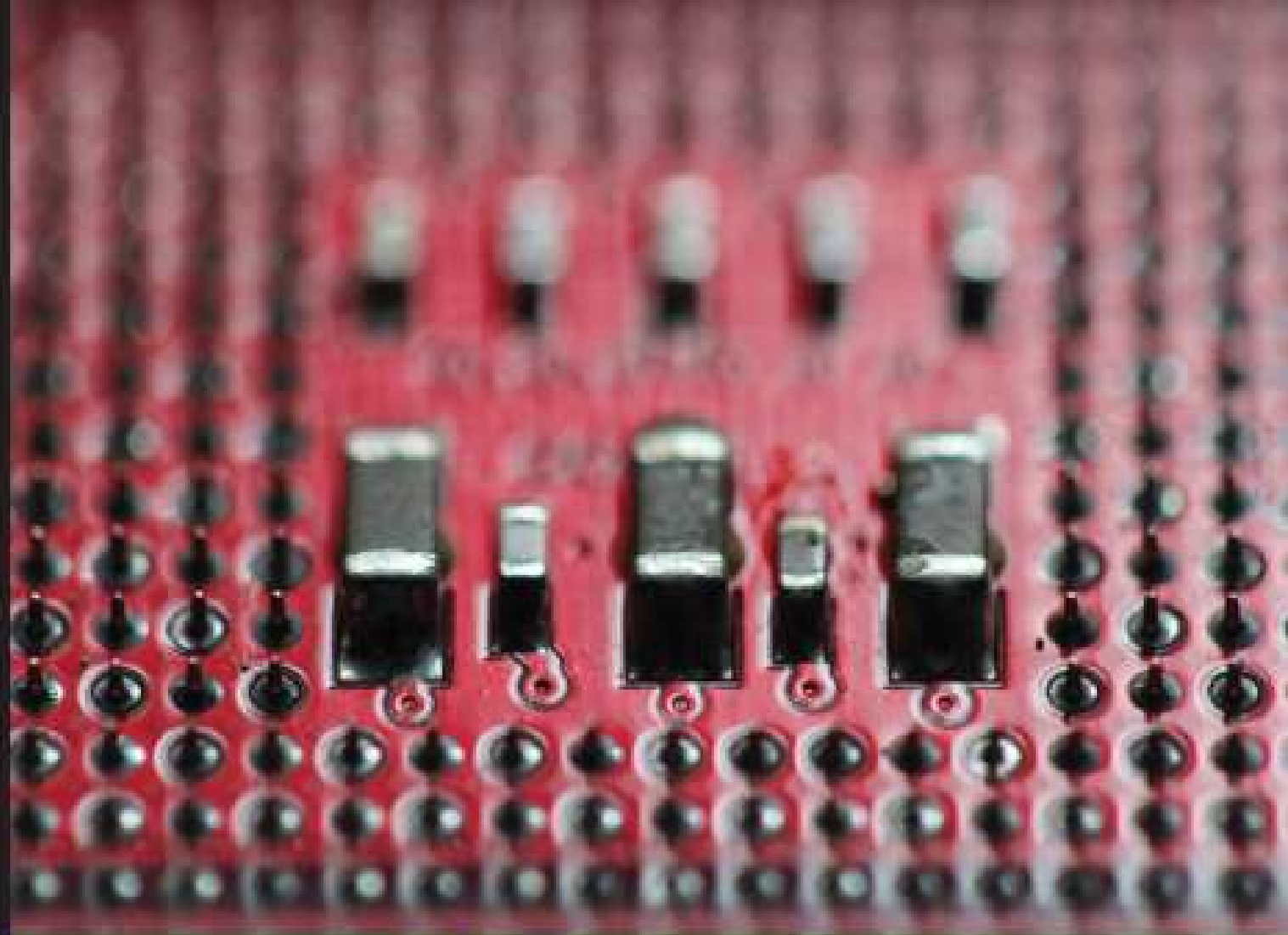




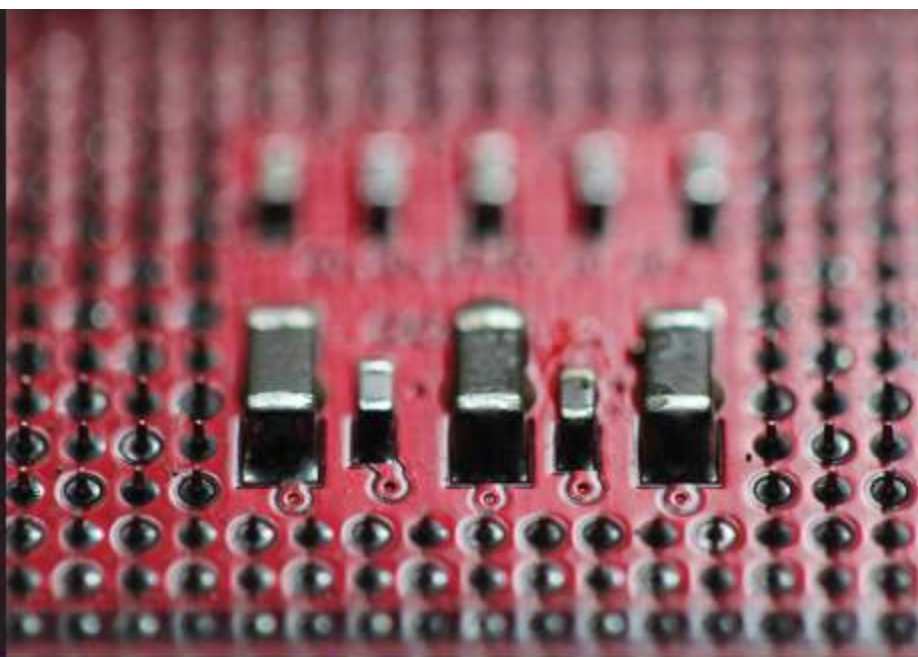
Quick answers to common problems

Bioinformatics with R Cookbook

Over 90 practical recipes for computational biologists to model
and handle real-life data using R

Paurush Praveen Sinha

[PACKT] open source 
PUBLISHING community experience matters



Quick answers to common problems

Bioinformatics with R Cookbook

Over 90 practical recipes for computational biologists to model
and handle real-life data using R

Paurush Praveen Sinha

[PACKT] open source*
PUBLISHING community experience distilled

Table of Contents

[Bioinformatics with R Cookbook](#)

[Credits](#)

[About the Author](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers, and more](#)

[Why Subscribe?](#)

[Free Access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Downloading the color images of this book](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Starting Bioinformatics with R](#)

[Introduction](#)

[Getting started and installing libraries](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Reading and writing data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Filtering and subsetting data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Basic statistical operations on data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Generating probability distributions](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Performing statistical tests on data](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Visualizing data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Working with PubMed in R](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Retrieving data from BioMart](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[2. Introduction to Bioconductor](#)

[Introduction](#)

[Installing packages from Bioconductor](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Handling annotation databases in R](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Performing ID conversions](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[The KEGG annotation of genes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[The GO annotation of genes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[The GO enrichment of genes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[The KEGG enrichment of genes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Bioconductor in the cloud](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[3. Sequence Analysis with R](#)

[Introduction](#)

[Retrieving a sequence](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Reading and writing the FASTA file](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Getting the detail of a sequence composition](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Pairwise sequence alignment](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Multiple sequence alignment](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Phylogenetic analysis and tree plotting](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Handling BLAST results](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Pattern finding in a sequence](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See](#)

[4. Protein Structure Analysis with R](#)

[Introduction](#)

[Retrieving a sequence from UniProt](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Protein sequence analysis](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Computing the features of a protein sequence](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Handling the PDB file](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Working with the InterPro domain annotation](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Understanding the Ramachandran plot](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Searching for similar proteins](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Working with the secondary structure features of proteins](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Visualizing the protein structures](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[5. Analyzing Microarray Data with R](#)

[Introduction](#)

[Reading CEL files](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Building the ExpressionSet object](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Handling the AffyBatch object](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Checking the quality of data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Generating artificial expression data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Data normalization](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Overcoming batch effects in expression data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[An exploratory analysis of data with PCA](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Finding the differentially expressed genes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Working with the data of multiple classes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Handling time series data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Fold changes in microarray data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[The functional enrichment of data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Clustering microarray data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Getting a co-expression network from microarray data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[More visualizations for gene expression data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[6. Analyzing GWAS Data](#)

[Introduction](#)

[The SNP association analysis](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Running association scans for SNPs](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[The whole genome SNP association analysis](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Importing PLINK GWAS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Data handling with the GWASTools package](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Manipulating other GWAS data formats](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[The SNP annotation and enrichment](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Testing data for the Hardy-Weinberg equilibrium](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Association tests with CNV data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Visualizations in GWAS studies](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[7. Analyzing Mass Spectrometry Data](#)

[Introduction](#)

[Reading the MS data of the mzXML/mzML format](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See](#)

[Reading the MS data of the Bruker format](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Converting the MS data in the mzXML format to MALDIquant](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Extracting data elements from the MS data object](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Preprocessing MS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Peak detection in MS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Peak alignment with MS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Peptide identification in MS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Performing protein quantification analysis](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Performing multiple groups' analysis in MS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Useful visualizations for MS data analysis](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[8. Analyzing NGS Data](#)

[Introduction](#)

[Querying the SRA database](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Downloading data from the SRA database](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Reading FASTQ files in R](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Reading alignment data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Preprocessing the raw NGS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Analyzing RNAseq data with the edgeR package](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[The differential analysis of NGS data using limma](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Enriching RNAseq data with GO terms](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[The KEGG enrichment of sequence data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Analyzing methylation data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Analyzing ChipSeq data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Visualizations for NGS data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[9. Machine Learning in Bioinformatics](#)

[Introduction](#)

[Data clustering in R using k-means and hierarchical clustering](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Visualizing clusters](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Supervised learning for classification](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Probabilistic learning in R with Naïve Bayes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Bootstrapping in machine learning](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Cross-validation for classifiers](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Measuring the performance of classifiers](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Visualizing an ROC curve in R](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Biomarker identification using array data](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[A. Useful Operators and Functions in R](#)

[B. Useful R Packages](#)

[Index](#)

Bioinformatics with R Cookbook

Bioinformatics with R Cookbook

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2014

Production reference: 1160614

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-313-2

www.packtpub.com

Cover image by Aniket Sawant

([<aniket_sawant_photography@hotmail.com>](mailto:aniket_sawant_photography@hotmail.com))

Credits

Author

Paurush Praveen Sinha

Reviewers

Chris Beeley

Yu-Wei, Chiu (David Chiu)

Commissioning Editor

Kunal Parikh

Acquisition Editor

Kevin Colaco

Content Development Editor

Ruchita Bhansali

Technical Editors

Arwa Manasawala

Ankita Thakur

Copy Editors

Dipti Kapadia

Insiya Morbiwala

Project Coordinator

Binny K. Babu

Proofreaders

Simran Bhogal

Maria Gould

Ameesha Green

Paul Hindle

Indexer

Mariammal Chettiyar

Graphics

Sheetal Aute

Abhinash Sahu

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Paurush Praveen Sinha has been working with R for the past seven years. An engineer by training, he got into the world of bioinformatics and R when he started working as a research assistant at the Fraunhofer Institute for Algorithms and Scientific Computing (SCAI), Germany. Later, during his doctorate, he developed and applied various machine learning approaches with the extensive use of R to analyze and infer from biological data. Besides R, he has experience in various other programming languages, which include Java, C, and MATLAB. During his experience with R, he contributed to several existing R packages and is working on the release of some new packages that focus on machine learning and bioinformatics. In late 2013, he joined the Microsoft Research-University of Trento COSBI in Italy as a researcher. He uses R as the backend engine for developing various utilities and machine learning methods to address problems in bioinformatics.

Successful work is a fruitful culmination of efforts by many people. I would like to hereby express my sincere gratitude to everyone who has played a role in making this effort a successful one. First and foremost, I wish to thank David Chiu and Chris Beeley for reviewing the book. Their feedback, in terms of criticism and comments, was significant in bringing improvements to the book and its content. I sincerely thank Kevin Colaco and Ruchita Bhansali at Packt Publishing for their effort as editors. Their cooperation was instrumental in bringing out the book. I appreciate and acknowledge Binny K. Babu and the rest of the team at Packt Publishing, who have been very professional, understanding, and helpful throughout the project. Finally, I would like to thank my parents, brother, and sister for their encouragement and appreciation and the pride they take in

my work, despite of not being sure of what I'm doing. I thank them all. I dedicate the work to Yashi, Jayita, and Ahaan.

About the Reviewers

Chris Beeley is a data analyst working in the healthcare industry in the UK. He completed his PhD in Psychology from the University of Nottingham in 2009 and now works within Nottinghamshire Healthcare NHS Trust in the involvement team providing statistical analysis and reports from patient and staff experience data.

Chris is a keen user of R and a passionate advocate of open source tools within research and healthcare settings as well as the author of *Web Application Development Using R with Shiny*, Packt Publishing.

Yu-Wei, Chiu (David Chiu) is one of the co-founders of the company, NumerInfo, and an officer of Taiwan R User Group. Prior to this, he worked for Trend Micro as a software engineer, where he was responsible for building up Big Data platforms for business intelligence and customer relationship management systems. In addition to being an entrepreneur and data scientist, he also specializes in using Hadoop to process Big Data and applying data mining techniques for data analysis. Another of his specialties is that he is also a professional lecturer who has been delivering talks on Python, R, Hadoop, and Tech Talks in Taiwan R User Group meetings and varieties of conferences as well.

Currently, he is working on a book compilation for Packt Publishing called *Machine Learning with R Cookbook*. For more information, visit his personal website at ywchiu.com.

I would like to express my sincere gratitude to my family and friends for supporting and encouraging me to complete this book review. I would like to thank my mother, Ming-Yang Huang (Miranda Huang); my mentor, Man-Kwan Shan; Taiwan R User Groups; and other friends who gave me a big hand.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

In recent years, there have been significant advances in genomics and molecular biology techniques, giving rise to a data boom in the field. Interpreting this huge data in a systematic manner is a challenging task and requires the development of new computational tools, thus bringing an exciting, new perspective to areas such as statistical data analysis, data mining, and machine learning. R, which has been a favorite tool of statisticians, has become a widely used software tool in the bioinformatics community. This is mainly due to its flexibility, data handling and modeling capabilities, and most importantly, due to it being free of cost.

R is a free and robust statistical programming environment. It is a powerful tool for statistics, statistical programming, and visualizations; it is prominently used for statistical analysis. It has evolved from S, developed by John Chambers at Bell Labs, which is a birthplace of many programming languages including C. Ross Ihaka and Robert Gentleman developed R in the early 1990s.

Roughly around the same time, bioinformatics was emerging as a scientific discipline because of the advent of technological innovations such as sequencing, high throughput screening, and microarrays that revolutionized biology. These techniques could generate the entire genomic sequence of organisms; microarrays could measure thousands of mRNAs, and so on. All this brought a paradigm shift in biology from a small data discipline to one big data discipline, which is continuing till date. The challenges posed by this data shoot-up initially compelled researchers to adopt whatever tools were available at their disposal. Till this time, R was in its initial days and was popular among statisticians. However, following the need and the competence of R during the late 90s (and the following decades), it started gaining popularity in the field of computational biology and bioinformatics.

The structure of the R environment is a base program that provides basic programming functionalities. These functionalities can be extended with smaller specialized program modules called packages or libraries. This modular structure empowers R to unify most of the data analysis tasks in one program. Furthermore, as it is a command-line environment, the prerequisite programming skill is minimal; nevertheless, it requires some programming experience.

This book presents various data analysis operations for bioinformatics and computational biology using R. With this book in hand, we will solve many interesting problems related to the analysis of biological data coming from different experiments. In almost every chapter, we have interesting visualizations that can be used to present the results.

Now, let's look at a conceptual roadmap organization of the book.

What this book covers

[Chapter 1](#), *Starting Bioinformatics with R*, marks the beginning of the book with some groundwork in R. The major topics include package installation, data handling, and manipulations. The chapter is further extended with some recipes for a literature search, which is usually the first step in any (especially biomedical) research.

[Chapter 2](#), *Introduction to Bioconductor*, presents some recipes to solve basic bioinformatics problems, especially the ones related to metadata in biology, with the packages available in Bioconductor. The chapter solves the issues related to ID conversions and functional enrichment of genes and proteins.

[Chapter 3](#), *Sequence Analysis with R*, mainly deals with the sequence data in terms of characters. The recipes cover the retrieval of sequence data, sequence alignment, and pattern search in the sequences.

[Chapter 4](#), *Protein Structure Analysis with R*, illustrates how to work with proteins at sequential and structural levels. Here, we cover important aspects and methods of protein bioinformatics, such as sequence and structure analysis. The recipes include protein sequence analysis, domain annotations, protein structural property analysis, and so on.

[Chapter 5](#), *Analyzing Microarray Data with R*, starts with recipes to read and load the microarray data, followed by its preprocessing, filtering, mining, and functional enrichment. Finally, we introduce a co-expression network as a way to map relations among genes in this chapter.

[Chapter 6](#), *Analyzing GWAS Data*, talks about analyzing the GWAS data in order to make biological inferences. The chapter also covers multiple association analyses as well as CNV data.

[Chapter 7](#), *Analyzing Mass Spectrometry Data*, deals with various aspects of analyzing the mass spectrometry data. Issues related to reading different data formats, followed by analysis and quantifications, have been included in this chapter.

[Chapter 8](#), *Analyzing NGS Data*, illustrates various next generation sequencing data. The recipes in this chapter deal with NGS data processing, RNAseq, ChipSeq, and methylation data.

[Chapter 9](#), *Machine Learning in Bioinformatics*, discusses recipes related to machine learning in bioinformatics. We attempt to reach the issues of clustering classification and Bayesian learning in this chapter to infer from the biological data.

[Appendix A](#), *Useful Operators and Functions in R*, contains some useful general functions in R to perform various generic and non-generic operations.

[Appendix B](#), *Useful R Packages*, contains a list and description of some interesting libraries that contain utilities for different types of analysis and visualizations.

What you need for this book

Most importantly, this book needs R itself, which is available for download at <http://cran.r-project.org> for all major operating systems. The instructions to get the additional R packages and datasets are provided in the relevant recipes of the book. Besides R, the book will need some additional software namely, Java Development Kit, MySQL GraphViz, MUSCLE, libxml2, and libxml(2)-devel as prerequisites for some of the R packages. They are available at their respective pages.

Who this book is for

People often approach programming with great apprehension. The purpose of this book is to provide a guide for scientists working on diverse common problems in bioinformatics and computational biology. The book also appeals to programmers who are working in bioinformatics and computational biology but are familiar with languages other than R.

A basic knowledge of computer programming as well as some familiarity with the basics of bioinformatics is expected from the readers. Nevertheless, a short groundwork has been presented at the beginning of every chapter in an attempt to bridge the gap, if any.

The book is not any text on basic programming using R or on basics of bioinformatics and statistics. Appropriate theoretical references have been provided whenever required, directing the reader to related reference articles, books, and blogs. The recipes are mostly ready for use but it is strongly recommended that you look at the data manually to get a feel for it before you start analyzing it in the recipes presented.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can use the `install.packages` function to install a package from CRAN that has many mirror locations."

Any command-line input or output is written as follows:

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

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "From the **Packages** menu in the toolbar, select **Install package(s)...**"

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [<feedback@packtpub.com>](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/3132OS_ColoredImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details

of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Starting Bioinformatics with R

In this chapter, we will cover the following recipes:

- Getting started and installing libraries
- Reading and writing data
- Filtering and subsetting data
- Basic statistical operations on data
- Generating probability distributions
- Performing statistical tests on data
- Visualizing data
- Working with PubMed in R
- Retrieving data from BioMart

Introduction

Recent developments in molecular biology, such as high throughput array technology or sequencing technology, are leading to an exponential increase in the volume of data that is being generated. Bioinformatics aims to get an insight into biological functioning and the organization of a living system riding on this data. The enormous data generated needs robust statistical handling, which in turn requires a sound computational statistics tool and environment. R provides just that kind of environment. It is a free tool with a large community and leverages the analysis of data via its huge package libraries that support various analysis operations.

Before we start dealing with bioinformatics, this chapter lays the groundwork for upcoming chapters. We first make sure that you know how to install R, followed by a few sections on the basics of R that will rejuvenate and churn up your memories and knowledge on R programming that we assume you already have. This part of the book will mostly introduce you to certain functions in R that will be

useful in the upcoming chapters, without getting into the technical details. The latter part of the chapter (the last two recipes) will introduce Bioinformatics with respect to literature searching and data retrieval in the biomedical arena. Here, we will also discuss the technical details of the R programs used.

Getting started and installing libraries

Libraries in R are packages that have functions written to serve specific purposes; these include reading specific file formats in the case of a microarray datafile or fetching data from certain databases, for example, GenBank (a sequence database). You must have these libraries installed in the system as well as loaded in the R session in order to be able to use them. They can be downloaded and installed from a specific repository or directly from a local path. Two of the most popular repositories of R packages are **Comprehensive R Archive Network (CRAN)** and Bioconductor. CRAN maintains and hosts identical, up-to-date versions of code and documentation for R on its mirror sites. We can use the `install.packages` function to install a package from CRAN that has many mirror locations. Bioconductor is another repository of R and the associated tool with a focus on other tools for the analysis of high throughput data. A detailed description on how to work with Bioconductor (<http://www.bioconductor.org>) is covered in the next chapter.

This recipe aims to explain the steps involved in installing packages/libraries as well as local files from these repositories.

Getting ready

To get started, the prerequisites are as follows:

- You need an R application installed on your computer. For more details on the R program and its installation, visit <http://cran.r-project.org>.
- You need an Internet connection to install packages/libraries from web repositories such as CRAN and Bioconductor.

How to do it...

The initialization of R depends on the operating system you are using. On Windows and Mac OS platforms, just clicking on the program starts an R session, like any other application for these systems. However, for Linux, R can be started by typing in R into the terminal (for all Linux distributions, namely, Ubuntu, SUSE Debian, and Red Hat). Note that calling R via its terminal or command line is also possible in Windows and Mac systems.

This book will mostly use Linux as the operating system; nevertheless, the differences will be explained whenever required. The same commands can be used for all the platforms, but the Linux-based R lacks the default **graphical user interface (GUI)** of R. At this point, it is worth mentioning some of the code editors and **integrated development environments (IDEs)** that can be used to work with R. Some popular IDEs for R include RStudio (<http://www.rstudio.com>) and the Eclipse IDE (<http://www.eclipse.org>) with the `StatET` package. To learn more about the `StatET` package, visit <http://www.walware.de/goto/statet>. Some commonly used code editors are Emacs, Kate, Notepad++, and so on. The R GUI in Windows and Mac has its own code editor that meets all the requirements.

Windows and Mac OS GUIs make installing packages pretty straightforward. Just follow the ensuing steps:

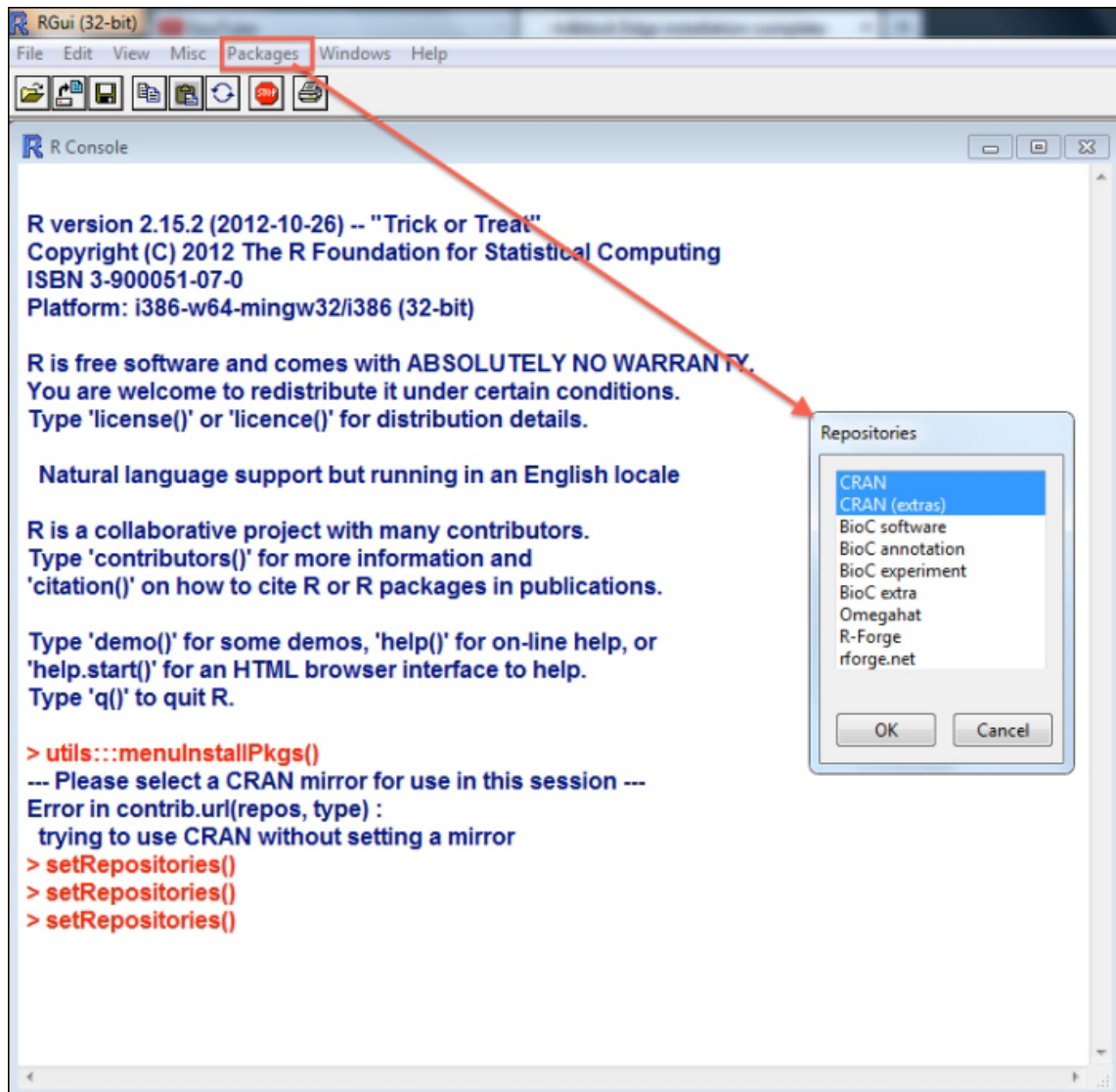
1. From the **Packages** menu in the toolbar, select **Install package(s)....**
2. If this is the first time that you are installing a package during this session, R will ask you to pick a mirror. A selection of the nearest mirror (geographically) is more feasible for a faster download.
3. Click on the name of the package that you want to install and then on the **OK** button.

R downloads and installs the selected packages.

Note

By default, R fetches packages from CRAN. However, you can change this if necessary just by choosing **Select repositories...** from the **Packages** menu. You are required to change the default repository or switch the repository in case the desired package is available in a different repository. Remember that a change in the repository is different from a change in the mirror; a mirror is the same repository at a different location.

The following screenshot shows how to set up a repository for a package installation in the R GUI for Windows:



4. Install an R package in one of the following ways:

- From a terminal, install it with the following simple command:

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

- From a local directory, install it by setting the repository to null as follows:

```
>
```

```
install.packages("path/to/mypackage.tar.g  
z", repos = NULL, type="source")
```

- Another way to install packages in Unix (Linux) is without entering R (from the source) itself. This can be achieved by entering the following command in the shell terminal:

```
R CMD INSTALL  
path/to/mypackage.tar.gz
```

5. To check the installed libraries/packages in R, type the following command:

```
> library()
```

6. To quit an R session, type in `q()` at the R prompt, and the session will ask whether you want to save the session as a workspace image or not or whether you want to cancel the quit command. Accordingly, you need to type in `y`, `n`, or `c`. In a Windows or Mac OS, you can directly close the R program like any other application.

```
> q()  
Save workspace image [y/n/c]: n
```

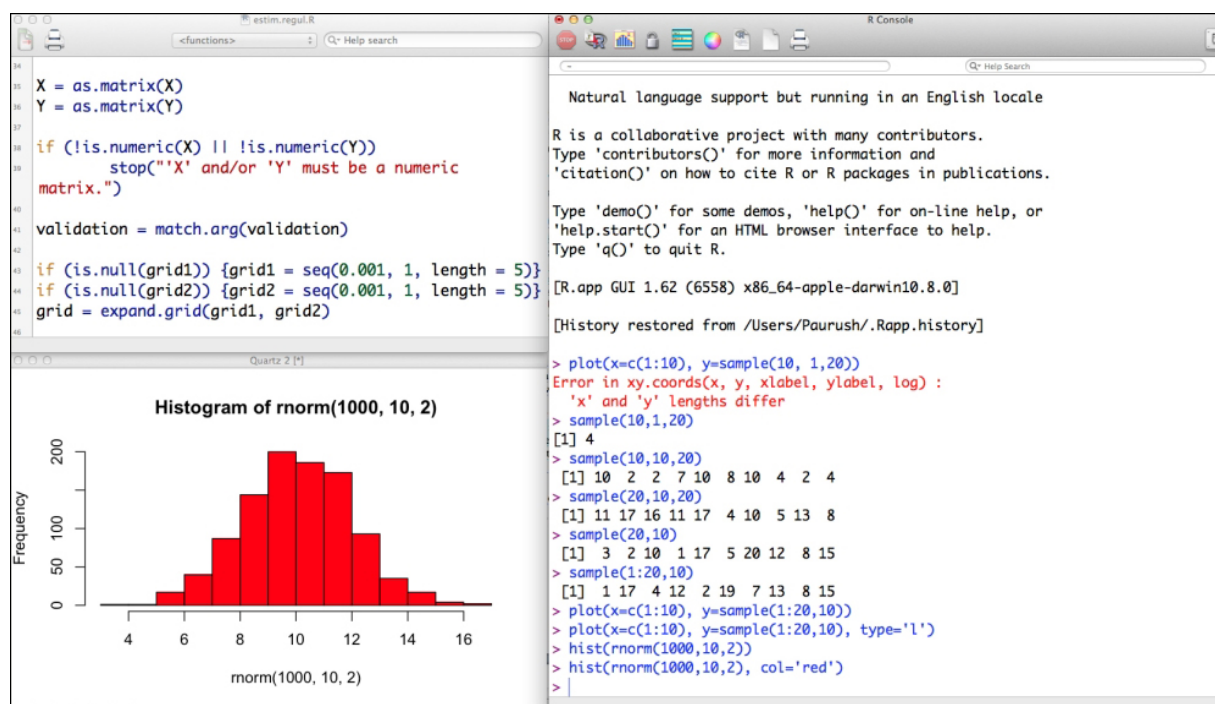
Tip

Downloading the example code

You can download the example code files for all Packt books that you have purchased from your account at <http://www.packtpub.com>. If you purchased this book from elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

How it works...

An R session can run as a GUI on a Windows or Mac OS platform (as shown in the following screenshot). In the case of Linux, the R session starts in the same terminal. Nevertheless, you can run R within the terminal in Windows as well as Mac OS:



The R GUI in Mac OS showing the command window (right), editor (top left), and plot window (bottom left)

The `install.packages` command asks the user to choose a mirror (usually the nearest) for the repository. It also checks for the dependencies required for the package being installed, provided we set the `dependencies` argument to `TRUE`. Then, it downloads the binaries (Windows and Mac OS) for the package (and the dependencies, if required). This is followed by its installation. The function also checks the compatibility of the package with R, as on occasions, the library cannot be loaded or installed due to an incorrect version or missing dependencies. In such cases, the installed packages are revoked. Installing from the source is required in cases where you have to compile binaries for your own machine in terms of the R version or so. The availability of binaries for the package makes installation easier for naive users. The filenames of the package binaries have a `.tgz/.zip` extension. The value of `repos`

can be set to any remote source address for a specific remote source. On Windows, however, the function is also encoded in terms of a GUI that graphically and interactively shows the list of binary versions of the packages available for your R version. Nevertheless, the command-line installation is also functional on the Windows version of R.

There's more...

A few libraries are loaded by default when an R session starts. To load a library in R, run the following command:

```
> load(package_name)
```

Loading a package imports all the functions of this specific package into the R session. The default packages in the session can be viewed using the following `getOption` command:

```
> getOption("defaultPackages")
```

The currently loaded libraries in a session can be seen with the following command:

```
> print(.packages())
```

An alternative for this is `sessionInfo()`, which provides version details as well.

All the installed packages can be displayed by running the `library` function as follows:

```
> library()
```

Besides all this, R has a comprehensive built-in help system. You can get help from R in a number of ways. The Windows and Mac OS platforms offer help as a separate HTML page (as shown in the following screenshot) and Linux offers similar help text in the running terminal. The following is a list of options that can be used to seek help in R:

```
> help.start()
> help(sum) # Accesses help file for function sum
> ?sum # Searches the help files for function sum
> example(sum) # demonstrates the function with an
example
> help.search("sum") # uses the argument
character to search help files
```

All of the previous functions provide help in a unique way. The `help.start` command is the general command used to start the hypertext version of the R documentation. All the help files related to the package can be checked with the following command:

```
> help(package="package_name")
```

The following screenshot shows an HTML help page for the `sum` function in R:

← → http://127.0.0.1:17159/library/base/html/sum.html R: Sum of Vector Elements ×

sum {base} R Documentation

Sum of Vector Elements

Description

sum returns the sum of all the values present in its arguments.

Usage

sum(..., na.rm = FALSE)

Arguments

... numeric or complex or logical vectors.

na.rm logical. Should missing values (including NaN) be removed?

Details

This is a generic function: methods can be defined for it directly or via the [Summary](#) group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

If na.rm is FALSE an NA or NaN value in any of the arguments will cause a value of NA or NaN to be returned, otherwise NA and NaN values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, NULL is accepted and treated as if it were `integer(0)`.

Value

The sum. If all of ... are of type integer or logical, then the sum is integer, and in that case the result will be NA (with a warning) if integer overflow occurs. Otherwise it is a length-one numeric or complex vector.

Reading and writing data

Before we start with analyzing any data, we must load it into our R workspace. This can be done directly either by loading an external R object (typical file extensions are `.rda` or `.RData`, but it is not limited to these extensions) or an internal R object for a package or a TXT, CSV, or Excel file. This recipe explains the methods that can be used to read data from a table or the `.csv` format and/or write similar files into an R session.

Getting ready

We will use an iris dataset for this recipe, which is available with R Base packages. The dataset bears quantified features of the morphologic variation of the three related species of Iris flowers.

How to do it...

Perform the following steps to read and write functions in R:

1. Load internal R data (already available with a package or base R) using the following `data` function:

```
> data(iris)
```

2. To learn more about iris data, check the `help` function in R using the following function:

```
> ?iris
```

3. Load external R data (conventionally saved as `.rda` or `.RData`, but not limited to this) with the following `load` function:

```
> load(file="mydata.RData")
```

4. To save a data object, say, `D`, you can use the `save` function as follows:

```
> save(D, file="myData.RData")
```

5. To read the tabular data in the form of a `.csv` file with `read.csv` or `read.table`, type the following command:

```
> mydata <- read.table("file.dat", header =  
TRUE, sep="\t", row.names = 1)  
> mydata <- read.csv("mydata.csv")
```

6. It is also possible to read an Excel file in R. You can achieve this with various packages such as `xlsx` and `gdata`. The `xlsx` package requires Java settings, while `gdata` is relatively simple. However, the `xlsx` package offers more functionalities, such as read permissions for different sheets in a workbook and the newer versions of Excel files. For this example, we will use the `xlsx` package. Use the `read.xlsx` function to read an Excel file as follows:

```
> install.packages("xlsx", dependencies=TRUE)  
> library(gdata)  
> mydata <- read.xls("mydata.xls")
```

7. To write these data frames or table objects into a CSV or table file, use the `read.csv` or `write.table` function as follows:

```
> write.table(x, file = "myexcel.xls", append  
= FALSE, quote = TRUE, sep = " ")  
  
> write.csv(x, col.names = NA, sep = ",")
```

How it works...

The `read.csv` or `write.csv` commands take the filename in the current working directory—if a complete path has not been specified—and based on the separators (usually the `sep` argument), import

the data frames (or export them in case of `write` commands). To find out the current working directory, use the `getwd()` command. In order to change it to your desired directory, use the `setwd` function as follows:

```
> setwd("path/to desired/directory")
```

The second argument `header` indicates whether or not the first row is a set of labels by taking the Boolean values `TRUE` or `FALSE`. The `read.csv` function may not work in the case of incomplete tables with the default argument `fill`. To overcome such issues, use the value, `TRUE` for the `fill` argument. To learn more about optional arguments, take a look at the help section of the `read.table` function. Both the functions (`read.table` and `read.csv`) can use the headers (usually the first row) as column names and specify certain column numbers as row names.

There's more...

To get further information about the loaded dataset, use the `class` function for the dataset to get the type of dataset (object class). The data or object type in R can be of numerous types. This is beyond the scope of the book. It is expected that the reader is acquainted with these terms. Here, in the case of the iris data, the type is a data frame with 150 rows and five columns (type the `dim` command with `iris` as the argument). A data frame class is like a matrix but can accommodate objects of different types, such as character, numeric, and factor, within it. You can take a look at the first or last few rows using the `head` or `tail` functions (there are six rows by default) respectively, as follows:

```
> class(iris)
> dim(iris)
> head(iris)
> tail(iris)
```

The following `WriteXLS` package allows us to write an object into an Excel file for the `x` data object:

```
> install.packages(WriteXLS)
> library(WriteXLS)
> WriteXLS(x, ExcelFileName = "R.xls")
```

The package also allows us to write a list of data frames into the different sheets of an Excel file. The `WriteXLS` function uses Perl in the background to carry out tasks. The `sheet` argument can be set within the function and assigned the sheet number where you want to write the data.

The `save` function in R is a standard way to save an object. However, the `saveRDS` function offers an advantage as it doesn't save both the object and its name; it just saves a representation of the object. As a result, the saved object can be loaded into a named object within R that will be different from the name it had when it was originally serialized. Let's take a look at the following example:

```
> saveRDS(myObj, "myObj.rds")
> myObj2 <- readRDS("myObj.rds")
> ls()
[1] "myObj" "myObj2"
```

Another package named `data.table` can be used to perform data reading at a faster speed, which is especially suited for larger data. To know more about the package, visit the CRAN page for the package at <http://cran.r-project.org/web/packages/data.table/index.html>.

The `foreign` package (<http://cran.r-project.org/web/packages/foreign/index.html>) is available to read/write data for other programs such as SPSS and SAS.

Filtering and subsetting data

The data that we read in our previous recipes exists in R as data frames. Data frames are the primary structures of tabular data in R. By a tabular structure, we mean the row-column format. The data we store in the columns of a data frame can be of various types, such as numeric or factor. In this recipe, we will talk about some simple operations on data to extract parts of these data frames, add a new chunk, or filter a part that satisfies certain conditions.

Getting ready

The following items are needed for this recipe:

- A data frame loaded to be modified or filtered in the R session (in our case, the iris data)
- Another set of data to be added to item 1 or a set of filters to be extracted from item 1

How to do it...

Perform the following steps to filter and create a subset from a data frame:

1. Load the iris data as explained in the earlier recipe.
2. To extract the names of the species and corresponding sepal dimensions (length and width), take a look at the structure of the data as follows:

```
> str(iris)
'data.frame': 150 obs. of 5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4
4.6 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9
3.4 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7
```

```

1.4 ...
  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4
0.3 ...
  $ Species      : Factor w/ 3 levels
"setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1
...

```

3. To extract the relevant data to the `myiris` object, use the `data.frame` function that creates a data frame with the defined columns as follows:

```

>
myiris=data.frame(Sepal.Length=iris$Sepal.Length,
Sepal.Width= iris$Sepal.Width, Species=
iris$Species)

```

4. Alternatively, extract the relevant columns or remove the irrelevant ones (however, this style of subsetting should be avoided):

```

> myiris <- iris[,c(1,2,5)]

```

5. Instead of the two previous methods, you can also use the removal approach to extract the data as follows:

```

> myiris <- iris[,-c(3,4)]

```

6. You can add to the data by adding a new column with `cbind` or a new row through `rbind` (the `rnorm` function generates a random sample from a normal distribution and will be discussed in detail in the next recipe):

```

> Stalk.Length <-c
(rnorm(30,1,0.1),rnorm(30,1.3,0.1),
rnorm(30,1.5,0.1),rnorm(30,1.8,0.1),
rnorm(30,2,0.1))
> myiris <- cbind(iris, Stalk.Length)

```

7. Alternatively, you can do it in one step as follows:

```
> myiris$Stalk.Length =
c(rnorm(30,1,0.1),rnorm(30,1.3,0.1),
rnorm(30,1.5,0.1),rnorm(30,1.8,0.1),
rnorm(30,2,0.1))
```

8. Check the new data frame using the following commands:

```
> dim(myiris)
[1] 150    6
> colnames(myiris)# get column names for the
data frame myiris

[1] "Sepal.Length" "Sepal.Width"
"Petal.Length"  "Petal.Width"  "Species"
"Stalk.Length"
```

9. Use `rbind` as depicted:

```
newdat <- data.frame(Sepal.Length=10.1,
Sepal.Width=0.5, Petal.Length=2.5,
Petal.Width=0.9, Species="myspecies")
> myiris <- rbind(iris, newdat)
> dim(myiris)
[1] 151    5
> myiris[151,]
      Sepal.Length Sepal.Width Petal.Length
Petal.Width      Species
151          10.1          0.5          2.5
0.9 myspecies
```

10. Extract a part from the data frame, which meets certain conditions, in one of the following ways:

- One of the conditions is as follows:

```
> mynew.iris <- subset(myiris,
Sepal.Length == 10.1)
```

- An alternative condition is as follows:

```
> mynew.iris <-
myiris[myiris$Sepal.Length == 10.1, ]
```

```

> mynew.iris
      Sepal.Length Sepal.Width
Petal.Length Petal.Width  Species
      151         10.1      0.5
2.5         0.9 myspecies
> mynew.iris <- subset(iris,
Species == "setosa")

```

11. Check the following first row of the extracted data:

```

> mynew.iris[1,]
Sepal.Length Sepal.Width Petal.Length
Petal.Width  Species
1         5.1         3.5         1.4
0.2 setosa

```

You can use any comparative operator as well as even combine more than one condition with logical operators such as `&` (AND), `|` (OR), and `!` (NOT), if required.

How it works...

These functions use R indexing with named columns (the `$` sign) or index numbers. The `$` sign placed after the data followed by the column name specifies the data in that column. The R indexing system for data frames is very simple, just like other scripting languages, and is represented as `[rows, columns]`. You can represent several indices for rows and columns using the `c` operator as implemented in the following example. A minus sign on the indices for rows/columns removes these parts of the data. The `rbind` function used earlier combines the data along the rows (row-wise), whereas `cbind` does the same along the columns (column-wise).

There's more...

Another way to select part of the data is using `%in%` operators with the data frame, as follows:

```

> mylength <- c(4,5,6,7,7.2)

```

```
> mynew.iris <- myiris[myiris[,1] %in% mylength,]
```

This selects all the rows from the data that meet the defined condition. The condition here means that the value in column 1 of `myiris` is the same as (matching) any value in the `mylength` vector. The extracted rows are then assigned to a new object, `mynew.iris`.

Basic statistical operations on data

R being a statistical programming environment has a number of built-in functionalities to perform statistics on data. Nevertheless, some specific functionalities are either available in packages or can easily be written. This section will introduce some basic built-in and useful in-package options.

Getting ready

The only prerequisite for this recipe is the dataset that you want to work with. We use our iris data in most of the recipes in this chapter.

How to do it...

The steps to perform a basic statistical operation on the data are listed here as follows:

1. R facilitates the computing of various kinds of statistical parameters, such as mean standard deviation, with a simple function. This can be applied on individual vectors or on an entire data frame as follows:

```
> summary(iris) # Shows a summary for each  
column for table data
```

Sepal.Length	Sepal.Width	Petal.Length
Petal.Width		
Min. :4.300	Min. :2.000	Min. :1.000
Min. :0.100		
1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600
1st Qu.:0.300		
Median :5.800	Median :3.000	Median :4.350
Median :1.300		
Mean :5.843	Mean :3.057	Mean :3.758

```

Mean      :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
3rd Qu.:1.800
  Max.    :7.900   Max.      :4.400   Max.      :6.900
Max.      :2.500
      Species
  setosa    :50
 versicolor:50
 virginica  :50

> mean(iris[,1])
[1] 5.843333
> sd(iris[,1])
[1] 0.8280661

```

2. The `cor` function allows for the computing of the correlation between two vectors as follows:

```

> cor(iris[,1], iris[,2])
[1] -0.1175698
> cor(iris[,1], iris[,3])
[1] 0.8717538

```

3. To get the covariance for the data matrix, simply use the `cov` function as follows:

```

> Cov.mat <- cov(iris[,1:4])
> Cov.mat
Sepal.Length Sepal.Width Petal.Length
Petal.Width
Sepal.Length    0.6856935 -0.0424340
1.2743154    0.5162707
Sepal.Width     -0.0424340  0.1899794
-0.3296564   -0.1216394
Petal.Length    1.2743154 -0.3296564
3.1162779    1.2956094
Petal.Width     0.5162707 -0.1216394
1.2956094    0.5810063

```

How it works...

Most of the functions we saw in this recipe are part of basic R or generic functions. The `summary` function in R provides the summaries of the input depending on the class of the input. The function invokes

various functions depending on the class of the input object. The returned value also depends on the input object. For instance, if the input is a vector that consists of numeric data, it will present the mean, median, minimum, maximum, and quartiles for the data, whereas if the input is tabular (numeric) data, it will give similar computations for each column. We will use the `summary` function in upcoming chapters for different types of input objects.

The functions accept the data as input and simply compute all these statistical scores on them, displaying them as vector, list, or data frame depending on the input and the function. For most of these functions, we have the possibility of using the `na.rm` argument. This empowers the user to work with missing data. If we have missing values (called `NA` in R) in our data, we can set the `na.rm` argument to `TRUE`, and the computation will be done only based on non-NA values. Take a look at the following chunk for an example:

```
> a <- c(1:4, NA, 6)
> mean(a) # returns NA
[1] NA
> mean(a, na.rm=TRUE)
[1] 3.2
```

We see here that in the case of missing values, the `mean` function returns `NA` by default as it does not know how to handle the missing value. Setting `na.rm` to `TRUE` actually computes the mean of five numbers (1, 2, 3, 4, and 6) in place of 6 (1, 2, 3, 4, NA, and 6), returning 3.2.

To compute the correlation between the sepal length and sepal width in our iris data, we simply use the `cor` function with the two columns (sepal length and sepal width) as the arguments for the function. We can compute the different types of correlation coefficients, namely Pearson, Spearman, Kendall, and so on, by specifying the apt value for the method arguments in the function. For more details, refer to the `help (?cor)` function.

Generating probability distributions

Before we talk about anything in this section, try the `?Distributions` function in your R terminal (console). You will see that a help page consisting of different probability distributions opens up. These are part of the base package of R. You can generate all these distributions without the aid of additional packages. Some interesting distributions are listed in the following table. Other distributions, for example, **multivariate normal distribution (MVN)**, can be generated by the use of external packages (MASS packages for MVN). Most of these functions follow the same syntax, so if you get used to one, others can be achieved in a similar fashion.

In addition to this simple process, you can generate different aspects of the distribution just by adding some prefixes.

How to do it...

The following are the steps to generate probability distributions:

1. To generate 100 instances of normally distributed data with a mean equal to 1 and standard deviation equal to 0.1, use the following command:

```
> n.data <- rnorm(n=100, mean=1, sd=0.1)
```

2. Plot the histogram to observe the distribution as follows:

```
> hist(n.data)
```

3. Check the density of the distribution and observe the shape by typing the following command:

```
> plot(density(n.data))
```

Do you see the bell shape in this plot?

4. To identify the corresponding parameters for other prefixes, use the following [help](#) file example:

```
> ?pnorm
```

The following table depicts the functions that deal with various statistical distributions in R (R Base packages only):

Distribution	Probability	Quantile	Density	Random
Beta	pbeta	qbeta	dbeta	rbeta
Binomial	pbinom	qbinom	dbinom	rbinom
Cauchy	pcauchy	qcauchy	dcauchy	rcauchy
Chi-Square	pchisq	qchisq	dchisq	rchisq
Exponential	pexp	qexp	dexp	rexp
F	pf	qf	df	rf

Distribution	Probability	Quantile	Density	Random
Gamma	<code>pgamma</code>	<code>qgamma</code>	<code>dgamma</code>	<code>rgamma</code>
Geometric	<code>pgeom</code>	<code>qgeom</code>	<code>dgeom</code>	<code>rgeom</code>
Hypergeometric	<code>phyper</code>	<code>qhyper</code>	<code>dhyper</code>	<code>rhyper</code>
Logistic	<code>plogis</code>	<code>qlogis</code>	<code>dlogis</code>	<code>rlogis</code>
Log Normal	<code>plnorm</code>	<code>qlnorm</code>	<code>dlnorm</code>	<code>rlnorm</code>
Negative Binomial	<code>pnbinom</code>	<code>qnbinom</code>	<code>dnbinom</code>	<code>rnbinom</code>
Normal	<code>pnorm</code>	<code>qnorm</code>	<code>dnorm</code>	<code>rnorm</code>
Poisson	<code>ppois</code>	<code>qpois</code>	<code>dpois</code>	<code>rpois</code>
Student t	<code>pt</code>	<code>qt</code>	<code>dt</code>	<code>rt</code>
Studentized Range	<code>ptukey</code>	<code>qtukey</code>	<code>dtukey</code>	<code>rtukey</code>
Uniform	<code>punif</code>	<code>qunif</code>	<code>dunif</code>	<code>runif</code>

How it works...

The `rnorm` function has three arguments: `n` (the number of instances you want to generate), the desired mean of the distribution, and the desired standard deviation (`sd`) in the distribution. The command thus generates a vector of length n , whose mean and standard deviations are as defined by you. If you look closely at the functions described in the table, you can figure out a pattern. The prefixes `p`, `q`, `d`, and `r` are added to every distribution name to generate probability, quintiles, density, and random samples, respectively.

There's more...

To learn more about statistical distribution, visit the Wolfram page at <http://www.wolframalpha.com/examples/StatisticalDistributions.html>.

Performing statistical tests on data

Statistical tests are performed to assess the significance of results in research or application and assist in making quantitative decisions. The idea is to determine whether there is enough evidence to reject a conjecture about the results. In-built functions in R allow several such tests on data. The choice of test depends on the data and the question being asked. To illustrate, when we need to compare a group against a hypothetical value and our measurements follow the Gaussian distribution, we can use a one-sample t-test. However, if we have two paired groups (both measurements that follow the Gaussian distribution) being compared, we can use a paired t-test. R has built-in functions to carry out such tests, and in this recipe, we will try out some of these.

How to do it...

Use the following steps to perform a statistical test on your data:

1. To do a t-test, load your data (in our case, it is the sleep data) as follows:

```
> data(sleep)
```

2. To perform the two-sided, unpaired t-test on the first and second columns (the values for the two conditions), type the following commands:

```
> test <- t.test(sleep[,1]~sleep[,2])
> test
Welch Two Sample t-test
data:  sleep[, 1] by sleep[, 2]
t = -1.8608, df = 17.776, p-value = 0.07939
alternative hypothesis: true difference in
```

```

means is not equal to 0
95 percent confidence interval:
-3.3654832  0.2054832
sample estimates:
mean in group 1 mean in group 2
0.75           2.33

```

3. Create a contingency table as follows:

```

> cont <- matrix(c(14, 33, 7, 3), ncol = 2)
> cont
[,1] [,2]
[1,]  14   7
[2,]  33   3

```

4. Create a table that represents two types of cars, namely, sedan and convertible (columns) and two genders, male and female, and a count of these that own the types of cars along the rows. Thus, you have the following output:

```

> colnames(cont) <- c("Sedan", "Convertible")
> rownames(cont) <- c("Male", "Female")
> cont
Sedan Convertible
Male      14      7
Female    33      3

```

5. In order to find the car type and gender, carry out a Chi-square test based on this contingency table as follows:

```

> test <- chisq.test(as.table(cont))
> test
Pearson's Chi-squared test with Yates'
continuity correction
data:  as.table(cont)
X-squared = 4.1324, df = 1, p-value = 0.04207

```

6. For a Wilcoxon signed-rank test, first create a set of vectors containing observations to be tested as x and y, as shown in the following commands:

```

> x <- c(1.83, 0.50, 1.62, 2.48, 1.68,

```

```
1.88, 1.55, 3.06, 1.30)  
> y <- c(0.878, 0.647, 0.598, 2.05, 1.06,  
1.29, 1.06, 3.14, 1.29)
```

7. This is simply followed by a command that you need to execute to run the Wilcoxon signed-rank test as follows:

```
> test <- wilcox.test(x, y, paired = TRUE,  
alternative = "greater")
```

8. To look at the contents of the object test, check the structures as follows and look at the specific values of the components:

```
> str(test)  
> test$p.value
```

How it works...

The t-test (in our case, it is two sample t-tests) computes how the calculated mean may deviate from the real mean by chance. Here, we use the sleep data that already exists in R. This sleep data shows the effect of two drugs in terms of an increase in the hours of sleep compared to the sleep data of 10 control patients. The result is a list that consists of nine elements, such as p-value, confidence interval, method, and mean estimates.

Chi-square statistics investigate whether the distributions of the categorical variables differ from one another. It is commonly used to compare observed data with the data that we would expect to obtain according to a specific hypothesis. In this recipe, we considered the scenario that one gender has a different preference for a car, which comes out to true at a p-value cutoff at 0.05. We can also check the expected values for the Chi-square test with the

`chisq.test(as.table(cont))$expected` function.

The Wilcoxon test is used to compare two related samples or repeated measurements on a single sample, to assess if their population mean ranks differ. It can be used to compare the results of two methods. Let x and y be the performance results of two

methods, and our alternative hypothesis is that x is shifted to the right of y (greater). The p-value returned by the test facilitates the acceptance or rejection of the null hypothesis.

There's more...

There are certain other tests, such as the permutation test, Kolmogorov-Smirnov test, and so on, that can be done with R using different functions for appropriate datasets. A few more tests will be discussed in later chapters. To learn more about statistical tests, you can refer to a brief tutorial at

<http://udel.edu/~mcdonald/statbigchart.html>.

Visualizing data

Data is more intuitive to comprehend if visualized in a graphical format rather than in the form of a table, matrix, text, or numbers. For example, if we want to visualize how the sepal length in the Iris flower varies with the petal length, we can plot along the x and y axes, respectively, and visualize the trend or even the correlation (scatter plot). In this recipe, we look at some common way of visualizing data in R and plotting functions with R Base graphics functions. We also discuss the basic plotting functions. These plotting functions can be manipulated in many ways, but discussing them is beyond the scope of this book. To get to know more about all the possible arguments, refer to the corresponding help files.

Getting ready

The only item we need ready here is the dataset (in this recipe, we use the iris dataset).

How to do it...

The following are the steps for some basic graph visualizations in R:

1. To create a scatter plot, start with your iris dataset. What you want to see is the variation of the sepal length and petal length. You need a plot of the sepal length (column 1) along the y axis and the petal length (column 4) along the x axis, as shown in the following commands:

```
> sl <- iris[,1]
> pl <- iris[,4]
> plot(x=pl, y=sl, xlab="Petal length",
       ylab="Sepal length", col="black",
       main="Varition of sepal length with petal
       length")
```

Or alternatively, we can use the following command:

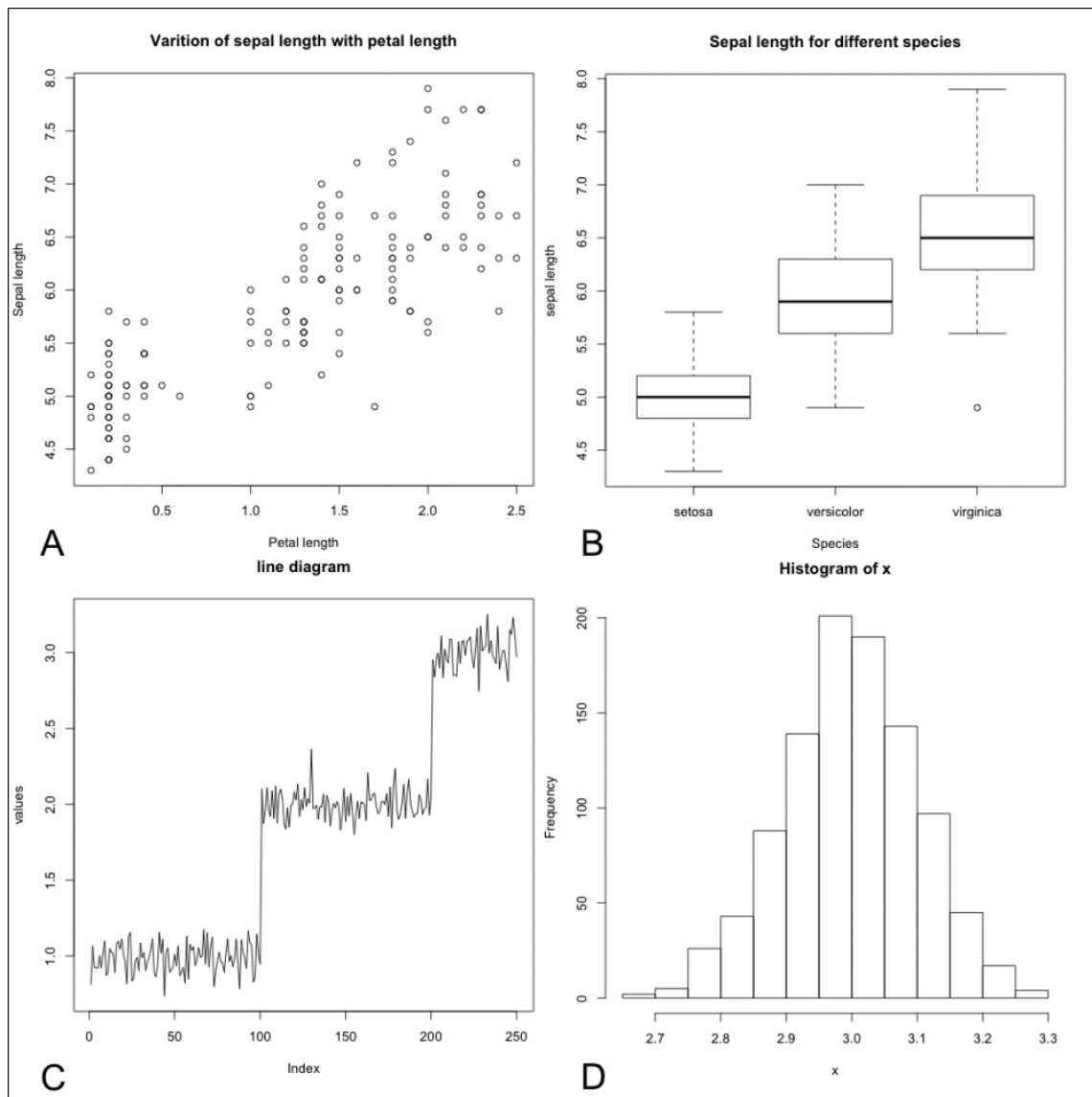
```
> plot(with(iris, plot(x = Sepal.Length,  
y=Petal.Length))
```

2. To create a boxplot for the data, use the `boxplot` function in the following way:

```
> boxplot(Sepal.Length~Species, data=iris,  
ylab="sepal length", xlab="Species",  
main="Sepal length for different species")
```

3. Plotting a line diagram, however, is the same as plotting a scatter plot; just introduce another argument type into it and set it to `'l'`. However, we use a different, self-created dataset to depict this as follows:

```
> genex <- c(rnorm(100, 1, 0.1), rnorm(100, 2,  
0.1), rnorm(50, 3, 0.1))  
> plot(x=genex, xlim=c(1,5), type='l',  
main="line diagram")
```



Plotting in R: (A) Scatter plot, (B) Boxplot, (C) Line diagram, and (D) Histogram

4. Histograms can be used to visualize the density of the data and the frequency of every bin/category. Plotting histograms in R is pretty simple; use the following commands:

```
> x <- rnorm(1000, 3, 0.02)
> hist(x)
```

How it works...

The `plot` function extracts the relevant data from the original dataset with the column numbers (`s1` and `p1`, respectively, for the sepal length and petal length) and then plots a scatter plot. The `plot` function then plots the sepal length along the y axis and the petal length along the x axis. The axis labels can be assigned with the argument for `xlab` and `ylab`, respectively, and the plot can be given a title with the `main` argument. The plot (in the **A** section of the previous screenshot) thus shows that the two variables follow a more or less positive correlation.

Scatter plots are not useful if one has to look for a trend, that is, for how a value is evolving along the indices, which can prove that it is time for a dynamic process. For example, the expression of a gene along time or along the concentration of a drug. A line diagram is a better way to show this. Here, we first generate a set of 250 artificial values and their indices, which are the values on the x scale. For these values, we assume a normal distribution, as we saw in the previous section. This is then plotted (as shown in the **B** section of the previous screenshot). It is possible to add more lines to the same plot using the `line` function as follows:

```
> lines(density(x), col="red")
```

A boxplot can be an interesting visualization if we want to compare two categories or groups in terms of their attributes that are measured in terms of numbers. They depict groups of numerical data through their quartiles. To illustrate, let us consider the iris data again. We have the name of the species in this data (column 5). Now, we want to compare the sepal length of these species with each other, such as which one has the longest sepal and how the sepal length varies within and between species. The data table has all this information, but it is not readily observable.

The `boxplot` function has the first argument that sorts out what to plot and what to plot against. This can be given in terms of the column names of the data frame that is the second argument. Other

arguments are the same as other `plot` functions. The resulting plot (as shown in the **C** section of the previous screenshot,) shows three boxes along the x axis for the three species in our data. Each of these boxes depicts the range quartiles and median of the corresponding sepal lengths.

The histogram (the **D** section of the previous screenshot) describes the distribution of data. As we see, the data is normally distributed with a mean of 3; therefore, the plot displays a bell shape with a peak of around 3. To see the bell shape, try the `plot(density(x))` function.

There's more...

You can use the `plot` function for an entire data frame (try doing this for the iris dataset, `plot(iris)`). You will observe a set of pair-wise plots like a matrix. Beside this, there are many other packages available in R for different high-quality plots such as `ggplot2`, and `plotrix`. They will be discussed in the next chapters when needed. This section was just an attempt to introduce the simple plot functions in R.

Working with PubMed in R

Research begins with a survey of the related works in the field. This can be achieved by looking into the literature available. PubMed is a service that provides the option to look into the literature. The service has been provided by NCBI-Entrez databases (shown in the following screenshot) and is available at

<https://www.ncbi.nlm.nih.gov>. R provides an interface to look into the various aspects of the literature via PubMed. This section provides a protocol to handle this sort of interface. This recipe allows the searching, storing, and mining, and quantification meta-analysis within the R program itself, without the need to visit the PubMed page every time, thus aiding in analysis automation. The following screenshot shows the PubMed web page for queries and retrieval:

The screenshot displays the PubMed website interface. At the top, there is a navigation bar with links for "NCBI", "Resources", and "How To". The main header features the "PubMed" logo, a search bar with a "Search" button, and a "Sign in to NCBI" link. Below the header, a large banner for "PubMed" states: "PubMed comprises more than 22 million citations for biomedical literature from MEDLINE, life science journals, and online books. Citations may include links to full-text content from PubMed Central and publisher web sites." To the right of the banner is a "PubReader" section with the text: "A whole new way to read scientific literature at PubMed Central" and an image of a tablet displaying a document.

Below the banner, there are three columns of links:

- Using PubMed**
 - [PubMed Quick Start Guide](#)
 - [Full Text Articles](#)
 - [PubMed FAQs](#)
 - [PubMed Tutorials](#)
 - [New and Noteworthy](#)
- PubMed Tools**
 - [PubMed Mobile](#)
 - [Single Citation Matcher](#)
 - [Batch Citation Matcher](#)
 - [Clinical Queries](#)
 - [Topic-Specific Queries](#)
- More Resources**
 - [MeSH Database](#)
 - [Journals in NCBI Databases](#)
 - [Clinical Trials](#)
 - [E-Utilities](#)
 - [LinkOut](#)

At the bottom of the page, there is a "You are here: NCBI > Literature > PubMed" breadcrumb trail and a "Write to the Help Desk" link. Below this, there are five columns of links:

- GETTING STARTED**
 - [NCBI Education](#)
 - [NCBI Help Manual](#)
 - [NCBI Handbook](#)
 - [Training & Tutorials](#)
- RESOURCES**
 - [Chemicals & Bioassays](#)
 - [Data & Software](#)
 - [DNA & RNA](#)
 - [Domains & Structures](#)
 - [Genes & Expression](#)
 - [Genetics & Medicine](#)
 - [Genomes & Maps](#)
 - [Homology](#)
 - [Literature](#)
 - [Proteins](#)
 - [Sequence Analysis](#)
 - [Taxonomy](#)
 - [Training & Tutorials](#)
- POPULAR**
 - [PubMed](#)
 - [Nucleotide](#)
 - [BLAST](#)
 - [PubMed Central](#)
 - [Gene](#)
 - [Bookshelf](#)
 - [Protein](#)
 - [OMIM](#)
 - [Genome](#)
 - [SNP](#)
 - [Structure](#)
- FEATURED**
 - [Genetic Testing Registry](#)
 - [PubMed Health](#)
 - [GenBank](#)
 - [Reference Sequences](#)
 - [Map Viewer](#)
 - [Human Genome](#)
 - [Mouse Genome](#)
 - [Influenza Virus](#)
 - [Primer-BLAST](#)
 - [Sequence Read Archive](#)
- NCBI INFORMATION**
 - [About NCBI](#)
 - [Research at NCBI](#)
 - [NCBI News](#)
 - [NCBI FTP Site](#)
 - [NCBI on Facebook](#)
 - [NCBI on Twitter](#)
 - [NCBI on YouTube](#)

Getting ready

It's time to get practical with what we learned so far. For all the sessions throughout this book, we will use the Linux terminal. Let's start at the point where it begins, by getting into the bibliographic data. The `RISmed` package facilitates the analyses of NCBI database content, written and maintained by Kovalchik. The following are the requirements to work with PubMed in R:

- An Internet connection to access the PubMed system
- An `RISmed` package installed and loaded in the R session; this can be done easily with the following chunk of code:

```
> install.packages("RISmed")  
> library(RISmed)
```

To look into the various functionalities, you can use the following `help` function of R:

```
> help(package="RISmed")
```

How to do it...

The following steps illustrate how to search and retrieve literature from PubMed using R:

1. Load the default data in `RISmed`. The default data available with the package is for `myeloma`. Start by loading this data as follows:

```
> data(myeloma)
```

2. Now, find the `myeloma` object that was loaded in your R workspace with the `ls()` command as follows (note that you might see some other objects as well besides the `myeloma`):

```
> ls()  
[1]myeloma
```

3. To see the contents of the `myeloma` object, use the following command:

```
> str(myeloma)
```

4. Take a look at each element of the data using `RISmed`, which has the following specific functions:

```
> AbstractText(myeloma)  
> Author(myeloma)  
> ArticleTitle(myeloma)  
> Title(myeloma)  
> PMID(myeloma)
```

5. Create your customized query. What we had till now for `RISmed` was based on a precompiled data object for the package. It will be interesting to discuss how we can create a similar object (for instance, `cancer`) with a query of our choice. The function that facilitates the data retrieval and creation of a `RISmed` class is as follows:

```
> cancer <- EUtilsSummary("cancer[ti]",  
  type="research", db="pubmed")  
> class(cancer)
```

How it works...

Before we go deep into the functioning of the package, it's important to know about E-utilities. `RISmed` uses E-utilities to retrieve data from the Entrez system. In this chapter, however, our focus is on bibliographic data. E-utilities provide an interface to the Entrez query and database system. It covers a range of data, including bibliographic, sequences, and structural data managed by the NCBI. Its functioning is very simple; it sends the query through a URL to the Entrez system and retrieves the results for the query. This enables

the use of any programming language, such as Perl, Python, or C++, to fetch the XML response and interpret it. There are several tools that act as a part of E-utilities (for details, visit <http://www.ncbi.nlm.nih.gov/books/NBK25497/>). However, for us, Efetch is interesting. It responds to an input of unique IDs (for example, PMIDs) for a database (in our case, PubMed) with corresponding data records. The Efetch utility is a data retrieval utility that fetches data from the Entrez databases in the requested format, with the input in the form of corresponding IDs. The `RISmed` package uses it to retrieve literature data.

The first argument of the `EUtilsSummary` function is the query term, and in the square brackets, we have the field of the query term (in our case, `[ti]` is for title, `[au]` is for author, and so on). The second argument is the E-utility type and the third one refers to the database (in our case, PubMed). The `myeloma` object of the `RISmed` class has information about the query that was made to fetch the object from PubMed, the PMIDs of the search hits, and the details of the articles, such as the year of publication, author, titles, abstract and journal details, and associated mesh terms.

All these commands of the package used with/for `myeloma` return a list of lengths equal to the number of hits in the data object (in our case, the `myeloma` object). Note that the `Title` function returns the title of the journal or publisher and not the title of the article, which can be seen with `ArticleTitle`.

Now, let's take a look at the structure of the `cancer` object that we created:

```
> str(cancer) # As in August 2013
Formal class 'EUtilsSummary' [package "RISmed"]
with 5 slots
..@ count : num 575447
..@ retmax : num 1000
..@ retstart : num 0
..@ id : chr [1:1000] "23901442" "23901427"
"23901357" "23901352" ...
```

```

..@ querytranslation: chr "cancer[ti]"

> cancer@id[1:10]
[1] "23905205" "23905156" "23905117" "23905066"
    "23905042" "23905012"
[7] "23904955" "23904921" "23904880" "23904859"

```

The `cancer` object consists of five slots: `count`, `retmax`, `retstart`, `id`, and `querytranslation`. These variables are assigned as the subclasses of the `cancer` object. Therefore, in case we need to get the PMIDs of the retrieval, we can do so by getting values for the `id` component of the `cancer` object with the following code:

```
> cancer@id
```

One important point that should be noted is that this query retrieved only the first 1000 hits out of 575,447 (the default value for `retmax`). Furthermore, one should follow the policies for its uses to avoid overloading the E-Utilities server. For further details, read the policy at <http://www.ncbi.nlm.nih.gov/books/NBK25497/>.

Now, we are left with the creation of a `RISmed` object (the same as the `myeloma` object). To get this, we use the following `EUtilsGet` function:

```

> cancer.ris <- EUtilsGet(cancer, type="efetch",
db="pubmed")
> class(cancer.ris)

```

This new object, `cancer.ris`, can be used to acquire further details as explained earlier.

For more operations on PubMed, refer to the `help` file of the `RISmed` package.

A drawback of the `RISmed` package is that, in some cases, due to the incorrect parsing of text, the values returned could be inaccurate. A more detailed explanation of this package can be found by seeking

help for the package as described earlier. To get to know more about the [RISmed](#) package, refer to the CRAN package home page at <http://cran.r-project.org/web/packages/RISmed/RISmed.pdf>.

Some interesting applications on the [RISmed](#) package are available on the R Psychologist page at <http://rpsychologist.com/an-r-script-to-automatically-look-at-pubmed-citation-counts-by-year-of-publication/>.

Retrieving data from BioMart

So far, we discussed bibliographic data retrieval from PubMed. R also allows the handling of other kinds of data. Here, we introduce another R package called `biomaRt`, developed by Durinck and co-workers. It provides an interface to a collection of databases implementing the BioMart suite (<http://www.biomart.org>). It enables the retrieval of data from a range of BioMart databases, such as Ensembl (genes and genomes), Uniprot (information on proteins), HGNC (gene nomenclature), Gramene (plant functional genomics), and Wormbase (information on *C. elegans* and other nematodes).

Getting ready

The following are the prerequisites:

- Install and load the `biomaRt` library
- Create the data ID or names you want to retrieve (usually gene names—we use BRCA1 for a demo in this recipe), such as the ID or the chromosomal location

How to do it...

Retrieving the gene ID from HGNC involves the following steps, where we first set the mart (data source), followed by the retrieval of genes from this mart:

1. Before you start using `biomaRt`, install the package and load it into the R session. The package can directly be installed from Bioconductor with the following script. We discuss more about Bioconductor in the next chapter; for the time being, take a look at the following installation:

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

```
> biocLite("biomaRt")
> library(biomaRt)
```

2. Select the appropriate mart for retrieval by defining the right database for your query. Here, you will look for human `ensembl` genes; hence, run the `useMart` function as follows:

```
> mart <- useMart(biomart = "ensembl", dataset
= "hsapiens_gene_ensembl")
```

3. Now, you will get the list of genes from the `ensembl` data, which you opted for earlier, as follows:

```
> my_results <- getBM(attributes =
c("hgnc_symbol"), mart = mart)
```

4. You can then sample a few genes, say 50, from your retrieved genes as follows:

```
> N <- 50
> mysample <- sample(my_results$hgnc_symbol,N)
> head(mysample)
[1] "AHSG"      "PLXNA4"    "SYT12"     "COX6CP8"
"RFK"       "POLR2LP"
```

5. The `biomaRt` package can also be used to retrieve sequences from the databases for a gene, namely `"BRCA1"`, as shown in the following commands:

```
> seq <- getSequence(id="BRCA1",
type="hgnc_symbol", seqType="peptide", mart =
mart)
> show(seq)
```

6. To retrieve a sequence that specifies the chromosome position, the range of the position (upstream and downstream from a site) can be used as well, as follows:

```
> seq2 <- getSequence(id="ENST00000520540",
```

```
type='ensembl_transcript_id',seqType='gene_flank',upstream = 30,mart = mart)
```

7. To see the sequence, use the `show` function as follows:

```
> show(seq2)
gene_flank  ensembl_transcript_id
1 AATGAAAAGAGGTCTGCCCGAGCGTGCGAC
ENST00000520540
```

How it works...

The `source` function in R loads a new set of functions in the source file into the R-session; do not confuse it with a package. Furthermore, during installation, R might ask you to update the Bioconductor libraries that were already installed. You can choose these libraries as per your requirements.

The `biomaRt` package works with the `BioMart` database as described earlier. It first selects the mart of interest (that is why, we have to select our mart for a specific query). Then, this mart is used to search for the query on the `BioMart` database. The results are then returned and formatted for the return value. The package thus provides an interface for the BioMart system.

Thus, the `biomaRt` package can search the database and fetch a variety of biological data. Although the data can be downloaded in a conventional way from its respective database, `biomaRt` can be used to bring the element of automation into your workflow for bulk and batch processing.

There's more...

The `biomaRt` package can also be used to convert one gene ID to other types of IDs. Here, we illustrate the conversion of RefSeq IDs to gene symbols with the following chunk of code:

```

> mart <- useMart(biomart = "ensembl", dataset =
"hsapiens_gene_ensembl")
> geneList <- read.csv("mylist.csv")
> results <- getBM(attributes = c("refseq_mrna",
"hgnc_symbol"), filters = "refseq_mrna", values =
geneList[,2], mart = mart)
> results

```

	refseq_mrna	hgnc_symbol
1	NM_000546	TP53
2	NM_001271003	TFPI2
3	NM_004402	DFFB
4	NM_005359	SMAD4
5	NM_018198	DNAJC11
6	NM_023018	NADK
7	NM_033467	MMEL1
8	NM_178545	TMEM52

Though `biomaRt` enables the conversion of the ID for biological entities for most of our work, in this book, we also use some other packages that are handier and will be illustrated in the next chapter.

See also

- The BioMart home page at <http://www.biomart.org> to know more about BioMart
- The *BioMart: driving a paradigm change in biological data management* article by Arek Kasprzyk (<http://database.oxfordjournals.org/content/2011/bar049.full>)
- The *BioMart and Bioconductor: a powerful link between biological databases and microarray data analysis* article by Durinck and others (<http://bioinformatics.oxfordjournals.org/content/21/16/3439.long>), which discusses the details of the `biomaRt` package

Chapter 2. Introduction to Bioconductor

In this chapter, we will cover the following recipes:

- Installing packages from Bioconductor
- Handling annotation databases in R
- Performing ID conversions
- The KEGG annotation of genes
- The GO annotation of genes
- The GO enrichment of genes
- The KEGG enrichment of genes
- Bioconductor in the cloud

Introduction

Bioconductor is an open source project with a collection of R packages for the analysis and comprehension of high-throughput genomic data. It has tools and packages for microarray preprocessing, experimental design, integrative and reproducible approaches to bioinformatics tasks, and so on. The Bioconductor website (<http://www.bioconductor.org>) provides details on installation, package repository, help, and other documentation.

The following screenshot shows the Bioconductor web page with a red box that indicates an important information tab that tells you more about the workflows and other help pages:

The screenshot shows the Bioconductor website. The top navigation bar is teal with links for Home, Install, Help, Developers, and About. The 'Help' link is highlighted with a red box. A red arrow points from this box to a note: 'Information on workflows vignettes etc.'. Below the navigation bar, the page is divided into three columns. The left column is titled 'About Bioconductor' and describes the project. The middle column is titled 'Use Bioconductor for...' and lists three categories: Microarrays, Annotation, and Cloud-enabled cis-eQTL search and annotation. The right column contains detailed descriptions for each category. The 'Recent Courses' section is also visible at the bottom right.

Bioconductor
OPEN SOURCE SOFTWARE FOR BIOINFORMATICS

Search:

[Home](#) [Install](#) [Help](#) [Developers](#) [About](#)

Information on workflows vignettes etc.

About Bioconductor

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. Bioconductor uses the R statistical programming language, and is open source and open development. It has two releases each year, [749 software packages](#), and an active user community. Bioconductor is also available as an [Amazon Machine Image \(AMI\)](#).

Use Bioconductor for...

- **Microarrays**
Import Affymetrix, Illumina, Nimblegen, Agilent, and other platforms. Perform quality assessment, normalization, differential expression, clustering, classification, gene set enrichment, genetical genomics and other workflows for expression, exon, copy number, SNP, methylation and other assays. Access GEO, ArrayExpress, Biomart, UCSC, and other community resources.
- **Annotation**
Use microarray probe, gene, pathway, gene ontology, homology and other annotations. Access GO, KEGG, NCBI, Biomart, UCSC, vendor, and other sources.
- **Cloud-enabled cis-eQTL search and annotation**
Bioconductor can be used to perform detailed analyses of relationships between DNA variants and mRNA abundance. Genotype (potentially imputed) and expression data are organized in packages prior to analysis, using very concise representations. SNP and probe filters can be specified at run time. Transcriptome-wide testing can be carried out using multiple levels of concurrency (chromosomes to nodes, genes to cores is a common approach). Default outputs of the cloud-oriented interface ciseqByCluster include FDR for all SNP-gene pairs in cis, along with locus-specific annotations of genetic and genomic contexts.
- **Recent Courses**
Explore material from courses in [2013](#) and [2014](#).

This chapter presents some recipes to help you solve bioinformatics problems specifically related to metadata in biology with the packages available in Bioconductor.

Installing packages from Bioconductor

Bioconductor, as mentioned before, has a collection of packages and software that serve different purposes in analyzing biological data or other objectives in bioinformatics. In order to be able to use them, you must install and load them into their R environment or R session. In this recipe, we will talk about these installations.

Getting ready

The following are the prerequisites for the recipe:

- An R session that is running
- An Internet connection to reach a remote Bioconductor repository

How to do it...

To install Bioconductor libraries in R, we perform the following steps:

1. Use the `biocLite.R` script to install Bioconductor libraries (Version 2.12). To install core libraries, type the following commands in an R command window:

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

2. Install specific libraries, for example, `GenomicFeatures` and `AnnotationDbi` with the following command:

```
> biocLite(c("GenomicFeatures",
"AnnotationDbi"))
```

3. To install a package from its source, simply type the following command:

```
> biocLite("IRanges", type="source")
```

4. To look into the code for `biocLite`, type the following commands:

```
> biocLite  
function (pkgs = c("Biobase", "IRanges",  
"AnnotationDbi"), suppressUpdates = FALSE,  
suppressAutoUpdate = FALSE, siteRepos =  
character(), ask = TRUE, ...)
```

How it works...

A web accessible installer script is called `biocLite`. It gets loaded into the R session on specifying the source Bioconductor repository. The `biocLite` function checks for the availability of the instructed libraries in the repository and installs or updates it. It ensures that the libraries from the appropriate version of Bioconductor are installed and that all of them remain up to date. The function accepts attributes such as `pkgs` for the names of packages (as a vector of package names) and updates the existing package option.

Setting the source for `biocLite` loads the `BiocInstaller` package and therefore, there is no need to load the library again. In order to change the default Bioconductor and CRAN mirrors, you can simply use the following functions:

```
> chooseBioCmirror()  
> chooseCRANmirror()
```

Installing the `BiocInstaller` package from a source is necessary in cases where you need to get binaries that are suitable for your version of R.

In our case, we installed two libraries, `GenomicFeatures` and `AnnotationDbi`, from Bioconductor. The `GenomicFeatures` library is for transcript-centric annotations, whereas the `AnnotationDbi` library provides an interface and database connection functions to annotate data packages.

There's more...

You can use the `setRepositories()` function to choose package repositories interactively. This function without any arguments will give you the options of the package repositories that you can choose from; otherwise, we can set a repository of our choice by specifying an appropriate value for the `addURLs` argument. For details, type `?setRepositories` in your R terminal or console.

The `biocLite` function can be instructed to use the CRAN repository too. For further information regarding the functionalities of `biocLite`, use the help file as follows:

```
> ?biocLite
```

Note

We can upgrade the installed Bioconductor libraries with the `biocLite("BiocUpgrade")` function. Furthermore, whenever we install a new Bioconductor library, the session will ask for the installation of all the available updates (for the installed libraries); at this point, we can choose to update or not according to our convenience. However, we can also set the `suppressUpdates` argument accordingly while running the `biocLite` function. For details, type `?biocLite`.

Handling annotation databases in R

While working with high-throughput data, for example, array data, we deal with probes. In order to really understand the data, we need a gene level. This section explains and presents the methods that can be used to perform such a mapping from probe IDs to the genes on a human Affymetrix data.

Getting ready

The prerequisites for this recipe are as follows:

- Prerequisite packages such as `Biobase` and `annotate` can be installed from Bioconductor as explained in the previous recipe
- An R session with Bioconductor basic packages should be installed
- A list of genes to be annotated

How to do it...

To annotate our data, perform the following steps:

1. First install the annotation data for the chip (Affymetrix Human Genome U133) from Bioconductor and load it in the R session as follows:

```
> source("http://Bioconductor.org/biocLite.R")
> biocLite("hgu133a.db")
> library("hgu133a.db")
```

2. Now, we must assign the Entrez IDs from the database to a list for mapping purposes. This creates an Entrez ID map for the probe in the chip as follows:

```
> myMap <- hgu133aENTREZID
```

3. Then, we get the mapped probes using the keys in the assigned database as a list with the following commands:

```
> mapped_probes <- mappedkeys(myMap)
```

4. Finally, we can retrieve the Entrez IDs of the probes that of interest (in our case, we take the first five probes) as follows:

```
> MyEntrez <-  
as.list(myMap[mapped_probes[1:5]])
```

5. The IDs can then be fetched by accessing the respective indices of the list as follows:

```
> mylength=3 # for entire list  
mylength=length(MyEntrez)  
for(i in 1:mylength){  
  MyEntrez[[i]]  
}
```

Make sure that `mylength` is less than `length(MyEntrez)` to avoid the occurrence of an error while running the code.

How it works...

The `Biobase` package contains standardized data structures to represent genomic data. The `annotate` package provides interface functions for getting data out of specific metadata libraries in Bioconductor as well as functions to query different web services. The chip annotation package for Affymetrix Human Genome U133 Set is `hgu133a.db`. It contains mappings between an Affymetrix's identifiers and accessions. To check the contents of this package, use the `ls` command as follows:

```
> ls("package:hgu133a.db")
```

This gives a list of 36 different objects within the package. Most of these database packages have such objects containing mappings of the identifiers to different IDs including gene symbols, Entrez identifiers, `RefSeq` IDs, and so on.

The `hgu133aENTREZID` object provides mappings of manufacturer (Affymetrix) identifiers to a vector of Entrez gene identifiers. The `mappedkeys` function gets the probe identifiers that are mapped to an Entrez gene ID as an object and later the selected IDs can be extracted from this object (in our case, the first one).

There's more...

R and Bioconductor provide various databases for the annotation of biological entities such as a gene and proteins with certain attributes. These attributes range from **gene ontology (GO)** to chip (microarray) annotations. There are several other kinds of mappings available with this package, and every other mapping is available for these annotation packages. The `ls("package:<package_name>")` command gives an overview of all such possible mappings in any package.

To know more about these annotation packages, visit the Bioconductor web page at <http://www.Bioconductor.org/packages/release/data/annotation/>.

