

Introduction to Bioinformatics

Laboratory Manual

Flavio Palmieri

Alexandre Perera i Lluna

2026-10-21

Table of contents

Preface	5
How the labs are organised	5
Tools for working with R	5
I Foundations	7
1 Introduction to R	8
1.1 Starting with R	8
1.1.1 Lab requirements	9
1.1.2 Starting R	9
1.1.3 Getting help, finding functions	9
1.1.4 Current path	10
1.1.5 The R session workspace	10
1.1.6 Saving and restoring the workspace	11
1.1.7 Quitting R, batch processing	11
1.2 Simple types	11
1.2.1 Vectors	11
1.2.2 Logical vectors	11
1.2.3 Data types	12
1.2.4 Indexing	12
1.2.5 Factors	13
1.3 Not-so-simple types	14
1.3.1 Arrays	14
1.3.2 Matrix operators	15
1.3.3 Lists and data frames	15
1.4 Input / output	17
1.5 Control statements	17
1.5.1 Conditionals	17
1.5.2 Loops	17
1.6 Defining functions	17
1.6.1 The <code>apply()</code> family and friends	19
1.7 Graphical output	22
1.7.1 The tidyverse: pipes and data wrangling	22
1.8 Packages	28
1.9 Non-guided work with datasets	28
1.10 Further reading	28
1.11 Report	29
2 Searching NCBI and Ensembl	30
2.1 NCBI	30
2.1.1 Genomes	31
2.1.2 Gene search	31
2.1.3 PubMed	32

2.2	Ensembl	32
2.3	Protein databases	32
2.3.1	UniProtKB (Swiss-Prot)	33
2.3.2	Research Collaboratory for Structural Bioinformatics (RCSB)	33
2.4	Programmatic access from R	33
2.4.1	org.Hs.eg.db	34
2.4.2	Live database access with rentrez	38
2.5	Further reading	39
2.6	Report	40
II	Genetics & genomes	41
3	Genetic Association: Quality Control and Stratification	42
3.1	Data exploration	43
3.1.1	Phenotype data	43
3.1.2	Genotype data	45
3.2	Association	46
3.2.1	Initial scan	46
3.2.2	Quality control	48
3.2.3	Final scan	51
3.3	Further reading	52
3.4	Report	53
4	Genome-Wide Association: a Thrombophilia Case	54
4.1	Data exploration	55
4.1.1	Genotype data	56
4.2	Familial structure	56
4.3	Association	57
4.4	Further reading	60
4.5	Report	60
5	Computing Homology	61
5.1	Homology	61
5.2	Alignment types	62
5.3	Working with protein sequences	62
5.3.1	BLASTP at NCBI	64
5.3.2	TBLASTN at NCBI	65
5.3.3	BLASTP at EBI	65
5.4	Working with DNA sequences	67
5.4.1	BLASTN at NCBI	67
5.4.2	BLASTX at NCBI	68
5.4.3	TBLASTX at NCBI	68
5.5	SARS-CoV-2	69
5.6	Doing it in R	70
5.7	Further reading	72
5.8	Report	72

III Patterns & expression	73
6 Pattern Recognition with R: Eigenfaces	74
6.1 Dataset description	74
6.2 Creating pattern vectors	77
6.3 Karhunen–Loève decomposition (PCA)	79
6.3.1 PCA for dataset visualisation	81
6.3.2 PCA for dataset modelling	84
6.4 Pattern clustering	88
6.5 Feature clustering	89
6.6 Pattern recognition	91
6.7 Further reading	94
6.8 Report	95
7 Differential Expression	96
7.1 Data description	96
7.2 Gene filtering	98
7.3 Data visualisation	101
7.4 Clustering	102
7.5 Classification	104
7.6 A modern counterpart: RNA-seq with DESeq2	106
7.7 Further reading	107
7.8 Report	108
8 Functional Enrichment	109
8.1 Functional analysis in differential expression	110
8.1.1 Gene set functional enrichment	110
8.1.2 ORA vs GSEA	112
8.2 Enrichment in metabolomics with FELLA	112
8.2.1 Building a KEGG object	113
8.2.2 Enrichment analysis on liver tissue	113
8.2.3 FELLA graphical user interface	114
8.3 Further reading	114
8.4 Report	114
References	115
Appendices	116
Glossary	116
Session information	118

Preface

Welcome to the **Introduction to Bioinformatics** laboratory manual, the companion practical for the *Introduction to Bioinformatics* course of the Màster en Enginyeria Biomèdica at the Universitat Politècnica de Catalunya.

This manual is the Quarto edition of the lab book originally written in **Sweave**/LaTeX by Flavio Palmieri and Alexandre Perera i Lluna (b2slab.upc.edu). Each chapter is a self-contained laboratory that mixes explanatory text, runnable **R** code, and exercises.

How the labs are organised

Every lab follows the same rhythm: a short motivation, worked examples you can run line by line, and a set of exercises (marked with a `>` callout) that you are expected to complete and write up in a report.

The laboratories are:

1. **Introduction to R** — the R language, data types, and graphics.
2. **Searching NCBI** — querying biological databases.
3. **Association studies** — genotype–phenotype association.
4. **Gait / GWAS** — genome-wide association on a quantitative trait.
5. **Homology** — sequence alignment and homology search.
6. **Eigenfaces** — principal component analysis on images.
7. **Differential expression** — transcriptomics analysis.
8. **Functional enrichment** — over-representation and pathway analysis.

Tools for working with R

The code in this book is written in **R**. The labs are designed to be worked through in your own script or notebook: type the code, run it, and inspect the results as you go. You will need:

1. A computer.
2. **R**, the language interpreter, installed from <https://www.R-project.org>. **R** itself is free and open-source software (GPL).
3. An editor or integrated development environment (IDE) — see below.

Most of these editors are free and open source; the exceptions are noted.

- **RStudio Desktop** (<https://posit.co/download/rstudio-desktop/>) is the classic **R** IDE: console, editor, plots, help, and package management in one workbench. It is the environment we recommend for this course. **RStudio** is open source (AGPLv3); the company behind it, Posit, also sells a commercial server edition, but the desktop IDE is fully free.

- **Positron** (<https://positron.posit.co/>) is a newer, polyglot data-science IDE from Posit with first-class support for both R and Python. It is built on Code OSS — the open-source core of Visual Studio Code from which editors like VSCodium are also derived — but Positron itself is *source-available* rather than open source: you can read and build the code, yet its Elastic License 2.0 restricts offering it as a paid service.
- **Visual Studio Code / VSCodium** with the R extension, **Emacs** with [ESS](#), **Vim**/Neovim with [Nvim-R](#), and **Jupyter** with an R kernel are further open-source options; any plain text editor plus a terminal running R works too.

The labs have been tested on GNU/Linux and macOS (both x86-64 and ARM); Windows may require some extra setup.

i About the exercises and solutions

This PDF edition contains the numbered **exercises** you are expected to solve, but **no worked solutions** — they are deliberately left out so you work through the labs yourself. The solutions are available as *collapsed callouts* right after each exercise in the HTML edition of this manual.

! Important

R is case sensitive. Pay attention to upper- and lower-case letters in function and variable names throughout these labs.

Part I

Foundations

1 Introduction to R

This first laboratory introduces **R**, the language and environment you will use throughout the course. By the end of this lab you will be able to:

- start and navigate an R session, get help, and manage your workspace
- work with the core data structures — vectors, factors, arrays, lists, and data frames
- use indexing, control statements, and the `apply()` family of iterators
- define your own functions
- produce and save graphical output
- read data into R and describe a dataset statistically
- chain operations with the pipe, wrangle data frames with `dplyr/tidyr`, and plot with `ggplot2`

! Important

R is case sensitive! `Mean` and `mean` are not the same thing.

Lab at a glance

Goal: get comfortable with R — data types, indexing, functions, graphics, and an introduction to the tidyverse (pipes, `dplyr`, `tidyr`, `ggplot2`). **Prerequisites:** none. **Time:** ~2.5 h. **Packages:** base R + `dplyr`, `tidyr`, `ggplot2`. **Data:** built-in (`iris`). **Deliverable:** a lab report.

1.1 Starting with R

R is a scientific software package developed by *The R Foundation for Statistical Computing* and distributed from their website <https://www.R-project.org>.¹

The name is partly based on the first names of the first two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs language **S** (Hornik, 2007). S is a very high-level language and environment for data analysis and graphics, born from discussions at Bell Labs in 1976. R is one particular implementation of the S language.

Although R is slightly different from MATLAB, its functionality is similar and it offers powerful packages for pattern recognition and multivariate analysis. It has a strong, active community and is under continuous development. Ports exist for many members of the Unix family (GNU/Linux, macOS, Solaris, ...) as well as Microsoft Windows.

Several user interfaces are available — an Emacs mode ([ESS](#)), Rcmdr, and [RKWard](#) among them. We strongly recommend using [RStudio](#) in this class.

¹Some sections of this introduction are adapted from (Venables & Smith, 2006). The original version of this document was written in LaTeX with embedded R code through the **Sweave** tool; this edition is produced with [Quarto](#).

1.1.1 Lab requirements

To follow this laboratory you will need:²

1. A computer.
2. R, installed from <https://www.R-project.org>.
3. A text editor or RStudio. **We strongly recommend RStudio** for this class.
4. A web browser to read the HTML edition of this manual (a PDF reader is only needed for the PDF edition).

1.1.2 Starting R

You can work in RStudio (recommended), in the native R application on macOS or Windows, or in a terminal by typing R. A current session begins with output similar to this (the version and platform will differ):

```
$ R

R version 4.x.y (YYYY-MM-DD) -- "release nickname"
Copyright (C) YYYY The R Foundation for Statistical Computing
Platform: your-platform

R is free software and comes with ABSOLUTELY NO WARRANTY.
...
Type 'q()' to quit R.

>
```

Launch RStudio in the usual way for your operating system and create an RStudio Project for your work on these labs.

1.1.3 Getting help, finding functions

The first thing to learn in any language is how to ask for help. In R we use `help()`. For instance, to check the syntax of `getwd()`:

```
help("getwd")
```

The command `help.start()` launches the help system in a browser. You can call `help(print)` and then run your first *Hello World*:

```
print("Hello World")
```

```
[1] "Hello World"
```

²You can install any package with `install.packages("package name")`. On Ubuntu Linux a base R installation is `sudo apt-get install r-recommended`. RStudio can be downloaded from <https://posit.co/download/rstudio-desktop/>.

A very useful way to find help is the `example()` function, which runs example uses of a command:

```
example(ls)
```

The help system can be searched for a pattern with `help.search()`.

1.1.4 Current path

Use a project so code and data paths are reproducible. In RStudio, choose **File → New Project** and keep scripts and data inside that project. `getwd()` shows the current project directory. Although `setwd()` changes it temporarily, avoid machine-specific absolute paths in work you submit:

```
getwd()
```

1.1.5 The R session workspace

Assignment uses the `<-` operator. To assign a value to a variable `a`:

```
a <- pi
sqrt(2) -> b
a
```

```
[1] 3.141593
```

```
b
```

```
[1] 1.414214
```

List the objects living in memory with `objects()` (note we also define a small function `myfunc` here so the next example has something to remove):

```
myfunc <- function(x) x^2
objects()
```

```
[1] "a"      "b"      "myfunc"
```

Remove objects from your session with `rm()`:

```
rm(myfunc, a)
objects()
```

```
[1] "b"
```

1.1.6 Saving and restoring the workspace

It is easy to save variables to an R data file (.RData):

```
xx <- runif(20)
yy <- list(a = 1, b = TRUE, c = "oops")
save(xx, yy, file = "xxyy.Rdata")
```

To save the complete workspace:

```
save.image(file = "backup.RData")
```

1.1.7 Quitting R, batch processing

You quit an R session with `q()`. R can also be called for batch processing from the shell:

```
$ R [options] [< infile] [> outfile]
```

where the commands to process are read from `infile` and output is logged to `outfile`.

1.2 Simple types

1.2.1 Vectors

The simplest structure is a numeric vector, built with the concatenate function `c()`:

```
x <- c(2.3, 3, 5, 7, 7.5, 7, 5, 4)
y <- c(x, 3, x)
```

Note that `c(2.3, 3, 5, 7, 7.5, 7, 5, 4) -> x` has the same effect. Simple arithmetic works as in other languages (`+` `-` `*` `/` `^`), as do functions such as `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`.

A regularly spaced vector is generated with `seq()`:

```
sq <- seq(5, -2, by = -0.5)
```

Arguments in R are matched by name, and in the absence of a name, by position, so the following produces the same result:

```
sq <- seq(by = -0.5, to = -2, from = 5)
```

1.2.2 Logical vectors

Test the effect of comparing a number with a vector:

```
big <- x > 5
big
```

```
[1] FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE
```

💡 > Exercise 1.1 — Binary selection indices

Can you interpret the output of `which(big)`? Try it.

1.2.3 Data types

R functions behave differently depending on the class of their input. The `c()` function also concatenates characters, and `summary()` behaves differently for numeric and character data:

```
y <- c(2, -3, 3, -1, 3, 5, 6, 12)
cv <- c("X1", "Y2", "X1", "Y1", "X2", "Y3", "X2", "Y3")
summary(cv)
```

```
Length      Class      Mode
      8 character character
```

```
summary(y)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-3.000  1.250   3.000   3.375  5.250  12.000
```

1.2.4 Indexing

Any object can be indexed. Vector elements are retrieved with the `[]` operator:

```
x[3]
```

```
[1] 5
```

```
x[c(3, 5)]
```

```
[1] 5.0 7.5
```

```
x[big]
```

```
[1] 7.0 7.5 7.0
```

```
x[which(big)]
```

```
[1] 7.0 7.5 7.0
```

```
(x + 10)[5]
```

```
[1] 17.5
```

R also supports advanced, name-based indexing for good metadata management:

```
geneexp <- c(1.3, -4, 3, 4, 0.3)
names(geneexp) <- c("ACP1", "COX2", "F7", "F11", "ABG1")
geneexp
```

```
ACP1 COX2  F7  F11 ABG1
1.3 -4.0  3.0  4.0  0.3
```

```
coag_factors <- geneexp[c("F7", "F11")]
coag_factors
```

```
F7 F11
3  4
```

1.2.5 Factors

Factors are the natural representation for categorical data. Any vector can be coerced with `as.factor()` or created with `factor()`:

```
cvf <- as.factor(cv)
print(cvf)
```

```
[1] X1 Y2 X1 Y1 X2 Y3 X2 Y3
Levels: X1 X2 Y1 Y2 Y3
```

```
summary(cvf)
```

```
X1 X2 Y1 Y2 Y3
2  2  1  1  2
```

For example, suppose we have a vector of *FVII* blood concentrations for individuals carrying different diseases:

```
disease <- c("Hemophilia A", "Thrombocytopenia", "Ehlers-Danlos syndrome",
             "Hemophilia", "Ehlers-Danlos syndrome", "Thrombocytopenia", "Vasculitis",
             "Vasculitis", "Vasculitis", "Thrombocytopenia", "Myeloproliferative",
             "Thrombocytopenia", "Myeloproliferative", "Hemophilia", "Thrombocytopenia",
             "Ehlers-Danlos syndrome",
             "Thrombocytopenia", "Thrombocytopenia", "Thrombocytopenia", "Vasculitis",
```

```

      "Hemophilia A", "Ehlers-Danlos syndrome", "Thrombocytopenia", "Myeloproliferative",
      "Ehlers-Danlos syndrome", "Hemophilia B", "Hemophilia A", "Vasculitis",
      "Ehlers-Danlos syndrome", "Vasculitis")
diseasef <- factor(disease)
diseasef

```

```

[1] Hemophilia A      Thrombocytopenia  Ehlers-Danlos syndrome
[4] Hemophilia        Ehlers-Danlos syndrome Thrombocytopenia
[7] Vasculitis        Vasculitis        Vasculitis
[10] Thrombocytopenia  Myeloproliferative Thrombocytopenia
[13] Myeloproliferative Hemophilia        Thrombocytopenia
[16] Ehlers-Danlos syndrome Thrombocytopenia  Thrombocytopenia
[19] Thrombocytopenia  Vasculitis        Hemophilia A
[22] Ehlers-Danlos syndrome Thrombocytopenia  Myeloproliferative
[25] Ehlers-Danlos syndrome Hemophilia B      Hemophilia A
[28] Vasculitis        Ehlers-Danlos syndrome Vasculitis
7 Levels: Ehlers-Danlos syndrome Hemophilia Hemophilia A ... Vasculitis

```

```
summary(diseasef)
```

Ehlers-Danlos syndrome	Hemophilia	Hemophilia A
6	2	3
Hemophilia B	Myeloproliferative	Thrombocytopenia
1	3	9
Vasculitis		
6		

1.3 Not-so-simple types

1.3.1 Arrays

Arrays are created by changing the dimension attribute of a vector:

```

z <- rnorm(10)
z

```

```

[1] -1.07506633  1.51879819  1.12760736  0.77967322 -0.58281648
[6] -0.95012719  1.91927289 -0.70562058 -0.09312588 -1.36857363

```

```

dim(z) <- c(5, 2)
z

```

	[,1]	[,2]
[1,]	-1.0750663	-0.95012719
[2,]	1.5187982	1.91927289
[3,]	1.1276074	-0.70562058

```
[4,] 0.7796732 -0.09312588
[5,] -0.5828165 -1.36857363
```

Empty matrices can be built with `array()` (e.g. `array(0, c(5, 2))`). Elements are accessed with `[]`:

```
z[3, 2]
```

```
[1] -0.7056206
```

```
z[, 2]
```

```
[1] -0.95012719 1.91927289 -0.70562058 -0.09312588 -1.36857363
```

```
z[1, ]
```

```
[1] -1.0750663 -0.9501272
```

Arrays are combined with `cbind()` and `rbind()` (column and row binding), and transposed with `t()`:

```
tz <- t(z)
```

1.3.2 Matrix operators

Several operators exist for matrix manipulation. Matrix multiplication uses `%*%`:

```
tz %*% z
```

```
      [,1]      [,2]
[1,] 5.681579 3.865795
[2,] 3.865795 6.965917
```

Other operators of interest include `diag()`, `crossprod()`, `solve()`, `eigen()`, and `svd()`.

1.3.3 Lists and data frames

A `list` is an ordered collection of objects that need not be of the same type. Components are accessed with `[[ ]]` (by index) or `$` (by name):

```
l <- list(name = "FVII", type = "Coagulation Factor", range = c(100, 100000))
l[[2]]
```

```
[1] "Coagulation Factor"
```

```
l$type
```

```
[1] "Coagulation Factor"
```

```
l[[3]][2]
```

```
[1] 1e+05
```

```
ty <- "type"  
l[[ty]]
```

```
[1] "Coagulation Factor"
```

`str()` prints a human-readable summary of complex objects:

```
str(l)
```

List of 3

```
$ name : chr "FVII"  
$ type : chr "Coagulation Factor"  
$ range: num [1:2] 1e+02 1e+05
```

A **data frame** is a special list of class `data.frame` — essentially a matrix whose columns may have different types. Define a vector of *FVII* levels (in arbitrary units):

```
fvii <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,  
         61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,  
         59, 46, 58, 43)
```

Then build a data frame pairing the disease factor with the FVII levels:

```
blood <- data.frame(Disease = diseasef, FVII = fvii)  
head(blood)
```

	Disease	FVII
1	Hemophilia A	60
2	Thrombocytopenia	49
3	Ehlers-Danlos syndrome	40
4	Hemophilia	61
5	Ehlers-Danlos syndrome	64
6	Thrombocytopenia	60

`head()` shows the first six rows; see also `tail()`.

1.4 Input / output

An important part of R deals with reading data from files. The easiest way is `read.table()`, which reads a delimited file into a data frame. Related functions are `read.csv()` and `read.delim()`. For very large files, prefer lower-level primitives such as `scan()`.

Objects such as data frames can be edited interactively with `edit()` or `fix()`, e.g. `blood <- edit(blood)`.

1.5 Control statements

1.5.1 Conditionals

```
if (1 == 2) { print("yes") } else { print("no") }
```

```
[1] "no"
```

The condition must evaluate to a single logical value. Comparison operators are `&&` `||` `>=` `>` `<` `<=` `==` `!=`; the `&` and `|` operators apply element-wise to vectors. There is also a vectorised `ifelse(condition, a, b)`.

1.5.2 Loops

```
for (ic in c("joan", "helena", "maria")) { print(ic) }
```

```
[1] "joan"  
[1] "helena"  
[1] "maria"
```

For loops appear much less often than in compiled languages, because R provides compact iteration via `apply()`, `sweep()`, `mapply()`, `tapply()` and friends. There are also `repeat` and `while`:

```
k <- 3  
while (k > 0) { print(k <- k - 1) }
```

```
[1] 2  
[1] 1  
[1] 0
```

1.6 Defining functions

R lets you define your own functions:

```
FunctionName <- function(x, y) { x + y }  
FunctionName(3, 4)
```

```
[1] 7
```

For example, the following implements $f(k) = \sum_{x=1}^{k-1} x$:

$$f(k) = \sum_{x=1}^{k-1} x \quad (1.1)$$

```
SerialSum <- function(k) {  
  out <- 0  
  while (k > 0)  
    out <- out + (k <- k - 1)  
  out  
}  
SerialSum(3)
```

```
[1] 3
```

```
SerialSum(5)
```

```
[1] 10
```

! Important

The return value of a function is the result of its **last expression**.

Typing a function's name shows its definition:

```
SerialSum
```

```
function (k)  
{  
  out <- 0  
  while (k > 0) out <- out + (k <- k - 1)  
  out  
}  
<bytecode: 0x10517cef0>
```

Binary operators can also be defined, with syntax like `"%!" <- function(X, y) { ... }`.

💡 > Exercise 1.2 — Weighted mean of squares

Create a function that, given a vector $x = x_1 \cdots x_N$, computes

$$y = \frac{\sum_k k x_k^2}{N} \quad (1.2)$$

1.6.1 The `apply()` family and friends

The `apply()` family does most of the work when an iterator is needed. With a function defined, it is easy to iterate it over a vector with `sapply()`, which also *simplifies* the output to a vector:

```
myfun <- function(x) sqrt(x) * x
sapply(1:10, myfun)
```

```
[1] 1.000000 2.828427 5.196152 8.000000 11.180340 14.696938 18.520259
[8] 22.627417 27.000000 31.622777
```

`apply()` iterates a function over rows or columns (margins). Here, using the *iris* dataset:

```
data(iris)
head(iris)
```

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

```
apply(iris[, -5], 2, sd)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
0.8280661	0.4358663	1.7652982	0.7622377

```
apply(iris[1:10, -5], 1, summary)
```

	1	2	3	4	5	6	7	8	9	10
Min.	0.20	0.200	0.200	0.200	0.20	0.400	0.300	0.200	0.200	0.10
1st Qu.	1.10	1.100	1.025	1.175	1.10	1.375	1.125	1.175	1.100	1.15
Median	2.45	2.200	2.250	2.300	2.50	2.800	2.400	2.450	2.150	2.30
Mean	2.55	2.375	2.350	2.350	2.55	2.850	2.425	2.525	2.225	2.40
3rd Qu.	3.90	3.475	3.575	3.475	3.95	4.275	3.700	3.800	3.275	3.55
Max.	5.10	4.900	4.700	4.600	5.00	5.400	4.600	5.000	4.400	4.90

💡 > Exercise 1.3 — `apply()` over rows and columns

Explain the function calls above and the output they generate.

`split()` retrieves the strata of a data frame given a factor, returning a named list:

```
iris.strata <- split(iris, iris$Species)
length(iris.strata)
```

```
[1] 3
```

```
names(iris.strata)
```

```
[1] "setosa"      "versicolor" "virginica"
```

```
summary(iris.strata$versicolor)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Min. :4.900	Min. :2.000	Min. :3.00	Min. :1.000
1st Qu.:5.600	1st Qu.:2.525	1st Qu.:4.00	1st Qu.:1.200
Median :5.900	Median :2.800	Median :4.35	Median :1.300
Mean :5.936	Mean :2.770	Mean :4.26	Mean :1.326
3rd Qu.:6.300	3rd Qu.:3.000	3rd Qu.:4.60	3rd Qu.:1.500
Max. :7.000	Max. :3.400	Max. :5.10	Max. :1.800

Species

setosa	: 0
versicolor	:50
virginica	: 0

`by()` extends `split()` so you can apply a method to each stratum:

```
by(iris[, 1:2], iris$Species, summary)
```

```
iris$Species: setosa
  Sepal.Length  Sepal.Width
Min.   :4.300   Min.   :2.300
1st Qu.:4.800   1st Qu.:3.200
Median :5.000   Median :3.400
Mean   :5.006   Mean   :3.428
3rd Qu.:5.200   3rd Qu.:3.675
Max.   :5.800   Max.   :4.400
```

```
-----
iris$Species: versicolor
  Sepal.Length  Sepal.Width
```

```

Min.      :4.900    Min.      :2.000
1st Qu.:5.600    1st Qu.:2.525
Median :5.900    Median :2.800
Mean      :5.936    Mean      :2.770
3rd Qu.:6.300    3rd Qu.:3.000
Max.      :7.000    Max.      :3.400

```

```

iris$Species: virginica
Sepal.Length Sepal.Width
Min.      :4.900    Min.      :2.200
1st Qu.:6.225    1st Qu.:2.800
Median :6.500    Median :3.000
Mean      :6.588    Mean      :2.974
3rd Qu.:6.900    3rd Qu.:3.175
Max.      :7.900    Max.      :3.800

```

Retrieving `fvii` and `diseasef` from Section 1.2.5, it is easy to compute the mean FVII concentration per disease with `by()`:

```
by(fvii, diseasef, mean)
```

```

diseasef: Ehlers-Danlos syndrome
[1] 54

```

```

diseasef: Hemophilia
[1] 61

```

```

diseasef: Hemophilia A
[1] 56

```

```

diseasef: Hemophilia B
[1] 46

```

```

diseasef: Myeloproliferative
[1] 58

```

```

diseasef: Thrombocytopenia
[1] 53.22222

```

```

diseasef: Vasculitis
[1] 54.83333

```

 > Exercise 1.4 — `tapply()`

Compute the same per-disease means through the `tapply()` function. Can you explain what `tapply()` does?

The `aggregate()` function accepts R's formula interface for compact computations:

```
aggregate(. ~ Species, iris, mean)
```

	Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	5.006	3.428	1.462	0.246
2	versicolor	5.936	2.770	4.260	1.326
3	virginica	6.588	2.974	5.552	2.026

💡 > Exercise 1.5 — Range per species

Can you compute the maximum and minimum of each variable for each species in the iris dataset?

1.7 Graphical output

The most frequently used plotting function is `plot()`, a generic that behaves differently depending on the class of its first argument. Graphical output goes to the current graphics device — a window (X11/Windows) or a file (PDF, PNG, ...); see `help(Devices)`.

Plotting the numeric `fvii` variable, coloured by disease, gives Figure 1.1:

```
plot(fvii, col = diseasef, pch = 16)
```

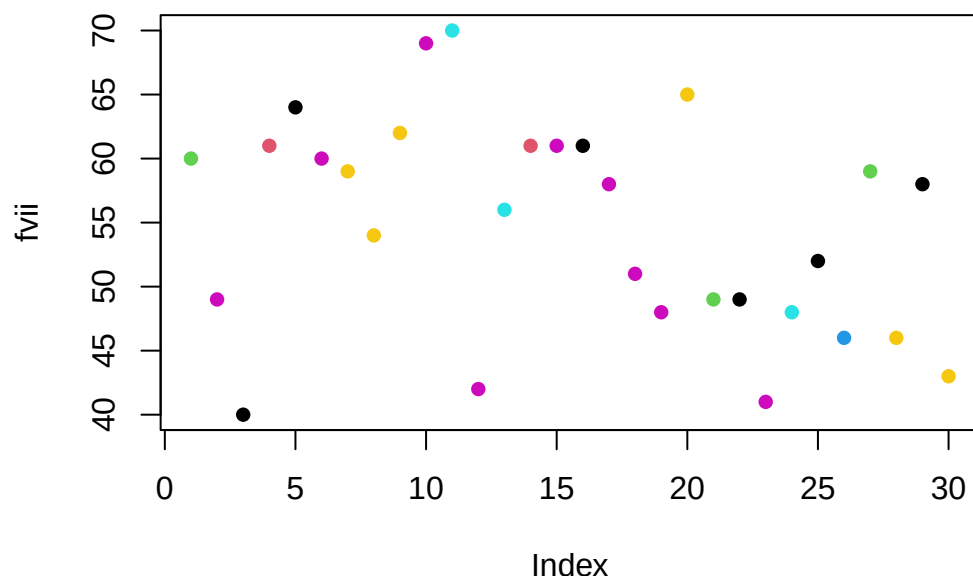


Figure 1.1: FVII concentration for each individual, coloured by disease.

Other standard plotting functions include `hist()`, `dotchart()`, `image()`, `contour()`, and `persp()`, together with lower-level primitives such as `points()`, `lines()`, `text()`, `abline()`, `polygon()`, and `legend()`. See the help for `plot()` and `par()`.

Beyond base graphics, the modern R ecosystem builds plots with **ggplot2** (a grammar of graphics) and adds interactivity with **plotly** — these are what the later labs use for their figures.

1.7.1 The tidyverse: pipes and data wrangling

Alongside base R, most contemporary data analysis uses the **tidyverse** — a family of packages (`dplyr`, `tidyr`, `ggplot2`, `readr`, ...) sharing a consistent “tidy data” philosophy: one observation

per row, one variable per column, one dataset per file. Its tools all take a *data frame in, data frame out* view, which makes them composable in a way base R functions are not. You will meet both styles in this course; base R is closer to the statistical machinery, the tidyverse is often clearer for data wrangling and plotting.

1.7.1.1 The pipe

Real analyses chain many steps: subset, filter, transform, summarise, plot. Writing them as nested calls quickly becomes unreadable — R evaluates them inside-out:

```
# Nesting: hard to read (and hard to write) inside-out.  
round(mean(sqrt(mtcars$disp)), 2)
```

```
[1] 14.63
```

The **pipe operator** rewrites the chain so it reads top-to-bottom, left-to-right, the way you think about the analysis. Base R ships the native pipe `|>`: it takes the result on its left and passes it as the **first argument** of the call on its right.

```
# The same computation, read aloud: "take disp, sqrt it, average it, round it."  
mtcars$disp |>  
  sqrt() |>  
  mean() |>  
  round(2)
```

```
[1] 14.63
```

`round(2)` works because the piped value fills the first argument and 2 fills the next — `round(x, 2)`. A pipe is *just* a reordering of arguments; the two forms produce identical results. Two details are worth knowing:

- The magrittr pipe `%>%` (from the package of the same name, loaded with the tidyverse) is the historical operator you will see in older code; the native `|>` behaves the same for everyday use. This manual uses `|>`.
- When the value is *not* the first argument, name it explicitly with the placeholder `_` (native pipe) or `.` (magrittr): for example `blood |> aggregate(FVII ~ Disease, data = _, mean)`.

