Y_test2,  nn_predict(B, X_test1))
```

```
## [1] 0.4123
```

The errors are slightly worse but that was very quick.

Why don't we increase the number of iterations?

```
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
##   8.203   0.132   8.193
```

```
accuracy(Y_train2, nn_predict(B, X_train1))
```

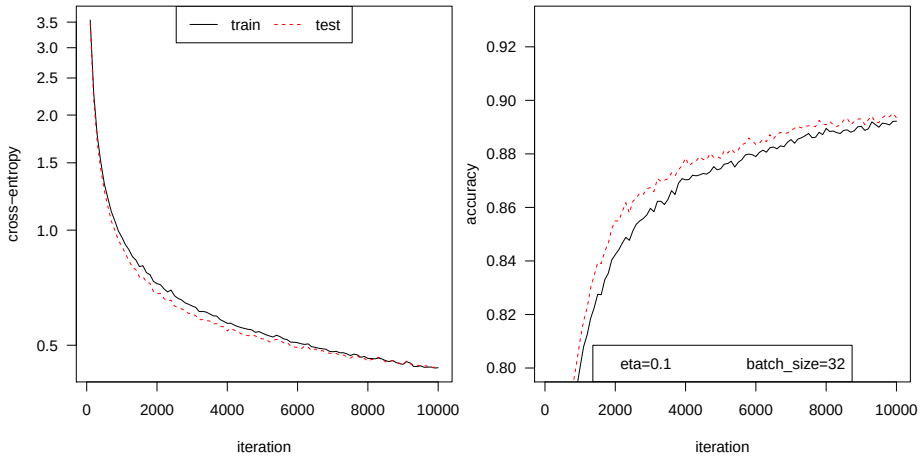
```
## [1] 0.8932667
```

```
accuracy(Y_test2,  nn_predict(B, X_test1))
```

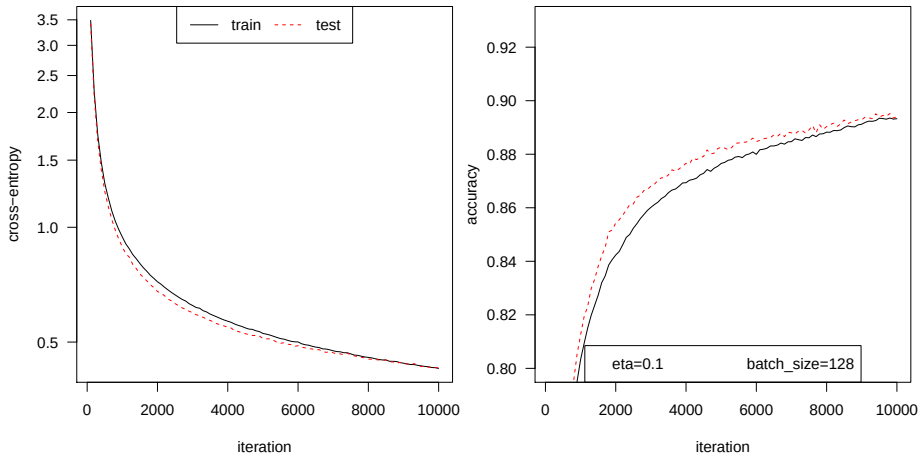
```
## [1] 0.8939
```

This is great.

Let's take a closer look at how the train/test error behaves in each iteration for different batch sizes.




```
##    user  system elapsed
##  67.989  14.658  35.058
```




```
##    user  system elapsed
## 149.924  54.333  57.563
```

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.

> (*) Good luck fitting a logistic regression model to MNIST with `optim()`'s BFGS – there are 7850 variables.

Training deep neural networks with SGD is slow too, but there is a trick to propagate weight updates layer by layer, called *backpropagation* (actually used in every neural network library), see, e.g., (Goodfellow, Bengio, and Courville 2016).

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).

6.4.2 Optimisers in Keras

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 fancy 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 will get into their details in a dedicated course covering deep neural networks in more detail (see, e.g., (Goodfellow, Bengio, and Courville 2016)), but you already have developed some good intuitions!

6.4.3 Note on Search Spaces

Most often, the choice of the search space D in an continuous optimisation problem can be:

- $D = \mathbb{R}^p$ – continuous unconstrained (typical in ML)
- $D = [a_1, b_1] \times \cdots \times [a_n, b_n]$ – continuous with box constraints

  see `method="L-BFGS-B"` in `optim()`
- constrained with k linear inequality constraints

  $a_{1,1}x_1 + \cdots + a_{1,p}x_p \leq b_1, \ldots, a_{k,1}x_1 + \cdots + a_{k,p}x_p \leq b_k$

  (*) supported in linear and quadratic programming solvers, where the objective function is from a very specific class

6.4.4 Further Reading

Recommended further reading:

- (Nocedal and Wright 2006)
- (Fletcher 2008)

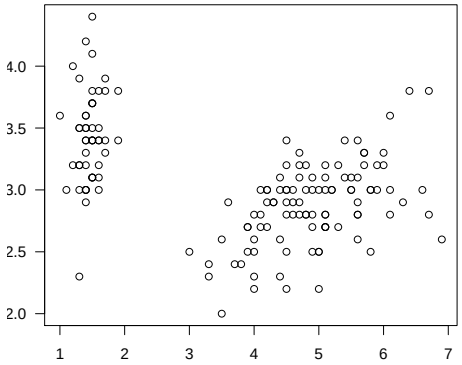
Chapter 7

Clustering

7.1 Unsupervised Learning

7.1.1 Introduction

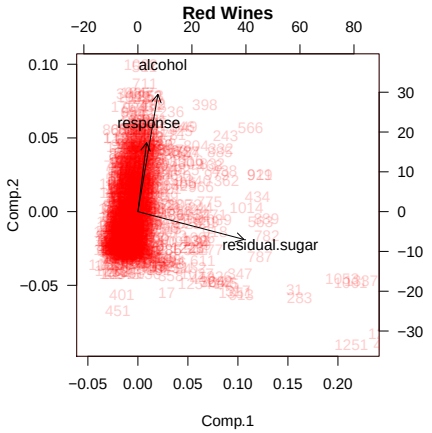
In **unsupervised learning** (learning without a teacher), the input data points $\mathbf{x}_{1,\cdot}, \ldots, \mathbf{x}_{n,\cdot}$ are not assigned any reference labels.



Our aim now is to discover the **underlying structure in the data**.

7.1.2 Main Types of Unsupervised Learning Problems

In **dimensionality reduction** we seek a meaningful *projection* of a high dimensional space (think: many variables/columns).

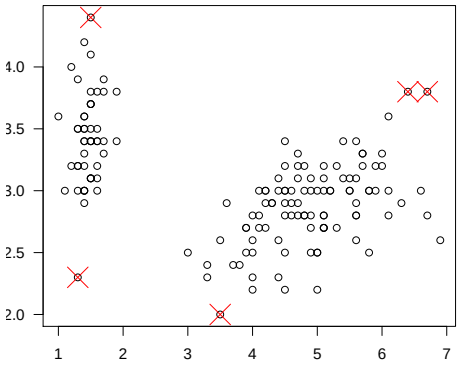


This might enable us to plot high-dimensional data or understand its structure better.

Example methods:

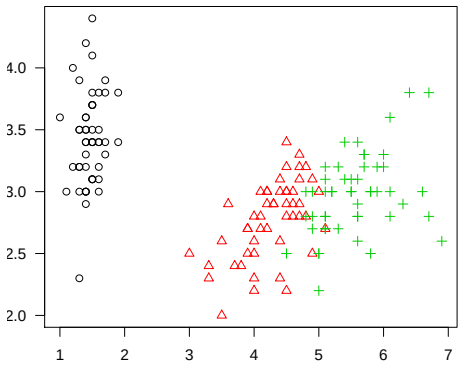
- Multidimensional scaling (MDS)
- Principal component analysis (PCA)
- Kernel PCA
- t-SNE
- Autoencoders (deep learning)

In **anomaly detection**, out 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.

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 Clustering

More formally, given $K \geq 2$, **clustering** aims is to find a *special kind* of a K**-partition** of the input data set $\mathbf{X}$.

$\mathcal{C} = \{C_1, \ldots, C_K\}$ is a K-partition of $\mathbf{X}$ of size n, whenever:

- $C_k \neq \emptyset$ for all k (each set is nonempty),
- $C_k \cap C_l = \emptyset$ for all $k \neq l$ (sets are pairwise disjoint),
- $\bigcup_{k=1}^{K} C_k = \{1, \ldots, n\}$ (no point is neglected).

This can also be thought of as assigning each point a unique label $\{1, \ldots, K\}$ (think: colouring of the points, where each number has a colour).

> "$\mathbf{x}_{i,\cdot}$ is labelled j iff it belongs to cluster C_j, i.e., $i \in C_j$".

Example applications of clustering:

- *taxonomization*: 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*:

$$\left\{ {n \atop K} \right\} = \frac{1}{K!} \sum_{j=0}^{K} (-1)^{K-j} \binom{K}{j} j^n;$$

e.g., already $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$ and $\left\{ {n \atop 3} \right\} = O(3^n)$ – that is a lot.

Certainly, we are not just interested in "any" partition.

However, even one of the most famous textbooks provides us with only a vague hint:

> 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, Tibshirani, and Friedman 2017).

There are 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-maximization 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 clustering approaches:

- *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

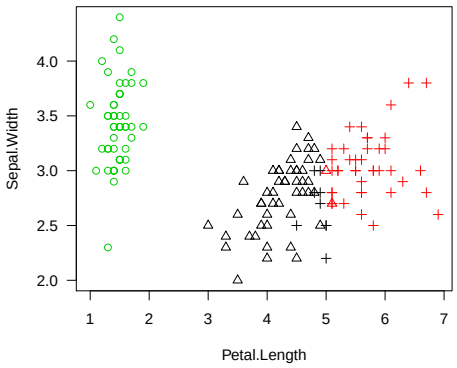
Let us apply K-means clustering to find $K = 3$ groups in the famous Fisher's `iris` data set (variables `Sepal.Width` and `Petal.Length` variables only)

