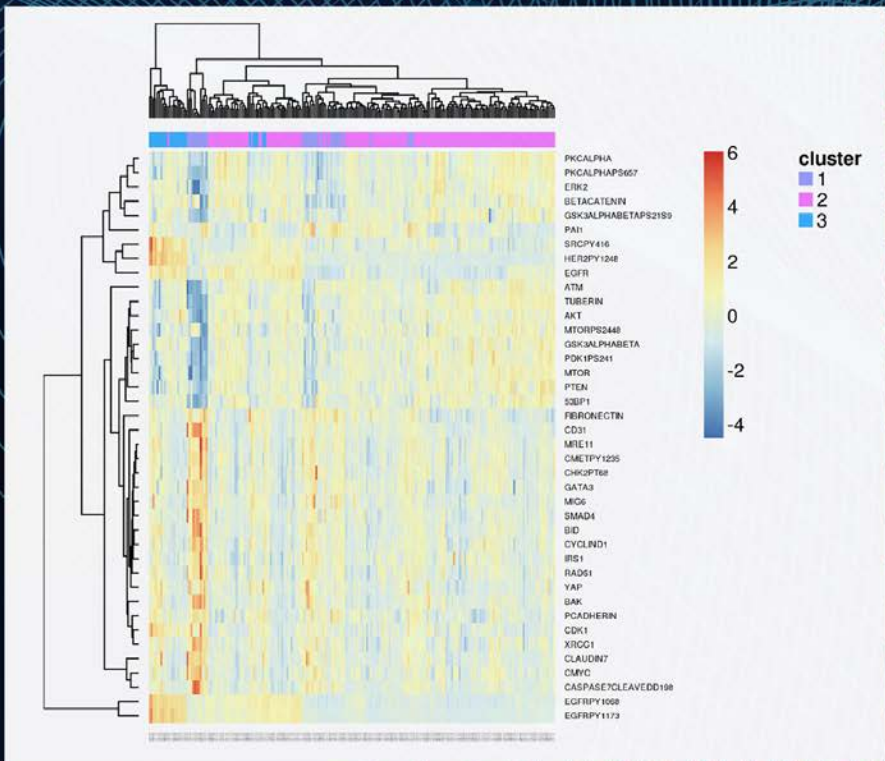


Introduction to Bioinformatics with R

A Practical Guide for Biologists



Edward Curry



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Introduction to Bioinformatics with R

Chapman & Hall/CRC Mathematical and Computational Biology

About the Series

This series aims to capture new developments and summarize what is known over the entire spectrum of mathematical and computational biology and medicine. It seeks to encourage the integration of mathematical, statistical, and computational methods into biology by publishing a broad range of textbooks, reference works, and handbooks. The titles included in the series are meant to appeal to students, researchers, and professionals in the mathematical, statistical, and computational sciences and fundamental biology and bioengineering, as well as interdisciplinary researchers involved in the field. The inclusion of concrete examples and applications and programming techniques and examples is highly encouraged.

Series Editors

Xihong Lin

Mona Singh

N. F. Britton

Anna Tramontano

Maria Victoria Schneider

Nicola Mulder

Introduction to Proteins

Structure, Function, and Motion, Second Edition

Amit Kessel, Nir Ben-Tal

Big Data in Omics and Imaging

Integrated Analysis and Causal Inference

Momia Xiong

Computational Blood Cell Mechanics

Road Towards Models and Biomedical Applications

Ivan Cimrak, Iveta Jancigova

An Introduction to Systems Biology

Design Principles of Biological Circuits, Second Edition

Uri Alon

Computational Biology

A Statistical Mechanics Perspective, Second Edition

Ralf Blossey

Computational Systems Biology Approaches in Cancer Research

Inna Kuperstein and Emmanuel Barillot

Introduction to Bioinformatics with R

A Practical Guide for Biologists

Edward Curry

Analyzing High-Dimensional Gene Expression and DNA Methylation Data with R

Hongmei Zhang

For more information about this series please visit: <https://www.crcpress.com/Chapman--HallCRC-Mathematical-and-Computational-Biology/book-series/CHMTHCOMBIO>

Introduction to Bioinformatics with R

A Practical Guide for Biologists

Edward Curry



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

First edition published 2020
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

© 2021 Taylor & Francis Group, LLC

CRC Press is an imprint of Taylor & Francis Group, LLC

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

ISBN: 9781138498952 (hbk)
ISBN: 9781138495715 (pbk)
ISBN: 9781351015318 (ebk)

Typeset in CMR
by Nova Techset Private Limited, Bengaluru & Chennai, India

Contents

Acknowledgements	xi
1 Introduction	1
1.1 Why informatics is important for biologists	1
1.2 How to use this book	2
2 Introduction to R	5
2.1 Obtaining R	5
2.1.1 Downloading R	5
2.1.2 Installing R	6
2.2 R console	6
2.2.1 Starting the R console	7
2.3 The R workspace	7
2.3.1 Creating/deleting objects	8
2.3.2 The working directory	8
2.4 Data handling	10
2.4.1 Basic data types	10
2.4.2 Vectors	11
2.4.3 Arrays	11
2.4.4 Lists	12
2.4.5 Data frames	14
2.4.6 Data input/output	15
2.5 More advanced concepts: Scripts and functions	16
2.5.1 Simple scripts	16
2.5.2 Functions	17
2.5.3 Using ‘apply’	19
2.5.3.1 apply	19
2.5.3.2 sapply	20
2.5.3.3 lapply	22
2.5.3.4 mapply	23
2.6 Plots	24
2.6.1 Simple scatterplot	24
2.6.2 Arguments of <code>plot()</code>	25
2.6.3 Multiple plots on one graph	25
2.6.4 Scatterplots of multiple variables	25
2.6.5 Box plots	25
2.6.6 Saving images to file	27

2.7	More advanced graphics with <i>ggplot2</i>	27
2.8	Using R help	30
3	An Introduction to LINUX for Biological Research	31
3.1	UNIX	31
3.2	Linux survival guide	32
3.3	Useful dependencies and programs	37
4	Statistical Methods for Data Analysis	39
4.1	What are statistical methods, and why do we use them in biological research?	39
4.1.1	A worked example	40
4.1.2	A brief summary	43
4.2	What do I need to understand statistics?	43
4.2.1	Probability	43
4.2.1.1	Random variables	43
4.2.1.2	Probability distributions	45
4.2.1.3	Hypothesis testing	47
4.2.2	Linear algebra	52
4.2.3	Summary	53
4.3	Normalization: Removing technical variation	53
4.3.1	Centering and scaling	55
4.3.2	An illustrative example	58
4.3.3	Quantile normalization	59
4.3.4	Batch effects	59
4.4	Correlation	60
4.4.1	Pearson correlation coefficient	60
4.4.2	Spearman's rank correlation	61
4.4.3	Examples	61
4.5	Clustering	65
4.5.1	Clustering illustration using R	66
4.6	Linear regression models	69
4.6.1	Limma	72
4.6.1.1	Installing limma	73
4.6.1.2	Categorical explanatory variables	73
4.6.1.3	Continuous explanatory variables	76
4.7	Multiple hypothesis testing	78
4.8	Survival analysis	79
4.8.1	Kaplan-Meier plots	79
4.8.2	Cox proportional hazards regression models	81
4.9	Projection methods	81
4.9.1	PCA	82
4.9.2	PLS	85
4.10	Resampling: Permutation tests and the bootstrap	86

4.11	Stability and robustness	87
4.12	Summary	87
5	Analyzing Generic Tabular Numeric Datasets in R	89
5.1	Introduction	89
5.2	Loading data into R	89
5.3	Data visualisation	92
5.3.1	Scatter plots	92
5.3.2	Box plots	93
5.3.3	Bar charts	94
5.4	Correlation and clustering	94
5.4.1	Correlation	95
5.4.2	Clustering	98
5.4.3	Heatmaps	101
5.5	Statistical analysis using linear models	103
5.5.1	Comparison of two groups	104
5.5.2	Alternative models	106
5.6	Summary	107
6	Functional Enrichment Analysis	109
6.1	Introduction	109
6.2	Loading gene sets into R	109
6.3	Over-representation	112
6.3.1	Online tools	113
6.3.2	Testing gene sets in R	113
6.4	Systematic enrichment	117
6.4.1	Online tools	117
6.4.2	Testing gene sets in R	117
6.5	Summary	120
7	Integrating Multiple Datasets in R	121
7.1	Introduction	121
7.2	Data import	123
7.3	Exploratory data analysis	123
7.4	Integrating multiple datasets	131
7.4.1	Survival analysis	134
7.5	Multiple molecular endpoints	141
7.6	Summary	143
8	Analyzing Microarray Data in R	145
8.1	Bioconductor	146
8.2	Accessing microarray data from GEO	147
8.3	Single-channel array analysis	148
8.4	Loading data	148
8.5	Data visualisation	149
8.5.1	Image plots	150

8.5.2	MA plots	151
8.5.3	Scatterplots	151
8.5.4	Box plots	153
8.6	Normalizing data	155
8.7	Differential expression (linear models)	158
8.7.1	Design matrix	159
8.7.2	Fitting linear models	160
8.7.3	Making use of the results	161
8.7.4	Postscript: Assumptions	164
8.8	Clustering and correlation	164
8.8.1	Expression profiles	164
8.8.2	Correlation	165
8.9	Clustering	169
8.9.1	Filtering	171
8.10	Survival analysis	175
8.10.1	Kaplan-Meier plots	178
8.10.2	Cox proportional hazards regression	183
8.11	Footnote: Correlation to explore associated functions	187
9	Analyzing DNA Methylation Microarray Data in R	189
9.1	Introduction	189
9.2	Importing raw data	190
9.3	Quality control	191
9.4	Normalization and estimating methylation level	193
9.5	Analyzing beta values	194
9.6	Using previously preprocessed data	197
9.7	Further analyses using minfi	200
10	DNA Analysis with Microarrays	203
10.1	Introduction	203
10.2	Genotyping	203
10.2.1	Normalization	204
10.2.2	Genotype calling	205
10.2.3	Downstream analysis: Genome-wide association tests	208
10.3	Copy number analysis	210
10.3.1	Normalization	211
10.3.2	Copy number estimation	212
10.3.3	Segmentation	212
10.3.3.1	Hidden Markov model	213
10.3.3.2	Circular binary segmentation	216
10.3.4	Downstream analysis	217
10.3.4.1	Mapping CNA data to genes	217
10.3.4.2	Finding frequently-mutated genes	220
10.4	Summary	221

11 Working with Sequencing Data	223
11.1 Introduction	223
11.2 Sequence data analysis tasks	224
11.3 Quality control	224
11.3.1 Base call quality filtering	226
11.3.2 Adapter trimming	228
11.4 Alignment	230
11.4.1 Bowtie	231
11.4.2 BWA	232
11.4.3 Post-alignment filtering	233
11.4.4 Removing duplicate reads	233
11.5 Obtaining sequencing data from the SRA	235
12 Genomic Sequence Profiling	239
12.1 Introduction	239
12.2 SNV: Single nucleotide variants	239
12.3 Variant filtering and annotation	241
12.4 Indels: Short insertions and deletions	244
12.5 SV: Structural variants	245
12.6 Making use of variant calls	246
12.7 Summary	256
13 ChIP-seq	259
13.1 Introduction	259
13.2 Cross-correlation	259
13.3 Filtering blacklisted reads	263
13.4 Peak calling	263
13.5 Peak annotation	265
13.6 Quantitative comparisons of ChIP-seq libraries	267
13.7 Summary	270
14 RNA-seq	271
14.1 Introduction	271
14.2 Obtaining RNA-seq data from GEO	272
14.3 Transcript quantification via pseudoalignment	273
14.3.1 Building a transcript index	273
14.3.2 Quantifying transcripts using reads	274
14.3.3 Downstream analysis	275
14.4 Analysis with transcriptome assembly	278
14.4.1 Building the transcriptome directly	279
14.4.2 Transcript quantification	280
14.4.3 Downstream analysis	282
14.5 Summary	285

15 Bisulphite Sequencing	287
15.1 Introduction	287
15.2 Alignment and methylation calls	289
15.3 Downstream analysis	290
15.4 Summary	293
16 Final Notes	295
Index	297

Acknowledgements

This book would not have been possible without the excellent students I taught over nearly a decade at Imperial College London. I consider myself extremely grateful to have had the opportunity to teach them, and to learn from them. Likewise, to work with and learn from colleagues who have become life-long friends: Adam Beech, Emma Bell, Charlotte Wilhelm-Benartzi, Nair Bonito, Paula Cunnea, Kirsty Flower, Ian Garner, Ian Green, Erick Loomis, Alun Passey, Euan Stronach, Angela Wilson and many others. I feel I particularly need to thank Professor Bob Brown for his professional support and guidance, and James Flanagan for running the MRes Cancer Informatics course with me. I am also grateful to Philippe Sanseau and the Computational Biology team at GSK, for welcoming me into an exciting research environment.

Special thanks to my wife Vaughan and my family for their invaluable personal support, and to David Grubbs at Taylor & Francis for giving me the opportunity to turn my collection of tutorials into this book.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

1

Introduction

1.1 Why informatics is important for biologists

This is really all about data. In particular, it's about working with so much data that learning to program computers to perform calculations for us will save a lot of time, and probably make possible analysis that would otherwise be impossible. In biological research, the amount of data available to researchers has increased so much over recent years this has been described as a 'data explosion'[1].

Much of this biological data is freely available for any researcher to access and use in their own work. Therefore, any biological scientist who learns skills to enable obtaining, preprocessing and analyzing publically-available datasets, is giving themselves an advantage when it comes to making the most out of their own opportunities.

One consequence of this increase in biological data is that many of the recent paradigms of molecular biology come from computational analysis of large collections of data. In terms of developing an intuition for what is shown when results from computational analysis is presented in a paper, there is no substitute for first-hand experience of using a method for data analysis in your own research (of course, a theoretical understanding of the method in question is also important!). In reality, it is becoming increasingly difficult to understand the current state of the art in biological research without some experience and understanding of computational biology.

In 2014, the UK's MRC and BBSRC (Medical Research Council and Biotechnology & Biological Sciences Research Council) produced a report of 'skills vulnerabilities', which reflected important research capabilities lacking in the UK. Both in 2014 and in a 2017 update¹, computational methods for biological research were identified as key weaknesses. In fact, the following specific points were highlighted:

- Data analytics, especially bioinformatics, appear to be particularly vulnerable.

¹<https://mrc.ukri.org/documents/pdf/review-of-vulnerable-skills-and-capabilities/>

- Informatics skills are applicable to many areas of both the biosciences and the medical sciences.
- Maths, statistics and computational biology skills are lacking particularly at the postgraduate and postdoctoral levels, with many respondents reporting difficulties in recruiting adequately skilled researchers at these levels; shortages are not just restricted to the UK.

So there is a recognized international shortage of bioinformatics skills, and these skills are increasingly fundamental across all areas of biological research. You were probably already aware of this given you're reading this, but it hopefully serves as a motivating reminder that learning the bioinformatics skills taught in this book will be worth the effort you put in!

1.2 How to use this book

This book was developed over a decade of my experience training biologists to empower their own research through making better use of computers. I think there are three key aspects of this training, which are in essence the intended learning outcomes of this book:

1. theoretical *understanding* of how a set of computational analysis steps produce a result that yields biological insight
2. ability to *plan* a set of analysis steps that, when carried out on a given dataset, will yield biological insight
3. practical experience of *enacting* those plans on real datasets to produce novel, valuable research results

For the first of these, reading the chapters of this book should help. Reading this book should also help with the second. But the only way to gain the skills to carry out data analysis to give research results is to do it. There is simply no substitute for practical experience. Furthermore, the more experience you get carrying out data analysis, the more instinctively you will be able to plan analyses for your own research and to think of the best datasets to work with. Because there is no substitute for practice, this book is designed to give all the practical guidance someone needs to be able to carry out a set of analysis procedures. We will cover the procedures that are particularly useful for harnessing different types of biological data.

Because a lot of data analysis tools are not implemented in tools with convenient graphical user interfaces (GUIs), there is no avoiding a bit of coding. While at first this will almost certainly be frustrating to those new to a command line interface, with time and practice you will find that the automation

you can implement empowers you to achieve all sorts of things that would otherwise be impossible (or at least impractical). To help in this process, (all) required computer code is provided, which are effectively individual commands given to the computer. Each line² of code is followed with detailed descriptions of every part of every command.

The first chapters of this book introduce R and the Unix command shell, which will be indispensable tools for data analysis. This will involve learning some of the building blocks for programming computers to perform many tasks in one go, without requiring continued instruction from a human. Many of the methods we use are theoretically simple enough to calculate by hand with a small set of observations, but the beauty of using command-line tools is that you can program them to perform huge numbers of repetitive tasks very quickly and automatically. One should also not underestimate the importance and power of ‘data wrangling’, which acknowledges that the format in which you obtain data is rarely exactly the format that you need it in to perform the analyses you want.

The fourth chapter explains the mathematical theory behind the analysis methods that are employed throughout this book. To understand the theory, we’ll make use of the R environment to look at a few practical examples. Generally, I take the philosophy that a solid understanding of a few very versatile methods is the best strategy to enable a great variety of applications with as little effort as possible. A recurring theme of my research supervision is that the simpler your approach to demonstrate a finding, the better (as long as it’s appropriate): it will be understandable to more people, and therefore have greater impact, and will be less likely to be misinterpreted.

[Chapters 5 to 7](#) use real research examples to build up your practical experience of obtaining and analyzing biological datasets, utilizing the statistical analysis methods described in [Chapter 4](#). The examples use already-processed datasets, so that the focus is on the analysis rather than worrying about formats. The complexity of the tasks and the datasets involved builds through these chapters, so that by the end of [Chapter 7](#) we are systematically evaluating patterns of variation of hundreds of features from multiple platforms used to characterize different aspects of the same samples.

And finally, the bulk of this book by volume guides you through the specifics of working with different types of biological datasets. I have included those I think are the most frequently-encountered across molecular biology research, but this is certainly influenced by my own background in cancer research. The choice of data types to cover also balances the accessibility of obtaining,

²Note that this is a line as the computer sees it, which really means one complete set of instructions. A line of code may span multiple lines on a page or screen!

pre-processing and analyzing the data, so that we get the most out of the least effort.

A word of warning: it is easy to feel isolated in research, and that can be problematic when you find yourself, still new to bioinformatics, as the expert for your research group or team. There is an excellent blog post from Mick Watson³ on problems facing ‘lonely bioinformaticians’. Most importantly, don’t be afraid of looking to others for help.

You can do this! Stick with it, and you should find that you’re able to make more use of the data you generate and the vast accumulation of molecular biology data that is already in the public domain.

Bibliography

[1] V Marx. “The big challenges of big data,” *Nature* 498:255-260 (2013).

³<http://www.opiniomics.org/a-guide-for-the-lonely-bioinformatician/>

2

Introduction to R

In a practical guide to data analysis, we will help to learn how to use some tools that enable us work with data. As any collection of information can potentially become a dataset that we can use to find answers to real-world questions, one of the things that will create opportunities to utilize data is a tool to create and structure datasets from a wide range of sources. Another thing that will create opportunities is a tool to perform a large number of repetitive tasks without requiring individual instruction each time: this enables searching for informative patterns within datasets. And another thing that will create opportunities for us is a tool that performs statistical calculations for us, without having to look up tables of distributions.

In this book, we make use of R, because it incorporates all the useful tools mentioned in the previous paragraph. R has excellent capabilities for bioinformatics in particular, and for data analysis more generally. Using R involves working in a command-line environment, writing instructions to the computer to tell it what tasks you want it to perform. These written instructions can be incredibly powerful, as they can be complex computer programs in themselves, but this way of working is probably unfamiliar to most of the target audience for this book. So this first chapter guides you through obtaining and setting up R (it's free and available on any operating system), and familiarizing yourself with some of the capabilities which we will come to rely on later.

2.1 Obtaining R

R is a statistical programming environment with a wide range of pre-coded functions available through downloadable packages. A large number of functions useful for statistical analysis of data are available in the pre-installed packages, and there is a great resource of packages developed and maintained by the academic community, which can be accessed through the R environment.

2.1.1 Downloading R

Being community-maintained, the R program itself is constantly being updated. Sometimes these updates will mean code that worked on a previous

version of R no longer runs as it used to. Sometimes, the contributed packages for R are updated in a way that means they will no longer be compatible with older versions of R. In practice, this is usually the reason to prompt me to update my base R environment. The home of R is CRAN¹: the **C**omprehensive **R** **A**rchive **N**etwork.

Links to download pages for different computer platforms (Linux/Mac/Windows) are available on the CRAN website, the address of which is given as a footnote to this page. In particular, the webpages with installation instructions for Mac and Windows are:

- Mac – <http://cran.r-project.org/bin/macosx/>
- Windows – <http://cran.r-project.org/bin/windows/base/>

All the code presented in this tutorial was tested for use with R version 3.5.2, and all contributed packages (from either CRAN or Bioconductor) as of November 2018. If you have problems with executing the code in these tutorials, and have exhausted mistakes in copying the code as a cause, then you may wish to obtain R version 3.5.2². You will need to install this from source, which is a little more complicated, but instructions can be found at: <https://cran.r-project.org/doc/manuals/r-release/R-admin.html>.

2.1.2 Installing R

Platform-specific instructions for installing R are provided at the following locations.

2.2 R console

Assuming you now have R successfully installed, it's time to start exploring how to use it! The interface with the R program is achieved through the text-based console. We will use the standard interface that comes with R, and which doesn't need a graphical interface. Many people find the RStudio³ environment helpful, so after you've tried a few of the chapters in this book you might want to see if the RStudio interface suits you more. RStudio still contains the main R console, so all the R code in this book applies in exactly the same way.

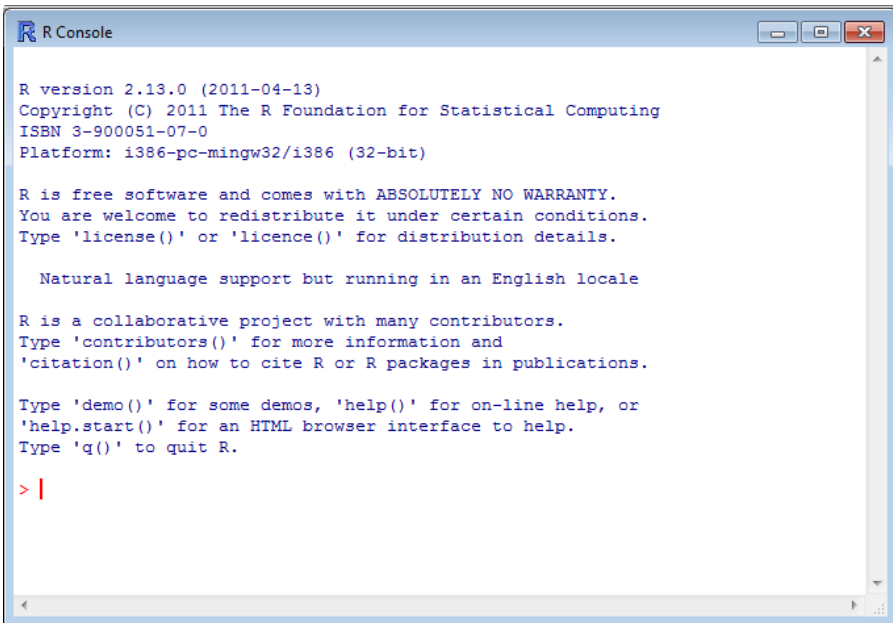
¹<http://cran.r-project.org>

²source code found at <https://cran.r-project.org/src/base/R-3/>

³<https://www.rstudio.com/products/rstudio/download/>

2.2.1 Starting the R console

This depends a bit on the platform: for a Windows machine the ‘RGui’ program should have been created during the install process; for Mac and Linux you can start R simply by entering ‘R’ at the command prompt. When the R program starts up, you’ll be presented with a command console. This is the place you can enter your commands, on the line starting with the *prompt* `>`. In a Windows environment, the R console looks like [Fig. 2.1](#). Throughout these tutorials, commands to be entered into the R console will be written in **terminal font** and will follow the command prompt `>`.



```
R Console
R version 2.13.0 (2011-04-13)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i386-pc-mingw32/i386 (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

FIGURE 2.1

The R console in Windows.

2.3 The R workspace

R is a statistical programming environment which stores all objects it can work on in its *workspace*. The general concept in using R is that you enter commands which will relate to some objects that you have specified: the command will search through the workspace to find the object(s) you specify, it will then perform some operation that uses the information contained in the specified object(s), and then provides some output. This output can be assigned to a new

object on the workspace, or it will simply be printed out to the console. To view the contents of the current workspace, use the `ls` function:

```
> ls()
```

The general structure of an R command, assuming you wish to keep the result, is made of three parts:

1. Output variable name: this is the name you wish to give the new object that is created from the output of the command.
2. Assignment operator: a left-facing arrow (made from the less-than symbol followed by the minus symbol) '`<--`' tells R that you wish to create a new object with the output of the command⁴.
3. Function call: this is the main content of the command. To call a function to be executed by R, you enter its name immediately followed by brackets⁵ (...). In between the brackets are any pieces of information that you wish the function to use: the values of these 'arguments' may be references to objects on the workspace, or they may be new objects that are created 'on the fly' (which can be something as simple as a single number).

2.3.1 Creating/deleting objects

We can therefore use this general structure of an R command to create objects to reside on the workspace. An annotated walkthrough of some simple examples is shown in [Fig. 2.2](#).

The workspace can be saved to file using the 'save.image' function, specifying the name of the file you wish to create⁶.

```
> save.image("filename.Rdata")
```

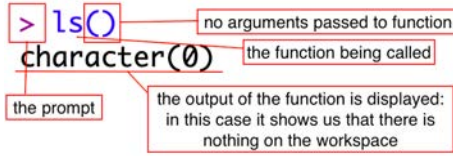
2.3.2 The working directory

In addition to manipulation of the objects in the workspace, R can also interact with files on your computer (or in fact on other computers, if it has access!). The interaction with files on the computer occurs through reading/writing to specified locations. An example of changing the working directory is shown in [Fig. 2.3](#).

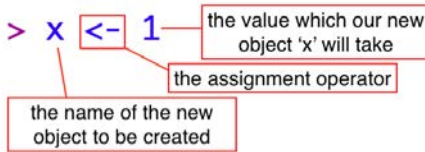
⁴It is also possible to use a single equal sign '='.

⁵The exceptions to this rule are the basic mathematical and logical operators: `+`, `-`, `/`, `*`, `>`, etc...

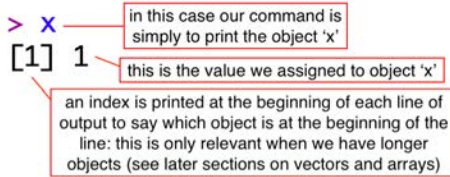
⁶When creating *any* file with R you always need to specify the extension!



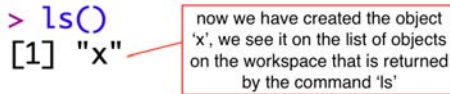
(a) List the workspace contents



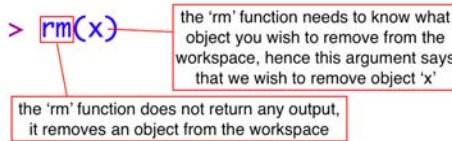
(b) Creating a new object



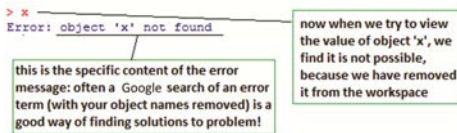
(c) Viewing the new object



(d) List the workspace contents (again)



(e) Remove an object from workspace



(f) Try to view the now non-existent object

FIGURE 2.2

Some simple examples of R commands that are creating, showing or removing objects from the R workspace.

For R to know where to find the file you want to use, the complete address of the file on your filesystem must be specified: this is known as the full or complete ‘path’ to the file. The first example given in Fig. 2.3 shows the path of a directory called ‘teaching’ on my Windows computer’s filesystem. The exception to this is the ‘working directory’: this is when R will look for any files referred to without reference to the full path. Out of context this may seem confusing, but the examples of Fig. 2.3 should clarify the difference between a full path and a direct reference to a file/folder in the working directory.

```

> setwd('/Users/ed/Documents/teaching/')
> getwd()
[1] "/Users/ed/Documents/teaching"
> setwd('TutorialsFullSet/')
> list.files()
[1] "ch1_statsIntro.pdf"
[2] "ch10_SeqGeneral.pdf"
[3] "ch11_genomeSeq.pdf"

```

Annotations:

- the 'setwd' function sets the working directory to the specified argument
- this is the full path to the folder we wish to read/write files to/from
- the 'getwd' function prints out the path to the current working directory
- you can refer directly to files/folders in the current working directory, without having to specify the full path
- the 'list.files' function prints out a list of all files in the current working directory

FIGURE 2.3

An illustration of the tools within R used to manipulate the working directory.

2.4 Data handling

The objects that can be manipulated in R come in a wide variety of types. These can be thought of in terms of basic *data types*, which can be organised into higher-order *data structures*.

2.4.1 Basic data types

- character – any combination of characters can be a ‘character’-type object, delimited by quotes ""
- numeric – a number value
- logical – TRUE or FALSE

Single objects of one of the basic data types can be organised together into a number of different structures, some of which will be described below.

2.4.2 Vectors

In the context of R, a vector is a simple one-dimensional list of objects which are all of the same basic type. To create a vector, use the ‘concatenate’ function `c()`:

```
> vec <- c(1,4,3)
> vec
[1] 1 4 3
```

Vectors of sequential integers can be created using a colon (`:`) between the initial and final numbers:

```
> vec2 <- c(2:11)
> vec2
[1] 2 3 4 5 6 7 8 9 10 11
```

One of the major utilities of creating vectors is that each object within a vector has a specified position, and so the objects can be retrieved through indexing. Indexing of vectors in R is done with square brackets.

```
> vec[1]
[1] 1
```

In the above case, we are retrieving the first object from vector we called `vec`.

To retrieve multiple elements at once, an index can in fact be a vector itself:

```
> vec2[vec]
[1] 2 5 4
```

Negative indices mean that the indexed elements won’t be returned:

```
> vec2[-1]
[1] 3 4 5 6 7 8 9 10 11
> vec2[-vec]
[1] 3 6 7 8 9 10 11
```

This becomes a remarkably powerful tool!

2.4.3 Arrays

Arrays are multi-dimensional vectors, which can be thought of as a table of objects which are all the same basic data type. Arrays are created in R

using the function `array` and filled with any specified values in order of the dimension (i.e. row first, column second, higher dimensions subsequently...):

```
> A <- array(c(1:6),dim=c(2,3))
> A
[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
```

Just like vectors, arrays are also indexed with square brackets, but with each dimension given its own index separated by a comma. To get the second column of the first row of `A`:

```
> A[1,2]
[1] 3
```

Entire rows or columns of an array may be retrieved by leaving the corresponding index empty.

```
> A[2,]
[1] 2 4 6
```

Vectors may be used to index particular elements, rows or columns of an array. The following example creates a vector ‘on the fly’ in order to retrieve the first and third columns of array `A`:

```
> A[,c(1,3)]
[,1] [,2]
[1,] 1 5
[2,] 2 6
```

2.4.4 Lists

A ‘list’ in R refers to a particular data structure, which differs from a vector in that the individual elements can be of different basic data types. In fact, the individual elements of lists may be data structures such as vectors or lists themselves. This makes lists powerful tools, but because the elements may be of different types they are more difficult to use.

```
> mylist <- list(1,2,3)
> mylist
[[1]]
[1] 1
```

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

```
[[3]]  
[1] 3
```

To index one of the elements from within a list, you need to use TWO sets of square brackets `[[index]]`.

```
> mylist[[1]] <- c(1,2,3)  
> mylist  
[[1]]  
[1] 1 2 3
```

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

```
[[3]]  
[1] 3
```

An individual element of a list in R may be a list itself, which opens up the possibility of having one object that is a list of lists of lists of lists.... For those interested, this property means that the list is what is known as a 'recursive' data structure. As an illustration:

```
> mylist[[2]] <- list(1,2,3)  
> mylist  
[[1]]  
[1] 1 2 3
```

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

```
[[3]]  
[1] 3
```

Now the second element of the list `mylist` above is itself a list. You can see that indexing of successive levels of lists are done sequentially, such that `[[2]][[1]]` refers to the first element of a list that is itself the second element of the list being indexed. The use of lists can become arbitrarily complex, although it is rare that you will need to be familiar with such use.

The final point regarding lists that may be relevant is that elements can be named when they are created. Naming an element means that it can be indexed by its name. In the example below, we create a list with three elements, the third of which is called `foo`. Referring to named elements of lists can be done with the dollar sign `$`:

```
> mylist <- list(1,2,foo=3)
> mylist
[[1]]
[1] 1 2 3

[[2]]
[1] 2

$foo
[1] 3
> mylist$foo
[1] 3
```

2.4.5 Data frames

Vectors, arrays and lists are all quite common data structures across many computer programming languages, but there is an additional data structure that is particular to R: the ‘data frame’. A data frame is a bit of a cross between an array and a list: it is essentially a list in which the elements must all be the vectors of the same length, although they can be of different basic data types. This enables the data frame to represent tables across which objects of different types may be considered related. Indexing of data frames may be done exactly as for an array (in which case a smaller data frame is retrieved), or as for a list (in which case a vector corresponding to the indexed column is retrieved).

As an example, we’ll create a data frame with two columns, one of which represents the numbers one to three in their numeric form, and the other in their textual form:

```
> mydf <- data.frame(numbers=c(1:3),names=c("one","two","three"))
> mydf
  numbers names
1 1 one
2 2 two
3 3 three
```

You’ll see from the above that R automatically gives row numbers to data frames, but these row numbers (or names, as they can be character objects too) are treated separately from the indexable columns.

The special thing about data frames is that this is the standard format in R for tables of data, as it can represent multiple attributes of a set of things (with one row per ‘thing’).

2.4.6 Data input/output

To write an object (here `mydf`) from the workspace to a file (here specified as “`mydf.txt`”) in the working directory use:

```
> write.table(mydf,file="mydf.txt",sep="\t",quote=FALSE,row.names=FALSE)
```

The argument `sep="\t"` specifies that the file should be a tab-separated table, which is a useful format as it can easily be loaded into Excel. The argument `quote=FALSE` is used because otherwise R places double quotes " around all characters in the table, and it’s generally a good policy to remove these. Finally, the `write.table` function automatically writes an additional row to the top of the table, containing the column names, and an additional column column at the left-most end of the table, containing the row names/numbers. If you wish to get rid of either of these, you have to specify `row.names=FALSE` and/or `col.names=FALSE` as appropriate. If you wish to include both column names and row names, the column names will be offset from the columns they refer to unless you specify `col.names=NA`⁷.

To read in a tab-separated table from a file and store it as an object (here called `B`), use:

```
> B <- read.table(file="mydf.txt",sep="\t",header=TRUE)
```

Again, we need to specify the `sep` argument of the function to be the tab character (which is referred to as “`\t`”). Additionally, the `header=TRUE` argument explains that the first row of the file should be used as the column headers. Now the object `B` is created on the workspace as a data frame. In fact, `B` is now exactly the same as `mydf`. We can index the new object as we would any data frame:

```
> B$names
[1] one two three
Levels: one three two
```

The last line of output above is included because when a data frame is created, character vectors are actually converted to a slightly different structure known as a *factor*. This is done because factors are like vectors that can contain objects of a mix of basic data types, and so is safer for R to use when

⁷In practice, I take the policy that if a row name is worth keeping it should probably be stored as a variable in the data frame anyway, and so typically I just specify not to include any row names.

it is reading objects from a file. It can cause problems though: the individual elements of a factor may not be treated as characters or numbers, even if they are. While the columns of a data frame may be stored as factors when read in from file, it is generally safer to convert these to the correct basic data type when using the column, for example:

```
> as.character(B$names)
[1] "one" "two" "three"
```

N.B. If converting a factor in a data frame to a numeric vector, check that this has been done correctly: it will not work properly if there are any non-numeric values in the corresponding column of the data table, although R will complete the operation and will not throw up any error messages⁸.

2.5 More advanced concepts: Scripts and functions

R can be used as a statistical data analysis program, but it is so much more: as a functional programming environment, you can combine sequences of commands together to create programs within R. This can be an incredibly powerful tool for automating tasks that would otherwise be impractically time-consuming, soul-destroyingly boring or mind-bogglingly complex!

2.5.1 Simple scripts

A sequence of commands may be loaded or pasted into the R console and run as a ‘script’. Some particularly useful functions to be used for scripts include `if` conditional statements and `for` loops. In conjunction, these can be used to search through a list or vector of elements to find all those elements for which a particular property is true.

For example, if we had a vector of characters `x` that contained the names of fruit:

```
> x <- c("apple", "orange", "banana", "mango", "pineapple")
```

And we wanted to know the positions in the vector of the element `"mango"` and store them in the vector `position`, we could run the simple script:

```
position <- c()
> for(i in 1:length(x)){
```

⁸I have personally found this to be problematic before when loading data tables from third parties, where numbers and non-numeric characters had been muddled up into the same column. The take-home message: check your data *outside* R too!

```
+ if(x[i]=="mango"){
+   position <- c(position,i)
+ }
+}
> position
[1] 4
```

An alternative approach would be to use logical indexing, which returns the vector elements with positions corresponding to `TRUE` values in the logical index vector. To illustrate the use of this technique, the same problem as above could be addressed with the single line of code:

```
> position <- c(1:length(x))[x=="mango"]
> position
[1] 4
```

In practice, the simplest way to do this would be to use the inbuilt function ‘which’:

```
> position <- which(x=="mango")
> position
[1] 4
```

2.5.2 Functions

In addition to using R’s in-built functions, or those provided through packages, you can define your own functions to perform certain operations on objects to be specified. Functions will not affect ‘global variables’ so all side-effects must be made by returning affected variables as the result of the function. Functions are treated in a similar way to variables, including their assignment and deletion.

```
> foo1 <- function(arg1,arg2){}
```

Here, `foo1` is a trivial function taking two arguments (`arg1` and `arg2`) and doing nothing. The commands specifying what the function will do are specified within the curly brackets (`{}`) following the arguments. Functions in R should only be used to manipulate objects passed to them as arguments as this ensures the correct inputs are always present. As stated above, functions in R only have an effect through the object they return.

To run a function, you enter its name and its arguments supplied inside normal brackets. For example, if we wanted to run the function `foo1` defined above, we would enter:

```
> foo1(arg1="a",arg2=2)
```

Now, in this case, it wouldn't matter what we supplied for the two named arguments of the function as it doesn't use them. But this would almost never arise, because if a function doesn't use an argument then there's no need to include it in the definition. So our function `foo1` could instead be defined:

```
> foo1 <- function(){}
```

Now we can call the function as follows:

```
> foo1()
```

And we get the output:

```
NULL
```

To demonstrate returning objects using functions, here is another example function:

```
> foo2 <- function(arg1,arg2){
> + if (arg1 > arg2) {
> + out <- arg1 }
> + else {
> + out <- arg2 }
> + out
> + }
```

This function, `foo2`, takes arguments (`arg1` and `arg2`) and returns whichever of these is greater (but if they are equal it will return `arg2`). The returned object is specified in the last line before the closing bracket (`}`). So now if we run the function to return the larger of two numbers:

```
> foo2(arg1=1,arg2=2)
```

We get this output:

```
[1] 2
```

That is, a single *numeric* object with value 2.

We could try it again, but with different numbers (note, while you don't actually need to specify which argument is which when invoking the function, it's good practice because they won't necessarily get used in the order you expect!):

```
> foo2(arg1=4,arg2=2)
[1] 4
```

2.5.3 Using ‘apply’

There is a family of functions in R: `apply`, `sapply`, `lapply` and `mapply`, which can be used to apply any other function (where appropriate) simultaneously to a large number of different inputs. We will consider each in turn, with examples to illustrate how they can be used. Given that you’re taking a lot on board in a short space of time, this is likely to seem complicated. Don’t worry, that’s completely reasonable! But the `apply` function can be extremely useful in the analysis of numerical data tables, so do try to get familiar with it and how it can be used. You shouldn’t need to worry too much about the others: they are included here because they work in a similar way.

2.5.3.1 `apply`

The `apply` function takes as its input arguments: a numeric matrix, a function to apply to each row/column of that matrix, and a ‘margin’ indicator to denote whether to apply the function to each row (`MARGIN=1`) or each column (`MARGIN=2`). Let’s illustrate this by creating a 4×5 matrix containing the numbers 1 to 20:

```
> A <- array(1:20,dim=c(4,5))
```

We could alternatively make the same matrix using the `matrix` function, but this time specifying the dimensions as separate arguments *nrow* (the number of rows) and *ncol* (the number of columns):

```
A <- matrix(1:20,nrow=4,ncol=5)
```

We can inspect the matrix A:

```
> A
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	5	9	13	17
[2,]	2	6	10	14	18
[3,]	3	7	11	15	19
[4,]	4	8	12	16	20

Note that the table’s values are filled in column-by-column. Now let’s say we simply want to add up each row and find the totals. We can use the inbuilt function `sum` and apply this across the matrix row-wise (`MARGIN=1`):

```
> apply(A,MARGIN=1,sum)
```

We get the output:

```
[1] 45 50 55 60
```

This is a numeric vector with one value for each row.

We could instead apply the `sum` function across each column, just by changing ‘`MARGIN=1`’ to ‘`MARGIN=2`’:

```
> apply(A,MARGIN=2,sum)
```

We should then get the output:

```
[1] 10 26 42 58 74
```

Using inbuilt functions (e.g. `mean`, `median`, `sum`, `product`, `sd`⁹, etc.) in this way can be extremely powerful, but `apply` even allows us to apply arbitrarily defined functions that we can create for specific purposes. Let’s say that for some reason, for each row in the table we needed to take the largest value from the first 3 columns and add it to the smaller value from the second 2 columns. We could define a new function to do this, calling its numeric vector input argument x for simplicity’s sake:

```
> newfun <- function(x){
+   max(x[1:3])+min(x[4:5])
+ }
```

Now we can use `apply` again to obtain the result of applying this function to each row of matrix A :

```
> apply(A,MARGIN=1,newfun)
[1] 22 24 26 28
```

A final cautionary note: in these simple function definitions we haven’t told R to carry out *type-checking* of the arguments. That means it doesn’t know that the argument x to the function `newfun` is supposed to be a numeric vector, nor that it needs to be of length (at least) 5. If we supply an inappropriate argument, R will attempt to evaluate the function on the arguments that are provided, only throwing an error if it comes across a function call which is constrained to certain types. For this reason, along with many others, it is always advisable to construct a test scenario for your script or program and making sure that the output is as expected.

2.5.3.2 `sapply`

Where `apply` evaluates a function using each row or column of a matrix as its input, `sapply` evaluates a function using each individual element of a vector or list. I tend to use this most when manipulating character variables. For example, let’s go back to the character vector used in Section 5.1:

```
> x <- c("apple","orange","banana","mango","pineapple")
```

⁹calculates standard deviation

We can append a set of characters to these, for example if we wanted to add ‘_juice’ to each, using the `paste` function:

```
> y <- paste(x,"juice",sep="_")
```

Here we have created a new object `y`, which is the character vector created by the `paste` inbuilt function. This takes a number of character (vector) arguments and appends them together with a separator specified by the argument `sep=` (in this case, an underscore `_`). Let’s look at this:

```
> y
[1] "apple_juice" "orange_juice" "banana_juice" "mango_juice"
[5] "pineapple_juice"
```

To remove these from each, we will make use of the `strsplit` function. This splits up a character string, based on a specified delimiter. For example, if we wanted to split the first element of `y` around the underscore, we could run:

```
> strsplit(y[1],split="_")
```

This gives us output:

```
[[1]]
[1] "apple" "juice"
```

Notice that the result returned is a list with one element, a vector containing two character strings. We could return just the first part of this by indexing the first element of the list (with double square brackets), and the first character string (with a single square bracket):

```
> strsplit(y[1],split="_")[[1]][1]
```

Now we just get:

```
[1] "apple"
```

This is the point at which we have the function we are looking for. Let’s define the function separately and call it *fruit*:

```
> fruit <- function(x){
+ strsplit(x,split="_")[[1]][1]
+ }
```

Note that the argument for the function, even through it’s called `x`, won’t get confused with our character vector called `x`.

So now we can use `sapply` to apply the function to each element of the vector `y`:

```
> sapply(y,fruit)
```

This gives us the output we are looking for, which should be the same as what we started with.

2.5.3.3 `lapply`

`lapply` is very similar to `sapply`, but the output is returned as a list. This is particularly useful when the function we are applying, or the list we are applying the function to, may have elements of varying lengths. To illustrate this use, we will start with a list of three numeric vectors of different lengths:

```
> x <- list(1,c(1,2,3),c(4,5))
```

This should give us the following:

```
> x
[[1]]
[1] 1

[[2]]
[1] 1 2 3

[[3]]
[1] 4 5
```

Say we have a function where we want to filter out all appearances of the value 2:

```
> notwos <- function(x){
+   setdiff(x,2)
+ }
```

Here we make use of the *set difference* operator, which will return the first argument but having removed all elements that match any elements of the second argument (in this case, a numeric vector with one element, the number 2).

So we can apply this function to the input list using `lapply`:

```
> lapply(x,notwos)
[[1]]
[1] 1
```

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

```
[[3]]
[1] 4 5
```

So we see that we have results that are of different lengths, and we need to represent this result as a list. As a matter of fact, the `sapply` function can deal with such situations, by running `lapply` instead. Try it for yourself, replacing `lapply` above with `sapply`. So the main reason you would want to stick to `lapply` is if you wanted to keep the resulting output as a list, even if it would be possible to keep the output as a table.

2.5.3.4 `mapply`

The `mapply` function takes multiple input arguments and applies the function using the first element of each argument, then the second, and so on. For a simple example, we will make two numeric vectors of length 3:

```
> x <- c(1:3)
> y <- c(2:4)
```

Now x has the values 1 to 3, and y has the values 2 to 4. We can define a simple function to take two input arguments and add them together:

```
> addtwo <- function(x,y){
+ x+y
}
```

Now we can use `mapply` to add the first element of x to the first element of y , then the second element of x to the second element of y , and finally the third elements of each:

```
> mapply(x=x,y=y,FUN=addtwo)
```

Note that here we have specified the function with the named argument `FUN`, and we have named each of x and y . We should see the output:

```
[1] 3 5 7
```

Now, some of you may notice that this is in fact the same way that the simple addition operator `+` handles vector arguments, and so is a rather redundant application of `mapply`. Situations that do require using `mapply` do not occur that frequently, and so this is only really included here for completeness: there is no need to become particularly familiar with this part!

2.6 Plots

One of the main advantages of using R for data analysis is its relatively simple yet powerful graphical capabilities. This is particularly useful in scientific research, where we often lean on visual representations of data to support our theoretical arguments. It is also very helpful when working with large volumes of data, where it is rarely feasible to develop an intuitive understanding of dataset by looking at vast tables of numbers.

2.6.1 Simple scatterplot

The command `plot()` will plot the values of an input vector in sequence, automatically scaling the axes. The values of each element in the vector will be shown on the y-axis, with the x-axis values coming from the positions of each element in the vector. If we create a vector and then plot it as follows, the resulting figure should be created as in [Fig. 2.4](#):

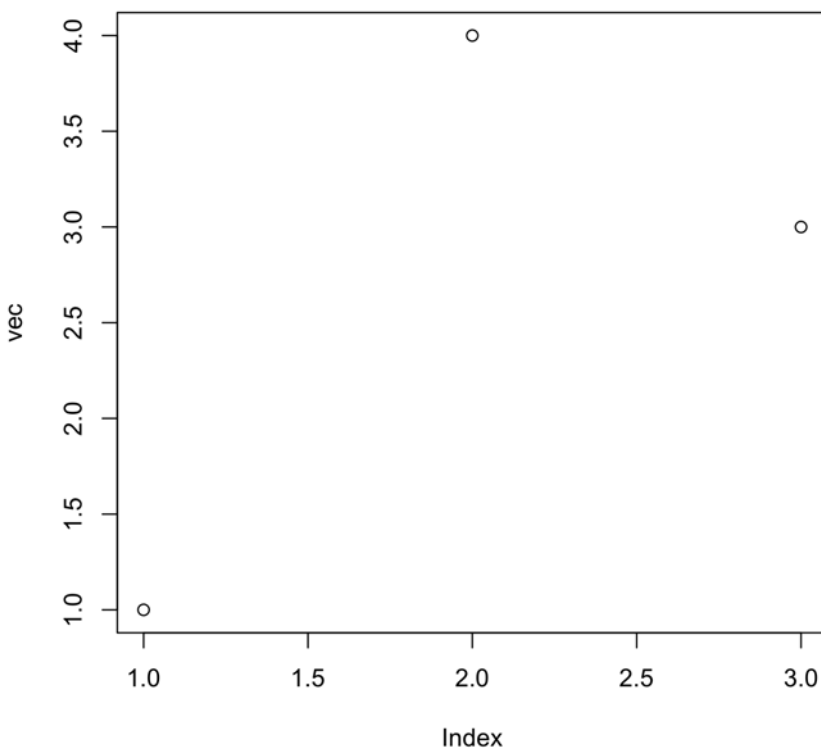


FIGURE 2.4

Illustration of the most basic of the plotting capabilities of R.

```
> vec <- c(1,4,3)
> plot(vec)
```

2.6.2 Arguments of `plot()`

There are many arguments that can be specified for the `plot()` function. Commonly useful examples are provided below:

- `type="l"`: plots a line, connecting each point
- `col="red"`: plots the points in red. Other colours may be specified (e.g. “green”, “blue” etc.)
- `xlim=c(0,10)`: sets the x-axis to go from 0 to 10
- `ylim=c(0,10)`: as above, but for the y-axis

2.6.3 Multiple plots on one graph

If there is a plot already drawn, the function `points()` can be used in place of `plot()` in order to draw subsequent plots on the same graph.

```
> plot(vec2,type="l",ylim=c(1,11))
> points(vec,type="l",col="red")
```

2.6.4 Scatterplots of multiple variables

In the above examples, the vectors are being shown on the y-axis of a scatter plotting the values of the elements in the vector against their corresponding position indices (shown on the x-axis). We can use the same `plot` function to plot two vectors against each other. The result of the following example is shown in [Fig. 2.5](#), which plots the elements of `vec` squared on the y-axis against the corresponding elements of `vec` on the x-axis. It should be noted that for such plots, the two vectors must be of the same length.

```
> plot(x=vec,y=vec^2)
```

In fact, the earlier examples can all be thought of as plotting the vector as `y` with `x` set to the sequence of integers from 1 to the length of the plotted vector.

2.6.5 Box plots

Creating box plots in R is relatively simple, although it is based around the list data structure. By using lists, each element of the list can be represented

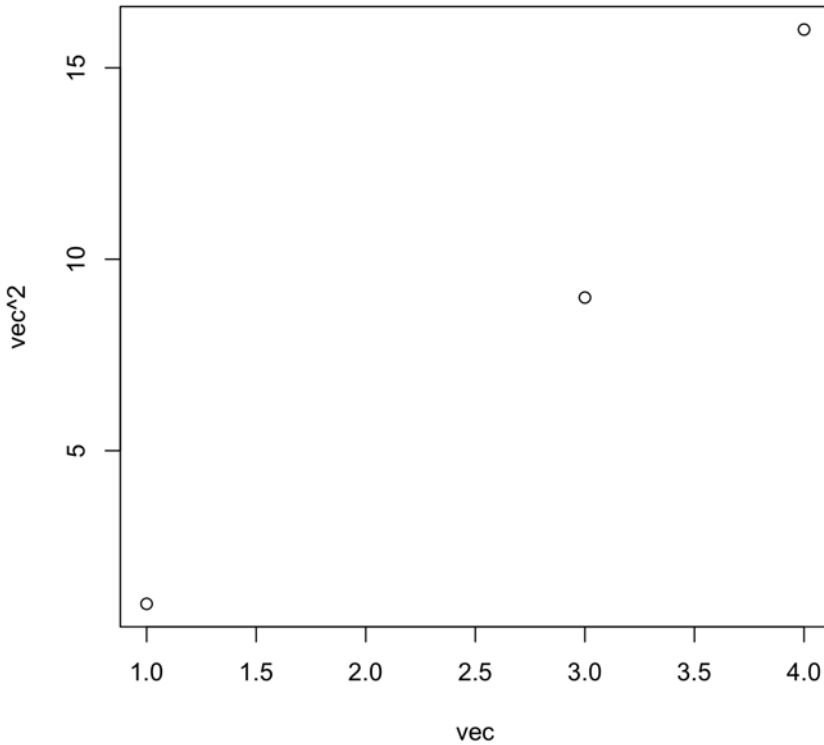
**FIGURE 2.5**

Illustration of basic two-variable scatter plotting capabilities of R.

by its own box, and thus doesn't need to be the same length. For example, if we create a new list called `mylist2` as follows:

```
> mylist2 <- list(c(1:3),2,c(1:5))
[[1]]
[1] 1 2 3
[[2]]
[1] 2
[[3]]
[1] 1 2 3 4 5
```

Now we can easily create a boxplot of these elements, which should appear as [Fig. 2.6](#):

```
> boxplot(mylist2)
```

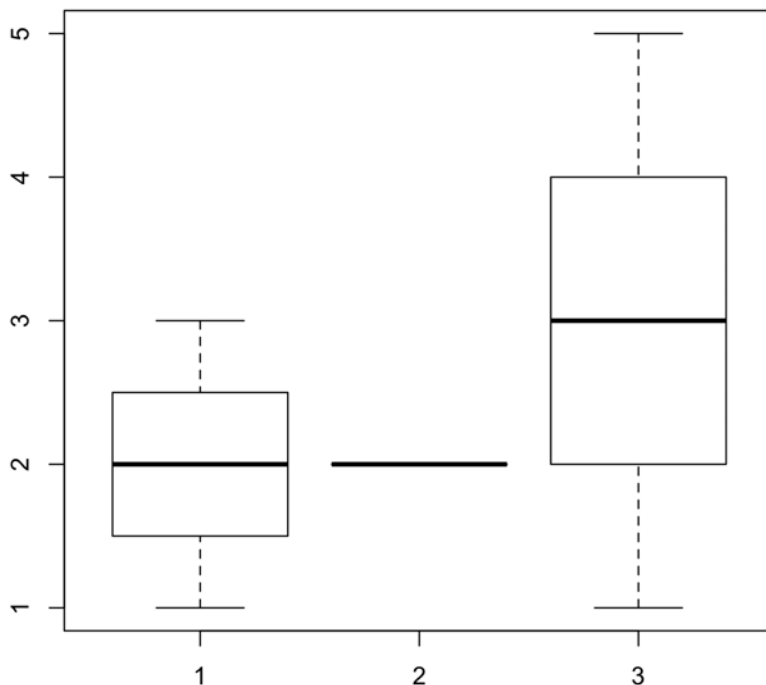
**FIGURE 2.6**

Illustration of a simple boxplot in R.

2.6.6 Saving images to file

The default graphical device on which to draw the output of any graphing function is a window that will open on the user's display. There is a set of functions in R that can create a *file* in which to draw graphical outputs: `png()`, `jpeg()`, `bitmap()`. These functions *open* a file of the corresponding type, and once the graphing commands have been completed, the file must be closed using the separate function `dev.off()`. For example, we could save the result of Fig. 2.6 to a *png* file called 'MyBoxplot.png' with the following set of commands:

```
> png(file='MyBoxplot.png')
> boxplot(mylist2)
> dev.off()
```

2.7 More advanced graphics with ggplot2

The graphical capabilities presented in the previous section are very useful, but the appearance is somewhat functional. A very nice set of graphing tools

has been developed for R, called *ggplot2*. The manual can be found at: <https://ggplot2.tidyverse.org>.

The *ggplot2* package is based on the concept of a ‘grammar of graphics’, a formal framework to use for data visualization. In R, this comes in the form of using the `ggplot` function to set up a graph using a *data frame* as input, and specifying *aesthetics*: mappings of columns from the data frame to attributes of the graph. Then additional *layers* can be added, which determine how the data frame is represented. For ideas of how the *ggplot2* package can be used to visualize data, browse the reference section of the manual (especially the *geom_* functions): <https://ggplot2.tidyverse.org/reference/index.html>.

We will illustrate one relatively simple example, which will create a plot that would take a lot more effort using the basic R graphing capabilities. First though, we need to install the package:

```
> install.packages('ggplot2')
```

Now we’ll make a data frame with a numerical value, let’s say this is the viability of a cell line, treated with a drug in two different cell culture conditions. Because we will draw the viability values randomly, we are setting the seed of the random number generator to a specific value: this works as a ‘cheat’ to make sure the random numbers that you create are the same as the ones I created. So the graphs should look the same.

```
> set.seed(10)
```

This is how to use the `set.seed` function, which you’ll not is a rare function in R in that it doesn’t create an output directly. Instead, it alters the internal state of the R workspace. Now we can generate some (pseudo-)random numbers to draw:

```
> viability <- rnorm(40)
```

We have created a vector called `viability`, with 40 values drawn from a standard normal distribution, using the `rnorm` function.

```
> treatment <- rep(c('control','treated'),20)
```

Now we have created a vector called `treatment`, repeating the two character strings ‘control’ and ‘treated’ 20 times. Note, we have one character for each viability value.

```
> culture <- rep(c('media1','media2'),each=20)
```

This command creates the final information we need, a vector called `culture` that repeats the character string ‘`media1`’ 20 times, then the character string ‘`media2`’ 20 times. We will combine these together into a data frame:

```
> plotdf <- data.frame(viability=viability,
+ treatment=treatment,culture=culture)
```

We have used the function `data.frame` to create a data frame with three named columns. This is only possible because the three vectors we used for the columns are all the same length. You can inspect the first few rows of the data frame `plotdf` using the `head` function:

```
> head(plotdf)
viability treatment culture
1 0.01874617 control media1
2 -0.18425254 treated media1
3 -1.37133055 control media1
4 -0.59916772 treated media1
5 0.29454513 control media1
6 0.38979430 treated media1
```

Now we can use the *ggplot2* package to create a set of boxplots, drawing the viability for each treatment, for each culture medium:

```
> library(ggplot2)
```

First, load the *ggplot2* package.

```
> ggplot(plotdf,aes(x=culture,y=viability,
+ fill=treatment)) +
+ geom_boxplot() +
+ geom_point(position=position_jitterdodge())
```

This has used the `ggplot` function with the first argument being the data frame to use, and the second argument the output of a function called `aes`. This function maps columns of the input data frame to characteristics of the plot: in this case, the value of the ‘`culture`’ column gives the x-axis position; the value of the ‘`viability`’ column gives the y-axis position; the value of the ‘`treatment`’ column gives the colour to fill the boxplots, and also will be used to resolve the x-axis position. The result should look like that in [Fig. 2.7](#). Hopefully this illustrates how powerful the *ggplot2* package can be. But it can be a bit complex, so we’ll mostly use the basic graphical capabilities in the examples throughout this book, to focus on the data analysis.

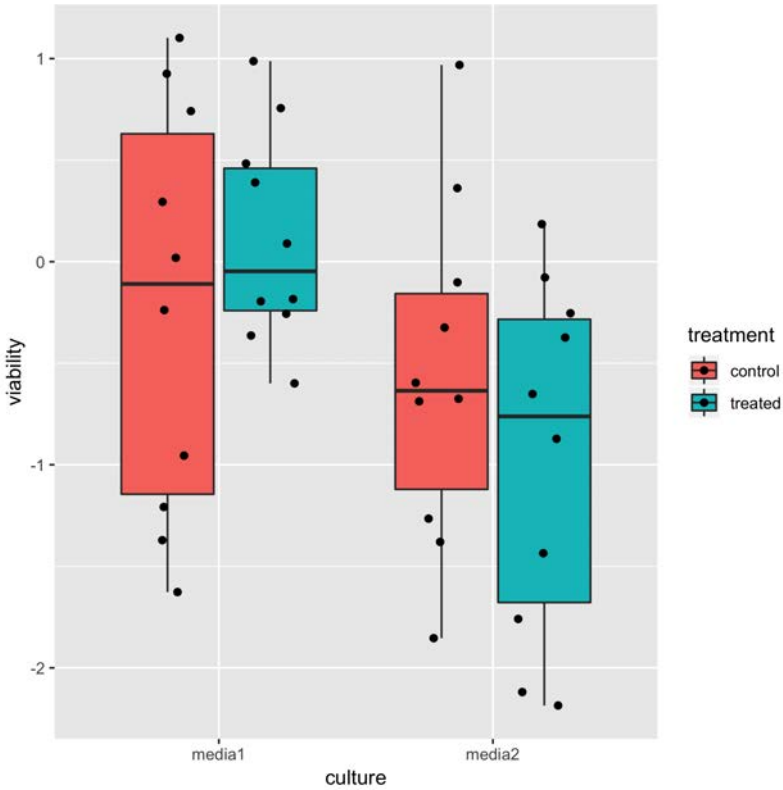
**FIGURE 2.7**

Illustration of a more complete boxplot using the *ggplot2* package.

2.8 Using R help

The R manual contains detailed instructions on any function provided in R. The entry for any particular function can be obtained through the R console simply by entering a question mark ? immediately followed by the function name. For example, if I wanted to learn more about the histogram function `hist` I would enter:

```
> ?hist
```

Additionally, a more detailed R tutorial is provided by CRAN at <http://cran.r-project.org/doc/manuals/R-intro.html>. This is a very useful resource for exploring in greater depth the topics introduced here, although given its detail it may be slightly harder to follow.

3

An Introduction to LINUX for Biological Research

Hopefully after the introduction to working with R, you should now be a little more comfortable with a command-line interface. You may be more used to working with a graphical environment in Windows or MacOS, or perhaps a Linux distribution, but there are some data processing tasks you may encounter in biological research for which it really helps to use the command-line interface of a Linux environment. If you don't find yourself in a situation where you need this, you might prefer to skip onto the next chapter for now. You can always come back to this short reference chapter when you need it.

3.1 UNIX

Technological advances have resulted in computer hardware getting cheaper at a rate that means the computational power available for a given cost has increased exponentially over most of the last century. In this context, considering that next-generation sequencing platforms first found mainstream use around 10 years ago, a computer that would be capable of analyzing the dataset you have just received from your sequencing facility might be on your desk now but in the early days of next-gen sequencing it would only have been available through a high-performance computing facility. Many of the software tools that have been created for analysis of sequencing data were therefore developed for high-performance computing machines, which tend to be set up on UNIX-based operating systems that scale up very well. The downside of this scalability is that these operating systems tend not to be as easy to use as those developed for the era of personal computing, like Windows and MacOS¹. The most likely situation is that you will be able to access a computer running a Linux distribution, you might even install Linux on your own computer. Linux is an open-source operating system that has been developed by different communities into different distributions (e.g. Ubuntu, Debian, RedHat, CentOS, Gentoo, etc.). For what it's worth, I currently run

¹Mac OSs are actually based on UNIX too, but they are best known for their graphical windowing system.

two CentOS computers and one RedHat computer which I use myself and run as servers for other researchers in my department to use. I have happily used Ubuntu in the past. If you have a powerful computer running Windows and plenty of spare disk space, you may wish to create a Linux partition for that computer. An alternative if you don't want to 'dual-boot' (i.e. have a computer that can choose between two different operating systems) is Cygwin², which is a program that creates an environment providing a lot of Linux functionality and can run through Windows. Regardless of what system you have access to, the fact that you will now be familiar with command-line-based computing (thanks to our use of R) will stand you in good stead for running commands on a Linux system. These systems will have a 'shell', which will typically open as you log in. If not, you can start the shell by opening a *terminal*. This is your interface with the computer, in which you can view and enter text, navigate through the file system by changing your working directory much as you do in R, and through which you enter commands to run. The most common shell environment is the 'Bourne Again SHell' (*bash*). There is a great tutorial at <http://www.tldp.org/LDP/Bash-Beginners-Guide/html/>.

3.2 Linux survival guide

For working in a Linux environment, knowledge of a few shell commands is absolutely essential. You will find it incredibly useful to develop some more advanced use of the shell as a scripting/programming environment, but for now let's stick with the essentials. As mentioned in the previous section, the shell is your text-based interface to the operating system, and the Linux-based functionality of your computer. Throughout these tutorials, we will show *bash* commands with the dollar-sign prompt `$` in much the same way as we show R commands with the `>` prompt. I would say that the things you will *need* to do in the course of analyzing sequence data will be:

- get information about a bash command with `man`
- navigate through the filesystem by changing directories with `cd`
- create new directories on the filesystem with `mkdir`
- move/rename files with `mv`
- copy files with `cp`
- list files in the current working directory with `ls`
- download files from the internet with `wget`
- extract *.zip* files with `unzip`

²<http://www.cygwin.com/>

- extract `.gz` files with `gunzip`
- extract `.tar` files with `tar`
- write the output of a command to a new file with `>`

Additional utilities I think you will find the most helpful to use:

- viewing text files with `less` or `more`
- displaying the current working directory with `pwd`
- counting the number of words or lines in a file with `wc`
- extracting only one column from a text file with `cut`
- displaying only lines in a text file that contain some specified pattern with `grep`
- passing the output of a command directly to another command with the 'pipe' character `|`
- running Java executables (`.jar` files) with `java`
- running Python scripts (`.py` files) with `python`
- running Perl scripts (`.pl` files) with `perl`

In the rest of this section of the tutorials, we will go through a series of example tasks that show us how we use these commands to achieve the things we will probably need to do if we are to analyze sequencing data on a Linux machine. The first thing to say is that you can find information about how to use any bash command by entering `man` followed by the command in question. For example, if I wished to find out about the `man` function itself, I would enter:

```
$ man man
```

The output from this would be:

```
man(1)
```

```
NAME
```

```
man - format and display the on-line manual pages
```

```
SYNOPSIS
```

```
man [-acdDfFhkKtwW] [--path] [-m system] [-p string] [-C  
config_file] [-M pathlist] [-P pager] [-B browser] [-H htmlpager]  
[-S section_list] [section] name ...
```

And then a description of what the `man` command does and how it is used.

The long line following ‘SYNOPSIS’ is the instruction of how to use the command: you enter the command itself (in this case `man`) followed by any of the possible options which may change how the command executes, then finally the argument ‘name’, which it describes as the name of the manual page you wish to display. All the square brackets [...] indicate that what’s inside is optional, and so in this case you can just use the command with two words as we did in our example. You will see that options are generally preceded with a dash -, and some of them tell the command how to use the bit that follows (for example, ‘-B `firefox`’ would tell the `man` command to use *Firefox* as the program to open up HTML manual files).

Navigating through the filesystem with only a text interface perhaps takes a little getting used to, but you’ll soon find it seems natural. The `cd` function is pretty simple to use, in that you enter ‘`cd`’ and then the directory you wish to move into (this obviously is based around the concept that the shell and programs invoked from it read and write files in the current working directory, unless explicitly specified otherwise). To set up a sensible example, we will also use the command `pwd` to tell us what directory we are currently in and the command `ls` to show what files there are in this directory. Let’s say I have just logged into my system and want to see what directory the shell has started in:

```
$ pwd
/home/ed
```

The output here says that shell started in a directory called ‘`ed`’ within the directory ‘`home`’. I can then look at what files are in this directory by simply entering ‘`ls`’:

```
$ ls
Desktop Downloads work
```

As a trivialized example, this shows me three files which are all directories. If I wish to enter the directory ‘`work`’, I enter:

```
$ cd work
```

Now when I ask what directory I am in, it tells me that I’m in the ‘`work`’ directory inside ‘`ed`’ inside ‘`home`’:

```
$ pwd
/home/ed/work
```

The output of the `pwd` command is known as a *path*, and is effectively the address of the file on the filesystem. There are a couple of special paths: .

means the current directory and `..` means the directory containing the current directory. Therefore we can navigate up through the filesystem hierarchy³ by entering:

```
$ cd ..
```

If we are not sure what directory we are starting (or more likely we are writing a script and don't know where we may be starting from in the future), then we could tell the shell to go straight to the 'work' directory by specifying the full (*absolute*) path:

```
$ cd /home/ed/work
```

Now let's say we wanted to create a new directory called 'testing'. We can do this using the `mkdir` command:

```
$ mkdir testing
```

If we list the files in the (previously-empty) 'work' directory, we see:

```
$ ls
testing
```

As an example that will help illustrate a number of other tasks at once, we will download a zipped tar archive (i.e. a compressed directory containing files we wish to use on our computer) from an internet address. Let's say that we want to use the Samtools set of programs for manipulating sequence data files. We first need to retrieve the file from the internet. Many files hosted on projects such as SourceForge or GitHub are easiest to retrieve through a standard browser (e.g. Firefox or Chrome), downloading the file either directly onto the Linux computer or to a local computer then transferring the file via an *scp* client (such as WinSCP or FileZilla). The Samtools download page is at <http://sourceforge.net/projects/samtools/files/samtools/1.2/samtools-1.2.tar.bz2>. Your Linux computer may have a text-based browser such as *ELinks* installed, which can be invoked from the shell as follows:

```
$ elinks http://sourceforge.net/projects/samtools/files/latest/download
```

However you have downloaded it, let's say we have the file 'samtools-1.2.tar.bz2' on the filesystem of our Linux machine. First we need to decompress the zipped file with *bzip*:

```
$ bunzip2 samtools-1.2.tar.bz2
```

³Note that when you are already in the root directory `/` then you cannot go any higher.

Now if we list the files in our current directory that start ‘samtools,’ we see that there is a file ‘samtools-1.2.tar’:

```
$ ls samtools*
samtools-1.2.tar
```

We extract the *tar* archive as follows:

```
$ tar -xf samtools-1.2.tar
```

Now the files are all extracted to a newly-created directory ‘samtools-1.2’. We can enter this directory and look at the files contained therein:

```
$ cd samtools-1.2
$ ls
```

If an archive such as this contains a file called ‘README’ then as a general rule it is worth reading before trying anything else. This can be inspected by invoking the bash program *more*:

```
$ more README
```

This tells us we need to see the file ‘INSTALL’ for building and installation instructions. So we need to open that file (this time we’ll use the bash program *less*):

```
$ less INSTALL
```

A bit more complicated, this file tells us we need to *compile* samtools. This is because the samtools kit is provided as a set of *source code* files. These are computer code written in a programming language (in this case **C**), which need to be converted into executable instructions the computer can use directly. We see from the instructions that installation can be carried out with these simple steps:

```
$ make
$ make install
```

Note, these steps require a **C** compiler (e.g. *gcc*) to be installed on the system, and may require administrator privileges on the machine. There may be additional dependencies (programs on which the thing you’re trying to install rely), and unfortunately these will vary depending on your exact setup. I find a lot of headaches can be resolved with an internet search of the exact error message thrown!

Assuming the install was successful, we can now invoke samtools from bash:

```
$ samtools
```

(Of course, without supplying any input, the program will just print out its usage instructions.)

It is important to remember that for many programs you can invoke them via shell commands in any directory, provided you specify the absolute path to the executable file for the program you are running. When this happens, most programs will by default write their output to the directory from which you called the shell command, not the directory in which the executable file is kept. There are a host of tools accessible through the shell that can enable creation of very powerful scripts that will carry out a great deal of work in an automated fashion, but such shell scripting tools would require a detailed course to themselves, and that is outside the scope of our learning here. The TLDP bash tutorial⁴ should be a reference guide to assist with queries you may have, but at the very least you should now be able to install and run the sequence data analysis tools referred to in the rest of this tutorial.

3.3 Useful dependencies and programs

Sir Isaac Newton was not the only one to stand on the shoulder of giants. As a central practice of software engineering, it means there exists a great wealth of scientific programming packages which have been created to make other programming tasks easier. Many of the tools that you will use for processing sequence data will make use of some of these programs or ‘libraries’ and while I can’t give an exhaustive list, the following are sufficiently frequently used that it will be handy to have them installed⁵ (either by yourself or by your sysadmin):

- gcc – C is one of the most widely-used programming languages in the world. The Gnu C Compiler, and in particular its *developer* resources are needed to configure, make and install many programs written in C.
- Java – Java is another widely-used programming languages. A difference between Java and C is that most Java programs won’t actually need to be separately installed on your computer but can be downloaded as a `jar` file which can be run directly through Java.

⁴<http://www.tldp.org/LDP/Bash-Beginners-Guide/html/>

⁵The process of installing these will depend very much on your computer.

- Python – Python is a relatively recently developed programming language which has really taken off in popularity. A number of HTS programs use Python. There are two main releases of Python currently in use, Python 2.7 and Python 3. If you are using a Python-dependent program, do ensure you have the required version installed. Along with Python, the numerical programming library *NumPy* and scientific programming library *SciPy* are also worth installing.
- Samtools – Samtools, BCFtools and HTSlib are available from <http://www.htslib.org/>, and Samtools in particular is very useful (if not essential) for working with sequencing data.
- BEDtools – BEDtools contains a lot of functionality that is especially useful for ChIP-seq studies and their results. It can be obtained from <https://github.com/arq5x/bedtools2> and the documentation at <http://bedtools.readthedocs.org/> is extensive.

4

Statistical Methods for Data Analysis

In this chapter we will try to get a good understanding of some analytical tools that will empower you to use datasets to answer real-world questions. You will see how some of the programming tricks can be used with R's built-in number-handling functions to put the theory into a practical context.

We start with an illustration of why we use statistical methods in research, to introduce the theory of probabilities, then describe some methods that use probabilities to help us attach confidence to observations in data.

4.1 What are statistical methods, and why do we use them in biological research?

In applying statistical methods to data analysis, we are typically aiming to *describe* some properties of the data, or we wish to *infer* properties of what the data represent. I find the distinction between these aims quite revealing about why statistical methods are important in biological research. It should be intuitive that descriptive statistics are useful: these are the means of characterizing or summarizing properties of the data we have collected. But assuming we undertake research according to a traditional scientific method, we can advance knowledge through *testing* hypotheses: we use data to help us determine whether a hypothesis is supported by observation or should be rejected.

Because it is rarely possible to obtain perfectly accurate measurements, nor to have complete control over all potential sources of variation in the systems being measured, we have to consider the fact that if we obtained more data, it probably wouldn't be the same as the data we already have. The measurements we have are just representatives (this is a *sample*) of all such measurements that could have been obtained, if we had repeated the experiment infinitely (this is a *population*). We use statistical inference to estimate properties of that infinite set of all measurements that could have been obtained, by modelling these uncontrolled variations in the data as *randomness*. In expanding on this concept, the following paragraph may seem complicated, but it is probably the most important part of this book....