1.7.1.2 Data wrangling with dplyr

`dplyr` is the workhorse for manipulating data frames. It has a small set of verbs, each doing one obvious thing to rows or columns:

Verb	What it does
<code>filter()</code>	keep rows matching a condition
<code>select()</code>	keep (or drop) columns by name
<code>mutate()</code>	add new columns (or modify existing ones)
<code>arrange()</code>	sort rows

Verb	What it does
<code>group_by()</code> + <code>summarise()</code>	per-group summaries
<code>slice_max()</code> / <code>slice_min()</code>	rows with the largest/smallest values

The same per-disease FVII means from the `by()` example read, in `dplyr`:

```
library(dplyr)
blood |>
  group_by(Disease) |>
  summarise(mean_FVII = mean(FVII), n = n())
```

```
# A tibble: 7 x 3
  Disease          mean_FVII     n
  <fct>          <dbl> <int>
1 Ehlers-Danlos syndrome    54     6
2 Hemophilia                61     2
3 Hemophilia A              56     3
4 Hemophilia B              46     1
5 Myeloproliferative       58     3
6 Thrombocytopenia        53.2     9
7 Vasculitis               54.8     6
```

Notice the pipe: `group_by()` returns a grouped table, `summarise()` receives it and collapses each group to one row. A longer, more realistic wrangle on the `iris` data shows the verbs composing:

```
iris |>
  filter(Petal.Length > 1.5) |> # drop the tiny setosa flowers
  mutate(Petal.Ratio = Petal.Length / Petal.Width) |>
  group_by(Species) |>
  summarise(n = n(),
            mean_ratio = mean(Petal.Ratio),
            sd_ratio = sd(Petal.Ratio)) |>
  arrange(desc(mean_ratio)) |>
  as.data.frame()
```

```
Species  n mean_ratio sd_ratio
1  setosa 13   6.364103 2.2982566
2 versicolor 50   3.242837 0.3124565
3 virginica 50   2.780662 0.4073670
```

Each line is one thought (“filter, mutate, group, summarise, sort”); if a result surprises you, comment out the tail of the pipe and inspect the intermediate table. That debuggability is the practical reason pipes dominate modern R code.

💡 > Exercise 1.6 — Pipes

Rewrite each of these nested calls as a pipe, then check the results match:

```
round(colMeans(iris[, 1:4]), 1)
sqrt(sum(mtcars$hp))
paste("mean =", round(mean(iris$Sepal.Length), 2))
```

💡 > Exercise 1.7 — dplyr wrangling

Using `dplyr` on the `blood` data frame, produce a table with: the number of individuals, the mean and standard deviation of FVII, and the minimum FVII, all **per disease** — without using `summarise` more than once.

💡 > Exercise 1.8 — Selection and sorting

With the `iris` data: (a) build a table with only `Species` and the two petal columns; (b) keep *versicolor* flowers with a petal width above 1.5 cm; (c) sort `mtcars` by horsepower, descending — which are the three most powerful cars?

1.7.1.3 Reshaping with tidyr

Where `dplyr` manipulates rows and columns of a tidy table, `tidyr` moves data *between* layouts. The two fundamental operations are `pivot_longer()` (wide → long, i.e. gather columns into a key-value pair) and `pivot_wider()` (long → wide, the inverse). Plotting and grouping tools in the tidyverse want **long** format, so you will pivot constantly:

```
library(tidyr)
iris_long <- iris |>
  mutate(id = row_number()) |>
  pivot_longer(~c(Species, id),
               names_to = "measurement",
               values_to = "value")
head(iris_long)
```

keep track of each flower
all columns except these...
...become this key column
...and this value column

```
# A tibble: 6 x 4
  Species    id measurement  value
  <fct>   <int>   <chr>      <dbl>
1 setosa     1 Sepal.Length  5.1
2 setosa     1 Sepal.Width   3.5
3 setosa     1 Petal.Length  1.4
4 setosa     1 Petal.Width   0.2
5 setosa     2 Sepal.Length  4.9
6 setosa     2 Sepal.Width   3
```

After the pivot, the four numeric columns became two columns (`measurement`, `value`) and the table grew from 150 to 600 rows — same information, tidy layout. `pivot_wider(names_from = measurement, values_from = value)` reverses it.

💡 > Exercise 1.9 — Wide to long

Why does `pivot_longer(-c(Species, id))` need the `id` column added first? What happens to the result if you drop it — try `iris |> pivot_longer(-Species)`? And how would you pivot only `Sepal.Length` and `Petal.Length` (leaving the widths as columns)?

1.7.1.4 Plotting with ggplot2

`ggplot2` implements the **grammar of graphics**: a plot is built by adding layers to a mapping between data columns and visual properties. A ggplot has three ingredients:

1. **data** — the data frame;
2. **aesthetic mapping** — `aes()`, which maps columns to x/y position, colour, size, ...;
3. **geometries** — `geom_*()` layers that actually draw (points, boxplots, lines, ...).

The long-format table from Section 1.7.1.3 is exactly what ggplot wants — one measurement per row — so the four iris variables can be drawn in a single faceted plot:

```
library(ggplot2)
ggplot(iris_long, aes(Species, value, fill = Species)) +
  geom_boxplot(show.legend = FALSE) +
  facet_wrap(~ measurement, scales = "free_y") +
  labs(x = NULL, y = "cm") +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
```

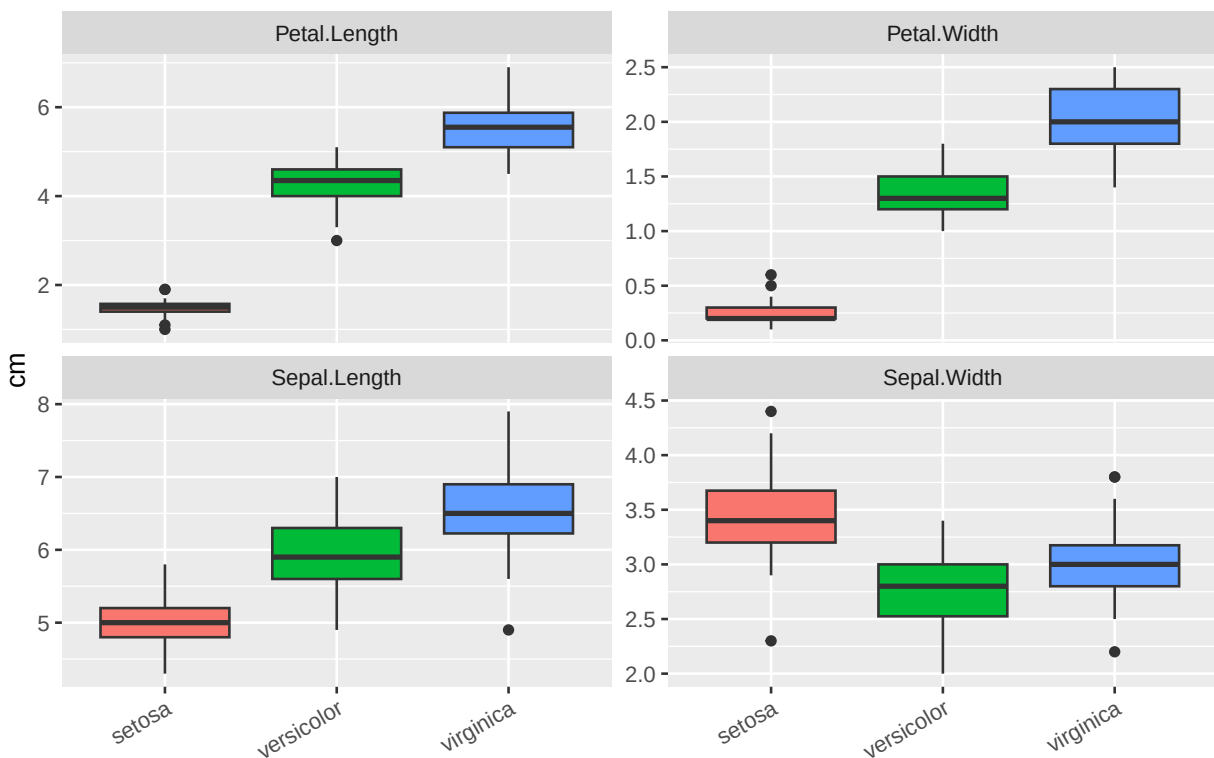


Figure 1.2: All four iris measurements in one ggplot: one row per measurement, faceted by species.

The `blood` data gets the same treatment — a ggplot version of the base Figure 1.1, but with the grouping variable doing the colouring *inside* the aesthetics:

```
ggplot(blood, aes(Disease, FVII, colour = Disease)) +
  geom_jitter(width = 0.2, alpha = 0.6, show.legend = FALSE) +
  stat_summary(fun = mean, geom = "point", shape = 18, size = 3) +
  labs(x = NULL, y = "FVII (arbitrary units)") +
  coord_flip()
```

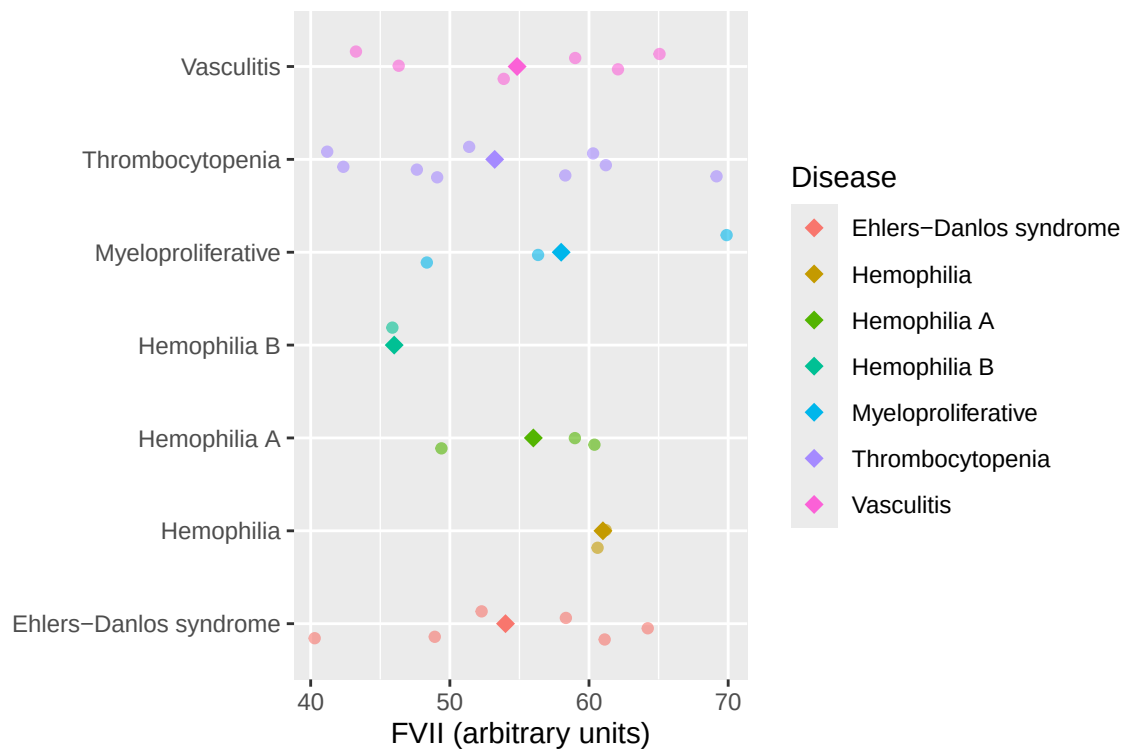


Figure 1.3: FVII by disease with ggplot2: points jittered to reduce overlap, mean marked.

Every layer adds to the previous one — data, then aesthetics, then geometry, then statistics, then labels and theming. Compare this with the base-graphics version: more code, but the intent of each line is explicit, legends generate themselves, and the same recipe scales to any data frame.

💡 > Exercise 1.10 — ggplot2 basics

With the `iris_long` table, make: (a) a scatter of `value` versus `measurement`, coloured by species; (b) the same but with `facet_wrap(~ Species)`; then (c) reproduce Figure 1.1 as closely as possible in ggplot2 (scatter of FVII against row number, coloured by disease).

💡 > Exercise 1.11 — A full pipeline

Starting from the `blood` data frame, write **one piped pipeline** (no intermediate variables) that adds a column `FVII_low = FVII < median(FVII)`, computes per-disease counts of low/high FVII, and feeds the result straight into a ggplot stacked bar chart.

1.8 Packages

Packages bundle functions, data, and documentation for a specific purpose. List installed packages with `library()`, and list currently loaded packages with `search()`:

```
search()
```

```
[1] ".GlobalEnv"      "package:ggplot2"  "package:tidyr"
[4] "package:dplyr"   "package:stats"    "package:graphics"
[7] "package:grDevices" "package:utils"    "package:datasets"
[10] "package:methods" "Autoloads"        "package:base"
```

There are thousands of contributed packages, catalogued at [CRAN](#). A set specialised in bioinformatics is [Bioconductor](#). Install a package from the command line with `install.packages("<packagename>")`.

1.9 Non-guided work with datasets

Check the output of the `data()` function, read its help, and explain what it does. Then browse the *datasets* package and choose a dataset.

i > Assignment

For your chosen dataset:

1. Describe its size and type, including the number of variables and observations.
2. Describe the meaning and type of each variable.
3. Provide a statistical description of the dataset.
4. Agree with your teacher on three statistical questions to answer with it.
5. Report your results both numerically and graphically.

Always include the R code you wrote to answer the questions.

1.10 Further reading

- R Core Team, *An Introduction to R* — the reference the early sections adapt.
- Wickham & Grolemund, *R for Data Science (2e)* — the tidyverse approach you met at the end.
- The built-in tutorial: run `help.start()` → *Manuals* in R.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

1.11 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

2 Searching NCBI and Ensembl

This lab guides you through a routine session with online biological databases. The first sections give an overview of some of the existing internet databases for sequence and general information search; the last part uses **R** to gain programmatic access to annotation datasets through **Bioconductor** libraries.

By the end of this laboratory you will be able to:

- navigate the main **NCBI** resources (Entrez, Gene, Genome, PubMed) and extract genome, gene, and literature information
- retrieve gene and transcript information from **Ensembl**
- look up protein sequences and structures in **UniProtKB/Swiss-Prot** and the **RCSB PDB**
- query genome-wide annotation from R using the Bioconductor `org.Hs.eg.db` package
- query NCBI live from R with **rentrez** — and know when to prefer a local snapshot over a live query

Installing Bioconductor packages

This lab uses packages from **Bioconductor**. If your R installation is missing a package that belongs to Bioconductor (rather than CRAN), install it like this:

```
if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")
BiocManager::install("<packageName>") # a specific package
```

Lab at a glance

Goal: query biological databases by hand and, reproducibly, from R. **Prerequisites:** [Introduction to R](#). **Time:** ~2 h. **Packages:** `org.Hs.eg.db`, `dplyr`, `rentrez` (and `biomaRt` for live Ensembl access, shown not run). **Deliverable:** a lab report.

2.1 NCBI

NCBI stands for the *National Center for Biotechnology Information*. It is a division of the National Library of Medicine, which belongs to the National Institutes of Health (NIH). Created in 1988 as a US national resource for molecular-biology information, all of its databases are public. Besides molecular-biology information, NCBI provides tools for analysing that data and is active in basic research, databases, and education.

NCBI links many databases through the **Entrez** search system and provides tools for sequence search, genome browsing, data download, 3D structure visualisation, and literature discovery.

Important resources include **Bookshelf**, **Gene**, **Nucleotide**, **Protein**, **Assembly**, **Taxonomy**, **PubMed**, **GEO**, **SRA**, **dbGaP**, **dbSNP**, **ClinVar**, and the **Conserved Domain Database**.

Useful analysis and download tools include **BLAST**, **Genome Data Viewer (GDV)**, **NCBI Datasets**, **ORF Finder**, and the browser-based **iCn3D** structure viewer.

2.1.1 Genomes

NCBI Datasets is the current entry point for finding assemblies and downloading genome, transcript, protein, and annotation files; Genome Data Viewer (GDV) provides the interactive chromosome view (the older Map Viewer was retired in 2018). A good first exercise is to compare the *reference assemblies* of three very different organisms.

> Exercise 2.1 — Genomes

1. Open the [NCBI Datasets genome page](#).
2. Search for the reference genomes of *Caenorhabditis elegans*, *Severe acute respiratory syndrome coronavirus 2*, and *Mus musculus* and fill in a comparison table like the one below (report it in your lab report):

organism	accession	assembly level	genome size	annotated genes
<i>C. elegans</i>				
SARS-CoV-2				
<i>M. musculus</i>				

Explain why “number of chromosomes” is not a useful field for every viral or draft assembly.

3. Open the *C. elegans* or mouse reference assembly in [Genome Data Viewer](#). Choose a chromosome, navigate to a gene, zoom from chromosome to exon scale, and identify the assembly and RefSeq annotation release shown.
4. From NCBI Datasets, download a small data package containing the genome data report and GFF3 annotation. Describe the files and identify which one contains feature coordinates.

2.1.2 Gene search

From the NCBI entry point <https://www.ncbi.nlm.nih.gov/>, open the **Gene** database and look for the human gene that encodes a receptor involved in *familial hypercholesterolemia* (FH).

> Exercise 2.2 — Gene search

1. What is the full name of the gene? What is its official *Symbol*? Which organisation standardises the official symbols?
2. Describe the *gene type*. What other possibilities exist for the *Gene Type* field? (Hint: look at the help.)
3. On which chromosome is this gene? Give the *location* of the gene.
4. Search for and download the gene sequence. Compute its length in base pairs.
5. Search for other *loci* related to familial hypercholesterolemia (see <https://www.ncbi.nlm.nih.gov/>)

- lm.nih.gov/omim/143890); write down their position and length.
6. Describe what the *phenotypes* and *homology* fields mean on a gene entry.
 7. To which databases do the links in the *Pathway* section connect? Describe the site you reach and what it contains.

2.1.3 PubMed

PubMed indexes biomedical literature. Its field tags make searches reproducible and help separate words in titles/abstracts from author affiliations.

> Exercise 2.3 — PubMed

1. Search for what *COVID-19* is in PubMed / *Entrez* (Publications), <https://pubmed.ncbi.nlm.nih.gov/>. Explain what this database is.
2. Compare the broad query *COVID-19 Wang* with the author-field query *COVID-19 AND Wang[Author]*.
3. Entrez allows control over the search by specifying fields. Search for *COVID-19[Title/Abstract] AND China[Affiliation]* in PubMed. Explain what the fields mean and what other search options exist (the *help* button is your friend).

2.2 Ensembl

The Ensembl Project is based at the **European Molecular Biology Laboratory's European Bioinformatics Institute (EMBL-EBI)**. It produces and integrates genome annotations for vertebrates and other eukaryotes. Its website and APIs expose genes, transcripts, comparative genomics, variation, and regulation data.

> Exercise 2.4 — Ensembl

1. Point your browser to <https://www.ensembl.org>.
2. Look for the gene *LDLR* in *Homo sapiens*.
3. Within the gene entry, navigate to *Sequence* in the left bar; observe the sequence and the marked exonic regions.
4. Open the transcript Ensembl marks as **canonical** and record its stable transcript and protein IDs and lengths. Do not use “the first transcript”: display order can change between releases.
5. Export one exon in FASTA format and record the Ensembl release and human assembly so your answer remains reproducible.

2.3 Protein databases

PubMed can retrieve protein information, but dedicated protein databases are also useful. A good example is UniProt (Swiss-Prot).

2.3.1 UniProtKB (Swiss-Prot)

We will inspect the reviewed UniProtKB/Swiss-Prot record for the SARS-CoV-2 spike glycoprotein. The protein is viral; *Homo sapiens* is its host, not its source organism.

> Exercise 2.5 — UniProtKB

1. Point a browser to <https://www.uniprot.org/>.
2. Search for accession **P0DTC2** (*SPIKE_SARS2*) and confirm that the organism is SARS-CoV-2. (P59594 is the related spike protein from the 2003 SARS-CoV.)
3. Explain the function of the SARS-CoV-2 spike protein and distinguish the S1 receptor-binding and S2 membrane-fusion roles.
4. Retrieve the amino-acid protein sequence in FASTA format.
5. What is the molecular weight (mass) of the protein, in kDa?

2.3.2 Research Collaboratory for Structural Bioinformatics (RCSB)

RCSB PDB stores experimentally determined macromolecular structures and links to computed structure models. Connect at <https://www.rcsb.org>.

> Exercise 2.6 — RCSB

1. What does RCSB mean?
2. Search for PDB entry **6M0J**, the SARS-CoV-2 spike receptor-binding domain bound to human ACE2.
3. Describe the differences between UniProtKB and the PDB. What type of information can you obtain from RCSB, apart from the protein sequence?
4. Identify the chains corresponding to viral spike and human ACE2 (UniProt **Q9BYF1**), the experimental method, resolution, and biological assembly. Hint: the complex is a 1:1 construct (one RBD chain bound to one ACE2 chain), which helps you tell spike apart from ACE2 when more than two chains are present in the asymmetric unit.
5. Inspect the interface in the browser-based Mol* viewer. Download the mmCIF file (the archival standard); optionally compare it with the legacy PDB format in [PyMOL](#).

About the PDB file format

The primary information stored in the PDB archive consists of coordinate files for biological molecules, listing the atoms in each protein and their 3D locations. The PDB archive's standard text format is **PDBx/mmCIF**; the older fixed-column PDB format remains available for compatible structures. A typical legacy PDB-formatted file has a large *header* section summarising the protein, citation information, and the details of the structure solution, followed by the sequence and a long list of atoms and their coordinates. The archive also contains the experimental observations used to determine the coordinates. Further information about the format is available [here](#).

2.4 Programmatic access from R

Everything so far happened in a browser. In this section we move the same queries into R, which is what makes them reproducible and automatable. Through Bioconductor, R offers many local

annotation packages (see the [AnnotationData BiocViews](#)), while services such as `biomaRt` and `rentrez` provide version-aware access to live databases. We start with a local snapshot and finish with live queries.

2.4.1 `org.Hs.eg.db`

`org.Hs.eg.db` is a Bioconductor package providing genome-wide annotation for *Homo sapiens*, primarily based on mappings using Entrez Gene identifiers. Common package families include organism-level mappings such as `org.Hs.eg.db` and `org.Mm.eg.db`, assay annotation such as `hgu133plus2.db`, ontology metadata in `GO.db`, transcript models in `TxDb.*`, and Ensembl-based models in `EnsDb.*`. Always record the package version and genome assembly; identifier mappings and annotations change between releases.

Most annotation packages use four standard access methods: `columns()`, `keytypes()`, `keys()`, and `select()`.

Load `org.Hs.eg.db` and inspect its contents.¹

```
suppressPackageStartupMessages({  
  library(org.Hs.eg.db)  
})  
ls("package:org.Hs.eg.db")
```

```
[1] "org.Hs.eg"                "org.Hs.eg_dbconn"  
[3] "org.Hs.eg_dbfile"        "org.Hs.eg_dbInfo"  
[5] "org.Hs.eg_dbschema"      "org.Hs.eg.db"  
[7] "org.Hs.egACCNUM"         "org.Hs.egACCNUM2EG"  
[9] "org.Hs.egALIAS2EG"       "org.Hs.egCHR"  
[11] "org.Hs.egCHRLNGTHS"      "org.Hs.egCHRLOC"  
[13] "org.Hs.egCHRLOCEND"      "org.Hs.egENSEMBL"  
[15] "org.Hs.egENSEMBL2EG"     "org.Hs.egENSEMBLPROT"  
[17] "org.Hs.egENSEMBLPROT2EG" "org.Hs.egENSEMBLTRANS"  
[19] "org.Hs.egENSEMBLTRANS2EG" "org.Hs.egENZYME"  
[21] "org.Hs.egENZYME2EG"      "org.Hs.egGENENAME"  
[23] "org.Hs.egGENETYPE"       "org.Hs.egGO"  
[25] "org.Hs.egGO2ALLEGS"      "org.Hs.egGO2EG"  
[27] "org.Hs.egMAP"            "org.Hs.egMAP2EG"  
[29] "org.Hs.egMAPCOUNTS"     "org.Hs.egOMIM"  
[31] "org.Hs.egOMIM2EG"        "org.Hs.egORGANISM"  
[33] "org.Hs.egPATH"           "org.Hs.egPATH2EG"  
[35] "org.Hs.egPFAM"           "org.Hs.egPMID"  
[37] "org.Hs.egPMID2EG"        "org.Hs.egPROSITE"  
[39] "org.Hs.egREFSEQ"         "org.Hs.egREFSEQ2EG"  
[41] "org.Hs.egSYMBOL"         "org.Hs.egSYMBOL2EG"  
[43] "org.Hs.egUCSCCKG"        "org.Hs.egUNIPROT"
```

To learn which kinds of data are retrievable via `select()`, list the available columns:

¹`suppressPackageStartupMessages()` hides the package start-up messages, which is convenient when generating a document.

```
columns(org.Hs.eg.db)
```

```
[1] "ACCNUM"      "ALIAS"      "ENSEMBL"    "ENSEMBLPROT"
[5] "ENSEMBLTRANS" "ENTREZID"   "ENZYME"     "EVIDENCE"
[9] "EVIDENCEALL"  "GENENAME"   "GENETYPE"   "GO"
[13] "GOALL"        "IPI"        "MAP"        "OMIM"
[17] "ONTOLOGY"     "ONTOLOGYALL" "PATH"       "PFAM"
[21] "PMID"         "PROSITE"    "REFSEQ"     "SYMBOL"
[25] "UCSCKG"       "UNIPROT"
```

You can retrieve help for a particular column with, e.g., `help("GO")`.

Query keys are available through `keytypes()`:

```
keytypes(org.Hs.eg.db)
```

```
[1] "ACCNUM"      "ALIAS"      "ENSEMBL"    "ENSEMBLPROT"
[5] "ENSEMBLTRANS" "ENTREZID"   "ENZYME"     "EVIDENCE"
[9] "EVIDENCEALL"  "GENENAME"   "GENETYPE"   "GO"
[13] "GOALL"        "IPI"        "MAP"        "OMIM"
[17] "ONTOLOGY"     "ONTOLOGYALL" "PATH"       "PFAM"
[21] "PMID"         "PROSITE"    "REFSEQ"     "SYMBOL"
[25] "UCSCKG"       "UNIPROT"
```

Get keys of a particular type with `keys()` — for example the first gene symbols, OMIM entries, and GO entries:

```
head(keys(org.Hs.eg.db, keytype = "SYMBOL"))
```

```
[1] "A1BG" "A2M" "A2MP1" "NAT1" "NAT2" "NATP"
```

```
head(keys(org.Hs.eg.db, keytype = "OMIM"))
```

```
[1] "138670" "103950" "108345" "243400" "612182" "107280"
```

```
head(keys(org.Hs.eg.db, keytype = "GO"))
```

```
[1] "GO:0002764" "GO:0005576" "GO:0005615" "GO:0005886" "GO:0031093"
[6] "GO:0034774"
```

Now define a general query on the `org.Hs.eg.db` object through `select()`.² You specify: **keys** (what to look up), **columns** (what to return), and **keytype** (the type of key you pass in). For example, query all *Gene Ontology* entries for coagulation Factor V, whose gene symbol is F5:

²Use `mapIds()` if you need only a single column.

```
select(org.Hs.eg.db, keys = "F5",
       columns = c("GO", "GENENAME"), keytype = "SYMBOL")
```

	SYMBOL	GO	EVIDENCE	ONTOLOGY	GENENAME
1	F5	GO:0005507	IEA	MF	coagulation factor V
2	F5	GO:0005515	IPI	MF	coagulation factor V
3	F5	GO:0005576	NAS	CC	coagulation factor V
4	F5	GO:0005576	TAS	CC	coagulation factor V
5	F5	GO:0005615	IEA	CC	coagulation factor V
6	F5	GO:0005788	TAS	CC	coagulation factor V
7	F5	GO:0005886	TAS	CC	coagulation factor V
8	F5	GO:0007596	IBA	BP	coagulation factor V
9	F5	GO:0008015	IBA	BP	coagulation factor V
10	F5	GO:0016020	HDA	CC	coagulation factor V
11	F5	GO:0030134	TAS	CC	coagulation factor V
12	F5	GO:0031091	IBA	CC	coagulation factor V
13	F5	GO:0031093	TAS	CC	coagulation factor V
14	F5	GO:0032571	IEA	BP	coagulation factor V
15	F5	GO:0033116	TAS	CC	coagulation factor V
16	F5	GO:1903561	HDA	CC	coagulation factor V

`select()` scales to the whole genome, and the *shape* of the result teaches you something about annotation itself. As a worked example, count how many OMIM records each UniProt accession links to, and look at the distribution (Figure 2.1): most proteins carry none or a few cross-references, while a long tail of heavily curated proteins links to dozens of OMIM entries.

```
library(dplyr)
omim_uniprot <- AnnotationDbi::select(org.Hs.eg.db,
                                     keys = keys(org.Hs.eg.db, keytype = "OMIM"),
                                     columns = "UNIPROT", keytype = "OMIM") |>
  filter(!is.na(UNIPROT)) |>
  distinct(OMIM, UNIPROT) |>
  count(UNIPROT, name = "n_omim")

all_uniprot <- data.frame(UNIPROT = keys(org.Hs.eg.db, keytype = "UNIPROT")) |>
  left_join(omim_uniprot, by = "UNIPROT")
all_uniprot$n_omim[is.na(all_uniprot$n_omim)] <- 0

ggplot(all_uniprot, aes(n_omim)) +
  geom_histogram(bins = 19, fill = "#0e8d7c") +
  scale_y_log10() +
  labs(x = "OMIM cross-references per UniProt accession",
       y = "proteins (log scale)")
```

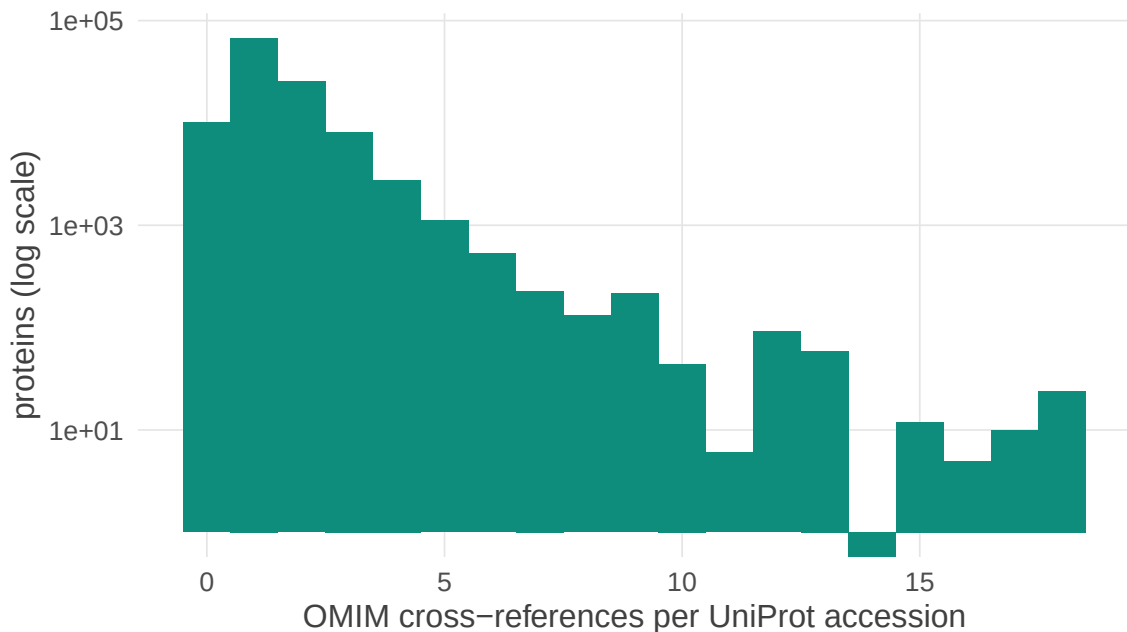


Figure 2.1: Number of distinct OMIM identifiers cross-referenced per UniProt accession (log scale).

i A note on name clashes

`dplyr` also has a `select()` function, and whichever package was loaded *last* wins. Once `library(dplyr)` has run, call the annotation query with its full name `AnnotationDbi::select(...)` (or `org.Hs.eg.db`'s own `select`) to be unambiguous — this is the standard way R resolves name conflicts between packages.