Performing ID conversions

A common task when analyzing data in bioinformatics is the conversion of IDs and gene symbols for proper representation and visualization. The previous recipe talked about such a mapping. However, it may not always be related to a certain chip type or manufacturer's ID. This recipe presents an example conversion of Entrez gene IDs to gene symbols and vice versa.

Getting ready

The prerequisites for the recipe are as follows:

- An R session with the `AnnotationDbi` package loaded in the session
- A list of gene symbols to be converted to Entrez IDs

How to do it...

The task first requires you to load the relevant mapping database (names starting with `org` and ending with `.db` with a two letter acronym for the organism name in between). In this recipe, we use the database for human, which is `org.Hs.eg.db` (`Hs` for *Homo sapiens*).

1. First, we need to install and load the `org.Hs.eg.db` package using `biocLite` as follows:

```
> biocLite("org.Hs.eg.db")  
> library(org.Hs.eg.db)
```

2. Now, we assign the genes of interest as Entrez gene IDs to a vector as follows:

```
> myEIDs <- c("1", "10", "100", "1000",
```



```
"37690")
```

3. To get the gene symbols of these Entrez gene IDs, we use the `mget` function and the `org.Hs.egSYMBOL` mapping function as follows:

```
> mySymbols <- mget(myEIDs, org.Hs.egSYMBOL,
  ifnotfound=NA)
```

4. The `mySymbols` list can be observed as a named vector with the `unlist` command as follows:

```
> mySymbols <- unlist(mySymbols)
> mySymbols
1      10      100      1000      37690
"A1BG" "NAT2" "ADA"  "CDH2"      NA
```

5. In order to change the names back to Entrez IDs, we first select the informative symbols (removing `NA` from our list) and then assign them to a vector as follows:

```
> y <- mySymbols[!is.na(mySymbols)]
```

6. Finally, we simply use another mapping function called `org.Hs.egSYMBOL2EG` in the same manner that we used earlier by typing the following commands:

```
> myEIDs <- unlist(mget(mySymbols,
  org.Hs.egSYMBOL2EG, ifnotfound=NA))
> myEID
A1BG NAT2 ADA CDH2
"1"  "10" "100" "1000"
```

How it works...

Installing and loading the `org.Hs.eg.db` package brings the human Entrez annotation data into the R session. Then, an `org` Entrez ID vector is created and assigned to an object. The `mget` function is inherited from the `AnnotationDbi` package and allows you to manipulate the `Bimap` object (see `?AnnotationDbi::mget` for further

details). The `mget` function accepts a vector and the mapping function (in our case, `org.Hs.egSYMBOL`) to map Entrez IDs to symbols, and `org.Hs.SYMBOL2EG` for the other way round. The `ifnotfound` attribute for the `mget` function simply returns `NA` for the IDs that are not mapped in the database. At the same time, multiple mapping is also possible, for example, a symbol can be mapped to more than one Entrez identifier.

The `mget` function returns a named list object. The names are the same as the input vectors (Entrez IDs or symbols). The `unlist` function simply presents the output as a named vector.

There's more...

There are various other kinds of mappings possible. To get to know more about these, enter `ls("package:org.Hs.eg.db")`. For example, it is also possible to convert Entrez IDs to `RefSeq`, `unigene`, and `Ensembl` IDs and vice versa with the `ls("package:org.Hs.eg.db")` function.

Similar packages exist for different organisms such as yeast (*Saccharomyces cerevisiae*), drosophila (*Drosophila melanogaster*), and arabidopsis (*Arabidopsis thaliana*).

The KEGG annotation of genes

Kyoto Encyclopedia of Genes and Genomes (KEGG) provides data resources of known biological pathways. Annotating a gene or a set of genes/proteins with their respective KEGG pathways gives us a clue about the biological role of the genes/proteins. This recipe will explain an approach to showing the enzyme involved in the pathway via annotations using the KEGG package in R.

Getting ready

We need the following prerequisites to start with this recipe:

- An R session with an installed KEGG package
- A list of genes to be annotated

How to do it...

We can annotate the biomolecules with the corresponding KEGG pathways using the following steps:

1. First, we need to install and load the KEGG library into our R session, as we have been doing before, using the following commands:

```
> biocLite("KEGG.db")  
> library(KEGG.db)
```

2. In the case that the list of genes we have are in the form of gene symbols, it is always a good practice to convert them into official Entrez IDs as follows:

```
> myEID <- unlist(mget(y, org.Hs.SYMBOL2EG,  
  ifnotfound=NA))  
> myEID <- as.character(myEID)
```

3. These Entrez IDs can then be mapped onto the KEGG pathway identifiers by using the following commands:

```
> kegg <- mget(as.character(myEID),  
KEGGEXTID2PATHID, ifnotfound=list(NA))
```

4. Further more, in the same way, the retrieved pathway identifiers can be used to fetch the pathway names as follows:

```
> myPathName <- unlist(mget(x,  
KEGGPATHID2NAME, ifnotfound=list(NA)))
```

5. Another way to do this is to simply use the semantic information of the KEGG knowledge base as follows:

```
> KEGGPATHID2NAME$"00140"  
[1] "C21-Steroid hormone biosynthesis"
```

6. You can also retrieve the involved gene for a pathway in a similar way (as at April 2014) by using the following command:

```
> KEGGPATHID2EXTID$hsa00140  
[1] "1109" "1583" "1584" "1585" "1586" "1589"  
"3283" "3284"  
[9] "3290" "3291" "6718"
```

7. For pathway information related to other organisms, the organism code for humans (`hsa`) can be simply replaced with the code for the organism of choice (in our case, `sce` for yeast) as follows:

```
> KEGGPATHID2EXTID$sce00072  
[1] "YML126C" "YPL028W"
```

How it works...

The preceding function works in the same manner as the other annotation packages explained earlier. However, as stated

previously, the annotation is based on the indexes of the database in the package, which in the case of this database can be mapped with names as well. The KEGG pathways use an identifier system where the name of an organism is embedded within the identifier. The letters refer to Homo sapiens' `KEGGPATHID2EXTID$hsa00140` function that converts the KEGG path ID to the Entrez gene that is mapped to the pathway ID, `hsa00140`.

There's more...

The `KEGGREST` library can query the KEGG pathway database via the KEGG REST server for all the genes, enzymes, compounds, and reactions that are involved in the interactions in the desired pathway by using the following commands:

```
source("http://Bioconductor.org/biocLite.R")
biocLite("KEGGREST")
library(KEGGREST)
genes <- keggGet("hsa:1045")
```

To get to know more about the KEGG REST server, visit the KEGG API page at <http://www.kegg.jp/kegg/docs/keggapi.html>. The details on the KEGG pathways can be found in the *KEGG for integration and interpretation of large-scale molecular data sets* article by Kanehisa and others (<http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3245020/?report=reader>).

The GO annotation of genes

GO provides a controlled vocabulary of terms to describe a gene product's characteristics via their annotation. GO terms for biomolecules reflect what is currently known about a gene in terms of function, physiology, and cellular location. Thus, GO terms provide a good description of a dataset under study at the biological level. In this recipe, we will illustrate how to annotate a set of genes with the associated GO terms.

Getting ready

The prerequisites for GO annotation are as follows:

- An R session with an installed `go.db` package
- A list of genes to be annotated

How to do it...

To annotate a set of genes with GO terms, perform the following steps:

1. First, we need to load the annotation database library named `org.Hs.eg.db` into our R session, as we have been doing before, by using the following command:

```
> library(org.Hs.eg.db)
```

2. We assume that the genes are in the form of Entrez gene IDs; otherwise, as explained before, it's good to convert them as follows:

```
> myEIDs <- c("1", "10", "100")
```

3. We now use these keys to find the GO annotation for each of our genes as follows:

```
> myGO <- org.Hs.egGO[[myEIDs[1]]]
```

4. We can do this in one go for all the genes using the `mget` function as follows:

```
> myGO_All <- mget(myEIDs, org.Hs.egGO)
```

5. It is also possible to perform the mapping in the reverse manner, that is, from GO term ID to Entrez genes as follows:

```
> GOgenes <-  
org.Hs.egGO2ALLEGS[["GO:0008150"]]
```

6. To get genes for all the input GO terms, we again use the `mget` function as follows:

```
> GOterm <- c("GO:0008150", "GO:0001909")  
> GOgenes_All <- mget(GOterm,  
org.Hs.egGO2ALLEGS)
```

How it works...

This is again a simple case of database mapping as we have seen in the previous cases. However, here the result of annotation is a list of all available GO terms for each input gene. Each element of this list is another list (we may call it a sublist) that contains the following three elements:

- The GO term ID, named `GOID`.
- Evidence for the annotation, named `Evidence`.
- The GO category (depending on the availability of annotations) for the annotation term and can be either **Biological Process (BP)**, **Molecular Function (MF)**, or **Cellular Components (CC)**. This element in the result is named `Ontology`.

The fifth and sixth steps simply return the Entrez IDs of all the genes that are annotated with the query GO term as a vector.

There's more...

Another database package named `GO.db` can provide the mappings between GO identifiers and their respective terms. It can be installed from Bioconductor as any other package. To do the mappings, one needs a list of GO term IDs and keywords for each slot as follows:

```
> biocLite(GO.db)
> library(GO.db)
if(length(xx) > 0){
  myGO =list(GOID(xx[[1]]), Term(xx[[1]]),
  Synonym(xx[[1]]), Secondary(xx[[1]]),
  Definition(xx[[1]]), Ontology(xx[[1]]))
}
myGO
```

See also

- The *Gene Ontology: tool for the unification of biology* article by the Gene Ontology Consortium(http://www.geneontology.org/GO_nature_genetics_2000.pdf), which provides detailed information on the Gene Ontology project
- The Gene Ontology project's home page (<http://www.geneontology.org>)
- The Bioconductor page at <http://www.bioconductor.org/packages/2.13/data/annotation/html/GO.db.html>, which provides more information on the `GO.db` package

The GO enrichment of genes

Once we have a set of genes/proteins in the required format, it is extremely interesting to annotate them with their roles in biology via functional enrichments such as GO terms. The aim of enrichment is to find a significant set of functions (here, we use GO terms or pathways in the case of pathway enrichment) for the given genes.

Enrichment is achieved by looking for the over-representation of GO terms within the group of genes. The significance of over-representation is calculated with statistical tests such as the Fisher test.

Talking about GO, pathways, and statistical tests, is beyond the scope of this book; refer to the relevant citations for details.

Getting ready

The following are the prerequisites to go ahead with the steps for the functional enrichment of genes:

- An R session with Bioconductor packages installed and loaded
- An Internet connected system to install a new relevant package from the repository
- A dataset to perform the analysis

How to do it...

Let's carry out the following steps to perform the GO enrichment analysis:

1. Install and load the `topGO` library and the `ALL` dataset for GO enrichment analysis with the help of the following commands:

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

```
> biocLite(c("topGO", "ALL"))
> library(topGO)
```

2. Load the dataset to be analyzed for enrichment into the R session (in our case, the `ALL` dataset from the `ALL` library) as follows:

```
> library(ALL)
> data(ALL)
> data(geneList)
```

3. Set the annotation for the Affymetrix chip data in the `ALL` dataset using the following commands:

```
> affyLib <- paste(annotation(ALL), "db",
  sep=".")
> library(package=affyLib,
  character.only=TRUE)
```

4. We can check how many genes we can consider as differentially expressed genes ($p\text{-value} < 0.01$) from the `geneList` data for GO enrichment by using the following command:

```
> sum(topDiffGenes(geneList))# should come as
50
```

5. When the input data and libraries are ready, we must first create a `topGOdata` object. This is followed by an enrichment analysis as follows:

```
> myGOData <- new("topGOdata", ontology="BP",
  allGenes=geneList, geneSel=topDiffGenes,
  nodeSize=10, annot= annFUN.db,
  affyLib=affyLib)
```

6. The enrichment test (Fisher test) can be performed as follows:

```
> Myenrichment_Fisher <- runTest(myGOData,
  algorithm= "classic", statistic="fisher")
```

7. To check the enrichment scores in our results, type the following command:

```
> score(Myenrichment_Fisher)
```

8. We can try other enrichment tests such as the **Kolmogorov–Smirnov test (KS test)** by typing the following command:

```
> Myenrichment_KS <- runTest(myGOData,  
algorithm= "classic", statistic="ks")
```

9. We can see the results for any of these tests in the form of a table with the following `GenTable` function:

```
> enrich_table <- GenTable(myGOData,  
classicFisher=Myenrichment_Fisher,topNodes =  
20)  
> head(enrich_table)
```

How it works...

The `topGOdata` object consists of genes, their scores, GO annotation, and GO hierarchy information as a nested list. All the enrichment analysis is performed on this data. To create the `topGOdata` object, you require some other inputs such as `nodeSize` to prune the GO tree (GO has a hierarchical tree structure) and a library to map the genes to GO.

Once we have the object, we go ahead with a test for enrichment. In our case, we use a Fisher test to find the enriched genes. The input to the `runTest` function is a `topGOdata` object along with the test statistics (here, the Fisher test) and the algorithms for the test (in our case, the classical algorithm). The function returns a `topGOresult` object as follows:

```
> result # (As on April 15th 2014)  
Description: Simple session  
Ontology: BP
```

```
'classic' algorithm with the 'fisher' test
860 GO terms scored: 46 terms with p < 0.01
Annotation data:
Annotated genes: 315
Significant genes: 50
Min. no. of genes annotated to a GO: 10
Nontrivial nodes: 776
```

The other testing methods can be used as well such as the KS test, as explained earlier, and can then be combined as follows, resulting in a data frame with ID, scores, and GO terms:

```
> allRes <- GenTable(myGOData, classicFisher =
Myenrichment_Fisher, classicKS = Myenrichment_KS,
ranksOf = "classicFisher", topNodes = 10)
> str(allRes)
```

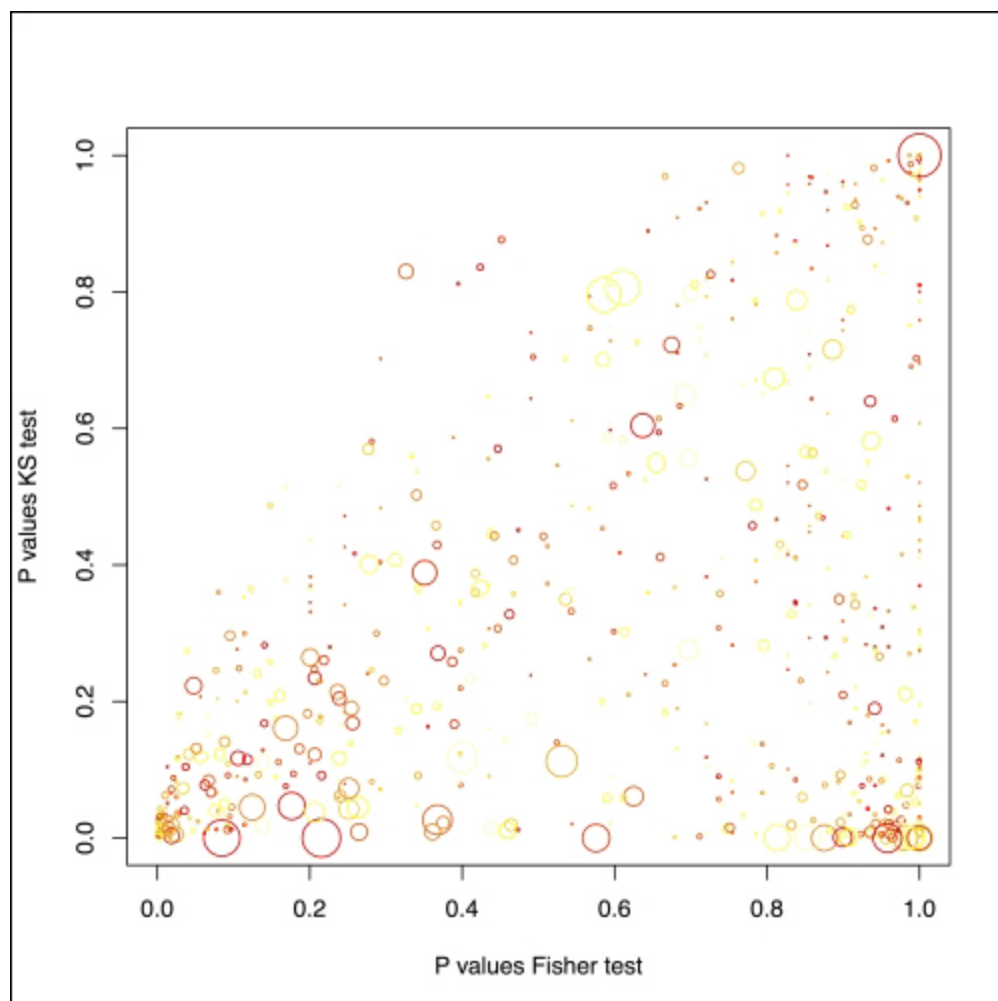
There's more...

You can observe the performance of the tests against each other by plotting the p-values against each other. To do so, we need to extract the GO terms and the corresponding number of genes along with other information with the `termStat` function using the following commands:

```
> gostat <- termStat(myGOData,
names(score(Myenrichment_Fisher)))
> plot(score(Myenrichment_Fisher),
score(Myenrichment_KS)
[names(score(Myenrichment_Fisher))], xlab="P
values Fisher test", ylab="P values KS test", cex=
(gostat$Annotated/max(gostat$Annotated))*4,
col=heat.colors(gostat$Significant))
```

Here, the `gostat` object was used to create mappings for size and color in the plot. The `heat.colors` function generates a set of colors based on the values passed as arguments. The following is a scatter plot that shows the comparison between the enrichment of genes for two different methods namely the Fisher test and KS test. The

scatter plot shows that the results depend on the test used. The color and size of the circle in the following screenshot represent the significance and the number of genes annotated, respectively:



Furthermore, a related GO graph can be observed with the `showSigOfNodes` function as follows (however, this requires the `Rgraphviz` package installed on your computer):

```
> showSigOfNodes(myGOData,  
  score(Myenrichment_Fisher), firstSigNodes=5,  
  useInfo="all")
```

See also

- The Wolfram MathWorld page at <http://mathworld.wolfram.com/FishersExactTest.html> for the Fisher test and <http://mathworld.wolfram.com/Kolmogorov-SmirnovTest.html> for the KS test, which provide detailed information on the statistical test that is used to perform the enrichment test

The KEGG enrichment of genes

The last recipe introduced us to the enrichment term for gene ontology. However, it is possible to do similar things with pathways or, more precisely, the KEGG pathways. This recipe illustrates such an enrichment test for a set of genes.

Getting ready

The following are the prerequisites to enrich a gene set with the KEGG pathways:

- An R session with the `clusterProfiler` package installed as follows:

```
> source("http://Bioconductor.org/biocLite.R")  
> biocLite("clusterProfiler")
```

- The list of genes (Entrez IDs) to be enriched

How to do it...

For KEGG enrichment, we need to perform the following steps:

1. To perform the KEGG pathway enrichment, first load the `clusterProfiler` library:

```
> library(clusterProfiler)
```

2. Then, create a list of genes from our dataset (in our case, `gcSample` data, which is a list of five sets), as follows:

```
> data(gcSample)  
> genes <- gcSample[[3]]
```

3. Once we have our gene set ready, we can do the enrichment tests as follows:

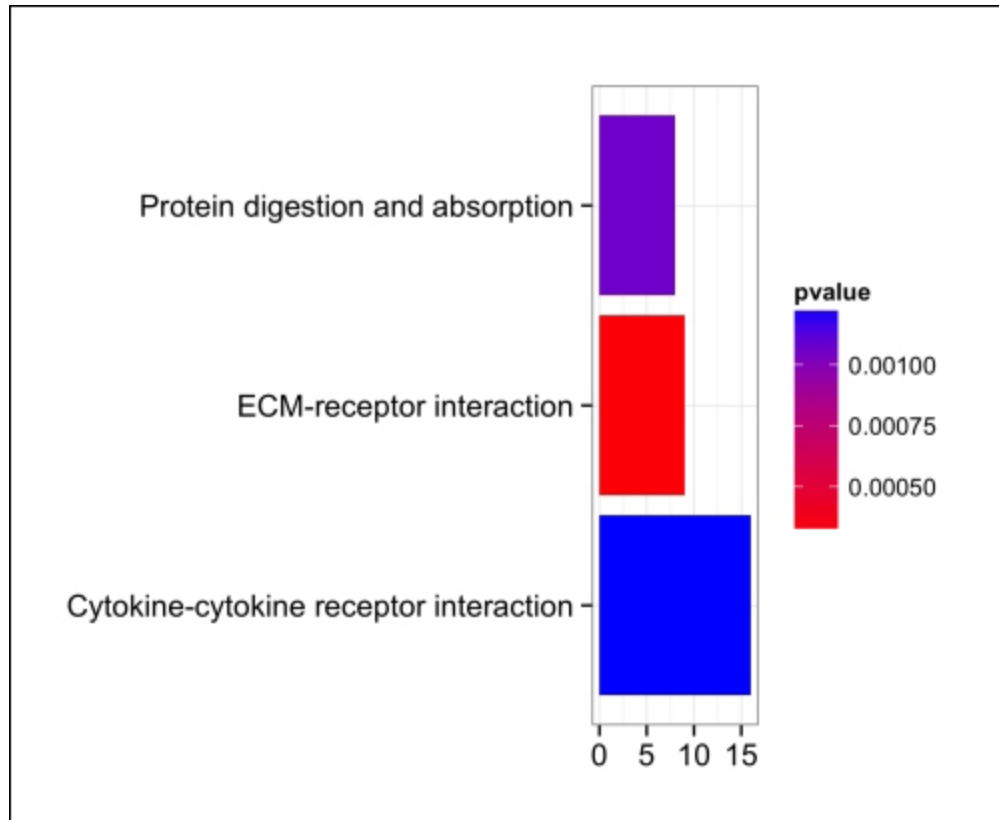
```
> kegg_enrichment <- enrichKEGG(genes,  
  pvalueCutoff=0.01)
```

How it works...

The `enrichKEGG` function takes the list of genes as input and returns the enrichment of KEGG categories within the gene set. The `pvalueCutoff` parameter filters the output to a certain p-value for significant enrichment. The enrichment can be visualized in the terms of a graph (as seen in the previous screenshot) and requires an `igraph` package to plot as follows:

```
> summary(kegg_enrichment)  
> plot(kegg_enrichment)
```

The following plot shows the KEGG pathway enrichment of genes (the p-values for the different pathways on a colored scale and the number of genes in each KEGG pathway along the x axis):



Enrichment analysis based on KEGG and GO has been a powerful tool for computational biologists. There are many other methods and tools. GSEA is one of the most popular tools and is considered as a pioneer in the field.

See also

- The *GSEA* article, which provides a decent theoretical background on enrichment
- The *Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles* article by Subramanian and others (<http://www.pnas.org/content/102/43/15545.long>)

Bioconductor in the cloud

Cloud computing offers on-demand computing and storage resources along with scalability and low maintenance costs. Cloud computing refers to the computing that involves a large number of computers connected via a communication network such as the Internet. It enables distributed computing over a network, providing you with the ability to run a program or an application on many connected computers at the same time. Cloud computing has become an appealing, **high performance computing (HPC)** platform for ad hoc analytics because it offers a high-performance computing solution with scalability and low maintenance when compared to setting up an HPC infrastructure. In the following cases, running a cloud-based Bioconductor can be useful (or required):

- You do not want to install Bioconductor on your personal machine.
- You can have a long-running task and you don't want to use your CPU. Tasks such as running a huge cluster or dynamic simulation for mathematical models in biology require higher RAM or (and) CPU configurations depending on the dataset and algorithm. You do not want to keep your machine busy for a long time and therefore, you can think of using clouds.
- You need to parallelize a task usually for reasons described previously. Although it is possible to parallelize a task locally using the packages such as `mccore` or `foreach`, you might want to use more nodes than locally available.
- Run R in your web browser (using RStudio Server). The RStudio Server provides a browser-based interface to R, which runs on a remote server and acts as an IDE.

These points make the integration of Bioconductor tools over cloud an interesting option. This recipe will provide basic information on such applications.

Getting ready

The following are the prerequisites for the recipe:

- You need an **Amazon web services (AWS)** account (<http://aws.amazon.com>). This may require credit card information (and even payments for paid services).
- A server with appropriate firewall settings.

How to do it...

The cloud setup for Bioconductor can be achieved in the following steps (note that this recipe will use AWS as a cloud service provider):

1. First, users should create an AWS account. For detailed information, visit <http://aws.amazon.com/getting-started/>.
2. Now, launch **Amazon Machine Image (AMI)** by visiting <https://console.aws.amazon.com/cloudformation/>. Here, you need to enter your login credentials for AWS.
3. AWS needs a stack name and a template to start with. (Download and save the `.pem` file.)
4. Later, we need to provide the version for Bioconductor when asked; it is advisable to go with the default values.
5. Once all the parameters are provided, you can review your sets and create the stack. The AMI will be ready soon.
6. Use `ssh` or `PuTTY` to connect to the server with an appropriate address. The `ssh` function is as follows:

```
$ ssh -X -i bioconductor-my-bioc.pem root@ec2-50-16-120-30.compute-1.amazonaws.com
```

7. Once you are able to access the terminal through `ssh` or any other means, start R by simply typing `R`.
8. Another way to run R via the web server is using the RStudio Server that provides a user friendly GUI.

9. To start RStudio, type in your web instance followed by 8787 in your browser. Your web instance provided by the cloud service provider will look like a web address, such as http://****.****.amazonaws.com. Thus, RStudio starts with http://****.****.amazonaws.com:8787 in the browser.

How it works...

Cloud-based R uses the sources of computing resources of the remote machines in the cloud. It makes it possible to use R as a browser kind of environment. The data and file can be moved by using RStudio GUI (for Windows) or with the `scp` command in SSH. Everything else functions in a similar way as in a local machine R. An important fact to be noted is that the use of PuTTY requires a `ppk` file that can be generated using the `ppm` file and PuTTYgen (a PuTTY utility). For more details, visit the AWS help page at <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html>.

See also

- The SAS and R blog (<http://sas-and-r.blogspot.it/2012/02/rstudio-in-cloud-for-dummies.html>), which gives some information on running R in the cloud
- The *Cloud Computing in Bioinformatics* chapter in *Distributed Computing and Artificial Intelligence* written by Bajo and others (http://link.springer.com/chapter/10.1007%2F978-3-642-14883-5_19) in order to learn more on cloud computing

Chapter 3. Sequence Analysis with R

In this chapter, we will cover the following recipes:

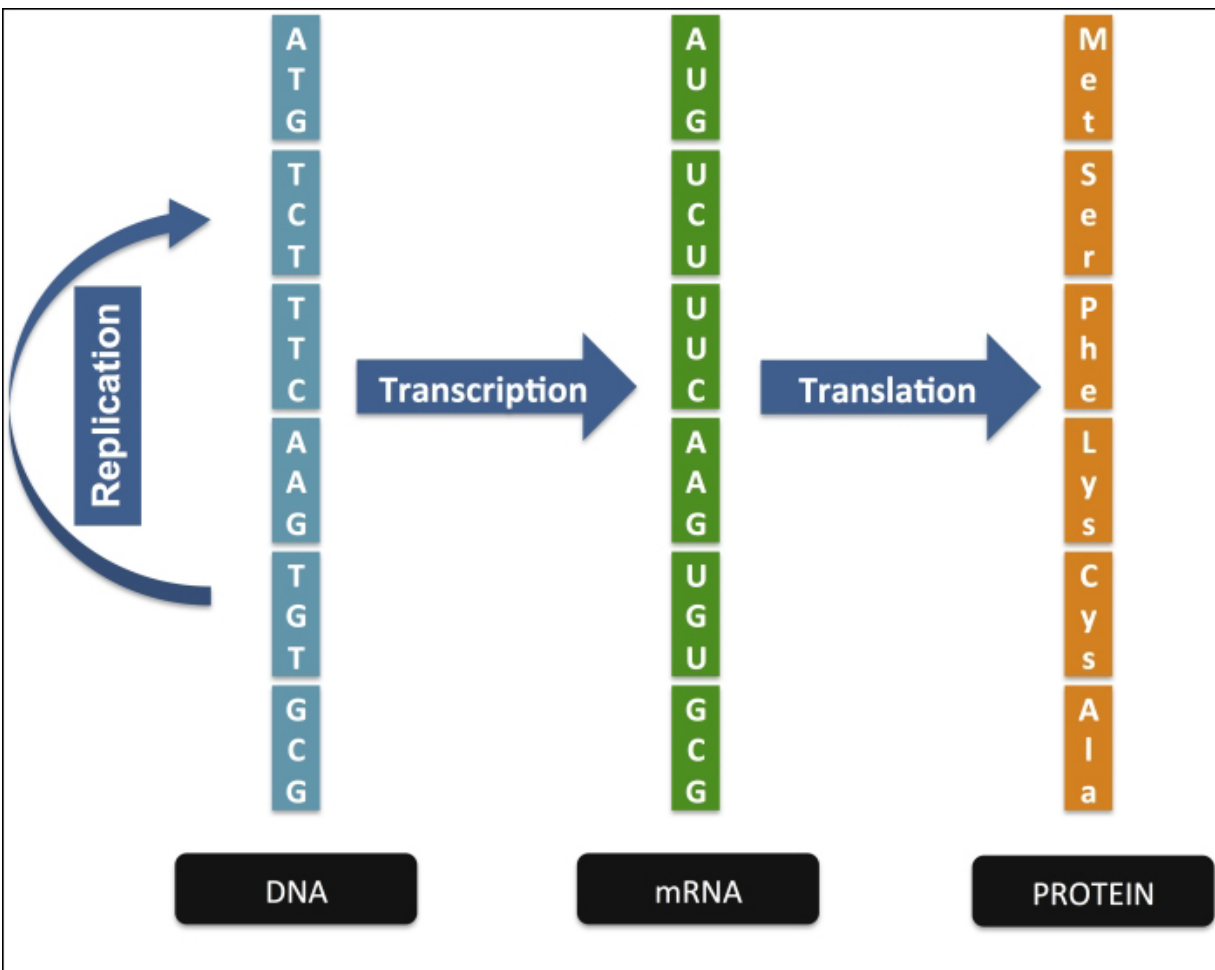
- Retrieving a sequence
- Reading and writing the FASTA file
- Getting the detail of a sequence composition
- Pairwise sequence alignment
- Multiple sequence alignment
- Phylogenetic analysis and tree plotting
- Handling BLAST results
- Pattern finding in a sequence

Introduction

Nucleic acids and protein sequences are extremely ubiquitous to the way we imagine biology today. Nucleic acids represent genes, RNAs, and so on, whereas proteins are considered the building blocks of life. These biomolecules actually depict the information content in the living system. They are represented in terms of a sequence of characters. The famous central dogma of molecular biology can be perceived as one type of a set of characters being converted into another at the sequence level. For example, **messenger RNAs (mRNAs)** are converted into proteins or, rather, polypeptides via translation. Thus, it is the sequence of DNA that will eventually determine the protein sequence. This makes sequence analysis important for various applications ranging from the comparison of biomolecules for studying evolution, mutation to identification of interesting sites in biomolecules, and so on.

It was the huge growth in sequence data that paved the way for the evolution of the bioinformatics domain. Data is usually a stream of characters. The characters in the case of DNA are ATGC (these

represent the bases in the nucleotides—remember that RNA has U instead of T), whereas in the case of proteins, they consist of the letters that represent amino acids. Computer scientists sometimes call these sequences the bitcode of life. Analyzing the properties of these molecules as well as understanding the underlying mechanism of the transformation of one type of information into another or the next level is one of the keys to deciphering the living system. The following figure illustrates how genetic information is transferred to a protein. From the information science perspective, it can be seen as the transfer of one type of sequence to another via the process of transcription and translation. The reverse transcription has not been included in the following figure:



The central dogma of molecular biology at the sequence level

Sequence analysis is the most basic task in bioinformatics. In general, it refers to processing sequence data of DNA or proteins to harness the underlying information about the function, structure, or evolution of the biomolecule. Analyzing sequences allows us to find similarities or dissimilarities between them for comparison purposes. One can use the sequence data to identify the chemical properties of the sequence based on its content. Furthermore, we can also use them to compute their structure. In addition, they are also applied to create a homology-based model to predict unknown three-dimensional structures of proteins that can be used in drug discovery. However, the purposes are not limited to these mentioned aspects.

This chapter will introduce the different kinds of sequence analysis that can be done with R and the corresponding R libraries and packages.

Retrieving a sequence

To start the task of sequence analysis, the first thing we need is a sequence of DNA or protein. Such sequence data can be retrieved either by visiting the database hosting page via a browser or by accessing the data mart form within R via corresponding commands/functions. We had a look at some of the methods in the previous chapter. In this recipe, we introduce some more methods to retrieve a sequence.

Getting ready

Retrieving sequences from databases requires the following:

- Access to the database via the Internet
- A sequence ID or keyword for searching
- The `biomaRt` and `seqinr` libraries installed and loaded in the R session

How to do it...

To fetch a sequence of the data via R, perform the following steps:

1. Get a sequence of the data. The simplest way to get a sequence of interest is to visit the database web page and manually download it, or to get it via the `bioMart` package that was explained in [Chapter 1](#), *Starting Bioinformatics with R*.
2. Sequences can also be retrieved via the `seqinr` package. To use the `seqinr` package, first load the library with the following command:

```
> install.packages("seqinr")  
> library(seqinr)
```


3. Choose the data bank you need to fetch the sequence. First, you can check the available data banks and then choose one of them. For instance, in our case, we choose `genbank` as follows:

```
> choosebank()
[1] "genbank"      "embl"      "emblwgs"
"swissprot"
[5] "ensembl"      "ensembl60" "hogenom"
"hogenomdna"
[9] "hovergendna"  "hovergen"   "hogenom5"
"hogenom5dna"
[13] "hogenom4"     "hogenom4dna"
"homolens"      "homolensdna"
[17] "hobacnucl"    "hobacprot"   "phever2"
"phever2dna"
[21] "refseq"       "greviews"
"bacterial"     "protozoan"
[25] "ensbacteria"  "ensprotists"
"ensfungi"      "ensmetazoa"
[29] "ensplants"    "mito"
"polymorphix"   "emglib"
[33] "taxobacgen"   "refseqViruses"

> choosebank("genbank")
```

4. Once you have the data bank open, query the bank with the search of your interest using the following `query` command (note that this step takes time to complete):

```
> query("BRCA1", "SP=Homo sapiens AND
K=BRCA1")
```

5. Have a look at the returned value of the `query` command by looking at the various components of the object as follows:

```
> attributes(BRCA1)
$names
[1] "call"      "name"      "nelem"
"typelist" "req"       "socket"
$class
[1] "qaw"
```

6. To check the set of all of the sequences that were retrieved, type the following command and get the name, length, and other attributes for every available sequence in the query:

```
> BRCA1$req
```

7. To fetch a specific sequence by its accession number, use the `AC` attribute in the `query` command as follows:

```
> query("BRCA1", "SP=Homo sapiens AND  
AC=U61268")
```

8. To fetch a specific sequence from the `query` object, use the following `getsequence` command (in our case, we use it for the first sequence in the `BRCA` object):

```
> myseq <- getSequence(BRCA1$req[[1]])
```

9. Take a look at the object that was created with the following `getsequence` command:

```
> myseq  
[1] "a" "t" "g" "g" "a" "t" "t" "t" "a" "t"  
"c" "t" "g" "c" "t" "c" "t" "t" "c"  
[20] "g" "c" "g" "t" "t" "g" "a" "a" "g" "a"  
"a" "g" "t" "a" "c" "a" "a" "a" "a"  
[39] "t" "g" "t" "c" "a" "t" "t" "a" "a" "t"  
"g" "c" "t" "a" "t" "g" "c" "a" "g"  
[58] "a" "a" "a" "a" "t" "c" "t" "t" "a" "g"  
"a" "g" "t" "g"
```

10. It is always advisable to close the data bank after the query in order to avoid multiple open banks. Do this with the following `closebank` function:

```
> closebank()
```

How it works...

The working of `biomaRt` has been explained in [Chapter 1, Starting Bioinformatics with R](#). The `seqinr` package works in a similar way. The selected data bank is queried for the set of input search terms (`k` for keyword and `sp` for species) and all the available sequences for the search attributes are assigned to the defined object that was named and returned (in our case, `BRCA1`). Out of these lists, we can filter out the irrelevant hits by looking at the attributes of the list, such as the name of the sequence. In our presented code, we used the position of the relevant sequence in the list. The `getsequence` command then fetches the sequence from all the available information during retrieval.

There's more...

The sequence retrieval from the remote database, as seen in the `seqinr` package, requires permission to access the database via the network ports. It might happen that some of the ports in your network are blocked and you receive corresponding errors (usually displayed as `Error in socketConnection(.....)`). For this, you need to contact your network administrator and get the blocks removed. Besides the `keyword` attribute for `query`, there are other possible attributes that can be used. Note that you can seek help for this just by typing `?query` or `?seqinr::query` in your R console. The use of double colons (`::`) with the package name on the left and the function name on the right is required to avoid confusion when the function shares the namespace in two (or more) different packages. It simply specifies that the function to be used comes from a particular package. You can fetch the annotation (the other attributes of a sequence) with the following command:

```
> annots <- getAnnot(BRCA1$req[[1]])
```

See also

- The detailed description of GenBank at <https://www.ncbi.nlm.nih.gov/genbank/>.

- The *GenBank* article by Benson and others at <http://nar.oxfordjournals.org/content/41/D1/D36.long>, which provides a description of the latest release of GenBank. (Note that the sequin known for submitting sequences to GenBank has nothing to do with `seqinr`. This is just an acronym for sequences in R.)
- The `seqinr` package's web page at <http://pbil.univ-lyon1.fr/software/seqinr/home.php?lang=eng>, which provides addition information about the package.

Reading and writing the FASTA file

The FASTA format is a simple and widely used format for storing biological (DNA or protein) sequences. It begins with a single-line description that starts with a > symbol. The description consists of virtually anything regarding the sequence but usually carries the sequence name, ID, name of the species, name of the author, and so on. The line that follows carries the sequences (nucleotide or protein). The following is an example of a FASTA file for a protein sequence taken from **Protein Data Bank (PDB)** or (ID- 2BQ0):

```
>2BQ0 : A | PDBID | CHAIN | SEQUENCE
MTMDKSELVQKAKLAEQAERYDDMAAAMKAVTEQGHELSNEERNLLSVAY
KNVVGARRSSWRVISSIEQKTERNEKKQMGKEYREKIEAELQDICNDVL
ELLDKYLI PNATQPESKVFY LKMGDYFRYLSEVASGDNKQTTVSNQQAY
QEAFEISKEMQPTHPIRLGLALNFSVFYIEILNSPEKACSLAKTAFDEA
IAELDTLNEESYKDSTLIMQLLRDNLTWTSENQGDGENLYFQ
```

FASTA is a standard format for storing sequence data in databases such as GenBank. It is important to have an idea about importing this file format into an R session and exporting it from one. This recipe aims to address this issue.

Getting ready

This recipe requires a FASTA file in the local or remote directories to be read into the R session. For this, you can simply copy and paste a FASTA sequence into a text editor and save it as a `.fasta` file (even `.txt` should serve the purpose). Besides this, the `seqinr` library will be required, which we discussed in the previous recipe.

How to do it...

In order to read a FASTA file into an R workspace and write a FASTA file with retrieved sequences, perform the following steps:

1. Load the `seqinr` library with the help of the following command:

```
> library(seqinr)
```

2. Read the FASTA file from its location (provide the complete path if not in the current working directory) as follows:

```
> mysequence <- read.fasta(file =  
"myfasta.fasta")
```

3. Choose GenBank as the data bank for your queries with the following command:

```
> choosebank("genbank")
```

4. Now, query GenBank for a keyword of interest (in our case, we look for `BRCA1`) by typing the following command:

```
> query(listname = "BRCA1", query="SP=homo  
sapiens AND K=BRCA1")
```

5. To look for the sequence identifiers, extract the name attribute of the sequences as follows:

```
> myseqs <- getSequence(BRCA1)  
> mynames <- getName(BRCA1)
```

6. Check the number of hits by simply checking the length of the `mynames` object that contains the names of the sequences found in the hits as a vector, with the help of the following command:

```
> length(mynames)  
[1] 103 # as on April 17th, 2014
```

7. Try writing the retrieved sequence data carrying information in different fields as multiple sequences (this is only applicable if more than one sequence was retrieved, otherwise you will receive a single sequence) in a FASTA file named `MyBRCA.fasta` as follows:

```
> write.fasta(myseqs, mynames, file.out =  
"MyBRCA.fasta")
```

8. As suggested earlier, close the data bank after the query by typing the following command in order to avoid working with multiple banks:

```
> closebank()
```

How it works...

The beginning of the sequence is thus recognized with this pattern and so is the other information about the sequence.

While querying for the desired sequence (`BRCA1`) in step 4, we define the search keyword `K` and the species we are interested in as `SP`. These terms are embedded as one query string text and used to search the data bank. The `query` function not only assigns the query but also creates an R object of the `qaw` class, which is a list with six fields, each containing different types of information on the retrieved results. The value argument, `listname`, should be a quoted character that will be used as the name of the `qaw` object that is created. The `query` argument actually carries the search request to be parsed by the data bank. It consists of selection criteria for the hits and is specified with different fields; for example, we used `K` and `SP` here, as explained earlier. Note that within the `query` argument, there should be no spaces around the `=` sign. There are many other fields in the `query` argument that can be used. To find out more about them, check the updated list at <http://pbil.univ-lyon1.fr/databases/acnuc/cfonctions.html#QUERYLANGUAGE> or

type `?query` in the R console (note that this provides the fields available only for the package version).

See also

- The FASTA file format at <https://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml>, which provides detailed information about the FASTA file
- The *Online synonymous codon usage analyses with the ade4 and seqinR packages* article by Charif and others at <http://bioinformatics.oxfordjournals.org/content/21/4/545>, which provides information about the applications that use the `seqinr` package

Getting the detail of a sequence composition

Once we have retrieved a sequence, we need to know more about it; for example, we need to know the nucleotide or amino acid frequency, and **Guanine and Cytosine nucleotide bases (GC content)**. To illustrate, if we have repetitive bases, say, AAAAAA, the sequence is not very interesting because of low information content. The contents of a sequence also help in determining certain properties of the entire molecule, for example, the acid basicity hydrophobicity in proteins based on the amino acids present in the sequence.

An interesting aspect is the GC content in a nucleotide. It refers to the fraction of Guanine (G) and Cytosine (C) in the sequence, and certain genomes, especially among the bacteria, show a significant difference on this scale and variations in terms of genomic regions. To illustrate, some Actinobacteria can have more than 70 percent of GC, whereas some Proteobacteria can have less than 20 percent of GC. Furthermore, the GC content is also used to predict the annealing temperature of the sequence during PCR experiments. These aspects make the analysis of the contents in a sequence important. In this recipe, we discuss how to get such information out of sequence data.

Getting ready

The task in this recipe requires the following:

- A `seqinr` package in the R session
- The sequence to be analyzed that will be retrieved during the steps

How to do it...

To compute some details of the sequence composition, perform the following steps:

1. Load the `seqinr` library by typing the following command:

```
> library(seqinr)
```

2. Get the sequence of interest from the database (Entrez gene, PDB, and so on) or file using `read.fasta`. In our case, we will fetch two RNA polymerase (beta subunits) sequences from GenBank (note that the sequences belong to two different species, *Mycobacterium tuberculosis* (Actinobacterium) and *Escherichia coli* (Proteobacterium)) as follows:

```
> choosebank("genbank")
> query(listname = "actino",
query="SP=Mycobacterium tuberculosis AND
K=rpoB") # Gets the M.tuberculi sequences
> query(listname = "proteo",
query="SP=Escherichia coli AND K=rpoB") # Gets
the E.coli sequences
```

3. Take a look at the summary of the sequences, such as the names and length, with the help of the following commands:

```
> actino$req # 694 Sequences as on April 17th
2014
> proteo$req # 171 Sequences as on April 17th
2014
```

4. From these sequences, choose one sequence from each species (*M. tuberculosis* and *E. coli*) that will be compared. In our case, we go with JX303316 in the former and AE005174.RPOB in the latter species (the sequence index numbers 644 and 1, respectively, in the corresponding `qwa` objects) as follows:

```
> myActino <- getSequence(actino$req[[644]])
> myProteo <- getSequence(proteo$req[[1]])
```

5. To compute the number of each base in the sequence, use the generic `table` function as follows:

```
> table(myActino)
myActino
  a    c    g    t
679 1084 1190 584
> table(myProteo)
myProteo
  a    c    g    t
1013 1026 1090 899
```

6. To get the fraction for each base, do it in on a line by dividing the outcome of the `table` function by the length of the sequence (multiply it with 100 to get the result in a percentage) as shown in the following command:

```
> table(myProteo)/length(myProteo)
myProteo
a          c          g          t
0.2514271531 0.2546537602 0.2705385952
0.2231322909
```

7. If you decide to manually copy the sequence from the web page of the database, it will be considered a character object; it must be converted into a vector of characters as follows:

```
> myseq <- "AAAATGCAGTAACCCATGCCAAAATGCAGTAA"
> myseq <- strsplit(myseq, "")
> myseq <- unlist(myseq)
```

8. To compute the individual frequencies of the nucleotides, use the following `table` function in the same way as used earlier:

```
> table(myseq)
myseq
A    C    G    T
15    7    5    5
```

9. Use similar commands for the protein sequence as follows:

```

myseq <-
"MTMDKSELVQKAKLAEQAERYDDMAAAMKAVTEQGHELSNEERNL
LSVAYKNVVGARRSSWRVISSIEQKTERNEKKQMGKEYREKIEAE
LQDICNDVLELLDKYLIPNATQPESKVFYLMKMGDYFRYLSEVASG
DNKQTTVSNQQAYQEAFEISKKEMQPTHPIRLGLALNFSVFYYEIL
NSPEKACSLAKTAFDEAIAELDTLNEESYKDSTLIMQLLRDNLWT
SENQGDEGENLYFQ"
> myseq <- strsplit(myseq, "")
> myseq <- unlist(myseq)

```

Alternatively, use the following command in one line:

```

> myseq <- s2c(myseq)

> table(myseq)
myseq
 A  C  D  E  F  G  H  I  K  L  M  N  P  Q  R  S
T  V  W  Y
20  2 13 29  7  8  2 10 20 24  8 14  5 16 10
18 12 11  2 12

```

10. Moving on the interesting aspects of the GC content of the sequences, simply use the `GC` command from the `seqinr` package for the two sequences that were retrieved as follows:

```

> GC(myActino)
[1] 0.6429177
> GC(myProteo)
[1] 0.5253227

```

11. Create a simple bar plot while comparing for different sequences in the GC content with a one-line command to visualize the results, as shown in the following command:

```

> barplot(c(Actinobacteria= GC(myActino),
Proteobacteria= GC(myProteo)), main="GC
content in different bacteria")= GC(myProteo),
main="GC content in different bacteria")

```

12. To know the frequency of every possible pair of nucleotides in the sequence, use the `count` function, as shown in the following example, for every character pair (you can also do this for triples and so on by choosing the right value for the `wordsize` argument):

```
> seqinr::count(myActino, wordsize=2)
  aa  ac  ag  at  ca  cc  cg  ct  ga  gc  gg
gt  ta  tc
131 232 194 121 219 287 431 147 291 340 325
234  38 225
  tg  tt
239  82
```

How it works...

In this recipe, we used the RNA polymerase beta from two bacterial species. The reason behind the selection of specific sequences from each species (JX303316 and AE005174.RPOB) was that they are the reads that represent similar proteins and are of comparable length. The functions in this recipe are mostly based on character matching and counting. The function runs through the input vector of characters, which is our sequence here, and increments the count of the word every time it encounters it. The final counts are then returned for each word. The count can be made for different word sizes, say, 2, 3, and so on. (However, a word size of only up to 3 is frequently used in the case of nucleotides as it represents the triplets in codons.)

The `gc` function of `seqinr` uses a similar method, but as an extension, it computes the fraction of G and C in the nucleotide. We can do this manually using the values of C (index 2) and G (index 3) as follows:

```
> myGC <- sum(table(myseq)[2], table(myseq)
[3])/sum(table(myseq))
> myGC
[1] 0.375
```

You can further express it as a percentage by multiplying the result with 100 as follows:

```
> myGC*100  
37.5
```

In our examples, to compare GC content of RNA polymerase from two different species, we need to see how it can vary even among closely related species, which proves the point that we made during the introductory section of the chapter. However, there can be an exception in our finding, and we need to perform such an analysis on more species in order to reach a statistically significant result. We talked mostly about nucleotide sequences in this recipe; protein sequences will be dealt with in detail in the next chapter.

Tip

If the connection to a data bank shows an error in the second or further queries (though it worked well for the first), restart your R session (obviously after saving the existing objects). If it does not work even after that, you will need to contact your system administrator.

See also

- The *DNA helix: the importance of being GC-rich* article by Vinogradov AE at <http://nar.oxfordjournals.org/content/31/7/1838.full>, which provides detailed information on the importance of GC content in nucleic acids

Pairwise sequence alignment

As mentioned in the introductory section of this chapter, while analyzing genes or proteins sequences, we often need to compare them to know their similarities and differences. This serves the purposes from various perspectives, such as for evolutionary studies and to understand the structure and function of a novel sequence by comparing it to the known one. To know whether the two gene/protein sequences we are studying are similar or different at the quantitative level, we measure their similarities.

Before making comparative statements about two sequences, we have to produce a pairwise sequence alignment. Pairwise alignment refers to the optimal way of arranging two sequences in order to identify regions of similarity therein. In other words, we need to find the optimal alignment between the two sequences. Some of the questions that pop up are as follows:

- How should we optimize such an arrangement of two sequences?
- How should we score matches and mismatches?
- Should we score protein sequences differently than we score DNA sequences?

There are several algorithms and metrics that try to answer these questions and are designed to perform such alignments, and opting for one of them depends on the biological question that needs to be answered. There are global alignment methods that aim to align every residue in the sequences and are used when sequences are similar and of comparable length (they need not be equal). An example of such a method is the Needleman-Wunsch algorithm. We also have a local alignment technique that attempts to align regions of high similarity in the sequences, and the Smith-Waterman algorithm is an example of such a technique.

There are many other methods and algorithms for sequence alignment, though discussing them in detail is beyond the scope of this book. Nevertheless, we will put forward some of the key concepts in this recipe. Here, we will introduce some R-based methods to perform the pairwise alignment of two sequences.

Getting ready

Sequence alignment requires the following:

- The `Biostrings` library installed and loaded in the R session
- Sequences of our interest

The `Biostrings` library can be installed and loaded as we did the other Bioconductor libraries. We will need the sequences to be aligned.

The alignment also computes an alignment score for which we need to define the scoring system that depends on the type of sequence (nucleotide or protein). This will be discussed in the upcoming parts of the recipe.

How to do it...

To do a pairwise sequence alignment, perform the following steps:

1. Start with installing and loading the `Biostrings` library by typing the following commands:

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

2. Read the two sequences. In the following example, we have shown manually typed-in sequences for convenience, but the same method can be used for other types of sequences as well:


```
> sequence1 <- "GAATTCGGCTA"  
> sequence2 <- "GATTACCTA"
```

3. Assign penalties for the mismatch and gap to get a scoring matrix for the nucleotides as follows:

```
> myScoringMat <-  
  nucleotideSubstitutionMatrix(match = 1,  
    mismatch = -1, baseOnly = TRUE)  
> myScoringMat
```

4. Assign gap penalties for the alignments as follows:

```
> gapOpen <- 2  
> gapExtend <- 1
```

5. Run the following `pairwiseAlignment` function to perform a global alignment for the sequences:

```
> myAlignment <- pairwiseAlignment(sequence1,  
  sequence2,      substitutionMatrix =  
  myScoringMat, gapOpening = gapOpen,  
  gapExtension = gapExtend, type="global",  
  scoreOnly = FALSE)  
> myAlignment
```

6. In the case of protein sequences, we have scoring matrices that are called substitution matrices. To see them, use the following function:

```
> data(package="Biostrings")  
Data sets in package 'Biostring':  
BLOSUM100 Scoring matrices  
BLOSUM45 Scoring matrices  
BLOSUM50 Scoring matrices  
BLOSUM62 Scoring matrices  
BLOSUM80 Scoring matrices
```

7. Assign/select one of these substitution matrices to align the protein sequences as follows:

```
> data(BLOSUM62)
> subMat <- "BLOSUM62"
```

8. Now, perform the alignment of the protein sequences with the selected substitution matrix as follows:

```
> sequence1 <- "PAWHEAE"
> sequence2 <- "HEAGAWGHE"
> myAlignProt <- pairwiseAlignment(sequence1,
  sequence2, substitutionMatrix = subMat,
  gapOpening = gapOpen, gapExtension =
  gapExtend, type="global", scoreOnly = FALSE)
```

How it works...

The alignment recipe can be divided into two phases: the scoring matrix formulation and the alignment itself. In the nucleotide alignment, the function defining the scoring matrix creates a 4 x 4 matrix for the nucleotides. The diagonal alignment in the matrix represents the match and the other alignments represent the mismatch as shown in the following example:

	A	C	G	T
A	1	-1	-1	-1
C	-1	1	-1	-1
G	-1	-1	1	-1
T	-1	-1	-1	1

The alignment we used was a global alignment, by setting the type argument to `global`. Every time the alignment finds a match or mismatch, the corresponding score is added to the existing score (set to 0 at the beginning). Furthermore, we also use a `gapOpening` penalty of 2 and a `gapExtension` penalty of 1 for our scoring task (see the `gapOpening` and `gapExtension` arguments respectively). The gap opening and gap extension penalties used every time introduce a gap or extend an existing gap to get a better alignment. These parameters are used to run the Needleman-Wunsch algorithm for the global alignment. The algorithm uses the dynamic programming

approach to find the optimal global alignment between two sequences with the scoring that we defined here. Let's look at the alignment created for our sequences:

```
> show(myAlignment)
Global PairwiseAlignmentsSingleSubject (1 of 1)
pattern: [1] GAATTCGGCTA
subject: [1] GATTAC--CTA
score: 1
```

We can see that in the alignment produced, we have two gaps inserted to get the optimal alignment.

Protein scoring matrices are larger and more complicated looking since there is a different probability for each amino acid pairing. Therefore, several matrices have been proposed for amino acid substitutions in protein sequences. Therefore, in the case of protein sequences, the substitution matrices define the scoring matrix from the `Biostrings` library. It contains several matrices (as can be seen by typing `data(package="Biostrings")` in the R console). Based on substitution, the **Blocks Substitution Matrix (BLOSUM)** is a family of matrices, which is computed based on substitutions encountered in a conserved set of proteins. The number that is used as a suffix for each member of the BLOSUM family indicates the percentage similarity in the proteins used to compile the matrix. Thus, the higher the BLOSUM number, the more similar are the proteins, and it can be better used to analyze closely related proteins. Another popular family of the substitution matrix is PAM. In this recipe, we used BLOSUM62, which is popularly used to compare highly similar proteins. We get the following alignment:

```
> show(myAlignProt)
Global PairwiseAlignmentsSingleSubject (1 of 1)
pattern: [1] P---AW-HE
subject: [1] HEAGAWGHE
score: 14
```

Let's see what happens if we run a local alignment with our peptide (protein) sequence:

```
> myAlignProt <- pairwiseAlignment(sequence1,
sequence2, substitutionMatrix = subMat, gapOpening
= gapOpen, gapExtension = gapExtend,
type="local", scoreOnly = FALSE)
> show(myAlignProt)
Local PairwiseAlignmentsSingleSubject (1 of 1)
pattern: [2] AW-HE
subject: [5] AWGHE
score: 25
```

We can see that the local alignment simply returns the highly similar regions in both the sequences using the Smith-Waterman algorithm.

An interesting visualization for comparing two protein or nucleotide sequences in order to identify regions of similarity between them is a dot plot. It is a two-dimensional grid/matrix with two axes, representing the sequences under comparison. Every dot in the grid shows the match in the sequence at that corresponding position. A prominent diagonal line represents a perfect match of a fragment of the sequence. If the slant is along the diagonal and is complete, it depicts the complete similarity of the sequences (as shown in the following screenshot). You can construct a dot plot using the following command from the `seqinr` package. We will try it in this recipe with a FASTA file (available at the book's page).

The file contains protein sequences for hemoglobin beta subunits from humans (*homo sapiens*) and chimpanzees (*pan troglodytes*). We can see how similar these sequences are by typing the following commands:

```
> library(seqinr)
> myseq <- read.fasta(file = "myfasta.fasta")
> dotPlot(myseq[[1]], myseq[[2]], col=c("white",
"red"), xlab="Human", ylab="Chimpanzee")
```

We can also try to compute the alignment and similarities between these sequences to confirm the findings as shown in the previous commands. We can play around with many different sequences provided in the FASTA file.

After doing all this, let's do a cool test for our alignments to know if the alignment we got was significantly better or not. We will do it via a permutation test. A permutation test is based on the random sampling of observed data (in our case, the template sequence) to determine how unusual an observed outcome is (in our case, the alignment score). The idea is to shuffle our template sequence via sampling from the set of characters in the sequence and align it to our pattern sequence. The alignment with every sampled sequence generates a score, and we check how often this score was better than the observed alignment of our original sequences. The sampling is done with a replacement, and the probabilities for the selection of each character during sampling are equal to its fraction in the sequence.

To conduct the test, simply load the R code file, `sequencePermutationTest.R` (available on the book's home page), using the `source` function as follows:

```
> source("sequencePermutationTest.R")
```

To run the function, use the following command—note that the time taken will depend on the number of permutations (`n`) and sequence lengths:

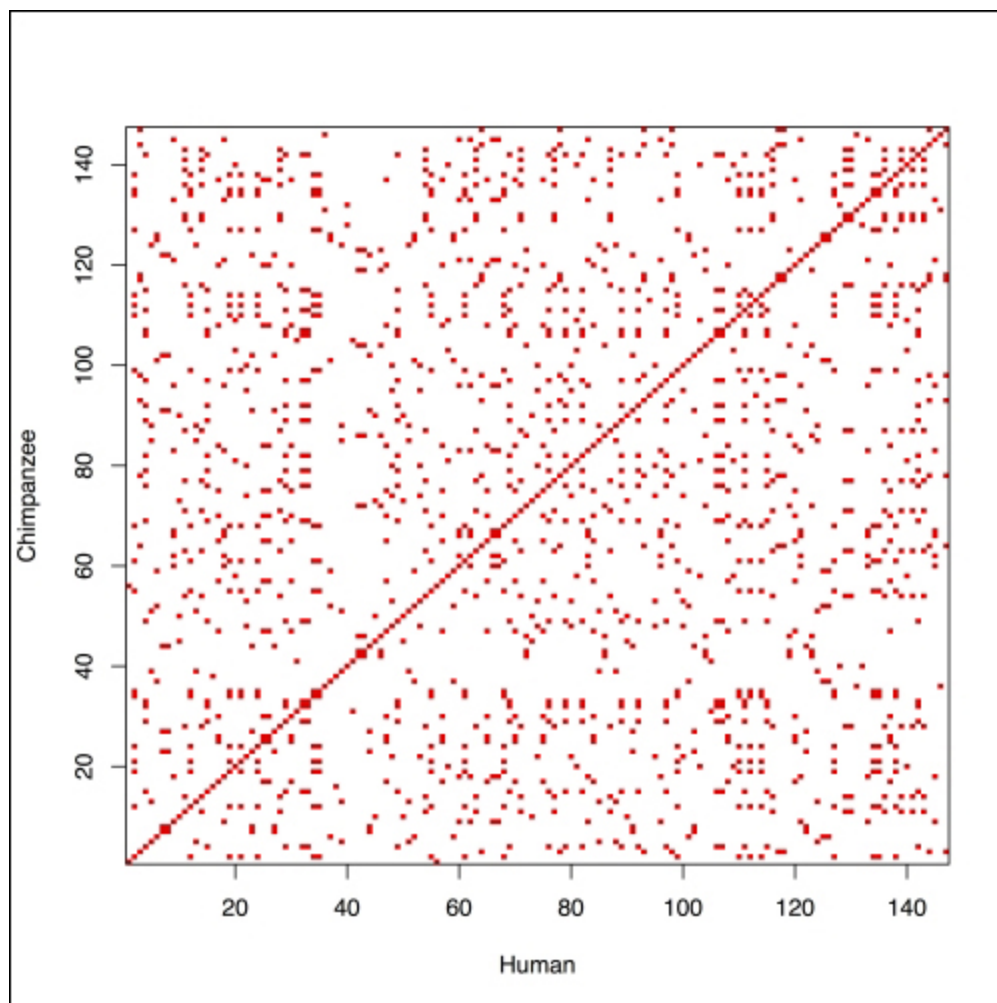
```
> pvalue <- permutes.seq(seq1=sequence1,
  seq2=sequence2, n=100)
```

What we get for our alignment is `~0.2`, which indicates that the alignment score we got is not significant at a p-value threshold of 0.05. Let's take a closer look at the function by opening it to

understand it better. We can also create a plot in the function by setting the `plot` argument to `TRUE` as follows:

```
> pvalue <- permutes.seq(seq1=sequence1,  
seq2=sequence2, n=1000, plot=TRUE)
```

The following screenshot shows the dot plot for hemoglobin beta protein sequences in humans and chimpanzees (note that the diagonal line shows a perfect match of the sequences):



See also

- The MIT open courseware page at <http://ocw.mit.edu/courses/electrical-engineering-and-computer->

science/6-096-algorithms-for-computational-biology-spring-2005/lecture-notes/lecture5_newest.pdf, which provides information on the alignment algorithm

- The article titled *A general method applicable to the search for similarities in the amino acid sequence of two proteins* by Needleman and Wunsch at <http://www.sciencedirect.com/science/article/pii/0022283670900574>, which provides detailed information on specific theoretical aspects of the Needleman-Wunsch algorithm for global alignment
- The *Identification of common molecular subsequences* article by Smith and others at <http://www.sciencedirect.com/science/article/pii/0022283681900875>, which provides information about the theoretical paper on the Smith-Waterman algorithm (local alignment)
- The *Amino acid substitution matrices from protein blocks* article by S Henikoff and J G Henikoff at <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC50453/>, which describes the BLOSUM matrices
- The article titled *A Model of Evolutionary Change in Proteins* by Dayhoff and others at <http://www.bio-recipes.com/Dayhoff/dayhoff1978.pdf>

Multiple sequence alignment

There are occasions in the sequence analysis where we need to compare many sequences against each other. A general example of such a case is phylogenetic analysis, which will be discussed in the next recipe. Such an analysis reveals how similar or dissimilar the sequences are against each other. In this recipe, we explain how to do **multiple sequence alignment (MSA)** using R.

Getting ready

The following are the prerequisites for this recipe:

- A new package named `muscle` to be installed and loaded in the R session.
- The sequences of our interest, that is, the sequences to be aligned. For sequences, we will use the protein sequences of Cytochrome oxidase (COX-2) from five different species (provided as the FASTA file on the book's home page).
- MUSCLE algorithms that are very well established and frequently used by bioinformaticians.

How to do it...

Follow the ensuing steps to perform the multiple sequence alignment:

1. Start with installing and loading the required packages by typing the following commands:

```
> install.packages("muscle")  
> library(muscle)
```

2. Then, get all the sequences to be aligned. They can be read directly from a FASTA file (available as a file on the book's home

page) as follows:

```
> mySeq <-  
muscle::read.fasta("fastaMSA.fasta")
```

3. Take a look at the sequences within the file that was read, as follows:

```
> myseq
```

4. Now, you need to fetch the sequence from the repositories and write them to a FASTA file before using them as explained in the previous recipes.
5. Now, use the `muscle` function to perform the alignment with the following command:

```
> MyMSA <- muscle(mySeq, out = NULL, quiet =  
FALSE)
```

6. Save the alignment as follows:

```
> muscle::write.fasta(MyMSA, file="aln.fasta")
```

7. To print the alignment on screen, use the following function:

```
> print(MyMSA, from = 1, to = aln$length)
```

How it works...

The `muscle` function depicted in this recipe uses the established **multiple sequence comparison by log-expectation (MUSCLE)** algorithm for MSA. The `read.fasta` function (from the `muscle` package) reads the sequences into a `fasta` object. The `fasta` object contains the sequences in the FASTA file as a data frame. The `muscle` function actually calls a C implementation of the MUSCLE algorithm. The algorithm works in three stages, namely, progressive alignment, improvement, and refinement. There are other web-based

programs, for instance, `ClustalW`, but MUSCLE algorithm has become a popular choice because of its speed and accuracy.

In our results, we see how closely related the proteins are, with minor differences, representing the closeness of the involved species at this level.

You can also read an alignment file, `myMSA.phy` (a dummy file name), into an R session using the `read.alignment` function from the `seqinr` package as follows. These files can be obtained from web-based or other standalone alignment tools for MSA, such as `ClustalW` (<http://www.genome.jp/tools/clustalw/> or <https://www.ebi.ac.uk/Tools/msa/clustalw2/>) and `Phylip` (<http://evolution.genetics.washington.edu/phylip.html>). Let's take a look at the following command:

```
> virusaln <- read.alignment(file = "myMSA.phy",  
format = "phylip")
```

This can be done for other alignment files using the `Biostrings` package's `readDNAMultipleAlignment` function for clustal files. The `MultipleAlignment` class of `Biostrings` also provides utilities to handle MSAs in R. For details, refer to the documentation of the `Biostrings` packages at <http://www.bioconductor.org/packages/2.13/bioc/manuals/Biostrings/man/Biostrings.pdf>.

Another package called `ape` also has functions for MSA. We will use this package in our phylogenetic analysis, which is explained in the next recipe.

See also

- The *Multiple sequence alignment: algorithms and applications* article by Gotoh at <http://www.ncbi.nlm.nih.gov/pubmed/10463075>, which provides

information about multiple sequence alignment algorithms and their application

- The *MUSCLE: multiple sequence alignment with high accuracy and high throughput* article by Edgar at <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC390337/?tool=pubmed>, which provides theoretical details of the MUSCLE algorithm

Phylogenetic analysis and tree plotting

Phylogenetic analysis is about finding the evolutionary relationship among species (organisms), in our case, based on sequence data. Once we have a set of sequences from different sources, it becomes very interesting to understand how close or distant they are in terms of molecular evolution. As a result of the mutations during evolution, there emerge differences at the sequence level. These differences can be represented in terms of distance measures (check the *See also* section at the end of this recipe). These measures can then be used to estimate the evolutionary relations among the species, often represented as phylogenetic trees. This relation is very often depicted in terms of a phylogenetic tree. So far, we looked into the various aspects of sequence retrieval, alignment, and analysis. This section aims to entail the recipe to perform a phylogenetic analysis on sequence data.

Getting ready

The prerequisites for the recipe are as follows:

- An R session with the `ape` package installed and loaded
- Sequences to be analyzed (at least their IDs)

How to do it...

To perform the phylogenetic analysis on sequences of your choice, follow the ensuing steps:

1. Install and load the `ape` package by typing the following command:

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

```
> library(ape)
```

2. Define the sequences of your interest as a set in terms of IDs as follows:

```
> myset <- c("U15717", "U15718", "U15719",  
             "U15720", "U15721", "U15722", "U15723",  
             "U15724")
```

3. Fetch the sequences of your interest, as provided in the following example:

```
> myseqs <- read.GenBank(myset)
```

4. Compute the distance matrix for the sequences with the following `dist.dna` function:

```
> mydist <- dist.dna(myseqs)  
> mydist
```

5. To get the `phylo` object for the phylogenetic trees, use the following `triangMtd` function that uses the `triangles` method to reconstruct the tree:

```
> myphylo <- triangMtd(mydist)
```

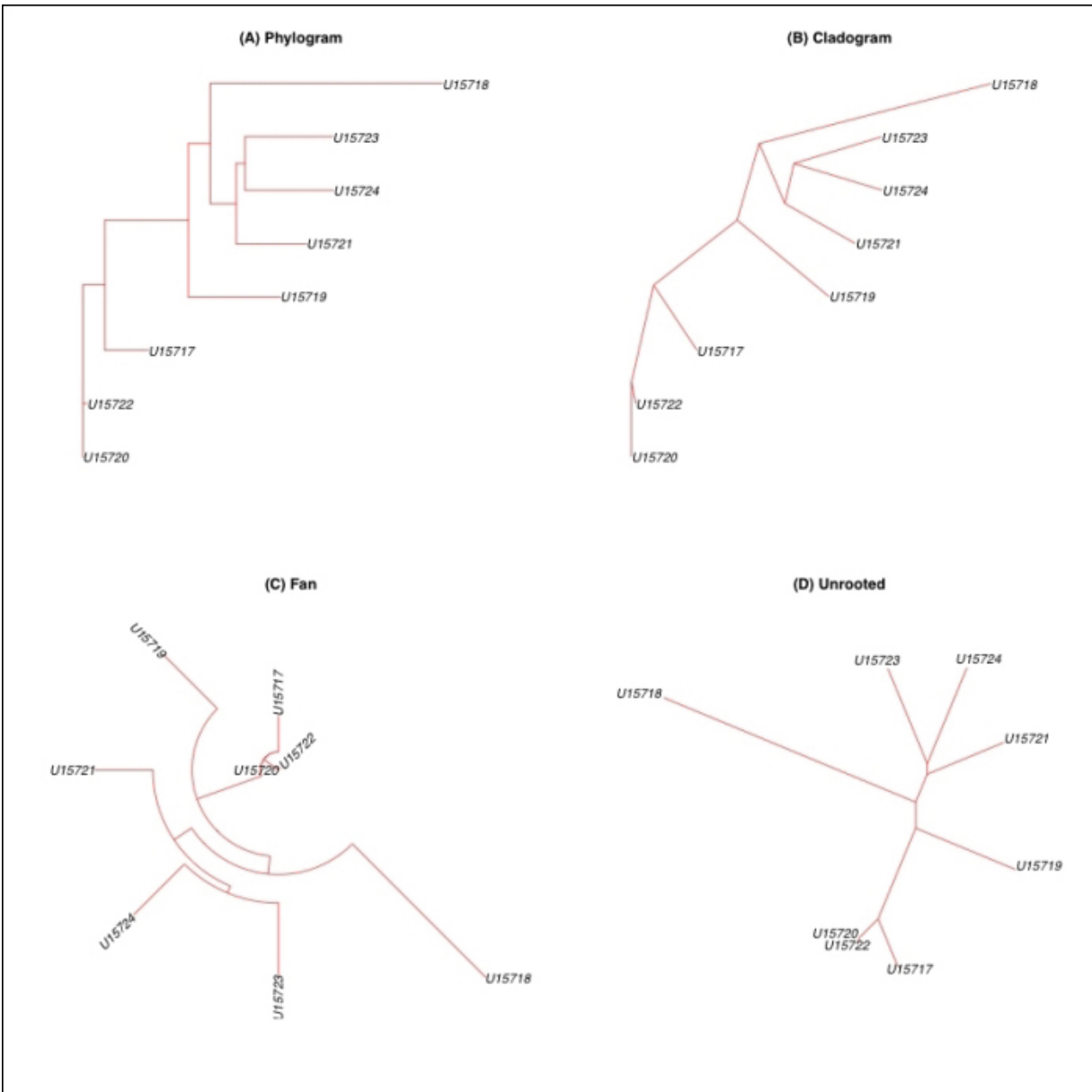
6. At this stage, we can do a bootstrap on the `phylo` object, if needed, using the `boot.phylo` function. For detailed information, type `?boot.phylo` in the R session.
7. Create the different kinds of phylogenetic trees for your analysis as follows:

```
> plot(myphylo, type="phylogram",  
       edge.color="red", cex=1, edge.width=1, main="  
       (A) Phylogram")  
> plot(myphylo, type="cladogram",  
       edge.color="red", cex=1, edge.width=1, main="  
       (B) Cladogram")  
> plot(myphylo, type="fan", edge.color="red",  
       cex=1, edge.width=1, main="(C) Fan")
```

```
> plot(myphylo, type="unrooted",  
edge.color="red", cex=1, edge.width=1, main="  
(D) Unrooted  
")
```

How it works...

Once the sequences are fetched from the database by the system, the process of phylogenetic analysis starts. The first step is the computation of pairwise distances between sequences. This is based on different models of DNA evolution, for example, the Kimura model, Jukes Cantor model, and so on. The distances are based on the rates at which one nucleotide replaces another during evolution. For details, try `?dist.dna`. The returned value is an object of the `dist` class (set `as.matrix = TRUE` to get the results in the form of a matrix). The `dist` object is then converted into a `phylo` object that can be used to plot phylogenetic trees. These distances actually depict how far the species are, or rather, the sequences on the evolutionary scale. A neighbor joining method creates the phylogenetic tree object. It joins the two closest neighbors so that the distances of these two clusters from the remaining clusters are as far as possible. Various types of plots for phylogenetic trees can be plotted. In this recipe, we used four different ways to visualize the phylogenetic tree. The phylogram shows the evolutionary history where the length of each branch stem indicates the amount of evolution (such as the number of nucleotide substitutions that occurred between the connected branch points). The second type of tree that we plotted is a cladogram. It does not represent the actual relation among the species but the branching during the evolution and branches join at hypothetical ancestors. The third type of tree is a fan-shaped tree that shows radical branches for species. The unrooted trees illustrate the relatedness of the leaf nodes, completely ignoring the ancestry. The following screenshot shows the different types of phylogenetic trees:



While performing a phylogenetic analysis, it is important to keep in mind the biological question that you are trying to answer. For instance, when studying closely related species at the protein level, the signals will be very small as proteins evolve at a much slower rate. It is therefore advisable to use nucleotide sequences for such an analysis. On the contrary, for distant species, the DNA signal is too noisy, and hence protein sequences are preferred. To illustrate this, the nucleotide sequences of a pair of homologous genes will have a higher information content compared to the corresponding

amino acid sequences as mutations, since nonsynonymous changes affecting the DNA might not affect the amino acid sequence.

Another important thing regarding the `ape` package is that it allows you to save the tree as a Newick-format tree file using the `write.tree` function. For further details, refer to `?write.tree`.

See also

- The *APE: Analyses of Phylogenetics and Evolution in R language* article by Paradis and others at <http://bioinformatics.oxfordjournals.org/content/20/2/289.abstract>, which provides more theoretical information on the `ape` package
- The *Models of Molecular Evolution and Phylogeny* article by Liò and Goldman at <http://genome.cshlp.org/content/8/12/1233.full>, which provides an extensive overview of the models of the DNA sequence evolution
- The NCBI book, *Genomes*, (Chapter 16, *Molecular Phylogenetics*) at <http://www.ncbi.nlm.nih.gov/books/NBK21122/>, which provides a great background in phylogenetic analysis

Handling BLAST results

Basic Local Alignment Search Tool (BLAST) is a basic tool for bioinformaticians. The tool aims at looking for local similarities across sequences against a database of relevant sequences. It takes an input sequence (nucleotide or protein) and a reference database (genomic data or EST) and performs the local alignment of the former against the latter. This is very useful in understanding the function and comparison of new sequences as well as to understand the relation among them. You can BLAST any of the sequences that we retrieved in the preceding sections on the NCBI BLAST web page (<http://blast.ncbi.nlm.nih.gov/Blast.cgi>).

BLAST results are usually an HTML page but can be downloaded as a table (the red rectangle in the previous screenshot). This recipe talks about the handling of these BLAST result tables. The following screenshot shows the BLAST result web page:

BLAST® Basic Local Alignment Search Tool

Home Recent Results Saved Strategies Help

My NCBI [Sign In] [Register]

NCBI/ BLAST/ blastp suite/ Formatting Results - N3DN06DB01R

Edit and Resubmit Save Search Strategies Formatting options Download YouTube How to read this page Blast report description

sp|P02062|HBB_HORSE

RID N3DN06DB01R (Expires on 04-20 07:34 am)

Query ID lc|160988 Database Name nr
 Description sp|P02062|HBB_HORSE Description All non-redundant GenBank CDS
 Molecule type amino acid translations+PDB+SwissProt+PIR+PRF excluding
 Query Length 146 environmental samples from WGS projects
 Program BLASTP 2.2.29+ Citation

Other reports: Search Summary Taxonomy reports Distance tree of results Multiple alignment

+ Graphic Summary
 - Descriptions

Sequences producing significant alignments:

Select: All None Selected:0

Alignments Download GenPept Graphics Distance tree of results Multiple alignment

	Description	Max score	Total score	Query cover	E value	Ident	Accession
☐	hemoglobin subunit beta [Equus caballus]	296	296	100%	1e-100	100%	NP_001157490.1
☐	RecName: Full=Hemoglobin subunit beta; AltName: Full=Beta-globin; AltName: Full=Hemo	296	296	100%	1e-100	100%	P02062.1
☐	RecName: Full=Hemoglobin subunit beta; AltName: Full=Beta-globin; AltName: Full=Hemo	293	293	100%	4e-99	99%	P67823.1
☐	Chain B, Three Dimensional Fourier Synthesis Of Horse Deoxyhaemoglobin At 2.8 Angstror	292	292	100%	5e-99	98%	2DHB_B
☐	PREDICTED: hemoglobin subunit beta-like [Pteropus alecto]	260	260	100%	1e-85	86%	XP_006915761.1

Getting ready

The basic requirement of this recipe is a BLAST result. For this, we need our computer to be connected to the Internet to download the results and libraries.

Input the sequence we retrieved in the previous recipes. The input can either be made in the textbox by pasting the sequence or path to a FASTA file or a sequence ID. Then, we can run the BLAST program of our interest to get the results. The result table can then be downloaded as a table.

If the results are downloaded from the NCBI standalone BLAST, the [RFLPtools](#) package meant for the DNA fragment analysis and BLAST report analysis can easily be used.

How to do it...

To work with the BLAST results, perform the following steps:

1. Install and load the following functions of `RFLPtools` into your R session that reads the BLAST results:

```
>
install.packages("RFLPtools",dependencies=TRUE
)
> library(RFLPtools)
```

2. Use the built-in blast result from the package for a demo here.
To specify the file, type the following commands:

```
> DIR <- system.file("extdata", package =
"RFLPtools")
> MyFile <- file.path(DIR, "BLASTexample.txt")
```

3. To read the file, simply use the following command:

```
> MyBLAST <- read.blast(file = MyFile)
```

4. If the BLAST results are from a standalone BLAST, use the RFLP library's `read.blast` function, as follows, to read the `MyblastRes.txt` BLAST file into a data frame:

```
> MyBLAST <- read.blast(file="MyblastRes.txt")
```

5. In case you did not use the BLAST program but used the web-based tools to get the `myAlign.txt` file, you can still read the file as a data frame with the following set of commands:

```
> MyBLAST2 <- read.csv(file="myAlign.txt",
head=TRUE, sep=",")
```

6. Take a look at the read in data contents:

```
> head(MyBlast2)
```

7. The `RFLPtool` package also allows the computing/extracting of similar matrices out of the data frame as follows:

```
> mySimMat <- simMatrix(MyBLAST)
```

Then, you can name the columns of the data per your convenience and use it for further analysis.

How it works...

The web-based BLAST algorithm primarily outputs an HTML file together with some other formats. However, the alignment is available for download in a tabular format. The recipe just explains a way to import these files as a data frame. The `read.blast` functions do the same with the file generated by the standalone BLAST. For instance, such data is available as an example in the `RFLPtools` package. It can be loaded as follows:

```
> data(BLASTdata)
> head(BLASTdata)
```

This gives us an idea about the contents of the BLAST result data. The data consists of various elements, such as the name, identity, sequence length, alignment length, E-values, and some other attributes of the BLAST search. Try to look at the column names of the package data as follows:

```
> colnames(BLASTdata)
[1] "query.id"      "subject.id"
[3] "identity"      "alignment.length"
[5] "mismatches"    "gap.opens"
[7] "q.start"       "q.end"
[9] "s.start"       "s.end"
[11] "evalue"        "bit.score"
```

See also

- The Nature Scitable page at <http://www.nature.com/scitable/topicpage/basic-local-alignment-search-tool-blast-29096>, which provides information on the BLAST program
- The *Basic local alignment search tool* article by Altschul and others at <http://www.ncbi.nlm.nih.gov/pubmed/2231712>, which provides more theoretical detail on the BLAST program
- The detailed documentation on the `RFLPtools` package, which is available on the package's CRAN page at <http://cran.r-project.org/web/packages/RFLPtools/index.html>

Pattern finding in a sequence

So far, we have learned a lot about fetching a sequence, aligning, and matching it with others. It is equally important to check for some patterns in the gene apart from the general nucleotide content analysis we had in the previous recipe. It can be used to detect various interesting subsets of characters in nucleotide or protein sequences, for example, the start and stop codons in nucleic acids. Start and stop codons mark the site at which the translation into protein sequences begins and the site at which the translation ends. The common example of a start codon in mRNAs is AUG and for a stop codon, some examples are UAG, UGA, and UAA. Even in the case of protein sequences, we can search conserved motifs. This recipe will talk about finding patterns in the sequence that can be interesting from the point of view of functional and molecular biology, such as gene finding in a sequence or ORF finding in a sequence. ORFs are the frames in a genomic sequence that are not interrupted by stop codons.

Getting ready

To start with, we just need a nucleotide sequence to work with, either in the R session or as a file, in the `Bioststrings` library.

How to do it...

To find a pattern in a sequence, perform the following steps:

1. Looking for the start and stop codons in R can be seen as a simple pattern-matching problem and can be solved with the `Bioststrings` library. Load this library with the following function:

```
> library(Bioststrings)
```

2. Then, create the sequence to be analyzed as follows (in our case, we use a random sequence, but we can use the sequences fetched from GenBank as well):

```
> mynucleotide <-  
DNASTring("aacataatgcagtagaaccatgagccc")
```

3. Look for a pattern of your interest, such as a start codon `ATG`, as shown in the following example:

```
> matchPattern(DNASTring("ATG"), mynucleotide)
```

4. Similarly, look for the pattern for the stop codons, such as `TAA` or other stop codons, as shown in the following example:

```
> matchPattern("TAA", mynucleotide)
```

5. Now, combine these two aspects into a single function to return the overall results for all the codons as in the following function:

```
myCodonFinder <- function(sequence){  
  startCodon = DNASTring("ATG") # Assign  
  start codons  
  stopCodons = list("TAA", "TAG", "TGA") #  
  Assign stop codons  
  codonPosition = list() #initialize the  
  output to be returned as a list  
  codonPosition$Start =  
  matchPattern(startCodon, sequence) # search  
  start codons  
  x=list()  
  for(i in 1:3){ # iterate over all stop codons  
    x[[i]]= matchPattern(DNASTring  
      (stopCodons[[i]]), sequence)  
    codonPosition$Stop=x  
  }  
  return(codonPosition) # returns results  
}
```

6. Now, paste the code for the previous function into the R session and run it with your sequence object `mynucleotide` as follows:

```
> StartStops <- myCodonFinder(mynucleotide)
```

Alternatively, save the source code as a file, say, `myCodonFinder.R`, and source it into the R session as follows:

```
> source("myCodonFinder.R")
```

7. To find the genes in the nucleotide sequence, tweak the function by looking for a start codon and then a potential stop codon with a set of codons in between.

The modification can also be used to find **open reading frames (ORFs)**.

How it works...

The `matchPattern` function of `Biostrings` is an implementation to identify the occurrences of a particular pattern or motif in a sequence. It requires a string as an input (not a vector of characters) that is created by the `DNASTring` function. If you intend to use a vector of characters from the *Retrieving a sequence* recipe, you must convert it into a string via the use of the `c2s` and `DNASTring` functions (`AAString` for a protein sequence and `RNAString` for RNAs). The `matchPattern` function returns a table with columns that represent the start, end, and width of the hit or match (the width is obviously the same as the length of the pattern). All the hits found in the sequence are arranged in the rows of the returned object.

In our function, to find all the start and stop codons, we combine two pattern searches; first, we combine the search for the start codon, and second, we look iteratively for all the three stop codons. We can modify this function to find an ORF. This is left for the reader to try.

The `Biostrings` package has many other interesting functions to solve more complicated issues regarding patterns in sequences. Another similar example is the `matchLRPatterns` function that finds the paired matches in a sequence. It can be used as follows:


```
> mytarget <- DNASTring("AAATTAACCCTT")  
> matchLRPatterns("AA", "TT", 5, mytarget)
```

It is recommended that you take a deeper look at these two packages (`Biostrings` and `seqinr`) to learn more about such methods.