Say we obtained three repeat measurements of a gene's expression in two experimental conditions (e.g. a transformed cell line and a control cell line), we could use descriptive statistics to show that the mean measurements from the two experimental conditions were different. This observation from the samples we have obtained could lead us to a hypothesis regarding the populations they represent: do the measurements we have obtained suggest that the means of the two populations are different? We are now thinking about what we'd find if we had repeated the experiment an infinite number of times, which acknowledges the fact that the data we have includes some uncontrolled variation. Typically we evaluate the extent to which the data supports a hypothesis by considering the opposite of our hypothesis: in this case, that the means of the two populations are the same. This is termed the *null hypothesis*. Using our example, the principle of statistical hypothesis testing can be outlined as follows:

- Use the properties of the samples to infer the *distribution* of values that the populations would take if the null hypothesis is true (in this case, that the means were the same).
- Work out the probability that any two samples of three measurements (one from each population) would have as big a difference in their means as the measurements we obtained from our experiment.
- Based on this probability, does our experimental data provide sufficient evidence to reject the null hypothesis?

Most readers of this book will be familiar with this procedure, as it's what you've done if you've ever applied a t-test. When doing this, were you aware that the p-value from the t-test is the probability that any two samples drawn from normally-distributed populations with the same mean and with the variances of the observed samples would have as big a difference in their means as the observed samples? If not, does knowing that now change your interpretation of t-test p-values?

4.1.1 A worked example

Assuming we are now at least a little familiar with R, we can use some simple programming to illustrate this theory. We can try sampling two sets of three measurements from the same normally-distributed population and compare their means. For this, we will use the `rnorm` function which draws random numbers from a specified normal distribution. Just before we try this, we can use a bit of a cheat to make sure the random numbers you create are the same as the ones I create, by using a function called `set.seed` to specify a state for R's random number generator:

```
> set.seed(10)
> a <- rnorm(3,mean=1,sd=1)
> b <- rnorm(3,mean=1,sd=1)
```

These two lines of code each create an object (one called **a**, one called **b**), each storing a *vector* of three numbers.

Let's look at the means of each set of three numbers:

```
> mean(a)
[1] 0.487721
> mean(b)
[1] 1.028391
```

Here we have used the function *mean* to compute the mean of a vector of numbers. Are these means the same? Are they significantly different? What do we even mean by a statistically significant difference? We actually know that the population from which each sample was drawn is the same, so if the means of the samples are estimates of the means of the populations they were drawn from, they should be the same. But they won't be exactly the same, because we have randomly drawn only three numbers from these distributions. This random variation is what we are trying to quantify using statistical theory. Let's consider what happens if we draw samples of three numbers from two *different* populations (represented by normal distributions with different means, but the same standard deviation)?

```
> x <- rnorm(3,mean=1,sd=1)
> y <- rnorm(3,mean=2,sd=1)
```

Let's look at the means of each set of three numbers:

```
> mean(x)
[1] -0.06614162
> mean(y)
[1] 2.533694
```

What is the difference between them?

```
> mean(x) - mean(y)
[1] -2.599836
```

Are these now significantly different? We can get into the meaning of statistical significance by making use of the capabilities of programming computers for data analysis. Let's say we would consider the observed difference in the means between our two sets of three numbers significant if it would be a 1/100 event if the two population means were the same. So, let's repeat our comparison of three numbers drawn from the same normally distributed population, say, 100 times:

```
> meanDiffs <- c()
```

Here we have made an empty vector, to which we will add our results. We will use a *for* loop to run the same sequence of commands (between the brackets { and }) a specified number of times (in this case 100 times).

```
> for(i in 1:100){
+ x <- rnorm(3,mean=1,sd=1)
+ y <- rnorm(3,mean=1,sd=1)
+ meanDiffs <- c(meanDiffs,
+ abs(mean(x)-mean(y)))}
```

Let's consider this line-by-line: the first line establishes that there is going to be a loop, in which an object called *i* is given each value from 1 to 100; each time, a vector called *x* is created from three numbers randomly drawn from a normal distribution with mean=1 and standard deviation=1; each time, another vector called *y* is created the same way; the vector *meanDiffs* is reassigned a new value, which is the old vector *meanDiffs* concatenated¹ to the result of a sum: the absolute value of the difference between the mean of *x* and the mean of *y*. So at the end of this, we should have a vector called *meanDiffs* that has 100 numbers in it. How many of those are at least as big (in magnitude, we will ignore the sign) as the one we originally observed between the two variables sampled from different populations?

```
> sum(meanDiffs>=2.599836)
[1] 1
```

We use the *sum* function because the greater-than-or-equal-to operator *>=* will return a vector of TRUE or FALSE values, one for each element of *meanDiffs*, and a sum of TRUE/FALSE values gives the number of TRUE values. So the difference we observed would have been a 1 in 100 event if the populations that our samples *x* and *y* came from had the same mean: this is quite unlikely, so we could feel fairly confident in assuming that they had come from different distributions.

This was quite a crude example of statistical hypothesis testing. In real applications, most people will assume the observed samples were drawn from normally distributed populations, compute the statistic *t*, and then check the value against a *theoretical* distribution (e.g. how many values would be as big as our observed one if we had an infinite *for* loop) to estimate the *p-value*. Or people would use a computer to perform this calculation. We will discuss *distributions* more in this chapter.

¹To concatenate means to stick together.

4.1.2 A brief summary

You may be wondering why so much discussion of statistics in an introductory book on bioinformatics. It's partly because we use bioinformatics to help us deal with large amounts of data, and it typically gets more difficult to control all sources of variation the larger a dataset gets. And it's partly because bioinformatics enables us to test large numbers of different hypotheses and find which are best supported by the available data. By understanding how these statistical analysis methods work, you will be able to think of ways to apply them to collections of data in ways which will provide evidence for answers to previously unanswered questions. By linking a biological question to a set of data with a statistical analysis method, you will advance your field of research.

4.2 What do I need to understand statistics?

Two basic areas of mathematics underpin a lot of statistical analysis of biological data. As these really are fundamental to understanding how the current knowledge of molecular biology has been attained, they are worth serious attention. Some mathematical expressions may use funny-looking symbols, but if the symbols are appropriately defined (as they always should be) then the expression should make logical sense. In fact, you may come to appreciate the convenience of having so many unambiguous definitions! So please don't be put off by any equations, and please do put the effort into working out what they all mean. Students invariably say that one of the most valuable things they learn from my courses is finding methods presented in the scientific literature more accessible to understand and to use.

4.2.1 Probability

This section may seem quite abstract, but it is crucial in helping you understand exactly what the results mean for all the statistical tests you will apply in this work. As mentioned in the previous section, one of the principal motivations for bioinformatics is enabling computational applications of statistical hypothesis tests and evaluations of different models fitted to large datasets. Being able to apply statistical methods is useful but can have serious adverse consequences if you don't truly understand what the results represent.

4.2.1.1 Random variables

You may have seen in the example of the t-test that the result and its interpretation refer to *probability*. In my opinion, you can't understand much statistics unless you have a basic grasp of probability theory. Probability theory involves

the study of *random variables*. Randomness is one of those concepts which is quite often misunderstood: just because there is randomness in some set of observations, it doesn't mean the observations lack any patterns. Nor does it mean that observations of a random variable aren't predictable in any way. As described in the introduction of this chapter, incorporating randomness in analysis of data is the way to account for uncontrolled sources of variation affecting the observations.

As a loose definition, a random variable refers to some quantity that can take certain values and individual observations of its values can be made. More formally, this means the random variable is a mathematical *function* that produces measurable outputs (these will be our *samples*) from the set of all possible values. So to define a random variable, one needs to define the values it may take. To give an example, if a question that we may ask people is 'Have you seen the film Jurassic Park? (answer yes or no)' then we can consider the responses to be a random variable (R.V.) X , taking values **yes** or **no**. To use notation to define this, we would write:

$$X \in \{yes, no\}. \quad (4.1)$$

In this equation, the \in symbol means that the RV's values are taken from the *set* defined on its right-hand side. The curly brackets $\{\dots\}$ defines a set through listing its elements. We may additionally define properties of the RV's values, such as if it is more likely that observations of the RV will have certain values. It may also be possible to define conditions on when the RV may be more likely to take certain values. This variable can be represented in R as an object of the *logical* data type.

To describe a specific sample (let's call it \mathbf{x}) from the random variable X , we may define the set of observed values. We would again define our variable X , now stating that it could be observed n times, and each time its value would be either **yes** or **no**:

$$\mathbf{x} = \{x_1, x_2, \dots, x_n\}. \forall i \in \{1, \dots, n\}, x_i \in \{yes, no\}. \quad (4.2)$$

In Equation 4.2, the \forall symbol means *for all* values of i in the set $\{1, \dots, n\}$, the following condition is met. As we haven't yet defined n , this is not any more informative than the definition given in Equation 4.1. We have used the under-scored numbers (or letters) to refer to elements in the set, which as defined here are ordered. x_1 is the 1st observation of X , x_2 is the 2nd, and x_i is the i th. Why do this? It enables us to give more explicit conditions on the variables we are defining: for example, one can define sequences of observations.

Now, this sample \mathbf{x} could be represented in the R environment by a vector of 10 elements, each taking a logical value. Let's try this, making use of R's

pseudo-random number generation again. First, we'll set the random number generation seed:

```
> set.seed(10)
```

Next, we can generate a vector of logical (True/False) values by using the fact that a comparison of two numbers results in a *logical* value. Rather like the `rnorm` function, the `runif` function draws random numbers from a uniform distribution: this just means that any value within the specified range (typically from 0 to 1) is as likely as each other. So if we draw 10 random numbers from the range (0, 1), each will have an equal chance of being less than or greater than 0.5. Putting this together, if we draw 10 random numbers from this uniform distribution and test each one in turn for being greater than 0.5, we will have a vector of 10 observations from the population of our theoretical random variable X :

```
> x <- runif(10)>0.5
```

4.2.1.2 Probability distributions

Having only defined the values the random variable can take, it isn't particularly helpful yet. We don't know whether a given person is more likely to answer 'yes' or 'no'. We don't really know how the random variable, or the property it is modelling, *behaves*. Random variables are useful because they give us a way of *modelling* events. To make the model useful, we can define the behaviour of the random variable. This is done through defining the *probability* of the RV taking certain values, potentially under certain constraints or conditions. If there were a 50% chance of any given person answering 'yes' or 'no' to the question of having seen Jurassic Park, this would correspond to the example we just created in the R environment. In mathematical notation, this random variable would be described as follows:

For random variable X ,

$$Pr(X = \text{yes}) = 0.5 \tag{4.3}$$

$$Pr(X = \text{no}) = 0.5.$$

In this equation, $Pr(\dots)$ denotes the probability of some condition being met. For example, the probability that the random variable X takes the value *yes*. The distribution of any random variable is defined through a function, known as the *probability distribution function*, which gives a formula to calculate the probability of observing a given value, for all possible values the random variable could take. In the example above, there are only two possible values that the RV X could take, so the *probability distribution function* is fairly easy to define. This RV X is actually an example of a well-known and well-defined probability distribution, known as the *Bernoulli* distribution. A Bernoulli distribution is a random variable with two possible values, so that the probability of the second value is 1 minus the probability of the first. Why?

Probabilities define our understanding of how likely some outcome is to be observed. A probability of 1 represents complete certainty that the given outcome will be observed, and a probability of 0 represents complete certainty that it won't. For any given event, the sum of the probabilities of all possible outcomes must necessarily equal 1. This is because we have complete certainty that one of the possible outcomes will be observed.

So the random variable X follows a Bernoulli distribution, which we could define using the probability of a 'yes' (which we will denote p):

$$X \sim \text{Bernoulli}(p). \quad (4.4)$$

The notation \sim denotes that a random variable follows some given distribution. As there are many convenient probability distributions that can describe commonly-occurring types of random variable, this given distribution will often be a previously-defined distribution. In that case, the specific instance will be defined using *parameters* of the model. In Equation 4.4, the Bernoulli distribution has one parameter, p .

Let's say that we know want to study the total number of 'yes' and 'no' answers across a group of people asked the same question. As every individual who didn't answer 'yes' answered 'no', we only need to consider the total number of one of the answers. Let's consider the total number of 'yes's. It will help if we consider the True/False observations of the random variable X to represent the numbers 1 and 0, respectively. This is a sufficiently common convention that it is encoded into R: follow the steps in the previous section to create the object \mathbf{x} , and inspect the values.

```
> x
[1] TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE
```

Now convert the logical values to numbers using the `as.numeric` function, and you should see that the 'TRUE' elements become the number 1 and the 'FALSE' elements become the number 0:

```
> as.numeric(x)
[1] 1 0 0 1 0 0 0 0 1 0
```

Using this convention, we can define a new random variable Y as the sum of the numerical values (0 or 1) represented by a sample from the random variable X :

$$Y = \sum_{i=1}^n X_i. \forall i \in \{1, \dots, n\}, X_i \sim \text{Bernoulli}(p = 0.5). \quad (4.5)$$

This definition has used some more symbols: $\sum_{i=1}^n X_i$ means to add up the values of X_1 , X_2 , and so on up to X_n . The variable i allows us to refer to elements in a sequence. The definition in Equation 4.5 is still vague: we haven't defined the number of people who are answering the question n . But even so, we can use the fact that a sum of independent and identically distributed² Bernoulli random variables, as this has some known properties characterized with a *binomial* distribution:

$$Y \sim B(n, p = 0.5). \quad (4.6)$$

How is the binomial distribution defined? It will be defined through a probability distribution function, and for the binomial distribution this is expressed through a single mathematical formula with three parameters:

For Binomially-distributed random variable X ,

$$Pr(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}. \quad (4.7)$$

In this equation, $Pr(X = k)$ means the probability of the RV X taking the value k . With this expression, we can calculate the probability that any sum of n Bernoulli RVs, each with probability of a success being p , is equal to the value k . The symbol $\binom{n}{k}$ is a combinatorial function, giving the number of possible combinations of k elements taken from a set of n elements. The details of this expression aren't really important for us, one of the useful capabilities of the R environment is that it has inbuilt functions for computing all sorts of probability distributions. The real point of this example is to show that we have a way of calculating the probability of different outcomes from a random variable, or from a set of samples of random variables. This opens the door to statistical inference....

4.2.1.3 Hypothesis testing

Coming back to our question of whether or not people had seen the film Jurassic Park, we can use this statistical model for *inference*. Let's say we asked 10 people: 7 answered 'yes' and 3 answered 'no'. This represents a sample from Y created by adding together the results of sampling from X 10 times. Seeing this data, we may suspect that there is a bias towards the answer 'yes' across the population (that is, all possible people we could have asked). One might be inclined to state that as the sample wasn't split evenly into 5 'yes's and 5 'no's, it shows that the probability of an individual answering 'yes' was not 0.5. But now we can use the probability distribution function for Y to quantify the *uncertainty* in this assertion, given the data. We do this by constructing a *null hypothesis* that the probability of any one person answering 'yes' was 50% (that is $X \sim Bernoulli(p)$, and therefore $Y \sim B(10, p = 0.5)$). Then we

²This is an important property, often abbreviated to *i.i.d.*

can compute the probability that a sampled value of Y would be *at least that different* to a 50/50 split. Our sampled value was $Y = 7$, meaning that to be at least that different to a 50/50 split, we would need $Y \in \{0, 1, 2, 3, 7, 8, 9, 10\}$. We can compute the probability that one randomly-sampled Y might take any of these values by adding together the probabilities for each of the values. This can be summed up succinctly using mathematical notation:

If an individual's answer is equally likely to be 'yes' or 'no', model X as random variable

$$X \sim \text{Bernoulli}(p = 0.5)$$

models total number of 'yes' responses across a sample as Y

$$Y \sim B(n, p = 0.5)$$

$$\Pr(Y \leq 3, \text{ OR } Y \geq 7) = \sum_{i \in \{0, 1, 2, 3, 7, 8, 9, 10\}} \Pr(Y = i) = 0.344. \quad (4.8)$$

Let's see how we can calculate this using R. We will use the `dbinom` function, which computes the probability distribution function for a binomially-distributed random variable, once you have specified the number of samples from the Bernoulli-distributed variable and that Bernoulli distribution's probability of a True value. To calculate the probability of observing exactly 7 'yes' responses to 10 yes/no questions, for which each question has equal probability of yes or no (probability of yes is 0.5):

```
> dbinom(7,prob=0.5,size=10)
[1] 0.1171875
```

We specify the Bernoulli distribution's probability of a 'True' result (a 'yes' answer to our question) with the argument `prob=`, and we specify the number of times we are sampling from the Bernoulli distribution (the number of people we ask the question) using the argument `size=`. Now, if the probability of a yes or no response is equal (both 0.5), then presumably the probability of 7 no responses would be the same as the probability of 7 yes responses? Let's check this (7 no responses would mean 3 yes responses):

```
> dbinom(3,prob=0.5,size=10)
[1] 0.1171875
```

That should hopefully be reassuring. Now we know how the function works, let's sum up the probabilities for each possible value we are interested in:

```
> sum(dbinom(c(0,1,2,3,7,8,9,10),prob=0.5,size=10))
[1] 0.34375
```

The total probability is 0.34375, so there's just over a 1 in 3 chance of a random sample of 10 individuals yielding as unequal a distribution of 'yes's

and ‘no’s as 7 to 3. In a statistical hypothesis test, this probability of the null hypothesis is often referred to as the *p-value*. I would be very surprised if you hadn’t come across a p-value before, but did you know that this is exactly what it represents?

It may help to illustrate the example with a visual examination of the probability distribution functions. Let’s plot the probability of an outcome on the y-axis against the outcome in question on the x-axis. The Bernoulli distribution with probability 0.5 will be as shown in Fig. 4.1.

And the corresponding Binomial distribution for the sum of 10 independent samples from the Bernoulli distribution is shown in Fig. 4.2.

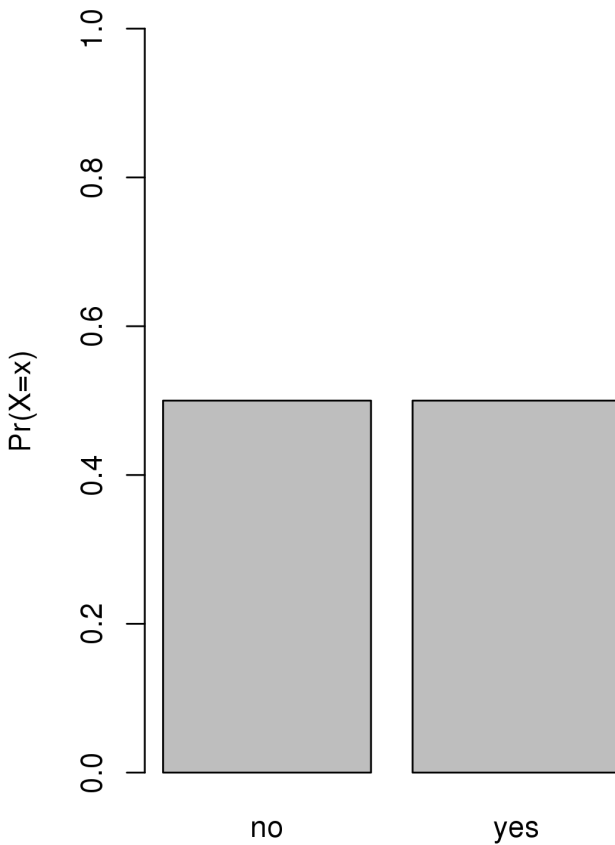
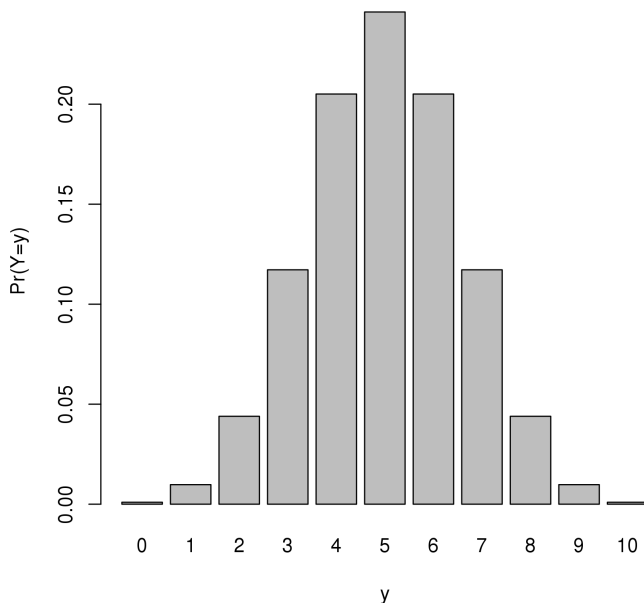


FIGURE 4.1
Probability distribution for Bernoulli trial with equal chance of ‘yes’ or ‘no’.

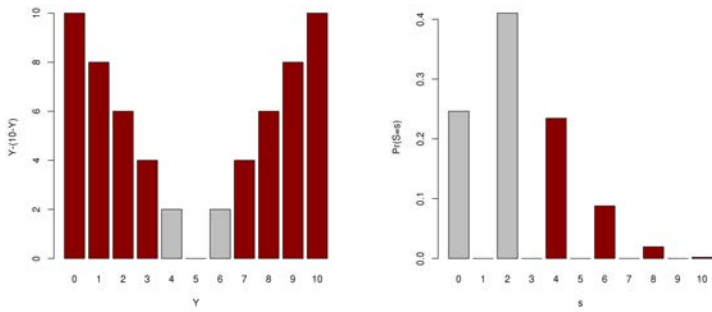
**FIGURE 4.2**

Probability distribution for sum of 10 independent samples from a Bernoulli trial with equal chance of ‘yes’ (which is assigned value 1) and ‘no’ (which is assigned value 0).

We have used a model of expected observations under a null hypothesis to perform a statistical test. This helped us consider whether or not to reject the null hypothesis that our sample of 10 responses were drawn from a population in which the answers ‘yes’ and ‘no’ were equally likely. This is the essence of statistical hypothesis testing. There are many different types of statistical hypothesis test that have previously been defined (and often named), using different models of different characteristics of sets of observations (modelled as samples from random variables). But every statistical test involves using probability models for random variables to find out how unlikely the observed result would be under the null hypothesis.

Often, this process involves calculating a *statistic*, which is a value that represents a characteristic of interest in the observed data. In our example, in effect we have calculated a statistic that quantifies the bias of responses towards one outcome or the other: the difference (regardless of the direction) between the two totals. We could express this as $S = |Y - (10 - Y)|$, where the notation $|x|$ means the absolute value of some number x . The point of producing this statistic is that it is a number that summarizes the property we are interested in, and for which we know the distribution under the null hypothesis.

The probability of a random sample from the modelled distribution having a statistic greater than or equal to the one we observe from our sample (i.e. the *p-value* of the hypothesis test) is equivalent to the area under the probability distribution for the *statistic*³. This is illustrated in Fig. 4.3, with panel (a) showing the values of the statistic for each value of *Y*, and the final p-value equal to the sum of the heights of the bars shaded red in panel (b)⁴.



(a) Statistic *S*, computed for all possible totals of ‘yes’ responses from 10 independent samples from a Bernoulli trial. (b) Probability distribution for statistic *S*, characterizing the inequality of the responses in our sample from a Bernoulli trial.

FIGURE 4.3

Illustrations of the concept of computing a statistic relating to a random variable, and the probability distribution of that statistic. In this case, the statistic represents the imbalance of ‘yes’ vs ‘no’ responses when 10 people are asked the same (yes/no) question.

A final comment on this matter illustrates why sample size is important. If we were instead to have asked 100 people whether or not they had seen Jurassic Park, found the same proportion answering yes (70%), and repeated our test, what would the probability be of observing that result under the null hypothesis? Now this $p = 7.8 \times 10^{-5}$.

My aim with this section was to introduce the formal aspects of statistical analysis methods in a way which is accessible to people who haven’t previously been taught statistics. This should actually make it easier to understand all the analyses performed throughout the rest of the book. The following section briefly introduces another set of mathematical notations which will help understand methods which are useful for analysis of biological data.

³Note that this also applies for continuous-valued statistics, such as the *t* statistic that is calculated in order to perform a student’s t-test.

⁴If this had been a continuous-valued probability distributions, the p-value would be calculated as the area shaded in red.

4.2.2 Linear algebra

Because many biological datasets can be thought of as values of the same set of variables (e.g. levels of expression of different genes) in a set of biological samples (e.g. cell lines with different treatment protocols), it is often convenient to think of these datasets as *matrices*. Some widely-used statistical methods involve performing mathematical operations on matrices (termed *linear algebra*), so it is worth becoming just a little familiar with these principles. A matrix is a grid of numbers, each of which can be referred to by row and column indices (in that order). As an example, let's say A is a 3×2 matrix (this means it has 3 rows and 2 columns):

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}. \quad (4.9)$$

With matrices defined in this way, it can be convenient to use a special form of multiplication, even though this seems at first quite complicated! To multiply two matrices involves creating a new matrix with the number of the first matrix, and the number of columns of the second matrix. The result of the multiplication is best expressed with the formula:

$$\text{If } AB = C, C_{ij} = \sum_{k=1}^p A_{ik} * B_{kj}. \quad (4.10)$$

So the value of the first column for the first row of AB will be the sum of each value of the first row of A multiplied by the corresponding value in the first column of B . And so on. Note that it is only possible to multiply two matrices together if the second matrix has the same number of rows as the number of columns in the first matrix. That is, if A is an $n \times p$ matrix and B is a $p \times m$ matrix, AB will then be an $n \times m$ matrix.

Here's an illustration using R:

```
> A <- matrix(c(1,2,3,4,5,6),nrow=3,ncol=2)
```

We have made an object A , containing a *matrix* of the numbers 1-6. A is a 3×2 matrix, because it has 3 rows and 2 columns. Now we'll make another matrix and call it B :

```
> B <- matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)
```

Now we've made an object B , which is another matrix of the numbers 1-6, but this time with 2 rows and 3 columns. So the result of a matrix multiplication between A and B should be a matrix with 3 rows and 3 columns. The matrix multiplication operator in R is `%*%`. Let's check:

```
> A%*%B
[,1] [,2] [,3]
```

```
[1,] 9 19 29  
[2,] 12 26 40  
[3,] 15 33 51
```

This definition of multiplication for matrices means that to *invert* a matrix means to find the matrix which, when multiplied to the original will produce a vector with a value of 1 for each diagonal element and 0 for all others. This matrix of diagonal 1s is known as the *identity matrix*, as it is the matrix which, when multiplied to any other matrix will result in that matrix unchanged.

A matrix with just one row or one column is usually referred to as a *vector*, and this can be helpful as a way of describing a set of observed measurements. That is, a set of values sampled from a population of values modelled as a random variable. And if multiple measurements are obtained from the same objects, the set of vectors representing each measurement's values can be treated as a matrix. This is usually the case for the datasets we will be working with.

4.2.3 Summary

A little familiarity with linear algebra will help with understanding the way statistical methods are applied to biological datasets, and will make a lot of the code for invoking these methods using computer programs more intuitive. Understanding the principles of statistical methods, in terms of probability models for random variables, is essential for understanding the consequences and limitations of almost any analysis you perform. Use this section as a reference to refer back to, to help understand any of the methods described later on. We will now describe a set of statistical methods that are particularly useful in biological investigations. These will include both descriptive and inferential statistics. As mentioned in the introduction to this book, my philosophy in training people to be effective biological researchers involves covering a fairly small set of methods which are very broadly applicable. There will be methods that you come across which aren't mentioned here, but if you understand these methods and are comfortable applying them to biological datasets, by thinking of interesting biological hypotheses that you can use available datasets to investigate, you can achieve a lot!

4.3 Normalization: Removing technical variation

The main advantage of many modern measurement technologies is that they enable automated data capture on a scale that would be impossible manually. The other side to this scale of data generation is the fact that individual

measurements haven't been curated. If you make a set of measurements yourself, you will typically take repeated measurements and use controls to help distinguish *technical variation*, which are the differences between measurements that arise from the process of making the measurement, rather than differences between levels of the feature being measured. To give an example that will hopefully be familiar to many, consider using qRT-PCR to quantify expression of a gene in two cell lines. Technical variation could arise from differences in the amounts of starting material, from differences in amplification efficiency. Repeated measurements are used to be able to obtain representative values, which can be considered as sampling from a population of all possible values of taking the same measurements from the same cells. Control measurements are taken from each sample to adjust for some technical bias, for example differences in reagent volumes or efficiency. Some high-throughput measurement platforms include internal control measurements, but we can also use the fact that so many measurements are obtained to adjust for technical variation. And while most widely-used technologies have associated procedures for detecting outlier measurements⁵ along with other quality control steps, the process of *normalization* refers to attempts to remove systematic biases that otherwise confound comparisons between different measured entities.

One very simple example is in comparing the results of high-throughput sequencing assays. The measurement values from such assays are usually counts of reads that map to particular genomic features. Given that the total number of reads per sequencing run is typically variable, would it be fair to compare counts of a feature (say, a gene) from two samples with different total numbers of reads? Making the example more concrete, let's say we had sequenced cDNA from two samples and wanted to compare the levels of expression of a gene: one sample has 500 reads mapping to the gene, but 1 million reads overall; the other sample has 400 reads mapping to the gene, but 500,000 reads overall. Given the first sample has more reads overall, but we assume both samples started with the same total amount of RNA (to be converted to cDNA before sequencing), we would expect any randomly-chosen gene to be likely to have more reads from that sample. So we should take into account this source of technical variation (or bias), and we could normalize the read counts to make comparisons more fair simply by dividing the counts for each sample by the total number of reads for that sample. Dividing a set of measurements by a factor is known as *scaling*, which we will look at shortly. The concept of counts-per-million-mapped-reads, or 'tags per million', sometimes abbreviated (CPM or TPM, respectively) is widely seen in the presentation of sequencing-based assay results.

As another example, microarrays provide a convenient platform for automating the process of collecting large numbers of measurements based on the

⁵those so far from what would be expected that they are almost certainly erroneous

intensity of fluorescence of individual spots on a slide. These intensity measurements from a microarray can be influenced by variations in the sample preparation process, the manufacture of the array, the hybridization process and the fluorescence quantification, in addition to the property of interest: the abundance of each transcript in the sample [3]. When comparing expression measurements from different arrays it is therefore appropriate to ‘normalize’ the measurements to reduce as much of the variation due to technical reasons (rather than biological differences between the samples). A demonstration of the need for normalization of Affymetrix GeneChip data is given in [4] along with the description of the widely-used robust multi-array average (RMA) measure of expression from GeneChips. RMA corrects for array-specific background intensity, performs quantile normalization to ensure that the distribution of intensity values across each array is the same, and then uses a simple additive linear model to estimate the expression level of each transcript based on the normalized measurements from each probe in the corresponding probe set and each probe’s specific hybridization affinity estimates. Owing to the normalization of probe-level measurements, RMA can only be applied to normalize measurements from different arrays of the same platform (so that the same probes are present on all arrays to be normalized).

4.3.1 Centering and scaling

If normalization is the removal of systematic bias across sets of measurements to facilitate comparison, the simplest approach to this is to make sure the values you are trying to compare occupy the same range. *Centering* is the process of making a distribution centred on (i.e. the average value is) zero. It will not change the spread of the values, which may well be skewed in favour of one side of the average. Subtracting the mean of the distribution from all values will centre a distribution on zero, although the median tends to be a more robust estimator of population mean from a small sample size and ensures an equal number of points either side of zero after centering.

Scaling is the process of standardizing the spread of values from a distribution, so that differences representing a similar proportion of a distribution’s range result in a difference with a similar magnitude. The simplest way to scale different distributions is to divide all the values by the standard deviation. Like centering, scaling a distribution in this way won’t affect the pattern of the spread of values, so will preserve any skew or multi-modality (multiple distinct peaks around which values are more commonly observed).

When we are dealing with normal distributions, the process of centering and scaling is referred to as *standardization*, because it will result in a standard normal distribution (i.e. with mean zero and standard deviation of 1).

Let’s look at the effects of centering and scaling using R: we will create two variables and compare their distributions using boxplots, before and after

centering and scaling. Remember the `set.seed` function to make sure the results are reproducible, even though we're using random numbers.

```
> set.seed(10)
> x <- rnorm(10,mean=1,sd=2)
```

In this command, after having set the seed for the random number generator (to the completely arbitrary value of 10), we create an object `x` which contains a vector of 10 numbers randomly sampled from a normal distribution with mean 1 and standard deviation 2.

```
> y <- rnorm(10,mean=2,sd=1)
```

And now we have another object `y` which contains a vector of 10 numbers randomly sampled from a normal distribution with mean 2 and standard deviation 1. So we should find that the values of `y` tend to be higher than the values of `x`, but that the `x` values have a larger spread. We can investigate this using a boxplot:

```
> boxplot(list(x=x,y=y))
```

Remember that the `boxplot` function will draw a separate box for each element of the list supplied as the argument to the function. In this case, we have created a new list as the argument for the `boxplot` function, which has two named elements: the element called `x` contains the values of our vector `x`, and the element called `y` contains the values of our vector `y`. The result should appear as in the top left-hand panel of [Fig. 4.4](#).

Now we can center the vectors `x` and `y` by subtracting the median:

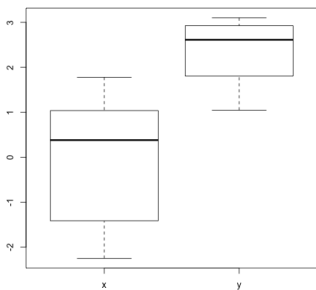
```
> xc <- x - median(x)
```

So we have created a new object called `xc`, which contains the difference between each value of `x` and the median of `x`. We can take the same approach with the vector `y`:

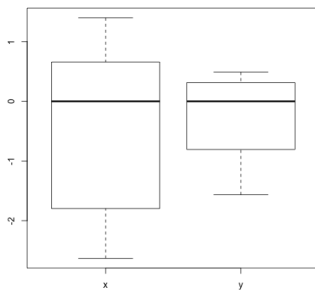
```
> yc <- y - median(y)
```

With our centred vectors `xc` and `yc`, we can now draw another boxplot to compare these distributions. The result should appear as in the top right-hand panel of [Fig. 4.4](#), and we should see that the two distributions are now both centred on zero.

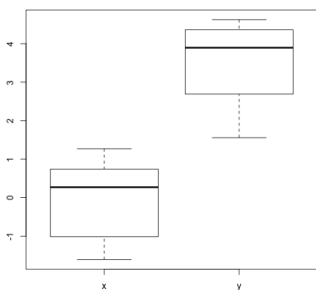
```
> boxplot(list(x=xc,y=yc))
```



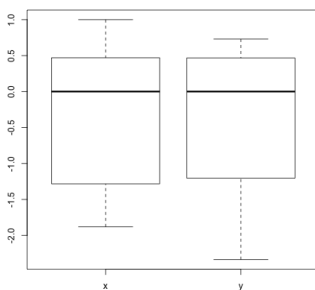
(a) Distribution of values in the vectors x and y , illustrated with a box-and-whiskers plot



(b) Distribution of values in the median-centered vectors x_c and y_c , illustrated with a box-and-whiskers plot



(c) Distribution of values in the standard deviation-scaled vectors x_s and y_s , illustrated with a box-and-whiskers plot



(d) Distribution of values in the standardized vectors x_z and y_z , illustrated with a box-and-whiskers plot

FIGURE 4.4

Illustration of the effect of centering and scaling variables.

As for scaling, this will be similar to centering, but instead of subtracting the median we divide by the standard deviation:

```
x_s <- x/sd(x)
```

So we have created a new object called x_s , which contains each value of x divided by the standard deviation of x . Similarly for y :

```
y_s <- y/sd(y)
```

Again, let's use a boxplot to compare the scaled distributions of `xs` and `ys`, which should appear as the bottom-left panel of [Fig. 4.4](#):

```
> boxplot(list(x=xs,y=ys))
```

Finally, we can apply both centering and scaling, which will *standardize* the distributions so that they both have the same mean and standard deviation. Standardized values are sometimes referred to as *z-scores*:

```
> xz <- (x-median(x))/sd(x)
> yz <- (y-median(y))/sd(y)
```

Note the use of the brackets here: it is usually safe to make explicit which order you want operators to be applied. Finally, we will make another boxplot, which should appear as the bottom-right panel of [Figure 4.4](#). Hopefully it will be clear how this would make individual values more directly comparable between the two vectors?

```
> boxplot(list(x=xz,y=yz))
```

It is important to remember that this approach assumes that the two sets of values would have the same spread, in the absence of technical variation. This is not necessarily the case, although one typically expects it to be if a very large number of values are being compared. We will consider this in more detail in the following section.

4.3.2 An illustrative example

As with any normalization procedures, a decision needs to be made how to apply these to minimize variation within/between variables due to technical causes, whilst maintaining as much of the variation due to our experimental factors (e.g. biological causes) as possible. To provide an example, let us imagine that we have results from an siRNA screen measuring viability of a panel of cell lines following parallel transfections with hundreds of different knock-down constructs. It would be expected that we may see one construct which consistently causes more lethality than another, across all the cell lines. But because we are not confirming the transfection and knock-down efficiency resulting in each measurement, we wouldn't be able to tell whether this is due to the impact of an equivalent level of knock-down, or if one construct is consistently more effective than the other. This reflects the drawback of high-throughput experiments: they generate a lot of data, but there will almost inevitably be unknown sources of technical variation. So if we were to centre and scale all the siRNA screen viability scores, for each construct, then we will highlight differences between the profiled cell lines in such a way that minimizes technical variation in the knock-down and measurement process.

However, we might also consider that some cell lines may have greater transfection efficiency (across our whole screen) than others: so after centering and scaling each construct's distribution of viability scores, we may then want to center and scale each cell line's distribution of (already centred and scaled) viability scores. Then our subsequent analysis will focus on only relative differences between constructs and cell lines, but will be considerably more robust to technical variation. So it is worth being aware of the trade-off between interpretability and reliability.

4.3.3 Quantile normalization

Centering and scaling can be useful tools for normalization, but they will not alter the shape of a distribution, just the range across which its values lie. If we have a dataset in which we are comparing sufficiently many values that we expect the overall shapes of a number of distributions to be similar, for example when we have thousands of measurements of gene expression levels obtained for a number of samples that each had the same total amount of mRNA, then we can use *quantile normalization* to make the distributions the same shape. Using quantile normalization across a number of distributions works by ranking each distribution's values in turn, then one-by-one from top- to bottom-ranked features in each distribution setting all the distributions' equivalently-ranked values to the average value at that rank. To clarify that, the top-ranked feature in each distribution will all have the same value, which will be the average of each distribution's top-ranked values prior to normalization. The second-ranked feature in each distribution will all take the same value, which will be the average of each distribution's second-ranking values prior to normalization. And so on, so that the patterns of the distributions all end up the same, but preserve their original ranking of values. We are utilizing the fact that we have a large number of measurements for each sample to infer that there is a very low probability of seeing systematic shifts from one sample to another due to anything other than technical bias.

4.3.4 Batch effects

There exist subtle sources of technical variation which can arise due to changes in measurement platforms or processes from one batch to the next. Some platforms are more prone to such batch effects than others, but it is *always* worth assuming that samples processed in different batches will have some differences that are due to batch effects and not any experimental variables. How do we then mitigate against these? There are various normalization procedures which attempt to remove batch effects from datasets, such as *COMBAT* and *SVA*. Such approaches will only work when study designs are suitably balanced across batches, so that a model can be constructed to find what variation exists across the experimental factors *within each batch* and then assume that any remaining systematic variation from batch to batch should be removed.

One approach to treating batch effects is to include the batch as a term in a linear model as you would a possible confounding variable. But never forget: if experimental variables are separated across different batches, there is **no** way of telling which differences are due to the experimental variable of interest and which differences are purely due to batch effects.

4.4 Correlation

An intuitive approach to the task of extracting biologically relevant information from a whole set of gene expression data involves grouping together genes that share similar expression patterns, as discussed in [5]. There exist a great many approaches to gene expression data analysis based on the principle that, if the expression of a number of genes is changing in similar ways across a group of microarrays, they are likely to be involved together in some sort of biological process(es) that are occurring. This is colloquially known as ‘guilt-by-association’. One of the simplest ways of assessing similarity in expression pattern is by calculating the Pearson correlation coefficient between the expression profiles of each possible pair chosen from genes represented in the dataset.

4.4.1 Pearson correlation coefficient

The Pearson correlation coefficient measures the linear dependence between two random variables. It is defined as the covariance of the two variables divided by the product of their standard deviations, as shown in Equation 4.11.

$$\rho_{x,y} = \frac{E[(X - \mu_x)(Y - \mu_y)]}{\sigma_x \sigma_y}. \quad (4.11)$$

The resulting $\rho_{x,y}$ value lies between -1 (when the variables are perfectly negatively-correlated) and 1 (when the variables are perfectly correlated), and has the property that it is invariant to changes in location and/or scale of either variable. This means that either variable may be multiplied by some constant factor, or have some constant added to all the values, without changing the correlation coefficient between the two variables.

The values of Pearson correlation coefficients calculated from samples of n observations each from two *independent*, normally-distributed random variables will follow a student’s t-distribution with $n - 2$ degrees of freedom. The corresponding student’s t-distribution can therefore be used to estimate the statistical significance of an observed correlation, in the form of the probability that at least as extreme a Pearson correlation coefficient would be obtained from a random sample of two independent, normally distributed variables.

The Pearson correlation coefficient reflects the extent of a *linear* correlation between two variables. This means that two variables are perfectly correlated if and only if every pair of objects with different values for one of the variables also have a *proportional* difference between their values for the other variable. It also means that a small number of objects with very large differences from the average values in either (or both) variables will contribute more to the result than a large number of objects with small differences from the averages. If either of these characteristics are likely to be undesirable for a given analysis, then it may be worth considering rank-based correlation as an alternative.

4.4.2 Spearman's rank correlation

A common trick for making certain statistical hypothesis tests more readily generalizable is to consider the ranks of a set of objects when ordered on their values for a given variable, rather than the values themselves. This can be convenient because the properties of statistics computed on the ranks will always be the same, regardless of the properties of the underlying values. By computing the Pearson correlation coefficient between the *ranks* of each measured object in the two variables under consideration, we obtain the Spearman correlation coefficient. This value lies between 1 and -1 . A value of 1 means that the variables *monotonically* increase together: if an object has a higher value than another object in one variable, it also has a higher value in the other variable. That is, with two variables x and y , if $x_i > x_j$ then $y_i > y_j, \forall i, j$. A value of -1 means that one variable monotonically decreases as the other increases.

The statistical significance of an observed Spearman correlation coefficient can be computed. This represents the probability that two sets of randomly sorted rankings would have a trend at least as close to monotonic as the real rankings based on the two observed variables. Computation of exact p-values is a combinatorial problem (i.e. very high computational complexity) and is therefore only possible for relatively small numbers of observations, although approximations can be derived for larger values of n . It is worth being aware that when using rank-based statistics it can be important to consider how frequently ties will occur, and how they will be resolved. As an extreme example, if a variable only takes a few discrete values, ties will be very frequent, and statistics such as the Spearman correlation coefficient will not be particularly meaningful.

4.4.3 Examples

Let's use the R environment to show how we can calculate correlations, and statistical inference involving correlations. The approach I've taken to illustrate this is to generate two random samples from a given distribution. These should be independent, and so should have a very low correlation. But if we

create a third variable from the first random sample with a small amount of noise (random variation) added, then this third variable should be highly correlated to the first (with the extent of correlation determined by the standard deviation of the random noise added). This should all become clear as we go through the example. Again, as we're using random number generator, we will set the random seed so that the results are reproducible.

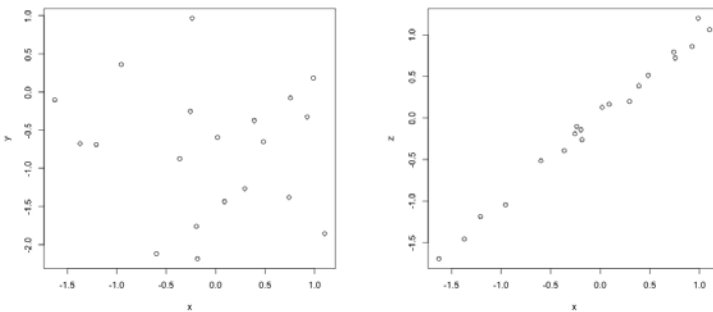
```
> set.seed(10)
> x <- rnorm(20)
> y <- rnorm(20)
```

So we've set the random seed, then created two vectors x and y , each independently sampled from a normal distribution with mean 0 and standard deviation 1. We can use the R function `cor` to compute the correlation between two vectors (of equal length). The default mode for this function is to compute the Pearson correlation coefficient.

```
> cor(x,y)
[1] -0.106711
```

So it appears that these two vectors are slightly negatively correlated to each other. On average, the positions with the higher values of x tend to have the lower values of y . It is often helpful to visualize the relationship between two sets of values like this using a scatterplot. This is very simple using the R function `plot`. The results will appear as in [Fig. 4.5](#).

```
> plot(x,y)
```



(a) Corresponding values of the elements in the independently sampled (uncorrelated) vectors x and y , plotted against each other

(b) Corresponding values of the elements in the correlated vectors x and z , plotted against each other

FIGURE 4.5

Illustration of uncorrelated and correlated variables.

Should we consider this negative correlation between the vectors meaningful: was our random sample of two vectors actually highly correlated? Let's consider this a question of statistical inference, which we will evaluate through hypothesis testing. The *null hypothesis* is that the two vectors \mathbf{x} and \mathbf{y} were *independently* drawn from two normal distributions. Note: correlations are invariant to centering and scaling, as they give the *relative* change in one variable as there is a change in the other variable. So our statistical significance *p-value* will be the probability that we would observe a correlation with as big magnitude (we don't mind whether it's positive or negative) from any two independent random samples from a normal distribution. We could attempt to approximate this probability distribution by setting up a *for* loop to compute a large number of correlation coefficients from random samples of this size, and computing the proportion that had an absolute value at least as large as 0.106711 (the absolute value of the correlation we observed between \mathbf{x} and \mathbf{y}). But, because the distribution of correlation coefficients is actually well characterized, R can compute it directly using the function `cor.test`:

```
> cor.test(x,y)
```

You should see in the output from this function that, as well as giving the correlation coefficient, it has computed 95% confidence intervals on the correlation coefficient, a t-statistic value and degrees of freedom, and a p-value corresponding to this distribution. So a p-value of 0.654 implies that we would expect to observe correlations of this magnitude more than half of the time we randomly sampled 20 values from two independent normal distributions, so the observed correlation is not very remarkable!

Now let's try creating a third variable which does depend on the variable \mathbf{x} :

```
> z <- x+rnorm(20,sd=0.1)
```

So we have added to the vector \mathbf{x} another random sample of 20 values from a normal distribution, but this time the standard deviation is only 0.1, so these values should be a lot smaller than the values of \mathbf{x} they are being added to. We have saved the result of this as an object in our R workspace called \mathbf{z} . Now let's compute the correlation between \mathbf{x} and \mathbf{z} :

```
> cor(x,z)
[1] 0.9949293
```

Given that correlation coefficients range from -1 to $+1$, a value of 0.995 seems very high! What is the probability of observing such high a correlation coefficient from independently sampled variables:

```
> cor.test(x,z)
```

The result of this function states $p\text{-value} < 2.2e-16$, i.e. $p < 2.2 \times 10^{-16}$. This says the p-value is so small that it might not be possible to estimate it exactly, and it should be enough to consider that it is very small. So these variables are very highly positively correlated. Again, it is probably helpful to visualize these with a scatterplot, and you can use the two plots from [Fig. 4.5](#) to get a sense of what correlation between variables represents.

```
> plot(x,z)
```

In the above example, we knew that the variables were normally distributed, because we had sampled them from normal distributions. So the inference from the Pearson correlation coefficient was exact. What about if we repeated the same approach, but had sampled the variables from *uniform* distributions: these will be very much not normally-distributed? Let's try it, where all we need to do is replace the function `rnorm` used to create `x` and `y` with the function `runif`. Note, we still add normally-distributed random noise to `x` to get `z` (this is to ensure the offsets between `z` and `x` will typically be small and centred on zero).

```
> set.seed(10)
> x <- runif(20)
> y <- runif(20)
> cor(x,y)
[1] -0.02601963
> z <- x+rnorm(20,sd=0.1)
> cor(x,z)
[1] 0.919642
```

So again, we have the two independent samples `x` and `y` with very little correlation (the fact that it is also negative is purely coincidental), but the variable `z` (created by applying small offsets to the sample `x`) is highly positively correlated with `x`. What about estimating statistical significance? We know that the samples were not normally-distributed, so the p-values derived from the Pearson correlation coefficient will not be exactly appropriate. Perhaps we should instead use the Spearman correlation coefficient for inference, as this doesn't assume that the variables are normally distributed? Let's try both, and see how different the results are:

```
> cor.test(x,y)
```

This performs hypothesis testing against a null hypothesis of independently sampled variables, based on the Pearson correlation coefficient. It gives a p-value of 0.9133.

```
> cor.test(x,y,method='spearman')
```

By setting the argument `method` to `'spearman'`, we perform hypothesis testing against a null hypothesis of independently sampled variables, but based on the Spearman correlation coefficient. It gives a p-value of 0.8016. So this isn't that different to the value based on the Pearson correlation coefficient. How about for the deliberately correlated variables `x` and `z`?

```
> cor.test(x,y)
```

Hypothesis test based on Pearson correlation coefficient gives a p-value of 9.886×10^{-09} .

```
> cor.test(x,z,method='spearman')
```

Hypothesis test based on Pearson correlation coefficient gives a p-value of 4.506×10^{-06} . So we have very small estimated probabilities of observing such correlated variables as a result of independent samples from any (unspecified) distributions, similar to when we use an incorrect assumption of normally distributed variables.

4.5 Clustering

Based on the principle of 'guilt by association', one of the most widely used methods of illuminating order from a large set of data is that of *clustering*. Its goal is to classify entities into (unspecified) groups based on their profiles. These entities could either be the objects represented in a dataset or the attributes available for each object. Clustering is generally a form of *unsupervised learning*⁶ in which a *distance metric* (a measure of dis-similarity between a pair of entities) is used to allow the most similar entities to be grouped together.

One common distance metric for numerical data is known as the *Euclidean distance*: this is the square root of the sum of squared differences between each element of the two entities. Say we had two vectors $\mathbf{x} = \{x_1, \dots, x_n\}$ and $\mathbf{y} = \{y_1, \dots, y_n\}$, then the Euclidean distance would be expressed as $\sum_{i=1}^n \sqrt{(x_i - y_i)^2}$. This may seem quite complicated, but think about what it represents if $n = 2$: \mathbf{x} and \mathbf{y} then each have two elements, and the distance between \mathbf{x} and \mathbf{y} is the length of the line connecting the points if the first element gives a horizontal co-ordinate and the second element gives a vertical co-ordinate. Then the formula for Euclidean distance is Pythagoras' theorem, where the distance is the hypotenuse of the triangle. Another distance

⁶so called because we don't use any *known* assignment of some entities to groups in order to estimate the others: that would be *supervised learning*

metric can be derived from the correlation between variables. Remember that perfectly linearly-correlated vectors of values have a Pearson correlation coefficient of 1, independent vectors have a Pearson correlation coefficient of 0, and perfectly anti-correlated vectors have a Pearson correlation coefficient of -1 . We can compute a distance between two vectors just by subtracting their correlation coefficient from 1: then perfectly-correlated vectors will have distance 0, and the distance increases with decreasing correlation up to perfectly anti-correlated vectors with distance 2.

Using the Pearson correlation coefficient to identify genes with similar expression patterns across a dataset is a simple, but effective form of clustering. There are many different approaches for performing clustering on a whole dataset at once, which is a little more complicated, and one of the most widely used of these clustering approaches is described below.

Hierarchical clustering was applied to gene expression microarray data to great effect in [2] and has since become one of the most widely-used methods to analyse high-dimensional molecular datasets. It works toward a goal of producing a ‘binary tree’ representation of the features (e.g. genes) and/or samples in the dataset. For example, a binary tree for the samples in the dataset might be produced on the basis of a similarity score between each sample and the others. This consists of a recursive⁷ organisation of the entities through merging them into pairs. One advantage of applying the hierarchical clustering technique to complex, high-dimensional datasets is that the tree structure enables examination of different levels of clustering, which can lead to classifications and visualisations of the data that are both intuitive and useful for exploration [2]. A hierarchical cluster *dendrogram* (a tree diagram, example as in Fig. 4.6) can be used to assign the objects to clusters by cutting the tree horizontally at a specified height, and each branch of the tree at that height represents a different cluster. An example of such a hierarchical clustering of the samples from a microarray dataset has been used in the *heatmap* shown in Fig. 4.7. A heatmap is a way of visualizing a (potentially large) matrix of numbers all together: the numbers are mapped to a colour scale, so we can see differences in the patterns of colours across rows and columns. This is particularly useful when clustering has been applied to order the rows and columns.

4.5.1 Clustering illustration using R

To practise clustering datasets, it will illustrate this better if we create a dataset that has some structure. We could create a dataset with structure in

⁷A recursive method is one which is applied to an input, then applied again to the result of this application, and so on until some criterion for finishing is met. In the case of producing a binary tree by merging entities into pairs, the process finishes when the two entities being merged contain all the individual entities between them.

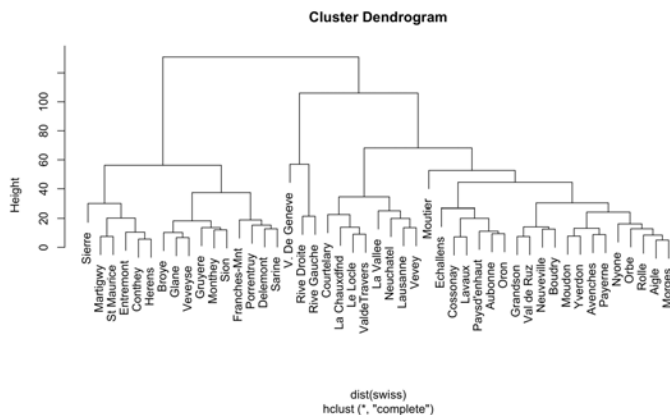


FIGURE 4.6

Clustering dendrogram representing a hierarchical grouping of columns of a data matrix, based on their profile across rows.

a similar way to creating 2 correlated variables, or alternatively we could load one of a handful of example datasets in R. You can list the available datasets, or load one of those datasets, using the R function `data`:

```
> data()
```

This lists the names of all the inbuilt datasets, and a brief description of the data they contain. Because it is a relatively convenient size for this demonstration, we will use the dataset ‘swiss’, which includes fertility rates and socioeconomic indicators from Swiss towns in 1888.

```
> data(swiss)
```

Now an object `swiss` has been added to the workspace, which contains a `data.frame`. Note that, unlike most molecular biology datasets, this data frame is structured with the features in the columns and the objects (the towns) in the rows.

If we recall that clustering will need some measure of the dissimilarity between all pairs of entities, we can use the function `dist` to compute the Euclidean distance between all rows of a numeric matrix (or in this case, a data frame with exclusively numeric columns).

```
> dist(swiss)
```

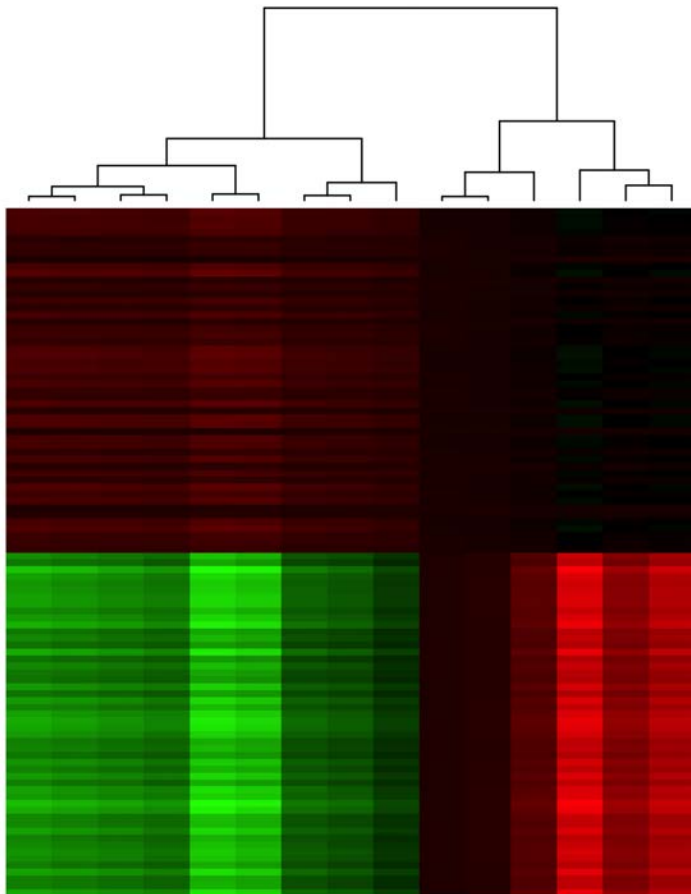



FIGURE 4.7

Heatmap of a high-dimensional dataset, with each sample represented by a column and each feature represented by a row. Connected lines above the heatmap form the sample clustering dendrogram, with the dissimilarity between two samples reflected by the height above the heatmap at which respective branches of the tree join. The level of each feature in each sample is represented by the colour scale from green (low) to red (high).

Note, the output of this will be quite a large (47×47) matrix, as the data frame has 47 rows. We can apply a hierarchical clustering algorithm to the distance matrix using the R function `hclust`:

```
> hclust(dist(swiss))
```

In the previous command we have first applied the `dist` function to the data frame called `swiss`, and then applied the `hclust` function to the result.

This creates the clustering dendrogram, but it would help to visualize this. Helpfully, the function `plot` knows how to deal with a dendrogram, and the result should appear as in [Fig. 4.6](#).

```
> plot(hclust(dist(swiss)))
```

Remember that the inbuilt dataset `swiss` has the features as columns, and most molecular biology dataset have the features as rows, but if we wanted to cluster the columns of the matrix then we just need to calculate the distance matrix on the *transpose* of the matrix, using the R function `t`:

```
> hclust(dist(t(swiss)))
```

4.6 Linear regression models

A linear model represents the values of some observed variable as a linear combination (consider this as a weighted sum) of any number of (independent) ‘explanatory variables’, and an error term. This is commonly defined as in Equation 4.12:

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_N X_N + \epsilon. \quad (4.12)$$

In Equation 4.12: Y is the observed variable, β_0 is the intercept, $\beta_{1\dots N}$ a set of N coefficients, $X_{1\dots N}$ a set of N variables included in the model (each with a corresponding coefficient), and ϵ the residual error (the discrepancy between the value explained by the model and the actual value of Y).

You can think about this in terms of the formula for a straight line in a two-dimensional co-ordinate system. Y is a vector of y-axis co-ordinates, expressed as an intercept (β_0) plus some multiple (β_1) of the x-axis co-ordinates (X_1). In a situation where there is not a perfect fit between the two variables X_1 and Y (that is, the Pearson correlation coefficient between them is not 1 or -1), then a vector of offsets ϵ needs to be included. Between any two numerical vectors, we can use this formula to express one as a linear function of the other, and find the best value for the intercept β_0 and the coefficient β_1 by finding the values which minimize the sum of the squared errors $\sum_{i=1}^n \epsilon_i^2$ (in model fitting, these individual errors ϵ_i are known as the *residuals*). From this we can see that the square root of this sum-of-squared errors would be the Euclidean distance between Y and the linear model’s estimate for Y , which is given by $\beta_0 + \beta_1 * X_1$.

Let’s try fitting a linear model using R. We will start by loading the ‘swiss’ dataset again:

```
> data(swiss)
```

Now we have added a pre-built data frame to our workspace, called `swiss`. To fit a linear model, we can use the function `lm`. The first argument of this function is a *formula*, which is specified with the tilde symbol `~`. You may recognize this from our earlier descriptions of the distribution of random variables. In a formula specification in R, the `~` is placed between the outcome variable and the explanatory variable(s). We can either give vectors in the formula specifications, or names of columns in a dataframe. If the latter, then we also need to pass an argument to the `lm` function specifying what data frame to use. So if we wanted to fit a linear model of the fertility rate of the Swiss towns based on education levels as an explanatory variable, using the 1888 data frame:

```
> lm(Fertility ~ Education, data=swiss)
```

The output of this function are the values of the coefficients for the intercept (β_0) and the one explanatory variable, ‘Education’ (β_1). We can see that this coefficient is -0.8624 : meaning that as the value of the Education variable increases by 1, on average the value of the Fertility variable decreases by 0.8624. That’s the *average effect*⁸, but it doesn’t give us an indication of the extent to which this average effect applies across all the points in the dataset. We can use linear models to perform statistical inference, and will do so shortly.

The real convenience of the linear model framework is that it allows arbitrarily complex experimental designs to be implemented in the same way as a straightforward comparison of two groups. It becomes very hard to visualize a line where one co-ordinate is expressed as a function of more than 1 or 2 other co-ordinates, but using the formula in Equation 4.12, it is easy to add any number of X variables and corresponding coefficients β to the model. Explanatory variables in the model (the different X_i vectors) could for example be a set of clinical variables, or in the case of a comparison between two groups, a single indicator variable (taking values 1 or 0) representing which group each object belongs to. As we have seen, linear regression software implemented in R will find the *maximum likelihood* estimates for the values of the model coefficients (these estimates are referred to as $\hat{\beta}_i$), which are the values of the β_i s which minimize the sum-of-squared errors. But from the fitted models, we can use the convenient property that if the residuals are normally distributed, and the coefficients are normally distributed under the null hypothesis (that is, if we obtain samples from independent distributions), then the coefficient estimates divided by the standard error of the coefficient estimate gives a statistic that follows a t-distribution under the null hypothesis. It probably isn’t necessary to know all these details, but for completeness sake, the standard error of the

⁸N.B. We actually need to be very careful using words like ‘effect’ which apply some causality, because this model can only assess a linear association between variables. There is no way to know what, if anything, is *causing* this observed association.

coefficient is computed as the square root of the standard error of the residuals (let's call this s) divided by the variance estimate for the explanatory variable in question (let's call this $V_i = \sum_j (x_{ij} - \bar{x}_i)$). So the full equation for the t statistic for an explanatory variable in a linear model is given by Equation 4.13:

$$t_i = \frac{\hat{\beta}_i}{\sqrt{\frac{s}{V_i}}}. \quad (4.13)$$

You will not need to know the precise details of these computations, as R computes them for you. But it is good to have an idea how the linear model can be used for statistical inference regarding linear association (dependence) between pairs of variables. Going back to our example in R, if we use the `swiss` dataset again to fit the linear model, but this time save the result as an object on the workspace:

```
> data(swiss)
> m1 <- lm(Fertility ~ Education, data=swiss)
> summary(m1)
```

Here we have made use of the `summary` function for linear model objects (such as the one we have created and called `m1`). You will see the output includes the distribution of the residuals, and the values for the $\hat{\beta}_i$ coefficient estimates, their corresponding standard error, t-statistics and p-values. It also lists the residual standard error, the R-squared value and an F-statistic and corresponding p-value. With only one explanatory variable in the model (apart from the intercept), this p-value should be the same as that derived from the t-statistic for the explanatory variable.

But what if we include more variables in the model? The F statistic (and moderated F statistic) can be derived for any *combination* of terms in the model, enabling estimation of the significance of the entire model. This is particularly helpful when we have categorical variables that can take more than 2 values, or when we have a model that includes a number of variables of interest. Let's try fitting another model on the same dataset, but this time including 2 explanatory variables: the level of education and the infant mortality rate.

```
> m2 <- lm(Fertility ~ Education + Infant.Mortality, data=swiss)
```

We have included multiple explanatory variables in the formula through use of the addition `+` symbol. The `summary` function will work as before, but now for the new model object we have created called `m2`:

```
> summary(m2)
```

Now you should see that there is a coefficient, a t-statistic and a p-value for each explanatory variable, which characterizes the association between that variable and the outcome variable *adjusting* for the other explanatory variables in the model (effectively, the expected average effect if the other variables were held constant). Interestingly, if you compare the results of `m2` with those from `m1`, you can see that adding the additional explanatory variable has reduced the magnitude of the coefficient estimate for the Education variable, but as it has also reduced the standard error of the estimate the association is actually more statistically significant when adjusting for the Infant.Mortality variable. And we can see that this model has an F-statistic derived p-value which differs from either individual variable's corresponding t-statistic p-value, because now the F-statistic summarizes the association between *both* explanatory variables and the outcome.

One final note on linear models with multiple explanatory variables, the cautionary tale of *collinearity*. If two explanatory variables are themselves very strongly associated (i.e. they have a Pearson correlation coefficient close to 1 or -1), then the computations made to estimate the coefficients end up being very *unstable*. We'll look a bit at stability later, but in this case it is sufficiently bad that if two perfectly-correlated explanatory variables are included in the model, the calculation for the coefficient estimates is *impossible*! R will not necessarily warn you of this potential pitfall, so it is good to test for associations between the explanatory variables before fitting such models.

4.6.1 Limma

The *limma* package in R provides computationally efficient tools for fitting large numbers of linear models to high-dimensional datasets. Examples of its application will appear throughout the later chapters of this book, but it is worth mentioning here that it involves a final step beyond the linear model fitting and inference. It makes use of a *Bayesian* approach to statistics, in which the observed evidence refines our prior beliefs. In this example, the prior belief is that it is very unlikely that the random variable populations from which our observed data has been drawn are perfectly associated. So this *prior* belief is used to provide a *moderation* of the observed t-statistics, based on sharing information of the sample variances (and therefore the standard errors for the coefficients) across all fitted models, to reduce the possibility of large statistics arising from very low estimates of sample variance. This is important for many high-dimensional datasets, because such large statistics are particularly likely to occur when the experiment involves large numbers of measurements from small numbers of samples.

One of the utilities of the *limma* package is that by defining one set of values for the explanatory (X) variables, a single function will fit linear models and extract statistics for the corresponding coefficients for any number of different outcome (Y) variables in turn. The package works using a *design matrix*

that you have to create, specifying the values of the explanatory variables (X_1, \dots, X_N in Equation 4.12) for each instance, and expects a matrix giving the values for each of the outcome variables (each outcome variable represents a different Y as in Equation 4.12) for each instance. Because `limma` was designed for application to datasets with large numbers of measured features (e.g. genes) in a relatively small number of instances (e.g. biological samples), it assumes the different outcome variables will be in the rows of a matrix⁹. This may all seem rather complicated, but we can illustrate the process of using the `limma` package with the same ‘swiss’ dataset as for the previous part of this section.

These are somewhat arbitrary examples, but they should make it clear how you would use the approach to work with a high-dimensional molecular biology dataset. We’ll try two different forms of explanatory variable, one categorical and one numeric. If these seem a bit complex, don’t worry for now. We will revisit this a number of times through real applications in biological research, and with repetition it will become clear.

4.6.1.1 Installing `limma`

The `limma` package is part of *Bioconductor*, which is a set of R packages specifically for computational biology. To install the `limma` package, we can load the code for the `biocLite` function directly from the Bioconductor website, then run the function specifying we want to install the package ‘`limma`’:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("limma")
```

Alternatively, there is a newer package manager for Bioconductor called *Bioc-Manager*. This can be installed using the standard `install.packages` function, and then a function `install` can be called from this package using a double-colon (`::`):

```
> install.packages("BiocManager")
> BiocManager::install("limma")
```

4.6.1.2 Categorical explanatory variables

For the categorical variable, let’s say we wanted to use linear models to see if any of the variables recorded in the Swiss towns dataset had significantly different values for towns with a letter ‘y’ in the name, compared with towns without a letter ‘y’ in the name. It is a very arbitrary example!

First, we have to create a design matrix. This will have one column for each explanatory variable, including an intercept. We can create a matrix by using

⁹Arrays in R work better when the number of rows is much larger than the number of columns, rather than the other way round.

the R function `cbind` to combine vectors together as columns of an array. In this case, our first column will be the intercept, which takes a value 1 for every town.

```
> intercept <- rep(1,nrow(swiss))
```

Here we create an object called `intercept`, which contains a vector where each element is the value 1, and it has the length of the number of rows in the `swiss` data frame.

The second column will be our variable interest specifying which of the two groups each town belongs to: those with a ‘y’ in the name, and those without. Group membership can be encoded in explanatory variables of a linear model using values 0 and 1 to represent the two groups, respectively. We can create a vector in which every value is 0, and then replace the values with 1s for each element corresponding to the second group. In the case of this example, that means finding the towns which have a ‘y’ in their name. We will make use of R’s *regular expression* search function `grep`: this function looks through a vector and returns the indices for the elements of the vector which match the specified pattern (in our case, the pattern will be ‘includes a letter y’).

```
> lettery <- rep(0,nrow(swiss))
```

First, we have created an object called `lettery`, which contains a vector where each element is value 0. Now we need to find which town names (the rownames of the data frame `swiss`) contain a letter ‘y’:

```
> lettery[grep('y',rownames(swiss))] <- 1
```

In this command, we have used the `grep` function to find the *indices* (more than one index) of the elements in the vector `lettery` to set to the value 1.

Now, we can combine the two columns together to form the design matrix:

```
> design1 <- cbind(intercept,lettery)
```

Here we have used the `cbind` function to combine two vectors together as columns of an array. The result is then used to create an object called `design1`.

With a design matrix created, we can load the *limma* package¹⁰ and then use the `lmFit` function to fit the specified linear model for each variable in the dataset:

```
> fit1 <- lmFit(t(swiss),design1)
```

¹⁰It is a Bioconductor package, for instructions see <https://bioconductor.org/packages/release/bioc/html/limma.html>.

An object called `fit1` has been created, storing the linear model coefficients for each row of the input matrix (`swiss` has been transposed with the function `t`), with the explanatory variables specified in the `design1` matrix. To compute the empirical Bayes moderated statistics for the model fits, we still need to apply the `eBayes` function.

```
> fit1 <- eBayes(fit1)
```

We have created a new `fit1` object (replacing the old one), now storing the moderated t-statistics for each model.

Finally, we can inspect the results using the `topTable` function. This returns the computed statistics for a specified set of explanatory variables (columns of the design matrix), for all of the outcome variables (rows of the input data matrix). In this example, we want to view the statistics for the second column of the design matrix, which corresponds to the indicator for whether or not the town had a ‘y’ in its name. Within the `topTable` function, we will specify that we wish to extract statistics for model coefficient 2:

```
> results1 <- topTable(fit1,coef=2)
```

We have created an object called `results1`, storing the results of the linear model fit for each row of the input matrix (i.e. each column of the original `swiss` data frame). Now inspect this object just by entering the object names:

```
> results1
```

It should give the following output:

```
logFC AveExpr t P.Value adj.P.Val B
Education -4.547619 10.97872 -1.511452 0.1374733 0.3227601 -4.593995
Catholic 16.314329 41.14383 1.250145 0.2175324 0.3227601 -4.594620
Infant.Mortality 1.200866 19.94255 1.215578 0.2303116 0.3227601 -4.594695
Fertility 4.700433 70.14255 1.195725 0.2378960 0.3227601 -4.594737
Examination -2.833333 16.48936 -1.118879 0.2689668 0.3227601 -4.594895
Agriculture 6.822727 50.65957 0.952495 0.3457954 0.3457954 -4.595204
```

This table includes columns for:

- log fold-change: average increase in the outcome variable as the explanatory variable increases by 1)
- the average value of the outcome variable
- t-statistic for the term in the fitted model

- corresponding p-value and adjusted p-value (because many models have been tested, it is prudent to apply *multiple hypothesis testing* adjustment, which will be explained in more detail next)
- the Bayesian log-odds of an association between the outcome and explanatory variables: this tends to be less intuitive to interpret than the p-values, so I wouldn't worry about it (you'll notice the ordering is the same as the ordering by unadjusted p-values anyway)

So it appears that none of the variables in this dataset are significantly different for towns with a 'y' in the name, as compared to towns without a 'y' in the name. That's not surprising, as it was a totally arbitrary characteristic on which to group the towns, with no reason at all to believe any of the recorded attributes would be different. But it shows how to create a design matrix with a categorical variable. The results in this case are equivalent to applying a t-test for each recorded attribute, comparing the values across the two groups of towns, with the empirical Bayes moderation. However, it is trivial to evaluate more complex models in exactly the same way, incorporating multiple explanatory variables just by adding more columns to the design matrix. Then the statistical hypothesis testing extends beyond what could be achieved with a t-test, because associations between variables can be tested while adjusting for other potential confounding factors.

4.6.1.3 Continuous explanatory variables

While the previous example used a numeric vector of 1s and 0s to represent membership of different groups, the model fitting and evaluation process is exactly the same for evaluating associations between continuous (numeric) variables. Let's say now that we wanted to find which of the recorded attributes of the towns in the Swiss 1888 dataset were most strongly associated with the infant mortality rate. We will still need to create a design matrix with one column for the intercept and one column for the value of the infant mortality rate.