💡 > Exercise 2.7 — `org.Hs.eg.db`

Using the *genome-wide annotation for Human* package (`org.Hs.eg.db`), work through these questions in order — each builds on the previous one. As you go, keep a single tidy data frame of OMIM × UniProt (or OMIM × SYMBOL) pairs and reuse it rather than re-querying the package:

1. How many distinct OMIM identifiers does the annotation package contain? (Note that OMIM identifiers can represent gene *and* phenotype records.)
2. What fraction of the UniProt accessions in `org.Hs.eg.db` have at least one OMIM cross-reference? And what fraction of gene symbols (SYMBOL) do?
3. List the five proteins linked to the largest number of distinct OMIM identifiers, with their gene names. What do these proteins have in common?
4. Compute the number of genes (SYMBOL) and the number of proteins (UNIPROT) in the package. Why do the two counts differ — and why is neither count “the number of genes in the human genome”?
5. Build a table of the gene–protein correspondence. Which are the 10 genes with the largest number of related proteins? Find all UniProt IDs for the first gene in that list, and explain why it has so many.

2.4.2 Live database access with rentrez

`org.Hs.eg.db` is a **local snapshot**: it installs once, works offline, and is perfectly reproducible — but it only knows what was true at its release date, and NCBI and Ensembl keep moving. For **live** queries use the packages that drive the same web APIs you clicked through above: **rentrez** speaks the NCBI E-utilities behind Entrez, and **biomaRt** speaks to the Ensembl BioMart behind the Ensembl browser. A pragmatic workflow is to prototype live, then pin a snapshot (package version or API date) for anything you publish.

rentrez follows the web workflow you did by hand: `entrez_search()` finds records, `entrez_summary()` fetches their metadata, `entrez_fetch()` downloads full records. Let us revisit the LDLR gene from Exercise 2.2 — this time from R:

```
library(rentrez)
ldlr <- entrez_search(db = "gene", term = "LDLR[Gene Name] AND Homo sapiens[Organism]")
ldlr$ids
```

```
[1] "3949"
```

```
entrez_summary("gene", id = ldlr$ids)
```

esummary result with 20 items:

[1] uid	name	description
[4] status	currentid	chromosome
[7] geneticsource	maplocation	otheraliases
[10] otherdesignations	nomenclaturesymbol	nomenclaturename
[13] nomenclaturestatus	mim	genomicinfo
[16] geneweight	summary	chrsort
[19] chrstart	organism	

The summary list behaves like a data frame: `extract_from_esummary()` pulls the fields you need. Note how the *live* answer — description, chromosome, coordinates, last update — matches what you found on the web page:

```
ldlr_sum <- entrez_summary("gene", id = ldlr$ids)
extract_from_esummary(ldlr_sum, c("name", "description", "chromosome"))
```

\$name

```
[1] "LDLR"
```

\$description

```
[1] "low density lipoprotein receptor"
```

\$chromosome

```
[1] "19"
```

The full record (the equivalent of the web page's *Genomic regions, transcripts, and products* table) comes from `entrez_fetch()`:

```
ldlr_tab <- entrez_fetch("gene", id = ldlr$ids, rettype = "gene_table", retmode = "text")
substr(ldlr_tab, 1, 400) # first lines of the gene table
```

```
[1] "LDLR low density lipoprotein receptor[Homo sapiens]\nGene ID: 3949, updated on 13-
Sep-2026\n\n\nReference GRCh38.p14 Primary Assembly NC_000019.10 from: 11089463 to: 1113382"
```

The same search pattern drives PubMed. The query from Exercise 2.3 becomes an `entrez_search()` call, and `retmax` caps how many records come back:

```
es <- entrez_search(db = "pubmed",
  term = "COVID-19[Title/Abstract] AND China[Affiliation]",
  retmax = 3)
es$count # total number of matching publications
```

```
[1] 46620
```

```
es$ids # the first 3 PMIDs
```

```
[1] "42747775" "42746260" "42745786"
```

💡 > Exercise 2.8 — rentrez

1. Fetch the *titles* and *first authors* of the three publications found above (`extract_from_esummary()` on the PubMed summaries; the fields are `title` and `sortfirstauthor`).
2. The first `rentrez` query in this section used a *search*. Repeat it the other way round: fetch the full **gene_table** record for Entrez Gene ID **3949** directly with `entrez_fetch()` (hint: `id = 3949`).
3. `es$count` changes daily. Which of the two numbers you computed in this section would you report in a paper, and how would you make it reproducible?

`biomaRt` is the Ensembl counterpart (it needs network access and an Ensembl mirror, so it is shown here rather than run). It queries the same Ensembl database you browsed in Exercise 2.4:

```
library(biomaRt)
mart <- useEnsembl("genes", dataset = "hsapiens_gene_ensembl")
getBM(attributes = c("hgnc_symbol", "chromosome_name", "start_position", "end_position"),
  filters = "hgnc_symbol", values = "LDLR", mart = mart)
```

2.5 Further reading

- [NCBI Datasets documentation](#) and the [BLAST topics guide](#).
- [Ensembl Help](#) — gene/transcript pages and BioMart.
- The `AnnotationDbi` vignette (`vignette("IntroToAnnotationPackages")`) for `select()/mapIds()`.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

2.6 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

Part II

Genetics & genomes

3 Genetic Association: Quality Control and Stratification

In this lab we explore the steps taken in a genome-wide association (GWA) analysis: exploring a dataset, running a first association scan, discovering that it is confounded, cleaning the data, and re-running the scan. We use the maintained **gaston** package (Hervé Perdry & Claire Dandine-Roulland) together with base R, so the lab no longer depends on the archived GenABEL toolkit.¹

By the end of this laboratory you will be able to:

- explore the phenotype and genotype data of a GWA dataset and summarise its quality
- read minor-allele-frequency and Hardy–Weinberg equilibrium (HWE) statistics
- run an initial genome-wide association scan and read a Manhattan plot
- detect and remove genetic outliers using a genetic relationship matrix and PCA
- apply quality-control filters and re-run the scan, and interpret why the results change

i Requirements

The code in this chapter runs with **gaston** only. The data live in the repo under `data/ge03d2.{bed,bim,fam}` (genotypes) and `data/ge03d2.phe` (phenotypes).² The figures below are generated by executing the code.

The dataset, `ge03d2`, targets **type 2 diabetes** — a disorder characterised by high blood glucose in the context of insulin resistance and relative insulin deficiency. Unlike type 1 diabetes, the insulin resistance is generally “post-receptor”: the problem is with the cells that respond to insulin rather than with insulin production. Type 2 diabetes has a strong heritable component — having first-degree relatives with the disease substantially increases risk — and a mutation in the *Islet Amyloid Polypeptide* gene produces an earlier-onset, more severe form (Cho et al., 2003; Sakagashira et al., 1996; Zimmet et al., 2001). The dataset simulates some of the genetic structure of type 2 diabetes.

Lab at a glance

Goal: run and quality-control a genome-wide association scan. **Prerequisites:** [Introduction to R](#); basic genetics. **Time:** ~2 h. **Packages:** `gaston`. **Data:** `data/ge03d2` (PLINK bed) + `data/ge03d2.phe`. **Deliverable:** a lab report.

Start your R session, load `gaston`, and read the genotypes and phenotypes:

¹Earlier editions of this lab used **GenABEL** (`polygenic` + `mmscore`); GenABEL has since been archived from CRAN. `gaston` reproduces the same family-aware analysis — `lmm.aireml` estimates the polygenic variance components and `association.test(..., method = "lmm")` is the mixed-model score test — and is what we use here.

²`ge03d2` is the simulated type-2-diabetes example that shipped with the GenABEL “getting started with GWA” tutorial (package `GenABEL.data`). Because GenABEL is archived, we exported it once to PLINK bed format with `scripts/make-ge03d2-bed.R` and committed the result, so the lab reads it with `gaston` and never loads GenABEL. Provenance and archived sources: [GenABEL.data on the CRAN archive](#), [the GenABEL archive](#), and Aulchenko et al. (2007), *Bioinformatics* **23**(10):1294–1296.

```
library(gaston)
geno <- read.bed.matrix("data/ge03d2")
```

Reading data/ge03d2.rds
Reading data/ge03d2.bed

```
pheno <- read.table("data/ge03d2.phe", header = TRUE)
```

`geno` is a **bed.matrix**: one row per individual, one column per marker, with annotation tables `@ped` (individuals) and `@snps` (markers). We ask **gaston** to compute the per-marker and per-individual summary statistics, and the HWE test:

```
geno <- set.stats(geno, verbose = FALSE) # call rates, MAF, heterozygosity
geno <- set.hwe(geno, verbose = FALSE)   # Hardy-Weinberg exact-test p-values
geno
```

A `bed.matrix` with 950 individuals and 7589 markers.
snps stats are set
There are 2 SNPs with a callrate equal to zero
There is one monomorphic SNP
ped stats are set

The individuals in `@ped` are in the same order as the rows of `pheno`, so we can use the phenotype columns directly. The autosomes are chromosomes 1–3; the X chromosome is handled separately.

```
auto <- geno@snps$chr %in% c("1", "2", "3")
```

3.1 Data exploration

We first explore the features of the dataset: the number of samples, the phenotypes and their distribution, and the number and quality of the genotypes.

3.1.1 Phenotype data

The phenotype table holds the traits for each individual. `str()` reveals the internal structure; `summary()` gives per-trait distributions:

```
str(pheno)
```

```
'data.frame': 950 obs. of 8 variables:
 $ id      : chr  "id4" "id10" "id25" "id33" ...
 $ sex     : int   0 1 0 0 0 0 1 1 0 1 ...
 $ age     : num   51.6 53.7 66 44.7 49.9 ...
 $ dm2     : int   1 1 1 1 1 1 1 1 1 1 ...
 $ height  : num   152 170 165 174 157 ...
```

```
$ weight: num 83 65.6 69.6 69.7 84.3 ...
$ diet : int 0 0 0 0 0 0 0 0 0 0 ...
$ bmi : num 36 22.6 25.6 22.9 34.1 ...
```

```
summary(pheno[, c("sex", "age", "dm2", "bmi")])
```

sex	age	dm2	bmi
Min. :0.0000	Min. :18.64	Min. :0.0000	Min. :13.49
1st Qu.:0.0000	1st Qu.:39.99	1st Qu.:0.0000	1st Qu.:24.28
Median :0.0000	Median :49.66	Median :0.0000	Median :27.94
Mean :0.4895	Mean :50.00	Mean :0.4874	Mean :29.68
3rd Qu.:1.0000	3rd Qu.:59.34	3rd Qu.:1.0000	3rd Qu.:34.11
Max. :1.0000	Max. :87.60	Max. :1.0000	Max. :64.22
			NA's :2

The target trait `dm2` is the binary type-2-diabetes status (0 = control, 1 = case). `table()` shows the case/control balance:

```
table(dm2 = pheno$dm2)
```

```
dm2
 0   1
487 463
```

💡 > Exercise 3.1 — Phenotypes

1. What is the number of individuals in the study?
2. Split the traits by diabetes status (e.g. `by(pheno[, c("age", "bmi", "height", "weight")], pheno$dm2, summary)`) and check which traits differ most between the diabetic and normal populations.
3. Use a boxplot to explore graphically the relationship between the phenotypes. Which variables differ most between the diabetic and normal populations? Discuss.

For example, a boxplot of body-mass index by diabetes status (Figure 3.1):

```
boxplot(bmi ~ dm2, data = pheno,
        xlab = "Type 2 Diabetes", ylab = "Body Mass Index")
```

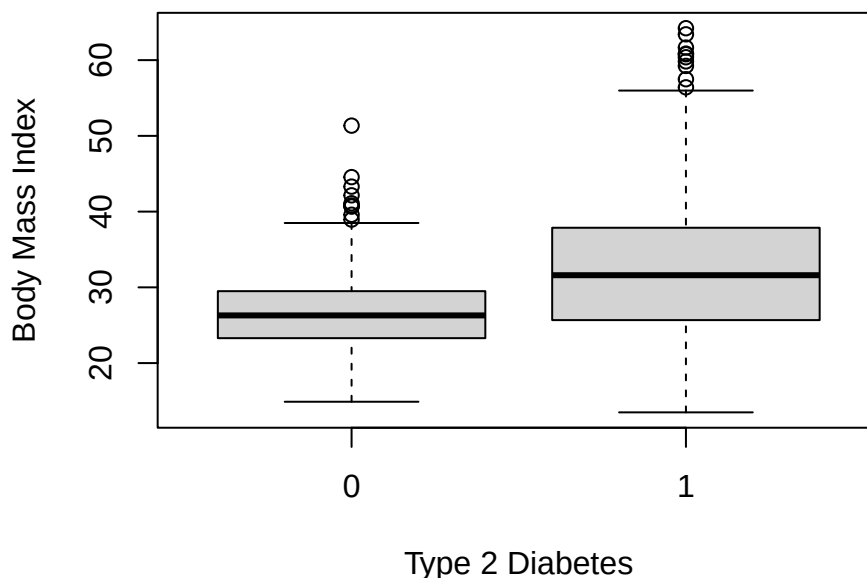


Figure 3.1: Body-mass index by type 2 diabetes status.

3.1.2 Genotype data

The `@snps` table now carries, for every marker, the call rate, the minor allele frequency (`maf`), the observed heterozygosity (`hz`) and the Hardy–Weinberg p-value (`hwe`). We summarise the autosomal markers:

```
summary(geno@snps[auto, c("callrate", "maf", "hz", "hwe")])
```

callrate	maf	hz	hwe
Min. :0.0000	Min. :0.00000	Min. :0.0000	Min. :0.0000
1st Qu.:0.9863	1st Qu.:0.05906	1st Qu.:0.1041	1st Qu.:0.2024
Median :0.9884	Median :0.14271	Median :0.2406	Median :0.4709
Mean :0.9823	Mean :0.18125	Mean :0.2538	Mean :0.4692
3rd Qu.:0.9905	3rd Qu.:0.28638	3rd Qu.:0.4086	3rd Qu.:0.7353
Max. :0.9979	Max. :0.50000	Max. :0.5464	Max. :1.0000
	NAs :2	NAs :2	NAs :2

Minor allele frequency measures how common the rarer allele is; markers that are (nearly) monomorphic carry almost no information. **Hardy–Weinberg equilibrium** relates genotype frequencies to allele frequencies in a large, randomly mating population; a strong departure often flags a genotyping error — or genuine biology such as selection or population structure. Count the markers that fail typical thresholds:

```
c(monomorphic      = sum(geno@snps$maf[auto] == 0, na.rm = TRUE),
   low_callrate     = sum(geno@snps$callrate[auto] < 0.95, na.rm = TRUE),
   hwe_p_below_1e_6 = sum(geno@snps$hwe[auto] < 1e-6, na.rm = TRUE))
```

monomorphic	low_callrate	hwe_p_below_1e_6
1	76	337

You can compare the HWE distribution between the two populations (diabetes / no diabetes) by re-computing it within each group:

```
hwe_by <- function(rows) {  
  g <- set.hwe(geno[rows, auto], verbose = FALSE)  
  sum(g@snps$hwe < 1e-6, na.rm = TRUE)  
}  
c(controls = hwe_by(pheno$dm2 == 0),  
  cases    = hwe_by(pheno$dm2 == 1))
```

```
controls    cases  
      61      360
```

💡 > Exercise 3.2 — HWE

Do you see a difference in the number of HWE-violating markers between the two populations? Can you explain why a population might be out of HWE?

3.2 Association

3.2.1 Initial scan

The target phenotype (dm2) is binary, so a first association solution is a **logistic regression per marker**, adjusting for age, sex and BMI. `gaston::association.test()` does exactly this with `response = "binary"`; we pass the covariates (plus an intercept column) in `X`. We restrict the scan to individuals with complete covariates and to the autosomes.

```
ok <- with(pheno, !is.na(dm2) & !is.na(age) & !is.na(sex) & !is.na(bmi))  
Xc <- cbind(1, pheno$age[ok], pheno$sex[ok], pheno$bmi[ok])  
gwa1 <- association.test(geno[ok, auto], Y = pheno$dm2[ok], X = Xc,  
                        method = "lm", response = "binary", test = "wald")  
nrow(gwa1)
```

```
[1] 7020
```

The scan output has one row per marker with an association p-value. A **Manhattan plot** shows genome-wide significance ($-\log_{10} p$) along the genome (Figure 3.2):

```
manhattan(gwa1, pch = 16)
```

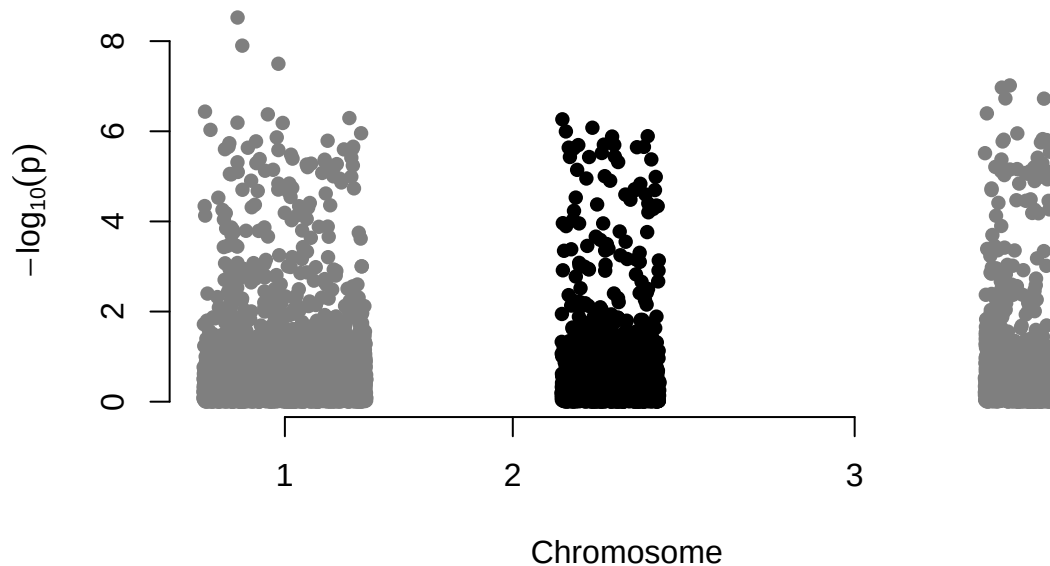


Figure 3.2: Manhattan plot of genome-wide significance (initial scan, before quality control).

The most significant markers are ranked by p-value:

```
head(gwa1[order(gwa1$p), c("chr", "pos", "id", "p")], 10)
```

	chr	pos	id	p
819	1	1047389	rs7903146	2.992996e-09
912	1	1191492	rs9350372	1.258915e-08
1655	1	2300686	rs8284432	3.177882e-08
6102	3	10762647	rs6520012	9.625007e-08
5946	3	10518480	rs2975760	1.074630e-07
6014	3	10627900	rs6577581	1.865315e-07
6873	3	11813645	rs6934167	1.888137e-07
32	1	47084	rs2689086	3.664227e-07
5600	3	10058730	rs3421751	4.010638e-07
1429	1	1976325	rs4232295	4.213910e-07

3.2.1.1 Is the scan well calibrated?

Before trusting any hit we check **calibration** with a QQ plot — observed $-\log_{10}$ p-values against those expected under the null — and the **genomic inflation factor** λ , the ratio of the observed median test statistic to its null expectation. $\lambda \approx 1$ means confounding is under control; $\lambda \gg 1$ signals population structure or genotyping artefacts inflating the whole scan (Figure 3.3).

```
genomic_lambda <- function(p) {
  p <- p[is.finite(p) & p > 0]
  median(qchisq(1 - p, df = 1)) / qchisq(0.5, df = 1)
}
lambda1 <- genomic_lambda(gwa1$p)
```

```
qqplot.pvalues(gwa1$p)
title(sub = sprintf("Initial scan: lambda = %.2f", lambda1))
```

QQ plot of p-values

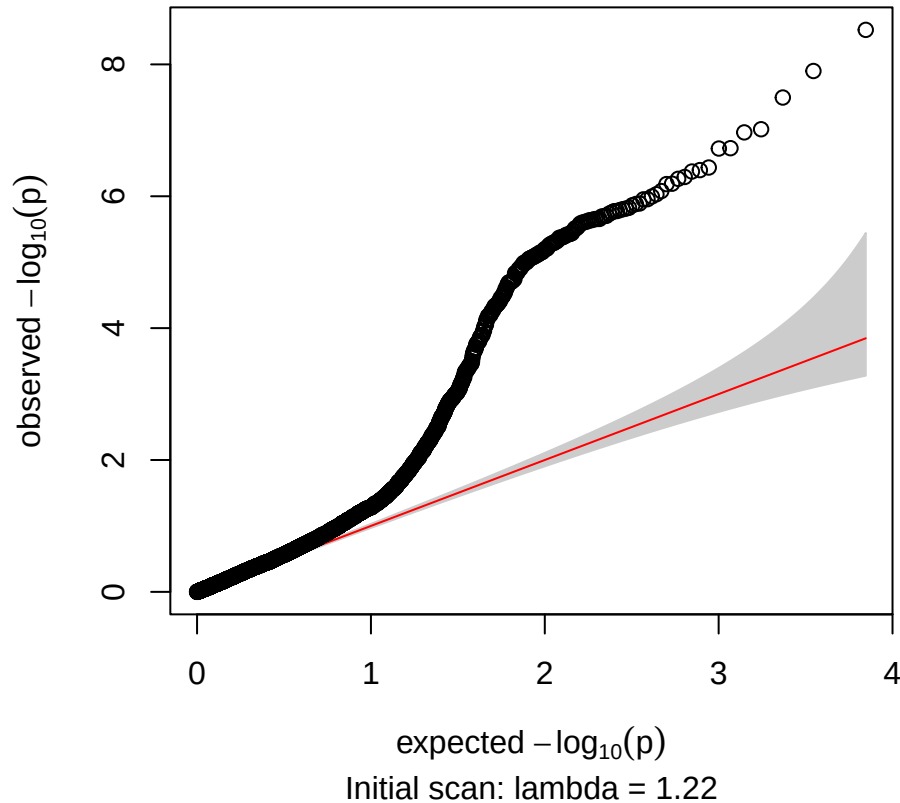


Figure 3.3: QQ plot of the initial scan. Points lifting off the diagonal early indicate inflation.

The inflation factor is $\lambda = 1.22$ — above 1, a red flag that the raw scan is confounded (exactly what the quality control below will address).

! Multiple testing

With thousands of markers tested at once, an uncorrected $p < 0.05$ yields hundreds of false positives. Judge hits against a genome-wide threshold — Bonferroni ($0.05/m$ for m markers) or a controlled false-discovery rate (FDR, Benjamini–Hochberg via `p.adjust(gwa1$p, "BH")`).

💡 > Exercise 3.3 — Initial scan

Can you find whether any of the markers in the table have a clinical relationship? What genome-wide significance threshold would you use for m markers?

3.2.2 Quality control

The odd HWE distribution and probable genotyping errors are blinding our analysis. Before a second scan we clean the dataset. We drop **markers** with a low call rate, a very low minor allele

frequency, or a gross HWE violation, and we drop **individuals** with a low call rate or extreme genome-wide heterozygosity (a classic signal of sample contamination or poor DNA quality).

```
snp_ok <- geno@snps$callrate >= 0.95 &
          geno@snps$maf >= 0.01 &
          (is.na(geno@snps$hwe) | geno@snps$hwe >= 1e-6)
c(autosomal_markers = sum(auto), pass_qc = sum(auto & snp_ok))
```

```
autosomal_markers      pass_qc
              7020          6389
```

```
het <- geno@ped$hz
ind_ok <- geno@ped$callrate >= 0.95 &
          abs(het - mean(het)) <= 3 * sd(het)
c(individuals = nrow(geno), low_callrate = sum(geno@ped$callrate < 0.95),
  het_outliers = sum(abs(het - mean(het)) > 3 * sd(het)),
  pass_qc = sum(ind_ok))
```

```
individuals low_callrate het_outliers      pass_qc
              950              4              8              938
```

Next we look for **genetic outliers** — a hidden subpopulation that, mixed with the rest, inflates the whole scan. We build the **genetic relationship matrix** (GRM) from common autosomal markers and take its leading **principal components**: individuals cluster by ancestry along the first PCs.³ Figure 3.4 shows the result.

```
gc <- geno[ok, auto & geno@snps$maf > 0.05 & geno@snps$callrate > 0.95]
standardize(gc) <- "p"
K <- GRM(gc)
pcs <- eigen(K, symmetric = TRUE)$vectors[, 1:2]
colnames(pcs) <- c("PC1", "PC2")
km <- kmeans(pcs, centers = 2, nstart = 50)
major <- as.integer(names(which.max(table(km$cluster))))
table(cluster = km$cluster)
```

```
cluster
 1  2
25 923
```

```
plot(pcs, col = ifelse(km$cluster == major, "grey40", "firebrick"),
     pch = 16, xlab = "PC 1", ylab = "PC 2")
legend("topright", c("main population", "outliers"),
     col = c("grey40", "firebrick"), pch = 16, bty = "n")
```

³This replaces the older Identity-By-State + multidimensional-scaling recipe. Both project a matrix of genetic similarity between every pair of individuals into a few dimensions; the eigendecomposition (PCA) of the GRM is the modern standard and is what tools such as PLINK and EIGENSTRAT use.

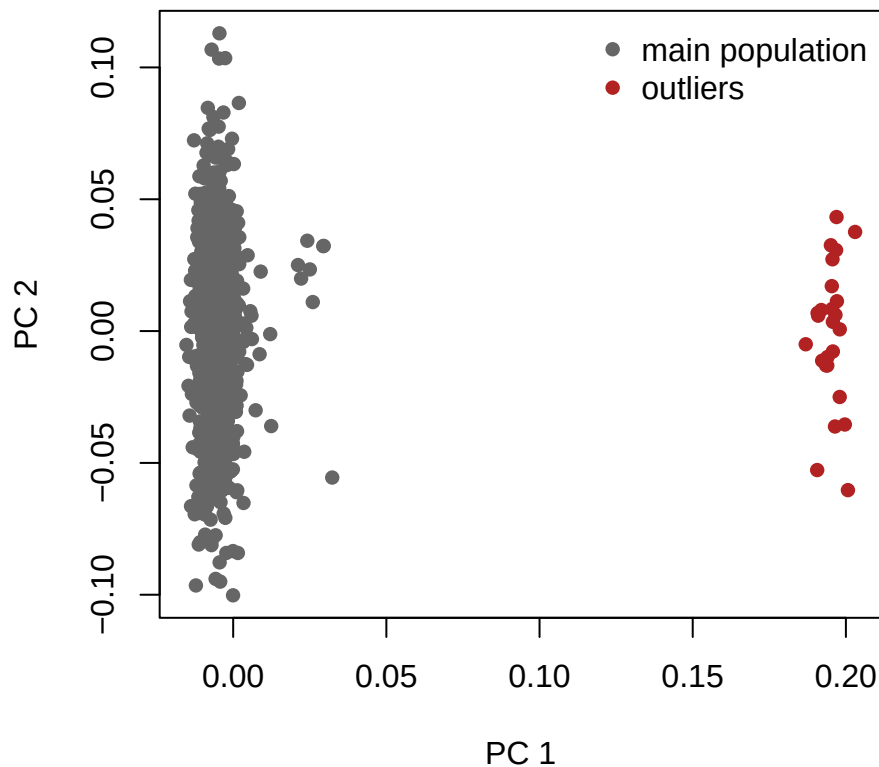


Figure 3.4: Genetic outliers in the first two principal components of the GRM. The minor cluster is the outlier subpopulation.

The minor cluster is discarded as an outlier population, as can be understood graphically from Figure 3.4.

i An alternative to *removing* the outliers is to keep everyone and add the leading principal components as covariates in the association model ($X = \text{cbind}(1, \text{PC1}, \text{PC2}, \text{age}, \text{sex}, \text{bmi})$). Both strategies control the stratification; here we remove the small outlier cluster so the effect on λ is easy to see.

💡 > Exercise 3.4 — Outliers

1. Re-run the k-means with a different number of centres, or colour the PCA by `dm2`. Does the minor cluster correspond to cases, controls, or neither?
2. A **sex check** compares the reported sex against the heterozygosity on the X chromosome (true males are hemizygous, so their X heterozygosity is ~ 0). Using `geno@snps$chr == "X"`, compute per-individual X heterozygosity and look for mismatches. Why is this an important QC step?

We now assemble the clean dataset: the individuals in the main cluster that also pass the per-individual QC, and the markers that pass the per-marker QC.

```
keep_ind <- which(ok)[km$cluster == major]
keep_ind <- keep_ind[ind_ok[keep_ind]]
geno_qc <- geno[keep_ind, auto & snp_ok]
pheno_qc <- pheno[keep_ind, ]
c(individuals = nrow(geno_qc), markers = ncol(geno_qc))
```

```

individuals      markers
          911      6389

```

3.2.3 Final scan

With the dataset clean — free of the stratified outliers, low-quality markers and individuals — we re-run the genome-wide scan against the type 2 diabetes phenotype (Figure 3.5):

```

Xc2 <- cbind(1, pheno_qc$age, pheno_qc$sex, pheno_qc$bmi)
gwa2 <- association.test(geno_qc, Y = pheno_qc$dm2, X = Xc2,
                        method = "lm", response = "binary", test = "wald")
lambda2 <- genomic_lambda(gwa2$p)

```

```

manhattan(gwa2, pch = 16)

```

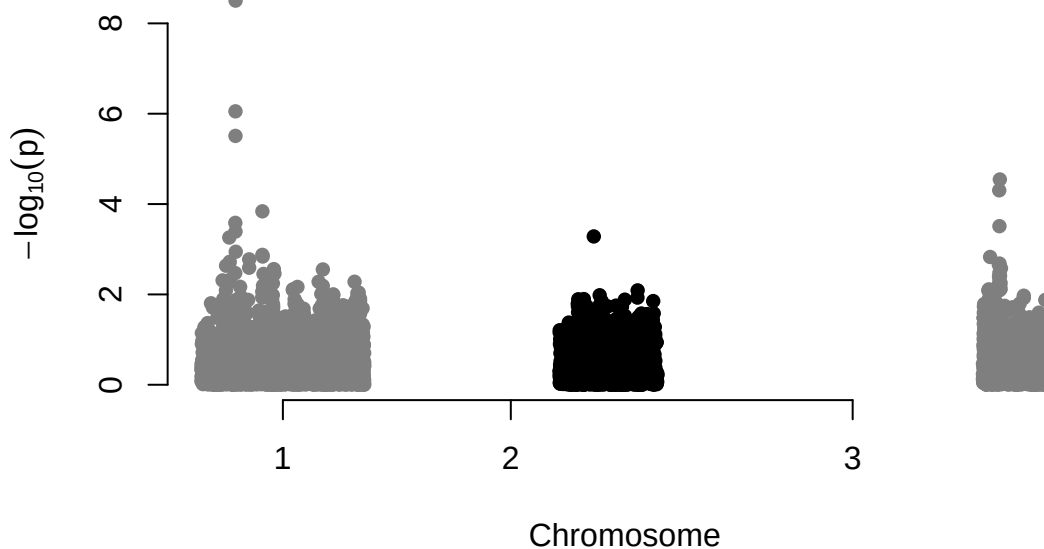


Figure 3.5: Manhattan plot of genome-wide significance after quality control of the dataset.