```
X <- as.matrix(iris[,c(3,2)])
# never forget to set nstart>>1!
km <- kmeans(X, centers=3, nstart=10)
km$cluster # assigned labels
```

```
##   [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [30] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 1 1 1 1
##  [59] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1
##  [88] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1 2 2 2
## [117] 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2
## [146] 2 1 2 2 2
```

> Later we'll see that `nstart` is responsible for random restarting the (local) optimisation procedure, just like we did in the previous chapter.

```
plot(X, col=km$cluster, pch=as.integer(iris$Species), las=1)
```



The colours indicate the detected clusters,

while the plotting characters – the true iris species

> Note that they were not used during the clustering procedure! (we're dealing with unsupervised learning)

A contingency table for detected vs. true clusters:

```
(C <- table(km$cluster, iris$Species))
```

```
##
##     setosa versicolor virginica
##   1      0         48         9
##   2      0          2        41
##   3     50          0         0
```

```
sum(apply(C, 1, max))/sum(C) # accuracy
```

```
## [1] 0.9266667
```

The discovered partition matches the original one very well.

7.2.2 Problem Statement

The aim of K-means clustering is to find K "good" cluster centres $\boldsymbol{\mu}_{1,\cdot}, \ldots, \boldsymbol{\mu}_{K,\cdot}$.

Then, a point $\mathbf{x}_{i,\cdot}$ will be assigned to the cluster represented by the closest centre.

Closest == w.r.t. the squared Euclidean distance.

Assuming all the points are in a p-dimensional space, $\mathbb{R}^p$,

$$d(\mathbf{x}_{i,\cdot}, \boldsymbol{\mu}_{k,\cdot}) = \|\mathbf{x}_{i,\cdot} - \boldsymbol{\mu}_{k,\cdot}\|^2 = \sum_{j=1}^{p} (x_{i,j} - \mu_{k,j})^2$$

The i-th point's cluster is determined by:

$$\mathrm{C}(i) = \arg \min_{k=1,\ldots,K} d(\mathbf{x}_{i,\cdot}, \boldsymbol{\mu}_{k,\cdot}),$$

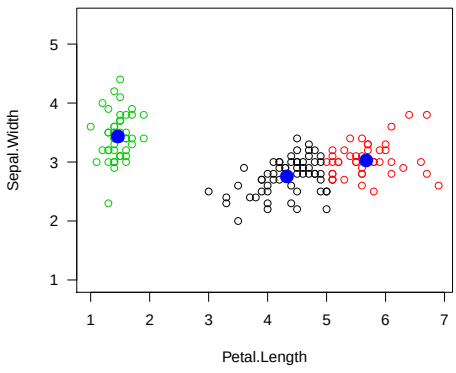
where arg min == argument minimum == the index k that minimises the given expression.

In the previous example, we have:

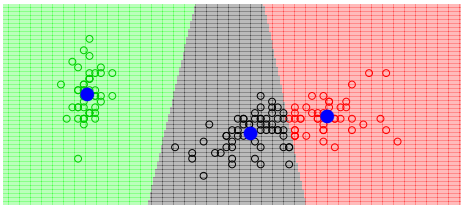
```
km$centers
```

```
##   Petal.Length Sepal.Width
## 1     4.328070    2.750877
## 2     5.672093    3.032558
## 3     1.462000    3.428000
```

```
plot(X, col=km$cluster, las=1, asp=1) # asp=1 gives the same scale on both axes
points(km$centers, cex=2, col=4, pch=16)
```



Here is the partition of the whole $\mathbb{R}^2$ space into clusters based on the closeness to the three cluster centres:



(*) For the interested, see "Voronoi diagrams".

To compute the pairwise distances, we may call `pdist::pdist()`:

```
library("pdist")
D <- as.matrix(pdist(X, km$centers))
head(D) # D[i,j] = distance between x[i,] and μ[j,]
```