```
> intercept <- rep(1,nrow(swiss))
```

As in the previous example, we create an object called `intercept`, which contains a vector where each element is the value 1, and it has the length of the number of rows in the `swiss` data frame.

Next, we want to obtain the values for the explanatory variable of interest, which in this case is the 6th column of the `swiss` data frame:

```
> infm <- swiss[,6]
```

Here we have created an object called `infm`, containing a vector with the values of the 6th column of the data frame `swiss`, which correspond to the

infant mortality rates, our explanatory variable of interest to include in the models.

Then we create the design matrix by binding the two vectors together as columns:

```
> design2 <- cbind(intercept, infm)
```

As in the previous example, we have used the `cbind` function to create an array called `design2`, which is our design matrix for this task.

We now want to use this design matrix to fit linear models (with the `lmFit` function), but we need to consider that our explanatory variable is one of the outcome variables to be tested! So in this case, our input data matrix will only be made up of the first 5 columns of the `swiss` data frame, transposed so that the variables to be tested are the rows:

```
> fit2 <- lmFit(t(swiss[,1:5]), design2)
```

In this command, the `lmFit` function has been used to fit linear models with explanatory variables defined by the design matrix `design2`, and outcome variables as columns 1 to 5 of the `swiss` data frame (then transposed with the `t` function so that the variables are the rows, and the instances (towns) are the columns).

We then compute the empirical Bayes moderated statistics with the `eBayes` function, and extract the values with the `topTable` function (again specifying that it is the coefficients for the second column of the design matrix that we are interested in):

```
> fit2 <- eBayes(fit2)
> results2 <- topTable(fit2, coef=2)
```

You should note that these steps appear exactly as with the `fit1` model that was fitted using the `design1` matrix. If we inspect the results, they should appear as follows:

```
> results2
logFC AveExpr t P.Value adj.P.Val B
Fertility 1.7864860 70.14255 3.0751656 0.003504682 0.01752341
-1.923477
Catholic 2.5128022 41.14383 1.2184526 0.229150537 0.57287634
-5.365024
Examination -0.3123054 16.48936 -0.7549719 0.454043429
0.63692306 -5.797767
```

```
Education -0.3278817 10.97872 -0.6646396 0.509538451 0.63692306  
-5.859141  
Agriculture -0.4745338 50.65957 -0.4152930 0.679822741  
0.67982274 -5.988589
```

So in this more realistic application, where the explanatory variable of interest was actually meaningful in the context of the study the data came from, we find one of the outcome variables ('Fertility') to be significantly associated with the explanatory variable ('Infant.Mortality') after adjusting for the fact that we have performed multiple independent hypothesis tests. More on that in the next section.

4.7 Multiple hypothesis testing

Consider a hypothesis test such as the ones described throughout this chapter. We typically use the distribution of a statistic under the null hypothesis against which we are testing (e.g. complete independence between two variables) to compute a p-value, and then use this p-value to inform our belief in the alternative hypothesis (e.g. that the two variables weren't independent because they showed a linear correlation). Then remember that if a statistical test of the association between two variables yields a p-value of 0.01, this implies that there was a 1 in 100 chance that random samples from two independent variables would show such an association. This means that if you looked at 100 pairs of unassociated variables, you'd only expect 1 to show as strong an association as the variables of interest. But when working with high-dimensional datasets, such as those arising frequently in molecular biology, we may be testing the association between hundreds, thousands or even millions of pairs of variables!

Let's say we had data from a genome-wide association study (GWAS), which measured 2 million single-nucleotide polymorphisms (SNPs) and the disease status of a cohort of individuals. If we tested each SNP in turn for an association between the genotype and the disease status of the corresponding individuals, we would expect 20,000 unassociated SNPs to appear associated with $p < 0.01$, just through random sampling ($2 \text{ million} \times 0.01 = 20,000$).

This issue is the concept of a Family-Wise Error Rate: what is the probability of a *set* of hypothesis tests yielding any individually significant results when applied to data following the null hypothesis (in this case, no association). Fortunately, there are a number of methods which deal with this problem, which are known as multiple hypothesis testing adjustment (or multiple testing correction). The simplest such approach, known as Bonferroni adjustment,

is just to multiply the p-value from each test by the number of tests performed. There are more sophisticated methods too, including the *false discovery rate* estimating methods. R implements these in the `p.adjust` function. For example, if we had a vector of p-values `p`:

```
> p.adjust(p,method='fdr')
```

This will implement the Benjamini-Hochberg procedure for estimating false discovery rate across a set of hypothesis tests. This is the method which is applied in the *limma* package.

4.8 Survival analysis

Many investigations of data from clinical samples involve estimating the association between some measured values (e.g. expression level of a particular gene) and clinical variables describing the time until occurrence of a particular event, if that event occurred. Such clinical variables are generally described as survival variables, because a common example is that of the recorded time until death of the patient. Because patients may still be alive when analysis is performed, or may have died for some reason unrelated to the investigation, or may have left the study so that follow-up data cannot be obtained, this type of variable must be treated with care. Cases where the patient has died for some unrelated reason or have left the study are described as being censored.

The goal of survival analysis is typically based around estimation of the survival function: the probability of survival beyond time, t . With estimates for the survival function, the influence of a particular variable of interest on the survival probability can be investigated.

4.8.1 Kaplan-Meier plots

The Kaplan-Meier estimator provides a method for estimating the survival function of an event, in the presence of censored data. It is the non-parametric maximum likelihood estimate for probability of survival of a member of a given population beyond specified time t , given in Equation 4.14:

$$S(t) = \prod_{t_i < t} \frac{n_i - d_i}{n_i}. \quad (4.14)$$

In Equation 4.14, n_i is the number of survivors at time t_i minus the number who have been censored by time t_i , and d_i is the number of (relevant) deaths at time t_i .

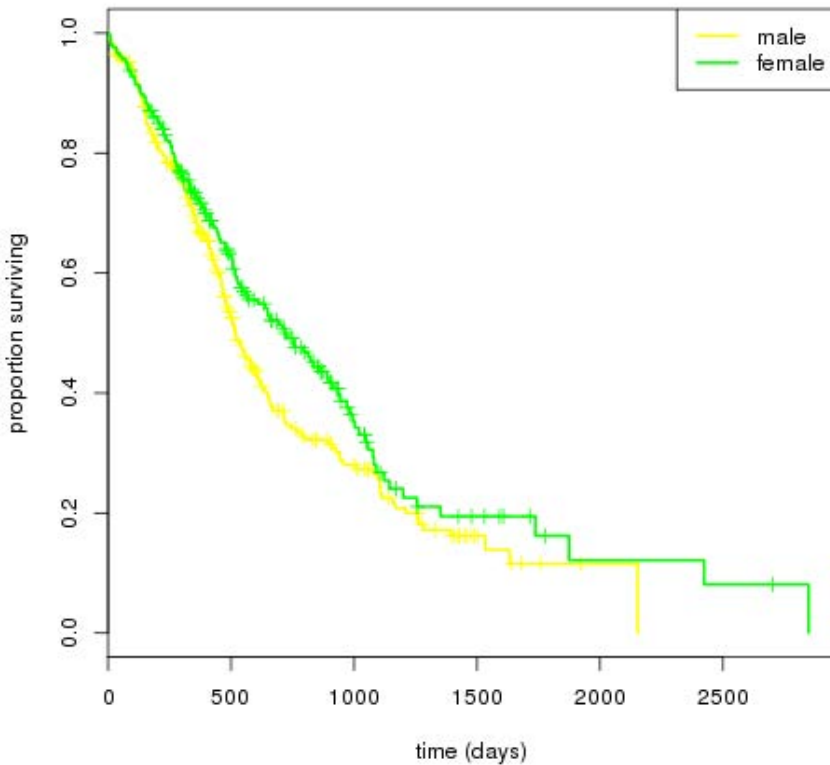


FIGURE 4.8

Example of a Kaplan-Meier plot showing survival function curves for two groups of patients. Censored samples are indicated with vertical tick-marks on the plot.

The Kaplan-Meier estimate of the survival function can be used to produce a plot illustrating patient survival, and for comparing the survival functions of two groups of patients (as shown in Fig. 4.8). The application to analysis of molecular data can be in the form of estimating the significance of the difference between the survival function for patients with low expression of a given gene and the survival function for patients with high expression of that gene. The logrank test can be used to estimate the significance of the difference in survival between two distinct groups. This evaluates the probability that we would observe as uneven a number of events in each group, relative to the total observation time across all the patients in the group, if the patients in each group were randomly sampled from populations with the same survival function ($S(t)$ in Equation 4.14). Alternatively, regression models may be used to estimate the quantitative influence of the actual values

of explanatory variables (such as expression of a given gene) on the survival function.

4.8.2 Cox proportional hazards regression models

The proportional hazards regression approach to analysis of survival data, introduced in [1], is based on the hazard function. The hazard function is related to the survival function in that it defines the probability density function for the event (e.g. death) occurring in relation to time – this can be used to calculate the estimated probability that the event will occur in a given time interval. The Cox proportional hazards regression model is based on assumptions that there is a baseline hazard function, $\Lambda_0(t)$ that describes the change in probability of the event occurring over time for constant values of the explanatory variables, that the effect of changing each explanatory variable by a constant value is independent of the time t , and that the effects of the individual explanatory variables multiply together to give the overall effect on hazard function. With these assumptions, a model can be formulated similar to the linear models described in the previous section, but that quantitatively characterise the relationship between each explanatory variable and the hazard. The proportional-hazards regression model is shown in Equation 4.15.

$$\Lambda(t|X) = \Lambda_0(t)e^{\beta_1 X_1 + \dots + \beta_N X_N} \quad (4.15)$$

The coefficients β_i of the regression model give rise to a hazard ratio e^{β_i} for each explanatory variable: the increase in probability of event occurrence at any time t arising from a unit increase in the value of the variable. The significance of each variable's hazard ratio can be evaluated in a similar way¹¹ to the estimation of statistical significance of linear model coefficients.

It is difficult to illustrate performing survival analysis in R without real censored datasets. As these will require a bit more processing, we'll save them for later chapters. But don't worry, complete examples will be provided.

4.9 Projection methods

In datasets that feature a large number of variables, to describe the relationship between entities precisely requires a very high-dimensional space. Imagine we have a measurement for expression of two proteins from each of a collection tissue samples: the measurement of each protein can represent one axis in a

¹¹Although the mathematics are more complicated, as we have to deal with the fact that we have a limited observation of the outcome variable for any cases where the event has not occurred (e.g. when a patient was still alive at the last follow-up).

co-ordinate system in which we can plot all the tissue samples, reflecting all the information we have about the relationships between them. When we have measurements for thousands of genes or proteins per sample, these represent a co-ordinate system in thousands of dimensions. It can be of considerable value to find ways of representing each point in fewer dimensions. This is achieved through constructing ‘latent variables’, which are combinations of each individual dimension of the original dataset. A transformation to latent variables can be especially useful when groups of individual variables are highly correlated with each other, as this can make some multivariate analysis tasks difficult and means that a few latent variables can represent a reasonably large proportion of the information contained in the dataset. By prioritizing latent variables that capture a larger proportion of the information contained in the dataset, it is possible to represent key features with less data.

4.9.1 PCA

The most well-known example of a projection method is Principal Component Analysis (PCA). In technical terms, PCA uses the eigenvector decomposition of the pair-wise covariance matrix to construct a system of orthogonal¹² latent variables which can be ordered by the proportion of the total variation of the dataset that each explains. In practical terms, this means that by using a few tricks of linear algebra, we can find combinations of the original variables which each characterize a distinct aspect of the variation across all the data. These combinations of the original variables are the latent variables, in which the samples can now be represented: this carries the advantage that a few latent variables are a lot more likely to be of value in describing the relationships between the objects; the disadvantage is that these latent variables no longer represent any readily-interpretable features (e.g. level of expression of a particular gene).

Performing PCA for a matrix involves creating a loadings matrix, which has a row for each column of the original matrix and a column for each principal component. The values are the weights by which each object represented by the original matrix (one row) can be represented in a principal component by a weighted sum of all the values. So by multiplying the original matrix by the loadings matrix, a new matrix is obtained with the values for each object *projected* onto each principal component.

It is important to note that representations of high-dimensional entities in only a few latent variables may only capture a small part of the similarities and differences between them across the full set of variables, and should therefore be interpreted with caution. In fact, the proportion of variance in the full dataset

¹²Completely uncorrelated with one another.

explained by each principal component can be calculated relatively easily¹³, and should be reported. Additionally, each principal component is a linear combination of all input variables in the dataset, which is typically a small contribution of many individual variables. It is very difficult to assign a specific real-world interpretation of such a combination of variables, so be wary of over-interpreting correlations between principal components and experimental variables, such correlations don't imply that the component 'means' or 'represents' the correlated variable(s).

We can illustrate some of the utility of performing PCA using the pre-loaded numeric dataset on fertility rate of Swiss towns in 1888. The R function for performing PCA is `prcomp`:

```
> data(swiss)
> swisspc <- prcomp(swiss)
```

The above command (after having loaded the `swiss` dataset into the workspace) applies the `prcomp` function to the resulting numeric dataframe, and stores the result in an object that we have called `swisspc`. This object is a list, and we can get a clue as to what the elements represent by inspecting their names:

```
> names(swisspc)
[1] "sdev" "rotation" "center" "scale" "x"
```

The element `sdev` gives the square roots of the eigenvalues of the principal components. Given the relationship between the eigenvalues and the proportion of variance explained by each component, we can compute this proportion as follows:

```
> (swisspc$sdev2)/sum(swisspc$sdev2)
[1] 7.232244e-01 1.947375e-01 6.255898e-02 1.132090e-02
```

I have only included the first four values here, but this shows that the first principal component explains 72% of the variation in the dataset. The second principal component explains another 19% of the variation in the dataset, and so on.

The element `rotation` gives what are sometimes referred to as the *loadings* of the variables in the dataset onto the principal components. That is, how each principal component is calculated from the individual variables in the dataset. As it is a simple linear combination (i.e. a weighted sum), to find the *projection* of the first entity of the dataset onto the first principal component we just

¹³The proportion of variance explained by each component is its squared eigenvalue divided by the sum of squared eigenvalues for all components.

multiply the first row of the data matrix to the first column of the rotation matrix. With one slight complication: the `prcomp` function first *centers* the columns of the data matrix. But conveniently, the offsets to use (remember that centering subtracts the mean from the data) are given as the element `center` in the output of the `prcomp` function. So let's try this:

```
> sum((swiss[1,]-swisspc$center)*swisspc$rotation[,1])
[1] 37.02011
```

Which conveniently leads to the other output of the `prcomp` function, the element `x`. This is the *projection* of the rows of the data matrix onto each principal component. Let's look at these values for the first 4 rows, for the first 3 principal components:

```
> swisspc$x[1:4,1:3]
PC1 PC2 PC3
Courtelary 37.088222 -14.53971 -24.34887
Delemont -42.758618 -13.77867 -13.34005
Franches-Mnt -51.282839 -16.46204 -25.00101
Moutier 7.427068 -2.54634 -21.61737
```

The element `x` of the list `swisspc` is a matrix, so the above command used square brackets `[` and `]` to index rows 1 to 4 and columns 1 to 3. We should be able to see that the value for the first row in the data table, for the first principal component, is almost identical to the one we calculated earlier.

The projections of elements onto principal components can be useful for providing a lower-dimensionality representation of a dataset. For example, we know that the first two principal components of this dataset reflect over 90% of the variation of the dataset, so considering only those two principal components doesn't lose a large amount of information. Representing entities using only two variables is useful for visualization, because it is easy to produce two-dimensional plots. So let's use the matrix `x` from this PCA calculation to provide `x` and `y` co-ordinates for a plot. We can then use the R function `text` to draw text labels on a plot system, so I'll start with the seemingly unusual step of drawing invisible points in a two-dimensional space:

```
> plot(x=swisspc$x[,1],y=swisspc$x[,2],type='n')
```

Here the argument `type` tells R how to display each point, and setting `type='n'` results in no points being shown. That is what we want, as it creates the co-ordinate system and the graphical window into which we can draw the text labels.

```
> text(rownames(swisspc$x),x=swisspc$x[,1],
+ y=swisspc$x[,2])
```

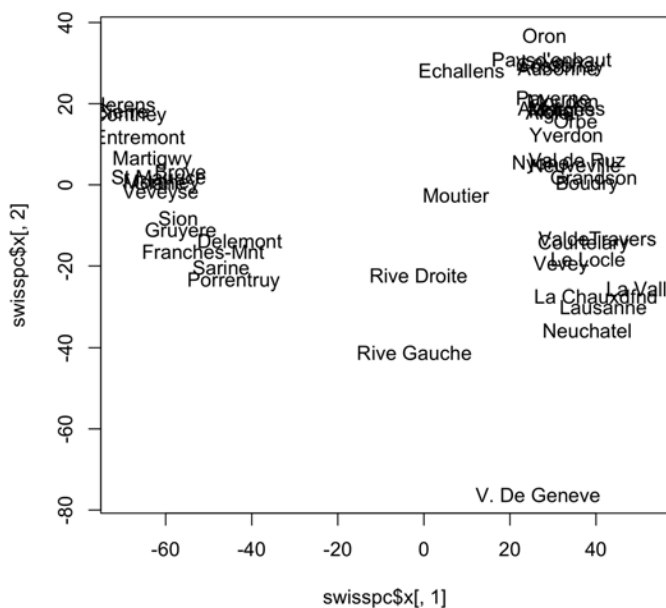


FIGURE 4.9

Projection of Swiss towns from 1888 dataset onto first two principal components from PCA.

In this command, we have used the `text` function with three arguments: the text to draw, for which we have used the row names of the `x` projection matrix; an x-coordinate and a y-coordinate, for which we have used the first and second column of the projection matrix, respectively. The result should appear as in [Fig. 4.9](#).

4.9.2 PLS

A related method called Projection onto Latent Structures (PLS, also known as Partial Least Squares Regression) can be used to find a set of orthogonal latent variables which are ordered on decreasing proportion of variation in some *outcome* variable. (Where in contrast PCA finds latent variables ordered on decreasing proportion of variation across the variables of the input dataset). In a simple two-class case where the outcome variable takes values 0 or 1, this is known as Linear Discriminant Analysis (LDA). Such projections can be especially useful in attempting to build a classifier based on a large number of input variables, whilst reducing the tendency to place too much emphasis on only a few values (which can give rise to over-fitting). As with PCA though, caution should be maintained when interpreting the loadings of the

discriminant/predictive latent structures: a single latent structure may have a large contribution from a few variables, but those contributions will still often only capture a small part of the variation represented by the whole latent variable.

4.10 Resampling: Permutation tests and the bootstrap

If the hypothesis tests we have covered use a specific model for the distribution of a statistic under a null hypothesis, what can we do if we don't know of an appropriate model to use for the statistic? Computing power gives us one possible solution, which is to use available data to create random samples under the null hypothesis, and to use many such samples to estimate the distribution of the statistic. This approach is called *resampling*, and the estimated distribution is known as the *empirical* distribution, because we are not explicitly modelling the distribution, just assuming that our randomly-generated sample under the null hypothesis is representative of the whole population.

Reverting to our earlier example, we could take this approach if we wished to test how unlikely a sample of 10 answers to a yes/no question resulted in as uneven a distribution as 7 to 3, if the probability of any response being 'yes' or 'no' was equal (both 0.5). Remember that our statistic of interest was $S = Y - (10 - Y)$, so compute this statistic from a large number of (say 1000) random samples, each selecting 10 answers from 'yes' or 'no' randomly and with equal probability. We will then have 1000 values of the statistic S , representing its distribution under the null hypothesis, and so our p-value is just the proportion of those 1000 values which are greater than or equal to our observed value. Putting this formally, and adding 1 to both parts of the fraction to make sure we never estimate a p-value of 0:

$$Pr(S \geq s) = \frac{1}{1+n} + \sum_{i \in \{1, \dots, n\}} S_i \geq s, \text{ for large } n. \quad (4.16)$$

One approach for generating samples under a null hypothesis is to *permute* the data (labels). A permutation is a random re-ordering. So if we were using a correlation coefficient to assess the association between two variables observed for a set of samples, we could re-order one of the variables and compute the correlation coefficient. Repeating this a large number of times will give us an empirical distribution for the correlation coefficient under a null hypothesis of no association. Such permutation tests are very helpful for estimating statistical significance of observations from data for which there is no standard hypothesis test, or for which the assumptions in such tests are expected to be invalid. But note, it relies on having a large number of possible combinations of the observed variables. If we only have data from a few samples to re-order,

we will only ever get a limited number of values for the statistic, no matter how many permutations we perform.

4.11 Stability and robustness

One thing to consider with any application of a statistical method, is how *stable* were the results? In this context, stability represents the extent to which the results would change if the input values changed. Obviously, if our observed data (e.g. sample from a random variable) changes, the results of a given hypothesis test are likely to change. And the distribution of objects across clusters is likely to change. But by how much? If a small change in one value from the observed data makes a big difference in the outputs from a statistical analysis method, then this may suggest that we should interpret the result with caution. It is often possible to try this yourself, deliberately making small changes or excluding some values for some of the objects being measured, repeating your analysis and checking that the results aren't completely different.

4.12 Summary

The limited set of statistical tools described in this document can be an incredibly powerful means of discovering patterns and making inferences based on the wealth of biological data provided by high-throughput platforms, particularly in the context of publicly available resources.

As detailed in the accompanying set of tutorials, the R statistical programming environment provides a means of performing the analyses described above. If a potential user of these tools does not understand the essential mathematical concepts presented in this document, they are strongly advised to study further reference material in order to understand the results of data analyses!

Bibliography

- [1] D.R. Cox, "Regression models and life-tables," *J. Royal Stat. Soc. B* 34(2):187-220 (1972).

- [2] M. Eisen *et al*, “Cluster analysis and display of genome-wide expression patterns,” *PNAS* 95:14863-14868 (1998).
- [3] A. Hartemink *et al*, “Maximum likelihood estimation of optimal scaling factors for expression array normalization,” *Proceedings of SPIE* 4266:132 (2001).
- [4] R. Irizarry *et al*, “Exploration, normalization and summaries of high density oligonucleotide array probe level data,” *Biostatistics* 4:249-264 (2003).
- [5] J. Quackenbush, “Computational analysis of microarray data,” *Nature Reviews Genetics* 2:418-427 (2001).

5

Analyzing Generic Tabular Numeric Datasets in R

5.1 Introduction

This chapter will give an example of some of the basic data-handling aspects of R, and how these can be used to work with tables of numerical data that arise very frequently in biological research. Though each worked example in this tutorial is a specific case, the methods used can be extrapolated to other situations and thus the examples are intended to illustrate how certain features of R can be used to facilitate analysis of virtually *any* numeric data table. Such data tables are typically plain text files (the file extension is usually *.txt* or *.csv*, but in fact it can be anything) with a row on each line and the boundaries of each column marked by some *delimiter* (most commonly a tab or comma).

5.2 Loading data into R

Obviously, a critical step in using R to perform numerical analysis of a dataset is loading the data into the R workspace. There is one function that can be used in most circumstances, which is `read.table`. An important fact that should always be remembered when attempting analysis of tabular data is that R will convert any table of values read into the workspace from a file into a *dataframe*: every row **must** have an entry for every column, and vice versa, or the table will not be read in. In a practical sense, this means that any blank or missing values should be explicitly entered 'NA'. It can often be helpful to use a spreadsheet program like Excel to produce the table.

To provide an accessible example from molecular biology, we will obtain data from Iorns *et al* (2009), an siRNA screen performed on a panel of cancer cell lines¹. The study was published in PLoS ONE, and a table giving the changes in cell viability following transfection of each of 779 siRNAs is freely available

¹<http://www.plosone.org/article/info%3Adoi%2F10.1371%2Fjournal.pone.0005120>

GENE	Ac	Z Scores	MCF7	HELA	CALS1	A549	H226
AAK1	NM_014911	1.00	0.01	-1.27	-0.47	-0.69	
AATK	XM_375495	-0.37	-1.43	-0.76	-0.44	-0.16	
ABI1	NM_005470	-1.51	0.67	-2.64	-1.02	-2.17	
ABL1	NM_007313	-0.89	0.40	0.03	-0.94	-0.22	
ABL2	NM_005158	1.05	0.16	0.36	0.06	-1.16	
ACK1	NM_005781	0.00	-0.20	-0.01	-1.68	-0.36	
ACVR1	NM_001105	1.90	0.52	0.14	0.28	-0.30	
ACVR1B	NM_004302	0.21	0.53	-0.27	0.34	0.03	
ACVR1C	NM_145259	1.74	0.35	-0.37	0.51	-0.35	
ACVR2	NM_001616	1.26	0.79	-0.27	0.79	0.17	
ACVR2B	NM_001106	0.53	1.52	-0.23	0.82	0.81	
ACVRL1	NM_000020	0.69	0.80	-0.64	0.68	1.25	
ADAM9	NM_003816	0.46	0.75	0.36	0.77	0.01	
ADCK1	NM_020421	0.07	0.94	-0.03	0.12	0.42	
ADCK2	NM_052853	-3.06	0.74	0.34	-0.59	-1.99	
ADCK4	NM_024876	0.81	0.82	0.18	-0.01	-1.16	
ADCK5	NM_174922	-0.67	0.42	-2.29	0.67	0.49	
ADK	NM_001123	1.86	0.89	0.25	-0.12	1.00	
ADRA1A	NM_000680	0.44	-0.63	-1.64	-0.35	0.48	

FIGURE 5.1

Excel table of siRNA screen data.

from the journal's website. We will download Table S1², which should open in any spreadsheet program³ and appear as in Fig. 5.1.

Now if we copy the cells making up the data table (C5:I784) and paste this into a new sheet, as in Fig. 5.2, then we can save the sheet as a tab-separated text file. Be sure to specify the file extension when naming the file: let's call it *siRNAscreen_data.txt*.

Now we can open an R workspace and read in the table. Make sure the working directory is set to the folder where you saved the siRNA screen data table, then enter the following command:

```
> siRNA.screen <-read.table("siRNAscreen_data.txt",sep="\t",header=TRUE)
```

The `read.table` function takes a number of arguments, some of which we haven't used here, but the most important ones are included.

- **file**: the full name of the file to read in (specified within double-quotes)
- **sep**: the field delimiter (specified within double-quotes)⁴

²<http://www.plosone.org/article/fetchSingleRepresentation.action?uri=info:doi/10.1371/journal.pone.0005120.s001>

³For a spreadsheet program I have used Microsoft Excel, but free alternatives include OpenOffice or Google Sheets.

⁴You may notice that a tab is encoded as "\t". A comma or space can simply be entered as is (", " or " ", respectively), and a new line is encoded "\n".

GENE	Ac	MCF7	HELA	CAL51	A549	H226
AAK1	NM_014911	1.00	0.01	-1.27	-0.47	-0.69
AATK	XM_375495	-0.37	-1.43	-0.76	-0.44	-0.16
ABI1	NM_005470	-1.51	0.67	-2.64	-1.02	-2.17
ABL1	NM_007313	-0.89	0.40	0.03	-0.94	-0.22
ABL2	NM_005158	1.05	0.16	0.36	0.06	-1.16
ACK1	NM_005781	0.00	-0.20	-0.01	-1.68	-0.36
ACVR1	NM_001105	1.90	0.52	0.14	0.28	-0.30
ACVR1B	NM_004302	0.21	0.53	-0.27	0.34	0.03
ACVR1C	NM_145259	1.74	0.35	-0.37	0.51	-0.35
ACVR2	NM_001616	1.26	0.79	-0.27	0.79	0.17
ACVR2B	NM_001106	0.53	1.52	-0.23	0.82	0.81
ACVRL1	NM_000020	0.69	0.80	-0.64	0.68	1.25
ADAM9	NM_003816	0.46	0.75	0.36	0.77	0.01
ADCK1	NM_020421	0.07	0.94	-0.03	0.12	0.42
ADCK2	NM_052853	-3.06	0.74	0.34	-0.59	-1.99
ADCK4	NM_024876	0.81	0.82	0.18	-0.01	-1.16
ADCK5	NM_174922	-0.67	0.42	-2.29	0.67	0.49
ADK	NM_001123	1.86	0.89	0.25	-0.12	1.00
ADRA1A	NM_000680	0.44	-0.63	-1.64	-0.35	0.48

FIGURE 5.2

Excel table of siRNA screen data.

- **header:** a logical value indicating whether or not the first row contains column headers rather than data

Now we can inspect the first few rows of the data frame `siRNA.screen` that has been created read in from the file using the `head` function:

```
> head(siRNA.screen)
  GENE      Ac MCF7. HELA. CAL51. A549. H226.
1 AAK1 NM_014911  1.00  0.01  -1.27 -0.47 -0.69
2 AATK XM_375495 -0.37 -1.43  -0.76 -0.44 -0.16
3 ABI1 NM_005470 -1.51  0.67  -2.64 -1.02 -2.17
4 ABL1 NM_007313 -0.89  0.40   0.03 -0.94 -0.22
5 ABL2 NM_005158  1.05  0.16   0.36  0.06 -1.16
6 ACK1 NM_005781  0.00 -0.20  -0.01 -1.68 -0.36
```

In addition to the cell viability z-scores, this data frame contains both the Gene Symbols and RefSeq IDs for each siRNA's target. For our calculations it can often be helpful to separate annotation information from the purely numerical data. For this purpose, we will create a separate table called `siRNA.zscores`. As these values are all numeric, we can convert the resulting data frame to a two-dimensional *numeric* array using the `as.matrix` function. Finally, we can annotate the rows of this array with the corresponding GeneSymbol.

```
> siRNA.zscores <- as.matrix(siRNA.screen[,-c(1,2)])
> rownames(siRNA.zscores) <- as.character(siRNA.screen$GENE)
```

5.3 Data visualisation

In the initial stage of data analysis, it can be very helpful to examine your data visually to check that there are no obvious anomalies. We will investigate a number of characteristics of the dataset using a few simple techniques.

5.3.1 Scatter plots

The cell viability measurements can simply be plotted as numeric values using the `plot` function. In the following example, we compare the z-scores following each transfection in MCF7 cells with the corresponding transfection in HeLa cells. The result should be the plot given in [Fig. 5.3](#).

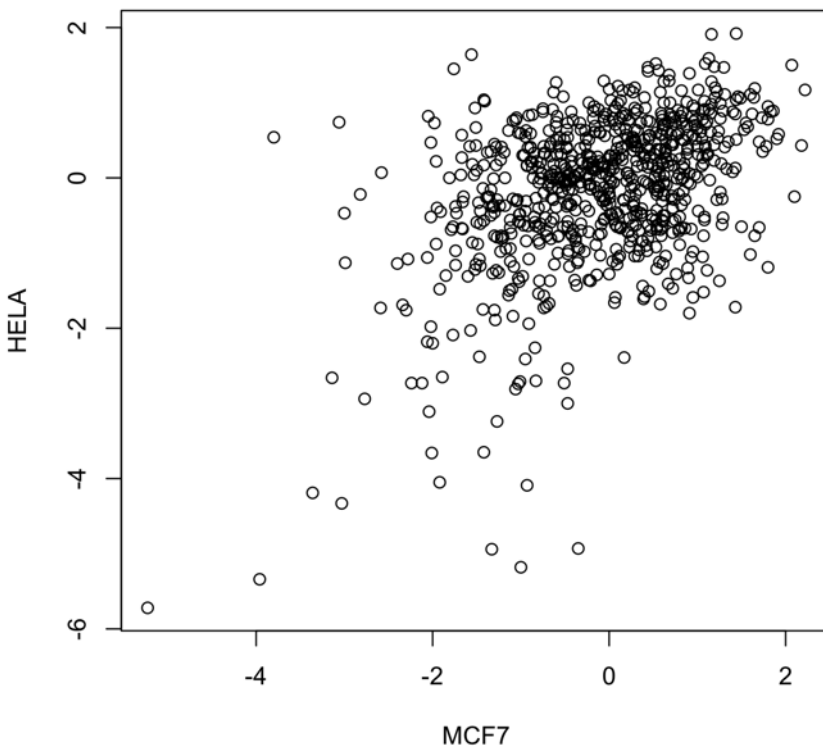


FIGURE 5.3

Scatter plot showing the distributions of cell viability z-scores of the MCF7 and HeLa cell lines under each siRNA transfection.

```
> plot(x=siRNA.zscores[,1],y=siRNA.zscores[,2],  
+ xlab=colnames(siRNA.zscores)[1],ylab=colnames(siRNA.zscores)[2])
```

5.3.2 Box plots

In this dataset we have a lot of measurements for each cell line. If the assay is unbiased and comprehensive enough, we may expect that, averaged across all the siRNAs, the cell lines would have similar sensitivities. We can inspect this by producing box plots showing the distribution of *all* cell viability z-scores for each cell line, using the `boxplot` function. The result should appear as in Fig. 5.4.

```
> boxplot(lapply(c(1:5),function(x)siRNA.zscores[,x]),  
+ names=colnames(siRNA.zscores))
```

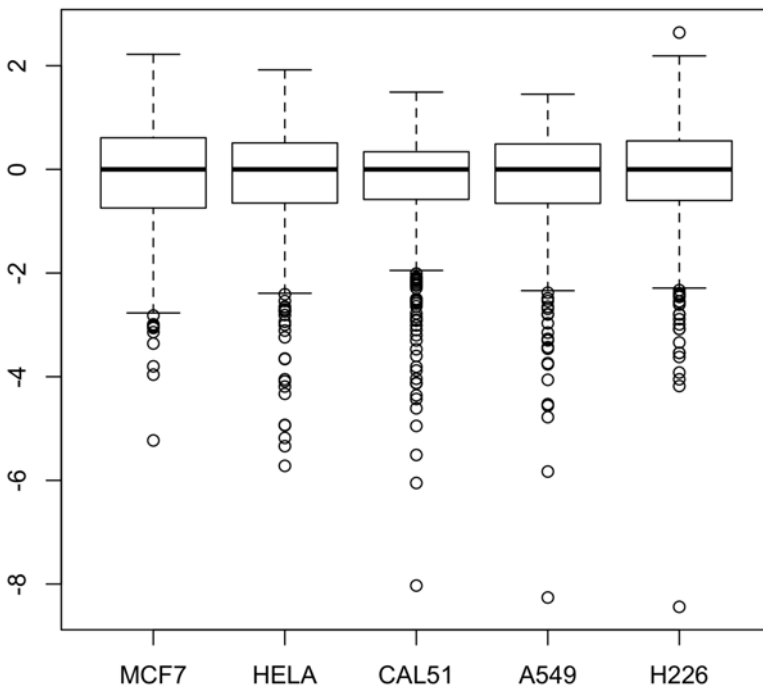


FIGURE 5.4

Box-and-whiskers plot showing the distributions of cell viability z-scores for each cell line, over all the siRNA transfections.

In the above command, we have used the `lapply` function to convert each column of the `siRNA.zscores` object into a separate element of a *list* object, which is the required argument of the `boxplot` function. This is done using the `lapply` function by applying, to each number from 1 to 5, a simple function that outputs the corresponding column from the `siRNA.zscores` object.

These box plots show that the distribution of scores for each cell line seem to have been centred and scaled so that they each have a median of zero and a similar range. You can use the `apply` function to compute different statistics for each column of the matrix of z-scores to confirm this. For example, find the median value from each cell line:

```
> apply(siRNA.zscores,MARGIN=2,median)
```

And the standard deviation:

```
> apply(siRNA.zscores,MARGIN=2,sd)
```

Had inspection of the distributions shown systematic differences between some of the cell lines, it might have been worth applying some normalization procedure to minimize the potential impact of technical biases that could cause such differences. For example, if one cell line has a low transfection rate, this will affect all (or at least most) of the shRNA constructs' viability scores, yet is not necessarily related to sensitivity of the cell line to knock-down of each target gene.

5.3.3 Bar charts

Bar charts represent a common way to illustrate visually differences in values. For example, we can use the `barplot` function to create a bar chart showing the first row of the data matrix: this shows the cell viability scores of each cell line following transfection with AAK1 siRNA. The resulting plot should appear as in [Fig. 5.5](#).

```
> barplot(siRNA.zscores[1,])
```

5.4 Correlation and clustering

Statistical correlation measures provide a means of assessing the similarity between trends in data. Clustering is generally the task of associating similar entities from a dataset: this could involve finding groups of genes with similar expression levels across a set of samples, or it could involve finding groups of

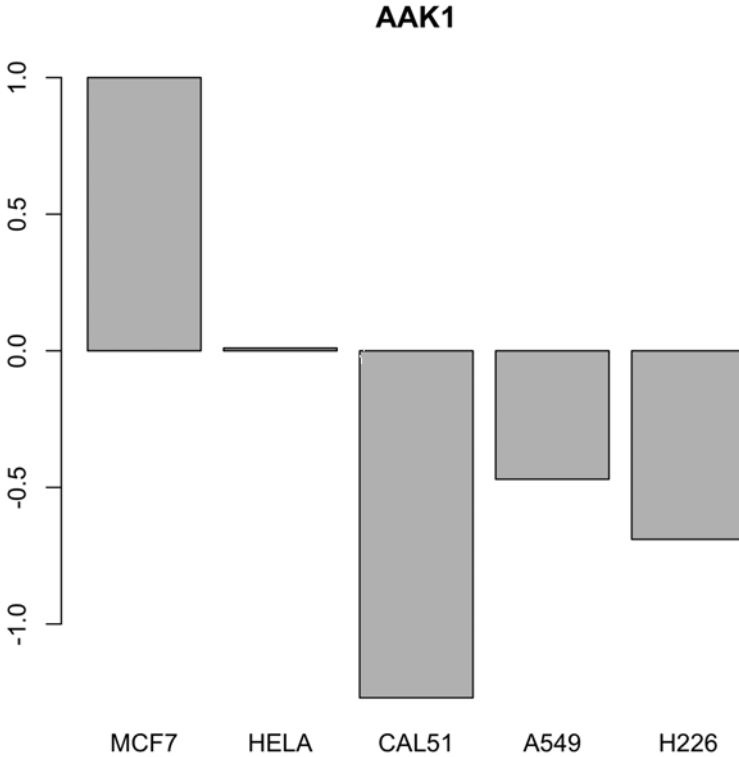


FIGURE 5.5
Bar chart showing the cell viability z-scores for each cell line, following transfection of siRNA to AAK1.

samples with similar expression levels of certain sets of genes. Both correlation and clustering are therefore concepts that can help us with the task of characterising similarity between elements from a dataset, but they are quite different concepts and must be applied in different ways.

5.4.1 Correlation

There are many different effects that can manifest in correlated data. In this siRNA screen, genes with correlated profiles (similar trends in cell viability z-scores across the panel of cell lines) may be part of the same pathway, such that additions to certain pathways may result in a number of genes being synthetically-lethal in combination with some characteristic of the cell lines.

If we pick a gene of interest, say PIK3CA, we can calculate correlation coefficients of any other genes' viability profiles with that of PIK3CA, using the

`cor` function. Approximate statistical significance can also be obtained using the `cor.test` function, with the caveat that this assumes normally distributed data and there are so few samples that it is impossible to tell how unrealistic this assumption may be. It may be useful in our research to test *all* features for their similarity in trend (changes across the dataset) to a particular feature of interest. In this example, we can evaluate the correlation of each gene in turn for its linear correlation with PIK3CA. This will be done by calculating each gene's correlation coefficient, the corresponding p-value, and an adjusted p-value taking into account the fact that we have performed multiple statistical tests (using the Benjamini-Hochberg method). Finally, the results will be summarized in a data frame. Given that we want to apply the same function (computing the Pearson correlation coefficient) to each row of the z-score matrix in turn, this example will make use of the `apply` function. For a recap of this function, see Section 2.5.3.

```
> PIK3CA.cors <- apply(siRNA.zscores,MARGIN=1,function(x)
+ cor(x,siRNA.zscores[which(rownames(siRNA.zscores)=="PIK3CA"),]))
```

Here we use the `apply` function to apply another function to each row of the data table in turn. In this case, the function being applied computes the correlation coefficient between a set of values and the cell viability z-scores from PIK3CA siRNA transfection.

If you didn't want to use the `apply` function, we could have obtained the same result using a `for` loop, having first created a vector of missing values to store the results for each row of the table:

```
> PIK3CA.cors <- rep(NA,nrow(siRNA.zscores))
> for(i in 1:nrow(siRNA.zscores)){
+ PIK3CA.cors[i] <- cor(siRNA.zscores[i,],
+ siRNA.zscores[which(rownames(siRNA.zscores)=="PIK3CA"),])}
```

Next, we can use a similar approach (either `apply` or a `for` loop) to compute the p-value estimates for hypothesis of the two tested genes' z-scores being more correlated than would be expected for two variables *independently* sampled from normal distributions. Here is the way to calculate these estimates using `apply`:

```
> PIK3CA.corPvals <- apply(siRNA.zscores,MARGIN=1,function(x)
+ cor.test(x,siRNA.zscores[which(rownames(siRNA.zscores)=="PIK3CA"),
+ ])$p.value)
```

Again `apply` is used, but this time we obtain the p-value from the correlation test by selecting the `p.value` field from the result of `cor.test` using the `$` symbol.

To perform the same calculations without the `apply` function, but using a `for` loop instead:

```
> PIK3CA.corPvals <- rep(NA,nrow(siRNA.zscores))
> for(i in 1:nrow(siRNA.zscores)){
+ PIK3CA.corPvals[i] <- cor.test(siRNA.zscores[i,],
+ siRNA.zscores[which(rownames(siRNA.zscores)=="PIK3CA"),])$p.value}
```

Whichever approach was used to compute these results, we can now create a data frame to store the correlation test results along with the gene names:

```
> PIK3CA.df <- data.frame(Gene=rownames(siRNA.zscores),
+ cor=PIK3CA.cors,p.value=PIK3CA.corPvals)
```

This command creates the data frame listing the gene names, correlation coefficients and p-values.

```
> PIK3CA.df <- PIK3CA.df[order(PIK3CA.df$p.value,decreasing=FALSE),]
> PIK3CA.df$adj.p.val <- p.adjust(PIK3CA.df$p.value,method="BH")
```

Following sorting of the table on p-value, the adjusted p-values are calculated using the `p.adjust` function, then added to the table. The first few rows of the table should now look like this:

```
> head(PIK3CA.df)
Gene cor p.value adj.p.val
534 PIK3CA 1.0000000 0.0000000000 0.00000000
209 DUSP22 0.9978449 0.0001200562 0.04676191
619 PTPN5 -0.9907038 0.0010744492 0.23591179
694 STK22C -0.9895644 0.0012777074 0.23591179
302 HIPK1 -0.9882775 0.0015209038 0.23591179
516 PFKFB2 -0.9867994 0.0018170356 0.23591179
```

It is always a good idea to inspect the underlying data giving rise to a result from statistical analysis. In this case, a line graph for the most correlated gene (DUSP22) and PIK3CA itself may be informative. This will give rise to the plot shown in [Fig. 5.6](#):

```
> plot(siRNA.zscores[which(rownames(siRNA.zscores)=="PIK3CA"),],
+ type="l",col="red",ylab="z-score",xlab="cell line")
> points(siRNA.zscores[which(rownames(siRNA.zscores)=="DUSP22"),],
+ type="l",col="blue")
> legend("topright",legend=c("PIK3CA","DUSP22"),lty=c(1,1),
+ col=c("red","blue"))
```

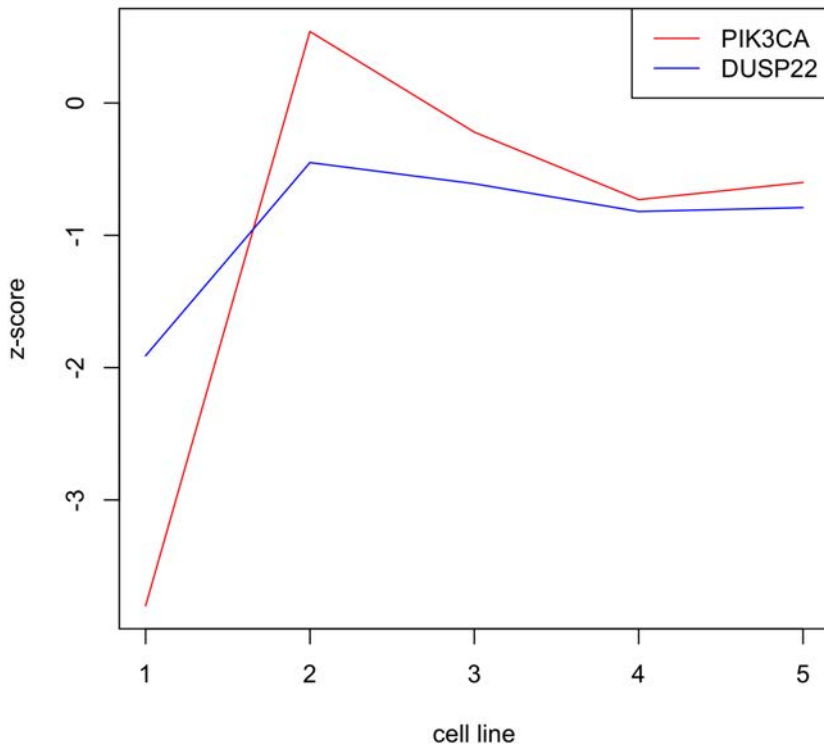


FIGURE 5.6

Line graph showing profiles of cell line viability effect of siRNA transfection for PIK3CA and DUSP22.

5.4.2 Clustering

Clustering is the principle of grouping similar entities together, but in any situation there are typically many ways of defining the notion of similarity⁵. Any individual approach will have strengths and weaknesses, and subtleties associated with correct interpretation of the results. For simplicity's sake, we will just use one type of approach: hierarchical clustering. This is actually a clustering task, utilizing a dissimilarity measure (a distance metric) and a linkage method. The examples provided here will use complete linkage method (the default implementation in R's `hclust` function) with two different distance metrics, in order to demonstrate some use in exploratory data analysis.

The first metric we will try is the Euclidean distance, which is the sum of the squared differences in each individual characteristic's value from one entity

⁵In fact, most clustering algorithms use *dissimilarity*, rather than similarity, but this is a purely technical distinction.

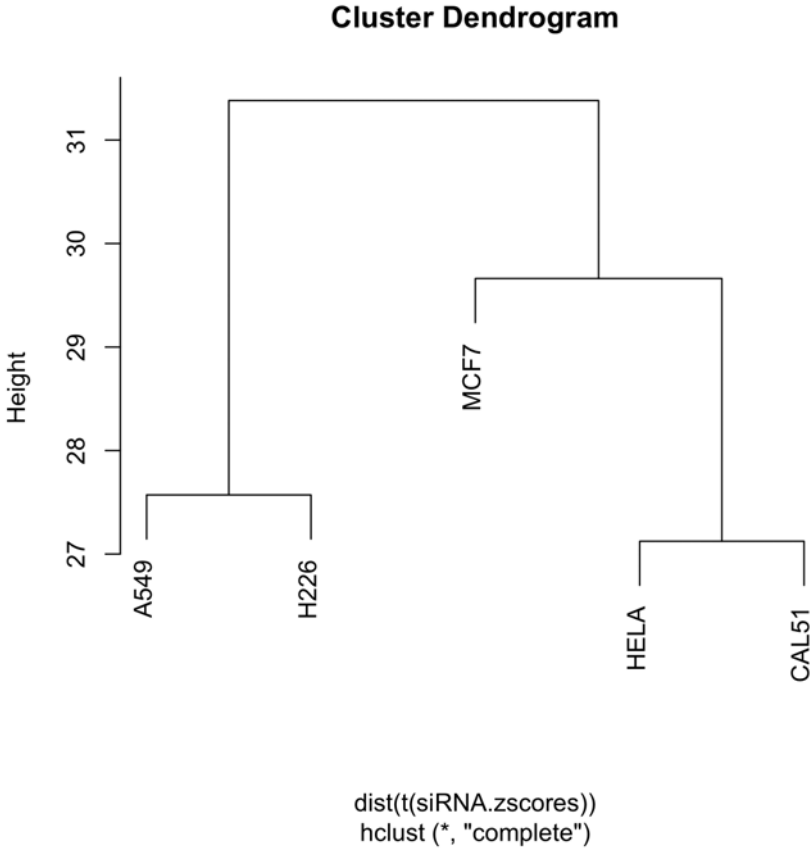


FIGURE 5.7

Hierarchical clustering dendrogram showing similarity between cell lines in terms of their shared siRNA sensitivities, based on Euclidean distance.

(set of values) to the other. The `dist` function in R creates a matrix of all the pairwise Euclidean distances between the rows of a data table. The result can be passed to the hierarchical clustering function `hclust`, which will produce a cluster dendrogram based on the distance matrix, as in [Fig. 5.7](#).

```
> plot(hclust(dist(t(siRNA.zscores))))
```

Note, we use the matrix transpose function `t()` to generate distance matrix for the columns, rather than the rows, of the siRNA screen data table.

As the Euclidean distance is based on adding squared differences in individual measurements, certain measurements can contribute to the overall distance a

lot more than others. In some situations, giving more weight to larger absolute differences (which typically come from high-valued measurements) is preferred, but in this case we may not wish for any individual siRNAs to contribute too much to the overall distances. An alternative approach is to use correlation as a measure of similarity, and as we know a correlation coefficient will lie between -1 and 1 , subtracting the correlation coefficient from 1 will give us a single distance for any two sets of values. More similar sets of values have higher correlation coefficients and thus lower distances. We can re-create the hierarchical clustering dendrogram now using the correlation-based distance, giving rise to [Fig. 5.8](#).

```
> plot(hclust(as.dist(1-cor(siRNA.zscores))))
```

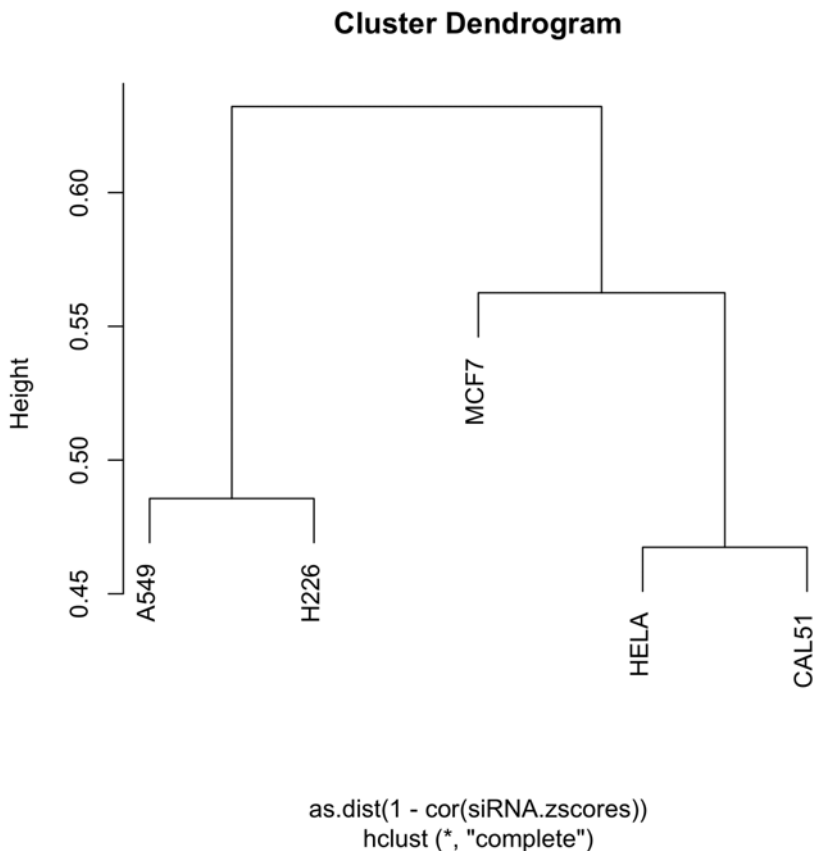


FIGURE 5.8

Hierarchical clustering dendrogram showing similarity between cell lines in terms of their shared siRNA sensitivities, using correlation-based distance.

In the above command, the `cor` function produces a matrix of the pair-wise column-to-column correlation coefficients from the data table, and the `as.dist` function has to be used to convert the matrix into the object required by the `hclust` function. Note, this example bases the dissimilarity values on the Pearson correlation coefficients, but it would be possible to use the Spearman correlation coefficients instead, by setting the `method` argument in the `cor` function: `cor(siRNA.zscores,method="spearman")`.

When the resulting dendrogram (Fig. 5.8) is compared with that from Fig. 5.7, we see that the similarities between HeLa and CAL51, and between A549 and H226, are consistent across these two definitions of similarity. This exploratory analysis suggests that it may be interesting to look at what genes are most clearly responsible for these similarities, to provide some biological insight on the similarity of gene additions in the different cell lines.

5.4.3 Heatmaps

Heatmaps represent a useful tool to assist visual inspection of large tables of values, particularly in conjunction with clustering techniques that group together similar rows and columns from within the table. A heatmap consists of a grid of coloured blocks, with each block representing a single measurement. The utility comes from defining a colour scheme such that low-valued measurements show up in one colour, high-valued measurements in another, and the shade of the colour indicates where on this scale each value lies.

As a purely illustrative example, we will construct a heatmap of *all* values from the siRNA screen. I personally find that the best way to draw heatmaps in R is using the `aheatmap` function from the *NMF* package. This must first be installed (if not already):

```
> install.packages("NMF")
```

Then the package can be loaded into the workspace:

```
> library(NMF)
```

Finally, the heatmap command can be entered:

```
> aheatmap(siRNA.zscores,Rowv=NA,scale="row")
```

You can see the result in Fig. 5.9, which shows that there are some patterns in the sensitivities of the cell lines to the knock-down of individual genes, represented by visible ‘blocks’ of colour. However, there are so many genes plotted that this particular heatmap probably doesn’t improve our understanding of the cell lines, or the genes in the screen, in any great way. A heatmap is often a more informative tool when some selection of the data is

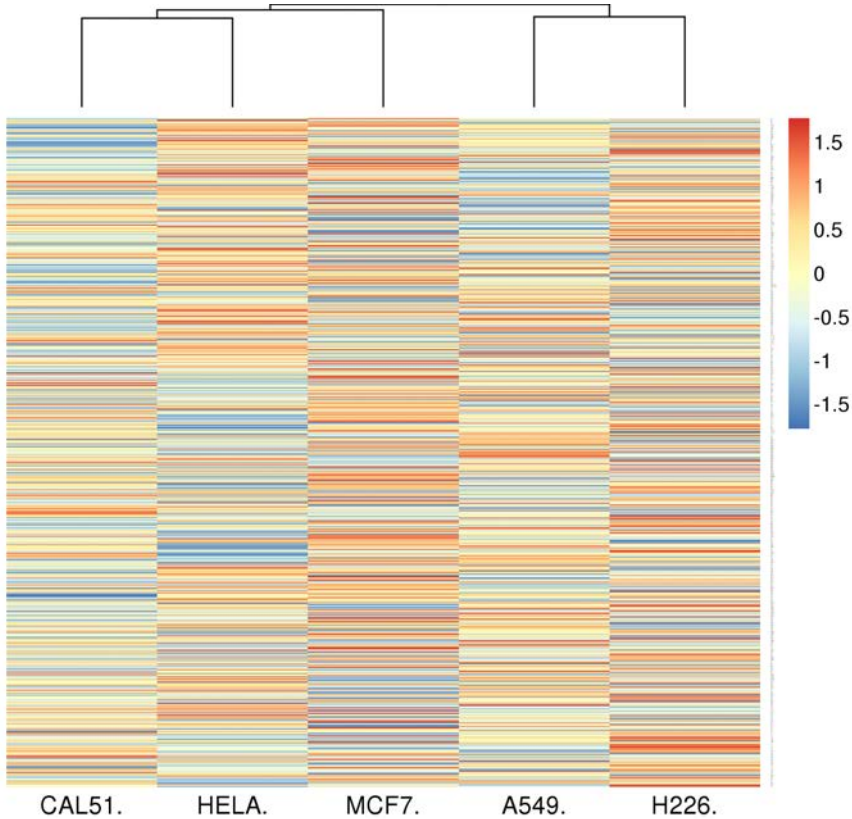
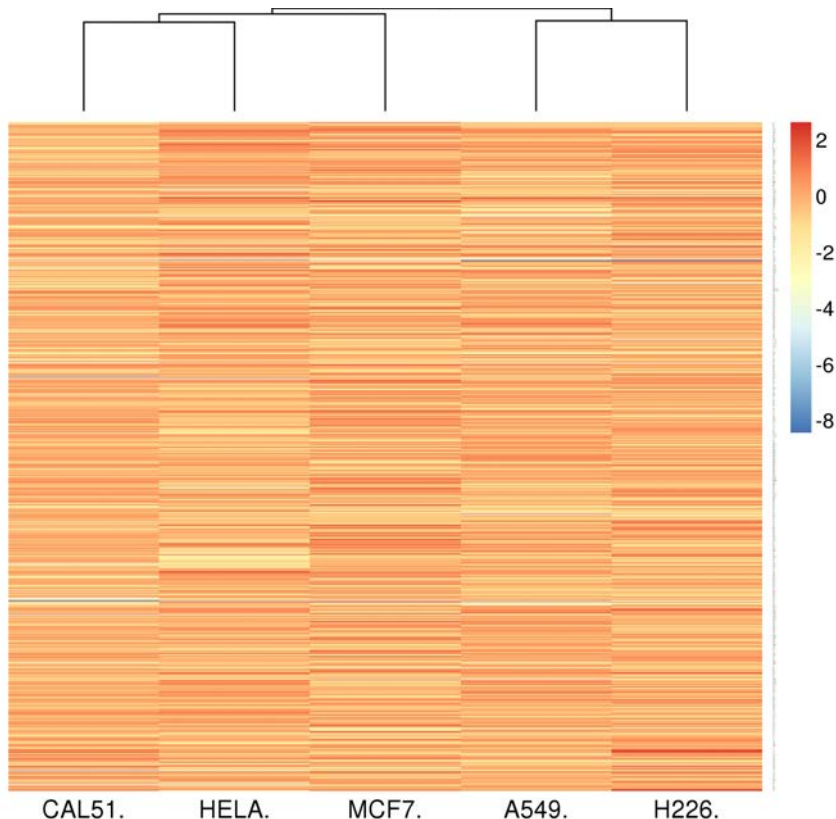


FIGURE 5.9

Heatmap showing clustering of cell viability effects from sets of siRNAs in each cell line.

applied, in order to extract features of interest. We will see such an application in the following section.

By default, the `heatmap` function clusters both rows and columns of the table. While we do want this, the gene-wise (row) dendrogram involves too many elements to be informative, so we have instructed the function to draw only the column dendrogram. This is achieved by passing the argument `Rowv=NA`. Finally, scaling the values on each row (with `scale="row"`) forces the heatmap to show *relative* values for each gene. Skipping this row-wise scaling (setting `scale="none"`) offers the advantage of the fact that a given colour in the grid implies a single underlying value, regardless of what gene the measurement relates to. However, it allows the possibility of a single outlying value to skew the colour scale sufficiently that most of the plot is meaningless. By way of

**FIGURE 5.10**

Heatmap showing clustering of cell viability effects from sets of siRNAs in each cell line, with an absolute (rather than relative) colour scale.

an example, Fig. 5.10 shows the same heatmap as Fig. 5.9 but without the row-wise scaling, generated with the following command:

```
> aheatmap(siRNA.zscores,Rowv=NA,scale="none")
```

5.5 Statistical analysis using linear models

Even though *limma* stands for ‘Linear Models for Microarray Analysis’, the package from Bioconductor⁶ can be used to apply the linear modelling framework for statistical analysis to any numerical dataset, not just microarrays⁷.

⁶<http://bioconductor.org>

⁷Later on we’ll make use of some of its functionality designed specifically for sequencing data.

It is particularly suited to ‘high-dimensional’ datasets that have a large number of measurements taken from a relatively small sample set, as is the case in this siRNA example. If the reader is unfamiliar with the use of linear models (and the *limma* package), you are referred to Section 4.6. There are more applications of the *limma* package in [Chapter 8](#), so if it still seems a bit obscure don’t worry, there will be more opportunities for it to make sense.

5.5.1 Comparison of two groups

It was observed in the clustering analysis that the HeLa and CAL51 cell lines shared similarities in terms of siRNA sensitivities, as did A549 and H226. One question we may have is ‘what genes are primarily responsible for these similarities and differences?’. This is a standard two-class problem in which we wish to find measurements that have consistent, clearly different values across two sets of entities. In this case the entities are the cell lines, and the measurements are the cell viability z-scores. If we define the HeLa and CAL51 cell lines as one group and the A549 and H226 cell lines as another group, we can fit a linear model for each gene, with the relative viability effect described as a linear function of a group membership variable (taking value 1 for one of the groups and 0 for the other group). First, a *design matrix* needs to be created to define the values of the explanatory variables (in this case, an intercept term and the group membership). Then we can use the `lmFit` function from the *limma* package to fit linear models to the data and use these to evaluate statistical significance that each siRNA’s viability score clearly discriminates between the two groups. The statistics extracted from such linear models for two-class analysis are t-statistics, and thus this is equivalent to performing a t-test.

Let’s first look at the column names of the `siRNA.zscores` matrix, to decide which of the columns correspond to each of our two groups.

```
> colnames(siRNA.zscores)
```

We should see:

```
[1] "MCF7." "HELA." "CAL51." "A549." "H226."
```

So the first column isn’t in either group (we should exclude it from our analysis), the 2nd and 3rd columns belong to the first group, and the 4th and 5th columns belong to the second group. If we exclude the first column from the table, then our explanatory variable of interest could take value 0 for the first two columns of the new table and 1 for the last two columns. We will use the `cbind` function to create the *design matrix*, as in section 6 of [Chapter 4](#). Using this function, we can give names to the columns we are binding together:

```
> design2 <- cbind(intercept=1,grp2=c(0,0,1,1))
```

So we have created an array called `design2` by binding together two vectors: one, called `intercept` always takes value 1; the other, called `grp2`, is created by concatenating (the function `c`) together two 0s and then two 1s.

Now we can use the `lmFit` function to fit the linear models for every row of the `siRNA.zscores` matrix, remembering that we wish to exclude the first column:

```
> grpfit <- lmFit(siRNA.zscores[,2:5], design=design2)
```

The `lmFit` is the main numerical part of the data analysis, in which linear models are fitted for each row in the table to see how closely their values match each output variable defined in the design matrix.

We then want to use the `eBayes` function performs empirical-Bayes moderation of the t-statistics, essentially ‘borrowing’ information across the models: generally, this is a good idea when there is a large number of rows in the table and relatively few columns. And finally the `topTable` function displays a table of the elements from the table with the most statistically significant association with the specified model and contrast, as shown in [Fig. 5.11](#). We can see that the gene with the most consistent difference between these two groups of cell lines is `WEE1`, which features prominently in the Iorns *et al* 2009 paper from which the data was obtained.

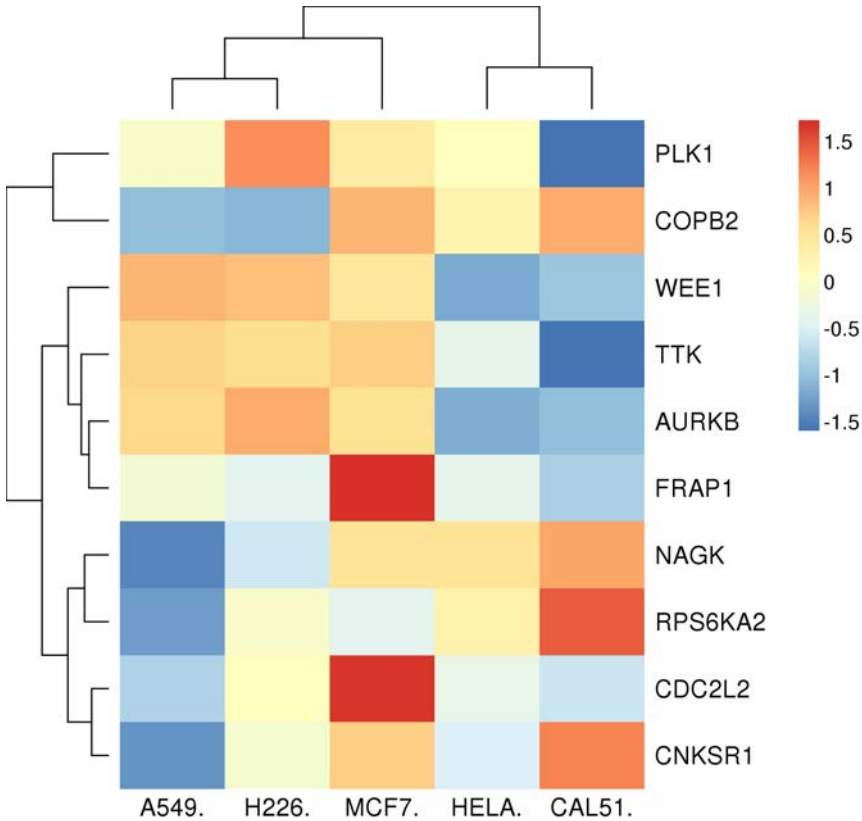
```
> grpfit2 <- eBayes(grpfit)
> topTable(grpfit2, coef=2)
```

Having identified a list of genes that discriminate the HeLa & CAL51 cell lines from A549 & H226, we can use a heatmap to visualise the patterns involved:

```
> topTable(grpfit, coef=2)
  ID logFC AveExpr      t      P.Value  adj.P.Val      B
772 WEE1  4.895 -2.4475  9.125515 1.125442e-05 0.008767192  2.9943763
162 COPB2 -3.775 -6.4625 -5.918846 2.805102e-04 0.109258733  0.6673013
745 TRAD -2.525 -0.9375 -4.668252 1.367310e-03 0.272786504 -0.6587367
58  AURKB  2.430 -2.3450  4.605205 1.489743e-03 0.272786504 -0.7330198
458 NAGK -2.925 -2.3425 -4.487610 1.750876e-03 0.272786504 -0.8735038
506 PCTK3  2.850 -0.9500  4.066144 3.175911e-03 0.306835362 -1.3973856
299 GUK1  3.525 -2.7275  4.043858 3.279848e-03 0.306835362 -1.4259543
68  BLNK -2.235 -1.2575 -4.031645 3.338345e-03 0.306835362 -1.4416472
562 PMVK -2.470 -1.7750 -3.939318 3.818288e-03 0.306835362 -1.5610979
320 IKBKE  2.025 -0.4975  3.868166 4.238209e-03 0.306835362 -1.6541251
```

FIGURE 5.11

Annotated table of siRNAs with different effects on cell viability in HeLa & CAL51, compared to A549 and H226.

**FIGURE 5.12**

Heatmap showing clustering of cell viability effects from siRNAs that best discriminate HeLa and CAL51 from A549 and H226.

```
> aheatmap(siRNA.zscores[topTable(grpfit2)$ID,], scale="row")
```

Here we obtain the gene names from the output of the `topTable` function using the `$` operator, and use these names to index the corresponding rows from the siRNA screen data table. The resulting heatmap should appear as in [Fig. 5.12](#).

5.5.2 Alternative models

The MCF7 cell line doesn't appear to cluster particularly closely with any other cell lines, so it might be interesting to find genes that are specifically critical to MCF7 cells and none of the other profiled cell lines. We take a

```
> topTable(mcf7.fit,coef=2)
```

	ID	logFC	AveExpr	t	P.Value	adj.P.Val	B
534	PIK3CA	-3.5475	-0.962	-5.493505	0.0005000999	0.3895778	-3.257460
119	CDK4	-2.5400	0.272	-4.136689	0.0029854111	0.5552405	-3.513727
708	STK38	-2.5325	-0.554	-3.829396	0.0046437445	0.5552405	-3.591301
110	CDC2L2	2.2275	-3.702	3.787632	0.0049361878	0.5552405	-3.602504
587	PRKCL2	2.0600	-1.038	3.713153	0.0055074594	0.5552405	-3.622889
669	SLK	-1.9875	0.010	-3.415842	0.0085899584	0.5552405	-3.709579
570	PRKACA	-1.9900	-0.078	-3.340133	0.0096365260	0.5552405	-3.733045
640	RIPK3	2.1450	-0.066	3.322594	0.0098975883	0.5552405	-3.738562
656	RPS6KC1	-1.8100	0.388	-3.241761	0.0112003483	0.5552405	-3.764390
748	TRIB3	1.8200	-0.346	3.142238	0.0130550855	0.5552405	-3.797090

FIGURE 5.13

Annotated table of siRNAs with MCF7-specific effects on cell viability.

similar approach to the one for the two-group test, but rather than using a *contrast matrix* we specify the contrast in the design matrix by using an intercept and a second variable to indicate whether the column represents the MCF7 cell line or not. The output variable from the design matrix needn't necessarily be an indicator variable (taking only 1 or 0 as a value), but in this case it is⁸.