See

- The BioWeb page at http://bioweb.uwlax.edu/genweb/molecular/seq_anal/translation/translation.html, which provides more information about ORFsmolecular biology
- The NCBI page at http://www.ncbi.nlm.nih.gov/Class/MLACourse/Modules/MolBioReview/codons_start_stop.html, which provides a description of the codons in molecular biology
- The Biostrings package article written by Pages and others at <http://www.bioconductor.org/packages/2.13/bioc/html/Biostrings.html>, which provides detailed documentation on the Biostrings package

Chapter 4. Protein Structure Analysis with R

In this chapter, we will cover the following recipes:

- Retrieving a sequence from UniProt
- Protein sequence analysis
- Computing the features of a protein sequence
- Handling the PDB file
- Working with the InterPro domain annotation
- Understanding the Ramchandran plot
- Searching for similar proteins
- Working with the secondary structure features of proteins
- Visualizing the protein structures

Introduction

Proteins are very versatile biomolecules. The structural and functional roles of proteins have been investigated through experimental studies as well as numerous computational techniques. We learned about sequence analysis in [Chapter 3](#), *Sequence Analysis with R*, where we focused mainly on matching and alignments. However, proteins, in contrast to nucleotides, show a great variety of three-dimensional conformations necessary for their diverse structural and functional roles. The amino acids in a protein dictate important terms in its three-dimensional structure. The frequency of particular amino acid residues in the sequence plays an important role in determining the secondary structures of the protein, namely, α helix, β sheet, and turns. For the details of this dependence, visit <http://www.ncbi.nlm.nih.gov/books/NBK22342/table/A354/?report=objectonly>.

The steric conformation in certain amino acids such as valine, threonine, and isoleucine makes them more adaptable in β sheets. However, predicting the secondary structure of peptides is slightly more complicated than these frequencies; nevertheless, amino acid sequences can be extremely useful for the purpose of estimating the overall protein structure.

Besides structure, the physicochemical properties of a protein are also determined by the analogous properties of the amino acids (ultimately the sequence) in it. For example, the acidity, basicity, and so on, can be decided based on the fraction of acidic or basic amino acids in the protein. All the properties of proteins, starting from structure to physicochemical properties, are critical for their biological role.

Although there are many promising software tools for protein structure analysis and visualization, R offers fewer options that serve the purpose. Nevertheless, it is good to have a glance at these possibilities with R. This chapter will illustrate how to work with proteins at the sequence and structure levels. Here, we cover the important and fundamental aspects and methods of protein bioinformatics, including sequence, structure analysis, and so on.

Retrieving a sequence from UniProt

To begin with the protein analyses, we first need to retrieve a protein sequence. UniProt is a high-quality and freely accessible database of protein sequences and functional information. It should be noted that certain other databases such as [RefSeq](#) also provide protein sequence information, but UniProt is a manually curated one and hence better for analytical purposes in this recipe. We use this as the source for our sequence information. This recipe will explain a few ways to retrieve sequences from UniProt using R.

Getting ready

Retrieving sequences from the UniProt database simply requires an identification of the protein sequence that we want to retrieve. We will look for the **NAD kinase (NADK)** enzyme in humans (UniProt ID: O95544).

How to do it...

To retrieve the protein sequences from UniProt, perform the following steps:

1. The first method we present uses the `biomaRt` package (as explained in [Chapter 3](#), *Sequence Analysis with R*). To use it, first load the `biomaRt` package as follows:

```
> library(biomaRt)
```

2. Followed by this, select the UniProt mart called `unimart` in `biomaRt` as follows:

```
> myMart <- useMart("unimart",
```

```
dataset="uniprot")
> myMart
```

- Now, check the attributes in the selected mart by typing the following command so as to decide what entities can be retrieved:

```
> listAttributes(myMart)
```

- Retrieve the interesting attributes in the following way using the `getBM` function:

```
> myProt <-
getBM(attributes=c("pdb_id","protein_name",
"ensembl_id","go_id", "name"),
filter="accession", values="O95544",
mart=myMart)
> myProt
```

	pdb_id	protein_name	ensembl_id
go_id		name	
1	3PFN	NAD kinase	ENSG00000008130
GO:0016310		NADK_HUMAN	
2	3PFN	NAD kinase	ENSG00000008130
GO:0019674		NADK_HUMAN	
3	3PFN	NAD kinase	ENSG00000008130
GO:0044281		NADK_HUMAN	
4	3PFN	NAD kinase	ENSG00000008130
GO:0046034		NADK_HUMAN	
5	3PFN	NAD kinase	ENSG00000008130
GO:0006767		NADK_HUMAN	
6	3PFN	NAD kinase	ENSG00000008130
GO:0003951		NADK_HUMAN	
7	3PFN	NAD kinase	ENSG00000008130
GO:0005829		NADK_HUMAN	
8	3PFN	NAD kinase	ENSG00000008130
GO:0005524		NADK_HUMAN	
9	3PFN	NAD kinase	ENSG00000008130
GO:0006766		NADK_HUMAN	
10	3PFN	NAD kinase	ENSG00000008130
GO:0006741		NADK_HUMAN	
11	3PFN	NAD kinase	ENSG00000008130
GO:0046872		NADK_HUMAN	

- To retrieve the sequence, set up an Ensemble mart with the following function:

```
> ensemblMart <- useMart(biomart = "ensembl",
  dataset = "hsapiens_gene_ensembl")
```

6. Now, use the `getSequence` function of R to fetch the sequence of interest as follows:

```
> myProtein <-
  getSequence(id=myProt$ensembl_id[1],
  type="ensembl_gene_id", seqType = "peptide",
  mart=ensemblMart)
```

7. Check the first retrieved sequence, which is a component of the `myProtein` object, as follows:

```
> myProtein$peptide[1]
```

8. Another method to get the sequence is the use of the `UniProt.ws` package. To do this, first install and load the `UniProt.ws` package as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("UniProt.ws")
> library(UniProt.ws)
```

9. Once you have the library loaded in the session, check the available species in UniProt (in this case, with the pattern, `sapiens`) as follows:

```
> availableUniprotSpecies(pattern="sapiens")
  taxon ID      Species name
1    63221 Homo sapiens neanderthalensis
2     9606      Homo sapiens
```

10. Select the taxonomic ID for the human sequence as shown in the following command:

```
> taxId(UniProt.ws) <- 9606
```

11. Check the information available in the database package with the following R commands:

```
> head(keytypes(UniProt.ws))
> head(cols(UniProt.ws))
```

12. Now, select the information you want from the database using Entrez IDs as the keys and columns of your choice as follows:

```
> keys <- "O95544"
> cols <- c("PDB", "UNIPROTKB", "SEQUENCE")
> kt <- "UNIPROTKB"
```

13. To fetch the sequence, use the `select` function of the `UniProt.ws` package as follows:

```
> myProtein2 <- select(UniProt.ws, keys, cols,
kt)
```

14. The following `myProtein2` variable will have the desired sequences:

```
> myProtein2$SEQUENCE
```

How it works...

The working principle of `biomaRt` has already been explained in the previous chapter. The UniProt mart, `unimart`, is a BioMart database used for querying **UniProt Knowledgebase (UniProtKB)** data, with a cross-querying facility to join to/from other BioMart databases. The `getBM` function uses the following three arguments:

- **Attribute:** This is the field to be fetched for the query
- **Filters:** These restrict the query to match certain criteria (in our case, the UniProt ID)
- **Mart:** This is created by the `usemart` function to identify the target database

The value returned is a data frame, where each row represents an entry for the corresponding protein. To get the sequence, we execute the `getSequence` function that accepts the Ensemble IDs as input. However, before this, we need to create another mart object for the Ensemble databases, as we did for UniProt, to use it to fetch the sequence. The `getSequence` function accepts the ID that we want to retrieve from the `myProt` object, the ID type (in our case, Ensemble), the type of sequence we want to retrieve (in our case, we look for peptide sequences), and finally, the Ensemble mart. The result is another data frame. Type in `head(myProtein)` to know about the contents of the data frame.

The alternative presented here is the `UniProt.ws` package. This package is an R interface for UniProt web services. UniProt (<http://www.uniprot.org>) is a resource for protein sequences and annotation data that can be extracted from the host page of the database. The various kinds of data that can be extracted from UniProt includes peptide sequences, the GO annotation, interactions, protein families, and so on. Loading the package in an R session creates a `UniProt.ws` object (check with the `ls()` function). The `UniProt.ws` package can also access this data from within R and is preferred to fetch the data from remote sources and the analysis of data that avoids dealing manually with browser-based data fetching as well as the automation of the analysis pipeline.

The database contains protein data for a number of species. Therefore, we first get the list of species matching a pattern with the function and then select a taxonomic ID. The most important function in the package is the `select` function that extracts data from UniProt. The function needs three inputs, columns (`cols`), keys (`keys`), and key-types (`kt`), together with the `UniProt.ws` object. The columns input shows which kinds of data can be returned for the `UniProt.ws` object, and the keys input consists of the keys for the database contained in the object. The `keytype` argument can be used to return specific keys. For example, we used UniProt's KB keys here; we could have used the `ENTREZ_GENE` key type if the input type was

Entrez IDs. The `select` function then returns the corresponding columns from the database.

See also

- The documentation page for the database at <http://www.uniprot.org/help/about>, which provides more information about the UniProt database.
- The *UniProt: the Universal Protein knowledgebase* article by Apweiler and others at http://nar.oxfordjournals.org/content/32/suppl_1/D115.long, which provides a detailed description about the database and its schema.
- The other ways to retrieve protein sequences, such as **Protein Data Bank (PDB)**. PDB (www.rcsb.org) that focuses mainly on information about experimentally determined structures of proteins, nucleic acids, and complex assemblies, which will be discussed in the upcoming recipe, *Protein sequence analysis*.
- Another possible source of protein data, SWISSPROT. It is the manually annotated and reviewed section of the UniProt knowledge base that can be accessed using the `biomaRt` or `seqinR` packages. Use the help pages of the corresponding packages for further information.

Protein sequence analysis

We have already seen the recipe for a general sequence analysis (both for nucleic acids and proteins) in the previous chapter. However, while doing a protein sequence alignment, we can directly fetch the sequence data from PDB (www.rcsb.org) and do the alignment within the program in R. Furthermore, the position of a protein multiple sequence alignment plays an important role, and color is often used to indicate amino acid properties to aid in judging the conservation of a given amino acid substitution. This will ultimately affect the structural, functional, and physicochemical properties of proteins. Here, we will present a recipe to create such an alignment and coloring of alignment with a protein sequence alignment and generate a readily reusable HTML format.

Getting ready

As we focus mainly on a multiple sequence alignment, this recipe, besides an Internet connection, needs MUSCLE software. MUSCLE is an open source multiple alignment software. We will also use the [bio3d](#) R package in this recipe, which has utilities for protein structure and sequence-related analysis.

How to do it...

To analyze the protein sequences, perform the following steps:

1. To install MUSCLE on a Linux (Ubuntu) system, use the following command from your shell terminal. For some Linux distributions, `sudo` is sometimes not available. Refer to the installation documentation at <http://www.drive5.com/muscle/manual/install.html> for details:

```
$ sudo apt-get install muscle
```

2. Then, install and load the `bio3d` package into the R session. The package is available as the source at <http://thegrantlab.org/bio3d/download>. Binary for Windows are also available at http://thegrantlab.org/bio3d/phocadownload/Bio3D_version2.x/bio3d_2.0-1.tar.gz. To download it from terminal in Linux, use the `wget` command as follows:

```
> wget
```

3. To install the package, use the following `install.packages` function (remember to move to the directory where the source file is located or specify the entire path):

```
> install.packages("bio3d_2.0-1.tar.gz",  
  repos=NULL, type="source")
```

Furthermore, there are certain dependencies for the `bio3d` package, and these must be installed before you install `bio3d` itself. The details of the dependencies are available at <http://thegrantlab.org/bio3d/download>.

4. Before you begin working with the package, load it into your R session as follows:

```
> library(bio3d)
```

5. Now, fetch the set of proteins from PDB (another source to get sequences) using the `read.pdb` function that reads the PDB files from PDB (the others used depend on UniProt or Ensemble) as follows:

```
> pdb1 <- read.pdb("1BG2")  
> pdb2 <- read.pdb("2VVG")  
> pdb3 <- read.pdb("1MKJ")
```

6. From these proteins, extract the sequences that will be aligned with the following function using `aa321`; it reads the three-letter

representation for amino acids in the PDB files and converts them to one-letter codes (hence 3-2-1):

```
> s1 <- aa321(pdb1$seqres)
> s2 <- aa321(pdb2$seqres)
> s3 <- aa321(pdb3$seqres)
```

7. To align these sequences, first create a matrix of sequences as follows:

```
> raw <- seqbind(seqbind(s1, s2), s3)
```

8. Now, align the set of sequences with the `seqaln` function as follows:

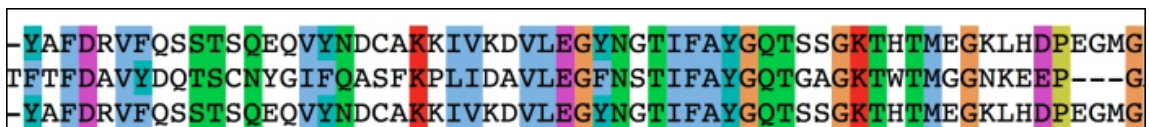
```
> aln <- seqaln(raw, id=c("1BG2", "2VVG ", "1MKJ
"))
```

9. Save the alignment and observe it as a colored HTML file with the following function:

```
> aln2html(aln, append=FALSE,
file="Myalign.html")
```

10. Now, open the created HTML file in a browser to visualize the results.

The following screenshot shows a part of the alignment (not the entire alignment—scroll through your HTML file to see the entire alignment) for three protein sequences produced as HTML files with color codes (in this case, it is shown in grayscale due to printing limitations):



```
-YAFDRVFSSTSQEQVYNDCAKKIVKDVLEGYNGTIFAYGQTSSGKTHTMEGKLHDPEGMG
TTFDAVYDQTSNCGIFQASFPLIDAVLEGFNSTIFAYGQTGAGKTWTMGGNKEEP---G
-YAFDRVFSSTSQEQVYNDCAKKIVKDVLEGYNGTIFAYGQTSSGKTHTMEGKLHDPEGMG
```

How it works...

We have already seen some of the alignment methods in [Chapter 3, Sequence Analysis with R](#). In this recipe, we introduced a new package called `bio3d` that specializes in dealing with protein sequences and deals with the PDB file. A PDB file is a representation of the macromolecular structure data derived from X-ray diffraction and NMR studies. It contains the sequences and coordinates of amino acid residues. The `read.pdb` function goes to the PDB database and reads the corresponding PDB file. In our case, we first fetch the protein sequence for the PDB database. The amino acid residues are in three-letter representations, and the `aa321` function converts them into one-letter representations. Another similar function, `aa123` (1-2-3), can do the conversions the other way round. This is followed by the combining of the sequences into a matrix of sequences with the `seqbind` function (in our case, we did it in a nested way). The `alignment` function then uses the MUSCLE program to align the sequences.

Here, our input sequences are kinesin proteins from three different organisms. They are microtubule-associated, force-producing proteins that may play a role in organelle transport. The alignment shows the differences as well as the conserved domains (colors in the alignments) in our HTML file. For more details, refer to the InterPro page at <http://www.ebi.ac.uk/interpro/entry/IPR019821>.

There's more...

The function can also be used for sequences retrieved from other sources of sequence data, such as `seqinR` or `UniProt`. However, `bio3d` can deal with the PDB files directly. Note that as these methods use an external program, MUSCLE, one must install this software before performing an alignment. For more information, visit <http://www.drive5.com/muscle/>.

See also

- The *Bio3d: an R package for the comparative analysis of protein structures* article by Grant and others at

<http://bioinformatics.oxfordjournals.org/content/22/21/2695>,
which provides information about the `bio3d` package

- The documentation page for PDB at <http://www.wwpdb.org/docs.html>, which provides a description about PDB files

Computing the features of a protein sequence

A sequence can provide certain information about the protein. Its properties include the amino acid composition, hydrophobicity, and so on. Once we have retrieved a sequence, we need to know more about it, such as the nucleotide frequency and GC content (see [Chapter 3](#), *Sequence Analysis with R*) present in it. In this recipe, we discuss how to get such information out of sequence data.

Getting ready

To get the features from the sequence data, use the `protr` package available with CRAN. It can be installed like any other CRAN package, as explained earlier.

How to do it...

To compute features in protein sequences, perform the following steps:

1. First, install and load the `protr` package as follows:

```
> install.packages("protr")
> library(bio3d)
> library(protr)
```

2. Now, the sequence needs to be analyzed. To do this, retrieve the sequence from PDB by typing the following commands:

```
> pdb1 <- read.pdb("1BG2")
> s1 <- aa321(pdb1$seqres)
```

3. The `protR` package needs the sequence as a sequence string. Therefore, first collapse a vector of characters into a string as follows:

```
> s1 <- paste(s1, sep="", collapse="")
```

Alternatively, type the following commands:

```
> library(seqinr)
> s1 <- c2s(s1)
```

4. Now, extract and compute the features from this sequence. Start with the amino acid composition using the following `extractAAC` function:

```
> extractAAC(s1)
```

5. Check whether the amino acid types of the protein sequence are in the 20 default types using the following function:

```
> protcheck(s1)
```

6. Compute the next feature, that is, the amphiphilic pseudo amino acid composition, which gives an idea related to the sequence order of a protein and the distribution of the hydrophobic and hydrophilic amino acids along its chain, with the help of following command:

```
> extractAPAAC(s1, props = c("Hydrophobicity",
  "Hydrophilicity"), lambda = 30, w = 0.05,
  customprops = NULL)
```

7. Compute some other features such as the composition descriptor, transition descriptor, and dipeptide composition with the help of the following commands:

```
> extractCTDC(s1)
```



```
> extractCTDD(s1)
> extractDC(s1)
```

How it works...

The `protr` package aims at protein sequence feature extraction, which could be used for various purposes. It implements most of the utilities to compute protein sequence descriptors. **Amino Acid Composition (AAC)** is the fraction of each amino acid type within a protein. The `extractAAC` function computes the amino acid composition as the fraction of each amino acid within a protein. Similarly, most of the features we discussed in this recipe are computed based on the sequence content. The descriptors look for individual amino acids, and the overall descriptor is computed based on these.

See also

- The Protr Vignette page at <http://cran.r-project.org/web/packages/protr/vignettes/protr.pdf>, which provides detailed information about the descriptors that can be computed using the `protr` package

Handling the PDB file

PDB is a worldwide archive of the structural data of biological macromolecules and has a considerable amount of protein data that includes structure as well as sequence. The data in PDB is available in the PDB file format. The PDB format is the standard for files containing atomic coordinates together with sequences and other information. Handling a PDB file is the first step to analyzing a protein structure. This recipe introduces the function to read a PDB file in R.

Getting ready

The prerequisites for this recipe are as follows:

- A few R packages, such as `bio3d`
- A machine that is connected to the Internet to access the PDB database together with the PDB ID that we need to read

How to do it...

Perform the following steps to work with the PDB file:

1. First, load the `protr` and `bio3d` libraries as follows:

```
> library(protr)
> library(bio3d)
```

2. Now, read a PDB file for the human kinesin motor domain with the ID as `1BG2` from PDB as follows:

```
> pdb <- read.pdb("1BG2")
```

3. Check the different parts and components of the created PDB object as follows:

```
> class(pdb)
> attributes(pdb)
> head(pdb)
> head(pdb$atom[, c("x", "y", "z")])
```

4. To get the C-alpha coordinates in the protein molecule, simply access the corresponding record of the PDB object by typing the following command:

```
> head(pdb$atom[pdb$calpha, c("resid",
"eley", "x", "y", "z")])
      resid eley x          y          z
[1,] "ASP" "CA"  "45.053" "-2.661" "39.856"
[2,] "LEU" "CA"  "44.791" "-1.079" "43.319"
[3,] "ALA" "CA"  "42.451" "-3.691" "44.790"
[4,] "GLU" "CA"  "40.334" "-4.572" "41.737"
[5,] "CYS" "CA"  "36.711" "-3.532" "42.171"
[6,] "ASN" "CA"  "36.940" "-0.636" "44.620"
```

5. To access the sequence from the PDB object, access the sequence record as follows:

```
> aa321(pdb$seqres)
```

6. Use the `write.pdb` function to write a PDB file to a local object. The `read.pdb` function can be used to read this file into the R session again as follows:

```
> write.pdb(pdb, file="myPDBfile.pdb")
> read.pdb("myPDBfile.pdb")
```

How it works...

The `read.pdb` function gets the PDB data for the input ID into the R session as a PDB object. The created object has all the records regarding the residue coordinates, sequences, and torsion angles. To extract the sequences or other elements from the protein, we simply use the names of the nested objects within the PDB object.

Working with the InterPro domain annotation

InterPro is a resource that provides the functional analysis of protein sequences by classifying them into families and predicting the presence of domains and important sites. To classify proteins in this way, InterPro uses predictive models called signatures from several databases that make up the InterPro consortium. It includes Gene3D, PANTHER, PRINTS, ProDom, PROSITE, TIGRFAMs, and so on. InterPro is a database of protein families, domains, and functional sites, in which identifiable features found in known proteins can be applied to new protein sequences in order to functionally characterize them.

Getting ready

This recipe needs some new libraries to be installed, such as `muscle`, in our R session. Besides this, we also need the sequences of our interest, that is, the sequences to be aligned. We show two different MSA algorithms in this recipe, which are very well established and used by bioinformaticians.

How to do it...

Perform the following steps for the InterPro annotation of proteins:

1. First, load the `biomaRt` library into the R session as follows:

```
> library(biomaRt)
```

2. Then, select the mart you intend to use, as earlier with the Ensemble mart for Homo sapiens, by typing the following command:

```
> myMart <- useMart("ensembl", dataset =  
"hsapiens_gene_ensembl")
```

3. Now, get the input ID for the annotation by typing the following command (for demo purposes, use the Ensemble ID from the *Retrieving a sequence from UniProt* recipe of this chapter):

```
> ensemblID <- "ENSG00000008130"
```

4. To fetch the InterPro annotation, use the `getBM` function of `biomaRt`. Now, we select the `UniProt` mart and database that will be used to fetch the domain data as follows:

```
> unip <-  
useDataset("uniprot", mart=useMart("unimart"))  
> interpro <-  
getBM(attributes=c("interpro_id", "go_id"), filters="ensembl_id",  
values="ENSG00000008130", mart=unip)
```

The retrieved object is a data frame, with each row that represents an annotation for the ID that is being searched for. The output is as follows:

```
> head(interpro)  
  interpro_id go_id  
1   IPR016064 GO:0016310  
2   IPR002504 GO:0016310  
3   IPR017438 GO:0016310  
4   IPR017437 GO:0016310  
5   IPR016064 GO:0019674  
6   IPR002504 GO:0019674
```

How it works...

The entire recipe works as any other `biomaRt` annotation retrieval we have seen before. The function gets the set of attribute columns defined in the `getBM` function from the mart and is returned as a data

frame. Since each query ID can have more than one domain, there can be more than one row for each domain in the returned results.

There's more...

Another possible way to learn about the protein family and domains is with the use of PFAM domains. The PFAM database contains information about protein domains and families. You can use the [PFAM.db](#) package, which is available at <http://bioconductor.org/packages/devel/data/annotation/html/PFAM.db.html>. For more information, refer to the help page for the package.

See also

- The *InterPro in 2011: new developments in the family and domain prediction database* article by Hunter and others at <http://nar.oxfordjournals.org/content/40/D1/D306.full>, which provides more information about InterPro

Understanding the Ramachandran plot

The Ramachandran plot is a way to visualize the dihedral angle Ψ against Φ of amino acid residues in the protein structure. In principle, it shows the possible conformations of the Ψ and Φ angles for amino acid residue in a protein due to steric constraints. It also serves the purpose of structure validation. This recipe shows how to draw such a plot for a protein using a PDB file.

Getting ready

The prerequisites for the recipe are as follows:

- The `bio3d` library loaded in the R session.
- The PDB file that we need to draw a Ramachandran plot for. The PDB file can either be on a local storage or read directly from PDB.

How to do it...

To visualize the Ramachandran plot for the protein `1BG2`, perform the following steps:

1. First, load the `bio3d` library into the R session as follows:

```
> library(bio3d)
```

2. Next, read the PDB file. In this case, read the file directly from the remote data bank as follows (continue using the `1BG2` protein as in the examples before this):

```
> pdb <- read.pdb("1BG2")
```

3. Extract the torsion angles, Ψ and Φ , with the following function:

```
> tor <- torsion.pdb(pdb)
```

4. Now, simply plot the components of the torsion angles with a generic plot function of R as follows:

```
> plot(tor$phi, tor$psi, main="(A)  
Ramachandran plot 1BG2")
```

5. You can enhance the visualization by slightly sophisticated code, but first extract the torsion angles separately as follows:

```
> scatter.psi <- tor$psi  
> scatter.phi <- tor$phi
```

6. Then, create/brew the color of the contours for the new Ramachandran plot with the help of the following command:

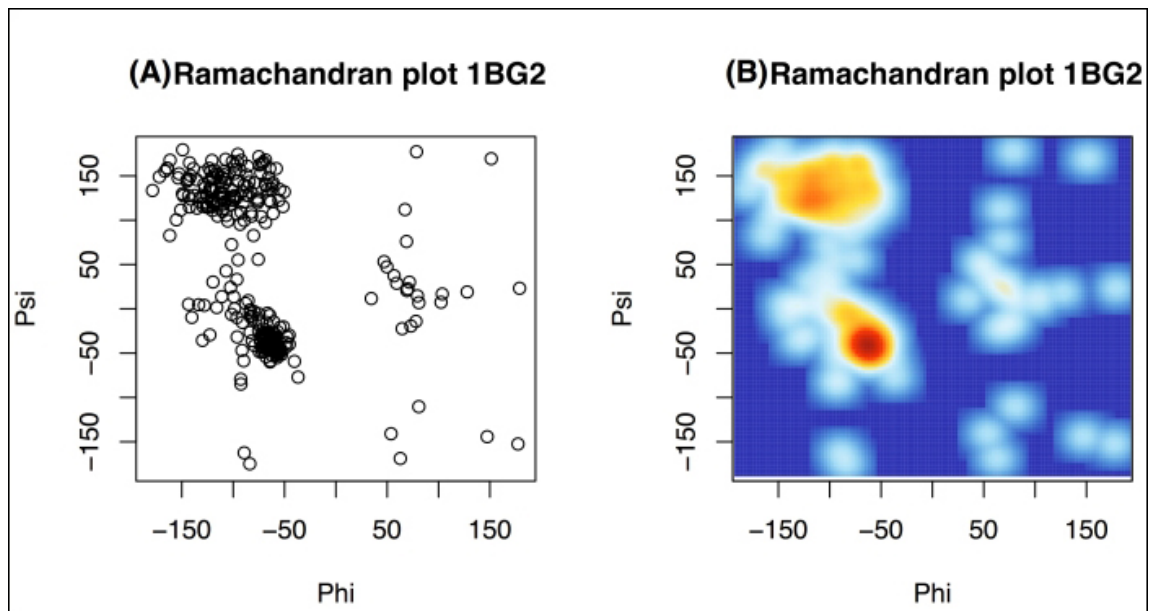
```
> library(RColorBrewer) # load RColourBrewer  
package  
> k <- 10 # define number of colours  
> my.cols <- rev(brewer.pal(k, "RdYlBu")) #  
Brew color palette
```

7. With these colors and torsion angles in place, create a better looking Ramachandran plot using the following command:

```
> smoothScatter(x=scatter_phi, y=scatter_psi,  
colramp=colorRampPalette(my.cols),  
xlim=c(-180,180), ylim=c(-180,180), xlab="Phi",  
ylab="Psi", main="(B) Ramachandran plot 1BG2",  
pch=19, cex=0.00)
```

The following plot shows a Ramachandran plot that has been drawn using the recipe code as a simple scatter plot, and a Ramachandran plot with color contours (note that both the plots show the Psi angles along the y axis and the Phi angles along

the x axis, and the corresponding areas can also be found by comparison easily):



Ramachandran plots for the PDB protein, 1BG2

How it works...

The recipe shown here takes the input PDB file and extracts the Ψ and Φ angles. The PDB file, as seen earlier, has various elements. The torsion angles Ψ and Φ are extracted by the `torsion.pdb` function. The `plot` function simply draws them along the y and x axes, respectively. The points in a Ramachandran plot refer to the pair of torsion angles observed in the structure. When we do a colored contour plot, the blue hues here represent the space not occupied by the torsion angles, whereas the white and red hues represent where the structure shows the corresponding torsion angles. The overall plot outlines the favored conformations of the protein.

See also

- The fourth chapter of the *Lehninger Principles of Biochemistry* book by Nelson and Cox, which provides more information

about the Ramachandran plot

Searching for similar proteins

Once we have a candidate protein, it is an interesting idea for a bioinformatician to look for similar proteins. This can be achieved simply by doing a blast for the sequence. While doing a blast online, we search a target sequence database for similar sequences and use the hit to look for similar relevant proteins based on their similarity scores. R allows us to do something similar. This recipe explains the method to do a blast for a PDB file directly from an R session.

Getting ready

To look for similar proteins, we need to have a protein sequence for which we use our PDB file and an active Internet connection to perform a BLAST using a web-based sequence repository.

How to do it...

Do a search for a similar protein from within R with the following set of steps:

1. First, load the `bio3d` library to read the PDB file as follows:

```
> library(bio3d)
```

2. Now, read the protein of your interest directly from PDB as follows:

```
> pdb <- read.pdb("1BG2")
```

3. Then, extract the sequence record from your PDB as a one-letter representation file using the `aa321` function as follows:

```
> mySeq <- aa321(pdb$seqres)
```

4. Run BLAST from R using the following `blast.pdb` function with the previous extracted sequence (note that this can take a while):

```
> myBlast <- blast.pdb( seq.pdb(pdb) )
```

5. We can see the head of the blast result as a table by typing the following command:

```
> head(myBlast$hit.tbl)
```

6. To get the top hit that consists of the most similar sequences, take a look at the most similar hits and plot them based on their similarity as follows:

```
> top.hits <- plot(myBlast)
> head(top.hits$hits)
  pdb.id   gi.id      group
1 "6Q21_A" "231226"    "1"
2 "6Q21_B" "231227"    "1"
3 "6Q21_C" "231228"    "1"
4 "6Q21_D" "231229"    "1"
5 "1IOZ_A" "15988032"  "1"
6 "1AA9_A" "157829765" "1"
```

How it works...

What we did in the preceding code chunks is very similar to performing a BLAST over a web-based tool. The recipe starts with extracting the sequence from the PDB file that was fetched. This sequence is then sent across the remote repository for a BLASTP search, that is, a search against protein sequences in the data repository. The search results consist of a tabular element that contains the information about the PDB IDs of the hit proteins and other IDs together with their grouping. For more information, check with `?blast.pdb` and `summary(blast)`.

Working with the secondary structure features of proteins

The structure of a protein is an important factor behind its biological role. Proteins have four major levels of structure definition. They are named the primary, secondary, tertiary, and quaternary structure. The primary structure refers to the linear sequence of amino acids in the protein, while the secondary structure is the local substructure, usually in the form of alpha helix and beta sheets together with coils. The tertiary and quaternary structures explain the three-dimensional structures and arrangement of subunits of a protein, respectively. The secondary structure of a protein is the specific geometric shape induced in the protein by the intramolecular and intermolecular hydrogen bonding of the amide groups of its residues. The α helix and β sheet are the most common secondary structure elements found in proteins. These elements play a critical role in the functioning of proteins. This recipe explains how to get the secondary structure elements of a protein.

Getting ready

To get an idea about the secondary structure of a protein, we need the `dssp` and `bio3d` packages from CRAN. Note that we can also use the stride program available at <http://webclu.bio.wzw.tum.de/stride/>. However, in this recipe, we will use `dssp` for demo purposes.

How to do it...

To deal with the secondary structure in proteins, perform the following steps:

1. First, install the `dssp` package in your machine by typing the following command (the software can be installed from the shell

terminal in Linux; for other operating systems, visit <http://swift.cmbi.ru.nl/gv/dssp/>):

```
$ sudo wget
ftp://ftp.cmbi.ru.nl/pub/software/dssp/dssp-
2.0.4-linux-i386 -O /usr/local/bin/dssp
```

2. In case there is a lack of `sudo` rights, install the software in your personal directory or contact the system administrator. The detailed instruction on the installation of `bio3d` allows the installation of `dssp` in parallel.
3. Following this, make the `dssp` package executable with the following `chmod` command in shell (the `dssp` package is now installed and executable):

```
$ sudo chmod a+x /usr/local/bin/dssp
```

4. Now, start the R session and then install and load the `bio3d` library as follows:

```
> library(bio3d)
```

5. Load the protein within the `Rknots` package in the R session as follows (remember that in our case, we use a different function to load a PDB file):

```
> protein <- read.pdb("1BG2")
```

6. To get the secondary structure, simply use the following `dssp` function:

```
> SS <- dssp(pdb)
```

7. The returned object is a table of residues and their secondary structure in the form of alpha helix, beta sheets, and so on. Check it out by typing the following command:

```
> head(SS)
```

How it works...

The function in this recipe uses `dssp`. It is a database of the secondary structure assignments (and much more) for all the protein entries in the PDB. Once the R function has the PDB file, it goes to the path given as an input in the `dssp` function to execute the original `dssp` program and then returns the results obtained from the database. It should be noted that `dssp` is a secondary structure database as well as a program. In this recipe, we used the `dssp` program.

Visualizing the protein structures

Visualizing the three-dimensional structure of a protein is crucial for several purposes in protein research. The ability to visualize these structures is critical to areas such as drug design and protein modeling. There exist many tools for structure visualization and, more importantly, interactive 3D visualization. Common examples of these are RasMol, Jmol, and Cn3D. The RasMol tool accepts the PDB and MMDB formats, whereas the Cn3D tool reads the ASN.1 format, which is more refined. Besides this, Cn3D can also read the structure (in the form of ASN.1 files) directly from the remote databases and provides faster rendering. Honestly speaking, as of now, R does not provide a tool that is any close to the available popular tool for the purpose. However, for the sake of completeness, we introduce a simple method in this recipe to look at the structure of a protein in R.

Getting ready

In this recipe, we need the [Rknots](#) package for visualization purposes along with the protein data (or PDB ID) of our interest. The [Rknots](#) package has actually been designed for the topological analysis of polymers.

How to do it...

To visualize the protein structure from within R, perform the following steps:

1. First, install and load the [Rknots](#) package from the CRAN repository as follows:


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

2. Then, load the proteins from PDB as follows:

```
> myprotein <- loadProtein("1BG2")
```

3. You can then see the structure of the protein with the help of the following commands, but in 2D:

```
> plotDiagram(myprotein$A, ends = c(), lwd = 2.5)
```

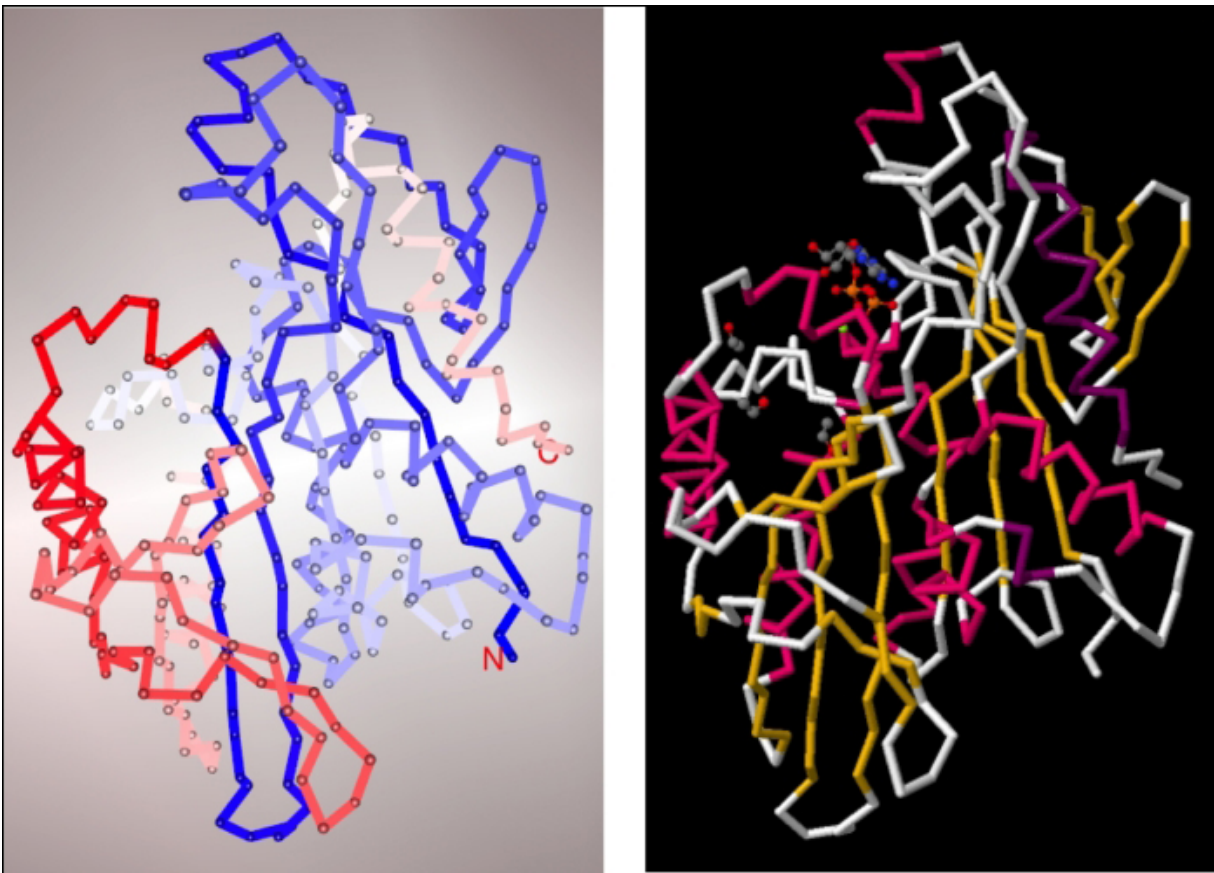
4. Install RGL graphics for 3D visualization on your machine, and plot the visualization with the following function:

```
> ramp <- colorRamp(c('blue', 'white', 'red'))
> pal <- rgb(ramp(seq(0,1,length=100)),
max=255)
> plotKnot3D(myprotein$A, colors=list(pal),
lwd=8, radius=0.4, showNC=TRUE, text=FALSE)
```

How it works...

The `loadProtein` function retrieves the PDB information for the input ID in the R session. The `plot` function then creates the structure in 2D. To visualize the structure, we created a color ramp with varying hues from red to blue (100 equal divisions in the color palette—argument `length` is equal to 100). Following this, we plotted the backbone structure that uses RGL graphics. The method in this recipe has been presented for completeness. It is advisable that you use other tools for better visualization of protein structures, such as RasMol, Jmol, Cn3D, or VMD. However, it is possible to write the data to a local PDB file using the `write.pdb` function and then visualize it using your favorite tool (in our case, we used Jmol). We can do it for our `1BG2` PDB file and compare the results. However, you can do it by directly downloading the file from PDB. This can be useful for locally created files. The following screenshot shows the

visualization of the protein 1BG2 that uses the [Rknots](#) package (left) and the [Jmol](#) package (right)—note that the structures have been positioned and oriented to make them comparable:



Chapter 5. Analyzing Microarray Data with R

This chapter will deal with the following recipes:

- Reading CEL files
- Building the ExpressionSet object
- Handling the AffyBatch object
- Checking the quality of data
- Generating artificial expression data
- Data normalization
- Overcoming batch effects in expression data
- An exploratory analysis of data with PCA
- Finding the differentially expressed genes
- Working with the data of multiple classes
- Handling time series data
- Fold changes in microarray data
- The functional enrichment of data
- Clustering microarray data
- Getting a co-expression network from microarray data
- More visualizations for gene expression data

Introduction

Microarrays are one of the most popular tools to understand biological phenomenon by large-scale measurements of biological samples, typically DNA, RNA, or proteins. The technique has been used for a range of purposes in life science research, ranging from gene expression profiling to SNP or other biomarker identification, and further, to understand relations between genes and their activities on a large scale. However, reaching such inferences from the huge amount of data available on a single chip is challenging in terms of biology as well as statistics. The domain of microarray data analysis has revolutionized the research in biology during the last

decade. In this chapter, we will explore the different methods that are used to analyze gene expression data, which is received from microarrays that serve different biological goals. There can be many types of microarrays depending on the biological sample used. Likewise, there are different techniques in use to produce these arrays, namely, Affymetrix, Illumina, and so on. In this chapter, we mainly focus on microarrays to measure gene expression with nucleic acid samples and use Affymetrix CEL file data for explanations. Nevertheless, most of the techniques can be used on other platforms with slight modifications.

The chapter will start with the reading and loading of microarray data, followed by its preprocessing, analysis, mining, and finally, its visualization using R. Then, the solution related to the biological use of the data will be presented.

Reading CEL files

The CEL file contains all the intensity-related information of the pixels on the array. The files include the intensity itself, the standard deviation, the number of pixels, and other meta information. To get a feel of the file, we can open the CEL file with any text editor and take a look at it. Each experiment usually has more than one sample and replicates; therefore, there will be one CEL file present for each sample or replicated file. The CEL files must be read to get the data in a workable format in the R workspace. This recipe will explain how we can read such a file in the R workspace.

Getting ready

To read the files, the first thing we need are the input CEL files. To do this, we will use the NCBI GEO database for a simple dataset on breast cancer. Furthermore, we will need some R packages. All this is explained in detail in the following section.

How to do it...

To get acquainted with a CEL file, try to read it in your R session using the following steps:

1. Download [GSE24460](http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE24460) from NCBI GEO (<http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE24460>). This gives you a file named `GSE24460_RAW.tar` in your desired directory.
2. Unzip this file to get CEL files. The files have also been provided with the code files (available on book's web page) the sub directory `<GSE24460_RAW>`.
3. Next, install and load the `affy` library into the R session as follows:

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

```
> biocLite("affy")
> library(affy)
```

4. To read all the files in the directory, use the `ReadAffy` function as follows:

```
> myData <-
  ReadAffy(celfile.path="path/to/dir")
```

5. If you wish to read only one or a couple of files, specify the filename as follows:

```
> myData1 <- ReadAffy(filenamees =
  filenames.of.your.data)
```

How it works...

In this recipe, the data we used comes from a study on the breast cancer cell. The `ReadAffy` function is a wrapper for another function called `read.affybatch` that allows the reading of the CEL file along with other relevant data into an `AffyBatch` object. The `ReadAffy` function gets the list of all of the CEL files (even the compressed TAR files) in the current working directory and passes it to the `read.affybatch` function to create the `AffyBatch` object from these files. It is possible to provide arguments for other information such as `phenoData` and specific CEL filenames and paths to another directory with the same function. When no arguments are passed into the `ReadAffy` function (as `ReadAffy()`), the function reads all the CEL files in the working directory.

The R library called `affyio` also contains routines to parse Affymetrix datafiles based on file format information. However, the `affy` library has an advantage because it has a lot of other functionalities for data analysis.

There's more...

In this recipe, we mainly focused on Affymetrix CEL files. The use of tabular files is getting popular for the gene expression data. This gives us the flexibility to read the data using the `read.csv` or similar other functions. When reading these files, we can extract the data columns from the expression data and use it accordingly.

See also

- The article titled *affy—analysis of Affymetrix GeneChip data at the probe level* by Gautier and others at, which provides the theoretical aspects of the `affy` library in Bioconductor
- The article titled *Getting Started in Gene Expression Microarray Analysis* by Slonim and Yanai at <http://www.ploscompbiol.org/article/info%3Adoi%2F10.1371%2Fjournal.pcbi.1000543> to understand how microarrays are really produced
- The *Prolonged Drug Selection of Breast Cancer Cells and Enrichment of Cancer Stem Cell Characteristics* article by Calcagno and others at <http://jnci.oxfordjournals.org/content/102/21/1637.long>, which provides information about the data we used in this recipe

Building the ExpressionSet object

The `ExpressionSet` class in Bioconductor represents a combination of several different sources of information into one data structure. For an array, it contains the intensities, phenotype data, and experiment information as well as annotation information. When we read a set of CEL files using the `ReadAffy` or `read.affyBatch` function, an `AffyBatch` object is created that extends the `ExpressionSet` structure. The `AffyBatch` object is probe-level data, whereas `ExpressionSet` is probeset-level data, which is extended to a probe level by `AffyBatch`. Sometimes, we have intensity values in the form of a table, matrix, or data frame together with phenotype data, experiment details, and annotations as separate objects (or files). We must create an `ExpressionSet` object from these individual files from scratch to facilitate the analysis work. This recipe will present the solution to this problem.

Getting ready

For this recipe, we need files with different types of information, such as assay data, phenotypic metadata, feature annotations and metadata, and a description of the experiment. We create these files by simply writing the required components of the `myData` object created in the previous recipe to different files. We need the chip annotation information for this as well.

How to do it...

To build `ExpressionSet`, perform the following steps:

1. Install and load the `Biobase` library, if not already loaded (it gets loaded by default when you load the `affy` library), as follows:


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

2. As a demo expression file and the **phenotypic data (pData)** file, we will use the built-in data for the `Biobase` library, whose location can be fetched as follows:

```
> DIR <- system.file("extdata",
package="Biobase")
> exprsLoc <- file.path(DIR, "exprsData.txt")
> pDataLoc <- file.path(DIR, "pData.txt")
```

3. Read the table from the text file that contains the expression values using the usual `read.table` or `read.csv` function as follows:

```
> exprs <- as.matrix(read.csv(exprsLoc, header
= TRUE, sep = "\t", row.names = 1, as.is =
TRUE))
```

4. Now, check the object created previously. It should be a matrix:

```
> class(exprs)
> dim(exprs)
```

5. Now, read the phenotype information file in a similar way using the `read.csv` function as follows:

```
> pData <- read.table(pDataLoc, row.names = 1,
header = TRUE, sep = "\t")
> pData <- new("AnnotatedDataFrame", data =
pData)
```

6. Compile the experiment information, an object of the `MIAME` class with slots for investigator name, lab name, and so on as follows:

```
> exData <- new("MIAME", name="ABCabc",
```

```
lab="XYZ Lab", contact="abc@xyz", title="",  
abstract="", url="www.xyz")
```

7. It is also important to know the chip annotation as it is a part of the `ExpressionSet` object for this data. Use the `hgu95av2` chip annotation for this purpose.
8. Now, create a new `ExpressionSet` object using the information compiled in the previous steps as follows:

```
> exampleSet <- new("ExpressionSet", exprs =  
exprs, phenoData = pData, experimentData =  
exData, annotation = "hgu133a2")
```

9. To check your object, simply type in the object name or check the structure with the `str` function as follows:

```
> str (exampleSet)
```

10. Test the validity of the object created before continuing with the analysis, as follows:

```
> validObject(exampleSet)
```

11. To convert an `AffyBatch` object to `ExpressionSet`, simply use the `AffyBatch` components directly to create a new `ExpressionSet` object, as shown in step 6.

How it works...

In this recipe, we read different information files individually using the conventional `read.csv` function in a matrix or data frame. The expression data is a matrix that contains the intensities measured, whereas the phenotypic data carries information about the conditions (for example, control or disease) of the data and samples. The experimental data simply has certain formal information, and it is not obligatory to fill it in. As the order is very important for the final `eSet`, we check the validity of the created object. The annotation chip used is because the built-in data for the package actually comes from the `hgu133a2Affymetrix` chip. For example, if the sample names in the

expression data and phenotypic data are different, the function will return the object as invalid. These individual objects are then assembled into `ExpressionSet` by creating a new object. Each component of `ExpressionSet` has its own role. The `exprs` object is the expression data, the phenotypic data summarizes information about the samples (for example, the sex, age, and treatment status—referred to as covariates), and the annotated package provides basic data manipulation tools for the metadata packages. This can be done with any platform, be it Affymetrix or Illumina.

Handling the AffyBatch object

The `AffyBatch` object will be used throughout the chapter for data analysis purposes. As we have seen, it can be created by reading the CEL files for an experiment together with the other allocated information. The `AffyBatch` object has various components. This recipe aims to look at the various components of such an object.

Getting ready

For our use, we will use the `AffyBatch` object created in the previous recipes (the `myData` object from the *Reading CEL files* recipe).

How to do it...

Perform the following operations to get information about the `AffyBatch` object:

1. Check an `AffyBatch` object by simply typing in the object name as follows:

```
> myData
```

2. Check the structure of the object as follows:

```
> str(myData)
```

3. Check the phenotype data of the object using the `pData` and `phenoData` functions as follows:

```
> pData(myData)
> phenoData(myData)
```

4. To get the expression data as a matrix, use the `exprs` function as follows:

```
> exprs(myData)
```

These values can be written to a separate file using the `write.csv` function.

5. Use the following annotation function to get the annotation name of the object:

```
> annotation(myData)
```

6. To get the probe names or sample names in the data, use the `probeNames` and `sampleNames` functions as follows:

```
> probeNames(myData)
> sampleNames(myData)
```

How it works...

The `AffyBatch` object has a complex structure that consists of many components and subcomponents, as observed while looking at the structure of the object. At every step of the recipe, we check individual components of the complete `AffyBatch` object. The components of the `AffyBatch` object, such as expression data, can be extracted and written into a separate file using the `write.csv` or similar functions.

Checking the quality of data

The quality of data can be affected at each step of the microarray experiment pipeline. Quality-related problems could stem from hybridization due to uneven fluorescence on the chip that causes variable intensity distributions. A nonspecific binding or other biological/technical reasons can create background noise in the data. Another possible situation can be an inappropriate experimental design that may affect the dataset as a whole. Using such data will result in wrongful or inconclusive inference during data analysis. Therefore, as analysts we must ensure the data quality before we begin the data analysis. This is achieved by looking for outlying arrays, distributions within arrays, batch effects, and so on. There are various analyses and diagnostic plots that can be used to compute these measures that explain the quality of array data under analysis. This recipe will explain various diagnostic steps for the quality check of data.

Getting ready

To begin with, we need the array data. We will be using the same data that we used in the previous recipes. In this recipe, we will also introduce a new R package that can serve most of our quality assessment processes.

How to do it...

In order to look at the data quality, create some diagnostic tests and plots as follows:

1. Install and load the `arrayQualityMetrics` library from the Bioconductor repository as follows:

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

```
> biocLite("arrayQualityMetrics")
> library(arrayQualityMetrics)
```

2. Use the following `arrayQualityMetrics` function to create plots to assess the data quality:

```
> arrayQualityMetrics(myData,
  outdir="quality_assessment")
```

3. Go to the created subdirectory (in your case, `quality_assessment`) in the current working directory to check the created HTML page (`index.html`) and plots.
4. Open the HTML file in a browser of your choice. Use the following line of code for this purpose:

```
> browseURL(file.path("quality_assessment",
  "index.html"))
```

5. You can create these plots and assessments individually as well. For example, to create an MA plot, use the `MAplot` function as follows:

```
> MAplot(myData, pairs=TRUE,
  plot.method="smoothScatter")
```

6. To plot the log densities, type in the following command:

```
> plotDensity.AffyBatch(myData)
```

7. To create the boxplots, simply use the `boxplot` function in the `AffyBatch` object as follows:

```
> boxplot(myData)
```

8. To get the RNA degradation plot, use the `AffyRNAdeg` function and then `plotAffyRNAdeg` as follows:

```
> rnaDeg <- AffyRNAdeg(myData)
> plotAffyRNAdeg(rnaDeg)
```

9. Check the details of the `rnaDeg` object as follows:

```
> summaryAffyRNAdeg(rnaDeg)
```

How it works...

The `ArrayQualityMetrics` function takes the data and performs different types of checks on it. The checks include measuring between arrays distances, **Principal Component Analysis (PCA)**, density plots, MA plots, and RNA degradation plots. The outlier detection among the arrays is reflected by measurements such as the distance between the arrays, boxplots, and MA plots. A detailed description is available in the HTML file produced. For example, to measure the distances d_{ij} between the arrays i and j , we can use the following formula:

$$d_{ij} = \text{mean} | I_{ia} - I_{ja} |$$

In the preceding formula, I_{ia} and I_{ja} are the intensity measurements for the a^{th} probe in the arrays i and j , respectively.

The MA plot, also referred as a derivation from the Bland Altman plot, is based on two components, namely, M and A. The M component represents the ratio between two channels (or two arrays), thus giving an indication which color is binding more at a given spot. The A component is a measure of the log2 transformed intensity at the spot. Plotting these two components in two dimensions (usually M along the y axis and A along the x axis) gives us an idea about the intensity bias in the data. This means the differences in two channels or two arrays can be helpful in detecting background or outliers such as phenomena in the data. For instance, we can use them for the pairwise comparison of arrays or to compare the intensities of two dyes in two-channel data. A deviation from the $M=0$ line (asymmetrical distribution along $M=0$) indicates intensity bias, outliers, or even **differentially expressed (DE)** genes. The deviation in the plot can be corrected to some extent by

normalization. (In fact, it is often used as an indicator for normalization together with the boxplot, which will be discussed later.) Any trend in the left side (lower A) indicates the presence of background, while a trend on the right (higher A) depicts saturation.

Most of the spots in the plot are usually around the $M=0$ line with small interquartile ranges across the arrays and probably represent **non-differentially expressed (non DE)** genes. However, one should reach such conclusions only after background correction and normalization.

The next type of plot we discussed is the intensity plot that shows the density estimates of the data. Decent data has a similar shape and range across the arrays. This plot gives the different types of information in the data. A data with high background noise will shift the entire distribution towards the right. If the distribution shows a diminished right tail, it indicates signal loss. The upper bulge of the distribution indicates signal saturation. A multimodal shape (that is, more than one mode has peaked) in the plot accounts for a spatial artifact (such as regional bias in the array).

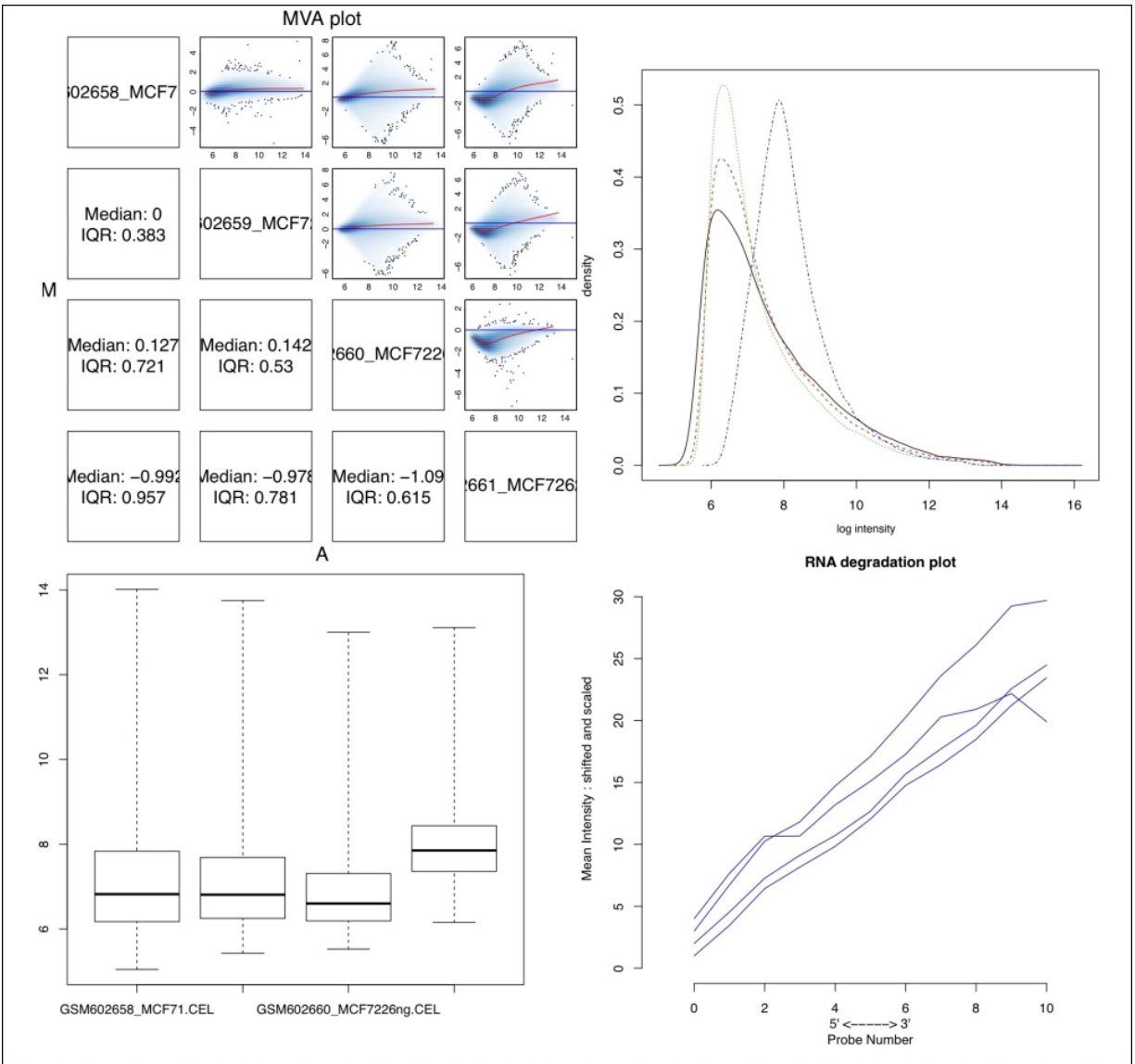
The boxplots of high-quality data show similar width and positions, and represent the distribution of signal intensities in the data. The distribution is usually done on the log scale to make the plot readable. A major deviation in the boxplot might represent an experimental flaw or noise in that particular array. A **Kolmogorov Smirnov (KS)** statistic on these distributions is used to detect outliers in the data. Another important feature of the boxplot is that deviation can be overcome by the normalization of data most of the time, which will be covered in upcoming recipes.

The RNA degradation plot gives an indication for the quality of samples used in array hybridization. Generally, mRNAs have a certain lifespan, after which they are degraded and hence not effective to measure the expression levels. The degradation starts at the 5' end moving towards the 3' end. Therefore, as an effect of this degradation, the intensities should be lower at the 5' end compared to the 3' end. The expression (intensity measurement) of all the

probes on an array gives the level of degradation in the sample. We represent this as an RNA degradation plot, where probes are numbered sequentially from the 5' end to the 3' end of the molecule. So, plotting the intensities should show an upward trend along the probe numbers (more degradation at the 5' end, hence low intensity, and vice versa). In this recipe, we checked whether the lines in the plot are following a consistent trend. A deviation indicates issues with the sample used for hybridization.

All these plots, plus some others, with a short abstract description are reported in the form of an HTML file in the desired output directory. In this recipe, we mentioned four of these plots. The general meaning of each plot is explained in the HTML file itself; however, it is advisable that you use your own reasoning when inferring from these plots as every experiment can differ from each other and should be analyzed accordingly.

The `affy` package has the functions to do most of the analyses and can create plots that are explained in this recipe individually. The following screenshot shows the MA plots for the data (top left), with the blue line indicating $M=0$ and the red line indicating the loess fit. You can also see the density plots (top right) with deviations among the arrays. In the following screenshot, the boxplots (bottom left) with non-normalized data have been indicated by the deviation in the position and size of the boxes and the RNA degradation plot (bottom right) with one array that shows the deviation (for a description of each plot, refer to the *How it works...* section of this recipe):



Diagnostic (quality check) plots for raw data

Generating artificial expression data

Developing new methods of analyzing expression data requires proper testing and performance checks on large, high-quality datasets obtained from many experimental conditions. This requires benchmark data with known parameters. Such data from experiments is usually not available, and performing such experiments in a wet lab is not economical. Therefore, generating well-characterized, synthetic datasets that allow thorough testing of learning algorithms in a fast and reproducible manner is needed. It is common practice to use simulated data (artificial) sets for such purposes. This recipe will present the approach to generate such datasets.

Getting ready

Before we start generating the artificial data, we must know how do we want our data to look in terms of the bounds (upper and lower), fraction of DE genes, amount of data, and associated statistical parameters, which will be explained in the *How it works...* section of this recipe.

How to do it...

In this section, we will generate a dataset of 35,000 genes with 1 percent of DE genes. In order to do so, perform the following steps:

1. Before starting the data generation process, first install and load the `madsim` library from the CRAN repository as follows:

```
> install.packages("madsim")  
> library(madsim)
```

2. Now, define your first set of parameters for the simulation process for the overall statistical parameters and the distribution along arrays as follows:

```
> fparams <- data.frame(m1 = 7, m2 = 7, shape2  
= 4, lb = 4, ub = 14, pde = 0.02, sym = 0.5)
```

3. Define the second set of parameters that consists of the statistical parameters that define the level of expression in the genes as follows:

```
> dparams <- data.frame(lambda1 = 0.13,  
lambda2 = 2, muminde = 1, sdde = 0.5)  
> sdn <- 0.4  
> rseed <- 50
```

4. Then, define the number of genes you require in the expression data as follows:

```
> n <- 35000
```

5. Now, generate the synthetic data as follows:

```
> myData <- madsim(mdata=NULL, n=35000,  
ratio=0, fparams, dparams, sdn, rseed)
```

6. While looking at the structure of the object created by `madsim`, you can see that it consists of three components:

```
> str(myData)
```

7. To visualize the data, create an MA plot for any sample, say, #sample 1, using the following function:

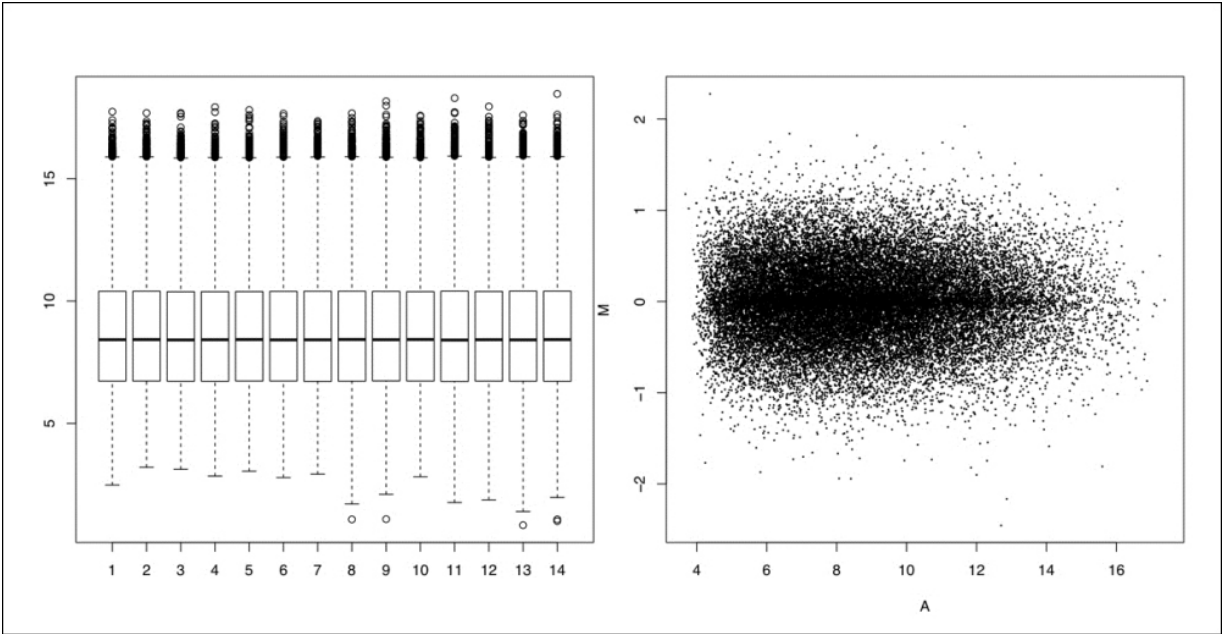
```
> library(limma)  
> plotMA(myData[[1]], 1)
```

How it works...

The `madsim` package generates data for two biological conditions when the characteristics are known in terms of statistical parameters. The parameters used in the `madsim` model allow the user to generate data with varying characteristics. They use the following four components to generate the expression values of a gene:

- The expression levels of non-DE genes
- The expression levels of DE genes
- Noise
- Technical noise

The overall expression level is a function (in our case, sum) of these four components. The `madsim` function uses a beta distribution to generate n values between 0 and 1. The shape parameters in the argument, `fparams`, define this beta distribution—a continuous, statistical distribution defined by two shape parameters. It is then scaled to fit the upper and lower bounds that are defined. A set of randomly selected DE genes as per the `pde` argument in `fparam` is extracted. The properties to generate the expression values are specified using the `dparm` argument (standard deviation and mean); this results in the differential expression of these genes in simulation. The following plot shows the boxplot and MA plot for artificially generated data:



There's more...

There are other ways to simulate microarray data. The package named `optBiomarker` has the `simData` function that can simulate artificial data as well. The package can be installed from the CRAN repository. To know more, refer to the help manual of the package. The details of the `optBiomarker` package are available at <http://cran.r-project.org/web/packages/optBiomarker>.

See also

- The article titled *A Flexible Microarray Data Simulation Model* by D Dembélé at <http://www.mdpi.com/2076-3905/2/2/115>, which provides more information about the `madsim` package

Data normalization

Microarrays are high-throughput methods that measure the expression levels of thousands of genes simultaneously. Each sample receives different conditions. A small difference in RNA quantities or/and experimental errors may cause the intensity level to vary from one replicate to the other. This can be irrespective of the biological expression of genes. Handling this inherent problem requires the normalization of data. This minimizes the technical effects, rendering the data comparable. This recipe will explore a few of the many normalization methods developed for data normalization in R.

Getting ready

To start with the normalization recipe, we need to define our data as an `ExpressionSet` or `AffyBatch` object. Here, we will use the breast cancer data object that we created in our first recipe.

How to do it...

Start with the normalization process using the following steps:

1. Get the data into the R session as an `AffyBatch` object or a matrix, already shown in previous recipes, as follows (in our case, we use the `myData` object that was created in the *Reading CEL files* recipe; let's repeat the step again for convenience):

```
> myData <-  
  ReadAffy(celfile.path="path/to/dir")
```

There are many possible types of normalizations. In this recipe, we focus on three methods: the `vs`, `loess`, and `quantile` normalizations.

2. To do a `vsn` normalization, you need a library named `vsn`. Install and load it into the R session as follows:

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

3. Perform the `vsn` normalization on an `AffyBatch` object with the following command:

```
> myData.VSN <-
  normalize.AffyBatch.vsn(myData)
```

4. Create a boxplot for the normalized data and compare it with the boxplot for non-normalized data in the previous recipe as follows:

```
> boxplot(myData.VSN)
```

5. The `loess` normalization uses the `affy` library. To do the `loess` normalization, use the `normalize.AffyBatch.loess` function as follows:

```
> myData.loess <-
  normalize.AffyBatch.loess(myData)
```

6. The `quantile` normalization uses the `affy` library. To do the `quantile` normalization, use the `normalize.AffyBatch.quantile` function as follows:

```
> myData.quantile <-
  normalize.AffyBatch.quantiles(myData)
```

7. Again, plot the boxplot for these two methods in a way similar to the one shown in step 5.

Normalization can also be performed using the `normalize.loess` and `normalize.quantile` functions with the data matrix as the input argument if the data is only a matrix of intensities.

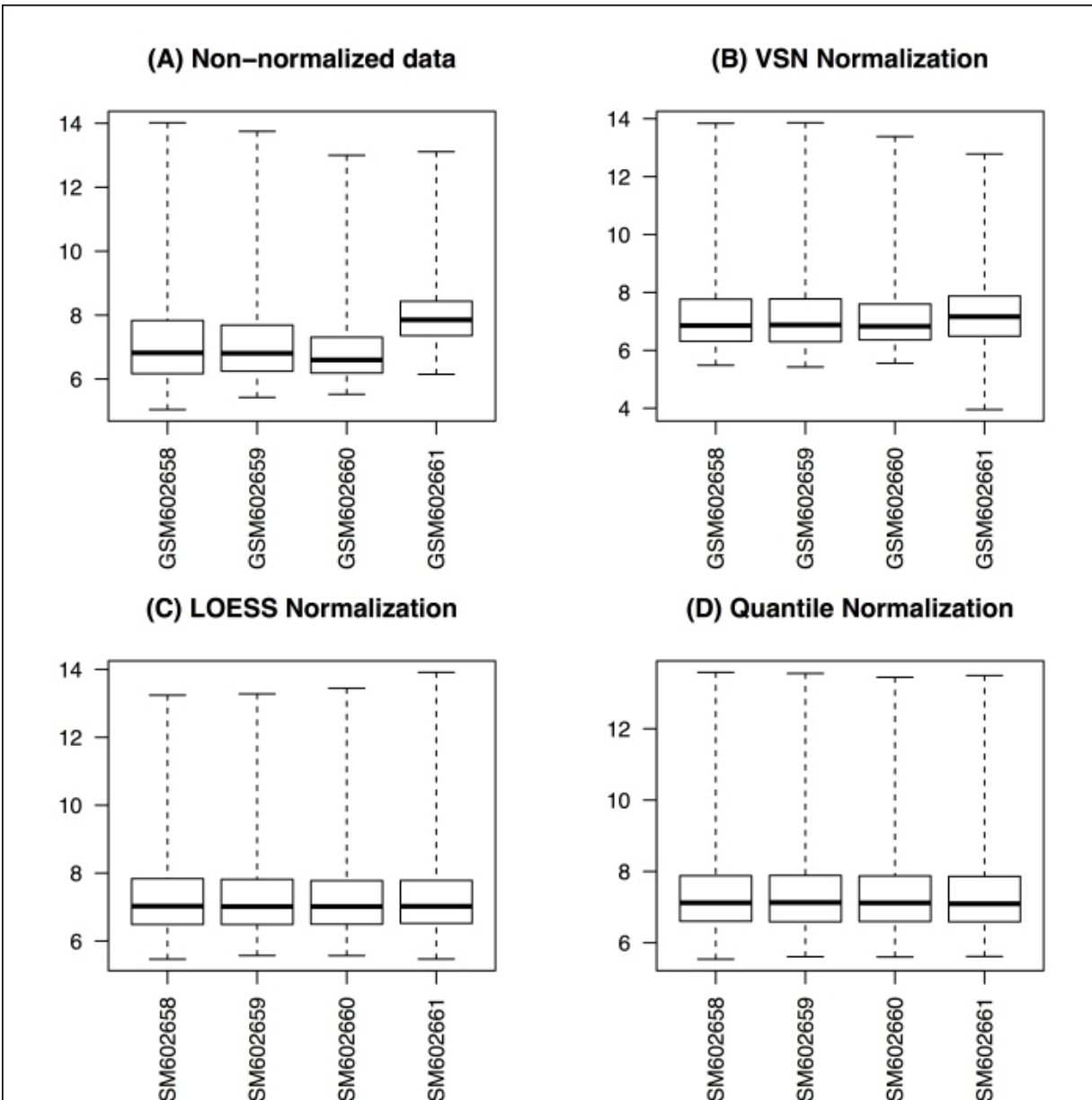
8. Do a second round of quality check for the normalized data and observe the effect of normalization in the same way as before using `arrayQualityMetrics`.

How it works...

Variance Stabilization and Normalization (VSN) is based on the assumption that the variance of microarray data depends on the signal intensity and there exists a transformation that keeps the variance approximately constant. This means that the `vsn` method finds a transformation of the intensity measures in the data so as to keep the variance of intensity approximately independent of its mean. The `normalize.AffyBatch.vsn` function is actually a wrapper for the `vsn` function in the `vsn` library (not the `affy` library).

The second method, the `loess` normalization, uses a locally weighted regression to normalize the data. The method fits a smoothing curve to a dataset. The degree of smoothing is determined by the window width parameter. A larger window width results in a smoother curve, and a smaller window results in a more local variation. The `normalize.AffyBatch.loess` function actually uses the `loess` function of R to fit and smooth the data. The window size used by default is 2/3, but can be modified with the `span` argument.

Finally, the `quantile` normalization uses a simpler concept of adjusting the quantiles of the distribution in an array to make all the quantiles alike and a common median center. This makes the histograms of the arrays look alike. The following boxplots show the boxplots for non-normalized data (A) and the normalized data using the `vsn` normalization (B), the `loess` normalization (C), and the `quantile` normalization (D):



There's more...

There are some other preprocessing steps that can be performed on the raw data. They include background correction, log transformation, and so on. They are mostly included within the normalization methods. The background correction in the `affy` package can be looked into with the `?bgcorrect.methods` function. The log transformation can be set to the Boolean values `TRUE` or `FALSE` by changing the `log.it` argument in the normalization functions of `affy`.

There are other methods of normalization within the `affy` library, namely, `qspline`, `invariantset`, `contrasts`, and so on. To check the various available methods, type in the following command in your R session with the `affy` library preloaded:

```
> normalize.AffyBatch.methods()
```

The normalization method is a matter of choice and need. It depends a lot on the data platform and biological experiments. So far, the methods explained in this recipe are the most popular ones among bioinformaticians.

See also

- The *Microarray data normalization and transformation* article by J Quackenbush at <http://www.nature.com/ng/journal/v32/n4s/full/ng1032.html>, which provides a basic insight into the different normalization methods

Overcoming batch effects in expression data

Batch effects are the systematic errors caused when samples are processed in different batches. They represent the nonbiological differences between the samples in an experiment. The reason can be the difference in sample preparation or hybridization protocol, and so on. It can be reduced, to some extent, by careful, experimental design but cannot be eliminated completely unless the study is performed under a single batch. Batch effects render the task of combining data from different batches difficult. This ultimately reduces the power of statistical analysis of the data. This needs appropriate preprocessing before the batches are combined. This recipe will present these preprocessing techniques.

Getting ready

The recipe will require a dataset that shows the batch effect without any preprocessing. We consider the `bladderbatch` data, which consists of five batches. The data is a part of the `bladderbatch` package. All of the other R libraries will be introduced in the next section.

How to do it...

In order to remove the batch effects, perform the following steps:

1. First, load all the required libraries. If they are not installed, you first need to install them as you did before from their respective repositories using the following commands:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("sva", "bladderbatch"))
> library(sva) # contains batch removing
```

```
utilities  
> library(bladderbatch) # The data to be used
```

2. Then, load `bladderdata` using the following `data` function:

```
> data(bladderdata)
```

3. Now, extract the expression matrix and `pheno` data from it as follows:

```
> pheno <- pData(bladderEset)  
> edata <- exprs(bladderEset)
```

4. Before you start working with the data, take a look at the phenotypic information by typing the following command:

```
> pheno
```

5. You can see that the first eight samples are normal cells but split into two batches (batch number 2 and 3). Use these eight samples to demonstrate how to remove the batch effects. To select these samples, use the following function:

```
> myData <-  
bladderEset[, sampleNames(bladderEset)[1:8]]
```

6. To have a look at the batch effect, perform a quality check on the data. Do this with the help of the following `arrayQualityMetrics` function:

```
> arrayQualityMetrics(myData, outdir="qc_be")
```

7. Take a look at the heatmap and the clustering tree produced for the samples to check for the batch effect.
8. Now, create the model matrix for the dataset as follows (note that only the first and third columns have been used from the model matrix as the data has only one condition):

```
> mod1 <- model.matrix(~as.factor(cancer),
```

```
data=pData(myData)[,c(1,3)]
```

9. Then, define the batches by typing the following command:

```
> batch <- pData(myData)$batch
```

10. Extract the expression matrix from the expression set object `myData`, where the batch effect has to be removed, as follows:

```
> edata <- exprs(myData)
```

11. Once you have all the objects ready, run the `ComBat` function as follows:

```
> combat_edata <- ComBat(dat=edata,  
batch=batch, mod=mod1, numCovs=NULL,  
par.prior=TRUE)
```

12. Now, create an expression set object with everything as your original input data, except the expression matrix—which is replaced by the matrix received as a result of the `ComBat` function in the last step—as follows:

```
> myData2 <- myData  
> exprs(myData2) <- combat_edata
```

13. Now, rerun the following `arrayQualityMetrics` function to check for the elimination of batch effects with this new object as the input:

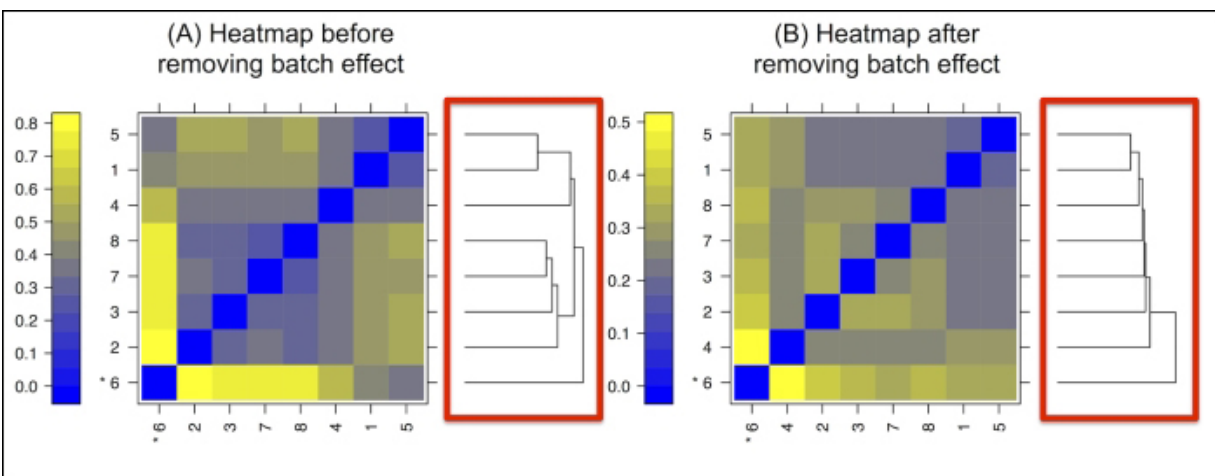
```
> arrayQualityMetrics(myData2,  
outdir="qc_nbe")
```

14. Again, take a look at the heatmap and clustering tree generated by the preceding function and check whether the batch effects in the data have been eliminated.

How it works...

The data we used originates from a bladder cancer study, where expression profiling was used to examine the gene expression patterns in **superficial transitional cell carcinoma (STCC)** with and without surrounding **carcinoma in situ (CIS)**. The data was produced on different dates and, hence, typically shows substantial batch effects; these can lead to confusing or incorrect biological conclusions, owing to the influence of technical artifacts.

The function accepts the data matrix that contains the expression values, the batch number for each sample in the input data, and the model matrix that represents the sample information. For other optional input arguments, type in `?ComBat` in your R console. The `ComBat` function uses an empirical Bayes method to combine the batches. It estimates parameters for the location and scale adjustment (LS) of each batch for each gene independently. Pooling information for multiple genes in each batch brings out the pattern that genes with similar expressions follow. This information is then used to adjust the batches in order to eliminate the batch effect. The patterns in the heatmap may show two factors: the intended biological effect or unintended batch effects. The batches can be observed in the heatmap and the clustering tree. We can see the branching as well as distinct color patterns in the heatmap for every batch, while after running the `ComBat` function some of these branches in clustering trees merge. The following heatmaps shows the clustering tree with separate (A) and merged (B) batches, which has been boxed externally (not a part of R code) for visual emphasis:



Heatmaps for the first eight samples (normal cells) in the bladder data that show the batch effect (A) and the removal of the batch effect (B)

See also

- The article titled *Adjusting batch effects in microarray expression data using empirical Bayes methods* by Johnson and Li, which provides sufficient theoretical details on batch effects
- The article titled *Adjustment of systematic microarray data biases* by Benito and others at <http://bioinformatics.oxfordjournals.org/content/20/1/105.long>, which provides information on methods such as mean clustering based methods, **support vector machine (SVM)**-based methods, and **distance weighted discrimination (DWD)**-based methods to eliminate the batch effect
- The *Gene Expression in the Urinary Bladder: A Common Carcinoma in Situ Gene Expression Signature Exists Disregarding Histopathological Classification* article by Dyrskjøt and others at <http://cancerres.aacrjournals.org/content/64/11/4040.long>, which provides details on the data used in this recipe

An exploratory analysis of data with PCA

Let's imagine that we measure approximately 20,000 human genes in 10 samples and get a matrix of 20,000 x 10 measurements. Now, imagine that each of these 20,000 genes (serving as features) as a cloud of values in a multidimensional space. Envisioning a pattern in such a cloud is difficult; therefore, we need to transform the multidimensional cloud in lower dimensions to explain and graphically represent the patterns in the data. Organizing and combining the features in order to explain the maximum variability in the data can help achieve this. **Principal Components Analysis (PCA)** is a method that achieves this by performing a covariance analysis between factors. This finds the orthogonal components that represent the data and each component (called principal components) that represents the dimension where the features are more extended.

Thus, PCA projects data onto a lower dimensional space and can be used as an exploratory method to serve purposes such as finding patterns in data and noise reduction. In this recipe, we will deal with PCA for microarray data.

Getting ready

As usual, we will start with our breast cancer data, as we did in the previous cases.

How to do it...

To perform the PCA for your data and to show the first few principal components, perform the following steps:

1. You first need data to work with. Use the data that you used before from the breast cancer example as follows:

```
> library(affy)
> myData <-
  ReadAffy(celfile.path="path/to/dir")
```

2. Then, select your data for the analysis by typing the following command:

```
> myData.pca <- exprs(myData)
```

3. Transpose the matrix to get the genes (features) as columns by typing the following command:

```
> myPca <- prcomp(myData.pca, scale=TRUE)
```

4. To check the principal components, take a look at the summary of created objects as follows:

```
> summary(myPca)
```

5. Create a vector of colors for every sample in the data as follows:

```
> colors <-
  c("green", "cyan", "violet", "magenta")
```

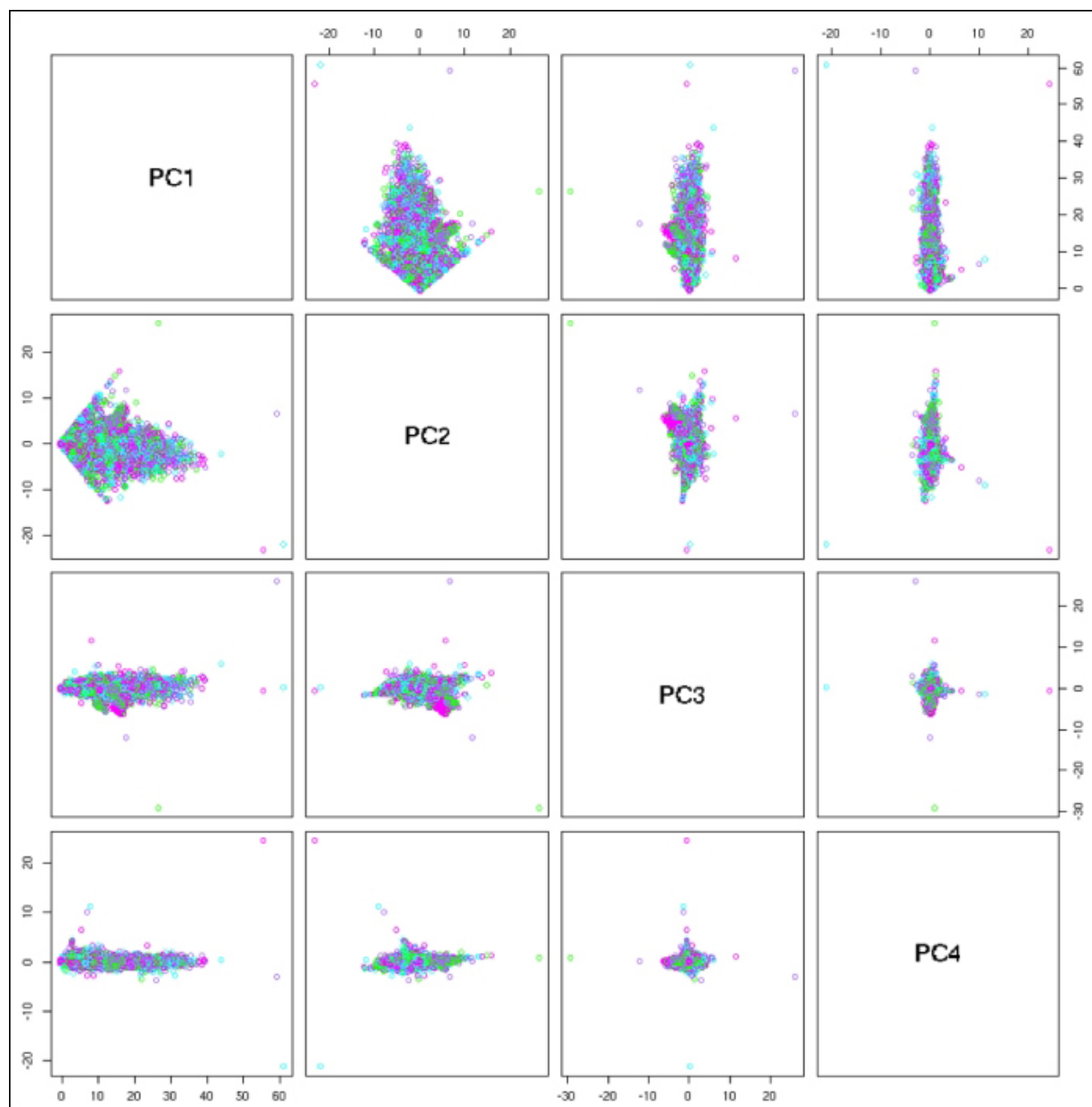
6. To plot the principal components, use the `pairs` function as follows (note that it might take some time depending on the size of the data):

```
> pairs(myPca$x, col=colors)
```

How it works...