```
##          [,1]     [,2]       [,3]
## [1,] 3.022380 4.297590 0.09501583
## [2,] 2.938649 4.272217 0.43246734
## [3,] 3.061196 4.375298 0.27969268
## [4,] 2.849538 4.172638 0.33019394
## [5,] 3.048705 4.309613 0.18283319
## [6,] 2.868316 4.065708 0.52860963
```

...

Therefore, the cluster memberships (arg mins) can be determined by:

```
(idx <- apply(D, 1, which.min))
```

```
##   [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [30] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 1 1 1 1
##  [59] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1
##  [88] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1 2 2
## [117] 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2
## [146] 2 1 2 2 2
```

```
all(km$cluster == idx) # sanity check
```

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

7.2.3 Algorithms for the K-means Problem

How to find "good" cluster centres?

In the K-means clustering, we wish to find $\boldsymbol{\mu}_{1,\cdot}, \ldots, \boldsymbol{\mu}_{K,\cdot}$ that minimise the total within-cluster distances (distances from each point to each own cluster centre):

$$\min_{\boldsymbol{\mu}_{1,\cdot},\ldots,\boldsymbol{\mu}_{K,\cdot} \in \mathbb{R}^p} \sum_{i=1}^{n} d(\mathbf{x}_{i,\cdot}, \boldsymbol{\mu}_{C(i),\cdot}),$$

Note that the $\boldsymbol{\mu}$s are also "hidden" inside the point-to-cluster belongingness mapping, C.

Expanding the above yields:

$$\min_{\boldsymbol{\mu}_{1,\cdot},\ldots,\boldsymbol{\mu}_{K,\cdot} \in \mathbb{R}^p} \sum_{i=1}^{n} \left(\min_{k=1,\ldots,K} \sum_{j=1}^{p} (x_{i,j} - \mu_{k,j})^2 \right).$$

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.

`kmeans()` in R implements the Hartigan-Wong, Lloyd, Forgy and MacQueen algorithms.

> (*) Technically, there is no such thing as "the K-means algorithm" – all the above are particular heuristic approaches to solving the K-means clustering problem as specified by the aforementioned optimisation task.
>
> By setting `nstart = 10` above, we ask the (Hartigan-Wong) 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 distance) amongst them. This does not guarantee finding the optimal solution, especially in high-dimensional spaces, but increases the likelihood of such.

Lloyd's algorithm (1957) is sometimes referred to as "the" K-means algorithm:

1. Start with random cluster centres $\boldsymbol{\mu}_{1,\cdot}, \ldots, \boldsymbol{\mu}_{K,\cdot}$.
2. For each point $\mathbf{x}_{i,\cdot}$, determine its closest centre $C(i) \in \{1, \ldots, K\}$.

3. For each cluster $k \in \{1, \ldots, K\}$, compute the new cluster centre $\boldsymbol{\mu}_{k,\cdot}$ as the componentwise arithmetic mean of the coordinates of all the point indexes 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.

(*) Example implementation:

```
K <- 3

# Random initial cluster centres
M <- jitter(X[sample(1:nrow(X), K),])
M
```

```
##      Petal.Length Sepal.Width
## [1,]     1.106172    2.994400
## [2,]     1.394794    3.321308
## [3,]     6.754820    3.808716
```

```
# Let D[i,k] bet the distance between the i-th point and the k-th centre:
D <- as.matrix(pdist(X, M))
# Let idx[i] be the centre closest to the i-th point
idx <- apply(D, 1, which.min)
```

```
repeat {
    # Previous cluster belongingness:
    old_idx <- idx
    # Split X into a list of K data frames based on old_idx info
    X_split <- split(as.data.frame(X), old_idx)
    # Compute componentwise arithmetic means of each data frame - new centres
    M <- t(sapply(X_split, colMeans))
    # Recompute cluster belongingness
    D <- as.matrix(pdist(X, M))
    idx <- apply(D, 1, which.min)
    # Check if converged already:
    if (all(idx == old_idx)) break
}
```

```
M # our result
```

```
##   Petal.Length Sepal.Width
## 1     1.462000    3.428000
## 2     4.328070    2.750877
## 3     5.672093    3.032558
```

```
km$center # result of kmeans()
```

```
##   Petal.Length Sepal.Width
## 1     4.328070    2.750877
## 2     5.672093    3.032558
## 3     1.462000    3.428000
```

7.3 Hierarchical Methods

7.3.1 Introduction

In K-means, we need to specify the number of clusters, K, in advance.

What if we don't know it?

There is no guarantee that a $(K+1)$-partition is "similar" to the K-one.

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.

Here we are interested in **agglomerative** 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 containing all the points.

Moving from the i-th to the $(i+1)$-th level, we select a pair of clusters to be merged.

7.3.2 Example in R

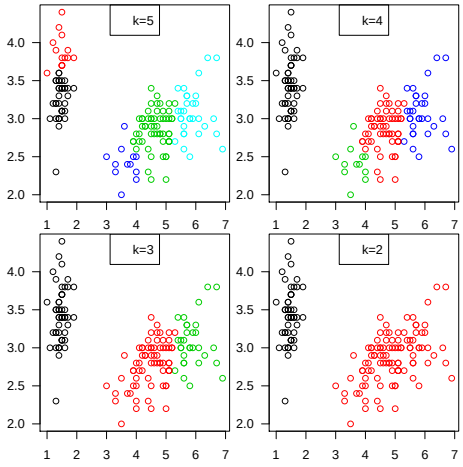
```
# Distances between all pairs of points:
D <- dist(X)
# Apply Complete Linkage (the default, see below):
h <- hclust(D) # method="complete"
print(h)
```

```
##
## Call:
## hclust(d = D)
##
## Cluster method   : complete
## Distance         : euclidean
## Number of objects: 150
```

```
cutree(h, k=3) # extract the 3-partition
```

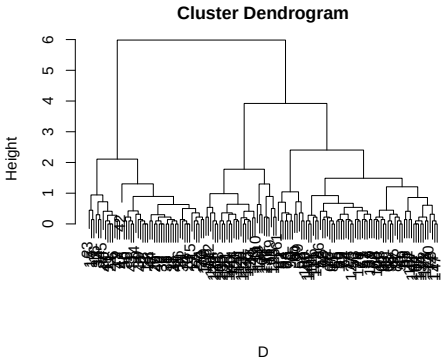
```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [30] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
##  [59] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [88] 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 3 3 3 3 2 3 3 3 2 3 3 2 2 2 3
## [117] 3 3 3 2 3 2 3 2 3 3 2 2 3 3 3 3 3 2 3 3 3 3 2 3 3 2 2 3 3 3
## [146] 2 2 2 3 2
```

Different cuts of the hierarchy:



A **dendrogram** depicts the distances (* as defined by the linkage function) between the clusters merged at every stage. This can provide us with the insight into the underlying data structure.