```
> library(limma)
> design <- cbind(Intercept=1,MCF7=c(1,0,0,0,0))
> mcf7.fit <- lmFit(siRNA.zscores,design=design)
> mcf7.fit <- eBayes(mcf7.fit)
> topTable(mcf7.fit,coef=2)
```

The results of the final `topTable` command should be as in [Fig. 5.13](#), showing that PIK3CA is quite clearly MCF7-specific in its effect on cell viability. We know this already from the line graph shown in [Fig. 5.6](#), with most of the z-scores for PIK3CA around zero apart from cell line 1 (MCF7) which had markedly reduced viability upon PIK3CA siRNA transfection. This observation is also highlighted in the Iorns *et al* 2009 paper.

5.6 Summary

The purpose of this tutorial is to show how R can be used to apply a set of simple statistical analysis tools to a wide range of data, provided it is in a tabular form. An important step is the formatting of the data table and

⁸In fact, this is another two-class problem, with one class being MCF7 and the other being everything else.

loading into the R workspace, for which guidelines have been given. While the example dataset used throughout this tutorial came from an siRNA screen, it could as well have been a normalised table of data from microarrays, a proteomics array, or many others. DNA methylation data represents a special case, for which all these analysis are applicable following a transformation of the data, covered in the appendix.

As a final caveat to all of the statistical analysis demonstrated in this tutorial, this study involves no replicates and the sample size is so small that the actual value of the statistics have little meaning in relation to traditional statistical significance thresholds: the study is **so** under-powered to detect any reasonable statistical associations. However, I hope that these examples demonstrate the fact that even in the absence of application of significance thresholding criteria, this sort of statistical analysis is still a very useful tool to identify important features from large numerical datasets.

6

Functional Enrichment Analysis

6.1 Introduction

When analyzing large datasets, we sometimes find there are so many features showing some characteristics of interest that it can be difficult to get a sense of what these mean at a functional level. To assist in higher-level interpretation of systematic differences reflected in a dataset, we can make use of defined sets of genes that each represent some known biological characteristics. By testing the *enrichment* of gene sets, which means evaluating how much more those specific genes are reflected in the results than would be expected just by chance.

In this tutorial we will look at two different approaches to testing enrichments: *over-representation* takes two lists and counts the overlap between them; *systematic enrichment* uses a scoring (or ranking) of all the features in a dataset, and evaluates how skewed the scoring (or ranking) is for a specific subset of those features (e.g. a defined gene set). There are a number of widely used online tools for performing both of these types of enrichment analysis, which have conveniently pre-loaded gene sets and all you need to do is upload a list of genes, with or without scores. It is also useful to be able to perform the relevant statistical tests yourself, so that you can evaluate enrichments involving gene sets not already defined in any online tools, and so that you can evaluate enrichments in results that aren't directly mappable to annotations using existing tools. This tutorial will also enable you to compare the results of analyses of different datasets to see if they agree more than would be expected by chance.

6.2 Loading gene sets into R

The first part of this tutorial involves accessing defined lists of genes in R, which can be used for functional enrichment analysis. There are many

databases containing such gene sets, including: MSigDB (<http://software.broadinstitute.org/gsea/msigdb/>), ConsensusPathDB (<http://www.consensuspathdb.org/>), KEGG (<http://www.genome.jp/kegg>), Gene Ontology (<http://geneontology.org/>).

Because it is freely and easily accessible without registration, we will download the table of biological pathway annotations from ConsensusPathDB. Open a browser and go to the website at <http://www.consensuspathdb.org/>, then click on the ‘download / data access’ link on the left-hand panel, as indicated in Fig. 6.1.

In the main frame of the page, select ‘gene symbol (HGNC symbol)’ from the drop-down menu appearing in the line of text saying ‘Biological pathways (as defined by source databases) with their genes identified with...’, then click on the text. This will download a text file called ‘CPDB_pathways_genes.tab’. Move this file into a directory you can access and then start R.

We can read the text file into R using the `read.table` function:

```
> cpdb.pathways <- read.table("CPDB_pathways_genes.tab",
+ sep="\t", head=T)
```

This command creates a *data frame* called `cpdb.pathways` from the text file ‘CPDB_pathways_genes.tab’, using a tab to separate the columns, and treating the first row of the file as the table’s header (i.e. the data frame’s column names). Take a look at the first row of the data frame:

```
> cpdb.pathways[1,]
```

We can see that the fourth column ‘hgnc_symbol_ids’ contains the list of gene symbols for each pathway, separated by commas. As the gene sets may be of different lengths, we ideally want to have a list where each element is a vector of the gene symbols belonging to the corresponding pathway. We can apply an R function called `strsplit` to the comma-separated gene symbols from the ‘cpdb.pathways’ data frame, which splits each single character object into a vector of separate characters. So we can a list of gene symbol vectors with the following command:

```
> cpdb.genes <- strsplit(as.character(cpdb.table$hgnc_symbol_ids),
+ split=",")
```

In this command, we have created an object called `cpdb.genes` from the output of the `strsplit` function. I have added an extra function call to apply `as.character` to the comma-separated gene symbol sets we access from the data frame, because the data frame structure may not specifically know that

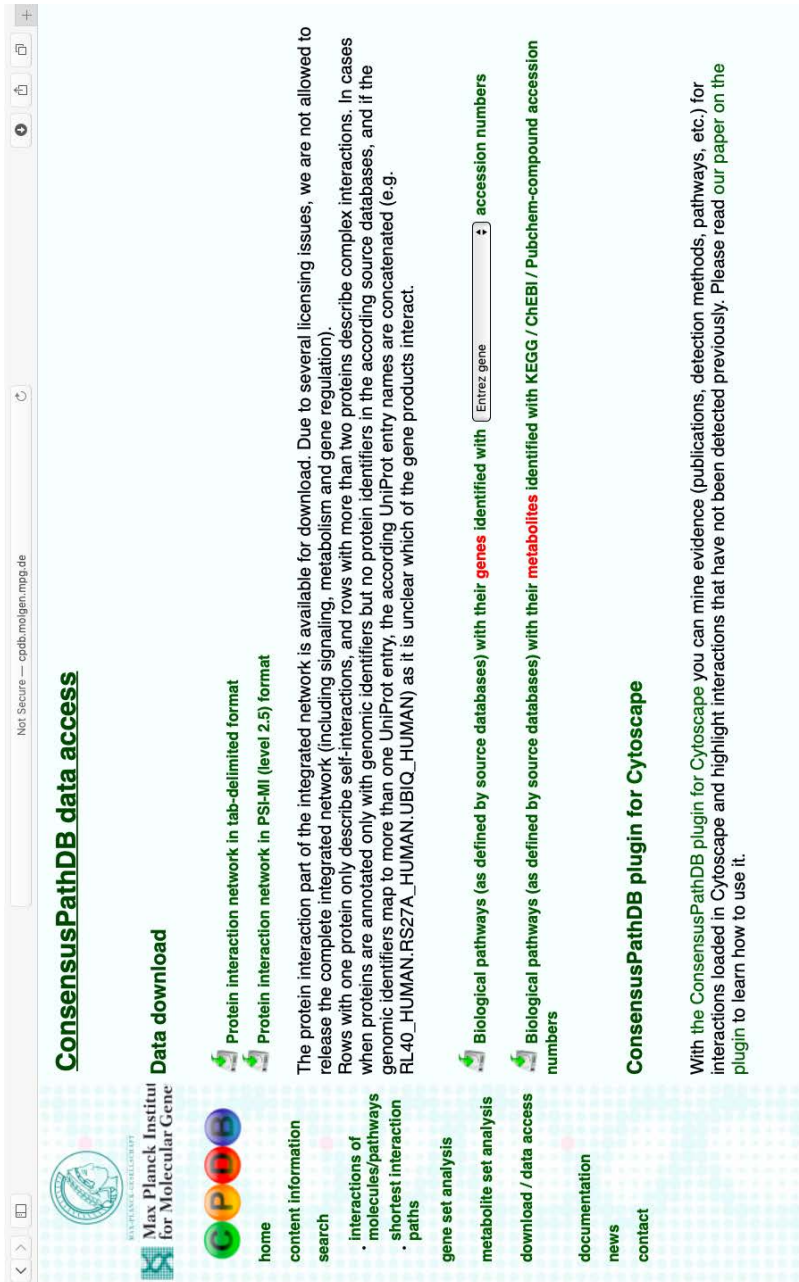


FIGURE 6.1 ConsensusPathDB website showing link to download.

each element object can be treated as a character. The other argument we have passed to the `strsplit` function specifies that the symbol to use to split up each character object (into a vector of smaller character objects) is a comma.

We now have a data frame which includes the names of each pathway, and a list of character vectors containing the gene symbols that have been annotated as belonging to each pathway. From this point, we can use these objects to test for enrichments of each pathway in some other selected set of gene symbols.

6.3 Over-representation

In over-representation analysis, we are testing whether the overlap between two lists is greater than what we would expect by chance (if the two lists were sampled from the same distribution *completely independently*). For this we need to know not only the contents of the two lists, but also the full set of potential items that the lists *could* have included. For example, this could be the set of all genes in the genome according to the annotations we used for our analysis. Or it could be the set of all genes represented on a given measurement platform that was used to produce our dataset. This is important to remember, because it is quite possible for the set of measured entities to be already biased towards some biological processes, if those were of particular interest to the designers of the platform. In such circumstances, without taking that bias into consideration, you would expect randomly selected lists of features to show similar enrichments. The over-representation problem is relatively easy to define in terms of probabilities, and is often described as like drawing coloured balls from an urn (a bag may be more relatable, but for some reason the probability theorists love their urns). If we randomly choose a certain number of balls out of a container that has in it a certain number of white balls and a certain number of black balls, then we want to know what is the probability of seeing at least as many black balls as we have. Relating that back to the case of gene sets, if we have a set of genes selected on the basis of some analysis of a dataset and we look at the overlap with an annotated gene set: the container of balls is the set of all genes we could have selected from our analysis (i.e. all the mapped features); the black balls are the genes in the annotated gene set we are testing for enrichment; the balls that are drawn from the container are the genes in our list of features of interest obtained from analysis of our dataset. This situation is described by the hypergeometric probability distribution, which gives us computable probabilities for any settings of the variables in the problem.

6.3.1 Online tools

DAVID (<https://david.ncifcrf.gov/>) and ConsensusPathDB (<http://www.consensuspathdb.org/>) both have easy-to-use interfaces enabling upload of a list of identifiers and selection (or upload) of a reference set of all possible identifiers you could have selected. They will then perform the over-representation analysis for you, testing all their defined gene sets one after the other, and presenting the results of tests which showed significant enrichment.

6.3.2 Testing gene sets in R

Being able to test for over-representation yourself will enable you to test associations that don't involve standard gene sets, such as comparing the results of separate analyses performed across two different datasets. In order to test the significance of an observed overlap between two sets, we need four values that we will describe using mathematical notations for *sets*:

- $|A \cap B|$: the number of elements that appear in both sets
- $|A \cap C|$: the number of elements in the first set that *could have been* in the second set
- $|B \cap C|$: the number of elements in the second set that *could have been* in the first set
- $|C|$: the number of elements that could have been in either set

These descriptions use the notation $|...|$ to describe the size of a set, and refer to three specific sets that are relevant for over-representation analysis: A is the first gene set, B is the second gene set, C is the set of identifiers from which A and B were both selected. It is not always entirely obvious how to represent C . If A and B are sets of identifiers selected from analyses of two different datasets, then C is the set of all identifiers that have values in *both* datasets. If B is a set of gene symbols representing a given pathway, then how do we know which gene symbols *could* be in *any* pathway? We can make an assumption that any defined gene symbol could appear, or perhaps we instead use for C the full list of gene symbols that appear in *any* pathway gene sets from the database we have obtained set B . Because there is not always one way to do this, make sure you are clear of your process when describing your analyses.

As a fairly trivial example, we will go back to the list of gene symbol vectors that we have obtained from ConsensusPathDB earlier in this tutorial. We can look at the name of the first pathway in the table `cpdb.table` by telling R to show us the first row from the 'pathway' column:

```
> cpdb.table[1,"pathway"]
```

We should see the following output:

```
[1] Alanine, aspartate and glutamate metabolism - Homo sapiens
(human)
```

So we know that the first set of genes reflects the alanine, aspartate and glutamate metabolism pathway. A look back at the entire first row of the data frame shows us this is defined in the database KEGG. Let's now look at the 32nd row in the table:

```
> cpdb.table[32,]
```

This gene set reflects the arginine and proline metabolism pathway, as defined in the KEGG database. So we can use the 1st vector of gene symbols from our list `cpdb.genes` to be our set A , and the 32nd vector of gene symbols from `cpdb.genes` to be our set B . After finding the number of gene symbols that overlap between these two vectors, and the list of all possible gene symbols, we can then compute the probability that there would be so large an overlap purely by chance.

First, compute $|A \cap B|$, the number of overlapping elements across the two lists. We can use the function `intersect(A,B)` to find elements that are in both A and B :

```
> intersect(cpdb.genes[[1]], cpdb.genes[[32]])
```

We can see that three gene symbols overlap, but to avoid having to count this by hand in case it's much larger, we can use the `length` function:

```
> length(intersect(cpdb.genes[[1]], cpdb.genes[[32]]))
```

We see the output:

```
[1] 3
```

It is trivial to find the lengths of A and B :

```
> length(cpdb.genes[[1]])
[1] 35
```

```
> length(cpdb.genes[[32]])
[1] 49
```

So $|A \cap B| = 3$, $|A| = 35$ and $|B| = 49$. What about set C ? Let's assume that the full set of gene symbols that could have been in any pathway is the set of gene symbols which are annotated in at least one pathway. We can find

this using the `unique` function (which removes duplicates from a vector), and the `unlist` function, which will collapse a list of multiple vectors into a single (long) vector:

```
> length(unique(unlist(cpdb.genes)))
[1] 11196
```

So $|C| = 11196$. We can now use these values to give as arguments to the `phyper` function, which computes the cumulative density function from the hypergeometric distribution. That is, $P(X \leq x)$: the probability of observing an overlap less than or equal to some specified value x in some random variable X . Because the p-value we want to indicate the statistical significance of the overlap is the probability of a randomly-chosen set of identifiers from C having greater than or equal overlap to A as the observed overlap between A and B , we actually want to find $P(X \geq x) = 1 - P(X \leq x - 1)$. One other subtlety of using this function is that the arguments it needs are $|A \cap B|$, $|A|$, $|B|$ and $|C \setminus A|$ (this means the elements in C **not** in set A), but we can use these as follows:

```
> 1-phyper(3-1, 35, 11196-35, 49)
```

And we see the output:

```
[1] 0.0004672633
```

So the probability of observing so large an overlap purely by chance is less than 1 in 1000. Now, by itself that can be fairly useful, but really comes into its own when we combine it with some simple scripting. Let's use this to find which other pathways have the most significant overlaps with the alanine, aspartate and glutamate metabolism pathway. First, for neatness, let's create a function that will calculate this p-value for us, given 3 gene sets A , B and C :

```
getORpval <- function(A,B,C){
+ overlap = length(intersect(A,B))
+ sizeA = length(intersect(A,C))
+ sizeC = length(setdiff(C,A))
+ sizeB = length(intersect(B,C))
+ 1-phyper(overlap-1,sizeA,sizeC,sizeB)}
```

It should hopefully be clear how this new function, which we have called `getORpval`, just goes through each of the calculations we made for the previous example. We can check this, by using the new function to repeat the example:

```
> getORpval(cpdb.genes[[1]],cpdb.genes[[32]],unlist(cpdb.genes))
[1] 0.0004672633
```


So we see this gives us the same answer as before. **NB:** it is *always* a good idea to create examples like this to check any code that you write, particularly when setting up a script to perform many analyses in one go.

To compute the p-values for *all* the gene sets, we could create a vector of all missing values, then fill in the values one-by-one using a *for* loop:

```
> allpath.pvals <- rep(NA,nrow(cpdb.table))
```

This creates a vector called `allpath.pvals` by repeating the special NA value once for each row in the `cpdb.table` data frame.

```
> for(i in 1:length(allpath.pvals)){
+ allpath.pvals[i] = getORpval(cpdb.genes[[1]],cpdb.genes[[i]],
+ unlist(cpdb.genes))}
```

Here we have created a loop over an iterator variable `i`, which takes values from 1 up to the length of the `allpath.pvals` vector, which is the same length as the number of rows in the `cpdb.table` data frame. Within the loop, it applies the function we have defined called `getORpval` with set *A* being the vector of gene symbols from the first pathway, set *B* being the set of gene symbols from the *i*th pathway (i.e. each pathway in turn), and set *C* being the set of gene symbols in any pathway, as before. This may take a little time to run, but the end result will be a vector of p-values, one for each pathway.

We can now use R to help present these p-values in a more useful format, as a sorted table listing the pathway names, the corresponding p-values, and because we have applied many identical hypothesis tests, a multiple-testing-adjusted p-value. Let's create a data frame to present these outputs:

```
> OR.df <- data.frame(pathway=cpdb.table$pathway,p.value=allpath.pvals,
+ adj.P.val=p.adjust(allpath.pvals))
```

Here we have used the `p.adjust` function to perform multiple testing correction across our vector of p-values, using its default settings (which use the *FDR* method). We can use the `order` function to re-order the rows of the table from smallest p-value to largest:

```
> OR.df <- OR.df[order(OR.df$p.value,decreasing=F),]
```

Now inspect the first rows of the table:

```
> OR.df[1:5,]
```

6.4 Systematic enrichment

To avoid limiting our analyses to approaches that require setting arbitrary thresholds for selecting output gene lists, we can alternatively evaluate the unlikelihood of an observed trend or systematic shift towards extreme values of any given test statistic. Because these approaches use as their input a list of scores for all measured features in the dataset, and compare the scores for the gene set of interest against all the other features, any bias towards which features are represented in the dataset is automatically taken into account. There can be many ways of defining a systematic shift towards extreme values of a test statistic, and therefore many (each valid) ways of testing a hypothesis of enrichment of a given set of identifiers. We will give examples of some approaches.

6.4.1 Online tools

The best known tool for enrichment analysis is GSEA¹, although Consensus-PathDB² also has an enrichment analysis interface as part of its ‘gene set analysis’ tools. As with over-representation analysis, these resources have pre-loaded gene sets that make functional annotation through enrichment analysis relatively simple to perform.

6.4.2 Testing gene sets in R

There are tools in R packages that make it relatively straightforward to test the statistical significance of systematic trends in sets of results. We will make use of some here, which will hopefully provide you with some valuable tools for your own research. In order to test a systematic shift in the values of a test statistic for some subset of the features in a dataset, we need three things:

- a vector of identifiers for *all* features represented in the dataset
- a vector of values of the statistic of interest, one value for each feature of the dataset
- a set of identifiers denoting the features to evaluate for enrichment over randomness

For the vector of test statistic values, we need a way of computing a number to reflect some characteristic of interest for each feature. Examples could be a t-statistic from a linear model comparing the values of the feature across two

¹<http://software.broadinstitute.org/gsea/index.jsp>

²<http://www.consensuspathdb.org/>

sets of samples, a fold-change between the averages across two sets of samples, the frequency of mutation of a gene across a population, or potentially anything that is of interest. For a simple example to run through here, we will re-load the siRNA screen data that was analyzed in [Chapter 5](#) ‘Analyzing generic tabular numeric data in R.’

Open an R workspace and read in the table. Make sure the working directory is set to the folder where you saved the siRNA screen data table, then enter the following command:

```
> siRNA.screen <- read.table("siRNAscreen_data.txt",sep="\t",
header=TRUE)
```

Convert the resulting data frame to a two-dimensional *numeric* array using the `as.matrix` function. Finally, we can annotate the rows of this array with the corresponding GeneSymbol.

```
> siRNA.zscores <- as.matrix(siRNA.screen[,-c(1,2)])
> rownames(siRNA.zscores) <- as.character(siRNA.screen$GENE)
```

Now let’s use the *limma* package to compute t-statistics that characterize the statistical significance of the difference in viability score for the MCF7 cell line compared to all others:

```
> library(limma)
> design <- cbind(Intercept=1,MCF7=c(1,0,0,0,0))
> mcf7.fit <- lmFit(siRNA.zscores,design=design)
> mcf7.fit <- eBayes(mcf7.fit)
```

Now we’ll save all the statistics into a data frame so we can use them later:

```
> allMCF7stats <- topTable(mcf7.fit,coef=2,
+ number=nrow(siRNA.zscores))
```

To evaluate enrichments of defined gene sets in the results of this analysis, we need to define some gene sets of interest. Let’s use the ConsensusPathDB pathway annotations. Repeat the steps in Section 6.2 of this chapter to create the data frame `cpdb.table` and the list of gene sets `cpdb.genes`.

The *limma* package also includes a function that can be used for evaluating the statistical significance of enrichments according to some defined test statistic. It does this using a rank sum method: after ranking all the dataset’s features based on their values of the provided test statistic, it compares the sum of those ranks for the selected features against the expected distribution

of sums of ranks for randomly-chosen sets of (the same number of) features. The function to perform this is called `geneSetTest`. In the following example, we will use this function to test for enrichment of the ‘alanine, aspartate and glutamate metabolism’ pathway genes in terms of their MCF7-specific impact on viability, among the 779 genes targeted in the siRNA screen:

```
> geneSetTest(statistics=allMCF7stats$t,
+ index=which(allMCF7stats$ID %in% cpdb.genes[[1]]),
+ alternative="down")
```

In this command, we have passed three arguments to the `geneSetTest` function: ‘statistics’ is the vector of values of the test statistics, which in this case are the t-statistics from the MCF7-specific linear model we fit using *limma*; ‘index’ is a vector stating which of the values represent the subset we are testing for enrichment, which in this case are the ones for which the gene symbol is in the CPDB gene vector corresponding to the pathway; ‘alternative’ states which alternative hypothesis to test against, where the null hypothesis is no difference from the sum of ranks of randomly selected genes. The above command evaluates the probability that a randomly-selected subset of genes from the siRNA screen would have as least as high a sum of ranks (reflecting low values of the test statistic) as that observed for the pathway in question: in other words, how unlikely is it to see as great a systematic shift towards smaller values in MCF7 than the other cell lines (i.e. MCF7-specific cell killing), across the entire set of genes reflecting the pathway. Having run the command above, you should see the resulting p-value:

```
[1] 1
```

So the corresponding term is not enriched at all in these statistics. This procedure forms the basis of testing enrichment analyses, but again we may wish to screen a whole database of gene sets to see which ones are most enriched. We can do this in a similar way to the over-representation analysis. However, given we only have 779 genes represented in the dataset, it is quite likely that some pathways may have no genes at all represented in the dataset. Because of this, we may wish to only perform the tests for pathways that map to at least a few (say, 3 or 4) genes in the siRNA screen. We can easily implement this by making use of a *conditional* statement using the `if` function. To compute the p-values for *all* the gene sets, we could create a vector of all missing values, then fill in the values one-by-one using a `for` loop:

```
> allpath.pvals2 <- rep(NA,nrow(cpdb.table))
```

This creates a vector by repeating the special `NA` value once for each row in the `cpdb.table` data frame.

Now set up a loop to go through each pathway in turn, testing them for systematically low relative viability values in MCF7:

```
> for(i in 1:length(cpdb.genes)){
+ if(length(intersect((allMCF7stats$ID,cpdb.genes[[i]])))>3){
+ allpath.pvals2[i] <- geneSetTest(statistics=allMCF7stats$t,
+ index=which(allMCF7stats$ID %in% cpdb.genes[[i]]),
+ alternative="down")
+ }
```

Note that in this loop we have two curly brackets opened and closed: one marking which code to execute within the `for` function and one marking which code to execute if the conditions for the `if` function are met.

We can now use R to help present these p-values in a more useful format, as a sorted table listing the pathway names, the corresponding p-values, and because we have applied many identical hypothesis tests, a multiple-testing-adjusted p-value. Let's create a data frame to present these outputs:

```
> enrichment.df <- data.frame(pathway=cpdb.table$pathway,
p.value=allpath.pvals2,
+ adj.P.val=p.adjust(allpath.pvals2[!is.na(allpath.pvals2)]))
```

Here we have used the `p.adjust` function to perform multiple testing correction across our vector of *non-missing* p-values, using its default settings (which use the *FDR* method). Note, the statement within the square brackets is indexing elements from the `allpath.pvals2` vector which are not missing (! denotes negation of a logical test). As before, we can use the `order` function to re-order the rows of the table from smallest p-value to largest:

```
> enrichment.df <- enrichment.df[order(enrichment.df$p.value,
decreasing=F),]
```

6.5 Summary

In this chapter we have explored ways of testing for greater-than-expected overlap or systematic shift in values for some set(s) of features of interest. These examples should hopefully have illustrated how a database of gene sets can be used to get a sense of the bigger picture represented by the trends discovered in some analysis of a high-dimensional dataset.

Integrating Multiple Datasets in R

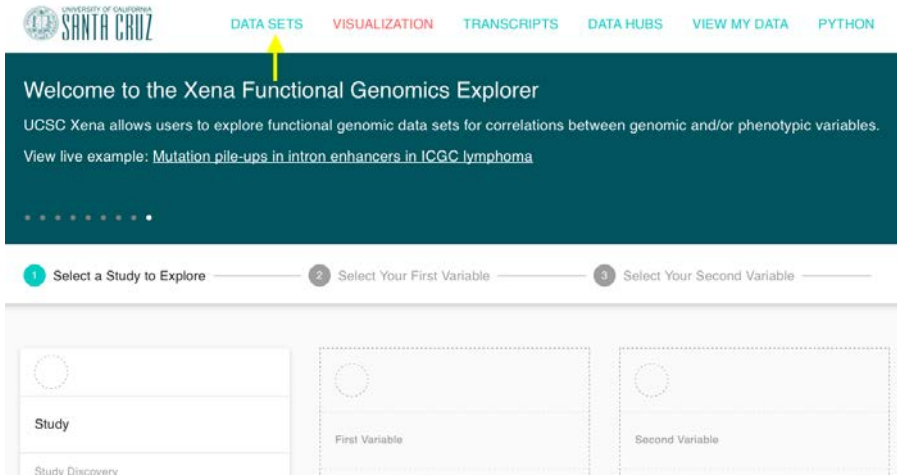
7.1 Introduction

It really is a useful skill to be able to manipulate datasets so they can be loaded into R, then explore structures and patterns in the data, relating different entities to each other. Having said that, often the real power in analyzing a dataset is to look for associations with characteristics of a different dataset. For example, it is interesting to see the samples in a molecular profiling dataset segregate into clearly distinct clusters, but it adds a lot to our interpretation of these clusters if the samples in different clusters tend to have different phenotypic traits. As another common example, we may wish to find proteins or genes whose expression levels in tumours are associated with the patients' clinical outcomes: this typically necessitates integration of data from multiple sources.

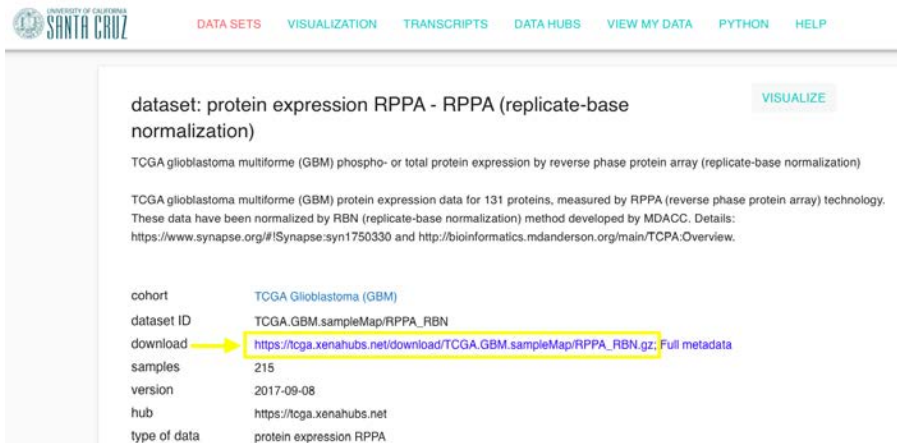
This tutorial is based around an example analyzing data from The Cancer Genome Atlas (TCGA). It is designed to illustrate a practical scenario which is likely to arise frequently in cancer research, although the principles will apply to virtually any analysis of molecular biology datasets, especially those with a clinical context. The TCGA data is all available from UCSC's Xena browser, which can be found at: <https://xenabrowser.net/heatmap/>

The default viewer will have a dataset pre-loaded, but it is not necessarily the one we want, so click on the 'Add Datasets' button on the left-hand side, as illustrated in [Fig. 7.1](#).

Clicking on this button will open a new window, with all the cancer datasets processed for the browser listed. Scroll down to the dataset 'TCGA Glioblastoma (GBM)', and click on this link. This will take you to another page listing all the available datasets for this cohort. Click on the links for 'Phenotypes' and 'RPPA (replicate-base normalization)'. For each dataset, you should see an information page which includes a download link, highlighted in [Fig. 7.2](#).

**FIGURE 7.1**

Screenshot showing Xena Browser, with ‘Data Sets’ link indicated by arrow.

**FIGURE 7.2**

Screenshot showing Xena Browser, with download link for TCGA Glioblastoma RPPA dataset indicated by arrow and box.

If you download the files for both of these datasets, you will also need to unzip them with a tool that can unzip *gzip*-compressed files. With the files downloaded and extracted, we should be in a position to load the data into R as tables.

7.2 Data import

The first file we wish to load in is called ‘RPPA_RBN’ as this contains the feature-level data from all the tumour samples in the dataset. In this instance, the features represent proteins (or antibodies to proteins). We can read in the file as follows:

```
> rppa.data <- read.table("RPPA_RBN", sep="\t", head=TRUE, row.names=1)
```

This command creates a *data frame* object called `rppa.data`¹, using the contents of the file ‘RPPA_RBN’. It specifies that the separator between columns should be a tab (`sep="\t"`), that the first row gives column headers (`head=TRUE`), and that the first column gives the row names (`row.names=1`). We can check the dimensions of the data frame using the function `dim`:

```
> dim(rppa.data)
[1] 131 215
```

This tells us that the data frame `rppa.data` has 131 rows and 215 columns.

This is all well, but we probably want to perform some numerical operations on the data. Many of the functions that R provides to do this will not work on data frames, because they need to know that the data contained in the table are all numerical values. To enable this, we simply convert the data frame to a matrix:

```
> rppa.data <- as.matrix(rppa.data)
```

Note that if the `rppa.data` object had contained any non-numerical values for anything other than column or row names, then this conversion would not work properly.

7.3 Exploratory data analysis

With the glioblastoma protein data loaded into R, we may wish to explore some of the properties of the dataset. For example, we could look to see if there is any clear grouping of the samples or proteins into clusters. Creating a cluster

¹I have called it this because the protein expression data in TCGA comes from a Reverse Phase Proteomics Array (RPPA) platform, from MD Anderson.

dendrogram for the either samples or proteins is relatively straightforward. For the proteins, this would be:

```
> plot(hclust(dist(rppa.data)))
```

This uses the `dist` function to create a matrix of the pairwise Euclidean distances (i.e. the sum of the squared differences between each sample's values for the two proteins in question), then the `hclust` function to produce a hierarchical clustering structure, and finally the `plot` function to draw this. The result should look like the dendrogram shown in [Fig. 7.3](#).

As the Euclidean distances are based on the differences in absolute values for each pair of proteins, if the data hasn't been centred so that the average values for each protein are the same, differences in antibody affinity might bias the results. If the distances are based on the pairwise correlations, they wouldn't be affected by this. So we can create the same structure using a different distance metric:

```
> plot(hclust(as.dist(1-cor(t(rppa.data)))))
```

Here we convert the correlation matrix produced by `cor` into a *distance matrix* object using the `as.dist` function. You should notice that the `cor` function was applied to `t(rppa.data)` rather than just `rppa.data`, this is because it operates column-wise where the `dist` function operates row-wise. The result should be as appears in [Fig. 7.4](#).

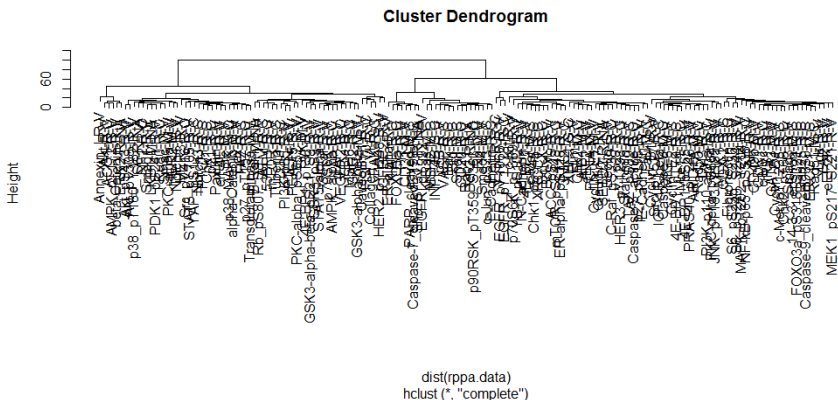
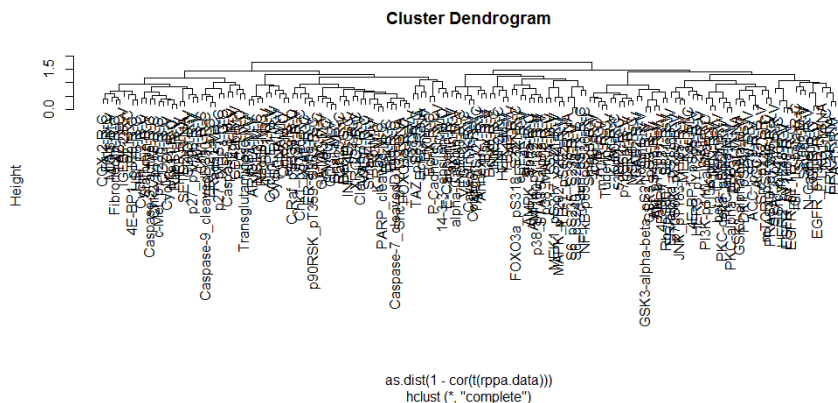


FIGURE 7.3

Hierarchical clustering dendrogram showing similarity structure between proteins in TCGA glioblastoma RPPA dataset, using Euclidean distance.

**FIGURE 7.4**

Hierarchical clustering dendrogram showing similarity structure between proteins in TCGA glioblastoma RPPA dataset, using correlation-based distances.

You can see that the results are different, but these dendrograms are perhaps not the most useful because they contain rather too many labels to be interpreted readily! We could create dendrograms showing structure of the samples simply by taking away the ' $t()$ ' from the correlation distance matrix computation, and by adding it into the Euclidean distance computation (so that the command becomes `dist(t(rppa.data))`).

Let's say we have decided that we would like to look for clusters of tumours that have similar protein expression profiles as measured on this RPPA platform, and that the sample-wise hierarchical clustering dendrogram reveals three major groups. To assign samples to groups, we can use the function `cutree()` to 'cut' the dendrogram at a given height, so that all the samples in the same branch at that height are assigned to the same cluster. We can run this example in two parts:

```
> rppa.hclust <- hclust(dist(t(rppa.data)))
> rppa.clusters <- cutree(rppa.hclust,k=3)
```

Here we have first created an object which we have named `rppa.hclust`, which contains the result of the hierarchical clustering applied to the sample-wise Euclidean distance matrix. The second command then creates an object called `rppa.clusters`, which is a vector of the cluster IDs for each sample (column) of the `rppa.data` matrix.

A benefit of generating cluster assignments in R is that we can use these in subsequent analyses. For example, we may wish to look at the overall distribution of protein expression values using a heatmap, and can then label

the heatmap with the cluster assignments. To create the heatmap, there is a helpful package called *gplots* which provides functions to generate a colour gradient. We load the package using:

```
> library(gplots)
```

If the *gplots* package isn't installed on your system, try installing it using:

```
> install.packages("gplots")
```

You should be able to load the package once it is successfully installed². Then to draw the heatmap of the RPPA dataset, we can use the following command:

```
> heatmap(rppa.data,col=bluered(100),scale="none")
```

Here the `heatmap` function generates the plot, with colours in a 100-point scale from blue (lowest) to red (highest). The colours are generated by `bluered(100)`. You could use `greenred(100)` for the green-red colour scheme, but the blue-red scheme is generally considered better as fewer people are affected with colourblindness in this spectrum. We have specified `scale="none"` to make the colours in the heatmap directly reflect the values from the data matrix. In many cases this makes some rows or columns appear entirely high or low, and so sometimes scaling by row (`scale="row"`) or by column (`scale="col"`) helps.

To add the cluster assignments to the heatmap, we need to create a vector of values that says which colour to make each sample. We could use the numerical cluster assignments as indices to select from a vector of colour names that we specify:

```
rppa.cols <- c("yellow","red","green")[rppa.clusters]
```

Here we have used the `c()` function to create a vector with three colour names, which are then selected from using the cluster assignments. We can add this to the heatmap using the `ColSideColors` argument:

```
> heatmap(rppa.data,col=bluered(100),scale="none",ColSideColors=
rppa.cols)
```

²Note that if you do not have access privileges to install packages, for whatever reason, you can tell R to use your current (writeable) directory instead by entering the command `.libPaths(getwd())`. The `'.'` here before `libPaths` is essential!

We might want to add a figure legend to this plot:

```
> legend("topleft",legend=c("C1","C2","C3"),fill=c("yellow","red",
+ "green"))
```

This command puts a figure legend in the top-left corner of the plot window, specifying the legend text to contain C1, C2 and C3, represented by boxes filled with the colours yellow, red and green, respectively. Using the cluster assignments the end result should resemble [Fig. 7.5](#).

If we are particularly interested in the clustering structure that emerged from our initial exploratory analysis of the data, we might wish to characterize these clusters further. One way to do that would be to find the proteins which best discriminate between the samples of different clusters. We could achieve

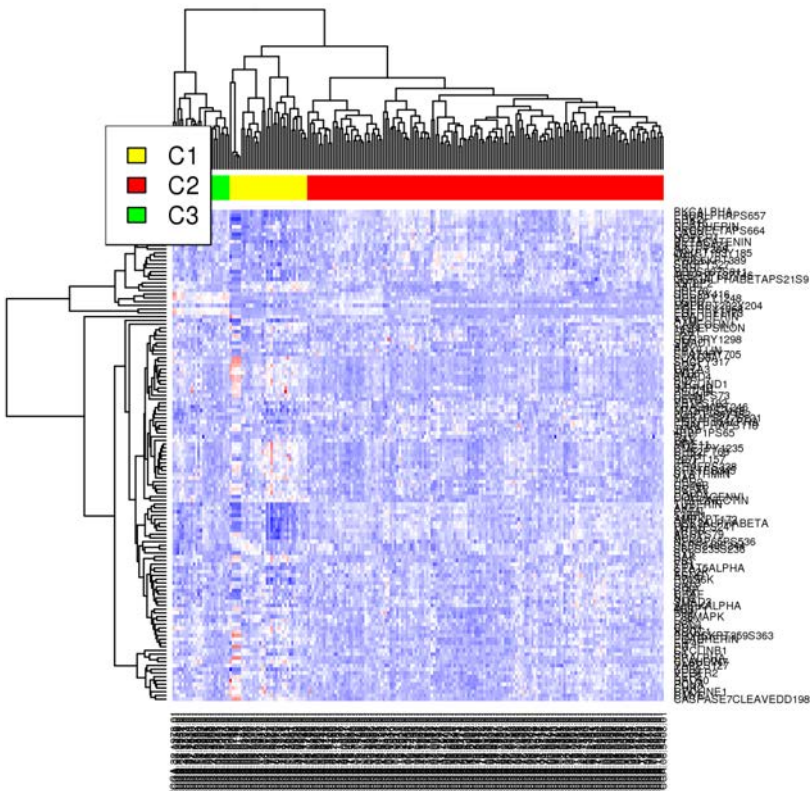


FIGURE 7.5

Heatmap showing RPPA-derived protein expression levels for TCGA glioblastoma tumour samples, with cluster assignments indicated by colour bar.

this by using the linear model framework available in the *limma* package. We first need to load the *limma* package into R:

```
> library(limma)
```

If *limma* is not already installed, it can be obtained directly through R (from Bioconductor):

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("limma")
```

Once it is installed, it should now be possible to load the package.

With *limma* loaded, the first step in fitting linear models to all rows of a large dataset involves creating a design matrix. This is a representation of all the explanatory variables we wish to include in our linear model. This approach was covered in Section 4.6, and used for a molecular biology application in Section 5.5. In this example, if we want to construct a linear regression model relating protein expression levels to the hierarchical clustering assignments in the TCGA glioblastoma RPPA dataset, we should use three variables in the design matrix: one for the intercept (this will allow for differences in the baseline/background protein expression levels), one to represent expression specific to cluster 2, and one to represent expression specific to cluster 3. Note that we don't need to consider expression specific to cluster 1, because if a sample isn't in cluster 2 or cluster 3 then it has to be in cluster 1 (therefore a 4th variable would be redundant). You might wonder why we don't just have one variable for the intercept and one variable for the cluster assignment: because we are using linear regression, this would attempt to find protein expression levels that increase or decrease sequentially from cluster 1 to cluster 2 and then to cluster 3, as the assignments would be treated as numeric values. Cluster membership is a binary (in-out) variable, and so we need one variable for each cluster (beyond the default case where all samples belong to a single cluster). So, with that explained, we construct our design matrix as follows:

```
> design <- cbind(intercept=1,c2=as.numeric(rppa.clusters
==2),c3=as.numeric(rppa.clusters==3))
```

This rather long command is putting 3 columns together to form a table (using `cbind`). The first column (which we have labelled 'intercept') has all values equal to 1. The second and third columns are converted from TRUE/FALSE values to 0/1, with the second column equal to 1 when the corresponding sample is assigned cluster 2 and equal to 0 otherwise (and similarly for the third column, but for cluster 3). If you run `dim(design)` you should see that this table has 214 rows (one for each column of the protein expression data matrix) and 3 columns.

Fitting linear models to each protein's expression levels is trivial once the design matrix has been created appropriately. We first use the `lmFit` function to create a model fit object:

```
> rppa.cluster.fit <- lmFit(rppa.data,design=design)
```

Then we use the `eBayes` function to generate *moderated* t-statistics for each model fit (this borrows information across the whole set of model fits in order to estimate where significance of a fit has been overly optimistic, or not optimistic enough):

```
> rppa.cluster.fit <- eBayes(rppa.cluster.fit)
```

We can now inspect the most significant proteins to discriminate between the clusters by using the `topTable` function:

```
> topTable(rppa.cluster.fit,coef=c(2,3))
```

The additional argument `coef=c(2,3)` specifies that the linear model coefficients we are interested in correspond to the second and third columns of the design matrix (we are not interested in the statistics associated with the intercept). Helpfully, the statistics corresponding to the terms of the model fits for the cluster assignments are combined, and a p-value for the joint effect is displayed. To store the names of the proteins showing this differential expression, we can extract the row names from this table, but we may wish to specify a number of proteins to include and an adjusted p-value threshold (in case the specified number includes features that weren't significantly associated with the specified variables):

```
> rppa.cluster.proteins <- rownames(topTable(rppa.cluster.fit,coef=c(2,3),n=40,p.val=0.05))
```

With a set of proteins most significantly discriminating between the clusters, the first step in interpreting the clusters would probably be to look at what those proteins are and which clusters have high/low expression. We may wish to visualize the expression values though, and so we can re-create our heatmap, this time using only the most differentially-expressed proteins:

```
> heatmap(rppa.data[rppa.cluster.proteins,],col=bluered(100),scale="row",ColSideColors=rppa.cols)
```

You should see that all we needed to do to select only a subset of the rows from the data table was to use square brackets `[rppa.cluster.proteins,]` to specify our list of row names to select (indexing arrays is described in [Chapter 2](#) of these tutorials). One other change to the previous heatmap command

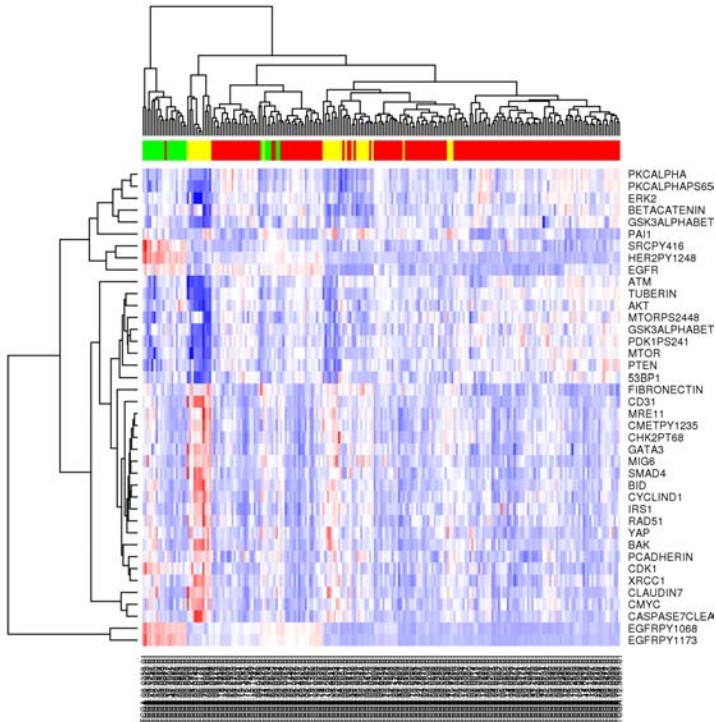


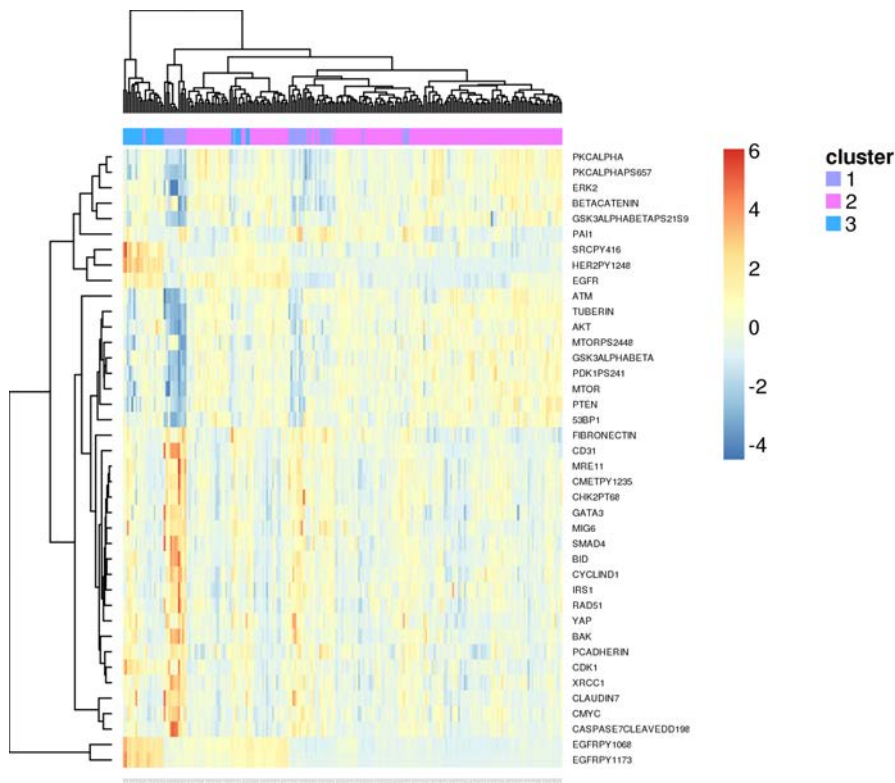
FIGURE 7.6

Heatmap showing RPPA-derived protein expression levels for TCGA glioblastoma tumour samples, with hierarchical cluster assignments indicated by colour bar. Only the 40 most differentially-expressed proteins (i.e. those discriminating between the cluster assignments) were used for this plot.

is that here we have specified `scale="row"`, this is to highlight the protein-wise differences between the clusters. The result should appear as in Fig. 7.6, which seems to show that one cluster (C2) has very high EGFR and pEGFR expression and one cluster (C1) has largely distinct expression across all other proteins in the panel (non-EGFR proteins are either higher or lower than in the other two clusters).

In Chapter 5, we encountered the `aheatmap` function from the *NMF* package as another way of drawing heatmaps. To make more complex heatmaps, such as Figure 6, this function can in fact be simpler. We don't necessarily need to specify the colour-schemes for either the main heatmap or the annotation bars, and it's possible to add as many annotation bars (to columns or rows) as desired. First, load the *NMF* package:

```
> library(NMF)
```


**FIGURE 7.7**

Heatmap showing RPPA-derived protein expression levels for TCGA glioblastoma tumour samples, with hierarchical cluster assignments indicated by colour bar. Drawn using the `aheatmap` function from the `/textitNMF` package.

Then, call the `aheatmap` function to draw the specified matrix, with annotation variables passed as *factor*-valued columns of a data frame:

```
> aheatmap(rppa.data[rppa.cluster.proteins,], scale="row",
+ annCol=data.frame(cluster=factor(rppa.clusters)))
```

This should result in [Figure 7.7](#).

7.4 Integrating multiple datasets

The previous analysis looks quite nice for a quick exploration of a dataset with no prior knowledge. However, what we would really like to do is relate

the protein expression to clinical characteristics of the tumours, or patient outcomes. For this we need to access a different dataset and integrate values from the RPPA data table to the clinical data. This will require constructing a mapping from the features of one dataset to the features of the other. In our example, we will use the tumour TCGA barcodes (which are the column names of the ‘RPPA_RBN’ file) which are also present in the ‘GBM_clinicalMatrix’ file. So first we load the clinical data file into the R workspace:

```
> clin.data <- read.table("GBM_clinicalMatrix",sep="\t",
head=TRUE,row.names=1)
```

As with the RPPA data, we have specified tab-separated columns, column headers and that the first column gives the row names. As this data table contains non-numeric values, we do not wish to try to convert it to a numeric matrix. We can inspect the first few rows and columns of the newly-created data frame `clin.data` by indexing it with square brackets:

```
> clin.data[1:5,1:3]
```

This shows us that the tumour barcodes are in the rows and the clinical characteristics are in the columns. We can check the dimensions of the whole data frame:

```
> dim(clin.data)
```

This command should give the following output:

```
[1] 629 138
```

We see here that there is data here for 138 clinical characteristics of 629 tumours, which is a lot more than we need here! You can get an idea of what the different clinical characteristics are by inspecting the column names of the data frame:

```
> colnames(clin.data)
```

Our first hurdle will be to align the rows of the clinical data frame with the columns of the RPPA data matrix. If we inspect the output of `rownames(clin.data)` and `colnames(rppa.data)`, we see that the column headers of the RPPA data matrix have ‘.’ where the rownames of the clinical data frame have ‘-’. We can correct this using the `gsub` function in R, which finds all instances of a pattern in the elements of a character vector and swaps them for a specified replacement:

```
> rownames(clin.data) <- gsub(rownames(clin.data),pattern="-",
+ replace=".")
```

Now we can test to see how many of the column names of the RPPA dataset have a matching row name in the clinical dataset:

```
> sum(colnames(rppa.data) %in% rownames(clin.data))
```

This command uses the slightly odd-looking operator `%in%`, which sees if there is any element in the object following the operator which matches each element of the object preceding the operator. In this case, the output is:

```
[1] 215
```

This indicates that every column from `rppa.data` has a corresponding row in `clin.data`. The reverse will not be the case, as there are so many more rows in `clin.data` than there are in `rppa.data`.

Now that the clinical data can be matched with the protein expression data, perhaps we wish to find out if there are any proteins that vary according to the sex of the patients. Then we could set up a design matrix with a column for the intercept and a column for the sex term (e.g. taking value 1 if the patient's sex field is 'FEMALE'):

```
> sex.design <- cbind(intercept=1,female=as.numeric(clin.data[
+ colnames(rppa.data),"gender"]=="FEMALE"))
```

Here we have used the column names of the RPPA data matrix to select rows of the clinical data frame, and selected the column "gender" from the clinical data frame (which gives the patients' sex), then tested each value to see if it is 'FEMALE'. Now we can proceed with the linear model fit:

```
> sex.lm <- lmFit(rppa.data,design=sex.design)
> sex.lm <- eBayes(sex.lm)
```

And we can inspect the top hits for the sex term:

```
> topTable(sex.lm,coef=2)
```

We can see that there are no proteins with so clear a difference based purely on sex, as nothing retains significance after multiple testing corrections. We can inspect this visually using a heatmap, given a colour-code to label the tumour samples:

```
> sex.cols <- c("yellow","green")[sex.design[,2]+1]
```

What I have done here is to create an index for the character vector specifying the two colours for our label, where the index uses the sex term of the

design matrix (1 if the corresponding clinical data field was ‘FEMALE’, 0 otherwise) plus 1, so that the value is 2 for tumours with the clinical data field ‘sex’ equal to ‘FEMALE’ and 1 otherwise. For convenience, we can select the IDs of the most differentially-expressed proteins:

```
> sex.proteins <- topTable(sex.lm,coef=2)$ID
```

Here we have used the ID field of the output from `topTable` as the row names appeared to be just numbers. Then we can create the heatmap:

```
> heatmap(rppa.data[sex.proteins,],col=bluered(100),
+ ColSideColors=sex.cols)
> legend("topleft",legend=c("male","female"),fill=c("yellow",
"green"))
```

The resulting plot shows that actually even with only the proteins with the most sex-distinct expression profiles, the tumour samples don’t segregate very clearly by sex. That’s probably reassuring in a way, but I have used this as an example because it can sometimes be worth checking factors that aren’t part of your hypothesis, just to make sure they don’t affect things too much. If they do, you can always take them into account in your models: in the *limma* context, that would mean adding a column to the design matrix for the variable that is not part of your hypothesis but may affect the expression levels, then not selecting the corresponding coefficient when extracting statistics using `topTable`.

7.4.1 Survival analysis

When investigating associations with clinical outcomes, not all information can be treated the same way. A particularly distinct example is that of survival times. Because patient survival (and some other events such as relapse) is only observed until the most up-to-date follow-up information, for all those patients who haven’t had the event (in this case, death) occur within the observation window, we can’t know how much longer they would have gone without the event occurring. We only know that it didn’t occur in the time from them entering the study until the last follow-up. There have been mathematical methods developed for dealing with this situation, some of which are described in Section 4.8. These are implemented in the R package *survival*, which should be installed and loaded as follows:

```
> install.packages("survival")
> library(survival)
```

The *survival* package needs to know both the event time (or time to last followup) and whether or not the event occurred within the observation window.

In the case of finding proteins with expression associated to patient survival in the TCGA glioblastoma multiforme dataset, we could use the column ‘days_to_last_followup’ from the clinical data frame for our event time and the column ‘vital_status’ for our event indicator. We then use the function `Surv` to create the required object:

```
> os.time <- clin.data[colnames(rppa.data),"days_to_last_followup"]
```

Here we have created a numeric vector called `os.time`, with the values of the ‘days_to_last_followup’ column of the clinical data frame, matched to the tumour samples in the RPPA data matrix.

```
> os.event <- as.numeric(clin.data[colnames(rppa.data),  
+ "vital_status"]=="DECEASED")
```

Here we have created a numeric vector called `os.event`, taking value 1 when the ‘vital_status’ column of the clinical data frame has the value ‘DECEASED’, and taking value 0 otherwise.

```
> gbm.os <- Surv(os.time,os.event)
```

Finally, we have created a survival object called `gbm.os` using the two vectors created previously.

Cox Proportional Hazards models can be fitted using the *survival* package in R, which enables a statistical evaluation of the relationship between a predictor variable (e.g. protein expression) and a censored outcome (e.g. patient survival). It is relatively simple to invoke, as the following example producing a proportional hazards regression model for patient survival predicted by the expression level corresponding to the first row of the RPPA data matrix (an arbitrarily-chosen example):

```
> coxph(gbm.os ~ rppa.data[1,])
```

The tilde `~` is used to describe that we wish to associate the outcome on the left hand side with the predictor(s) on the right hand side. In this case we have only one predictor, but it is possible to include more. You should see that this example produces a model with a hazard ratio of 0.64 and a p-value of 0.27. This hazard ratio means that on average, an increase in the expression measurement for that protein (‘14-3-3_epsilon’) of 1 decreases the probability of death having occurred during any given time interval by $\frac{1}{0.64} = 1.56$. However, there is sufficient variation in the observed effect that there is a 27% chance that a randomly-generated variable would associate with the outcome to the same degree of accuracy (i.e. the model fit wasn’t good enough to read much into the result). On its own this can be moderately useful, but is

especially powerful when we combine it with some programming functionality within R. For example, we can use a `for` loop to fit a model for each row of the RPPA data matrix in turn (provided we first create vectors to store the results):

```
> all.hrs <- rep(NA,nrow(rppa.data))
> all.pvals <- rep(NA,nrow(rppa.data))
> for(i in 1:nrow(rppa.data)){
+ coxphmodel <- coxph(gbm.os ~ rppa.data[i,])
+ all.hrs[i] <- summary(coxphmodel)$coef[1,2]
+ all.pvals[i] <- summary(coxphmodel)$coef[1,5]}
```

In this series of commands, we have first used the `rep` function to make a vector full of missing values ('NAs'), with the same length as the number of rows in the data matrix `rppa.data`. Then a `for` loop is set up to use the *iterator* variable `i` to select each row of the `rppa.data` matrix and each element of the vectors `all.hrs` and `all.pvals` in turn. The loop runs the sequence of commands between the curly brackets `{` and `}`, changing the value of the variable named in the `for` function. In this case, there are three commands to be run:

1. Create an object called `coxphmodel` which stores the result of fitting a Cox proportional hazards regression model (using the function `coxph`) to the censored survival times (`gbm.os`) to row `i` of the matrix `rppa.data`.
2. Apply the `summary` function to the object that stores the fitted model `coxphmodel`, in order to extract statistics. This results in a list of outputs, from which the `coef` element is an *array* storing the model coefficients, hazard ratios and p-values for each variable included in the model. The 2nd column of the 1st row of this array is the hazard ratio for the association between the first variable in the model (in this case, row `i` of the matrix `rppa.data`) and the outcome. So we set element `i` of the vector `all.hrs` to be the 2nd column of the 1st row of the `coef` element of the result from the `summary` function.
3. The same as the previous command, just set value `i` of the vector `all.pvals` to the 5th column of the 1st row of the `coef` array, as this corresponds to the p-value for the association between the first variable and the outcome.

We should now have fitted 171 proportional hazards models, and summarized each by obtaining the hazard ratios and the p-values, using the function `summary`. We may wish to use these values to find the most interesting model fits (i.e. the proteins with the strongest association between expression level

and patient survival time). We can do this by combining the results together into a *data frame*, along with the corresponding protein names:

```
> rppa.coxph.df <- data.frame(Protein=rownames(rppa.data),
+ HR=all.hrs,p.value=all.pvals)
```

Each column of the *data frame* is provided as a separate argument to the `data.frame` function, with the column names being specified as the argument names. It is probably more useful if we sort the table so that the proteins with the smallest p-values are at the top:

```
> rppa.coxph.df <- rppa.coxph.df[order(rppa.coxph.df$p.value,
+ decreasing=FALSE),]
```

We use the `order` function to find the order of the values in the first argument (in this case the p-values), specifying `decreasing=FALSE` because we want the lowest values at the top (not the highest). To look at the first few rows of this table we run the following command:

```
> rppa.coxph.df[1:4,]
```

Then see the output:

```
Protein HR p.value
128 PAI1 1.3204615 0.0004154699
66 IGFBP2 1.3917282 0.0005891653
36 CHK2 0.4511387 0.0042050539
7 ACC1 0.6666609 0.0080703250
```

This looks encouraging, but we should remember that actually we performed 171 identical hypothesis tests, and therefore purely by chance we'd expect $171 * 0.05$ (more than 8) randomly generated variables to have an association to our outcome variable with a p-value of less than 0.05. We should take this into consideration by applying multiple testing correction, which can be done easily in R using the `p.adjust` function. We can add the adjusted p-values into the data frame as follows:

```
> rppa.coxph.df$adj.p.val <- p.adjust(rppa.coxph.df$p.value,
method="fdr")
```

Here we specified the values of a new element in the data frame `rppa.coxph.df` (as the column name 'adj.p.val' didn't already exist), and so it created a new column with the values generated by the `p.adjust` function. There are a number of possible options for methods to use for multiple testing adjustment, but

the ‘fdr’ (estimated family-wise false discovery rate) is usually a reasonable choice. We can now re-inspect the model fit statistics:

```
> rppa.coxph.df[1:4,]
Protein HR p.value adj.p
128 PAI1 1.3204615 0.0004154699 0.05442656
66 IGFBP2 1.3917282 0.0005891653 0.07659148
36 CHK2 0.4511387 0.0042050539 0.54245195
7 ACC1 0.6666609 0.0080703250 1.00000000
```

So, interestingly, only the first two proteins have a strong enough association that we expect it to be very unlikely to have appeared by chance, even when we test the association of 171 different variables.

To get an idea of what this represents, we can stratify patients based on expression level of the outcome-associated protein and visualize the survival curves of the two groups using a Kaplan-Meier plot. First, let’s create an indicator variable that says whether or not each tumour’s expression level of the top hit from the proportional hazard regression model fitting (‘PAI1’) is greater than its median expression value (this ensures that the patients are stratified into two equally-sized groups, although it does disregard the actual distribution of the expression values):

```
> pai1.high <- as.numeric(rppa.data["PAI1",]
+ >median(rppa.data["PAI1",]))
```

This command creates a numeric vector `pai1.high` that contains value 1 when the corresponding column of the ‘PAI1’ row of the `rppa.data` matrix is greater than the median value of the whole ‘PAI1’ row, and contains value 0 when the corresponding column is not greater than the median. Next, we can use the `survfit` function to create the Kaplan-Meier survival curves for each of the two groups:

```
> plot(survfit(gbm.os ~ pai1.high),col=c("black","red"),lwd=2)
```

Here we have specified that the two survival curves should be plotted with black and red lines (corresponding to expression \leq median and expression $>$ median, respectively) with width twice as thick as default (by specifying `lwd=2`). We probably also want to add a legend to the plot, saying which colour represents which patient group:

```
> legend("topright",legend=c("low-PAI1","high-PAI1"),fill=c("black"
+ ,"red"))
```

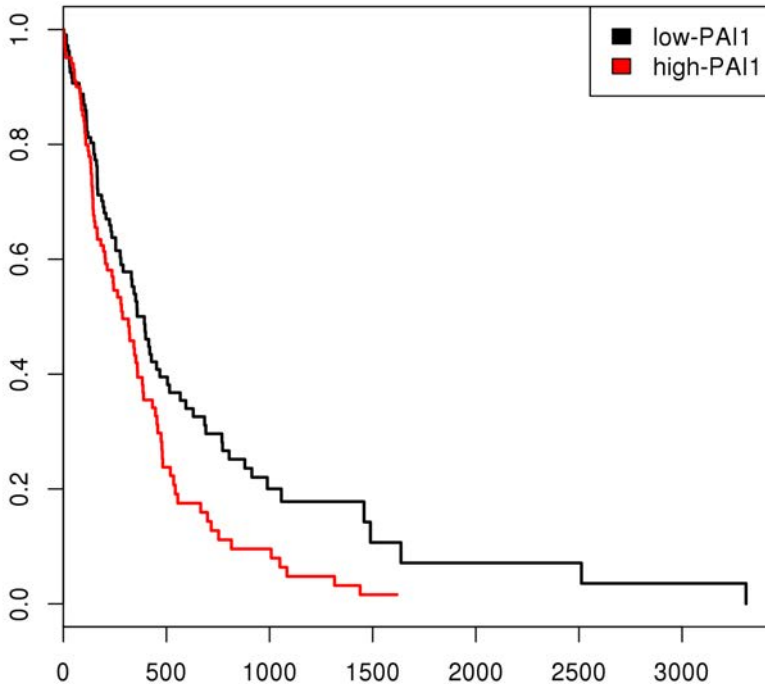


FIGURE 7.8

Kaplan-Meier plot showing overall survival for glioblastoma multiforme patients in TCGA dataset whose tumours had lower (black line) or higher (red line) expression of SERPINE1 as measured on the RPPA platform.

The result should appear as in Fig. 7.8, and illustrates the fact that patients whose tumours had higher expression of the PAI1 protein were more likely to die sooner than those with lower expression.

Not satisfied that we have found something interesting, it may be pertinent to think of possible confounding factors that might explain the result we see. The data we downloaded contains a range of clinical information about the patients and tumour samples, which may explain either protein levels or some of the other clinical variables. To pick an example, we can see that one of the columns of the clinical data frame is called ‘chemo_therapy’. If we use the function `table` to look at the distribution of values of this variable across the patients whose tumours were profiled on the RPPA platform, we see:

```
> table(clin.data[, "chemo_therapy"])
NO YES
139 95 395
```


So 139 don't have information about whether they received chemotherapy or not, 95 hadn't received chemotherapy, and 395 had received chemotherapy. It may not make sense to include survival data from patients who didn't have chemotherapy along with those who did, so perhaps we should repeat the model fit using only the patients who had received chemotherapy? To do this, we could create a vector containing the column numbers for tumours whose patients received chemotherapy:

```
> had.chemo <- which(clin.data[colnames(rppa.data),
+ "chemo_therapy"]=="YES")
```

Then we can repeat the model fit using only the subset of the patients who definitely received chemotherapy:

```
> coxph(gbm.os[had.chemo] ~ rppa.data["PAI1",
+ had.chemo])
```

It looks like the relationship in this subset of patients is pretty similar to the one we saw before. Perhaps there are other confounders though?

One last example for this tutorial is included to show how to include potential confounder variables into survival models in R. You simply add these terms into the formula following the \sim (tilde). For example, we could repeat the previous model fit but include age as a possible factor that might influence patient survival. First it makes it a little more convenient if we create a vector containing the ages for the patients in question:

```
> gbm.age <- clin.data[colnames(rppa.data)[had.chemo],
+ "age_at_initial_pathologic_diagnosis"]
```

Here we use the vector of indices `had.chemo` to select only column names corresponding to the patients who had received chemotherapy, then use this to look up the appropriate rows in the clinical data frame. Perhaps it would be informative to look at a histogram of the ages in this cohort? Let's leave that for now, and go straight on to re-evaluating the proportional hazards model fit to take into account the effect of age on patient survival:

```
> coxph(gbm.os[had.chemo] ~ gbm.age +
+ rppa.data["PAI1",had.chemo])
```

You will see that all we have done in relation to the previous model fit is added in the term `gbm.age` after the \sim , followed by a `+` (plus sign). The results of this model fit seem to suggest that there is a strong association between age and patient survival, and when taking that into account the association between SERPINE1 expression and patient survival is rather weaker than originally

estimated. In fact, if we had included age as an additional variable in the original set of model fits across the whole RPPA dataset (using only data from the cohort of patients who received chemotherapy) and then adjusted for the multiple hypothesis tests that were carried out, then the association between SERPINE1 expression and shorter patient survival would no longer be much stronger than the sort of associations you would expect to see appearing purely by chance when looking for associations among that many proteins.

7.5 Multiple molecular endpoints

In this tutorial so far we have seen integration of a molecular dataset (RPPA) with clinical data. Another situation which arises is one in which we have multiple ‘layers’ of molecular and genetic characteristics for the same samples, and wish to look for patterns in how one type of measurement influences another. For example, the Xena Browser lists multiple datasets for the Glioblastoma Multiforme cohort from TCGA, including: DNA copy number (SNP arrays), cytosine methylation (Illumina HumanMethylation arrays), gene expression (Affymetrix arrays and RNA-seq), gene-level somatic mutation (from sequencing) and protein expression (RPPA). If we had identified a protein from our RPPA dataset as being of particular interest, one way to characterize its behaviour would be to find genes whose expression is associated with the protein expression level. We could perform this analysis by downloading a gene expression dataset from the Xena Browser, following the steps in Sections 1–2 of this tutorial, but instead of downloading the ‘RPPA_RBN’ dataset shown in [Fig. 7.3](#), download the ‘gene expression array - AffyU133a’ dataset (this is the gene expression dataset with the most samples, and therefore has greatest likelihood of overlapping most samples with the RPPA dataset). Once you have downloaded the file ‘HT_HG-U133A.gz’, unzip it so that you have a file ‘HT_HG-U133A’ and then read this matrix into R:

```
> gx.data <- as.matrix(read.table("HT_HG-U133A", sep="\t",
head=TRUE, row.names=1))
```

This will take a bit longer than the RPPA dataset as this dataset is much larger! With the two molecular datasets (RPPA and Affymetrix array) loaded into R, analysis begins by finding the patients whose tumour samples match between the two datasets:

```
> shared.samples <- intersect(colnames(rppa.data), colnames(gx.data))
```

This command uses the `intersect` function to return the elements of the two vectors of column names which are present in both matrices.

Now it will be possible to find genes whose expression is significantly correlated with a candidate's protein expression (say, SERPINE1, to follow on from our previous example). We can make use of the fact that correlation between the measured levels of different molecular entities will be detectable as a linear association between variables: we can use the *limma* package to evaluate these relationships simultaneously. First we load the *limma* package:

```
> library(limma)
```

Then we need to set up our design matrix. For this we want to include an intercept and a variable giving the candidate's protein expression level, to which each gene's measured expression level will be fit using a linear regression model:

```
> cor.design <- cbind(intercept=1,PAI1=rppa.data[
+ "PAI1",shared.samples])
```

In this command, we have created a matrix called `cor.design` by joining two columns: the first column (named 'intercept') always takes value 1, and the second column (named 'PAI1') has the values from the matrix `rppa.data` for the row 'PAI1' and the columns listed in the vector `shared.samples` which we created previously. The `cor.design` table should have 185 rows.

With an appropriate design matrix created, we can fit linear regression models to all rows of the gene expression data matrix:

```
> cor.fit <- lmFit(gx.data[,shared.samples],design=cor.design)
```

You will see that in this command we have indexed the `gx.data` matrix by the shared column names (`shared.samples`). This is very important as it ensures that we are comparing the PAI1 protein expression in the design matrix to the gene expression from the same tumours (having used the same shared column names to index the values from `rppa.data` which went into the design matrix).

```
> cor.fit <- eBayes(cor.fit)
```

Now `cor.fit` contains all the statistics derived from the individual linear model fits. We can inspect these statistics for the two most significantly correlated probes from the gene expression dataset:

```
> topTable(cor.fit,coef=2,number=2)
logFC AveExpr t P.Value adj.P.Val B
SERPINE1 1.0462033 7.371430 12.526891 1.787062e-26 2.151980e-22 49.01303
LOX 0.6583057 5.323454 8.969556 3.263406e-16 1.493561e-12 26.22687
```

We can see which genes these are (which encouragingly includes SERPINE1, the gene encoding the PAI1 protein), and that they are both positively-correlated with the PAI1 protein expression (we know this because the \log_2 fold-change \log_2FC and the t statistic t are both greater than zero), which means that tumours with higher PAI1 protein level also had higher SERPINE1 and LOX gene expression. We could write out a list of the 100 most significantly-correlated genes³:

```
> pai1.100cor.genes <- rownames(topTable(cor.fit,coef=2,number=100))
> write.table(pai1.100cor.genes,file="GBM_PAI1_correlated100genes.txt"
+ ,quote=FALSE,row.names=FALSE,col.names=FALSE)
```

With a list of correlated genes, looking for biological processes which are known to involve an unusually large number of genes in the list can offer insight into what functions the protein of interest might be involved with in the samples that we are investigating. A helpful resource for this sort of exploratory *functional enrichment analysis* is DAVID [2]. Visit the webpage <http://david.ncifcrf.gov/summary.jsp>, and you can upload the file you just created ('GBM_PAI1_correlated100genes.txt') by clicking on the 'Browse...' button in the left-hand panel, and selecting 'OFFICIAL_GENE_SYMBOL' from the drop-down list under 'Step 2: Select Identifier'. Then upload the gene list, select species (Homo sapiens) and click on the button 'Functional Annotation Chart'.

It would be equally feasible to create a design matrix with the levels of a protein of interest and use the `lmFit` function to fit linear models to each of the proteins from the RPPA dataset in turn. The resulting model fit statistics should have the protein of interest perfectly-correlated with itself (i.e. a p-value of 0), but could be used to find potential regulators. The few most significantly-associated proteins could be uploaded to *STRING* (visit <http://string-db.org/> and click the 'multiple names' tab, then upload using the 'Browse...' button), and you can visualize interactions among the protein network. Another potential application would be to download the gene-level mutation indicators from the UCSC Cancer Genomics Browser, and find proteins or genes significantly correlated with mutation status of a gene of interest: this could give insight into the role of the gene in that tumour type. With a relatively simple set of data analysis tools at your command, and an ability to obtain data from the public access repositories, you can achieve a lot!

7.6 Summary

The aim of this particular tutorial is to give a working example of how straightforward it can be to explore datasets and even to search for patterns of

³Why 100? It's more or less completely arbitrary, although it is usually enough to give a signal in function enrichment analysis

association between multiple features from different datasets. The R syntax takes a bit of practice to get used to, and there are usually some little things that need to be done before different datasets can be linked together by common identifiers, but hopefully this shows you that a small set of analytical tools can be used in a whole variety of situations and can produce all sorts of results, including ones that may be of genuine interest in a field such as cancer research. The example using survival data is intended to illustrate the fact that there are often subtleties in the associations between variables in large datasets, which can be revealed through a process of exploration. Ultimately, the more data you have available, the more you can draw upon to explain the associations you see, and the more things you can rule out as confounding factors, the more likely the association you identified is to validate in a real-world setting!

Bibliography

- [1] J Zhu *et al*, “The UCSC Cancer Genomics Browser,” *Nature Methods* 6:239-240 (2009).
- [2] DW Huang & BT Sherman & RA Lempicki, “Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources,” *Nature Protocols* 4:44-57 (2009).

Analyzing Microarray Data in R

The availability of affordable, reliable gene expression microarrays through the 1990's and 2000's in many ways brought bioinformatics into the mainstream of molecular biology research. It became cheap enough for most research groups to generate tens to hundreds of thousands of measurements from their biological samples in an individual experiment. While microarrays for gene expression have largely been superseded by RNA-seq, there remains a vast wealth of data in the public domain from microarray platforms, and being able to analyze that data is rather like reading scientific literature. In fact, because there is so much information contained in these high-throughput molecular profiling studies, it is quite likely that a different scientific question would lead to different analyses and different findings being reported from the same dataset. There is much to be gained from revisiting previous data in the context of a new study!

Gene expression microarrays also benefit from standardized platforms with standardized pre-processing pipelines, which mean that it is relatively straightforward to go straight from a raw dataset to a normalized matrix of gene expression measurements. From that point, it becomes possible to apply the analytical methods we have already covered in the earlier chapters of this book. The examples provided in this chapter should also help consolidate what you have recently learned, and help develop an understanding of the ways analysis of large datasets can inform biological understanding. It is probably worth noting that there are two main designs of microarray: one-channel arrays measure intensity for a single fluorescent marker, and therefore one measurement is assigned to each probe (where each probe represents a unique target mRNA sequence); two-channel arrays label two different mRNA (usually actually cDNA) libraries with different fluorescent markers, and therefore two intensity signals (and two measurements) can be assigned to each probe. For the sake of simplicity, this chapter focuses on analysis of single-channel microarrays produced by Affymetrix, which are by far the most abundant in terms of publically-available gene expression data. There are of course other manufacturers of gene expression microarrays, but as all records on the GEO repository (which we will discuss in this chapter) require processed datasets to be made available, platform-specific instructions for pre-processing and normalizing all kinds of microarrays is unnecessary.

The first parts of this chapter will focus on obtaining publically-available microarray data in its raw (unprocessed) form, getting a sense of what the measurements represent, and preprocessing the data into a more readily-usable form. Then we will walk through a few standard analytical tasks: identifying genes that are differentially-expressed according to some experimental condition (variable) of interest; identifying features correlated to a gene of interest; exploring global transcriptomic relationships between samples using clustering; and finally accessing a dataset from a clinical study, complete with sample annotations, for performing survival analysis. These should cover the majority of applications routinely encountered in biomedical research.