The PCA computation via the `prcomp` function performs the principal component analysis on the data matrix. It returns the principal components, their standard deviations (the square roots of Eigen

values), and the rotation (containing the Eigen vectors). The following screenshot shows how the data appears when viewed along the selected pairs of principal components:



Scatter plots for all the pairwise combinations for the first four principal components

See also

- The book called *Principal Component Analysis* by Jolliffe at <http://www.springer.com/statistics/statistical+theory+and+metho>

[ds/book/978-0-387-95442-4](#), which provides detailed information on PCA

- The article called *What is principal component analysis?* by Ringnér at <http://www.nature.com/nbt/journal/v26/n3/full/nbt0308-303.html>, where the application of PCA in a microarray data analysis has been highlighted

Finding the differentially expressed genes

At the genome level, the content of every cell is the same, which means that similar genes are present (with a few exceptions) in similar cells. The question that arises then is, what makes cells (for example, control and treated samples) different from one another? This is the question we have most of the time while doing microarray-based experiments. The concept of differential gene expression is the answer to the question. It is well established that only a fraction of a genome is expressed in each cell, and this phenomenon of selective expression of genes based on cell types is the baseline behind the concept of differential gene expression. Thus, it is important to find which genes show differential expression in a particular cell. This is achieved by comparing the cell under study with a reference, usually called control. This recipe will explain how to find the DE genes for a cell based on the expression levels of the control and treatment cells.

Getting ready

The recipe requires the normalized expression data for treatment and control samples. More number of replicates is always statistically relevant for such analytical purposes. It must be noted that we always use normalized data for any differential expression analysis. As mentioned earlier, normalization makes the array comparable, and hence, using such transformed data to find differences makes the process unbiased and scientifically rational. In this recipe, we will use the `quantile` normalized data. Besides this, we need the experiment and phenotype details, which are part of the `affyBatch` or `ExpressionSet` object. We will also introduce the R library, `limma`, that houses one of the most popular methods in R for differential gene expression analysis. For demonstration, we will use

normal colon cancer preprocessed `affy` data from the `antiProfilesData` package.

How to do it...

To extract differentially expressed genes from the treatment and control expression data, perform the following steps:

1. Install and load the `limma` library into your R session together with the `affy` package as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("limma", "antiProfilesData"))
> library(affy) # Package for affy data
handling
> library(antiProfilesData) # Package
containing input data
> library(affyPLM) # Normalization package for
eSet
> library(limma) # limma analysis package
```

2. Get the data (the colon cancer data) from the `antiProfilesData` package into the R session and subset the first 16 samples that represent the normal and tumor samples (eight each) as we did in previous recipes by typing the following commands:

```
> data(apColonData)
> myData <- apColonData[,
sampleNames(apColonData)[1:16]]
> myData_quantile <-
normalize.ExpressionSet.quantiles(myData)
```

3. Prepare a design matrix based on the experiment details and `phenoData` as follows:

```
> design <- model.matrix(~0 +
pData(myData)$Status)
```

4. Fit a linear model using the expression data and design matrix as follows:

```
> fit <- lmFit(myData_quantile, design)
```

5. Check the `fit` object generated at this step by typing the following command to get more details:

```
> fit
```

6. Once you have a linear model fit, compute the moderated statistics for it using the following `eBayes` function:

```
> fitE <- eBayes(fit)
```

7. The output can be ranked and top-ranking genes extracted from it, as follows:

```
> tested <- topTable(fitE, adjust="fdr",  
sort.by="B", number=Inf)
```

8. To add the conditions of the p-values or other conditions to get the DE genes, extend the previous steps as follows:

```
> DE <- x[tested$adj.P.Val<0.01,]  
> dim[DE]  
[1] 884 6  
> DE <- x[tested$adj.P.Val< 0.01 &  
abs(x$logFC) >2,]  
[1] 403 6
```

How it works...

The `limma` library is used for analyzing gene expression microarray data, especially the use of linear models for analyzing gene expression data. The term `limma` stands for linear models for microarray data. It implements several methods of linear modeling for microarray data that can be used to identify DE genes. The approach described in this recipe first fits a linear model for each gene in the data given a set of arrays. Thereafter, it uses an empirical Bayes method to assess differential expression. This

computes the statistical test and corresponding score in the form of p-values, log fold change, and so on.

The input to the `lmFit` function that we used was the expression data (a matrix) and the design matrix. However, the input can also be `ExpressionSet`. Another important thing that must be mentioned is that we assigned the designed matrix here manually to illustrate the knowledge based on data. Nevertheless, the design matrix can also be directly created from the phenotype data as follows:

```
> design <- model.matrix(~-1 +  
  factor(pData(myData_quantile)$type))
```

The design matrix describes the experiment condition in each of its column. In our case, we have only two conditions, and hence, the single column design matrix with appropriate experiment indication works as well (the type in `design` in your R terminal). To learn more about creating design matrices, type in `?model.matrix` in the R terminal. To use multiple comparisons among different conditions, the `makeContrasts` function can be used (type in `?makeContrasts`). The contrast matrix can be used in the same way as the design matrix.

There's more...

There are several other packages for differential gene expression analysis. One of them is `EMA`, which can be used for many purposes. We will use the package for clustering and heatmap generation. To learn more about the use of the `EMA` package for differential gene analysis, check the corresponding help file as follows:

```
> install.packages("EMA")#requires dependencies  
from Bioconductor(siggenesand gcrma)  
> library(EMA)  
> help(package='EMA')# or visit http://cran.r-project.org/web/packages/EMA/index.html
```

See also

- The article titled *EMA- A R package for Easy Microarray data analysis* by Servant and others at <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2987873/>, which provides more detail on the EMA package
- The article titled *Testing for differentially expressed genes with microarray data* by Tsai and others at <http://nar.oxfordjournals.org/content/31/9/e52.full>, which provides a good deal of information on other kinds of tests
- The *Linear models and empirical bayes methods for assessing differential expression in microarray experiments* article by Smyth at <http://www.ncbi.nlm.nih.gov/pubmed/16646809>, which provides theoretical details on the `limma` package

Working with the data of multiple classes

Until now, we have seen analyses with two experimental groups, namely, treatment and control. However, there are experimental designs where we may need to compare more than two groups as well. To illustrate, let's consider a situation where we have three conditions and we need to compare them systematically against each other. This recipe will explain such a situation.

Getting ready

For this recipe, we will use another dataset from the `leukemiasEset` package. The data is from 60 bone marrow samples of patients with one of the four main types of leukemia (ALL, AML, CLL, and CML) and non-leukemia controls. However, for demonstration purposes, we will use only three samples from each of these categories.

How to do it...

In this recipe, you need to principally focus on creating the design matrix for the comparison, which is the key difference. Show this on imaginary data with three conditions and pairwise comparisons, as follows:

1. First, install and load the required libraries (in this case, `leukemiasEset`) as follows:

```
> biocLite("leukemiasEset")
```

2. Load the dataset that you want to start with from the `leukemiasEset` library as follows:

```
> library(leukemiasEset)
```

```
> data(leukemiasEset)
```

3. Then, define the three conditions with two replicates for each, as follows:

```
> pheno <- pData(leukemiasEset)
```

4. Now, select three samples from each set (use the corresponding indexes here) as follows:

```
> myData <- leukemiasEset[,  
  sampleNames(leukemiasEset)[c(1:3, 13:15,  
  25:27, 49:51)]]
```

5. Create the design matrix based on the condition variables, as explained earlier, using the phenotype data as follows:

```
> design <- model.matrix(~0 +  
  factor(pData(myData)$LeukemiaType))
```

6. Rename the columns of the design matrix as follows:

```
> colnames(design) <-  
  unique(as.character(pData(myData)$LeukemiaType  
  ))
```

7. To see the created design matrix, simply look at the design object by typing the following command:

```
> design # First three columns being different  
types of leukemiaand the fourth one being the  
control non leukemia samples
```

	ALL	AML	CLL	NoL
1	1	0	0	0
2	1	0	0	0
3	1	0	0	0
4	0	1	0	0
5	0	1	0	0
6	0	1	0	0
7	0	0	1	0
8	0	0	1	0

```

9      0      0      1      0
10     0      0      0      1
11     0      0      0      1
12     0      0      0      1

```

8. Now, proceed by fitting a linear model to the data as follows:

```
> fit <- lmFit(myData, design)
```

9. For pairwise comparisons, create a contrast matrix as follows:

```
> contrast.matrix <- makeContrasts(NoL- ALL,
NoL- AML, NoL- CLL, levels = design)
```

10. Fit the linear model using this contrast matrix with the help of following command:

```
> fit2 <- contrasts.fit(fit, contrast.matrix)
```

11. Perform the empirical Bayes analysis of the model by typing the following command:

```
> fit2 <- eBayes(fit2)
```

12. Now, extract the differentially expressed gene for each of the pairwise comparisons using the `coef` argument in the `topTable` function. For the first pairwise comparison, set `coef=1` to compare non-leukemia control with **Acute Lymphoblastic Leukemia (ALL)** leukemia as follows:

```
> tested2 <-
topTable(fit2, adjust="fdr", sort.by="B", number=
Inf, coef=1)
> DE2 <- tested2[tested2$adj.P.Val < 0.01,]
dim(DE2)
[1] 252 6
```

How it works...

The major difference in this recipe compared to the two conditions recipe (*Finding the differentially expressed genes* recipe) is the use of the contrast matrix and the `coef` argument in the `topTable` function. The contrast matrix enables the pairwise comparison for the computation of p-values. The model fitted returns a set of p-values for each comparison. The desired set of DE genes for a specific comparison can then be extracted using the proper `coef` value for comparisons.

Handling time series data

As we discussed different conditions or treatments during experiments, the use of time as a treatment is among the popular methods. A cell sample is given a certain treatment, and its expression can change along the course of time. For illustration, we can consider that during stem cell or embryonic development, the expression of genes at different time points will vary. Handling such time course expression data, though not much different from the standard protocol described earlier, needs small modifications in our recipe.

Getting ready

To start with the recipe, we need time course data. In this recipe, we will use the yeast dataset from the `Mfuzz` package. This, of course, requires the installation of the `Mfuzz` package from Bioconductor. This can be followed if you do not want to install the package. In this recipe, we will use the library installation method.

How to do it...

Perform the following steps to deal with the time series data—note that the data being used in the recipe is from yeast (*Saccharomyces cerevisiae*):

1. Install and load the `Mfuzz` library as follows:

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

2. As the dataset is `ExpressionSet`, which needs some other packages for direct handling (the `affy` package needs the `AffyBatch` object), use the `affyPLM` library as follows:

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

3. Now, load the data using the following `data` function:

```
> data(yeast)
```

4. Check the quality of the data using the density plots, boxplots, and so on, as follows:

```
> plotDensity(yeast)
> boxplot(yeast)
```

5. To normalize the data, use `normalize.ExpressionSet.quantiles` from the `affyPLM` package as follows (here, use `quantile` normalization):

```
> yeast_norm <-
  normalize.ExpressionSet.quantile(yeast)
```

6. Perform the quality assessment for the normalized data again, as shown in the fourth step.

7. To check the attributes of the data, get the component details of the `ExpressionSet` object as follows:

```
> pData(yeast_norm)
```

You can see that the data is 16 samples large, starting from a time point of 0 to 160 with an interval of 10 units.

8. The design matrix can accordingly be created for a time series where there will be controls and time points. For example, if you have two replicates, C, for each control and time points, T1 and T2, use the following commands:

```
> times <- pData(yeast_norm)$time
```



```

> times <- as.factor(times)
> design <- model.matrix(~0
+factor(pData(yeast_norm)$time))
> colnames(design)[1:17] <- c("C", paste("T",
0:16, sep=""))

```

9. Create a contrast matrix using the *0th* point as a reference and all the other time points as the treatment (usually done for samples from the same culture), as follows:

```

> cont <- makeContrasts(C-T1, C-T2, C-T3, C-
T4, C-T5, C-T6, C-T7, C-T8, C-T9, C-T10, C-
T11, C-T12, C-T13, C-T14, C-T15, C-T16,
levels=design)

```

10. Now, use this matrix to fit the linear model, followed by the `eBayes` function to compute statistics, as follows:

```

> fit <- lmFit(yeast_norm, cont)
> fitE <- eBayes(fit)

```

11. Then, filter the top-ranking genes using the `topTable` function as follows:

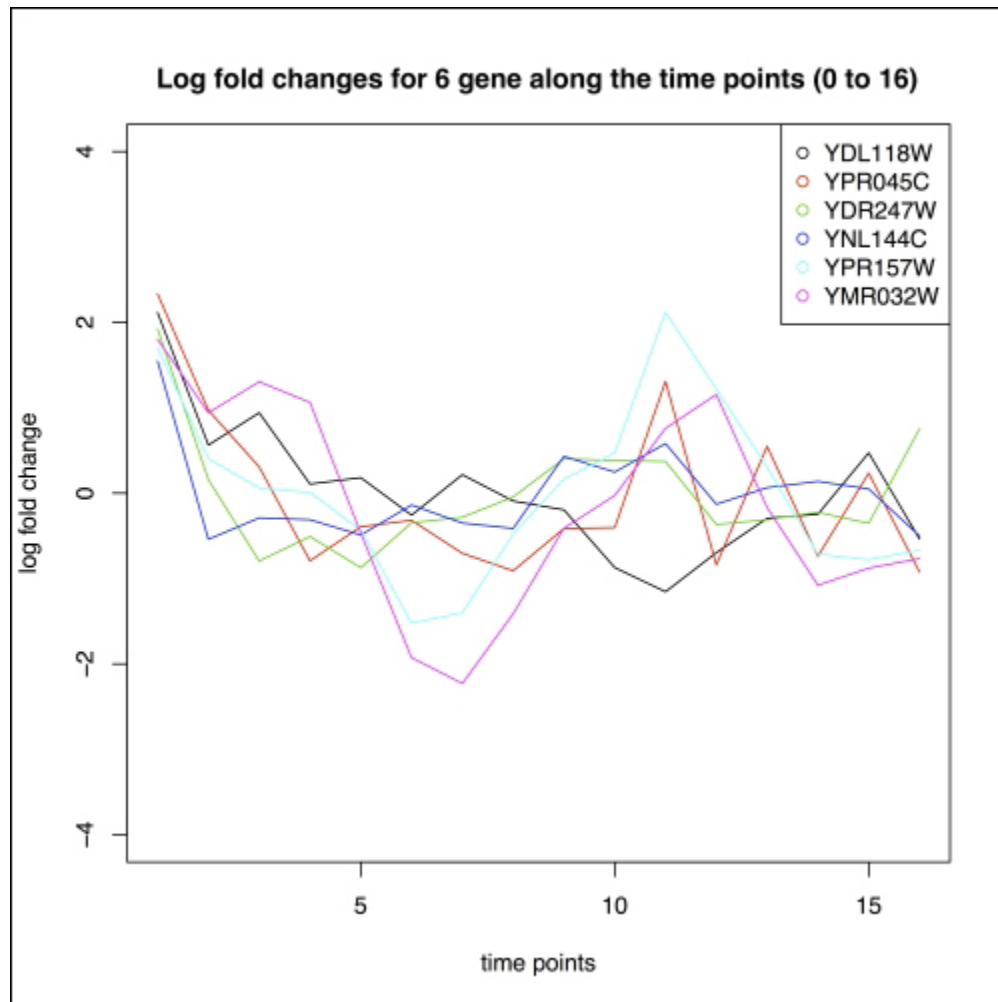
```

> x <- topTable(fitE, adjust="fdr",
sort.by="F", number=100)
> x[x$adj.P.Val< 0.05,]

```

How it works...

The working of the method is very similar to the one in the static data shown in the previous recipe. The only difference here is that we use the time factor while creating the design and contrast matrices. Such data can also be analyzed using the design matrix only in simple conditions and complete data. The following plot shows the log fold change for the yeast data along the time points for first six genes showing oscillations:



There's more...

The `betr` package is another method for filtering out the DE gene from time course data. It also based on the empirical Bayes approach to make such estimation and has one direct function named `betr`. For further information and help, install the `betr` package from Bioconductor and type in `?betr` after loading the `betr` library. It is left to the analyst to use the best possible method for the purpose depending on the data, experiment, and biological question. The detailed explanation of the `betr` package is available on the Bioconductor web page <http://www.bioconductor.org/packages/devel/bioc/vignettes/betr/inst/doc/betr.pdf>.

The `affy` library has a `justRMA` function that can perform the normalization as well. It uses the `quantile` normalization only and can work on the CEL files directly, without the need of `AffyObject`.

Fold changes in microarray data

Fold change refers to the ratio of final value to initial value. In terms of gene expression, it can be defined as the ratio of the final quantification of mRNA to the initial content. The initial and final stages can be the time points or treatment and control conditions. It represents the change rather than an ambiguous absolute quantity. It has been suggested that while extracting DE genes from a dataset, fold changes can serve as more reproducible identifiers. This recipe will explain the use of fold changes for such purposes.

Getting ready

We will continue to use the leukemia dataset for our work, and here, we directly use the results from the *Finding the differentially expressed genes* recipe. In this recipe, we will use the results only from the ALL type of leukemia.

How to do it...

Fold changes are a part of tables that are derived from `limma`. Some other interesting operations on (or about) fold change are explained in this recipe. Perform the following steps:

1. Use the result from the `limma` analysis to get the fold changes. The table generated has a column for the fold change associated with the probes (an example table has been shown here). Refer to the *Working with the data of multiple classes* recipe in this chapter to create the `tested2` object. Take a look at the following object:

```
> head(DE2)
```

The following screenshot represents a table of top-ranked genes after the `limma` analysis:

	logFC	AveExpr	t	P.Value	adj.P.Val	B
ENSG00000152078	4.510507	4.856523	28.13988	4.463747e-11	9.004270e-07	14.01472
ENSG00000117519	-4.185175	4.791585	-22.73888	3.878292e-10	3.911645e-06	12.69738
ENSG00000145850	4.142236	4.507655	17.38636	5.759942e-09	2.925048e-05	10.72782
ENSG00000170180	5.681327	5.734169	17.37423	5.800214e-09	2.925048e-05	10.72231
ENSG00000087586	3.952183	5.720789	16.45393	9.977396e-09	3.111188e-05	10.28705
ENSG00000047597	5.362419	5.108415	16.32474	1.079114e-08	3.111188e-05	10.22315

2. Extract the relevant columns into a separate data frame for the top 10,000 probes as follows:

```
> myTable <- topTable(fit, number=10000)
> logratio <- tested2$logFC
```

3. The fold change to log values and vice versa; the `gtools` library can be used as follows:

```
> library(gtools)
> LR <- foldchange2logratio(foldchange,
base=2)
> FC <- logratio2foldchange(logratio, base=2)
```

4. Now, visualize the log fold change and p-value relations in a volcano plot as follows:

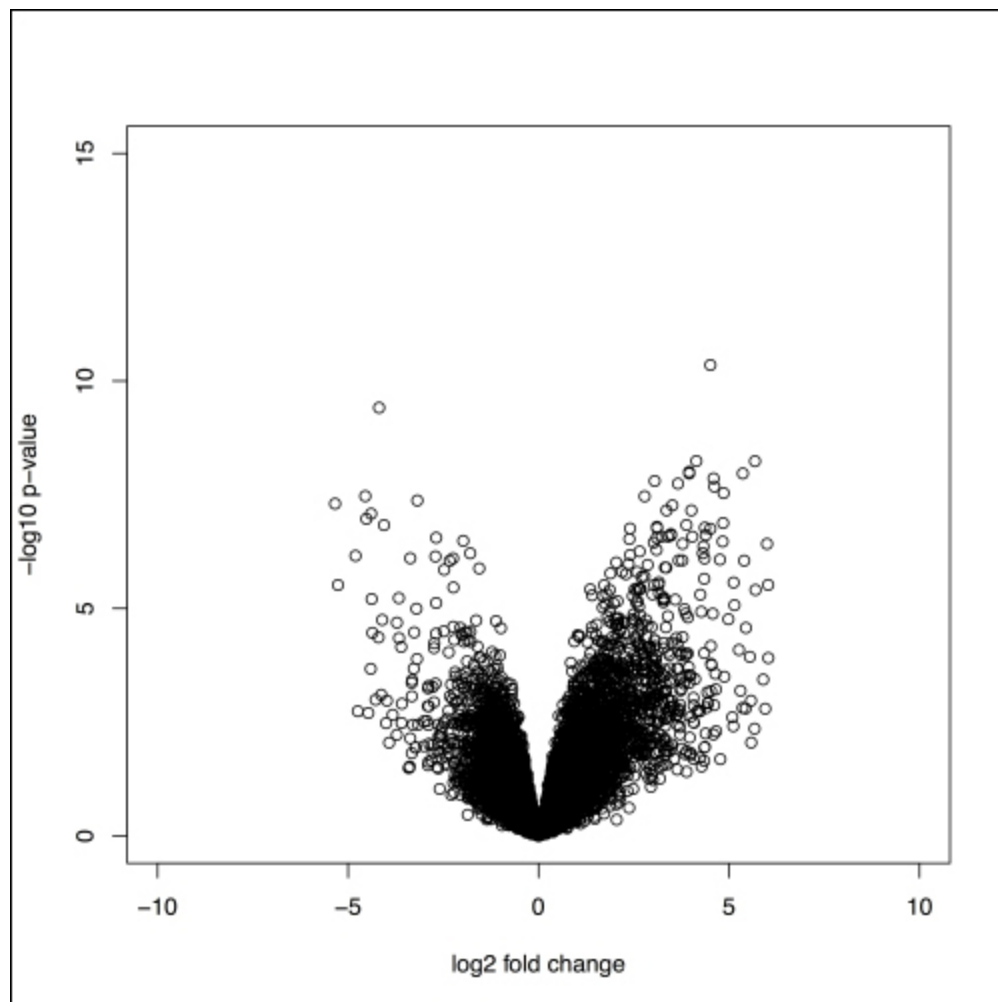
```
> plot(tested2$logFC, -
log10(tested2$P.Value),xlim=c(-10, 10),
ylim=c(0, 15), xlab="log2 fold change",
ylab="-log10 p-value")
```

5. While selecting significant genes from the `limma` generated table, use the log fold change column as another criterion to select DE genes in combination with the p-values as follows:

```
> myTable[tested2$P.Val< 0.05&logFC>1.5,]
```

How it works...

The working of the preceding code is straightforward and self-explanatory. The log-fold changes are computed based on the final (treatment) and initial (control) values. The log used is to the base of 2. The volcano plot simply creates the plot of log fold change to log odds in the data. The plotting is a simple scatter plot with log fold changes along the x axis and $-\log(p\text{-values})$ along the y axis. Transforming the p-values into a log scale gives better resolution for visualization. The following plot shows the log fold change versus the log p-value plot:



The functional enrichment of data

Once we know the DE genes from our array data, we have all the genes that somehow play a role in the cell. In order to know more about this set of genes at a biological level, we need to know their biological role in terms of their function. Analyzing the GO categories in the set can do this. This recipe is about the enrichment of gene sets with GO terms.

Getting ready

The recipe explained here will require the gene set data that comes from our analysis and the annotation package indicated in the `ExpressionSet` object (`hgu95av2.db`).

How to do it...

Perform the following steps to describe the enrichment of genes with GO terms via a hypergeometric test:

1. First, install and load the annotation database and `GOstats` library as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("hgu95av2.db", "GOstats"))
> library(hgu95av2.db)
> library(GOstats)
> library(biomaRt)
```

2. Prepare the input data from the results of the leukemia data analysis (*Working with the data of multiple classes* recipe). Create two sets, one that consists of all the genes in the data and the other that consists of DE genes, as follows:

```
> all_genes <- rownames(tested2)
> sel_genes <- rownames(DE2)
```

3. Map these sets to their Entrez IDs as follows:

```
> mart <- useDataset("hsapiens_gene_ensembl",
useMart("ensembl")) # set the mart
```

```
> all_genes <- c(getBM(filters= "ensembl_gene_id",
attributes= c("entrezgene"), values= all_genes,
mart= mart)) # get entrez ids for all genes
> sel_genes <- c(getBM(filters= "ensembl_gene_id",
attributes= c("entrezgene"), values= sel_genes,
mart= mart)) # get entrez ids for DE
```

4. Now, define a cutoff for the test statistics as follows:

```
> hgCutoff <- 0.05
```

5. The next thing you need is a `GOHyperGParams` object that will be used as an input parameter for the enrichment computations. It can be computed with the following function:

```
> params <- new("GOHyperGParams",
geneIds=sel_genes, universeGeneIds= all_genes,
annotation="hgu95av2.db", ontology="BP",
pvalueCutoff=hgCutoff, conditional=FALSE,
testDirection="over")
```

6. Once you have your `GOHyperGParams` object, perform a hypergeometric test to get the p-value for the GO annotations as follows:

```
> hgOver <- hyperGTest(params)
```

7. Check the summary of the object that was created by typing the following command:

```
> summary(hgOver)
```

8. Get the number of genes associated with the different categories as follows:

```
> geneCounts(hgOver)
> universeCounts(hgOver)
```

9. Also, plot the GO **directed acyclic graph (DAG)** as follows:

```
> plot(goDag(hgOver))
```

10. Finally, generate the report as an HTML file that can be read using any browser, as follows:


```
> htmlReport(hgCondOver, file="ALL_hgco.html")
```

How it works...

The `GOHyperGParams` object is a parameter object. This makes it easier to organize and execute the hypergeometric test on GO annotations for the gene set. The object has slots for the GO category (BP, MF, or CC), genes (Entrez IDs), and GO structure condition and annotation. The `hyperGTest` function implements the hypergeometric test using the set of parameters in the `GOHyperGParams` object. It computes the over or underrepresentation of the GO terms in the gene set. However, the computation ignores the GO structure, treating every annotation as independent. Here comes the use of the structure condition with the specification of the argument conditional set to `TRUE` in the `GOHyperGParams` function. It allows the use of the GO DAG structure to test the leaves of the graph, that is, those terms with no child terms. GO has a hierarchical structure that follows a DAG topology, making higher nodes more abstract than child nodes. The following table is for top-ranked GO terms enrichments. The first column is for the GO IDs; the second is for the p-value that is received from the hypergeometric test; the third is for the odds ratio; and the rest is for the expected count, actual count, size, and actual term, respectively.

We can observe the terms related to blood circulation in the leukemia data:

GOBPID	p-value	OddsRatio	ExpCount	Count	Size	Term
GO:0006779	0.000	47.005	0	8	19	porphyrin-containing compound biosynthetic process
GO:0006778	0.000	30.847	0	9	28	porphyrin-containing

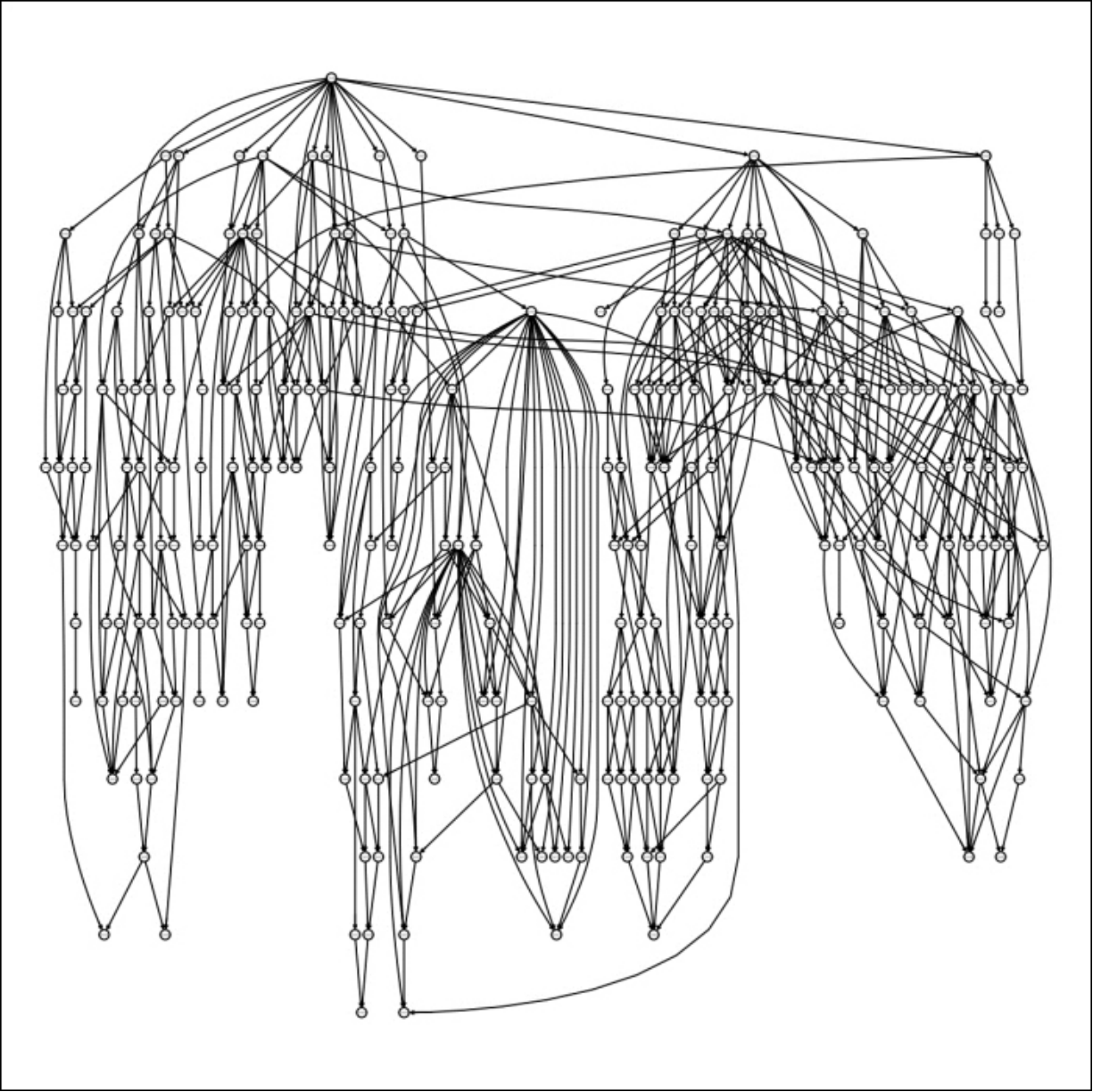
GOBPID	p-value	OddsRatio	ExpCount	Count	Size	Term
						compound metabolic process
GO:0033014	0.000	43.083	0	8	20	tetrapyrrole biosynthetic process
GO:0046501	0.000	95.408	0	6	10	protoporphyrinogen IX metabolic process
GO:0033013	0.000	18.876	1	9	40	tetrapyrrole metabolic process
GO:0042168	0.000	32.030	0	7	21	heme metabolic process
GO:0006782	0.000	105.139	0	5	8	protoporphyrinogen IX biosynthetic process
GO:0006783	0.000	42.375	0	6	15	heme biosynthetic process
GO:0051188	0.000	10.366	1	10	73	cofactor biosynthetic process

GOBPID	p-value	OddsRatio	ExpCount	Count	Size	Term
GO:0051186	0.000	6.226	2	12	139	cofactor metabolic process
GO:0042440	0.000	12.082	1	7	44	pigment metabolic process

There's more...

KEGG is another source of pathways and function information; however, it is not available for open use in its updated form in R. It is a comprehensive database of various pathways (for example, signaling pathways, metabolic pathways, and so on). To view the database, visit the home page at <http://www.genome.jp/kegg/>. We can use `KEGGHyperGParams` instead of `GOHyperGParams` for the KEGG enrichment of genes. While running the function, all `GOHyperGParams` are replaced with `KEGGHyperGParams`, and the condition is set to `FALSE`. The analysis also needs the `KEGG.db` package. For more information about `KEGG.db` (it should be noted that the `KEGG.db` package has not been updated till date), visit <http://www.bioconductor.org/packages/2.13/data/annotation/manuals/KEGG.db/man/KEGG.db.pdf>.

The `MLP` package can also be used for the enrichment of genes with various pathway databases. For details, refer to the Bioconductor home page at <http://bioconductor.org/packages/devel/bioc/manuals/MLP/man/MLP.pdf>. The following plot shows part of the directed acyclic graph of the GO categories in the given data:



Clustering microarray data

Clustering is about aggregating similar genes together in a group (called cluster) and away from other such groups. When genes get clustered together (falling in the same group/cluster), it means they follow a similar pattern based on the expression data under the given conditions. This recipe presents the widely used concept of hierarchical clustering in gene expression analysis.

Getting ready

The clustering recipe presented here will use the normalized breast cancer data from the earlier recipes. However, we will use only part of it—say, the top 1500 genes—for a faster computation.

How to do it...

To perform clustering on gene expression data, carry out the following steps:

1. First, create your dataset for clustering purposes from the leukemia data again. Use only the first 100 data instances for demonstration purposes, as follows:

```
> c.data <- exprs(eset[1:100,])
```

2. Then, do an array clustering. Use the following [EMA](#) package to perform clustering:

```
> install.packages("EMA")  
> library(EMA)
```

3. To perform the clustering of arrays, simply use the `c.data` object with clustering from the [EMA](#) library as follows:

```
> c.array <- clustering(data=c.data,  
metric="pearson", method="ward")
```

4. Create the dendrogram plot for the cluster by plotting the clusters as follows:

```
> plot(c.array)
```

5. To cluster the gene, simply transpose the data matrix and use it as input for the data argument in the clustering function, and define the similarity metric and clustering method as follows:

```
> c.gene <- clustering(data=t(c.data),  
metric="pearsonabs", method="ward")
```

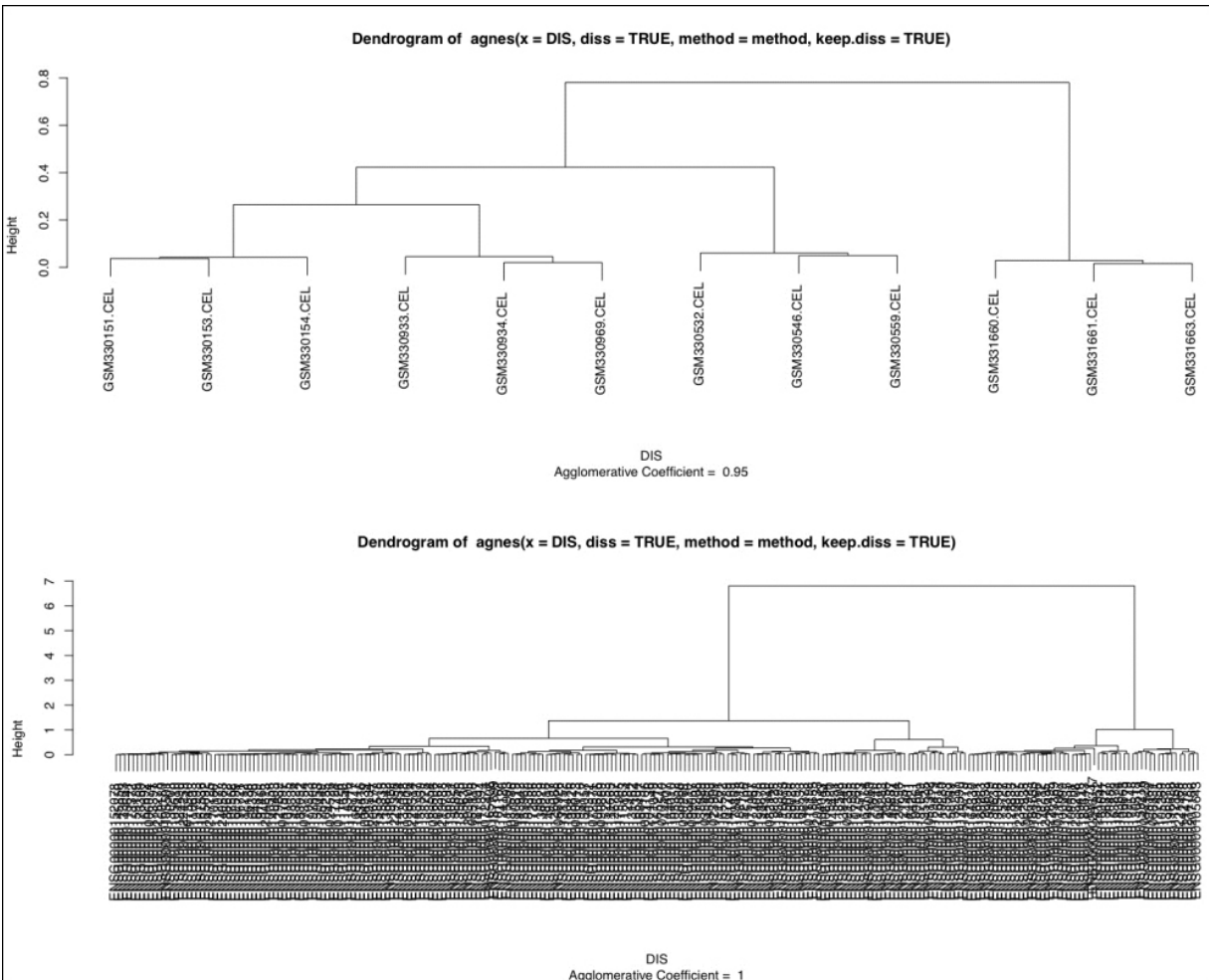
6. Plot the results as follows (note that for readability issues, the following screenshot shows the results for only 100 genes):

```
> plot(c.gene)
```

A more detailed visualization in the form of a heatmap has been presented in the *More visualizations for gene expression data* recipe of the chapter.

How it works...

The clustering method presented in this recipe computes the similarity across the data points (arrays or genes) from the expression data. The similarity measure that can be used are Pearson, Spearman correlation coefficient, Euclidean, Manhattan, and jaccard distances. Based on the similarity scores, a distance matrix is generated that is used as cluster data points. Various methods have been implemented for clustering, which include the average, single, complete, or ward method. We can also use other clustering methods such as kmeans or PAM. The following plot shows the clustering dendrogram for arrays (top) and selected genes (bottom):



There's more...

Clustering will be explained in further detail in [Chapter 9, Machine Learning in Bioinformatics](#). There are many other packages that can be used to perform clustering, and the choice depends on the user. Visit the Wolfram's page (<http://reference.wolfram.com/mathematica/guide/DistanceAndSimilarityMeasures.html>) to know more about different similarities and distance measures.

Getting a co-expression network from microarray data

The generation of networks from gene expression data has shown an upward trend in the recent past. Networks at an abstract level represent the relations between the genes based on the data. There are many possible ways to draw out these relationships from the data. In this recipe, we will explore the relations based on the correlation among the genes.

Getting ready

We will select a small fraction of the dataset to explain this recipe. This is simply to make the process faster and computationally less consuming. Moreover, it is good practice to reduce such networks to the more important genes, thus making the network less noisy.

How to do it...

To generate the co-expression networks using the [WGCNA](#) package, perform the following steps:

1. Install and load the following libraries into the R session:

```
> install.packages(c("WGCNA", "RBGL"))  
> library(WGCNA)  
> library(RBGL)
```

2. Now, take the dataset. It's better to use only significant genes at this stage as it will reduce the noise and simultaneously consume less time. Only 50 genes have been used for demonstration purposes in this recipe, as follows:

```
> myData_Sel <- exprs(eset[rownames(DE2)
```



```
[1:25],1:3])
```

3. The data has sample names in the columns and genes in the rows. Transpose the data as follows:

```
> myData_Sel <- t(myData_Sel)
```

4. To compute the adjacency matrix, either run the correlation computation (it takes a lot of time) or use the `adjacency` function from `WGCNA` as follows:

```
> myMat <- adjacency(myData.Sel,  
  type="signed")
```

The results give a square matrix of size equal to the number of genes. Each entry in the matrix is the connectivity between the genes.

5. The values in the resulting adjacency matrix can be set to either 0 (edge absent) or 1 (edge present) via dichotomization in different ways to get the final adjacency matrix. Simply use a threshold of 0.90 (a high value to get the most correlated vertices), as one of the simplest methods, but not the optimal one. It can be done as follows:

```
> adjMat <- myMat  
> adjMat[abs(adjMat)>0.90] <- 1  
> adjMat[abs(adjMat)<=0.90] <- 0  
> diag(adjMat) <- 0
```

6. The final adjacency matrix can be converted into a `graphNEL` object to be rendered as a graph with nodes and edges, as follows (note that plotting the graph might take some time depending on the network size):

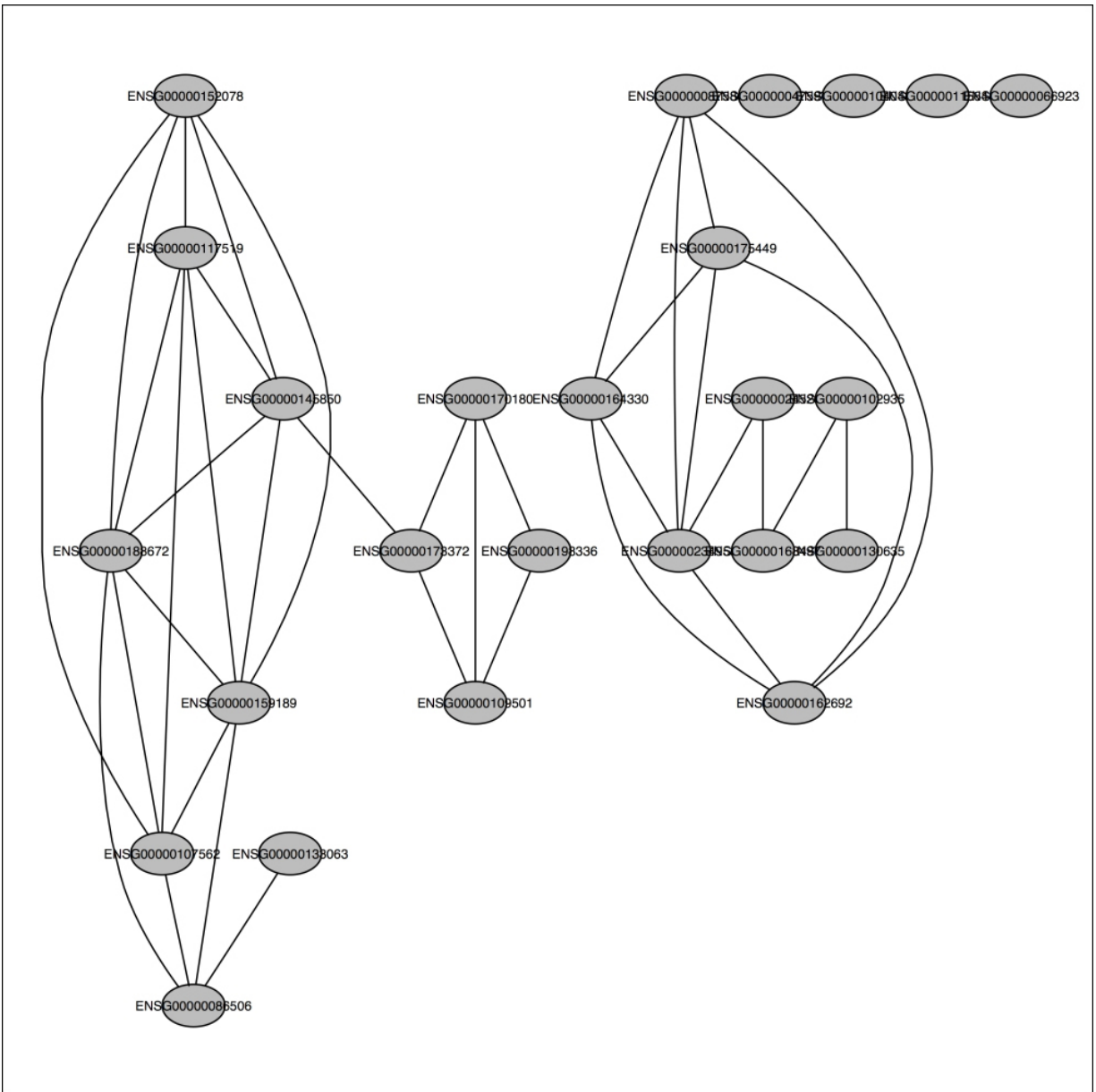
```
> myGraph <- as(adjMat, "graphNEL")  
> myGraph  
> plot(myGraph,
```

```
nodeAttrs=makeNodeAttrs(myGraph, fontsize=28,  
fillcolor="grey")
```

How it works...

The method explained in this recipe is based on the computation of the relationship between the genes in terms of correlation or similarity measures. The function computes the pairwise similarity or correlation among the genes based on the expression data and returns this as a matrix. The threshold set defines only highly correlated or similar genes connected by an edge in the network otherwise no connection between the genes. This adjacency matrix is then used to get the graph object.

The following plot shows the gene co-expression network among the 25 top genes obtained after a DE analysis of the normal cells in leukemia:



There's more...

Besides the [WGCNA](#) package, you can also compute the distances or correlation yourself to compute such networks. The [WGCNA](#) package makes life rather easy. There are several other methods of network inference depending on the experiment and type of data. Many of these methods are supported in R via specific R packages. It will be beyond the scope of current text to deal with these methods. You can refer to the relevant sources for detailed information

See also

- The article titled *Inferring cellular networks – a review* by Markowetz and Spang (<http://www.biomedcentral.com/1471-2105/8/S6/S5/>) presents an overview on some of these methods

More visualizations for gene expression data

This recipe will present some interesting and useful visualizations for expression data. The visualizations or plots explained in this recipe are heatmaps, Venn diagrams, and volcano plots.

Getting ready

For all the plots, we will either use datasets already used in the chapter or create artificial data while explaining the steps.

How to do it...

To generate the different types of plots, perform the following steps:

1. Start with heatmaps. From the *Clustering microarray data* recipe, use the generated clusters, namely, `c.array` and `c.gene`.
2. To do a heatmap, use the following `clustering.plot` function of the `EMA` library and pass the previously mentioned clusters as an argument:

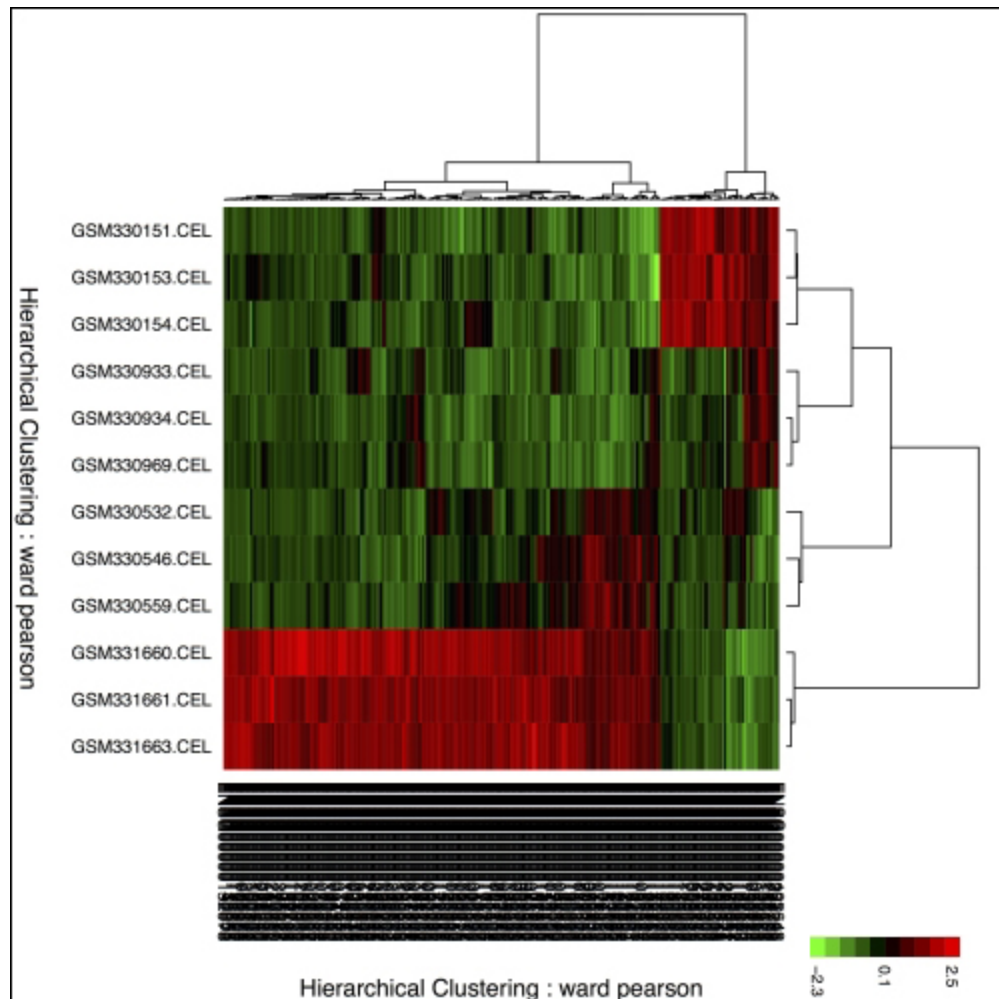
```
> library(EMA)
> clustering.plot(tree=c.array,
  tree.sup=c.gene, data=c.data)
```

This generates the required heatmap.

One can also use the base function `heatmap` to do the same, as follows:

```
> heatmap(c.data)
```

The following figure shows a heatmap for selected genes from the [leukemiaexpression](#) data, which features normal cells and the different types of leukemia:



3. The next plot to create is a Venn diagram. It can be used to show the common and unique contents of two variables. This will require the following [VennDiagram](#) library, which can be installed and loaded as follows:

```
> install.packages("VennDiagram")
> library(VennDiagram)
```

4. Now, create artificial data that consists of five variables (as sets) as a named list object by typing the following commands:

```

> set <- list()
> for(i in 1:5){
  set[[i]]=sample(LETTERS[1:20], replace=TRUE,
prob=rep(0.05,20))
}
names(set)=c(paste("S", 1:5, sep=""))

```

5. To plot the Venn diagram, use the five sets created in the list set and create the `gList` object as follows:

```

> venn.plot <- venn.diagram(x = set, filename
= NULL,cat.cex = 1.5,alpha = 0.50,col =
"black",fill = c("dodgerblue", "goldenrod1",
"darkorange1", "seagreen3", "orchid3"),cex=
c(1.5, 1.5, 1.5, 1.5, 1.5, 1, 0.8, 1, 0.8, 1,
0.8, 1, 0.8, 1, 0.8, 1, 0.55, 1, 0.55, 1,
0.55, 1, 0.55, 1, 0.55, 1, 1, 1, 1, 1, 1.5),
cat.col = c("dodgerblue", "goldenrod1",
"darkorange1", "seagreen3",
"orchid3"),cat.fontface = "bold",margin =
0.05)

```

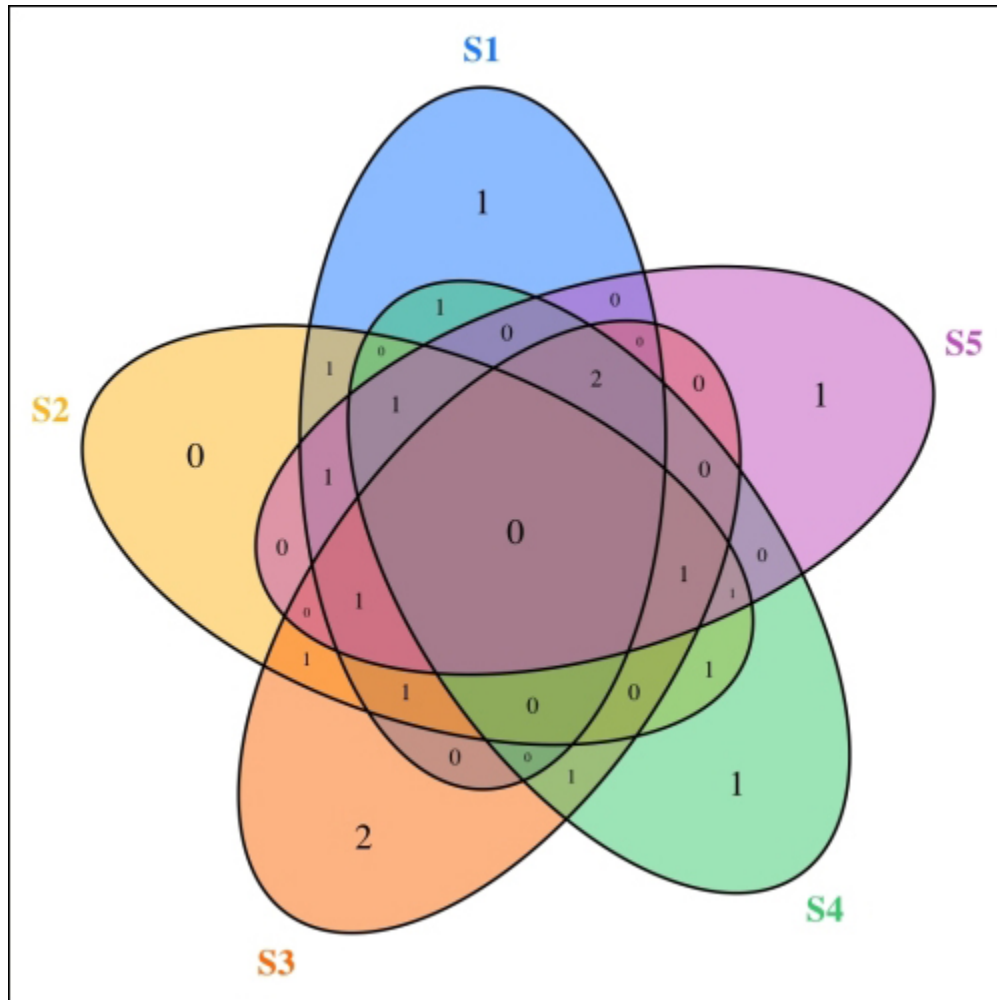
6. Create the plot onto a PDF file using the `grid.draw` function as follows:

```

> pdf("venn.pdf")
> grid.draw(venn.plot)
> dev.off()

```

The following figure shows the Venn diagram for five sets:



7. The volcano plots are used to plot the fold changes and p-values against each other. You have already seen this while working on the *Working with the data of multiple classes* recipe. Here, you will deal with a more intuitive representation of the volcano plot using the following `ggplot2` library:

```
> library(ggplot2)
```

8. Now, select the top-ranked genes from your `limma` analysis as a data frame by typing the following command:

```
> head(tested2)
```

9. Use the threshold the p-values and log fold change to define colors in the plot as follows:


```
> threshold <- as.factor(tested2$logFC>1.5  
&tested2$P.Value< 0.1)
```

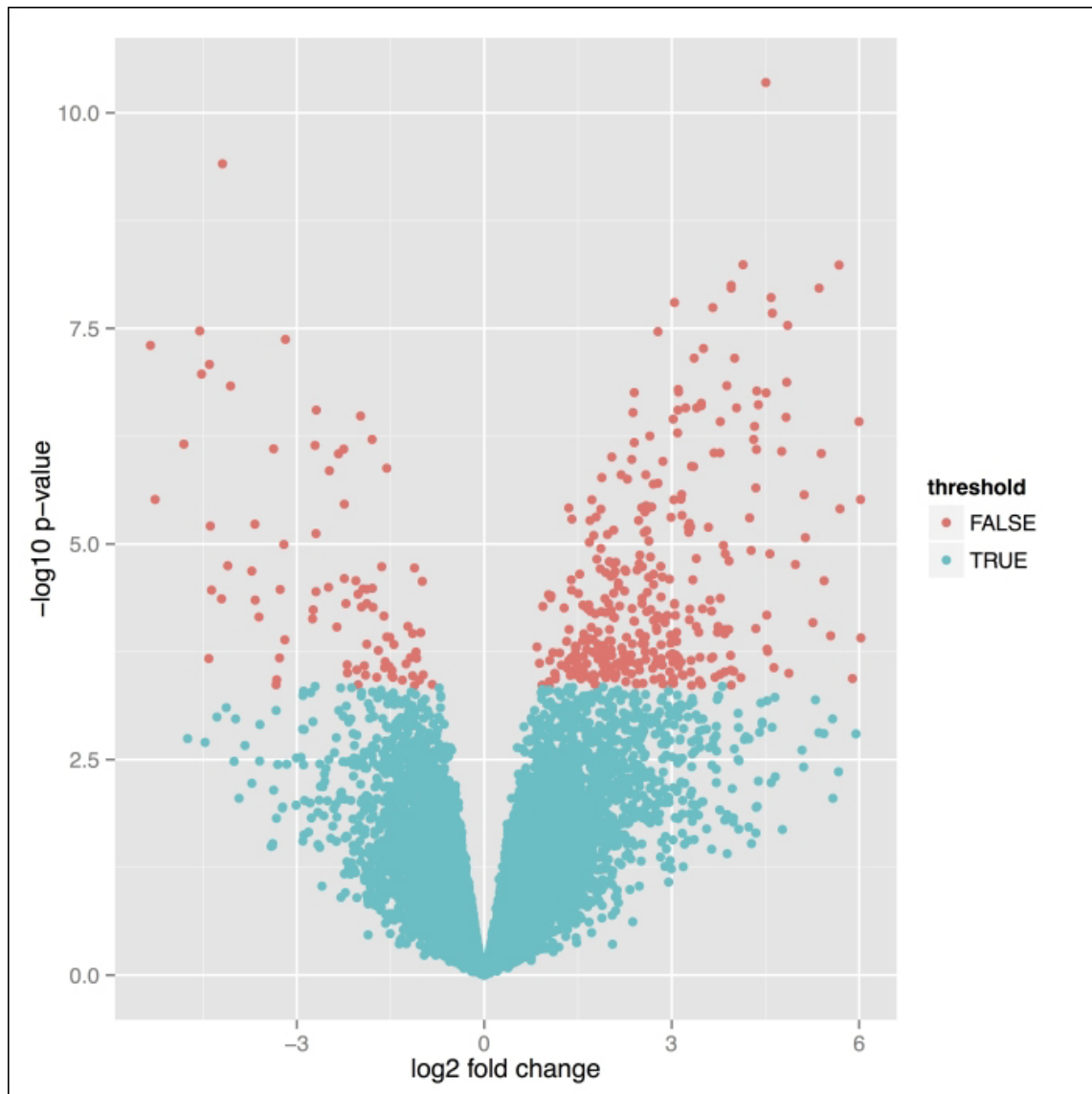
10. Now, use the data and the `threshold` object to create a `g` object as follows:

```
> g <- ggplot(data = tested2, aes (x= logFC, y  
= -log10(P.Value), colour =threshold))  
+geom_point() + opts(legend.position =  
"right") +xlab("log2 fold change") + ylab("-  
log10 p-value")
```

11. Finally, plot the object by simply typing in the object name or save it as an object as follows:

```
> g
```

In the following plot, the volcano plot uses `ggplot` for the expression data:



How it works...

The recipe of the heatmap uses the clustering results from the corresponding recipe. It uses two-way clustering for the arrays and genes. More arguments can be used to add information to the plot, such as labels and titles, and even the color palette can be changed. For more information, you can seek help in R using `?clustering.plot`.

The Venn diagram recipe uses a named list object that contains the sets that the Venn diagram has to be drawn for. The code explained

in this recipe is for five sets. In order to use the function for a different number of sets, parameters such as `colors` and `cex` must be modified. An example with a different number of sets has been explained in the help file for the function. Type in `?venn.diagram` for a detailed explanation. These plots can be used to show different attributes of results from data analysis. One possible scenario is that while comparing multiple treatments with a control, we can visualize how many DE genes/probes are shared across the experiments. Another possible situation can be the sharing of overexpressed GO terms in two or more analyses.

The third plot (volcano plot) is rather simple and uses the `ggplot` function instead of the usual `plot` function. It segregates the genes from other genes in terms of colors, showing the genes with p-values less than 0.05 and the absolute log fold change greater than 1.5.

Chapter 6. Analyzing GWAS Data

In this chapter, we will cover the following recipes:

- The SNP association analysis
- Running association scans for SNPs
- The whole genome SNP association analysis
- Importing PLINK GWAS data
- Data handling with the GWASTools package
- Manipulating other GWAS data formats
- The SNP annotation and enrichment
- Testing data for Hardy-Weinberg equilibrium
- Association tests with CNV data
- Visualizations in GWAS studies

Introduction

Genome-wide association studies (GWAS) is a method of identifying susceptibility loci for complex diseases. It is based on the technique of scanning genomes of many subjects in order to identify the genetic variation possibly responsible for a disease through statistical tests. The technique needs a large number of subjects in order to reliably identify the variation. With the advent of high-density microarray technologies for **Single Nucleotide Polymorphisms (SNP)** detection and the [HapMap](http://hapmap.ncbi.nlm.nih.gov) project together with the **Human Genome Project (HGP)**, this technique has been made possible. More information on the [HapMap](http://hapmap.ncbi.nlm.nih.gov) project is available at <http://hapmap.ncbi.nlm.nih.gov>. A number of SNP studies are available on SNPedia (<http://snpedia.com/index.php/SNPedia>), which shares information about the effects of variations in DNA based on peer-reviewed literature. A popular database for SNPs is [dbSNP](http://www.ncbi.nlm.nih.gov/SNP/), which is available on the Entrez page (<http://www.ncbi.nlm.nih.gov/SNP/>). **Online Mendelian Inheritance**

in Man (OMIM) is another source of information on human genes and genetic phenotypes, which is available at <http://omim.org>.

GWAS uses two groups of subjects: the control (for example, without disease or without the desired trait), and diseased (with desired traits). SNPs are detected from the subjects. They refer to a variation in the DNA sequence where a single nucleotide differs between two phenotypes or chromosomes. To illustrate, in a sequence ATCGTACG, the variant can be ATCGCACG, where T has been changed to C. To find the association of these SNPs with certain phenotypes, the presence of such SNPs is statistically tested with samples from these phenotypes. For example, the SNPs that are significantly more frequent in people who are suffering from the disease, compared to those who aren't, are said to be associated with the disease. We can illustrate this with an example of a disease-control GWA study. For control cases and disease cases, the allele count of SNPs is computed and then a statistical test, such as a Chi-squared test, is performed to identify variants associated with the disease under consideration. This involves the analysis of a huge amount of genotypic data. The GWAS data can be the size of hundreds of MBs up to GBs.

Before we start working with GWAS data, let's see what is there in it. GWAS data in any format is obtained from genome-wide studies. It includes information about the variants as well as their mappings onto chromosomes for every individual. This information is structured in different ways depending on the data format, such as tables or flat files. Some of these data formats will be discussed during the course of this chapter. To get a detailed idea of GWAS, refer to the *Genome wide association studies* article by Bush and Moore at <http://www.ploscompbiol.org/article/info%3Adoi%2F10.1371%2Fjournal.pcbi.1002822>.

This chapter will explain the recipes related to analyzing GWAS data in order to make biological inferences.

The SNP association analysis

GWAS has gained popularity as an approach to identify genetic variants associated with a disease phenotype. This is accomplished through statistical analysis of the GWAS data. These methods of statistical analyses attempt to establish the association between each individual SNP and the observed phenotype. In this recipe, we will perform such an association analysis.

Getting ready

To prepare for a GWAS analysis, we need data that contains all the information explained earlier. In this recipe, we will use the data available in packages. While going through the recipe, we will introduce the corresponding packages.

How to do it...

Perform the following steps to explain the SNP association analysis with diseases or other phenotypes:

1. Start with the installation of the `SNPassoc` package from CRAN by typing the following commands:

```
> install.packages("SNPassoc") # A package for  
association studies  
> library(SNPassoc)
```

2. Use the built-in data for this package, which can be loaded with the `data` function, as follows:

```
> data(SNPs)
```

3. The previous step loads two data objects in the workspace. Take a look at the data objects that were just loaded:

```
> head(SNPs)
> head(SNPs.info.pos)
```

4. Now, create an SNP object using the following `setupSNP` function with the SNP information in columns 6 to 40 (see the *How it works...* section):

```
> mySNP <- setupSNP(SNPs, 6:40, sep="")
```

5. With this data object, carry out an association test for the trait and variables of your interest with the following `association` function:

```
> myres <-
association(casco~sex+snp10001+blood.pre, data
= mySNP, model.interaction =
c("dominant","codominant"))
```

6. Take a look at the results by issuing the following command:

```
> myres
```

How it works...

Before starting with an explanation of the recipe, let's take a look at the SNP data that we are using. The data is a `data.frame` object that consists of 157 rows and 40 columns. Each row of the data represents a sample that is either a case or control. The first five columns represent the attributes for each sample and the columns from `6:40` are the SNP information. That is why we used these columns while creating the SNP class object. The first five columns are for sample identifiers, for case or control identification (1 is for case and 0 is for control, which means `casco`). This is followed by information about the blood pressure and protein level of each gender. The following screenshot shows the first six entries of the SNP data:

	id	casco	sex	blood.pre	protein	snp10001	snp10002	snp10003	snp10004	snp10005	snp10006
1	1	1	Female	13.7	75640.52	TT	CC	GG	GG	GG	AA
2	2	1	Female	12.7	28688.22	TT	AC	GG	GG	AG	AA
3	3	1	Female	12.9	17279.59	TT	CC	GG	GG	GG	AA
4	4	1	Male	14.6	27253.99	CT	CC	GG	GG	GG	AA
5	5	1	Female	13.4	38066.57	TT	AC	GG	GG	GG	AA
6	6	1	Female	11.3	9872.46	TT	CC	GG	GG	GG	AA

The other data that we loaded contains information about the SNP name, chromosome name, and genomic position, as shown in the following screenshot:

	snp	chr	pos
1	snp10001	Chr1	2987398
2	snp10002	Chr1	1913558
3	snp10003	Chr1	1982067
4	snp10004	Chr1	447403
5	snp10005	Chr1	2212031
6	snp10006	Chr1	2515720

The first six entries of the chromosomal location of the SNP data

The SNP object that we created has a `data.frame` structure as well, but looks slightly different in terms of how the SNP information is coded. (To take a look at it, type `head(mySNP)` in the R console.)

Now, let's have a look at our function to identify the relation. The `association` function carries out an association analysis between a single SNP and a dependent variable via model fitting and computing statistics. The arguments include the model to be fitted (formula object). This needs the trait of interest and the SNP under investigation separated by a `~` operator. It is possible to add other variables to the formula using the `+` operator. Adding more SNPs is possible, but the analysis is done only for the first SNP and adjusted by the others. The `association` function can model the dependence based on the codominant, dominant, recessive, overdominant, and

log-additive genetic models of inheritance. This is supplied to the function under the `model.interaction` argument.

In this recipe, we tried to find the dependence of case and control on the first SNP—`snp10001`—the sex, and the blood pressure. We used two genetic models: dominant and codominant. The result looks like a matrix. The columns contain information about the sample size and percentages for each genotype, the odds ratio and its 95 percent confidence interval (which takes the most frequent homozygous genotype as the reference), and the p-value corresponding to the likelihood ratio test obtained from a comparison with the null model. Besides, the matrix also has the **Akaike Information Criterion (AIC)** of each genetic model. Thus, the `myres` table reflects the dependence of phenotypes on SNP (together with other factors). The following screenshot shows the result of the association analysis test for `snp10001`:

SNP: snp10001 adjusted by: sex blood.pre										
	0	%	1	%	OR	lower	upper	p-value	AIC	
Codominant										
T/T	24	51.1	68	61.8	1.00			0.15410	195.8	
C/T	21	44.7	32	29.1	0.55	0.26	1.14			
C/C	2	4.3	10	9.1	1.74	0.35	8.63			
Dominant										
T/T	24	51.1	68	61.8	1.00			0.22859	196.1	
C/T-C/C	23	48.9	42	38.2	0.65	0.32	1.31			
Recessive										
T/T-C/T	45	95.7	100	90.9	1.00			0.28494	196.4	
C/C	2	4.3	10	9.1	2.22	0.46	10.70			
Overdominant										
T/T-C/C	26	55.3	78	70.9	1.00			0.07188	194.3	
C/T	21	44.7	32	29.1	0.52	0.25	1.06			
log-Additive										
0,1,2	47	29.9	110	70.1	0.87	0.51	1.49	0.60861	197.3	

There's more...

There are other packages for performing similar analyses. Some examples are `snp.plotter` and `GenABEL`. We will get to learn more about these in the upcoming recipes of the chapter.

See also

- The *SNPassoc: an R package to perform whole genome association studies* article by Juan R and others at <http://bioinformatics.oxfordjournals.org/content/23/5/654.2.full.pdf>, which provides the theoretical details of the `SNPassoc` package

Running association scans for SNPs

The previous recipe presented the approach to discover the dependence of a phenotype on a single SNP. However, in the case of phenotypes such as complex diseases, one has to keep an eye on all of the SNPs and scan the entire set. This recipe will discuss computing the significance of the dependence of phenotypes on all SNPs available in the data.

Getting ready

In this recipe, we will continue working on the same dataset of SNPs and the same `SNPassoc` library as the previous recipe. We will directly use the SNP object, `mySNP`, created previously.

How to do it...

We start our recipe step from the point where we created the SNP object named `mySNP`. This time, we try to understand another phenotype, say, the protein level. To run association scan for SNPs, perform the following steps:

1. Run the `WGassociation` function on your `mySNP` object for all of the genetic models as follows:

```
> myres <- WGassociation(protein~1, data =  
mySNP, model = "all")
```

The preceding step can also be run avoiding the formula by just providing the name of the phenotype under study since the function runs across all of the SNPs anyway, as follows:

```
> myres <- WGassociation(protein, data =  
mySNP, model = "all")
```

2. Take a look at the result matrix computed by issuing the following command:

```
> myres
```

3. To get the p-values for specific genetic models, use the name of the mode as the function and the complete result matrix as the argument, as follows:

```
> dominant(myres)  
> recessive(myres)
```

4. To get the detailed result for every SNP (similar to the previous recipe), use the following `WGstats` function:

```
> WGstats(myres)
```

5. View the overall summary with the `summary` function as follows:

```
> summary(myres)
```

6. Plot the p-values for all of the models and SNPs as follows:

```
> plot(myres)
```

7. The `WGassociation` function performs a whole genome association analysis by computing just the p-values for each SNP corresponding to the likelihood ratio test as follows:

```
> resHapMap <- WGassociation(protein,  
data=mySNP, model="log")
```

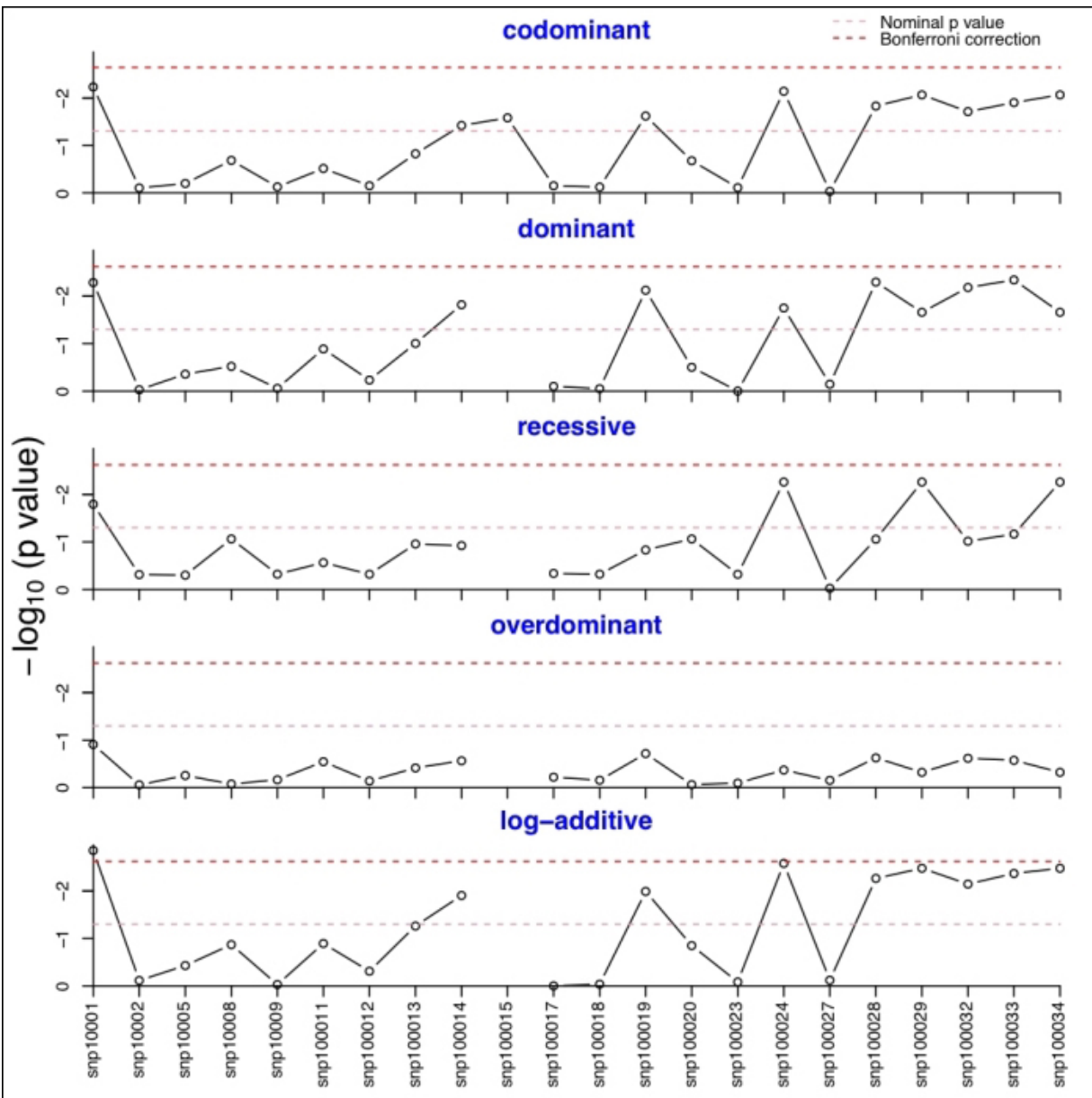
How it works...

We have already seen how the `associate` function works in the previous recipe. The `Wgassociation` function works in a similar fashion. The main difference is that it performs many association tests in parallel using the `mclapply` function. The function takes each SNP as a variable that the phenotype depends on and runs the `association` function on each of them in parallel. These parallel runs return a matrix for each SNP that the corresponding p-values are extracted from, and this ultimately returns as the overall result. The following screenshot shows the result matrix for the association scan of an SNP:

	comments	codominant	dominant	recessive	overdominant	log-additive
snp10001	-	0.00586	0.00516	0.01605	0.12306	0.00140
snp10002	-	0.78555	0.93303	0.48704	0.87288	0.76846
snp10003	Monomorphic	-	-	-	-	-
snp10004	Monomorphic	-	-	-	-	-
snp10005	-	0.63306	0.43881	0.50130	0.55427	0.37267
snp10006	Monomorphic	-	-	-	-	-
snp10007	Monomorphic	-	-	-	-	-
snp10008	-	0.20627	0.30005	0.08652	0.83655	0.13493
snp10009	-	0.74736	0.87204	0.47815	0.68152	0.93616

The plot shows the p-values for every SNP as an absolute log scale along the y axis and SNPs along the x axis. The facets represent the genetic model. The p-values presented here are equipped with the **Bonferroni correction**. While plotting a warning, it tells you how many SNPs are significant for every model of inheritance after the Bonferroni correction. Here, in our results, we have no significant association for the overdominant model. The following screenshot shows the plot for the association of all SNPs in the dataset. The plot shows the log p-values to the base 10 for different models along the y axis, and the SNPs are aligned along the x axis.

In the following screenshot, the horizontal dashed lines (red and pink) represent two different thresholds, namely, the Bonferroni correction and nominal p-value (0.05), respectively:



There's more...

We spoke about parallelization in this recipe using the `mclapply` function from the `multicore` package. This is a parallelized version of `mclapply` (see [Appendix B, Useful R Packages](#)). To know more about parallelization using `multicore` and `mclapply`, refer to the `multicore` package at <http://cran.r-project.org/web/packages/multicore/multicore.pdf>.

Furthermore, we also spoke about the Bonferroni correction. It is an adjustment made to the nominal p-values for multiple testing on the same data (the alpha error per test). It is used to reduce the chances of obtaining false-positive results or type I errors. For more information, you can refer to the explanation in the write-up at <http://www.stat.berkeley.edu/~mgoldman/Section0402.pdf>.

See also

- The *Whole genome association studies in complex diseases: where do we stand?* article by Need and Goldstein at <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3181943/>, which provides a more detailed explanation of associations in complex diseases

The whole genome SNP association analysis

So far, we have worked with SNP data where we had a few SNPs and made attempts to understand the phenotype dependence on these SNPs. However, we are yet to cover the most interesting part of association analysis, which is the whole genome analysis. Here, we attempt to study the association analysis of a trait across the whole genome. This recipe is an attempt at doing such an analysis.

Getting ready

Until now, we have been using a small data set. Now, we switch to a much larger one called the [HapMap](#) data set. The data is available within the [SNPassoc](#) package of R.

How to do it...

Follow the ensuing steps to perform a whole genome association analysis for SNPs:

1. Start with loading the [HapMap](#) data as follows:

```
> data(HapMap)
```

2. Check what the data looks like by looking at the first few entries with the [str](#) function as follows:

```
> str(HapMap)
> str(HapMap$SNPs$pos)
```

3. The next step is to create the SNP class object with the help of the following command:


```
> myHapMap <- setupSNP(HapMap, colSNPs=
3:9307, sort=TRUE,
info=HapMap.SNPs.pos, sep="")
```

- Now, run the following `WGassociation` function on this data object for the genetic model of your interest (in our case, we use `"dominant"`):

```
> myHapMapres <- WGassociation(group, data=
myHapMap, model="dominant")
```

- The result is a data frame. Check the resulting data frame as follows:

```
> head(myHapMapres)
> print(myHapMapres)
```

- Type the following command to plot the entire result in a better way, as shown in the following screenshot:

```
> plot(myHapMapres, whole=TRUE)
```

How it works...

The `HapMap` data used in this recipe consists of two groups of population, namely, CEU (Utah residents with ancestors from Northern and Western Europe) and YRI (Yoruba in Ibadan, Nigeria). Each group is represented by 60 samples (totally 120 samples). The following screenshot shows the head of the `HapMap` data with the sample ID group and nucleotide pairs at the SNP sites:

	id	group	rs10399749	rs11260616	rs4648633	rs6659552	rs7550396	rs12239794
1	NA06985	CEU	CC	AA	TT	GG	GG	GG
2	NA06993	CEU	CC	AT	CT	CG	GG	GG
3	NA06994	CEU	CC	AA	TT	CG	GG	GG
4	NA07000	CEU	CC	AT	TT	GG	GG	<NA>
5	NA07022	CEU	CC	AA	CT	GG	GG	GG
6	NA07034	CEU	CC	AA	TT	CG	GG	GG

The `HapMap.SNPs.pos` file has the genomic mapping information for the SNPs on the human chromosomes, as shown in the following screenshot:

	snp	chromosome	position
1	rs10399749	chr1	45162
2	rs11260616	chr1	1794167
3	rs4648633	chr1	2352864
4	rs6659552	chr1	2902617
5	rs7550396	chr1	3170389
6	rs12239794	chr1	3382844

The chromosomal/genomic position of the HapMap data

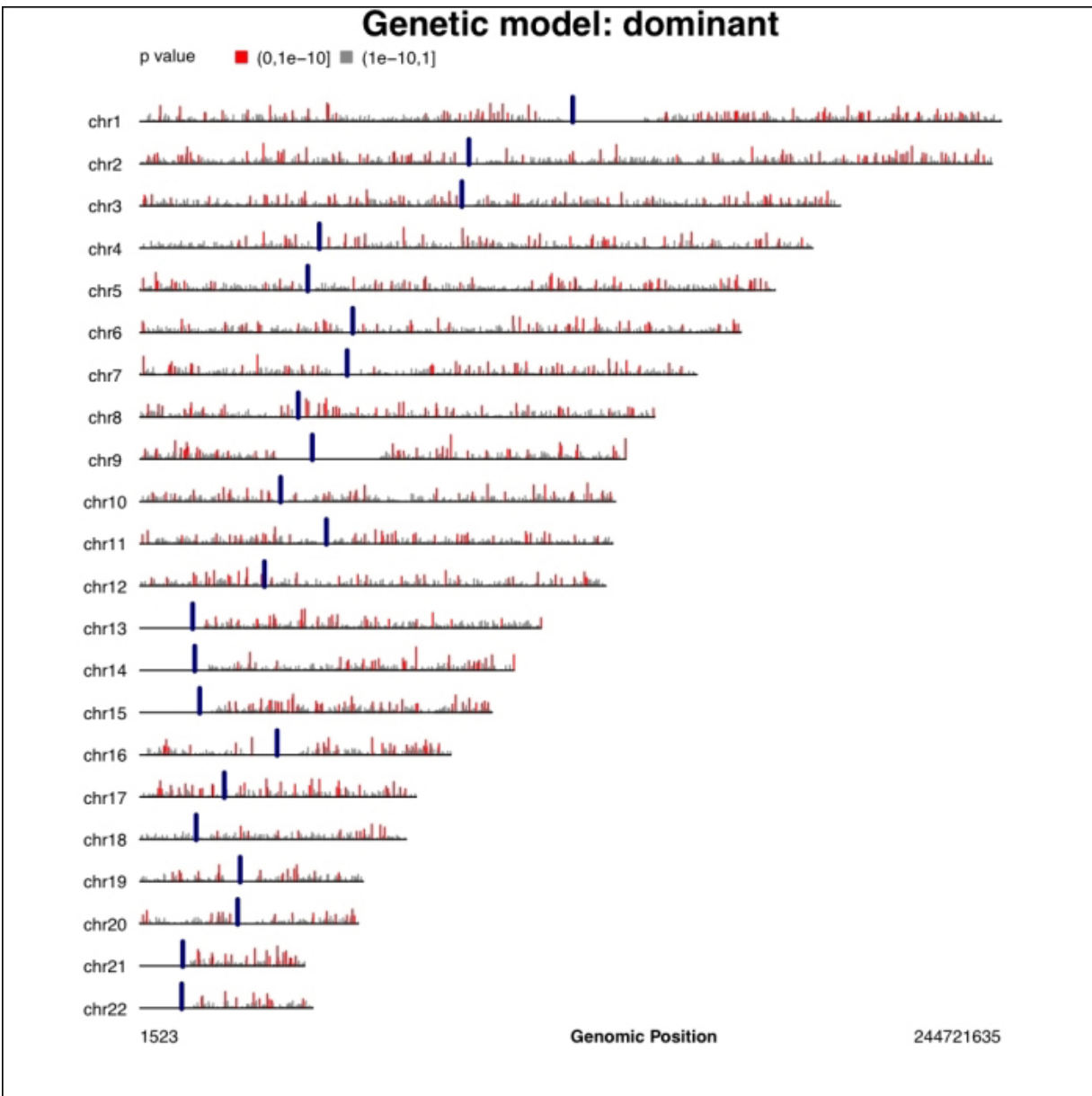
The working principle of this recipe is mostly the same as the previous recipe. The `setupSNP` function converts a `data.frame` object into the `snp` class to be used by the `WGassociation` function. The input required is the `data.frame` object and the column number where SNPs are located in the `data.frame` object. The difference is the plot that was created in this recipe. The created plot shows the p-values of the association of phenotypes mapped onto their corresponding genomic location on the human chromosome.

Before converting the data into an SNP object, it is always wise to look at it manually in order to understand what the available descriptors and variables are. It can also be useful in cases where the phenotype data is not available and one has to create it manually (artificially or from other sources). Analyzing huge data such as [HapMap](#) takes time and memory; therefore, you have to be patient with such an analysis.

Note

A system running low on memory may freeze your R session while running this function.

The following screenshot shows a plot with the associated p-values for the [HapMap](#) data over the specific chromosomal locations. The y axis depicts the $-\log$ p-values for the entire analysis along all chromosomes. Each vertical line corresponds to a position or an SNP on that specific chromosome, whereas the blue lines represent the centromere. The significant SNPs are in red (the others are in grey) as shown in the following screenshot (note that the model assumed is a dominant genetic model):



The `plot` function used in this recipe plots all of the SNPs and their significance along the chromosomal positions. We can set the `whole` argument to `FALSE` to get it together along one horizontal line.

There's more...

The `HapMap` data, as discussed earlier, is a huge collection of genetic variants from different populations. CEU and YRI are two of these populations that we used here. A complete detail of other descriptors is available at <http://hapmap.ncbi.nlm.nih.gov/citinghapmap.html>.