```
plot(h)
```



7.3.3 Agglomerative Hierarchical Clustering

Initially, $\mathcal{C}^{(0)} = \{\{1\}, \ldots, \{n\}\}$, i.e., each point is a member of its own cluster.

While a **hierarchical agglomerative clustering** algorithm is being computed, there are $n-k$ clusters at the k-th step of the procedure, $\mathcal{C}^{(k)} = \{C_1^{(k)}, \ldots, C_{n-k}^{(k)}\}$.

When proceeding from step k to $k+1$, we determine $C_u^{(k)}$ and $C_v^{(k)}$, $u < v$, to be **merged** together so that we get:

$$\mathcal{C}^{(k+1)} = \left\{C_1^{(k)}, \ldots, C_{u-1}^{(k)}, C_u^{(k)} \cup C_v^{(k)}, C_{u+1}^{(k)}, \ldots, C_{v-1}^{(k)}, C_{v+1}^{(k)}, \ldots, C_{n-k}^{(k)}\right\}.$$

Thus, $(\mathcal{C}^{(0)}, \mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(n-1)})$ is a sequence of **nested** partitions of $\{1, 2, \ldots, n\}$ with $\mathcal{C}^{(n-1)} = \{\{1, 2, \ldots, n\}\}$.

7.3.4 Linkage Functions

A pair of clusters $C_u^{(k)}$ and $C_v^{(k)}$ to be merged with each other is determined by:

$$\arg\min_{u<v} d^*(C_u^{(k)}, C_v^{(k)}),$$

where $d^*(A, B)$ is the *distance* between two clusters A and B.

Note that we usually only consider the distances between single points, not sets of points.

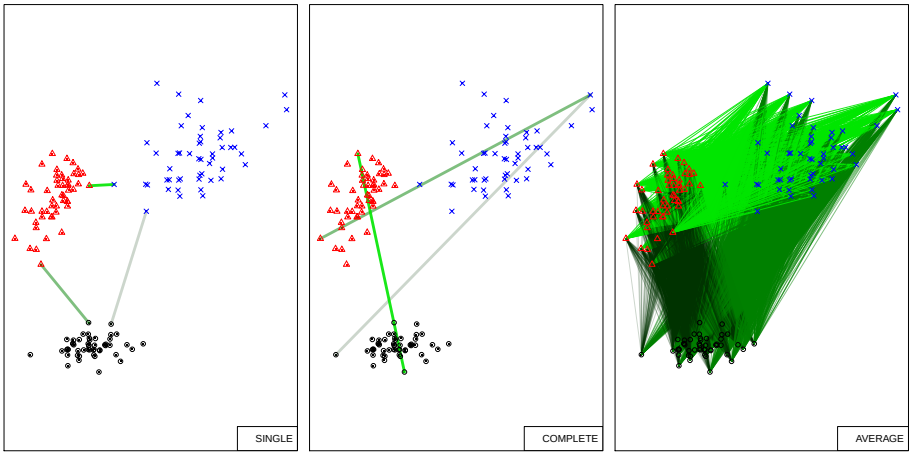
d^* is a suitable extension of a pointwise distance d (usually the Euclidean metric) to whole sets.

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.

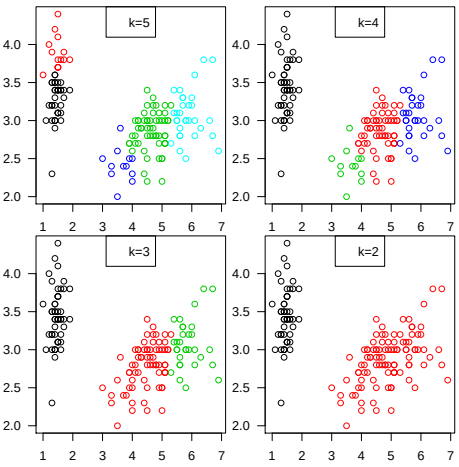
There are many popular choices of d^* (which in the context of hierarchical clustering we call a **linkage function**)

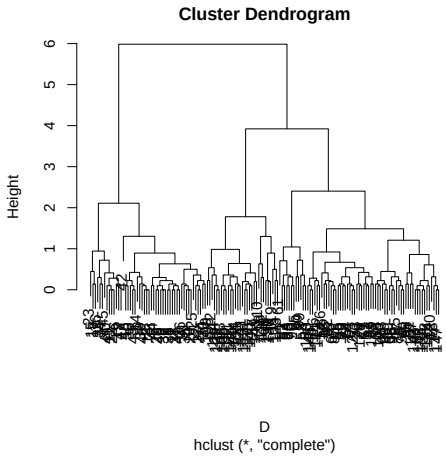
- Single linkage:
$$d_S^*(A, B) = \min_{\mathbf{a}\in A, \mathbf{b}\in B} d(\mathbf{a}, \mathbf{b})$$
- Complete linkage:
$$d_C^*(A, B) = \max_{\mathbf{a}\in A, \mathbf{b}\in B} d(\mathbf{a}, \mathbf{b})$$
- Average linkage:
$$d_A^*(A, B) = \frac{1}{|A||B|} \sum_{\mathbf{a}\in A} \sum_{\mathbf{b}\in B} d(\mathbf{a}, \mathbf{b})$$

Computing linkages – an illustration:

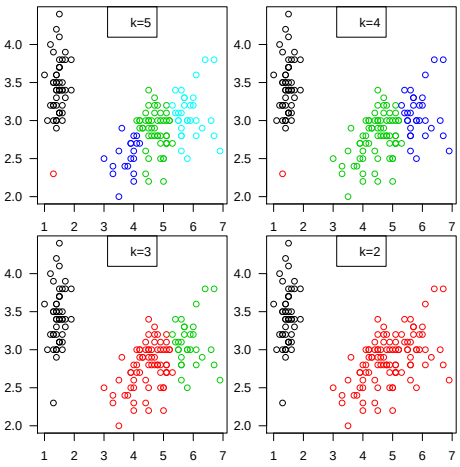


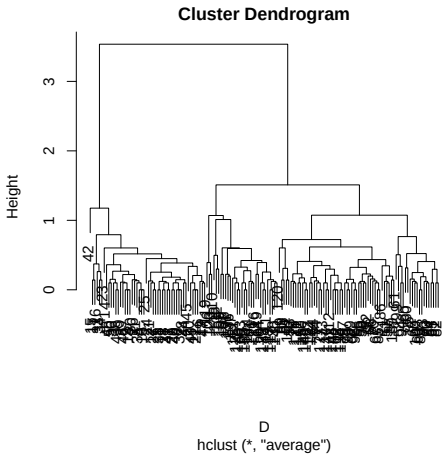
Complete linkage (again):



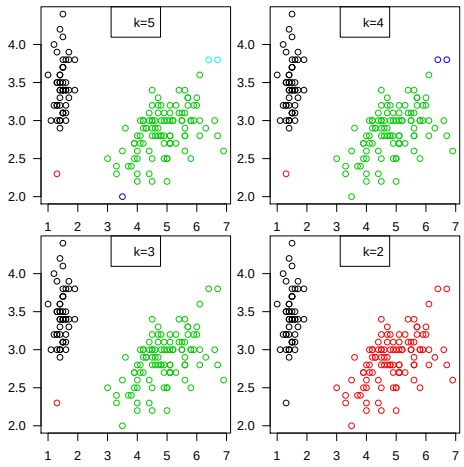


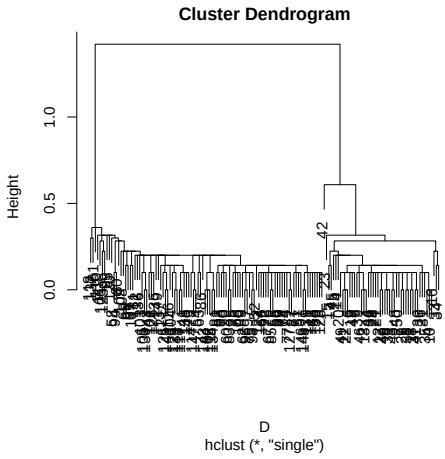
Average linkage:





Single linkage (this is a typical behaviour!):





7.4 Outro

7.4.1 Remarks

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 obtained a "fine grained" data compression and then combine the (sub)clusters into more meaningful groups using other methods.

Iterative K-means algorithms are very fast even for large data sets, but they may fail to find a desirable solution, see the next chapter for discussion.

On the other hand, hierarchical methods output a whole family of mutually nested partitions, which may provide us with insight into the underlying data structure.

Unfortunately, there is no easy way to assign new points to existing clusters.

Linkage scheme must be chosen with care.

These are generally slow – $O(n^2)$ to $O(n^3)$ complexity.

> Note that the `fastcluster` package provides a more efficient implementation of some methods available via a call to `hclust()`. See also the `genie` package for a robust algorithm based on the minimum spanning tree, which can be computed quickly.