8.1 Bioconductor

The computational tools in this chapter rely on the *Bioconductor* project. The preferred way to install the latest version of Bioconductor is to use the *BiocManager* package. This is obtained using the `install.packages` function:

```
> install.packages("BiocManager")
```

Then the function `install` can be called from the package. With no arguments specified, this function will install the core Bioconductor packages. Specific packages (and all their dependencies) can be installed by specifying the required package(s) as arguments to the `install` function. For example, to install the Bioconductor package *gcrma*:

```
> BiocManager::install("gcrma")
```

For older versions of R, the *BiocManager* package may not be available. There is another way to install the Bioconductor packages into your R environment from within the R console, using the commands:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite()
```

The first command loads the function `biocLite` that is defined on one of the Bioconductor organisation's computers. Then the second command runs this function to install the default set of Bioconductor packages to your system. If you ever need to install additional Bioconductor packages that aren't included in the basic installation, such as the packages *gcrma* and *annotate* that will appear later, simply enter the name of the package (in quotes) as an argument to the `biocLite` function like this:

```
> biocLite('gcrma')
```

8.2 Accessing microarray data from GEO

The **Gene Expression Omnibus**¹ is a comprehensive repository for functional genomics data. You may still be wondering if it's worth learning how to analyze data from gene expression microarrays, given that RNA-seq has become the industry standard for transcriptomic profiling. Hopefully a search of the GEO repository will help convince you of this: open the GEO web page in a browser, and click on the link to Browse by Platform. Sort the resulting table by number of samples, and see the number of samples for which data from different platforms is available. As of November 2018, a single microarray platform had 143,762 samples, as shown in Fig. 8.1.

Accession	Title	Technology	Organism(s)	Data rows	Samples	Series	Cor
GPL570	[HG-U133_Plus_2] Affymetrix Human Genome U133 Plus 2.0 Array	in situ oligonucleotide	<i>Homo sapiens</i>	54,675	143762	5019	Affyr
GPL17021	Illumina HiSeq 2500 (<i>Mus musculus</i>)	high-throughput sequencing	<i>Mus musculus</i>		111111	3792	GEC
GPL13112	Illumina HiSeq 2000 (<i>Mus musculus</i>)	high-throughput sequencing	<i>Mus musculus</i>		106684	5093	GEC
GPL11154	Illumina HiSeq 2000 (<i>Homo sapiens</i>)	high-throughput sequencing	<i>Homo sapiens</i>		100097	6592	GEC
GPL16791	Illumina HiSeq 2500 (<i>Homo sapiens</i>)	high-throughput sequencing	<i>Homo sapiens</i>		92950	4015	GEC
GPL13534	Illumina HumanMethylation450 BeadChip (HumanMethylation450_15017482)	oligonucleotide beads	<i>Homo sapiens</i>	485,577	82081	1163	Illum
GPL10558	Illumina HumanHT-12 V4.0 expression beadchip	oligonucleotide beads	<i>Homo sapiens</i>	48,107	77438	2309	Illum
GPL18573	Illumina NextSeq 500 (<i>Homo sapiens</i>)	high-throughput sequencing	<i>Homo sapiens</i>		76589	1402	GEC
GPL19057	Illumina NextSeq 500 (<i>Mus musculus</i>)	high-throughput sequencing	<i>Mus musculus</i>		68007	1399	GEC
GPL1261	[Mouse430_2] Affymetrix Mouse Genome 430 2.0 Array	in situ oligonucleotide	<i>Mus musculus</i>	45,101	53399	4116	Affyr

FIGURE 8.1

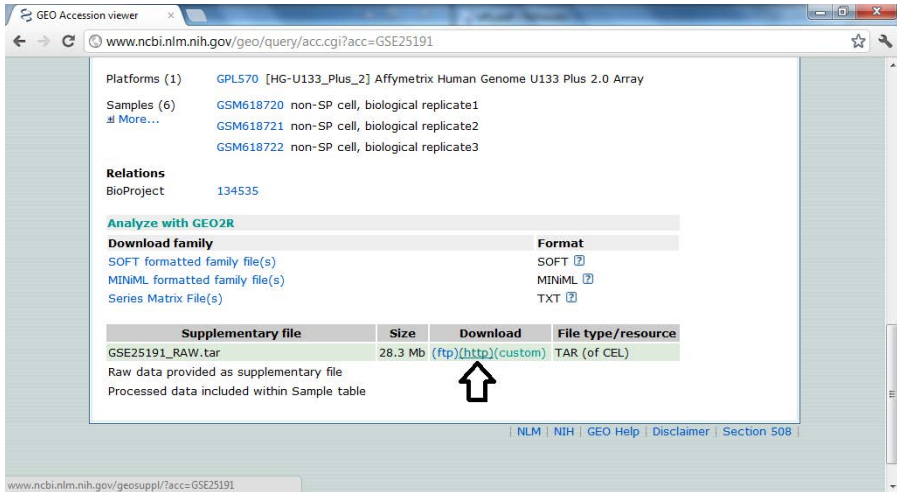
Screenshot showing table of platforms by number of samples.

For the worked example in this tutorial, we will use a gene expression microarray experiment that is on GEO with the experiment accession *GSE25191*. To find the record for this experiment, go to the GEO website and enter the accession number into the ‘GEO accession’ field.

You can download the raw data in a zipped ‘tar’ archive by clicking on one of the links near the bottom of the page, as illustrated in Fig. 8.2.

When the file has downloaded, you may need to extract the archive and decompress the individual compressed files (which will have the ‘.CEL.gz’ extension). The *affy* package can read in these compressed files, and it is probably easier

¹<http://www.ncbi.nlm.nih.gov/geo/>

**FIGURE 8.2**

Screenshot showing link to click to download raw data archive for GEO microarray experiment GSE25191.

to leave these as they are because the precise way to decompress the files will depend on your software (one common example is WinZip). It will make the rest of the exercise easier if all the individual raw data files (compressed or uncompressed) are placed into the same folder somewhere on your computer. The simplest application of Bioconductor's tools for manipulating Affymetrix data will typically load in ALL the '.CEL' (or '.CEL.gz') files from the current working directory.

8.3 Single-channel array analysis

The R package *affy* in Bioconductor provides most tools for reading, normalizing and analyzing single-channel microarrays (such as the Affymetrix genechip oligonucleotide arrays). Load the *affy* package in R using:

```
> library(affy)
```

8.4 Loading data

Raw data from Affymetrix microarrays is provided in 'cel' file format. If you wish to load into the workspace all raw data .CEL files currently in the working

directory, the ‘ReadAffy’ function exists to do this. This command is included here in case you wish to do some exploration of the raw data, although in practice it is rare that you would need to do this². **Note: in most cases you will want to jump straight to Section 8.6 to load data and normalize in a single step, then move onto Section 8.7. Sections 8.4–8.6 are included for educational purposes only.**

Use of the ‘ReadAffy’ function is straightforward:

```
> rawdata <- ReadAffy()
```

Now the object `rawdata` is created on the R workspace, and this contains all the information from the raw data files structured in a way that facilitates manipulation within R. The actual object is a sort of modified data frame called an *AffyBatch* object: this is particular to the Bioconductor packages.

All but the most recent Affymetrix microarrays have two versions of each probe on the array: the *perfect match* probe is the exact complement of the target sequence, but the *mismatch* probe differs by one base from the exact complement. The mismatch probes were included to get an estimate of non-specific binding from each probe. To obtain the intensity values of the perfect match or mismatch probes for each chip, use `pm()` or `mm()`, respectively. For most purposes of examining raw data, the perfect match values alone will suffice, so to avoid unnecessary complication we will ignore the mismatch values³. Therefore, to obtain raw expression values for the dataset:

```
> exprs <- pm(rawdata)
```

8.5 Data visualisation

With any microarray data, it is useful to examine your data visually to check that there are no obvious anomalies – there are plenty of possible sources for problems that can affect microarray experiments! Normalizations will usually smooth over such anomalies, so it is often worth having a look at the data before normalization.

²As you’re loading all the information for each microarray, the data files tend to be very large.

³The later Affymetrix microarrays no longer contain any mismatch probes, largely because it was found that the best normalization of the data didn’t use the mismatch probes.

8.5.1 Image plots

Assuming the layout of the chip is included in the ‘chip definition file’ that the *affy* package will access when loading the data, the raw data for each chip can be converted back into an image and plotted using the `image()` function. If we have the raw expression data in an object `rawdata`, as created in Section 8.3, to look at the image of the first array use:

```
> image(rawdata[,1])
```

In the above command, we have used the same method of indexing to get just one array’s worth of data from the *AffyBatch* object `rawdata` as the indexing that applies to arrays and data frames. The result of the command should be a figure like that shown in Fig. 8.3. The function `image` can be used to draw an unclustered heatmap of any numeric matrix, defaulting to a greyscale colour scheme.

GSM618720_1207_SK_3_4_H_IGNSP1.CE

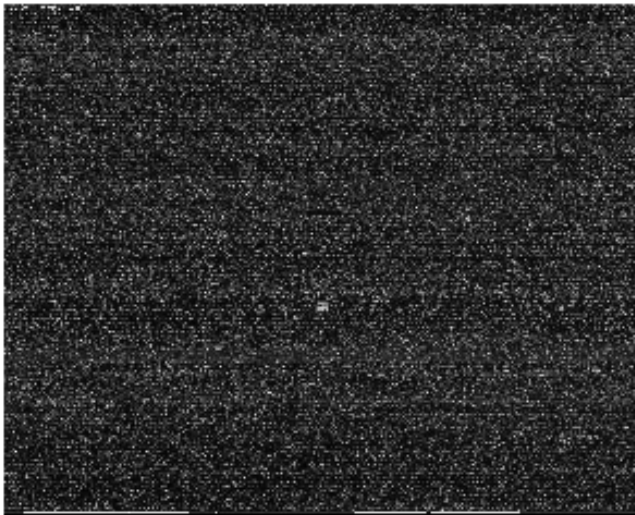


FIGURE 8.3

Reconstructed raw image from the first microarray in our experiment. This would show any obvious spatial biases affecting the intensity measurements from the array.

8.5.2 MA plots

An MA plot, or M vs. A plot, is a simple way of checking an array for intensity-dependent effects. For a two-channel array, it plots the log-ratio of intensity from one dye (Red) to the intensity from the other dye (Green) for each probe against the overall intensity of each spot, thus $M = \log_2\left(\frac{R}{G}\right)$ and $A = \frac{\log_2(R) + \log_2(G)}{2}$. If there are no intensity-dependent effects, which is what we would hope for, the MA plot should resemble a horizontal line around $M = 0$.

For a single-channel array like the Affymetrix chips, pairs of microarrays have to be used to construct an MA plot. The M values are defined as the ratio of intensities between the two chips for each probe, and the A values are the corresponding average intensities across the two chips. As with the two-channel case, if the MA plots involving a chip depart from the horizontal line around $M = 0$, it is likely that the intensity of the hybridisations measured on that chip may not be representative of the expression levels.

For MA plots of all pairwise combinations of chips in the experiment, given a table of expression values, you would use the function `plotMA()`. But as this may take quite a while, we will inspect a single MA plot comparing two individual arrays. We use the table of perfect match probes from our experiment, which we called `exprs`, selecting an appropriate subset of the expression table's columns. For example, to compare the first and third chips:

```
> plotMA(exprs[,c(1,3)])
```

This should result in [Fig. 8.4](#). Again, you should note that the standard approach to indexing arrays applies to the object `exprs`.

8.5.3 Scatterplots

A simple way to explore the relationships between different chips is to plot a scatter of the expression values for each probe-set in two chips. This is in fact equivalent to an MA plot rotated 45-degrees so that the horizontal line around $M = 0$ becomes the line $X = Y$ (where X s are the expression values for the first chip and Y s are the values for the second). This can be done using the basic `plot` function, and can take any of the usual parameters. To plot the same individual comparison as in the MA plot above, which should result in [Fig. 8.5](#), enter:

```
> plot(x=exprs[,1],y=exprs[,3])
```

As the first and third arrays are both non side-population samples, we would probably expect to see less variation than if we plotted the first array against

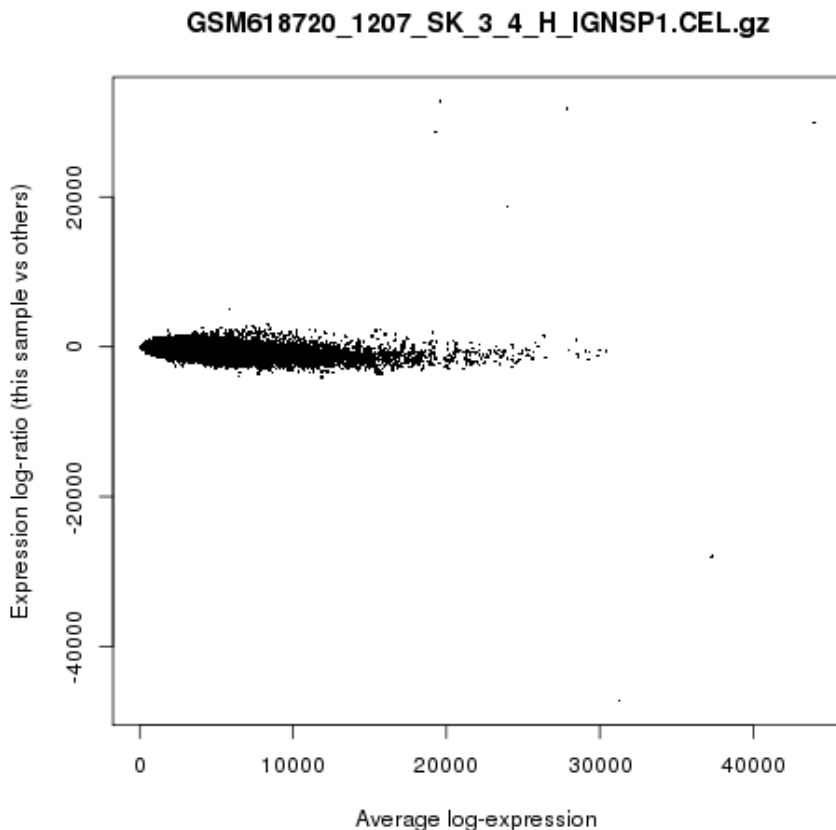
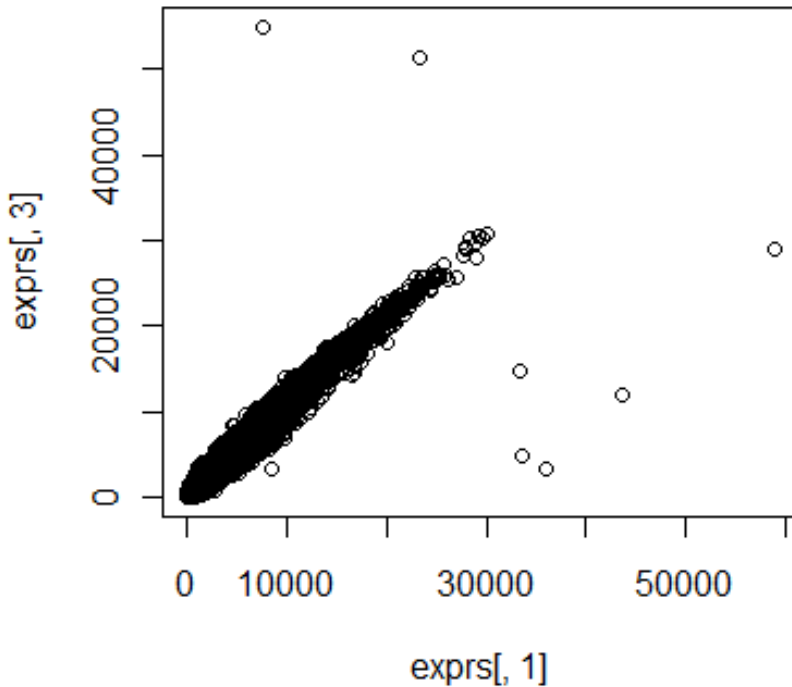


FIGURE 8.4

MA plot showing intensity ratios for all probes in a pair of microarrays, plotted against their average intensity. A departure from a horizontal line along $M = 0$ indicates intensity-dependent bias in the expression measurements from the arrays, but in this case we can see that aside from a handful of possible outliers, the general trend is as expected.

the sixth (which comes from a side-population sample). Sure enough, repeating the step above but replacing the third array with the sixth, we get the plot shown in [Fig. 8.6](#).

However, you may have spotted that the scale on the y-axis is different between [Fig. 8.5](#) and [Fig. 8.6](#), and this may potentially make the difference seem greater than it is. In order to compare the scatter plots more reliably, we should plot them in the same space. The following example uses the `points` function to

**FIGURE 8.5**

Scatter plot relating all probes' values in the first microarray of our experiment to those in the third microarray.

overlay a new set of points on an existing plot, and by specifying the colour as an argument to the plotting functions we can highlight which points come from which comparison:

```
> plot(x=exprs[,1],y=exprs[,6],col="blue")  
> points(x=exprs[,1],y=exprs[,3])
```

Now we can make a fairer comparison, and in [Fig. 8.7](#) we still see that the scatter comparing the first and sixth arrays (blue) is wider than that comparing the first to the third arrays (black).

8.5.4 Box plots

It is relatively straightforward to explore systematic bias in the overall distributions of intensity values from each microarray through the use of box

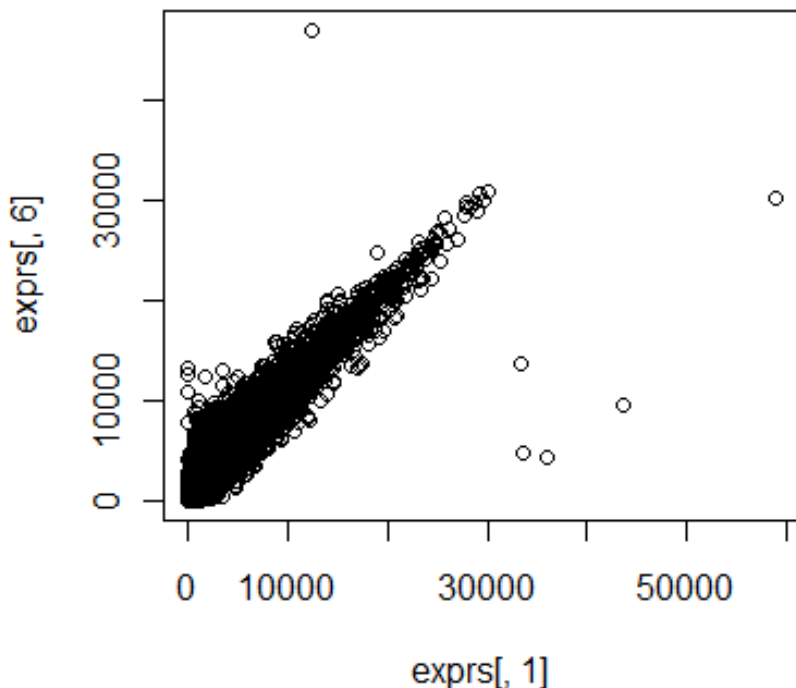


FIGURE 8.6

Scatter plot relating all probes' values in the first microarray of our experiment to those in the sixth microarray. The scatter between two more biologically dissimilar samples will tend to be wider than for pairs of more biologically similar samples.

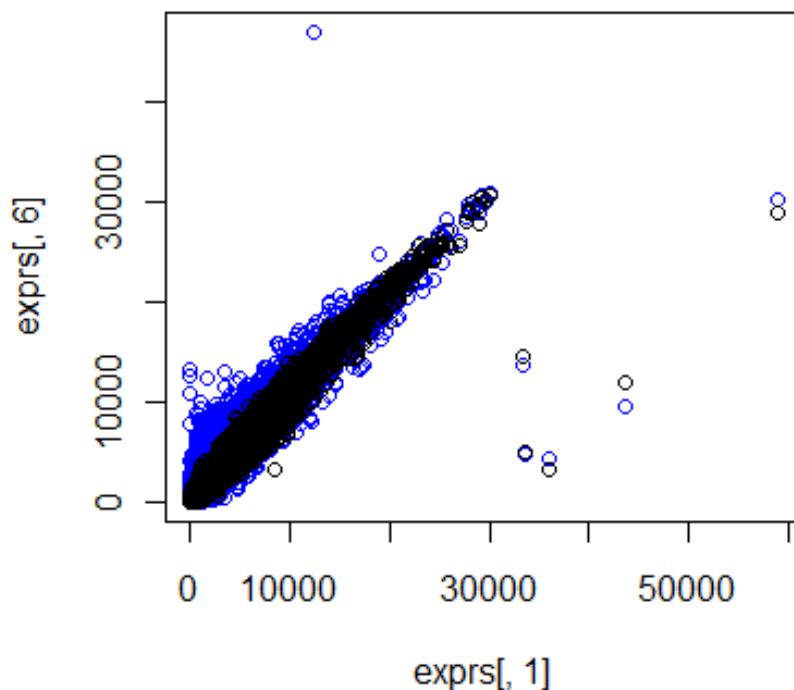
plots. The simplest application to our dataset can explore the perfect match intensities from each array:

```
> boxplot(exprs)
```

As we see in [Fig. 8.8](#), the vast majority of the values from each array are so much lower than the largest values that it becomes very difficult to see the main distinctions between the arrays. If we repeat on the log-transform of the intensity values as in [Fig. 8.9](#), any bias will be much easier to see.

```
> boxplot(log(exprs,base=2))
```

In the above command, the argument `base=2` specifies that we wish to use base 2 logarithms (often abbreviated \log_2). Logarithms with any base will typically have the same effect, but it has become standard with microarrays to use base 2 logarithms.

**FIGURE 8.7**

Combined figures showing two scatter plots that each compare the intensity values of probes from a pair of microarrays.

8.6 Normalizing data

Normalization of microarray data will remove systematic technical bias that affects the raw values. Bioconductor has implementations of a number of normalization methods for single-channel arrays. This includes (among others):

- `mas5()` – Affymetrix’s own normalization program.
- `rma()` – ‘Robust Multi-Chip’ average.
- `gcrma()` – A version of RMA that corrects for biases due to probe GC-content. This is provided in a package of its own, called *gcrma*. Install this with the command `biocLite('gcrma')`, and load the package with the command `library(gcrma)`.

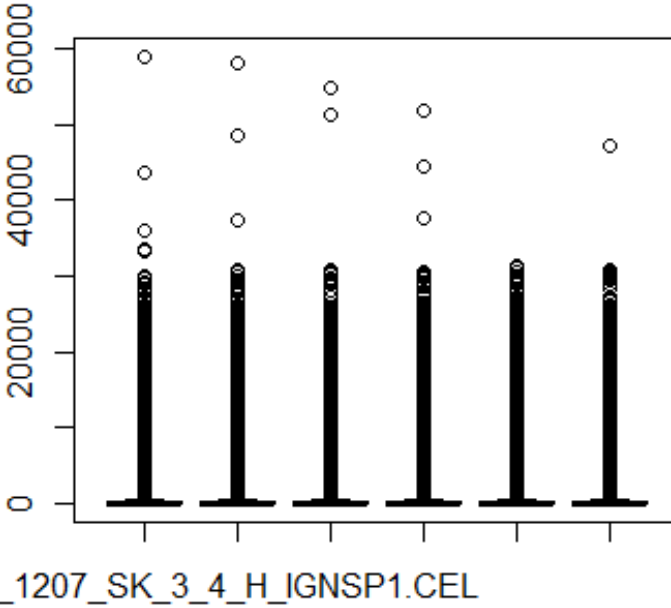


FIGURE 8.8
Box plot of the intensity values from each array.

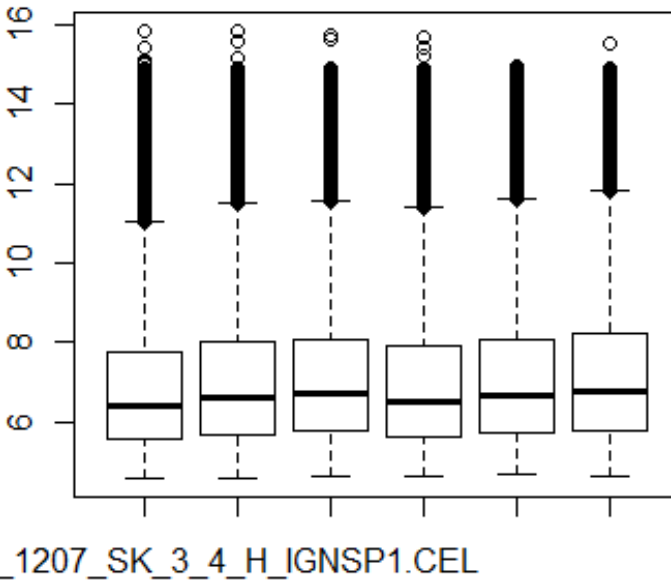


FIGURE 8.9
Box plot of the log-transformed intensity values from each array.

GCRMA is good but takes significantly longer than RMA, so you may just want to use RMA now:

```
> normdata <- rma(rawdata)
```

If you have a particularly large dataset, it is worth knowing about the `justRMA` command, which bypasses loading all the raw data into the workspace before normalization. Instead of the steps given in Section 8.4, an alternative would be to change R's working directory (using the function `setwd()`) to the folder containing the '.CEL' (or compressed '.CEL.gz') files, then enter:

```
> normdata <- justRMA()
```

The result of either of the above applications of the RMA method is an object on the R workspace `normdata`, which is a data structure called an *ExpressionSet*. This is another custom data structure used by Bioconductor, used because it conveniently organises the information required to work with a normalized microarray dataset. To obtain an array containing the normalized expression values, use the `exprs()` function:

```
> normexprs <- exprs(normdata)
```

Now we can repeat the box plot we performed in the previous section, using the RMA-normalized values. This will give us the result seen in Fig. 8.10,

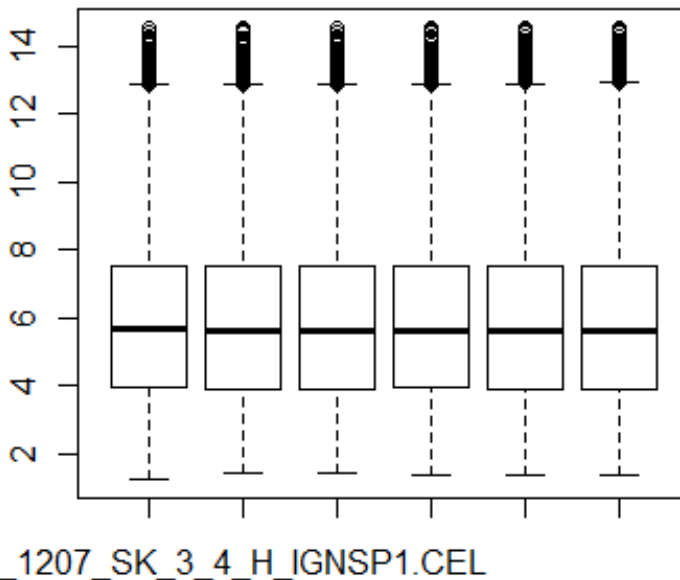


FIGURE 8.10

Box plot of the RMA-normalized expression values from each array.

in which we can see the overall distributions from the individual arrays have been transformed to fit the same pattern:

```
> boxplot(normexprs)
```

Vectors of the names of the samples and the identifiers for the probes on the microarrays can be retrieved using the functions `sampleNames()` and `featureNames()`, respectively. The outputs will be listed in the same order as they occur in the array of normalized expression values `normexprs`.

```
> samples <- sampleNames(normdata)
> probesets <- featureNames(normdata)
```

8.7 Differential expression (linear models)

The *limma* package uses linear models to analyse microarray data, which is a particularly powerful idea as the same framework can be used for both simple comparisons and arbitrarily complex experimental designs. The principle of *limma* is to evaluate how well the expression of each gene (Y in Equation 8.1) can be explained by some specified variables (X in Equation 8.1). The linear model framework is shown in Equation 8.1, which should be familiar:

$$Y = \beta_0 + X_1\beta_1 + \dots + X_n\beta_n + \epsilon. \quad (8.1)$$

In the equation, each X refers to a variable of interest for which we wish to consider an affect on gene expression, and the corresponding β values are the (unknown) coefficients that characterise the affect of each of these variables. β_0 is a special case, that of the intercept, which will capture the *baseline* level of expression of the gene across the dataset. The final component of the model is ϵ , which is the *residual* of the model fit: essentially all the variation of the gene's expression that is not explained by the rest of the model, this can be used to characterise how well the specified model fits the data.

In the process of evaluating linear models from microarray data analysis, the values of the X variables in the model framework of Equation 8.1 must be specified. This is done through the creation of a *design matrix* to be passed as an argument to the core *limma* function that does the linear model fitting and evaluation: `lmFit`.

8.7.1 Design matrix

The design matrix indicates, for each of the arrays in the dataset, which values of the various experimental variables (the X s in Equation 8.1) apply. It is both the reason for the power of the *limma* approach and the reason it seems a bit complicated at the start. The design matrix can be constructed in a number of ways, but the principle is that it will be a table with a column for each experimental variable and a row for each microarray in the dataset.

If there is only one experimental variable, the design matrix can be as simple as a vector with a number value for each microarray in the experiment. However, even if there is only one explicit experimental variable we normally need to include an intercept term.

For numeric variables, such as a simple indicator like the variable ‘SidePop’ specified in the targets file shown in Fig. 8.2, a design matrix can be constructed using the `cbind` function. This function in R constructs an array through combining the vectors passed to it as individual columns, as should be illustrated in the following example:

```
> design <- cbind(c(1,1,1,1,1,1),c(0,0,0,1,1,1))
> design
[,1] [,2]
[1,] 1 0
[2,] 1 0
[3,] 1 0
[4,] 1 1
[5,] 1 1
[6,] 1 1
```

In the above example, each of the two columns of the array `design` represent the values of one of the variables in our experiment, for each of the microarrays in the dataset. The first column is the intercept, which has a constant value of 1 across all microarrays. The second column indicates whether the corresponding microarray profiled samples from side-populations (value = 1) or non-side-populations (value = 0). Using this design to fit linear models with *limma* will enable us to evaluate the significance of impact on each gene’s expression of whether or not the samples are isolated side-populations or not.

In cases with categorical variables like this, each element of the design matrix takes a value of 1 if the microarray corresponding to that row of the matrix has the value corresponding to the column of the matrix (e.g. in this example the 2nd column effectively encodes *does this sample have the variable ‘SidePop’=1?*, so that `design[4,2]=1` and `design[3,2]=0`). For a slightly more complex example (with named columns), see below:

```

> design2 <- cbind(intercept=1,pair2=c(0,1,0,0,1,0),
pair3=c(0,0,1,0,0,1))
> design2
intercept pair2 pair3
1 1 0 0
2 1 1 0
3 1 0 1
4 1 0 0
5 1 1 0
6 1 0 1

```

Now in the new design matrix `design2` the second column encodes *does this sample have the variable 'Pair'=2?* and the third column encodes *does this sample have the variable 'Pair'=3?* When it comes to calculating the coefficients and significances corresponding to these two values of the 'Pair' variable, they will characterise how a gene's expression varies between those samples with 'Pair'=1 and those samples with the value corresponding to each column. When we come to using the results of `limma`'s linear modelling, these individual coefficients can be combined to perform an ANOVA-type analysis on the categorical variable. Naming the vectors that will make up the columns of the design matrix is generally a good idea, as it helps us remember which coefficient in the model corresponds to which experimental variable. You can alter the column names after creating a design matrix using the `colnames` function. For example, if we wanted to go back to add column names for the first design matrix:

```

> colnames(design) <- c("Intercept","Sidepop")

```

8.7.2 Fitting linear models

The main part of the `limma` analysis is fitting and evaluating linear models of each gene's expression, using the `lmFit` function. Assuming we have a normalized expression data matrix `normexprs` and a design matrix `design`, a model is simply fitted using the commands:

```

> fit <- lmFit(normexprs, design)
> fit <- eBayes(fit)

```

The second command performs an *empirical Bayes moderation* of the t-statistics associated with each coefficient in the linear model. Moderation of t-statistics makes the resulting analysis more robust to over-inflated significance estimates due to the fact that there are typically only a few measurements for each gene with each value of the linear model variables, and the variance estimate (which is part of the denominator of the t-statistic) can be artificially low.

Lists of the top differentially-expressed genes for specified coefficients can be obtained using the `topTable()` command. To find the top differentially-expressed genes corresponding to the difference between side-populations and non side-population samples, enter:

```
> topTable(fit,coef=2,adjust="fdr")
```

Here `coef` specifies which column of the design matrix to use for the comparison, and `adjust` specifies what statistical adjustments should be used to take into account the considerable multiple testing (in this case ‘false discovery rate’). In fact, it was unnecessary specifying this as `fdr` is the default multiple testing adjustment in the `topTable` function.

If we instead wished to examine the effect of the cultures from which the samples were isolated, we would have to fit a different set of linear models, using the design matrix `design2`:

```
> fit2 <- lmFit(normexprs,design2)
> fit2 <- eBayes(fit2)
```

Now we could find genes with expression variation best explained by changes to the categorical variable *Pair*, by specifying that we wish to include effects from both the second and third model terms. You may notice that this table has a column for `F` rather than `t` as before: this is because the significance of the combined effect of multiple linear model terms is evaluated with an `F` statistic rather than a `t`-statistic.

```
> topTable(fit2,coef=c(2,3))
```

8.7.3 Making use of the results

While it is exciting that we have been able to identify lists of probesets with significant differential expression, it is clear that in its current state the list output from the `topTable` function is of limited use. The first question most people have upon seeing such a table is: ‘what *are* those genes?’ The automatic annotation microarray experiments that performed by the `ReadAffy` or `justRMA` functions do not extend as far as looking up the gene symbols, so this is something we have to add to the results. First we need to load additional Bioconductor packages: one called *annotate*, and another that contains the annotation information for the microarray platform that has been used for the experiment. A list of the available annotation packages and the platforms they correspond to can be found at <http://www.bioconductor.org/packages/release/data/annotation/>, and the one to use for this purpose will have a name ending in ‘.db’. Once the appropriate packages have been loaded, you can add the gene symbols to

```
> fit1.out[1:10,c(1,7,2:5)]
      logFC      Symbol AveExpr      t      P.Value      adj.P.Val
244829_at  -3.512223 LINC00518  4.280104 -24.25504  1.294201e-07  0.007076045
201667_at  -4.290522      GJA1  7.151514 -12.56086  8.528587e-06  0.121794106
241079_at   2.113882      <NA>  4.534789  12.33570  9.551411e-06  0.121794106
1560422_at  1.938365      <NA>  4.240167  11.57664  1.419890e-05  0.121794106
204115_at   1.489710     GNG11  7.129100  11.41084  1.553265e-05  0.121794106
223875_s_at  2.110507     EPC1  6.656454  11.16331  1.780061e-05  0.121794106
207086_x_at -1.401677      <NA>  6.445216 -11.12425  1.819208e-05  0.121794106
207663_x_at -1.865800      <NA>  4.392493 -11.10017  1.843839e-05  0.121794106
241855_s_at  1.408529      <NA>  6.137816  10.67595  2.347218e-05  0.121794106
214844_s_at -2.815481     DOK5  6.797450 -10.53924  2.541796e-05  0.121794106
```

FIGURE 8.11

Annotated table of differentially-expressed genes.

the output of a linear model fit that has already been constructed. For example, to add gene symbol annotation to the first linear model that was fitted in Section 7.3, we first load the annotation package *hgu133plus2.db*:

```
> library(annotate)
> library(hgu133plus2.db)
```

We can use the `topTable` command (in the same way we did before) to create the output, but then add an additional column to the resulting table, specifying the gene symbol that corresponds to the probe-set in question:

```
> fit.out <- topTable(fit,coef=2,number=20)
> fit.out$Symbol <- unlist(mget(rownames(fit.out),hgu133plus2SYMBOL))
```

This command uses the `mget` function to look up the gene symbol annotations for each rowname in the `fit.out` data frame, then add this as an additional column. Inspecting the first few rows of the table `fit.out`, we should see the table shown in Fig. 8.11. A description of the various columns included in the table follows:

- **ID** – this is the identifier for the probe-set from which the corresponding measurements come.
- **Symbol** – the official symbol for the gene into which the probe-set maps.
- **logFC** – the log fold-change associated with the contrast, this is equivalent to the β term in the linear model given in Equation 8.1, and represents the typical change in expression value for the probe-set resulting from a unit increase of the variable associated with the term (X in Equation 8.1)⁴.

⁴To illustrate, if we have a two-class categorical variable then `logFC` is the average log fold-change of the probe's expression value between the two classes. However, if we were modelling the effect of age (in years, this is a numerical variable) on expression then `logFC` will be the average amount the probe's expression value increases with a one-year increase in age.

- `AveExpr` – the average expression value for the probe-set.
- `t` – the t statistic derived from the value of the coefficient and the residual.
- `P.Value` – the p-value associated with the t statistic for the term, an indication of how unlikely such a clear trend as that observed would arise in a random set of normally-distributed data.
- `adj.P.Val` – the adjusted p-value to take into account multiple testing, this gives an indication of the *family-wise error rate* for all probesets where the model fit was at least as good as it is for this one (i.e. all probes in the table from the top down to this one).
- `B` – the *log odds* of differential expression, the actual values of this depend on an assumption of a certain proportion of the probesets on the array are differentially expressed, which may be erroneous, but it is normally a good way of ordering the genes. In addition, as it is less intuitive to interpret than the p-value, typically it is the p-values and adjusted p-values that are used to summarise the goodness of the model fit.

The default use of the `topTable` command only returns information for the most significant 10 probesets, but in practice we will often wish to obtain information for many more. It is fairly straightforward to specify the number of probesets to include in the table, for example:

```
> topTable(fit,coef=2,number=20)
```

In addition, you can specify filters so that the table includes only those probesets with an adjusted p-value below a certain threshold, or those with a log fold-change above a certain threshold, or some combination of the above. For example, if we wished to return ALL probesets where the model fit had an adjusted p-value less than 0.05 and a fold-change of at least 2:

```
> topTable(fit,coef=2,number=nrow(normexprs),p.value=0.05,
+ lfc=log(2,base=2))
```

In the above command, the specified number of `nrow(normexprs)` is the total number of rows in the whole data matrix, which will correspond to the number of probesets on the microarray. Additionally, a two-fold expression change threshold was specified as `lfc=log(2,base=2)` because the models are fitted in terms of expression values that are base 2 logarithms.

For using the output in downstream analyses, or for publication purposes, you will need to export the table to file from R. This can be done using the `write.table` command that was introduced in the *Introduction to R* tutorial. To write the default table (removing the row numbers) out to a

tab-separated text file called 'fit2table.txt' in the current working directory, you could enter:

```
> fit.fullout <- topTable(fit,coef=2,number=nrow(normexprs),
+ p.value=0.05,lfc=log(2,base=2))
> fit.fullout$Symbol <- unlist(mget(rownames(fit.fullout),
+ hgu133plus2SYMBOL))
> write.table(fit.fullout,file="fittable.txt",sep=" ",
+ quote=FALSE,row.names=FALSE)
```

8.7.4 Postscript: Assumptions

It should not go without comment that the statistics provided by limma are calculated on the basis of assumptions about underlying distributions of the data. One of the assumptions that can be tested is that the errors in each fitted linear model are normally distributed. We can also assume that if most genes don't have expression associated with a given experimental variable, the distribution of p-values derived from fitting linear models to all genes should be uniform between 0 and 1.

8.8 Clustering and correlation

Statistical correlation measures provide a means of assessing the similarity between trends in data. This can, for example, be a useful way of associating different genes together based on the 'shape' of their expression profiles across the dataset. Clustering is generally the task of associating similar entities from a dataset: this could involve finding groups of genes with similar expression levels across a set of samples, or it could involve finding groups of samples with similar expression levels of certain sets of genes. Both correlation and clustering are therefore concepts that deal with the task of characterising similarity between elements from a dataset, but they are quite different concepts and must be applied in different ways.

8.8.1 Expression profiles

As correlations reflect trends in data, it is useful to be able to get an idea of what a trend in an expression dataset might represent. An 'expression profile' for a given probe-set simply shows that probe-set's measurements across the individual microarrays in a dataset, and can easily be plotted using the `plot` function. If we know the probe-set identifier for a particular gene and want to see how its expression values vary across the dataset, first find the relevant row of the expression table and then plot it as a line. For example, if we wished to inspect the expression profile for the most

significantly differentially-expressed probe-set (“244829_at”, as shown in the table in Fig. 8.11) in the side-population gene expression dataset:

```
> gIndex <- which(featureNames(normdata)=="244829_at")
> plot(normexprs[gIndex,], type="l")
```

This is particularly useful when used in conjunction with the `ylim=...` parameter and successive calls to the `points()` function with different line colours to compare how different genes vary across the experimental samples in the dataset.

8.8.2 Correlation

Based on the ‘guilt-by-association’ principle, an assumption that genes sharing a similar trend in their expression values are likely to be regulated in a similar way or involved in similar processes, we can use R’s correlation function to get a quantitative measure of similarity between the trends exhibited by two different probesets. At its simplest, we can compare two probesets with corresponding rows in the expression table `gIndex` and `gIndex2`:

```
> gIndex2 <- which(featureNames(normdata)=="201667_at")
> cor(normexprs[gIndex,], normexprs[gIndex2,])
[1] 0.9903677
```

The `cor` function returns the Pearson correlation coefficient between two vectors of numbers. This gives an indication of how ‘similar’ the profiles are across the samples of the dataset, in a scale from 0 (no correlation) to 1 (perfectly correlated) or -1 (perfectly anti-correlated). How to interpret the actual value of Pearson correlation really depends on the context. In some instances you would expect a reasonable correlation to manifest with a Pearson correlation coefficient of 0.4, and in other circumstances anything below 0.99 would imply a fair degree of dissimilarity. Therefore, the interpretation of the correlation coefficient depends on what you *expect*. This can be influenced by the expected level of heterogeneity in the data, the acceptable experimental or technical variation in the values, and the number of data points. This last issue can be incorporated into a test for significance of correlation, assuming both both vectors of numbers are normally-distributed. In R this is implemented in the `cor.test` function:

```
> cor.test(normexprs[gIndex,], normexprs[gIndex2,])
```

This provides both the correlation coefficient and a statistical significance estimate p-value. It is important to recall that this is based on an underlying assumption that the two sets of values are independent and normally-distributed. It is also important to remember that this p-value indicates the

statistical significance only: even if it would be highly unlikely to see such a correlation between two random normally-distributed variables of the appropriate size, it doesn't necessarily mean that the sets of values are actually well correlated. Again, the true significance of the level of correlation depends on what degree of correlation you might expect.

Returning to a theme that we visited earlier, it is always useful to explore a statistical analysis with visualisations of the underlying data. To investigate the expression patterns giving rise to the high correlation observed between the two probe-sets' profiles, we can repeat the plot from Fig. 8.12, adding in the profile for the second probe-set in a different colour using the `points` function.

```
> plot(normexprs[gIndex,], type="l")  
> points(normexprs[gIndex2,], type="l", col="green")
```

You should be able to see that this plot is of limited use: the Pearson correlation coefficient doesn't measure the similarity between sets of values, it

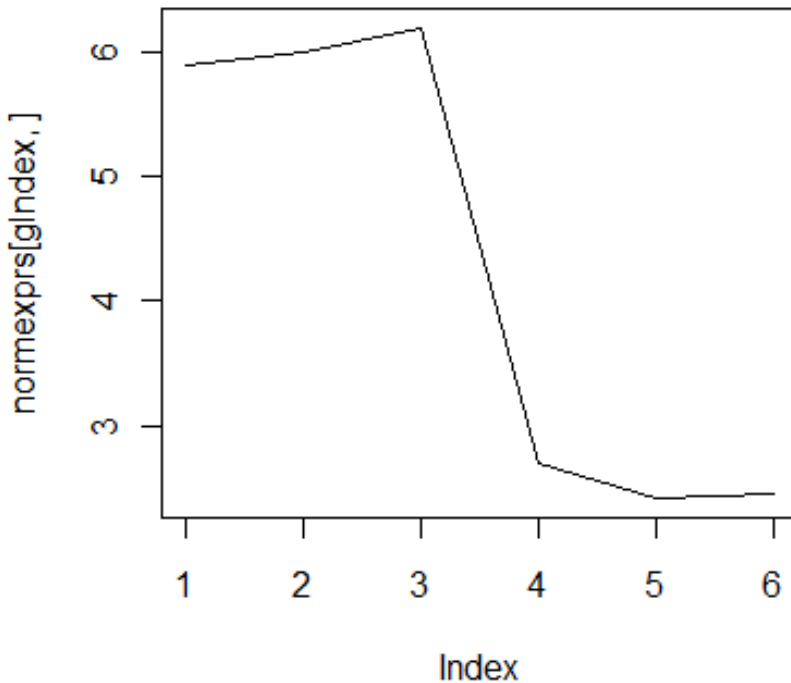


FIGURE 8.12

Profile plot showing expression of probe-set 244829_at across the side-population experiment.

measures the similarity of the trend across each set of values. The expression values for the 2nd probe-set are on a different scale to those of the 1st probe-set. Therefore for an informative plot, we need to rescale the y-axis so as to include all values from both probe-sets:

```
> ymin <- min(normexprs[c(gIndex,gIndex2),])
> ymax <- max(normexprs[c(gIndex,gIndex2),])
> plot(normexprs[gIndex,],type="l",ylim=c(ymin,ymax))
> points(normexprs[gIndex2,],type="l",col="green")
> legend("topright",legend=c("244829_at","201667_at"),
fill=c("black","green"))
```

Following the addition of the legend, the plot should look like that shown in [Fig. 8.13](#).

One application of correlation methods that can be useful to explore patterns of gene expression is to obtain an ordered list of all the probesets represented

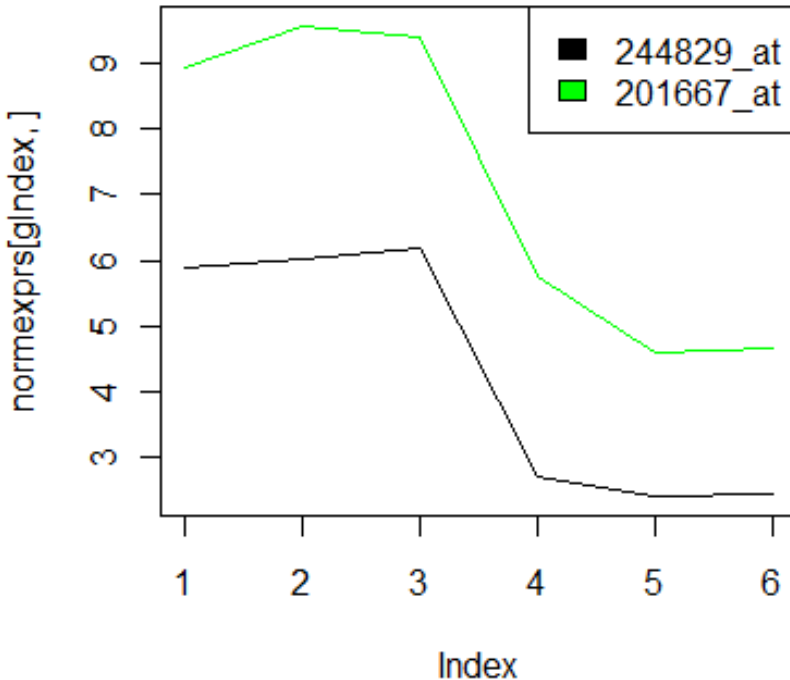


FIGURE 8.13

Plot showing expression profiles of two highly correlated probe-sets across the side-population microarray experiment.

on the array, sorted by their correlation to a gene of interest. For this we can utilise a `for` loop. Let us suppose we wish to find which genes have most highly-correlated expression with the probe “201667_at”, mapping to the gene GJA1, which was the 2nd most significantly differentially-expressed feature shown in the table in Fig. 8.11. First we can create a numeric vector with one element for each probe-set in the dataset:

```
> correlationScores <- rep(NA,nrow(normexprs))
```

Next we construct a *for* loop to calculate the correlations between the values from each probe-set in turn and the values from the probe-set measuring GJA1 expression:

```
> for(i in 1:nrow(normexprs)){
> correlationScores[i] <- cor(normexprs["201667_at",],normexprs[i,])
> }
```

Now we can construct a *data frame* to contain the correlation coefficients and the probe-set annotations:

```
> corTable <- data.frame(probeID=rownames(normexprs),
cor=correlationScores)
```

Given the probeIDs, we can add annotations of the gene symbols to these as in Section 8.7.3:

```
> corTable$symbol <- unlist(mget(corTable$probeID,hgu133plus2SYMBOL))
```

So now we have a data frame called `corTable`, with the probeIDs, Pearson correlation coefficients and gene symbols for each probe-set in the dataset (each row in the `normexprs` matrix). Finally, we sort the table according to the correlation values, then can use the functions `head` and `tail` to inspect the probe-sets most strongly positively correlated and most strongly negatively correlated with the GJA1 probe-set, respectively:

```
> corTable.sorted <- corTable[order(corTable$cor),]
> head(corTable.sorted)
> tail(corTable.sorted)
```

Positive correlation indicates that the shapes of the profiles are the same. Probe-sets exhibiting opposite trends of expression will have highly negative correlation.

If you are interested in both positive and negative correlation, sort on the absolute value (magnitude) of the correlation score:

```
> corTable.sorted2 <- corTable[order(abs(corTable$cor)),]
```

8.9 Clustering

Given the high-dimensionality of microarray datasets (i.e. there are a lot of probe-set measurements for each sample), it can be informative to separate the measurements into groups based on their similarity. This approach is called clustering, and can be performed on both the probe-sets and the samples of the expression dataset. An important consideration with clustering is that the way different entities are grouped together will depend not only on the method of clustering (for which there are many) but also, and possibly more importantly, the definition of the notion of dissimilarity or *distance* between the elements.

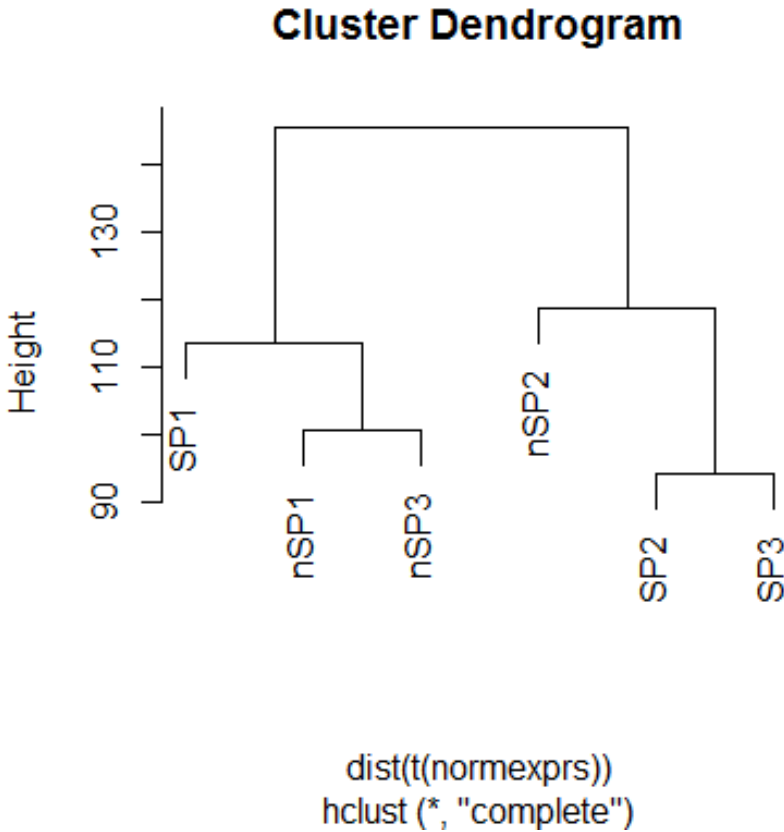
For example, to see a hierarchy or *dendrogram* representing the overall similarities of each chip in the dataset to the others, try:

```
> plot(hclust(dist(t(normexprs))))
```

The above command *transposes* the gene expression data matrix so that the probe-sets are the columns and the samples are the rows, so that the `dist` function will calculate the distances between the samples. As used above, the `dist` function calculates distance between any two rows of the table by adding up the squares of the differences between each column's value for the two rows in question. This has the effect that larger *absolute* differences in value will contribute more to the overall measure of dissimilarity between each pair of samples: probe-sets with a wider range of expression value will typically have greater influence on the resulting clustering than probe-sets with a lower range of expression. This is not necessarily a bad thing, in fact it may well be a useful feature, it just depends on the application! As this first command may well be illegible due to the long names of the samples (which were created from their raw data file names), you can specify the labels for each sample:

```
> plot(hclust(dist(t(normexprs))),  
+ labels=c("nSP1", "nSP2", "nSP3", "SP1", "SP2", "SP3"))
```

This will give the plot shown in [Fig. 8.14](#), which provides some interesting insights.

**FIGURE 8.14**

Dendrogram from hierarchical clustering of samples from the microarray experiment *GSE25191*.

Firstly, it appears that the side-population sample 1 is more similar to two of the non-side-population samples than it is the other side-populations. Also, the 2nd non-side-population sample appears more similar to 2 of the side-population samples than it is the other non-side-populations. Therefore, we may wish to repeat the earlier analysis while omitting these seemingly ‘mis-placed’ samples, illustrating the role that clustering can have in quality control. However, we may wish to keep all the samples in the analysis, as the differences may reflect a biological heterogeneity that should not be forgotten about in future consideration. Once again, the correct action to take depends on what you actually wish to use the analysis results for, but it is useful to be aware of the issues so that you can make the appropriate decision!

In order to see the overall (*global*) clustering of genes and chips, use the `heatmap()` function (which automatically computes hierarchical clusterings of both rows and columns of the expression table). For better colours (red-green as opposed to red-yellow) I use the `greenred()` function in the package *gplots*⁵. If you have an incredibly powerful computer, it may be possible to draw the appropriate heatmap, but typically the construction of the distance matrix used to make pair-wise comparisons between every possible combination of probe-sets is too big to process (in this case it will include nearly 1.5 billion numbers). Were we to wish to create this heatmap, you would use the commands (but be warned, this may stall):

```
> library(gplots)
> heatmap(normexprs,col=greenred(100),scale="row")
```

So for a simpler illustration of the utility of a heatmap, we can construct one to show the expression values of the two probe-sets we utilised earlier in this section exploring expression profiles, which should look like [Fig. 8.15](#):

```
> library(gplots)
> heatmap(normexprs[c(gIndex,gIndex2),],col=greenred(100),scale="row",
+ labCol=c("nSP1","nSP2","nSP3","SP1","SP2","SP3"))
```

In fact, heatmaps can be a useful means of visualising differential-expression. We can draw the heatmap of the most differentially-expressed probe-sets using:

```
> diffexpIDs <- as.character(topTable(fit2,coef=2)$ID)
> diffexprows <- which(featureNames(normdata) %in% diffexpIDs)
> heatmap(normexprs[diffexprows,],col=greenred(100),scale="row",
+ labCol=c("nSP1","nSP2","nSP3","SP1","SP2","SP3"))
```

This should appear as in [Fig. 8.16](#).

8.9.1 Filtering

As we have seen in the previous example, performing certain types of analysis on the datasets we work with can be computationally infeasible. This is not really surprising, as there's really a staggering amount of information that is being processed. It may be a waste of effort including information which may not be desperately relevant, such as probe-sets that will not impact much upon the clustering because they are expressed at such a low level or show only minimal variation across the samples. What's more, if there is a large number

⁵This is not a Bioconductor package, so you may need to install it using the command `install.packages('gplots')`.

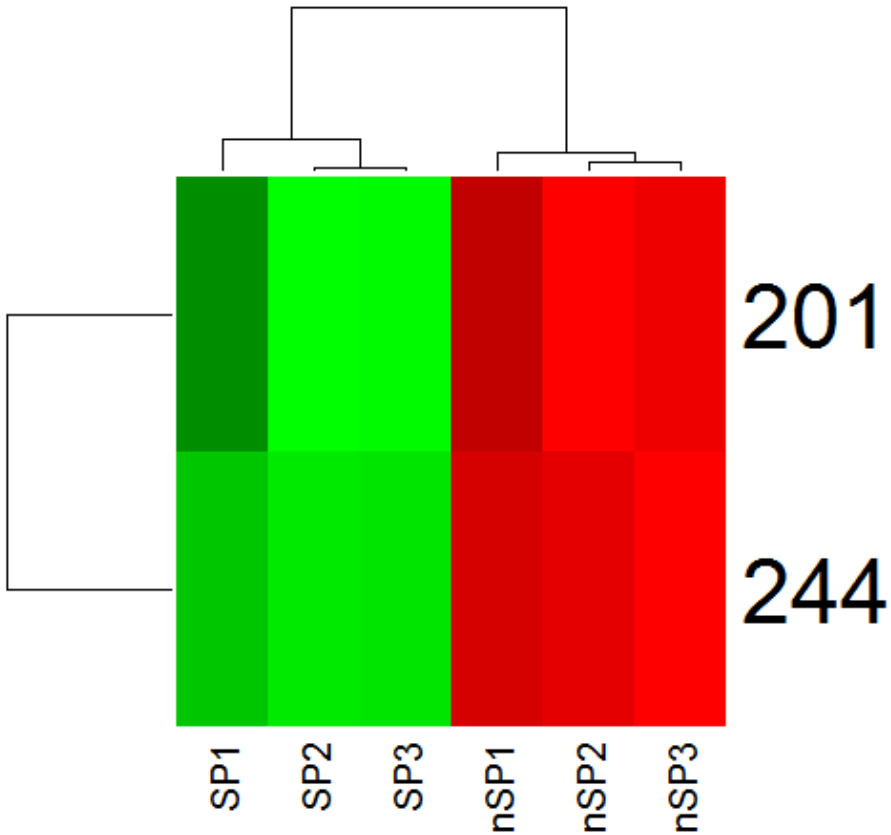
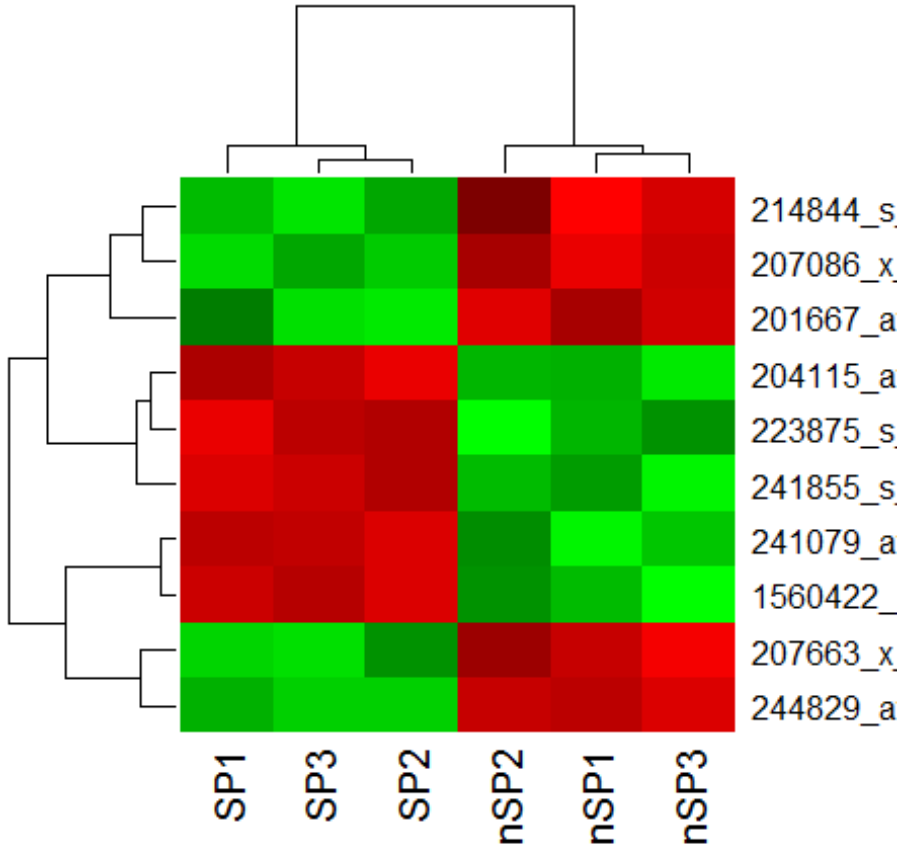


FIGURE 8.15

Heatmap showing per-sample clustering with expression levels from two probe-sets in the microarray experiment *GSE25191*.

of such uninformative probe-sets, they may actually add up to obscure some of the more important similarities and distinctions between the samples of the dataset. There are a number of reasons why it may be beneficial to omit some probe-sets or samples from the analysis:

- Reduce computational requirements
- Remove ‘noise’ from unreliable (low-level) measurements
- Omit samples that do not fit classification required for some analysis
- Increase power to detect significant patterns by reducing the numbers of tests performed, and thus reducing the family-wise error rate

**FIGURE 8.16**

Heatmap showing per-sample clustering with expression levels of the most differentially-expressed probe-sets in the microarray experiment *GSE25191*.

Some filtering requirements may involve context-specific knowledge, such as selecting a subset of the probe-sets on the microarrays to perform analysis on candidate genes only. However, two fairly standard criteria often apply:

1. Remove probe-sets recording only background expression levels
2. Remove probe-sets showing minimal variation across the dataset

For the first criterion, filtering out probe-sets with only low-level expression, the most common approach is to define some essentially arbitrary cut-off and discard all probe-sets with median value across the dataset lower than this threshold. For Affymetrix microarray datasets, such a threshold set to a \log_2 normalized expression value of 5 is fairly common. While this is rather

arbitrary and could involve discarding potentially very interesting probe-sets, such an approach is surprisingly widely used because it is nevertheless an effective way of minimising the contribution of noise to clustering patterns. We could find such probe-set by setting up a *for* loop to calculate the median for each probe-set in turn:

```
> medianexprs <- numeric(nrow(normexprs))
> for(i in 1:nrow(normexprs)){
+ medianexprs[i] <- median(normexprs[i,])}
> lowexprs <- which(medianexprs<5)
```

However, we can also do this more easily by making use of the R function `apply`, which applies a function to every row (specifying `MARGIN=1`) or column (specifying `MARGIN=2`) of a data matrix in turn:

```
> medianexprs <- apply(normexprs,MARGIN=1,median)
> lowexprs <- which(medianexprs<5)
```

For the second filter mentioned above, removing probe-sets with a low range of expression across a dataset, it is fairly standard practice to discard a certain proportion (say, 25%) of the probe-sets on a microarray. Again this is a totally arbitrary way of filtering out probe-sets, and all considerations that applied to the previous case also apply to this approach to filtering. We can perform this filtering by first calculating the standard deviation across the dataset for each probe-set:

```
> exprsds <- apply(normexprs,MARGIN=1,sd)
```

Then by sorting the resulting values, we can find which rows of the data matrix correspond to the lowest variation:

```
> sdorder <- order(exprsds,decreasing=FALSE)
```

And finally we can find out how many we wish to filter out, in this case I have specified 25%:

```
> numberToKeep <- round(nrow(normexprs)*0.25)
> lowsd <- sdorder[c(1:numberToKeep)]
```

Now we can combine the two filters and repeat the previous clustering but with these probe-sets removed from the dataset. The resulting dendrogram should appear as in [Fig. 8.17](#):

```
> plot(hclust(dist(t(normexprs[-c(lowexprs,lowsd),])),
+ labels=c("nSP1", "nSP2", "nSP3", "SP1", "SP2", "SP3")))
```

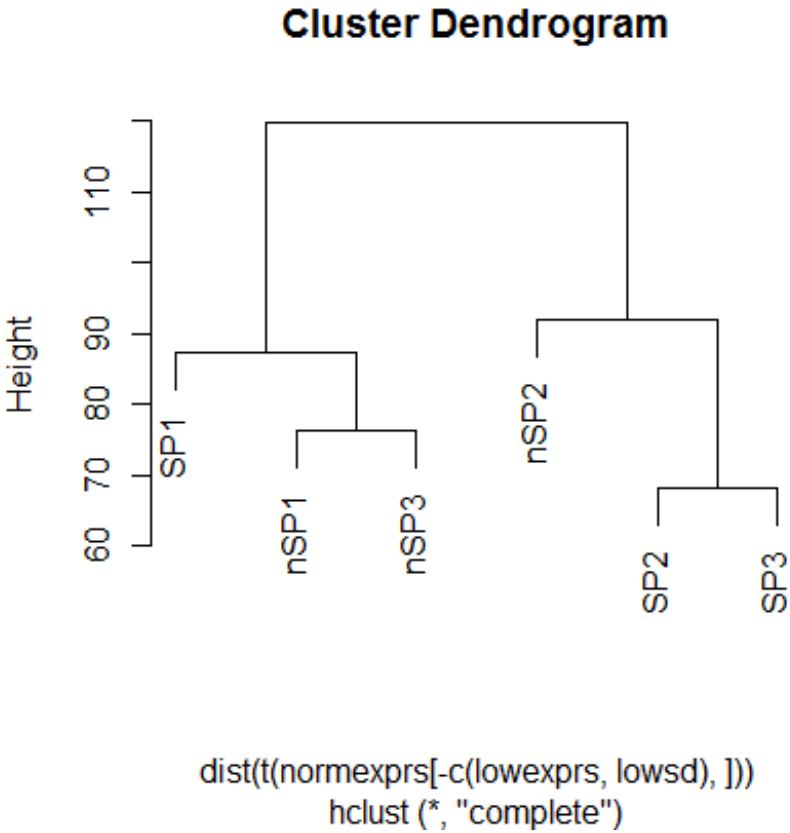


FIGURE 8.17 Hierarchical clustering dendrogram of side-population experiment, having filtered out low-level and low-variation probe-sets.

First we can note that the clustering appears very similar to that which was done previously, indicating that those low-level and low-variation probe-sets weren't significantly contributing to the clustering and that the differences seen come from the more reliable measurements.



8.10 Survival analysis

An introduction to microarray analysis for cancer research would not be complete without mention of survival analysis. To relate gene expression levels

to survival times, or any ‘time to event’ variable which involves incomplete follow-up, we need two pieces of information for each sample: one giving the time til the last follow-up, and one indicating whether at last follow-up the event had occurred or not. In a survival context, the indicator variable states whether the indicated time to last follow-up corresponds to the patient’s death or the patient leaving the study. A different method of analysis is needed for this type of variable, so that information from samples where the event hasn’t occurred (e.g. patient is alive at the end of the study) can be included.

One thing that became apparent when I was putting this together was that it is actually rather difficult to find publically available cancer datasets for which the patients’ clinical characteristics are available in a user-friendly manner! Usually accessing this data requires downloading multiple tables, creating separate spreadsheets and cross-referencing against each other. However, the Gene Expression Omnibus dataset with accession *GSE2034* has the required clinical characteristics in a table, which can be reached from the GEO record on <http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE2034>, and this dataset will provide illustrative examples for analysis that will typically be carried out on in-house datasets. Accessing even this clinical data is not totally straightforward: click on the ‘View full table...’ button towards the bottom of the page, copy all the text that appears and paste into a blank spreadsheet using a spreadsheet program (e.g. Excel, LibreOffice, etc.). Once the text is pasted in, remove all the header rows (those starting with a #), and even the column names. The remaining data should be in a single column: select this column then use the ‘Text to Columns’ feature to split the data across multiple columns using a *space* delimiter. Following these steps, you should have a table like that in [Fig. 8.18](#). Save this table as a tab-delimited text file: to be compatible with the code in the rest of this chapter, name the file “GSE2034clinical.txt”.

We can read this information into R as we would any table. We want to make sure we have the sample identifier (GEO accession number in column B), the time until progression/followup (column D), the numerical indicator of progression (column E) and the ER status (column F). With this information, we can load the raw gene expression microarray data, preprocess it, match the microarrays to clinical subjects, and perform survival analysis (adjusting for ER status and lymph node status if we so wished). First, read in the annotation file:

```
> clin.data <- read.table("GSE2034clinical.txt",sep="\t",head=F)
```

Note that we set `head=F`, as we removed the column names from the table. Let’s add column names back in, using the `colnames` function:

```
> colnames(clin.data) <- c('pID','filename','nodes',
+ 'PFStime','Progression','ERstatus','brainmets')
```

	A	B	C	D	E	F	G	H	I	J	K
1	3	GSM36793	negative	101	0	ER-	0				
2	5	GSM36796	negative	118	0	ER+	0				
3	6	GSM36797	negative	9	1	ER-	0				
4	7	GSM36798	negative	106	0	ER-	0				
5	8	GSM36800	negative	37	1	ER-	0				
6	9	GSM36801	negative	125	0	ER+	0				
7	11	GSM36834	negative	109	0	ER+	0				
8	14	GSM36835	negative	14	1	ER-	0				
9	15	GSM36836	negative	99	0	ER+	0				
10	17	GSM36837	negative	137	0	ER+	0				
11	18	GSM36838	negative	34	1	ER+	0				
12	19	GSM36839	negative	32	1	ER+	0				
13	20	GSM36855	negative	128	0	ER-	0				
14	21	GSM36858	negative	14	1	ER+	0				
15	22	GSM36859	negative	130	0	ER+	0				
16	27	GSM36860	negative	30	1	ER+	0				
17	28	GSM36861	negative	155	0	ER+	0				
18	29	GSM36862	negative	25	1	ER-	0				
19	31	GSM36870	negative	30	1	ER+	0				

FIGURE 8.18

Table of clinical characteristics for GEO dataset *GSE2034*, following removal of headers and splitting of single data column using a space as a delimiter.

Now we can use the GEO accession numbers to specify which CEL files to read in and apply RMA normalization. But we need to use the `paste` function to add `‘.CEL.gz’` onto the end of each GEO accession number in order to specify the gzipped CEL files. Of course, we also need to have downloaded and extracted the `tar` archive of raw data from the GEO web page (as in Section 2).

```
> clinexprs <- justRMA(filenamees=
+ paste(clin.data$filename, ".CEL.gz", sep=""))
```

See how we’ve used the `paste` function to add `‘.CEL.gz’` onto the end of each GEO accession number? This will result in a vector of file names being passed to the `justRMA` function, telling it which files to read and preprocess.

Now the expression data is loaded and preprocessed, it would be useful to attach gene symbols to the probe-set identifiers. We previously added annotation information to a model fit object through the `annotate` package in section 7.3 of this chapter, but an alternative is to add the gene information to the whole dataset. This involves a similar process to that used earlier:

```
> library(annotate)
> library(hgu133a.db)
> ID <- featureNames(clinexprs)
```

```
> Symbol <- getSYMBOL(ID,"hgu133a.db")
> fData(clinexprs) <- data.frame(ID=ID,Symbol=Symbol)
```

In the above commands we have loaded the appropriate annotation packages, created a vector `ID` listing the probe-set identifiers from our dataset, created a vector `Symbol` listing the Gene Symbols matching each probe-set identifier, then annotated the `clinexprs` dataset with a *data frame* listing both probe-set identifier and gene symbol. This uses the function `fData` to access the feature annotation information associated with the `clinexprs` dataset, and will be useful when we wish to look up the genes later.

Finally, in order to work with the dataset we need to create a table of expression values from the object that was created by `justRMA`, which we do using the `exprs` function:

```
> normexprs2 <- exprs(clinexprs)
```

To analyse survival data in R, we need to load the *survival* package. This contains the functions needed:

```
> library(survival)
```

An underlying object used by the survival package is created with the `Surv` function, associating the numeric variable giving the time to event with the numeric indicator variable:

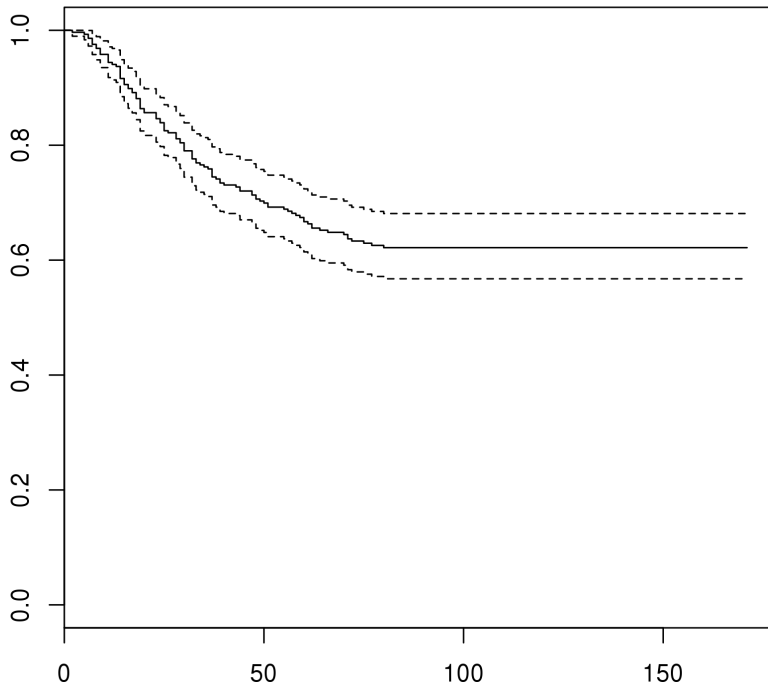
```
> clinSurv <- Surv(clin.data$PFStime,
+ clin.data$Progression)
```

8.10.1 Kaplan-Meier plots

A helpful way to investigate survival data involves producing the familiar Kaplan-Meier plots, which show how the probability of a patient from the given population surviving beyond a particular point decreases as the time goes on. In R, a Kaplan-Meier plot is based around a model fit that is created using the `survfit` function. For example, we can create a simple Kaplan-Meier plot for the whole patient cohort by relating the survival times to a constant (in this case 0):

```
> plot(survfit(clinSurv~0))
```

This will result in the plot shown in [Fig. 8.19](#), which has for its y-axis the proportion of the population that haven't progressed and its x-axis the time in number of weeks. This plot includes 95% confidence intervals, which while

**FIGURE 8.19**

Kaplan-Meier plot showing probability of patient recurrence in relation to time in number of weeks, for the whole of the cohort profiled in experiment *GSE2034*.

useful can clutter the plot, so we can repeat the plot without the confidence intervals as follows:

```
> plot(survfit(clinsurv~0),conf.int=FALSE)
```

While this utility is neat, it is the ability to explore *differences* in survival that is typically of more interest. The only modification we need to the previous command is on the right hand side of the tilde \sim . We need to specify what separates the different cohorts, using a *factor*. For example, we can use the ER status included in the clinical data targets file to see if disease progression is related to ER status:

```
> survfit(clinsurv~clin.data$ERstatus)
Call: survfit(formula = clinsurv ~ clin.data$ERstatus)
records n.max n.start events median 0.95LCL 0.95UCL
clin.data$ERstatus=ER- 77 77 77 27 NA NA NA
clin.data$ERstatus=ER+ 209 209 209 80 NA NA NA
```


The previous command gives a summary of the median survival time, with confidence intervals. However in the example given, not enough events occur to be able to estimate a median survival time (as less than half of either population have progressed by the end of the follow-up). We can test the statistical significance of the difference between the two populations by performing a *logrank* test, using the `survdif` function in the same way as we used the `survfit` function before:

```
> survdiff(clinsurv~clin.data$ERstatus).
```

And the `survfit` function can be used to produce a comparative Kaplan-Meier plot with both curves:

```
plot(survfit(clinsurv~clin.data$ERstatus),lty=c(1,2))
```

The `lty=c(1,2)` argument to the plotting function tells R that you wish to use a dotted line for the second value of the factor (in this case ER+), and gives the plot shown in [Fig. 8.20](#).

While separating the patients according to clinical characteristics may be of some interest, we have not yet used the gene expression data at all. As the above methods require categorical variables, the incorporation of gene expression data is dependent on discretization of the continuous numerical expression values. This will always involve introducing some essentially arbitrary cut-off to separate the samples, and is most commonly done using the median expression value so as to ensure an even split of the samples between the two groups. To repeat the Kaplan-Meier plot showing survival according to the level of expression of the first probe-set on the microarray, enter:

```
> plot(survfit(clinsurv ~ as.numeric(normexprs2[1,]  
>median(normexprs2[1,]))))
```

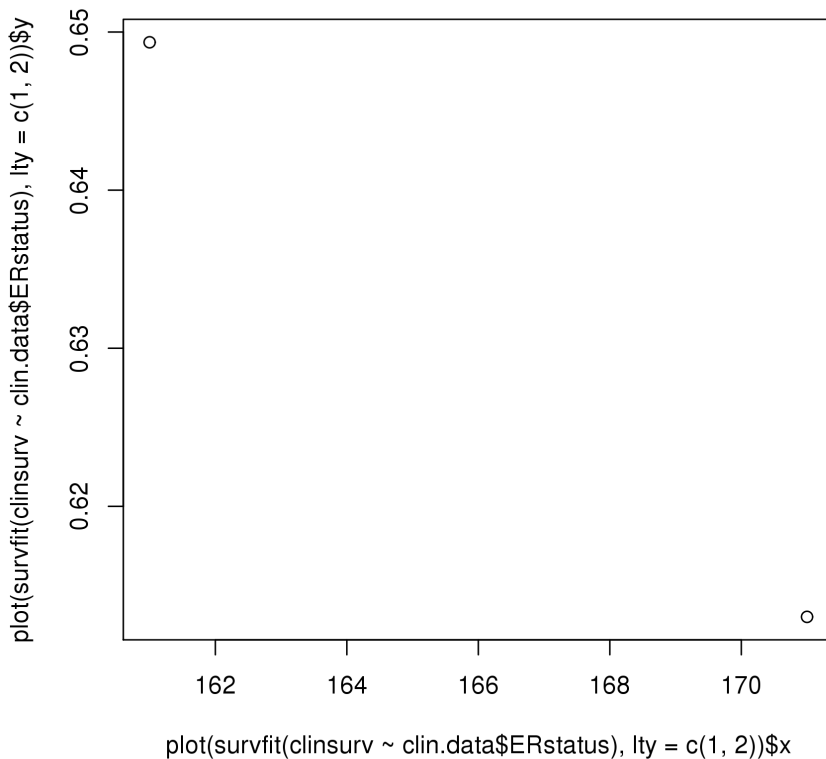
The above command creates a numerical indicator for whether each value in the first row of the normalized gene expression dataset `normexprs2` is greater than the median or not. This is a perfectly reasonable way to use the appropriate R functions, but it is perhaps easier to see what we are doing when the process is broken down into multiple steps:

First we select only the row of the expression dataset that corresponds to our gene of interest:

```
> exprow <- normexprs2[1,]
```

Next we find the median expression value for the chosen probe-set:

```
> expmedian <- median(exprow)
```

**FIGURE 8.20**

Kaplan-Meier plot showing the difference in survival curves for the two patient cohorts separated by ER status.

Next we create an indicator stating whether or not each expression value from the chosen probe-set is above its median:

```
> expIndicator <- as.numeric(exprow>expmedian)
```

Now you can create a *survfit* object that contains the information necessary for the figure:

```
> expFit <- survfit(clinsurv ~ expIndicator)
```

And finally you can make a plot using the object just created:

```
> plot(expFit)
```

The same principles used for plotting Kaplan-Meier curves in R can be used to test the significance of the difference in survival between the patients with above-median expression measurements from the first probe-set on the microarray and the patients with below-median expression measurements:

```
> survdiff(clinsurv ~ as.numeric(normexprs2[1,]
>median(normexprs2[1,])))
```

As a more useful example, say we wished to investigate the association between expression of STARD5 and disease progression. Then we could first find which of the rows of the dataset correspond to the gene of interest:

```
> STARD5row <- which(fData(clinexprs)$Symbol=="STARD5")
```

Because a single equals sign = in R can be used as an assignment operator, similar to <-, if you wish to test for equality between two objects you must use a double equals sign ==. This test for equality only works when comparing two single objects of the same basic data type.

Now we know which row we want to inspect, we can continue as before to evaluate the significance of the difference in disease progression times between patients with above-median expression of the gene and those with below-median expression of the gene:

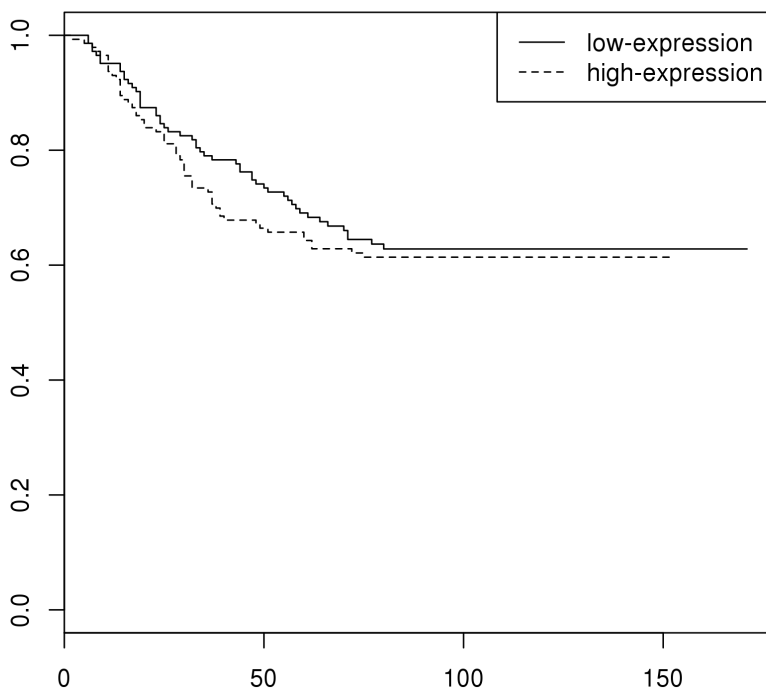
```
> survdiff(clinsurv ~ as.numeric(normexprs2[STARD5row,]
>median(normexprs2[STARD5row,])))
```

Similarly, the following code will create a Kaplan-Meier plot characterising the difference in disease progression times between patients with above-median expression of STARD5 vs those without:

```
> plot(survfit(clinsurv ~ as.numeric(normexprs2[STARD5row,]
+ median(normexprs2[STARD5row,]))) , lty=c(1,2))
```

The above use of the `plot` function has been split over two lines (which introduces the R prompt `+` to indicate that the command is incomplete⁶) solely for display purposes, and again has the argument to specify `lty=c(1,2)`, which tells R to draw the first data series as a solid line and the second data series as a dashed line. **NB:** the plus symbol `+` above is a prompt from R, it is not part of the command. Therefore, if you are copying this command into the R console in a single line, you do not enter the plus symbol.

⁶R can tell a command is incomplete if there is an open bracket that doesn't have a matched closing bracket.

**FIGURE 8.21**

Kaplan-Meier plot showing time-to-progression of patients partitioned into two equally-sized groups based on expression of STARD5.

The two data series correspond to the survival curve for patients with below-median expression of STARD5 and the survival curve for patients with above-median expression of STARD5, respectively. To show this on the plot we can construct a legend:

```
> legend("topright", legend=c("low-expression", "high-expression"),  
lty=c(1,2))
```

This command using the `legend` function first specifies where on the plot to draw the legend, then the text to describe the different elements, and finally the way to draw the symbol for each element in the legend. Following this command, the plot should look like [Fig. 8.21](#).

8.10.2 Cox proportional hazards regression

While Kaplan-Meier plots and logrank tests are useful to investigate differences in survival that correspond to differences in gene expression or clinical

variables, being restricted to referring to gene expression as a categorical variable limits the possible analysis and makes it dependent on arbitrary thresholds. A more quantitative approach involves the use of *proportional hazards regression* models. The mechanism of fitting so-called *cox* models for survival analysis in R is similar to that used for producing Kaplan-Meier plots. To fit a cox proportional hazards regression model relating time to progression to the quantitative measurement of expression from the first probe-set on the microarray:

```
> coxph(clinsurv ~ normexprs2[1,])
```

Or to assess the quantitative association between STARD5 expression and time to progression:

```
> coxph(clinsurv ~ normexprs2[STARD5row,])
```

One of the strengths of the *limma* package is being able to fit and evaluate linear models for tens of thousands of probesets in one command. Unfortunately it doesn't implement survival models, so we have to evaluate these one probe-set at a time. However, we can utilise the following simple *for*-loop to create a list in R where each element is a cox model fit relating the time to progression to the expression measurement from the corresponding probe-set on the microarray:

```
> phmodels <- list()
```

This command creates an empty list to start with.