Importing PLINK GWAS data

The previous recipes used one of the simplest forms that the GWAS data is handled in, that is, `data.frame`. However, it is not the only format for the GWAS data. As discussed earlier, the GWAS data normally has two types of information: the genotype data that has information about the SNP variants and the mapping information containing the SNP names and related information. One of the popular data formats is the PLINK format. The PLINK format of the GWAS data consists of two separate files, one containing the SNP information and the other containing the mapping information. For dependence analysis, it can be combined with the phenotype data separately. Working with such data is not as simple as importing an ordinary text, CSV file, or data frame. This recipe discusses working with the PLINK data by specially importing it in R.

Getting ready

The first two things we need for this recipe are the `.map` and `.ped` files that have to be read into R. In this recipe, we can use both types of data downloaded from the NCBI HapMap page at <http://hapmap.ncbi.nlm.nih.gov/downloads/> (take a look at the hyperlink for the Genotypes PLINK file).

Note that we need to download the `*.map` and `*.ped` files from this source. Make sure that they are corresponding files, that is, they have similar names with different extensions. However, these files can be huge in size, and processing them might take a while to download and work on (and might even be memory intensive). Therefore, for our demo, we already have these files available at the book's web page, together with the code file, and they are named `myPed.ped`, `myMap.map`, and `myPheno.ph`.

The packages that will be used in the recipe are explained in the next section.

How to do it...

In order to work with a real PLINK format in the [HapMap](#) file, perform the following steps:

1. Unzip the file downloaded from the HapMap page to a desired directory.
2. Set the working directory of your R session to this directory.
3. Install and load the [GenABEL](#) library as follows:

```
> install.packages("GenABEL")  
> library(GenABEL)
```

4. Now, convert the [ped](#) and [map](#) files into more usable genotype data using the following function:

```
> convert.snp.ped(pedfile=" myPed.ped",  
  mapfile="myMap.map", out="myOut.out")
```

5. We need a phenotype file for our data, which is a tab separated file, consisting of at least the sample names and the trait of interest as columns. The data will come from the corresponding studies. Here, we have the prepared data on the book's web page. The data required to be loaded into R is now ready in the form of the phenotype data and the genotype data:

```
> myData <-  
  load.gwaa.data(phenofile="myPheno.ph",  
  genofile="myOut.out")
```

6. Take a look at the structure of the [myData](#) object as follows:

```
> str(myData)
```

7. Access the genotype and phenotype data with the help of the following functions:

```
> phdata()  
> gtdata()
```

How it works...

The `HapMap` data consists of individual genotype data and phenotypes. The data downloaded from the `HapMap` project consists of two different types of information. The first one is the `.ped` file, which contains the genotype information, and the second one is the `.map` file, which consists of the SNP names and their mappings. We can see in the unzipped data that it has different files. For example, the data we are working on is the population panel from the `HapMap` project on Utah residents with Northern and Western European ancestry (CEU). The data has 90 individuals and corresponding SNPs on the eighteenth chromosome. The phenotype data contains sex and phenotype information of every individual in the data. While reading these files, the function gets the genotype and marker information from the data. This can be seen in the R console while running the function.

The final data object, `myData`, that was created has slots for the SNP data, mappings, and phenotype data. Use the following `str` function to look at the details of the `myData` object:

```
> str(myData)
```

In case we have data on some other phenotype of the patients, we can create our own `phenol` file. This is actually a table (in the form of a tab-separated file) that consists of sample names and traits of interest and is usually accompanied by sex information in the form of columns. It must be noted that all the IDs used in the `.ped` file should match those in the `pheno` file; otherwise, it would result in a fatal error, and no data will be loaded. Let's take a look at the following example:

```
> write.csv(data.frame(id=samples, sex=sex,  
  traitOfInterest=trait), file="myPheno.ph")
```

There's more...

PLINK is more or less considered a de facto standard for genotypic data. However, there are several other formats, such as simple CSV files, text files, and affymetric flat files.

See also

- http://www.gwaspi.org/?page_id=145 for a detailed description on the PLINK format

Data handling with the GWASTools package

We have dealt with two different data formats. In this recipe, we will introduce an R package that not only enables us to work with raw data but also helps in quality checks and many analytical steps. In this recipe, we will mainly talk about handling raw data and quality checks with `GWASTools`.

Getting ready

For this recipe, we will use the `GWASTools` package and the included data sets for demonstration purposes. Besides this, we will also use the data package named `GWASdata`.

How to do it...

Perform the following steps to walk through the use of the `GWASTools` package to work with GWAS data:

1. Start with installing the package from the Bioconductor repository by typing the following commands:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("GWASTools")
> biocLite(c("GWASdata", "gdsfmt"))
> library(GWASTools)
> library(GWASdata)
```

2. Now, load the `affy` raw data from the `GWASdata` package. It consists of two data sets, `affy_scan_annot` and `affy_snp_annot`, as follows:

```
> data(affy_scan_annot)
```

```
> data(affy_snp_annot)
```

3. Use the built-in data from the package, and to get the complete path and name of the data from the package, type the following commands:

```
> file <- system.file("extdata",  
  "affy_genotype.nc", package="GWASdata")  
> file
```

4. Now, use this file path and name as the argument of the `NcdfGenotypeReader` function to import or load the file in your R workspace as follows:

```
> nc <- NcdfGenotypeReader(file)
```

5. Check the number of scans and SNPs in the data with the following `nscan` and `nsnp` functions:

```
> nscan(nc)  
[1] 47  
> nsnp(nc)  
[1] 3300
```

6. Now, to extract the genotype components from the data, run the `getGenotype` function for the desired scans and SNPs (in our case, 5 and 20) as follows:

```
> geno <- getGenotype(nc, snp=c(1,20),  
  scan=c(1,5))
```

7. Now, get the scan and SNP data in the workspace as follows:

```
> data(affyScanADF) # ScanAnnotationDataFrame  
> data(affySnpADF) # SnpAnnotationDataFrame
```

8. To retrieve the phenotype data, type in the following command:

```
> varMetadata(affySnpADF)
```

9. To create a data object with scan and SNP attributes, use the following function (note that at this level, you can choose not to have the annotation component for the data as it is not obligatory to run the function):

```
> MyGenoData <- GenotypeData(nc, snpAnnot=
affySnpADF, scanAnnot= affyScanADF)
```

10. The data is ready. This has created an object of the `GenotypeData` class. Take a look at the structure of the created object with the following function:

```
> str(MyGenoData)
```

11. Compute the **missing call rate (MCR)** for all of the chromosomes as follows:

```
> mcr_chr <-
missingGenotypeByScanChrom(MyGenoData)
> head(mcr_chr$missing.counts)
      21 22 X XY   Y M
3      2  4 2  0 100 0
5      0  4 2  0   0 0
14     2  6 2  1 100 0
15     4  3 6  1 100 0
17     2  4 1  0   0 1
28     1  1 0  0   0 0
```

12. To compute the MCR sex-wise in the genotyping as a measure of quality, use the following function:

```
> mcr_sex <-
missingGenotypeBySnpSex(MyGenoData)
```

How it works...

The `NcdfGenotypeReader` function reads the genotype data stored in `NetCDF` files. A `NetCDF` file stores the genotype and intensity data and has dimensions of (SNP x Sample size). It uses the `netcdf` package

for this purpose. Once the data is read from the [NetCDF](#) file, it can be used to create a complete data object that we can work with. The principle involved here is the same as an E-set that we learned during the gene expression analysis in the previous chapter.

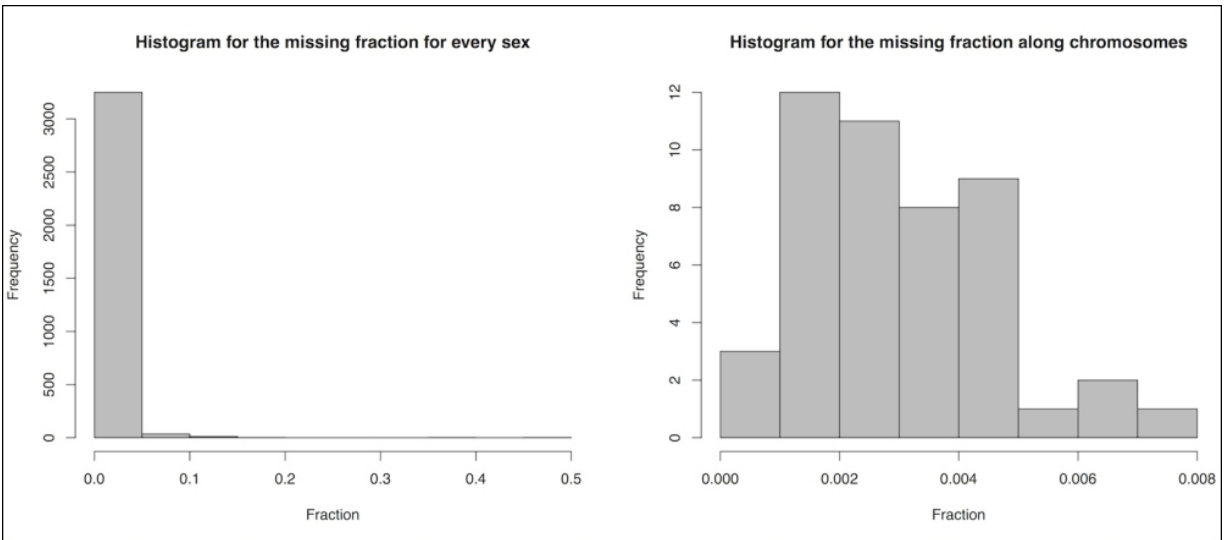
Missing genotype calls can bias the entire data towards one genotype or another. An MCR is defined as the fraction of missing calls per SNP over multiple samples (or per sample). It is one of the common quality measures for genotype batches. The [missingGenotypeBySnpSex](#) function tabulates the missing SNP counts, allele counts, and heterozygous counts for each sex. Note that the function excludes females when calculating the missing rate for Y chromosome SNPs. With these results, we can fetch the total missing counts, number of scans, and the missing fraction. Let's do a histogram for the fractions using the following functions:

```
> par(mfrow=c(1,2))
> hist(mcr_sex$missing.fraction, xlab="Fraction",
main="Histogram for the missing fraction for every
sex")
> hist(mcr_chr$missing.fraction, xlab="Fraction",
main="Histogram for the missing fraction along
chromosomes")
```

To compute the number of SNPs with MCR greater than 0.05, we can use the following command:

```
> sum(mcr_sex$missing.fraction>0.05)
[1] 50
```

The following screenshot shows the histogram of missing call fractions for each sex (left) and along chromosomes (right) (note that we have the missing call fraction along the x axis, and the y axis depicts the frequency of the occurrence in the data):



See also

- The CRAN page at <http://www.bioconductor.org/packages/2.13/data/experiment/manuals/GWASdata/man/GWASdata.pdf>, which provides a more detailed description about the functionalities in the `GWASTools` package
- The *GWASTools: an R/Bioconductor package for quality control and analysis of genome-wide association studies* article by Gogarten and others at <http://bioinformatics.oxfordjournals.org/content/28/24/3329.short>, which provides information regarding the practical backgrounds of the `GWASTools` package
- <http://www.unidata.ucar.edu/software/netcdf/examples/files.html>, which provides detailed information about the **network common data form (NetCDF)** format—a widely accepted format in the scientific community
- The *Quality Control and Quality Assurance in Genotypic Data for Genome-Wide Association Studies* article by Laurie and others at https://www.genevastudy.org/sites/www/content/files/GENEVA_QC_QA.pdf, which provides further detail on MCRs in genotyping

Manipulating other GWAS data formats

Until now, we have worked with structured data frames and PLINK files (`.ped` and `.map`). However, there are different formats for data, and using different libraries requires different data formats for analysis. This recipe will focus on reading some more data formats and their interconversion to other, usable formats.

Getting ready

As mentioned, we will discuss some more data formats during this recipe. Therefore, we need to have data in different formats. The data comes from experiments on different platforms. In the following demo steps, we provide only the reference names; you can change the names with your own files. Note that without having these, it is not possible to execute the code. However, the `pedin.18` file is available at <http://www.filewatcher.com/m/pedin.18.98100-0.html> and the `extra.ped` file at <http://pngu.mgh.harvard.edu/~purcell/plink/dist/example.zip>. Both these files are available with code files.

How to do it...

Perform the following steps to walk through all the data format handling:

1. Let's start with a few functions to convert files from different types of genotypic data to raw data. The `convert.snp.<desired format>` function can be used to convert formats into raw files, as you saw in the previous recipe. Type in the following command to convert the affy file into the raw data format:

```
> dir <- "<path/to/desired/directory>"
> library(GenABEL)
> convert.snp.affymetrix(dir, map, outfile,
skipaffym)
```

2. Similar to the preceding step, use the following functions to convert illumina, mach, and integer genotypic data into the raw data format:

```
>
convert.snp.illumina(infile="pedin.18",out="my
raw.raw ",strand="+")
>
convert.snp.ped(ped="myPed.ped",mapfile="map.1
8",out="myRaw.raw")
> convert.snp.text("genos.dat", " myraw.raw ")
```

3. To read a `.ped` file as a data frame, use the generic function of `read.csv` or the `read.pedfile` function from the `trio` library. The `extra.ped` file

```
> biocLite("trio")# please check the
compatibility with your R version or install
from source (see Chapter 1, Starting
Bioinformatics with R, for details)
> library(trio)
> mySNP <-
read.pedfile(file="path/to/codefiles/myPed.ped
")
```

4. Take a look at the data frame that was created with the following generic function:

```
> head(mySNP)
```

5. Take a look at column names for the object to check the difference between this data frame and the one we used in the `SNPassoc` package by typing the following command:

```
> colnames(mySNP)
```

6. The file that was read has to be converted into a data frame object usable by the function in the `SNPassoc` package. To do this, extract some features from your original matrix as follows:

```
> allsnps <- unlist(strsplit(colnames(mySNP)
[7:ncol(mySNP)], "[.]"))[seq(1,68,2)]
> onlysnp <- mySNP[, -c(1:6)]
> odds <- seq(1,ncol(onlysnp),2)
> allsnps <- allsnps[odds]
> evens <- odds+1
```

7. When you have all the dimensions and specifications of the original `.ped` data, use the following commands to create an empty data frame that matches the required dimensions:

```
> x <- matrix(0, nrow(onlysnp),
length(allsnps))
> x <- data.frame(x)
> colnames(x) <- allsnps
```

8. Now, use the following commands to fill in the data frame with SNP alleles:

```
> for(i in 1:length(odds)){
p=as.factor(paste(onlysnp[,odds[i]],
onlysnp[,evens[i]], sep=""))
x[,i]=p
}
```

9. Finally, merge the computed data frame with the other ID and variable information from the original `.ped` file as follows:

```
> mySNP=data.frame(mySNP[, c(1:6)],x)
```

10. Check your reformatted data with the `head` function as follows:

```
> head(mySNP)
```


This data frame can be used for further processing using the [SNPassoc](#) package that we explored in the previous recipes.

How it works...

The input files demonstrated in this recipe are not available within the package or provided with the code file as they are specific to the experiment platforms (affymetrics, illumina, and so on).

Nevertheless, some files may be fetched from the sources indicated in the *Getting ready* section. The filenames provided in this recipe are just for reference and are not real files. Change the values of the corresponding arguments to the names of your own input files.

The first few conversion functions shown in this recipe read the genotypic information from the file that has been assigned and convert the information for the [GenABEL](#) package into the internal raw format. The format consists of different components. The components are structured as [data.frames](#).

The second part of the recipe converts the [.ped](#) file from PLINK into [data.frame](#), as stated earlier. A [.ped](#) file can be read in many ways, even with the generic [read.csv](#) function. The presented function is one of the many in-built functions in R that handle the [.ped](#) files. The [.ped](#) file in this recipe is read as a [data.frame](#) object. The following screenshot shows the head of the [.ped](#) file data frame before conversion:

	famid	pid	fatid	motid	sex	affected	SNP1.1	SNP1.2	SNP2.1	SNP2.2	SNP3.1	SNP3.2	SNP4.1
1	CH18526	NA18526	0	0	2	1	G	G	C	C	T	T	A
2	CH18524	NA18524	0	0	1	1	G	G	C	C	T	T	A
3	CH18529	NA18529	0	0	2	1	C	G	C	C	T	T	C
4	CH18558	NA18558	0	0	1	1	G	G	C	C	G	T	A
5	CH18532	NA18532	0	0	2	1	G	G	C	C	T	T	A
6	CH18561	NA18561	0	0	1	1	G	G	G	C	G	T	C
7	CH18562	NA18562	0	0	1	1	G	G	C	C	T	T	A
8	CH18537	NA18537	0	0	2	2	G	G	G	C	G	T	C
9	CH18603	NA18603	0	0	1	2	G	G	C	C	T	T	A
10	CH18540	NA18540	0	0	2	1	G	G	C	C	T	T	A

To render this data frame usable in the [SNPassoc](#) package, we perform a few manipulations. The manipulations simply merge

alleles in two consecutive columns as the genotypes for every SNP. Finally, all other information must be merged into one data frame for completion. The following screenshot shows the head of the converted `.ped` file:

	famid	pid	fatid	motid	sex	affected	SNP1	SNP2	SNP3	SNP4	SNP5	SNP6	SNP7	SNP8	SNP9	SNP10
1	CH18526	NA18526	0	0	2	1	GG	CC	TT	AA	GG	GG	TA	GG	TG	CC
2	CH18524	NA18524	0	0	1	1	GG	CC	TT	AA	GG	AG	AA	GA	GG	CC
3	CH18529	NA18529	0	0	2	1	CG	CC	TT	CA	GG	GG	TA	GG	TG	CC
4	CH18558	NA18558	0	0	1	1	GG	CC	GT	AA	GG	GG	AA	GA	TG	CC
5	CH18532	NA18532	0	0	2	1	GG	CC	TT	AA	GG	GG	AA	GA	GG	CC
6	CH18561	NA18561	0	0	1	1	GG	GC	GT	CA	GG	AG	TA	GG	TG	CC
7	CH18562	NA18562	0	0	1	1	GG	CC	TT	AA	GG	GG	AA	GA	GG	CC
8	CH18537	NA18537	0	0	2	2	GG	GC	GT	CA	GG	GG	AA	AA	GG	CC
9	CH18603	NA18603	0	0	1	2	GG	CC	TT	AA	GG	GG	TA	GA	TG	CC
10	CH18540	NA18540	0	0	2	1	GG	CC	TT	AA	GG	GG	TA	GG	TG	CC

Once this data frame is ready, we can run the `setupSNP` function and prepare data for the association tests.

Note

All of these conversion functions do not return any value, but write the output to a file in the current directory. However, the chunk of code provided to convert the `.ped` file into a data frame can do both.

See also

- The corresponding GenABEL pages at <http://www.genabel.org/GenABEL/convert.snp.affymetrix.html> and <http://www.genabel.org/GenABEL/convert.snp.illumina.html> to learn more about different data formats such as affymetrix and illumina

The SNP annotation and enrichment

Once we have some SNPs that are potentially interesting from the point of view of association with the phenotype, it is extremely important and interesting to know about the gene or genomic region that they belong to. Finding these regions and associated functional enrichment can be a major bottleneck. This recipe will present methods that annotate SNPs with genes, genomic regions, and so on.

Getting ready

To perform the annotations, we simply use some random SNPs; however, the functions can be used for any vector or list of SNPs. The package we use to achieve this is [NCBI2R](#).

How to do it...

Perform the following steps to show the annotation of SNPs with genes or genomic regions:

1. Start with installing the [NCBI2R](#) library and loading it in your R session as follows:

```
> install.packages("NCBI2R")  
> library(NCBI2R)
```

2. Now, create a list of some random SNP IDs (it must be a valid [dbSNP](#) ID) with the help of the following command:

```
> mysnp1ist <-  
c("rs94", "rs334", "rs309", "rs333", "rs339")
```

3. Extract the SNPs from one of the blocks of data that you are working with as follows:

```
> mySNPs <- SNPs$id
```

4. To get the flank sequence for the set of SNPs, use the `GetSNPF flankSeq` function from the `NCBI2R` package as follows:

```
> myInfo <- GetSNPF flankSeq(mysnplist)
```

5. To fetch information about the SNPs, use the `GetSNPInfo` function as follows:

```
> myInfo <- GetSNPInfo(mysnplist)
```

6. To know about the pathways where the SNP related genes are involved, type in the following command:

```
> pathway <- GetPathways(myInfo$locusID)
```

7. To divide the SNP information for multiple genes into individual entries, use the package that implements the `SplitGenes` function as follows:

```
> mySNPs <- SplitGenes(myInfo)
```

8. To get the chromosome and genomic position, use the following function:

```
> GetSNPPosHapMap(mysnplist[1])
```

9. To perform the annotation for a set of genes in a file, run the `AnnotateSNPFile` function. The results will be generated and saved as an HTML file, as follows:

```
> write(mysnplist, file="myfile.txt")  
> resultsdf <-
```

```
AnnotateSNPFile("myfile.txt", "Annotation_output.html")
```

10. The following `GetGenesInSNPs` function returns a list of genes that contain the given SNP:

```
> geneList <- GetGenesInSNPs(mysnplist)
```

11. If we want to know the gene in a given region (by chromosome name or number and position), use the `GetGenesInRegion` function as follows:

```
> GetGenesInRegion("3", 1, 1000000)
```

How it works...

Most of the search and fetch functions explained previously query the HapMap page at <http://hapmap.ncbi.nlm.nih.gov/cgi-perl/gbrowse/hapmap> for the SNP IDs.

The `GetSNPFlankSeq` function fetches the flanking position and the variation code of a set of SNPs from the remote NCBI location. It must be noted that the SNP IDs provided as input for the function must be valid. The following screenshot shows the sequence flank that is retrieved for the list of SNPs:

```

marker variation
3   rs94         Y
2   rs334        N
1   rs309        Y

fiveprime
3
atcacaaaagaagtgaatatgccctgccccaccttaactgatgacattccaccacaaaagaagtgtaaatgg
ccggtccttgcccttaagtgatgacattaccttgtaaagtcctttcc
2
GCAATTTGTA CTGATGGTATGGGGCCAAGAGATATATCTTAGAGGGAGGGCTGAGGGTTTGAAGTCCAATC
CTAAGCCAGTGCCAGAAGAGCCAAGGACAGGTACGGCTGTCATCACTTAGACCTCACCTGTGGAGCCACAC
CCTAGGGTTGGCCAATCTACTCCAGGAGCAGGGAGGGCAGGAGCCAGGGCTGGGCATAAAAGTCAGGGCAG
AGCCATCTATTGCTTACATTTGCTTCTGACACAACCTGTGTTCACTAGCAACCTCAAACAGACACCATGGTGC
ATCTGACTCCTG
1
tatttttagtagcgacaggggtctcaccatggtggccaagctggtcttgaa

threeprime
3
ggctcatcctggctcaaaaagcacccccactgagcaccttgtagccccactcctgccaccagagaacaaa
cccccttgactgtaattttccttacgtacccaaatcctataaaacG
2
GGAGAAGTCTGCCGTTACTGCCCTGTGGGGCAAGGTGAACGTGGATGAAGTTGGTGGTGAAGCCCTGGGCAG
GTTGGTATCAAGGTTACAAGACAGGTTTAAGGAGACCAATAGAACTGGGCATGTGGAGACAGAGAAGACTC
TTGGGTTTCTGATAGGCACTGACTCTCTGCTATTGGTCTATTTCCACCTTAGGCTGCTGGTGGTCT
ACCTTGGACCCAGAGGTTCTTTGAGTCCTTGGGGATCTGTCCACTCCTGATGCTGTTATGGGCAACCTA
AGGTGAAGGCTC
1
tcctgatctcaggtgatccaccgcctcggcctccaaagtgctgggatt

flag
3   0
2   0
1   0

```

The `GetSNPInfo` function fetches information such as the genome position and gene details (symbol and Entrez ID) if the SNP is located inside of any genes (if it is within the boundaries of a gene). The returned value also has information about the chromosome and species. The following screenshot shows the information retrieved from the SNPs:

Note: Some SNP locations could not be found. They have a chrpos of zero.
Some markers were not found. Will attempt a second method.

marker	genesymbol	locusID	chr	chrpos	fxn_class	species	dupl_loc	current.rsid	flag
1 rs94			6	62315934		Homo sapiens	Y:23206876	rs94	1
2 rs334	HBB	3043	11	5248232	missense,reference	Homo sapiens		rs334	0
3 rs309			12	92630922		Homo sapiens		rs309	0

There's more...

The `NCBI2R` package annotates SNPs, genes, and so on, using the current information available at NCBI.

See also

- The `NCBI2R` package's blog page at <http://ncbi2r.wordpress.com/>, which provides a more detailed overview and description of the `NCBI2R` package

Testing data for the Hardy-Weinberg equilibrium

The **Hardy-Weinberg Equilibrium (HWE)** is a principle stating that the genetic variation in a population will remain constant from one generation to the next in the absence of disturbing factors. When mating is random in a large population with no disruptive circumstances, the law predicts that both genotype and allele frequencies will remain constant because they are in equilibrium. However, mutations, natural selection, nonrandom mating, genetic drift, and gene flow disturb this equilibrium. Any deviation from this equilibrium will thus indicate these phenomena, plus the quality of genotyping. Therefore, performing an HWE test on data can be helpful while analyzing genotypic data. The tests performed in this recipe for all or a selection of SNP genotypes as will indicate if the deviation from HWE is significant and hence indicative of one of the phenomena mentioned here. This is usually accomplished with the Fisher exact test or Chi-square tests.

Getting ready

We will perform the HWE test for two different types of data: first, for simulated data, and secondly for the SNP data on Alzheimer's, which is available with the code files on the book's web page. The data is that the late-onset Alzheimer's disease is prevalent in a Japanese population, and a number of SNPs have been genotyped in the APOE region. These SNPs are listed in physical order, with the distance between the first and last SNP around 55Kb. Besides this, we will use the [GWASExactHW](#) library to meet the objective.

How to do it...

To check the HWE for an SNP data set, perform the following steps:

1. First, load the `GWASExactHW` library in the R session as follows:

```
> install.packages("GWASExactHW")
> library(GWASExactHW)
```

2. Now, simulate some SNP data. To do this, artificially generate SNP variants via a multinomial distribution. Use the conditions of HWE to generate data as follows:

```
> pA <- runif(1)
> pAA <- pA^2
> pAa <- 2*pA*(1-pA)
> paa <- (1-pA)^2
> myCounts <- rmultinom(100, 500, c(pAA, pAa,
paa))
```

3. Now, create a `data.frame` object for the counts (frequencies) of all of the variants as follows:

```
> genotypes <- data.frame(t(myCounts))
> colnames(genotypes)= c("nAA", "nAa", "naa")
```

4. Then, use the `HWEExact` function to compute the p-values as follows:

```
> hwPvalues <- HWEExact(genotypes)
```

5. Use real SNP data for this part of the recipe as follows (note that the data is available with the code files on the book's web page):

```
> load("path/to/code/file/dir/Alzheimers.rda")
```

6. Performing the previous step loads an object named `Alzheimers`; take a look at its contents as follows. It is a `data.frame` object that contains the frequencies of each allele along columns for each SNP along the rows as follows:

```
> head(Alzheimers)
```

7. Use the data for the exact test with the following `HWEexact` function:

```
> myTest <- HWEexact(Alzheimers)
> names(myTest) <- rownames(Alzheimers)
```

8. Plot the p-values after the test as follows:

```
> plot(-log10(myTest), type="b", ylab="-
log10(p-value)", main="HWE p-values for SNPS")
> abline(h=-log10(0.05), col="red")
```

9. To extract the significant SNPs, look for a p-value lower than 0.05 as follows:

```
> sum(myTest<0.05)
[1] 12
```

10. To get the name of the significant SNPs from the HWE test, simply extract the ones for which the p-value is less than 0.05, as follows:

```
> names(myTest)[which(myTest<0.05)]
```

11. Now, imagine a situation where you do not have frequencies; rather, you have the alleles of SNPs in different samples (individuals). Recall the SNP data from the association study recipe.
12. Source the `SNPfreqCalc.R` function from the code files provided in your workspace. The function contains scripts to compute frequencies from such SNP data. Also, load the SNP data from the `SNPassoc` library as follows:

```
> source("path/to code/file/SNPfreqCalc.R")
> library(SNPassoc)
> data(SNPs)
```

13. Now, run the following function to get the frequencies:

```
> freq2 <- freqCalc(SNPs, pat="snp")
```

14. This gives a frequency table like the one you used for other data. Run the following `HWEexact` function on it, as you did earlier, to get the statistics:

```
> myTestSNPs <- HWEexact(freq2)
```

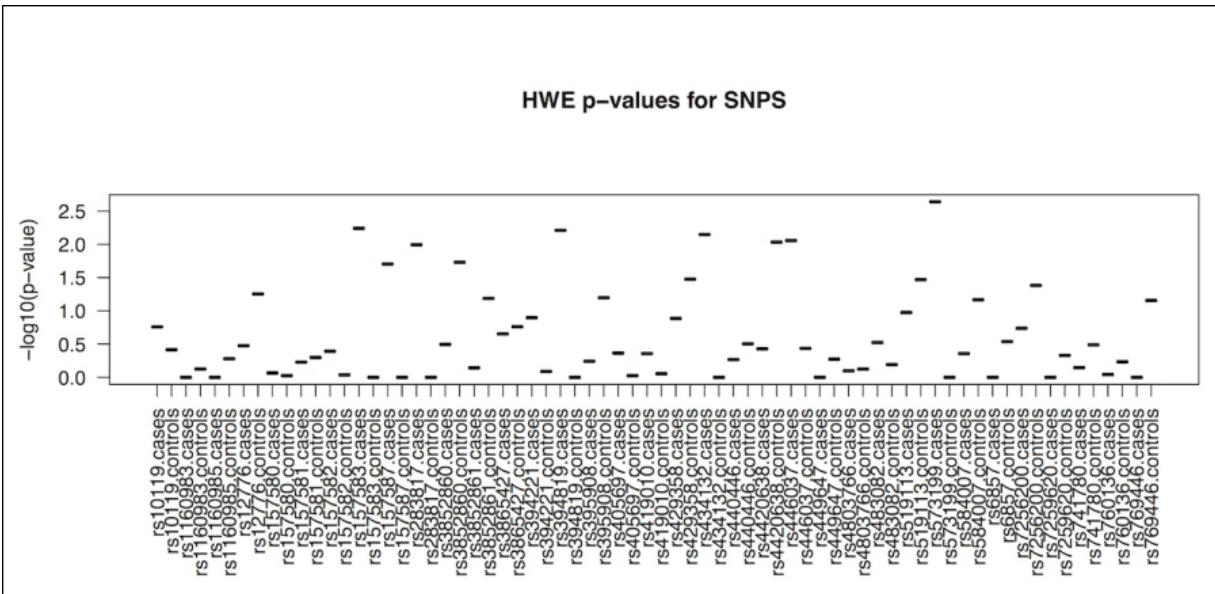
How it works...

The `HWEexact` function performs a Fisher test on all the SNPs in the data and returns the corresponding p-values. The simulated data first generates a random probability of the variants and then varies itself using these probabilities from a multinomial distribution and finally a frequency table for variants. This table acts as the input for the `HWEexact` function.

In the second case, we use the Alzheimer's data that has allele frequencies for every SNP. The `HWEexact` function runs on all the data and computes the exact scores for the allelic frequencies and returns a score for each SNP. We can simply plot the scores (p-values) on a log scale to find significantly deviant SNPs as per the HWE. It is known that in the case of Alzheimer's, the APOE region plays an important role. Our data has the SNPs `rs446037` through `rs429358` that lie in the APOE gene. The SNPs `rs429358` and `rs7412` (not genotyped, but adjacent to `rs429358`) together make up the alleles $\epsilon 2$, $\epsilon 3$, and $\epsilon 4$ of the APOE gene. The results obtained show the SNPs deviating from the HWE.

For the third type of data, we have the `freqCalc` function that computes a frequency table from the SNP files that contain the alleles. The function computes the frequency of each allele for every SNP across all samples (individuals) and finally removes the phenotype information available in the data by looking for a pattern of names provided for the SNP IDs. In this recipe, we used `snp` as a value argument `pat` for this. However, we can use the other patterns such as `rs` to identify the SNP. This function can accept most of the

patterns for the SNP names. The following screenshot shows a plot of negative logarithms for SNPs after the HWE test. As shown in the following screenshot, along the x axis, we have the SNPs, and the y axis represents the negative logarithm of the p-values (base 10):



See also

- The *A Note on Exact Tests of Hardy-Weinberg Equilibrium* article by Wigginton and others at <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1199378/>, which provides a detailed explanation of the test
- The *Genome-wide association study of Alzheimer's disease* article by Kamboh and others at <http://www.nature.com/tp/journal/v2/n5/full/tp201245a.html> to know more about the role of SNPs in Alzheimer's

Association tests with CNV data

So far, we have discussed SNPs a lot. **Copy Number Variation (CNV)** refers to a phenomenon where for certain genes, there exists a normal variation in the number of copies of one or more sections of the DNA. CNVs are an important component of structural variation in the human genome and are potentially critical in terms of their genetic contribution to the risk of common complex diseases. Just like the SNP-based association analysis that we saw so far, the CNV association analysis can play a crucial role in understanding many complex phenotypes and diseases. This recipe will introduce some methods to perform such an analysis.

Getting ready

We will use the `CNVassoc` package for our recipe here, which is available from the CRAN repository. The input data for our demo will be the built-in data in the package received from the MLPA assay for association between CNVs and disease.

How to do it...

Perform the following steps for the association analysis of the CNV association:

1. As usual, start with installing and loading the `CNVassoc` library as follows:

```
> install.packages("CNVassoc")
> library(CNVassoc)
```

2. After this, load the MLPA data in the workspace by typing the following command:

```
> data(dataMLPA)
```

3. Take a look at the MLPA data, which is a data frame in the `head` function, by typing the following function:

```
> str(dataMLPA)
> head(dataMLPA)
```

4. To check the intensities that correspond to the phenotype, plot the densities as follows:

```
>
plotSignal(dataMLPA$Gene1, case.control=dataMLPA$casco)
```

5. Now, as you have seen the data peak intensities for the two genes, you need to create a `cnv` object for the gene you want to study. Here, you will attempt to study Gene 2. Create the `cnv` object as follows:

```
> myCNV <- cnv(x = dataMLPA$Gene2, threshold.0
= 0.01, mix.method = "mixdist")
```

6. Now, use the `cnv` object to compute the association using the `CNVassoc` function, as follows:

```
> myModel <- CNVassoc(formula=casco ~ myCNV,
data = dataMLPA, model = "mul")
```

7. To look at the estimated parameters, type the following command:

```
> summary(myModel)
```

8. To do an Anova (with an additive model for example) or likelihood computation, use the corresponding function, as follows:

```

> myModel2 <- CNVassoc(formula=casco ~ myCNV,
data = dataMLPA, model= "add")
> anova(myModel, myModel2)
> logLik(myModel)

```

9. Finally, to determine whether a CNV is associated with the phenotype, perform a Wald or likelihood test on your model with the help of the following commands:

```

> CNVtest(myModel, type = "LRT")
----CNV Likelihood Ratio: Test----
Chi= 18.75453 (df= 2) , pvalue= 8.462633e-05

> CNVtest(myModel, type = c("Wald","LRT"))

> CNVtest(myModel, type = "Wald")
----CNV Wald test----
Chi= 17.32966 (df= 2) , pvalue= 0.0001725492

```

This returns the p-value for the association based on the test that was performed.

How it works...

The MLPA data set contains the case control status between the CNV and disease. The data contains the peak intensities for two genes that arise from the MLPA assay for 360 cases and 291 controls. The data has two genes: under study and corresponding covariates. The following screenshot shows the head of the MLPA CNV data:

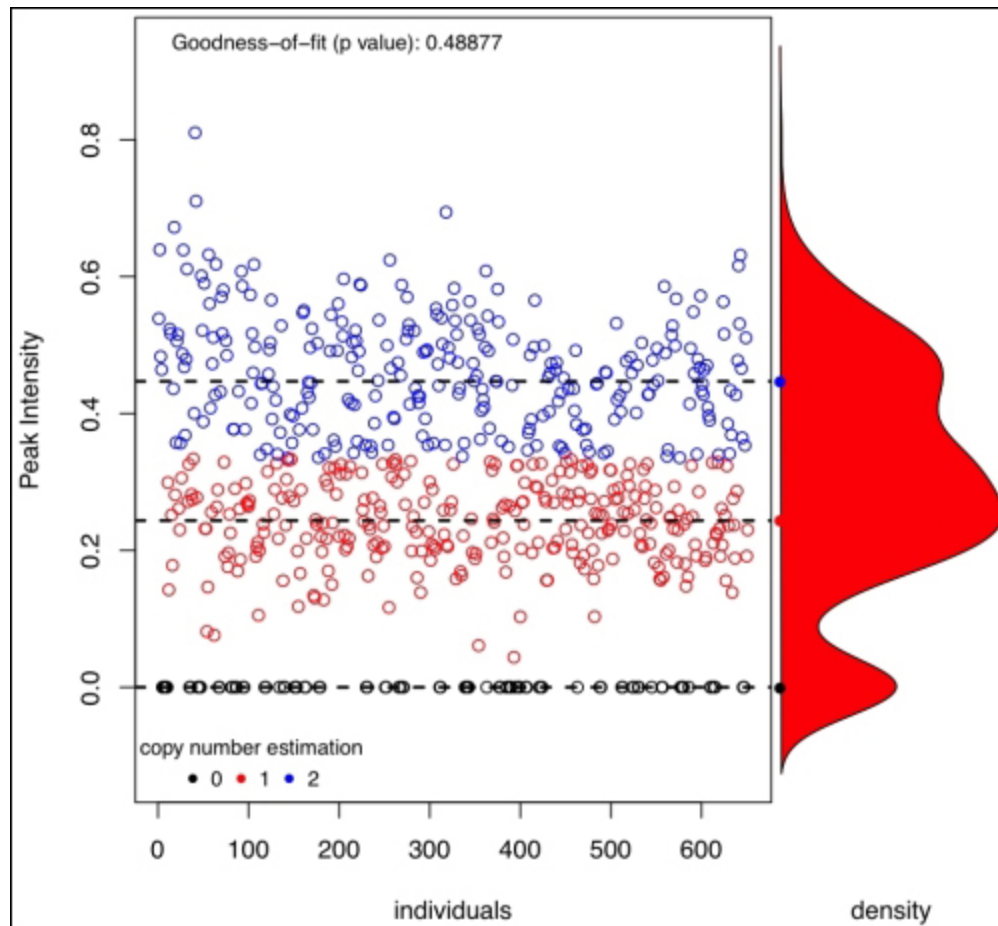
	id	casco	Gene1	Gene2	PCR.Gene1	PCR.Gene2	quanti	cov
1	H238	1	0.51	0.5385080	wt	wt	-0.61	10.83
2	H238	1	0.45	0.6392029	wt	wt	-0.13	10.69
3	H239	1	0.00	0.4831572	del	wt	-0.57	9.63
4	H239	1	0.00	0.4640072	del	wt	-1.40	9.87
5	H276	1	0.00	0.0000000	del	del	0.83	10.25
6	H276	1	0.00	0.0000000	del	del	-2.07	10.40

The `cnv` function uses the quantitative signal of individual probes to infer the copy number status for samples. The function is based on

the normal mixture model assumptions. The plot of the `cnv` object shows the peak intensities for a gene, representing the intensities of Gene 2 for each individual, distinguishing between cases and controls. The following `plotSignal` function plots the peak intensities for a gene individually (note that we can do similar plots for the other gene as well and compare them):

```
>
plotSignal(dataMLPA$Gene2, case.control=dataMLPA$case)
sco)
```

The following screenshot shows the plot signal for the intensity peaks in the CNV data for Gene 2. Along the x axis, we have the individuals, and along the y axis, the peak intensity. The plot shows the peaks for three copy number estimations: 0, 1, and 2 (in black, red, and blue). The densities are plotted along the vertical axis. The intensities of the peak in the following screenshot show the different densities for the different estimated copy numbers:



The `cnvassoc` function uses a latent class model to perform an association analysis between a CNV and the phenotype. It treats the phenotype as a binary variable. While using the function, the value for the `family` argument should be based on the distribution of the response variable. Finally, the `cnvtest` function computes the global significance p-value for the association.

There's more...

Note that illumina or affymetrix data, where log2 ratios are available instead of peak intensities, can be analyzed in the same way that we illustrated in this section. As stated earlier, the `cnvassoc` function uses a latent class for test association. This latent class incorporates uncertainties that model the degree of misclassification of the copy number status. More information about the latent class is available in the *Accounting for uncertainty when assessing association between*

copy number and disease: a latent class model article by Gonalez and others at <http://www.biomedcentral.com/1471-2105/10/172>.

The `multiCNVassoc` function can be used to test multiple CNVs. To learn more about the function, type in `?multiCNVassoc` in the R console. The `vignette` package offers some more information on the function (http://cran.fhcrc.org/web/packages/CNVassoc/vignettes/CNVassoc_vignette.pdf).

To perform a genome-wide CNV analysis, the `patchwork` package can be useful. It can be used to determine the copy number of homologous sequences throughout the genome and is handy for aneuploidy samples with moderate coverage. For further details, refer to the *Patchwork: allele-specific copy number analysis of whole-genome sequenced tumor tissue* article by Mayrhofer and others (<http://genomebiology.com/content/14/3/R24>).

Visualizations in GWAS studies

In order to present the results obtained from the GWAS analysis, we can perform several intuitive, rational, and appealing visualizations. The most commonly used ones include Q-Q plots, the Manhattan plot, regional association plots, and so on. This recipe presents the methods to generate some of these plots. It is extremely helpful if you're looking to publish your findings as an article or a talk.

Getting ready

We need results to create plots, and for this recipe, we will use our existing results, simulated results, and some ready-made results available within R packages.

How to do it...

To create some interesting visualization for your GWAS results, perform the following steps:

1. Start with a Q-Q plot. To do this, first load the `GWASTools` library as follows:

```
> library(GWASTools)
```

2. As input, take the whole genome association results from the previous recipe on SNP data. To refresh the memory, recompute the association as follows:

```
> myres <- WGassociation(protein, data=mySNP,  
  model="all")
```

3. Then, extract the p-values for the dominant model of inheritance from the results as follows:

```
> pvals <- dominant(myres)
```

4. Then, use the following `qqPlot` function with these p-values as input to generate a Q-Q plot:

```
> qqPlot(pvals)
```

5. The next plot will be a Manhattan plot. Start with the simulated p-value data of 1000 SNPs by typing the following command:

```
> n <- 1000  
> pvals <- sample(-log10((1:n)/n), n,  
  replace=TRUE)
```

6. Then, assign these p-values to random chromosomes as follows:

```
> chromosome <- c(rep(1,100), rep(2,150),  
  rep(3, 80), rep(4, 90), rep(5,100), rep(6,60),  
  rep(7, 70), rep(8, 70), rep(9,70),  
  rep(10,50), rep("X",110), rep("Y",50))
```

7. To create the plot, call the `manhattanPlot` function from the `GWASTools` library as follows:

```
> manhattanPlot(pvals, chromosome)
```

8. You can create a similar plot with the real results. Here, use the results from your `HapMap` data. Start with the extraction of the p-value for the dominant inheritance model from the whole genome SNP association analysis results, and take the negatives of the log values to the base 10, as follows:

```
> pvals=dominant(myHapMapres)  
> pvals=-log10(pvals)
```

9. Now, use the `HapMap` position data to get information about the chromosomal location of the SNPs as follows:

```
> chromosome <- HapMap.SNPs.pos$chromosome
```

10. As earlier, run the `manhattanPlot` function with these p-values and chromosome information as follows:

```
> manhattanPlot(pvals, chromosome, signif=1e-5)
```

11. The next plot is the regional association plot. To do this, get the gap package as well as the `CDKN` data as follows:

```
> install.packages("gap")
> library(gap)
> data(CDKN)
```

12. The `gap` library loads the `CDKN` data that has three subsets. To look at the data, use the `head` function on them as follows:

```
> head(CDKNlocus)
```

The following screenshot shows the contents of the `CDKN` data:

	CHR	POS	NAME	PVAL	RSQR
1	9	21880326	rs7865071	0.82418370	0.020
2	9	21884495	rs7389178	0.79907230	0.017
3	9	21913279	rs10811638	0.06569925	0.001
4	9	21917327	rs4977749	0.12945710	0.001
5	9	21919666	rs2518713	0.37621220	0.002
6	9	21944953	rs10757261	0.76992020	0.010

```
> head(CDKNmap)
```

	POS	THETA	DIST
1	19999135	0.1778173	0.00000
2	20000312	0.1791786	40.55383
3	20001576	6.9411662	40.55406
4	20001821	10.9959827	40.55576
5	20002125	11.3206207	40.55910
6	20002593	9.2593987	40.56440

```
> head(CDKNgenes)
```

	START	STOP	STRAND	GENE
1	116267060	117256871	-	ASTN2
2	27938527	28709303	-	LINGO2
3	8307267	9008737	-	PTPRD
4	70379521	70966068	-	TRPM3
5	123221486	123771971	-	DENND1A
6	98129920	98551034	-	GABBR2

13. The `CDKNlocus` object provides all of the information to create the plot. The `asplot` function uses the locus, map, and gene information from these data sets to create a regional association plot as follows:

```
> asplot(CDKNlocus, CDKNmap, CDKNgenes,
best.pval=5.4e-8, sf=c(3,6))
```

Extract similar information for the data of your interest from the analysis we did so far. For the genetic maps, use the data at the HapMap page

(http://www.hapmap.org/downloads/recombination/2006-10_rel21_phaseI+II/rates/).

14. You can create a regional plot using the `postgwas` library. Start with loading the library as follows:

```
> install.packages(c("postgwas", "taRifx")) #  
please install from source as explained in  
Chapter 1, Starting Bioinformatics with R, if  
incompatible with your R version  
> library(postgwas) # please check the  
availability of the package with your R  
version
```

15. Create a `data.frame` object for the SNP of your interest, as follows:

```
> snps <-  
data.frame(SNP=c("rs4648633", "rs6659552"))
```

16. Now, create the `gwas data.frame` object that consists of the chromosome, SNP, position, and p-values from the `HapMap` data as follows:

```
> gwasdata <- data.frame(CHR=  
HapMap.SNPs.pos$chromosome, SNP=colnames(HapMap  
) [3:ncol(HapMap)] ,  
BP=HapMap.SNPs.pos$position, P=dominant(myHapMa  
pres))
```

17. Convert the chromosome name `data.frame` to an appropriate form for the Biomart annotation (numbers). It uses the following `taRifx` library:

```
> library(taRifx)  
> gwasdata$CHR <- as.factor(destring(  
as.character(gwas  
data$CHR), keep="0-9.-"))
```

18. Save the data as a text file with the help of the following command:

```
> write.table(gwasdata, file="mygwas.txt",  
sep="\t", quote=FALSE)
```

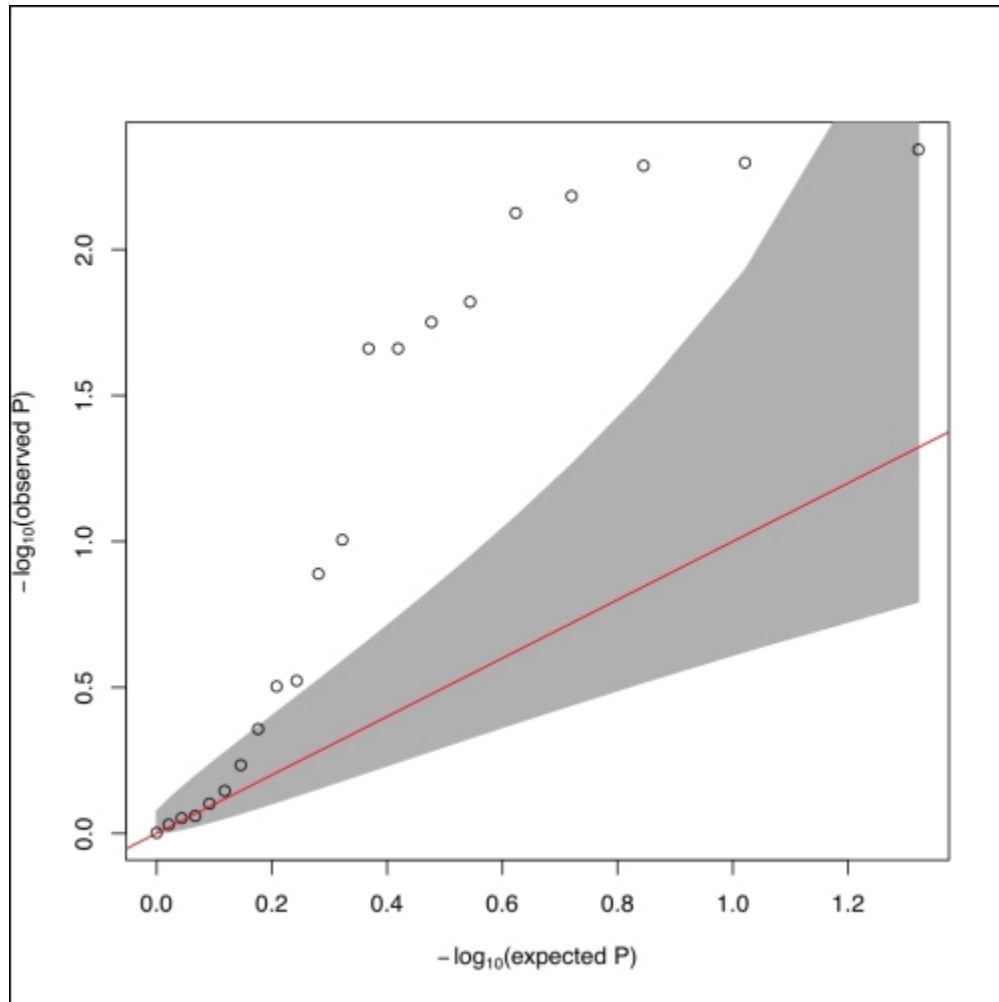
19. Run the following `regionalplot` function using this file and the SNP of your interest as the input argument:

```
> regionalplot(snps=snps,  
gwas.datasets="mygwas.txt")
```

How it works...

The first plot we created in this recipe is a Q-Q plot. The data used here comes from our association studies in the previous recipe. The study finds association between a certain protein and its genotype. The Q-Q plots in GWAS show the expected distribution of association test statistics (x axis) across the million SNPs compared to the observed values (y axis). A deviation from the diagonal $X = Y$ refers to a consistent difference between cases and controls across the whole genome. Thus, a Q-Q plot can be used to characterize the extent to which the observed distribution of the test statistic follows the expected (null) distribution in the given data.

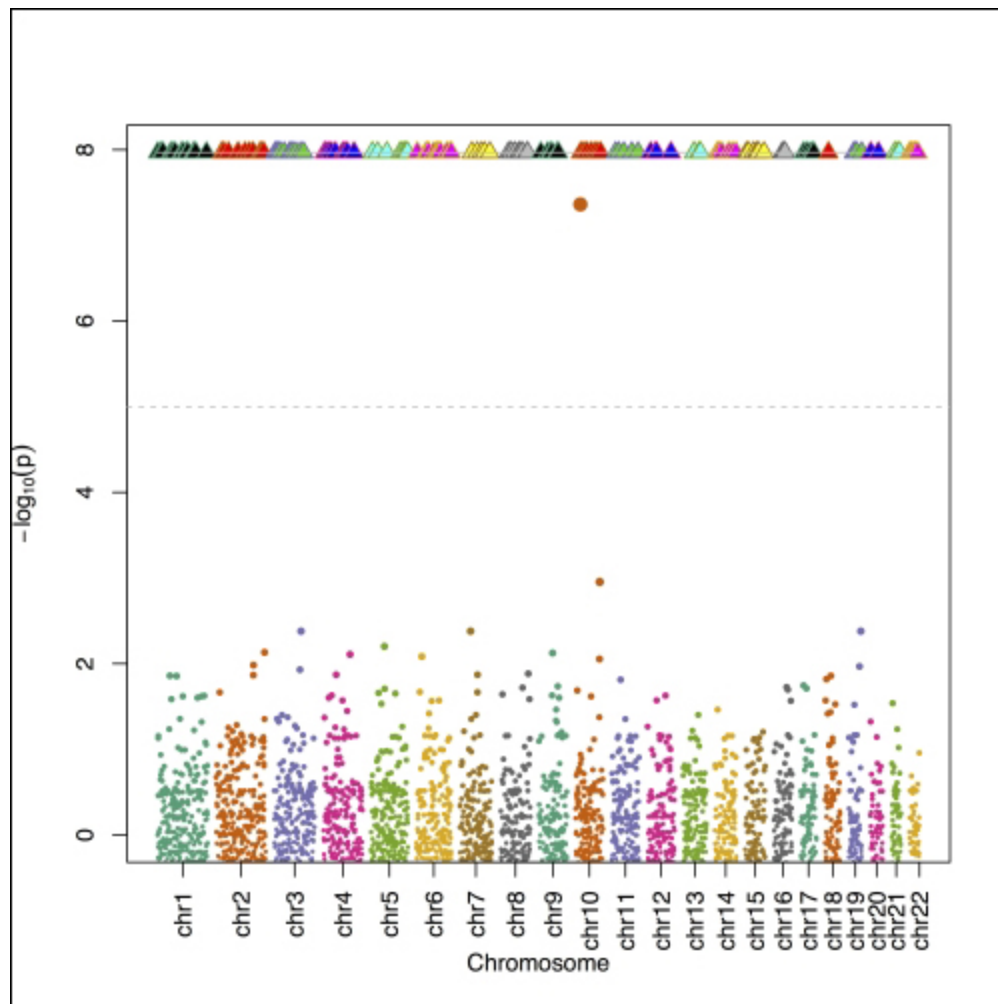
The following screenshot shows the Q-Q plot for the SNP data with the observed values along the y axis and the expected values along the x axis (test statistics)—note that the red line is $X = Y$:



The second plot we worked out here is a Manhattan plot. We used the results obtained on analyzing the [HapMap](#) data in the recipe for the whole genome SNP association analysis. As mentioned, it represents the significance of the association of SNPs with the trait or phenotype under study. The y axis is the $-\log_{10}$ transformation of p-values, where the p-values represent a measure for the strength of association. The plot is color coded by chromosome. In our case, we created plots for simulated data, which is straightforward. Then, we extracted results from our whole genome [HapMap](#) study and created a plot for this.

The following screenshot shows Manhattan plots for the [HapMap](#) data. As shown in the following screenshot, we have the chromosome

numbers on the x axis, and the y axis represents the $-\log_{10} P$ -values:

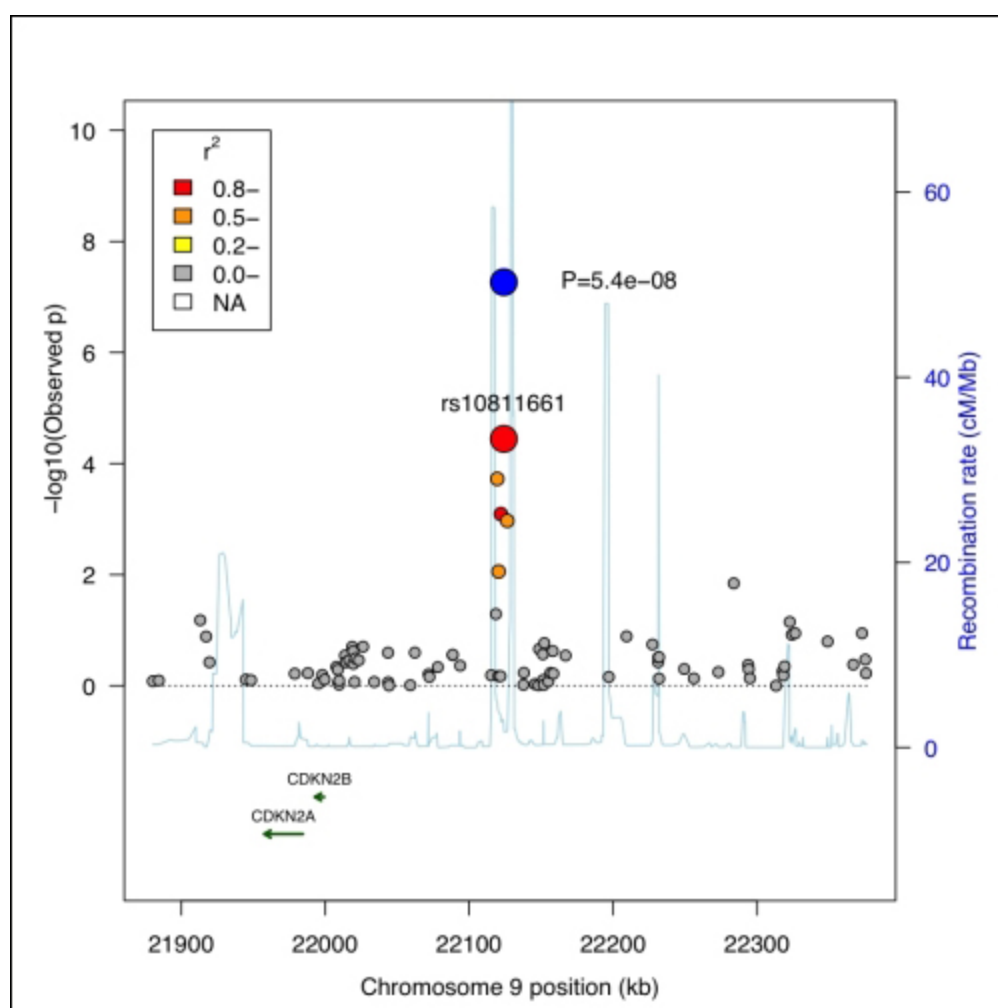


The regional association plot we created in this recipe uses three different `data.frames` as follows:

- The SNP and its association details
- The position and map details of the SNP
- The details of the genes on the genome

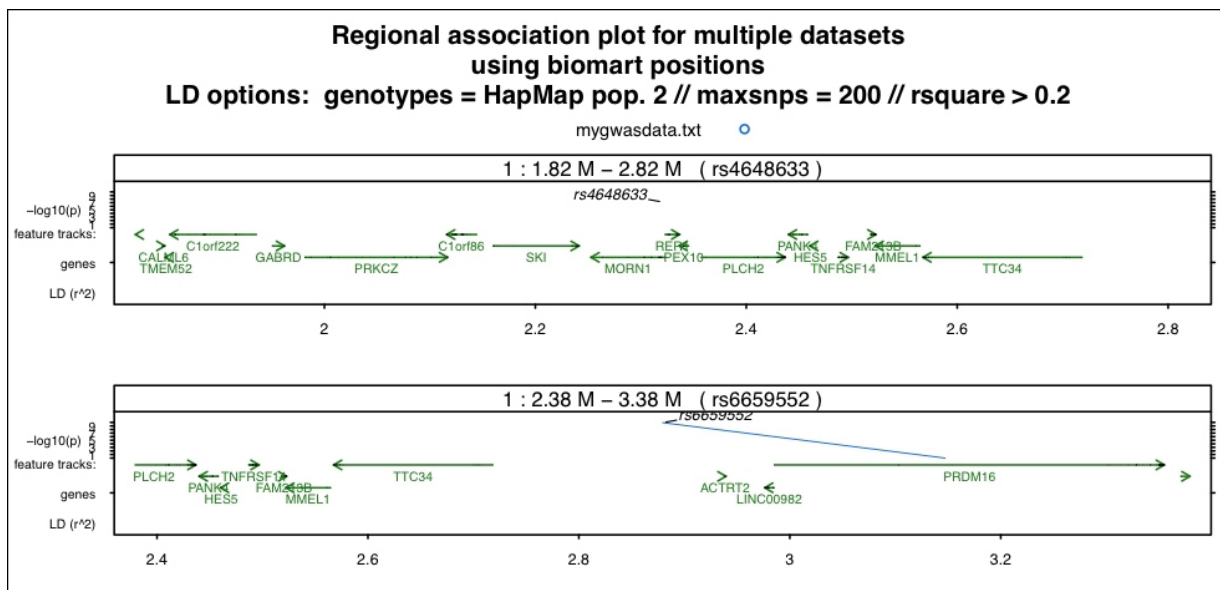
Our input data comes from a study of the CDKN regions of the ninth chromosome for association studies originating from Diabetes Genetics Studies. The resulting plot is for a particular locus that is based on the information about the recombination rates, **linkage disequilibria (LD)** between the SNP of interest and neighboring

ones, and the p-values of single-point association tests. These plots are not very common, but they are useful at times. The plot shows an annotation for the p-values of the SNP names and their location on the chromosome together with the LD measurements. The `asplot` function accepts the locus, genes, map, and analysis results (for statistical scores) to create the plot. Take a look at the structure of the individual objects used as input to get details of the plotting function. In the following screenshot, the x axis represents the location site in the ninth chromosome in terms of kilo bases, and the y axis shows the negative logarithm of observed p-values. The recombination rate is also along the y axis and shows the local linkage disequilibrium structure around the associated SNP. In the following screenshot, the circles represent the SNPs and their color depicts the r^2 :



The regional association plot that uses the gap package for the CDKN data

Now, we will discuss another type of regional association plot. This plot appears often in GWAS-related articles. The data used as input for the plot is again acquired from our HapMap analysis (`myHapMapres`) and converts into a `data.frame` object. This object contains the chromosome number, the location details, the SNPs, and the corresponding p-values (in our case, from the dominant model). The function can map more than one SNP with the neighboring genes and association measurements. It plots the regional plot for an SNP with a given window size (default 106). It helps in phenotypic comparison based on the data. Besides this, it also carries the gene information and some other information. With this function, we can include the p-value graphs of two datasets to do a comparison. We did it here for two SNPs (the `snps` object). We can add more SNPs or data in our file and plot them together as we did for these two SNPs. In the following screenshot, the x axis shows the position or location along the chromosomes and the y axis shows the $-\log_{10}$ p-values:



The regional plot for the selected SNPs from the HapMap data

There's more...

A Manhattan plot is basically a scatter plot accustomed to high-dimensional data. It is another common type of plot that one can find often in the articles published in the field of GWAS. In this recipe, we used the `GWASTools` package to do this plot; however, there are several other implementations for Manhattan plots. One can also do this simply with the `ggplot2` package. The blog page that can be of help to create such a plot using `ggplot2` is available at <http://www.r-bloggers.com/annotated-manhattan-plots-and-q-q-plots-for-gwas-using-r-revisited-2/>.

The `gap` package was not only used to plot the regional association, but it was also used for the data set. The package's page can be found at <http://cran.r-project.org/web/packages/gap>.

The `postgwas` package is an R package that has several functionalities for GWA studies. It can act as a substitute for many packages that we used in this chapter. To know more, refer to the *Postgwas: Advanced GWAS Interpretation in R* article by Hiersche and others at <http://www.plosone.org/article/info%3Adoi%2F10.1371%2Fjournal.pone.0071775;jsessionid=036863964F881DFB1D94CD429095E4AF>.

See also

- The detailed and interesting description of Q-Q plots at <https://www.stat.auckland.ac.nz/~ihaka/787/lectures-quantiles.pdf>, which provides detailed information about Q-Q and derived plots in order to understand the theoretical background in detail
- The HapMap site at http://www.hapmap.org/downloads/recombination/2006-10_rel21_phaseI+II/rates/
- The report titled *Genome-Wide Association Analysis Identifies Loci for Type 2 Diabetes and Triglyceride Levels* by Consortium at that appeared in the Science journal, which provides details of the data set

Chapter 7. Analyzing Mass Spectrometry Data

This chapter will cover the following recipes:

- Reading the MS data of the mzXML/mzML format
- Reading the MS data of the Bruker format
- Converting the MS data in the mzXML format to MALDIquant
- Extracting data elements from the MS data object
- Preprocessing MS data
- Peak detection in MS data
- Peak alignment with MS data
- Peptide identification in MS data
- Performing protein quantification analysis
- Performing multiple groups' analysis in MS data
- Useful visualizations for MS data analysis

Introduction

Mass spectrometry (MS) is a key analytical technology used in current proteomics to generate large amounts of high-quality data, addressing the issues of protein identification, annotation of secondary modifications, and determining protein abundance. The data consists of the intensity versus the mass-to-charge ratio (popularly called m/z ratio) in the biochemical or chemical sample, exhibiting the distribution pattern of the components in the sample. Nevertheless, in this chapter, we will mostly deal with biochemical samples. Among the many applications of MS, biomarker pattern discovery has aroused considerable interest.

However, the challenges in the analysis of MS data range from statistical (noise handling, normalization, and so on) to biological (peptide identification, quantification, and so on). Therefore, the role of bioinformatics becomes fundamental for the elaboration of MS

data and for the screening of biological knowledge in the data. This chapter aims to present solutions to some key problems in MS data analysis for proteomics studies.

The book titled *Analysis of Mass Spectrometry Data in Proteomics* by Mattiesen and Jensen outlines some basic analysis concepts in MS data analysis from the point of view of bioinformatics.

In order to work with MS data, it is extremely important to know about the data formats that we will handle in the chapter. Over the years, different manufacturers of mass spectrometers have developed various data formats for handling and presenting MS data. This makes it difficult to have a uniform protocol to directly manipulate the data. To address this limitation, several open XML-based data formats have recently been developed by the Trans-Proteomic Pipeline at the Institute for Systems Biology and as an innovation in the public sector. These data formats are described here.

The data format used in MS actually depends on the hardware being used to produce the data. However, there has been a recent development in making these formats more or less consistent to facilitate data manipulation by the diverse scientific community. The two most widely used formats for MS data are [mzXML](#) and [mzML](#).

The [mzXML](#) format is an XML-based file format for proteomics mass spectrometric data. This data format has information that ranges from the hardware to processing as well as peaks. In recent times, [mzXML](#) has emerged as a popular format because of its flexibility and efficient handling of different data attributes. More details on the format can be found in the article titled *A common open representation of mass spectrometry data and its application to proteomics research* by Pedrioli and others. The article is available at <http://www.nature.com/nbt/journal/v22/n11/full/nbt1031.html>.

The [mzML](#) format is a hybrid format of [mzXML](#) and an obsolete format named [mzData](#). You can find more details on the data format specifications in the *mzML: A single, unifying data format for mass*

spectrometer output article by Eric Deutsch at
<http://onlinelibrary.wiley.com/doi/10.1002/pmic.200890049/abstract>.

In our recipe, we will focus on these two data formats, which are popular in the proteomics community, and binary files available within R packages. Nevertheless, we will have a look at the required information to work with some other formats too.

Reading the MS data of the mzXML/mzML format

The data available in the MS community is usually in the form of an XML file. It contains different fields with the allocated information. In order to start working with the MS data, you need to import this data into the R session. This recipe presents the steps to import XML-format-based MS data in the R workspace.

Getting ready

The recipe needs some sample data to work with. The data can be obtained from web resources or from within R packages. Besides the data, we will also need some R packages, which will be explained in the *How to do it...* section of this recipe. The example files used in this recipe are also provided with the code files on the book's web page.

How to do it...

The following steps will explain the approach to be taken in order to import XML-based MS data in our work session:

1. First, you need to get the data from <http://onlinelibrary.wiley.com/doi/10.1002/pmic.200890049/abstract> via a browser.
2. Fetch the file directly from the Web using the `wget` function of Unix as follows (no need to start R at this point; a Unix session will do):

```
$ wget regis-  
web.systemsbiology.net/rawfiles/lcq/7MIX_STD_1  
10802_1.mzXML
```

Make sure that you put this file in the directory to be used as the working directory for your R session, or use the path to the file.

3. Now, you will need the `readMzXmlData` library in the R session for the next set of steps; you can install the package from the CRAN repository and load it in the R session as follows:

```
> install.packages("readMzXmlData",  
dependencies = TRUE)  
> library(MALDIquant)  
> library(readMzXmlData)
```

4. Now, use the `mqReadMzXml` function to read the downloaded `mzXML` file as follows (remember that it might take a while to download the data):

```
> myData <-  
mqReadMzXml(path="7MIX_STD_110802_1.mzXML")
```

5. Use the following `class` function to check the object type for `myData`, which returns the `list` class:

```
> class(myData)
```

6. The number of elements in the list can be seen with the following function:

```
> length(myData)
```

7. Check whether the data has the `7161 MassSpectrum` class object in the list by looking at the summary for the `myData` object as follows:

```
> head(summary(myData))  
> dim(summary(myData))  
[1] 7161 3
```

8. Look at the structure of one data entry in your list by specifying the index in the list as follows:

```
> str(myData[[1]])
```

9. In order to be able to read all the `mzXML` files in a directory, use the `readMzXmlDir` function as follows:

```
> myData <-  
  readMzXmlDir(mzXmlDir="path/to/Directory")
```

10. To read the `mzML` files, use the following `mzR` library that can be installed again from the CRAN repository:

```
> install.packages("mzR") # in case of  
  installation warning please install the 0.11.0  
  version from source  
> library(mzR)
```

11. To read the `mzML` file, use the `openMSfile` function as follows (an example file is provided with the code files):

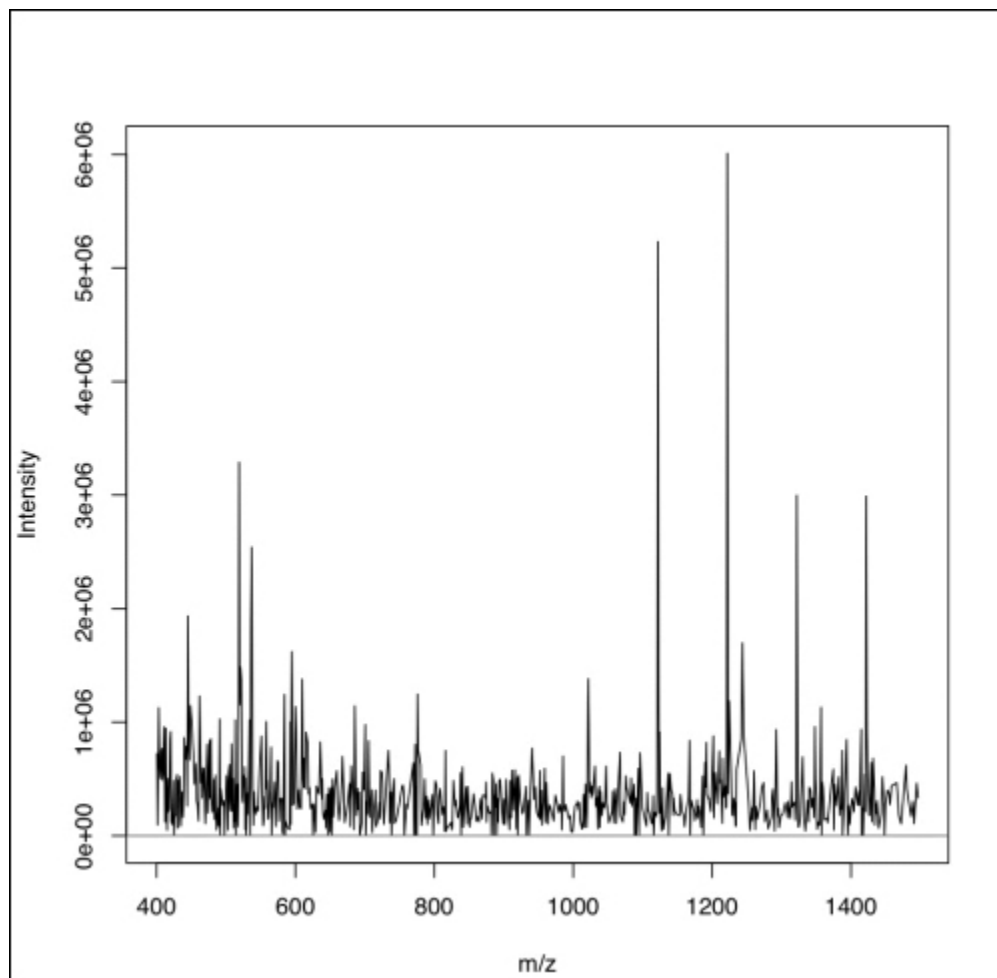
```
> myData <-  
  openMSfile("path/to/code/files/tiny.pwiz.1.1.m  
  zML")
```

How it works...

All the MS file formats discussed in this recipe share their structures with a set of metadata, followed by a list of spectra with the masses and intensities. In addition, each spectrum has its own set of metadata, such as the retention time and acquisition parameters. The mass and intensities can be plotted on the x and y axes, respectively, with the `plot` function. The plot shows the intensity (y axis) obtained for different masses (x axis). This gives us a clue for the peaks in the MS data as follows, which we will talk about in the upcoming recipes:

```
> plot(myData[[1]], col="black")
```

The following plot shows different fragments with their corresponding m/z ratios along the x axis and the observed intensities along the y axis:



Mass versus intensity plot for a spectrum

The functions explained earlier are based on the parsing of XML files. The read functions provided read in the fields in the XML file and put with the appropriate subclasses in the list (the object class). This can be seen in the following screenshot:

```

> str(myData[[1]])
Formal class 'MassSpectrum' [package "MALDIquant"] with 3 slots
 ..@ mass      : num [1:705] 401 402 403 404 406 ...
 ..@ intensity: num [1:705] 722826 95972 1126969 607818 504474 ...
 ..@ metaData  :List of 20
 .. ..$ file      : chr "/home/praveen/bookcode/ms/7MIX_STD_110802_1.mzXML"
 .. ..$ scanCount : num 7161
 .. ..$ startTime : num 0.00683
 .. ..$ endTime   : num 200
 .. ..$ parentFile :List of 1
 .. .. ..$ :List of 3
 .. .. .. ..$ fileName: chr "file:///Rdf3/data2/search/ppatrick/sashimi_repository/LCQ/7MIX_STD_110802_1.RAW"
 .. .. .. ..$ fileType: chr "RAWData"
 .. .. .. ..$ fileSha1: chr "957f3baf650d4de3d87c04a9fc64baa13f6b363e"
 .. ..$ msInstrument :List of 6
 .. .. ..$ msManufacturer: chr "ThermoFinnigan"
 .. .. ..$ msModel       : chr "LCQ Deca XP"

```

A part of the structure information for an MS object class, showing some of the fields in the data

Thus, the data not only consists of the mass-to-charge charge ratio or intensity, but also the fields for metadata such as experiment information and instrument details.

The parsers to read the data are implemented in C/C++ to make it faster and efficient and are accessed by R through wrapper functions.

See

- The CRAN page at <http://cran.r-project.org/web/packages/readMzXmlData/>, which provides details on the `readMzXmlData` packages
- The article titled *A cross-platform toolkit for mass spectrometry and proteomics* by Chambers and others at <http://www.nature.com/nbt/journal/v30/n10/full/nbt.2377.html>, which provides the theoretical background on `mzR` and associated tools
- The Bioconductor page at <http://www.bioconductor.org/packages/release/bioc/vignettes/mzR/inst/doc/mzR.pdf>, which provides details of the other functions in the `mzR` package

Reading the MS data of the Bruker format

MS data is usually available as XML-based files. Nevertheless, it is possible to use data obtained directly from the instrument. `Bruker XMASS` and `flexAnalysis` save peak lists in a simple XML format. You can use converters, such as `CompaxxXport`, to get these files converted to the `mzXML` format. However, we can also read these files from within R directly. This recipe will explain the steps to be performed.

Getting ready

This recipe will require the data in the `Bruker` format and the R package `readBrukerFlexData`.

How to do it...

Perform the following steps for this recipe:

1. First, install and load the `readBrukerFlexData` library from the CRAN repository as follows:

```
> install.packages("readBrukerFlexData")
> library(readBrukerFlexData)
```

2. Now, you need to define the directory where your datafiles are located.
3. Use the example file from the `readBrukerFlexData` package. The path to the file can be extracted using the `system.file` function in R as follows:

```
> exampleDirectory <- system.file("Examples",
package="readBrukerFlexData")
```

4. To read all the files in the directory assigned earlier, use the following command:

```
> myData_Bruker <-  
  readBrukerFlexDir(file.path(exampleDirectory))
```

5. Take a look at the data by using the generic `summary` function as follows:

```
> summary(myData_Bruker)
```

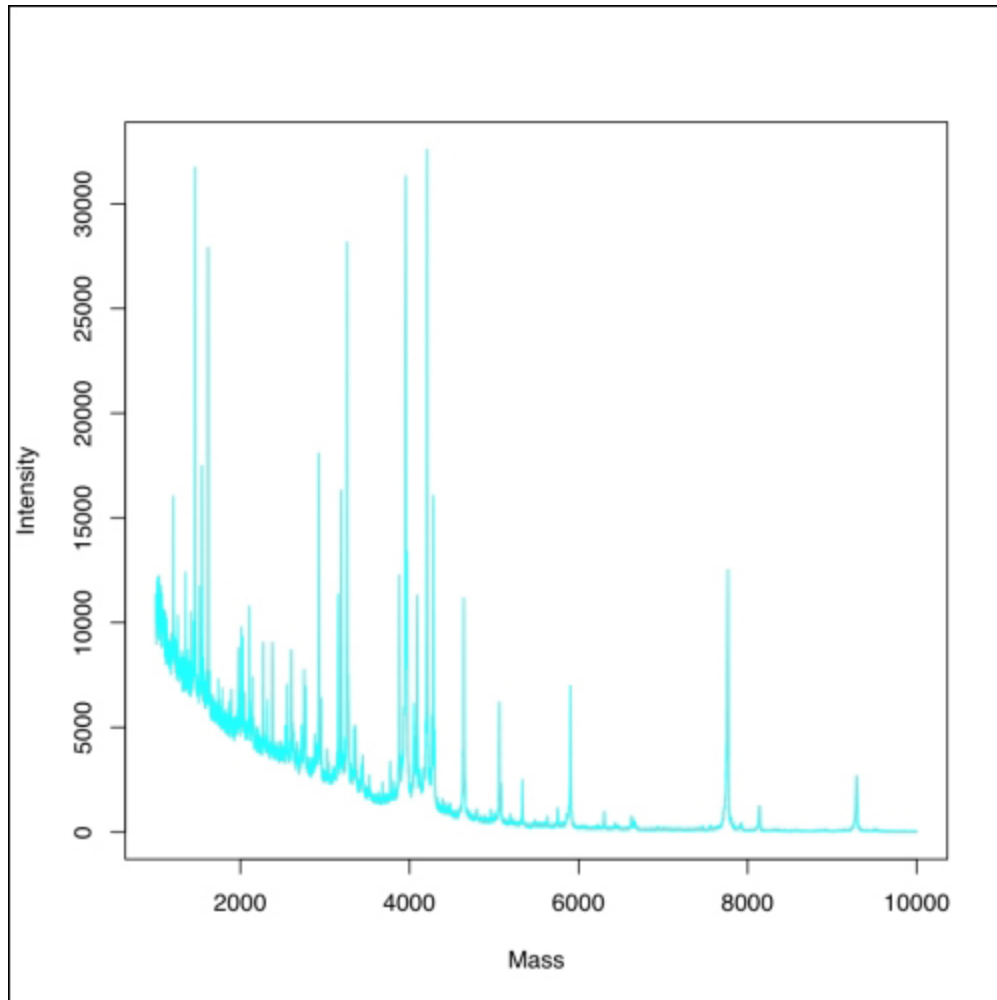
6. Take a look at the structure of the object obtained using the following `str` function:

```
> str(myData_Bruker)
```

7. To plot the intensity value versus mass in MS data use the corresponding fields of data as follows:

```
> plot(myData_Bruker[[1]]$spectrum$mass,  
       myData_Bruker[[1]]$spectrum$intensity,  
       xlab="Mass", ylab="Intensity", col="skyblue")
```

In the following screenshot, the peaks indicate higher intensities for the fragments with the corresponding m/z ratio:



Mass versus intensity plot for the Bruker format data

How it works...

The `readBrukerFlexData` function works in a similar way as the previous recipe. However, the structure of the imported data is slightly different. We can check with the `str` function. The `mzXML` and `Bruker` data has similar information but is organized differently. The `Bruker` data returns an object where every element of the list has sublists, namely `spectrum` and `metaData`, with mass and intensity being the subcomponents of the spectrum slot, whereas the `mzXML` data has three slots called `mass`, `intensity`, and `metaData`. The following screenshot shows the structure of the `mzXML` imported data:


```

List of 2
 $ spectrum:List of 2
  ..$ mass      : num [1:22431] 1000 1000 1000 1001 1001 ...
  ..$ intensity: num [1:22431] 11278 11350 10879 10684 10740 ...
 $ metaData:List of 18
  ..$ file           : chr "/home/praveen/bookcode/ms/A1-0_A1.mzXML"
  ..$ scanCount      : num 1
  ..$ parentFile     :List of 1
  .. ..$ :List of 3
  .. .. ..$ fileName: chr "Z:/home/sebastian/dokumente/studium/sapdmd/dev/r-
kages/readBrukerFlexData-dev/readBrukerFlexData/inst/Examples/2010_05_19_Gi"
_truncated__
  .. .. ..$ fileType: chr "RAWData"
  .. .. ..$ fileSha1: chr "2b6113bf97a3a2c83424f71070f8de8d23bd1919"
  ..$ msInstrument   :List of 7
  .. ..$ msManufacturer: chr "Bruker Daltonics"
  .. ..$ msModel       : chr "autoFlex"
  .. ..$ msIonisation  : chr "MALDI"
  .. ..$ msMassAnalyzer: chr "TOF"
  .. ..$ msDetector    : chr "MS:1000026"
  .. ..$ software      :List of 3
  .. .. ..$ type       : chr "acquisition"

```

See also

- The CRAN page and manual at <http://cran.r-project.org/web/packages/readBrukerFlexData/readBrukerFlexData.pdf>, which provides more information on the `readBrukerFlexData` package

Converting the MS data in the mzXML format to MALDIquant

Before we start this recipe, it is important to introduce the [MALDIquant](#) R package. [MALDIquant](#) is an R package that carries many functions for a basic analysis of MS data. We will use this package for some of our upcoming recipes. Therefore, it is essential that you go through the approaches that can be used to get data in an appropriate format to be processed by the package.

Getting ready

We will need two packages for this recipe, namely [MALDIquant](#) and its auxiliary package, [MALDIquantForeign](#). Both packages are available in the CRAN repository.

How to do it...

To get data from a different format to be usable in the [MALDIquant](#) format, perform the following steps:

1. First, install the two packages mentioned earlier, as follows:

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

2. After installing the packages, load them in the R session as follows:

```
> library(MALDIquantForeign)
```

3. Start with the following [import](#) function to import a file in the desired directory in the R workspace to be used by [MALDIquant](#):

```
> myData_MALDI <- import("path/to/dir",
```

```
type="auto")# We can use"." in case the file  
is already in our working directory
```

4. Now, take a look at the class of the imported object:

```
> class(myData.MALDI [[1]])  
[1] "MassSpectrum"  
> attr(,"package")  
[1] "MALDIquant"
```

Looking at the structure of one of the components (spectra) of the imported data reveals that it is very similar to the [mzXML](#) object we created in the *Reading MS data of the mzXML/mzML format* recipe:

```
> str(myData.MALDI [[1]])
```

How it works...

The [MALDIquantForeign](#) package is an auxiliary package for the [MALDIquant](#) package. This package reads and writes different file formats of MS data into/from the [MALDIquant](#) objects. It supports both the [import](#) and [export](#) functions. In this recipe, we focused on the [import](#) function. The [import](#) function parses the input files and writes the contents to the required data format.

See also

- The CRAN page for the [MALDIquantForeign](#) package at <http://cran.r-project.org/web/packages/MALDIquantForeign/MALDIquantForeign.pdf>
- The developer's page at <http://strimmerlab.org/software/malDIquant/>

Extracting data elements from the MS data object

In the recipes discussed so far, we focused on different data formats in MS. As we have seen, MS data has different components or slots. Each slot contains a specific type of information. This recipe will present the methods that can be used to fetch specific content from the data.

Getting ready

In this recipe, we will need a dataset and the `MALDIquant` package, which have already been introduced.

How to do it...

The following steps deal with the extraction of interesting components from MS data in the `MALDIquant` format:

1. Start by loading the `MALDIquant` package in the R session as follows:

```
> library(MALDIquant)
```

2. Now, load the data with the following function. Use the data from the `MALDIquant` package that can be loaded with the following command:

```
> data("fiedler2009subset",  
      package="MALDIquant")
```

3. Take a look at the data by typing the following command:

```
> summary(fiedler2009subset)
```

4. For further analysis, we will use only the first element from the list of 16 elements for demonstration, as follows:

```
> myData <- fiedler2009subset[[1]]
```

The data is an instance of the `MassSpectrum` class, as shown in the following example:

```
> class(myData)
[1] "MassSpectrum"
> attr(,"package")
[1] "MALDIquant"
```

5. You need to know the structure of the data to extract useful elements from it. Take a look at the detailed structure by typing the following command:

```
> str(myData)
```

6. To extract the m/z ratio, extract the `mass` component as follows:

```
> m_z <- mass(myData)
```

Alternatively, you can also extract elements via a component named `index` as follows:

```
> m_z <- myData@mass
```

7. In a similar way, extract the `intensity` component as follows:

```
> myIntensity <- intensity(myData)
```

8. Again, use the name indexing to get the same result by typing the following command:

```
> myIntensity <- myData@intensity
```

9. Take a look at the experiment and the sample details for the data as follows:

```
> metaData(myData)
```

The plots we made in the previous recipes for mass and intensity can be made simply by using the individual components with the following command:

```
> plot(m_z, myIntensity, type="l",  
col="black")
```

How it works...

The recipe explained here is based on the structure of the data as mentioned earlier. The input data is from an experiment on eight subjects (four healthy control subjects and four diseased subjects) with two replicates of each. The aim of collecting the data is to study the peptide associated with pancreatic cancer. The recipe exploits the structure of the `MassSpectrum` data. Looking at the structure of the data (as seen in the following example), allows us to intuitively extract the elements of interest from the data:

```
> str(myData)
Formal class 'MassSpectrum' [package "MALDIquant"]
with 3 slots
 ..@ mass      : num [1:42388] 1000 1000 1000 1000
1000 ...
 ..@ intensity: int [1:42388] 3149 3134 3127 3131
3170 3162 3162 3152 3174 3137 ...
 ..@ metaData :List of 42
 .. ..$ byteOrder      : chr "little"
 .. ..$ number         : num 42388
 .. ..$ timeDelay      : num 19886
```

The recipes shown here can be used to tabulate the intensities in each sample as a CSV file. The table thus created consists of the `mass` data as rows and `intensities` as columns.

There's more...

The MALDI in the `MALDIquant` package stands for Matrix-Assisted Laser Desorption/Ionization technology producing MS data. It allows you to analyze large organic molecules such as proteins.

See also

- The *Matrix-assisted laser desorption/ionization time-of-flight mass spectrometry in clinical chemistry* article by Marvin and others at <http://www.sciencedirect.com/science/article/pii/S0009898103003863>, which provides more information about the MALDI technology
- The software paper on `MALDIquant` titled *MALDIquant: a versatile R package for the analysis of mass spectrometry data* by Gibb and Strimmer at <http://bioinformatics.oxfordjournals.org/content/28/17/2270.long>
- The CRAN page at <http://cran.r-project.org/package=MALDIquant>, which provides details of the functionalities in the `MALDIquant` package

Preprocessing MS data

Every time an ionized protein or peptide hits the detector, the instrument registers it. This ionized fragment has m/z , and the detectors measure the intensity of this fragment, which is called a mass spectrum. Mass spectra have several imperfections, which can complicate their interpretation. Detecting the peaks in these spectra is challenging because some peptides with low abundance may be buried by the noise, causing a high false positive rate of peak detection. The chemical, ionization, and electronic noise often results in a decreasing curve in the background of MS data, which is referred to as a baseline. The existence of a baseline produces a strong bias in peak detection. Therefore, it is desirable to normalize, smoothen, and remove the baseline before peak detection. These are some common issues that a data analyst has to deal with while working with MS data. We will deal with some of these preprocessing steps in this recipe with our focus on the MALDI technique.

Getting ready

For this recipe, we will use the [MALDIquant](#) library and the data within this package.

How to do it...

The following preprocessing steps entail the preprocessing of MS data:

1. First, load the [MALDIquant](#) package in the R session as follows:

```
> library(MALDIquant)
```

2. Now, load the data from the [MALDIquant](#) package, as we did earlier, by typing the following command:


```
> data("fiedler2009subset",  
package="MALDIquant")
```

3. Select one of these 17 spectra data for your demonstration as follows:

```
> myData <- fiedler2009subset[[1]]
```

4. Start with the transformation of the intensity in the data. The `transformIntensity` function can be used for this in the following manner:

```
> spec <- transformIntensity(myData,  
method="sqrt")
```

Other possible options for the `method` argument in the function are `log`, `log2`, and `log10`. Use this transformed intensity as the input for the next steps.

5. Next, smoothen the intensity using the `smoothIntensity` function as follows:

```
> spec <- smoothIntensity(spec,  
method="MovingAverage")
```

The function can use different smoothing functions including `SavitzkyGolay` and `MovingAverage`.

6. Now, perform the baseline correction. The `removeBaseline` function can be used for this purpose as follows:

```
> spec <- removeBaseline(spec)
```

7. Finally, normalize the data with the following `calibrateIntensity` function:

```
> spec <- calibrateIntensity(spec,
```

```
method="TIC")
```

8. Estimate the noise of MS data as follows:

```
> n <- estimateNoise(spec)
```

How it works...

Transforming the intensity improves the resolution of the data. This means that the data gets more distinguishable for classifiers and other methods. Several kinds of transformation are possible, as we have seen, including square root transformation and log transformation (to the base 2, 10, and natural logarithm). A square root transformation avoids skewness in the distribution, whereas a log transformation may result in a slightly skewed distribution; however, a significant error is rare. Therefore, the choice of transformation depends on the user. If the log transformation gives unexpectedly high skewed results, the square root transformation should be tried.

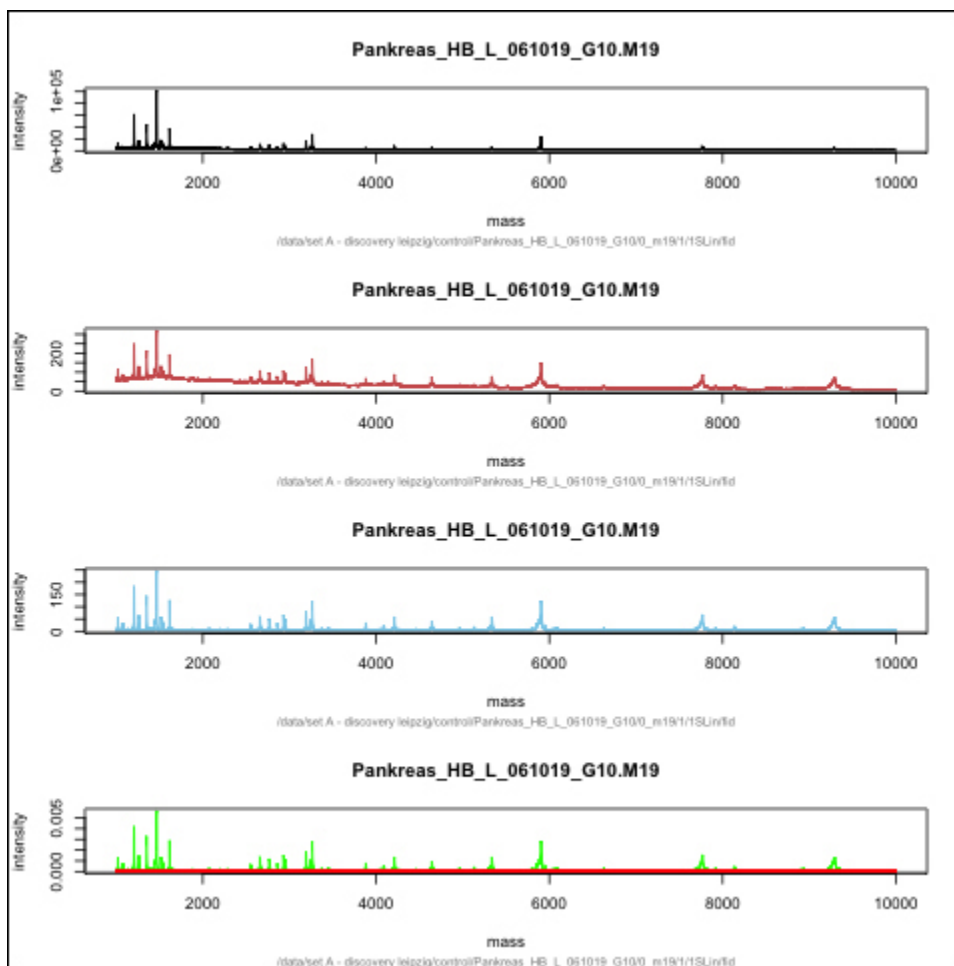
The MS data can have noises due to the experiment's design or other factors. Smoothing reduces the noise level in the whole spectrum. It can use various methods as mentioned in the *How to do it...* section of this recipe. The `MovingAverage` method runs a simple two-side moving average. The `SavitzkyGolay` smoothing uses a Savitzky-Golay filter based on partial least squares.

Baseline correction is another noise reduction method for MS data. It uses one of the SNIP, TopHat, ConvexHull, and median methods to do so. Take a look at the list of references provided in the *See also* section of this recipe to learn about the individual methods.

Normalization of the data is performed by calibration via the `calibrateIntensity` function. The implemented methods for normalization include `TIC`, `PQN`, and `median`. The `TIC` method sets the **Total Ion Current (TIC)** of the spectrum to one for normalization, whereas the `median` method sets the median of the intensities to one. The `PQN` method is a bit complicated. It computes the TIC calibration

for the data and also computes the median spectrum. Then, the quotients of all the intensities are computed with the median reference, and the median for this spectrum is computed. Finally, the intensities are divided by the median obtained in the last step to get the normalized data. We can plot the objects using the `plot(object)` function, where `object` is the spectrum computed at every step. A more detailed reference for the PQN approach is available in the *Probabilistic Quotient Normalization as Robust Method to Account for Dilution of Complex Biological Mixtures. Application in ^1H NMR Metabonomics* article by Dieterle and others at <http://pubs.acs.org/doi/abs/10.1021/ac051632c>.

The following screenshot shows the raw values, square root transformation, baseline correction, and the calibrated intensity with noise in red (starting from top to bottom):



See also

- The *Mass Spectrometry for Microbial Proteomics* book by Shah and Gharbia to learn more on the analysis and preprocessing of MS data
- The *PRE-PROCESSING MASS SPECTROMETRY DATA* article by Coombes (<http://www.unical.it/portale/strutture/dipartimenti/deis/home/tagarelli/research/CBM07.pdf>), which is another interesting article on the preprocessing of MS data
- The following two articles to learn about the Savitzky-Golay filter:
 - *Smoothing and Differentiation of Data by Simplified Least Squares Procedures* by Savitzky and Golay (<http://pubs.acs.org/doi/abs/10.1021/ac60214a047>)
 - *Application hints for Savitzky-Golay digital smoothing filters* by Ziegler and others (<http://pubs.acs.org/doi/abs/10.1021/ac00234a011>)

Peak detection in MS data

Usually, peptide signals appear as local maxima called peaks in MS spectra. These high, sharp peaks contain information about molecules with high concentrations. A peak detection procedure calculates the so-called line spectrum for a raw spectrum, which is a list of the positions and intensities of the peaks. The high sharp peaks of this spectrum have information about molecules with high concentrations. A peak detection procedure calculates a list of the positions and intensities of the peaks. Peak detection is usually a starting and crucial step of MS data analysis.

As we have already seen the preprocessing steps in the previous recipe's functions, it gets easier for us to go through the peak detection recipe.

Getting ready

We will continue using the `MALDIquant` package for peak detection. In this recipe, we will use the data from the same package for demonstration purposes.

How to do it...

To identify peaks in MS data, perform the following steps:

1. Start by loading the `MALDIquant` library as follows:

```
> library(MALDIquant)
```

2. Now, load the data as follows:

```
> data("fiedler2009subset",  
package="MALDIquant")  
> myData <- fiedler2009subset[[1]]
```

3. You need to preprocess the data before peak detection. Start by transforming the values of the intensities to their square roots by typing the following command:

```
> spec <- transformIntensity(myData,  
method="sqrt")
```

4. Next, smoothen these transformed intensities as follows:

```
> spec <- smoothIntensity(spec,  
method="SavitzkyGolay")
```

5. To avoid bias during peak detection, perform the baseline correction as follows:

```
> spec <- removeBaseline(spec)
```

6. Finally, normalize the data with the following command:

```
> spec <- calibrateIntensity(spec, method=  
"TIC")
```

7. Now, call the `detectPeaks` function to compute the peaks in the normalized data as follows:

```
> peak <- detectPeaks(spec, method="MAD",  
SNR=2)
```

8. Take a look at the structure of the object by typing the following command, and you will see a structure similar to the structure of the original data:

```
> str(peak)
```

9. To plot the peak points, first plot the intensity line graph as follows:

```
> plot(spec)
```

10. To this plot, add the peak points with the following `points` function:

```
> points(peak, col="red")
```

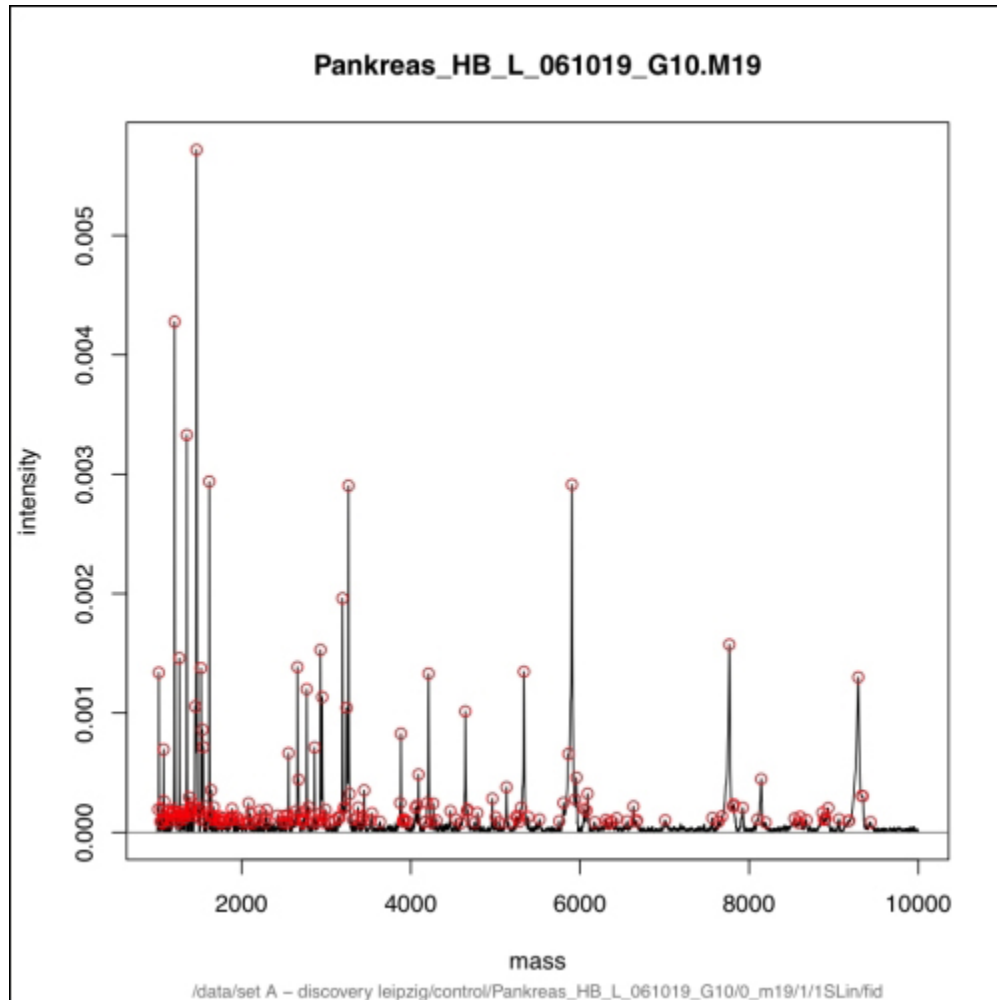
How it works...

The first few steps of the recipe show the preprocessing of data as explained in the previous recipe. The `detectPeaks` method looks for peaks in the MS data (represented by a `MassSpectrum` object). A peak is a local maximum above a user-defined noise threshold. The function estimates the noise in the data using the `method` argument of the function. It accepts two values (methods), namely `MAD` and `SuperSmoother`. Another argument is `SNR` which refers to the signal-to-noise ratio in the data. This argument needs a numeric value (the default is 2). The program computes the signal noise first. In order to identify a local maximum as a peak, the value should be higher. The following condition is required by the function:

$$peak > noise \times SNR$$

All the values where the preceding condition is met are detected as peaks.

Calculating the **median absolute deviation (MAD)** estimates the noise in the data. For the `SuperSmoother` method, it uses Friedman's super smoother approach to compute noise. We can see the peak on the mass versus intensity plot in the following figure:



Identified peaks (red circles) in m/z versus intensity plots

See also

- The *Comparison of public peak detection algorithms for MALDI mass spectrometry data analysis* article by Yang and others at <http://www.biomedcentral.com/1471-2105/10/4>, which provides the theoretical background on various peak detection algorithms
- The article titled *A VARIABLE SPAN SMOOTHER* by Friedman at <http://www.slac.stanford.edu/cgi-wrap/getdoc/slac-pub-3477.pdf>, which provides the conceptual details of the super smoother approach

Peak alignment with MS data

For the comparison of peaks across different spectra, it is essential to conduct alignment. Furthermore, instrument resolution or instrument calibration may affect the quality of the datasets because of the variance of the mass/charge ratio at any point. Therefore, the alignment of spectra within datasets is also required. This recipe will deal with the alignment of peaks in MS data.

Getting ready

We will continue using the [MALDIquant](#) package for peak detection. In this recipe, we will use the data from the same package for demonstration purposes.

How to do it...

The following steps will take us through the recipe for peak alignment in MS data:

1. Start by loading the [MALDIquant](#) library as follows:

```
> library(MALDIquant)
```

2. Now, load the data as follows:

```
> data("fiedler2009subset",  
package="MALDIquant")  
> myData <- fiedler2009subset
```

3. You need to preprocess the data before peak detection. Start by transforming the values of the intensities to their square roots as follows:

```
> spec <- transformIntensity(myData,
```

```
method="sqrt")
```

4. These transformed intensities are then smoothed as follows:

```
> spec <- smoothIntensity(spec,  
method="SavitzkyGolay")
```

5. To avoid bias during peak detection, perform the baseline correction as follows:

```
> spec <- removeBaseline(spec)
```

6. Normalize the data with the following command:

```
> spec <- calibrateIntensity(spec,  
method="TIC")
```

7. Finally, use the `alignSpectra` function to align the processed and normalized peaks as follows:

```
> spectra <- alignSpectra(spec)
```

8. Now, check the original and returned objects, which are a list of spectra, and plot the elements as follows:

```
> summary(myData)  
> summary(spectra)  
> plot(myData[[1]])  
> plot(spectra[[1]])
```

How it works...

In order to match peaks that have the same mass values, `MALDIquant` uses a statistical regression-based approach. First, landmark peaks are identified, which occur in most spectra. Subsequently, a non-linear `warping` function is computed for each spectrum by fitting a local regression to the matched reference peaks. This also allows you to merge the aligned spectra from technical replicates.

There's more...

The `alignment` method implemented in the `MALDIquant` package is a hybrid of two popular methods, `DISCO` and the self-calibrated `warping`. To know the conceptual details of the `DISCO` approach, refer to the *DISCO: Distance and Spectrum Correlation Optimization Alignment for Two-Dimensional Gas Chromatography Time-of-Flight Mass Spectrometry-Based Metabolomics* article by Wang and others (<http://pubs.acs.org/doi/abs/10.1021/ac100064b>).

The second method is explained in detail in the *Self-Calibrated Warping for Mass Spectra Alignment* article by Peter He and others (<http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3085421/>).

Peptide identification in MS data

Now, we move onto the biological inference from the MS data. Identifying peptides from MS data is one of the popular tasks in MS data analysis. Nowadays, protein identification is regularly performed by **tandem mass spectrometry (MS-MS)**. Because of the difficulty faced in measuring intact proteins using MS-MS, a protein is typically digested into peptides with enzymes and the MS-MS spectrum of each peptide is measured. Here, we present a recipe to identify peptides in MS data.

Getting ready

In this recipe, we will introduce a new R package named `protViz`. The MS data we use in this recipe is artificially created from the previous data. The peptide we attempt to detect is the human serum albumin (ALB).

How to do it...

The following steps depict how to retrieve peptide data from `SwissProt`:

1. Start by getting the suitable peptide data from `SwissProt-ExPasy` for the human albumin protein.
2. Visit web.expasy.org/peptide_mass.
3. Enter the UniProt KB identifier for albumin (ALBU_HUMAN) in the query box and click on the **Perform** button (leave the other parameter settings at default).

What we get is the result of the digestion of the protein ALBU_HUMAN by the trypsin enzyme. The following screenshot shows the ExPASy web page:


 SIB ExPASy
 Bioinformatics Resource Portal

PeptideMass

PeptideMass

The entered protein is: ALBU_HUMAN

The selected enzyme is: Trypsin

Maximum number of missed cleavages (MC): 0

All cysteines in reduced form.

Methionines have not been oxidized.

Displaying peptides with a mass bigger than 500 Dalton.

Using monoisotopic masses of the occurring amino acid residues and giving peptide masses as [M+H]⁺.

You have selected ALBU_HUMAN (P02768) from UniProtKB/Swiss-Prot:

Serum albumin precursor

Signal and propep in positions 1-22 have been removed.

- Chain Serum albumin at positions 25 - 609 [Theoretical pI: 5.67 / Mw (average mass): 66472.21 / Mw (monoisotopic mass): 66428.93]

mass	position	#MC	modifications	peptide sequence
2917.3229	311-337	0		SHCIAEVENDEMPADLPSLA ADFVESK
2593.2425	139-160	0		LVRPEVDVMCTAFHDNEETF LK
2433.2635	45-65	0		ALVLIAFAQYLQQCPFEDHV K
2404.1709	470-490	0		MPCAEDYLSVVLNQLCVLHE K
2203.0012	525-543	0		EFNAETFTFHADICTLSEK
2045.0953	397-413	0		VFDEFKPLVEEPQNLIK
1915.7731	265-281	0		VHTECCHGDLLECADDR
1853.9102	509-524	0		RPCFSALEVDETYVPK

- You will need the [protViz](#) package from CRAN for this recipe. Install and load the package in the R session as follows:

```
> install.packages("protViz"
dependencies=TRUE)
> library(protViz)
```

- Now, take only the first few peptides from this result for further analysis.
- Put this in your workspace as a vector of characters as follows:

```
> myPeptide <- "SHCIAEVENDEMPADLPSLAADFVESK"
```

- Now, create the spectrum data artificially. For your convenience, use one of the spectrum data used so far that can be loaded with the following function:

```
> myData <- fiedler2009subset[[1]]
```

- You need to preprocess the data before peak detection. Start by transforming the values of the intensities to their square roots as

follows:

```
> spec = transformIntensity(myData,  
method="sqrt")
```

9. These transformed intensities are then smoothed as follows:

```
> spec = smoothIntensity(spec,  
method="SavitzkyGolay")
```

10. To avoid bias during peak detection, perform the baseline correction as follows:

```
> spec = removeBaseline(spec)
```

11. Finally, normalize the data with the following commands:

```
> spec = calibrateIntensity(spec,  
method="TIC")
```

12. Create the artificial spectra data object in the form of a list as follows:

```
> mySpec <- list(title="artificial", charge=2,  
mZ=spec@mass, intensity=spec@intensity)
```

13. Once you have the data to be compared ready, get the number of characters in your peptide using the `nchar` function as follows:

```
> n <- nchar(myPeptide)
```

14. Now, compute the fragment ions of the peptide sequence by typing the following command:

```
> fi <- fragmentIons(myPeptide)
```

15. Extract the ions of your choice (here, the b and y ions) as follows:

```
> by_mZ <- c(fi[[1]]$b, fi[[1]]$y)
```

16. Now create a label vector for the ions by typing the following command:

```
> by_label <- c(paste("b", 1:n,  
  sep=""), paste("y", 1:n, sep=""))
```

17. Then, find the index of the nearest neighbors of the ion data in the spectra as follows:

```
> idx <- findNN(by_mZ, mySpec$mZ)
```

18. Finally, compute the error in the m/z matches, which was calculated in the previous steps, as follows:

```
> mz_error <- abs(mySpec$mZ[idx]-by_mZ)
```

19. To plot this error, use the generic `plot` functions as follows:

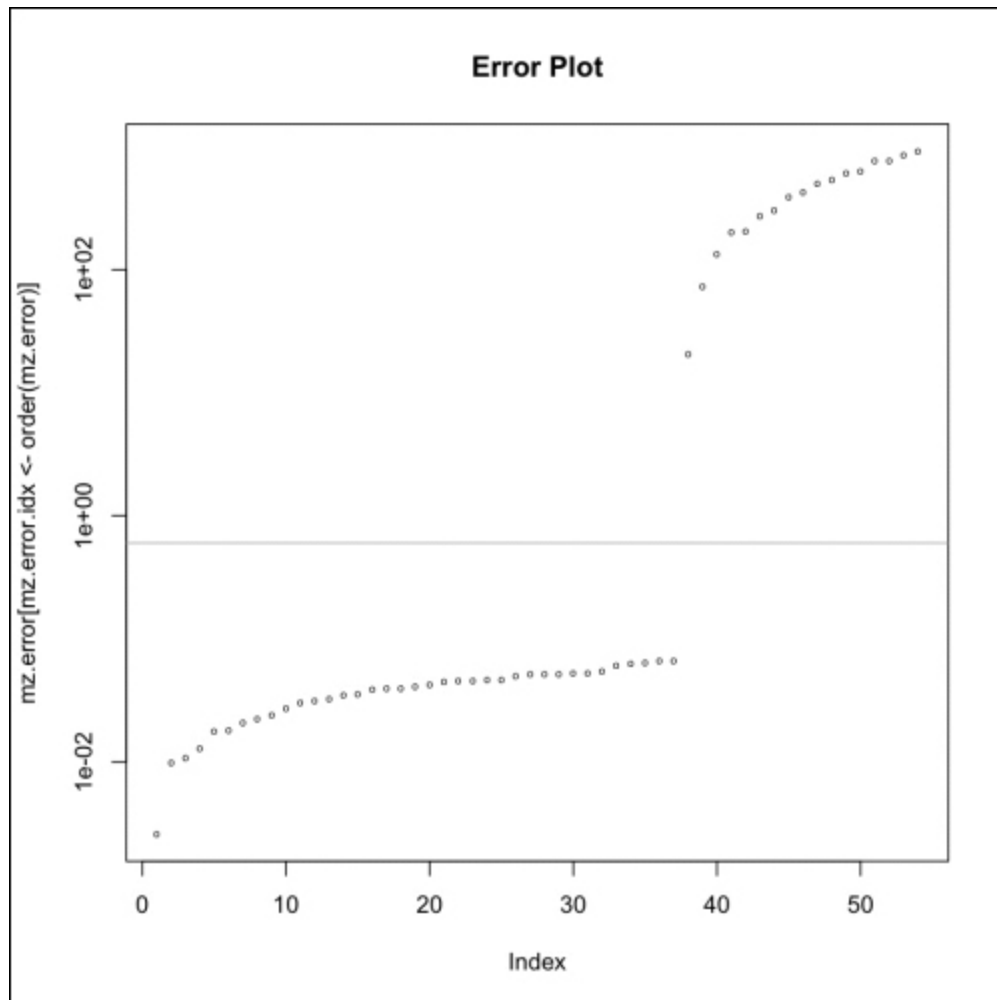
```
> plot(mz_error[mz_error.idx<-  
  order(mz_error)], main="Error Plot", pch='o',  
  cex=0.5, log='y')  
> abline(h=0.6, col="grey")
```

How it works...

A peptide sequence is detected in the spectra via fragment ion matching. A hit is considered if this error is lower than a threshold called `fragmentIonError`. In order to compute the error, we use the ions in the peptide sequence using the `fragmentIons` function. We use m/z computed for these ions to be mapped onto the spectra with the nearest match using the `findNN` function. We can choose the ions that should be mapped onto the spectra data (we used the b and y ions here). If the difference between the computed mass and measured mass (in MS data) is lower than the threshold, it is termed as a hit. A higher hit rate and low error rate increases the confidence

in the presence of the peptide under investigation. We can see the high error rate in our plot here.

The plot shows the error of assignment between the MS data and the singly-charged fragment ions of the `myPeptide` object. The more the theoretical ions match the actually measured ion series, the more likely it is that the measured spectrum is indeed from the inferred peptide, and thus, the protein is identified. The following screenshot shows an error plot for peptide detection:



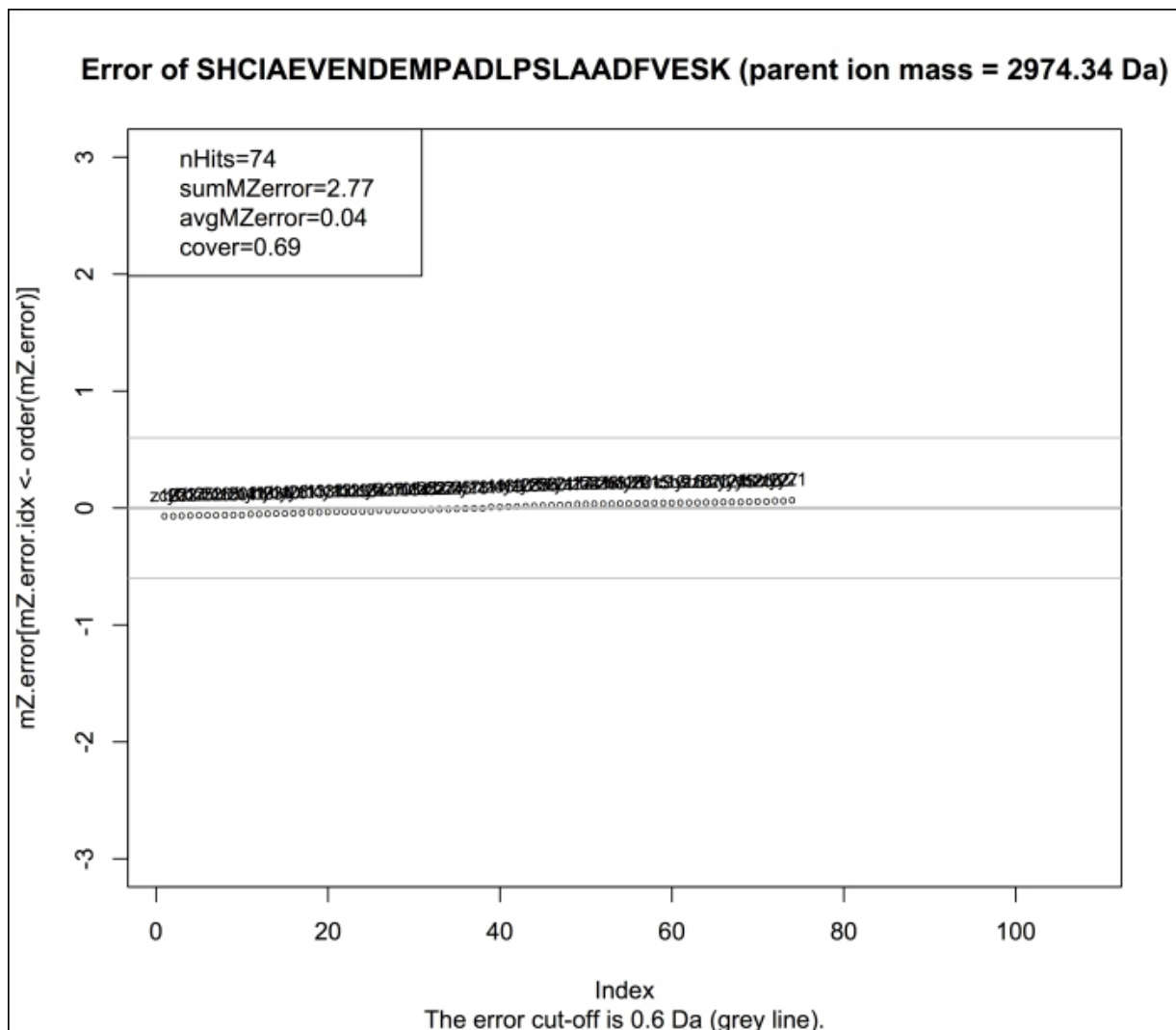
There's more...

The `psm` function is a generalized function to compute the error. Nevertheless, it is advisable to use the recipe described here in order to compute the match between the spectrum and the peptide.

This is because it provides higher resolution flexibility and resolution. The `psm` function can be used in the following way:

```
> match <- psm(myPeptide, mySpec)
```

We can compare the two plots, and the resolution in the analysis is evident. The following screenshot shows an error plot obtained using the `psm` function for the `fiedler2009subset` data (first element):



The ion types in the peptide fragment follow a nomenclature system that is referred to as b, y, c, and z. To find out more about this nomenclature, refer to the *Proposal for a common nomenclature for*

sequence ions in mass spectra of peptides article by Roepstorff and Fohlman.

See also

- The *Protein Sequencing and Identification Using Tandem Mass Spectrometry* article by Kinter and Nicholas, which gives more information on the identification of peptides in MS data
- The CRAN page <http://cran.r-project.org/web/packages/protViz/index.html>, which provides more information on the `protViz` package and also has links to the user manual and the vignette files

Performing protein quantification analysis

So far, we spoke in terms of the qualitative use of MS data. However, proteomic analysis has moved from the qualitative to the quantitative approach, and the MS technique has played a key role in this move. This recipe will focus on the quantification aspects of MS data analysis.

Getting ready

We will continue to use the `protViz` library for this recipe. The inbuilt data within this package will serve as the demo input for us.

How to do it...

To perform the MS-data-based quantification, complete the following steps:

1. Start by loading the `protViz` library in the R session as follows:

```
> library(protViz)
```

2. Then, load the data by typing the following command:

```
> data(pgLFQfeature)
```

3. Take a look at the structure of the `pgLFQfeature` object to understand its features and contents by typing the following command:

```
> str(pgLFQfeature)
```

You can see that the data contains abundant m/z values and protein information.

4. To compute the volume of each protein associated with the features in the data, run the `pgLFQtnpq` function as follows:

```
myAbundance <-  
pgLFQtnpq(QuantitativeValue=pgLFQfeature$"Normalized abundance",  
peptide=pgLFQfeature$peptideInfo$Sequence,  
protein=pgLFQfeature$peptideInfo$Protein,N=2,  
plot=FALSE)
```

The returned object, `myAbundance`, is a matrix that gives the details of the abundant proteins in every sample, as follows:

```
> class(myAbundance)  
> head(myAbundance)
```

The following screenshot shows the first six rows and eight columns from the head for the `myAbundance` object:

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]
ZZ_FGCZCont0179	63903627	59534454	50423729	70165609	51562584	77970377	48701488	65112693
P06899	2436975	3692945	2494733	3841880	2488065	3833229	2572906	3275921
P04406	31252475	31000263	27021699	31399587	25355491	27057299	22794612	26195901
P60709	67652202	69624753	43747487	78348612	51937499	79087413	39542716	42958682
P63261	76490709	64628409	47910815	80188664	44058618	59892262	42843683	51180369
P68104	15832621	16368242	14930934	16116918	14694761	15813410	17859790	19490708

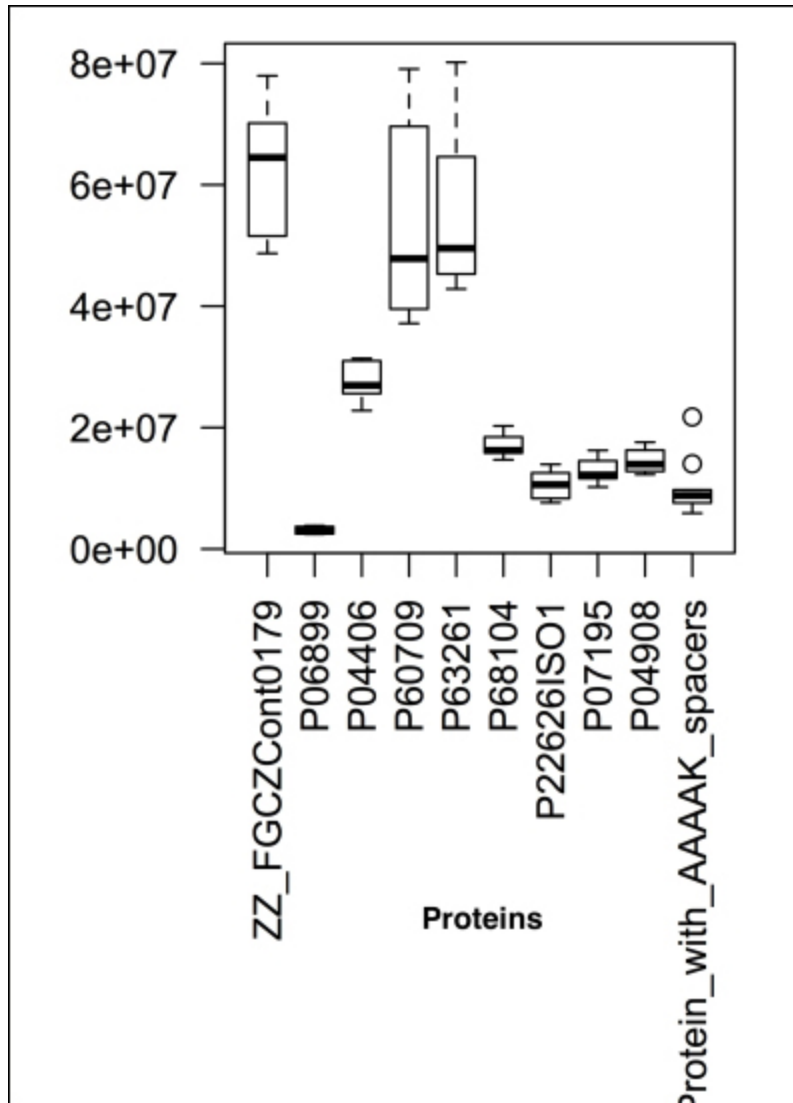
5. Create a boxplot for the abundances as follows (for the first 10 proteins only):

```
> boxplot(t(myAbundance)[1:10,1:10], xlab="",  
ylab="value", las=2)
```

How it works...

The data contains the digested proteins from human HeLa cells infected with Shigella bacteria grown during a certain period of time.

It is structured in such a way that six biological replicates from four conditions (Not_infected, Infected_1hr, Infected_2hr, and Infected_3hr) are measured; thus, we have 24 samples in total. In this recipe, we used only the second sample for demonstration. The following screenshot shows the boxplot with the abundance protein (y axis) for only the first 10 peptides along the x axis for better visualization:



The abundance analysis plots

The function uses a TopN strategy, which uses only the top n intense features to calculate the protein volume. This approach should reveal a quantitative protein value, which should make the protein

itself comparable within one condition. This allows you to estimate the protein stoichiometry and simplifies the modeling and calculations with copy numbers per cell. The Boolean argument plot creates a matrix plot for the proteins when set to `TRUE` together with the matrix computation.

See also

- The *Implementation and evaluation of relative and absolute quantification in shotgun proteomics with label-free methods* article by Grossman and others (<http://www.sciencedirect.com/science/article/pii/S1874391910001703>), which provides information about the datasets used in this recipe and also explains the TopN strategy to detect protein abundance

Performing multiple groups' analysis in MS data

Another important analysis for MS (or proteomics) data from a biological point of view is the comparison between two groups. The groups can be samples under two different conditions such as treatment and control. The comparison will aim to find whether certain proteins or molecules have significantly different abundance in one of the groups. This recipe will entail the approach to deal with such cases.

Getting ready

We do not need any specific packages for the analysis, but for the data, we will use the **Isobaric Tags for Relative and Absolute Quantitation (iTRAQ)** data from [protViz](#).

How to do it...

To perform the analysis, carry out the following steps:

1. We start by loading the [protViz](#) library to get the data as follows:

```
> library(protViz)
```

2. Now, load the [iTRAQ](#) data that you will use for the analysis as follows:

```
> data(iTRAQ)
```

3. Take a look at the data as follows:

```
> head(iTRAQ)
```

4. To look at the contents of the data (11 columns), check the columns by typing the following command (note that columns 3 to 10 have the data values and columns 1, 2, and 11 have protein information):

```
> colnames(iTRAQ)
```

5. The data has two groups under different conditions. Create empty frames for these two conditions as follows:

```
> condition_1 <- numeric()
> condition_2 <- numeric()
```

6. Columns 3 to 6 in the data belong to one group, and 7 to 10 belong to the other group. Fill the two frames created in the preceding step with values from the corresponding columns as follows:

```
for (i in c(3:6)){
  condition_1 = cbind(condition_1,
    asinh(tapply(iTRAQ[,i], paste(iTRAQ$prot),
      sum, na.rm=TRUE)))
}
for (i in 7:10){
  condition_2 = cbind(condition_2,
    asinh(tapply(iTRAQ[,i], paste(iTRAQ$prot),
      sum, na.rm=TRUE)))
}
```

Now, we have the data for five proteins for two groups with four samples in each group.

7. Now, in order to check the significance of the difference in the two groups, perform a t-test. For this, first create an empty vector where you will store the test score for each test as follows:

```
> pv <- c()
```


8. Now, compute the t-test for each protein and append the p-value to the vector created in the last step as follows:

```
for(i in 1:nrow(condition_2)){  
  pv = c(pv,  
    t.test(as.numeric(condition_1[i,]),  
    as.numeric(condition_2[i,]))$p.value)  
}
```

The p-values are the test scores that define whether the difference in abundance of the peptides between the two groups is significant or not. Conventionally, use a p-value cut-off of 0.05. You can see two proteins, O95445 and P02652, showing a significant difference with the p-values ~0.032 and 0.041. You can go for a multiple testing correction if required.

How it works...

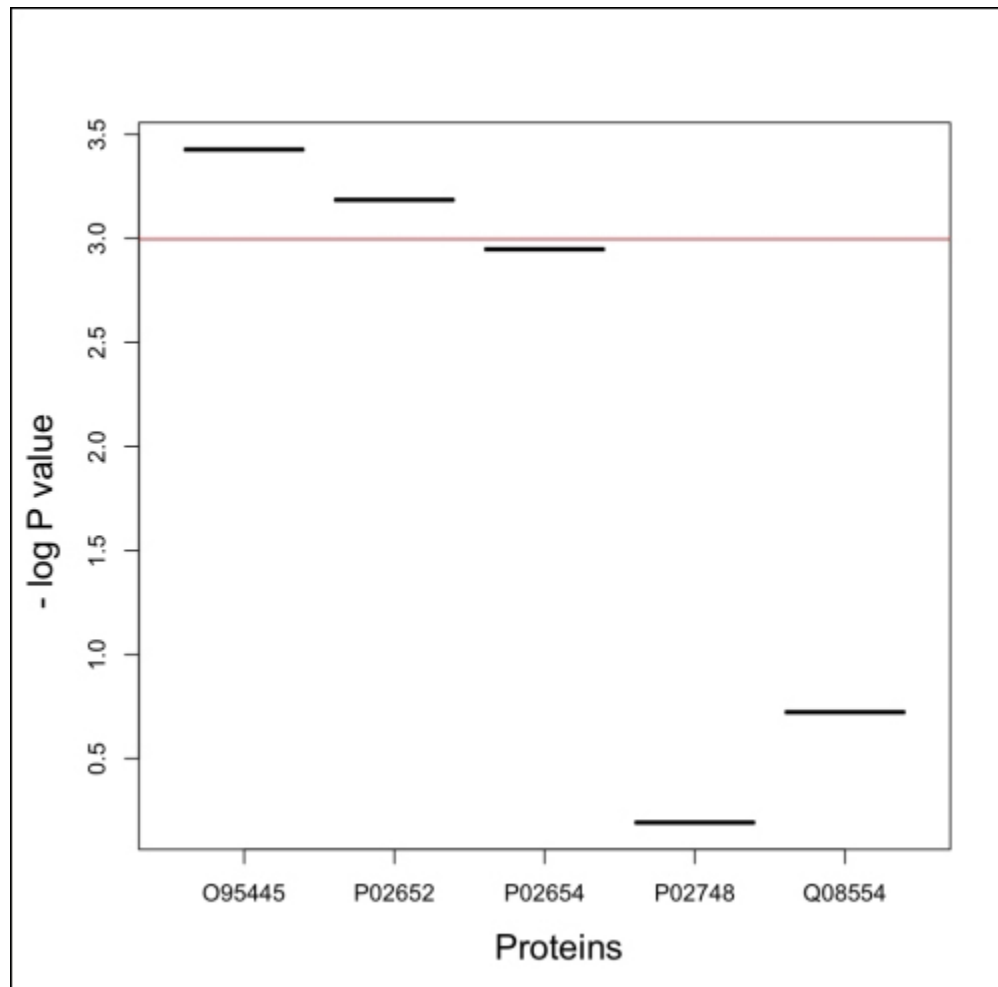
The `iTRAQ` data used in this recipe comes from two groups, which are disease and control, for Rheumatoid arthritis. The data has the peptides for five identified proteins. First, we compute the measurement for every protein by summing the corresponding peptides in each sample. This yields our two data frames, namely `condition_1` and `condition_2`. This is followed by the computation of the t-test scores. The test returns the p-value for the difference in measurement of every protein under two different conditions. If the returned p-value for a protein is less than 0.05, we accept the difference in measurement to be significant.

We can create a plot to show this. Here, we use the negative of the logarithm of the p-value for better visualization. The two proteins are the Apolipoproteins (check the UniProt IDs at the UniProt site, <http://www.uniprot.org>).

```
> names(pv) <- levels(iTRAQ$prot)  
> plot(x = factor(names(pv)), y = -log10(pv), xlab
```

```
= "Proteins",ylab = "- log P value", )
> abline(h = -log10(0.05), col = "red")
```

In the following screenshot, the red line indicates the p-value threshold of 0.05, and the x axis indicates the proteins (UniProt IDs):



Negative of the log p-values for the two groups of proteomic data analysis

See also

- The *Discovery of serum proteomic biomarkers for prediction of response to infliximab (a monoclonal anti-TNF antibody) treatment in rheumatoid arthritis: An exploratory analysis* article by Ortea and others at <http://www.sciencedirect.com/science/article/pii/S1874391912006550>, which provides more information about the dataset

Useful visualizations for MS data analysis

In this recipe, we will try out some interesting visualizations for MS data. The visualizations we will look at include peak labeling visualization, multiple spectra plotting, and fragment ion visualization.

Getting ready

We will use the same data and libraries that we have already used in our previous recipes.

How to do it...

To create some useful visualizations for MS or proteomic data analysis, perform the following steps:

1. First, start by labeling the peaks in your MS datasets. Here, you will label the top 10 spectra in your plot with their masses.
2. Load the `MALDIquant` library and the `fiedler2009subset` data as follows:

```
> library("MALDIquant")
> data("fiedler2009subset", package =
"MALDIquant")
```

3. Use only the first dataset from the entire set for demonstration purposes, as shown in the following example:

```
> spec <- fiedler2009subset[[1]]
```

4. Then, do the preprocessing of the data as described in the corresponding recipe by typing the following commands:

```
> spec <- transformIntensity(spec, method =  
"sqrt")  
> spec <- smoothIntensity(spec, method =  
"MovingAverage")  
> spec <- removeBaseline(spec)
```

5. Now, plot the spectra as follows:

```
> plot(spec)
```

6. Then, run the peak detection in the spectra as follows:

```
> p <- detectPeaks(spec)
```

7. Add these peaks as points in the plot by typing the following command:

```
> points(p)
```

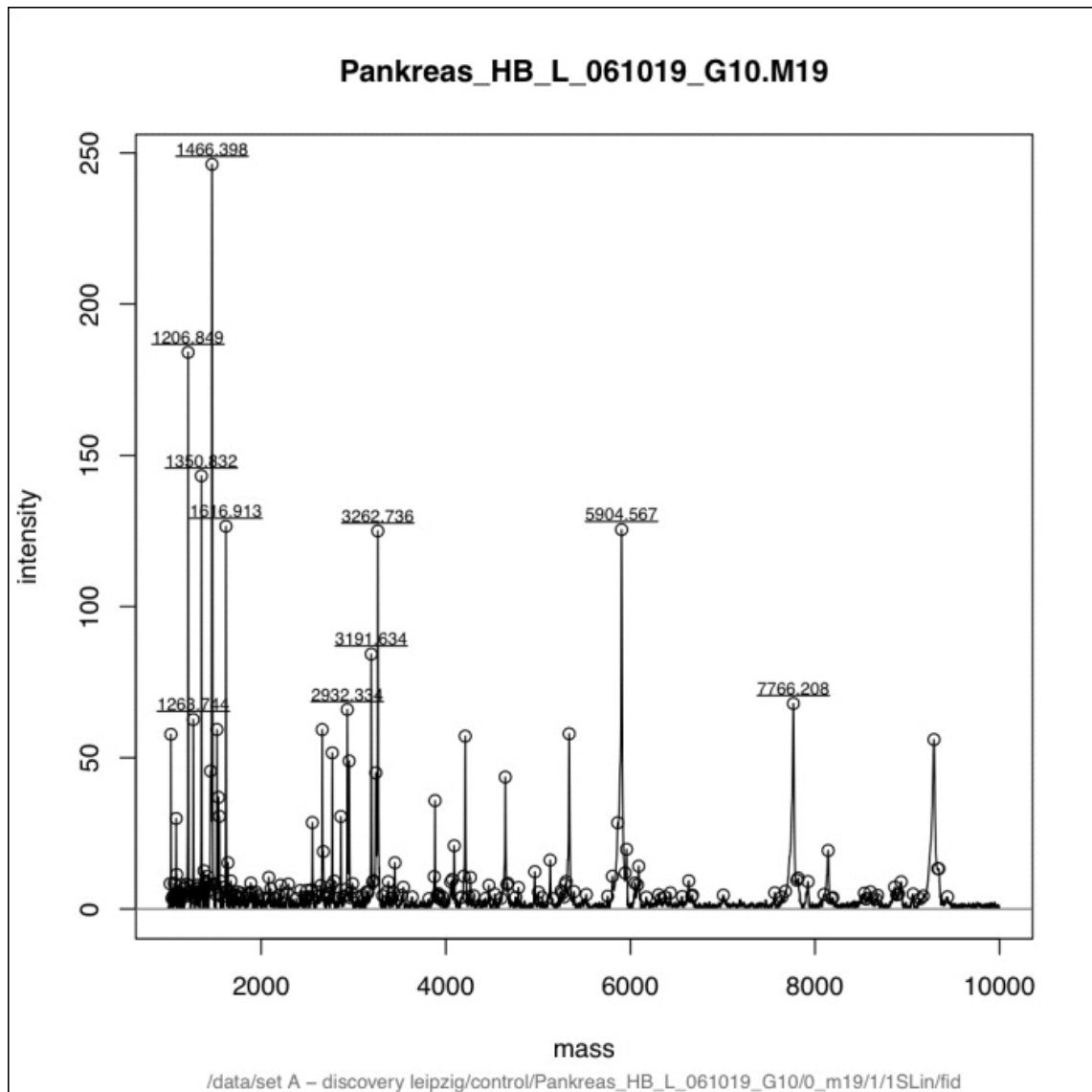
8. As you need to label the top 10 spectra in this plot, you first need to identify them by sorting them as follows:

```
> top10 <- intensity(p) %in%  
sort(intensity(p), decreasing = TRUE)[1:10]
```

9. Finally, run the `labelPeaks` function in the following way to add labels in the plot:

```
> labelPeaks(p, index = top10)
```

The following screenshot shows the visualization with the labeled peaks:



10. Our next visualization is about plotting different spectra on the same plot, which can be used specially to show alignment.
11. Again, use the `fiedler2009subset` data and assign it to the `spectra` variable as follows:

```
> spectra <- fiedler2009subset
```

12. Fetch the length of the data which is in the form of a list as follows:

```
> l <- length(spectra)
```

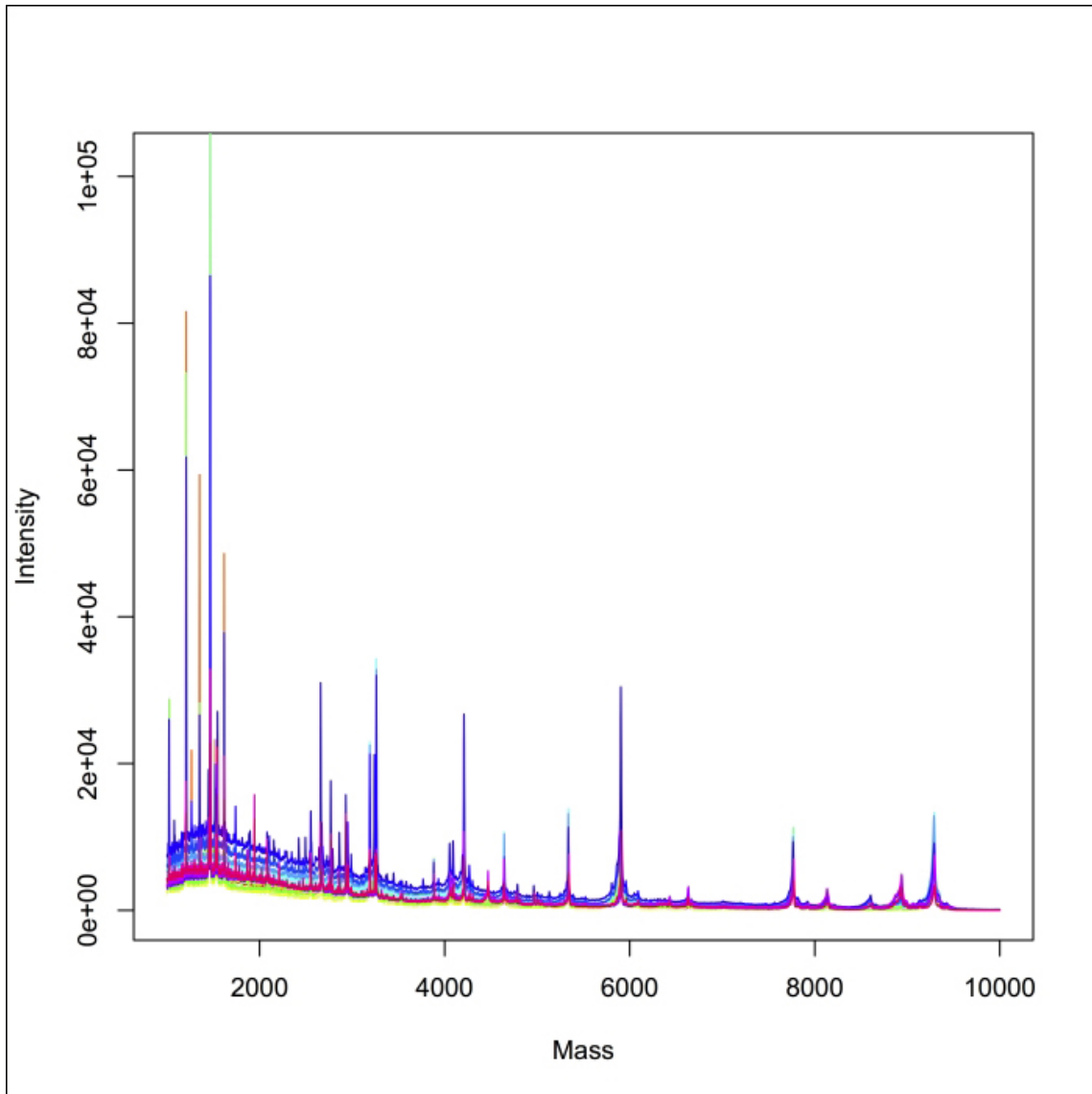
13. Initiate a plot for the first element of the list as follows (set `type = "n"`):

```
> plot(spectra[[1]]@mass,  
       spectra[[1]]@intensity, type = "n", , xlab =  
       "Mass", ylab = "Intensity")
```

14. Add the lines for all the other spectra data in the list via a loop in different colors as follows (try `?rainbow` in R):

```
> for(i in 1:l){  
  lines(spectra[[i]]@mass,  
        spectra[[i]]@intensity, type = "l", col =  
        rainbow(l)[i])  
}
```

In the following screenshot, you can see multiple spectra intensities plotted together with different colors:



15. The next plot is to visualize the m/z for the different ions in a peptide.
16. Let your peptide be in the format as shown in the following command (taken from the query to [ExPasy](#) for [ALBU_HUMAN](#)):

```
> myPeptide <-  
c("SHCIAEVENDEMPADLPSLAADFVESK",  
  "LVRPEVDVMCTAFHDNEETFLK",  
  "ALVLIAFAQYLQQCPFEDHVK")
```

17. Compute the parent ion mass for these peptides as follows:

```
> pim <- parentIonMass(myPeptide)
```

18. Also, compute the fragment ions for our peptides. This can be done with the following command:

```
> fi <- fragmentIons(myPeptide)
```

For each peptide in your input, you get the b, y, c, and z ions.

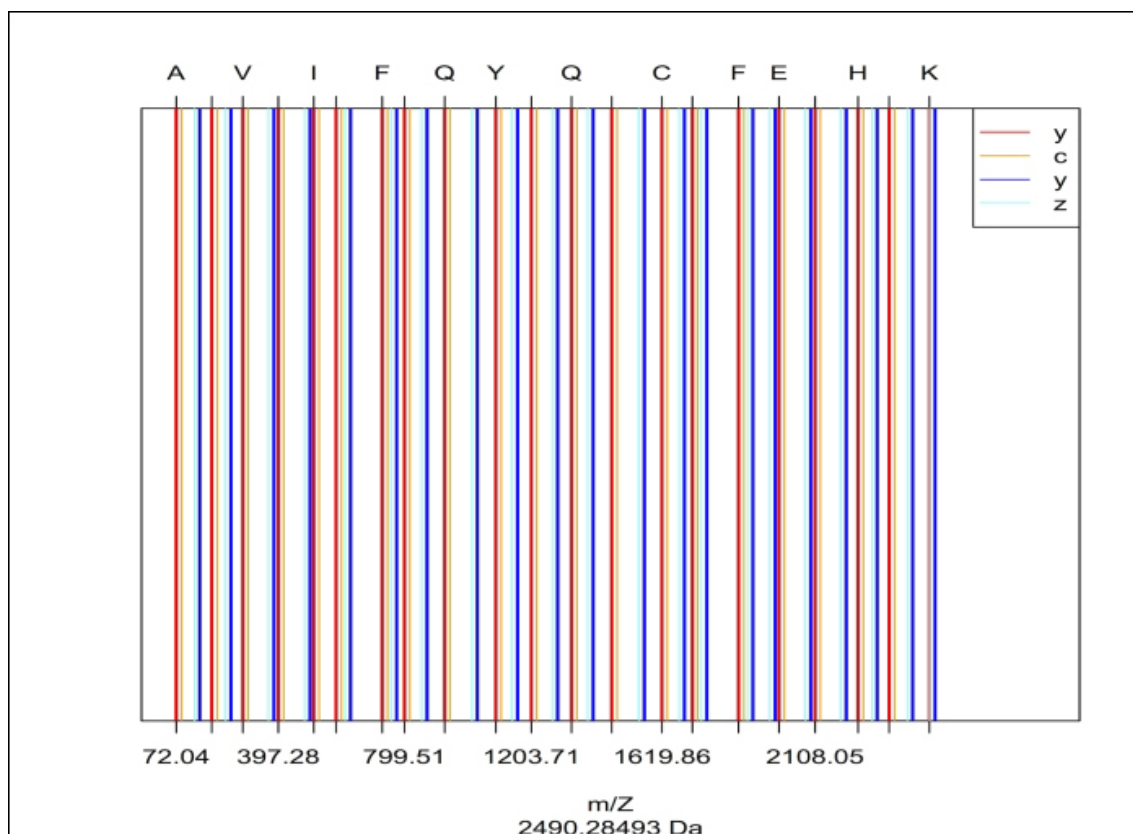
19. We can visualize m/z for each of these ion types in a different color iteratively for each peptide as follows:

```
for (i in 1:length(myPeptide)){
  plot(0,0, xlab = 'm/Z', ylab = '', xlim =
    c(min(c(fi[i][[1]]$b,fi[i][[1]]$y)),
      max(c(fi[i][[1]]$b,fi[i][[1]]$y))+350), ylim =
    c(0,1), type = 'n', axes = FALSE, sub = paste(
    pim[i], "Da")) # creates the plot area
  box() # adds a box to the plot area
  axis(1,fi[i][[1]]$b,round(fi[i][[1]]$b,2))
  # add axis label
  pepSeq<-strsplit(myPeptide[i], "") # splits the
  peptide residues
  axis(3,fi[i][[1]]$b,pepSeq[[1]]) # adds
  residue labels at top
  abline(v = fi[i][[1]]$b, col='red',lwd=2) #
  adds plot for the b type ions
  abline(v = fi[i][[1]]$c, col='orange') # adds
  plot for the c type ions
  abline(v = fi[i][[1]]$y, col='blue',lwd=2) #
  adds plot for the y type ions
  abline(v = fi[i][[1]]$z, col='cyan') # adds
  plot for the z type ions
}
```

20. Now, add a legend to your plot as follows:

```
> legend("topright", legend = c("y","c","y",
  "z"), col = c("red","orange","blue","cyan"),
  lty = 1, merge = TRUE)
```


The following screenshot shows the m/z for different types of ions (in different colors) for the first peptide, SHCIAEVENDEMPADLPSLAADFVESK:



How it works...

The first plot we did was the labeling of the peak. The peaks with the top 10 m/z values are labeled accordingly. Our plot used the `labelPeaks` function of the `MALDIquant` package. The `protViz` package also has a `peakplot` function to do the labeling. To find out more, look at the help file for `protViz` mentioned in the previous recipes.

The second plot simply exploits the generic plotting function lines of R iterated via a for loop. At every iteration, one dataset is added to the plot. It can be used to show a spectrum after and before the alignment for instances or peaks of two different conditions.

The third and final visualization presented in the recipe shows the location of the fragmented ions of a peptide in the m/z space. It

shows the different types of ions in different colors.

Chapter 8. Analyzing NGS Data

In this chapter, we will cover the following topics:

- Querying the SRA database
- Downloading data from the SRA database
- Reading FASTQ files in R
- Reading alignment data
- Preprocessing the raw NGS data
- Analyzing RNAseq data with the edgeR package
- The differential analysis of NGS data using limma
- Enriching RNAseq data with GO terms
- The KEGG enrichment of sequence data
- Analyzing methylation data
- Analyzing ChipSeq data
- Visualizations for NGS data

Introduction

HGP aimed to determine the sequences that make up human DNA. It was completed in 2003, ahead of schedule and within the budget. However, it did not end there. The project not only seeded the many sequencing projects, but also encouraged the development of technologies that could enable faster and economical sequencing of genomes. A single human genome is not enough to understand genetic information on health; rather, many genomes are required for such studies. The technologies used during the HGP were slow and expensive. The demand for cheaper and faster sequencing methods has driven the development of **Next Generation Sequencing (NGS)**. The NGS platforms perform massively parallel sequencing, where millions of fragments of DNA from a single sample are sequenced in parallel, facilitating high-throughput sequencing. This allows an entire genome to be sequenced faster and at a lower cost.

NGS is a term used to describe a number of different modern sequencing technologies. It includes some popular technologies, explored as follows:

- Illumina (Solexa) sequencing
- Roche 454 sequencing
- Ion torrent (proton and PGM sequencing)
- SOLiD sequencing

These technologies allow faster and cheaper sequencing of DNA and RNA than the previously used Sanger sequencing, and as such have revolutionized the study of genomics and molecular biology. This chapter will deal with the results obtained from these technologies and some related technologies. To know more on NGS technologies and their application, you can refer to Metzker's *Sequencing technologies-the next generation* (<http://www.nature.com/nrg/journal/v11/n1/full/nrg2626.html>).

One of the most popular formats of sequence data in NGS is the FASTQ format. Before we go through the NGS data analysis in detail, it would be interesting to look at the FASTQ data format. The FASTQ data format consists of four lines. The first line is for the sequence name, the second is for the sequence itself, the third is for optional information about the sequence, and the fourth is for a confidence or accuracy measurement of bases. The following screenshot shows an instance of the FASTQ data format:

```
@ERR056989.2 GRJP5WI01AODNT/2
GCGAAGTAGCATGAGCAGGACGCGATGACGAGCAGCAGGAGCATGACCATGAGCGTCTGCGCGGCAGCGC
+
:9;00012333358995../07;=;;;=?@@@@@??<<=&??; ;@@@BB@@??==@?;511111371
```

As all the other information in the FASTQ format is rather simple and should be easily understandable, let's discuss the quality scores. The data quality in NGS is measured in terms of a metrics called the Phred score. A Phred score is assigned to each base during the sequencing process; therefore, we have a corresponding character

for each base in FASTQ data. Mathematically, the Phred score (Q) is defined as follows:

$$Q = -10 \log_{10} P$$

In the preceding formula, **P** is the estimated error probability in base call (process of assigning bases to peaks). This establishes a logarithmic relationship between the quality and base calling error, which allows you to work with very small errors (close to zero) and deal in high accuracies numerically. Thus, 99.999 percent (1 in 100,000) accuracy in base calling yields a score equal to 50. To know more about the Phred score, refer to *Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities* by Ewing and Green (<http://genome.cshlp.org/content/8/3/186.long>).

The FASTQ format displays this quality measure in terms of ASCII characters (usually, the (33+Q)th ASCII character is used to represent a Q value). That's why, we see such characters throughout every fourth line of every FASTQ file.

The FASTQ data allows the storage of the sequence and quality information for each read in a compact text-based format. To know more about the FASTQ format, refer to *The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants* article by Cock and others (<http://nar.oxfordjournals.org/content/38/6/1767.full>).

Querying the SRA database

Sequence Read Archive (SRA) is a database that contains DNA sequencing data in terms of short reads generated by high-throughput sequencing obtained via different platforms. The sequences are usually less than 1,000 base pairs in length. The database is available at <http://www.ncbi.nlm.nih.gov/sra>. This recipe will introduce steps to query and access the data residing in the SRA database from within R.

Getting ready

As it is a querying recipe, we will need our machine to be connected to the Web, and we will need the queries we want to perform. The packages to be used will be introduced as we need them.

How to do it...

The following steps will help us run the desired queries on the SRA database:

1. First, install and load the `SRADB` library in your R session as follows:

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

2. Now, fetch the metadata file from the SRA server as follows (note that this might take a while; therefore, it may be a good idea to grab a coffee now):

```
> sqlFile <- getSRADBFile()
```

3. Create a connection to the `SQLite` file for querying, as follows:

```
> sraCon <- dbConnect(SQLite(), sqlFile)
```

4. There are a few things you can do to investigate the content of the database. This will help you to define the queries that should be run later on. Use the following function to determine the fields in the data:

```
> sraTables <- dbListTables(sraCon)
> dbListFields(sraCon, "study")
```

5. Now, frame your first query to get the accessions and titles from the SRA study where, the study type contains the keyword `embryo`. The query can be seen as a typical SQL query:

```
> myHit <- dbGetQuery(sraCon, paste("select
study_accession, study_title from study
where", "study_description like '%embryo'", sep="
"))
```

You will see that the query returns one hit, as follows:

```
> myHit
  study_accession  study_title
1 SRP001353 GSE17621: RNA-Seq of Drosophila
developmental stage 16-18 hr
```

6. To do a free text search for the terms of interest, use the following functions:

```
> myHit <- getSRA( search_terms = "brain",
out_types = c('run', 'study'), sraCon)
> head(myHit)
  run_alias  run      run_date
updated_date  spots
1 R00GC_ DRR000341 <NA> 2013-07-27 66057520
2 R00GA_R01GA DRR000339 <NA> 2013-07-27
11595633
3 R00GB_R01GB DRR000340 <NA> 2013-07-27
7684664
```

```

4 R01GC DRR000342 <NA> 2013-07-27 36153241
5 riken_grau-0002_Run_0002 DRR000834 2010-01-
12 2013-07-28 157457
6 bsi_hirase-0001_Run_0001 DRR000996 2010-05-
24 2013-07-28 18838171

```

7. Combine key words for appropriate logic as follows:

```

> myHit <- getSRA( search_terms = 'Alzheimers
OR "EPILEPSY"', out_types = c('sample'),
sraCon)

```

8. Take a look at some of the retrieved results by typing the following command:

```

> head(myHit)
sample_alias      sample      taxon_id common_name
anonymized_name individual_name
1      cec411      ERS354366   9606   human   <NA>
<NA>
2      GSM803420 SRS266589   9606   <NA>   <NA>
<NA>
3      455-325    SRS348506   9606   <NA>   <NA>
<NA>
4      455-4923   SRS348502   9606   <NA>   <NA>
<NA>
5      <NA>       SRS299958   NA     <NA>
<NA> <NA>
6      455-326    SRS348505   9606   <NA>
<NA> <NA>

```

How it works...

The `SRADB` package makes accessibility to the metadata associated with submission, study, sample, experiment, and run, much easier and faster (the downloaded `SQLite` database in the second step of the recipe). The `SRADB` package works via this metadata from the NCBI SRA database. The `dbConnect` function first connects the R system to these local database systems, and all our queries are locally processed based on the data contained there. In this recipe, the queries we tried with the `dbGetQuery` function are passed in the form of SQL queries, which is a `Select From Where` framework. This

part actually requires the `RSQLite` package, which is installed when installing the `SRADB` package, as a dependency. The `getSRA` function can actually do a full text search in the SRA data again via `RSQLite` and fetch the data in the selected fields for the query.

See also

- The SRA handbook from NCBI (<http://www.ncbi.nlm.nih.gov/books/NBK47528/>), which provides detailed information on the SRA database at the Entrez framework
- The *SRADB: query and use public next-generation sequencing data from within R* article by Zhu and others at <http://www.biomedcentral.com/1471-2105/14/19>, which provides information on the `SRADB` package

Downloading data from the SRA database

We have just explored how to query the SRA database. However, we can use the [SRADB](#) package to download the FASTQ data based on our queries. This recipe aims to address this issue.

Getting ready

For this recipe, we will need the queries we want to search for or the ID of the data we want to retrieve.

How to do it...

The following steps should be performed to download data from the SRA database:

1. First, load the [SRADB](#) library as follows:

```
> library(SRADB)
```

2. Now, frame a query. For instance, use the same query as in the previous recipe to search for [ALZHEIMERS](#) or [EPILEPSY](#), as shown in the following command (note that the step requires the [sraCon](#) object created in the previous recipe):

```
> myHit <- getSRA( search_terms = 'ALZHEIMERS  
OR "EPILEPSY"', out_types = c('sample'),  
sraCon)
```

3. Now, choose two of the hits and find their accessions (IDs) using the following [sraConvert](#) function:

```
> conversion <- sraConvert(
```

```
c('ERS354366','SRS266589'), sra_con = sraCon)
```

4. Take a look at the object created, as follows:

```
> conversion
      sample submission      study experiment
run
1 ERS354366 ERA252779 ERP003987 ERX324104
ERR351268
2 SRS266589 SRA046961 SRP008797 SRX100465
SRR351673
3 SRS266589 SRA046961 SRP008797 SRX100465
SRR351672
```

5. To get information on one of the experiments, use the `getSRAinfo` function:

```
> rs <- getSRAinfo( c("SRX100465"), sraCon,
sraType = "sra")
```

6. This gives us the FTP addresses and other experiment details for the search as follows:

```
> rs
$ ftp
```

- `ftp://ftp-trace.ncbi.nlm.nih.gov/sra/sra-instant/reads/ByExp/sra/SRX/SRX100/SRX100465/SRR351672/SRR351672.sra`
- `ftp://ftp-trace.ncbi.nlm.nih.gov/sra/sra-instant/reads/ByExp/sra/SRX/SRX100/SRX100465/SRR351673/SRR351673.sra`

```
      experiment      study      sample      run
size(KB)      date
1 SRX100465 SRP008797 SRS266589 SRR351672
831796 Jan 21 2012
2 SRX100465 SRP008797 SRS266589 SRR351673
1147321 Jan 21 2012
```

7. To download the data of your interest (the run), use the `getSRAfile` function as follows (note that this requires some time to download):

```
> getSRAfile( c("SRR351672", "SRR351673"),  
sraCon, fileType ='fastq')
```

How it works...

This recipe is an extension of the previous recipe. The step where the recipe deviates is the conversion of the IDs from the results of the query for free text search. This gives us the accession IDs for the hits retrieved from SRA. Then, we collect the downloaded information on the data. Finally, the `getSRAfile` function fetches the file from the SRA database. This part of the recipe actually downloads the data from the remote host. We can also use the FTP address directly to download the file. To look at other methods to fetch the file, look at the manual for the `SRADB` package available at <http://www.bioconductor.org/packages/release/bioc/manuals/SRADb/man/SRADb.pdf>.

Reading FASTQ files in R

A FASTQ file, as explained earlier, can have several short sequence reads in it. It is the universally accepted format in the NGS community and for most alignment programs such as Bowtie, which uses these files as input. In order to analyze the data, we need to read in this data in the workspace. This recipe addresses this issue of reading FASTQ files in R.

Getting ready

For our recipe, we will need a FASTQ file to be read and the [ShortRead](#) package. We can download a file from the SRA database, or use the example file in the [ShortRead](#) package. The following screenshot shows a FASTQ file:

```
@SRR038845.3 HWI-EAS038:6:1:0:1938 length=36
CAACGAGTTCACACCTTGGCCGACAGGCCCGGGTAA
+SRR038845.3 HWI-EAS038:6:1:0:1938 length=36
BA@7>B=>:>>7@7@>>9=BAA?;>52;>:9=8.=A
@SRR038845.41 HWI-EAS038:6:1:0:1474 length=36
CCAATGATTTTTTTTCCGTGTTTCAGAATACGGTTAA
+SRR038845.41 HWI-EAS038:6:1:0:1474 length=36
BCCBA@BB@BBBBBAB@B9B@=BABA@A:@693:@B=
@SRR038845.53 HWI-EAS038:6:1:1:360 length=36
GTTCAAAAAGAACTAAATTGTGTCAATAGAAAACCTC
+SRR038845.53 HWI-EAS038:6:1:1:360 length=36
BBCBBBBBBB@@BAB?BBBBBCBC>BBBAA8>BBBAA@
```

How to do it...

The following steps explain the functions that can be used to read a FASTQ file in the R workspace:

1. To download files from within R, use the following functions:

```
> install.packages("R.utils")
> library(R.utils)
>
download.file(url="ftp://ftp.ddbj.nig.ac.jp/dd
bj_database/dra/fastq/SRA000/SRA000241/SRX0001
22/SRR000648.fastq.bz2", destfile =
"SRR000648.fastq.bz2")
```

Alternatively, you can move to the working directory using the Linux terminal and download an example FASTQ file from the following link. Note that the following commands are Linux commands and not R commands:

```
$ cd <path/to working/directory>
$ wget
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fas
tq/SRA000/SRA000241/SRX000122/SRR000648.fastq.
bz2
```

You can also download the files via browsers on any platform (Windows, Mac OS, or Linux).

2. Now, unzip the downloaded file in your working directory as follows (another Linux command):

```
$ bunzip2 SRR000648.fastq.bz2
```

As an alternative, you can get the FASTQ file by unzipping it via a GUI-based program (such as Bzip2 or 7zip). This also works well for any other OS platform.

To unzip from within R, use the following command:

```
> bunzip2(list.files(pattern = ".fastq.bz2$"))
```

3. Now, install and load the `ShortRead` package from Bioconductor as follows:

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

4. To read the downloaded FASTQ file as an R object, use the `readFastq` function as follows. Note that FASTQ files are usually heavy and might require huge memory space while reading (it might need approximately 8 GB); therefore, check your machines before working on such files:

```
> MyFastq <- readFastq(getwd(),
  pattern=".fastq")
```

5. Look at the first four lines of the file by the generic `readLines` function as follows:

```
> readLines("SRR000648.fastq", 4)
[1] "@SRR000648.1 EUEMSW405C5XBJ length=218"
[2]
"TTGCGTGCGGAGCATTTCATTTACAAAGCCGTGGGAAGTCA
TAAAATTAATTCGGCATAACCATGTGTTCAAGTGTGTATACAGTGA
GAGTAGCTCCCGCTGCCCATTTGTTCAAGACTTCTGCTTCTTGTGG
GTTGTGGCTGGAATGAAGACACCGGTCTTCACTGCAATAAGGTTGT
TGTTTCTGATTTCCAACCCCAAATCCGTCCAGCTTCTGAGACACGC
AACAGGGGATAGGCAAGGCACACAGGGGATAGG"
[3]
[4] "C==<==FB1==<<HD8+
<FB1==FB1<=B;C<C<<<EA/A9=;
<<=FC6' C=C=FB1<C<<<=C=C<<=<==C=<<<=<<=<====
<====<=<==<<;<FB0==<<GB1<C=<FB0=C<=:C=
<====C=;B;;=B=C<8<?6<=B;B;<<B;=8:;C<C=;;C=
<<<====C;;A9C<C=8B;;FB0<<=<GC3C=C<FB4$A=(C=:
<A9=;<B<=8<8:99<;<=A<<8HD8,<<<A<:A;B<97==
<<HD8,<9;A;"
```

How it works...

The function that reads the FASTQ files works by reading the lines. It reads all the FASTQ-formatted files that match a given pattern in the directory being scanned. The function returns the sequence and the quality scores from the FASTQ file compiled into an R object. We simply used `fastq$` as the pattern and the current working directory (`getwd()`) as the directory being scanned.

See also

- Darren Wilkinson's research blog at <http://darrenjw.wordpress.com/2010/11/29/a-quick-introduction-to-the-bioconductor-shortread-package-for-the-analysis-of-ngs-data/>, which provides an introduction to the `ShortRead` package
- The user manual that details each functionality of the package and is available from the `ShortRead` package's homepage at <http://www.bioconductor.org/packages/release/bioc/html/ShortRead.html>

Reading alignment data

The outputs of NGS experiments are sequence reads that have to be aligned and mapped to a reference genome. The first step in NGS data analysis is to align reads to the reference genome. This task of alignment is computationally demanding due to the huge volumes of NGS data and reference genomes. However, there are tools beyond R to do this. Most commonly used alignment tools include BWA and Bowtie. It is beyond the scope of the book to go into the details of these two methods. Nevertheless, a short explanation has been offered on Bowtie in the *See also* section of this recipe.

After mapping the reads in the FASTQ file to the reference genome, what we get is a **sequence alignment map (SAM)** or BAM (binary version of SAM) file.

This recipe will discuss how to read/load these files in the R workspace.

Getting ready

We will need the `Rsamtools` package to read the BAM files. The example data used in this recipe is available from UCSC for demo purposes. The data is actually a sequence set sample (NA12878) from the 1000 Genome Project.

How to do it...

The following steps will explain the method to be used to deal with a BAM file:

1. First, download an example file from within R, as follows:

>

```
download.file(url="http://genome.ucsc.edu/goldenPath/help/examples/bamExample.bam", destfile = "bamExample.bam")
```

Alternatively, you can also use Linux terminal commands without starting R, as follows:

```
$ cd path/to/working/directory
$ wget
http://genome.ucsc.edu/goldenPath/help/examples/bamExample.bam
$ R
```

2. To read the BAM file, use the `scanBam` function from the `Rsamtools` package as follows (the file has been provided with the code files on the book's web page for the ease of access):

```
> library(Rsamtools)
> bam <- scanBam("bamExample.bam")
```

3. Take a look at the attributes for the first list element of the read data as follows:

```
> names(bam[[1]])
```

4. Check the count of the records in the data as follows:

```
> countBam("bamExample.bam")
      space start end width      file
records nucleotides
1      NA      NA  NA   NA bamExample.bam
36142      1474950
```

5. If you want to read only selected attributes, set them as parameters by typing the following commands:

```
> what <- c("rname", "strand", "pos",
"qwidth", "seq")
> param <- ScanBamParam(what=what)
> bam2 <- scanBam("bamExample.bam",
```

```
param=param)
> names(bam2[[1]])
```

6. Read the data as a `DataFrame` object using the following function:

```
> bam_df <- do.call("DataFrame", bam[[1]])
> head(bam_df)
```

7. From this `DataFrame` object, extract the sequences that fulfill certain conditions as follows:

```
> table(bam_df$rname == '21' & bam_df$flag ==
16)
FALSE  TRUE
31894  4248
```

How it works...

The `scanBam` function together with another input function called `countBam` imports the binary alignment maps in an NGS analysis. The `what` argument defines the attributes that have to be imported from the data. More information on the function is available within the help file (`?scanBam`). The imported files are stored as a list, with each element being an alignment. The list can be easily transformed into a `data.frame` object as shown in step 6. As a `data.frame` object, each entry in the list is a row and the attributes of the alignment are shown in terms of columns. Finally, we use the `table` function to determine how many of the sequences in the file meet the desired conditions.

In this recipe, the conditions for a demo were `rname`, which refers to the reference sequence's name, and `flag`, indicating the bitwise flag in the data. The bitwise flag can define the different properties of the segments in the data, namely, multiple segments, unmapped segments, the first segment, and so on. In this recipe, we used a value of 16 that indicates reads for reverse strands. For detailed information on flags, refer to the tool available at <http://picard.sourceforge.net/explain-flags.html>.

See also

- The article titled *The Sequence Alignment/Map format and SAMtools* by Li and others at <http://bioinformatics.oxfordjournals.org/content/25/16/2078.full>, which provides more details on the BAM and SAM files
- The Broad Institute's page at <http://www.broadinstitute.org/igv/bam>, which provides substantial information on the BAM and SAM files
- The SAMtools page at <http://samtools.sourceforge.net>, which provides various utilities for manipulating alignments in the SAM format

Preprocessing the raw NGS data

FASTQ data has the sequences (the bases) as the corresponding quality scores (Phred) in terms of ASCII characters, as explained in the introductory part of the chapter. Once read into the R workspace, the data is ready to be analyzed. However, it needs some preprocessing to meet the desired conditions on quality and data instance according to our interest. For example, we need higher Phred scores and a particular strand. This preprocessing involves quality assessment and filtering. This recipe will deal with these aspects, specifically filtering and quality checks.

Getting ready

For this recipe, we will use the data downloaded from the SRA database. We will also continue to use the `ShortRead` library.

How to do it...

The following steps will take us through the preprocessing of data, starting from data retrieval to filtering:

1. First, download the required files as follows (note that the download might take a while):

```
>
download.file(url="ftp://ftp.ddbj.nig.ac.jp/dd
bj_database/dra/fastq/SRA000/SRA000241/SRX0001
22/SRR000648.fastq.bz2", destfile=
"SRR000648.fastq.bz2")
>
download.file(url="ftp://ftp.ddbj.nig.ac.jp/dd
bj_database/dra/fastq/SRA000/SRA000241/SRX0001
```

```
22/SRR000657.fastq.bz2", destfile=
"SRR000657.fastq.bz2")
```

Alternatively, you can also download the files in the working directory using a Linux terminal, and download an example FASTQ file from the following link:

```
$ cd <path/to working/directory>
$ wget
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA000/SRA000241/SRX000122/SRR000648.fastq.bz2
$ wget
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA000/SRA000241/SRX000122/SRR000657.fastq.bz2
```

2. Unzip the downloaded files from within R as follows:

```
> install.packages("R.utils") # install the
R.utils from CRAN
> library(R.utils)
> bunzip2(filename=list.files(pattern =
".fastq.bz2$")[1]) # Unzips and removes the
original bunzip file
```

Alternatively, you can also download the files in a Linux terminal as follows:

```
$ bunzip2 *.fastq.bz2
```

3. To assess the quality, use the `FastQuality` function from the `ShortRead` library as follows:

```
> library(ShortRead)
> myFiles <- list.files(getwd(), "fastq",
full=TRUE)
> myFQ <- lapply(myFiles, readFastq)
> myQual <-
FastQuality(quality(quality(myFQ[[1]])))
```

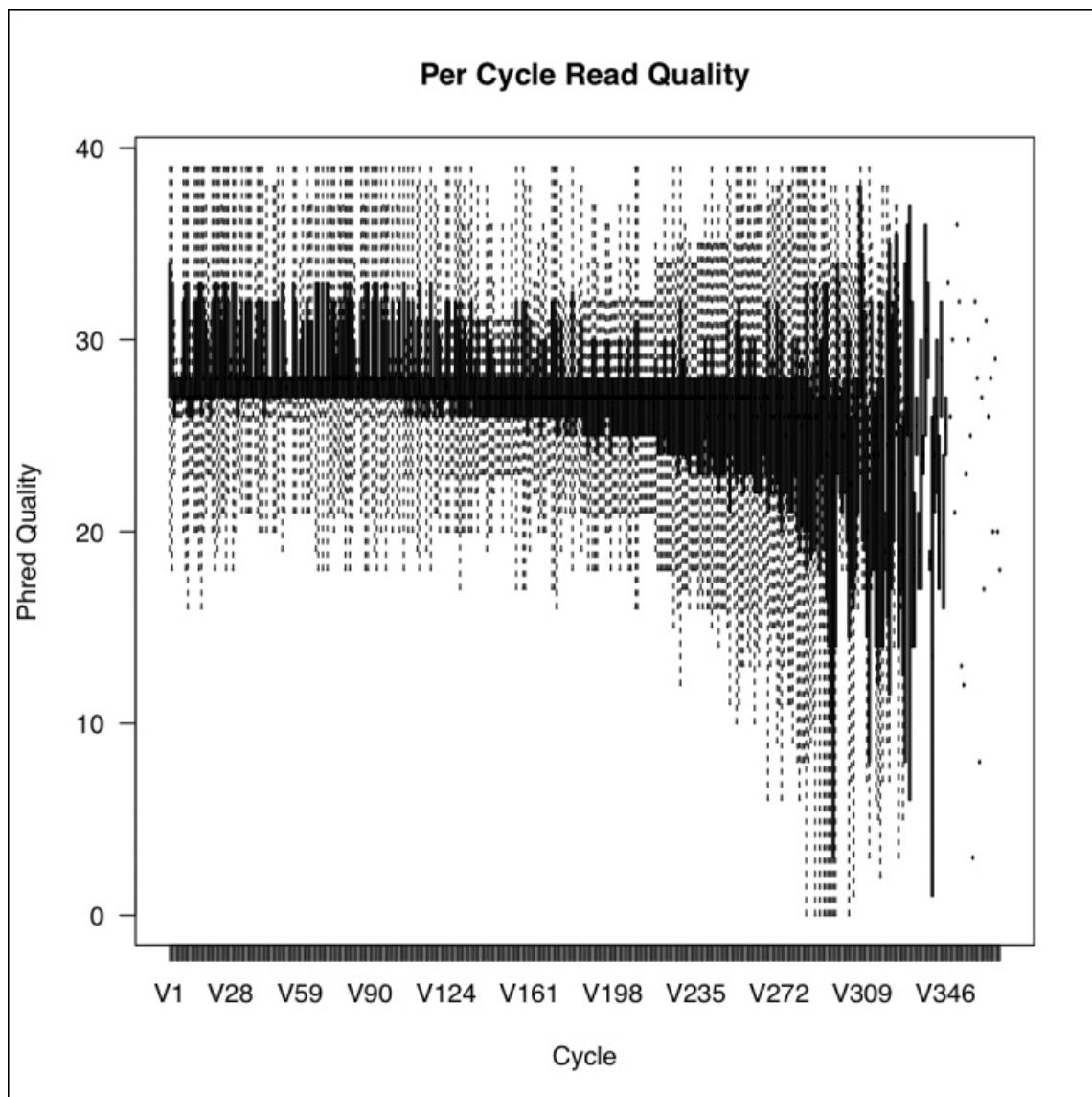
4. Convert the quality measure in terms of a matrix as follows:

```
> readM <- as(myQual, "matrix")
```

5. Visualize the results as a boxplot with the help of the following command:

```
> boxplot(as (data.frame(readM)), outline =  
FALSE, main="Per Cycle Read Quality",  
xlab="Cycle", ylab="Phred Quality")
```

In the following boxplot, the x axis shows the cycles and the y axis represents the Phred quality score available within the FASTQ file:



6. Another interesting preprocessing step involves filtering the sequences while reading alignment data. For this, you first need alignment data as follows (note that [Bowtie](#) data is available on the book's web page under the name [myBowtie.txt](#)) and you must either copy it to your working directory or use this directory as working directory to use the [myBowtie.txt](#) file:

```
> myData <- readAligned("myBowtie.txt", type =
"Bowtie")
```

7. To check the read alignments, look at the [myData](#) object as follows:


```

> myData
class: AlignedRead
length: 1000 reads; width: 35 cycles
chromosome: chr5 chr10 ... chr15 chr16
position: 151311502 35505989 ... 25552487
33204792
strand: - - ... + +
alignQuality: NumericQuality
alignData varLabels: similar mismatch

```

8. There are different filters that can be used, such as chromosomes, sequence length, strands, and so on. To use these, first define the required filter (in this case, define a filter for the + strand) as follows:

```

> strand <- strandFilter("+")

```

9. Then, use the created filter as follows:

```

> myRead_strand <- readAligned("myBowtie.txt",
filter=strand, type = "Bowtie")

```

10. Combine more than one filters and then use with the `compose` function as follows:

```

> chromosome <- chromosomeFilter("3")
> myFilt <- compose(strand, chromosome)
> myRead_filt <- readAligned("myBowtie.txt",
filter=myFilt, type = "Bowtie")

```

11. Again, take a look at the filtered data (`myRead_filt`) as follows:

```

> myRead_filt
class: AlignedRead
length: 55 reads; width: 35 cycles
chromosome: chr3 chr13 ... chr3 chr13
position: 149312235 27275680 ... 136314155
52437523
strand: + + ... + +
alignQuality: NumericQuality
alignData varLabels: similar mismatch

```

How it works...

The quality check function described in this recipe uses the Phred scores assigned in terms of ASCII characters in the FASTQ files, and computes the actual quality score for each cycle. This is reflected in the plot we saw in the previous boxplot. The filtering function simply checks the data and extracts the alignment reads that meet the filtering conditions. The other filters that can be used include `idFilter`, `positionFilter`, and so on. We can see the difference between the `myData` object that has 1,000 reads with both the `+` and `-` strands, whereas the filtered object, `myRead_filt`, has only 55 reads with the `+` strand.

There's more...

Another interesting quality assessment is available on the University of California Riverside page (www.ucr.edu). To load these assessment R functions from the remote host, we can simply use the following function (note that we will use this in one of our visualization recipes):

```
>
source("http://faculty.ucr.edu/~tgirke/Documents/R
_BioCond/My_R_Sc")
```

We can then use various functions loaded for the quality assessment of the FASTQ files. Some of the interesting functions include `seeFastqPlot` and `seeFastq`. To know more, visit http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/My_R_Sc.

Analyzing RNAseq data with the edgeR package

In order to determine whether the count for a transcript is significantly different or differentially expressed under the treatment condition, we need to do a differential count analysis for the data. This is analogous to the differential gene analysis explained in [Chapter 5](#), *Analyzing Microarray Data with R*. We will do such an analysis using the `edgeR` package in this recipe.

Getting ready

We will need two R packages, namely, `edgeR` and `goseq`, for the recipe. In terms of dataset, we will demonstrate in terms of the default data for these packages. The data comes from an experiment that examined the effect of androgen stimulation on a human prostate cancer cell line, `LNCaP`. The data has four control and three treated samples.

How to do it...

To perform differential analysis for the `RNAseq` data, carry out the following steps:

1. First, start with the installation and loading of the required libraries as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("edgeR")
> biocLite("goseq")
> library(edgeR)
> library(goseq)
```

2. Now, read in the input data from the `goseq` library data directory as follows:

```
> myData <- read.table(system.file("extdata",
  "Li_sum.txt", package='goseq'), sep = '\t',
  header = TRUE, stringsAsFactors =
  FALSE, row.names=1)
```

3. Take a look at the content of a part of the data as follows:

```
> head(myData)
```

```
      lane1 lane2 lane3 lane4 lane5 lane6 lane8
ENSG00000215688      0      0      0      0      0
0      0
ENSG00000215689      0      0      0      0      0
0      0
ENSG00000220823      0      0      0      0      0
0      0
ENSG00000242499      0      0      0      0      0
0      0
ENSG00000224938      0      0      0      0      0
0      0
ENSG00000239242      0      0      0      0      0
0      0
```

4. The first four columns in your data are controls and the last three are the treatment samples (see the *Getting ready* section of this recipe). Assign these attributes to the data as follows:

```
> myTreat <-
  factor(rep(c("Control", "Treatment"), times =
    c(4, 3)))
```

5. Now, create a `DGElist` object using all the count data and treatment information as follows:

```
> myDG <- DGElist(myData, lib.size =
  colSums(myData), group = myTreat)
```

The `DGElist` object is a list with two components: counts and sample (treatment information), as shown in the following example:

```

> myDG
An object of class "DGEList"
$counts
      lane5 lane6 lane8 lane1 lane2 lane3 lane4
ENSG00000215688 0 0 0 0 0 0
0 0
ENSG00000215689 0 0 0 0 0 0
0 0
ENSG00000220823 0 0 0 0 0 0
0 0
ENSG00000242499 0 0 0 0 0 0
0 0
ENSG00000224938 0 0 0 0 0 0
0 0
49501 more rows
$samples
      group lib.size norm.factors
lane1 Control 1178832 1
lane2 Control 1384945 1
lane3 Control 1716355 1
lane4 Control 1767927 1
lane5 Treatment 2127868 1
lane6 Treatment 2142158 1
lane8 Treatment 816171 1

```

6. Now, estimate the dispersion in the data by typing the following command:

```

> myDisp <- estimateCommonDisp(myDG)

```

7. This is followed by an exact test as follows:

```

> mytest <- exactTest(myDisp)

```

8. Extract the top DE tags ranked by the p-value (or the absolute log fold change) using the following `topTags` function:

```

> myRes <- topTags(mytest, sort.by = "PValue")

```

9. To see the results, simply check the head of the `data.frame` object as follows:

```
> head(myRes)
```

```
Comparison of groups: Treatment-Control
                        logFC    logCPM
PValue      FDR
ENSG00000127954 11.557868 6.680782 2.574972e-
80 1.274766e-75
ENSG00000151503 5.398963 8.499425 1.781732e-
65 4.410322e-61
ENSG00000096060 4.897600 9.446635 7.983756e-
60 1.317479e-55
ENSG00000091879 5.737627 6.281841 1.207655e-
54 1.494654e-50
ENSG00000132437 -5.880436 7.951978 2.950042e-
52 2.920896e-48
ENSG00000166451 4.564246 8.458263 7.126763e-
52 5.880292e-48
```

How it works...