```
##
## Attaching package: 'fastcluster'

## The following object is masked from 'package:stats':
##
##     hclust
```

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.

Moreover, some unsupervised methods do not easily generalise to unobserved data.

Also many clustering methods are based on the notion of pairwise distances but these tend to behave weirdly in high-dimensional spaces ("the curse of dimensionality").

However, clustering methods can aid with supervised tasks – instead of fitting a single "large model", it might be useful to fit separate models to each cluster.

7.4.2 Other Noteworthy Clustering Algorithms

Other noteworthy clustering approaches:

- Genie (see R package `genie`)
- DBSCAN, HDBSCAN*
- k-medoids, k-medians
- Spectral clustering
- BIRCH

7.4.3 Further Reading

Recommended further reading:

- (James et al. 2017: Section 10.3)

Other:

- (Hastie, Tibshirani, and Friedman 2017: Section 14.3)

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** $f : D \to \mathbb{R}$:

$$\min_{x \in D} f(x) \quad \text{or} \quad \max_{x \in D} f(x),$$

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 $i = 1, ..., M$:
    a. $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} +$ [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
- ε = 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:

$$\min_{\boldsymbol{\mu}_{1,\cdot},\ldots,\boldsymbol{\mu}_{K,\cdot}\in\mathbb{R}^p} \sum_{i=1}^{n} \left(\min_{k=1,\ldots,K} \sum_{j=1}^{p} (x_{i,j} - \mu_{k,j})^2 \right).$$

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 $\boldsymbol{\mu}_{1,\cdot}, \ldots, \boldsymbol{\mu}_{K,\cdot}$.
2. For each point $\mathbf{x}_{i,\cdot}$, determine its closest centre $C(i) \in \{1, \ldots, K\}$.
3. For each cluster $k \in \{1, \ldots, K\}$, compute the new cluster centre $\boldsymbol{\mu}_{k,\cdot}$ as the componentwise arithmetic mean of the coordinates of all the point indexes 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.

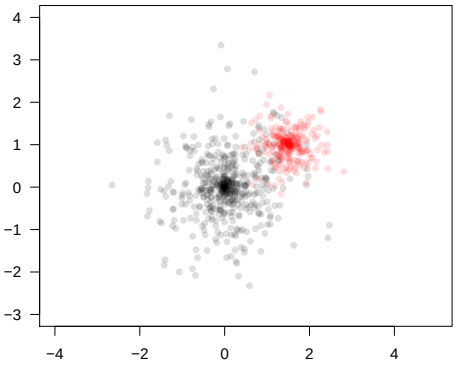
We will need some benchmark data.

```
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:

```
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:

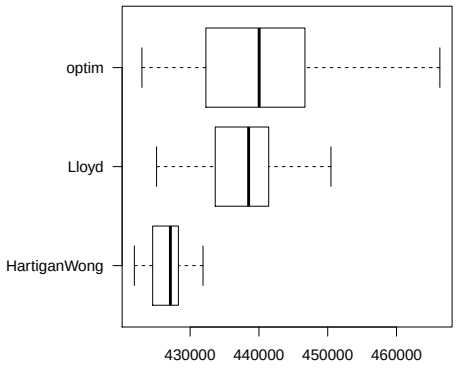
```
min_HartiganWong <- function(mu0, X)
    get_fitness(
        # algorithm="Hartigan-Wong"
        kmeans(X, mu0, iter.max=100)$centers,
    X)
min_Lloyd <- function(mu0, X)
    get_fitness(
        kmeans(X, mu0, iter.max=100, algorithm="Lloyd")$centers,
    X)
min_optim <- function(mu0, X)
    optim(mu0,
        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)
```




```
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    1.130455    1.145806    1.176184    1.186706
## 2        Lloyd    1.000000    1.000000    1.000000    1.000000
## 3        optim 1638.084281 1579.560210 1533.456729 1553.742477
##            uq         max neval
## 1    1.246964    1.195926    10
## 2    1.000000    1.000000    10
## 3 1539.866205 1362.555916    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 and 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**.

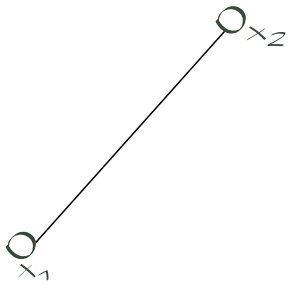
8.2.2 Convex Combinations (*)

A **convex combination** of a set of points $x_1, \ldots, x_n \in D$ is a *linear combination*

$$\theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

for some $\theta_1, \theta_2, \ldots, \theta_n$ that fulfils $\theta_1, \theta_2, \ldots, \theta_n \geq 0$ and $\theta_1 + \theta_2 + \cdots + \theta_n = 1$.

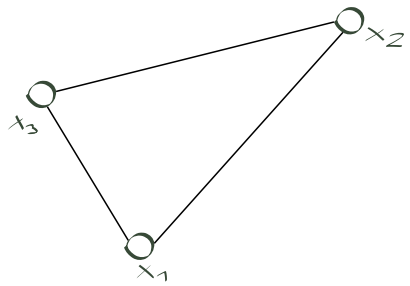
> Think of this as a weighted arithmetic mean of these points.