```
> for(i in 1:nrow(normexprs2)){
```

This sets up a loop, creating a variable *i* that at each step through the loop takes the next value in a sequence from 1 to the number of rows in the normalized gene expression dataset.

```
+ phmodels[[i]] <- coxph(clinsurv ~ normexprs2[i,])}
```

This creates a new element in the list *phmodels* at each step through the loop, setting its value to the output of the *coxph* function. The curly bracket *{* closes the loop.

We can inspect the Cox proportional hazards regression model fit for any probe-set in the dataset by indexing the corresponding element from the list. For example, if we wished to see the model corresponding to STARD5 expression:

```
> phmodels[[STARD5row]]
```

Again, this is all very well, but we don't have a real indication of which genes have a significant expression association with disease progression. For this, we can use the Cox model fit objects in the list `phmodels` and a *for* loop in a similar manner to the one used above:

```
> phmodels.pvals <- numeric(length(phmodels))
```

First we create a vector of the same length as the `phmodels` object.

```
> for(i in 1:length(phmodels)){
```

This sets up the loop over variable `i` from 1 to the length of the `phmodels` object.

```
+ phmodels.pvals[i] <- summary(phmodels[[i]])$coefficients[5]}
```

The above command explains that to each Cox model fit object in turn, first apply the `summary` function to the Cox model fit object to access various aspects of statistical information regarding the model, then select the `coefficients` element of the result, and finally take only the 5th column of the resulting table, which is the p-value associated with the model term... far from being particularly user-friendly! However, the reason for needing all this extra is the fact that there is an awful lot more information provided about each model fit, and it is worth remembering that it is not only the p-value that matters.

So the above tells us the p-value associated with the proportional hazards regression model, but it doesn't indicate whether higher expression of the gene is associated with better or worse prognosis! For this, we need to find the *hazard ratio* or the coefficient value (β in the proportional hazards regression model). We can find these values in a very similar way to the approach applied above to obtain the p-values. For example, to get the model coefficient:

```
> phmodels.coefs <- numeric(length(phmodels))
> for(i in 1:length(phmodels)){
+ phmodels.coefs[i] <- summary(phmodels[[i]])$coefficients[1]}
```

The only difference to the commands used to obtain the p-values is in the final step it is the 1st element of the `coefficients` element of the model summary we wish to retain. Now we can see for any probe-set if the corresponding model coefficient is greater than zero then higher expression corresponds with an increased risk of disease progression, but if the model coefficient is less than zero then higher expression corresponds with a decreased risk of disease progression.

As the final part of this analysis, we may wish to find *which* of the genes have a significant expression association with disease progression. There is an inbuilt R function `order` that can be used to obtain the ordering of the p-values:

```
> survorder <- order(phmodels.pvals,decreasing=FALSE)
```

The above command uses the fact that the `sort` function can either return the sorted values or the ordering that indicates *which* value comes at each ranked position when sorted.

We can find the probe-set identifiers corresponding to each gene simply by using the `survorder` object that we have created as an index to the list of all probe-sets on the microarrays:

```
> survIDs <- featureNames(clinexprs)[survorder]
```

Similarly, we can get the corresponding gene symbols:

```
> survGenes <- fData(clinexprs)$Symbol[survorder]
```

Finally, I would advise using the `p.adjust` function to take into account the fact that we've performed a large number of statistical tests and so the *family-wise error rate* is important in hypothesis testing! multiple hypothesis testing. This can be done simply using:

```
> coxpvals.adj <- p.adjust(phmodels.pvals,method="fdr")
```

You can inspect the probesets with expression measurements most significantly associated with a difference in disease progression, and their corresponding p-values, by constructing an array with the `cbind` function and then re-ordering according to the sort that was performed earlier:

```
> survtable <- cbind(fData(clinexprs),phmodels.coefs,phmodels.pvals,
+ coxpvals.adj)[survorder,]
```

It may also be helpful to specify the column names for this table:

```
colnames(survtable) <- c("ID","Symbol","coef","P.Value","adj.p.val")
```

Now entering `head(survtable)` will print out the top few rows of the table to the display in the R console. The table `survtable` can be written out to file as any array object.

Note: all survival analyses illustrated in this section correspond to *univariate* analysis: we are only evaluating the relationship between gene expression (or a

single clinical variable) and the survival variable (time to disease progression), ignoring all other influences. In many situations we would expect certain other clinical characteristics (these are sometimes described as *confounding variables* and may include age, tumour stage etc.) to have a considerable impact upon some survival variable, and *multivariate* models can be constructed to evaluate the association between the survival variable to your variable of interest (say, expression of a candidate gene) whilst simultaneously taking into account the effects due to the confounding clinical variables. Multivariate survival analysis is not altogether different from the univariate analyses described in this section, but they will be covered later in an ‘advanced topics’ tutorial.

8.11 Footnote: Correlation to explore associated functions

One way of gaining insight into the functional roles of a given gene in a given context is to obtain a set of genes with correlated expression in the biological context of interest (assuming there is sufficient variation to make this worthwhile), and test the resulting list of genes for over-representation of functional annotations (e.g. pathways) as outlined in [chapter 6](#). For this we can repeat the correlation analysis applied in section 8 of this chapter, but let’s see how this would work for the clinical gene expression dataset. Suppose we wish to find which genes have most highly-correlated expression with STARD5 across this whole cohort of breast tumours. First we create a vector of missing values, with one element for each probe-set in the dataset:

```
> correlationScores <- rep(NA,nrow(normexprs2))
```

Next we construct a *for* loop to calculate the correlations between the values from each probe-set in turn and the values from the probe-set measuring STARD5 expression:

```
> for(i in 1:nrow(normexprs2)){  
> correlationScores[i] <- cor(normexprs2[STARD5row,],normexprs2[i,])  
> }
```

Now we can construct a table (a *data frame*) to contain the correlation coefficients and the probe-set annotations, by using the `cbind` function to append the correlation coefficients to the data frame of feature annotations associated with the `clinexprs` object, which we access with the `fData` function:

```
> corTable <- cbind(fData(clinexprs),correlationScores)
```


Finally, we can sort the table according to the correlation values, then use the functions `head` and `tail` to inspect the probe-sets most strongly positively correlated and most strongly negatively correlated with the STARD5 probe-set, respectively:

```
> corTable.sorted <- corTable[order(correlationScores,decreasing=T),]  
> head(corTable.sorted)  
> tail(corTable.sorted)
```

Positive correlation indicates that the shapes of the profiles are the same. Probe-sets exhibiting opposite trends of expression will have highly negative correlation.

We may then wish to test for functional enrichment among the highly-correlated genes, or to look for other possible drivers of the observed association with survival.

9

Analyzing DNA Methylation Microarray Data in R

9.1 Introduction

The addition of methyl groups to cytosine residues in DNA forms part of the mechanisms by which cells can regulate their patterns of gene expression. As such, the profile of which CpG loci (so-called because it is only the CG dinucleotides that can become methylated) are methylated in a set of cells has become a well-studied characteristic of biological variation. Such variation in DNA methylation profiles has been shown to be a key molecular driver of some cancers, and underpins cellular specification in mammalian development. With interest in DNA methylation profiles has come technology to profile the levels of multiple CpG loci simultaneously, in an attempt to build up such profiles. One widely-used approach is based on bisulphite conversion and microarrays, where sodium bisulphite converts unmethylated Cs to Us but affects no change on methylated Cs, then microarray probes can distinguish between targeted sequences with a C in the query CG site or those with a T. While there have been a range of platforms for this purpose, Illumina's '450K' array (and the subsequent 'EPIC' array) is sufficiently dominant that we can focus on this platform.

This tutorial assumes familiarity with R and the analytical methods described in previous chapters of this set. It will walk through the process of importing Illumina 450K data files into R using the package *minfi*[1], carrying out quality control and normalization, ultimately resulting in representations of the methylation profiles which can be analyzed using any of the techniques applied in previous chapters. The *minfi* package is obtained through R as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("minfi")
> biocLite("IlluminaHumanMethylation450Kmanifest")
```

This second package we are installing is required by the *minfi* package to perform some of its functions.

9.2 Importing raw data

The raw data format from Illumina methylation microarrays is the *IDAT* file. These can be imported into R using the *minfi* package. As an example to illustrate how simple this can be, let's download some data to try it out on... There is DNA methylation data available in GEO just as there is gene expression data. We can obtain data from a study with accession GSE69118¹, which contains the data used to show that ER-regulated enhancers show differential methylation in endocrine therapy resistant cell lines relative to drug sensitive parent cell lines [2]. The raw data can be downloaded by clicking on the link illustrated by the arrow in Fig. 9.1.

Having downloaded the TAR archive, you will need to extract the raw data files. There will be two IDAT files for each sample, this is because the microarrays use two dyes: one to quantify the 'methylated' probes and one to quantify 'unmethylated' probes. Let's use only the two replicates of the parental line MCF7 and the two replicates of the Tamoxifen resistant derivative TamR (these have accessions 'GSM1693089', 'GSM1693090', 'GSM1693091' and 'GSM1693092'). Start R in the 'GSE69118' directory containing the IDAT files, load the *minfi* package and then read in the IDAT files:

```
> library(minfi)
> basenames <- unique(substr(list.files(),start=1,stop=28))
```

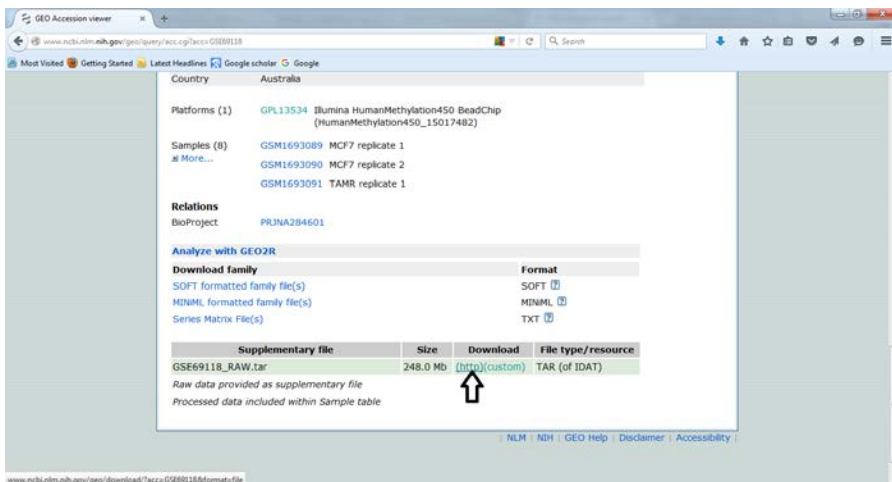


FIGURE 9.1

Gene Expression Omnibus record for experiment GSE69118. Click on link shown with arrow to download tar archive of raw data.

¹<http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE69118>

minfi expects a list of one identifier per sample, which it will use to look for files ending in ‘_Grn.idat’ and ‘_Red.idat.’ This command lists the files in the directory, takes the first 28 characters of them (using the `substr` function), then keeps only one instance of each identifier. This should look as follows:

```
> basenames
[1] "GSM1693089_8769527090_R01C01" "GSM1693090_8769527090_R06C01"
[3] "GSM1693091_8769527090_R03C01" "GSM1693092_8769527090_R02C02"
> RGSet <- read.metharray(basenames)
```

This is the part which actually reads in the data, creating an ‘RGset’ object with four samples. This is the structure that *minfi* uses to manage the raw data, and we will want to use some of the package’s features to the data before we can perform any meaningful analysis.

9.3 Quality control

The first step in processing data is to make sure that there aren’t any obvious signs that there are problems (remembering the old adage ‘rubbish in, rubbish out’). *minfi* has inbuilt functionality precisely designed for this purpose with Illumina 450K methylation arrays. You can use the `qcReport` function to create a PDF report outside R, for example:

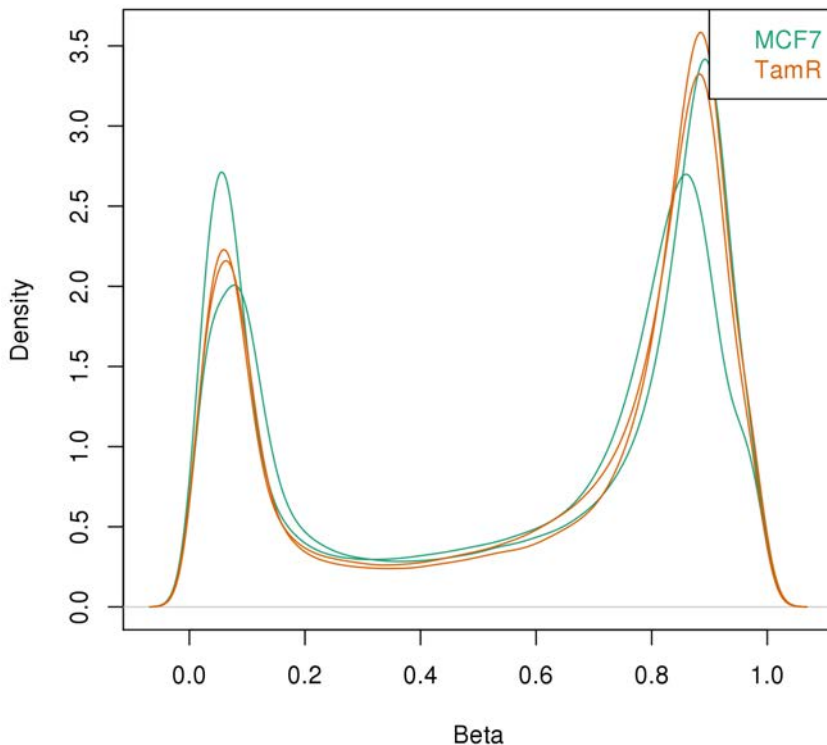
```
> qcReport(RGSet, sampNames=c("MCF7.rep1", "MCF7.rep2", "TamR.rep1",
+ "TamR.rep2"), sampGroups=rep(c("MCF7", "TamR"), each=2),
pdf="GSE69118QC.pdf")
```

Here we have specified the names of the samples.

I tend to find it more convenient to generate certain QC figures within R. The first of these is a density plot, showing the distribution of methylation ‘beta’ values from each sample:

```
> densityPlot(RGSet, sampGroups=c("MCF7", "MCF7", "TamR", "TamR"))
```

Here we have specified the fact that each set of replicates comes from the same sample group. This should produce the plot shown in [Fig. 9.2](#). You can see that most values are either less than 0.2 or greater than 0.8, which reflects the fact that most CpG sites in the genome are pretty consistently methylated or unmethylated across each sample.

**FIGURE 9.2**

Density plot showing the distribution of values from each sample in a study. The two groups of replicates have been labelled with separate colours.

The next plot it can be important to check is the `controlStripPlot`. This compares the intensity values of different control probes on the array. There are sets of different control probes², but the default ones are the bisulphite conversion controls, which will be plotted with the following command:

```
controlStripPlot(RGSet, sampNames=c("MCF7.rep1", "MCF7.rep2",  
+ "TamR.rep1", "TamR.rep2"))
```

The results of this can be seen in Figure 9.3, and should not differ too much from one sample to the next. A sample with outliers should be treated with caution, if not left out of analysis completely.

²See the manual at <http://bioconductor.org/packages/release/bioc/html/minfi.html> for details.

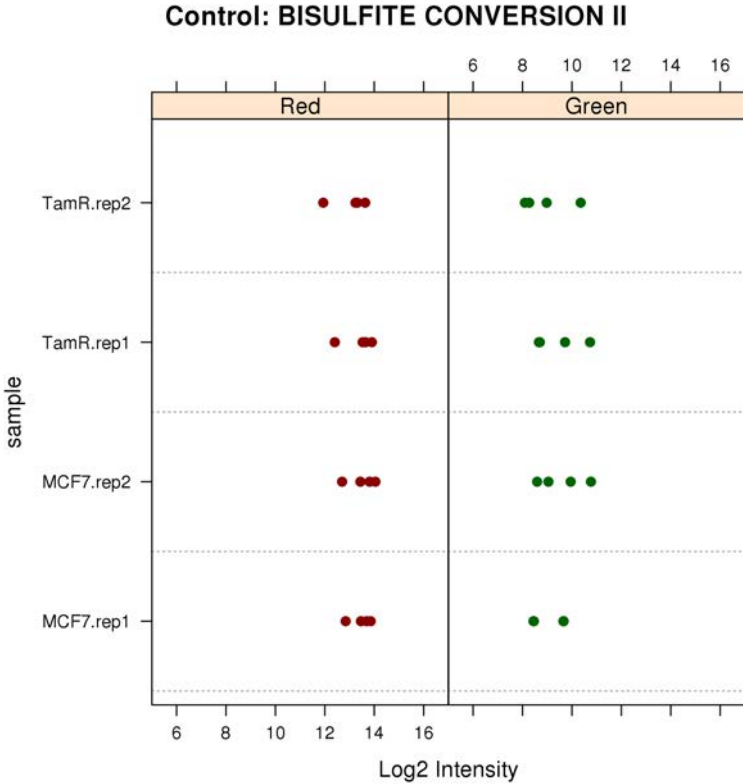


FIGURE 9.3

Strip plot showing the values from the bisulphite conversion control probes for each sample in a study.

9.4 Normalization and estimating methylation level

Having checked that there is nothing untoward in the data for each sample, it is usually pertinent to normalize the dataset, which will attempt to remove technical biases resulting in variation in the methylation values from one sample to the next. If we don't wish to perform any normalization, we can obtain methylation estimates for each probe in each sample:

```
> MSet <- preprocessRaw(RGSet)
> betaVals <- getBeta(MSet,type="Illumina")
```

The methylation values are termed 'beta' values because the density plots of these values are characterized by a β distribution.

Assuming that we want to remove the background signal from the arrays, which is almost ubiquitous, we carry out the following instead:

```
> MSet.bgnorm <- preprocessIllumina(RGSet,bg.correct=TRUE,
normalize="no")
```

NB: this may take a long time! Following this step, running the `getBeta` function will give beta values that have been background-corrected.

Another normalization that is sometimes applied to large datasets is *quantile normalization*. This ensures that each column of a data matrix follows the same distribution, under the assumption that overall total signal should be similar. This is almost always appropriate for gene expression studies, where the same total amount of RNA is used for each sample, but is not necessarily the case with DNA methylation: even in the same amount of DNA, some samples may have more or less methylation than others. However, if we don't expect this to be the case for predominantly biological reasons, then we can perform quantile normalization. There is a function called `normalize.quantiles` in the package *preprocessCore*³ which will perform generic quantile normalization on a data matrix, but there is one further peculiarity of the 450K arrays that means it is well to use a bespoke method. That is, the arrays actually contain two different types of probes, which will have different distributions, and so should be quantile-normalized separately. Helpfully, the *minfi* package makes it easy for us to apply a quantile normalization approach that takes this into account:

```
> MSet.qnorm <- preprocessSWAN(RGSet,MSet.bgnorm)
```

Note that this has used both the `RGSet` and the background-normalized `MSet` object we created previously. So our final processed methylation beta values would be generated:

```
> betaVals.final <- getBeta(MSet.qnorm)
```

9.5 Analyzing beta values

Many statistical methods assume that observed data reflect samples from normally-distributed random variables. Especially when the sample sizes are small, or when each entity being measured may have very different distributions, this is often a *useful* if not altogether *accurate* assumption. However, the measurements from quantitative DNA methylation assays (such as

³available from Bioconductor

quantitative bisulphite sequencing or Illumina microarrays) are known to be beta-distributed. This means that the values are typically sufficiently different from a normal-distribution that transforming beta-distributed values to normally-distributed values can improve the appropriateness of the statistical analysis tools used in this tutorial[3]. A simple calculation can do just this, known as the M-transform:

$$M = \log_2 \left(\frac{\beta}{1 - \beta} \right). \quad (9.1)$$

In fact, when working with Illumina methylation data in *minfi*, we can simply generate M-values from an MSet with the `getM` function. For example, assume we had the object `MSet.qnorm` in the R workspace as produced in the previous section:

```
> MVals <- getM(MSet.qnorm)
```

Then with the M-values, we could apply the empirical Bayes moderated t-statistic linear regression implemented in *limma*, as described in Section 4.7. Nevertheless, I will illustrate the principle here: first creating a design matrix, then fitting the linear models to each probe, and finally extracting the statistics for differentially-methylated probes.

```
> library(limma)
```

We need the *limma* package to proceed, if you don't have this then install from Bioconductor⁴.

```
> design <- cbind(intercept=1,tamR=c(0,0,1,1))
```

Here we construct a matrix called `design` with two columns: the first, called 'intercept', has all values equal to 1; the second, called 'tamR', has value 0 for the first two rows (corresponding to the two MCF7 cell lines) and value 1 for the second two rows (corresponding to the two TamR cell lines). This is a simple way of setting up a two-group comparison.

```
> tamR.fit <- lmFit(MVals,design=design)
```

This command uses the `lmFit` function from the *limma* package, which fits a linear model to every row in the matrix of M-values that we created using the functionality from *minfi*.

```
> tamR.fit <- eBayes(tamR.fit)
```

Here we generate moderated t-statistics, taking into account the fact that we have large numbers of measurements from a small number of samples, in an

⁴> `biocLite("limma")`

attempt to get better estimates of the variability in each probe's measurements we would have seen if we had more replicates.

```
> topTable(tamR.fit,coef=2)
```

This creates a table of the most significantly differentially-methylated probes, specifying that we are only interested in the second column of the design matrix, which will characterize the differences between the two MCF7 replicates and the two TamR replicates. In order to return the full table of all differentially-methylated probes (with adjusted $p < 0.05$), use the following slight alteration:

```
> fulldiffmeth <- topTable(tamR.fit,coef=2,number=nrow(MVals),
p.value=0.05)
```

A list of Illumina probe identifiers is all well and good, but what do these actually represent? Where in the genome are they situated? As with most of the other datasets we have analyzed, we need to find a representation of the data that means something to us. In this case, we can find the genomic co-ordinates of the probes, official symbol of the nearest gene(s), and additional annotation information (such as whether the CpG locus lies within a 'CpG Island', 'shore' etc) in the platform annotation file from GEO. The 450K array has GEO accession GPL13534, and the annotation file can be downloaded directly from:

["http://www.ncbi.nlm.nih.gov/geo/download/?acc=GPL13534&format=file&file=GPL13534_HumanMethylation450_15017482_v.1.1.csv.gz"](http://www.ncbi.nlm.nih.gov/geo/download/?acc=GPL13534&format=file&file=GPL13534_HumanMethylation450_15017482_v.1.1.csv.gz).

Then we can use this file in R, setting the probe identifiers as row names of the resulting data frame:

```
> gpl13534.annot <- read.table("GPL13534_HumanMethylation450_15017482_
v.1.1.csv",sep=" ",skip=7,head=TRUE,row.names=1,fill=TRUE)
```

With the table loaded in, it is actually pretty simple to add the annotation information. For example, to add gene symbols to the table of differentially methylated probes:

```
> fulldiffmeth$Symbol <- as.character(gpl13534.annot[rownames
(fulldiffmeth),
"UCSC_RefGene_Name"])
```

And to add the characterization of whether the CpG site is in the gene body, promoter, or further upstream:

```
> fulldiffmeth$Group <- as.character(gpl13534.annot[rownames
(fulldiffmeth), "UCSC_RefGene_Group"])
```

Now if we inspect the top few probes, we see:

```
> head(fulldiffmeth[,c(1,4,5,7)],n=5)
logFC P.Value adj.P.Val Symbol
cg04455286 9.596053 4.419923e-07 0.008245484
cg26764180 9.356045 4.766909e-07 0.008245484 FAM122C
cg09813276 9.172361 5.461757e-07 0.008245484 ARMCX1
cg18245890 8.294991 9.093927e-07 0.008245484 BCL6
cg00518941 8.760508 9.675355e-07 0.008245484 PRKCE
```

You can carry out any of the other data analysis steps detailed in [Chapter 4](#), or in [Chapter 8](#) (Sections 7 onwards), using either beta or M values as you consider more appropriate. As a final example, we will generate a heatmap to visualize the methylation values of the 100 most significantly differentially-methylated probes. By default, the `heatmap` function in R performs unsupervised hierarchical clustering using a Euclidean distance metric (for more details, see [Section 4.10](#)). This means that if the samples split by group, then the probes we have selected do indeed separate the samples. Convenient plotting colours are accessed through the package *gplots*⁵:

```
> heatmap(betaVals.final[rownames(fulldiffmeth)[1:100],]
,col=bluered(100),ColSideColors=rep(c("cyan","pink"),each=2))
```

This uses `bluered(100)` to create a colour gradient from blue (lowest) to red (highest). We also specify a colour bar to code the samples by their group: cyan for MCF7, pink for TamR.

```
> legend("topleft",legend=c("MCF7","TamR"),fill=c("cyan","pink"))
```

This has added a legend to the plot, so we know what the colour code means. The result should be as seen in [Fig. 9.4](#).

9.6 Using previously preprocessed data

For another example, to illustrate the fact that we are not restricted to the use of Illumina 450K datasets, we will use a publicly available Illumina Human-Methylation27 BeadChip array dataset, profiling frozen tissue samples from normal ectocervix and squamous cell carcinoma of the cervix. This dataset can be retrieved from GEO with accession GSE36637⁶. The normalised beta-values

⁵Install with: `> install.packages("gplots")`.

⁶<http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE36637>

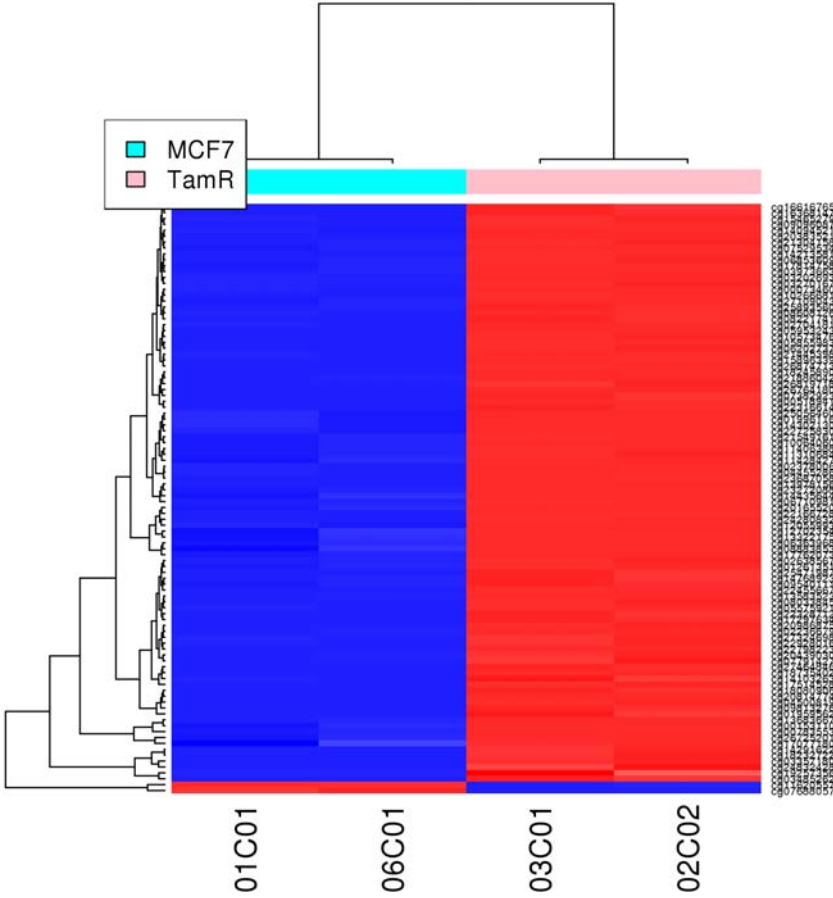


FIGURE 9.4 Heatmap of methylation beta values, for 100 most significantly differentially methylated in comparison between MCF7 and TamR cell lines. As almost all are in the same direction (more methylated in TamR than MCF7) it suggests that perhaps there was some bias in the data which our normalization didn't remove.

can be obtained from the *Series Matrix File* on the GEO webpage for the experiment. After downloading and unzipping this Series Matrix File (which is a gzipped text-file with extension *.txt.gz*), you will need to open the file with a spreadsheet tool and strip out the header information: this takes up the first 69 rows and the very last row of the table. I have saved the resulting file as a tab-delimited text file called "GSE36637_dataonly.txt". As a check that the data has been loaded correctly, take note that the final table has 10 columns (one of which gives the probe identifiers) and 27,579 rows (one of which gives the column headers).

We can load this data table into the R workspace using the `read.table` function, as we have seen before:

```
> meth.table <- read.table("GSE36637_dataonly.txt", sep="\t",
header=TRUE)
> dim(meth.table)
[1] 27578 10
```

We have used the `dim` function, which returns the dimensions of an *array* or data frame, to check that the table does indeed have the dimensions expected (27,578 rows and 10 columns).

As with the siRNA screen dataset used as an example earlier in this tutorial, it will be helpful to create a purely numeric data matrix containing the methylation beta values, with each row labelled by the probe identifier:

```
> beta.values <- as.matrix(meth.table[, -1])
> rownames(beta.values) <- meth.table$ID_REF
```

It is relatively straightforward to use the M-transform to create a table of more normally-distributed *M-values* from the table of beta-values:

```
> m.values <- log(beta.values/(1-beta.values), base=2)
```

Now if we use *limma* to identify probes with statistically significant differences in methylation measurement between the normal ectocervix samples and the cancer samples, we can compare the results from performing the analysis on the beta-values to those from the m-values. First, we need to create the design matrix. Given that the first four columns in the matrix correspond to non-cancerous samples, and the remaining five columns correspond to cancerous samples, our explanatory variable of interest could be numerically encoded as a vector of four 0s then five 1s:

```
> design <- cbind(intercept=1, cancer=c(rep(0,4), rep(1,5)))
```

Here we have created a design matrix specifying an intercept and a variable `cancer` indicating whether or not the sample was cancerous.

```
> beta.fit <- lmFit(beta.values, design)
> beta.fit <- eBayes(beta.fit)
```

Here we have used the design matrix to fit linear models to the table of beta-values.

```
> m.fit <- lmFit(m.values, design)
> m.fit <- eBayes(m.fit)
```

We have fitted linear models as before, but to the table of m-values. So now we can compare the most significantly differentially-methylated probes from each of the two tables:

```
> topTable(beta.fit,coef=2)
ID logFC AveExpr t P.Value adj.P.Val
13187 cg13270853 0.6065 0.6744444 52.64286 3.031232e-11 8.359531e-07
7415 cg07442479 0.4950 0.3700000 27.53366 4.656947e-09 5.197644e-05
20596 cg20645065 0.5930 0.3544444 25.63057 8.101947e-09 5.197644e-05
3603 cg03574571 -0.1800 0.7200000 -24.88095 1.018951e-08 5.197644e-05
23653 cg23694248 0.5240 0.3711111 24.67005 1.088162e-08 5.197644e-05

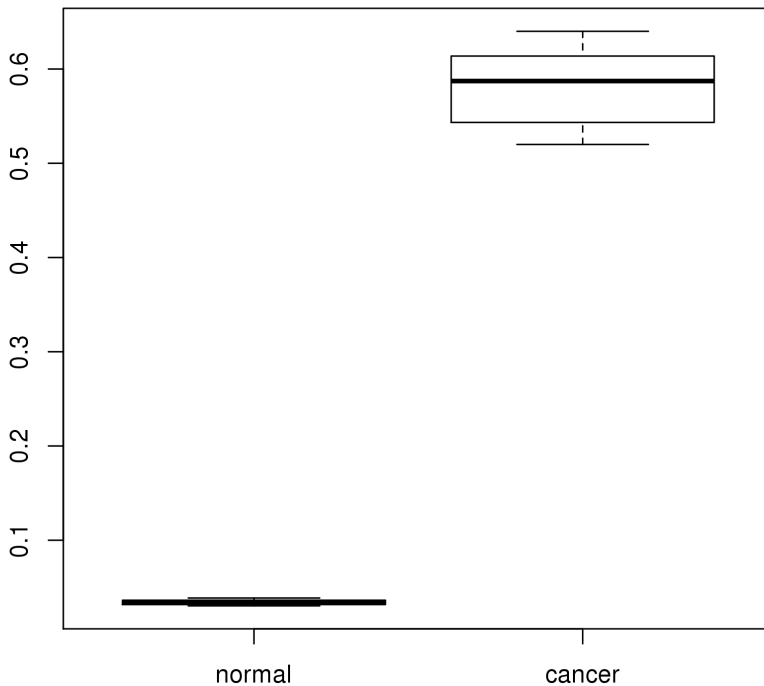
> topTable(m.fit,coef=2)
ID logFC t P.Value adj.P.Val
7422 cg07447922 5.376914 29.59677 1.066920e-09 1.328091e-05
20596 cg20645065 6.014610 28.36971 1.510658e-09 1.328091e-05
13829 cg13877915 5.313083 26.92885 2.317002e-09 1.328091e-05
13972 cg14011639 4.148885 26.84753 2.375191e-09 1.328091e-05
18880 cg18888520 5.272871 26.80282 2.407882e-09 1.328091e-05
```

You should be able to observe from these tables that the results are different, demonstrating the fact that performing this transformation of the data is not a pointless task. However, it is always a good idea to produce plots using the beta values, as these are far more readily interpretable (multiply the beta value by 100 to get the percentage)! For example, we can create a boxplot to inspect the difference in methylation levels between the cancer and normal samples, according to one of the most significant probes from the differential methylation analysis. The following command should give rise to the plot shown in [Fig. 9.5](#):

```
> boxplot(list(normal=beta.values[7422,c(1:4)],
+ cancer=beta.values[7422,c(5:9)]))
```

9.7 Further analyses using minfi

The *minfi* package enable a range of other analysis tasks to be performed on DNA methylation datasets, including identification of *differentially-methylated regions* (rather than individual CpG sites), estimation of chromatin compartmentalization based on long-range correlations of methylation, and a number of methods for application to human blood DNA samples (typically obtained for epidemiological studies). Rather than give examples for these further

**FIGURE 9.5**

DNA methylation beta-values from probe cg07447922, comparing normal ecto-cervix to cervical cancer samples.

analytical procedures here, we refer the reader to the *minfi* documentation on the Bioconductor website:

[“https://bioconductor.org/packages/minfi/”](https://bioconductor.org/packages/minfi/).

We will revisit analysis of DNA methylation data in [Chapter 15](#), which will demonstrate how to work with bisulphite-sequencing data.

Bibliography

- [1] MJ Aryee *et al*, “Minfi: a flexible and comprehensive Bioconductor package for the analysis of Infinium DNA methylation microarrays”, *Bioinformatics* 30(10):1363–1369.

- [2] A Stone *et al*, “DNA methylation of oestrogen-regulated enhancers defines endocrine sensitivity in breast cancer,” *Nature Communications* 6:7758 (2015).
- [3] P Du *et al*, “Comparison of Beta-value and M-value methods for quantifying methylation levels by microarray analysis,” *BMC Bioinformatics* 11:587 (2010).

10

DNA Analysis with Microarrays

10.1 Introduction

Allele-specific hybridisation of microarray probes can be used to determine SNP genotypes (commonly polymorphic positions in the genome). Genotyping microarrays typically work with sets of *polymorphic probes*: a number of near-identical probe sequences matching the target genomic sequence, but varying in a single base (that of the target SNP position). Each probe will hybridise more efficiently to the exact complementary sequence, and thus the ratio of intensities arising from the different forms of the polymorphic probes can be used to estimate ratios of the SNP alleles present in the sample. While individual probe sets may have particularly high or low intensities due to a number of technical effects, a region of consecutive probe sets with elevated or lower intensities than expected (according to the overall average) can indicate a region of DNA copy-number gain or loss. Therefore, this tutorial covers both aspects of analysis of high-density DNA hybridisation arrays: genotyping and copy-number analysis.

In this tutorial we will focus on the most widely-used platform, the Affymetrix SNP 6.0 microarray. Although, owing to the size of many datasets it is likely that the majority of the analysis topics covered in this tutorial will need to be carried out by a bioinformatics support facility.

10.2 Genotyping

Genome-wide SNP arrays generate a **lot** of data... *after* data processing and genotype-calling there will be of the order of a million measurements for every sample! Additionally, because there are so many measurements for every sample, adequately powering a genome-wide association (GWAS) study to deconvolute the shared genotypic differences between samples and detect significant associations between individual SNP alleles and some phenotypic trait of interest requires a large number of samples. Many GWAS studies involve thousands of samples. Processing all this data, even for individual arrays, in

R can cause considerable computational burden. In order to perform this data processing for SNP arrays, the *crlmm* package in R makes use of a package called *ff*, which keeps the data on the hard disk rather than loading it all into memory. To install and load the required packages, follow these steps (this may take a while):

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("crlmm")
> library(crlmm)
> biocLite("genomewidesnp6Crlmm")
```

The `biocLite` function, downloaded from Bioconductor's website, installs the *crlmm* package and all of its dependencies. In order to process raw data from any microarray, we need to install the annotation package corresponding to the array, in this case *genomewidesnp6Crlmm* provides the required annotation information to run *CRLMM* on Affymetrix Genomewide SNP 6.0 microarrays.

The goal in processing genome-wide SNP arrays is to estimate the *genotype call* for each SNP, for each sample. The genotype call refers to the relative number of major/minor alleles, and is commonly represented 'AA' (homozygous major allele), 'AB' (heterozygous) or 'BB' (homozygous minor allele). Details on how the *CRLMM* (Corrected Robust Linear Model with Maximum likelihood distance) algorithm calculates these can be found in [1]. In an ideal world, with intensities of the different probes being directly proportional to the number of copies of each allele present in the sample, simple thresholds could be applied to tell if the ratio of intensities most closely represented 1:0 (AA), 1:1 (AB) or 0:1 (BB). However, owing to technical sources of variation such as background noise, spatial artefacts and off-target hybridization, a more sophisticated approach must be taken. With the *CRLMM* approach, a set of arrays are normalized together to remove technical variation, then the genotype calls are generated on the basis of log-ratios between the A and B alleles for each SNP, adjusting for probe sequence, fragment length and total signal intensity effects.

10.2.1 Normalization

As with gene expression microarrays, individual probe-level measurements from genotyping arrays need to be *normalized* to remove as much technical variation from the data as possible, without obscuring any underlying biological signal. The reader is referred to [Chapter 8](#) for a description of some of the sources of technical bias in microarray measurements.

The *crlmm* package implements a modified version of the *RMA* algorithm, known as *snpRMA*, which has been specifically adapted to work on Affymetrix Genotyping arrays. In fact, the raw data (CEL file) normalization

is incorporated as part of the genotype-calling function `crlmm`, and does not need to be called explicitly. However, it is worth knowing that this normalization is performed during this process.

10.2.2 Genotype calling

In practical terms, using the *crlmm* package to generate genotype calls is quite straightforward. However, a number of things need to be put in place first. Our example dataset for this section will be a set of 24 small cell lung cancer cell line samples. Results of the study have been published [3] and the data is publicly available from GEO with accession number GSE40142. Download and unpack all the raw data: note, this will require 2GB of hard disk space. Once the .CEL files have been extracted, set the R working directory to the one containing these .CEL files. Make sure the *crlmm* package is loaded into the R workspace, then proceed with the genotype calling as follows:

```
> celFiles <- list.celfiles(full.names=TRUE)
> genotypeCalls <- crlmm(celFiles)
```

Here we define the list of cel files to be analysed, then invoke the `crlmm` function.

There is a chance that one or more of the CEL files may be corrupted in the course of downloading and extracting into the filesystem. You can use the `validCEL` function to find any corrupted files before running the genotype calling. If, for example, the file GSM986170.CEL was corrupted, you would see the following:

```
> validCEL(celFiles)
Problem reading ./GSM986170.CEL
```

Corrupted file won't be read in correctly and would cause an error in the genotype calling procedure, preventing any result from being created. We can remove any corrupted files using the `setdiff` function:

```
> celFiles <- setdiff(celFiles, "./GSM986170.CEL")
```

The `setdiff` function returns as its output what is left of the first argument (in this case `celFiles`) after removing any elements that match the second argument (in this case, `./GSM986170.CEL`, which is the location of the corrupted file). Now when we rerun the file check, it reports no errors:

```
> validCEL(celFiles)
[1] "Successfully read all cel files"
```

In this case, we can now attempt to rerun the genotyping function on the new vector of CEL file names:

```
> genotypeCalls <- crlmm(ceFiles)
```

Assuming that the `crlmm` function has been run successfully on the raw data files, to retrieve the genotype calls from the output, use the `calls` function (indexing the first three rows and first three columns of the resulting matrix, just to make it readable):

```
> calls(genotypeCalls)[c(1:3),c(1:3)]
SNP_A-2131660 3 3 3
SNP_A-1967418 3 3 3
SNP_A-1969580 3 3 3
```

As seen in the above command, the `calls` function returns a matrix of numerical values, in which 1 represents an AA genotype for the corresponding SNP, 2 represents AB and 3 represents BB. By itself, the result is perhaps of little value. However, with annotation of the SNPs, we may be able to interpret more from the data! The probe set identifier corresponding to each row can be retrieved using the `featureNames` function:

```
> SNPids <- featureNames(genotypeCalls)
```

To use the data a little more wisely, we need to know what these SNP IDs actually represent. For this, we need to access annotation information. A table with annotation listing the chromosome, position and dbSNP ID corresponding to each SNP array probe IDs is available for download from GEO, under the record GPL6801. Download this table and save it on your filesystem in the current working directory. Load the table into R, and then put the corresponding rows in order:

```
> snp.annot <- read.table("GPL6801-4019.txt"),sep="\t",header=TRUE
> snp.annot <- snp.annot[as.character(snp.annot$ID) %in% SNPids,]
> dbSNPids <- as.character(snp.annot$SNP_ID)
> SNPchromosomes <- snp.annot$Chromosome
> SNPlocations <- as.numeric(as.character(snp.annot[,7]))
```

Here we have kept only annotated SNP probes which are measured in the dataset (i.e. they are in the vector `SNPids`). Then for convenience, we have extracted the columns in the annotation table giving dbSNP IDs, chromosome and position, respectively.

Now, if we were particularly interested in SNP alleles in the region of a particular gene, say `PIK3CA`, we can find the corresponding genotype calls. First, we need to look up the gene's location, using the same genome build as that

used for the microarray annotation package (in this case, hg19)! Doing this using the UCSC genome browser¹ tells us that the genomic co-ordinates of the gene are Chr3:178,866,311-178,952,497. We can use the `which` function to find the relevant rows from the genotype call table:

```
> PIK3CA.SNPs <- which(SNPchromosomes==3 & SNPlocations>178866311
+ & SNPlocations<178952497)
```

Now we can inspect the genotype calls for each tumour sample by using this `PIK3CA.SNPs` object to index the full table:

```
> calls(genotypeCalls)[PIK3CA.SNPs,]
```

And we can find how many of the cell lines have each genotype, for each SNP, using the function `table`. For example, for the first SNP mapping to the `PIK3CA` gene:

```
> table(calls(genotypeCalls)[PIK3CA.SNPs[1],])
```

In this function, one row of the matrix resulting from applying the function `calls` to the object `genotypeCalls` has been indexed. Then the function `table` counts the number of elements in the vector passed to it which have each value. The output should appear as below:

```
1 2 3
19 3 2
```

The `table` function outputs a vector of named numeric elements, one for each unique value of the input. In this example, all possible values (1=AA, 2=AB, 3=BB) have been observed, with the respective counts 19, 3 and 2. But if we want to specify all the values that could have been observed, we can first convert the input into **R**s *factor* datatype, specifying the *levels* (i.e. the values) that the factor will take. This is particularly useful when comparing sets of genotype counts, as it ensures we will be comparing like with like. To repeat the previous command but using the `factor` to specify the possible genotypes as 1 (AA), 2 (AB) or 3 (BB):

```
> table(factor(calls(genotypeCalls)[PIK3CA.SNPs[1],],levels=1:3))
```

The `factor` function has been used here with two arguments: the first is the vector of values, the second (called `levels`) is the vector of *possible* values. If the vector of values includes some elements not in the list of possible values, those impossible values will be replaced with `NA`s to denote missing values.

¹<http://genome.ucsc.edu/cgi-bin/hgGateway>

10.2.3 Downstream analysis: Genome-wide association tests

For the dataset in question, we do not have any further data associated with the samples. If we did, it would be possible to apply the *limma* methods for analysis using linear models just as were used for gene expression microarray data in Sections 8.6 and 8.7. In this way, we could find SNP alleles associated with any desired variables. However, it is worth bearing in mind that the hypothesis testing from linear models as in the *limma* framework is best suited to variables which are at least approximately normally-distributed. For the case of genotyping data, we know that is not the case. Instead, for a two-class comparison we may wish to test statistical *independence* of the genotype with the variable of interest: we can do this using a χ^2 (*Chi-squared*) test.

As mentioned, in this example we do not have any functional data associated with the genotyped samples. However, let's pretend that half of the cell lines were derived from patients affected by a disease, and half from unaffected individuals. For convenience, let's say that columns 1–12 were the patients, and columns 13–24 were the unaffected individuals. What we want to test then, is for each genotyped locus what the probability would be of obtaining such different proportions of genotypes (the relative frequency of AA, AB and BB calls) across the two groups of individuals if the calls for each group had been sampled from the same distribution. To illustrate the process for a single SNP locus, we can compare the distributions of genotypes for the first locus mapping to the PIK3CA gene (using the mappings from the previous section):

```
> ctab <-cbind(table(factor(calls(genotypeCalls)[PIK3CA.SNPs[1],
1:12],levels=1:3)),
+ table(factor(calls(genotypeCalls)[PIK3CA.SNPs[1],13:24],
levels=1:3)))
```

Note how this repeats the previous command to count the different genotypes of the selected SNP, having converted the vector of genotype calls to a *factor*, specifying the possible values with the `levels` argument. But rather than counting the genotypes across all columns of the matrix returned by the `calls` function, it first counts only across columns 1 to 12 and then counts only across columns 13 to 24. The `cbind` function is used to combine these two vectors of genotype counts to form the columns of an array, which is stored on the workspace as the object `ctab`. This is the 3×2 contingency table that can be tested for independence.

The final part of the procedure for testing for independence of two distributions is to apply the Chi-squared test using the `chisq.test` function.

```
> chisq.test(ctab)
```

Output should appear as follows:

```
Pearson's Chi-squared test
data: ctab
X-squared = 2.386, df = 2, p-value = 0.3033
Warning message:
In chisq.test(ctab) : Chi-squared approximation may be incorrect
```

As with many applications, once you are able to evaluate associations for a single molecular feature (in this case a SNP), it becomes relatively simple to extend this to *all* available features. To perform a genome-wide association evaluation, create vectors of missing values to store the Chi-squared test p-values, and then we can put these into a data frame along with any feature annotations, and sort the table.

```
> gwas.pvals <- rep(NA,nrow(calls(genotypeCalls)))
```

This command creates a vector called `gwas.pvals`, with the same number of elements as the number of rows in the matrix resulting from applying the `calls` function on the `genotypeCalls` object. The next step is to run the `for` loop to compute the chi-squared p-values for each SNP having genotypes for which we can reject the null hypothesis of independence between genotypes and the phenotype of interest (in this example, the phenotype is just an arbitrary distinction of the cells into two groups, but in real applications this could be something meaningful).

```
> for(i in 1:nrow(calls(genotypeCalls))){
> ctab <- cbind(table(factor(calls(genotypeCalls)[i,1:12],
levels=1:3)),
+ table(factor(calls(genotypeCalls)[i,13:24],levels=1:3)))
+ gwas.pvals[i] <- chisq.test(ctab)$p.value}
```

In this set of commands, a `for` loop is run with a variable `i` taking values from 1 to the number of genotyped features. For each value of `i`, a new array (still called `ctab`) is created containing the genotype call counts for the two groups (each group of samples represented as a column in this array). The `i`th element of the `gwas.pvals` vector is then set to the p-value from the result of applying a Chi-squared test for independence to the contingency table `ctab`. Now a data frame can be created containing the SNP IDs and any available annotation information:

```
> gwas.table <-data.frame(probeID=rownames(calls(genotypeCalls)),
+ SNP.ID=dbSNPids,chr=SNPchromosomes,loc=SNPlocations,
+ p.value=gwas.pvals)
```

So we have used the `data.frame` function to create a data frame called `gwas.table`, containing named columns: `probeID`, `SNP.ID`, `chr`, `loc` and `p.value`, using vectors created throughout this section. Finally, we can add adjusted p-values to the data frame using the `p.adjust` function, and sort the table so that the loci with the most significant associations are at the top.

```
> gwas.table$adj.p <- p.adjust(gwas.table$p.value)
> gwas.table <- gwas.table[order(gwas.table$p.value,decreasing=F),]
```

The output of `p.adjust` function has been added as a new column of the `gwas.table` data frame, and named `adj.p`. Then the `gwas.table` object has been replaced by a version of the same data frame but with the rows sorted on increasing values of the `p.value` column. What are the 3 SNPs with the genotypes most significantly-associated with (that is, a difference in the proportions of genotypes across) the specified sample grouping?

```
> gwas.table[1:3,]
```

One limiting factor in the utility of such downstream analysis is the effect of multiple comparisons: applying the same statistical test a million times makes it quite likely that at least **something** would appear statistically significant purely by chance. So false-positives can be a real problem because so many more variables are measured than there are samples for which measurements are available. In order to reduce vulnerability to this effect, it is normal to apply some filtering in order to select just a subset of the profiled SNPs, which some prior information suggests are more likely to be functionally relevant if observed to associate with the output variables. Such filters could be selecting SNPs lying in or near candidate genes (such as commonly mutated genes), selecting only non-synonymous SNPs, or selecting SNPs with previously-described disease associations. A wealth of information can be found from the dbSNP database², with SNP IDs mappable back to rows of the genotype calls table through the microarray annotation provided by GEO under the accession GPL6801.

10.3 Copy number analysis

While high-density DNA microarrays were originally conceived with genotyping in mind, working with genetic copy number analysis is, in many ways, easier to manage. In fact, genotyping microarrays usurped array-based

²<http://www.ncbi.nlm.nih.gov/projects/SNP/>

Comparative Genomic Hybridization (aCGH) platforms as the standard tool for copy-number profiling, and are only now beginning to be replaced by low-coverage whole genome sequencing. There is a fairly wide range of programmes available for determining copy number from genotyping arrays, and a number of common platforms used for genotyping analysis. However, for simplicity and consistency with the other tutorials in this set, we will focus on using the R package *crlmm*, available through Bioconductor³ and the Affymetrix Genome-wide SNP 6.0 microarray platform.

Some data processing steps apply to both oligonucleotide SNP arrays and aCGH platforms: normalization of probe-level measurements to reduce the impact of technical variation on copy-number calls, some (typically model-based) estimation of genetic copy number at each profiled position/region, and the application of a *segmentation algorithm* to transform the individual copy-number estimates into a genome-wide profile by predicting the most likely positions of breakpoints around each copy-number aberration (CNA). aCGH platforms require {test,control} pairs of samples by design, but with SNP arrays if the samples being profiled consist of {test,control} pairs, the data processing steps may utilise information from the control to identify regions of *difference* between the two samples. Without control samples, data analysis aims to identify regions of consistently deviating intensity relative to the individual sample's average. Some strategies involve using a single control sample to provide a common reference for a large set of test samples, but the methods described here have been shown to work effectively even without a control sample.

10.3.1 Normalization

While the normalization procedure was performed as part of the genotype calling in *crlmm*, copy number estimation using this package requires explicit data normalization. We must first read the raw data and create an object on the workspace (the data is stored the file system, but we need an R object to point to the right files):

```
> library(ff)
> cnSet <- genotype(celFiles,cdfName="genomewidesnp6",
+ batch=rep("batch",length(celFiles)))
```

To use the `genotype` function, we must specify the *chip definition file* to be "genomewidesnp6" and indicate that all samples belong to the same batch. Also, the *ff* library must be loaded for these functions to operate properly.

³<http://bioconductor.org>

10.3.2 Copy number estimation

The copy number estimation implemented in the `crlmmCopynumber` function is described in detail in [2]. Invocation of this function to calculate the segmented genome-wide copy number estimates is simple, but execution may take a **long** time, particularly for larger datasets:

```
> crlmmCopynumber(cnSet)
```

The `cnSet` object we have just created contains information mapping each row in the numerical output table to genomic co-ordinates. For example, we can obtain a subset of all the copy number estimates by selecting only those rows mapping to chromosome 8:

```
> chr8.cnset <- cnSet[which(chromosome(cnSet)=="8"),]
```

We can use the `totalCopynumber` function to get the copy number calls for each probe from the `cnSet` object:

```
> CNcalls <- totalCopynumber(cnSet, j=c(1:length(sampleNames(cnSet))))
```

The `totalCopynumber` function requires you to specify which samples you wish to tabulate the copy number estimates from: in this case, that is all of the samples in the dataset.

The resulting table has a very large number of measurements. These can be taken straight into downstream analysis, with annotation for each row of the table provided with a few functions:

```
> cnv.probes <- featureNames(cnSet)
> cnv.chromosomes <- chromosome(cnSet)
> cnv.positions <- position(cnSet)
```

This table can be taken straight into downstream analysis, but the results may be liable to probe-specific effects as the copy-number calls have not been smoothed out over regions (which makes the results far more robust).

10.3.3 Segmentation

Following raw copy number estimation, the final stage of processing SNP array data to obtain copy number estimates is to use information from the consecutive copy number estimates arising from individual SNP probes in order to estimate region-wide copy-number. In essence, this smooths out the individual calls to make them less prone to errors arising from technical variation in probe-level intensity measurements. There are two main approaches to copy number segmentation with relatively easy to use implementations in R. We will look at one example of each.

10.3.3.1 Hidden Markov model

One common approach to segmentation is to apply *Hidden Markov Model* (HMM) methods, such as implemented in the *VanillaICE* package. These methods tend to work best when non-contaminated data is available. For processing data from primary tumour samples, you are advised to follow the instructions in the *Circular Binary Segmentation* section, which follows this. To run the *HMM* algorithm, first we must install and load the *VanillaICE* package⁴:

```
> biocLite("VanillaICE")
> library(VanillaICE)
```

The `hmm` function only runs on the chromosomes 1-22, so we subset the `cnSet` object with only those copy-number calls representing chromosomes 1-22:

```
> CNVmodel <- hmm(cnSet[which(chromosomes(cnSet)
+ %in% c(1:22)),])
```

The `CNVmodel` object created with this command is in the *RangedData* class of R objects: this allows values to be defined over many regions of different sizes in some set of *spaces*. In this case, the `space` and `ranges` part of the object are somewhat abstract, but other values in the *RangedData* table are readily interpretable:

- `chrom`: the chromosome in which the CNA region lies
- `num.mark`: the number of *markers* (that is probes) lying in the region
- `id`: the sample to which the region refers
- `state`: the inferred HMM state for the region. With default settings, the states 1-6 correspond to copy-numbers 0,1,2,2,3,4
- `LLR`: the log-likelihood ratio. The higher the value, the more significant the call of aberrant copy number in the region.

The `CNVmodel` object can be indexed to obtain a subset of the segmented copy-number regions. For example, to obtain all the CNA regions from one particular sample (e.g. GSM986170_CBE_ROM3-P1-1_H196_031209_GWS6.CEL) enter:

```
> CNVmodel[which(CNVmodel[[3]]==
+ "GSM986170_CBE_ROM3-P1-1_H196_031209_GWS6.CEL"),]
```

⁴The `biocLite` function needs to be loaded using: `source("http://bioconductor.org/biocLite.R")`.

Here we have used the fact that the third value term of the `CNVmodel` object listed the sample names for each region, so the `which` function returns the indices of those regions that correspond to the chosen sample.

To find the genomic positions covered by the probes in a given segment (or set of segments), you can use the annotation provided in the `cnSet` object. However, first you must find which of the probes are covered by the segment. To obtain this information for the first segment in the `CNVmodel` table, we would proceed as follows:

```
> biocLite(oligoClasses)
> library(oligoClasses)
```

First, the *oligoClasses* package must be installed (if necessary) and loaded into the workspace.

```
> reg1.markers <- as.matrix(findOverlaps(CNVmodel[1,],
+ subject=cnSet[autosomal,]))
```

In this command, we have used the `findOverlaps` function from the *oligoClasses* package to obtain the probe indexes contained in the region defined by the first segment of the `CNVmodel` object. The `subject=cnSet[autosomal,]` argument specifies the object that was passed to the `hmm` function in order to calculate the copy number segments.

Now we can construct a *data frame* containing the probe ID, chromosome name and chromosomal position of each probe, using the `reg1.markers` object created in the previous command to index the `cnSet` annotation information:

```
> reg1.probes <- data.frame(ID=featureNames(cnSet[autosomal,])
+ [reg1.markers[,2]], chr=chromosome(cnSet[autosomal,])
+ [reg1.markers[,2]], pos=position(cnSet[autosomal,])
+ [reg1.markers[,2]])
```

The first few rows of this annotated data frame should appear as follows:

```
> head(reg1.probes)
ID chr pos
1 SNP_A-1991776 10 291134
2 SNP_A-1991777 10 294953
3 SNP_A-1991778 10 309526
4 SNP_A-4266693 10 514032
5 SNP_A-1991794 10 514138
6 SNP_A-1991801 10 533331
```

It would help to have a vector containing the copy-number states represented by each HMM state (we will use this later):

```
> CNstates <- c(0,1,2,2,3,4)
```

Copy-numbers for individual samples, and for individual genes can therefore be looked up according to which region of the `CNVmodel` object applies to them. However, most downstream analysis is greatly simplified if we convert these probe-wise or segment-wise representations of copy number into gene-wise representations. This will be covered in the final section of this tutorial.

It may be useful to create a table containing the full, segmented copy-number representation in probe-wise format. This will likely require a lot of memory to create, especially for datasets with many samples. One way to do this would be first to create the table, initialising all values to a copy number of 2 (normal). Then, for each region of the `CNVmodel` object in turn, find which rows (probes) and column (sample) of the new table are referred to, and set the relevant values to the appropriate copy number:

```
> fullCNVtable <- array(2,dim=dim(calls(cnSet[autosomal,])))
> rownames(fullCNVtable) <- featureNames(cnSet[autosomal,])
> colnames(fullCNVtable) <- sampleNames(cnSet[autosomal,])
```

We need to be explicit with the fact that the `CNVmodel` object refers only to regions from the subset of the `cnSet` object that is indexed by the `autosomal` vector.

```
> for(sID in sampleNames(cnSet)){
+ for(region in which(CNVmodel[[3]]==sID)){
+ probeIndex <- as.matrix(findOverlaps(CNVmodel[region,],
+ subject=cnSet[autosomal,]))[,2]
+ fullCNVtable[probeIndex,which(colnames(fullCNVtable)==sID)]
+ <- CNstates[state(CNVmodel)[region]]
+ }}
```

This sequence of commands sets up two nested *for*-loops: the first goes through each sample in turn, referring to the sample in question as `sID`, the second goes through each region of the `CNVmodel` object that refers to the sample in question, referring to the region in question as `region`. Then two commands are executed for every region, for every sample: the first creates an index object called `probeIndex`, which finds which of the probes are covered by the region (this is as we have done previously). The final command sets the values of the table `fullCNVtable` indexed by the `probeIndex` object and the sample name `sID` to be the copy-number state estimate provided by the `CNVmodel` object (this is the sixth set of values associated with each region) for the specified

region. This loop may take a long time to run, but the output can be written to file using the `write.table` function and is a simple table of values that can be analysed using the methods covered in previous tutorials for other numeric data tables (such as gene expression data).

10.3.3.2 Circular binary segmentation

Another standard approach to segmentation, the *Circular Binary Segmentation* algorithm, as implemented in the *DNAcopy* package, can be used. This is particularly appropriate for tumour datasets, where sample heterogeneity is likely to give rise to fractional copy-number gains or losses.

```
> CNcalls <- totalCopynumber(cnSet, j=c(1:length(sampleNames(cnSet))))
> autosomal <- which(chromosome(cnSet) %in% c(1:22))
```

These first commands create a table of the copy number calls from the `cnSet` object, then index only those on chromosomes 1 to 22.

```
> CNV.object <- CNA(CNcalls[autosomal,], chrom=
+ chromosome(cnSet[autosomal,]), maploc=position(cnSet[autosomal,]))
> CNV.smoothed <- smooth.CNA(CNV.object)
> CNV.out <- segment(CNV.smoothed)
```

The `CNV.out` object created by the above command contains a *data frame*, which is very similar to the `CNVmodel` object described in the *HMM* section immediately prior to this. The only real differences lie in the ways to access the information contained in the object. We can inspect the first few rows of this data frame using `$out` to refer to the object:

```
> head(CNV.out$out)
```

The segments data frame contained in `CNV.out$out` can be used for anything that the segment regions object `CNVmodel` can be used for in terms of downstream analysis, as will be covered in the next section. There are plotting tools provided in the *DNAcopy* package that make creating graphical representations of the output relatively simple. To produce a plot of the estimated copy number at each position in a sample, you can use the following command (which should give rise to the plot shown in [Fig. 10.1](#)):

```
> plotSample(CNV.out, sampleid=sampleNames(cnSet)[1])
```

Additionally, with the `CNV.out` object, frequencies of gain or loss of segments can be retrieved using the `glFrequency` function:

```
> glFrequency(CNV.out)
```

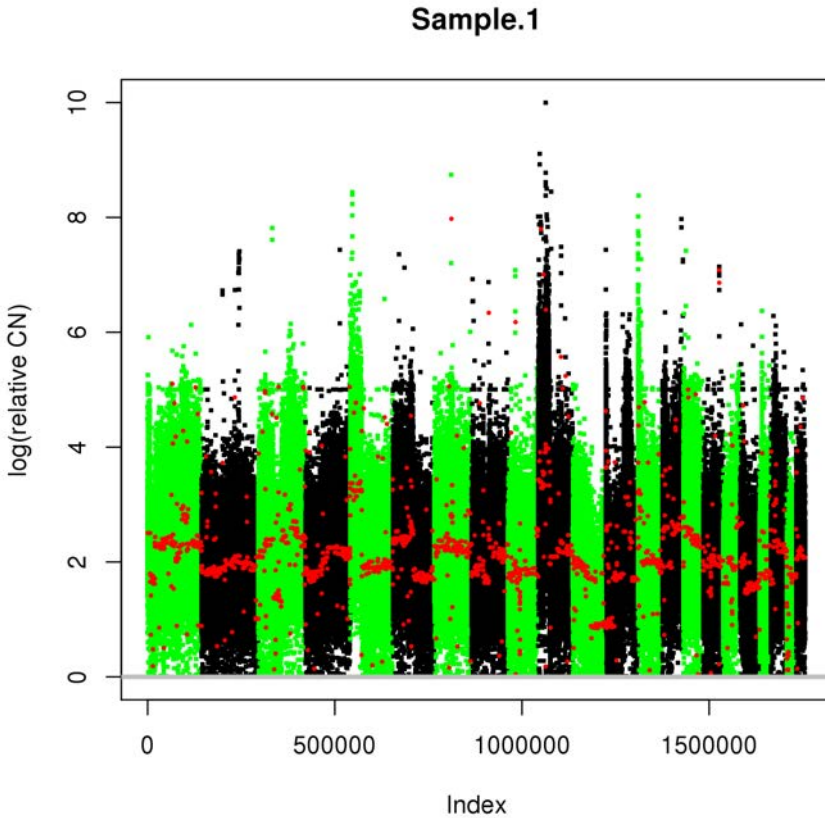


FIGURE 10.1

Plot of copy number estimates in sample GSM986170_CBE_ROM3-P1-1_H196_031209_GWS6.CEL from the circular binary segmentation algorithm. Separate chromosomes are shown in different colours.

10.3.4 Downstream analysis

10.3.4.1 Mapping CNA data to genes

Genetic copy number data is often available in probe-wise or segment-wise representations. Probe-wise representations contain all the information from a dataset, but suffer the same limitations as genome-wide genotyping data in having extremely large numbers of measurements for every sample. Additionally, adjacent probes will typically have the same values in most samples. So there may be many thousands of probes with exactly the same statistical association to some variable of interest. Add this to the fact that a biological

interpretation of genetic copy number typically revolves around the genes covered by regions of CNA, and obtaining a gene-wise representation of the results of copy number analysis seems to make sense.

To create a gene-wise representation from segment-wise or probe-wise representations of copy number data, a table of all genes' chromosomes and positions is required (or another mapping from probeID to gene symbol). For this, we will make use of the *biomaRt* package in R: BioMart is a tool from Ensembl which enables download of selections of the genome-wise annotation information that is displayed in the Ensembl genome browser, and *biomaRt* provides a simple way to access all this information directly through R.

First, install (if necessary) and load the *biomaRt* package:

```
> biocLite("biomaRt")
> library(biomaRt)
```

Using the *biomaRt* package requires specification of a database in which to look up the required information, then (if required) specifying filters so that only relevant information is returned. We will start by creating an object that points to the appropriate database (the human gene annotation data) for our query:

```
> ensembl <- useMart("ensembl",dataset="hsapiens_gene_ensembl")
```

One function, `getBM` allows us to retrieve desired information. It requires the specification of *attributes* to include in the output. A full list of attributes can be viewed with the following command:

```
> listAttributes(ensembl)
```

The information we need is: gene symbol, chromosome, start position and end position. We can specify the appropriate attributes and obtain the table of annotation information:

```
> geneInfo <- getBM(attributes=c("hgnc_symbol",
+ "chromosome_name","start_position","end_position")
+ ,filters="chromosome_name",values=c(1:22),mart=ensembl)
```

In the above command, the `getBM` function has been used with arguments `filters` and `values` specified to restrict the output to only those genes on chromosomes 1 to 22 (i.e. those genes for which we actually have segmented copy number estimates).

Before we use this table, we will filter out those entries without gene symbols:

```
> geneInfo <- geneInfo[which(!geneInfo$hgnc_symbol==""),]
```

We may also wish (or need) to include only one entry per unique gene symbol. Any selection is likely to be arbitrary, so we could just choose the first entry in the table for each gene:

```
> geneInfo <- geneInfo[(!duplicated(geneInfo$hgnc_symbol)),]
```

This command uses the `duplicated` function to find the indexes of all non-duplicated gene-symbols, and therefore removes all duplicate rows from the `geneInfo` table.

With the requisite gene annotation information, constructing a gene-wise copy number table for the dataset is very similar to constructing a probe-wise representation from the segmented data (as was carried out previously). We first construct a table with a row for each gene and a column for each sample, with values initialised to copy number 2 (normal). Then we find which region in the segmented table corresponds to each gene, for each sample, and places the copy number estimate into the appropriate position in the table.

For the output from the *HMM* algorithm, the table `CNVmodel`, use the following sequence of commands:

```
> CNVcalls.byGene <- array(2,dim=c(nrow(geneInfo),
+ length(sampleNames(cnSet))))
> rownames(CNVcalls.byGene) <- geneInfo$hgnc_symbol
> colnames(CNVcalls.byGene) <- sampleNames(cnSet)
> for(segment in c(1:nrow(CNVmodel))){
+ segment.genes <-which(geneInfo$chromosome_name==CNVmodel[[1]][segment]
+ & geneInfo$start_position>start(ranges(CNVmodel[segment,])[1]))
+ & geneInfo$end_position<end(ranges(CNVmodel[segment,])[1]))
+ segment.sample <- which(sampleNames(cnSet)==CNVmodel[[3]][segment])
+ CNVcalls.byGene[segment.genes,segment.sample] <-
+ CNStates[state(CNVmodel)[segment]]
}
```

This complex command is straightforward when broken down into its parts. There is one loop that goes through each segment listed in the `CNVmodel` object in turn, finding which genes lie in the region covered by the segment and which sample the segment refers to. To get a list of the genes lying in a given region, the `which` function is used to combine three tests (with a `&` symbol): which genes are on the same chromosome as the segment, which genes have start positions higher than the genomic co-ordinate of the beginning of

the segment, and which genes have end positions less than the genomic coordinate of the end of the segment. Thus, all (and only those) genes lying entirely within the segment will satisfy all three tests and be output by the `which` function.

For output from the *CBS* algorithm, the procedure is almost identical, just with slightly different ways to access the appropriate results from the `CNV.out` function:

```
> CNVcalls.byGene <- array(2,dim=c(nrow(geneInfo),
+ length(sampleNames(cnSet))))
> rownames(CNVcalls.byGene) <- geneInfo$hgnc_symbol
> colnames(CNVcalls.byGene) <- sampleNames(cnSet)
> for(segment in c(1:nrow(CNV.out$out))){
+ segment.genes <-which(geneInfo$chromosome_name==CNV.out$out[segment,2]
+ & geneInfo$start_position>CNV.out$out[segment,3]
+ & geneInfo$end_position<CNV.out$out[segment,4])
+ segment.sample <- which(sampleNames(cnSet)==CNV.out$out[segment,1])
+ CNVcalls.byGene[segment.genes,segment.sample] <-
+ CNV.out$out[segment,6]
}
```

The resulting table, `CNVcalls.byGene`, can be used just as any other numeric data table for downstream analysis as detailed in [Chapters 5](#) and [8](#). For example, the `limma` package can be used to evaluate associations between gene-level copy number estimates and any experimental variables of interest, as specified through a design matrix for a linear model.

10.3.4.2 Finding frequently-mutated genes

A simple analysis that is particularly useful with copy number data is calculation of the *frequencies* of copy-number aberrations (CNAs) are for each gene.