Compare these results with the first markers from the Section 3.2.1.

```

head(gwa2[order(gwa2$p), c("chr", "pos", "id", "p")], 10)

```

	chr	pos	id	p
752	1	1047389	rs7903146	3.131508e-09
747	1	1043860	rs289981	8.855194e-07
749	1	1044233	rs7064741	3.095833e-06
5416	3	10518480	rs2975760	2.852695e-05
5400	3	10500286	rs6083205	4.960372e-05
1237	1	1870260	rs3075920	1.446580e-04
743	1	1039030	rs8398809	2.605098e-04
5409	3	10506420	rs9468596	3.084826e-04
750	1	1044797	rs6227551	4.059546e-04
3892	2	7048058	rs6079246	5.206342e-04

Re-checking calibration after QC, the inflation should have fallen back towards 1 (Figure 3.6):

```
qqplot.pvalues(gwa2$p)
title(sub = sprintf("After QC: lambda = %.2f", lambda2))
```

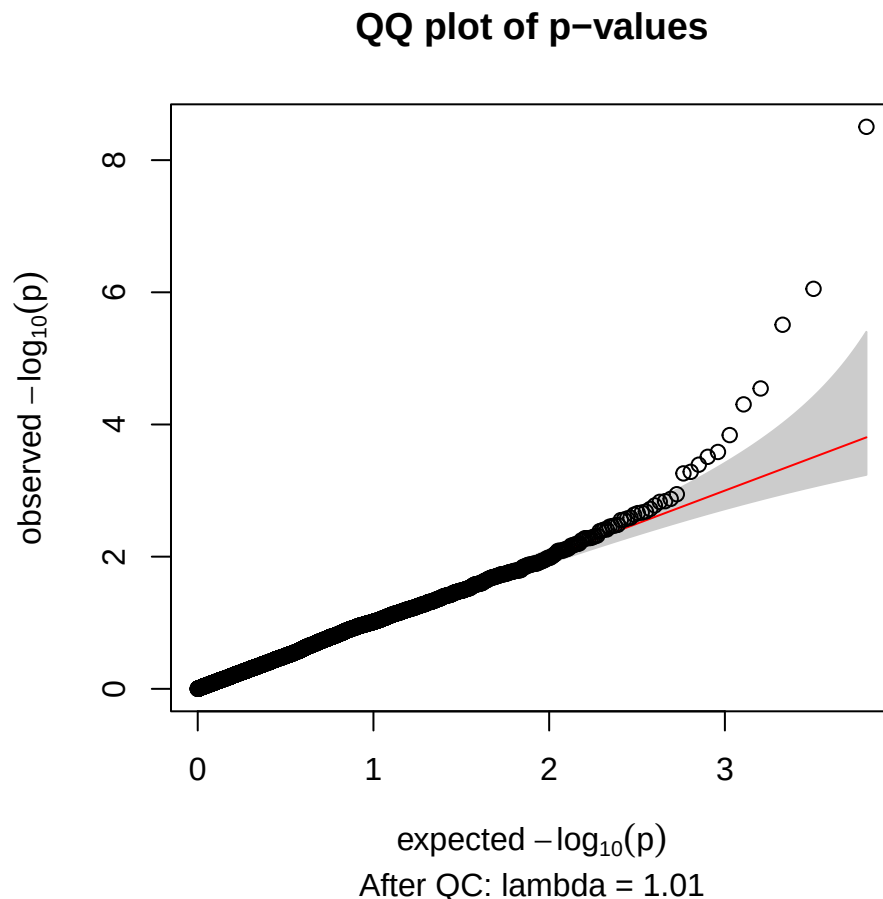


Figure 3.6: QQ plot after quality control — the bulk of points now track the null diagonal, with a few genuine hits departing at the tail.

Inflation dropped from $\lambda = 1.22$ (initial) to $\lambda = 1.01$ after QC: removing the stratified outliers and bad markers de-confounded the scan. The strongest surviving hit is **rs7903146**, a variant in the *TCF7L2* gene and the best-established common risk allele for type 2 diabetes — a reassuring positive control.

💡 > Exercise 3.5 — Final scan

1. Look up the top markers (e.g. rs7903146) in [dbSNP](#) or the GWAS Catalog. Do any have a clinical relationship to type 2 diabetes?
2. Can you explain, in your own words, why the results differ from the initial scan?

3.3 Further reading

- Aulchenko *et al.* (2007), *Bioinformatics* **23**:1294–1296 — the original GenABEL genome-wide association framework.
- The [gaston vignettes](#) — `association.test`, the GRM, and the mixed models used next.

- [PLINK2 documentation](#) on the bed/bim/fam format.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

3.4 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

4 Genome-Wide Association: a Thrombophilia Case

In this lab we draft a genome-wide association (GWA) study. Genome-wide association is the overall name for a technique that seeks positions in the genome *associated* with a given phenotype. We carry out this laboratory with the maintained **gaston** package (Hervé Perdry & Claire Dandine-Roulland), which fits the linear mixed models needed to account for family relatedness.¹

By the end of this laboratory you will be able to:

- load a real family-based GWA dataset in PLINK/bed format and explore it
- look up SNP markers in dbSNP
- reason about familial structure and kinship, and visualise it
- estimate trait heritability and run a **mixed-model** genome-wide scan that accounts for relatedness
- read a Manhattan and QQ plot, and identify and interpret the top associated SNP

Lab at a glance

Goal: run a family-based GWAS with a linear mixed model and reason about relatedness and confidentiality. **Prerequisites:** [Genetic Association](#). **Time:** ~2 h. **Packages:** **gaston**. **Data:** the GAIT dataset in **gaston/bed** format (see below). **Deliverable:** a lab report.

For this session we use real data from the **GAIT** project (Genetic Analysis of Idiopathic Thrombophilia), kindly provided by Dr. Soria (Unit of Genomics and Complex Diseases, Institut de Recerca de l'Hospital de la Santa Creu i Sant Pau). GAIT is an NIH-funded project studying strong linkage signals for QTLs influencing quantitative risk factors for thrombosis, detected in the 21 extended families that comprise the GAIT sample.

! Confidential data

This dataset is **not public**. The information it contains is privileged, confidential, and protected from disclosure. Unauthorised use, reproduction, or distribution of the dataset — or any portion of it, or any result derived from it — is strictly forbidden, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law. If you received this data in error, notify us immediately (alexandre.perera@upc.edu).

The genotypes are stored in the widely used **PLINK bed** format, which **gaston** reads by slices. A bed dataset is three files: **.bed** (genotypes), **.fam** (individuals), and **.bim** (marker map). If you only have the legacy LabGWAS1.Rdata, convert it once with the helper script (`Rscript scripts/make-gaston-bed.R`), which writes **objQ2.bed/.bim/.fam** and **objQ2.phe** into **data/**.

¹Earlier editions of this lab used **GenABEL** (`polygenic + mmscore`); GenABEL has since been archived from CRAN. **gaston** reproduces the same family-aware analysis — `lmm.aireml` estimates the polygenic variance components and `association.test(..., method = "lmm")` is the mixed-model score test — and is what we use here.

```
library(gaston)
geno <- read.bed.matrix("data/objQ2")
```

Reading data/objQ2.rds
Reading data/objQ2.bed

```
pheno <- read.table("data/objQ2.phe", header = TRUE)
```

`geno` is a **bed.matrix**: two annotation tables, `@ped` (individuals) and `@snps` (markers), plus a pointer to the packed genotypes. `pheno` holds the phenotype table.

4.1 Data exploration

```
geno
```

A `bed.matrix` with 377 individuals and 266966 markers.
snps stats are set
ped stats are set

The `@ped` table counts, per individual, the genotypes equal to 0/1/2 and missing (N0, N1, N2, NAs), the call rate and heterozygosity; `@snps` holds the alternate allele frequency `p`, its expected genotype `mu = 2p`, and the genotype standard deviation `sigma`. Under Hardy–Weinberg equilibrium `sigma` should track $\sqrt{2p(1-p)}$ (Figure 4.1):

```
plot(geno@snps$p, geno@snps$sigma, xlim = c(0, 1), pch = ".",
     xlab = "allele frequency p", ylab = expression(sigma))
t <- seq(0, 1, length = 101)
lines(t, sqrt(2 * t * (1 - t)), col = "#c0432f", lwd = 2)
```

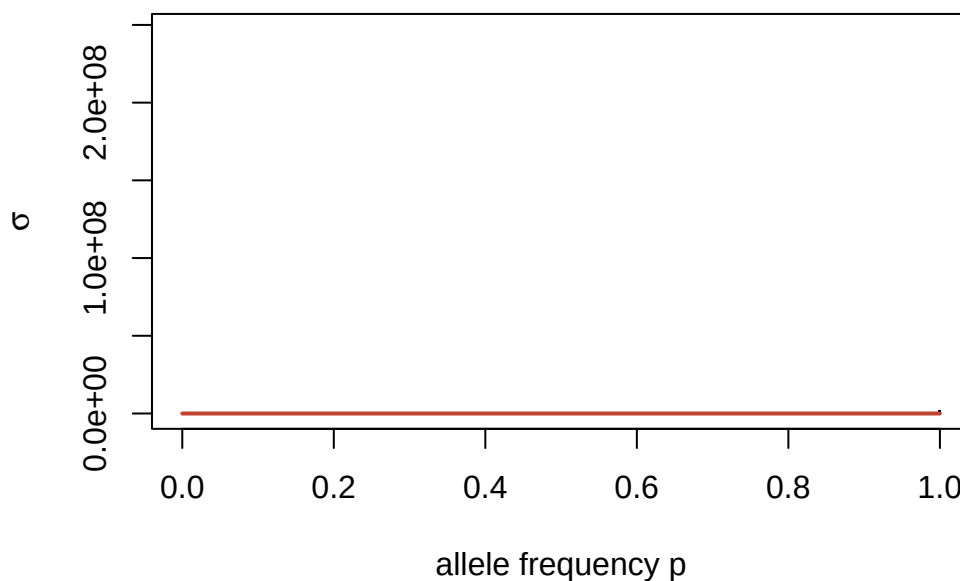


Figure 4.1: Genotype standard deviation against alternate-allele frequency; the red curve is the Hardy–Weinberg expectation.

💡 > Exercise 4.1 — Data exploration

1. What is the number of individuals in the study? And the number of families (`pheno$FAM`)?
2. How many members were recruited from each family?
3. Does the p - σ cloud follow the H-W expectation? Where would you expect departures?
4. Build a table of the Smoking (`SMOKIN`) vs Gender (`sex`) data.

4.1.1 Genotype data

The genotypes cast to a numeric 0/1/2 matrix with `as.matrix()`; the `rs` codes are the column names. The first few markers:

```
as.matrix(geno)[1:5, 1:5]
```

	rs3934834	rs3737728	rs6687776	rs9651273	rs4970405
1203	0	1	0	1	0
1204	0	0	0	1	0
1208	0	1	0	1	0
1209	0	1	0	1	0
1210	0	1	0	2	0

Look up a marker on the dbSNP website (<https://www.ncbi.nlm.nih.gov/snp/>) — a public catalogue of Single Nucleotide Polymorphisms. Marker names are `rs` accessions (Reference SNP records).

💡 > Exercise 4.2 — Markers

On which chromosome and position is the first SNP located (`geno@snps[1, ]`)?

4.2 Familial structure

Each individual carries family (`FAM`) and parent (`FA`, `MO`) information. Most individuals are related, and this kinship must be accounted for when computing association statistics — otherwise relatedness inflates the test (recall the genomic-inflation λ from the [Genetic Association](#) lab).

We summarise relatedness with the **Genetic Relationship Matrix** (GRM), the standardised whole-genome analogue of the Identity-By-State idea: for individuals i, j over N SNPs with allele frequency p_k ,

$$K_{i,j} = \frac{1}{N} \sum_{k=1}^N \frac{(x_{i,k} - 2p_k)(x_{j,k} - 2p_k)}{2p_k(1 - p_k)} \quad (4.1)$$

```
K <- GRM(geno)
K[1:5, 1:5]
```

	1203	1204	1208	1209	1210
1203	0.988952637	-0.005105605	0.486439437	-0.017846238	0.373444021
1204	-0.005105605	1.000755310	-0.008204698	0.007200110	-0.016324446
1208	0.486439437	-0.008204698	0.990920424	-0.001985076	0.447332799
1209	-0.017846238	0.007200110	-0.001985076	1.020578146	-0.000605118
1210	0.373444021	-0.016324446	0.447332799	-0.000605118	0.990267515

A hierarchical clustering of $1 - K / \max K$ recovers the family blocks (Figure 4.2):

```
d <- as.dist(1 - K / max(K))
hc <- hclust(d, method = "single")
plot(hc, labels = geno@ped$famid, cex = 0.4, main = "", xlab = "", sub = "")
```

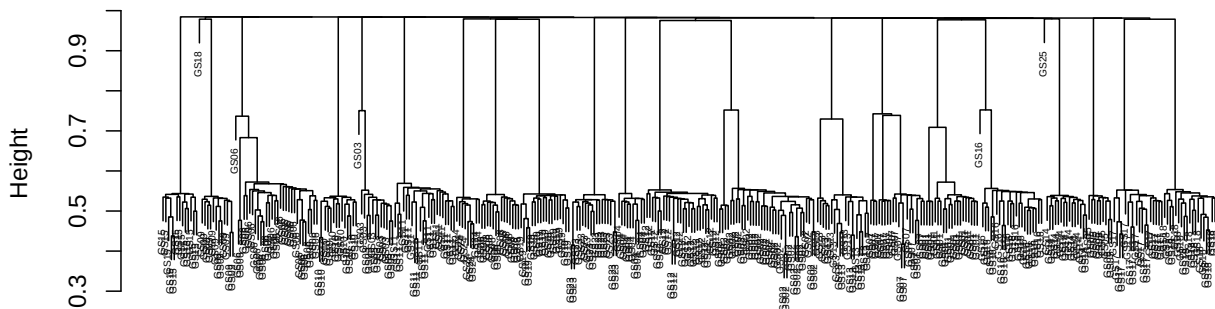


Figure 4.2: Hierarchical clustering from the genetic relationship matrix, labelled by family — members of a pedigree group together.

💡 > Exercise 4.3 — Kinship

1. How many unrelated individuals (founders) are in the dataset (FA/MO missing)?
2. Why is the GRM computed from autosomal SNPs only?
3. Comment on Figure 4.2: do the clusters match the families?

4.3 Association

The target phenotype **XXX** is a critical, highly heritable clinical trait related to hemostasis. Note that **XXX** is a deliberately **anonymised label**: because the dataset is under confidentiality the real trait name is replaced by a placeholder, so treat it as a generic quantitative phenotype and focus on the *method* (kinship accounting), not the specific analyte. We first fit the **polygenic model** — a linear mixed model with a random effect structured by the GRM — to estimate how much of the trait variance is genetic (heritability, h^2). We restrict to individuals with complete phenotype and covariates:

```
Y <- pheno$XXX
Xc <- cbind(AGE = pheno$AGE, sex = pheno$sex)
ok <- complete.cases(Y, Xc)
fit <- lmm.aireml(Y[ok], X = cbind(1, Xc[ok, ]), K = K[ok, ok], verbose = FALSE)
h2 <- fit$tau / (fit$tau + fit$sigma2)
round(c(genetic_var = fit$tau, resid_var = fit$sigma2, heritability = h2), 3)
```

genetic_var	resid_var	heritability
900.390	196.474	0.821

The heritability is high — most of the variance in XXX is genetic, which is why a relatedness-aware model matters. Now the genome-wide **mixed-model score test** (*gaston*'s equivalent of GenABEL's `mmscore`): each SNP is tested for association with XXX, adjusting for age, sex, and the full kinship:

```
assoc <- association.test(geno[ok, ], Y = Y[ok], X = Xc[ok, ],
                        method = "lmm", K = K[ok, ok], test = "score")
```

```
[Iteration 1] theta = 533.503 537.38
[Iteration 1] log L = -1434.21
[Iteration 1] AI-REML update
[Iteration 1] ||gradient|| = 0.0351274
[Iteration 2] theta = 133.063 887.195
[Iteration 2] log L = -1428.5
[Iteration 2] AI-REML update
[Iteration 2] ||gradient|| = 0.0477275
[Iteration 3] theta = 182.052 911.125
[Iteration 3] log L = -1427.16
[Iteration 3] AI-REML update
[Iteration 3] ||gradient|| = 0.0052973
[Iteration 4] theta = 195.404 901.948
[Iteration 4] log L = -1427.13
[Iteration 4] AI-REML update
[Iteration 4] ||gradient|| = 0.000214133
[Iteration 5] theta = 196.419 900.487
[Iteration 5] log L = -1427.13
[Iteration 5] AI-REML update
[Iteration 5] ||gradient|| = 8.26675e-06
```

```
assoc <- assoc[!is.na(assoc$p), ]
```

A **Manhattan plot** shows $-\log_{10} p$ along the genome (Figure 4.3):

```
manhattan(assoc, main = "XXX association (mixed model)")
abline(h = -log10(5e-8), col = "#c0432f", lty = 2)
```

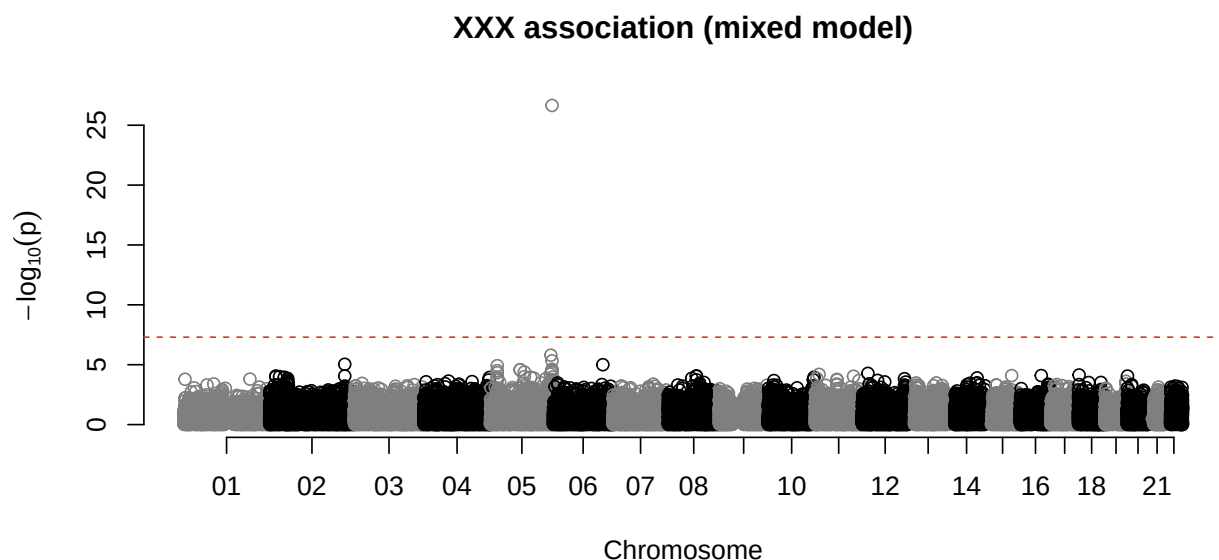


Figure 4.3: Manhattan plot of the mixed-model association to the XXX phenotype. The dashed line is genome-wide significance ($5e-8$).

The scan is well-calibrated because relatedness is modelled (Figure 4.4):

```
qqplot.pvalues(assoc$p)
```

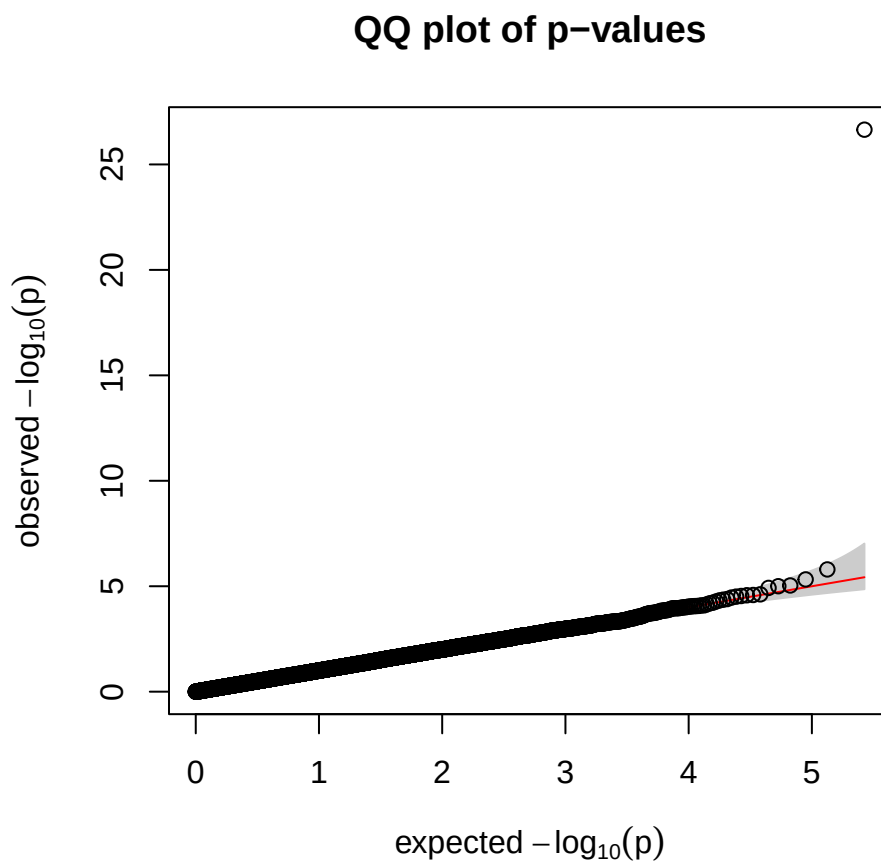


Figure 4.4: QQ plot of the mixed-model association p-values.

The top associated SNPs:

```
head(assoc[order(assoc$p), c("chr", "pos", "id", "A1", "A2", "freqA2", "p")], 10)
```

	chr	pos	id	A1	A2	freqA2	p
86341	05	176775080	rs2731672	NA	NA	0.17837838	2.242433e-27
85954	05	173485857	rs7712944	NA	NA	0.06081081	1.596850e-06
86338	05	176722325	rs6556312	NA	NA	0.19864865	4.748208e-06
29843	02	215372636	rs7591615	NA	NA	0.22043011	9.195526e-06
101801	06	141900444	rs10081087	NA	NA	0.32392473	9.968469e-06
71774	05	19775090	rs1158558	NA	NA	0.17204301	1.200234e-05
86333	05	176659241	rs3088050	NA	NA	0.19407008	2.423393e-05
77468	05	85232149	rs6874335	NA	NA	0.09946237	2.601159e-05
77469	05	85256710	rs4440403	NA	NA	0.08423913	2.672850e-05
86343	05	176792899	rs2287694	NA	NA	0.07297297	2.893642e-05

💡 > Exercise 4.4 — Top hit

1. Locate the SNP with the smallest p-value — its rs id, chromosome, and position. Look it up in dbSNP.
2. Which gene lies at that locus, and what is its role in coagulation/thrombosis?
3. Compare the mixed-model result with a naïve test that ignores kinship (`association.test(geno[ok,], Y[ok], X = Xc[ok,], method = "lm")`). What changes, and why does ignoring relatedness mislead?

❗ Ethics — genomic data privacy

Genotype data are inherently identifying: a few dozen SNPs can re-identify an individual, and a genome reveals information about relatives who never consented. That is why the GAIT set is access-controlled and why real GWAS data live in managed-access repositories (dbGaP, EGA) under data-use agreements. Share summary statistics — never individual genotypes — and follow the study's consent and governance terms.

4.4 Further reading

- Zhou & Stephens (2012), *Genetics* — the polygenic linear mixed model (`lmm.aireml`) implemented here.
- [gaston vignette “linear mixed model”](#) — `association.test(..., method = "lmm")`.
- Soria *et al.*, the GAIT project papers, for the thrombophilia phenotype background.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

4.5 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

5 Computing Homology

Sequence comparison algorithms are a key piece in the family of tools used in bioinformatics. These algorithms look for protein or DNA sequences that resemble a sample sequence you already know. This laboratory introduces the **BLAST** set of tools and multiple alignment.

By the end of this laboratory you will be able to:

- explain sequence homology, orthology/paralogy, and how e-values quantify similarity
- choose the right BLAST flavour (blastp, tblastn, blastx, tblastx, blastn) for a task
- run protein and DNA similarity searches at NCBI and EBI and interpret the results
- read a BLAST alignment (identities, positives, gaps, masked regions)
- apply homology search to a real problem: characterising the SARS-CoV-2 genome

Significant similarity between two or more sequences suggests they share a common ancestor, similar structure, and biological function. This is surprisingly useful: it can reveal the role of a sequence of unknown function through a similar sequence of known function, possibly even from a different organism.

Lab at a glance

Goal: understand sequence homology and run similarity searches and alignments. **Pre-requisites:** [Introduction to R](#); [Searching NCBI & Ensembl](#). **Time:** ~2.5 h. **Packages:** Biostrings, palign, ape (and rentrez/rBLAST for the optional programmatic search). **Deliverable:** a lab report.

5.1 Homology

Two sequences are **homologous** when they share a common ancestor and thus show clear similarities. The concept of homology is central to comparative biology and fundamental to understanding sequence function.

When sequences are homologous, they are also *conserved*. Conservation refers to a high degree of similarity in protein or orthologous¹ DNA sequences and protein structures. A highly conserved region or protein is usually related to an important function: a mutation in a highly conserved region would likely lead to a non-viable organism, or one strongly disfavoured by natural selection.

From a practical point of view, two sequences can be considered homologous if 25% of amino acids are identical (proteins) or 70% of nucleotides are identical (DNA) — though this holds only for sufficiently long sequences (>100 residues). Homology algorithms report the **e-value** (the odds that the similarity is due to chance), the length of the similar segments, and the

¹Homologous sequences are **orthologous** if they were separated by a speciation event: when a species diverges into two, the divergent copies of a single gene are orthologous. They are **paralogous** if separated by a gene duplication event: if a gene is duplicated to occupy two positions in the same genome, the two copies are paralogous.

number of insertions/deletions. A reasonable e-value falls below 10^{-4} ; a useful rule of thumb for interpretation:

- $E < 10^{-100}$: identical sequence (same gene/protein and organism)
- $10^{-100} < E < 10^{-50}$: almost identical sequence
- $10^{-50} < E < 10^{-10}$: closely related sequence
- $10^{-5} < E < 1$: could be a true homologue
- $E > 1$: most likely not related

5.2 Alignment types

Over the past decades, computing sequence similarity has improved dramatically. Where similarity was once found by visual inspection, a query is now compared automatically against databases of millions of sequences.

The most famous algorithm in bioinformatics is **BLAST** — *Basic Local Alignment Search Tool*. Several flavours target different search types:

1. **Nucleotide BLAST** — search a nucleotide database with a nucleotide query (blastn, megablast, discontinuous megablast).
2. **Protein BLAST** — search a protein database with a protein query (blastp, psi-blast, phi-blast).
3. **blastx** — search a protein database with a translated nucleotide query.
4. **tblastn** — search a translated nucleotide database with a protein query.
5. **tblastx** — search a translated nucleotide database with a translated nucleotide query.

These methods are summarised in Figure 5.1.

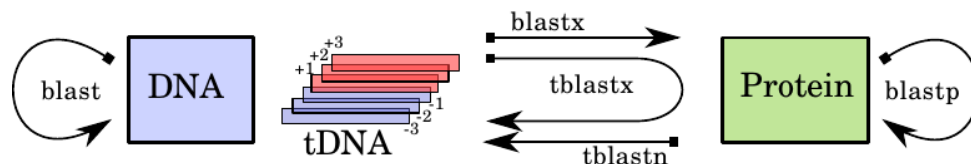


Figure 5.1: Scheme of the main BLAST search methods.

5.3 Working with protein sequences

This section applies when you have a protein sequence and want to find similar proteins, or DNA that could code for similar proteins. Following Figure 5.1, two methods are available: **blastp** (protein query against a protein database) and **tblastn** (protein query against a nucleotide database).

The same BLAST query on different servers can yield different results, because each interface may work with different databases or versions. We use the two main BLAST interfaces (NCBI and EMBnet) and stick to NCBI for the following sections.


BLAST
Basic Local Alignment Search Tool

[Home](#)
[Recent Results](#)
[Saved Strategies](#)
[Help](#)

My NCBI 
[\[Sign In\]](#) [\[Register\]](#)

► **NCBI/ BLAST Home**

BLAST finds regions of similarity between biological sequences. [more...](#)

[Learn more](#) about how to use the new BLAST design

BLAST Assembled Genomes

Choose a species genome to search, or [list all genomic BLAST databases](#).

☐ Human	☐ Oryza sativa	☐ Gallus gallus
☐ Mouse	☐ Bos taurus	☐ Pan troglodytes
☐ Rat	☐ Danio rerio	☐ Microbes
☐ Arabidopsis thaliana	☐ Drosophila melanogaster	☐ Apis mellifera

Basic BLAST

Choose a BLAST program to run.

nucleotide blast	Search a nucleotide database using a nucleotide query <i>Algorithms: blastn, megablast, discontinuous megablast</i>
protein blast	Search protein database using a protein query <i>Algorithms: blastp, psi-blast, phi-blast</i>
blastx	Search protein database using a translated nucleotide query
tblastn	Search translated nucleotide database using a protein query
tblastx	Search translated nucleotide database using a translated nucleotide query

Specialized BLAST

Choose a type of specialized search (or database name in parentheses.)

- ☐ Search [trace archives](#)
- ☐ Find [conserved domains](#) in your sequence (cds)
- ☐ Find sequences with similar [conserved domain architecture](#) (cdart)
- ☐ Search sequences that have [gene expression profiles](#) (GEO)
- ☐ Search [immunoglobulins](#) (IgBLAST)
- ☐ Search for [SNPs](#) (snp)
- ☐ Screen sequence for [vector contamination](#) (vecscreen)
- ☐ [Align](#) two sequences using BLAST (bl2seq)

News

BLAST 2.2.18 now available
The binaries for BLAST 2.2.18 are now available at our [FTP site](#)
2008-04-01 07:50:00

 [More BLAST news...](#)

Tip of the Day

How to Search Custom Databases in Web-Blast Using Entrez Queries

A powerful feature of the BLAST Web interface is the ability to limit BLAST searches to a subset of any database using a standard Entrez query. Skillful use of Entrez queries allows the equivalent of on-the-fly construction of databases of exact composition

 [More tips...](#)

Figure 5.2: NCBI BLAST entry site.

5.3.1 BLASTP at NCBI

We will answer whether there are any proteins similar to the human Low-density lipoprotein receptor precursor, found in the SWISS-PROT protein database.

💡 > Exercise 5.1 — BLASTP at NCBI

1. Find the BLAST entry point at NCBI (<https://blast.ncbi.nlm.nih.gov/>).
2. Look for the amino-acid sequence of the LDL receptor (precursor) for *Homo sapiens* — from the gene search as **LDLR**.
3. Insert the amino-acid sequence or the accession number into the *blastp* suite (it accepts a sequence or a SWISS-PROT accession P0XXXX). Run a BLASTP search with default parameters.
4. Which databases did the algorithm cover?
5. Return to the *blastp* form and select SWISS-PROT in the database selector. List and explain the first 5 hits.
6. Which is the closest protein? What percentage of coincidence do the two share?
7. Open the Distance Tree view and describe the result.



Figure 5.3: NCBI BLAST: partial results in tree form.

The graphic display with bars shows which parts of the query sequence match the found protein sequence. Each bar represents a similar region, and its colour indicates match quality — from black (worst) to red (best). Clicking a bar takes you to the corresponding alignment. The hit list below shows the sequence accession, name, and description; each entry links to its database record.