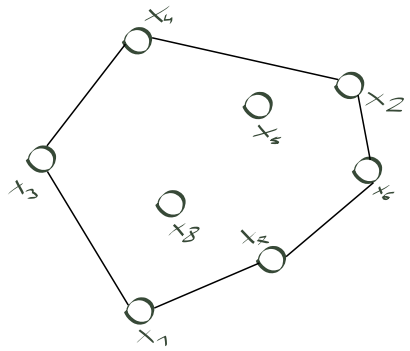
The set of all convex combinations of two points $x_1, x_2 \in D$:

$$\{\theta x_1 + (1-\theta)x_2 : \theta \in [0,1]\}$$

is just the line segment between these two points.

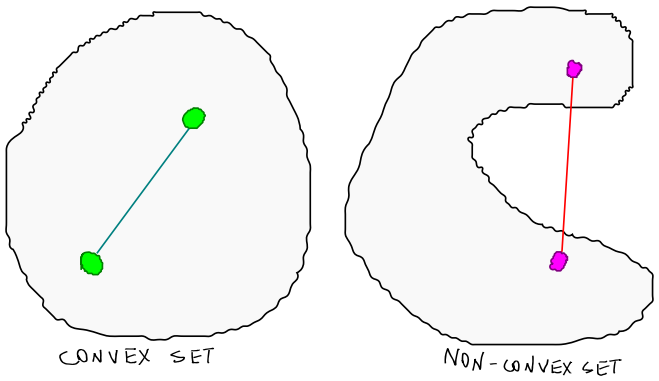


The set of all convex combinations of 3 points yields a triangle (unless D is one-dimensional).



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.

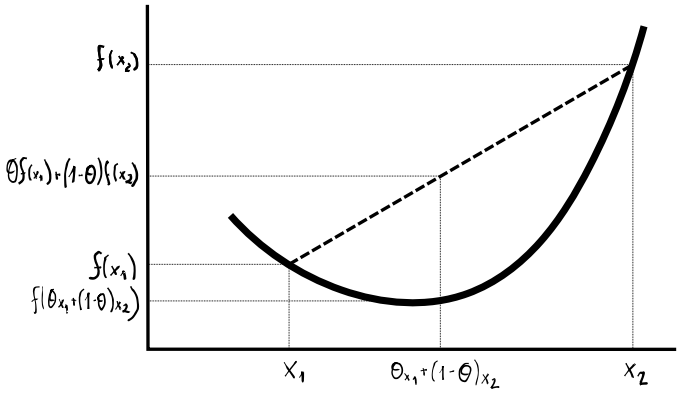
> In two dimensions, think of a rubber band stretched around all the points like a fence.



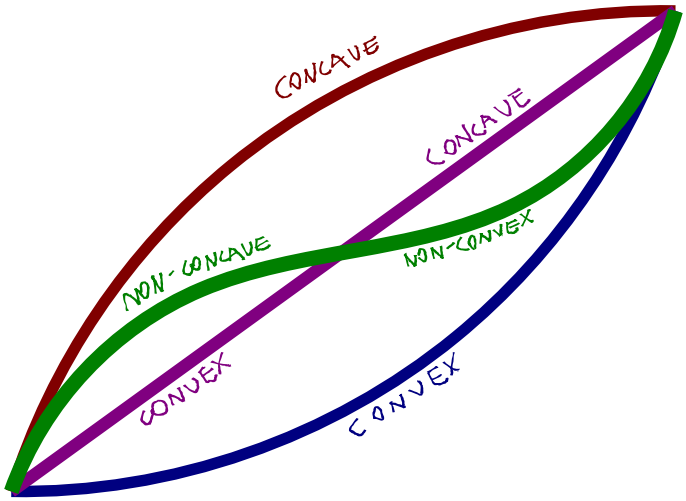
8.2.3 Convex Functions (*)

We call a function $f : D \to \mathbb{R}$ **convex**, whenever:

$$(\forall x_1, x_2 \in D)(\forall \theta \in [0,1]) \quad f(\theta x_1 + (1-\theta)x_2) \leq \theta f(x_1) + (1-\theta) f(x_2)$$



(function value at any convex combination of two points is not greater than that combination of the function values at these two points)



Convex: $f(\theta x_1 + (1-\theta)x_2) \leq \theta f(x_1) + (1-\theta)f(x_2)$

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 and 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_{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 $\mathbb{R}^{K \times p}$.

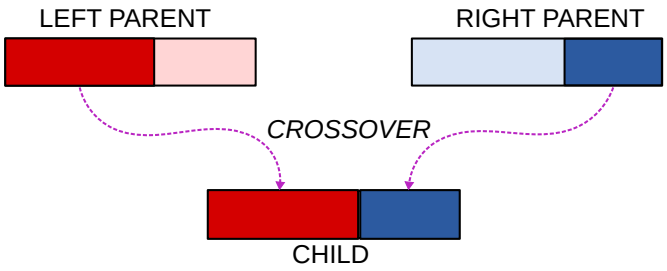
Let's assume that the fitness of each individual should be a function of the rank of the objective function's value (smallest objective == highest rank == best fit).

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)),]
        }
    }
    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_mus(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

    $a_{1,1}x_1 + \cdots + a_{1,p}x_p \leq b_1, \ldots, a_{k,1}x_1 + \cdots + a_{k,p}x_p \leq b_k$

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 and 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

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 $\mathcal{U} = \{U_1, \ldots, U_m\}$ denote the set of m users.

Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ 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:

- $r_{u,i} \in \{0, 1, \ldots, 5\}$ (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 $\{U_{v_1}, \ldots, U_{v_K}\} \in \mathcal{U} \setminus \{U_u\}$ be the set of users maximising $s^U_{u,v_1}, \ldots, s^U_{u,v_K}$ and having $r_{v_1,i}, \ldots, r_{v_K,i} > 0$. 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_{u,v_ℓ}
>
> 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 indexes 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:

$$s^I_{i,j} = S(\mathbf{r}_{\cdot,i}, \mathbf{r}_{\cdot,j}).$$

```
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 $\{I_{j_1}, \ldots, I_{j_K}\} \in \mathcal{I} \setminus \{I_i\}$ be the set of items maximising $s^I_{i,j_1}, \ldots, s^I_{i,j_K}$ and having $r_{u,j_1}, \ldots, r_{u,j_K} > 0$. 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_{i,j_ℓ} or the mode.