```
> amplification.counts <- apply(CNVcalls.byGene,MARGIN=1,
+ function(x)sum(x>2))
```

Here we have used the `apply` function to count the number of times each row in the table (i.e. each gene) has a copy number greater than 2 (i.e. it is amplified). We could use a similar command to count deletion events:

```
> deletion.counts <- apply(CNVcalls.byGene,MARGIN=1,
+ function(x)sum(x<2))
```

Now we can create a data frame and find the most common events:

```
> CNA.frame <- data.frame(ID=rownames(CNVcalls.byGene),
+ AmpCount=amplification.counts, DelCount=deletion.counts)
```

Ordering on the number of CNA events shows which positions are most commonly amplified or deleted.

```
> head(CNA.frame[order(CNA.frame$AmpCount,decreasing=TRUE),])
ID AmpCount DelCount
TET1P1 TET1P1 24 0
RPL29P29 RPL29P29 24 0
LINC00433 LINC00433 24 0
MRPS29P2 MRPS29P2 24 0
RAPGEF4-AS1 RAPGEF4-AS1 24 0
ALDH7A1P2 ALDH7A1P2 24 0
```

If we had a gene of interest (e.g. MYC), we could look up the corresponding row in the table to count the amplifications and deletions:

```
> CNA.frame[which(CNA.frame$ID=="MYC"),]
```

This analysis could also be performed on a table of gene expression data, with some set threshold, in order to find genes that are commonly over-expressed (or silenced) in a particular set of samples. However, determination of an appropriate threshold for such data is difficult, because there is no way to estimate a reference *background* level (as there is with copy number data).

10.4 Summary

Owing to its extremely high density of information, analysis of high-throughput genotyping microarray data can be fairly complex. As the computational resources required to process large datasets may be prohibitive, it is likely that many of the steps covered in this tutorial will be carried out by a bioinformatics service from the facility running your microarrays. However, there is still a great wealth of datasets available for which analysis would be computationally feasible, but even the basic steps of data processing to obtain tables of values that can be used for subsequent downstream analysis can be difficult.

It is therefore the aim of this tutorial to give you the tools to be able to process raw Affymetrix SNP array data to the stage at which downstream analysis is possible, and therefore to enable you to utilise more of the data available to you in domain data repositories (such as GEO) or from your own research centres. The end-points of the data processing described in this tutorial can feed directly into data analysis techniques detailed in the earlier tutorials from this set, and so hopefully you will be able to use these together

to harness the wealth of data that is publically available to gain insight into genetic events relevant to your biological and clinical research questions.

Bibliography

- [1] B Carvalho, H Bengtsson, TP Speed and RA Irizarry, “Exploration, normalization, and genotype calls of high-density oligonucleotide SNP array data,” *Biostatistics* 8(2):485–499 (2007).
- [2] RB Scharpf, I Ruczinski, B Carvalho, B Doan, A Chakravarti and RA Irizarry, “A multilevel model to address batch effects in copy number estimation using SNP arrays,” *Biostatistics* 12(1):33-50 (2011)
- [3] ML Sos *et al*, “A framework for identification of actionable cancer genome dependencies in small cell lung cancer,” *PNAS* 109(42):17034-9 (2012).

11

Working with Sequencing Data

11.1 Introduction

This chapter of the tutorials is not like the other chapters so far. I have tried to make these tutorials a *complete* introduction for each topic covered, with instructions for downloading and installing R and all required packages and data. Unfortunately, because of the sheer amount of data generated in today's typical sequencing experiments, analysis within R is not always feasible because it is designed to process all data in memory (RAM). Even using more memory-efficient tools, processing next-generation sequencing data will usually require the use of a very powerful computer. And even if you have such a computer at your disposal, it will quite often be administered by someone else. In fact, because of these difficulties I chose not to include sequence data analysis in my written tutorials for quite some time! I have tried to include walkthroughs of processes in R where possible, although I will also show how all steps can be performed using other software tools. This will include links to download pages for the tools, but *exact* installation instructions will often vary from platform to platform, and so you may need to request your local sysadmin or bioinformatician to ensure the installation of the tools and dependencies (other programs the tools need to run) for you. In order to simplify use of the many standalone software tools developed for analysis of high-throughput sequencing data, the examples given in these chapters assume that a UNIX-type environment is available. This could be through running the commands directly on a computer running a Linux distribution, through using the Terminal App in MacOS, or through a Linux Virtual Machine in Windows¹. If all else fails, nearly all the data processing steps here (but not the downstream analysis) can be performed online thanks to the Galaxy Project: <https://usegalaxy.org>. There will be limits on the sizes of datasets you can upload to the Galaxy Project servers, but it provides a graphical interface for applying many tools to the data you have uploaded, and you could then download the results and follow the R-based downstream analysis instructions in the subsequent chapters of this book.

¹If this seems difficult, it is even possible to run Linux operating systems directly from external media: look up creating a LiveCD, LiveDVD or LiveUSB.

11.2 Sequence data analysis tasks

There are a number of tasks which need to be performed to get the most out of a sequencing experiment, which are frequently linked together into ‘workflows’. The nature of these workflows varies for the different applications of high-throughput sequencing (e.g. genome sequencing, ChIP-seq, RNA-seq), in that these require different tasks. However, there are some tasks which are universally required across most applications. The main content of this tutorial will be to go through these tasks one by one, starting off with the universal tasks before progressing through workflows for each of the main applications of high-throughput sequencing (HTS). Note, there will be many ways of performing each of these tasks, I am only showing you the ways that I find work best for me! You may find that your sequencing facility or collaborators have already carried out a number of these tasks for you, or if you’re obtaining publically available data it may be most convenient to download ready-processed datasets. It is still worthwhile knowing what tasks ought to have been carried out in order to generate the processed data, and hence familiarize yourself with the whole of this tutorial, but for quick results you might want to jump to the relevant chapter: this might be [Chapter 12](#) if you already have variant calls in VCF files, or [Chapter 13](#) if you have ChIP-seq peaks in BED files.

11.3 Quality control

The real advantage of the so-called *next-generation sequencing* platforms is their throughput, not particularly their accuracy. While the platforms certainly are pretty accurate sequencers, their throughput means that typical values on an Illumina HiSeq 2000 of 1 in 500 bases called erroneously could mean that a single run of 600 Gb of sequence would contain over a billion errors! If the error rate is low and the errors are evenly distributed throughout the reads, then this should be fairly easy to work around. However, if the errors are particularly concentrated around certain features, or if the sequencing itself is biased to certain features, it may be worth specifically taking this into account. Fortunately, the technologies have been developed to help us out in this respect, in that the certainty of the base-calling is one of the pieces of information that is stored for each base-call of sequence data (in the Fastq format). These quality scores are encoded in a number Q such that the probability of base-call error $P = 10^{-\frac{Q}{10}}$. As a guide, $Q = 20$ means 1% expected probability of error, while $Q = 30$ means 0.1% expected probability of error. $Q = 30$ or above is often considered a ‘rule-of-thumb’ for good quality reads.

There are other aspects of the sequencing chemistry that can cause the read sequences that are returned to misrepresent the actual sequences of the DNA fragments in the library. The most common of these is probably the (partial) sequencing of the adapters and/or PCR primers that were used in order get a measurable signal for the sequencer.

There are a few general-purpose tools that generate QC reports directly from the Fastq files which is the standard ‘raw’ output from the sequencers. One simple but effective such tool is *FastQC* from the Babraham Institute, which is available to download at <http://www.bioinformatics.babraham.ac.uk/projects/fastqc/>.

FastQC has a graphical interface and produces a traffic-light-based navigator through the tests it carries out, for which an example is shown in Fig. 11.1. In Fig. 11.1 you can see that the quality scores across this library tend to decrease towards the end of the reads: that trend is typical of high-throughput sequencing runs, but the extent to which the quality scores decrease in this example is not typical and suggests there may be some problems with the input.

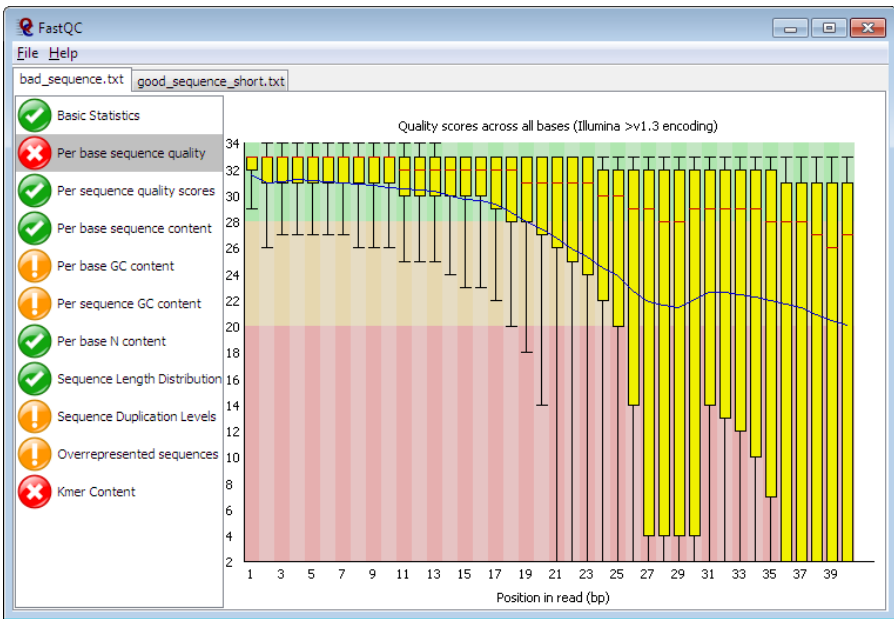


FIGURE 11.1

Per-base quality score chart produced from FastQC. Each boxplot shows the distribution of base call quality scores at that position in all reads in the input Fastq file.

Note that FastQC is a tool to draw to your attention potential problems with a sequence library, it doesn't include functionality to take action according to these problems. Additionally, as it is based around visual reports presented through a GUI, it would be extremely time-consuming to apply to a very large set of libraries in turn. If you plan an analysis of a large number of datasets it would perhaps be better to use processed data from a trusted research group or consortium (e.g. ENCODE), or to implement rule-based filtering and read trimming as we'll come to next.

11.3.1 Base call quality filtering

As we have based most of this tutorial set on the use of R, it makes sense to take advantage of the relevant functionality in R that comes from the *Bioconductor* project for working with sequence data [1]. One such area is in handling Fastq files, for which the *ShortRead* package can import Fastq files and access both sequence characters and numeric quality scores for all reads in the resulting object. Install the ShortRead package by running the following within the R environment:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("ShortRead")
```

Now, say we have a sequenced library in a file called 'cancertest.fq' and we wish to remove the poor-quality reads. The file I have used² is available for download from: <https://github.com/edcurry/bioinfo-book/cancertest.fq>. Once that file is downloaded, we can first import the file after loading the ShortRead package:

```
> library(ShortRead)
> reads <- readFastq("cancertest.fq")
```

There, that was easy! But we now need to access the quality scores, which is where things get slightly more complicated. The function `quality` returns the quality scores within a *BStringSet* object, which we can make more usable by converting it to a matrix³:

```
> quals <- as(quality(reads), "matrix")
```

In some recent versions of *Bioconductor*, that command will not work. If this is the case for you, the following work-around creates the same output:

```
> quals <- array(NA, dim=c(length(reads), width(reads[1])))
> for(i in 1:nrow(quals)){
```

²Derived through randomly-sampling reads from publically-available ChIPseq runs

³NB: there is a limit on the size of matrices it can create like this, and so if your Fastq file has over 20 million reads then you will have to break it into 20 M read chunks and apply the filtering separately, then combine the filtered read sets at the end.

```
+ quals[i,] <- as.numeric(quality(reads)[[i]])-33  
+ }
```

This approach creates an array full of missing values, with a row for each read in the `reads` object and a column for each sequencing cycle (assuming the first read is representative). The `for` loop steps through each read in turn, extracts the quality scores, converts them to a numeric vector and subtracts 33 to convert to the correct scale (if any scores end up less than zero, omit this subtraction).

We can in fact use R to create a similar plot to the one generated by FastQC, the result of which is shown in [Fig. 11.2](#), and shows that in this instance it is the start of each read which tend to have lower quality scores, albeit these are still mostly above our $Q = 30$ ‘good-quality’ standard:

```
> boxplot(lapply(1:ncol(quals),function(x)quals[,x]))
```

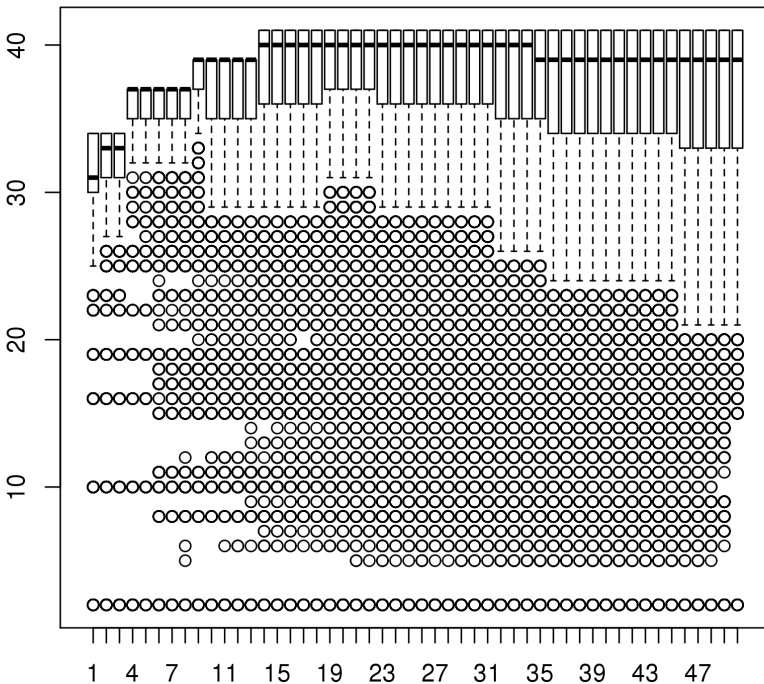


FIGURE 11.2

Per-base quality score boxplot produced in R. Each boxplot shows the distribution of base call quality scores at that position in all reads in the input FastQ file.

How you actually choose to filter reads may depend on what you expect of the sequence library and what you are using it for. To provide an example, if the ‘cancertest.fq’ file represents 50 bp Illumina reads, and I expect them to be pretty good quality, I might say that I want to discard any reads that have more than 5 base calls with quality lower than $Q = 30$. To achieve this I make use of the `apply` function to count the number of values below 30 in each row:

```
> goodq.reads <- reads[apply(quals,MARGIN=1,function(x)
+ sum(as.numeric(x<30))<=5)]
```

A quick examination of the length of the resulting filtered set tells us about 65% of the reads in this library pass the quality filter: that is not a lot! However, by trimming the ends of the reads (which are often where base call quality scores degrade) it may be possible to improve this number whilst applying exactly the same filter.

```
> length(goodq.reads)/length(reads)
[1] 0.64435
```

We might also want to discard any reads with ‘N’ base-calls (i.e. the sequencer couldn’t work out which base was in the corresponding position). For this we can use the `sread` function to get the individual base calls as a *DNAStringSet* object, to which we can apply the `alphabetFrequency` function and extract the column of the resulting matrix which represents ‘N’s:

```
> goodq.reads.noN <- goodq.reads[alphabetFrequency(
+ sread(goodq.reads))[, "N"]==0]
```

We can now export the filtered set of reads back out to file:

```
> writeFastq(goodq.reads.noN,file="cancertest_filtered.fq",
+ compress=FALSE)
```

11.3.2 Adapter trimming

As mentioned earlier, ‘contamination’ of adapter or primer sequences in read files can occur for a number of reasons, although is particularly common when short DNA fragments are being sequenced (so that the read length is longer than the fragment length). There are a number of tools for identifying recurring short sequences and removing these from reads in a Fastq file. One such example is the tool *Trimmomatic* [2], which is written in Java and can be obtained from <http://www.usadellab.org/cms/index.php?page=trimmomatic>. Download the Trimmomatic binary zip and then extract on a Linux system:

```
$ wget http://www.usadellab.org/cms/uploads/supplementary/  
Trimmomatic/Trimmomatic-0.33.zip  
$ unzip Trimmomatic-0.33.zip  
$ cd Trimmomatic-0.33
```

Then you can run the Trimmomatic program on the ‘cancertest_filtered.fq’ file by copying it into the Trimmomatic directory and then invoking:

```
$ java -jar trimmomatic-0.33.jar SE cancertest_filtered.fq  
cancertest_trimmed.fq ILLUMINACLIP:adapters/TruSeq3-SE.fa:2:30:10  
LEADING:3 TRAILING:3 SLIDINGWINDOW:4:25 MINLEN:36
```

Note that as well as specifying the input file `cancertest_filtered.fq`, an output file `cancertest_trimmed.fq` is also named: this output file doesn’t need to exist yet, the program will create it. The ‘ILLUMINACLIP’ argument tells the program to trim out known sequences from adapters used in the Illumina *TruSeq3* chemistry. The numbers ‘2:30:10’ in the argument each specify a threshold to be used in the trimming. First: up to two mismatches between read and adapter sequence should be allowed. Second: palindromic adapter sequences in paired-end reads will be removed if they have a matching score of at least 30. Third: any sequence to be trimmed must match an adapter sequence with a score of at least 10. The matching score algorithm is given in the Trimmomatic documentation, but it recommends setting this last value in the range 7–15, so 10 seems like a reasonable compromise. The other arguments specify additional read-trimming instructions: ‘LEADING’ removes consecutive bases that are lower than the specified quality score (therefore 3 is a very tolerant threshold!), starting at the beginning of the read. ‘TRAILING’ does the same but starting at the end of the read. ‘SLIDINGWINDOW’ will cut reads if they have a run of 4 consecutive bases with average quality $Q < 25$. The final argument ‘MINLEN’ specifies that reads should be discarded entirely if they are trimmed (in any way) to a length of less than 36 bp.

Paired-end reads are generated when each DNA fragment is sequenced from both ends at the same time. These libraries are often used when accurately mapping reads is more important than having a high resolution of coverage, for example in variant calling or bisulphite-sequencing. Trimmomatic can also be used for paired-end read libraries, which will have the sequenced ends of each fragment (known as ‘mate pairs’) kept in two separate files. If we instead had two Fastq files ‘cancertest_mate1.fq’ and ‘cancertest_mate2.fq’ which are the left and right ends of each pair of reads, then we could apply the same Trimmomatic approach as follows:

```
$ java -jar trimmomatic-0.33.jar PE cancertest_mate1.fq  
cancertest_mate2.fq
```

```
cancertest_trimmed ILLUMINA_CLIP:adapters/TruSeq3-SE.fa:2:30:10  
LEADING:3 TRAILING:3 SLIDINGWINDOW:4:25 MINLEN:36
```

In this command, rather than specifying a single output file, an output ‘base name’ is given: the program will use this base to name four separate output files: two files of paired reads (in this case they will be called ‘cancertest_trimmed_1P.fq’ and ‘cancertest_trimmed_2P.fq’), then two additional files for the reads in each input file that don’t have a mate-pair (because the other mate-pair may have been filtered out).

Note that Trimmomatic allows you to specify what adapter sequences you wish to remove, which could be any over-represented k-mer sequence as highlighted in the FastQC report (for example). Another widely-used tool for adapter trimming is *cutadapt*, which can be obtained from <http://cutadapt.readthedocs.org>.

11.4 Alignment

HTS technologies make use of parallelism in sequencing, in that rather than sequencing the entirety of a chromosome’s DNA in one pass, the DNA is split into millions of short fragments that are each sequenced simultaneously. The cDNA that is sequenced for an RNA-seq experiment, which is itself created from RNA, is split into short fragments. In order to make use of a vast number of short reads, we need to find out where in the profiled genome these reads map. This is the key to quantifying the numbers of fragments spanning any given genomic position or feature (a given TSS, for example), the key to working out which positions in the genome are reliably showing polymorphisms or mutations, and the key to working out any structural rearrangements of the sequenced genome (or transcriptome). Alignment involves assessing the closeness of matching of each sequence read to be matched against every possible position in the reference genome, then annotating each read with the best-matching position(s) or no position (if none was found to match sufficiently well). This process is typically based around some scoring criteria to determine how badly each mismatch between the read and matched reference sequence affects the mapping, which may vary according to the nature of the library being sequenced. Of course, this process requires a reference genome against which the sequence reads can be mapped. The most recent human reference genome release is known as GRCh38, but as there is a lot more annotation and existing experimental data available for hg19 (GRCh37) you may wish to use this instead. There are a number of reference genomes for different organisms available from UCSC at <http://hgdownload.soe.ucsc.edu/downloads.html>.

Alignment will usually be carried out by a sequencing facility as it is often a computationally intensive task, but if you want to perform alignment yourself the tools *Bowtie* and *BWA* are effective and widely-used. The output of alignment will be in SAM format (or BAM, which is just a compressed non-human-readable version of a SAM file). A SAM file consists of a header containing information about the collection of aligned reads and one row for each read, including: an identifier for the read, the mapped genomic co-ordinates, the sequence mapped, the number of mismatches to the reference, whether or not the read is part of a pair, and additional information. Full details can be found at <http://samtools.github.io/hts-specs/SAMv1.pdf>.

11.4.1 Bowtie

Bowtie is a fast, freely-available alignment tool initially described in [3]. It now comes in two flavours, *Bowtie 1* and *Bowtie 2*[4]. We will focus on Bowtie 2, which is probably better for alignment of longer reads, particularly those that have multiple-base insertions or deletions relative to the reference genome, as is often the case in cancer genomes. Bowtie 2 mapping is filtered according to a score which considers the number of matching bases, the number of mismatching bases, the number and length of gaps in the alignment, and the uniqueness of the alignment (i.e. the number of possible positions in the genome the read could have mapped). In Bowtie 1 filtering is carried out according to user-specified thresholds of the number of mismatches, and number of possible matched positions in the reference genome.

The Bowtie 2 project homepage is <http://bowtie-bio.sourceforge.net/bowtie2/index.shtml>, and the current release can be downloaded from <http://sourceforge.net/projects/bowtie-bio/files/bowtie2/2.2.6/>.

To install on a Linux machine, download the file *bowtie2-2.2.6-linux-x86_64.zip*. Save this file into the desired directory and run:

```
$ unzip bowtie2-2.2.6-linux-x86_64.zip
```

This download comes with a number of precompiled reference genomes indices, including the standard human reference hg19 (GRCh37). If you wish to align sequence reads to a reference genome not included in the Bowtie 2 bundle, for example another organism or a specific human genome assembly, you will need to download it from a repository (many are available at http://support.illumina.com/sequencing/sequencing_software/igenome.html) or to create an index yourself. Say, for example, I had the sequence for a cancer test genome with chromosomes 1, 2, X and Y in separate Fasta files in the same directory as the *bowtie2-build* executable, I would create an index thus:

```
$ ./bowtie2-build -f chr1.fa,chr2.fa,chrX.fa,chrY.fa cancertest
```

To run Bowtie 2 to align a Fastq file ‘cancertest.fq’ from single-end sequencing to the reference genome we just indexed, I would enter:

```
$ ./bowtie2 -x cancertest -U cancertest.fq -S cancertest_aligned.sam
```

Then we can look at the sam file:

```
$ head cancertest_aligned.sam
```

We can convert SAM to BAM files using samtools:

```
$ ./samtools view -bS cancertest_aligned.sam > cancertest_aligned.bam
```

Paired-end reads need to be aligned slightly differently. Let’s say we have two Fastq files ‘cancertest_mate1.fq’ and ‘cancertest_mate2.fq’ which are the forward and reverse ends for each pair of reads, then we would proceed:

```
$ ./bowtie2 -x cancertest -1 cancertest_mate1.fq -2  
cancertest_mate2.fq -S cancertest_alignedPE.sam
```

11.4.2 BWA

Another fast, freely-available alignment tool is *BWA*, the Burrows-Wheeler Aligner [5]. Again there are multiple versions of BWA that are optimized for different sorts of alignment tasks, which are explained in the BWA manual at <http://bio-bwa.sourceforge.net/bwa.shtml>, but it should be obvious which we are using. There are a number of parameters that can be specified, such as a maximum number of gaps in each alignment and a maximum allowed number of mismatches: I would recommend reading about these in the manual if you suspect you might want to use something other than the default values. To obtain the BWA program, download the latest archive (currently ‘bwa-0.7.12.tar.bz2’) from <http://sourceforge.net/projects/bio-bwa/files/> and save this to the computer you wish to work on. Unzip and unpack the archive (you will need to have installed the *bzip2-devel* package for your Linux distribution):

```
$ bunzip2 bwa-0.7.12.tar.bz2  
$ tar -xf bwa-0.7.12.tar
```

Now you will have the BWA programs and associated data in a directory called `bwa-0.7.12` (or the number of whichever version you obtained). Tell the Linux shell you wish to enter this directory with:

```
$ cd bwa-0.7.12
```

The BWA-MEM algorithm is recommended for generating either single-end or paired-end alignments from Fastq files. To run BWA-MEM to align a Fastq file ‘cancertest.fq’ from single-end sequencing to the custom reference genome, again we first need to index the reference (after using the *cat* command in Linux to combine the individual chromosome sequences together into a genome reference file):

```
$ cat chr1.fa chr2.fa chrX.fa chrY.fa > cancertest_Reference.fa
$ ./bwa index -p cancertest cancertest_Reference.fa
```

Now with an indexed reference genome, I would enter:

```
$ ./bwa mem cancertest_Reference.fa cancertest.fq
> cancertest_aligned.sam
```

Let’s say now we have two Fastq files ‘cancertest_mate1.fq’ and ‘cancertest_mate2.fq’ which are the left and right ends of each pair of reads, then we would proceed:

```
$ ./bwa mem -M cancertest cancertest_mate1.fq cancertest_mate2.fq
>
cancertest_alignedPE.sam
```

Another approach using BWA that works is to first obtain the co-ordinates of each read in the pairs directly from the reference genome, then generate alignments from the resulting mappings:

```
$ ./bwa aln cancertest_Reference.fa cancertest_mate1.fq > 1.sai
$ ./bwa aln cancertest_Reference.fa cancertest_mate2.fq > 2.sai
$ ./bwa sampe cancertest_Reference.fa 1.sai 2.sai cancertest_mate1.fq
cancertest_mate2.fq > cancertest_alignedPE.sam
```

11.4.3 Post-alignment filtering

As well as the quality scores associated with each base call in a sequence read, the information gained through attempting to match a sequence against a reference genome can also indicate possible errors in the sequenced library that will want to be removed before further analysis of the data. Alignment statistics can also indicate the presence of problems that make an entire library unreliable.

11.4.4 Removing duplicate reads

Amplification artifacts during the library preparation or sequencing processes can result in multiple reads reflecting a single DNA fragment. These can bias

any analysis based on quantifying reads, such as RNA-seq, identifying peaks in ChIP-seq data or setting a minimum number of supporting reads for variant calls. Due to the fact that sequencing errors can occur after duplication events, not all duplicate reads may even have exactly the same sequence. Unfortunately, this means that it is impossible to distinguish between an artificially duplicated stack of sequence reads mapping to the same genomic co-ordinate or multiple reads of identical DNA fragments. When wishing to quantify the numbers of reads mapping to certain regions, it becomes a point of analysis strategy whether or not to discard duplicate reads. This decision may be guided by the library complexity (i.e. number of distinct reads in the library relative to the total number of reads), for if the complexity is low it can be an indication of over-amplification of the input DNA and duplication artifacts may be a big problem. If one decides to remove duplicates from a set of aligned sequence reads, there are a number of ways to carry this out. One such method is available through *Picard Tools*, which can be downloaded from <http://broadinstitute.github.io/picard/>. The *MarkDuplicates* program can be run in such a way as to keep only one read mapping to each distinct set of genomic co-ordinates from an input SAM/BAM file. Start by downloading and unpacking the *Picard Tools* suite:

```
$ wget https://github.com/broadinstitute/picard/releases/download/
1.139/picard-tools-1.139.zip
$ unzip picard-tools-1.139.zip
$ cd picard-tools-1.139
```

Now we have a directory called ‘picard-tools-1.139’ which includes the *jar* Java package file ‘picard.jar.’ This program can be run through invoking *java* from the shell:

```
$ java -jar picard.jar
```

(Running the program with no arguments or inputs will result in the usage documentation being printed to screen.)

Assuming we have an aligned read SAM file called ‘cancertest_aligned.sam,’ we first want to create a binary BAM file and then make sure the reads are sorted, for which we use *samtools view* and *samtools sort*, respectively:

```
$ samtools view -b cancertest_aligned.sam > cancertest_aligned.bam
$ samtools sort cancertest_aligned.bam cancertest_sorted.bam
```

Now we can call the *MarkDuplicates* program:

```
$ java -jar PicardTools/picardtools.jar MarkDuplicates
I=cancertest_sorted.bam O=cancertest_noDups.bam
M=duplicationMetrics REMOVE_DUPLICATES=TRUE ASSUME_SORTED=true
```

This is rather a long command call, in which we specify: an input file with the argument `I=`, an output filename with the argument `O=`, a file to create with additional information about the duplicates with the argument `M=`, the fact that we wish to remove the duplicates from the output is specified with `REMOVE_DUPLICATES=TRUE` and we pass an additional argument in the form `ASSUME_SORTED=true` as *PicardTools* has very stringent rules about its input files and if we have previously sorted the bam file with *Samtools* this is more likely to run smoothly!

An alternative approach when analyzing data downstream in R is using the *ShortRead* package, but at present there is not a satisfactory implementation of an export from R to SAM/BAM, so I would generally recommend trying to get the *PicardTools* functionality to work. Within R:

```
> library(ShortRead)
> cancertest.unfiltered <- readAligned("cancertest_aligned.bam",
+ type="BAM",filter=nFilter())
> cancertest.filtered <- cancertest.unfiltered[
+ !sruplicated(sread(cancertest.unfiltered)) & !is.na(position(
+ cancertest.unfiltered))]
```

This will create an *AlignedRead* object in the workspace called `cancertest.filtered` which doesn't include any duplicated reads.

11.5 Obtaining sequencing data from the SRA

The *Sequence Read Archive* (SRA), <https://www.ncbi.nlm.nih.gov/sra> is the NCBI's major repository for sequencing data, including from high-throughput sequencing experiments. In fact, the Gene Expression Omnibus links to SRA for retrieval of raw data from sequencing-based experiments whose processed data is described in a GEO record.

Sequencing data stored on SRA is highly-compressed, and therefore represents an efficient source for downloading the raw sequencing data, but requires special tools to decompress these records into usable *fastq* files. The *SRA-toolkit* provides functionality to retrieve data from the SRA, and to extract the retrieved records to Fastq files. Instructions for obtaining the SRAtoolkit can be found at: <https://github.com/ncbi/sra-tools/wiki/01.-Downloading-SRA-Toolkit>.

Note that precompiled binaries are available for two Linux distributions (CentOS and Ubuntu), for MacOS and for Windows. These should cover most examples, and can therefore be downloaded and used straight away. If these

won't work for you, try following the instructions for downloading the source code from the GitHub repository and then compiling⁴: <https://github.com/ncbi/sra-tools/wiki/Building-and-Installing-from-Source>.

Once installed, there are two tools we need to use in order to obtain data from the SRA. The first of those, *prefetch*, retrieves the data as compressed by the SRA. Let's say we wanted to obtain the ChIP-seq data from an ovarian cancer DNA sample, cited in [6]. The manuscript lists an SRA accession for the project of 'SRP016075'. Searching for the corresponding record on the SRA database, we find a list of the runs. Specifically look at the ChIP-seq with the H3K4me3 antibody, which can be found at: <https://www.ncbi.nlm.nih.gov/sra/SRX193578>.

From the corresponding web page, there is a *run* identifier listed: 'SRR600956'. To download the corresponding data, call the *prefetch* program (from the location into which it has been installed, which we will assume is from the current directory 'sra-tools/') with the run accession number as its only argument:

```
$ sra-tools/prefetch SRR600956
```

Then to obtain use this data to reconstruct the Fastq file(s), use the *fastq-dump* program:

```
$ sra-tools/fastq-dump SRR600956
```

The result should be a Fastq file which constitutes the raw data from the corresponding sequencing run. In [Chapter 14](#), there is an example of reconstructing two Fastq files from a paired-end sequencing run retrieved from SRA. Hopefully these examples show how publically-available sequencing data can be retrieved and processed in terms of quality control, filtering and alignment. Downstream processing differs for different applications of high-throughput sequencing, and so these will be covered in subsequent chapters.

Bibliography

- [1] W Huber *et al*, "Orchestrating high-throughput genomic analysis with Bioconductor," *Nature Methods* 12:115-121 (2015).
- [2] AM Bolger & M Lohse & B Usadel, "Trimmomatic: a flexible trimmer for Illumina sequence data," *Bioinformatics* 30(15):2114-2120 (2014).

⁴This is aimed at UNIX-based computers.

- [3] B Langmead *et al*, “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome,” *Genome Biology* 10:R25 (2009).
- [4] B Langmead & SL Salzberg, “Fast gapped-read alignment with Bowtie2,” *Nature Methods* 9:357-359 (2012).
- [5] H Li & R Durbin, “Fast and accurate short read alignment with Burrows-Wheeler transform,” *Bioinformatics* 25(14):1754-1760 (2009).
- [6] N Chapman-Rothe *et al*, “Chromatin H3K27me3/H3K4me3 histone marks define gene sets in high-grade serous ovarian cancer that distinguish malignant, tumour-sustaining and chemo-resistant ovarian tumour cells,” *Oncogene* 32(38):4586 (2013).



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

12

Genomic Sequence Profiling

12.1 Introduction

Representing what is probably the most obvious application of high-throughput sequencing technology, the DNA sequences from a population of cells can be identified. Working out a full genome sequence directly from a library of short reads is known as *de novo genome assembly*. This is computationally very challenging and requires an extremely large amount of sequence data, and is also rather rare, so we won't cover it here. Instead, by mapping sequence reads to an appropriate reference genome it is possible to use the information about where the reads don't match the reference in order to infer the correct sequence of the DNA being profiled. This approach to DNA profiling is known as *variant calling* and can be carried out at a genome-wide level, or for only targeted (captured) genomic regions, in largely the same way. Different types of genetic variation tend to be discovered and characterized in different ways, and so there are separate processing steps in order to discover SNVs, Indels and large structural variation. As with all the topics covered in this tutorial, there is a plethora of available software tools for carrying out these tasks, and the ones I describe represent just one or two examples of widely-used tools. These may be widely used on account of their software availability and reliability as much as anything to do with the algorithms underpinning them.

12.2 SNV: Single nucleotide variants

Far from sharing a single 'reference' genome, there is a great deal of variation in the exact sequence of DNA from different individuals, and even from different cells within the same individual as they age or are affected by disease. Most of the *characterized* human genetic variation exists in the form of single nucleotide variants (SNVs), which are single bases that vary from the majority allele (defined by populations that have been profiled through a number of international genome sequencing consortia). By identifying the set of SNVs for a DNA sample, it becomes possible to look for genetic causes of phenotypic

characteristics of the cells from which the DNA was obtained. When the DNA samples are blood from patients suffering from a given disease and a matched healthy cohort, one might be able to identify genetic variants predisposing individuals to risk of suffering from the disease. When the DNA samples are taken from a panel of tumour cell lines with differential response to a drug, it might be possible to identify genetic variants underpinning drug resistance or sensitivity. Of course, there will typically be a large number (potentially millions) of SNV differences between any pair of DNA samples from different genetic backgrounds (i.e. not clonally related), and so working out *which* of these seem most likely to contribute to the phenotype(s) of interest can be a challenge. In the next section we will cover some computational tools to attempt this, but as with so many things there is no wizardry that can take the place of a well-designed study.

There are many tools that can perform SNV identification, two of the most widely used seem to be *VarScan* [1] and the *GATK* [2]. The *GATK* is a very powerful toolkit containing functionality for performing a wide range of sequence data analysis tasks, with more information at <https://www.broadinstitute.org/gatk/>.

In the examples to follow we will use *VarScan* as I have found this to be the easiest to configure to run successfully on mapped sequence read files generated by different alignment tools on different machines. In order for *VarScan* to compute variant calls, we need to generate multi-way alignment files known as ‘pileups’. This can be done using the *mpileup* program from *samtools*. *Samtools* needs to be downloaded from <http://htslib.org> and then installed onto your computer before proceeding, following the instructions (for help, refer to the guide on using UNIX systems in [Chapter 3](#)). Once *samtools* has been installed, you can invoke the *mpileup* program to run on an aligned read BAM file.

As an example, say we have a file ‘cancertest_aligned.sam’ on our filesystem, which we have converted to BAM format, sorted and removed duplicate reads as in [Chapter 11](#). Next we need to generate the pileup file:

```
$ samtools mpileup -f cancertest.Reference.fa cancertest.noDups.bam  
> cancertest.pileup
```

From the pileup file we can now use *VarScan* to identify SNVs with evidence to support statistically significant occurrence relative to random sequencing errors. *VarScan* needs to be downloaded from <https://github.com/dkoboldt/varsan>. If you open this page in a browser, you should see a link to download the folder as a ZIP archive. Move the file you download (using the bash command `mv`) so that you have a file called ‘varsan.zip’ in your directory. Unzip the file:

```
$ unzip varsan.zip
```

This will create a directory which may have a long name, but you can rename the directory (again with the bash command `mv`) so that it is just called ‘varscan’. Now you can run the *VarScan* program through *Java* as it is supplied as a jar package (as was the case for *PicardTools*).

```
$ java -jar varscan/VarScan.v2.3.7.jar mpileup2snp pancertest.pileup
--output-vcf > pancertest.snvs.vcf
```

Note that the name of this *VarScan* *jar* file will vary depending on the version. You can see that I have used version 2.3.7, but be prepared to change this part of the command if you use a different version.

VarScan has a large set of configurable arguments which can alter the mode of operation, the type of information reported in the output, or the criteria for calling variants. The reader is referred to the *VarScan* manual for details, but you will see how we use some of these throughout this chapter.

You may notice that the pileup file is very large on the file system. You can see this if you use the shell to list files in your working directory, listing their size in a human-readable format:

```
$ ls -sh
```

We can avoid keeping this large file on the filesystem by using the Unix ‘pipe’ to pass the output from the *mpileup* command directly as input to the *varscan* command. This is particularly useful when running analyses on large numbers of files or on particularly large files.

```
$ samtools mpileup -f pancertest.Reference.fa pancertest.noDups.bam|java
-jar varscan/VarScan.v2.3.7.jar mpileup2snp --output-vcf >
pancertest2.vcf
```

12.3 Variant filtering and annotation

If we have a large set of variant calls from a sample, we may wish to attempt to narrow these variants down to a subset that are most likely to be making a phenotypic impact. A number of filtering and annotation steps can assist with this, and there is an array of computational tools to perform these tasks. Filtering involves removing those variants that don’t (or in some cases do) meet some specified criteria. Examples of common criteria include:

- number of supporting reads
- variant allele frequency

- presence in database of common variants
- non-synonymous variant in protein-coding gene
- conservation across multiple species
- overlap with identified regulatory region
- predicted phenotypic consequence

The first two of the listed criteria are more about distinguishing clonal variants from sequencing errors or rare sub-clonal variants that may affect too few cells to impact on the phenotype. These filters can typically be applied in the variant calling pipeline. If we are using VarScan to call SNVs from a pileup, we can specify a number of criteria using the following arguments: `--min-coverage`, `--min-reads2`, `--min-avg-qual`, `--min-var-freq` and `--p-value`. For example, if we wished to find SNVs from the file ‘cancertest.pileup’ with a minimum of 20 unique reads supporting each variant, a maximum p-value (estimating the probability of seeing that level of support for the variant by chance) of 0.001, and a minimum allele frequency of 0.1 (i.e. 1 in 10), we would adapt the example from Section 12.2 to the following:

```
$ java -jar varscan/VarScan.v2.3.7.jar mpileup2snp cancertest.pileup
--min-reads2 20 --min-var-freq 0.1 --p-value 1e-3
--output-vcf > cancertest.snvs_filtered.vcf
```

Any real DNA sample is likely to vary substantially in sequence from the appropriate organism’s reference genome, but many of those variations are likely to represent common genetic characteristics of different populations within a species. As such common sequence variations must be at least widely-tolerated (if not advantageous in certain environments) it is less likely that they are pathogenic. If we are studying variants associated with disease, as is common in human genetic studies, we may wish to flag SNVs that reflect common genetic variation. A number of initiatives have set out to characterize common human genetic variation, including the HapMap and 1000 Genomes Project, and there is a wealth of information in dbSNP [3] at <http://www.ncbi.nlm.nih.gov/SNP/>.

The dbSNP collection of common human genetic variation can be downloaded in VCF format straight onto a Linux computer as follows:

```
$ wget ftp://ftp.ncbi.nih.gov/snp/organisms/human_9606_b144_
GRCh37p13/VCF/common_all_20150605.vcf.gz
$ gunzip common_all_20150605.vcf.gz
```

This table of known common variants can be used to add annotation to a VCF file using the tool SnpSift [4]. SnpSift is implemented in *Java* and can

be downloaded as part of the SnpEff package from http://sourceforge.net/projects/snpeff/files/snpeff_latest_core.zip/download. The ZIP archive needs to be extracted:

```
$ unzip snpEff_latest_core.zip
```

Then a VCF file can be annotated with known variants from dbSNP as follows, with this example assuming we downloaded and unpacked the dbSNP common variants into the current directory, we have extracted the SnpEff ZIP file so that the SnpSift *jar* file is in the directory ‘snpEff,’ and in our current working directory we have the VCF file we wish to annotate, ‘cancertest_snvs_filtered.vcf’:

```
$ java -jar snpEff/SnpSift.jar annotate -v common_all_20150605.vcf
cancertest_snvs_filtered.vcf > cancertest_snvs_annotated.vcf
```

We can use the bash tool *grep* (regular expression search) to take this a step further and provide us a new VCF file that contains only those variants that were **not** in the dbSNP database:

```
$ grep -v rs cancertest_snvs_annotated.vcf > cancertest_snvs_novel.vcf
```

The *grep* command will find all rows in the input file (in this case ‘cancertest_snvs_annotated.vcf’) that contain the search pattern (in this case ‘rs’). When used with the argument *-v* it returns only those rows that don’t match the search pattern, which in this case will be all the entries in the VCF file that haven’t had a dbSNP ID (which begins ‘rs’) associated with them.

A set of novel, well-supported SNVs identified from a DNA sample could be extremely interesting for us, but there are likely to be a large number of non-functional variants (or at least those less likely to have a dramatic phenotypic impact). It may be that the most important variants to us are those which change a protein-coding gene sequence in such a way that is likely to cause an amino acid substitution. There are tools to help prioritize such variation, and one such example is the Ensembl Variant Effect Predictor, which is a very powerful tool implemented in *Perl* and can be downloaded from <http://www.ensembl.org/info/docs/tools/vep/script/index.html>. One of the useful things about Variant Effect Predictor is that it can add SIFT and PolyPhen functional effect predictions (along with many others) directly from the Linux shell, and can therefore be scripted to run automatically on a large number of files. In fact, the Variant Effect Predictor is available online with a graphical interface, at <http://www.ensembl.org/Multi/Tools/VEP>. But to install the command-line version of the tool, which will enable scripted processing of large numbers of input files, follow the instructions at <http://www.ensembl.org/info/docs/tools/vep/script/index.html>:


```
$ wget https://github.com/Ensembl/ensembl-tools/archive/
release/86.zip
$ unzip ensembl-tools-release-86.zip
$ cd ensembl-tools-release-86/scripts/variant_effect_predictor/
perl INSTALL.pl
```

To run the tool to generate usable output, use *perl* to run the program, remembering to specify the full path to the ‘ensembl-tools-release-86’ directory in which the program’s code now sits. For example:

```
$ perl ensembl-tools-release-86/scripts/variant_effect_predictor/
variant_effect_predictor.pl -i concertest_snvs_novel.vcf
-o concertest_snvs_novel_VEP.vcf --vcf --cache --merged
```

To include the SIFT and PolyPhen annotations:

```
--sift b --polyphen b
```

We may need to specify the port to the established database (mine is 3337):

```
--port 3337
```

Add the following argument to tell the annotation tool to use the GRCh37 version of the human genome (hg19):

```
--assembly GRCh37
```

The output of running this as one command will be another VCF file, but with a host of additional columns of annotation information for each variant, where available. We can use these in conjunction with the regular expression search tool *grep* in order to make a new VCF file with only the rows (i.e. variants) featuring a given text term (e.g. ‘deleterious’):

```
$ grep deleterious concertest_snvs_novel_VEP.vcf >
concertest_snvs_novelDeleterious_VEP.vcf
```

12.4 Indels: Short insertions and deletions

As well as single nucleotide variants, short insertions and deletions (or *Indels*) are another common form of genetic variation. While fundamentally very similar to SNVs in concept, the challenge of identifying such differences between a sequence represented by a set of short reads and a reference genome is

computationally different from that of identifying SNVs. Having said this, one advantage of the *VarScan* tool we used in the previous section to call SNVs is that the same program can be used to call Indels with only slight differences. Let's say we have the same pileup file that we used to call SNVs in Section 4.1, called 'cancertest.pileup.' We use the *mpileup2indel* mode of *VarScan* to generate another VCF file, this time detailing the indels:

```
$ java -jar varscan/VarScan.v2.3.7.jar mpileup2indel cancertest.pileup
--output-vcf > cancertest_indels.vcf
```

It should be reassuring to note that Indels in a VCF file can be filtered and annotated in the same way as SNVs, and so all of Section 12.3 applies to the output generated here. In fact, you may wish to merge SNVs and Indels together into one VCF file, which can be done using the *cat* shell command. For example, if we had SNVs in a file 'cancertest_snvs.vcf' and Indels in a file 'cancertest_indels.vcf' we could merge these as follows:

```
$ cat cancertest_snvs.vcf cancertest_indels.vcf
> cancertest_allvariants.vcf
```

12.5 SV: Structural variants

As well as the small-scale variation in genetic material across a population, a cell's DNA can be affected by large-scale structural rearrangements. This is particularly prevalent in cells which have compromised DNA damage checkpoint and repair pathways as in cancer, such that replicating DNA from distinct chromosomes can be aberrantly joined. These structural variants (SVs) manifest as large-scale deletions, duplications or translocations of DNA. An (extreme) illustration of the extent of structural variation that can arise in a cell is shown in [7] (Figure 12.1A in the paper), to show structural variants identified in the HeLa cell line. Identification of translocation breakpoints can suggest genes whose transcription may be disrupted, or may suggest potential fusion genes that could be created. Increased DNA copy number is typically linked to increased transcription, and therefore copy number variation (CNV) represents a means of genetic influence on phenotypes.

As with all the sequence data analysis tasks we are attempting in these tutorials, there are many tools available for use to identify SVs. One such tool is *DELLY* [8], which can be found online at <https://github.com/tobiasrausch/delly>. There are a number of Linux executable files, and source code bundles available at <https://github.com/tobiasrausch/delly/releases/>. As with other GitHub-hosted projects, you can download the folder as a ZIP

archive by clicking the ‘Download ZIP’ button. If you unzip this archive, you will need to install it by entering the directory (with `cd`) and running the following from the shell:

```
$ make all
```

Then Delly can be run on an input BAM file in a number of modes, prefixed with the argument `-t`, with each mode specifying a different type of SV to identify:

- DEL: deletions
- DUP: tandem duplications
- INV: inversions
- TRA: translocations

For example, if we wished to identify translocations from an aligned read BAM file ‘`cancertest_noDups.bam`’ we could execute the following (assuming that the Delly program was installed into a subdirectory named ‘`delly`’ which is in the current working directory, along with the reference genome file and the BAM file):

```
$ delly/src/delly -t TRA -o cancertest_translocations.vcf -g  
cancertest_Reference.fa cancertest_noDups.bam
```

By replacing `TRA` in the above command with any of `DEL`, `DUP` or `INV`, and probably renaming the output file (which follows the `-o` argument), we could find deletions, duplications or inversions, respectively. There are a number of other filters which can be applied to the output from Delly, with more details to be found on the GitHub site, but at least the above instructions should enable you to run the basics. Note, for identifying large-scale SVs you typically need a high-depth library of long, paired-end reads, and so these will generally be rather expensive sequencing experiments!

12.6 Making use of variant calls

The standard file format for specifying sequence variants is *VCF*. Details of the VCF format, in particular the values that the wide range of flags can take and what these mean, can be found at <http://samtools.github.io/hts-specs/VCFv4.1.pdf>. Following general information provided in a header, a ‘`.vcf`’ file will contain one row per called variant, listing: an ID for the variant, its chromosome and genomic co-ordinate, the variant allele (i.e. sequenced

base(s) that differ from reference genome), the corresponding reference allele (i.e. expected sequence), the variant allele frequency and a list of further annotations if provided (e.g. *RS* IDs from dbSNP). Although VCF is the standard file format, the majority of publically-available mutation information (from humans) do not come in VCF format. This is in part due to the fact that profiles of genetic variants from tissue samples could potentially identify patients who enrolled in studies under conditions of anonymity.

If we have a set of mutation calls (either a set of individual VCF files for each DNA sample, or a single file containing variants from a set of DNA samples) we will probably want to investigate whether any called variants are associated with some phenotypic characteristics of the cells from which the DNA was extracted (or clinical information corresponding to the patients from which DNA was extracted). As this is a non-trivial process, I will provide an example below.

Let's say we have downloaded the annotated mutation calls from the TCGA ovarian cancer study and we wish to look for associations between mutations in the *BRCA* genes and chemotherapy response. First, we can go to the ICGC¹ data portal to retrieve all the data in a convenient format. Open up the Data Portal at <http://icgc.org>, which should look like the screenshot in Fig. 12.1. Click on the 'DCC Data Releases' tab along the top, and you should be taken to a page looking like Fig. 12.2.

The screenshot shows the ICGC Data Portal interface. At the top, there is a navigation bar with five tabs: 'Cancer Projects', 'Advanced Search', 'Data Analysis', 'DCC Data Releases', and 'Data Repositories'. Below the navigation bar, the main content area is divided into two columns. The left column is titled 'Cancer genomics data sets visualization, analysis and download.' and contains a 'Quick Search' box with a 'Search' button. Below the search box, there is an example search query: 'e.g. BRAF, KRAS G12D, DO35100, MU7870, F1998, apoptosis, Cancer Gene Census, Imatinib, GO:0016049'. Below the search box, there is an 'Advanced Search' section with three buttons: 'By donors', 'By genes', and 'By mutations'. The right column is titled 'Data Release 27' and shows the date 'April 30th, 2018'. Below the title, there is a table of statistics:

Cancer projects	84
Cancer primary sites	22
Donor with molecular data in DCC	20,487
Total Donors	24,077
Simple somatic mutations	77,462,290

At the bottom of the right column, there is a 'Download Release' button.

FIGURE 12.1
Screenshot from ICGC data portal.

¹International Cancer Genome Consortium.

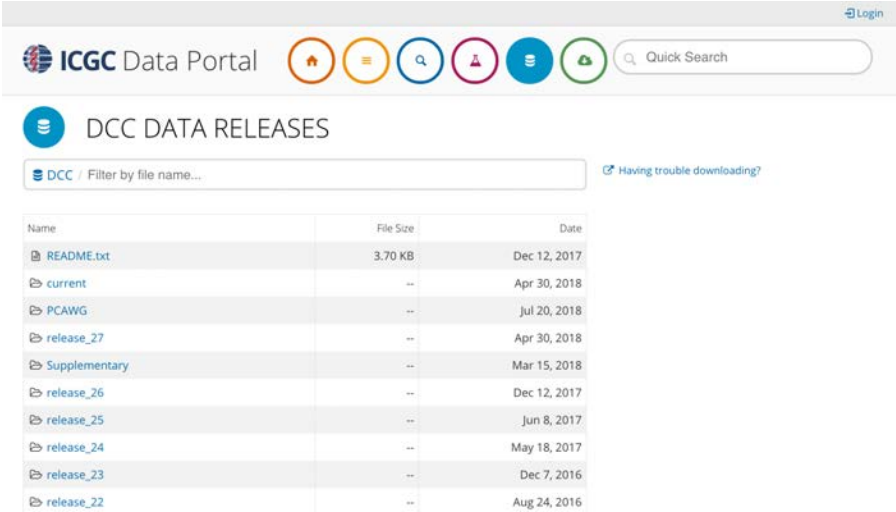


FIGURE 12.2
Screenshot from ICGC data portal.

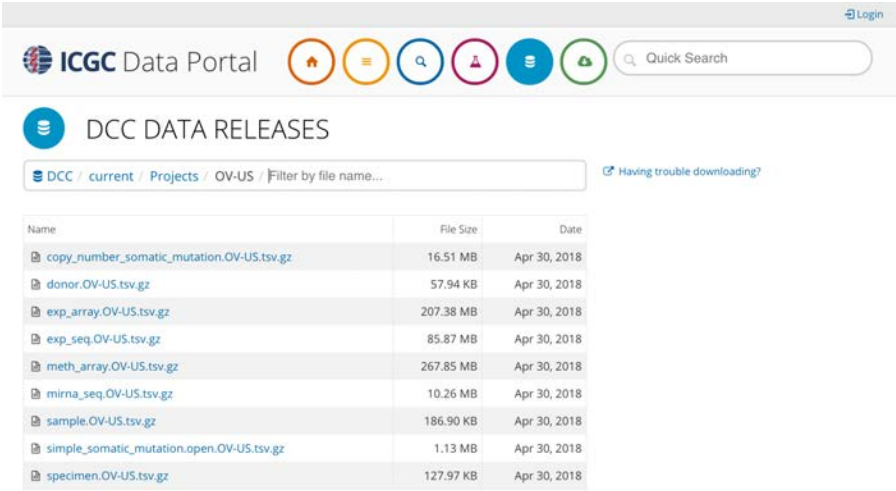


FIGURE 12.3
Screenshot from Genomic Data Commons data portal: TCGA Ovarian Cancer project.

Select the folder ‘Current’ and then you will see a list of all the projects for which data is available. Click on ‘OV-US (TCGA)’, which should take you to the TCGA Ovarian Cancer project page (Fig. 12.3). Download and unzip the files ‘simple_somatic_mutations.open.OV-US.tsv.gz’ and ‘donor.OV-US.tsv.gz’.

Our downstream analysis can be performed using R, although we assume that the mutation and clinical data files are all in the working directory from which we started R.

First, we want to find which patients have both variant calls and clinical information downloaded. We do this by loading the individual tables, then matching up the donor IDs from the two tables:

```
> snv.table <- read.table("simple_somatic_mutation.open.OV-US.tsv",
+ sep="\t",head=TRUE,row.names=NULL)
```

This creates a data frame called *snv.table*, containing the mutation calls and associated information. This table has 1 row for each mutation, indicating its genomic location and which patient the mutation call came from. So we will first want to create a vector listing all the patients for which we have mutation data:

```
> snv.patients <- unique(as.character(snv.table$icgc_donor_id))
```

Now we want to read in the clinical data table:

```
> clin.data <- read.table("donor.OV-US.tsv",sep="\t",
+ head=TRUE,row.names=1)
```

This creates a data frame called *clin.data*, containing the clinical follow-up information from the ovarian cancer patients, with the patient IDs as the row names. To match up the mutation data to the patient data, we need to check which patients have both clinical and mutation data:

```
> both.patients <- intersect(snv.patients,rownames(clin.data))
> length(both.patients)
[1] 118
```

We can see that there are 118 patients with both mutation calls and clinical information. This is in fact because there are only mutation calls from 118 patients included in the *snv.file*:

```
> length(snv.patients)
[1] 118
```

So we now have a table of mutations and a set of clinical characteristics, but how can we test for associations between the two? First of all, we can try matching the mutations to genes, to see if *any* mutations at given genes associate with treatment outcomes. You will see from the mutation file we have downloaded that this task has already been done for us, with columns of

the table ‘gene.affected’ and ‘transcript.affected’. Run through the example anyway because a typical VCF file may not have such information. We can approach this task in a similar way to the copy-number segmentation table in [Chapter 10](#). But we’ll add in an extra step, because if we split the one table of variant calls into a list of tables (one table per patient), then the code will be very similar to what we’d use if we had a separate VCF file for each patient. This will be the case if you have followed the earlier steps in this chapter.

```
> mutation.tables <- lapply(both.patients,function(x)snv.table[
+ which(snv.table$icgc_donor_id==x),])
```

In this command we use the `lapply` function to make a list, applying a function (with a single argument `x`) to each of the characters in `both.patients` in turn, so that each element in the list is a table with the rows of `snv.table` for which the patients we previously extracted from each row match the particular barcode which the function is being applied to. The result is a list with 118 elements, each containing the mutation information corresponding to the patient matching the corresponding barcode from `both.patients`. The reason I have structured this task in such a way is so that if you were to have a set of VCF files, you would end up with a similar list of tables if you were to read the VCF files into R in a batch, by running the following commands (assumes you have VCF files in your current working directory):

```
> vcf.fileNames <- list.files(pattern=".vcf")
```

This creates a character vector named `vcf.fileNames` with one entry for each file in the current working directory that contains the pattern ‘.vcf’.

```
> mutation.tables <- lapply(vcf.fileNames,read.table,sep="\t")
```

This creates a list called `mutation.tables` for which each element is a data frame containing the information in the VCF file of the corresponding element of `vcf.fileNames`.

To map the variants to genes, we can follow an approach similar to that taken in [Section 10.3.4](#) on SNP Arrays. This uses the *biomaRt* package to bring in data from Ensembl to R. First of all, we need to obtain the genomic co-ordinates for each gene:

```
> library(biomaRt)
> ensembl <- useMart(host="feb2014.archive.ensembl.org",biomart=
+ "ENSEMBL_MART_ENSEMBL",dataset="hsapiens_gene_ensembl")
> geneInfo <- getBM(attributes=c("hgnc_symbol",
+ "chromosome_name","start_position","end_position"),mart=ensembl)
```

The `getBM` function has created a data frame, which we have called `geneInfo`, which lists each Ensembl-annotated gene's official gene symbol along with its chromosome, start and end positions. It will be more convenient for us if we remove the Ensembl-annotated genes that don't have an official gene symbol:

```
> geneInfo <- geneInfo[which(!geneInfo$hgnc_symbol==""),]
```

And we also may wish to remove duplicate entries for different gene definitions with the same symbol:

```
> geneInfo <- geneInfo[!duplicated(geneInfo$hgnc_symbol),]
```

Now we can create a matrix indicating the mutation status of each gene, with a row for each gene and a column for each patient. A convenient way of encoding this information is with a value of 0 in an entry indicating the corresponding patient does not have a variant mapping to the corresponding gene. Create a matrix of the appropriate size with all values equal to 0, then for each patient fill in a 1 for any genes that are covered by variants:

```
> mut.genes <- array(0,dim=c(nrow(geneInfo),length(mutation.tables)))
> rownames(mut.genes) <- geneInfo$hgnc_symbol
> colnames(mut.genes) <- both.patients
> for(i in 1:length(both.patients)){
+ this.variant.genes <- c()
+ for(j in 1:nrow(mutation.tables[[i]])){
+ this.variant.genes <- c(this.variant.genes,geneInfo$hgnc_symbol[
+ which(geneInfo$chromosome_name==mutation.tables[[i]][j,"chromosome"]
+ & geneInfo$start_position<mutation.tables[[i]][j,"chromosome_end"]
+ & geneInfo$end_position>mutation.tables[[i]][j,"chromosome_start"])]))}
+ mut.genes[this.variant.genes,i] <- 1}
```

This rather intimidating-looking loop is actually relatively straightforward, running two nested `for` loops. The first loop goes through each patient ID, first creates an empty vector that will ultimately list genes that have variants mapping from the corresponding patient's sample, then initiates the second loop. This second loop goes through each row in the patient's `mutation.tables` entry, and adds any genes that span the corresponding variant co-ordinates to the vector `this.variant.genes`. Finally, the values in the `mut.genes` matrix corresponding to those genes are set to 1.

Some brief exploration tells us a bit about this dataset:

```
> sum(mut.genes)
[1] 9921
```


Here we see that there are 9,921 mutated genes in total, and we can break those down into numbers for each patient with `colSums`:

```
> range(colSums(mut.genes))
[1] 16 253
```

So this varies between a minimum of 16 mutated genes for one patient, up to a maximum of 253 for another.

We create a histogram of the mutation counts per gene, as follows:

```
> hist(rowSums(mut.genes))
```

The `hist` function plots a histogram, the result should be as shown in [Fig. 12.4](#).

There will be a large number of genes that are never mutated in this collection of tumour samples, so we might as well remove these from the table:

```
mut.genes <- mut.genes[rowSums(mut.genes)>0,]
```

Now we can check the new dimensions of the `mut.genes` matrix:

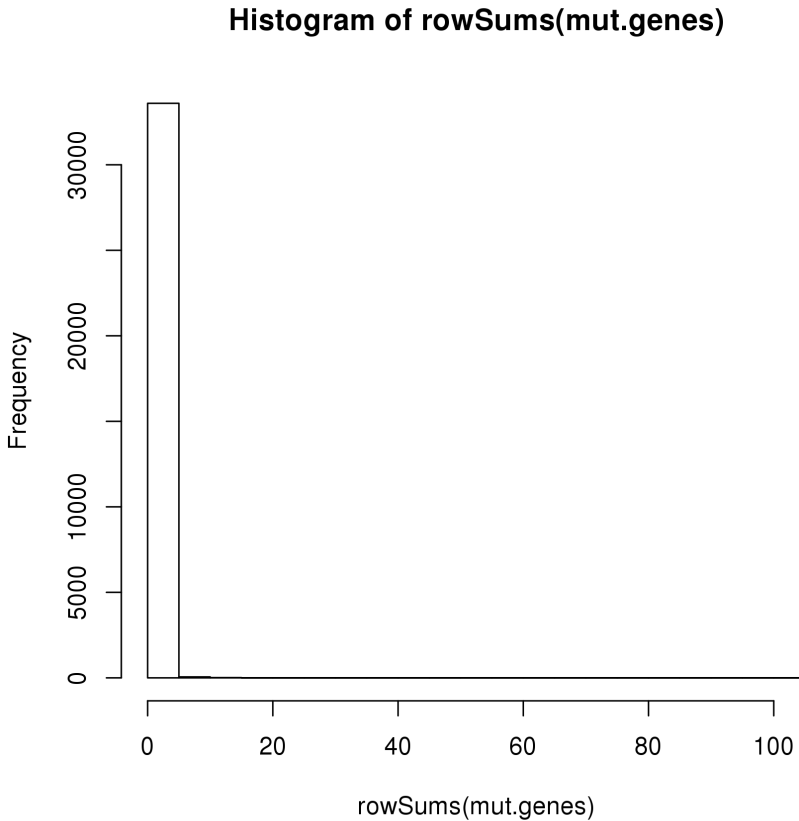
```
dim(mut.genes)
[1] 6519 118
```

We may want to test to see if there are individual genes for which a mutation is associated with worse outcomes. One way of representing this could be to look at the overall survival times, in the context of the vital status indicators. For this we need to create a special type of object in R, using the *survival* package. The survival variables need to have one vector specifying the ‘time-to-event’ and another vector indicating whether or not the event occurred. In this case, time-to-event is `donor_survival_time` if the patient’s vital status is deceased, but it is `donor_interval_of_last_followup` if the vital status is alive. We will first create a vector called `surv.time`, containing the survival time (or an NA if the patient was still alive at last followup):

```
> surv.time <- clin.data[both.patients,"donor_survival_time"]
```

Then create a vector with the indicator stating whether or not the event (i.e. death) had occurred within the observation window.

```
> surv.event <- rep(NA,length(both.patients))
> surv.event[which(clin.data[both.patients,"donor_vital_status"]
+ == "deceased")] <- 1
```

**FIGURE 12.4**

Histogram of mutation counts for each gene in TCGA ovarian cancer cohort with validated mutations (118 patients).

```
> surv.event[which(clin.data[both.patients,"donor_vital_status"]  
+ == "alive")] <- 0
```

We can use the event indicator as an index to specify which patients will need to replace missing values from the `donor_survival_time` column of the clinical data table:

```
> surv.time[which(surv.event==0)] <- clin.data[both.patients,  
+ "donor_interval_of_last_followup"][which(surv.event==0)]
```

Finally, load the *survival* package and create a survival object with the `Surv` function”

```
> library(survival)
> patient.os <- Surv(surv.time,surv.event)
```

We can now use the `survdiff` function within the *survival* package to test the statistical significance of the separation between the survival curves for patients with and without a mutation in a given gene. This uses a χ^2 test to compare the counts of events (i.e. deaths) that occur in each group of patients, scaled by the sum of observation time across all patients in the corresponding group. The p-value returned by this hypothesis represents the probability that a random allocation of events to patients would result in so uneven a distribution of events across the two groups, taking into account the differences in patient follow-up times.

```
> survdiff(patient.os ~ mut.genes["PIK3CA",])
Call:
survdiff(formula = patient.os ~ mut.genes["PIK3CA", ])
N Observed Expected (O-E) $\hat{2}$ /E (O-E) $\hat{2}$ /V
mut.genes["PIK3CA", ]=0 112 67 64.87 0.07 1.18
mut.genes["PIK3CA", ]=1 6 2 4.13 1.10 1.18
```

Chisq= 1.2 on 1 degrees of freedom, p= 0.3

We have seen the result for an individual gene, but what about testing all genes? It probably isn’t worth considering the differences between curves when only a few patients have mutations, but we need to be pragmatic bearing in mind the relatively small cohort size (118) and low prevalence of individual point mutations in ovarian cancer. So we can apply a proportional hazards regression model (this is another approach to test the impact of a variable, or set of variables, on survival outcomes) to the rows of the `mut.genes` table which contain more than four 1s:

```
coxph.pvals <- apply(mut.genes[which(rowSums(mut.genes)>4),],
+ MARGIN=1,function(x)summary(coxph(patient.os ~ x))$coefficients[1,5])
```

This command uses the `apply` function to apply a custom function (extracting the p-value from the Cox proportional hazards regression model fit) to each row of the `mut.genes` table in turn. We can then sort the values to find the genes with the most significant difference in patient survival:

```
> sort(coxph.pvals,decreasing=F)[1:3]
MY05C BRCA2 MUC17
0.01979490 0.02729627 0.02745394
```

If we were being strict about this we should adjust the p-values to take into account the fact that we have run multiple hypothesis tests:

```
> coxph.adjusted <- p.adjust(coxph.pvals,method="BH")
```

Now we see if any of the individual model fits remains significant when we take into account the number of hypothesis tests we carried out:

```
> sort(coxph.adjusted,decreasing=F)
```

In a real scenario, we would almost certainly want to compute the hazard ratios of the models, and the confidence intervals for the fitted hazard ratios. This would be very similar to the example presented in [Chapter 7](#), so we leave that here as an exercise for the reader.

Furthermore, the fact that we had so few observations to draw on mean that any association would be tenuous at best! But, if we had a larger cohort harbouring a mutation, we may wish to visualize the difference in survival curves in Kaplan-Meier plots. Again this is similar to examples shown in [Chapter 8](#), using gene expression data.

```
plot(survfit(patient.os ~ mut.genes["BRCA2"],)  
+ ,lty=c(2,1),col=c("black","red"))
```

This command, using the function `survfit`, specifies the use of a dashed black line for the group with values = 0 (i.e. patients without a validated mutation in the BRCA2 gene) and a solid red line for the group with values = 1. The result should look like that shown in [Fig. 12.5](#).

As an extension to this topic, we may wish to see if combinations of mutations are associated with outcomes, given that individual mutations may not. By using penalized multivariate regression, we can use this approach to find the variables (and combinations of variables) which have the biggest impact on the outcome. This is sufficiently more complex that it would require further study, and so we will not be covering it in this book, but I mention the concept as something that you may wish to explore if the simpler approaches do not yield anything. The basic principle is that by allowing the model to include a large number of variables, but penalizing the scores of model fit for each additional variable included, a balance can be struck between finding subtle patterns in datasets and over-fitting to a small set of observations. You can find more about the *glmnet* package which we use for penalized multivariate regression in R here:

https://cran.r-project.org/web/packages/glmnet/vignettes/glmnet_beta.html

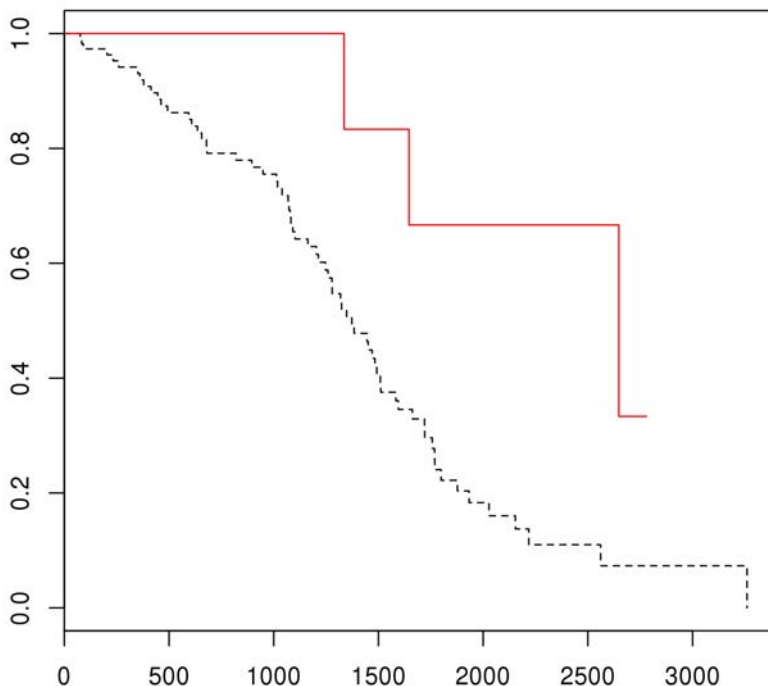


FIGURE 12.5

Kaplan-Meier plot showing overall survival curves for patients in the TCGA ovarian cancer cohort with a BRCA2 mutation (solid red line), and patients without a validated BRCA2 mutation (dashed black line). X-axis gives survival time in days; y-axis gives proportion of patients in the corresponding group surviving beyond the specified time.

12.7 Summary

This chapter covered variant calling, filtering and annotation from high-throughput sequencing data that has been preprocessed to the point of non-duplicated reads mapped to a reference genome (as described in the previous chapter). The downstream analysis section hopefully illustrates with a real example how the resulting variant calls can be used for clinical research. Of course, HTS data can also be used to call genotypes for known polymorphic sites, in which case associations with phenotypes can be tested as in [Chapter 10](#). This approach could also be applied to contingency tables produced by separating samples into two groups based on the presence or absence of specific variants (or any variant mapping to a given gene). There are also

specific technical errors that can be introduced through targeted sequencing approaches, but as a starter, the methods illustrated in this chapter should work for most applications.

Bibliography

- [1] DC Koboldt *et al*, “VarScan: variant detection in massively parallel sequencing of individual and pooled samples,” *Bioinformatics* 25(17):2283-2285 (2009).
- [2] MA DePristo *et al*, “A framework for variation discovery and genotyping using next-generation DNA sequencing data,” *Nature Genetics* 43:491-498 (2011).
- [3] ST Sherry *et al*, “dbSNP: the NCBI database of genetic variation,” *Nucleic Acids Research* 29(1):308-311 (2001).
- [4] P Cingolani *et al*, “Using *Drosophila melanogaster* as a model for genotoxic chemical mutational studies with a new program, SnpSift,” *Frontiers in Genetics* 3:35 (2012).
- [5] P Kumar & S Henikoff & PC Ng, “Predicting the effects of coding non-synonymous variants on protein function using the SIFT algorithm,” *Nature Protocols* 4(7):1073-1081 (2009).
- [6] K Wang & M Li & H Hakonarson, “ANNOVAR: Functional annotation of genetic variants from next-generation sequencing data,” *Nucleic Acids Research* 38:e164 (2010).
- [7] JJM Landry *et al*, “The genomic and transcriptomic landscape of a HeLa cell line,” *G3* 3(8):1213-1224 (2013).
- [8] T Rausch *et al*, “DELLY: Structural variant discovery by integrated paired-end and split-read analysis,” *Bioinformatics* 28(18):i333-i339 (2012).



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

13

ChIP-seq

13.1 Introduction

The capacity for high-throughput sequencing technology to go some way towards relative quantification of different DNA sequences in a library can be used identify genomic locations of certain chemical modifications or DNA-binding elements. The general principal of a chromatin immunoprecipitation (ChIP) sequencing assay is to enrich a library through purification with an antibody that specifically binds to a target transcription factor (or DNA feature) of interest. By counting the number of reads that map to each genomic position, and estimating how unlikely it would be to see so many reads in a given position purely by chance, regions of statistically significant enrichment over background can be obtained. There are a few subtleties in the process which can result in technical artifacts biasing the results of a ChIP-seq experiment, and so this tutorial will go through some of the means of attempting to take these into account and produce as accurate and as useful results as possible.