The alignment section (Figure 5.4) shows your query sequence (top) and the found sequence

```

Score = 134 bits (336), Expect = 3e-30, Method: Compositional matrix adjust.
Identities = 70/209 (33%), Positives = 112/209 (53%), Gaps = 7/209 (3%)

Query   429   VVALDTEVASNRIYWSQLSORMICSTQLDRAHGVSSTDTVISRDIQAPDGLAVDWIHSNI   488
        +V +D +      +YW+D++ R I    L+      + +T+++ + +P+GLA+D I    +
Sbjct  1144  IVGIDYDCRERMVYWTDVAGRTISRAGLELG--AEPETIVNSGLISPEGLAIDHIRRTM  1200

```

Figure 5.4: NCBI BLAST: alignment section excerpt.

(bottom). In the middle row, an amino-acid letter indicates a match, a + indicates similar residues, and a space indicates a mismatch. The numbers on each side give the residue coordinates of the matching region. Regions marked X usually contain many repeats and are masked by BLAST so they don't interfere with the search.

> Exercise 5.2 — Alignments

1. Why would regions with repeats be masked by BLAST?
2. What are Conserved Domains (CD) in proteins? What is their effect if included in BLAST searches?

5.3.2 TBLASTN at NCBI

Here we seek DNA nucleotide sequences starting from a protein sequence, using **tblastn**. This takes a protein query and compares it against an NCBI nucleotide database translated in all six reading frames.

> Exercise 5.3 — TBLASTN

1. Run a *tblastn* query with the same protein sequence you used for BLASTP.
2. Comment on the type of the first results and the differences from the BLASTP results.

5.3.3 BLASTP at EBI

Protein sequencing determines the amino-acid sequence of a protein, helping to understand its structure and function. Imagine you work in a lab on bio-inspired technologies for computer displays, trying to identify a novel insect protein that would enable organic displays with high electroluminescence at very low power. You suspect you have isolated it — Problem Sequence 1 below.

> Exercise 5.4 — BLASTP at EBI

1. Connect to the EBI Sequence Similarity Searching website (<https://www.ebi.ac.uk/jd/ispatcher/sss>).
2. Choose PSI-BLAST (Protein Database Query) and paste Problem Sequence 1.
3. Explore the Results sections: Summary, Tool Output, Visual Output, Functional Predictions.
4. Can you hypothesise a function for this protein sequence? Did you find the protein you were looking for?
5. If you find the protein of Sequence 2 in a patient, what would you suggest?
6. If sequences A and B are both homologous to your query C, can you conclude that A



Figure 5.5: A Beckman-Coulter Porton LF3000G protein sequencer.

is homologous to B?

```
>sp|Problem Sequence 1
```

```
CGGAGKVEYITAPFYPLEDTTAGEQLHKAMKRYALVPGTIAFDAHIEVYQGFQAEYFEMS  
VRLAEAMKRYGLNTNHRIVVCSENSLQFFMPVLGALFIGVAVAPANDIYIERELLSMNI  
SQPTVVFVSEEGWQKILNVQKKLPPIIQKIIIMDSKTDYQGFQSMYTFVTSHLPPGFNEYD  
FVPESFDRDKTIALISNSSGSTGLPKGVALPHRTACVRFSHARDPIFGNQIIPDTAILS  
VPFHGFGFMFTTLGYLICGFRVVLMYRFEEELFLRSLQDYKIQSALLVPTLFSFFAKSTL  
IDKYDLSNLHEIASGGAPISKEVGEAVAKRFHLPGIRQGYGLTETTSAILITPEGDDKPG  
AVGKVVPFFEAKVVDLDTGKTLGVNQRGELCVRGPMIMSLYVNNPEATNALIDKGWLHS  
AVGKVVPFFEAKVVDLDTGKGLGPKGVALPHRTACVRFSHARDPIFGNQAVAKRFHLPLE  
GDIAYWDEDEHFFIVDLLKSLIKYKGYQVAAAALESILLQHPNIFDAGVAGLPDDDAGEL  
PAAVVVLEHGKTMTEKEIV
```

```
>sp|Problem Sequence 2
```

```
MFVFLVLLPLVSSQCVNLTRTQLPPAYTNSFTRGVYYPDKVFRSSVLHSTQDLFLPFFSNVTWFHAIHV  
SGTNGTKRFDNPVLPFNDGVYFASTEKSNIIRGWIFGTTLDSKTQSLILVNNATNVVIKVFCEQFCNDPF  
LGYYHKNKNSWMESEFRVYSSANNCTFEYVSQPFLMDLEGKQGNFKNLREFVFKNIDGYFKIYSKHTPI  
NLVRDLPQGFSALEPLVDLPIGINITRNQFNSAIGRSYLTPGDSSSGWTAGAAAYPIHKELERTFLLKYN  
ENGTITDAVDCALDPLSETKCTLKSFTVEKGIYQTSNFRVQPTESIVRFPNITNLCPFGEVFNATRFASV  
YAWNKRKISNCVADYSVLNSASFSTFKCYGVSPTKLNDLCFTNVYADSFVIRGDEVQRQIAPGQTGKIAD  
YNYKLPPDDFTGCVIAWNSNNEEEEEENYNLYRLFRKSNLKPFERDISTEIYQAGSTPCNGVEGFNCYF  
PLQSYGFQPTNGVGYQPYRVVLSFELLHAPATVCGPKKSTNLVKNKCVNFNFNGLTGTGVLTESNKKFL  
PFQQFGRDIADTTDAVRDPQTLEILDITPCSFGGVSVITPGTNTSNQVAVLYQDVNCTEVPVAIHADQLT  
PTWRVYSTGSNVFQTRAGCLIGAHEVNNSYECDIPIGAGICASYQTQTNSPRRARSVASQSIIAYTMSLG  
AENSVAYSNNIAIPTNFTISVTTEILPVSMTKTSVDCTMYICGDSTECNLLLQYGSFCTQLNRALTGI  
AVEQDKNTQEVFAQVKQIYKTPPIKDFGGFNFSQILPDPSKPSKRSFIEDLLFNKVTLADAGFIKQYGDC  
LGDIAARDLICSNNSNNSNVLPPLLTDEMAIYQTSALLAGTITSGWTFGAGAAALQIPFAMQMAYRFNGIG  
VTQNVLYENQKLIANQFNSAIGKIQDSLSTASASILLQHPNINQNKLNLTLVKQLSSNFGAISSVLNDI  
LSRLDKVEAEVQIDRLITGRQLSLQTYVTQQLIRAAEIRASANLAATKMSECVLGQSKRVDFCGKGYHLM  
SFPQSAPHGVVFLHVTYVPAQEKNFTTAPAICHGDKAHFPREGVFSNGTHWFTQRNFYEPQIITDNT  
FVSGNCDVVIGIVNNTVYDPLQPELDSFKEELDKYFKNHTSPDVLGDISGINASVVNIQKEIDRLNEVA  
KNL NESLIDLQELGKYEQYIKWPWYIWL
```

5.4 Working with DNA sequences

Finding similar DNA sequences is harder than finding similar proteins — the amount of data to search through is considerably larger. For DNA queries we have three main tools: **blastn**, **tblastx**, and **blastx**.

5.4.1 BLASTN at NCBI

BLASTN compares a nucleotide query against a nucleotide database. All current BLAST algorithms use BLAST 2.0, which allows gaps in alignments by default, enabling both local and global alignments — suitable for most applications.

5.4.2 BLASTX at NCBI

blastx takes a nucleotide query, translates it in all six frames, and compares those translations to a protein database — useful for finding proteins encoded by a nucleotide query. Because it combines significance statistics across frames, it is especially useful when the reading frame is unknown or the query contains frame-shift errors. It is often the first analysis performed on a newly determined nucleotide sequence and is used extensively for EST analysis; it is more sensitive than nucleotide BLAST because the comparison happens at the protein level.

💡 > Exercise 5.5 — BLASTX

1. Find the gene for the human HBG1 hemoglobin, gamma A.
2. Run a *blastx* with the gene sequence. Compare the results with the gene structure in Figure 5.6.
3. Explain why *blastx* is more sensitive than nucleotide–nucleotide BLAST.
4. Save the result in a file.

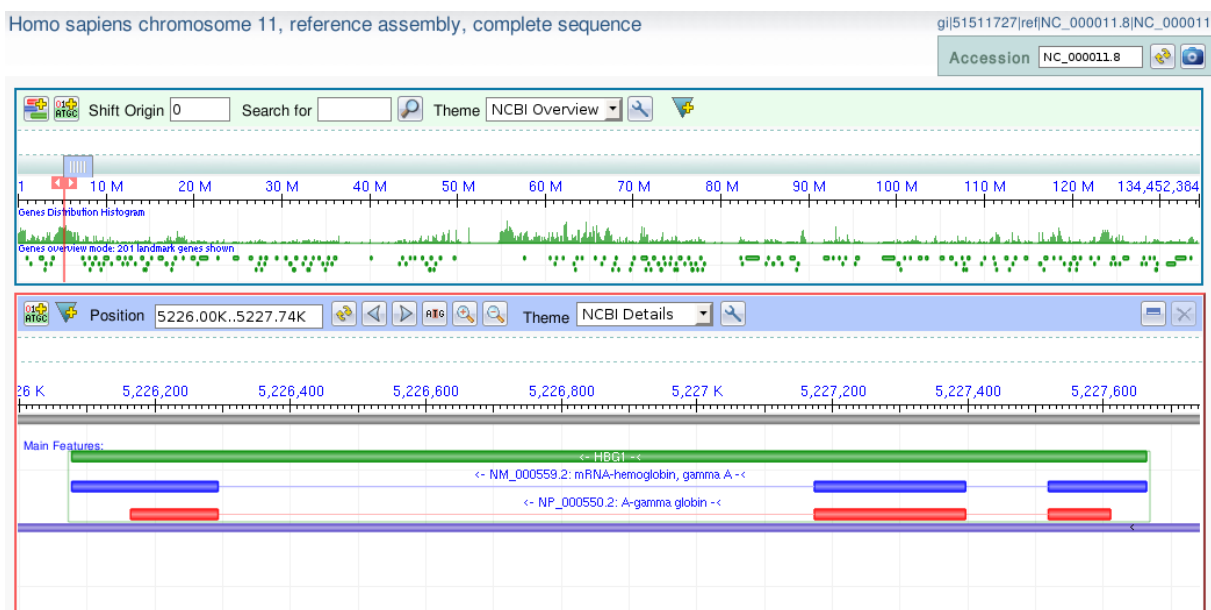


Figure 5.6: Structure of the HBG1 hemoglobin, gamma A gene from NCBI.

5.4.3 TBLASTX at NCBI

tblastx takes a nucleotide query, translates it in all six frames, and compares those to database sequences dynamically translated in all six frames — effectively a more sensitive blastp search without manual translation. It circumvents frame-shifts and ambiguities that may hide open reading frames, which is very useful for identifying potential proteins encoded by single-pass EST reads² and for discovering novel genes.

²An expressed sequence tag (EST) is a short sub-sequence of a transcribed, spliced nucleotide sequence (coding or not).

💡 > Exercise 5.6 — TBLASTX

1. Run a *tblastx* with the human HBG1 hemoglobin, gamma A gene sequence. Observe and discuss the results.
2. What is the main difference from the previous methods?
3. Compare the results with the gene structure.

5.5 SARS-CoV-2

Severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2) is the virus causing COVID-19. The outbreak, which presumably started from a local seafood market, grew to infect 2761 people in China by 26 January 2020 (Zhou et al., 2020). It is the seventh known coronavirus to infect people, after 229E, NL63, OC43, HKU1, MERS-CoV, and the original SARS-CoV (2003).

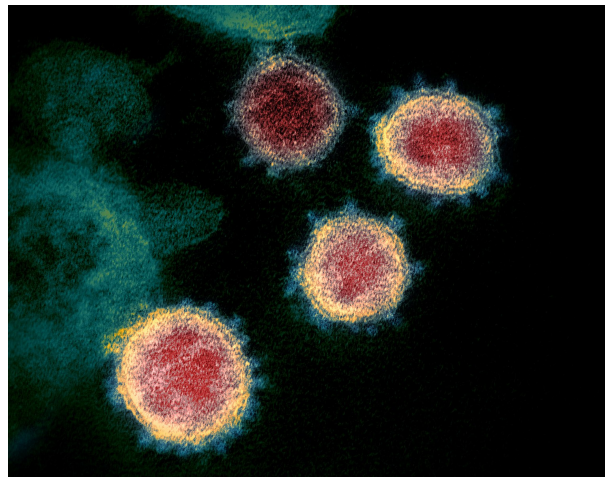


Figure 5.7: Transmission electron microscope image of SARS-CoV-2 (from Wikimedia).

The first known SARS-CoV-2 sequence was published on 2 February 2020 (revised 11 February) (Wu et al., 2020) under GenBank annotation *MN908947.3*, from a single patient — a worker at a local market admitted to the Central Hospital of Wuhan on 26 December 2019 with a severe respiratory syndrome. Metagenomic RNA sequencing of bronchoalveolar lavage fluid identified a new RNA virus of the family Coronaviridae, designated *WH-Human 1*. Phylogenetic analysis of the complete 29,903-nucleotide genome revealed the virus was most closely related (89.1% nucleotide similarity) to a group of SARS-like coronaviruses (genus *Betacoronavirus*, subgenus *Sarbecovirus*) previously found in bats in China.

The genes annotated on MN908947.3 are listed in Table 5.1.

Table 5.1: Genes listed on MN908947.3

#	Accession	Start	Stop	Gene symbol	Strand
1	MN908947.3	266	21555	orf1ab	plus
2	MN908947.3	21563	25384	S	plus
3	MN908947.3	25393	26220	ORF3a	plus
4	MN908947.3	26245	26472	E	plus
5	MN908947.3	26523	27191	M	plus
6	MN908947.3	27202	27387	ORF6	plus

#	Accession	Start	Stop	Gene symbol	Strand
7	MN908947.3	27394	27759	ORF7a	plus
8	MN908947.3	27894	28259	ORF8	plus
9	MN908947.3	28274	29533	N	plus
10	MN908947.3	29558	29674	ORF10	plus

💡 > Exercise 5.7 — SARS-CoV-2

1. Run a homology search on the full SARS-CoV-2 sequence. Explain which virus sequences are homologous to 2020 SARS-CoV-2, and in which regions.
2. Plan and run a strategy for analysing the different elements of the SARS-CoV-2 sequence — its proteins and active domains. Learn as much as you can about this virus in terms of homology to any known sequence.

5.6 Doing it in R

The web interfaces above are how you'll usually explore a single query, but for reproducible work you want the search and alignment **in code**. Bioconductor's **Biostrings** aligns sequences locally, and **ape** builds trees — both run offline. We illustrate the mechanics on a small set of synthetic sequences (one seed and progressively more mutated copies), so the results are reproducible anywhere:

```
library(Biostrings)
library(pwalign)      # pairwiseAlignment()/stringDist() live here since BioC 3.19
set.seed(1)
aa <- strsplit("ACDEFGHIKLMNPQRSTVWY", "")[[1]]
seed <- paste(sample(aa, 80, replace = TRUE), collapse = "")
mutate <- function(s, n) {
  v <- strsplit(s, "")[[1]]; pos <- sample(length(v), n)
  v[pos] <- sample(aa, n, replace = TRUE); paste(v, collapse = "")
}
seqs <- AAStringSet(c(seed = seed, near = mutate(seed, 6),
                      mid = mutate(seed, 18), far = mutate(seed, 34),
                      distant = mutate(seed, 52)))

seqs
```

AAStringSet object of length 5:

	width	seq	names
[1]	80	EHACMQVWALQLHKRFKQFFCL...MTEPISQYHPNSAPGTKHWVSM	seed
[2]	80	HHACMQVWANQLHKRFKQFFCL...MREPIMQYHPNSAPGTKHWVSM	near
[3]	80	RHACMQVWALQLHKRFKQFFCW...MTCPIQYAPNSAPGTKHWVSM	mid
[4]	80	KEKCMQVWLSQIHLLNQKECL...MTEPIIQFHRSTPGTEEWGYM	far
[5]	80	EWYGMQKGQSDTHFVGSIFTCA...MAPMISWYHPNHRPEMSAGVKM	distant

A **pairwise alignment** with BLOSUM62 scores similarity — high for the near copy, low for the distant one, exactly as an e-value would rank them:

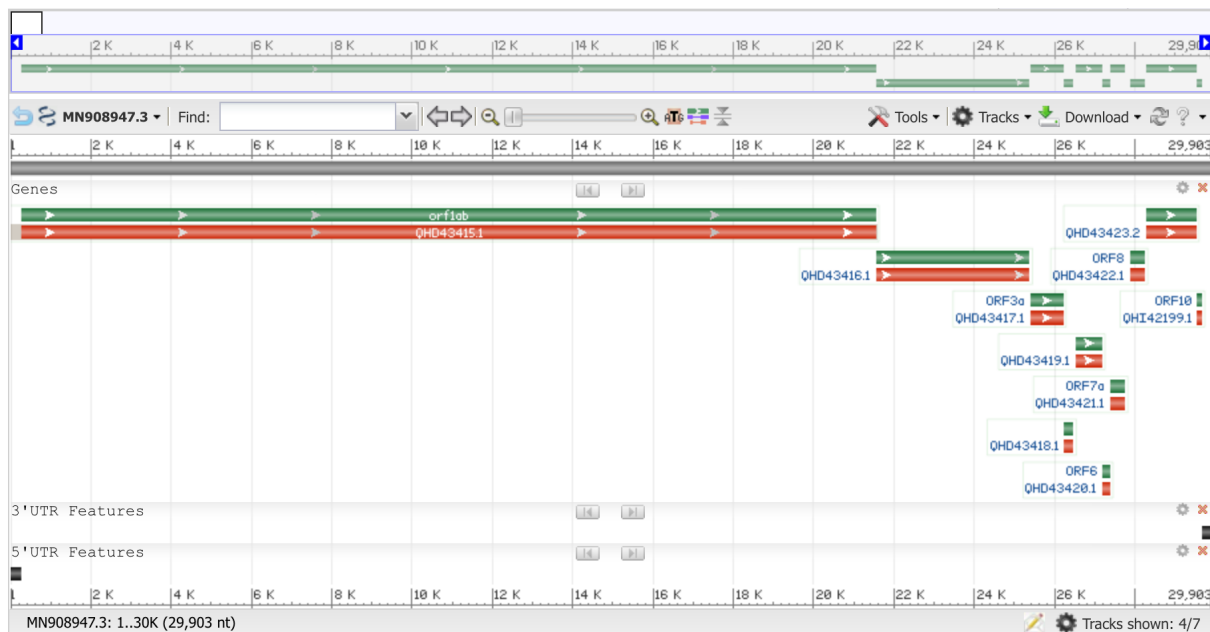


Figure 5.8: SARS-CoV-2 isolate Wuhan-Hu-1, complete genome (GenBank MN908947.3).

```
pa_near <- pairwiseAlignment(seqs[["seed"]], seqs[["near"]],
                             substitutionMatrix = "BLOSUM62",
                             gapOpening = 10, gapExtension = 4)
pa_far  <- pairwiseAlignment(seqs[["seed"]], seqs[["distant"]],
                             substitutionMatrix = "BLOSUM62",
                             gapOpening = 10, gapExtension = 4)
c(near = score(pa_near), distant = score(pa_far))
```

```
near distant
428      112
```

```
pid(pa_near) # percent identity of the close pair
```

```
[1] 92.5
```

From all-vs-all distances we can build and draw a **neighbour-joining tree** (Figure 5.9), the same idea as the BLAST “Distance Tree view”:

```
library(ape)
d <- stringDist(seqs, method = "hamming")
plot(nj(d), type = "unrooted", cex = 0.9, no.margin = TRUE)
```

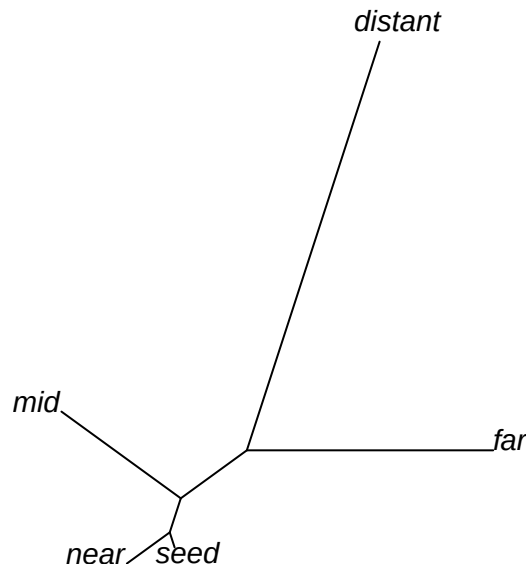


Figure 5.9: Neighbour-joining tree of the illustrative sequences, recovering the mutation order from the seed.

Searching a real database from R

To search NCBI itself, fetch sequences with `rentrez` and run BLAST locally with `rBLAST` (a wrapper around NCBI BLAST+). These need network access and a BLAST install, so they are shown, not run here:

```
library(rentrez); library(rBLAST)
ldlr <- entrez_fetch(db = "protein", id = "NP_000518", rettype = "fasta")
# ... write to a FASTA, then: blast(db = "swissprot") |> predict(query)
```

For multiple alignment and publication-quality trees, add `msa/DECIPHER` and `ggtree`.

5.7 Further reading

- Altschul *et al.* (1990), *J. Mol. Biol.* — the original BLAST paper.
- [NCBI BLAST documentation](#) and the [EBI Job Dispatcher](#) help.
- The `Biostrings` / `palign` and `ape` vignettes for the in-R alignment and tree code.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

5.8 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

Part III

Patterns & expression

6 Pattern Recognition with R: Eigenfaces

In this lab we use **R** with image data to learn the core methods of pattern recognition. Image datasets are a convenient test bench because most results are directly interpretable from visual inspection — the same procedures apply later to more abstract signals such as gas-sensor or omics data.

By the end of this laboratory you will be able to:

- import images into R and build a feature matrix from them
- compute Principal Component Analysis (the Karhunen–Loève decomposition) and interpret **eigenfaces**
- use PCA scores to visualise a high-dimensional dataset
- reconstruct images from a reduced set of components (PCA as a model)
- cluster patterns and features with *k*-means
- build a simple 1-NN classifier with train/validation splitting

The dataset shows the faces of each representative of the Spanish Chamber of Representatives, from the publicly available website congreso.es. A small Python script parsed the site to retrieve images, names, and information for each representative.

Requirements

This lab runs with the `jpeg`, `dplyr`, `class`, `ggrepel`, and `ggplot2` packages and the image dataset (the `db/` folder and `index.csv`), bundled in the repo under `data/`. A handful of figures are additionally shown as interactive `plotly` widgets in the HTML edition — hover them to explore the points — while the PDF keeps those figures static. The figures below are generated by executing the code. Write your own work in a `.R` script (e.g. `myRcode.R`) so you keep a copy of what you typed; run it with `source("myRcode.R")`.

Lab at a glance

Goal: learn PCA, visualisation, clustering and a simple classifier on image data. **Pre-requisites:** [Introduction to R](#). **Time:** ~2.5 h. **Packages:** `jpeg`, `dplyr`, `class`, `ggrepel`, `ggplot2` (+ `plotly` for the interactive figures). **Data:** the `db/` face set + `index.csv`. **Deliverable:** a lab report.

6.1 Dataset description

The whole dataset is one folder: an `index.csv` file plus a `db/` directory holding one `.jpg` per representative, in $86 \times 85 \times 3$ pixels (`db/orig/` stores the originals). In the repo the bundled copy is `data/`. If you decompress the provided zip somewhere else, point the knob below at your copy — every chunk in this lab reads through it, so no `setwd()` is ever needed. List the contents with `dir(dataDir)` and `dir(file.path(dataDir, "db"))`.

```
dataDir <- "data"
```

Import an image with `readJPEG()` from the `jpeg` package:

```
library(jpeg)
I <- readJPEG(file.path(dataDir, "db", "orig", "200p.jpg"))
```

Check the form of `I`:

```
dim(I)
```

```
[1] 94 80 3
```

We can plot an image with a small `iplot()` helper (Figure 6.1):

```
iplot <- function(m) {
  par(mai = c(0, 0, 0, 0), xaxs = "i", yaxs = "i")
  plot.new()
  rasterImage(m, 0, 0, 1, 1)
}
iplot(readJPEG(file.path(dataDir, "db", "99c.jpg")))
```



Figure 6.1: A representative from the dataset, plotted with `iplot()`.

The file `index.csv` describes each file in `db/`. Load it with `read.table()` and keep it tidy with a couple of `dplyr` verbs:

```
library(dplyr)
finfo <- read.table(file.path(dataDir, "index.csv"),
  sep = ";", header = TRUE, strip.white = TRUE) |>
  as_tibble() |>
  mutate(across(c(Gender, Party), as.factor))
head(finfo)
```

```
# A tibble: 6 x 5
```

Num	Name	Surname	Gender	Party
-----	------	---------	--------	-------

	<int>	<chr>	<chr>	<fct>	<fct>
1	321	Manuel	Marín González	H	GS
2	251	Carme	Chacón Piqueras	M	GS
3	132	Jordi	Vilajoana Rovira	H	GC-CiU
4	181	Gabriel	Cisneros Laborda	H	GP
5	154	Ignacio	Gil Lázaro	H	GP
6	59	María Jesús	Sainz García	M	GP

`sep = ";"` sets the field delimiter and `header = TRUE` uses the first line as column names. `strip.white = TRUE` trims the padding spaces around each field, so the labels in the figures below are clean. Since R 4.0, text columns always arrive as **character** — the old “every string becomes a factor silently” default was removed because it caused hard-to-debug surprises; instead, we elevate only the true categories with `mutate(across(...))`. Names and surnames stay character: they are identifiers, not levels. Each row gives an image number (`Num`, matching a file in `db/`)¹ plus the name, surname, gender, and party of the representative. Some quick statistics:

```
summary(finfo)
```

	Num	Name	Surname	Gender
Min.	: 1.00	Length:348	Length:348	H:223
1st Qu.	: 95.75	Class :character	Class :character	M:125
Median	:187.50	Mode :character	Mode :character	
Mean	:187.19			
3rd Qu.	:282.25			
Max.	:370.00			

	Party
GS	:164
GP	:146
GC-CiU	: 10
GER-ERC	: 8
GV	: 7
GIV-IU-ICV	: 5
(Other)	: 8

The breakdown by party and gender is one `dplyr` verb away — `count()` tallies rows per group:

```
finfo |> count(Party, Gender)
```

```
# A tibble: 15 x 3
  Party      Gender      n
  <fct>    <fct> <int>
1 GC-CiU    H         9
2 GC-CiU    M         1
3 GCC       H         3
4 GER-ERC   H         7
```

¹Mid-term substitutions leave gaps in the numbering: `Num` reaches 370 while the roster shows 348 representatives. Below, `X` is built in the row order of `index.csv`, so face *numbers* and matrix *rows* must not be conflated.

5	GER-ERC	M	1
6	GIV-IU-ICV	H	3
7	GIV-IU-ICV	M	2
8	GMx	H	2
9	GMx	M	3
10	GP	H	105
11	GP	M	41
12	GS	H	88
13	GS	M	76
14	GV	H	6
15	GV	M	1

6.2 Creating pattern vectors

For our purposes it is convenient to work with grayscale images, via a small `rgb2grey()` helper (Figure 6.2):

```
rgb2grey <- \(m) (m[, , 1] + m[, , 2] + m[, , 3]) / 3 # channel mean → one intensity pl
Ig <- readJPEG(file.path(dataDir, "db", "200c.jpg")) |> rgb2grey()
iplot(Ig)
```



Figure 6.2: A dataset face converted to grayscale (86×85 px).

The first step is to express an image as a vector of features, converting each pixel into a number capturing how dark it is:

```
vIg <- as.vector(Ig)
data.frame(pixel = seq_along(vIg), intensity = vIg) |>
  ggplot(aes(pixel, intensity)) +
  geom_line(colour = "#0e8d7c") +
  labs(x = "pixel", y = "intensity",
       title = "Vector representation of an image")
```

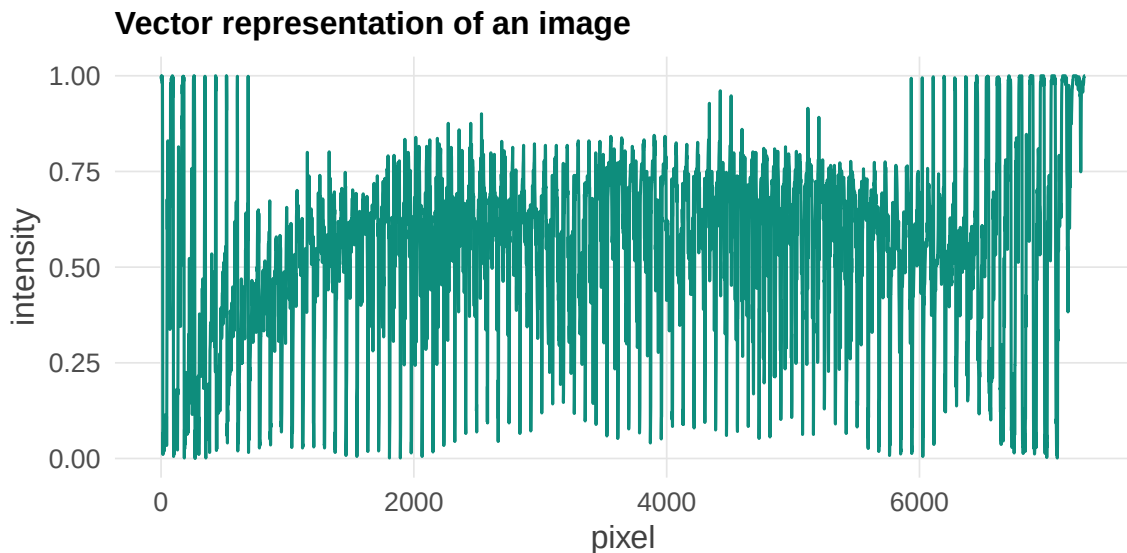


Figure 6.3: Vector representation of a face in gray tones; values normalized to $[0, 1]$.

Now `vIg` is a 7310-dimensional vector (86×85 pixels) containing all the grayscale information of the image (Figure 6.3).

Data is usually placed in a matrix — the **feature matrix** X (Figure 6.4) — where we can access the intensity of pixel 1 of face 200 as `X[200, 1]` (0 is black, 1 is white).

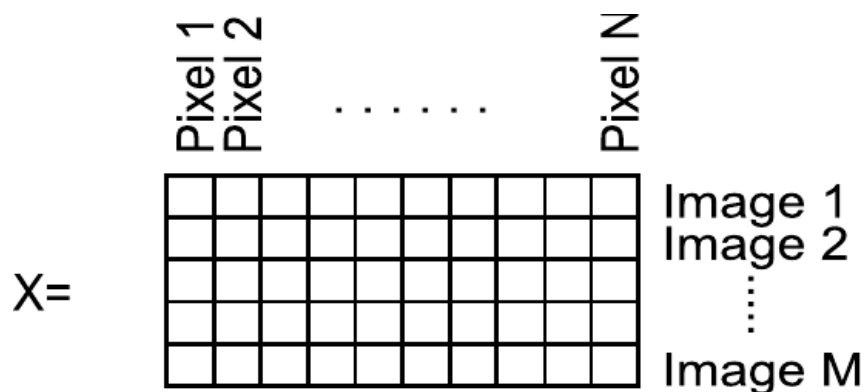


Figure 6.4: The feature matrix.