```
K <- 2
(sim <- order(SI["Apple",],decreasing=TRUE))
```

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

```
# sim gives the indexes 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 and 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
```

```
table(ratings$rating)
```

```
##
##   0.5     1   1.5     2   2.5     3   3.5     4   4.5     5
##  1370  2811  1791  7551  5550 20047 13136 26818  8551 13211
```

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 you 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 and others 2012)
- (Harper and Konstan 2015)

Other:

- (Koren 2009)
- (Töscher, Jahrer, and Bell 2009)
- (Piotte and 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

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, $A \times B = \{(a, b) : a \in A, b \in B\}$

$A^p = A \times A \times \cdots \times A$ (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

$\boldsymbol{x} = (x_1, \dots, x_n)$ – a sequence of n elements (n-ary sequence/vector)

if it consists of real numbers, we write $\boldsymbol{x} \in \mathbb{R}^n$

$\mathbf{x} = [x_1 \ x_2 \ \dots \ x_p]$ – 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 $b_{i,j} = a_{j,i}$.

$\|\boldsymbol{x}\| = \|\boldsymbol{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$ – the Euclidean norm

Notation Convention – Functions

$f : X \to Y$ 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 $e \simeq 2.718$

$\log x$ – 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

$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$

$\sum_{i=1,\ldots,n} x_i$ – the same

$\sum_{i \in \{1,\ldots,n\}} x_i$ – the same

note display (stand-alone) $\displaystyle\sum_{i=1}^{n} x_i$ vs text (in-line) $\sum_{i=1}^{n} x_i$ style

$\prod_{i=1}^{n} x_i = x_1 x_2 \ldots x_n$

Appendix A

Vector Algebra in R

A.1 Motivation

Vector and matrix algebra provides us with a convenient language for expressing computations on 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).

Using matrix notation leads to a more concise and readable code. It might also be faster to compute.

For instance, given two vectors $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_n)$ 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/n)
```

```
## [1] 4.55796
```

to mean:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-y_i)^2}$$

we'd rather write:

```
sqrt(mean((x-y)^2))
```

```
## [1] 4.55796
```

or even:

$$\frac{1}{\sqrt{n}}\|\boldsymbol{x}-\boldsymbol{y}\|_2$$

In order to be able to read such a notation, we only have to get to know the "building blocks". There are just a few of them, but it takes some time to become comfortable with them.

Also note that vectorised code is much faster and much more readable than the `for` loop-based one:

```
library("microbenchmark")
x <- runif(10000) # 10000 random numbers in [0,1]
y <- runif(10000)
print(microbenchmark(
    t1={s <- 0; n <- length(x);
        for (i in 1:n) s <- s + (x[i]-y[i])^2; sqrt(s/n)},
    t2=sqrt(mean((x-y)^2))
), signif=3, unit='relative')
```

```
## Unit: relative
##  expr  min   lq mean median   uq  max neval
##    t1 86.8 81.1 71.9   69.2 61.6 62.6   100
##    t2  1.0  1.0  1.0    1.0  1.0  1.0   100
```

A.2 Vector-Scalar Operations

Vector-scalar arithmetic operations such as $s\boldsymbol{x}$ (multiplication of a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ by a scalar s) result in a vector $\boldsymbol{y}$ such that $y_i = sx_i$, $i = 1, \ldots, n$.

The same rule holds for, e.g., $s + \boldsymbol{x}$, $\boldsymbol{x} - s$, $\boldsymbol{x}/s$.

```
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 $-\boldsymbol{x}$ we will mean $(-1)\boldsymbol{x}$:

```
-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 $2^{\boldsymbol{x}}$ or $\boldsymbol{x}^2$.

A.3 Vector-Vector Operations

Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_n)$ be two vectors of identical lengths.

Arithmetic operations $\boldsymbol{x}+\boldsymbol{y}$ and $\boldsymbol{x}-\boldsymbol{y}$ are performed *elementwise*, i.e., they result in a vector $\boldsymbol{z}$ such that $z_i = x_i + y_i$ and $z_i = x_i - y_i$, respectively, $i = 1, \ldots, n$.

```
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
```

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 vectors, 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
```

```
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 sensible result.

In R:

- comparison operators such as `<` (less than), `<=` (less than or equal), `==` (equal), `!=` (not equal), `>` (greater than) and `>=` (greater than or equal) as well as
- logical operators like `&` (and) and `|` (or)

are performed in the same manner as above, i.e.:

- they are elementwise operations and
- recycling rule is applied if necessary.

However, they return *logical* vectors in result.

```
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
```

Also note that in R there are 3 (!) logical values: `TRUE`, `FALSE` and `NA` (not available, missing, null).

Generally, operations on `NA`s yield `NA` in result unless other solution makes sense.

```
c(0, NA, 2)*c(1, 10, 100)
```

```
## [1]   0  NA 200
```

```
u <- c(TRUE, FALSE, NA)
v <- c(TRUE, TRUE, TRUE, FALSE, 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
```

A.4 Other Vector Operations

R implement a couple of *aggregation* functions:

- `sum(x)` $= \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$
- `prod(x)` $= \prod_{i=1}^{n} x_i = x_1 x_2 \ldots x_n$
- `mean(x)` $= \frac{1}{n} \sum_{i=1}^{n} x_i$ – arithmetic mean
- `var(x)` = `sum((x-mean(x))^2)/(length(x)-1)` $= \frac{1}{n-1} \sum_{i=1}^{n} \left(x_i - \frac{1}{n} \sum_{j=1}^{n} x_j \right)^2$ – variance
- `sd(x)` = `sqrt(var(x))` – standard deviation

see also: `min()`, `max()`, `median()`, `quantile()`.

> Remember that you can always access the R manual by typing `?functionname`, e.g., `?quantile`.

Mathematically, we will also be interested in the following norms:

- Euclidean norm:
$$\|\boldsymbol{x}\| = \|\boldsymbol{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$
this is nothing else than the *length* of the vector $\boldsymbol{x}$
- Manhattan (taxicab) norm:
$$\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|$$
- Chebyshev (maximum) norm:
$$\|\boldsymbol{x}\|_\infty = \max_{i=1,\ldots,n} |x_i| = \max\{|x_1|, |x_2|, \ldots, |x_n|\}$$

```
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:

$$\|\boldsymbol{x} - \boldsymbol{y}\| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

gives the *Euclidean distance* (metric) between the two vectors.

```
u <- c(1, 0)
v <- c(1, 1)
sqrt(sum((u-v)^2))
```

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

What is more, given two vectors of identical lengths, $\boldsymbol{x}$ and $\boldsymbol{y}$, we define their *dot product* (a.k.a. *scalar, inner product*) as:

$$\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{n} x_i y_i.$$

This is not the same as elementwise vector multiplication in R.

```
u <- c(1, 0)
v <- c(1, 1)
sum(u*v)
```

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

(*) Note that the squared Euclidean norm of a vector is equal to the dot product of the vector and itself, $\|\boldsymbol{x}\|^2 = \boldsymbol{x} \cdot \boldsymbol{x}$.

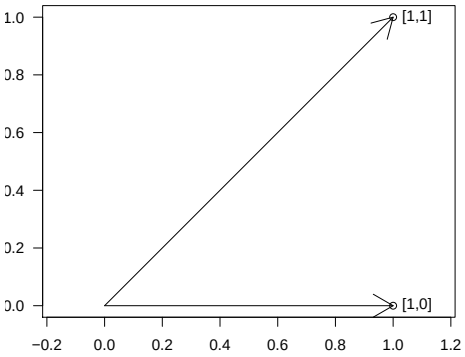
Interestingly, a dot product has a nice geometrical interpretation:

$$\boldsymbol{x} \cdot \boldsymbol{y} = \|\boldsymbol{x}\|\|\boldsymbol{y}\| \cos \alpha$$

where α is the angle between the two vectors.

Read: it is the product of the lengths of the two vectors and the cosine of the angle between them.

You can get the cosine part by computing the dot product of the *normalised* vectors, i.e., such that their lengths are equal to 1.