13.2 Cross-correlation

Initial quality control and alignment of ChIP-seq data should be carried out as with any high-throughput sequencing data. Following alignment, an additional assessment strategy can be followed, exploiting the expected distribution of reads that comes from sequencing either end of clusters of DNA fragments centered on the structure recognized by the antibody used for the IP. Note: this is mostly included for illustrative purposes, it's unlikely that this would be performed routinely. In fact, the MACS peak-calling program incorporates a form of this analysis! Across an entire ChIP-seq library, if the IP procedure results in isolation of many copies (from different cells) of the same DNA fragments (those linked to the antibody's target), then the count of overlapping reads (referred to as *coverage*) from one strand ought to be correlated with the coverage on the other strand, offset by the average fragment length. An excellent description of cross-correlation in ChIP-seq can be found in [1]. We

can inspect the cross-correlation of a ChIP-seq library using R, although it is worth bearing in mind that this is computationally-intensive work and so will take a long time! Using the *ShortRead* package we can first read in an aligned read BAM file, let's say called 'ChIPseq_aligned.bam':

```
> library(ShortRead)
> ChIP.reads <- readGAlignments("ChIPseq_aligned.bam",
param=ScanBamParam())
```

It simplifies things to compute the cross-correlations from different chromosomes separately, and to speed things up in our example we may wish to keep only data from a few chromosomes. Therefore we can define the set of chromosomes we want to keep, and keep only these reads:

```
> chrs <- c("chr1","chr2")
```

Here we have created a vector `chrs` that includes "chr1" and "chr2". We can keep only reads from these chromosomes with the following:

```
> ChIP.reads <- ChIP.reads[chromosome(ChIP.reads) %in% chrs]
```

We need to separate reads from each strand:

```
> ChIPreads.plus <- ChIP.reads[strand(ChIP.reads)=="+"]
> ChIPreads.minus <- ChIP.reads[strand(ChIP.reads)=="-"]
```

We can then use the `coverage` function provided in the *IRanges* package¹ which calculates coverage for *readAligned* objects in R:

```
> cov.plus <- coverage(ChIPreads.plus)
> cov.minus <- coverage(ChIPreads.minus)
```

Cross-correlation is the *offset* correlation between the coverage on the plus and minus strands. We therefore need to calculate this correlation for different values of the offset, which we do not know in advance (although we should be able to make a pretty good guess due to size selection in the library prep!). We therefore pick a range of offsets from zero to the upper end of what we might expect of DNA fragment sizes, which we can create using the `seq` function in R:

```
> offsets <- seq(from=0,to=200,length=10)
```

A convenient way to store the results of this analysis in R is a matrix, with a row for each offset and a column for each chromosome. Then it will be

¹This package is included with the *ShortRead* packages in Bioconductor.

straightforward to perform mathematical operations and generate plots from the different offsets together:

```
> crosscors <- array(NA,dim=c(length(offsets),length(chrs)))
```

We now can set up a loop running through each chromosome in turn, and a loop inside the first running through each offset in turn, computing the correlation between the read coverage counts for each strand and entering the result in the appropriate cell within the matrix:

```
> for(i in 1:length(chrs)){
+   maxlength <- min(c(length(cov.plus[[chrs[i]]]),length(cov.minus
+   [[chrs[i]]])))
+   for(j in 1:length(offsets)){
+     crosscors[j,i] <- cor(x=cov.plus[[chrs[i]]][c(1:[maxlength-
+     offsets[j]])],
+     y=cov.minus[[chrs[i]]][c(offsets[j]:[maxlength-1])])})}
```

Once again this may look pretty formidable, but it is fairly simple when you break it down. The first loop sets up a counter *i* to keep track of which chromosome we are looking at, then the second loop sets up a counter *j* to keep track of which offset we are using. We need to consider the length of the coverage object for the given chromosome, hence using the minimum value from either the plus or the minus strand. Within the second loop the cross-correlation is calculated using the `cor` function, passing two vectors as the arguments *x* and *y*, one of which is the coverage values on the plus strand from position 1 to the *j*th value of `offsets` subtracted from the maximum length value `maxlength`. The second vector argument is the coverage values on the minus strand from the *j*th value of `offsets` up to the maximum length value `maxlength` minus 1. This way the two vectors are always the same size, regardless of the value of offset used, and never extend beyond the calculated coverage values for the corresponding chromosome.

With these values calculated, we may wish to plot the averages across all computed chromosomes for each offset, so we can compare these. To create such a plot is relatively simple:

```
> plot(x=offsets,y=rowMeans(crosscors),type="l",ylab="cross-
+ correlation",xlab="offset")
```

The resulting plot should look like the one shown in [Fig. 13.1](#). The peak in the plot gives the average fragment length, which in this case is probably around 150 bp, but only if we assume that relatively high correlations at low offsets indicate technical issues. An important point to consider is that there is quite often a “phantom peak” at a very low offset, which can be

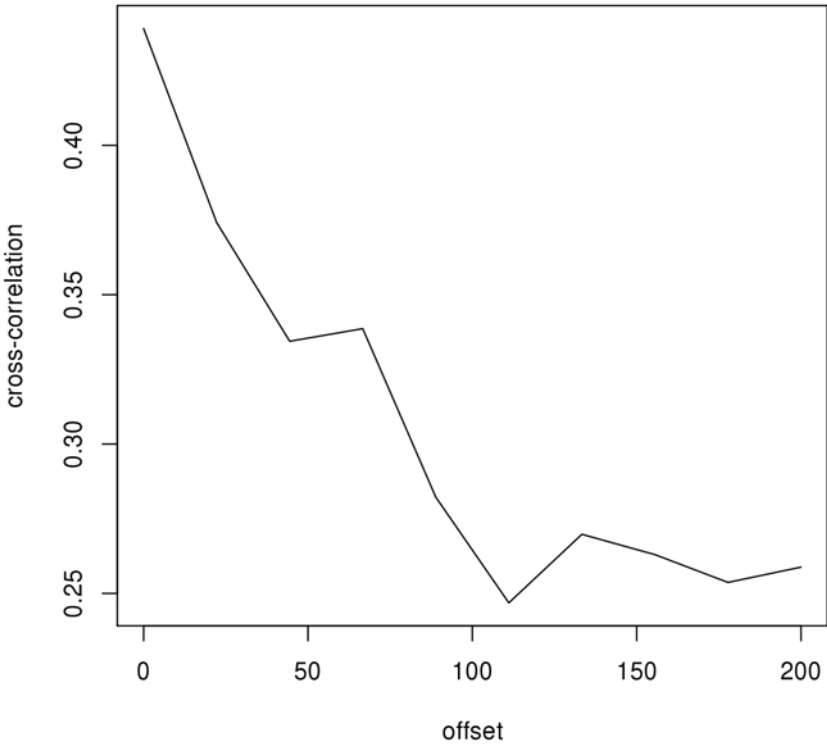


FIGURE 13.1

Plot of cross-correlation values from ChIP-seq library at different values of offset between plus and minus strands.

seen as the cross-correlation due to non-IP fragment enrichment. The ratio between the fragment-length peak cross-correlation and the phantom-peak cross-correlation has been proposed as an indicator of how successfully the IP procedure has been represented in the sequenced library [1]. Alternatively, the tightness of the cross-correlation peak around the expected fragment length (and the value of the cross-correlation) can also be used as an indicator of the IP enrichment in the library. If the cross-correlation peak is very broad or centred a lot lower than you expect the fragment length to have been based on your library prep protocol, these are warning signs that the dataset you have may not be very good quality. The results obtained on our dataset suggest that there may be some issues: the next section explains one possible cause and gives a solution.

13.3 Filtering blacklisted reads

Some regions of the genome are easier to sequence and to map than others. The extent of this variability means that some regions of the genome are in fact likely to dominate ChIP-seq library signals even when they are not represented with the feature that the antibody used for the IP is supposed to recognize [1]. For such super-alignable regions, which are helpfully well-characterized in some genomes, it makes sense to remove reads mapping to these regions from a ChIP-seq library prior to downstream analysis. You may find that after removing such ‘blacklisted’ regions and recalculating cross-correlations, that the resulting plots look considerably better. The process of removing reads from a library is actually relatively simple, provided we have a BED file containing the blacklisted genomic regions and a BAM file containing the aligned reads from the library we wish to analyze. You can download the blacklisted regions bedfile from <http://hgdownload.cse.ucsc.edu/goldenpath/hg19/encodeDCC/wgEncodeMapability>. Alternatively, on a UNIX-based system you can get the file directly from the bash shell:

```
$ wget http://hgdownload.cse.ucsc.edu/goldenpath/hg19/encodeDCC/wgEncodeMapability/wgEncodeDukeMapabilityRegionsExcludable.bed.gz
$ gunzip wgEncodeDukeMapabilityRegionsExcludable.bed.gz
```

Then we can invoke the *bedtools* program (assuming we have it installed) from the shell with the argument *intersect -v*:

```
$ bedtools intersect -v wgEncodeDukeMapabilityRegionsExcludable.bed
ChIPseq_aligned.bam > ChIPseq_aligned_DACfiltered.bam
```

Then we can use the file ‘ChIPseq_aligned_DACfiltered.bam’ in our downstream analysis. In a real study, the cross-correlation analysis would be expected to have much more positive results after performing this filtering!

13.4 Peak calling

The main objective with ChIP-seq experiments is to identify genomic regions that contain the feature recognized by the antibody. That is, we wish to find the regions that are significantly enriched during the IP process. There is a large body of work describing ways this can be performed, but in this tutorial we will stick with one of the most widely used programmes for so-called “peak-finding”, known as *MACS* [2]. There are now two main version of MACS, with MACS2 being what is probably the most widely-used ChIP-seq peak finder. MACS2 is implemented in the Python programming language, and

specifically requires Python version 2.7 to run. The easiest way to install MACS2 is through Python's *pip* package manager:

```
$ pip install MACS2
```

If this doesn't work, then you can download a *zip* archive of the source code from: <https://github.com/taoliu/MACS/archive/master.zip>.

Once this archive is unzipped, in the resulting directory run the following command:

```
$ python2.7 setup.py install
```

If you have any issues with this Python-based installation, the original version of MACS can still be downloaded and installed as follows:

```
$ wget https://github.com/downloads/taoliu/MACS/MACS-1.4.2-1.tar.gz
$ gunzip MACS-1.4.2-1.tar.gz
$ tar -xf MACS-1.4.2-1.tar
```

MACS2 can perform a number of different tasks, but the one we are most interested in is the *callpeak* mode. This can run using just a single input BAM file, if there is no control library against which to compare the ChIP aligned reads. For this scenario, we could find peaks in the file "ChIPseq_aligned_DACfiltered.bam" as follows:

```
$ macs2 callpeak -t ChIPseq_aligned_DACfiltered.bam
-n ChIPseq_noControl
```

However, we will typically have an input DNA (no antibody) or IgG (control antibody) library against which to compare the ChIP library. Let's say we also had a file called "InputDNA_aligned_DACfiltered.bam" which has been preprocessed, aligned and filtered in exactly the same way as the ChIP reads. Then we could find IP-enriched peaks using MACS2 as follows:

```
$ macs2 callpeak -t ChIPseq_aligned_DACfiltered.bam
-c InputDNA_aligned_DACfiltered.bam -n ChIPseq_withControl
```

The result of peak-calling will typically be stored as BED files, which are text files with one row for each peak, containing at the very least the chromosome, start and end positions for the peak. BED files generated by MACS will also contain a peak ID, an enrichment p-value derived from their statistical model of ChIP coverage relative to the genomic distribution (and possibly the enrichment in the same region of the input library as well), and a read count for the peak. The outputs we will typically use are in files with the extension ".narrowPeak", and we may just wish to extract the co-ordinates

for the peak regions to save these as “.bed” files. For the previous example, we could extract the first three columns using the UNIX command `cut`:

```
$ cut -f 1,2,3 ChIPseq_withControl_peaks.narrowPeak > ChIPseq_peaks.bed
```

Now there is a file on the system which contains the genomic co-ordinates of the regions enriched for reads in the ChIP-pulldown library relative to the genomic DNA control. Although for many purposes, the “narrowPeak” file could be used interchangeably. These files might be sufficient for downstream analysis and interpretation of results, but it will often help to map the enriched regions to genomic features of interest.

13.5 Peak annotation

Having identified regions enriched during the IP process, it is normal to wish to associate these to known genomic features of interest. Most commonly these are genes, although they could be regulatory features, genetic variants, or identified non-coding transcripts (etc). There are stand-alone tools for annotating peaks with genomic features, but we can use the same strategy within R that was used in [Chapter 10](#) to map the variants to genes, and that taken in [Section 4.3.4](#) on SNP Arrays. This approach uses the *biomaRt* package to bring in gene annotation data from Ensembl to R. First of all, we need to obtain the genomic co-ordinates for each gene:

```
> library(biomaRt)
> ensembl <- useMart(host="feb2014.archive.ensembl.org",biomart=
+ "ENSEMBL_MART_ENSEMBL",dataset="hsapiens_gene_ensembl")
> geneInfo <- getBM(attributes=c("hgnc_symbol",
+ "chromosome_name","start_position","end_position","strand")
+ ,mart=ensembl)
```

The `getBM` function has created a data frame, which we have called `geneInfo`, which lists each Ensembl-annotated gene’s official gene symbol along with its chromosome, start and end positions, and the strand. We use the ‘feb2014’ archive of ENSEMBL as that is the most recent version still using the hg19 release of the human genome. It will be more convenient for us if we remove the Ensembl-annotated genes that don’t have an official gene symbol:

```
> geneInfo <- geneInfo[which(!geneInfo$hgnc_symbol==""),]
```

And we also may wish to remove duplicate entries for different gene definitions with the same symbol:

```
> geneInfo <- geneInfo[!duplicated(geneInfo$hgnc_symbol),]
```

It is obviously essential to have the peak information, which we can read into R:

```
> ChIP.peaks <- read.table("ChIPseq.peaks.bed",sep='^',head=F)
> colnames(ChIP.peaks) <- c("Chromosome","Start_position","End_position")
```

With the data loaded into the workspace we could proceed as in the other examples, associating genes with a peak if the peak lies within the gene itself.

```
> marked.genes <- c()
+ for(i in 1:nrow(ChIP.peaks)){
+   marked.genes <- c(marked.genes,geneInfo$hgnc_symbol[
+     which(geneInfo$chromosome_name==ChIP.peaks[j,"Chromosome"]
+     & geneInfo$start_position<ChIP.peaks[i,"End_position"]
+     & geneInfo$end_position>ChIP.peaks[i,"Start_position"])]))
+   marked.genes <- unique(marked.genes)
```

Then the vector `marked.genes` lists all the genes containing peaks. Of course, it would also be possible to use any set of regions of interest, it wouldn't have to be the gene annotation information from Ensembl. All you would need to do is read in the coordinates of the named features of interest, as for the peaks here, and then change `$hgnc_symbol` to the appropriate column name containing the names of the genomic features.

However, as most ChIP-seq data deals with regulatory regions, we might instead wish to look in the regions upstream of each TSS. For this we will start by creating a new annotation table, relabelling each gene as starting at its TSS-2000 and ending at its TSS:

```
> geneInfo.TSS <- geneInfo
> geneInfo.TSS[which(geneInfo.TSS$strand==1),"end_position"] <-
+ geneInfo.TSS[which(geneInfo.TSS$strand==1),"start_position"]
> geneInfo.TSS[which(geneInfo.TSS$strand==1),"start_position"] <-
+ geneInfo.TSS[which(geneInfo.TSS$strand==1),"start_position"]-2000
> geneInfo.TSS[which(geneInfo.TSS$strand==-1),"start_position"] <-
+ geneInfo.TSS[which(geneInfo.TSS$strand==-1),"end_position"]
> geneInfo.TSS[which(geneInfo.TSS$strand==-1),"end_position"] <-
+ geneInfo.TSS[which(geneInfo.TSS$strand==-1),"end_position"]+2000
> marked.promoters <- c()
+ for(i in 1:nrow(ChIP.peaks)){
+   marked.promoters <- c(marked.promoters,geneInfo$hgnc_symbol[
+     which(geneInfo$chromosome_name==ChIP.peaks[j,"Chromosome"]
+     & geneInfo$start_position<ChIP.peaks[i,"End_position"]
+     & geneInfo$end_position>ChIP.peaks[i,"Start_position"])]))
+   marked.promoters <- unique(marked.promoters)
```

With a vector of gene symbols corresponding to those for which a ChIP-seq peak was detected in the promoter region, this may be sufficient for our needs. Such a list of genes could be evaluated for over-representation of functionally annotated genesets (e.g. as in [Chapter 6](#)), or they could in fact be used as a gene-set with which to test another dataset for systematic enrichment or over-representation.

13.6 Quantitative comparisons of ChIP-seq libraries

In some situations, the actual *levels* of enrichment of given genomic regions across different ChIP libraries might be of interest. This section outlines how counting the number of reads mapping to regions of interest, and then normalizing for the *sequencing depth*, can create quantitative enrichment matrices that can then be analyzed much like any other (see for example, [Chapter 5](#)). In order to do this, we will need one BED-format file specifying the regions of interest, and at least one BAM-file containing the mapped reads for each ChIP sample. We may also want to control for the technical bias in sequencing coverage by including a non-IP (or non-specific antibody IP) BAM file for each ChIP sample.

For example, let's say we have seven files: (1) "ChIPseq_peaks.bed" (the regions of interest); (2) "ChIPseq_cond1_rep1.bam" (mapped reads for ChIP-seq library in condition 1, 1st replicate); (3) "ChIPseq_cond1_rep2.bam" (mapped reads for ChIP-seq library in condition 1, 2nd replicate); (4) "InputDNA_cond1.bam" (mapped reads for input library in condition 1); (5) "ChIPseq_cond2_rep1.bam" (mapped reads for ChIP-seq library in condition 2, 1st replicate); (6) "ChIPseq_cond2_rep2.bam" (mapped reads for ChIP-seq library in condition 2, 2nd replicate); (7) "InputDNA_cond2.bam" (mapped reads for input library in condition 2). So we have sequenced two replicate IP samples from each condition, but only one input (no-IP) DNA control for each condition.

First, we need to read in the BED file and define the regions of interest using the `GRanges` function (in the `ShortRead` packages).

```
> library(ShortRead)
> peaks <- read.table("ChIPseq_peaks.bed", sep="^", head=F)
> peaks.gr <- GRanges(seqnames=as.character(peaks[[1]]),
+ ranges=IRanges(start=peaks[[2]], end=peaks[[3]]))
```

The first two commands here should be self-explanatory by now, but the third uses the `GRanges` function to create a special type of *GRanges* object on the

workspace, which we have called `peaks.gr`. This function requires at least two arguments: the `seqnames` and the `ranges`. The first of these is typically a vector of chromosome names, and the second is another special type of object which we have created using the `IRanges` function. The `IRanges` function requires a vector of start positions and end positions, with one element in each vector for each range.

The `GRanges` class of object enables some very useful functionality, including counting overlapping reads from a `GenomicAlignments` object. This is what we will do next, for the first ChIP sample, after reading it into the workspace using the `readGAlignments` function.

```
> ChIP1reads.rep1 <- readGAlignments(file="ChIPseq_cond1.rep1.bam",
+ param=ScanBamParam())
> ChIP1counts.rep1 <- summarizeOverlaps(features=peaks.gr,
reads=ChIP1reads.rep1)
```

So in these commands, we have first created an object called `ChIP1reads.rep1`, which contains the information for each mapped read in the BAM file “ChIPseq_cond1.rep1.bam”. Then we have used the `summarizeOverlaps` function to count the number of reads mapping to each range specified in the `peaks.gr` `GRanges` object.

One convenient and widely-used approach to quantifying sequencing coverage over specified ranges is to compute the *Reads Per Kilobase of region per Million mapped reads (RPKM)*. For paired-end libraries, it is usual to count the number of mapping *Fragments*, hence *FPKM*. We can obtain a vector of counts of mapped reads for each feature using the `assay` function applied to the output of `summarizeOverlaps`, we can obtain the length (in bp) of each region of interest by applying the `width` function to the `GRanges` object, and we can find the total number of mapped reads just by applying the `length` function to the `GenomicAlignments` object. Therefore, we can compute the RPKM scores for the `ChIP1reads.rep1` sample in each of the `peaks.gr` regions as follows:

```
> ChIP1rpkm.rep1 <- (1000*assay(ChIP1counts.rep1)*1e6/
length(ChIP1reads.rep1))
+ /width(peaks.gr)
```

This command computes the expression $1,000 * 1,000,000 * \frac{ReadCount_{region}}{width_{region} * ReadCount_{total}}$, and stores the result on the workspace as a vector called `ChIP1rpkm.rep1`.

As we also have an input DNA control for this sample, we may wish to compute instead the log-ratio of the RPKM scores for the ChIP compared to the control: $\log_2\left(\frac{ChIP_{RPKM}+1}{input_{RPKM}+1}\right)$. We add 1 to each of the values in the fraction to avoid undefined measures in the event of zero-coverage in a region. In our

current example, this means we need to generate the RPKM scores for the input sample for condition 1:

```
> input1reads <- readGAlignments(file="InputDNA_cond1bam",
+ param=ScanBamParam())
> input1counts <- summarizeOverlaps(features=peaks.gr,reads=input1reads)
> input1rpkm <- (1000*assay(input1counts)*1e6/length(input1reads))
+ /width(peaks.gr)
```

With the RPKM computed for both the ChIP sample and the input DNA control, we can now compute the log *enrichment* score for each region:

```
> ChIP1logFC.rep1 <- log(((ChIP1rpkm.rep1+1)/(input1rpkm+1)),base=2)
```

For the second replicate of the first sample, repeat the steps to compute the RPKM scores per-region for the corresponding BAM file, and then compute log-enrichment scores using the already-computed input DNA RPKM scores (as we have specified that the ChIP replicates share the same Input DNA control).

```
> ChIP1reads.rep2 <- readGAlignments(file="ChIPseq_cond1_rep2.bam",
+ param=ScanBamParam())
> ChIP1counts.rep2 <- summarizeOverlaps(features=peaks.gr,
reads=ChIP1reads.rep2)
> ChIP1rpkm.rep2 <- (1000*assay(ChIP1counts.rep2)*1e6/
length(ChIP1reads.rep2))
+ /width(peaks.gr)
> ChIP1logFC.rep2 <- log(((ChIP1rpkm.rep2+1)/(input1rpkm+1)),base=2)
```

I leave it as an exercise for the reader to repeat all these steps for the second ChIP condition, with the reminder that we will also need to compute RPKM scores for the two ChIP replicates and the input DNA library for this sample. Assuming this results in vectors of log-enrichment scores called `ChIP2logFC.rep1` and `ChIP2logFC.rep2`, we can use the `cbind` function to create a matrix that we can then analyze using the `limma` package:

```
> logFCmat <- cbind(ChIP1logFC.rep1, ChIP1logFC.rep2,
+ ChIP2logFC.rep1, ChIP2logFC.rep2)
```

Let's load the `limma` package, and specify a design matrix with one intercept column and one column that takes values 0 for condition 1 and 1 for condition 2:

```
> library(limma)
> design <- cbind(intercept=1,cond2=c(0,0,1,1))
```

Here we have used `cbind` to create a 2×2 design matrix, called `design`. Now we can use the `lmFit` function to fit linear models using these explanatory variables with each region in turn as the outcome variable:

```
> ChIPlm <- lmFit(logFCmat,design)
```

This command used the `lmFit` function to fit models to the `logFCmat` values, and stores the result in the object `ChIPlm`. Now we perform empirical Bayes moderation of the linear model t-statistics and extract the statistical analysis output for all features:

```
> ChIPlm <- eBayes(ChIPlm)
> diffChIPtable <- topTable(ChIPlm,coef=2,number=nrow(logFCmat))
```

This final command uses the `topTable` function with the argument `coef=2` to specify that it is the 2nd column of the design matrix we are interested in: this is the numeric indicator for whether or not the samples refer to condition 1 or condition 2. The resulting table can be mapped back to the regions of interest, and therefore can be used to identify regions with significantly different enrichment across the conditions of interest. By using the linear modelling framework, we can extend this analysis to more complex designs. A similar approach could also be used for quantitative analysis of ATAC-seq datasets, which profile chromatin accessibility using an engineered transposase.

13.7 Summary

In this chapter we have seen how to load mapped sequence reads into R and to perform quantitative analysis based on coverage. ChIP-seq libraries will often need to be filtered against blacklisted ultra-accessible genomic regions, and then we can find peak regions of interest using MACS. We have also seen how R can be used to perform quantitative comparisons between different ChIP libraries and identify significantly differentially-enriched regions for arbitrarily complex experimental designs.

Bibliography

- [1] TS Carroll *et al*, “Impact of artifact removal on ChIP quality metrics in ChIP-seq and ChIP-exo data,” *Frontiers in Genetics* 5:75 (2014).
- [2] Zhang *et al*, “Model-based analysis of ChIP-Seq,” *Genome Biology* 9(9):R137 (2008).

14.1 Introduction

The quantitative nature of high-throughput sequencing platforms has also been put to good use for measuring RNA levels. This has variously been optimized for quantification of gene expression, for identifying non-coding RNAs, for quantifying miRNAs, and probably many others. Note, there has been a recent expansion in applications of single-cell RNA sequencing, but the measurement of smaller volumes of RNA from a large number of cells introduces a set of additional analytical challenges. Because of this, we concentrate here on bulk RNA-seq analysis, and leave analysis of single-cell sequencing experiments as a future topic. A list of different RNA-seq applications can be found from Illumina's website at:

<http://www.illumina.com/techniques/sequencing/rna-sequencing.html>.

This includes descriptions of the library prep needed to measure mRNAs, total RNAs, and small RNAs. One distinct advantage of the RNA-seq strategy is that RNAs can be quantified without knowing in advance their exact sequences. However, because the RNA molecules to be profiled (after converting to cDNA) are derived from the genome as templates that can be combined in different ways, transcript mapping and quantification is not a trivial task. See [1] for a relatively old but good overview of RNA-seq experiments for gene expression profiling. A library of short cDNA fragments with ligated sequence adapters is then sequenced, reads are mapped to the genome: either as gapped reads spanning splice junctions, reads mapping uniquely to exonic regions of coding genes, or Poly-A terminal reads.

The exact data processing steps may vary for RNA-seq experiments quantifying different types of RNAs, but most will involve transcriptome assembly, transcript quantification and then normalization. Assuming measurements are obtained from multiple samples, it is likely that we would wish to identify differentially-expressed transcripts, which will be similar but potentially slightly different to the process of finding differentially expressed genes from microarray data (which was covered extensively in [Chapter 8](#) of these tutorials). The rich data produced in such sequencing experiments can be processed in myriad further ways, often with specific purposes, but these are sufficiently esoteric that it would be better to look up methodology in a publication

describing the specific example of interest and then attempt to replicate the analysis, adapting core principles learned here.

14.2 Obtaining RNA-seq data from GEO

In this tutorial we will follow 2 distinct pipelines for RNA-seq data analysis. The first will be a simple approach based on quantifying expression levels for known transcripts, without having to go through the process of fully mapping every read to the reference genome. The second approach will use a range of tools from the *Tuxedo Suite*, which enables identification of novel transcripts, and assessment of their differential expression.

For a practical example on which to try out our RNA-seq analysis, let's say we have a small RNA-seq experiment from a human cancer cell line: two replicates of a cell line treated with CRISPR-Cas9 targeting a gene of interest, and two replicates of the cell line treated with a control construct. This is more than hypothetical, the study in question can be found at <http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE72001>. With single-end reads we will have one file for each sample, with paired-end reads (which are more common in RNA-seq) we will have two files for each sample. Let's use the SRA toolkit to download the study's data and extract it into Fastq format:

```
$ sra-tools/prefetch SRR2156848
```

In the above command, we use assume the SRA toolkit has been installed to the directory 'sra-tools'. Refer back to [Chapter 11](#) for more details on installing this toolkit. When you execute this (and the following) commands, make sure you change this to the correct path to your SRA toolkit directory. The *prefetch* program prepares temporary files for obtaining sequence data efficiently from the SRA. With this run, you can use the *fastq-dump* to extract the data in Fastq format to a file in the current directory.

```
$ sra-tools/fastq-dump --split-3 SRR2156848
```

Note that as this sequencing run is a paired-end library, this command uses the 'split-3' argument to indicate that we wish to extract separate files for the different mate-pairs. Now if we look at the contents of the current directory, we should see the following:

```
$ ls
SRR2156848_1.fastq SRR2156848_2.fastq SRR2156848.fastq
```

How would you check that these files with extension ‘.fastq’ actually look like what we expect for a Fastq file? You could try printing the first few lines to the shell standard output:

```
$ head SRR2156848.1.fastq
```

With the first file downloaded, we can follow the same steps to download the other three Fastq files.

```
$ sra-tools/prefetch SRR2156849
$ sra-tools/fastq-dump --split-3 SRR2156849
$ sra-tools/prefetch SRR2156850
$ sra-tools/fastq-dump --split-3 SRR2156850
$ sra-tools/prefetch SRR2156851
$ sra-tools/fastq-dump --split-3 SRR2156851
```

14.3 Transcript quantification via pseudoalignment

A fast approach to analysis of RNA-seq data makes use of the fact that if you only wish to quantify the expression of a gene, you don’t need to know *where* in the gene the read maps to. In fact, it is enough to know which genes a read *might have* come from. This is the principle behind Kallisto[2]. In brief, Kallisto compares each RNA-seq read against an indexed table of transcript sequences to find the sets of transcripts that each read could have come from. These sets are then used to estimate the most likely set of transcript counts over the whole RNA-seq library. An executable version of Kallisto can be obtained from the Linux shell as follows:

```
$ wget https://github.com/pachterlab/kallisto/releases/
download/v0.43.1/kallisto_linux-v0.43.1.tar.gz
$ gunzip kallisto_linux-v0.43.1.tar.gz
$ tar -xf kallisto_linux-v0.43.1.tar
```

Now you should have a directory called ‘kallisto_linux-v0.43.1’ which contains the executable file *kallisto*. To run Kallisto we need an indexed transcript database and a set of RNA-seq reads to use to quantify the transcripts. We’ll now download and index a transcript database, then use RNA-seq reads for quantification.

14.3.1 Building a transcript index

Let’s first move the bash working directory into the folder where kallisto is installed:

```
$ cd kallisto_linux-v0.43.1
```

We can now pull the hg19 Ensembl reference transcriptome directly from their ftp site:

```
$ wget ftp://ftp.ensembl.org/pub/release-67/fasta/
homo_sapiens/cdna/Homo_sapiens.GRCh37.67.cdna.all.fa.gz
```

Now we need to unzip that:

```
$ gunzip Homo_sapiens.GRCh37.67.cdna.all.fa.gz
```

Now we can use Kallisto's *index* program to build the transcriptome index¹:

```
$ kallisto index -i hg19.ensembl
Homo_sapiens.GRCh37.67.cdna.all.fa
```

This command specified that we give this transcriptome index the name 'hg19.ensembl'.

Now that we have an indexed to match the reads against, we can use Kallisto for fast probabilistic transcript quantification from the RNA-seq libraries.

14.3.2 Quantifying transcripts using reads

To quantify transcripts with Kallisto we use the *quant* command in the software, specifying a transcript index to quantify, an output directory in which to write results to file, and a list of Fastq files to use for the transcript quantification:

```
$ kallisto quant -i hg19.ensembl -o SRR2156848
SRR2156848_1.fastq SRR2156848_2.fastq
```

If this has run correctly, there should be a new directory called *SRR2156848*, containing three files:

```
$ ls SRR2156848
abundance.h5 abundance.tsv run_info.json
```

We can now run the quantification for the remaining samples:

```
$ kallisto quant -i hg19.ensembl -o SRR2156849
SRR2156849_1.fastq SRR2156849_2.fastq
$ kallisto quant -i hg19.ensembl -o SRR2156850
SRR2156850_1.fastq SRR2156850_2.fastq
```

¹note that this will probably take some time to run

```
$ kallisto quant -i hg19.ensembl -o SRR2156851
SRR2156851_1.fastq SRR2156851_2.fastq
```

14.3.3 Downstream analysis

With the probabilistic transcript quantification completed, we can use Bioconductor tools to enable us to apply the exploratory statistical analysis and hypothesis testing tools we are familiar with in R.

There is a function called *readKallisto* in the *SummarizedExperiment* package, which enables straightforward import of Kallisto output. With each sample having its own directory containing the Kallisto output, we can import the transcript count estimates into R using:

```
> library(SummarizedExperiment)
> seqDirs <- c("SRR2156848", "SRR2156849", "SRR2156850", "SRR2156851")
> kset <- readKallisto(paste(seqDirs, "abundance.tsv", sep="/"),
as="matrix")
```

We now have a table of estimated transcript counts for each sample. We can see how many transcripts we have for each sample:

```
> colSums(kset)
SRR2156848 SRR2156849 SRR2156850 SRR2156851
2563611 2600800 2372309 2111474
```

And how many transcripts are detected in at least one sample:

```
> sum(rowSums(kset)>0)
[1] 94561
```

Before subsequent analysis, we might want to filter out those annotated transcripts with no reads:

```
> kset.nonzero <- kset[which(rowSums(kset)>0),]
```

We can apply any exploratory analysis technique to this counts matrix, but we should bear in mind that some samples may have more counts than others, and some transcripts are more likely to be represented than others. As an example, we will perform a principal component analysis of the transcriptomic profiles of these samples. First, we will scale the counts by total transcript count:

```
> kset.scaled <- kset.nonzero/rep((colSums(kset)/1e6),
each=nrow(kset.nonzero))
```


Now we compute the principal components, centering and scaling each transcript's measured levels so that each feature contributes equally to the PCA:

```
> kset.pc <- prcomp(t(kset.nonzero[which(apply(
kset.nonzero,1,sd)>0),]),scale.=T)
```

Now we can use the first two principal components as a co-ordinate system for visualizing the summarized transcriptomic profiles of each sample:

```
> plot(x=kset.pc$x[,1],y=kset.pc$x[,2],
xlab="PC1",ylab="PC2",type="n")
> text(x=kset.pc$x[,1],y=kset.pc$x[,2],
labels=rownames(kset.pc$x))
```

Here we have used the argument `type="n"` to create the plot with no symbols drawn, then used the function `text` to plot text (the sample names) at the corresponding co-ordinates. The result should look like the graph in [Fig. 14.1](#),

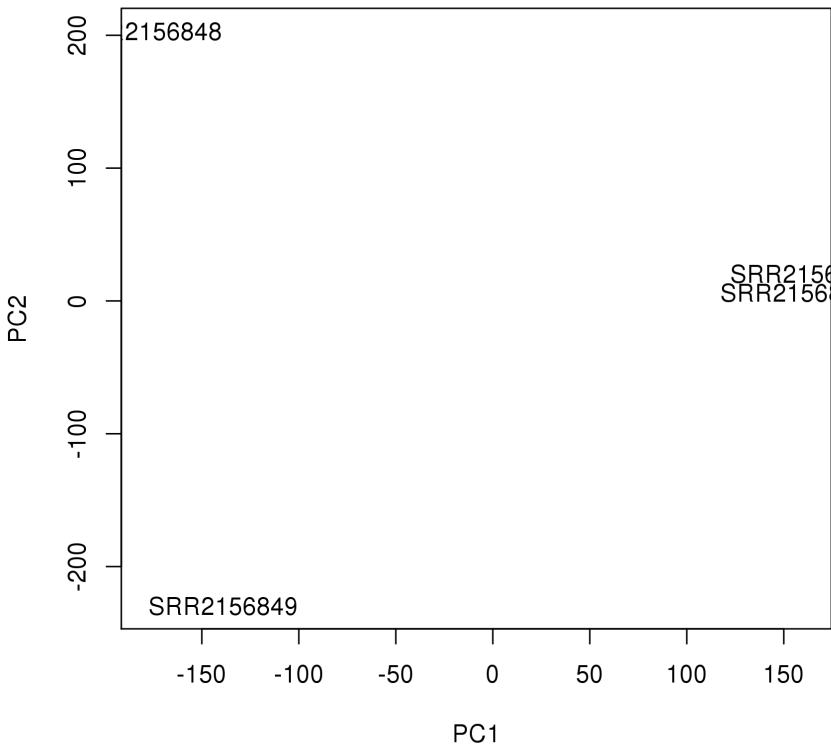


FIGURE 14.1

Principal Component Analysis plot for transcript quantification from RNA-seq using Kallisto.

making it clear that PC1 separates the two control samples (SRR2156848 and SRR2156849) from the two enhancer-targeting CRISPR-Cas9 samples (SRR2156850 and SRR2156851). PC2 separates the two control samples from each other, but you would find that PC3 separates the two enhancer-targeting CRISPR samples from each other. This is at least slightly reassuring, implying that there are considerable differences between the treated and control samples.

We can use the *voom* functionality within the *limma* package to make it appropriate to apply the differential-expression statistical analysis that we are already familiar with:

```
> library(limma)
> de.design <- cbind(intercept=1,eCas9=c(0,0,1,1))
> v <- voom(kset.nonzero,design=de.design)
> diffexp <- eBayes(lmFit(v,design=de.design))
```

We can inspect differentially-expressed transcripts using the *topTable* function. For example, finding the number of transcripts with statistically significant differential expression (setting $\alpha = 0.05$):

```
> nrow(topTable(diffexp,coef=2,number=nrow(kset.nonzero),p.value=0.05))
[1] 2369
```

Now let's map these transcript IDs to gene symbols using *biomaRt*:

```
> library(biomaRt)
> ensembl <- useMart(host="feb2014.archive.ensembl.org",biomart=
"ENSEMBL_MART_ENSEMBL",dataset="hsapiens_gene_ensembl")
> enstLookup <- getBM(attributes=c("ensembl_transcript_id",
"hgnc_symbol"),mart=ensembl)
> enstLookup <- enstLookup[!duplicated(enstLookup[,1]),]
> rownames(enstLookup) <- as.character(enstLookup[,1])
> de.transcripts <- topTable(diffexp,coef=2,
number=nrow(kset.nonzero),p.value=0.05)
> de.transcripts$SYMBOL <- enstLookup[rownames(de.transcripts),
"hgnc_symbol"]
```

And we can draw a heatmap featuring those transcripts (which should result in the plot shown in [Fig. 14.2](#)):

```
> library(gplots)
> heatmap(kset.nonzero[de.transcripts,],col=bluered(100))
```

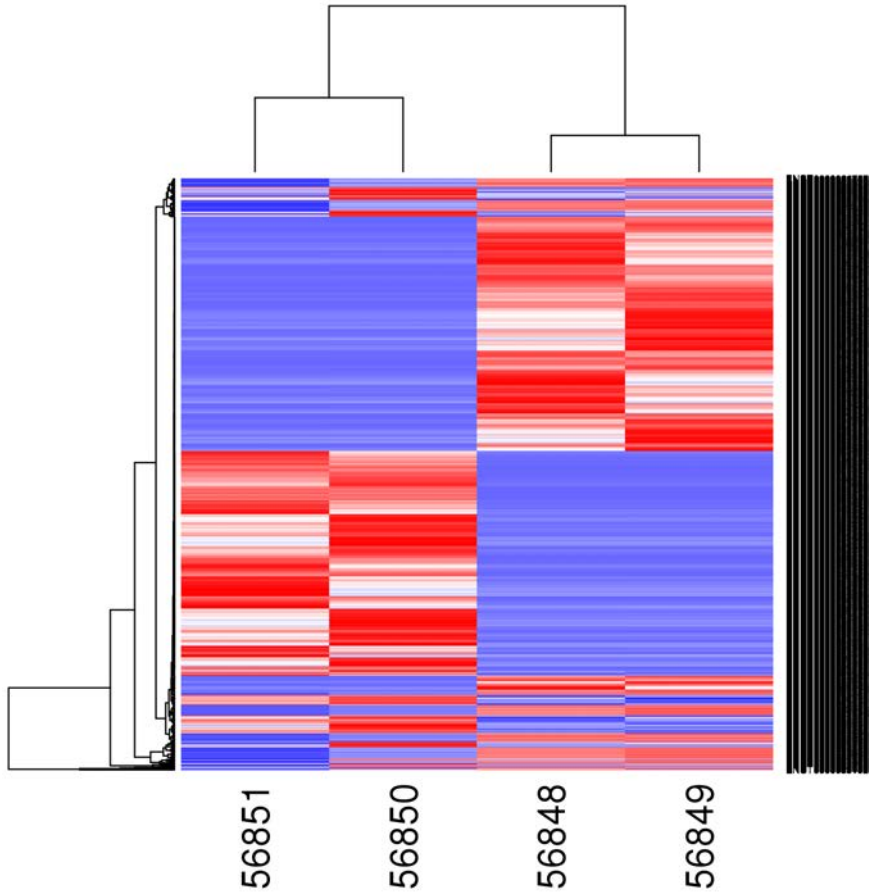


FIGURE 14.2

Heatmap for samples from CRISPR-Cas9 data in [4], based on transcript quantification from RNA-seq using Kallisto.

14.4 Analysis with transcriptome assembly

If we wish to profile expression of non-reference transcripts, we will need to map these reads to a genome so we know which reads came from which transcripts. However, short reads from RNA-derived cDNAs are likely to contain many informative reads that align to the reference genome with gaps (coinciding with splice junctions). Given this, it is desirable to take a slightly different approach to alignment compared to what we have used before. In fact, for processing RNA-seq with a known reference genome, we can use tools built on the *Bowtie* aligner we have already come across. The simpler strategy involves

using a predefined transcriptome, to give previously-defined coding regions of the genome a new set of co-ordinates, then aligning reads to this transcriptome as you would map DNA sequence reads to a genome. This section uses programs from the *Tuxedo Suite*: Bowtie, TopHat and Cufflinks. There are in fact a set of more recently-developed tools for the same tasks, but as they operate in a very similar way, rather than giving instructions for these too I refer you to the *Nature Protocols* paper describing these [5].

14.4.1 Building the transcriptome directly

TopHat is freely-available tool which aligns reads and identifies splice junctions, thereby providing the basis for quantification of reads across the transcripts that were seen in the library. It can be obtained directly from the Linux shell as follows:

```
$ wget https://ccb.jhu.edu/software/tophat/downloads/
tophat-2.1.0.Linux_x86_64.tar.gz
$ gunzip tophat-2.1.0.Linux_x86_64.tar.gz
$ tar -xf tophat-2.1.0.Linux_x86_64.tar
```

Now you should have a directory ‘tophat-2.1.0.Linux_x86_64’ which will contain everything you need to run TopHat². However, in order to use TopHat you need a Bowtie index for the genome against which you wish to align reads; this is covered in Section 11.4.1. TopHat can be run on reads in Fastq files, and it is a good idea to go through QC and filtering procedures beforehand, as you would prior to analysis of any type of sequence data.

We can illustrate this analytical approach using the same RNA-seq experiment as the previous section, which was obtained from the SRA in Section 14.2 of this chapter. TopHat assumes the paired-end read files for each sample are given names that end ‘_1’ for the ‘left’ mate-pairs and ‘_2’ for the ‘right’ mate pairs. Say we have files ‘controlrep1.1.fq’ and ‘controlrep1.2.fq’ for the left and right ends of paired reads from the first replicate of the control treatment, and these are in the directory into which you downloaded and unpacked the ‘tophat’ bundle. We also assume that *Bowtie* was installed with a human reference genome index ‘hg19’. Then we could produce the alignments and splice junction mappings for the sample as follows:

```
$ tophat-2.1.0.Linux_x86_64/tophat -o ControlRep1TopHat
hg19 controlrep1.1.fq controlrep1.2.fq
```

The argument `-o ControlRep1TopHat` specifies that a new directory called ‘ControlRep1TopHat’ will be created. Without this, it defaults to putting the

²Assumes that Bowtie is already installed!

output into a directory called ‘tophat_out’ (which is not necessarily a bad thing, but it probably makes sense to say which sample/experiment the output has come from).

The output from this invocation of TopHat will include the following files:

- accepted_hits.bam the aligned reads
- junctions.bed the called splice junctions listed in BED format
- insertions.bed novel insertions identified by TopHat
- deletions.bed novel deletions identified by TopHat

14.4.2 Transcript quantification

With splice junctions mapped (i.e. a reference transcriptome assembled) it then becomes possible to quantify the number of distinct reads mapping to different transcripts. This is not a straightforward task, as you can imagine when there are multiple transcripts from a single gene and the vast majority of reads are too short to represent a full transcript, most reads could have come from multiple different potential transcripts. However, there are tools that will carry out this task for us. *Cufflinks* is the tool that picks up where TopHat left off. This can be obtained from the Linux shell as follows:

```
$ wget http://cole-trapnell-lab.github.io/cufflinks/assets/downloads/cufflinks-2.2.1.Linux_x86_64.tar.gz
$ gunzip cufflinks-2.2.1.Linux_x86_64.tar.gz
$ tar -xf cufflinks-2.2.1.Linux_x86_64.tar
```

Invoking Cufflinks to run on aligned RNA-seq reads is simple. If we have processed the Fastq files from the CRISPR experiment as described in the previous section, we could generate transcript-level expression measurements for the first replicate of the control as follows:

```
$ cufflinks-2.2.1.Linux_x86_64/cufflinks ControlRep1TopHat/accepted_hits.bam
```

Cufflinks will generate three files of output:

- ‘transcripts.gtf’ – the definition of resolved transcripts from the input reads and assembled transcriptome
- ‘isoforms.fpkms_tracking’ – quantification of transcripts in terms of ‘fragments per kilobase of transcript length per million reads of total sequencing depth’ (FPKM)

- ‘genes.fpk_tracking’ – quantification of gene expression (total across all isoforms) in FPKM

As each sample in an experiment processed in this way will have its own directory following transcriptome assembly and alignment, we can set up a shell script to enter each directory in turn and run Cufflinks on the appropriate input file. While doing this, it is probably sensible to name the individual output files so we can remember which sample they came from. One way to do this involves setting up a `for` loop to run through each directory that was created:

```
$ for dir in `ls *TopHat`;
$ do
$ cd $dir
$ ../cufflinks-2.2.1.Linux_x86_64/cufflinks accepted_hits.bam
$ cp transcripts.gtf ../${dir%TopHat}transcripts.gtf
$ done
```

This script creates a variable `dir` with each value of the expression inside left-quotes ‘...’, that inside the loop can be referred to with `$dir`. By using curly brackets after the dollar-sign, the percent sign removes characters from the end of the value: `${dir%TopHat}` is therefore the value of variable `dir` but without ‘TopHat’ on the end.

We will have run Cufflinks on each of our aligned read libraries from an experiment, but these will each have their own ‘transcripts.gtf’, potentially all unique. To compare different libraries (e.g. from different samples) with each other, these transcript definitions need to be merged. The program *Cuffmerge*, which is included as part of Cufflinks, can do this for us. Cuffmerge is very easy to run, requiring only a text file listing the paths to the transcript definition GTF files (and, of course, that those files exist). For the previous example, given that we have copied all the individual GTF files into the same directory, we could use the shell tools to create this list of paths:

```
$ ls *.gtf > transcriptfiles.txt
```

This command creates a file called `transcriptfiles.txt` from the result of the `ls` command, which lists the files in the current directory matching the specified pattern (anything that ends in ‘.gtf’).

So we can use this result to then run Cuffmerge and generate a final transcript definition file:

```
$ cufflinks-2.2.1.Linux_x86_64/cuffmerge transcriptfiles.txt
```

The result of this will be a merged transcript definition file called ‘merged.gtf’.

Once transcript coverage has been quantified, it may be helpful to account for systematic differences between samples that could arise due to technical reasons. Bias due to sequencing depth will mostly be removed by the FPKM representation of expression level, which is the output from Cufflinks. Previously described batch-effect normalization approaches (Chapter 8 Section 11) may be effective, especially with randomized RNAseq library prep and sequencing batches. However, there is a tool specifically designed for normalizing the output of Cufflinks from multiple different samples, called *Cuffnorm*. Cuffnorm requires a transcript definition GTF file, such as the output from Cufflinks, and a set of aligned sequence SAM/BAM files, which come from TopHat. Following on from our example, we can use the merged transcript definition file and the list of all BAM files:

```
$ cuffnorm merged.gtf controlrep1_aln.bam controlrep2_aln.bam
treatmentrep1_aln.bam treatmentrep2_aln.bam
```

14.4.3 Downstream analysis

The output of Cuffnorm will be a collection of files containing tables of transcript/gene quantifications and annotations. These normalized counts can now be compared together, enabling a vast array of analytical techniques to be applied through environments such as R. As an example, we could try working through the isoform-level quantification data from the CRISPR experiment. To work with this, we would need to read in the count (or fpkm) table, perform analysis (e.g. differential-expression analysis through limma) and then annotate the results by reading in the annotation table and genomic feature data (e.g. through *biomaRt*) and mapping these to the genes:

```
> rnaseq.fpkm <- as.matrix(read.table("isoforms.fpkm_table",
+ sep="\t",head=TRUE,row.names=1))
```

Here we have read in the normalized FPKM transcript quantifications and used the table to create a *matrix* called `rnaseq.fpkm`. We may want to double-check which samples correspond to the column headers (q1_0,q2_0,q3_0 and q4_0), which we can do by inspecting the file ‘samples.table’:

```
> sample.lookup <- read.table("samples.table",sep="\t",head=TRUE)
```

We can see from this annotation table that the first two columns of the `rnaseq.fpkm` matrix correspond to the control samples, and the second two columns of the matrix correspond to the CRISPR-treated samples. With this in mind, we can load the *limma* package to find differentially-expressed transcripts, constructing a design matrix to represent the treatment effect and using this to fit linear models for each transcript (just as with normalized gene expression microarray data, as in Chapter 8 of these tutorials):

```
> library(limma)
> design <- cbind(intercept=1,crispr=c(0,0,1,1))
```

This command creates a design matrix with one term for the intercept (where all samples have value 1) and one term for the treatment (where control samples have value 0 and treated samples have value 1).

```
> crispr.fit <- lmFit(rnaseq.fpkm,design=design)
> crispr.fit <- eBayes(crispr.fit)
```

These commands first fit a linear model to the table of transcript counts, using the design matrix created above, then apply empirical Bayes moderation of t-statistics as described in [3]. We can inspect the most significantly differentially expressed transcripts using the `topTable` function³:

```
> topTable(crispr.fit,coef=2)
```

We can see that the multiple testing correction drastically reduces the apparent statistical significance of differential expression, and this is likely because at transcript isoform level there are a large number of individual tests being carried out with limited statistical power due to the small number of replicates. However, we could identify the set of isoforms with adjusted p-values less than 0.1 (meaning an estimated 1 in 10 of the resulting isoform list could pass these differential expression criteria purely by chance):

```
> limma.result <- topTable(crispr.fit,coef=2,number=nrow(rnaseq.fpkm),
+ p.value=0.1)
```

The resulting table has statistics for 37 differentially-expressed transcript isoforms, but these are annotated only by identifiers which don't mean anything outside the context of this analysis. To make sense of the results, there is annotation information provided by `Cuffnorm`, which we can load into R and map genomic co-ordinates to genes:

```
> transcript.annot <- read.table("isoforms.attr_table",
+ sep="\t",head=TRUE,row.names=1)
> diffexp.coords <- as.character(transcript.annot[rownames(
+ limma.result), + "locus"])
```

This command creates a character vector `diffexp.coords` containing the genomic co-ordinates for the differentially-expressed transcripts. Mapping these to genes will be similar to the approach taken to map ChIP-seq peaks (in [Chapter 11](#)):

³For details of the output of this function, see [Chapter 4](#).


```

> library(biomaRt)
> ensembl <- useMart(host="feb2014.archive.ensembl.org",biomart=
+ "ENSEMBL_MART_ENSEMBL",dataset="hsapiens_gene_ensembl")
> geneInfo <- getBM(attributes=c("hgnc_symbol",
+ "chromosome_name","start_position","end_position","strand")
+ ,mart=ensembl)
> geneInfo <- geneInfo[which(!geneInfo$hgnc_symbol==""),]
> geneInfo <- geneInfo[!duplicated(geneInfo$hgnc_symbol),]
> diffexp.genes <- rep(NA,length(diffexp.coords))
> for(i in 1:length(diffexp.genes)){
+ this.chr <- gsub(strsplit(diffexp.coords[i],split=":")[[1]][1],
pattern="chr",replace="")
+ this.pos <- strsplit(diffexp.coords[i],split=":")[[1]][2]
+ this.start <- as.numeric(strsplit(this.pos,split="-")[[1]][1])
+ this.end <- as.numeric(strsplit(this.pos,split="-")[[1]][2])
+ matching.genes <- which(geneInfo$chromosome_name==this.chr &
geneInfo$start_position<=this.end & geneInfo$end_position>=this.start)
+ if(length(matching.genes)>0) diffexp.genes[i] <- paste(geneInfo$
hgnc_symbol[matching.genes],collapse=";")}

```

The loop here goes through each of the co-ordinates (elements of the vector `diffexp.coords`) in turn, finding the chromosome (by taking only the part before the ‘:’ then stripping out the characters ‘chr’), finding the start and end positions (by taking only the part after the ‘:’, then only the part before the ‘-’ for start co-ordinate and only the part after the ‘-’ for the end co-ordinate), and then finding which rows of the table `geneInfo` match the corresponding co-ordinates. If there are any matches, the final command in this loop will retrieve the gene symbols for the matches and stick them together into a single character string (by using the function `paste` with argument `collapse`), and put these into the corresponding element of the vector `diffexp.genes`. It is probably most helpful if we add this information, and the genomic co-ordinates, to the results table from the *limma* analysis:

```

> limma.result$locus <- diffexp.coords
> limma.result$symbol <- diffexp.genes
As a final step, we may wish to write the results out to file:
> write.table(limma.result,file="GSE72001_DiffExpTranscripts.txt",
+ sep="\t",quote=FALSE,col.names=NA)

```

Now we can see which isoforms of which genes were found to be differentially expressed between these samples. A number of these look rather interesting in the context of cancer, which hopefully illustrates how the power to analyze RNA-seq data can be useful!

14.5 Summary

Two examples of processing RNA-seq data were illustrated in this chapter: one involving *pseudoalignment* for direct read counting against a reference transcriptome; the other involved *transcriptome assembly* to identify expressed RNA isoforms and then quantify expression through counting mapped reads. Which of these approaches you follow will depend on your priorities, although it is probably apparent that the pseudoalignment-based quantification is simpler and quicker, and for many applications will be sufficient.

Bibliography

- [1] Z Wang & M Gerstein & M Snyder “RNA-Seq: a revolutionary tool for transcriptomics,” *Nature Reviews Genetics* 10(1):57-63 (2009).
- [2] N Bray *et al* “Near-optimal probabilistic RNA-seq quantification,” *Nature Biotechnology* 34:525-527 (2016).
- [3] G Smyth “Linear models and empirical bayes methods for assessing differential expression in microarray experiments,” *Statistical Applications in Genetics and Molecular Biology* 3(1):1-25 (2004).
- [4] X Zhang *et al* “Identification of focally amplified lineage-specific super-enhancers in human epithelial cancers,” *Nature Genetics* 48(2):176-182 (2016).
- [5] M Pertea *et al* “Transcript-level expression analysis of RNA-seq experiments with HISAT, StringTie and Ballgown,” *Nature Protocols* 11:1650-1667 (2016).



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Bisulphite Sequencing

15.1 Introduction

Bisulphite sequencing, or the application of high-throughput sequencing technologies to bisulphite-converted DNA, has enabled new levels of resolution to be reached in the genome-wide profiling of cytosine (DNA) methylation. This predominantly comes in two forms: whole-genome bisulphite sequencing (WGBS) and reduced-representation bisulphite sequencing (RRBS). The idea behind WGBS is pretty straightforward in that sodium bisulphite converts unmethylated cytosines to uracils, so the proportion of methylated vs unmethylated DNA fragments covering any given position in the genome can be identified through counting the results of sequencing. The practicalities of WGBS mean that working with the data is less simple: it requires a LOT of sequence reads for enough resolution to call relatively small differences in methylation levels across populations of cells (consider that to call methylation in increments of 1% one would need 100 reads mapping to the same cytosine, 4 reads mapping to the same cytosine would give only increments of 25% at a time). If you are starting with approximately 75 GB of fastq data for a single sample, in the course of analysis (after qc, alignment and methylation calling) you might expect to have comfortably more than 100 GB per sample. And if you consider family-wise error rate across 40 million CpG sites in the human genome, you most likely need a large number of samples for enough power to detect significant differential methylation at individual loci! Given the computational requirements arising from the scale of the data, working with these results most likely requires the support of specialist facilities. Of course, it is also possible to use WGBS to measure the average methylation across genomic regions (such as enhancers), rather than individual CpG sites, but even this will need quite a lot of sequencing depth.

RRBS provides a means to target sequencing to DNA fragments that are enriched for CpG sites whilst remaining distributed across the whole genome [1]. The dramatic reduction in required sequencing depth makes this approach more feasible for analysis of datasets including more samples (or more biological replicates), but with the drawback of no longer covering all CpG sites in the genome.

Owing to the fact that the base-changes induced by bisulphite treatment are precisely what this application of sequencing is intended to measure, it makes sense to take a bespoke approach to alignment of the reads: so that we don't lose reads to 'mismatches' but recognize where these arise from bisulphite conversion of unmethylated cytosines. Preliminary quality control should be carried out as with any other high-throughput sequencing data (see [Chapter 11](#)), with an additional warning that adapter trimming is especially important for RRBS libraries as the size selection means that individual read sequencing will often over-run the ends of DNA fragments and into the adapters.

This chapter will utilize an example dataset that will be relatively convenient for demonstrating common WGBS/RRBS analysis tasks, obtained from the NCBI's Sequence Read Archive (SRA), with accession SRP058260: <http://trace.ncbi.nlm.nih.gov/Traces/study/?acc=SRP058260>.

This dataset contains RRBS reads from the MCF7 breast cancer cell line treated with paclitaxel (SRR2017566) and untreated control (SRR2017565). As we have seen in previous chapters, retrieving data from SRA is a little more complex than downloading microarray data from GEO, owing to its potential size. In fact the data retrieval is so specific it requires use of a bespoke toolkit produced by the SRA (called SRA Toolkit), which can be obtained from: <http://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?view=software>. This toolkit contains a program called *prefetch* which retrieves data from the SRA (you only need to specify an accession), and a program called *fastq-dump* which extracts usable fastq files from the '.sra' file. Assuming that we have installed the appropriate SRA Toolkit into a directory called 'sra-tools' (the precise name will depend on the operating system you are working from), we can obtain the fastq files for SRR2017565 and SRR2017566 from the bash shell as follows:

```
$ sra-tools/prefetch SRR2017565
$ sra-tools/fastq-dump --split-3 /ncbi/public/sra/SRR2017565.sra
```

Here we have to call the *fastq-dump* program with the argument `--split-3` to give separate fastq files for the two paired ends of each read, and I have assumed that the download destination for the *prefetch* tool is '/ncbi/public/sra' (this is where it is by default).

```
$ sra-tools/prefetch SRR2017566
$ sra-tools/fastq-dump --split-3 /ncbi/public/sra/SRR2017566.sra
```

Now you can inspect the quality using FastQC, remove low-quality reads using *Trimmomatic* or in R, and remove adapter sequences with *Trimmomatic*, all as described in [Chapter 9](#) of these tutorials.

15.2 Alignment and methylation calls

Assuming that we are starting with appropriately filtered fastq files, the first step of analysis specific to bisulphite sequencing data is the bisulphite-aware alignment. There are a few tools which can do this, including Bismark [2] and BSmooth [3]. I will provide examples using Bismark, purely for the reason that I am more familiar with this pipeline and therefore I know it works well. The Bismark program is available at http://www.bioinformatics.babraham.ac.uk/projects/bismark/bismark_v0.14.5.tar.gz.

The Bismark pipeline performs alignment to a bisulphite-converted reference genome using Bowtie, then extracts the methylation calls for cytosines in different genomic contexts (CpG, CHG or CHH). First, this process requires a bisulphite-converted reference genome. Let's say we have a folder containing all the hg19 (GRCh37) sequences in fasta format, called 'hg19', and we have installed Bismark into a directory called 'bismark_v0.14.5' so that both 'hg19' and 'bismark_v0.14.5' are subdirectories of our current working directory. Then we can generate a bisulphite-converted reference genome as follows:

```
$ bismark_v0.14.5/bismark_genome_preparation hg19/
```

This should result in a subdirectory called 'Bisulfite_Genome' being created with 'hg19', which can now be used for alignment of bisulphite sequencing reads from human samples.

The next stage is to run the main part of the *Bismark* pipeline. Following on from the example retrieved from SRA in the previous section, let's assume that we have fastq files 'SRR2017565.1.fastq' and 'SRR2017565.2.fastq' (corresponding to the left and right mate pairs of reads from the untreated control MCF7 RRBS library) and 'SRR2017566.1.fastq' and 'SRR2017566.2.fastq' (corresponding to the left and right mate pairs of reads from the paclitaxel-treated MCF7 RRBS library) in our current working directory in a LINUX filesystem. We can generate methylation-level aligned sequences with Bismark as follows:

```
$ bismark_v0.13.0/bismark ./hg19 -1 SRR2017565.1.fastq -2  
SRR2017565.2.fastq
```

This uses default alignment options in Bowtie; for more detail on how these can be altered see the Bismark User Guide¹. The output of the *bismark* program is a report (the file ending 'report.txt') and a SAM file containing aligned reads along with methylation calls.

¹http://www.bioinformatics.babraham.ac.uk/projects/bismark/Bismark_User_Guide.pdf

The SAM format for methylation calls is not particularly easy to use and contains a lot of additional information, so it is best to extract the methylation calls to a more readable format. Bismark contains a program specifically for this purpose, so if we assume that the previous command had output a SAM file called ‘SRR2017565.fastq_bismark.sam’ in our current working directory, we could extract the methylation calls with:

```
$ bismark_v0.14.5/bismark_methylation_extractor -p SRR2017565.  
fastq_bismark.sam
```

In this command the argument `-p` specifies that the SAM file came from a paired-end library. If it had been derived from single-end reads then you would need to specify `-s` instead. The result from running this command will be three text files, the one we are most likely to be most interested in is ‘CpG_context_SRR2017565.fastq_bismark.txt’ as this contains the methylation calls for the CpG sites (and methylation predominantly occurs at CpG sites). These can be either used directly for downstream analysis (e.g. through R), or converted to BedGraph format using the *bismark2bedGraph* tool:

```
$ bismark_v0.14.5/bismark2bedGraph CpG_context_SRR2017565.  
fastq_bismark.txt  
-o SRR2017565_CpGmethylation.bed
```

15.3 Downstream analysis

The BedGraph format is a tabular file with columns specifying chromosome, start position, end position and methylation call (as a percentage) for each CpG site covered by the sequence library. This can be loaded directly into R:

```
> SRR2017565.bed <- read.table("SRR2017565_CpGmethylation.bed",  
+ sep="\t",head=F,skip=1)
```

With the genomic co-ordinates and methylation calls, it is possible to go straight to visualizing the methylation levels across a given genomic region of interest. For example, if we were interested in the methylation across the gene *ESR1*, we could find the co-ordinates in the hg19 genome², which are chr6:152,128,454-152,424,408. Now, let’s say we wish to start looking at the methylation 2 kb upstream of this, so we want chr6 152,126,454 to 152,424,408. We can find the corresponding rows of the `SRR2017565.bed` data frame by

²For example, you can find any gene’s co-ordinates by entering the gene into the search box in the appropriate UCSC genome browser.

searching for all values where the start occurs before the end co-ordinate of interest and the end occurs after the start co-ordinate of interest:

```
> ESR1.sites <- which(SRR2017565.bed[,1]=="chr6" & SRR2017565.bed[,2]
+ <152424408 & SRR2017565.bed[,3]>152126454)
```

We could plot the values for these sites:

```
> plot(x=SRR2017565.bed[ESR1.sites,2],y=SRR2017565.bed[ESR1.sites,4])
```

And add a dashed vertical line to indicate the location of the TSS:

```
> abline(v=152128454,lty=2)
```

You should see the plot as in [Fig. 15.1](#). It's not particularly easy to see any clear pattern in the values though, because there are so many sites with measurements and the variation from position to position is quite high. It may help our interpretation if we add a smoothed line fit through the points, representing a sort of moving average within the region. This can be done using the `lowess` function in R, or by using functionality from the `bsseq` package which implements methods from BSmooth [\[3\]](#).

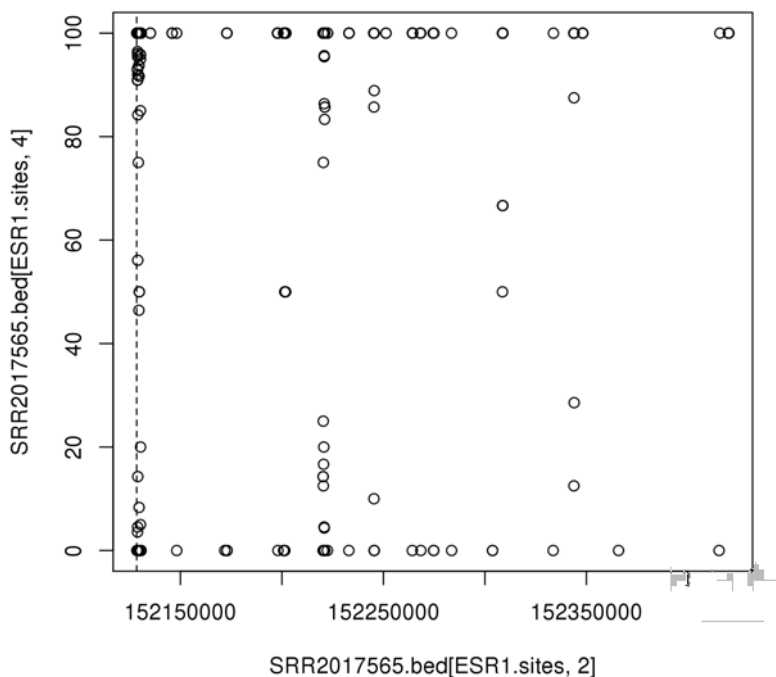


FIGURE 15.1

Methylation calls spanning ESR1 gene, plus 2 kb upstream.

If we use the `lowess` function to create a smoothed line, we can call the function using the same arguments as the `plot` command above:

```
> esr1.smooth <- lowess(x=SRR2017565.bed[ESR1.sites,2],
+ y=SRR2017565.bed[ESR1.sites,4])
```

And then we can recreate the plot using the co-ordinates from the smoothed line:

```
> plot(x=esr1.smooth$x,y=esr1.smooth$y,type="l")
```

Here we specify that we want to plot a line, not points. We can add the TSS indicator too:

```
> abline(v=152128454,lty=2)
```

Now the result should appear as in [Fig. 15.2](#), which seems a lot more informative. However, note with caution that the appearance of the line depends a lot on the ‘span’ of the smoother (how wide a region the moving average is

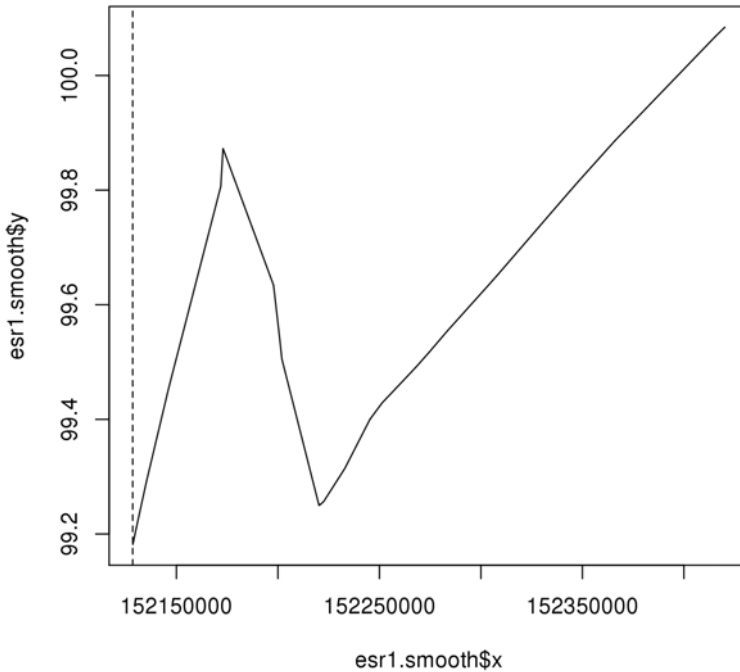


FIGURE 15.2

Smoothed methylation calls spanning ESR1 gene, plus 2kb upstream

calculated over). Try re-fitting the lowess smoother to the points, but varying the span by specifying different values for the argument `f` in the `lowess` function. For example, if we wanted to use 10 points in each window, we specify the proportion of points we want to influence the smoother at each position to be 10 divided by the total number of points:

```
> esr1.smooth10 <- lowess(x=SRR2017565.bed[ESR1.sites,2],  
+ y=SRR2017565.bed[ESR1.sites,4],f=10/length(ESR1.sites))
```

The `bsseq` package can be installed directly into R through Bioconductor. This package facilitates analysis involving smoothed DNA methylation levels[3]. As we have already performed smoothed methylation calling, I will leave the interested reader to visit the package's documentation at:
<https://www.bioconductor.org/packages/release/bioc/html/bsseq.html>.

15.4 Summary

In this chapter we have seen how to use Bismark to process Bisulphite-sequencing libraries from Fastq files through to smoothed per-base DNA methylation percentage calls, and mapping these to features of interest. Hopefully by this point in the book the reader feels sufficiently comfortable with scripting in R to create (or read in from file) a table of regions of interest (such as the promoter regions defined in Section 13.5), find overlapping CpG sites such as for the ESR1 gene example in this chapter, and compute average methylation levels. From that point, if multiple samples from different conditions are available, downstream analysis can proceed as with any other other numeric data (especially the DNA methylation data from [Chapter 9](#)).

Bibliography

- [1] A Meissner *et al* "Reduced representation bisulfite sequencing for comparative high-resolution DNA methylation analysis," *Nucleic Acids Research* 33(18):5868-5877 (2005).
- [2] F Krueger & SR Andrews "Bismark: a flexible aligner and methylation caller for Bisulfite-Seq applications," *Bioinformatics* 27(11):1571-1572 (2011).
- [3] K Hansen & B Langmead & RA Irizarry "BSmooth: from whole genome bisulfite sequencing reads to differentially methylated regions," *Genome Biology* 13:R83 (2012).



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Final Notes

The objectives of this book are to help the reader develop an *understanding* of computational analysis methods that are useful for biological datasets, an *ability to plan* a set of analytical steps to answer specific biological questions using appropriate data, and the *experience of enacting* those plans. This will not come without work, but hopefully this book gives you a good place to start.

This book is certainly not intended to be exhaustive. There are many advanced topics that you might be interested in if you are comfortable with the material presented in this book, for example:

- penalised multiple regression
- distance-based multivariate statistics
- non-linear machine learning
- image processing and analysis
- spectrometry data processing
- single-cell sequencing experiments

There are also new platforms being developed for molecular biology research all the time. This book is deliberately focused on genomics, epigenomics and transcriptomics, because in my experience they are the areas with more standardized processing pipelines and downstream analytical procedures. But I would hope that the chapters on R, statistical methodology and generic data analysis are transferrable to many areas of molecular biology research, and in fact far beyond biological research.

A final word of encouragement: it's unlikely to be easy, but keep at it as improving computational data analysis skills opens so many doors!



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Index

- ATAC-seq, 270
- bash, *see* UNIX shell
- Bioconductor, 73, 146, 204, 226
- Biomart, 265
- biomaRt, 218
- box plot, 93
- boxplot, 26, 55, 58, 154, 200, 227
- ChIP-seq, 259
 - peak calling, 263
 - peak annotation, 265
- chromatin immunoprecipitation,
see ChIP
- clustering, 65, 98
 - distance metric, 98, 124
 - hierarchical clustering, 66, 98,
169, 197
- Copy Number Variation, 211
- correlation, 60, 66, 95, 142, 165, 259
- CRAN, 6
- data frame, 14, 89
- data types, 10
- dendrogram, *see* hierarchical
clustering
- differential expression, 158, 277, 282
- distance metric, 65
 - Euclidean distance, 65
- DNA methylation, 189, 287
- enrichment analysis, 188
- Fastq, 224
- for loop, 16, 63, 136, 168, 184, 215,
251
- functional enrichment, *see*
enrichment analysis
- Gene Expression Omnibus, 197, 235,
272
- GEO, *see* Gene Expression Omnibus
- GSEA, *see* enrichment analysis
- GWAS, 203
- heatmap, 66, 101, 125, 130, 197
- hierarchical clustering, 124
- high-throughput sequencing, 54,
224
 - alignment, 231, 278, 289
 - filtering reads, 228
 - quality control, 225
- hypothesis testing, 47, 63, 71
 - Chi-squared test, 208
 - multiple hypothesis testing,
76, 78, 116, 120, 161,
210
 - non-parametric, 64
- indel, 244
- linear algebra, *see* matrix
- linear model, 69, 81
 - limma*, 72, 103, 118, 158, 199,
269, 277, 282
- linear models, 133, 142
 - limma*, 128
- Linux, *see* UNIX
- MA plot, 151
- matrix, 52, 69
- microarray, 54
 - DNA methylation microarray,
190
 - gene expression microarray, 55,
66, 145
 - SNP array, 203

- normalization, [54](#), [155](#), [193](#), [204](#), [211](#)
 - batch effect, [59](#)
 - quantile normalization, [59](#)
 - RMA, [55](#)
 - z-score, [58](#), [91](#)
- null hypothesis, *see* [hypothesis testing](#)
- over-representation, *see* [enrichment analysis](#)
- pathways, [110](#), [113](#)
- permutation testing, *see* [resampling](#)
- principal component analysis, [82](#), [275](#)
- probability distribution, [40](#), [45](#)
- pseudoalignment, [273](#)
- random variable, [44](#), [48](#)
- read counting, [273](#)
- reference genome, [231](#), [239](#)
- resampling, [86](#)
- RNA-seq, [271](#)
- RPPA, *see* [Reverse Phase Proteomics Array](#)
- RStudio, [6](#)
- scatter plot, [24](#), [62](#), [92](#), [151](#), [164](#)
- SCP, [35](#)
- segmentation, [211](#), [213](#), [216](#)
- Single Nucleotide Polymorphism, *see* [SNP](#)
- Single Nucleotide Variant, *see* [SNV](#)
- SRA, *see* [Sequence Read Archive](#)
- statistical inference, *see* [hypothesis testing](#)
- Structural Variant, *see* [SV](#)
- survival analysis, [79](#), [134](#), [175](#), [252](#)
 - Cox proportional hazards regression, [81](#), [184](#)
 - Kaplan-Meier, [79](#), [178](#)
- t-test, [40](#), [76](#), [104](#)
- TCGA, *see* [The Cancer Genome Atlas](#)
- UNIX, [31](#), [223](#)
- UNIX shell, [32](#)
- unsupervised learning, [65](#)
- variant annotation, [243](#)
- variant calling, [239](#)