Using the `Num` field of `finfo`, we build a feature matrix holding all images in one pipeline: take the image numbers, and for each of them read its file, convert to grayscale, flatten to a vector — the `\(f)` lambda does that — then stack the results into a matrix:

```
X <- finfo |>
  pull(Num) |>
  sapply(\(f) file.path(dataDir, "db", paste0(f, "c.jpg")) |>
    readJPEG() |>
    rgb2grey() |>
    as.vector()) |>      # one column per face
  t()                   # one ROW per face
dim(X)
```

```
[1] 348 7310
```

💡 > Exercise 6.1 — Feature matrix

Understand and explain the functionality of the code line above.

Now `X` holds the vectorised images of all members, and `finfo` holds the information on what each row of `X` contains — i.e. `finfo` is **metadata**.

6.3 Karhunen–Loève decomposition (PCA)

Before computing PCA we can inspect the mean image. Reconstruct it into an image matrix with a small `imagematrix()` helper (Figure 6.5):

```
imagematrix <- function(m, nrow, ncol) {  
  dim(m) <- c(nrow, ncol)  
  m  
}  
mX <- colMeans(X)  
mean_img <- mX |> imagematrix(nrow = nrow(Ig), ncol = ncol(Ig))  
iplot(mean_img)
```



Figure 6.5: Mean chamber representative.

PCA finds the principal orthogonal vectors that explain the main variance of the dataset. Each principal vector lives in feature space, so it is itself an image (an **eigenface**):

```
pcamod <- prcomp(X, center = FALSE)
```

How much of the variation does each component capture? The **scree plot** (Figure 6.6) shows the variance explained — bars per component, amber line accumulating — a handful of eigenfaces already account for most of the structure, which is why reconstruction from few components works.

```
scree <- pcamod$sdev^2 |>  
  (\(v) data.frame(PC = seq_along(v), variance = v / sum(v)))() |> # one row per PC  
  slice_head(n = 20) |>  
  mutate(cumulative = cumsum(variance))  
scree |>
```

```
ggplot(aes(PC, variance)) +
  geom_col(fill = "#0e8d7c") +
  geom_line(aes(PC, cumulative), colour = "#b0741a") +
  geom_point(aes(PC, cumulative), colour = "#b0741a", size = 1.6) +
  scale_y_continuous(labels = scales::percent) +
  labs(x = "principal component", y = "variance explained")
```

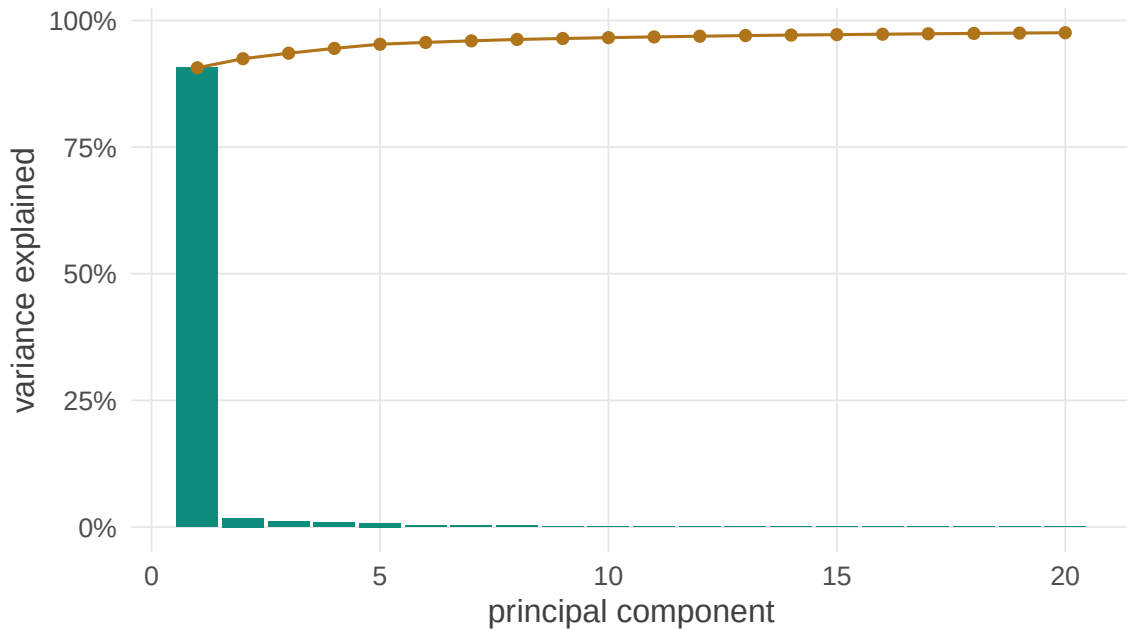


Figure 6.6: Variance explained by the first 20 principal components (bars) and their cumulative share (line).

Plot the first three principal images by translating the rotation vectors into image matrices (Figure 6.7).² Each principal vector is itself an image of dimension 7310 pixels.

```
normalize <- \(x, to = c(0, 1)) {
  xrange <- range(x)
  to[1] + (x - xrange[1]) / diff(xrange) * diff(to)
}
as_face <- \(v) imagematrix(normalize(v), nrow = nrow(Ig), ncol = ncol(Ig))
plot_faces <- \(imgs) {
  par(mfrow = c(1, length(imgs)), mai = c(0, 0, 0, 0))
  invisible(lapply(imgs, iplot))
}
signs <- c(-1, -1, 1) # PC1-PC2 look like negatives without the flip
plot_faces(1:3 |> lapply(\(i) as_face(signs[i] * pcamod$rotation[, i])))
```

²If an image looks like a negative, plot it with a leading minus sign (before `pcamod$rotation`).



Figure 6.7: First three principal components (eigenfaces) on non-centered data.

💡 > Exercise 6.2 — Eigenfaces

1. Give an interpretation of the first principal component (the first eigenface, or *eigen-representative*).
2. Repeat the PCA centering the model. Plot the first three components again, explain the effect of centering, and discuss its importance.
3. For both cases (centered and non-centered), plot the amount of variance captured by each component with `plot(pcamod)`. What are the differences? Can you explain them?

Centering the data before PCA (Figure 6.8) changes the components markedly:

```
pcamodc <- prcomp(X, center = TRUE)
plot_faces(1:3 |> lapply(\(i) as_face(pcamodc$rotation[, i])))
```



Figure 6.8: First three principal components on centered data.

6.3.1 PCA for dataset visualisation

PCA provides a decomposition ordered by the major covariance directions, so it doubles as a dimensionality-reduction algorithm. Each sample has a projection (its **scores**) onto the subspace spanned by a set of principal components. We can plot the scores on the first two dimensions to get an idea of a high- dimensional ($D = 7310$) distribution through a 2D representation. We use the **centered** model `pcamodc`, whose components describe variation *around* the mean face

— with the non-centered model the first component would largely encode the mean itself and dominate the plot.

A tidy score plot needs three ingredients: the sample scores, the metadata to colour by, and the variance explained by each component (so the reader knows how much of the cloud each axis actually shows). Two tiny helpers collect them: `pca_scores()` returns one tidy row per face, binding any metadata columns to the scores; `pca_var()` formats the axis titles with the explained variance; and `gg_interactive()` turns a `ggplot` into an interactive `plotly` widget wherever a browser can display it (the HTML edition) and keeps the plain `ggplot` anywhere JavaScript cannot run — the PDF:

```
pca_scores <- \(pca, info = NULL, axes = 1:2) {
  scores <- pca$x[, axes, drop = FALSE] |>
    as.data.frame() |>
    setNames(paste0("PC", axes))
  if (is.null(info)) scores else bind_cols(scores, info)
}
pca_var <- \(pca, axes = 1:2) {
  var <- (pca$sdev^2 / sum(pca$sdev^2))[axes]
  sprintf("%s (%.1f%%)", paste0("PC", axes), 100 * var)
}
gg_interactive <- \(plot, ...) {
  if (knitr::is_html_output())
    plotly::ggplotly(plot, ...)
  else
    plot
}
```

With the scores and metadata in one data frame, the party plot is a plain `ggplot()` (Figure 6.9): every face is one point, and each party is tagged once at its mean position, with `ggrepel` keeping the tags from colliding. The `text` aesthetic carries each representative's name on the points, so hovering identifies it in the interactive HTML edition:

```
party_scores <- pca_scores(pcamodc, finfo |> select(Name, Surname, Party))
party_centres <- party_scores |>
  group_by(Party) |>
  summarise(PC1 = mean(PC1), PC2 = mean(PC2), .groups = "drop")
party_plot <- party_scores |>
  ggplot(aes(PC1, PC2, colour = Party)) +
  geom_hline(yintercept = 0, colour = "grey80", linewidth = 0.4) +
  geom_vline(xintercept = 0, colour = "grey80", linewidth = 0.4) +
  geom_point(aes(text = paste0(Name, " ", Surname, " (" , Party, ")")),
    size = 1.1, alpha = 0.5, show.legend = FALSE) +
  ggrepel::geom_label_repel(data = party_centres, aes(label = Party),
    size = 2.6, seed = 42, show.legend = FALSE) +
  labs(x = pca_var(pcamodc)[1], y = pca_var(pcamodc)[2]) +
  coord_equal()
gg_interactive(party_plot, tooltip = "text")
```

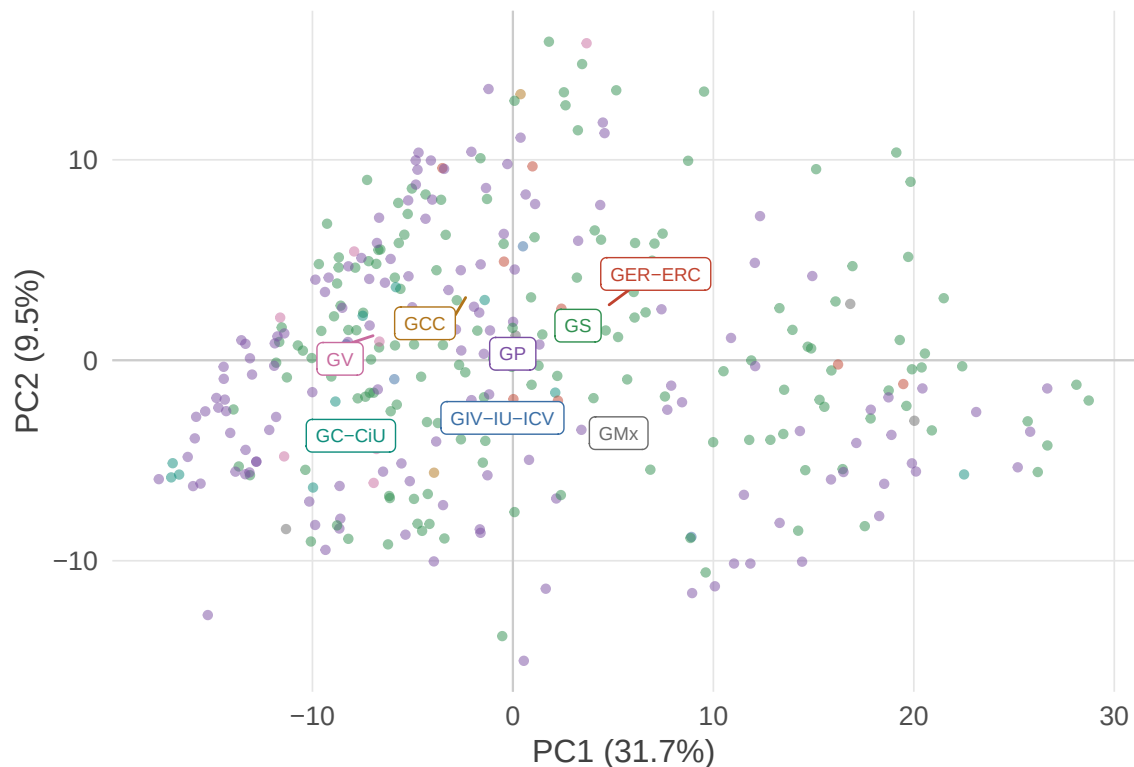


Figure 6.9: PCA score plot of political affiliation: one point per representative (hover to identify), one tag per party at the party mean. Axis titles give the variance explained by each component.

💡 > Exercise 6.3 — Score plots (party)

1. From Figure 6.9, what total amount of variance is captured by the first two principal components? (Check the axis titles.)
2. Can you visually see any relationship between political party and the features?
3. What classification performance would you expect from these score plots?

Repeating the projection while exposing gender instead (Figure 6.10):

```
gender_scores <- pca_scores(pcamodc, finfo |> select(Name, Surname, Gender))
gender_plot <- gender_scores |>
  ggplot(aes(PC1, PC2, colour = Gender,
             text = paste0(Name, " ", Surname, " - ", Gender))) +
  geom_hline(yintercept = 0, colour = "grey80", linewidth = 0.4) +
  geom_vline(xintercept = 0, colour = "grey80", linewidth = 0.4) +
  geom_point(size = 1.3, alpha = 0.6) +
  labs(x = pca_var(pcamodc)[1], y = pca_var(pcamodc)[2]) +
  coord_equal()
gg_interactive(gender_plot, tooltip = "text")
```

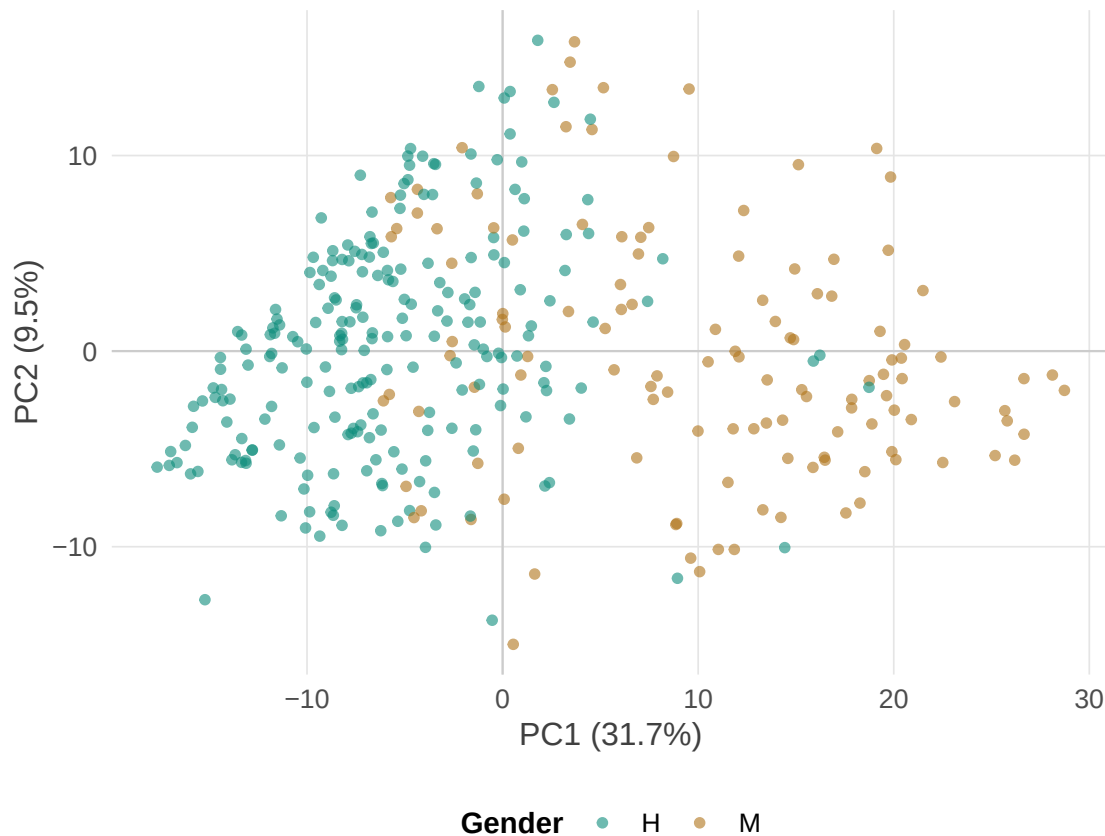


Figure 6.10: PCA score plot labelling gender; axis titles give the variance explained by each component. Hover in the interactive HTML edition to identify each representative.

A number backs the eye up — correlating each score vector with the gender indicator shows the signal lives on PC1 alone (magnitude about 0.7 — the sign is a rotation convention), with PC2 carrying none of it. That is exactly the relationship the last question of Exercise 6.4 asks you to interpret:

```
cor(select(gender_scores, PC1, PC2), gender_scores$Gender == "H")
```

```
[,1]
PC1 -0.71860268
PC2  0.03203211
```

💡 > Exercise 6.4 — Score plots (gender)

1. Can you visually see any relationship between gender and the features?
2. What performance would you expect from these score plots?
3. Why is this response different from Figure 6.9?
4. Relate this to the interpretation of the eigenvectors: which eigenvector is most related to gender?

6.3.2 PCA for dataset modelling

PCA is a way of building a *model* of a dataset. We can check the effect of keeping a limited set of components through a reconstruction experiment: a reconstruction keeps the projection of a face

on the first `comps` eigenfaces (the scores), multiplies by the rotation and adds the mean back:

$$\hat{X} = X_{\text{scores}} R^T + \bar{X}$$

Each column of Figure 6.11 is one representative (its number in `index.csv`); each row keeps `comps` components, from the full model of 330 at the top down to seven at the bottom.

```
# One tidy pixel table (row, col, value) for an image matrix, ggplot-ready.
image_long <- \(m) {
  data.frame(row = as.vector(row(m)),
             col = as.vector(col(m)),
             value = as.vector(m))
}

theme_image <- theme(                                # blank, square canvas for pixel tiles
  axis.text = element_blank(),
  axis.ticks = element_blank(),
  axis.title = element_blank(),
  panel.grid = element_blank()
)

# Reconstruct face `id` (a row of X) keeping only the first `comps` components.
reconstruct <- \(id, comps) {
  kept <- seq_len(comps)
  rx <- pcamodc$x[id, kept, drop = FALSE] %*%
    t(pcamodc$rotation[, kept, drop = FALSE])
  (rx[1, ] + pcamodc$center) |>                      # add the mean face back
    imagematrix(nrow = nrow(Ig), ncol = ncol(Ig))    # reshape to an 86x85 matrix
}

# Pixel table for a set of faces at once: each row of the matrix is reshaped,
# normalized and numbered (`face` = matrix row); optional per-face metadata
# columns (one row per face) are bound on.
faces_long <- \(m, meta = NULL) {
  px <- seq_len(nrow(m)) |>
    lapply(\(face)
      imagematrix(m[face, ], nrow = nrow(Ig), ncol = ncol(Ig)) |>
        normalize() |>
        image_long() |>
        mutate(face = face)) |>
    bind_rows()
  if (is.null(meta))
    px
  else
    bind_cols(px, meta |> slice(rep(seq_len(nrow(meta)), each = nrow(Ig) * ncol(Ig))))
}

idips <- c(10, 254, 94, 114, 229, 233, 251, 120)    # representatives to show
infactors <- c(330, 200, 100, 50, 25, 7)           # components kept
panels <- expand_grid(id = idips, comps = infactors) # one row per panel
recon_m <- t(vapply(seq_len(nrow(panels)),          # faces x 7310 matrix
  \(i) reconstruct(panels$id[i], panels$comps[i]),
  numeric(ncol(X))))
recon_long <- recon_m |>
```

```

faces_long(panels) |>
mutate(face = factor(finfo$Num[id], levels = finfo$Num[idips]),
       comps = factor(comps, levels = infactors)) |>
select(-id)
recon_long |>
ggplot(aes(col, row, fill = value)) +
geom_raster() +
facet_grid(rows = vars(comps), cols = vars(face)) +
scale_y_reverse(expand = expansion(0)) +
scale_x_continuous(expand = expansion(0)) +
scale_fill_gradient(low = "black", high = "white", guide = "none") +
coord_equal() +
theme_image

```



Figure 6.11: Reconstruction of eight representatives keeping `comps` principal components: full model (330) down to 7. Labels are the file numbers in `index.csv`.

💡 > Exercise 6.5 — Reconstruction

1. Visually estimate how many principal components are needed to reconstruct gender information.
2. How many are needed to recognise whether an individual wears glasses?

Note that here we included the target images in the PCA training set. Testing generalisation

fairly is one small step away: refit the PCA *without* four faces and reconstruct those unseen ones from it (Figure 6.12). The model can only speak the language of the faces it learned — glasses and fine detail fade much earlier than in the in-sample grid above:

```
set.seed(7)
unseen <- sample(nrow(X), 4) # hold-out faces
pca_hold <- prcomp(X[-unseen, ], center = TRUE) # the model, trained without them
reconstruct_unseen <- \(id, comps) {
  kept <- seq_len(comps)
  rx <- (X[id, ] - pca_hold$center) %*% # scores of a face it never saw
    pca_hold$rotation[, kept] %*% t(pca_hold$rotation[, kept])
  (rx[1, ] + pca_hold$center) |>
    imagematrix(nrow = nrow(Ig), ncol = ncol(Ig))
}
p_h <- expand.grid(id = unseen, comps = c(100, 25, 7))
hold_m <- t(vapply(seq_len(nrow(p_h)),
  \(i) reconstruct_unseen(p_h$id[i], p_h$comps[i]),
  numeric(ncol(X))))
hold_m |>
  faces_long(p_h) |>
  mutate(face = factor(finfo$Num[id]),
    comps = factor(comps, levels = c(100, 25, 7))) |>
  select(-id) |>
  ggplot(aes(col, row, fill = value)) +
  geom_raster() +
  facet_grid(rows = vars(comps), cols = vars(face)) +
  scale_y_reverse(expand = expansion(0)) +
  scale_x_continuous(expand = expansion(0)) +
  scale_fill_gradient(low = "black", high = "white", guide = "none") +
  coord_equal() +
  theme_image
```



Figure 6.12: Reconstruction from a PCA that never saw the faces shown (fit on the other 344), keeping 100 / 25 / 7 components. Labels are the file numbers in `index.csv`.

We can make the trade-off quantitative: reconstruction error falls as we keep more components, with diminishing returns once the leading eigenfaces are in (Figure 6.13):

```
infac <- c(2, 5, 10, 25, 50, 100, 200, 330)
recon_mse <- \(comps) {
  kept <- seq_len(comps)
  Xhat <- sweep(pcamodc$x[, kept] %*% t(pcamodc$rotation[, kept]),
               2, pcamodc$center, "+") |> as.matrix()
  mean((X - Xhat)^2)
}
recon <- data.frame(components = infac, mse = vapply(infac, recon_mse, numeric(1)))
recon |>
  ggplot(aes(components, mse)) +
  geom_line(colour = "#0e8d7c") +
  geom_point(colour = "#0e8d7c") +
  labs(x = "components retained", y = "mean squared error")
```

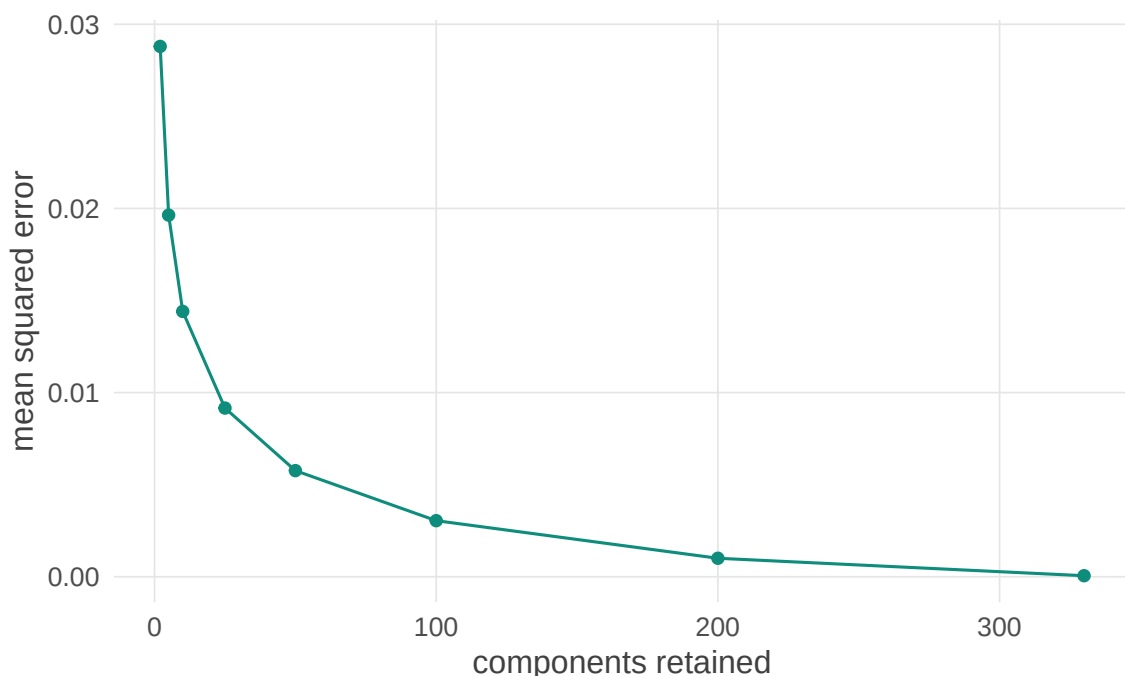


Figure 6.13: Mean squared reconstruction error against the number of principal components retained.

6.4 Pattern clustering

We can cluster faces with *k*-means (low Euclidean distance = similar). First project to a low-dimensional subspace with PCA, then cluster — and keep the tidy data with the metadata attached, since we will reuse it below:

```
set.seed(2025)
all15 <- pca_scores(pcamodc, finfo, axes = 1:5) # first 5 PCs per face + metadata
X5 <- all15 |> select(starts_with("PC")) |> as.matrix()
c1 <- kmeans(X5, centers = 15)
```

```
cl$size
```

```
[1] 22 16 20 22 37 19 25 27 39 33 23 24 15 14 12
```

To inspect a cluster, plot the images assigned to it. The largest cluster holds 39 faces; a dozen of them are shown in Figure 6.14 (labels are the file numbers in `index.csv`):

```
biggest <- which.max(cl$size)
members <- which(cl$cluster == biggest) |> head(12)
member_px <- X[members, , drop = FALSE] |>
  faces_long(data.frame(num = finfo$Num[members])) |>
  mutate(face = factor(num, levels = finfo$Num[members])) |>
  select(-num)
member_px |>
  ggplot(aes(col, row, fill = value)) +
  geom_raster() +
  facet_wrap(vars(face), ncol = 6) +
  scale_y_reverse(expand = expansion(0)) +
  scale_x_continuous(expand = expansion(0)) +
  scale_fill_gradient(low = "black", high = "white", guide = "none") +
  coord_equal() +
  theme_image
```

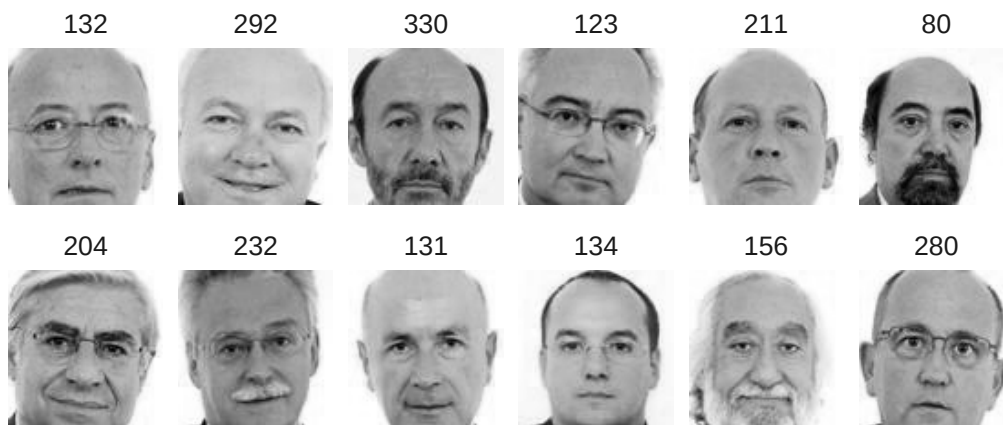


Figure 6.14: Twelve members of the largest k-means cluster, labelled by their file numbers in `index.csv`.

💡 > Exercise 6.6 — Clustering

Observe Figure 6.14 — is the clustering truly following face features? How would you adjust the number of clusters?

6.5 Feature clustering

We can also cluster *pixels* instead of faces — clustering **features** rather than patterns — which hints at which features behave similarly across samples. This is *k*-means on the transposed

matrix; with $t(X)$ the observations become the 7310 pixels. The assigned zone number is drawn as a gray level on the image (Figure 6.15):

```
set.seed(2025)
fcl <- kmeans(t(X), centers = 15)
zone_map <- expand.grid(row = 1:nrow(Ig), col = 1:ncol(Ig)) |>
  mutate(zone = fcl$cluster)
zone_plot <- zone_map |>
  ggplot(aes(col, row, fill = zone)) +
  geom_raster() +
  scale_y_reverse(expand = expansion(0)) +
  scale_x_continuous(expand = expansion(0)) +
  scale_fill_gradient(low = "white", high = "grey25", guide = "none") +
  theme_image +
  coord_equal()
gg_interactive(zone_plot, tooltip = c("x", "y", "fill"))
```



Figure 6.15: Clustering features (pixels) with k-means; the gray level encodes the assigned zone (hover in the HTML edition to read pixel coordinates).

💡 > Exercise 6.7 — Feature clustering

Change the number of clustered features (zones) and describe the effect of increasing it.

We can repeat the analysis focusing only on face features by building a mask and applying it to the dataset (Figure 6.16):

```
# Mask: keep only the central rows and columns of the image.
rows <- 20:80
cols <- 20:70
mask <- matrix(0, nrow = nrow(Ig), ncol = ncol(Ig))
mask[rows, cols] <- 1
nxc <- X[, which(as.logical(mask))] # keep only the masked features
set.seed(2025)
ccl <- kmeans(t(nxc), centers = 10)
zone_focus <- expand.grid(row = rows, col = cols) |> # one row per pixel inside the mask
```

```
mutate(zone = ccl$cluster)
zone_focus_plot <- zone_focus |>
  ggplot(aes(col, row, fill = zone)) +
  geom_raster() +
  scale_y_reverse(expand = expansion(0)) +
  scale_x_continuous(expand = expansion(0)) +
  scale_fill_gradient(low = "white", high = "grey25", guide = "none") +
  theme_image +
  coord_equal()
gg_interactive(zone_focus_plot, tooltip = c("x", "y", "fill"))
```



Figure 6.16: Clustering features with a narrow feature set (a central face mask).

💡 > Exercise 6.8 — Focused feature clustering

Can you explain the new results?