The [edgeR](#) package uses the count data by modeling it via an overdispersed Poisson model, assuming it to be a negative binomial distribution. Then, it follows an Empirical Bayes procedure to moderate the degree of overdispersion across genes by conditional maximum likelihood, conditioned on the total count for that gene (step 5). To compute the differential expression of a tag/gene, a Fisher's exact test (step 7) is performed, yielding the corresponding statistical scores. Finally, the top ranking tags (according to the p-values) are returned in steps 8 and 9. The [topTags](#) function, by default, returns the top 10 results; we can return desired number of top ranked tags by setting the argument [n](#) to this number in the function.

See also

- The article titled *edgeR: a Bioconductor package for differential expression analysis of digital gene expression data* by Robinson and others at

- <http://bioinformatics.oxfordjournals.org/content/26/1/139.long>, which provides more details on the edgeR package
- The articles titled *Moderated statistical tests for assessing differences in tag abundance* (http://bioinformatics.oxfordjournals.org/content/23/21/2881.abstract?ijkey=02b1e386cfb11240f48d2bafc520b091f53bfd95&keytype=tf_ipsecsha) and *Small-sample estimation of negative binomial dispersion, with applications to SAGE data* (http://biostatistics.oxfordjournals.org/content/9/2/321.abstract?ijkey=7c4f41ecdabc114236c79cea46db9d733c624d69&keytype=tf_ipsecsha) by Robinson and others, which provide more details about the statistical methods
 - The *Determination of tag density required for digital transcriptome analysis: Application to an androgen-sensitive prostate cancer model* article by Li and others at <http://www.pnas.org/content/early/2008/12/16/0807121105>, which provides information on the data used in this recipe

The differential analysis of NGS data using limma

We have discussed differential gene expression analysis in one of our previous recipes. Another popular pipeline for the purpose is provided by the `limma` software package, which we saw in [Chapter 5, *Analyzing Microarray Data with R*](#). It can handle multiple experiments via Empirical Bayes statistical methods and uses normalized read counts for each gene. This recipe will explain the use of the `limma` package from Bioconductor for differential gene analysis with NGS data.

Getting ready

Besides using the `limma` library, we will use the `Pasilla` dataset here. The dataset can be obtained from Bioconductor and consists of sequence counts from a perturbation experiment in *Drosophila*. To know more about the data, refer to the *Conservation of an RNA regulatory map between Drosophila and mammals* article by Brooks and others (<http://genome.cshlp.org/content/early/2010/10/04/gr.108662.110>).

How to do it...

The following steps will explain the use of the `limma` package in NGS data analysis:

1. Start with the input data, which can be loaded as a package from Bioconductor, as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("DESeq", "pasilla"))
> library(limma)
> library(DESeq)
```



```
> library(pasilla)
> data(pasillaGenes)
```

2. Check the data content as follows:

```
> pasillaGenes
```

3. Now, create an expression set using the counts from the `pasillaGenes` dataset as follows:

```
> eset <- counts(pasillaGenes)
```

4. You have seven samples in the dataset; the first three are the treatment samples and the last four are controls. Assign this to your `eset` data as follows:

```
> colnames(eset) <- c(paste("T", 1:3,
  sep="_"), paste("C", 1:4, sep="_"))
```

5. Take a look at the data by typing the following command:

```
> head(eset)
```

	T_1	T_2	T_3	C_1	C_2	C_3
C_4						
FBgn0000003	0	0	1	0	0	0
FBgn0000008	78	46	43	47	89	53
FBgn0000014	2	0	0	0	0	1
FBgn0000015	1	0	1	0	1	1
FBgn0000017	3187	1672	1859	2445	4615	2063
FBgn0000018	369	150	176	288	383	135

6. Now, create a design matrix as follows:

```
> design <-
  cbind(Intercept=1, trt=c(1,1,1,0,0,0,0))
```

7. Now, perform a `voom` transformation using the experiment design matrix as follows:

```
> eset_voom <- voom(eset, design, plot=FALSE)
```

8. Fit a linear model on your `eset` data and design matrix as follows:

```
> fit <- lmFit(eset_voom, design)
```

9. To perform the computation of statistics, use the following `eBayes` function:

```
> fitE <- eBayes(fit)
```

10. Now, to find the top genes and filter them based on the corresponding p-values, type the following commands:

```
> topAll <- topTable(fitE, n=nrow(eset),  
  coef = 2, adjust = "BH")  
> DEgenes <-  
  rownames(topAll[which(topAll$adj.P.Val<0.05),]  
  )
```

How it works...

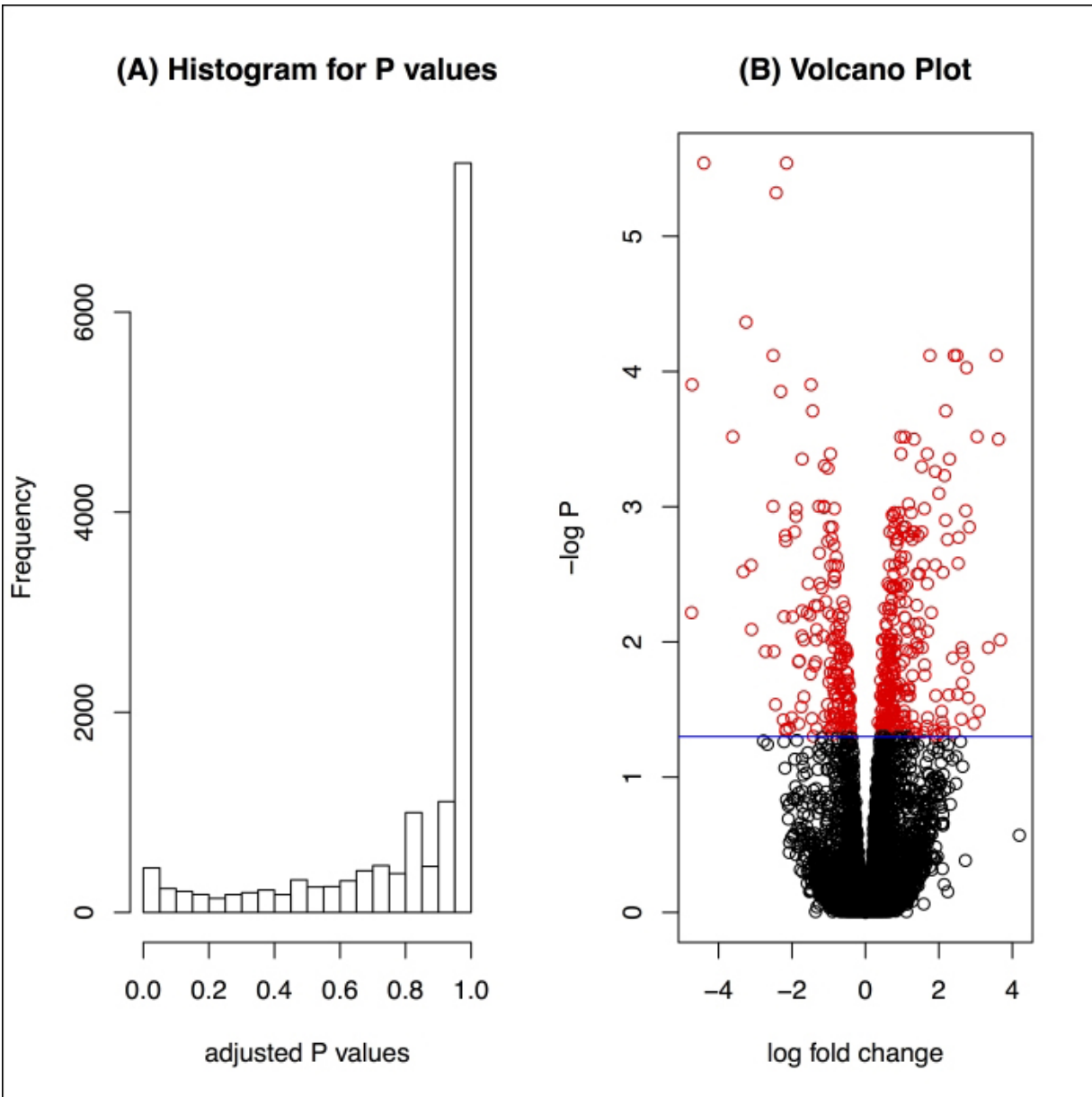
The `limma` analysis for a data has already been described in [Chapter 5, Analyzing Microarray Data with R](#). In this recipe, we used the count data in place of normalized expression values. Another difference is the use of the `voom` transformation that estimates the mean-variance relationship for the log counts, which robustly generates a precision weight for each individual normalized observation. To learn about the details of `limma`, refer to [Chapter 5, Analyzing Microarray Data with R](#).

The plot shown in the following screenshot shows a histogram of the tags with different ranges of p-values and a volcano plot for the `limma` analysis. The code to generate these plots is as follows:

```
> hist(topAll$adj.P.Val, xlab="adjusted P  
  values", main="(A) Histogram for P values")  
> clr <- rep("black", nrow(topAll)) # creates a  
  vector for color
```

```
> clr[which(topAll$adj.P.Val<0.05)] <- "red"# sets  
color for DE to red  
> plot(x = topAll$logFC, y = -  
log10(topAll$adj.P.Val), col = clr, xlab = "log  
fold change", ylab = "-log P",main = "(B) Volcano  
Plot") # Do a volcano plot  
> abline(h=-log10(0.05), col="blue")# Draw a  
horizontal line marking a P value threshold for  
0.05
```

The following screenshot shows the distribution of p-values in the analysis results and the number of tags with low p-values:



Histogram and the volcano plot for limma analysis

The results show that we have 445 tags that show DE tags, that is, tags that were significantly more expressive between the two conditions. We can check how many of these show a higher fold change, say two, as follows:

>

```
rownames(topAll[which(topAll$adj.P.Val<0.05&abs(topAll$logFC) > 2),])
```

We can see that 59 tags show a change of more than two folds (positive or negative). This renders these genes important for such studies.

See also

- The *voom: precision weights unlock linear model analysis tools for RNA-seq read count* article by Law and others at <http://genomebiology.com/2014/15/2/R29>, which provides more details on the `voom` transformation

Enriching RNAseq data with GO terms

The `RNAseq` data coming out of NGS provides great detail on the cellular transcriptional landscape. Besides measuring expression levels of the transcripts, they provide information on alternative splicing, allele-specific expressions, and so on. Thus, the `RNAseq` data gives a more comprehensive picture of differential expression in cells. However, adding the functional aspects can refine this analysis further with statistical robustness. This recipe aims to demonstrate the highlighting of the `RNAseq` data in terms of GO terms.

Getting ready

For our demonstration, we will continue using the data from the `goseq` package. We will also need the `goseq` library for our analysis.

How to do it...

To perform the enrichment analysis, carry out the following steps:

1. Start with loading the required packages, `goseq` and `edgeR`, as follows:

```
> library(goseq)
> library(edgeR)
> library()
```

2. Take a look at the genomes that are supported by the package as follows:

```
> supportedGenomes()
```

3. Now, load the data you will use for analysis from the `goseq` package as follows:

```
> myData <- read.table(system.file("extdata",  
  "Li_sum.txt", package="goseq"), sep = '\t',  
  header = TRUE, stringsAsFactors =  
  FALSE, row.names=1)
```

Alternatively, you can read the data directly from the file provided with the code files on the book's web page as follows:

```
> myData <-  
  read.table("path/to/code/files/Li_sum.txt",  
  sep = '\t', header = TRUE, stringsAsFactors =  
  FALSE, row.names=1)
```

4. The first four columns in your data are controls and the last three are the treatment samples. Assign these attributes to the data by typing the following command:

```
> myTreat <-  
  factor(rep(c("Control", "Treatment"), times =  
  c(4, 3)))
```

5. Now, create a `DGEList` object via the `edgeR` library using all the count data and treatment information, as follows:

```
> myDG <- DGEList(myData, lib.size =  
  colSums(myData), group = myTreat)
```

6. With the `DGEList` object, estimate the dispersion in the data followed by an exact test, as follows:

```
> myDisp <- estimateCommonDisp(myDG)  
> mytest <- exactTest(myDisp)
```

7. Use the genes from this analysis for enrichment. So, extract the genes with the desired p-values and log fold change condition

with their corresponding gene names, creating a named vector, as follows:

```
> myTags <-  
as.integer(p.adjust(mytest$table$PValue[mytest  
$table$logFC!=0], method="BH") < 0.05)  
> names(myTags) <-  
row.names(mytest$table[mytest$table$logFC!=0,]  
)
```

8. The vector you created in the previous step bears the status of every gene in terms of DE or non-DE tags. Check the corresponding numbers as follows:

```
> table(myTags)
```

9. Next, compute the probability weighting function for a set of genes based on their status as follows:

```
> wtFunc <- nullp(myTags,"hg19","ensGene")  
> head(wtFunc)
```

	DEgenes	bias.data	pwf
ENSG00000230758	0	247	0.03757470
ENSG00000182463	0	3133	0.20436865
ENSG00000124208	0	1978	0.16881769
ENSG00000230753	0	466	0.06927243
ENSG00000224628	0	1510	0.15903532
ENSG00000125835	0	954	0.12711992

10. Use the results obtained to compute the enrichment as follows:

```
> myEnrich_wall <-  
goseq(wtFunc,"hg19","ensGene",  
test.cats=c("GO:BP"))
```

11. Now, take a look at the enrichment as follows:

```
> head(myEnrich_wall)
```

The following screenshot shows the GO enrichment of the data:

	category	over_represented_pvalue	under_represented_pvalue	numDEInCat	numInCat
1988	GO:0008150	1.062594e-163	1	2341	10978
2381	GO:0009987	1.472828e-144	1	2164	10009
6376	GO:0044699	8.663861e-125	1	1810	7970
6395	GO:0044763	4.866315e-112	1	1673	7313
1989	GO:0008152	1.525368e-72	1	1592	7633
9568	GO:0071704	1.603034e-67	1	1526	7302

12. To look at the meaning of a GO category, use the [GO.db](#) package as follows:

```
> library(GO.db)
> GOTERM[[GO.wall$category[1]]]
```

How it works...

A large part of the recipe (up to step 9) is similar to the previous recipe, dealing with the [edgeR](#) package. Till step 9, we computed our genes or tags of interest and we have the Ensembl IDs for the tags. The [goseq](#) function used in step 10 fetched the GO category for the Ensembl genes obtained in step 9. In this recipe, we used only the biological process category, but for other categories, we can use the values [GO:CC](#) or [GO:MF](#). Besides fetching the GO annotation, the [goseq](#) function also computes the enrichment in terms of over and under representation of p-values. Finally, the use of the [GO.db](#) package gives us the actual GO term details for the categories retrieved.

There's more...

In this recipe, we observed one of the enrichment methods, but there are other possible methods as well in [goseq](#) such as sampling or hypergeometric. To know more about these, type [?goseq](#) in your R console.

See also

- The *Gene ontology analysis for RNA-seq: accounting for selection bias* article by Young and others at

<http://genomebiology.com/2010/11/2/r14>, which provides details about the `goseq` methods

The KEGG enrichment of sequence data

The previous recipe dealt with the GO category enrichment of tags. However, we can do a similar analysis in terms of KEGG annotations. This recipe deals with the KEGG enrichment of data.

Getting ready

We will continue using the same data and packages here as in the previous recipe.

How to do it...

To perform the KEGG enrichment of sequence data, carry out the following steps (note that the first few steps in this recipe are the same as in the previous one):

1. Start with loading the required packages, `goseq` and `edgeR`, as follows:

```
> library(goseq)
> library(edgeR)
> library(org.Hs.eg.db)
```

2. Load the data you will use for analysis from the `goseq` package as follows:

```
> myData <- read.table(system.file("extdata",
  "Li_sum.txt", package = 'goseq'), sep = '\t',
  header = TRUE, stringsAsFactors =
  FALSE, row.names = 1)
```

Alternatively, you can read the data directly from the file provided with the code files on the book's web page as follows:

```
> myData <-  
  read.table("path/to/code/files/Li_sum.txt",  
    sep = '\\t', header = TRUE, stringsAsFactors =  
    FALSE, row.names=1)
```

3. The first four columns in your data are controls and the last three are the treatment samples. Assign these attributes to the data and perform the differential tag computation as follows:

```
> myTreat <-  
  factor(rep(c("Control", "Treatment"), times =  
    c(4, 3)))  
> myDG <- DGEList(MyData, lib.size =  
  colSums(MyData), group = MyTreat)  
> myDisp <- estimateCommonDisp(myDG)  
> mytest <- exactTest(myDisp)
```

4. Use the genes from this analysis for enrichment, so extract the genes with the desired p-value and log fold change condition with their corresponding gene names, creating a named vector as follows:

```
> myTags <-  
  as.integer(p.adjust(mytest$table$PValue[mytest  
    $table$logFC!=0], method = "BH") < 0.05)  
> names(myTags) <-  
  row.names(mytest$table[mytest$table$logFC!=0, ]  
  )
```

5. Now, compile your KEGG data for enrichment, starting with the conversion of ENSEMBL IDs to Entrez as follows:

```
> en2eg <- as.list(org.Hs.egENSEMBL2EG)
```

6. Get all the KEGG IDs for the compiled Entrez IDs by typing the following command:

```
> eg2kegg <- as.list(org.Hs.egPATH)
```

7. Now to get the KEGG and Entrez IDs mapped together, type the following commands:

```
> grepKEGG <- function(id,mapkeys)
{unique(unlist(mapkeys[id], use.names =
FALSE))}
> kegg <- lapply(en2eg,grepKEGG,eg2kegg)
```

8. To compute the probability weighting function, type the following command:

```
> pwf <- nullp(MyTags,"hg19","ensGene")
```

9. Use the `goseq` function with KEGG mappings for the enrichment of tags as follows:

```
> KEGG <- goseq(pwf, gene2cat = kegg)
> head(KEGG)
```

The following screenshot shows the KEGG enrichment of the data:

	category	over_represented_pvalue	under_represented_pvalue	numDEInCat	numInCat
85	01100	3.397842e-16	1.0000000	221	915
88	03010	1.533182e-09	1.0000000	29	89
203	05200	4.456968e-08	1.0000000	72	250
113	04115	3.992541e-07	0.9999999	26	64
167	04914	3.186160e-06	0.9999991	28	76
77	00900	4.175229e-06	0.9999997	10	15

How it works...

This recipe is very similar to the previous recipe. However, in this recipe, we aim to find the KEGG annotation in place of GO categories. Until step 4, we fetch the differentially expressed tags from data as we did in our previous recipe followed by the extraction of genes of interest (meeting the p-value and log fold change values). Step 5 gets the Entrez mapping of the ENSEMBL genes,

which is followed by the corresponding KEGG mapping (of Entrez genes) in step 6. Finally, we extract the KEGG annotation of the interesting genes from the entire list and compute the enrichment scores.

Analyzing methylation data

DNA methylation data can serve as a tool to detect an epigenetic marker for which a detailed mechanism of mitotic inheritance has been described. It is usually done via methylation-specific DNA sequencing or via microarray data epigenetic regulation. Recent advances in NGS and microarray technology make it possible to map genome-wide DNA methylation at a high resolution and in a large number of samples. One of the common goals in analyzing methylation data is to identify **differentially methylated regions (DMRs)** when comparing control and treatment conditions. This recipe aims to address this issue.

Getting ready

For the purpose of our demonstrative analysis, we will use the `methyAnalysis` package and the built-in dataset.

How to do it...

The steps for analysis of methylation data are as follows:

1. Start with loading the library and dataset in your R workspace as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("methyAnalysis",
  "TxDb.Hsapiens.UCSC.hg19.knownGene"))
> library(methyAnalysis)
> data(exampleMethyGenoSet)
```

2. To get some information, such as samples and location for the data, look at different components, as follows (note that you can see eight samples):

```

> slotNames(exampleMethyGenoSet)
> head(locData(exampleMethyGenoSet))
> dim(exprs(exampleMethyGenoSet))
> pData(exampleMethyGenoSet)
> str(exampleMethyGenoSet)

```

3. The data has two sample conditions, which are `Type1` and `Type2`. Smoothen the input data with a desired window size (here, it is 200), as follows:

```

> methylSmooth <-
smoothMethyData(exampleMethyGenoSet, winSize =
200) #Might get some warning messages
> attr(methylSmooth, 'windowSize')

```

4. Extract the sample conditions from the `pData` component as follows:

```

> conditons <-
pData(exampleMethyGenoSet)$SampleType

```

5. Take the input data and sample type from the preceding step and detect the DMRs in the data as follows:

```

> myDMR <-
detectDMR.slideWin(exampleMethyGenoSet,
sampleType= conditons) #Might get some warning
messages

```

6. Take a look at the object created by typing the following command:

```

> head(myDMR)

```

The following screenshot shows the first six entries in the `myDMR` object:


```
GRanges with 6 ranges and 10 metadata columns:
      seqnames      ranges strand | PROBEID difference      p.value      p.adjust startWinIndex endWinIndex startLocation
      <Rle>         <IRanges> <Rle> | <factor> <numeric> <numeric> <numeric> <numeric> <numeric> <integer>
cg17035109 chr21 [10882029, 10882029] * | cg17035109 -1.8411605 0.06276449 0.1888488      1      1      10882029
cg06187584 chr21 [10883548, 10883548] * | cg06187584 -0.4566059 0.41601486 0.6149596      2      2      10883548
cg12459059 chr21 [10884748, 10884748] * | cg12459059 -0.3591179 0.36542152 0.5627904      3      5      10884748
cg25450479 chr21 [10884967, 10884967] * | cg25450479 -0.3591179 0.36542152 0.5627904      3      5      10884748
cg23347501 chr21 [10884969, 10884969] * | cg23347501 -0.3591179 0.36542152 0.5627904      3      5      10884748
cg03661019 chr21 [10885409, 10885409] * | cg03661019 -0.3532662 0.38065600 0.5782099      6      6      10885409
      endLocation mean_Type1 mean_Type2
      <integer> <numeric> <numeric>
cg17035109      10882029 -2.4183775 -0.57721699
cg06187584      10883548 -2.2297567 -1.77315084
cg12459059      10884969  0.2594151  0.61853304
cg25450479      10884969  0.2594151  0.61853304
cg23347501      10884969  0.2594151  0.61853304
cg03661019      10885409 -0.4170363 -0.06377013
---
seqlengths:
chr21
NA
```

- Finally, to identify the significant DMRs, run `identifySigDMR` in the output from the previous step as follows:

```
> mySigDMR <- identifySigDMR(myDMR)
```

- Get the annotation information for the regions from UCSC as follows:

```
> dmr_anno <-
  annotateDMRInfo(mySigDMR, 'TxDb.Hsapiens.UCSC.h
g19.knownGene')
```

- Finally, export the results of your analysis by typing the following command:

```
> export.DMRInfo(dmr_anno,
  savePrefix='testExample')
```

How it works...

The example data we used in this recipe is an extension of the `eSet` class, bearing slots for the different types of information such as samples, chromosomes, and genomic ranges. Our analysis begins with data smoothing. The function considers the correlation in the nearby CpG sites in terms of the defined window (here, it is 200), thus reducing the noise. With the less noisy data in place, `detectDMR.slideWin` checks whether the region (smoothed) is differentially methylated based on the t test or wilcox test, which is then merged to get the continuous regions by calling the

`getContinuousRegion` function. The results annotated with the genome data are finally exported as CSV files in step 10. The following screenshot shows the the exported CSV file:

CHROMOSOM	POSITION	PROBEID	difference	p.value	p.adjust	startWinInd	endWinInd	mean_Type1	mean_Type2	Transcript	EntrezID	GeneSymbol	distance2TS	nearestTx	PROMOTER
chr21	19191096	cg12430776	-1.0715235	0.00012359	0.00706139	279	281	-4.0922713	-3.0207478	uc002yko.4	54149	C21orf91	607	uc002yko.4	FALSE
chr21	34522588	ch.21.33444	-1.2848047	7.6775E-06	0.00309388	998	998	-4.5115553	-3.2267505	uc002yra.4	728409	C21orf54	19953	uc002yra.4	FALSE
chr21	37851847	cg02417033	-1.4922516	9.4161E-05	0.00676844	1486	1486	0.62012439	2.112376	uc002yvl.1	23562	CLDN14	541	uc002yvl.1	FALSE
chr21	38066047	cg10445315	-2.548081	1.1317E-05	0.00369206	1514	1514	0.57726092	3.12534194	uc002yvp.3	6493	SIM2	-5944	uc002yvp.3	FALSE
chr21	38075599	cg22711869	-1.5521615	0.00023321	0.00990044	1555	1555	0.96720578	2.51936732	uc002yvp.3	6493	SIM2	3608	uc002yvp.3	FALSE
chr21	38076709	cg22289831	-3.516629	2.6093E-07	0.00055331	1556	1557	0.23931295	3.75594193	uc002yvp.3	6493	SIM2	4718	uc002yvp.3	FALSE
chr21	38076869	cg21697851	-3.4648857	4.1049E-05	0.00435793	1556	1558	0.57098982	4.03587551	uc002yvp.3	6493	SIM2	4878	uc002yvp.3	FALSE
chr21	38080975	cg15750546	-2.7367846	8.0311E-07	0.00056766	1574	1576	0.09690416	2.83368875	uc002yvp.3	6493	SIM2	8984	uc002yvp.3	FALSE
chr21	38081100	cg20349024	-2.7367846	8.0311E-07	0.00056766	1574	1576	0.09690416	2.83368875	uc002yvp.3	6493	SIM2	9109	uc002yvp.3	FALSE
chr21	38081193	cg01090834	-2.7367846	8.0311E-07	0.00056766	1574	1576	0.09690416	2.83368875	uc002yvp.3	6493	SIM2	9202	uc002yvp.3	FALSE
chr21	39285679	cg01360586	-3.8751831	2.5315E-05	0.00435793	1748	1748	-1.4551853	2.41999787	uc002ywo.3	3763	KCNJ6	3062	uc011aej.2	FALSE
chr21	39748803	cg24018174	-2.8013075	0.00014243	0.00784499	1803	1803	-0.1759281	2.62537942	uc021wjd.1	2078	ERG	284901	uc010gny.1	FALSE
chr21	40033892	cg17274064	-3.6649371	1.5097E-08	6.4028E-05	1823	1823	-2.8550838	0.80985331	uc021wjd.1	2078	ERG	-188	uc021wjd.1	TRUE
chr21	42217001	cg02475236	-2.039757	7.2278E-05	0.00567651	2026	2026	0.4670764	2.50683341	uc002yqj.1	1826	DSCAM	2038	uc002yqj.1	FALSE
chr21	43652704	cg01881899	-2.0082803	0.00021902	0.00990044	2384	2384	0.30997823	2.31825851	uc002zar.3	9619	ABCG1	12696	uc002zar.3	FALSE
chr21	45139229	cg00784703	-1.0405872	0.00020737	0.00990044	2848	2849	-4.9126547	-3.8720676	uc002zdm.4	8566	PDXK	251	uc002zdm.4	FALSE
chr21	45139379	cg14522549	-1.0405872	0.00020737	0.00990044	2848	2849	-4.9126547	-3.8720676	uc002zdm.4	8566	PDXK	401	uc002zdm.4	FALSE
chr21	47876058	cg09387528	-2.7791643	4.4054E-06	0.00233542	4167	4167	0.11495104	2.89411533	uc002zjl.3	23181	DIP2A	-2804	uc002zjl.3	FALSE
chr21	47878552	cg19247551	-1.1235182	5.0071E-07	0.00056766	4172	4174	-5.4820183	-4.3585001	uc002zjl.3	23181	DIP2A	-310	uc002zjl.3	TRUE
chr21	47878727	cg15775835	-1.0214568	8.7542E-06	0.00309388	4172	4176	-4.9690927	-3.9476359	uc002zjl.3	23181	DIP2A	-135	uc002zjl.3	TRUE
chr21	47878746	cg12533308	-1.0214568	8.7542E-06	0.00309388	4172	4176	-4.9690927	-3.9476359	uc002zjl.3	23181	DIP2A	-116	uc002zjl.3	TRUE

A differentially methylated region, which we aimed to find in this recipe, is a region where most of the CpG sites are methylated. What we did here was the detection of probe methylation status from the `exampleMethyGenoSet` object. Finally, we mapped these regions onto the chromosomal maps. In this recipe, we saw the SIM2, ERG, DIP2, and so on, regions being substantially methylated on chromosome 21, where the p-value was less than 0.05. The input data for the recipe was an `eSet` object. However, it can be in other formats such as a CSV or text file. The approach to create `eSet` from these files has been already explained in [Chapter 5, Analyzing Microarray Data with R](#).

See also

- The article titled *Analysing and interpreting DNA methylation data* by Bock (<http://www.nature.com/nrg/journal/v13/n10/full/nrg3273.html>), which provides more details on using methylation data
- The *Comparison of Beta-value and M-value methods for quantifying methylation levels by microarray analysis* article by Du and others (<http://www.biomedcentral.com/1471-2105/11/587>), which provides theoretical details of the `methyAnalysis` package

Analyzing ChipSeq data

Chromatin immunoprecipitation sequencing (ChipSeq) is a powerful method to identify genome-wide DNA binding sites for a protein of interest. It is often used to determine the binding sites for transcription factors, DNA-binding enzymes, histones, chaperones, or nucleosomes. The workflow to produce the [ChipSeq](#) data starts from the cross-links bound proteins and chromatin. The chromatin is fragmented, and the DNA fragments bound to one protein are captured using an antibody specific to it. The ends of the captured fragments are sequenced using NGS. The computational mapping of the sequenced DNA leads to the identification of the genomic locations of these fragments, illuminating their role in DNA protein interactions and epigenetics research.

The [ChipSeq](#) data consists of short reads in a FASTQ file format. There is a short read after every fifth line of the file. This recipe will deal with the analysis of the [ChipSeq](#) data in R.

Getting ready

The recipe will require the [ChipSeq](#) package and the built-in data in the package for demonstration purposes. We will also need the mouse genome data for the biological mapping of results.

How to do it...

To analyze the [ChipSeq](#) data in R, perform the following steps:

1. Start with the loading of the required R libraries. You need two libraries, namely [chipseq](#) and [TxDb.Mmusculus.UCSC.mm9.knownGene](#), as follows:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("chipseq",
```

```
"TxDb.Mmusculus.UCSC.mm9.knownGene") )  
> library(TxDb.Mmusculus.UCSC.mm9.knownGene)  
> library(chipseq)
```

2. Now, you need the data to be analyzed. For demonstration purposes, use the built-in package dataset named `cstest` as follows:

```
> data(cstest)
```

3. Take a look at the data by typing the following command before you start analyzing it:

```
> cstest
```

4. Now, estimate the length of the fragments in the data by typing the following command:

```
> estimate.mean.fraglen(cstest$ctcf)  
chr10    chr11    chr12  
179.6794 172.4884 181.6732
```

5. Extend your fragments in order to cover the binding sites in the sequences. Use an extension length inspired by the fragment lengths in the last step (`length = 200` > `estimate.mean.fraglen(cstest$ctcf)`) as follows:

```
> ctcf_ext <- resize(cstest$ctcf, width = 200)  
> gfp_ext <- resize(cstest$gfp, width = 200)
```

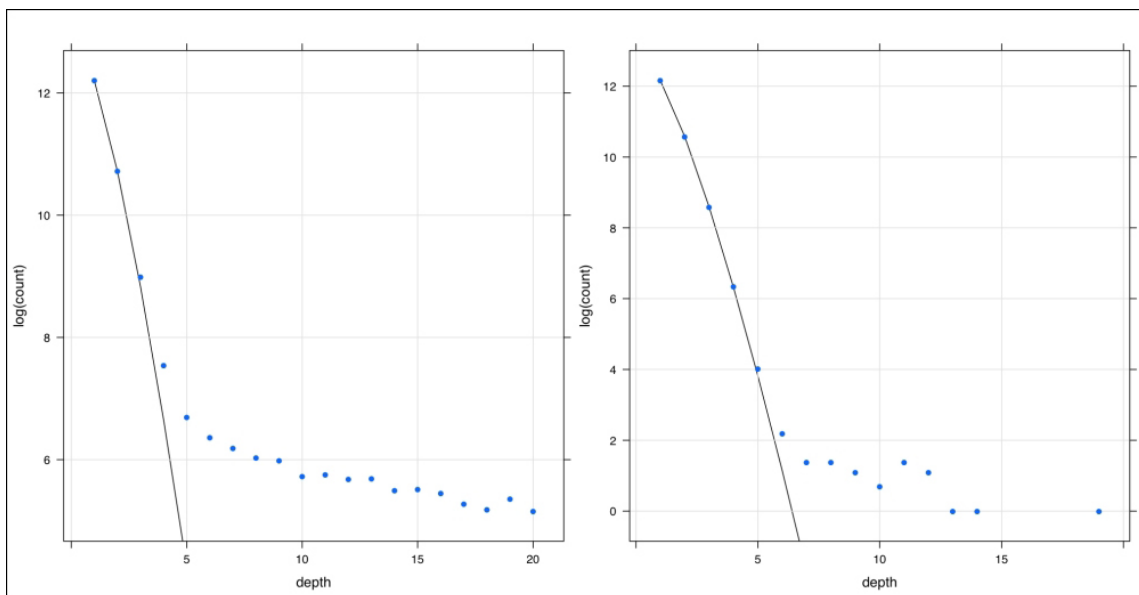
A useful summary of this information is the coverage, that is, how many times each base in the genome was covered by one of these intervals and can be computed as follows:

```
> cov_ctcf <- coverage(ctcf_ext)  
> cov_gfp <- coverage(gfp_ext)
```

6. Now, create a plot called islands for the regions of interest with the following functions. They are contiguous segments:

```
> par(mfrow = c(2, 1))
> islandDepthPlot(cov_ctcf)
> islandDepthPlot(cov_gfp)
```

In the following screenshot, the x axis shows the depth, whereas the y axis shows the corresponding log counts (overall, the plot shows the coverage for the sample data):



7. Now, compute the peak's cut off for a desired `fdr` (here, `0.01`) as follows:

```
> peakCutoff(cov_ctcf, fdr = 0.01)
[1] 5.091909
> peakCutoff(cov_gfp, fdr = 0.01)
[1] 6.979273
```

8. With this value, decide a cut-off value (7) to be used in order to get the peaks with high coverage in the segments of the data for both lanes, `ctcf` and `gfp`, as follows (note that the chosen cut off is based on your computation above 5.09 and 6.97):

```
> peaks_ctcf <- slice(cov_ctcf, lower = 7)
> peaks_gfp <- slice(cov_gfp, lower = 7)
```

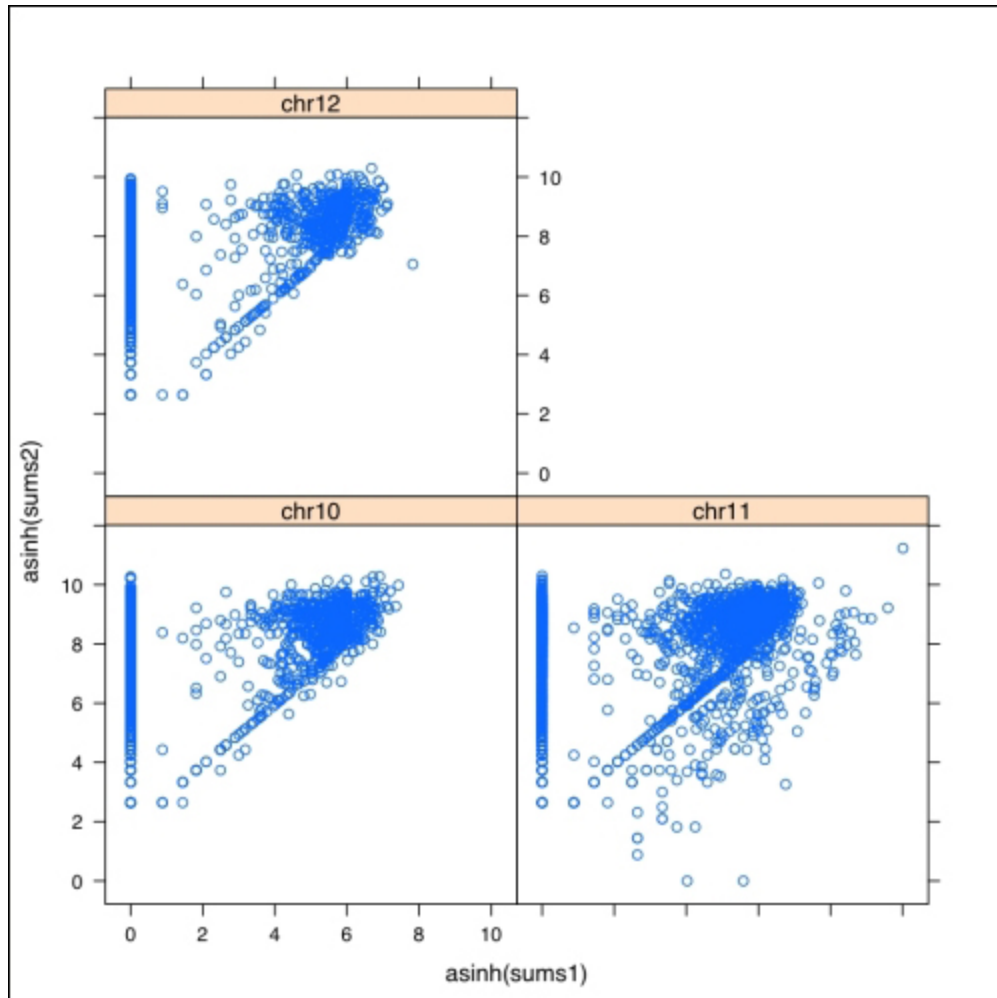
9. Now, compute the differential peaks with the following function to determine which peaks are different in the two samples:

```
> peakSummary <- diffPeakSummary(peaks_gfp,
  peaks_ctcf)
> head(data.frame(peakSummary))
```

10. Visualize the results in terms of an XY plot as follows:

```
> xyplot(asinh(sums2) ~ asinh(sums1) | space,
  data = as.data.frame(peakSummary))
```

In the following screenshot, you can see the summary of the peaks for three chromosomes:



11. Now, you have the peaks of your interest. Find if they are in the region of interest (promoter region) using the following commands:

```
> gregions <-
  transcripts(TxDb.Mmusculus.UCSC.mm9.knownGene)
> promoters <- flank(gregions, 1000, both =
  TRUE)
> peakSummary$inPromoter <- peakSummary %over%
  promoters
```

12. Take a look at the peaks in the promoter region by looking at the object created in the previous step as follows:

```
> which(peaks$inPromoter)
```


How it works...

In this recipe, the input data was from three chromosomes (10, 11, and 12) and in two lanes representing `ctcf` and `gfp` pull-down in mouse. The aim of our analysis is to find which peaks are different in two lanes (samples). The method explained in this recipe finds the coverage vectors for the data and computes statistical scores for the peaks therein. We start with finding the coverage for both the lanes by extending the reads to cover the binding sites. Thereafter, we define our peaks based on a cut-off value according to the desired `fdr` (0.01, as seen in step 7). The `diffPeakSummary` function then combines a set of peaks for the two lanes and summarizes them. Finally, we map the peaks onto the reference genome in step 11, and select the peaks that lie in the promoter region, which are of interest to find the binding sites (step 12).

See also

- The *ChIP-seq: welcome to the new frontier* article by Elaine R Mardis (<http://www.nature.com/nmeth/journal/v4/n8/abs/nmeth0807-613.html>), which provides a detailed description about the ChipSeq workflow
- The *Practical Guidelines for the Comprehensive Analysis of ChIP-seq Data* article by Bailey and others (<http://www.ploscompbiol.org/article/info%3Adoi%2F10.1371%2Fjournal.pcbi.1003326>), which provides a detailed description of the `ChipSeq` data analysis protocol
- The *ChIP-seq Analysis in R (CSAR): An R package for the statistical detection of protein-bound genomic regions* article by Muino and others (<http://www.newton.ac.uk/programmes/CGR/Muino,%20J.pdf>), which provides detailed information on the `ChipSeq` package
- The *BayesPeak-an R package for analysing ChIP-seq data* article by Cairns and others (), which provides information about the `BayesPeak` method

Visualizations for NGS data

This recipe will introduce some new visualizations that can be used to represent some results in NGS analysis. We will present two plots as follows:

- A quality assessment plot for NGS
- A map for the methylation data

Getting ready

The demonstration code presented in this recipe will use a function not available in the R package, but will obtain it from a remote host.

How to do it...

The steps for the above mentioned plots are as follows:

1. First, start with a comprehensive plot used for visualizing the quality in a FASTQ data. You can do this within R, as you did before, by typing the following commands:

```
> download.file(url =  
"http://biocluster.ucr.edu/~tgirke/HTML_Presen  
tations/Manuals/Rngsapps/chipseqBioc2012/data.  
zip", destfile = "data.zip")  
> unzip("data.zip")
```

Alternatively, you can do this in the Linux terminal as well by typing the following commands (note that this might take some time depending on your network speed and bandwidth):

```
$ cd <path/to/working/directory>  
$ wget  
http://biocluster.ucr.edu/~tgirke/HTML_Present
```

```

ations/Manuals/Rngsapps/chipseqBioc2012/data.zip
ip
$ unzip data.zip
$ cd Data/

```

2. Now, start your R session and load the following `ggplot2` library:

```

$ R
> library(ggplot2)

```

3. Load the function for your plot from a remote host as follows (note that it is advised to check the web pages to be used for source codes manually in order to avoid reaching malicious pages and availability):

```

>
source("http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/My_R_Scripts/fastqQuality.R")

```

4. Now, create a list of unzipped FASTQ files and assign names to each file for the purpose of annotation of plots as follows:

```

> fastq <- list.files(getwd(), "*.fastq$")
> names(fastq) <- paste("flowcell_6_lane",
1:4, sep="_")

```

5. With the FASTQ files ready, compute the quality related scores for each file as follows:

```

> fqlist <- seeFastq(fastq=fastq,
batchsize=1000, klength=5)

```

6. To visualize the results, plot the quality measurements for the first sample only as follows:

```

> seeFastqPlot(fqlist[1],
arrange=seq(along=fqlist))

```

7. The second visualization is a detailed visualization for methylation data. For this, use the analysis results from the

Analyzing methylation data recipe as follows:

```
> library(methyAnalysis)
> data(exampleMethyGenoSet)
```

8. Now, take a look at the CSV file that you exported in the *Analyzing methylation data* recipe and choose the gene you want to visualize on the chromosomal map (here, 54149, the first gene in the list).
9. To plot the map, use the `plotMethylationHeatmapByGene` function from the `methyAnalysis` package as follows:

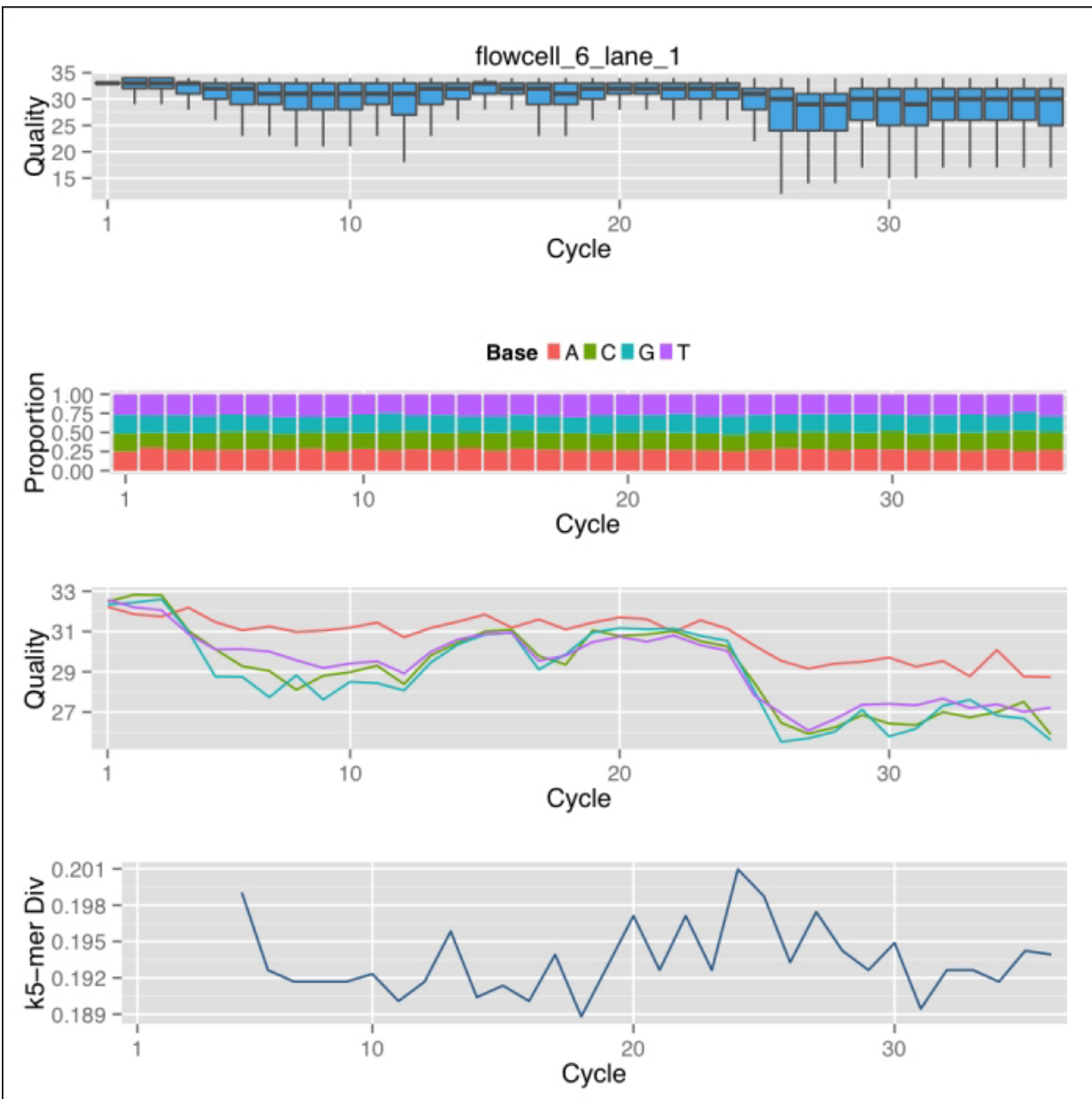
```
> plotMethylationHeatmapByGene('54149',
  methyGenoSet = exampleMethyGenoSet, phenoData
  = pData(exampleMethyGenoSet), includeGeneBody
  = TRUE, genomicFeature =
  'TxDb.Hsapiens.UCSC.hg19.knownGene')
```

How it works...

The quality plot (the first plot of the recipe) created here shows the following characters:

- A boxplot of quality (Phred score)
- Base (A, G, C, and T) proportion
- Mean base quality
- k-mer diversity

The following plot shows the quality and base compositions in every cycle:

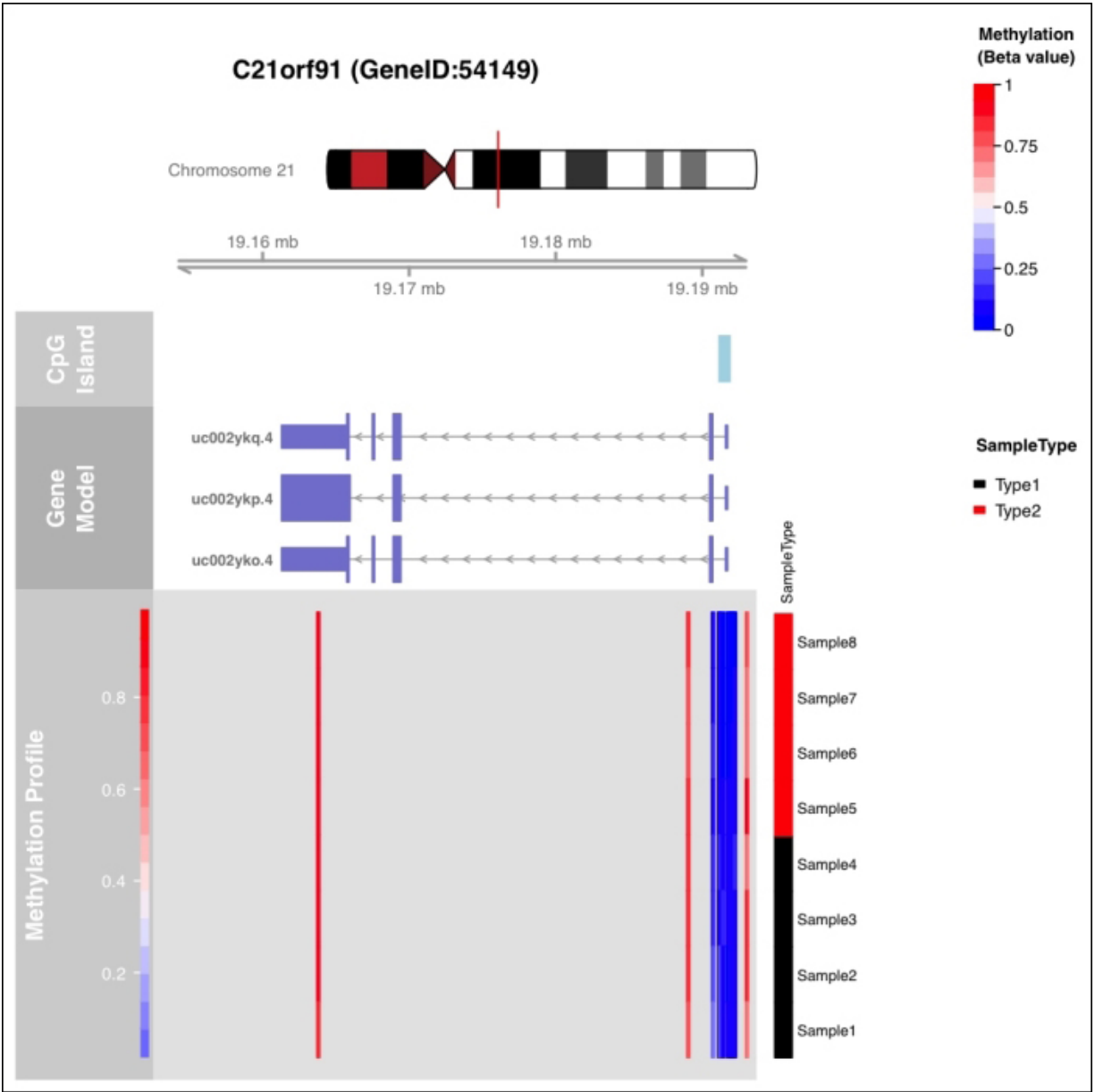


We can look into the content of the metrics computed for sample (lane) i and metrics j as follows:

```
> head(fqlist[[i]][[j]])
```

The second plot takes the gene 5149 and maps it onto chromosome 21 based on its location in the data from the genome map in `TxDb.Hsapiens.UCSC.hg19.knownGene`. Besides this, it also plots the

methylation beta or M values in the adjoining heatmap. The following screenshot shows the heatmap for methylation data with chromosomal location:



Chapter 9. Machine Learning in Bioinformatics

In this chapter, we will cover the following recipes:

- Data clustering in R using k-means and hierarchical clustering
- Visualizing clusters
- Supervised learning for classification
- Probabilistic learning in R with Naïve Bayes
- Bootstrapping in machine learning
- Cross-validation for classifiers
- Measuring the performance of classifiers
- Visualizing an ROC curve in R
- Biomarker identification using array data

Introduction

The exponential growth of the amount of available biological data raises two problems, that of efficient storage and management of data and of the extraction of useful information from such data. The second problem is one of the main challenges in computational biology; it requires the development of tools and methods that are capable of transforming all this heterogeneous data into biological knowledge about the underlying mechanism. To illustrate, think of gene expression data that consists of expression values from a series of DNA microarray experiments. The data comes from patients suffering from a disease, say, cancer. With this kind of data at our disposal, it is interesting to look for answers to the following questions:

- Which samples exhibit a higher similarity in their expression profiles?
- Is there some intra-sample similarity between the genes?

- Are there some genes that serve as the defining feature in certain types of cancer?

Machine learning is a domain in artificial intelligence that allows computers to learn these aspects, and more, from this kind of example data or past experience. Machine learning can create and optimize models that can predict outcomes based on the features in the known data (usually termed as training data).

The two major settings in which we perform machine learning are supervised learning and unsupervised learning. In supervised learning, we have the values of the samples in the training set and use this knowledge to predict values for the unknown sample: a test set. Classification is an instance of supervised learning where we have the categories (class labels) of samples known in the training set and create our models based on this set to predict the class labels for the unknown instances. Unsupervised learning uses a set of data instances without any known results. It finds patterns in the data and thus partitions the data into meaningful subsets (these categories are not predefined as in the case of supervised learning). Clustering is an example of unsupervised learning where exploratory data analysis is performed to find patterns to identify the similarities and dissimilarities among the data instances.

In this chapter, we will elaborate on how to deal with machine learning methods in bioinformatics using R. However, before we move on to specific problems in bioinformatics and their solutions via machine learning, we must first define a clear objective, and the data should be able to answer this question. The key issues that must be taken care of before we start with the formulation of learning models are as follows:

- **Assessing data quality:** Correctness and completeness of data is necessary to get the model to work in the way that it is supposed to. This includes checking the consistency, the attribute values and types, the missing values, and finding the outliers.

- **Data normalization:** Depending on the input data and the chosen algorithm, sometimes we need to standardize the data in order to make it comparable. However, it might not be required in cases where the unique scaling of data is significant. Similarly, the normalization method can be different depending on the types of features. Another important aspect is that one should first normalize the training data and then apply these normalization parameters to rescale the validation data.
- **Feature selection:** We need to get the features that show a dependency between the data instance and the objective (for example, class labels in the case of classification). Depending on the data, objective, and learning algorithm, all features might not be required to create a model; they either add noise, redundancy, or cause computational inefficiency for the model. Therefore, we need to identify useful features that can define the objective function. For example, avoiding a redundant feature or correlated feature is a good call.

There are many other aspects that should be taken care of, and they will be explained when required. It is beyond the scope of this book to go through all the details here. It is recommended that you go through a basic book on machine learning for more information (for example, *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman, which is available at http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf). The *Machine learning in bioinformatics* article by Larrañaga and others (<http://bib.oxfordjournals.org/content/7/1/86.long>) also gives a good overview of the field in the context of bioinformatics.

In all of the recipes presented here, the data we use has already gone through preprocessing. We will simply attempt to build our model based on the presented data (and later, validate it).

Data clustering in R using k-means and hierarchical clustering

Clustering is an unsupervised learning task in data mining. It aims to group a set of data points in such a way that those in the same group are more similar to each other than to those in other groups. These groups are unlabeled and are called clusters. The goal of clustering is to minimize the distances between the data points within the cluster and maximize the distances between the clusters. There are several functions available in R to perform different kinds of clustering. This recipe will explain some of these functions.

Getting ready

To perform clustering, we need our dataset to be clustered. We also need to decide the number of groups that we intend to organize our clusters into while using some of the clustering methods. The required R-packages will be introduced when needed. As a dataset, we use the yeast data on protein localization sites in gram-negative bacteria available from the book's web page ([yeast.rda](#)). The data has been selected for four localizations and 1299 instances. It has been filtered from the original data at <http://archive.ics.uci.edu/ml/machine-learning-databases/yeast/>. We have four localizations in the data, namely mitochondria, cytosol, nucleus, and membrane. To find out more about the features used, visit the UCI machine learning repository at <http://archive.ics.uci.edu/ml/machine-learning-databases/yeast/yeast.names>.

How to do it...

To perform clustering on part of the iris dataset, perform the following steps:

1. First, you need a dataset as the input for clustering. Use the yeast data, which was described earlier, and select the feature columns 2 to 10 (10th column is the class label) as follows:

```
> load("path/to/code/directory/yeast.rda") #  
located in the code file directory for the  
chapter, assign the path accordingly  
> myData <- yeast[,2:9]
```

2. Next, define the number of clusters you want for your dataset as follows (here, choose 4 because there are four localizations in the data):

```
> k <- 4
```

3. Start with k-means clustering. To perform k-means clustering, use the following function, taking the data and the number of clusters as input:

```
> kmeans_result <- kmeans(myData, k)
```

4. Now, check the number of proteins under each cluster by looking for and clustering the results as a table, as follows:

```
> table(kmeans_result$cluster)
```

5. Visualize the clusters along two selected variables of the dataset (here, the sepal length and width) as follows:

```
> par(mfrow=c(1,2))  
> plot(myData[c("mit", "gvh")], col =  
kmeans_result$cluster, main = "(A) Plot with  
clusters")
```

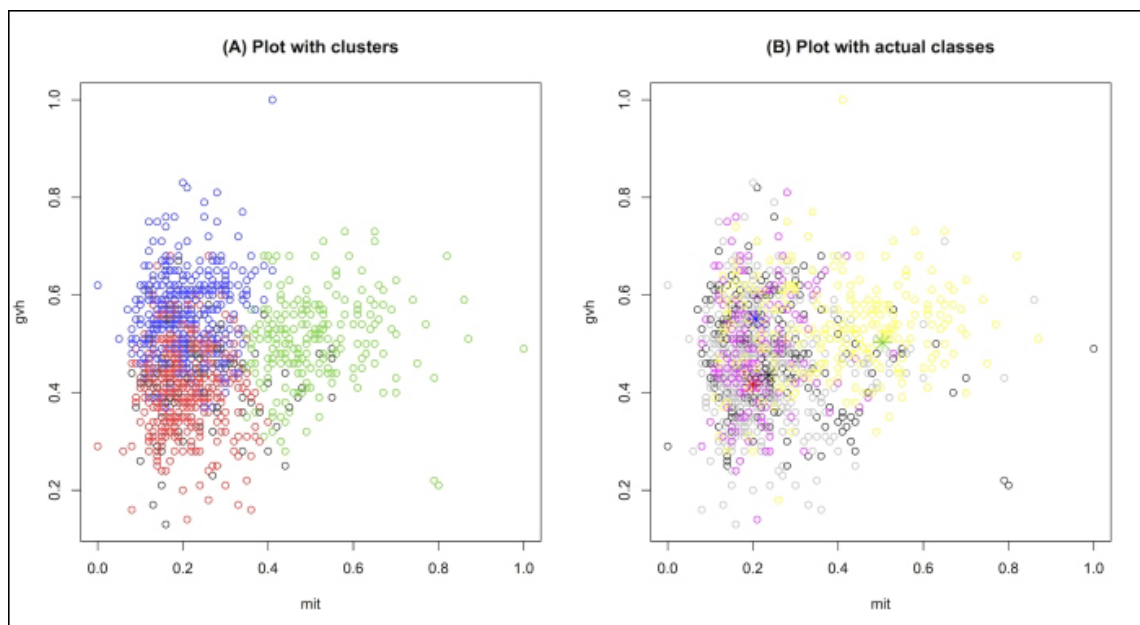
6. Plot the actual species and compare the clustering results with actual data by typing the following command:

```
> plot(myData[,c("mit", "gvh")], col =
yeast$class, main = "(B) Plot with actual
classes")
```

7. To observe the cluster centers in the plot, add them using the following command:

```
> points(kmeans_result$centers[,c("mit",
"gvh")], col = 1:4, pch = 8, cex=2)
```

In the following scatterplot, the colors represent the cluster computed in the dataset (A) and the actual classes in the input data (B). The centroids of the clusters are shown as stars—the colors correspond to the clusters in (A)—together with the data points in (B). In both plots, you can see the features `mit` and `gvh` along the x and y axes, respectively:



Clustering data points

8. The next step is to work with hierarchical clustering. For this, first create a distance matrix between the data points. However, for visualization purposes, reduce the dataset to 100 instances

(the data is provided separately under the name `yeast100.rda`). This can be done as follows:

```
>
load("path/to/code/directory/yeast100.rda")
# located in the code file directory for the
chapter, assign the path accordingly
> myData_100 <- yeast100[,2:9]
> myDist <- dist(myData_100) # might throw a
warning depending on the data but can be
ignored
```

9. The following command is used for hierarchical clustering:

```
> hc <- hclust(myDist, method="ave")
```

10. Now, check the clusters formed by typing the following command and cut the tree for three clusters:

```
> groups <- cutree(hc, k=k)
```

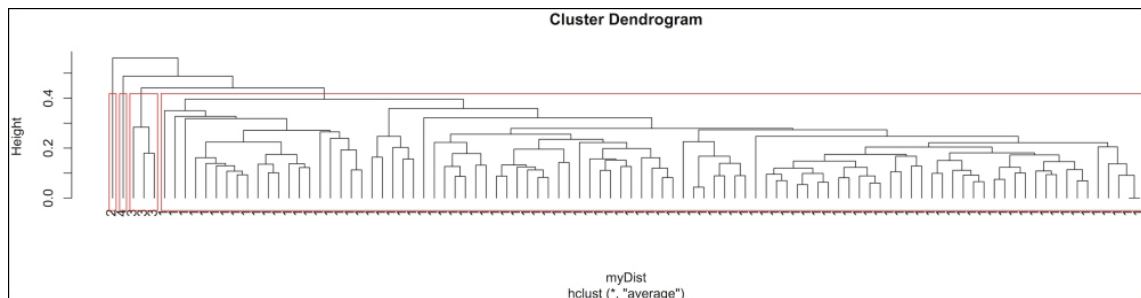
11. The tree thus formed can be plotted as a dendrogram, as follows:

```
> plot(hc, hang <- -1, labels=groups)
```

12. For better visualization, plot a rectangle around the cluster with the following function:

```
> rect.hclust(hc, k = 4, which = NULL, x =
NULL, h = NULL, border = 2, cluster = NULL)
```

In the following plot, you can see that the clusters have been boxed inside red boxes:



A dendrogram of the hierarchical cluster

How it works...

The clustering approaches shown previously use two different methods, namely k-means and hierarchical clustering. The first step is to prepare the data. The iris data is imported into the R session. We select the features in the data (columns 1 to 9, with the 10th being the class). We set the number of clusters to be 4 ($k=4$). The reason we use 4 is because we have four classes in the data as well. The class attribute from the data is removed; otherwise, it can cause bias in the clustering process. Once the data is ready, we apply the clustering algorithms implemented in the built-in R functions, such as `kmeans` and `hclust`.

The k-means algorithm randomly assigns clusters with the initial k to the input data points and then computes the means of the clusters, which are called centroids. This is updated and reassigned till the clustering converges.

The hierarchical clustering first computes the distances between the data points based on the feature measurements available. By default, the distance metrics used in the `dist` function are Euclidean (the other options are Manhattan, Maximum, and so on); note that these argument are passed in small letters such as `euclidean` and `manhattan` within the functions. This distance matrix is then used to build a dendrogram. Once the dendrogram is ready, an optimal cut is determined to get the desired number of clusters in the data (try different cluster numbers and see the results).

A tip while clustering, which can be rewarding, is to use hierarchical clustering first to visualize the clustering outcome and decide how many groups are to be categorized (by finding k), and then to use this k to compute clusters via other methods.

There's more...

There are other clustering methods in R, for example, the `som` and `kohonen` packages for self-organizing maps. Density-based clustering methods are implemented in `fpc` packages. Fuzzy clustering can be performed with `e1071` packages via the `cmeans` function. The `e1071` package is available from the CRAN repository at <http://cran.r-project.org/web/packages/e1071/index.html>.

See also

- The *A Survey of Clustering Data Mining Techniques* article by Berkhin at http://link.springer.com/chapter/10.1007/3-540-28349-8_2, which provides an overview of clustering techniques

Visualizing clusters

Visualizing clusters in a two- or three-dimensional space gives more intuitive information to the observer in terms of the location of the data points in the feature space and the segregation and aggregation of these data points. The visualization usually consists of the location of a data point along the feature axis. Another interesting plot for clustering is the silhouette plot, which depicts the quality of clustering. In this recipe, we work on a few types of visualizations of the clusters in k-means as we have already seen the visualization specific to hierarchical clustering.

Getting ready

What we need for the visualization is the data and the cluster information. Here, we use our previous recipe as the input.

How to do it...

Perform the following steps for cluster visualizations:

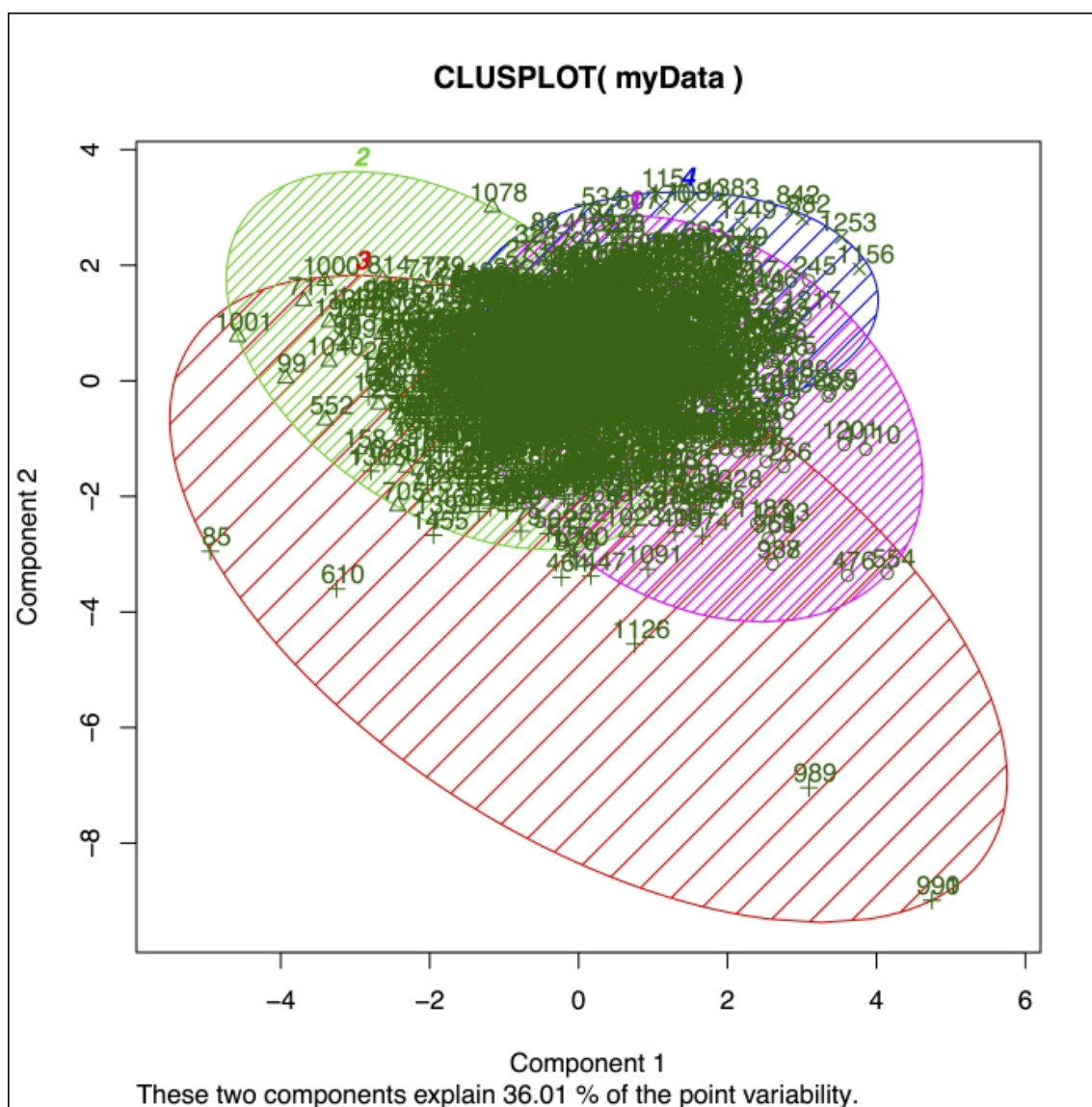
1. Create a simple plot of obtained clusters, plot the data against the clusters, and color the points based on the clusters that they are put into as follows:

```
> plot(myData, col = kmeans_result$cluster)
```

2. To perform shading on the clustering area, use the `fpc` package and shade the areas with corresponding colors, as follows:

```
> install.packages("fpc")
> library(fpc)
> clusplot(myData, kmeans_result$cluster,
color=T, shade=T, labels=2, lines=0)
```

In the following screenshot, the x and y axes represent the two components that explain the maximum amount of variability:



Clustering visualizations displaying clusters within shaded regions

3. To look at the centroid positions together with the data is a bit complicated. However, this can be achieved in the following way and requires the given [vegan](#) library:

```
> install.packages("vegan")
> library(vegan)
```


4. Now, assign the groups as the levels in the cluster results as follows:

```
> groups <-  
  levels(factor(kmeans_result$cluster))
```

5. After this, initialize a plot with the following command:

```
> ordiplot(cmdscale(dist(myData)), type = "n")  
# might issue a warning
```

6. Assign the colors to each cluster in the results, as follows:

```
> cols <-  
  rainbow(nlevels(factor(kmeans_result$cluster))  
)
```

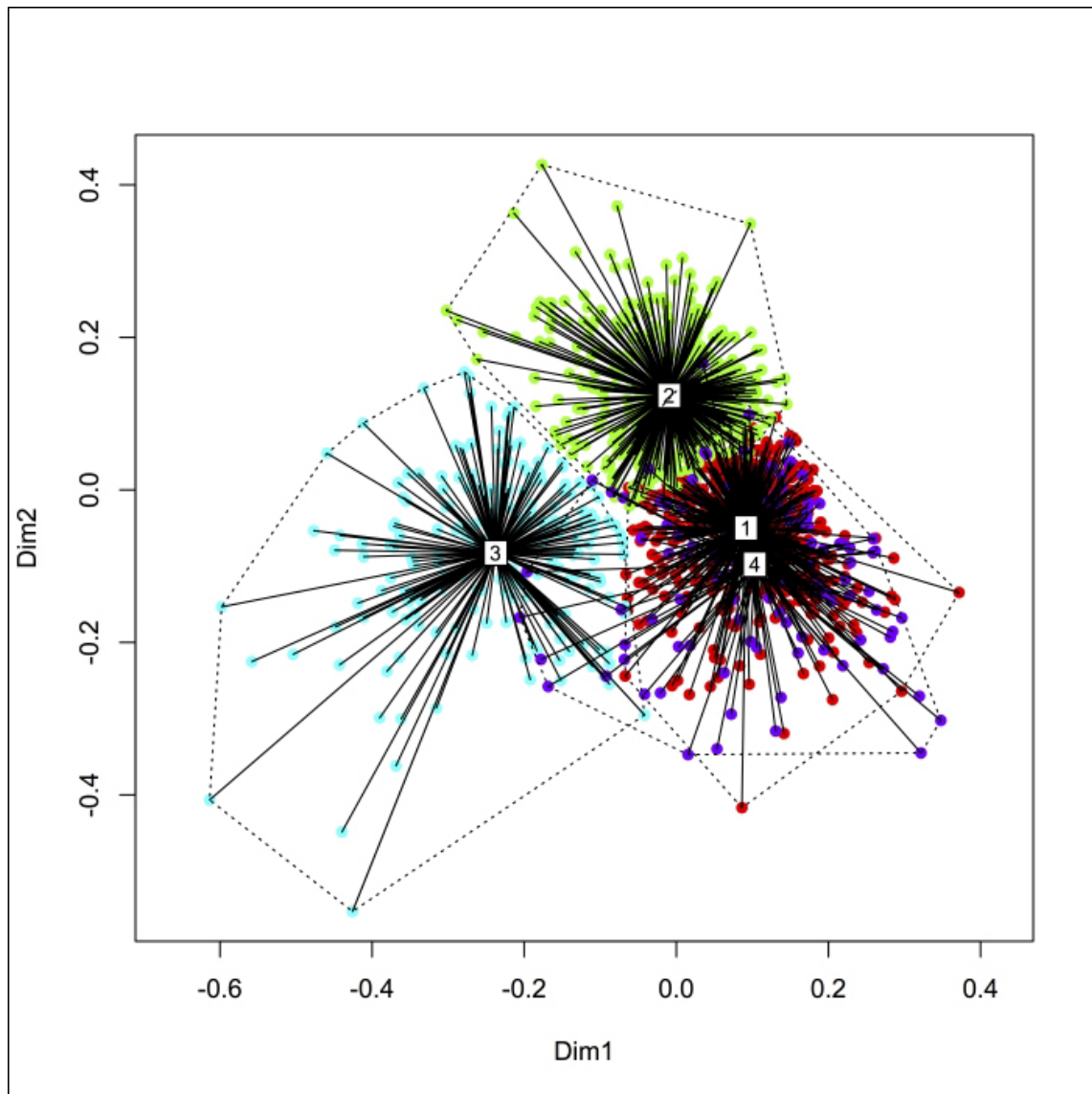
7. Now, plot each data point within the initialized plot as follows:

```
> for(i in seq_along(groups)) {  
  points(cmdscale(dist(myData))  
    [factor(kmeans_result$cluster) == groups[i],  
    ], col = cols[i], pch = 16)  
}
```

8. Then, add the spider web for each data point on the plot as follows:

```
> ordispider(cmdscale(dist(myData)),  
  factor(kmeans_result$cluster), label = TRUE)  
> ordihull(cmdscale(dist(myData)),  
  factor(kmeans_result$cluster), lty = "dotted")
```

In the following plot, the data points and their corresponding cluster centroids are connected by a spider web:



Clustering visualizations

9. Silhouette plots show the quality of clustering in terms of the segregation and aggregation of data points. It can be plotted as follows:

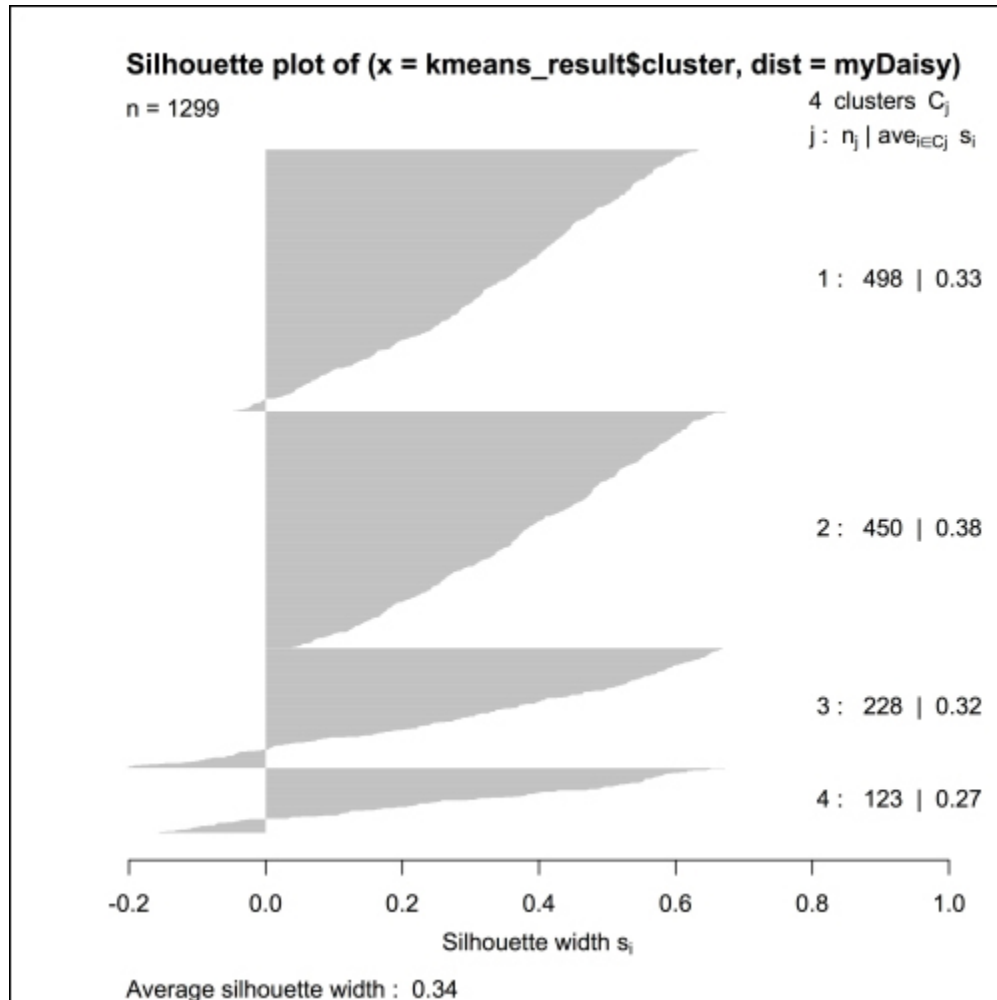
```
> myDaisy <- (daisy(myData))^2 # might issue a
warning
> mySil <- silhouette(kmeans_result$cluster,
myDaisy)
> plot(mySil)
```

How it works...

The simple plot explained in step 1 prepares a plot of the entire data matrix and simply colors the points depending on the clusters assigned to them. Using the `fpc` package, as seen in step 2, we encircle the region for each cluster around its centroid. The third plot that shows the centroid and spider webs from the data point to the centroid of the cluster uses the `vegan` library to plot functions. The `vegan` library has several utilities to compute distance matrices. Here, the `dist` function computes the pairwise distances within the data. The `cmdplot` function computes the classical, multidimensional scaling (principal coordinates analysis) of a data matrix and returns a set of points such that the distances between the points are approximately equal to the dissimilarities. The initial blank plot is filled with data points (`type=n`), as shown in step 7, followed by which the spider webs between the data points and the cluster centroid are added. This plot can be used to visualize the spread of data in the feature space. The silhouette plot represents the following aspects:

- The number of clusters, each depicted by a horizontal grey area.
- The height of these grey regions is proportional to the number of data point within the corresponding cluster.
- The grey area is formed by a collection of data within each cluster represented by lines that extend horizontally to its **silhouette width (SW)**, representing how well does a point fits in one cluster compared to other. The higher the SW, better is the fit.
- The silhouette index is the mean similarity of each point to its own cluster minus the mean similarity to the next most similar cluster.

The silhouette plot for k-means clustering is as follows:



Finally, the silhouette plot uses the cluster library of R to compute distances in the dataset as well as create the silhouette object for the cluster results. We can then plot the results using the `plot` function of R. The silhouette index provides a measure of how well each object lies within its cluster. Here, the plot shows the highest silhouette index for cluster 3.

See also

- The *Silhouettes: A graphical aid to the interpretation and validation of cluster analysis* article by Peter J. Rousseeuw at <http://www.sciencedirect.com/science/article/pii/0377042787901257>, which provides information about the silhouette index and plot

Supervised learning for classification

Like clustering, classification is also about categorizing data instances, but in this case, the categories are known and are termed as class labels. Thus, it aims at identifying the category that a new data point belongs to. It uses a dataset where the class labels are known to find the pattern. Classification is an instance of supervised learning where the learning algorithm takes a known set of input data and corresponding responses called class labels and builds a predictor model that generates reasonable predictions for the class labels in the unknown data. To illustrate, let's imagine that we have gene expression data from cancer patients as well as healthy patients. The gene expression pattern in these samples can define whether the patient has cancer or not. In this case, if we have a set of samples for which we know the type of tumor, the data can be used to learn a model that can identify the type of tumor. In simple terms, it is a predictive function used to determine the tumor type. Later, this model can be applied to predict the type of tumor in unknown cases.

There are some do's and don'ts to keep in mind while learning a classifier. You need to make sure that you have enough data to learn the model. Learning with smaller datasets will not allow the model to learn the pattern in an unbiased manner and again, you will end up with an inaccurate classification. Furthermore, the preprocessing steps (such as normalization) for the training and test data should be the same. Another important thing that one should take care of is to keep the training and test data distinct. Learning on the entire data and then using a part of this data for testing will lead to a phenomenon called overfitting. It is always recommended that you take a look at it manually and understand the question that you need to answer via your classifier.

There are several methods of classification. In this recipe, we will talk about some of these methods. We will discuss **linear discriminant analysis (LDA)**, **decision tree (DT)**, and **support vector machine (SVM)**.

Getting ready

To perform the classification task, we need two preparations. First, a dataset with known class labels (training set), and second, the test data that the classifier has to be tested on (test set). Besides this, we will use some R packages, which will be discussed when required. As a dataset, we will use approximately 2300 gene from tumor cells. The data has ~83 data points with four different types of tumors. These will be used as our class labels. We will use 60 of the data points for the training and the remaining 23 for the test. To find out more about the dataset, refer to the *Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks* article by Khan and others (<http://research.nhgri.nih.gov/microarray/Supplement/>). The set has been precompiled in a format that is readily usable in R and is available on the book's web page (code files) under the name `cancer.rda`.

How to do it...

To classify data points based on their features, perform the following steps:

1. First, load the following MASS library as it has some of the classification functions:

```
> library(MASS)
```

2. Now, you need your data to learn and test the classifiers. Load the data from the code files available on the book's web page (`cancer.rda`) as follows:

```
> load("path/to/code/directory/cancer.rda") #  
located in the code file directory for the  
chapter, assign the path accordingly
```

3. Randomly sample 60 data points for the training and the remaining 23 for the test set as follows—ensure that these two datasets do not overlap and are not biased towards any specific tumor type (random sampling):

```
> train_row <- sample(1:83, 60)  
> train <- mldata[train_row,] # use sampled  
indexes to extract training data  
> test <- mldata[-train_row,] # test set is  
select by selecting all the other data points
```

4. For the training data, retain the class labels, which are the tumor columns here, and remove this information from the test data. However, store this information for comparison purposes:

```
> testClass <- test$tumor  
> test$tumor <- NULL
```

5. Now, try the linear discriminate analysis classifier, as follows, to get the classifier model:

```
> myLD <- lda(tumor ~ ., train) # might issue  
a warning
```

6. Test this classifier to predict the labels on your test set, as follows:

```
> testRes_lda <- predict(myLD, test)
```

7. To check the number of correct and incorrect predictions, simply compare the predicted classes with the `testClass` object, which was created in step 4, as follows:

```
> sum(testRes_lda$class == testClass) #  
correct prediction
```

```
[1] 19
> sum(testRes_lda$class != testClass) #
incorrect prediction
[1] 4
```

8. Now, try another simple classifier called DT. For this, you need the `rpart` package:

```
> library(rpart)
```

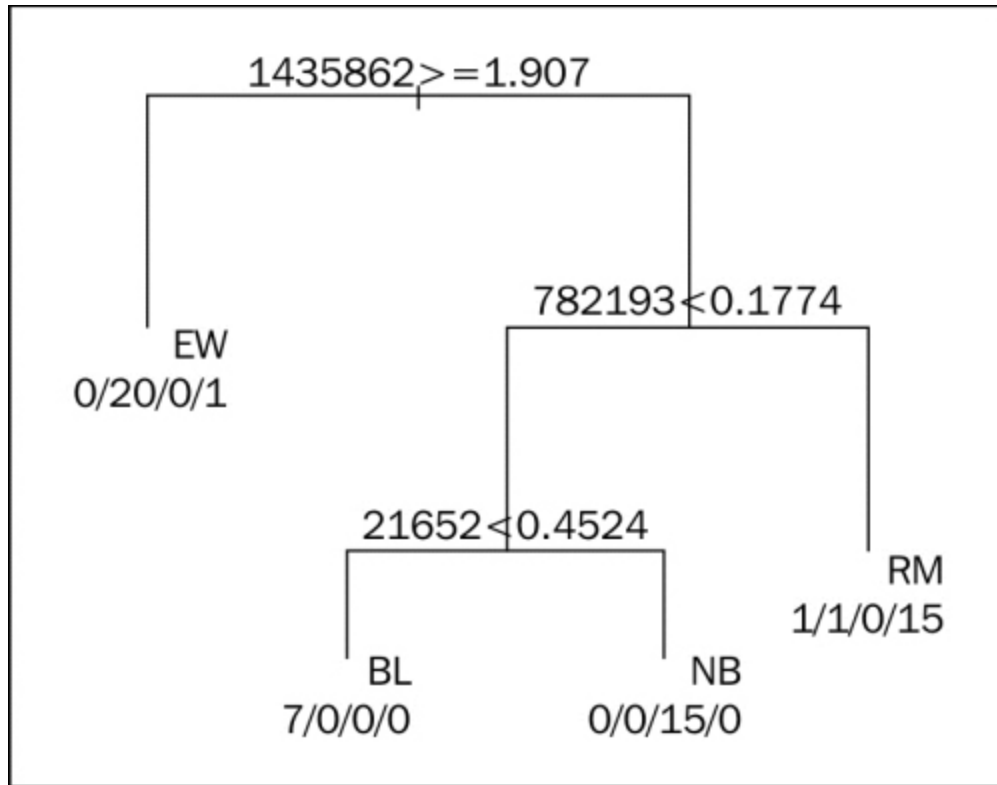
9. Create the decision tree based on your training data, as follows:

```
> myDT <- rpart(tumor~ ., data = train,
control = rpart.control(minsplit = 10))
```

10. Plot your tree by typing the following commands, as shown in the next diagram:

```
> plot(myDT)
> text(myDT, use.n=T)
```

The following screenshot shows the cut off for each feature (represented by the branches) to differentiate between the classes:



The tree for DT-based learning

11. Now, test the decision tree classifier on your test data using the following prediction function:

```
> testRes_dt <- predict(myDT, newdata= test)
```

12. Take a look at the species that each data instance is put in by the predicted classifier, as follows (1 if predicted in the class, else 0):

```
> classes <- round(testRes_dt)
> head(classes)
  BL EW NB RM
4   0  0  0  1
10  0  0  0  1
15  1  0  0  0
16  0  0  1  0
18  0  1  0  0
21  0  1  0  0
```

13. Finally, you'll work with SVMs. To be able to use them, you need another R package named `e1071` as follows:

```
> library(e1071)
```

14. Create the `svm` classifier from the training data as follows:

```
> mySVM <- svm(tumor ~ ., data = train)
```

15. Then, use your classifier, the model (`mySVM` object) learned to predict for the test data. You will see the predicted labels for each instance as follows:

```
> testRes_svm <- predict(mySVM, test)
> testRes_svm
```

How it works...

We started our recipe by loading the input data on tumors. The supervised learning methods we saw in the recipe used two datasets: the training set and test set. The training set carries the class label information. The first part in most of the learning methods shown here, the training set is used to identify a pattern and model the pattern to find a distinction between the classes. This model is then applied on the test set that does not have the class label data to predict the class labels. To identify the training and test sets, we first randomly sample 60 indexes out of the entire data and use the remaining 23 for testing purposes.

The supervised learning methods explained in this recipe follow a different principle. LDA attempts to model the difference between classes based on the linear combination of its features. This combination function forms the model based on the training set and is used to predict the classes in the test set. The LDA model trained on 60 samples is then used to predict for the remaining 23 cases.

DT is, however, a different method. It forms regression trees that form a set of rules to distinguish one class from the other. The tree learned on a training set is applied to predict classes in test sets or other similar datasets.

SVM is a relatively complex technique of classification. It aims to create a hyperplane(s) in the feature space, making the data points separable along these planes. This is done on a training set and is then used to assign classes to new data points. In general, LDA uses linear combination and SVM uses multiple dimensions as the hyperplane for data distinction. In this recipe, we used the `svm` functionality from the `e1071` package, which has many other utilities for learning.

We can compare the results obtained by the models we used in this recipe (they can be computed using the provided code on the book's web page).

There's more...

One of the most popular classifier tools in the machine learning community is WEKA. It is a Java-based tool and implements many libraries to perform classification tasks using DT, LDA, Random Forest, and so on. R supports an interface to the WEKA with a library named `RWeka`. It is available on the CRAN repository at <http://cran.r-project.org/web/packages/RWeka/>.

It uses `RWekajars`, a separate package, to use the Java libraries in it that implement different classifiers.

See also

- The *Elements of Statistical Learning* book by Hastie, Tibshirani, and Friedman at http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf, which provides more information on LDA, DT, and SVM

Probabilistic learning in R with Naïve Bayes

Until now, we have gone through some nonprobabilistic classification methods. However, it is important to have a look at the probabilistic methods of classification and data mining. To maintain simplicity, we describe the Naïve Bayes method for classification in this recipe.

The Naïve Bayes method is an extremely simple, probabilistic classification method. It is based on the assumption of independent features in datasets. This means that the presence or absence of a particular feature is independent (unrelated) from the presence (or absence) of any other feature. For illustration purposes, let's imagine that a disease D can be caused by the mutation of three genes, namely a, b, and c. Assuming that these genes are independent, the information from these genes may be combined in a Naïve Bayes classifier with a node for disease D as a parent to the independent variables a, b, and c.

Naïve Bayes classifiers require small training data to estimate the parameters (the means and variances of the variables) necessary for classification. Since it is based on the assumption that the variables are independent, only the variances of the variables for each class need to be computed and not the entire covariance matrix.

Getting ready

The prerequisites for the recipe are the same as those of the earlier recipes (for other classification recipes)—the training set and test set. However, we need the R library, [e1071](#), in addition to this. We continue to use the training and test sets created for previous classifiers.

How to do it...

To perform a Naïve Bayes classification, perform the following steps:

1. First, load the `e1071` library into the R session as follows:

```
> library(e1071)
```

2. Now, learn the classifier model based on the training set using the `naiveBayes` function of the `e1071` package as follows:

```
> model <- naiveBayes(tumor ~ , data = train)
```

3. Once you have your trained model, use it to predict the classes in the test set as follows:

```
> testRes_nb <- predict(model, test)
```

4. Compare the predicted classes with the actual classes by forming a contingency table as follows:

```
> table(testRes_nb, testClass)
```

```
testClass
testRes_nb BL EW NB RM
BL      3   0   0   0
EW      0   8   0   0
NB      0   0   3   0
RM      0   0   0   9
```

How it works...

The `e1071` library of R has the Naïve Bayes method implemented within it. The method computes the conditional posterior probabilities of the class based on the Bayes rule, given the feature values that are considered as independent of each other. These probability values create the classifier model, which is then used to predict the classes in the test set or new data points. The formula argument in the function represents the class labels and the variables that it depends on. These values must be present in the training data;

however, the class labels are missing in the test data. (Remove the label on purpose to verify the prediction power of the classifier.)

See also

- The *Understanding Probabilistic Classifiers* article by Garg and Roth at http://link.springer.com/chapter/10.1007%2F3-540-44795-4_16, which provides more details on probabilistic classifiers

Bootstrapping in machine learning

Learning from subsamples is one way to test the robustness of a machine learning algorithm and improve the accuracy of a learning algorithm. It is a simple approach for accuracy estimation and provides the bias or variance of the estimator. One such approach is bootstrapping. This is the practice of learning and estimating properties from subsamples of the learning data, as well as iteratively improving the performance. This section will explain bootstrapping with respect to R programming.

Getting ready

To bootstrap with any function, we need the function (for example, the type of classifier we want to use) and the dataset that we want to perform the bootstrapping on. In this recipe, we will use the iris dataset for this purpose.

How to do it...

Perform the following steps for bootstrapping using the iris data:

1. First, we need our dataset, and as mentioned earlier, we use the iris data for our purpose, as follows:

```
> data(iris)
> myData = iris[c(1:150),]
```

2. For bootstrapping, create a function that will be bootstrapped across the dataset. The function is an SVM classifier, and it returns the number of **true positive (TP)**, which serves as your performance value, as follows (this has been selected for

simplicity; a more complete measurement of performance discussed in the upcoming recipes can be used as well):

```
> corPred <- function(data, label, indices){
  train = myData[indices,] # indexes for
training data
  test = myData[-indices,] # indexes for
test data
  testClass = test [,label] # assigns class
labels (species)
  colnames(train)[ncol(train)]="Class" mySVM
= svm(Class ~ .,data = train, cost = 100,
gamma = 1) # learn model using SVM
  myPred = predict(mySVM, test) #
prediction on test set
  TP = sum(myPred == testClass) # calculate
True positives
  return(TP)
}
```

3. Now, write another function to bootstrap with the data and write the number of bootstraps to the function as input, as follows:

```
> myboot <- function(d, label, iter){
  bootres = c()
  for(i in 1:iter){
    indices = sample(1:nrow(d),
floor((2*nrow(d))/3)) # samples indexes
    res = corPred (d, label, indices) # runs
corPred function
    bootres = c(bootres, res) # append
results
  }
  return(list(BOOT.RES = bootres, BOOT.MEAN =
mean(bootres), BOOT.SD = sd(bootres)))
}
```

4. Then, run your bootstrap function as follows:

```
> res.bs <- myboot(d = myData,
label="Species",iter = 10000)
```


The results consist of the value for each iteration, that is, the mean of all of the bootstraps and the corresponding standard deviation.

How it works...

The major part of the recipe has two functions in steps 2 and 3. The function in step 2 extracts the data for the training and test sets from the original data and uses the data for learning and prediction, respectively. A command has been added to compute the total number of true positives as a performance measure. It is very simple to modify it to return other measurements such as sensitivity and specificity (this will be discussed in the forthcoming recipes). The second function in step 3 actually does the bootstraps by sampling data point indices for the training and test sets used in step 2. It also creates a vector of the performance measure by appending results from the iterations (the `BOOT.RES` element in the returned list). Thus, the overall value returned is a list that consists of the actual results (vector of measurements) and their mean and standard deviation. A higher mean and lower standard deviation indicates a consistent classifier with more accurate classifiers (in this case, it is TP). The fourth step simply calls the function in step 3 with appropriate arguments.

Cross-validation for classifiers

Getting a learning-based model is not enough to get optimal results. The question that remains is how well the model performs when applied to make new predictions for unseen data instances, which we refer to as predictive performance in technical terms. One way is to hold out a part of the available data as a test set. After training, this test set can be used to test the performance of the learned model. This basic idea for a whole class of model evaluation methods is termed cross-validation. **Cross-validation (CV)** is useful to overcome the problem of overfitting, which refers to a condition where the model requires more information than the data can provide.

There are several approaches to do CV, the simplest being what we just described, that is, holding a part of training data; the other popular methods are k-fold cross-validation and leave-one-out cross-validation. The former approach can be seen as an iteration of holding out a part of the dataset. Here, we divide the data into k subsets and iterate the learning and testing k number of times using $k-1$ sets for training and the k th set for testing (hence, k-fold cross-validation). The latter approach—leave-one-out—can be seen as k-fold cross-validation, where the value of k is set to the data instance n ($k = n$). This yields $n-1$ instances for training and the n th instance for testing. The smaller test set can lead to higher variability in predictive performance in the case of leave-one-out methods (fewer sample combinations), besides being more likely to be computationally expensive. In such cases, we go with k-fold cross-validation.

The question left to answer is: how do we define the number k ? This depends a lot on how many data instances we have; with a higher number of data instances available, we have more freedom to choose k . The most accepted values for k are 5 and k . However, it is advisable that you run iterations of a k-fold cross-validation for a robust model. Furthermore, we can include the feature selection

within CV, forming a nested k-fold cross-validation called the nested loop cross-validation. For the purpose of this book, we will not elaborate on this here.

This recipe is about performing a k-fold cross-validation in R for a learning method of our choice.

Getting ready

This CV recipe will need data and a learning algorithm. For this purpose, we use the iris data, but only the first 100 data instances. This is done to ensure that we have only two classes in our set to keep things simple. The learning algorithm is a matter of choice; we use the SVM learning from the [e1071](#) package here.

How to do it...

To do an n-fold cross-validation, perform the following set of steps:

1. First, load the [e1071](#) package in your R session as follows:

```
> library(e1071)
```

2. Create the dataset for learning and testing purposes, and use the iris data again and extract the first 100 instances from it as follows:

```
> data(iris)
> myData <- iris[1:100,]
```

3. Make a small modification in the extracted data to restrict the number of levels in the class labels to 2 with the following command:

```
> myData$Species <-
  factor(as.character(myData$Species))
```

4. Now, set the number of folds for the CV, that is, choose the number of folds, as follows:

```
> k=10
```

5. Then, create indices based on the number of folds for the CV by sampling from the folds (k) with the replacement, and create the sequence vectors for the folds ranging from 1 to k, as follows:

```
> index <-  
  sample(1:k,nrow(myData),replace=TRUE)  
> folds <- 1:k
```

6. Then, initialize a data frame to store your CV results as follows:

```
> myRes <- data.frame()
```

The following chunk of code is the main loop within which the data sampling and learning and CV actually occurs:

```
> for (i in 1:k){  
  training = subset(myData, index %in% folds[-  
i]) # create training set  
  test = subset(myData, index %in% c(i)) #  
  create test set  
  mymodel = svm(training$Species ~.,  
data=training) # train model  
  actual = test[,ncol(test)] # get actual  
  labes  
  temp = data.frame(predict(mymodel, test [, -  
ncol(test)])) # run model on test set  
  colnames(temp) = "Predicted"  
  results = data.frame(Predicted=temp,  
Actual=actual) # create data.frame for results  
  myRes = rbind(myRes, results) # append results  
  for each iteration  
}
```

7. Finally, prepare a contingency table for the overall CV results by typing the following command:

```
> table(myRes)
```

How it works...

Steps 1 to 3 prepare the data for the entire process. The next steps set the parameters, such as the number of folds for the CV. In this recipe, CV is actually done in step 7. Here, within the loop, the training and test sets are created based on the number of folds, the sets to train and test are sampled, and then the training and test sets are extracted. The SVM model is then learned from the training set and used to predict on the test set. The predicted and actual results are combined during the iterations and stored in the object `myRes`.

There's more...

You can use several packages to perform CV during your learning processes. R libraries, such as `randomForest` and `nlcv`, allow this simultaneously while the learning process by specifying particular arguments while running the learning function. Note that the `randomForest` package is also the name of a classification method. The package is not only meant for classification and regression based on a forest of trees but also carries utilities for CV. However, in this recipe, we used our own set of functions; they can be used with any choice of learning or statistical testing process.

See also

- The book titled *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman at http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf, which provides more details on CV
- The *Nested Loop Cross Validation for Classification using nlcv* article by Talloen at https://r-forge.r-project.org/scm/viewvc.php/*checkout*/pkg/nlcv/inst/doc/nlcv.pdf

[?root=nlc](#), which provides a description of nested loop cross-validation

Measuring the performance of classifiers

Measuring the accuracy of the learning algorithms is as important as the algorithms themselves. These measurements of performance define how accurately the data points have been classified or clustered. In this recipe, we will describe the computation of some of these measurements in R.

Getting ready

To start with, we need some of the results from recipes. We learned about the performance measure for clustering in the *Visualizing clusters* recipe when we talked about silhouette plots. In this recipe, we compute some of the measurements of performance to assess how good is our classifier. We use a binary classification here (with two class labels) for simplicity. Working with multiple classes will be explained in the *See also* section. The required R packages will be introduced in the next section.

How to do it...

To measure the performance of classifiers, perform the following steps:

1. First, install and load the `caret` library as follows:

```
> install.packages("caret")  
> library(caret)
```

2. Prepare your data and classifiers using the following function:

```
> data (iris)  
> myData <- iris
```

```

> indices <- sample (1:nrow(myData), floor
((2*nrow(myData))/3))
> train <- myData [indices,]
> test <- myData [-indices,] testClass <- test
[, "Species"]
> mySVM <- svm (Species ~ ., data = train,
cost = 100, gamma =1)

```

3. To compute the performance of a classifier, first get the results from the predictions of the SVM classifier and the actual class labels for the data points, as follows:

```

> myPred <- predict(mySVM, test)
> myLabels <- testClass

```

4. Then, compute the sensitivity and specificity of your learning method using the following function from the `caret` package:

```

> myCM <- confusionMatrix(myPred, myLabels)

```

5. Extract the various confusion matrices from the object created earlier using the performance function and by specifying the performance measure as follows:

```

> CMtable <- myCM$table

```

6. Compute the sensitivity, specificity, and so on, of the classifier for every class as follows:

```

> myPerf2 <- myCM$byClass

```

How it works...

The working of this recipe is very simple. The sensitivity and specificity functions from the `caret` package compare the predicted and original class labels. The `prediction` function from the `ROCR` package accepts two major arguments in terms of predicted classes and original classes but in numerical terms, which has been shown in the following step. This creates a prediction object that has all the

performance measures. The measurement of interest is extracted with the performance function. This function accepts the prediction object and the name of the measurement as the input argument, as depicted in steps 3, 4, and 5.

We can also use the [ROCR](#) package if we have only binary classes. For this, we convert the factor class label into a numeric type and then use the prediction function from the [ROCR](#) package as follows:

```
> preds <- prediction(as.numeric(myPred),  
  as.numeric(myLabels))
```

Visualizing an ROC curve in R

ROC curves provide the complete measure of performance. They are of Specificity versus 1-Sensitivity plots represented on the x and y axes, respectively. The visualization of these curves can provide the entire idea of the accuracy (sensitivity and specificity) of the classifier in one plot. By looking at these plots, you will get a clear picture of the area under the curve; the larger the area, the better the performance. The higher the area under this curve, the higher the sensitivity and specificity. In fact, the top-right region in the ROC curve is also termed as classifier heaven. Here, we intend on creating such visualizations.

Getting ready

We need our classifier results as input for the computation. We simply take the results of the SVM classifier to avoid repetition of such results.

How to do it...

To plot visualizations of the ROC curve for the methods explained earlier, perform the following steps:

1. First, load the `ROCR` library as follows:

```
> library(pROC)
```

2. Then, compute the `roc` object as follows (here, we do it for the LDA classifier of the tumor data):

```
> roc_lda <-  
plot.roc(as.numeric(testRes_lda$class),  
as.numeric(testClass))
```

3. Plot these values to get the ROC curve as follows:

```
> plot(roc_lda, col="grey")
```

4. For comparison purposes, compute the `roc` object for another classifier (here, we use `svm`) as follows:

```
> roc_svm <- plot.roc(as.numeric(testRes_svm),  
  as.numeric(testClass))
```

5. Plot the `roc` object for `svm` to the existing plot as follows:

```
> plot(roc_lda, col="grey")  
> lines(roc_svm, col="black")  
> legend("bottomright", c("svm", "lda"), fill  
  = c("black", "grey"))
```

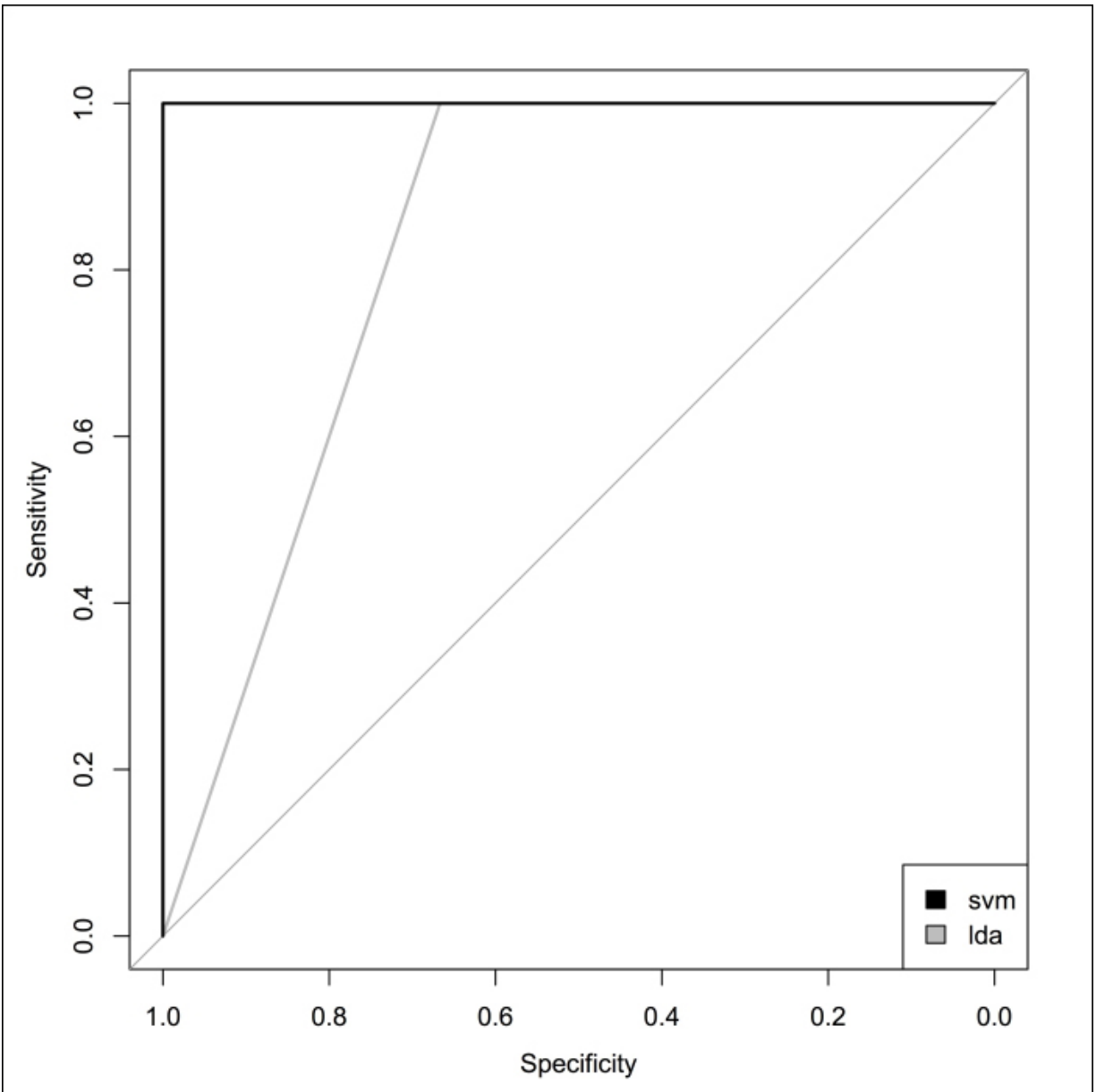
The `pROC` package also has a function that can specifically deal with multiple classes; use it, in this case, as follows:

```
> multiclass.roc(as.numeric(testRes_svm),  
  as.numeric(testClass), percent=TRUE)
```

How it works...

In the following diagram, the plot shows a perfect classification by the SVM classifier. This is an ideal case, with the area under the curve equal to 1. The ROC curve for the method explained in the recipe uses the `pROC` package and the learning approach used is `svm`. The graph shows an area under the curve that is equal to 1. This is the ideal **area under curve (AUC)** for machine learning and shows an accurate prediction for each data point (which rarely happens in real life).

The diagonal line is just a representation of 50 percent of AUC and does not belong to any classifier. The following diagram shows the ROC plot described previously:



As stated in the previous recipe, we extract the two performance measures, namely the **true positive rate (tpr)** and the **false positive rate (fpr)** from the `prediction` object. These measures are used to plot the ROC curve. The `plot.roc` function of the `pROC` package does this in a single step.

There's more...

Another interesting library to create ROC plots is the `pROC` package. This recipe works well for binary classification as well. However, it

could get complicated if we have multiple classes. For this purpose, we can use the `pROC` package. The `pROC` package provides the option of a multiclass ROC curve. It presents the mean AUC value for all of the classes. The function to be used is `multiclass.roc`. For further details, type the following commands:

```
> library(pROC)
?multiclass.roc
```

Biomarker identification using array data

Until now, we have seen some machine learning methods in R. It will be interesting to look at their application in bioinformatics. In life sciences, comparing groups of individuals to find significant differences is becoming more and more important. It might be a measurement or a substance that indicates the biological condition and can serve as biomarkers.

Let's imagine that we have gene expression data from patients and healthy controls. Genes identified based on this expression data can differentiate between people with diseased and healthy samples. These genes can serve as indicators or biomarkers for a biological state, such as a disease. This recipe talks about such an application to look for biomarkers using some learning techniques and statistical methods.

Getting ready

We need a few preparations before we go ahead. The first is the dataset where we will look for the biomarker. We also need an R package called `BioMark`. In this recipe, we will generate the dataset artificially.

How to do it...

The approach to find biomarkers from a dataset is illustrated as follows:

1. Start by installing and loading the `BioMark` library as follows:

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

```
> library(BioMark)
```

2. Once you have loaded the library, use the `gen.data` function to simulate a dataset for use. By default, it simulated the data for five biomarkers, but this can be altered with the `nbiom` argument as follows:

```
> simdata <- gen.data(ncontrol = 10, nvar =  
500, nsimul=1, group.diff = 1.5)
```

3. Check the contents of the dataset as follows:

```
> class(simdata)  
> str(simdata$X)
```

4. The real biomarkers in the generated dataset are the first five variables; therefore, you can assign it to an R object for evaluation purposes, as follows:

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

5. Now, use the `get.biom` function to look for the biomarkers in the dataset. Use the `pls` method for this purpose as follows:

```
> myBiom <- get.biom(X = simdata$X[1:20,,1], Y  
= simdata$Y, fmethod = "pls", type = "HC")
```

6. Once you have selected the biomarkers, take a look at the ranked variable's indices with the following command:

```
> selection(myBiom)
```

7. Once you have the results, observe the performance measure in terms of the ROC curve, as follows, for the top five biomarkers selected from the results:

```
> myROC <- ROC(1/coef(myBiom)$pls[[1]][1:5],  
myrealMarkers)  
> plot(myROC)
```

How it works...

The `BioMark` library selects a biomarker from a dataset based on statistical tests or from regression-based methods and their corresponding coefficients. The `gen.data` function generates a dataset with a defined amount of control (`ncontrol`), and by default, the same number of treated instances (to change this, use the `ntreated` argument). This generates a certain number of datasets. We set this number to 1 by setting the `nsimul` argument to be equal to 1. The result is a list of data class labels and a number of biomarkers (the default is 5). In order to use the dataset, we extract the data to be used by selecting only one dataset, by specifying the proper array indices (see the `x` argument in step 5). The `get.biom` function does the required test or learning to filter out the set of biomarkers in the dataset. It is possible to use three different methods for inferring the biomarkers. Besides the **partial least square (PLS)** regression that we use here, the other possible methods are PCR and t-tests. The value returned is an ordered list of indices of the variables (in our case, genes or proteins) that are found to act as biomarkers in the dataset.

There's more...

This recipe uses an existing package (`BioMark`). However, the process can be customized for a more tailored method, such as SVM or other techniques. This will involve a new implementation, which we will not present as a recipe here. Nevertheless, it is worthwhile to look at some of the approaches to do this in the literature.

Appendix A. Useful Operators and Functions in R

The commonly used operators in R are tabularized as follows:

Operator	Purpose	Example
=	Assignment	<pre>> sum(a = 5, b = 6) [1] 11</pre>
<-	Assignment	<pre>> a <- 5</pre>
+	Addition	<pre>> 10 + 1 [1] 11</pre>
-	Subtraction	<pre>> 10 - 1 [1] 9</pre>
*	Multiplication	<pre>> 10 * 2 [1] 20</pre>
/	Division	<pre>> 10 / 2 [1] 5</pre>
<code>^</code> or <code>**</code>	Exponentiation	<pre>> 10 ^ 2 [1] 100</pre>
<code>x %% y</code>	Modulus (x mod y)	<pre>> 11 %% 2 [1] 1</pre>

Operator	Purpose	Example
x %/% y	Integer division	> 10 %/% 3 [1] 3
<	Less than	> 3 > 5 [1] FALSE
<=	Less than or equal to	> 3 <= 5 [1] TRUE
>	Greater than	> 3 < 5 [1] TRUE
>=	Greater than or equal to	> 3 >= 5 [1] FALSE
==	Check equality	> 3 == 5 [1] FALSE
!=	Not equal to	> 3 != 5 [1] TRUE
!	NOT	> !a
	OR	> x y
&	AND	> x & y

Operator	Purpose	Example
<code>%*%</code>	Matrix product	<pre> > x=matrix(c(1,2,3,4),2,2) > y=matrix(c(2,1,1,1),2,2) > x %*% y [,1] [,2] [1,] 5 4 [2,] 8 6 </pre>
<code>%in%</code>	Matching operator	<pre> > 5 %in% c(1:10) [1] TRUE </pre>

The following table shows the useful generic R functions:

Function	Main arguments	Details
<code>as.*()</code>	Object to be created or tested	Forces an object to belong to a class; examples include the conversion of <code>data.frame</code> to matrix, but it can be used for many other classes: <pre> > f <- data.frame(a=c(1,2,3), b=c(4,5,6)) > class(f) [1] "data.frame" > d <- as.matrix(f) > class(d) [1] "matrix" </pre>

Function	Main arguments	Details
<code>barplot()</code>	Vector or matrix for which a plot should be produced	Creates a bar plot (horizontal or vertical by default), for example, <code>barplot(c(a=32, b=17, c=11))</code>
<code>boxplot()</code>	The values to be plotted in the box plot	Produces boxplots for the given values, for example, <code>boxplot(x=c(1,3,4,5,6,))</code>
<code>c()</code>	Vector or list entries	Concatenates the object vector or list as follows: <code>> x <- c(1,2,3,4)</code>
<code>cat()</code>	Concatenate and print	Prints the argument.
<code>cbind</code>	Vectors, matrices, or data frames to be combined	Bind the data along the columns as <code>data.frame</code> .

Function	Main arguments	Details
<code>ceiling()</code>	A numerical argument	Takes a numerical argument and returns the numeric vectors containing the smallest integers, not less than the elements of the original argument, with the following command: <pre>> ceiling(57.76) [1] 58</pre>
<code>colnames()</code>	Data frame or matrix	Returns the names of the columns of a data frame or a matrix-like object. If column names are not available, <code>colnames()</code> returns a <code>NULL</code> value.
<code>data.frame()</code>	Values to be accepted as data frame	To create a data frame class of R objects, use the following command: <pre>> f <- data.frame(a=c(1,2,3) , b=c(4,5,6))</pre>
<code>data.matrix()</code>	The data frame	To convert data frame to a matrix, use the following commands: <pre>> f <- data.frame(a=c(1,2,3) , b=c(4,5,6)) > d <- data.matrix(f)</pre>

Function	Main arguments	Details
<code>data()</code>	Loads a dataset in the library	Loads a library dataset in the R session as follows: <pre>> data(car90, package = "rpart")</pre>
<code>dev.off()</code>	The device (default current device)	Closes the graphics output device.
<code>dim()</code>	The matrix or data frame	Returns the dimensions of the data frame or the matrix (array) as follows: <pre>> f <- data.frame(a=c(1,2,3), b=c(4,5,6)) > dim(f) [1] 3 2</pre>
<code>dimnames()</code>	The matrix or data frame	Returns the names of the various dimensions in the data frame or the matrix (array): <pre>> f <- data.frame(a=c(1,2,3), b=c(4,5,6)) > dimnames(f) [[1]]</pre>

Function	Main arguments	Details
		<pre>[1] "1" "2" "3" [[2]] [1] "a" "b"</pre>
<code>floor()</code>	A numerical argument	Returns the smallest integers, not more than the elements of the original argument, as follows: <pre>> floor(57.76) [1] 57</pre>
<code>getwd()</code>	No argument	Retrieves the current working directory (absolute path).
<code>grep()</code>	The pattern (can be a regex) and the character vector	Searches for a pattern within a character vector as follows: <pre>> grep("rs", c("rs5427", "5427", "snp5437")) [1] 1</pre>
<code>head()</code>	The data frame or matrix and the number of elements, n (default 6),	Shows the first n elements of a data frame as follows: <pre>> data(iris) > head(iris, n=5)</pre>

Function	Main arguments	Details
	to be returned	
<code>hist()</code>	A vector for the histogram	Plots a histogram for the input vector.
<code>intersect()</code>	The two sets (vectors)	Finds the intersection of the two sets (vectors).
<code>length()</code>	The list or vector	Gives the length of the list, factors, or the vector.
<code>library()</code>	Package name	Loads an R package/library in the workspace as follows: <pre>> library(MAXX)</pre>
<code>list()</code>	Values to be put inside the list	Constructs a list together with the names of the elements as follows: <pre>> b <- list(1,3,5)</pre>
<code>load()</code>	Path to the object	Loads an object or data into the R session.

Function	Main arguments	Details
<code>match()</code>	Values to be matched and the values to be matched against	Gives the vector of position, where the match of the first argument was found in the second, as follows: <pre>> match("A", c("A", "D", "A", "S")) [1] 1</pre>
<code>matrix()</code>	Values in the matrix can be a number or a vector of the size same as the dimensions of the vector	Creates a new matrix as follows: <pre>> matrix(c(1:9), 3, 3) [,1] [,2] [,3] [1,] 1 4 7 [2,] 2 5 8 [3,] 3 6 9</pre>
<code>max()</code>	A vector or list	Finds the maximum value in a numeric list or vector as follows: <pre>> x <- sample(20, 10) > x [1] 12 5 19 6 15 16 1 11 2 4 > max(x) [1] 19</pre>
<code>mean()</code>	A vector or list	Finds the mean value in a numeric list or vector as follows:

Function	Main arguments	Details
		<pre> > x <- sample(20,10) > x [1] 12 5 19 6 15 16 1 11 2 4 > mean(x) [1] 9.1 </pre>
<code>min()</code>	A vector or list	Finds the minimum value in a numeric list or vector as follows: <pre> > x <- sample(20,10) > x [1] 12 5 19 6 15 16 1 11 2 4 > min(x) [1] 1 </pre>
<code>names()</code>	Object name	Gives the names in a named object, for example, a list or vector
<code>ncol()</code>	The matrix or data frame	Gives the number of columns in the data frame
<code>nrow()</code>	The matrix or data frame	Gives the number of rows in the data frame
<code>pdf()</code> , <code>png()</code>	The name and path of a	Opens the corresponding graphics device (PDF/PNG), and saves the plot

Function	Main arguments	Details
	PDF/PNG file	as a file in the assigned path or working directory
<code>range()</code>	A vector	Gives the range within the numerical vector as follows: <pre>> x <- sample(20,10) > range(x) [1] 2 20</pre>
<code>rbind</code>	Two or more data frames or matrices with the same number of columns and where appropriate, same column names	Appends the input along the rows as follows: <pre>> x <- sample(20,3) > y <- sample(20,3) > rbind(x,y) [,1] [,2] [,3] x 10 4 9 y 18 8 7</pre>
<code>read.csv()</code>	File's name and path to the file to be read	Reads the CSV file as a data frame object and writes it into the R session

Function	Main arguments	Details
<code>read.table</code>	File's name and the path, if necessary	Reads the tabular text file as a data frame
<code>rep()</code>	A number, character, vector, and so on, in addition to a positive number n	Creates a vector that repeats the first argument n times
<code>require()</code>	The name of a library in R	Loads an add-on R package (<code>require()</code> is used inside functions)
<code>rm</code>	R objects to be removed from the session	Removes/deletes an R object from the current R session
<code>round()</code>	A numeric vector and the number of decimal places desired	Rounds off the numerical argument to the specified number of decimal places as follows: <pre>> round(57.76, 1) [1] 57.8</pre>

Function	Main arguments	Details
<code>rownames()</code>	Data frame or matrix	Gives the row names of the input matrix or data frame. Can be used to assign row names as well: <pre>> data(iris) > rownames(iris)</pre>
<code>sample()</code>	The original set (vector) to sample from and the number of instances to be sampled	Randomly sample a definite number of elements from the given set of elements as follows: <pre>> sample(1:10,5) [1] 10 9 1 7 2</pre>
<code>save()</code>	Saves an object or session	Saves the assigned object as a file of the entire session. Here, the <code>obj</code> object is saved as the file <code>myObj.RData</code>: <pre>> save(obj, file="myObj.RData")</pre>
<code>sd()</code>	Numeric vector	Returns the standard deviation of the vector as follows: <pre>> sd(sample(1:10,5)) [1] 2.880972</pre>

Function	Main arguments	Details
<code>seq()</code>	Lower and upper limits	Generates a regular sequence between the lower and upper limits. The interval can be manipulated with the <code>by</code> argument (check <code>?seq</code> for details).
<code>setdiff()</code>	Two sets (vectors)	Returns the differences in the elements between the input sets.
<code>setwd()</code>	Path to the desired directory (as a character)	Sets the new directory path as the working directory or relative directory for the current R session as follows: <pre>> setwd("path/to/dir")</pre>
<code>sink()</code>	Desired name for a text file	Redirects all the information to be printed on the screen to the file in the argument as follows: <pre>> sink("filename.txt") > sum(1,2,3) > sink()</pre>
<code>sort</code>	A vector	Arranges the vectors in the ascending or descending order as desired by the logical argument.

Function	Main arguments	Details
<code>source()</code>	The path to a source file (*.R file)	Loads the source code and all the objects and functions in it into the R session from a local or remote file. It also reads, parses, and evaluates the expressions in the file.
<code>str()</code>	An object	Gives the structure of the object with the following commands: <pre>> data(iris) > str(iris)</pre>
<code>strsplit()</code>	Character vector and regular expression (regex) for splitting	Splits the character vector into substrings based on the regex provided: <pre>> strsplit("hsa:01234", ":") [[1]] [1] "hsa" "01234"</pre> Apply Command style to <pre>> strsplit("hsa:01234", ":") [[1]]</pre>
<code>subset()</code>	Original data frame,	Subsets the desired part of a data frame that meets specific conditions

Function	Main arguments	Details
	vector, matrix conditions, and desired columns	with the following commands: <pre>> myTest <- data.frame(x=c(1:10) , y=c(3:12)) > subset(myTest, x>4, c(x,y))</pre>
<code>sum</code>	A numeric vector	Computes the sum of all the elements of the vector as follows: <pre>> sum(c(1:10)) [1] 55</pre>
<code>summary</code>	An R object	Depending on the object structure, this operator gives a summary of the contents in the object with the following commands: <pre>> data(iris) > summary(iris)</pre>
<code>tail()</code>	The data frame or matrix and the number of n elements to	Shows the last n rows of <code>data.frame</code> with the following commands: <pre>> data(iris) > tail(iris, n=5)</pre>

Function	Main arguments	Details
	be returned (default 6)	
<code>union()</code>	Sets as vectors	Finds the union of the input sets as follows: <pre>> union(c("a", "b", "c"), c("a", "e")) [1] "a" "b" "c" "e"</pre>
<code>unique()</code>	A vector	Returns unique elements in the input vector, <code>data.frame</code> , or array.
<code>which</code>	The condition operation	Returns the parts of the original object that satisfy a desired condition. An example is given as follows: <pre>> which((1:20)%%2==0) [1] 2 4 6 8 10 12 [1] 14 16 18 20</pre>
<code>write.csv()</code>	Data frame or matrix and the filename	Writes a CSV file in the desired directory (the working directory is the default) with the values from <code>data.frame</code> or matrix.

Appendix B. Useful R Packages

The following are some of the useful R libraries and their applications. There are some packages that are interesting for bioinformatics but in the presented recipes, they were not needed and hence the corresponding chapter numbers are not a part of the following table:

Library name	Application	Repository	Chapter
<code>ade4</code>	A package for multivariate data analysis.	CRAN	
<code>affy</code>	This bears the functions for the exploratory analysis of Affymetrix Oligonucleotide Array.	Bioconductor	5
<code>ALL</code>	This is a data package with data for T and B cell acute lymphocytic leukemia from the Ritz Laboratory at the DFCI.	Bioconductor	5
<code>annotate</code>	This package provides interface functions to support user actions that rely on the different metadata packages	Bioconductor	2, 5

Library name	Application	Repository	Chapter
	provided through the Bioconductor project.		
AnnotationDbi	This package provides interface and database connection functions for annotation data packages.	Bioconductor	1, 2, 5, 8
ape	Analyses of Phylogenetics and Evolution (ape) contains functions to play around with phylogenetic trees and related data in R.	CRAN	3
ArrayQualityMetrics	This package generates a quality report for the microarray data.	Bioconductor	5
bio3d	This package contains utilities for the analysis of protein structure, sequence, and so on.	Grant Lab	4
BioMark	This is a package for biomarker selection via classification.	CRAN	9

Library name	Application	Repository	Chapter
<code>biomaRt</code>	This package provides an interface to a collection of databases that implement the BioMart software suite (http://www.biomart.org). It enables the retrieval of data within the R workspace, without the need to know the underlying database schemas and to write complex SQL queries.	Bioconductor	1, 2, 5
<code>BioNet</code>	This package can be used for network analysis.	Bioconductor	
<code>Biostrings</code>	This package has string-matching algorithms and related utilities mainly used in sequence analysis.	Bioconductor	2, 3
<code>CHNOSZ</code>	This package has functions for analysis in chemical thermodynamic modeling in biochemistry.	CRAN	3, 4

Library name	Application	Repository	Chapter
<code>chipseq</code>	An R package that analyzes the <code>chipseq</code> data.	Bioconductor	8
<code>clusterProfiler</code>	A package to analyze the GO- and KEGG-based functional profiles of genes and gene clusters.	Bioconductor	
<code>codelink</code>	An R package for the analysis of GE healthcare gene expression data.	Bioconductor	
<code>RColorBrewer</code>	A package to draw color palettes to draw maps that are shaded according to a variable.	CRAN	
<code>dplyr</code>	Package for data manipulation.	CRAN	
<code>e1071</code>	A package with many functions related to machine learning such	CRAN	9

Library name	Application	Repository	Chapter
	as SVM Naïve Bayes and so on.		
<code>edgeR</code>	An R package for the differential expression analysis of RNA-seq data.	Bioconductor	8
<code>gdata</code>	A data manipulation package.	CRAN	
<code>GenomicFeatures</code>	A package to retrieve and manage transcript-related features from UCSC genome Bioinformatics.	Bioconductor	8
<code>GenABEL</code>	A package for genome-wide association analysis with SNPs.	CRAN	6
<code>ggplot2</code>	This is a data visualization and plotting package that provides grammar for graphs.	CRAN	5, 6

Library name	Application	Repository	Chapter
<code>GO.db</code>	Annotation maps for Gene Ontology (GO) .	Bioconductor	5, 8
<code>gosim</code>	A package that contains functions for the computation of functional similarities between GO terms and a gene product.	CRAN	
<code>GOstats</code>	This package contains functions to interact with GO and microarray data.	Bioconductor	5
<code>graph</code>	A package with graph-handling functions.	Bioconductor	5
<code>GSEABase</code>	This package provides classes and methods that support Gene Set Enrichment Analysis (GSEA) .	Bioconductor	
<code>GWASTools</code>	An R package for genome-wide association studies	Bioconductor	6

Library name	Application	Repository	Chapter
	(GWAS) and data handling.		
Hmisc	This package contains many functions that are useful in data analysis, high-level graphics, utility operations, character string manipulation, conversion of R objects to LaTeX code, and so on.	CRAN	5
igraph	A package for simple graphs and networks as well as for graph analysis and plotting.	CRAN	5
KEGG.db	Annotation maps for KEGG.	Bioconductor	
KEGGgraph	An R package that provides an interface between the KEGG pathway and R and the required analysis functions.	Bioconductor	

Library name	Application	Repository	Chapter
KEGGREST	This provides an interface to the KEGG REST server.	Bioconductor	
limma	A package for linear models and differential expressions for microarray data.	Bioconductor	5, 8
lumi	This package has functions that are required for Illumina microarray data analysis.	Bioconductor	
MALDIquant	A package for MS studies providing utilities to analyze MALDI-TOF and other mass spectrometry data.	CRAN	7
MALDIquantForeign	An auxiliary package for MALDIquant to read data types other than MALDI.	CRAN	7
MASS	Some key statistical functions and data based on <i>Modern</i>	CRAN	9

Library name	Application	Repository	Chapter
	<i>Applied Statistics with S, Venable and Ripley.</i>		
<code>methyAnalysis</code>	An R package to analyze and visualize the DNA methylation data.	Bioconductor	8
<code>mlbench</code>	This package contains a number of interesting datasets for machine learning problems.	CRAN	
<code>mlogit</code>	A package for multinomial logit models.	CRAN	
<code>multicore</code>	A package for the parallel processing of R code on machines with multiple processor cores (for Linux-based platforms).	CRAN	
<code>muscle</code>	A package for multiple sequence alignment.	CRAN	3, 4

Library name	Application	Repository	Chapter
<code>mvtnorm</code>	This package has functions for multivariate normal distributions and related operations.	CRAN	9
<code>NCBI2R</code>	An R package to navigate and annotate genes and SNPs.	CRAN	6
<code>plotrix</code>	This package contains many plotting functions.	CRAN	
<code>plyr</code>	A package for data manipulation operations such as splitting, combining, and so on.	CRAN	
<code>RPostgreSQL</code>	This provides an interface to the <code>PostgreSQL</code> database system.	CRAN	
<code>reactome.db</code>	This provides annotation maps from <code>reactome</code> .	Bioconductor	

Library name	Application	Repository	Chapter
<code>pROC</code>	The <code>pROC</code> package has functions to compute receiver operating characteristics (ROC) and functions to analyze it.	CRAN	
<code>protr</code>	This package contains subroutines for protein sequence descriptor calculation and similarity computation.	CRAN	4
<code>protViz</code>	An R package for the analysis and visualization of MS data.	CRAN	7
<code>randomForest</code>	A package for random forest-based classification.	CRAN	
<code>RBGL</code>	This package implements several graph algorithms from the <code>Boost</code> library.	CRAN	5
<code>Rcurl</code>	These are functions to make HTTP requests	CRAN	

Library name	Application	Repository	Chapter
	and are used to fetch results from a web server.		
<code>reshape</code>	This package contains functions to restructure and aggregate data.	CRAN	
<code>reshape2</code>	This package restructures and reshapes the data.	CRAN	
<code>RFLPtools</code>	This R package has functions to analyze the DNA fragment samples (the <code>RFLP</code> data).	CRAN	3
<code>Rgraphviz</code>	A package to plot graphs (vertices and edges) in R.	Bioconductor	
<code>RISmed</code>	An R package to interact with NCBI data and download data.	CRAN	
<code>rjava</code>	This provides R to a Java interface.	CRAN	

Library name	Application	Repository	Chapter
Rknots	A package for the topological analysis of polymers, including proteins.	CRAN	4
RMySQL	This provides a database interface and MySQL driver for R.	CRAN	
ROCR	Another package to analyze ROCs.	CRAN	9
ROSE	Random over-sampling examples (ROSE) provides functions and utilities to deal with binary classification.	CRAN	
Rsamtools	The package provides an interface to the samtools , bcftools , and tabix utilities to handle SAM and BAM files.	Bioconductor	8
seqinr	A package for the analysis, visualization, and management of	CRAN	3, 8

Library name	Application	Repository	Chapter
	biological sequence (DNA and protein) data.		
<code>ShortRead</code>	This package contains functions and methods for handling high-throughput and short-read sequencing data.	Bioconductor	8
<code>SNPassoc</code>	A package for association studies of SNPs.	CRAN	6
<code>spliceR</code>	The package can be used for the classification of alternative splicing and prediction of coding potential from RNA-seq data.	Bioconductor	
<code>SRadb</code>	<code>SRadb</code> provides access to the metadata at the SRA database.	Bioconductor	8
<code>stringr</code>	This package contains functions to work with strings.	CRAN	

Library name	Application	Repository	Chapter
<code>topGO</code>	This package has functions to test the <code>GO</code> terms.	Bioconductor	5, 8
<code>vegan</code>	This package contains interesting functions mainly from the domain of ecology but can be used for purposes such as similarity score computation, diversity analysis, and so on, useful for metagenomic studies.	CRAN	3, 9
<code>xlsx</code>	The R functions to read/write/format Excel 2007 and Excel 97/2000/XP/2003 file formats.	CRAN	
<code>XML</code>	A package for parsing and generating XML files.	CRAN	

The following are the links for the R library repositories:

- **CRAN:** This R library repository is available at <http://cran.r-project.org/>

Use the following command for installation:

```
> install.packages("<package name>")
```

- **Bioconductor:** This R library repository is available at <http://www.bioconductor.org>

Use the following command for installation:

```
> source("http://bioconductor.org/biocLite.R")  
> biocLite("<package name>")
```

- **Grant Lab:** This R library repository is available at <http://thegrantlab.org/bio3d/index.php>

Use the following command for installation:

```
> install.package("<path/to/file.tar.gz>",  
  repos=NULL, type="source")
```

Index

A

- Acute Lymphoblastic Leukemia (ALL)
 - about / [How to do it...](#)
- AffyBatch object
 - about / [Building the ExpressionSet object](#)
 - handling / [Handling the AffyBatch object](#), [How it works...](#)
- Akaike Information Criterion (AIC)
 - about / [How it works...](#)
- alignment data
 - reading / [Reading alignment data](#), [How to do it...](#), [How it works...](#)
- Amazon Machine Image (AMI) / [How to do it...](#)
- Amino Acid Composition (AAC)
 - about / [How it works...](#)
- annotation, SNP
 - about / [The SNP annotation and enrichment](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- annotation database
 - working with / [Handling annotation databases in R](#), [How to do it...](#), [There's more...](#)
 - prerequisites / [Getting ready](#)

- reference link / [There's more...](#)
- artificial expression data
 - generating / [Generating artificial expression data](#), [Getting ready](#), [How it works...](#), [There's more...](#)
- association function
 - about / [How it works...](#)
- association scans
 - executing, for SNP / [Running association scans for SNPs](#), [How it works...](#), [There's more...](#)
- AWS
 - about / [Getting ready](#)
 - URL / [How to do it...](#)

B

- BAM file
 - about / [Reading alignment data](#)
- batch effects
 - about / [Overcoming batch effects in expression data](#)
 - overcoming, in expression data / [Overcoming batch effects in expression data](#), [How to do it...](#), [How it works...](#)
- betr package
 - about / [There's more...](#)
- Bioconductor

- about / [Getting started and installing libraries](#), [Introduction](#)
 - URL / [Getting started and installing libraries](#), [Introduction](#)
- Bioconductor libraries
 - installing / [Installing packages from Bioconductor](#), [How it works...](#), [There's more...](#)
 - prerequisites / [Getting ready](#)
- Biological Process (BP) / [How it works...](#)
- biomarker
 - identifying, machine learning used / [Biomarker identification using array data](#), [How to do it...](#)
- BioMark library
 - about / [How it works...](#)
- BioMart
 - data, retrieving from / [Retrieving data from BioMart](#), [How to do it...](#), [There's more...](#)
 - URL / [Retrieving data from BioMart](#)
- BLAST
 - about / [Handling BLAST results](#)
 - URL / [Handling BLAST results](#)
- BLAST results
 - about / [Handling BLAST results](#)
 - handling / [Handling BLAST results](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
- Blocks Substitution Matrix (BLOSUM) / [How it works...](#)
- Bonferroni correction
 - about / [How it works...](#)

- bootstrapping
 - about / [Bootstrapping in machine learning](#), [How it works...](#)
- Bowtie
 - about / [Reading alignment data](#)
- boxplot
 - about / [How it works...](#)
- Bruker format
 - reading, of MS data / [Reading the MS data of the Bruker format](#), [How it works...](#)
- BWA
 - about / [Reading alignment data](#)

C

- carcinoma in situ (CIS)
 - about / [How it works...](#)
- caret package
 - installing / [How to do it...](#)
 - about / [How it works...](#)
- CEL file
 - about / [Reading CEL files](#)
 - reading / [Reading CEL files](#), [How it works...](#)
- Cellular Components (CC) / [How it works...](#)

- centroids
 - about / [How it works...](#)
- ChipSeq
 - about / [Analyzing ChipSeq data](#)
- ChipSeq data
 - analysing / [Analyzing ChipSeq data](#), [How to do it...](#), [How it works...](#)
- classification
 - about / [Introduction](#), [Supervised learning for classification](#)
 - linear discriminant analysis (LDA) / [Supervised learning for classification](#)
 - decision tree (dt) / [Supervised learning for classification](#)
 - support vector machine (SVM) / [Supervised learning for classification](#)
 - performing / [Getting ready](#), [How to do it...](#), [How it works...](#)
 - probabilistic classification / [Probabilistic learning in R with Naïve Bayes](#)
- class labels
 - about / [Supervised learning for classification](#)
- cloud-based Bioconductor
 - usage / [Bioconductor in the cloud](#)
 - prerequisites / [Getting ready](#)
 - setting up / [How to do it...](#), [How it works...](#)
- cloud computing
 - about / [Bioconductor in the cloud](#)

- clustering
 - performing, on microarray data / [Clustering microarray data](#), [How it works...](#), [There's more...](#)
 - about / [Introduction](#)
 - k-means clustering, performing / [Data clustering in R using k-means and hierarchical clustering](#), [How to do it...](#), [There's more...](#)
 - hierarchical clustering, performing / [Data clustering in R using k-means and hierarchical clustering](#), [How to do it...](#), [There's more...](#)
 - density-based clustering / [There's more...](#)
 - Fuzzy clustering / [There's more...](#)
- clusters
 - about / [Data clustering in R using k-means and hierarchical clustering](#)
 - visualizing / [Visualizing clusters](#), [How to do it...](#), [How it works...](#), [See also](#)
- cmdplot function
 - about / [How it works...](#)
- CNV
 - about / [Association tests with CNV data](#)
- CNV association analysis
 - about / [Association tests with CNV data](#)
 - performing / [How to do it...](#), [How it works...](#), [There's more...](#)
- co-expression networks
 - generating, from microarray data / [Getting a co-expression network from microarray data](#), [How to do it...](#), [How it works...](#), [There's more...](#)

- ComBat function
 - about / [How it works...](#)
- CRAN
 - about / [Getting started and installing libraries](#)
- cross-validation (CV)
 - about / [Cross-validation for classifiers](#)
 - performing, for classifiers / [Cross-validation for classifiers](#), [How to do it...](#), [How it works...](#)
 - k-fold cross-validation / [Cross-validation for classifiers](#)
 - n-fold cross-validation / [Cross-validation for classifiers](#)

D

- data
 - writing / [Reading and writing data](#), [How it works...](#), [There's more...](#)
 - reading / [Reading and writing data](#), [How it works...](#), [There's more...](#)
 - filtering / [Filtering and subsetting data](#), [How to do it...](#), [How it works...](#)
 - subsetting / [Filtering and subsetting data](#), [How to do it...](#), [How it works...](#)
 - statistical operations, performing / [Basic statistical operations on data](#), [How to do it...](#), [How it works...](#)
 - statistical tests, performing / [Performing statistical tests on data](#), [How to do it...](#), [How it works...](#)
 - visualizing / [Visualizing data](#), [How to do it...](#), [How it works...](#), [There's more...](#)
 - retrieving, from BioMart / [Retrieving data from BioMart](#), [How to do it...](#), [There's more...](#)

- downloading, from SRA database / [Downloading data from the SRA database](#), [How it works...](#)
- data elements
 - extracting, from MS data / [Extracting data elements from the MS data object](#), [How to do it...](#), [There's more...](#)
- dbConnect function
 - about / [How it works...](#)
- decision tree (dt)
 - about / [Supervised learning for classification](#)
- differential gene expression
 - searching, in microarray data / [Finding the differentially expressed genes](#), [How to do it...](#), [How it works...](#)
- differentially expressed (DE) genes
 - about / [How it works...](#)
- differentially methylated regions (DMRs)
 - about / [Analyzing methylation data](#)
- directed acyclic graph (DAG)
 - about / [How to do it...](#)
- distance weighted discrimination (DWD)
 - about / [See also](#)
- dist function
 - about / [How it works...](#)

- DNA methylation data
 - analysing / [Analyzing methylation data](#), [How to do it...](#), [How it works...](#)

E

- e1071 package
 - about / [There's more...](#), [How it works...](#)
 - URL / [There's more...](#)
- Eclipse IDE
 - URL / [How to do it...](#)
- edgeR library
 - used, for analyzing RNAseq data / [Getting ready](#), [How to do it...](#), [How it works...](#)
- Efetch
 - about / [How it works...](#)
- ExPasy
 - URL / [How to do it...](#)
- ExpressionSet
 - building / [Building the ExpressionSet object](#), [How to do it...](#), [How it works...](#)

F

- false positive rate (fpr)

- about / [How it works...](#)
- FASTA file
 - reading / [Reading and writing the FASTA file](#), [How to do it...](#), [How it works...](#)
 - writing / [Reading and writing the FASTA file](#), [How to do it...](#), [How it works...](#)
 - about / [Reading and writing the FASTA file](#)
- FASTQ file
 - reading / [Reading FASTQ files in R](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
- features, protein sequence
 - computing / [Computing the features of a protein sequence](#), [How to do it...](#), [See also](#)
- Fisher test
 - about / [The GO enrichment of genes](#)
- fold change
 - about / [Fold changes in microarray data](#)
 - in microarray data / [Fold changes in microarray data](#), [How it works...](#)
- functional enrichment, microarray data
 - describing / [The functional enrichment of data](#), [How to do it...](#), [There's more...](#)

G

- gap package

- URL / [There's more...](#)
 - about / [There's more...](#)
- GC content
 - about / [Getting the detail of a sequence composition](#)
- gen.data function
 - about / [How it works...](#)
- GenBank
 - reference links / [See also](#)
- gene expression data
 - visualization / [More visualizations for gene expression data](#), [How to do it...](#), [How it works...](#)
- get.biom function
 - about / [How it works...](#)
- getBM function
 - attribute argument / [How it works...](#)
 - filter argument / [How it works...](#)
 - mart argument / [How it works...](#)
- getSRA function
 - about / [How it works...](#)
- ggplot2 package
 - URL / [There's more...](#)
 - about / [There's more...](#)
- GO

- about / [There's more...](#), [The GO annotation of genes](#)
 - URL / [See also](#)
- GO annotation
 - about / [The GO annotation of genes](#)
 - prerequisites / [Getting ready](#)
 - performing / [How to do it...](#), [How it works...](#)
- GO enrichment
 - about / [The GO enrichment of genes](#)
 - prerequisites / [Getting ready](#)
 - performing / [How to do it...](#), [There's more...](#), [See also](#)
- GOHyperGParams object
 - about / [How it works...](#)
- GO project
 - URL / [See also](#)
- goseq function
 - about / [How it works...](#)
- goseq library
 - using / [Getting ready](#)
- graphical user interface (GUI) / [How to do it...](#)
- GSE24460
 - URL, for downloading / [How to do it...](#)
- GSEA
 - about / [How it works...](#)

- GWAS
 - about / [Introduction](#)
- GWAS data
 - about / [Introduction](#)
 - PLINK GWAS data / [Importing PLINK GWAS data](#)
 - handling, GWASTools package used / [Data handling with the GWASTools package](#), [How to do it...](#), [How it works...](#)
- GWAS data formats
 - manipulating / [Getting ready](#), [How to do it...](#), [How it works...](#)
- GWAS results
 - visualizing / [How to do it...](#), [How it works...](#), [There's more...](#)
- GWASTools package
 - used, for handling GWAS data / [Data handling with the GWASTools package](#), [How to do it...](#), [How it works...](#)
 - reference links / [See also](#)

H

- HapMap data
 - about / [There's more...](#)
 - URL / [There's more...](#)
- hierarchical clustering
 - performing / [Data clustering in R using k-means and hierarchical clustering](#), [How to do it...](#), [There's more...](#)
- high performance computing (HPC)

- about / [Bioconductor in the cloud](#)
 - HWE
 - about / [Testing data for the Hardy-Weinberg equilibrium](#)
 - SNP data, testing for / [Testing data for the Hardy-Weinberg equilibrium](#), [How to do it...](#), [How it works...](#)
 - HWExact function / [How it works...](#)
 - hyperGTest function
 - about / [How it works...](#)
- 
- ID conversion
 - performing / [Performing ID conversions](#), [How to do it...](#), [There's more...](#)
 - prerequisites / [Getting ready](#)
 - installation, Bioconductor libraries / [Installing packages from Bioconductor](#), [How it works...](#), [There's more...](#)
 - installation, R libraries / [Getting started and installing libraries](#), [How to do it...](#), [How it works...](#), [There's more...](#)
 - integrated development environments (IDEs) / [How to do it...](#)
 - intensity plot
 - about / [How it works...](#)
 - InterPro
 - about / [Working with the InterPro domain annotation](#)
 - InterPro domain annotation
 - working with / [Working with the InterPro domain annotation](#), [Getting ready](#), [There's more...](#)

- Isobaric Tags for Relative and Absolute Quantitation (iTRAQ)
 - about / [Getting ready](#)

K

- k-fold cross-validation
 - about / [Cross-validation for classifiers](#)
- k-means clustering
 - performing / [Data clustering in R using k-means and hierarchical clustering](#), [How to do it...](#), [How it works...](#)
- KEGG
 - about / [The KEGG annotation of genes](#), [There's more...](#)
 - URL / [There's more...](#)
- KEGG annotation
 - about / [The KEGG annotation of genes](#)
 - prerequisites / [Getting ready](#)
 - performing / [How to do it...](#), [There's more...](#)
- KEGG API
 - URL / [There's more...](#)
- KEGG enrichment
 - about / [The KEGG enrichment of genes](#)
 - prerequisites / [Getting ready](#)
 - performing / [How to do it...](#), [How it works...](#)
 - performing, of sequence data / [The KEGG enrichment of sequence data](#), [How to do it...](#), [How it works...](#)

- Kolmogorov-Smirnov test (KS test) / [How to do it...](#)
- Kolmogorov Smirnov (KS) statistic
 - about / [How it works...](#)

L

- libraries, R
 - installing / [Getting started and installing libraries](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- limma library
 - about / [How it works...](#)
 - used, for analyzing NGS data / [The differential analysis of NGS data using limma](#), [How to do it...](#), [How it works...](#)
- linear discriminant analysis (LDA)
 - about / [Supervised learning for classification](#)
- linkage disequilibria (LD)
 - about / [How it works...](#)
- loess normalization
 - about / [How it works...](#)

M

- .map file
 - about / [Getting ready](#)
 - URL, for downloading / [Getting ready](#)
- machine learning

- about / [Introduction](#)
- supervised learning / [Introduction](#)
- unsupervised learning / [Introduction](#)
- bootstrapping / [Bootstrapping in machine learning](#), [How it works...](#)
- performance, measuring / [Getting ready](#), [How it works...](#)
- used, for biomarker identification / [Biomarker identification using array data](#), [How to do it...](#)
- machine learning, key issues
 - data quality / [Introduction](#)
 - data normalization / [Introduction](#)
 - feature selection / [Introduction](#)
- madsim function
 - about / [How it works...](#)
- MALDIquant
 - MS data, converting to / [Converting the MS data in the mzXML format to MALDIquant](#), [How to do it...](#)
 - about / [Converting the MS data in the mzXML format to MALDIquant](#)
- Manhattan plot
 - about / [How it works...](#)
- MA plot
 - about / [How it works...](#)
- mass spectrum
 - about / [Preprocessing MS data](#)
- median absolute deviation (MAD)

- about / [How it works...](#)
- Messenger RNAs (mRNAs) / [Introduction](#)
- microarray data
 - analyzing / [Introduction](#)
 - quality, checking / [Checking the quality of data](#), [Getting ready](#), [How it works...](#)
 - normalization / [Data normalization](#), [How to do it...](#), [How it works...](#), [There's more...](#)
 - analyzing, with PCA / [An exploratory analysis of data with PCA](#), [How it works...](#)
 - differential gene expression, searching in / [Finding the differentially expressed genes](#), [How to do it...](#), [How it works...](#)
 - fold change / [Fold changes in microarray data](#), [How to do it...](#), [How it works...](#)
 - functional enrichment, describing / [The functional enrichment of data](#), [How to do it...](#), [There's more...](#)
 - clustering / [Clustering microarray data](#), [How it works...](#), [There's more...](#)
 - co-expression networks, generating from / [Getting a co-expression network from microarray data](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- microarrays
 - about / [Introduction](#)
- missing call rate (MCR)
 - about / [How to do it...](#)
- Molecular Function (MF) / [How it works...](#)
- MS
 - about / [Introduction](#)
- MSA

- about / [Multiple sequence alignment](#), [How to do it...](#), [How it works...](#)
- MS data
 - about / [Introduction](#)
 - mzXML format, reading / [Reading the MS data of the mzXML/mzML format](#), [How to do it...](#), [How it works...](#)
 - mzML format, reading / [Reading the MS data of the mzXML/mzML format](#), [How to do it...](#), [How it works...](#)
 - Bruker format, reading / [Reading the MS data of the Bruker format](#), [How it works...](#)
 - converting, to MALDIquant / [Converting the MS data in the mzXML format to MALDIquant](#), [How to do it...](#)
 - data elements, extracting from / [Extracting data elements from the MS data object](#), [How to do it...](#), [There's more...](#)
 - preprocessing / [Preprocessing MS data](#), [How to do it...](#), [How it works...](#)
 - peak detection, performing in / [Peak detection in MS data](#), [How to do it...](#)
 - peak alignment, performing in / [Peak alignment with MS data](#), [How it works...](#)
 - peptides, identifying in / [Peptide identification in MS data](#), [How to do it...](#), [How it works...](#), [There's more...](#)
 - protein quantification analysis, performing in / [Getting ready](#), [How it works...](#)
 - multiple group analysis, performing in / [Performing multiple groups' analysis in MS data](#), [How to do it...](#), [How it works...](#)
 - visualization, creating / [How to do it...](#), [How it works...](#)
- MS data formats
 - mzXML / [Introduction](#)
 - mzML / [Introduction](#)
- multicore package
 - URL / [There's more...](#)

- multiple group analysis
 - performing, in MS data / [Performing multiple groups' analysis in MS data](#), [How to do it...](#), [How it works...](#)
- multiple microarray data
 - working with / [Working with the data of multiple classes](#), [How to do it...](#)
- multivariate normal distribution (MVN) / [Generating probability distributions](#)
- MUSCLE algorithm
 - about / [How it works...](#)
- mzML format
 - about / [Introduction](#)
 - reading, of MS data / [Reading the MS data of the mzXML/mzML format](#), [How to do it...](#), [How it works...](#)
- mzXML format
 - about / [Introduction](#)
 - reading, of MS data / [Reading the MS data of the mzXML/mzML format](#), [How to do it...](#), [How it works...](#)

N

- n-fold cross-validation
 - performing / [How to do it...](#), [How it works...](#)
- NAD kinase (NADK)
 - about / [Getting ready](#)

- Nature Scitable
 - URL / [See also](#)
- Naïve Bayes classification
 - about / [Probabilistic learning in R with Naïve Bayes](#)
 - performing / [How to do it...](#)
- NCBI
 - URL / [Working with PubMed in R](#)
- NetCDF file
 - about / [How it works...](#)
- network common data form (NetCDF)
 - about / [See also](#)
- NGS
 - about / [Introduction](#)
 - Illumina (Solexa) sequencing / [Introduction](#)
 - Roche 454 sequencing / [Introduction](#)
 - Ion torrent (proton and PGM sequencing) / [Introduction](#)
 - SOLiD sequencing / [Introduction](#)
- NGS data
 - raw NGS data, preprocessing / [Preprocessing the raw NGS data](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
 - analyzing, limma library used / [The differential analysis of NGS data using limma](#), [How to do it...](#), [How it works...](#)
 - visualization / [Visualizations for NGS data](#), [How to do it...](#), [How it works...](#)
- nlcv package

- about / [There's more...](#)
- non-differentially expressed (non DE) genes
 - about / [How it works...](#)
- normalization, microarray data
 - about / [Data normalization](#), [How to do it...](#), [How it works...](#), [There's more...](#)
 - Variance Stabilization and Normalization (VSN) / [How it works...](#)
 - loess normalization / [How it works...](#)
 - quantile normalization / [How it works...](#)

O

- Online Mendelian Inheritance in Man (OMIM)
 - URL / [Introduction](#)
- open reading frames (ORFs)
 - about / [How to do it...](#)
- overfitting
 - about / [Supervised learning for classification](#)

P

- .ped file
 - about / [Getting ready](#)
 - URL, for downloading / [Getting ready](#)
- pairwise sequence alignment

- about / [Pairwise sequence alignment](#)
 - performing / [How to do it...](#), [How it works...](#), [See also](#)
- partial least square (PLS)
 - about / [How it works...](#)
- PCA
 - about / [An exploratory analysis of data with PCA](#)
 - used, for analyzing microarray data / [An exploratory analysis of data with PCA](#), [How it works...](#)
- PDB file
 - handling / [Handling the PDB file](#), [How to do it...](#)
- peak alignment
 - performing, in MS data / [Peak alignment with MS data](#), [How it works...](#)
- peak detection
 - performing, in MS data / [Peak detection in MS data](#), [How to do it...](#)
- peaks
 - about / [Peak detection in MS data](#)
- peptides
 - identifying, in MS data / [Peptide identification in MS data](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- Phred score
 - about / [Introduction](#)

- phylogenetic analysis
 - about / [Phylogenetic analysis and tree plotting](#)
 - performing, on sequence / [How to do it...](#), [How it works...](#)
- phylogenetic tree
 - plotting / [Phylogenetic analysis and tree plotting](#), [How to do it...](#), [How it works...](#)
- PLINK GWAS data
 - about / [Importing PLINK GWAS data](#), [There's more...](#)
 - importing / [Importing PLINK GWAS data](#), [How it works...](#), [There's more...](#)
 - URL / [See also](#)
- plot.roc function
 - about / [How it works...](#)
- Principal Component Analysis (PCA)
 - about / [How it works...](#)
- probabilistic classification
 - about / [Probabilistic learning in R with Naïve Bayes](#)
 - Naïve Bayes classification / [Probabilistic learning in R with Naïve Bayes](#)
- probability distributions
 - generating / [Generating probability distributions](#), [How it works...](#)
- Protein Data Bank (PDB) / [Reading and writing the FASTA file](#)
 - about / [See also](#)

- URL, for documentation / [See also](#)
- protein quantification analysis
 - performing, in MS data / [Getting ready](#), [How it works...](#)
- protein secondary structure
 - visualizing / [Working with the secondary structure features of proteins](#), [How to do it...](#)
- protein sequence
 - retrieving, from UniProt / [Retrieving a sequence from UniProt](#), [How to do it...](#), [How it works...](#), [See also](#)
 - features, computing / [Computing the features of a protein sequence](#), [How to do it...](#), [See also](#)
- protein sequence analysis
 - performing / [Protein sequence analysis](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- protein structure
 - visualizing / [Visualizing the protein structures](#), [Getting ready](#), [How it works...](#)
- Protr Vignette
 - URL / [See also](#)
- PubMed
 - working with / [Working with PubMed in R](#), [Getting ready](#), [How it works...](#)
 - requirements / [Getting ready](#)



- Q-Q plots
 - about / [How it works...](#)
- quality, microarray data
 - checking / [Checking the quality of data](#), [Getting ready](#), [How it works...](#)
- quantile normalization
 - about / [How it works...](#)

R

- R
 - about / [Introduction](#)
 - libraries, installing / [Getting started and installing libraries](#), [How to do it...](#), [How it works...](#), [There's more...](#)
 - data, reading / [Reading and writing data](#), [How it works...](#), [There's more...](#)
 - data, writing / [Reading and writing data](#), [How it works...](#), [There's more...](#)
 - data, subsetting / [Filtering and subsetting data](#), [How to do it...](#), [How it works...](#)
 - data, filtering / [Filtering and subsetting data](#), [How to do it...](#), [How it works...](#)
 - statistical operations, performing on data / [Basic statistical operations on data](#), [How to do it...](#), [How it works...](#)
 - probability distributions, generating / [Generating probability distributions](#), [How it works...](#)
 - statistical tests, performing on data / [Performing statistical tests on data](#), [How to do it...](#), [How it works...](#)
 - data, visualizing / [Visualizing data](#), [How to do it...](#), [How it works...](#), [There's more...](#)

- PubMed / [Working with PubMed in R](#), [Getting ready](#), [How it works...](#)
- data, retrieving from BioMart / [Retrieving data from BioMart](#), [How to do it...](#), [There's more...](#)
- annotation database, working with / [Handling annotation databases in R](#), [How to do it...](#), [There's more...](#)
- Ramachandran plot
 - about / [Understanding the Ramachandran plot](#)
 - visualizing / [Getting ready](#), [How to do it...](#), [How it works...](#)
- randomForest package
 - about / [There's more...](#)
- raw NGS data
 - preprocessing / [Preprocessing the raw NGS data](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- ReadAffy function
 - about / [How it works...](#)
- regional association plot
 - about / [How it works...](#)
 - data.frames / [How it works...](#)
- RNA degradation plot
 - about / [How it works...](#)
- RNAseq data
 - analyzing, edgeR library used / [Getting ready](#), [How to do it...](#), [How it works...](#)
 - enriching, with GO terms / [Enriching RNAseq data with GO terms](#), [How to do it...](#), [How it works...](#)

- ROC curve
 - about / [Visualizing an ROC curve in R](#)
 - visualizing / [Visualizing an ROC curve in R](#), [How to do it...](#), [How it works...](#)
- ROCR package
 - about / [How it works...](#)
- Rsamtools library
 - using / [Getting ready](#)
- RStudio
 - URL / [How to do it...](#)
- RWeka library
 - URL / [There's more...](#)

S

- scanBam function
 - about / [How it works...](#)
- secondary structure, protein
 - visualizing / [Working with the secondary structure features of proteins](#), [How to do it...](#)
- sequence
 - retrieving / [Retrieving a sequence](#), [How to do it...](#), [There's more...](#)

- phylogenetic analysis, performing on / [How to do it...](#), [How it works...](#)
- sequence alignment
 - about / [Pairwise sequence alignment](#)
 - pairwise sequence alignment / [Pairwise sequence alignment](#), [How to do it...](#), [How it works...](#), [See also](#)
 - MSA / [Multiple sequence alignment](#), [How to do it...](#), [How it works...](#)
- sequence alignment map (SAM) file
 - about / [Reading alignment data](#)
- sequence analysis
 - about / [Introduction](#)
- sequence composition
 - determining / [Getting the detail of a sequence composition](#), [How to do it...](#), [How it works...](#)
- sequence pattern
 - searching / [Pattern finding in a sequence](#), [How to do it...](#), [How it works...](#), [See](#)
- ShortRead library
 - about / [Getting ready](#)
 - installing / [How to do it...](#)
- similar proteins
 - searching / [Searching for similar proteins](#), [How it works...](#)
- SNP

- about / [Introduction](#)
- association scans, executing / [Running association scans for SNPs](#), [How it works...](#), [There's more...](#)
- annotating / [The SNP annotation and enrichment](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- SNP association analysis
 - about / [The SNP association analysis](#)
 - performing / [How to do it...](#), [How it works...](#)
- SNP data
 - testing, for HWE / [Testing data for the Hardy-Weinberg equilibrium](#), [How to do it...](#), [How it works...](#)
- SNPedia
 - URL / [Introduction](#)
- SRA
 - about / [Querying the SRA database](#)
 - URL / [Querying the SRA database](#)
- SRA database
 - querying / [Querying the SRA database](#), [How to do it...](#)
 - data, downloading from / [Downloading data from the SRA database](#), [How it works...](#)
- SRAdb library
 - about / [How it works...](#)
 - URL / [How it works...](#)
- StatET package
 - URL / [How to do it...](#)

- statistical operations
 - performing, on data / [Basic statistical operations on data](#), [How to do it...](#), [How it works...](#)
- statistical tests
 - performing, on data / [Performing statistical tests on data](#), [How to do it...](#), [How it works...](#)
- superficial transitional cell carcinoma (STCC)
 - about / [How it works...](#)
- supervised learning
 - about / [Introduction](#)
 - classification / [Introduction](#), [Supervised learning for classification](#), [How to do it...](#), [How it works...](#)
- support vector machine (SVM)
 - about / [See also](#), [Supervised learning for classification](#)

T

- tandem mass spectrometry (MS-MS)
 - about / [Peptide identification in MS data](#)
- time series expression data
 - handling / [Handling time series data](#), [How to do it...](#), [There's more...](#)
- Total Ion Current (TIC)
 - about / [How it works...](#)

- true positive (TP)
 - about / [How to do it...](#)
- true positive rate (tpr)
 - about / [How it works...](#)

U

- UniProt
 - protein sequence, retrieving from / [Retrieving a sequence from UniProt](#), [How to do it...](#), [How it works...](#), [See also](#)
 - about / [Retrieving a sequence from UniProt](#)
 - URL / [How it works...](#), [How it works...](#)
- UniProt Knowledgebase (UniProtKB)
 - about / [How it works...](#)
- unsupervised learning
 - about / [Introduction](#)
 - clustering / [Introduction](#)

V

- Variance Stabilization and Normalization (VSN)
 - about / [How it works...](#)
- vegan library
 - about / [How it works...](#)
- visualization, clusters

- performing / [Visualizing clusters](#), [How to do it...](#), [How it works...](#)
- visualization, gene expression data
 - about / [More visualizations for gene expression data](#), [How to do it...](#), [How it works...](#)
- visualization, GWAS results
 - creating / [How to do it...](#), [How it works...](#), [There's more...](#)
- visualization, MS data
 - creating / [How to do it...](#), [How it works...](#)
- visualization, NGS data
 - about / [Visualizations for NGS data](#), [How to do it...](#), [How it works...](#)
- visualization, ROC curve
 - about / [How to do it...](#), [There's more...](#)

W

- WEKA
 - about / [There's more...](#)
- whole genome SNP association analysis
 - about / [The whole genome SNP association analysis](#)
 - performing / [The whole genome SNP association analysis](#), [How it works...](#)
- Wolframalpha

- URL / [There's more...](#)
- Wolfram MathWorld
 - URL / [See also](#)