```
len_u <- sqrt(sum(u^2)); len_v <- sqrt(sum(v^2))
(cos_angle_uv <- (sum(u*v)/(len_u*len_v)))
```

```
## [1] 0.7071068
```

```
acos(cos_angle_uv)*180/pi # angle in degs
```

```
## [1] 45
```

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.

```
sqrt(1:9)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
## [7] 2.645751 2.828427 3.000000
```

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
```

```
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==1` and `FALSE==0`.

A.5 Further Reading

Recommended further reading:

- (Venables, Smith, and the R Core Team 2020)

Other:

- (Deisenroth, Faisal, and Ong 2020)
- (Peng 2019)
- (Wickham and Grolemund 2017)

Appendix B

Matrix Algebra in R

B.1 Matrices

Vectors are 1-dimensional objects – they represent sequences of values.

Matrices are 2-dimensional objects – they represent tabular data.

```
A <- matrix(c(1, 2, 3, 4, 5, 6), byrow=TRUE, nrow=2)
dim(A) # number of rows, number of columns
```

```
## [1] 2 3
```

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Using mathematical notation, above we have defined $\mathbf{A} \in \mathbb{R}^{2\times 3}$:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Other ways to create a matrix:

```
rbind(1:3, 4:6, 7:9) # row bind
```

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

```
rbind(1:3, 4:6, 7:9) # column bind
```

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

On a side note, R `data.frame`s are similar to matrices but are used to store tabular data of potentially different types in each column.

Note that the omission of `byrow=TRUE` yields the following default behaviour of the `matrix()` function:

```
matrix(c(1, 2, 3, 4, 5, 6), nrow=2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

In other words, elements are read in a column-major manner.

(*) This is exactly how R stores the underlying data in RAM.

Also take notice of the fact that "flat" vectors are promoted to column vectors, i.e., matrices with one column:

```
as.matrix(1:3)
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

$\mathbf{A}^T$ denotes the matrix *transpose*:

```
t(A)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    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.

B.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:

```
(-1)*A
```

```
##      [,1] [,2] [,3]
## [1,]   -1   -2   -3
## [2,]   -4   -5   -6
```

B.3 Matrix-Matrix Operations

If $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times p}$ are two matrices of the same 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
```

B.4 Matrix Multiplication (*)

Mathematically, **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 $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

> (*) 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 the next chapter we will learn about the Pearson linear correlation coefficient which can be beautifully expressed this way.

> (*) As an exercise, I recommend that you multiply a few simple matrices of sizes 2×2, 2×3, 3×2 etc. using pen and paper and check the results in R. This will become easy once you get some practice.

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`.

B.5 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 a vector 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 $\mathbf{A}\mathbf{x}$ only if $\mathbf{x} \in \mathbb{R}^{p \times 1}$ is a column vector.

Similarly, $\mathbf{y}\mathbf{A}$ makes only sense whenever $\mathbf{y} \in \mathbb{R}^{1 \times n}$ is a row vector.

> 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 $\boldsymbol{x}$ is an ordinary vector (1-dimensional sequence).
>
> This is why, e.g., the i-th row of $\mathbf{X}$ is denoted with $\mathbf{x}_{i,\cdot}$ – to emphasise that it is a row vector.

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 $\mathbf{A} \in \mathbb{R}^{n \times p}$ 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:

- `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).

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
```

B.6 Further Reading

Recommended further reading:

- (Venables, Smith, and the R Core Team 2020)

Other:

- (Deisenroth, Faisal, and Ong 2020)
- (Peng 2019)
- (Wickham and Grolemund 2017)

Appendix C

Data Frame Wrangling in R

C.1 TO DO

C.1.1 TO DO

This chapter is under construction.

Due date: March 2020.

C.2 Further Reading

Recommended further reading:

- (Venables, Smith, and the R Core Team 2020)

Other:

- (Peng 2019)
- (Wickham and Grolemund 2017)

R packages dplyr and data.table implement the most common data frame wrangling procedures. You may find them very useful.

References

Bishop, Christopher. 2006. *Pattern Recognition and Machine Learning*. Springer-Verlag. https://www.microsoft.com/en-us/research/people/cmbishop/.

Boyd, Stephen, and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf.

Breiman, Leo, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees*. Chapman; Hall/CRC.

Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong. 2020. *Mathematics for Machine Learning*. Cambridge University Press. https://mml-book.com/.

Fletcher, Roger. 2008. *Practical Methods of Optimization*. Wiley.

Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. https://www.deeplearningbook.org/.

Harper, F. Maxwell, and Joseph A. Konstan. 2015. "The MovieLens Datasets: History and Context." *ACM Transactions on Interactive Intelligent Systems* 5: 19:1–19:19. https://doi.org/10.1145/2827872.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2017. *The Elements of Statistical Learning*. Springer-Verlag. https://web.stanford.edu/~hastie/ElemStatLearn/.

Herlocker, Jonathan L., Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. 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.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2017. *An Introduction to Statistical Learning with Applications in R*. Springer-Verlag. http://faculty.marshall.usc.edu/gareth-james/ISL/.

Koren, Yehuda. 2009. *The BellKor Solution to the Netflix Grand Prize*. https://netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf.

Lü, Linyuan, and others. 2012. "Recommender Systems." *Physics Reports* 519: 1–49. https://arxiv.org/pdf/1202.1112.pdf.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. Springer.

Peng, Roger D. 2019. *R Programming for Data Science*. https://bookdown.org/rdpeng/rprogdatascience/.

Piotte, Martin, and Martin Chabbert. 2009. *The Pragmatic Theory Solution to the Netflix Grand Prize*. https://netflixprize.com/assets/GrandPrize2009_BPC_PragmaticTheory.pdf.

Quinlan, Ross. 1986. "Induction of Decision Trees." *Machine Learning* 1: 81–106.

———. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers.

R Development Core Team. 2020. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. http://www.R-project.org.

Ricci, Francesco, Lior Rokach, Bracha Shapira, and Paul Kantor, eds. 2011. *Recommender Systems Handbook*. Springer. http://www.inf.unibz.it/~ricci/papers/intro-rec-sys-handbook.pdf.

Simon, Dan. 2013. *Evolutionary Optimization Algorithms: Biologically-Inspired and Population-Based Approaches to Computer Intelligence*. Wiley.

Therneau, Terry M., and Elizabeth J. Atkinson. 2019. *An Introduction to Recursive Partitioning Using the RPART Routines*. https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf.

Töscher, Andreas, Michael Jahrer, and Robert M. Bell. 2009. *The BigChaos Solution to the Netflix Grand Prize*. https://netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf.

Venables, W. N., D. M. Smith, and the R Core Team. 2020. *An Introduction to R*. https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.

Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science*. O'Reilly. https://r4ds.had.co.nz/.