6.6 Pattern recognition

The goal here is to discriminate gender from the image alone, using a 1-NN classifier with a two-fold validation scheme. Split the dataset 50/50 into training and validation, destroying any ordering. `dplyr` makes the split readable — add a `set` column, then `filter()` — and two details make it *sounder*: sampling **within** each gender keeps the H/M balance of both halves exactly the same (a stratified split; a purely random draw only balances it in expectation), and `n() %/% 2` tolerates an odd-sized subset:

```
set.seed(1234)
all15 <- all15 |>
  group_by(Gender) |>
  mutate(set = sample(rep(c("train", "validation"),
                          c(n() %/% 2, n() - n() %/% 2)))) |>
  ungroup()
table(all15$set, all15$Gender) # balance check
```

	H	M
train	111	62
validation	112	63

```
train <- all5 |> filter(set == "train")
val <- all5 |> filter(set == "validation")
Xtr <- train |> select(starts_with("PC")) |> as.matrix()
Xva <- val |> select(starts_with("PC")) |> as.matrix()
```

Predict the validation classes with `knn()` from the `class` package:

```
library(class)
pred <- knn(Xtr, Xva, train |> pull(Gender), k = 1)
```

Show performance with `table()`:

```
table(Predicted = pred, Observed = val$Gender)
```

	Observed	
Predicted	H	M
H	103	16
M	9	47

```
mean(pred == val$Gender)
```

```
[1] 0.8571429
```

The same result as a heat tile, which is worth its ink once class sets grow:

```
data.frame(table(Predicted = pred, Observed = val$Gender)) |>
  ggplot(aes(Observed, Predicted, fill = Freq)) +
  geom_tile() +
  geom_text(aes(label = Freq), size = 3.2) +
  scale_fill_gradient(low = "white", high = "#0e8d7c", guide = "none") +
  theme(panel.grid = element_blank()) +
  coord_equal()
```

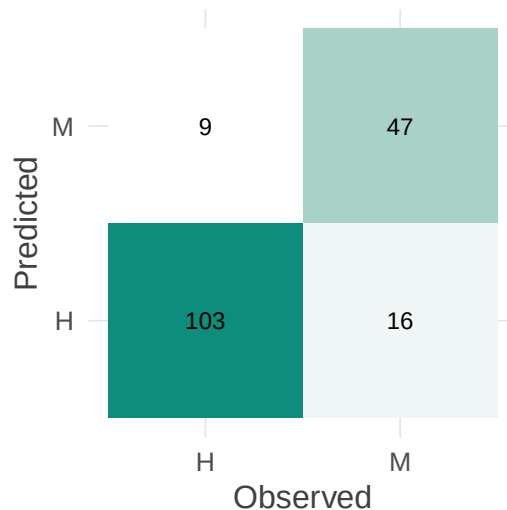


Figure 6.17: Confusion matrix of the 1-NN gender classifier on the validation half (counts).

In this run we correctly identify about 85% of genders, just from the image.

💡 > Exercise 6.9 — Recognition

1. Can you improve this result?
2. What is the ratio if you try to guess political affiliation instead?
3. Repeat the calculation with the reduced feature set from Figure 6.16.
4. Repeat the classification using the whole chamber as the training set and pictures of your classmates as the validation set. Discuss how the results compare.

A single 50/50 split is noisy. **Cross-validation** gives a more stable estimate and lets us tune k : we average accuracy over 5 folds for a range of k (Figure 6.18). The tidy way is to build the grid of (k, fold) combinations, run one classifier per row, and `summarise()` per k :

```
set.seed(42)
ks <- c(1, 3, 5, 7, 11, 15, 21)
fold_of <- sample(rep(1:5, length.out = nrow(all15))) # 5 random folds
fold_acc <- \(k, f) {
  train <- fold_of != f
  truth <- all15 |> pull(Gender)
  pred <- knn(X5[train, ], X5[!train, ], truth[train], k = k)
  mean(pred == truth[!train])
}
cv <- expand_grid(k = ks, fold = 1:5) |>
  rowwise() |>
  mutate(accuracy = fold_acc(k, fold)) |>
  ungroup() |>
  group_by(k) |>
  summarise(accuracy = mean(accuracy), .groups = "drop")
cv_plot <- cv |>
  ggplot(aes(k, accuracy)) +
  geom_line(colour = "#0e8d7c") +
  geom_point(colour = "#0e8d7c") +
  scale_y_continuous(labels = scales::percent) +
```

```
labs(x = "k (neighbours)", y = "CV accuracy")
gg_interactive(cv_plot)
```

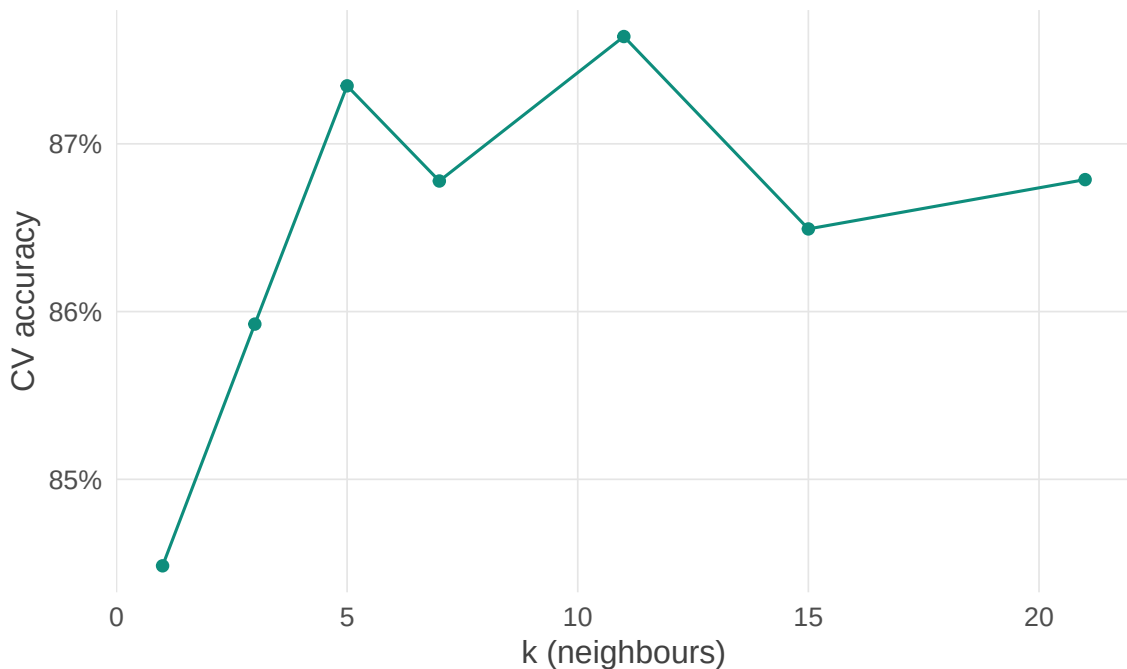


Figure 6.18: 5-fold cross-validated gender-classification accuracy as a function of the k-NN neighbourhood size.

! Ethics — inferring attributes from faces

This exercise trains a classifier to predict **gender** from face images, which is a useful teaching example of supervised learning — but automated inference of personal attributes (gender, ethnicity, sexual orientation, “trustworthiness”) from faces is scientifically fraught and ethically hazardous. Gender here is a coarse binary label from a historical roster, not a ground truth about identity; such systems encode the biases of their training data, misclassify people outside the labelled categories, and have been used to justify surveillance and discrimination. The failure is not uniformly distributed either: commercial gender classifiers err far more on darker-skinned women than on lighter-skinned men (Buolamwini & Gebru, 2018). Treat the result as a lesson in *what a linear model can pick up from pixels*, and discuss where deploying such a classifier would be inappropriate.

6.7 Further reading

- Turk & Pentland (1991) — the article that introduced eigenfaces for recognition (Turk & Pentland, 1991).
- [The prcomp docs](#) for PCA in base R; [the class docs](#) for `knn/knn.cv`; [the plotly docs](#) for the interactive figures.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

6.8 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

7 Differential Expression

In this lab we use a differential-expression dataset available within R. The **ALL** dataset contains a subset of 79 samples from patients with B-cell acute lymphoblastic leukemia. Acute lymphoblastic leukemia (ALL) is a cancer that starts from early white blood cells (lymphocytes) in the bone marrow. The dataset was created with the HG-U95Av2 Affymetrix GeneChip array by Chiaretti et al. (2004). Our interest is to compare samples carrying the **BCR/ABL** fusion gene — which results from a translocation between chromosomes 9 and 22 — with normal (**NEG**) samples.

By the end of this laboratory you will be able to:

- load and subset an **ExpressionSet** and access its expression matrix and phenotype data
- filter probesets by variability and fit a linear model with **limma** to rank differentially expressed genes
- visualise the data with PCA score plots
- cluster samples and genes (**HOPACH**, *k*-means) and draw a heatmap
- build and evaluate a *k*-NN classifier with leave-one-out validation and plot its decision boundary

Requirements

This lab runs with the Bioconductor packages **ALL**, **genefilter**, **limma**, **gplots**, and the CRAN packages **pIs** and **class** — all executed here. The **HOPACH** clustering section additionally needs the **hopach** package; where it is not installed those three chunks are shown but not run. Install any missing packages with `BiocManager::install("<packageName>")`.

Lab at a glance

Goal: find and interpret differentially expressed genes between two groups. **Prerequisites:** [Introduction to R](#); [Eigenfaces / PCA](#). **Time:** ~2.5 h. **Packages:** **ALL**, **genefilter**, **limma**, **ggplot2**, **pheatmap**, **DESeq2** (RNA-seq track). **Data:** **ALL** (bundled). **Deliverable:** a lab report.

7.1 Data description

Load the **ALL** data:

```
library("ALL")
data(ALL)
dim(ALL)
```

Features	Samples
12625	128

```
experimentData(ALL)
```

Experiment data

Experimenter name: Chiaretti et al.

Laboratory: Department of Medical Oncology, Dana-Farber Cancer Institute, Department of Me

Contact information:

Title: Gene expression profile of adult T-cell acute lymphocytic leukemia identifies disti

URL:

PMIDs: 14684422 16243790

Abstract: A 187 word abstract is available. Use 'abstract' method.

💡 > Exercise 7.1 — Objects

What has been loaded into the workspace? What kind of objects are they?

`pData()` accesses the phenotypic data and metadata:

```
pheno <- pData(ALL)
```

💡 > Exercise 7.2 — Phenotypes

What are the names of the phenotype variables? Explain each one. (Hint: `varLabels()` and `varMetadata().`)

The phenotype of interest is `mol.biol`. Build a subset containing only the BCR/ABL and NEG samples, restricted to B-cell lymphocytes:

```
subset <- intersect(grep("^B", as.character(pheno$BT)),  
                    which(pheno$mol %in% c("BCR/ABL", "NEG")))  
esub <- ALL[, subset]  
dim(esub)
```

Features	Samples
12625	79

💡 > Exercise 7.3 — Selection

Explain how the first line of the code above builds the selection.

`exprs()` extracts the expression matrix. We visualise the first 300 probes across all samples with `image()`:

```
eS <- exprs(esub)  
image(eS[1:300, ], ylab = "samples", xlab = "expr levels",  
      axes = FALSE, col = cm.colors(20))
```

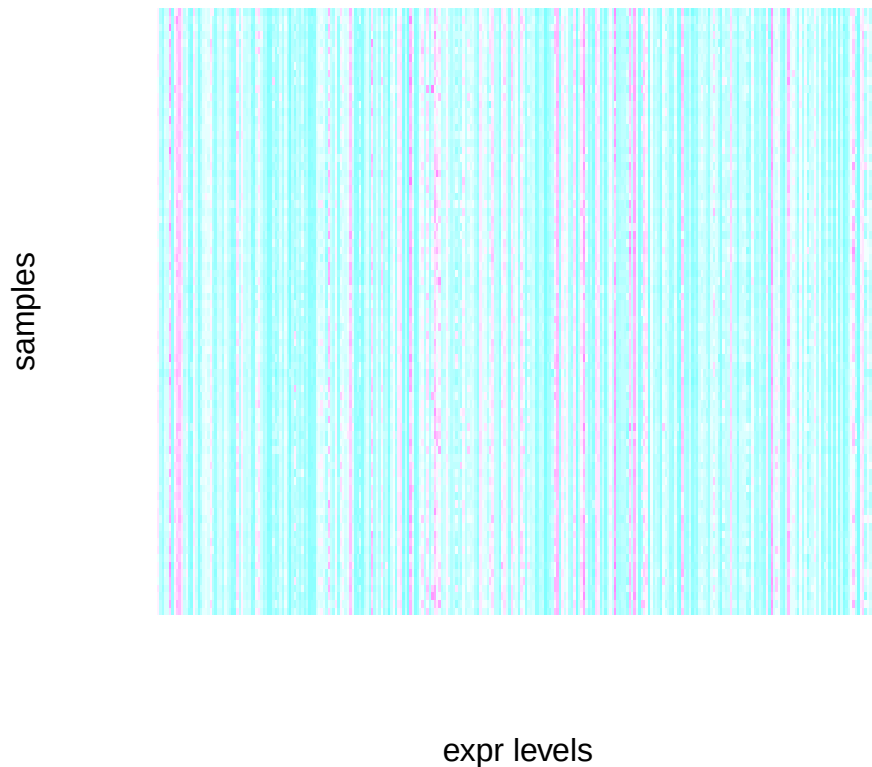


Figure 7.1: Visual representation of the first 300 expression probes across all samples.

💡 > Exercise 7.4 — MA plot

Draw an MA plot for two different samples and describe what you see.

7.2 Gene filtering

First filter out probesets with low variability across samples. We require expression above 100 fluorescence units in more than 25% of samples, and an interquartile range of at least 0.5 on the log2 scale, using Bioconductor's **genefilter**:

```
library(genefilter)
fa <- pOverA(0.25, log2(100))
fb <- function(x) (IQR(x) > 0.5)
fab <- filterfun(fa, fb)
selec <- genefilter(esub, fab)
esubf <- esub[selec, ]
esubf$mol <- factor(esubf$mol)
```

💡 > Exercise 7.5 — Filtering

1. How many probesets are finally selected?
2. Explain the filtering code.
3. Replot the MA plot, marking the discarded probes. Explain the results.

Next, use `limma` to find the most differentially expressed genes via an f -statistic. We fit a linear model of each probeset against the BCR/ABL vs NEG factor: `model.matrix()` sets up the model, `lmFit()` fits it, `eBayes()` computes the statistics, and `topTable()` ranks the genes:

```
library(limma)
initialmodel <- model.matrix(~ esubf$mol)
model <- lmFit(esubf, initialmodel)
modelstats <- eBayes(model)
topprobes <- topTable(modelstats, coef = 2, 25)
X <- esubf[rownames(topprobes)]
head(topprobes, 10)
```

	logFC	AveExpr	t	P.Value	adj.P.Val	B
1636_g_at	-1.1000116	9.196420	-9.201234	2.512298e-14	6.006904e-11	21.765333
39730_at	-1.1525269	9.000049	-8.707317	2.455619e-13	2.935693e-10	19.644093
1635_at	-1.2026753	7.897095	-7.389925	1.035653e-10	8.254157e-08	13.999019
40202_at	-1.7793784	8.621443	-6.378922	9.506229e-09	5.682349e-06	9.767965
37027_at	-1.3487023	8.444161	-5.805701	1.129901e-07	5.403189e-05	7.452028
36591_at	-1.1457106	9.234424	-5.388592	6.477019e-07	2.051406e-04	5.821221
40480_s_at	-0.8687145	7.757604	-5.381017	6.682399e-07	2.051406e-04	5.792099
33774_at	-0.9961947	8.035731	-5.374516	6.863757e-07	2.051406e-04	5.767118
39837_s_at	-0.4757069	7.144313	-5.265719	1.072178e-06	2.848419e-04	5.351197
37014_at	1.4137658	6.727731	5.196863	1.418928e-06	3.392657e-04	5.090057

A **volcano plot** shows the whole tested set at once — effect size (log fold change) against significance ($-\log_{10}$ adjusted p) — so the genes that are both strongly and reliably changed stand out in the upper corners (Figure 7.2):

```
library(ggplot2)
all_tt <- topTable(modelstats, coef = 2, number = Inf)
all_tt$sig <- with(all_tt, ifelse(adj.P.Val < 0.05 & abs(logFC) > 1,
                                ifelse(logFC > 0, "Up", "Down"), "n.s.))
ggplot(all_tt, aes(logFC, -log10(adj.P.Val), colour = sig)) +
  geom_point(alpha = 0.6, size = 1.1) +
  scale_colour_manual(values = c(Up = "#c0432f", Down = "#3a6ea5", n.s. = "grey70")) +
  geom_vline(xintercept = c(-1, 1), linetype = 2, colour = "grey60") +
  geom_hline(yintercept = -log10(0.05), linetype = 2, colour = "grey60") +
  labs(x = expression(log[2]~fold~change), y = expression(-log[10]~adjusted~p),
       colour = NULL, title = "BCR/ABL vs NEG")
```

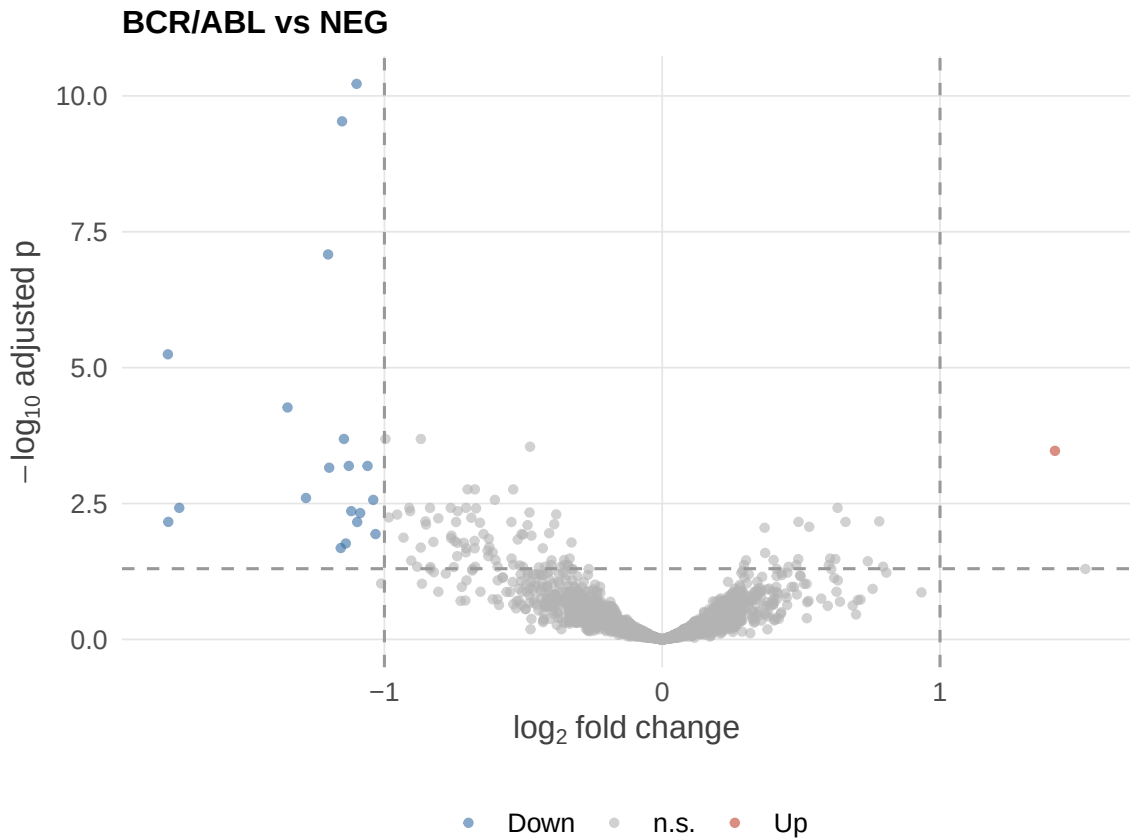


Figure 7.2: Volcano plot of BCR/ABL vs NEG. Points passing $FDR < 0.05$ and $|\log FC| > 1$ are highlighted.

The companion **MA plot** puts average expression on the x-axis, exposing whether fold changes depend on intensity (Figure 7.3):

```
ggplot(all_tt, aes(AveExpr, logFC, colour = adj.P.Val < 0.05)) +
  geom_point(alpha = 0.5, size = 1) +
  scale_colour_manual(values = c(`TRUE` = "#c0432f", `FALSE` = "grey70"),
    labels = c("FDR < 0.05", "n.s."), name = NULL) +
  geom_hline(yintercept = 0, colour = "grey40") +
  labs(x = "average expression (log2)", y = expression(log[2]~fold~change))
```

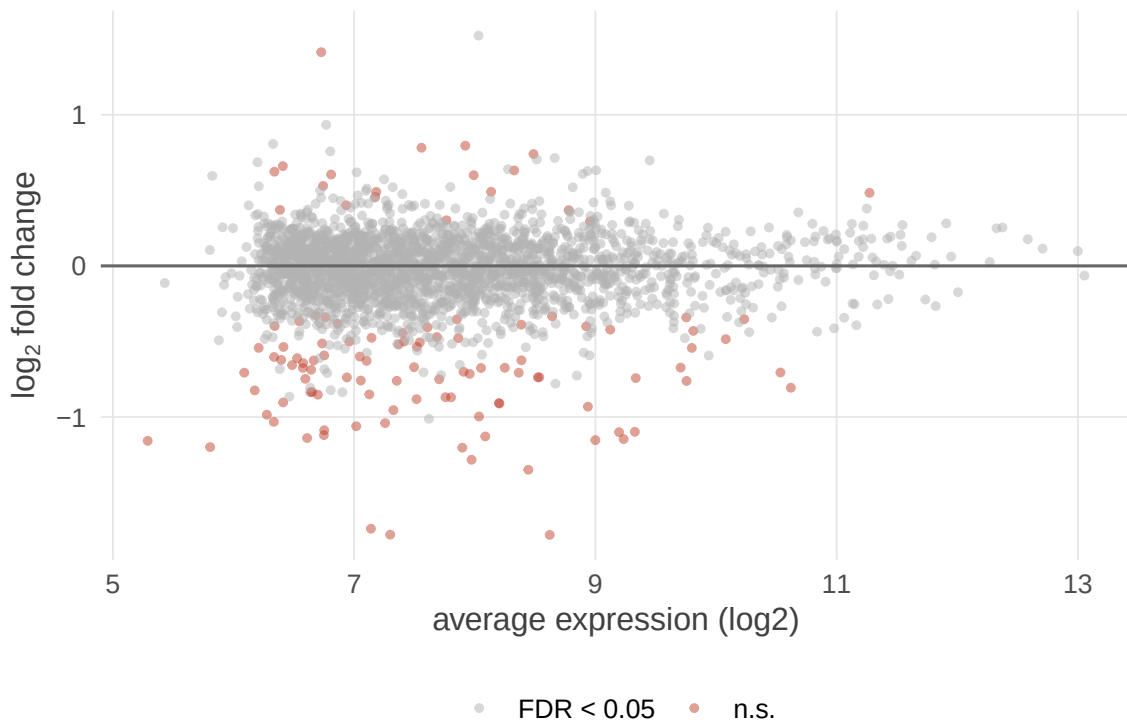


Figure 7.3: MA plot: log fold change against mean expression. Significant genes (FDR < 0.05) in red.

7.3 Data visualisation

Use a linear subspace (PCA) to visualise the dataset in lower dimension, colouring by mol (BCR/ABL vs NEG):

```
library(pls)
vmodel <- prcomp(t(exprs(X)))
scoreplot(vmodel, col = as.numeric(X$mol),
          pch = c(17, 16)[as.numeric(X$mol)])
legend("topright", legend = levels(X$mol),
      col = as.numeric(X$mol), pch = c(17, 16)[as.numeric(X$mol)])
```

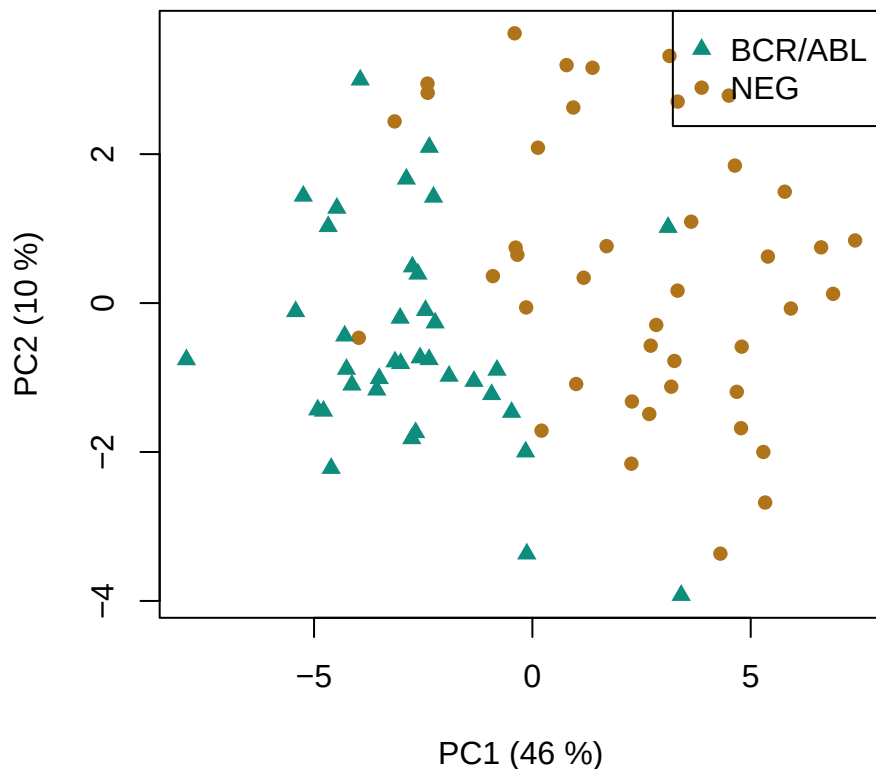


Figure 7.4: PCA score plot for BCR/ABL vs NEG on the ALL dataset.

💡 > Exercise 7.6 — Visualisation

1. What method are we using to visualise the dataset?
2. Using the projection vectors in `vmodel`, can you guess a ranking of the probes most related to BCR/ABL vs NEG?
3. Compare this with the first 10 probes from the linear model — are any probes common to both methods?

7.4 Clustering

A key goal of large-scale expression studies is to find biologically important clusters of samples and genes. The workhorse is **hierarchical clustering** of a distance matrix. We compute sample-to-sample distances on the top probes and draw them as an annotated heatmap with `pheatmap`, whose dendrograms cluster both axes (Figure 7.5):

```
library(pheatmap)
sampleDist <- dist(t(exprs(X)), method = "euclidean")
ann <- data.frame(subtype = X$mol)
rownames(ann) <- colnames(exprs(X))
pheatmap(as.matrix(sampleDist),
         annotation_col = ann, annotation_row = ann,
         show_rownames = FALSE, show_colnames = FALSE,
         annotation_colors = list(subtype = c("BCR/ABL" = "#c0432f", "NEG" = "#3a6ea5")),
         color = colorRampPalette(c("#0e8d7c", "white"))(50),
         main = "Clustering of individuals")
```

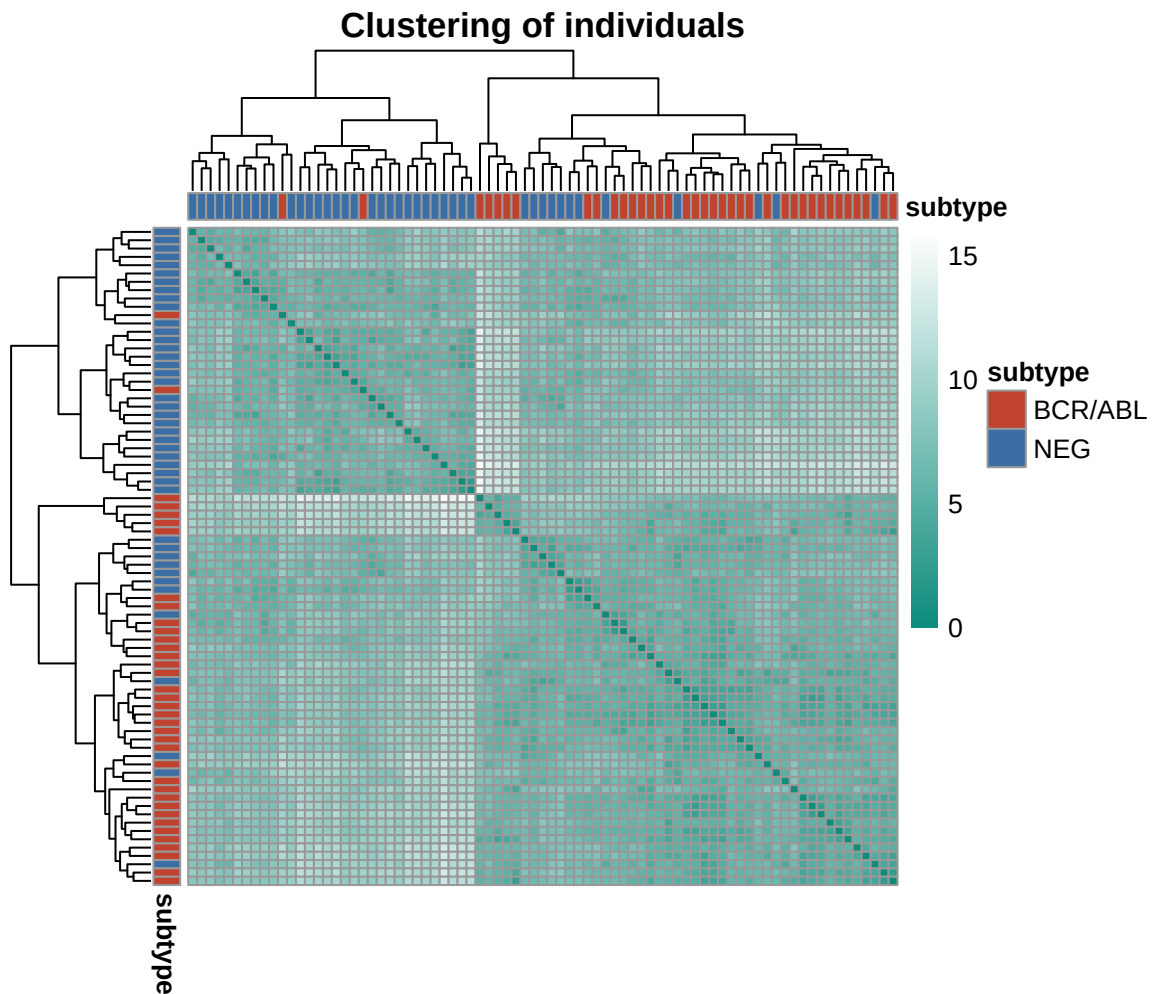


Figure 7.5: Sample-to-sample distance matrix on the top differentially expressed probes, clustered and annotated by molecular subtype.

Beyond hierarchical clustering

More elaborate algorithms exist — e.g. **HOPACH** (Hierarchical Ordered Partitioning And Collapsing Hybrid), which splits and re-collapses clusters to correct mistakes, or consensus clustering. With the `hopach` package installed the individuals cluster as `hopach(as.matrix(t(exprs(X))), dmat = distancematrix(..., d = "cosangle"))`.

💡 > Exercise 7.7 — Gene clustering

Repeat the clustering on the genes (rows) instead of the samples, and compare the gene groups with the differential-expression ranking.

💡 > Exercise 7.8 — k-means

1. Use two principal components to compute a two-cluster k -means.
2. Find the patients in each cluster and build a table of BCR/ABL vs NEG counts.
3. Is there any relationship between the k -means solution and the illness?

Expression is more commonly visualised with a **heatmap** — a false-colour representation with rows and columns ordered by clustering:

```
library(gplots)
cols <- c("#FF0000", "#0000FF")
names(cols) <- c("BCR/ABL", "NEG")
comap <- cols[as.character(esub$mol)]
heatmap(exprs(X), col = redgreen(100), ColSideColors = comap, scale = "none")
```

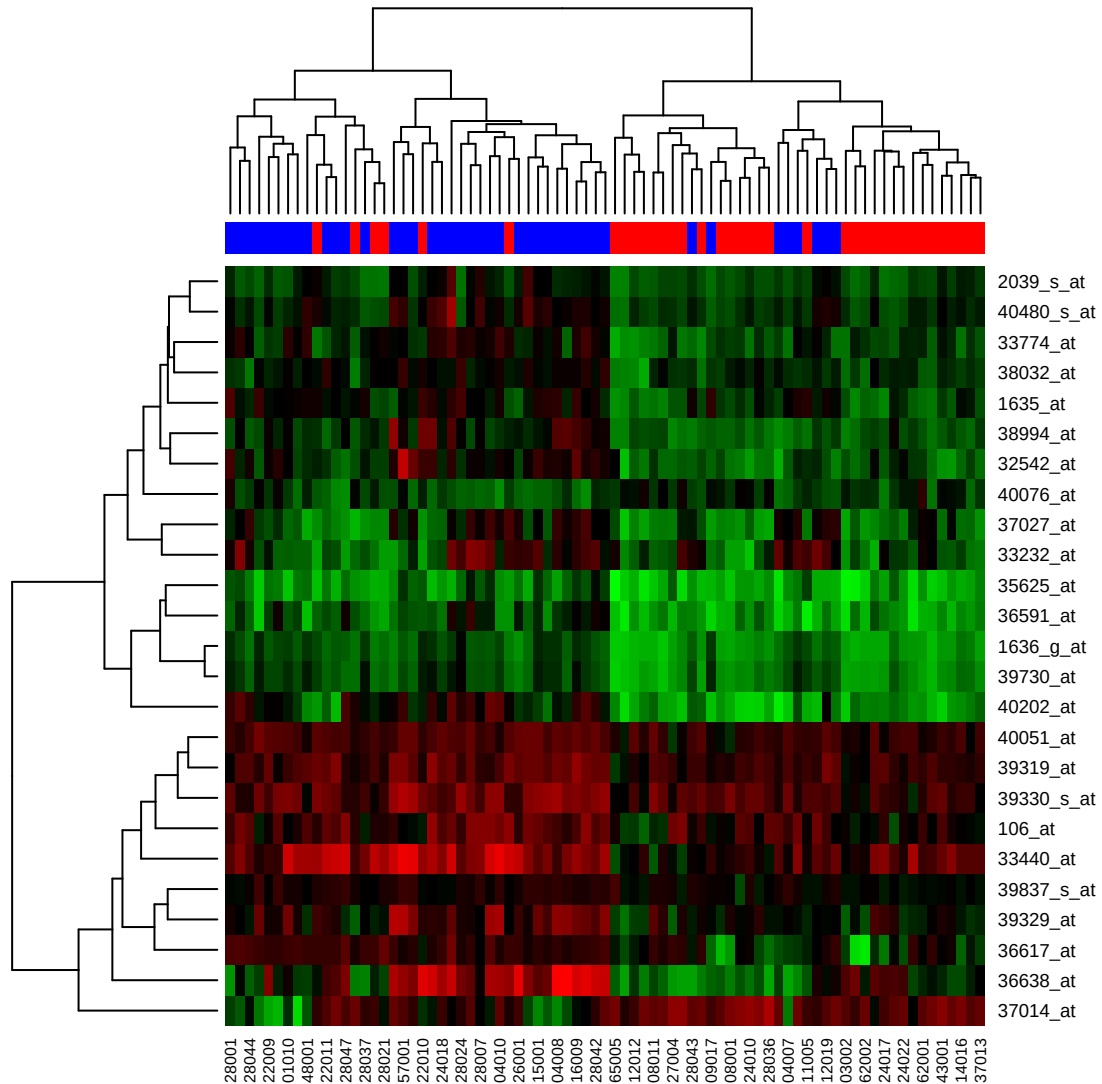


Figure 7.6: Heatmap of the two-way clustering of gene expression for BCR/ABL vs NEG.

💡 > Exercise 7.9 — Heatmap

1. Explain the code.
2. Interpret the resulting plot in terms of gene expression.

7.5 Classification

Can we guess BCR/ABL vs NEG from the expression profiles? As variables we use a 2D projection onto the PCA model from Section 7.3, via `predict()`:

```
factors <- 2
pX <- predict(vmodel)[, 1:factors]
dim(pX)
```

```
[1] 79 2
```

Check performance of a k -NN classifier with **leave-one-out** validation. LOO selects one sample j for prediction and trains on the remaining $i \neq j$, iterating over all N samples. In R this is `knn.cv()`; for $k = 1$:

```
library(class)
pred <- knn.cv(pX, X$mol, k = 1)
```

Check performance with `table()`:

```
table(pred, X$mol)
```

```
pred      BCR/ABL NEG
BCR/ABL      32  5
NEG           5 37
```

```
cr <- mean(pred == X$mol)
cr
```

```
[1] 0.8734177
```

> Exercise 7.10 — Classifier

1. What is `cr`?
2. Plot the classification ratio as you increase the number of factors — what is the optimum?
3. What is the optimum k (remember k should be odd)?

Finally, compute the classification boundary on the 2D model subspace by building a regular grid, applying k -NN at each node, and colouring the plane by predicted class:

```
k <- 1
grain <- 200
rgrid <- expand.grid(x = seq(1.10 * min(pX[, 1]), 1.1 * max(pX[, 1]), length = grain),
                    y = seq(1.10 * min(pX[, 2]), 1.1 * max(pX[, 2]), length = grain))
rgrid_p <- knn(train = pX, test = rgrid, cl = X$mol, k = k)
plot(rgrid, col = c("lightsalmon1", "lightblue1")[as.numeric(rgrid_p)],
     cex = 1, pch = 15, xlab = "PC1", ylab = "PC2")
points(pX, col = "black", bg = c("red", "blue")[as.numeric(X$mol)],
       cex = 1, pch = c(21, 22)[as.numeric(X$mol)], lwd = 1.5)
legend("topright", legend = levels(X$mol),
       col = c("red", "blue")[as.numeric(X$mol)], pch = c(21, 22)[as.numeric(X$mol)])
```

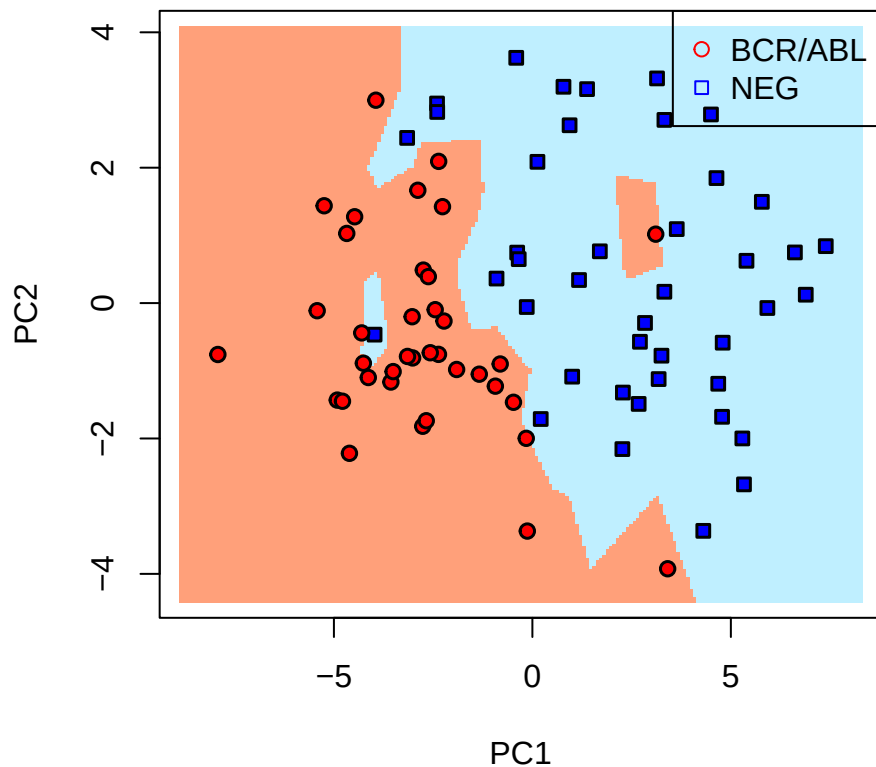


Figure 7.7: k-NN classification boundary for the first two PCs on the BCR/ABL vs NEG dataset.

💡 > Exercise 7.11 — Decision boundary

Modify the code with different values of k . Discuss the effect of k on the classification boundary.

7.6 A modern counterpart: RNA-seq with DESeq2

The ALL data above are **microarray** intensities, analysed with `limma`. Most expression studies today produce **RNA-seq counts**, where the analysis models discrete counts with a negative-binomial distribution. `DESeq2` is the standard tool. The workflow mirrors what you just did — fit a model, shrink, rank — but on counts. Here we use a self-contained simulated count dataset so it runs anywhere:

```
library(DESeq2)
set.seed(1)
dds <- makeExampleDESeqDataSet(n = 2000, m = 12, betaSD = 1) # 2000 genes, 12 samples
dds <- DESeq(dds, quiet = TRUE)
res <- results(dds)
summary(res)
```

```
out of 1995 with nonzero total read count
adjusted p-value < 0.1
LFC > 0 (up)      : 336, 17%
LFC < 0 (down)    : 317, 16%
```

```
outliers [1]      : 9, 0.45%
low counts [2]     : 193, 9.7%
(mean count < 2)
[1] see 'cooksCutoff' argument of ?results
[2] see 'independentFiltering' argument of ?results
```

DESeq2's own MA plot shrinks noisy low-count fold changes toward zero (Figure 7.8):

```
plotMA(res, main = "DESeq2 - simulated RNA-seq", colSig = "#c0432f")
```

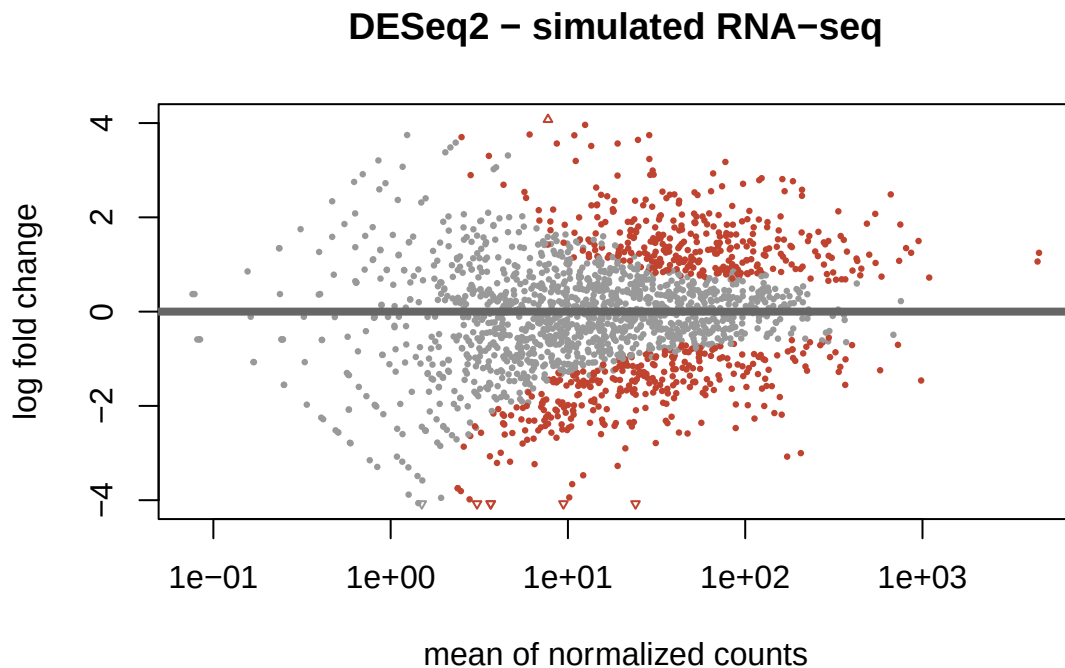


Figure 7.8: DESeq2 MA plot on the simulated RNA-seq data; significant genes ($\text{FDR} < 0.1$) in colour.

💡 > Exercise 7.12 — Microarray vs RNA-seq

Compare the `limma` (microarray) and `DESeq2` (RNA-seq) workflows: what does each assume about the data distribution, and where do the steps line up? Rerun the `DESeq2` fit with a larger `betaSD` and describe how the volcano/MA plots change.

7.7 Further reading

- Smyth (2004), `limma` — linear models + empirical Bayes for microarrays.
- Love *et al.* (2014), `DESeq2` — the negative-binomial RNA-seq model used in `DESeq2`.
- The [ALL vignette](#) for the dataset background.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

7.8 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

8 Functional Enrichment

The Human Genome Project produced the first reference sequence for most of the human genome (Lander et al., 2001). Modern reference assemblies and annotations have changed substantially since that draft, so exact gene counts and genome fractions must always be tied to an assembly and annotation release. Many proteomic, genetic, and metabolic analyses now produce a ranked list of *interesting* biomolecules — genes ranked by GWAS or differential-expression statistics, microarray probesets, or small molecules from LC-MS metabolomics.

Such ranked lists lack structure and context. Typical questions to answer from a list of biomarkers are (Tipney & Hunter, 2010): what does this gene and its protein product do; how and where; does it make sense on this list; does it interact with other genes/proteins; does its behaviour change during disease or therapy? Answering these gene-by-gene from the literature is cumbersome and biased. **Functional enrichment** analysis addresses this systematically.

By the end of this laboratory you will be able to:

- annotate a ranked gene list with symbols and names using `org.Hs.eg.db`
- perform GO and KEGG over-representation analysis with `clusterProfiler`
- visualise enrichment results (dotplots, concept networks)
- run network-based metabolomics enrichment by diffusion with `FELLA`

In this lab we perform two functional analyses: one with gene-expression data and one with metabolomics (LC-MS) data.

Requirements

The gene-annotation step runs with `org.Hs.eg.db` (using `resultTable.Rda` bundled in the repo under `data/`). The GO/KEGG enrichment (`clusterProfiler`) and the metabolomics diffusion (`FELLA`) need packages and live KEGG downloads that are not available at build time, so those chunks are shown but not executed here. Install the packages with `BiocManager::install("clusterProfiler")` / `BiocManager::install("FELLA")` to run them yourself.

Lab at a glance

Goal: turn a ranked gene list into biological meaning (GO, KEGG, GSEA). **Prerequisites:** [Differential Expression](#); [Searching NCBI & Ensembl](#). **Time:** ~2 h. **Packages:** `org.Hs.eg.db` (runs); `clusterProfiler`, `enrichplot`, `fgsea`, `FELLA` (install to run). **Deliverable:** a lab report.

8.1 Functional analysis in differential expression

8.1.1 Gene set functional enrichment

Load a sample results table (an .Rda from the course campus), similar to the results from the differential-expression lab:

```
library(org.Hs.eg.db)
load("data/resultTable.Rda")
resultTable
```

	logConc	logFC	LR.statistic	PValue	FDR
2152	-9.639471	-4.679498	378.0732	3.269307e-84	2.613484e-80
3199	-10.638865	-4.264830	291.1028	2.859424e-65	1.142912e-61
3202	-11.448981	-4.009603	222.3653	2.757135e-50	7.346846e-47
2159	-11.026699	-3.486593	206.7771	6.934967e-47	1.385953e-43
2147	-11.036862	3.064980	204.1235	2.630432e-46	4.205535e-43
5624	-12.276297	-4.695736	190.5368	2.427556e-43	3.234314e-40
7056	-12.063395	3.511937	188.4824	6.817155e-43	7.785191e-40
3579	-10.570501	2.758937	182.8937	1.131473e-41	1.130624e-38
7035	-12.378083	-4.654318	179.2331	7.126423e-41	6.329847e-38
8503	-12.027956	3.093029	158.9964	1.874723e-36	1.498654e-33

This small teaching table includes only 10 results ranked by p-value, with the log fold change ($\log_2 \text{FC} = \log_2 B - \log_2 A$) and **Entrez Gene IDs** as row names. Ten selected hits are enough to practise identifier mapping, but usually too few for a well-powered enrichment analysis.

> Exercise 8.1 — Annotation

Use `select()` and `merge()` to create a new data frame with SYMBOL and gene-name information.

```
resultTable$ENTREZID <- rownames(resultTable)
annots <- AnnotationDbi::select(org.Hs.eg.db, keys = resultTable$ENTREZID,
                                columns = c("SYMBOL", "GENENAME"), keytype = "ENTREZID")
resultTable <- merge(resultTable, annots, by = "ENTREZID", all.x = TRUE)
gene_ids <- unique(resultTable$ENTREZID)
head(resultTable)
```

	ENTREZID	logConc	logFC	LR.statistic	PValue	FDR	SYMBOL
1	2147	-11.036862	3.064980	204.1235	2.630432e-46	4.205535e-43	F2
2	2152	-9.639471	-4.679498	378.0732	3.269307e-84	2.613484e-80	F3
3	2159	-11.026699	-3.486593	206.7771	6.934967e-47	1.385953e-43	F10
4	3199	-10.638865	-4.264830	291.1028	2.859424e-65	1.142912e-61	HOXA2
5	3202	-11.448981	-4.009603	222.3653	2.757135e-50	7.346846e-47	HOXA5
6	3579	-10.570501	2.758937	182.8937	1.131473e-41	1.130624e-38	CXCR2

GENENAME

1	coagulation factor II, thrombin
2	coagulation factor III, tissue factor

```

3           coagulation factor X
4           homeobox A2
5           homeobox A5
6   C-X-C motif chemokine receptor 2

```

8.1.1.1 GO enrichment analysis

Install and load `clusterProfiler`, which tests a selected gene list against a defined background universe. In a real analysis, `background_ids` must contain all genes that could have been selected—for example, all genes retained after expression filtering—not every annotated human gene. The broader fallback below is included only because this teaching file does not contain the original tested universe:

```

library(clusterProfiler)
background_ids <- keys(org.Hs.eg.db, keytype = "ENTREZID") # replace when available
egoBP <- enrichGO(gene      = gene_ids,
                  universe   = background_ids,
                  OrgDb      = org.Hs.eg.db,
                  pAdjustMethod = "BH",
                  pvalueCutoff = 0.05,
                  ont        = "BP",
                  keyType    = "ENTREZID",
                  readable   = TRUE)

head(as.data.frame(egoBP))

```

Dotplots are a good graphical summary of the over-representation data:

```
dotplot(egoBP)
```

> Exercise 8.2 — GO

1. Draw a dot plot or enrichment map and interpret the gene ratio, adjusted p-value, and gene count.
2. Obtain over-represented GO terms for the Cellular Component and Molecular Function ontologies. Interpret the results.

8.1.1.2 KEGG enrichment analysis

Repeat the analysis for over-represented KEGG pathways. The `KEGG.db` annotation package has not been updated since 2012 (when KEGG went partly paywalled); `enrichKEGG` downloads the latest online KEGG data for the analysis:

```

kegg <- enrichKEGG(gene      = gene_ids,
                  organism    = "hsa",
                  pvalueCutoff = 0.05)

head(as.data.frame(kegg))

```

The relationship between KEGG pathways and the input gene list can be plotted with `cnetplot()` (a concept network); `enrichplot` also gives dot plots and enrichment maps:

```
library(enrichplot)
dotplot(ekegg) # top pathways by gene ratio
cnetplot(ekegg, categorySize = "pvalue") # pathway-gene network
emapplot(pairwise_termsim(egoBP)) # term-term similarity map
```

8.1.2 ORA vs GSEA

The analyses above are **over-representation analysis** (ORA): they ask whether a *cut* list of top genes is enriched for a term. **Gene-set enrichment analysis** (GSEA) instead uses the *whole ranked* list (e.g. by log fold change), so it can detect coordinated shifts of many small-effect genes that never pass a cutoff. GSEA requires the **complete**, uniquely named ranking—not this file of 10 preselected hits. Given a full differential-expression table called `full_results`, the workflow is:

```
# rank every tested gene by a signed statistic; names must be unique Entrez IDs
stopifnot(all(c("ENTREZID", "statistic") %in% names(full_results)))
full_results <- full_results[!duplicated(full_results$ENTREZID) &
                             !is.na(full_results$ENTREZID), ]
geneList <- sort(setNames(full_results$statistic, full_results$ENTREZID),
                 decreasing = TRUE)
gse <- gseGO(geneList, OrgDb = org.Hs.eg.db, ont = "BP", keyType = "ENTREZID")
ridgeplot(gse) # distribution of ranks per enriched set
gseaplot2(gse, geneSetID = 1)
```

> Exercise 8.3 — ORA vs GSEA

Using a complete ranked result table, run both ORA and GSEA. Where do they agree, and where does GSEA recover pathways ORA misses? Explain why GSEA cannot be validly run on the provided top-10 table.

8.2 Enrichment in metabolomics with FELLA

This case study examines the effects of environmental contamination on gilt-head bream (*Sparus aurata*) (Ziarrusta et al., 2018).¹ Fish were exposed to oxybenzone over 14 days, and changes were sought in brain, liver, and plasma using untargeted metabolomics (UHPLC-qOrbitrap MS, positive and negative modes, C18 and HILIC separation). Mortality and brain metabolites were unchanged, but liver and plasma showed perturbations — evidence of adverse effects beyond the well-studied hormonal activity.

We use network-based enrichment by diffusion with Bioconductor’s **FELLA** (Picart-Armada et al., 2017). FELLA starts from a list of affected metabolites and introduces a comprehensive output relating the top-ranked pathways to other biological entities, through a diffusion process on a holistic graph of the KEGG database. FELLA needs the KEGG graph and complementary data (a `FELLA.DATA` S4 object) and a list of affected metabolites as KEGG compounds (a `FELLA.USER` object).

¹Low concentrations of toxic products such as antidepressants in rivers can change swimming and feeding behaviour in some creatures. Such products reach the water environment — they are not fully removed in wastewater treatment — and can accumulate in fish brains (De Castro-Català et al., 2017).

8.2.1 Building a KEGG object

KEGG has no pathway annotations for *Sparus aurata* (Kanehisa et al., 2017), so it is common to use zebrafish (*Danio rerio*, organism code **dre**) as an approximation:

```
library(FELLA)
library(igraph)
library(magrittr)
tmpdir <- paste0("./kegg_database")

# Filter the dre01100 overview pathway, as in the article
graph <- buildGraphFromKEGGREST(organism = "dre", filter.path = c("01100"))

# Ensure the database does not already exist (FELLA will not overwrite it)
unlink(tmpdir, recursive = TRUE)
buildDataFromGraph(keggdata.graph = graph,
                   databaseDir = tmpdir, internalDir = FALSE,
                   matrices = "none", normality = "diffusion", niter = 100)
```

Load the **FELLA.DATA** object, which represents the relationships among KEGG entities (Pathways, Modules, Enzymes, Reactions, Compounds):

```
fella.data <- loadKEGGdata(databaseDir = tmpdir, internalDir = FALSE, loadMatrix = "none")
fella.data
```

8.2.2 Enrichment analysis on liver tissue

Find the KEGG identifiers associated with metabolic changes in liver tissue from (Ziarrusta et al., 2018), build a vector **cpd.liver** of KEGG compound IDs, and enrich it against **fella.data**:

```
cpd.liver <- c("C12623", "C01179", "C05350", "C05598", "C01586")
```

```
analysis.liver <- enrich(compounds = cpd.liver, data = fella.data,
                        method = "diffusion", approx = "normality")
```

Check whether all input metabolites were mapped:

```
analysis.liver %>% getInput %>% getName(data = fella.data)
```

Plot the resulting network of entities found by FELLA:

```
plot(analysis.liver, method = "diffusion", data = fella.data,
     nlimit = 250, plotLegend = FALSE)
```

Retrieve the sub-network as an **igraph** object, or as a table:

```
g.liver <- generateResultsGraph(object = analysis.liver, data = fella.data,
                              method = "diffusion")
tab.liver <- generateResultsTable(object = analysis.liver, data = fella.data,
```

```
head(tab.liver)                                method = "diffusion")
```

> Exercise 8.4 — Plasma metabolites

1. Repeat the analysis with the plasma metabolites reported in (Ziarrusta et al., 2018).
2. Discuss the enriched entities in relation to the paper's goals.

8.2.3 FELLA graphical user interface

FELLA also ships a graphical user interface (based on `shiny`, launched with `launchApp()`) with tabs for **Compounds upload** (submit the metabolite list and see mapped vs mismatching compounds), **Advanced options** (database, method and approximation choice, parameter tweaking, GO label for semantic similarity), **Results** (an interactive sub-graph of the top- k KEGG entries with a linked table), and **Export** (several tabular and network export options).

8.3 Further reading

- Yu *et al.* (2012), *OMICS* — `clusterProfiler` for ORA/GSEA.
- Subramanian *et al.* (2005), *PNAS* — the original GSEA method.
- The [FELLA vignette](#) for the metabolomics diffusion analysis.

Worked solutions for this lab are omitted from this PDF — find them as collapsed callouts in the HTML edition.

8.4 Report

Write a laboratory report following the structure of this text. It should contain an introduction, the goals of each section, and a description of the results of each part. Include any graphics relevant to making your work understandable. **The lab should be delivered within two weeks.**

References

- Buolamwini, J., & Gebru, T. (2018). Gender shades: Intersectional accuracy disparities in commercial gender classification. *Proceedings of the 1st Conference on Fairness, Accountability and Transparency, Proceedings of Machine Learning Research*, 81, 77–91.
- Cho, Y. M., Kim, M., Park, K. S., Kim, S. Y., & Lee, H. K. (2003). S20G mutation of the amylin gene is associated with a lower body mass index in korean type 2 diabetic patients. *Diabetes Res Clin Pract*, 60(2), 125–129.
- De Castro-Català, N., Muñoz, I., Riera, J. L., & Ford, A. T. (2017). Evidence of low dose effects of the antidepressant fluoxetine and the fungicide prochloraz on the behavior of the keystone freshwater invertebrate gammarus pulex. *Environmental Pollution (Barking, Essex : 1987)*, 231, 406–414. <https://doi.org/10.1016/j.envpol.2017.07.088>
- Hornik, K. (2007). *The R FAQ*. <http://CRAN.R-project.org/doc/FAQ/R-FAQ.html>
- Kanehisa, M., Furumichi, M., Tanabe, M., Sato, Y., & Morishima, K. (2017). KEGG: New perspectives on genomes, pathways, diseases and drugs. *Nucleic Acids Research*, 45, D353–D361. <https://doi.org/10.1093/nar/gkw1092>
- Lander, E. S., Linton, L. M., Birren, B., Nusbaum, C., Zody, M. C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., Funke, R., Gage, D., Harris, K., Heaford, A., Howland, J., Kann, L., Lehoczy, J., LeVine, R., McEwan, P., ... Consortium, I. H. G. S. (2001). Initial sequencing and analysis of the human genome. *Nature*, 409, 860–921. <https://doi.org/10.1038/35057062>
- Picart-Armada, S., Fernández-Albert, F., Vinaixa, M., Rodríguez, M. A., Aivio, S., Stracker, T. H., Yanes, O., & Perera-Lluna, A. (2017). Null diffusion-based enrichment for metabolomics data. *PloS One*, 12, e0189012. <https://doi.org/10.1371/journal.pone.0189012>
- Sakagashira, S., Sanke, T., Hanabusa, T., Shimomura, H., Ohagi, S., Kumagaye, K. Y., Nakajima, K., & Nanjo, K. (1996). Missense mutation of amylin gene (S20G) in japanese NIDDM patients. *Diabetes*, 45(9), 1279–1281.
- Tipney, H., & Hunter, L. (2010). An introduction to effective use of enrichment analysis software. *Human Genomics*, 4, 202–206.
- Turk, M., & Pentland, A. (1991). Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1), 71–86.
- Venables, W. N., & Smith, D. M. (2006). *An introduction to R: Notes on R: A programming environment for data analysis and graphics*. R Development Core Team.
- Wu, F., Zhao, S., Yu, B., Chen, Y.-M., Wang, W., Song, Z.-G., Hu, Y., Tao, Z.-W., Tian, J.-H., Pei, Y.-Y., et al. (2020). A new coronavirus associated with human respiratory disease in china. *Nature*, 579(7798), 265–269.
- Zhou, P., Yang, X.-L., Wang, X.-G., Hu, B., Zhang, L., Zhang, W., Si, H.-R., Zhu, Y., Li, B., Huang, C.-L., et al. (2020). A pneumonia outbreak associated with a new coronavirus of probable bat origin. *Nature*, 1–4.
- Ziarrusta, H., Mijangos, L., Picart-Armada, S., Irazola, M., Perera-Lluna, A., Usobiaga, A., Prieto, A., Etxebarria, N., Olivares, M., & Zuloaga, O. (2018). Non-targeted metabolomics reveals alterations in liver and plasma of gilt-head bream exposed to oxybenzone. *Chemosphere*, 211, 624–631. <https://doi.org/10.1016/j.chemosphere.2018.08.013>
- Zimmet, P., Alberti, K. G., & Shaw, J. (2001). Global and societal implications of the diabetes epidemic. *Nature*, 414(6865), 782–787. <https://doi.org/10.1038/414782a>

Glossary

A quick reference to key terms used across the labs.

Annotation — mapping identifiers (genes, probes, SNPs) to biological information (names, functions, locations, pathways).

BLAST — *Basic Local Alignment Search Tool*; finds regions of similarity between a query sequence and a database.

Bioconductor — an R project and package repository for the analysis of high-throughput genomic data.

Differential expression — genes whose expression differs systematically between conditions (e.g. disease vs control).

e-value — in a sequence search, the expected number of hits as good as the observed one that would occur by chance; smaller is more significant.

Eigenface / principal component — an orthogonal direction of maximum variance in a dataset; for face images, itself an image.

Enrichment (ORA / GSEA) — testing whether a gene set (e.g. a pathway) is over-represented among hits (over-representation analysis) or shifted in a ranked list (gene-set enrichment analysis).

Expression set (ExpressionSet) — a Bioconductor container bundling an expression matrix with sample (phenotype) and feature (gene) metadata.

FDR (false discovery rate) — the expected proportion of false positives among declared discoveries; controlled with methods such as Benjamini–Hochberg.

Founder — in a pedigree, an individual whose parents are not in the dataset.

Genomic inflation factor (lambda) — the ratio of observed to expected median test statistic in a GWAS; lambda near 1 indicates well-controlled confounding.

GWAS — genome-wide association study; tests each of many SNPs for association with a phenotype.

Homology / orthology / paralogy — shared ancestry between sequences; orthologues diverged by speciation, paralogues by duplication.

HWE (Hardy–Weinberg equilibrium) — the expected relationship between allele and genotype frequencies in an ideal population; departures can flag genotyping error or structure.

IBS / kinship — identity-by-state / relatedness measured from shared alleles across many markers.

Manhattan plot — a GWAS scatter plot of $-\log_{10}$ p-value against genomic position.

Multidimensional scaling (MDS) — a projection of a distance matrix into few dimensions, used here to reveal population structure.

QQ plot — quantile–quantile plot of observed vs expected p-values; used to judge calibration and inflation.

SNP — single-nucleotide polymorphism; a single-base variant in the genome.

Session information

The book is built with the environment below. Pin these versions (see the project DESCRIPTION and `renv.lock`) to reproduce the analyses exactly.

```
sessionInfo()
```

```
R version 4.3.3 (2024-02-29)
```

```
Platform: x86_64-pc-linux-gnu (64-bit)
```

```
Running under: Ubuntu 24.04.4 LTS
```

```
Matrix products: default
```

```
BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.12.0
```

```
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0
```

```
locale:
```

```
[1] C
```

```
time zone: Etc/UTC
```

```
tzcode source: system (glibc)
```

```
attached base packages:
```

```
[1] stats      graphics  grDevices  utils      datasets  methods    base
```

```
loaded via a namespace (and not attached):
```

```
[1] compiler_4.3.3 fastmap_1.1.1 cli_3.6.2      tools_4.3.3
```

```
[5] htmltools_0.5.7 yaml_2.3.8      rmarkdown_2.25 knitr_1.45
```

```
[9] jsonlite_1.8.8 xfun_0.41      digest_0.6.34  rlang_1.1.3
```

```
[13] evaluate_0.23
```

```
Quarto version: 1.6.40
```